

IBM Cúram Social Program Management
7.0.10.0

*IBM Universal Access Responsive Web
Application 2.7.0*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 248](#)

Edition

This edition applies to IBM® Cúram Social Program Management 7.0.10.0

Licensed Materials - Property of IBM.

© **Copyright International Business Machines Corporation 2018, 2020.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures.....	viii
Tables.....	ix
Chapter 1. IBM Cúram Universal Access.....	1
Chapter 2. IBM Cúram Universal Access release notes.....	2
2.7.0 release notes.....	2
What's new.....	2
Release notes.....	2
Known issues.....	3
Chapter 3. IBM Cúram Universal Access business overview.....	6
Screen.....	6
Filtered and eligibility screening types.....	6
Anonymous or authenticated screening.....	7
The ' Check what you might get ' page.....	8
The ' Here's what you might get ' screening results page.....	9
Screening from a citizen account.....	10
Apply for benefits.....	10
Starting and selecting an application.....	12
Managing existing applications.....	12
Saving an application.....	13
Resuming an application.....	14
Submitting an application.....	14
Printing an application.....	16
Withdrawing an application.....	17
Deleting an application.....	17
Verify.....	17
Citizen alerts and to-do messages.....	18
Viewing verifications.....	18
Submitting documents.....	19
Caseworker tasks.....	20
Update.....	20
Enter a life event.....	20
Appeal.....	22
Decide to appeal.....	22
Submit an appeal request.....	23
View your appeals.....	23
Appeals notices and notifications.....	24
Requesting an appeal from the citizen account.....	24
The citizen account.....	24
Creating a citizen account and logging in.....	24
The Dashboard page.....	25
The ' Your benefits ' page.....	27
The ' Your documents ' page.....	28
The Notices page.....	29
The Profile page.....	30

Chapter 4. Configuring IBM Cúram Universal Access.....	31
Configuring the browser.....	31
Configuring service areas and PDF forms.....	31
Configuring programs.....	32
Configuring a Program.....	32
Defining local offices for a program.....	36
Defining PDF mappings for a program.....	36
Defining program evidence types.....	36
Configuring screenings.....	36
Configuring a new screening.....	37
Configuring eligibility and screening details.....	37
Configuring screening display information.....	38
Defining programs for a screening.....	38
The screening auto-save property.....	39
Configuring rescreening.....	39
Pre-populating the screening script.....	39
Resetting data captured from a previous screening.....	39
Writing Rule Sets For Screening.....	40
Configuring applications.....	41
Configuring applications in the administration application.....	41
Configuring applications in system administration.....	43
Configuring other application settings.....	44
Configuring online categories.....	45
Configuring the citizen account.....	45
Configuring messages.....	45
Configuring last logged in information.....	54
Configuring contact information.....	54
Configuring user session timeout.....	55
Configuring appeal requests.....	56
Configuring communications on the Notices page.....	56
Configuring life events.....	56
Configuring a life event.....	57
Mapping life event information to evidence entities.....	58
Defining a question script, answer script, and schema.....	59
Categorizing life events.....	59
Defining Remote Systems.....	59
Chapter 5. Developing with the Universal Access Responsive Web Application.....	60
Installing the IBM Universal Access Responsive Web Application development environment.....	60
Prerequisites and supported software.....	60
Installing the IBM Cúram Universal Access development environment.....	64
Upgrading the IBM Universal Access Responsive Web Application.....	66
Starter pack and packages.....	67
Sample application project structure.....	69
Developing compliantly.....	71
Enforce code style with ESLint.....	71
Universal Access UI coding conventions.....	72
The sampleApplication feature.....	74
Manage state with React Hooks.....	76
Redux in Universal Access.....	77
Universal Access Redux modules.....	79
Social Program Management Web Development Accelerator.....	82
Error handling with a React higher-order component (HOC).....	84
Connectivity handling.....	85
Implementing a connectivity handler.....	86
Developing with routes.....	88

The Routes component.....	88
Adding routes.....	88
Replacing routes.....	89
Redirecting routes.....	89
Removing routes.....	89
Advanced routing.....	90
Connecting to Universal Access REST APIs.....	91
The mock server API service.....	91
The RESTService utility.....	93
Implementing file security and validation for the POST files API.....	94
Enabling the POST files API for file uploads.....	95
Adding metadata to file uploads.....	95
Universal Access REST APIs.....	96
Developing with headers and footers.....	102
Adding images, fonts, and files.....	103
Customizing the color scheme or typography.....	104
Developing toast notifications.....	107
Localization.....	108
Selecting a language.....	108
Configuring languages in the application.....	108
Translating your application.....	109
Regional settings.....	111
Development scenarios.....	111
Changing text in the application.....	111
Adding content to the application.....	113
Styling content with the Social Program Management Design System.....	114
Changing the application header or footer.....	115
Creating an IBM Cúram Social Program Management API.....	117
Connecting to REST APIs from the application.....	118
Testing REST API connections with Tomcat.....	121
Handling failures in the application.....	122
Implementing a loading mask.....	125
Reusing existing features.....	126
Implementing page view analytics.....	128
Implementing a test environment.....	129
End-to-end test environment.....	130
Jest and Enzyme test environment.....	142
Security, authentication, and authorization.....	147
Build secure web apps with the Social Program Management Design System.....	147
User account types.....	150
User account authorization roles and groups.....	151
Authentication.....	151
Authenticating with external security systems.....	154
Customizing account creation and management.....	171
Data caching.....	172
React environment variable reference.....	173

Chapter 6. Customizing IEG forms in the Universal Access Responsive Web

Application.....	181
IEG elements and attributes specific to the design system and Universal Access Responsive Web Application.....	181
IEG configuration not currently supported for the Universal Access Responsive Web Application.....	182
Configuring section navigation for forms.....	183
Configuring progress information for forms.....	183
Configuring dynamic titles on forms.....	183
Configuring rich text on forms.....	184
Configuring external links to open in a new tab or window.....	185

Configuring hint text for forms.....	185
Configuring explainer text for forms.....	186
Configuring the 'Help' label for forms.....	187
Configuring required or optional labels for form fields.....	187
Configuring input formats and constraints for form fields.....	188
Configuring phone numbers.....	189
Configuring date formats.....	190
Configuring currency symbols.....	190
Configuring code-table hierarchies for form fields.....	190
Implementing a combo box for form fields.....	191
Implementing search functions for ComboBox components.....	191
Configuring combo box scripts and schemas.....	193
Customizing script behavior with BaseFormContainer.....	194
Merging clusters with the cluster element grouping-id attribute.....	195
Configuring relationship pages questions.....	196
Configuring relationship starting dates on relationship summary pages.....	197
Chapter 7. Customizing IBM Cúram Universal Access.....	198
Customizing screening.....	198
Track the volume, quality, and results of screenings.....	198
Populating a custom screening results page.....	198
Customizing submitted applications.....	199
Customizing the intake application workflow.....	199
Customizing the generic PDF for processed applications.....	201
Using events to extend intake application processing.....	201
Customizing the concern role mapping process.....	201
How to send applications to remote systems for processing.....	202
Customizing the Citizen Account.....	202
Security and the Citizen Account.....	202
Messages.....	204
Customizing the Notices page.....	210
Customizing appeal request statuses.....	211
Error logging in the citizen account.....	212
Customizing life events.....	213
Enabling and disabling life events.....	214
How to build a life event.....	214
Customizing advanced life events.....	215
How to build a life event.....	215
Event APIs for life events.....	230
Customizing verifications.....	231
Enabling or disabling verifications.....	231
Customizing file formats and size limits for file uploads.....	232
Customizing a file upload lead time for verifications.....	232
Customizing how verification information is presented.....	233
Customizing verification names.....	235
Customizing caseworker tasks.....	237
Customizing appeals.....	237
Enabling and disabling appeals.....	238
Artifacts with limited customization scope.....	238
Chapter 8. Deploying your web application to a web server.....	240
Building the Universal Access Responsive Web Application for deployment.....	240
Install and configure IBM HTTP Server with WebSphere Application Server.....	240
Generating an IBM HTTP Server plug-in configuration.....	241
Configuring the IBM HTTP Server plug-in.....	241
Install and configure Oracle HTTP Server with Oracle WebLogic Server.....	242
Installing Oracle HTTP Server and its components.....	242

Configuring the Oracle HTTP Server plug-in.....	243
Installing and configuring Apache HTTP Server.....	244
Deploying your web application.....	244
Chapter 9. Troubleshooting and support.....	246
Citizen Engagement components and licensing.....	246
Citizen Engagement support strategy.....	247
Examining log files.....	247
Notices.....	248
Privacy Policy considerations.....	249
Trademarks.....	249

Figures

1. Key business flow for verifications..... 18

2. Key business flow for Appeals..... 22

3. SPM React and Java application development environments.....64

4. IdP-initiated flow in Universal Access.....156

5. SP-initiated flow in Universal Access..... 157

6. Universal Access SSO configuration components..... 159

7. Intake application workflow..... 199

Tables

1. Information messages for browser preferences.....	31
2. Appeal request acknowledgment.....	46
3. Appeal rejection.....	46
4. Application acknowledgment.....	47
5. Meeting invite.....	47
6. Meeting cancellation.....	48
7. Meeting update.....	48
8. Payment issued.....	51
9. Payment canceled.....	51
10. Payment due.....	52
11. Case suspended.....	52
12. Case unsuspended.....	52
13. Compatibility with IBM Cúram Social Program Management.....	60
14. Compatibility with IBM Cúram Social Program Management.....	61
15. The withErrorBoundary parameters.....	85
16. ACS trust association interceptor custom properties.....	162
17. Account configurations.....	171
18. Account events.....	172
19. Message properties files.....	204
20. Payment messages and related properties.....	208
21. Payment message expiry property.....	208
22. Meeting messages.....	209
23. Meeting message display date property.....	209

24. Application acknowledgment message expiry property.....	209
25. Application error codes.....	212

Chapter 1. IBM Cúram Universal Access

IBM Cúram Universal Access provides a configurable citizen-facing application that enables agencies to offer a web self-service solution to their citizens. Introduced in Universal Access v7.0.3, you can choose to use the IBM Universal Access Responsive Web Application instead of the classic client application. The Universal Access Responsive Web Application uses modern technologies, such as React JavaScript, and the IBM Social Program Management Design System to enable citizens to better access services in a browser from desktop, tablet, and mobile devices.

Use this information to customize Universal Access to provide your own custom citizen-facing web application. For information about working with the classic client application, see [IBM Cúram Universal Access with the classic client application](#).

The Universal Access Responsive Web Application asset is updated at more regular intervals than IBM Cúram Universal Access and the IBM Cúram Social Program Management Platform platform and has its own version numbering scheme.

Note: Online documentation for Universal Access is provided for the most recent version only. To read the documentation for older versions in PDF format, see the [IBM Cúram Social Program Management PDF library](#).

Chapter 2. IBM Cúram Universal Access release notes

Read about what's new and the release notes for recent versions of IBM Cúram Universal Access.

To read the release notes for older versions in PDF format, see the [IBM Cúram Social Program Management PDF library](#).

2.7.0 release notes

Read about enhancements and defect fixes in IBM Cúram Universal Access with IBM Universal Access Responsive Web Application 2.7.0.

For information about compatibility with IBM Cúram Social Program Management, see [Prerequisites and supported software](#).

For information about server-side changes, see the [IBM Cúram Social Program Management release notes](#) for your version.

What's new

Citizen Verifications - helping citizens to understand the verification process and enabling them to provide documents proactively

The new Citizen Verification feature in the IBM Universal Access Responsive Web Application informs citizens when information that they provide to an organization needs to be verified with supporting documentation, based on the SPM Verification Engine configuration. Citizens can then upload the requested documentation through the Universal Access Responsive Web Application, within the guidelines set out by an agency. Citizens see a new Your Documents page, where they can see their verifications and upload documents.

The feature also includes notifications to both citizens and caseworkers, alerting them to new and necessary steps to take. Caseworkers still take control of the verification of evidence, ensuring adherence to agency standards.

The feature includes new APIs to retrieve and display verification information for the citizen in Universal Access. APIs are also provided to upload documentation to IBM Cúram Social Program Management, which can then be sent to a designated content management system by using the existing CMIS integration point. The Social Program Management Design System was updated to include a file picker component that allows multiple documents to be selected and uploaded in a single transaction.

For more information about uploading documents, see [Submit documentation for verification](#).

Section navigation in IEG forms

Forms in Intelligent Evidence Gathering (IEG) are enhanced with new section navigation for the IBM Universal Access Responsive Web Application. Citizens can now see all the sections in a form, and easily navigate through available sections if they need to amend or add details. IEG section navigation uses the overflow menu from the SPM Design System.

For more information about section navigation, see [Controlling the Flow using Sections \(Universal Access Responsive Web Application\)](#).

Release notes

Support added for checking a user's account status when they submit an application

Previously, if a citizen was logged into a standard account, and their account was upgraded to a linked account in the background after they submitted an application, they would have to log out and in again before they could access their full linked account. Now, you can chose to configure your

application's .env file with a check for linked user account status on submission of an application so citizens can access their full linked account. This feature can be useful when an application is set up to automatically create linked accounts, such as in test or demo environments. (4451)

Checkbox group questions show 'Optional' for mandatory questions

An issue was fixed where Optional was incorrectly shown instead of Required on checkbox group questions that were configured to be mandatory. (5007)

Issue with log in for SP-initiated SAML SSO

Previously, when SP-initiated SAML SSO was configured, users were not able to log in again due to an issue with the SSO precheck, which was not triggered when they logged out. Now, a logout triggers the SSO precheck and users can log in as expected.

Validation improvement for the year in Date inputs

Previously, if you typed 01/01/0001 in a Date input, the date format would change due to an error in the validation of the year. This error was fixed and now when a year is entered into date fields they are validated correctly. (4878)

Screen reader issues related to dialogs

Previously dialogs were not announced correctly by screen readers. Screen readers now announce dialogs correctly, with their role (dialog or alertdialog), title, and description. (988)

JAWS screen reader issue with the date picker

Previously, the JAWS screen reader incorrectly read out the date input text field by reading all of the text in the cluster that contained the date picker along with the 12 months of the year. This issue was resolved, and now, when tabbing through the application, JAWS correctly reads the date input text field as expected. (4390)

VoiceOver screen reader issues on iPad

- Previously, VoiceOver read the text in the middle of the page when the **Request an appeal** button was clicked. This issue was resolved, and now VoiceOver correctly reads from the start of the page. (4714)
- Previously, VoiceOver was not reading the error heading and input error messages in IEG forms consistently on iPad. This issue was resolved, and now VoiceOver reads the IEG error heading and error messages as expected. (4219)

The Content Security Policy was updated so users can view a PDF from a thumbnail in Mozilla Firefox

The Content Security Policy was updated to allow PDF documents to be viewed when you select a thumbnail in Mozilla Firefox. To allow this in your custom application, add the following configuration to the main index.html file: (5010)

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-eval' 'unsafe-inline' resource://pdf.js" />
```

Known issues

Known issue with viewing a document or image from a thumbnail in Microsoft browsers

If you select a thumbnail to view a document or image in Microsoft Internet Explorer 11 or Microsoft Edge, an error occurs. (5009)

Known issue with reviewing an uploaded document on tablet and mobile devices

If you select an uploaded document to review it before you submit on tablet and mobile devices, an error occurs and it doesn't open. (4508)

Known issue on Microsoft Internet Explorer 11 with creating modules in the IBM Social Program Management Web Development Accelerator

When using Microsoft Internet Explorer 11, trying to create a module in the IBM Social Program Management Web Development Accelerator by pressing the **Confirm** button results in an error. (4838)

Known issue for translation support of IEG messages

The @spm/intelligent-evidence-gathering/locale folder, which is required to export English messages from intelligent-evidence-gathering for translation, is not included in this version. (4577)

Known issue with the SP-initiated flow for single sign-on (SSO)

The SP-initiated flow for SSO fails due to interference from polling APIs. The Assertion Consumer Service (ACS) request fails, blocking the authentication flow. (5107)

You can workaround this issue as follows:

1. Copy the SS0Verifier component from the application @spm/core-ui package:

```
cp node_modules/@spm/core-ui/src/components/SS0Verifier src/features/
```

2. Customize the src/features/SS0Verifier/SS0Verifier.js file so that it allows logging in as publiccitizen after SS0Precheck:

- a. Modify the AppSpinner import so that reads: `import { AppSpinner } from '@spm/core-ui';`
- b. Modify the async function executeSS0PreCheck so that it reads:

```
/**
 * check if the user is authenticated, if they are not authenticated
 * it checks the SSSO or authenticates the user as a PublicCitizen
 */
async function executeSS0PreCheck(dispatch) {
  // try to login
  const [success, response, authenticationCallback] = await authenticationLogin();

  // try to login in with SSO
  if (!success && Authentication.isSSOTypeSP() === 'true') {
    const [isLoggedInInTheSSO, ssoResponse, ssoCallback] = await
performSPIInitSS0AuthRequest(
  response
);
    if (isLoggedInInTheSSO && authenticationCallback) {
      authenticationCallback(ssoResponse, ssoCallback);
    } else {
      await loginAsPublicCitizen(dispatch);
    }
  } else {
    // if the user is not logged in, they must be logged in as a public citizen to access
    // the system configurations
    await loginAsPublicCitizen(dispatch);
  }

  // set so it does not verify if the user is logged again
  SS0VerifierActions.setSS0Verifier(dispatch, false);
}
```

3. Modify the src/App.js so that the @spm/core-ui SS0Verifier is no longer imported, and the local SS0Verifier is used instead:

```
import React from 'react';
import { BrowserRouter as Router } from 'react-router-dom';
import { Main } from '@govhhs/govhhs-design-system-react';
import {
  ApplicationHeader,
  ApplicationHeaderComponent,
  ApplicationFooter,
  ScrollToTop,
  SessionTimeout,
  ErrorBoundary,
  // SS0Verifier, -- this line commented out
```

```
    Toaster,  
    ConnectivityHandler,  
  } from '@spm/universal-access-ui';  
  import routes from './routes';  
  import ReduxInit from './redux/ReduxInit';  
  import IntlInit from './intl/IntlInit';  
  import SSOVerifier from './features/SSOVerifier/SSOVerifier.js' // -- this line added
```

4. Rebuild the application with `npm run build` and deploy it to your web server.

Chapter 3. IBM Cúram Universal Access business overview

IBM Cúram Universal Access is a citizen-facing web application that provides citizens with online facilities. Use this business overview to help you to map the existing Universal Access features and capability to your organization's business requirements during business analysis.

Screen

Citizens can self-check their eligibility for benefits and services before they submit an application. Checking for eligibility is implemented by using the Screening feature.

Screening has many advantages for both citizens and agencies:

- Citizens can check their eligibility for the benefits that the agency offers before they apply, and without having to go through the whole application process.
- Screening reduces the need for citizens to interact with the agency.
- Screening reduces the time and effort that caseworkers need to spend on screening tasks, freeing them up to concentrate on their core duties.
- Screening can quickly determine whether citizens are potentially eligible for one or more benefits based on a short set of guided questions and eligibility rules. Based on this determination, citizens can then decide whether to apply for the benefits.

Related concepts

[Printing an application](#)

Citizens can open and print an application form in two ways.

[Configuring screening display information](#)

You can configure the screening information display fields for each screening.

Filtered and eligibility screening types

To balance the need for quick screening results against the need to gather detailed citizen information, IBM Cúram Universal Access supports two types of screening. Screening results indicate the programs for which citizens might be eligible.

Filtered screening

Filtered screening allows citizens to quickly see whether they are eligible for benefits before going through the more detailed eligibility screening process. Asking questions like 'Are you married?' and 'Are you pregnant?' can quickly identify and eliminate programs for which citizens are unlikely to be eligible.

Filtered screening is defined by specifying a simple filter script and rules. Typically, a filtered screening script is not longer than two pages. If filtered screening is defined, the system immediately displays the filtered screening script when citizens select the screening. The system does not prompt citizens to select programs. Instead, the system runs the rules for all programs that are defined in the filtered screening rule set.

You can easily and quickly customize a filtered screening. For each screening, you configure the available programs and eligibility requirements. You then configure the script, rules, and data schema to collect and process citizen information, and define what information is displayed to citizens. When defined, citizens can screen themselves to identify programs that they might be eligible to receive. For more information, see *Configuring screenings*.

Program selection takes precedence over filtered screening. For more information about program selection, see [“Screen”](#) on page 6.

Eligibility screening

Eligibility screening determines citizens' potential eligibility to receive a program or programs. To gather the more detailed information that is needed to determine whether citizens qualify for benefits, eligibility screening uses a longer and more detailed IEG script. Typical questions can relate to the citizen's income, or resources, for example, savings, stocks, or bonds.

Eligibility screening consists of a script to collect data and a rule set to determine the citizen's potential eligibility for one or more programs.

Eligibility screening rules are run upon completion of the screening script and the results are displayed for citizens on the **Here's what you might get** page.

The eligibility screening rules are run only for programs that are associated with the screening.

The relationship between filtered and eligibility screening

Some points to note regarding the two screening types:

- Filtered screening is a precursor to eligibility screening.
- Filtered screening is optional. Citizens can screen for eligibility without doing a filtered screening.
- After completing a filtered screening, citizens must then complete an eligibility screening before they can apply for benefits.

Related concepts

The screening auto-save property

Use the screening **curam.citizenworkspace.auto.save.screening** property to set whether screenings are automatically saved for authenticated citizens.

[Configuring screenings](#)

Define different types of screenings that citizens can complete to identify programs that they might be eligible to receive.

Related information

[Natural Flow of an IEG Script](#)

Anonymous or authenticated screening

IBM Cúram Universal Access supports both anonymous and authenticated screening. Citizens who are not logged in, and want to retain a degree of anonymity, can screen themselves for benefit eligibility while unauthenticated. Citizens who are logged in can complete an authenticated screening.

Anonymous screening

Unauthenticated citizens can screen themselves for benefits without logging in but they cannot save their screening until they log in. Administrators can use an IEG script configuration to set if citizens have an option to save their progress. If an administrator sets the option to save progress on a particular script, unauthenticated citizens are taken to the **Log in** page when they select to save. When logged in or signed up, citizens' screening progress is saved and they are taken to the **Dashboard**. For more information on IEG script configuration, see *Configuring IEG*.

Related information

[Configuring IEG](#)

Authenticated screening

Citizens who are logged in to Universal Access can complete an authenticated screening.

Pre-populating citizen data

Citizens might want the convenience of having their data pre-populated when they start screening. You can use the system property *curam.citizenaccount.prepopulate.screening* to pre-populate citizen data into a screening. If citizens are linked users, their basic details are populated into the script if *curam.citizenaccount.prepopulate.screening* is enabled. If *curam.citizenaccount.prepopulate.screening* it

is disabled, citizens must complete their details. For more information, see *Pre-populating the screening script*.

Saving screenings for authenticated citizens

Authenticated citizens can save a screening and resume it later. As citizens progress through the script, information that is entered on the previous page is automatically saved each time that citizens click **Next** in the IEG script. If there is a timeout or the browser is closed accidentally, automatically saving the information prevents the loss of the screening information. Use the **curam.citizenworkspace.auto.save.screening** property to set whether screenings are automatically saved in the citizen account. For more information, see *The screening auto-save property*.

In-progress screenings

When citizens save an in-progress screening, or it is automatically saved by the system, an in-progress screening message is displayed in the citizens' dashboard as a reminder. Citizens can complete an in-progress screening or they can delete it. When citizens complete a screening, the **Here's what you might get** page is displayed and the in-progress message is removed. The screening also appears on the **Benefits checker** page on the **Dashboard**.

The Benefits you might get pane

Citizens can view completed screenings on the **Benefits you might get** pane in the citizen **Dashboard**. To ensure that the most recent results of a screening are kept relevant for the citizen, one screening of the same type can be in the complete state at a time. Citizens can use the **Benefits you might get** pane to view the results of the screening or delete the screening from the pane.

Configuring rescreening

Citizens might need to change a screening if they forget to provide some information or their circumstances change. In the administration application, the agency can set whether to allow citizens to change and resubmit their screening. If the setting is set to **Yes**, citizens can rescreen from the **Benefits you might get** pane or from the **Screening results** page. If the setting is **No**, citizens do not see these links, in this case if the citizen wants to rescreen, they must delete their screening and start again. For more information, see ["Configuring rescreening" on page 39](#).

Related concepts

Pre-populating the screening script

When citizens screen from within a citizen account, you can pre-populate information already known about the citizen performing the screening.

The screening auto-save property

Use the screening **curam.citizenworkspace.auto.save.screening** property to set whether screenings are automatically saved for authenticated citizens.

Related tasks

Configuring rescreening

Configure whether citizens can change and resubmit their screenings.

The 'Check what you might get' page

Screening starts when citizens select **Check what you might get** on the organization **Home** page.

When citizens select to create a new account, an account creation screen is displayed. After the citizen successfully creates the account, the citizen is automatically logged in to the system and the screening process proceeds.

If citizens are logged in and they click **Check** on any screening where they have a previously completed or in-progress screening of that type, they are alerted to the existence of that previous screening. Citizens can then either view the current progress of that screening or they can start screening again.

If citizens start screening again, any in progress screenings are overwritten. Any completed screening is only overwritten when citizens get to the screening results page.

The **Check what you might get** page lists and describes each of the screenings that are available.

Note: The **Check what you might get** page is laid out as follows:

- Page description - a banner indicating to citizens that they can screen themselves.
- A list of screenings with a description of what each screening is.
- A list of benefits with a description of what each benefit offers.

A screening might allow citizens to screen for one or more programs. Citizens are prompted to select the programs for which they want to be screened. However, there are three situations when citizens are not prompted to select programs:

- If filtered screening is defined for the screening. In this instance, citizens are prompted to select the programs for which they want to be screened when filtered screening is complete.
- If a single program is defined for the screening.
- If a screening has been configured to disable program selection by citizens. The Program Selection indicator determines whether citizens can select specific programs to screen for or whether they are brought directly into a screening script where they are screened for all programs associated with the screening. For more information, see *Defining Program Selection*.

Note: Program selection takes precedence over filtered screening. Also, if filtered screening is enabled but only one program configured, citizens are brought directly to eligibility screening for that single program.

Citizens select the screening and the programs for which they want to be screened and then click **Check**. The system then starts the associated IEG script so that screening can start.

Related concepts

Configuring screenings


Define different types of screenings that citizens can complete to identify programs that they might be eligible to receive.

The 'Check what you might get' page

Screening starts when citizens select **Check what you might get** on the organization **Home** page.

The 'Here's what you might get' screening results page

When a screening is submitted, the eligibility rules run and the list of programs is displayed with the results on **Here's what you might get** page.

- Programs that the citizens might be eligible for are marked with the Eligible  icon. Citizens can click **Apply** to apply for these programs online through the **Apply for benefits** flow.
- Programs for which eligibility cannot be determined are listed with a suitable message, which can be configured in the administration application. For example:

Based on what you have told us, we are unable to make a determination for Child Care Assistance.

Administrators can use Cúram Express® Rules (CER) to provide detailed explanatory text to help citizens understand the decisions that are made about potential eligibility. For more information, see *Working with Cúram Express Rules*

If citizens' circumstances change, they can screen again at any time by clicking **Check again for what you might get**.

Applying for benefits online and offline

The **Here's what you might get** page also indicates whether benefits can be applied for offline. Benefits that citizens can apply for offline are listed under the heading:

Or apply for these programs by filling out the form and sending or bringing it to your nearest office.

Programs that can be applied for offline typically have a **Download application** link.

How to apply

For each screening type, you can configure helpful, informative text to display on the **Here's what you might get** page header. This text is configured in the **How to apply** rich text editor in the administration application. For more information, see [“Configuring screening display information” on page 38](#).

The **How to apply** editor allows flexibility for the agency on how they want to communicate to the citizen the different ways they can apply. For example, the agency might advise citizens to apply online by clicking **Apply**. The page also allows citizens to print the application so the agency might advise citizens to mail the application to the agency.

The **How to apply** editor can include hyperlinks on the page. This is useful if the agency wants citizens to visit their local office. For example, the agency might choose to use Google Maps as a way to show the citizen where their local office is. The agency is free to use the maps provider of their choice that suits their needs.

Transferring data from screening to application

A sysadmin configuration setting allows citizens' screening data to be reused when they apply directly from the **Here's what you might get** page. When set to ON, some details based on the schema that is applied is transferred into the application. This existing information saves the citizen time when they are completing their application.

Related concepts

[Configuring screening display information](#)

You can configure the screening information display fields for each screening.

Related information

[Working with Cúram Express Rules](#)

Screening from a citizen account

Citizens can screen themselves for programs while logged in to their citizen account.

By using a short set of guided questions and eligibility rules, citizens can determine whether they might be eligible for one or more programs. Based on this determination, the citizen can decide whether to apply for the programs identified.

To perform a screening, citizens take the following steps:

1. Select **Check what you might get** on the organization **Home** page.
2. Select **Check** on the eligibility category.
3. Select the benefits they think they might get on the **Include benefits** page
4. Select **Continue** to start the check eligibility process.
5. Citizens then answer the questions on the screening script.
6. Select **Next** to navigate through the pages in the script.
7. When the process is complete, citizens are shown the benefits they might be eligible for on the **Here's what you might get** page.
8. Citizens can then **Apply** for benefits.

Related concepts

[Pre-populating the screening script](#)

When citizens screen from within a citizen account, you can pre-populate information already known about the citizen performing the screening.

Apply for benefits

Citizens can apply for benefits from the organization **Home** page or the **Dashboard**. Citizens must submit an application that includes personal details like income, expenses, employment, education. This

information is the evidence of the citizen's case. Agencies can use this information to determine eligibility for benefits. Citizens can also apply offline by downloading the application form, filling it in and sending it to the agency. Citizens can also contact their local agency office.

Before you begin

Citizens can apply for benefits by logging in to their account. Citizens who log in can save an application for a benefit before they submit it and then return later to complete the application. Citizens can also partially apply for benefits without logging in. If the configuration option *submit on completion* is set to **No**, citizens can submit a partial submitted application. Citizens do not have to be logged in to submit the partial application.

A customizable icon for each application is displayed with the application name, followed by a description of the application. The application and benefit descriptions are configurable in the administration configuration.

Note: The terms "benefit" and "program" are synonymous. An application might consist of one or more benefits. For example, the "Income Support" application might contain the "Food Assistance" and "Cash Assistance" benefits.

Procedure

1. Citizens click **Apply for benefits** on the organization **Home** page, the **Dashboard**, or the **Your benefits** tab.

Note: Benefits are displayed in alphabetical order by default, but you can override this order.

2. For each benefit type, citizens can take the following actions:
 - a) Click **Learn more** to find out more about the benefit. If the *More Info URL* setting is configured for the application, **Learn more** is conditionally displayed.
 - b) Click **Print application** to print the application form, complete it by hand and mail it to the agency. If the *PDF Application Form* setting is configured for the application, **Print application** is conditionally displayed.
 - c) Click **Apply** to start the application process for the benefit. **Apply** is conditionally displayed if **multiple applications** is set to **Yes** or if **multiple applications** set to **No** and the citizen has no existing, pending decision applications.
3. Citizens can also click **Check what you might get** to see what benefits they might qualify for.

Results

If citizens try to quit the application without saving it, the application displays a warning dialog box to prevent accidental loss of information.

Note: Citizens must click the application name on the page in to see the **Leave this application** dialog box. The application name is also conditionally enabled depending on whether the **quit and delete** option is enabled in the IEG script.

Clicking **Leave** brings citizens to the **Dashboard** if they are logged in, or to the organization **Home** if they are not logged in.

Clicking **Cancel** returns citizens to the point at which they left the application script with the previously entered data available. Citizens can cancel an application without saving at any point before they submit. Citizens can cancel only when the application is in progress, if they **Save** and **Exit** they can only **Delete** the application.

Citizens can also:

- Resume an application by selecting the **Continue** link on the **Your benefits** page, or by selecting **Continue** on any in-progress application alerts in the **Dashboard**.
- Withdraw an application. If available, the withdraw option is displayed for the pending decision application on the **Your benefits** page.

- Delete an application. Citizens can only delete in-progress applications that they have not yet submitted to the agency.

Starting and selecting an application

Citizens can select the benefits that they want to apply for.

Citizens start an application by selecting **Apply for benefits** on the **Organization** home page or selecting the **Benefits** navigation item. Citizens are then brought to the **Apply for benefits** page.

The **Apply for benefits** page describes each of the available applications. To make it easier for administrators to find the required application, they are grouped into categories, for example "unemployment services". The applications, and their categorization, are defined in the Universal Access Administration section of the Administration Application. Citizens can also click **Learn more** to learn more about each application or can click **Print application** to print to a PDF file.

Citizens can apply for a benefit. Citizens start an application for a benefit they have already applied for, they can resume the application, or they can start again.

A customizable icon is displayed for each benefit type along with the benefit name and a description of the benefit.

Citizens might use an application to apply for one or more programs. Typically, the system prompts citizens to select the programs they want to apply for. However, in two situations the system does not prompt the citizen to select programs:

- A single program is defined for the application.
- Each application is configured so that the citizen can select a program or automatically select all of the programs that are associated with the application.

Configuring the application process

You can configure the application process as follows:

- Each configured application is displayed. If an application has more than one associated program, it is displayed in the second column of the **Apply for benefits** page.
- A configuration property **program selection** is available at the application level. If the property is set to **Yes**, an **Include benefits** page is displayed allowing the citizen to select all, or a subset of the configured programs.
- If an application contains one program and the configuration property program selection is set to **Yes**, the **Include benefits** page is not displayed.
- If the program selection is set to **No** and the application contains multiple programs, all the programs are automatically applied for and the **Include benefits** page is not displayed.
- A configuration property for multiple applications is available at the program level. If this property is set to **No** there is an existing pending decision for the program, the **Apply** option is disabled.
- A system property *curam.citizenaccount.prepopulate.screening* sets whether the IEG script is pre-populated with any available citizen information.

When citizens select the applications and the programs they want to apply for, the system starts the associated IEG script. Citizens use the script to complete the selected applications.

Managing existing applications

When a citizen logs in, any existing applications are listed and the citizen is presented with different options that depend on the state of an application.

The agency can configure the system to specify whether citizens need to be authenticated before they apply for benefits:

- If authentication is enabled, citizens must either create a new user account or log in to an account before they start the application process.
- If authentication is disabled, citizens can proceed with the application without authentication.

The configuration property *curam.citizenworkspace.authenticated.intake* specifies whether citizens must log in to apply for benefits. If the property is set to **NO**, citizens do not have to log in to apply for benefits. If the property is set to **YES**, citizens must create an account or log in to an existing account to apply for benefits.

Depending on how authentication is configured, applications are managed in one of the following ways: Citizens can log in to their account, or they can sign up from the application overview page. Citizens can also be prompted to log in, sign up, or send application without an account at the end of the IEG application script.

If citizens create an account, they are automatically logged in to the system and the intake process starts. The system also checks whether they have any existing applications.

The configuration property *curam.citizenworkspace.authenticated.intake* is available at the application level. If this property is set to **No**, citizens can submit a partially completed application, if this property is set to **Yes**, citizens cannot submit a partially completed application.

Existing applications are in one of the following categories:

- **Application in progress.** The application is in progress but is not yet submitted. Citizens can either continue or delete applications in this category.
- **Pending decision.** The application is awaiting a decision from the case worker. Citizens can either download or withdraw applications in this category.
- **Active.** The caseworker has authorized the application.
- **Denied** The caseworker has rejected the application.
- **Authorization failed.** Citizens can download applications in this state.
- **Withdrawn.** Citizens can withdraw an application if it is in **Pending decision** or **Denied** status.

The application lists are displayed only if there are items in the list, that is, if there are no saved applications. If applications are listed, the citizen is presented with different options that depend on the state of an application. The citizen might resume or delete an incomplete application, withdraw a submitted application, or start a new application.

Related concepts

[Security, authentication, and authorization](#)

IBM Cúram Universal Access gives citizens access to their most sensitive personal data over the internet. Security must be a primary concern in the development of citizen account customizations. All projects that are built on Universal Access must focus on delivering security from beginning to end.

Saving an application

By default, applications are automatically saved for citizens who are logged in. Citizens can also manually save applications, including in-progress applications. Applications are not saved for citizens who are not logged in.

An application is automatically saved each time that citizens click **Continue** in the IEG script. Citizens can go to the **Benefits** page to resume or to delete an in-progress application.

A system property specifies whether applications are automatically saved. By default, this property is enabled. For more information, see *Configuring applications*.

When citizens quit an application, three options are depending on how the intake application is configured.

- Save the application
- Leave the application without saving
- Cancel the application

If citizens save the application and they are not logged in, the save application screen is displayed. Citizens can create an account, log in, or send the application without logging in.

If the **Submit on Completion Only** administration setting is selected, citizens can submit a partially completed application. Citizens see the option to submit a partially completed application on the Save and Exit modal when they save and exit an IEG script. If the **Submit on Completion Only** administration setting is not selected, citizens cannot submit a partially completed application.

Related concepts

Configuring applications

Use IBM Cúram Social Program Management administration and system administration applications to define the applications that are available for citizens. For each application, you can configure the available programs and an application script and data schema. You must also configure the remaining applications details, such as application withdrawal reasons.

Resuming an application

Logged-in citizens can resume an application by selecting the **Continue** link on either the **Dashboard** or the **Your benefits** page.

Selecting **Continue** in the citizen's **Dashboard** resumes the application from where the application was last saved. When an application is resumed, the data that is entered is automatically saved as citizens moves from page to page through the script.

When citizens resume an application, they are brought to where they left off when the application was saved.

Submitting an application

To allow citizens to submit an application to the agency, you must specify a submission script for the application in the administration system. After citizens submit an application, the way the script is processed depends on the configuration of the programs for which the citizen is applying.

The application might be submitted when citizens complete the intake script or when they exit a script before it completes. An intake application can be configured so that an agency can dictate whether an application script can be submitted before it is complete or not.

If citizens send an application to the agency, either by exiting or completing a script, the screen that is displayed depends on:

- Whether citizens are logged in
- Whether citizens must either create or log in to an account before the application is submitted.

If citizens are not logged in, they are prompted to log in or create a new account. If the property is enabled, citizens must log in to an existing account or create a new account before the application can be sent to the agency. For more information, see *Managing existing applications*.

Specifying log in requirements

The system can be configured so that:

- Citizens are not required to identify themselves to the system AND
- Citizens can send the application to the agency without logging in or creating an account.

Alternatively, the system can be configured so that citizens must create an account or log in. For more information, see *Managing existing applications*.

Managing in-progress and submitted applications

If citizens log in before they send the application to the agency, the system can determine whether:

- There is an in-progress application of the same type OR.
- Citizens previously submitted applications for the same programs that are still pending disposition, that is, awaiting a decision by the agency.

For an in-progress application of the same type, a page is displayed. From here, citizens can send the new application to the agency or keep the saved application, thus discarding the new application. The options available are to **Start again** or **Resume** the in-progress application.

If citizens submit applications for the same programs, the system determines whether they can still submit any of the programs to the agency for processing. Programs can be configured so that multiple applications can be submitted for the program at any time. For example, submitting a new application for cash assistance for a different household unit than a previously submitted application that the agency is processing. This screen indicates that the application cannot be submitted for all of the programs for which the citizen wants to apply. However, the application might still be sent to the agency. There are three options: continue to submit the application for the programs for which the citizen can apply, save the application, or delete the application.

The configuration property **Multiple application** is available at the program level. If this property is set to **No** and there is a pending decision for the program, the **Apply** option is visible but disabled.

Specifying a submission script

To submit an application to the agency, a submission script must be specified for the application in administration. The submission script is required because applications require additional information, which does not form part of the application, to be captured before the applications can be submitted. For example, a Cash Assistance application requires information that relates to the citizen's ability to attend an interview. This information would not be appropriate for another type of application that does not require an interview to be conducted, for example, unemployment insurance. Electronic signatures are another example of the type of information that would typically be captured by using a submission script. This data might not be captured as part of the script, as citizens can submit the application before they complete the script.

Processing a submitted script

The processing that happens on completion of the submission script depends upon the configuration of the programs for which citizens are applying. Program eligibility can be configured such that it might be determined by using IBM Cúram Social Program Management or a remote system. If IBM Cúram Social Program Management is specified as the eligibility system, an application case creation process is started. The application case creation process includes a search and match capability, which attempts to match citizens on a new application to registered persons on the system based on configured search criteria. When search and match finishes, one or more application cases are created. If the programs that are applied for are configured for different application case types, multiple application cases are created. If the application was submitted within the business hours of the root location for the organization, the application date on the application case is set to today's date. If the application is submitted outside of the business hours of the organization, the application date is set to the next business date.

Mapping the application data to case evidence tables

The data that is entered for the application might be mapped to case evidence tables. The mappings are configured for a particular program by using the Cúram Data Mapping Editor. For the appropriate evidence entities to be created and populated in response to an online application submission, a mapping configuration must be specified for a program.

Associating requested programs with application cases

When the application case is created, the programs that are requested by the citizen are associated with the relevant application case. Some organizations might impose time limits within which an application for a program must be processed. A number of timer configuration options are available for a particular program. These timers are set when a program is associated with an application case.

If the eligibility is determined by a remote system, configurations are provided to allow a web service to be started on a remote system.

Displaying submission confirmation

The submission confirmation page is displayed upon successful submission of an application to the agency. The submission confirmation page displays the reference number that is associated with the submitted application. Citizens can use this reference number in any further correspondence about application with the agency.

Configuring intake applications for PDFs

The citizen might also open and print a PDF. The configuration of the intake application determines the actual PDF that opens. The application can be configured to use a PDF designed specifically by the agency with the intake application, or, if no PDF form is specified, to use a generated generic PDF. If an agency-designed form is specified, this form is opened when the citizen clicks the **PDF** link. For programs with associated mapping configurations of type **PDF Form Creation**, the data that is entered during the online application is copied to the PDF form. The data is copied for each of the programs for which the citizen is applying with this mapping configuration. If a mapping configuration is not associated with a program, the information that is entered during the online application for that program is not copied to the PDF form. If a PDF form is not specified, a generic generated form opens instead. This form contains a copy of the information that is entered by the citizen when the citizen is completing the online application.

The agency can define additional information to be displayed on the generic generated form. Typically, the additional information that is required helps the agency to process the application quickly. Proof of identity is an example of this additional information. This additional information is configurable for each type of application.

Submission confirmation

When citizens successfully submit an application, going through the sign and submit screen, they are brought to an updated version of the **Overview**. The stages specific to the application process are now updated with a confirmation message to indicate that the application was successfully submitted:

- A customizable icon
- An application reference number
- Informational message for the citizen
- A **Save submitted application PDF** link that allows citizens to download the information that is entered as part of the application, in PDF format.

Submit documents for verification

If your organization includes the online submission of documents in their business process, citizens can upload and submit supporting documentation for information that they provide, so caseworkers can verify their changes. For more information, see [“Submitting documents” on page 19](#).

Related concepts

[Managing existing applications](#)

When a citizen logs in, any existing applications are listed and the citizen is presented with different options that depend on the state of an application.

Printing an application

Citizens can open and print an application form in two ways.

- Citizens are directed to a PDF that they can open, complete, and print.
- Citizens are taken through a script. After citizens complete or exit the script, they can open a PDF containing the information that they entered.

PDF forms can be configured to provide versions in all supported languages. The programs that can be applied for using the PDF form can also be configured.

Each PDF form that is defined in the administration system is displayed on the **Apply for Benefits** page. The **Apply for benefits** page is displayed when **Apply For Benefit** is selected from the organization **Home** page.

If **PDF Application Form** is configured for the application, **Print application** is displayed.

To open the PDF form, citizens click **Print application**. Citizens can also identify the address of the local office to which to send the form. A system property sets whether the system uses postal codes or counties for this function.

Withdrawing an application

Citizens can withdraw successful applications from the **Your benefits** page. If the application did not successfully submit, the **Withdraw** option is not displayed.

Citizens can withdraw a successfully submitted application or they can also withdraw applications for all or any one of the programs.

Citizens can withdraw each program individually. The reasons for withdrawing the program application can be configured for the intake application in the administration system.

The **Reason** field contains a list of configurable code table values that are defined by the administrator. The list of values is configured at application level.

The **First name**, **Last name**, and **Reason** fields are mandatory.

The submit action on the page withdraws the application. The system automatically updates the status of the programs that are associated with the application case to **Withdrawn** and sends a notification to the application caseworker.

The difference between deleting and withdrawing an application

The **Withdraw** action is different from the **Delete** action in that only a submitted application can be withdrawn and only an in-progress application can be deleted. Also, **Delete** physically deletes the application record, **Withdraw** changes the status of the application to *Withdrawn* after the citizen goes through a workflow.

Related concepts

[The citizen account](#)

When citizens create a secure citizen account, they can access a range of relevant information. Citizens can also use the citizen account to track and manage their interactions with the agency.

Deleting an application

Citizens can delete applications that are not yet submitted to the agency.

Citizens can delete applications from the **Dashboard** or the **Your benefits** pages. When citizens click the **Delete application** link for an in-progress application, a confirmation dialog box is displayed.

Verify

If your organization includes the online submission of documents in their business process, citizens are notified in the Universal Access Responsive Web Application when some of their information needs to be verified with supporting documentation. They can then provide that supporting documentation online. Both citizens and caseworkers receive notifications, alerting them to any steps to take. Case workers control the verification of evidence, ensuring adherence to agency standards.

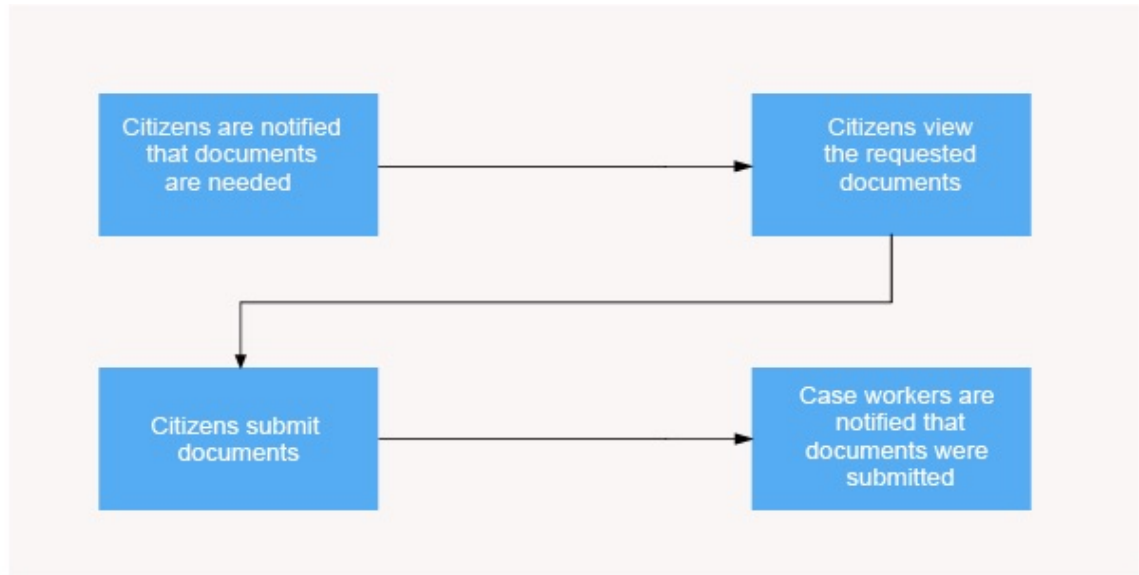


Figure 1. Key business flow for verifications

Related concepts

Customizing verifications

If your organization includes the online submission of documents in their business process, citizens can upload and submit documents from the IBM Universal Access Responsive Web Application to prove information that they provided in their benefit applications. You can customize a number of aspects of the verifications functionality in the application.

Related information

[Verification Engine](#)

Citizen alerts and to-do messages

When citizens must provide documentation to the agency, they see an alert in their dashboard, and a to-do message for each application or benefit case where documents are needed. Only linked users see the verification alert and to-do messages.

The alert is removed when there is at least one document submitted for each verification.

To-dos are grouped by case, so a citizen can have multiple to-dos if they have multiple applications or cases. The to-do messages for a case are removed when all documents are provided for that case.

Viewing verifications

Citizens who are linked users can upload and submit documents to prove information that they provide. For example, they can submit a birth certificate to prove a date of birth and a caseworker can then verify the evidence with the submitted document.

Verifications are displayed whenever they are generated for that user, either by online or in-person interactions with the organization. For example, verifications can be generated during an online benefit application process, or on receipt of a postal application form, or when a caseworker meets with a client in the office.

Verifications that are raised for any case members on a case or application for which the citizen is the primary client are displayed to the citizen. If there are verifications raised on behalf of an entire family, the verification is raised against the primary client and are displayed as such to the citizen.

Where there are multiple people on an application or case, you can see and can submit documentation for each person's outstanding verification requirements.

Verified and non-verified verifications are displayed, but canceled verifications are not displayed. When the information is verified, the verification's status is updated. If a verification item utilization expires, the verification is shown here again and a message indicates that more recent documentation is needed.

Related concepts

[The 'Your documents' page](#)

When a citizen who is a linked user is logged in, they see a **Your documents** page that provides a consolidated view of their verifications and submitted documents. Citizens can see what information they need to provide documentation for, information for which they have submitted documentation, and verifications that were done in the past.

Submitting documents

Citizens can add and submit one or more documents. For phones or tablets, the file picker uses the native functionality of the device so they can take a photo, select a picture or select a file. When they submit a document, they must specify the type of document from the list of eligible document types. The caseworker is notified that documents are ready to verify.

To prevent unnecessary submissions, citizens cannot submit further documents when the verification status is **Verified**.

A verification is displayed for each item of information for which caseworkers need documentation to verify. Citizens can see a list of the information to be verified and the eligible documents that a caseworker can use to verify that information.

When submitting documents on desktop devices, Citizens can select only files in valid file formats. If you are using a phone or tablet, the file picker uses the native functionality of the device so you can take a photo, select a picture or select a file. On mobile devices, an error is shown if you select an unsupported file format. By default, the allowed file formats are JPG, JPEG, PNG, TIFF, and PDF and the file size limit is 5 MB. You can customize the allowed file formats and file size limit.

Citizens specify the type of document from the list of eligible document types such as a passport. The eligible document types are based on the verification item utilization.

Citizens can check a document before they submit by clicking the thumbnail image to see the document. They can delete a document before they submit.

On submission, the verification status is updated and a task is generated for the caseworker in the IBM Cúram Social Program Management caseworker application.

Sharing and reusing documents

You can configure whether documents of a certain type can be shared and reused across verifications that require the same document type.

- If you choose that a document can only be used against that verification, and must be unique, then the document is not shared across any other verifications that might use the same document type. A citizen might need to resubmit the document multiple times if they have multiple cases with the same information.
- If you chose that a document can be shared and reused, the submitted documents can be associated with other relevant verification items. If previously submitted documentation is suitable, citizens can select that documentation to reuse for the verification.

Administrators can configure a verification item utilization by setting the **Usage Type** to Shared or Unique, see [Verification Item Utilizations](#).

Caseworker tasks

When documents are submitted for verification by a citizen, a task is generated for the caseworker that is assigned to the citizen's case.

The task indicates that evidence on the case or for the person is now ready for verification, based on the documents submitted.

A system configuration is available to determine whether the task is generated when all documents for an evidence record are received, or when all documents for all evidence records on the case are received.

When the caseworker opens the task, details of the evidence, its related documentation and whether it's ready for verification are visible to the caseworker for review.

Related tasks

Customizing caseworker tasks

When a citizen submits a document for a verification, a task is generated for the caseworker. Tasks are displayed to the caseworker that is assigned to the citizen's case when they log in to the caseworker application. System administrators can configure the system to display a task each time a citizen provides all documents for an individual evidence record on a case, or to display the task only when a citizen has provided all documents for every evidence record on a case.

Update

Citizens can update their details by submitting a change in their circumstances to the agency, which is implemented by using the Life Events feature. Examples of changes in circumstances include a change of address, a birth, or marriage. These significant events in citizens' lives might affect the benefits or services that they are receiving or are due to receive.

Key business flow scenario

James Smith is in receipt of child benefit and is also working full time. However, he just lost his job as the company that he works for is closing. James needs to tell the agency about losing his job so that he can get his benefit reviewed. Life Events allows James to communicate this change to the agency without visiting the office. This reduces the amount of interaction with the agency and saves valuable caseworker time.



[Click here for a video presentation that gives an overview of Life Events.](#)

Enter a life event

Citizens who are logged in can review their existing profile information on the **Profile page** and make any required changes. They can submit a change in their circumstances by selecting either **Review your profile** on their dashboard or selecting **Profile** to open the **Profile** page.

The 'Tell us if anything has changed' pane

The **Tell us if anything has changed** pane displays the available life events, for example, **Change of Address** or **Change in Employment Status**. Each configured life event is a card, with a description that is configurable by an administrator in Universal Access life event administration.

The administrator can categorize life events in Universal Access life event administration. For example, you can categorize changing jobs, income changes, and change of address life events under **Employment**. If a life event is not categorized, it appears in the **All** category tab. If citizens cannot immediately see the life event that they want, they can select **See more** to see the life events across all categories.

The life event Overview page

When citizens select a life event, the **Overview** page outlines the update process.

The steps list any information or documentation that they must provide, and approximately how long the submission takes to complete. The steps can also include how the agency might inform them of the change when the change of circumstance is complete.

When citizens read and understand the information that is presented, they can select **Start** to enter the submission form.

When citizens begin a submission form, they are presented with a guided set of questions that use Intelligent Evidence Gathering (IEG) to gather information. The IEG script for the form is defined in Universal Access life event administration.

The life event Summary page

After they enter information, a **Summary** page displays so they can review their changes before submission.

The life event Confirmation page

On successful submission of the life event, the **Confirmation** page is displayed.

The confirmation page can display information that is useful and relevant to the submitted life event. This information can be defined in Universal Access life event administration.

- Text can be added. For example, agencies can say that a change might take some time as a caseworker review is needed.
- The **Next steps** pane can display information such as actions that citizens might need to take after they submit the change. For example, citizens might need to update their rent if they move into a new home. The **Next steps** pane can also include links to external websites.

Citizens don't need to have a case on the system to submit a life event. If they don't, the submitted information isn't transmitted to a case owner. Instead, the submitted information is stored internally and the agency must decide what to do with the information.

The life event Consent page

An optional **Consent** page can be displayed so that citizens can consent to having their details sent to selected other agencies or third parties. Administrators can configure the **Consent** page to display for a life event in Universal Access life event administration. This action constitutes the citizens' consent to send information to the selected agencies. The information can be transmitted to a remote system through a web service or to the relevant case owners on an IBM Cúram Social Program Management system through the evidence broker.

The life events change history

Citizens can access their previously submitted life events from the dashboard by clicking **Review your profile > Previous changes**. The list of life events is sorted by the submission date. They can select a life event record from the history list to view a summary of the information that they submitted to the agency.

Submit documents for verification

If your organization includes the online submission of documents in their business process, citizens can upload and submit supporting documentation for information that they provide, so caseworkers can verify their changes. For more information, see [“Submitting documents” on page 19](#).

Related concepts

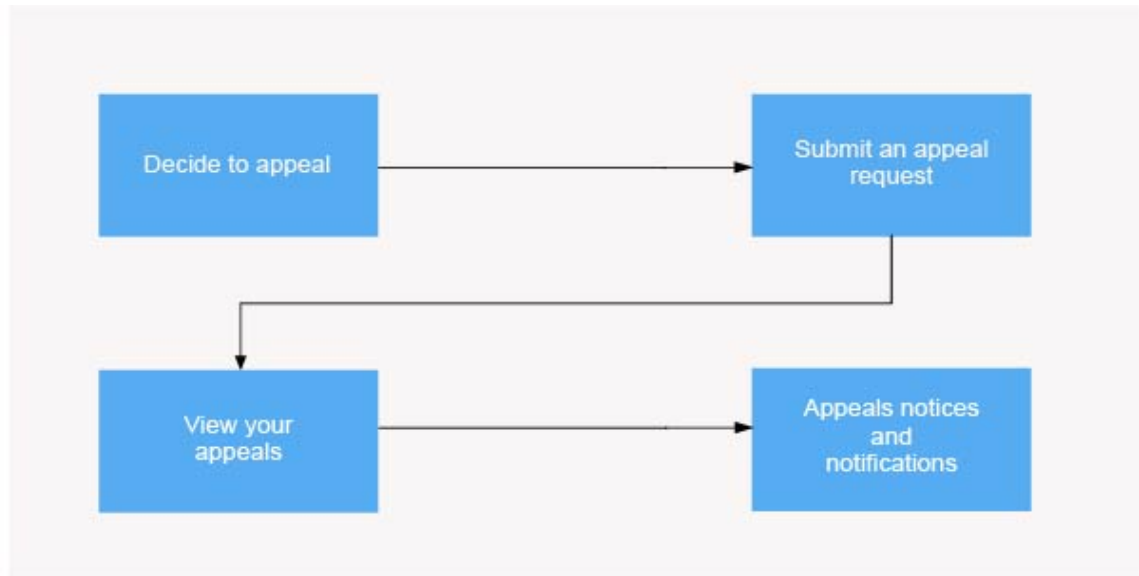
[Configuring a life event](#)

You can configure a life event in the administration application on the **New Life Event** page.

Appeal

If your organization includes appeals in their business process, citizens can appeal decisions on their benefits online from their citizen accounts on their own devices. If your organization uses the IBM Cúram Appeals application module, your organization can process appeals through the full appeals life cycle that is provided by that solution.

Figure 2. Key business flow for Appeals



1. [“Decide to appeal” on page 22](#)
2. [“Submit an appeal request” on page 23](#)
3. [“View your appeals” on page 23](#)
4. [“Appeals notices and notifications” on page 24](#)



[Click here for a video presentation that gives an overview of Appeals and Notices.](#)

Related tasks

[Customizing appeals](#)

You can customize appeals to suit your organization. You can integrate with an appeals system of your choice. If you are licensed for the IBM Cúram Appeals application module, the IBM Cúram Social Program Management appeals functionality is available on installation.

Decide to appeal

If citizens don't agree with a decision on their benefits, they can appeal the decision. They can appeal for themselves or a family member, and can appeal online regardless of how they originally applied. A citizen must have applied for at least one benefit in order to appeal.

By default, they can appeal:

- An eligibility determination.

- A change to their eligibility.
- Their calculated benefit.

Citizens are informed of their rights of appeal, and an overview page explains anything that citizens need to know before they request an appeal.

Submit an appeal request

After they read their appeal rights and understand the appeals process, citizens complete a form with all of the relevant information. This information can range from details of the benefit itself to supplemental information needed to establish informal reviews and hearings such as interpreters or emergency needs.

You can configure the form to ask for the specific information that is needed by your organization. The SPM Design System accommodates a wide range of question formats to enable the citizen to easily complete this form. You can use a summary page to provide further information in the form to help the citizen and to alleviate specific concerns.

After they enter and review their appeal request details, citizens sign and submit the request for appeal and get a confirmation of the submission. The confirmation page outlines the next steps and sets out the time frames for the organization to respond, and any communications to be expected.

Appeals processing

A caseworker or hearing official can receive notification of that appeal and begin processing.

- When the IBM Cúram Appeals application module is installed, the full appeals lifecycle and statuses in that solution are supported. A task is created and assigned to an appeal request work queue when the citizen submits the request. The appeal request is recorded against the citizen's person record. A PDF file is generated from the IEG script and is stored for caseworker reference as a communication against the appellant in the caseworker application.

A caseworker can then act on the request and either acknowledge the request and continue with the appeal process or reject the request. An acknowledgment or rejection message is displayed in the citizen's account. A list of submitted appeal requests is provided in the citizen's account and provides a view of the request's status.

- When the IBM Cúram Appeals application module is not installed, a citizen can request an appeal. They can receive an appeal request submitted status, and the organization must implement an appeals solution to handle the submitted appeal requests and other appeal lifecycle processing.
- Alternatively, an organization can implement a solution to have a third-party appeals system process the appeal and to generate the appropriate appeal lifecycle processing, statuses, and messaging.

View your appeals

Citizens can see their appeals on the **Appeals** page. All appeals that citizens submit are displayed and are updated with the appropriate color-coded statuses as they move through the Appeals lifecycle of hearings and decisions. At any stage, citizens can log in and understand what is happening with their appeal.

The **Appeals** page displays each appeal in a card, with copy of the original appeal details if needed. Typically, the details that are provided in the earlier form are added to a PDF, both the citizen and the caseworker receive a copy.

The statuses of appeals are updated as the appeal moves through the appeals lifecycle, as pre-configured for the IBM Cúram Appeals application module, or as configured for your organization's custom appeals process.

Appeals notices and notifications

Citizens receive both formal notices and informal notifications at specific milestones in the appeals process. These updates provide them with instant status updates, while they wait for formal notice of a decision or next steps.

Notices

Citizens can see communications in the **Notices** page, which are typically formal written communications about the appeal or hearing, typically issued to meet legal, regulatory, or state requirements. Notices are often created by using letterhead templates.

Notifications

Citizens can see messages in the **Notifications** pane on their dashboard, which are typically informal messages that inform the citizen of any significant point in a process. For example, for appeals, notification can inform citizens of any progression on their appeal request, such as when their appeal request was first acknowledged, or if their appeal was accepted or denied.

Requesting an appeal from the citizen account

When logged into their citizen account, a citizen can review their rights of appeal. They can request an appeal on a benefit decision if they are a participant on a IBM Cúram Social Program Management application or case.

Before you begin

For example, a citizen might be deemed ineligible on application, or have their benefit payments reduced. If they don't agree with the decision or the circumstances of the decision, they can appeal the decision.

Procedure

1. Go to the **Appeals** page.
2. Click **Request an appeal**. The appeals process overview page is displayed.
3. Review the overview of the appeals process, and when you are ready, click **Start**. The appeal request form opens.
4. Complete the appeal request form.
5. Sign and submit the form.
6. Your appeal request is complete. Review the **Confirmation and next steps** information.

The citizen account

When citizens create a secure citizen account, they can access a range of relevant information. Citizens can also use the citizen account to track and manage their interactions with the agency.

Creating a citizen account and logging in

Citizens can create a citizen account at any time, including during the check eligibility and application processes.

Creating an account

Citizens can select **Sign up** on the organization **Home** page to create an account. Citizens then enter their first and last names, an optional email address, and an account password. If citizens select **I don't have an email address**, they can specify a user name instead.

Administration configurations

- Number of login attempts before the account is locked out: 5
- Number of remaining login attempts before a user warning is displayed: 3
- Number of break-in attempts before an account is locked: 3

- Maximum and minimum characters in a user name
- Maximum and minimum characters in a password

For more information about user name and password length, see *Account management configurations*.

Logging in

To log in to the citizen account, citizens select **Log in** on the organization **Home** page. Depending on how they created their account, citizens enter either an email or user name. They then enter their password and click **Next**. You can configure the number of log-in attempts citizens have before their account is locked out. For example, if you set the number of login attempts to three, the account is locked for citizens who make more than three unsuccessful login attempts.

On a successful login, the **Citizen account** dashboard is displayed.

Related concepts

Screen

Citizens can self-check their eligibility for benefits and services before they submit an application. Checking for eligibility is implemented by using the Screening feature.

Account management configurations

Use the following configuration properties to define the behavior of password validations for citizen accounts.

The Dashboard page

When a citizen is logged in, their **Dashboard** shows an overview of their account.

If your organization uses Appeals, and a citizen has applied for at least one benefit, they also see an **Appeals** page. For more information, see [“Requesting an appeal from the citizen account” on page 24](#).

If your organization uses Verifications, and a citizen is a linked user with at least one verification that needs documentation, they also see a **Your documents** page. For more information, see [“The 'Your documents' page” on page 28](#).

System messages

System messages are broadcast to all logged-in citizens and are displayed at the top of the dashboard. For example, system messages can inform citizens about planned system outages.

Citizens who are linked users can see system messages about their verifications.

In-progress applications messages

Messages about current in-progress applications are displayed at the top of the screen. Citizens can either continue or delete their in-progress applications.

The 'Check what you might get' card

Citizens can click **Check what you might get** to check their own eligibility for benefits. For more information, see [“The 'Check what you might get' page” on page 8](#).

The 'Apply for benefits' card

Citizens can click **Apply for benefits** to apply for a benefit. For more information, see [“Starting and selecting an application” on page 12](#).

The 'Review your profile' card

Citizens can click **Review your profile** to update their profile with a change in circumstances. For more information, see [“Enter a life event” on page 20](#).

The Payments pane

Lists a summary of the most recent payments to the citizen. Citizens can click **All payments** to view the payment details or see their payment history. For more information, see [“The 'All payments' page” on page 26](#).

The 'Benefits you might get' pane

Lists a summary of any in-progress eligibility checks. Citizens can click recheck or delete individual eligibility checks.

The To-Dos pane

Lists actions that citizens must take to complete an application, including action messages that the caseworker creates for the citizen. For example, a citizen might need to provide supplementary information to support a benefit application.

Citizens who are linked users can see messages about their verifications.

The Meetings pane

Lists a summary of meetings that citizens were invited to including the dates of the meetings. The latest meeting is shown first.

The Notifications pane

Lists acknowledgments for all of the applications that citizens make. A date is included for most notifications. The most recent notification is shown first. Example notifications include application acknowledgment, appeal request messages, or service request messages.

Related concepts

[Customizing specific message types](#)

Organizations can customize the default message to create a referral message or a service delivery message.

The 'All payments' page

The **All payments** page shows more details about payments to the citizen. The messages that are associated with these payments can be retrieved from IBM Cúram Social Program Management or another remote system. Canceled or expired payments are also displayed.

Payments can be by check, electronic funds transfer (EFT), cash, or voucher. Depending on the payment type, the following details can be displayed for each payment:

Check

Payee address and check number

EFT

Bank sort code and bank account number

Cash

Payee address

Voucher

Payee address and voucher number

Citizen account messages

In addition to system messages and in-progress application messages, the **Payments**, **To-Dos**, **Meetings**, and **Notifications** panes on the **Dashboard** display citizen account messages. Messages from remote systems can also be displayed by using web services. For example, messages can be about meetings for the citizen, or activities that are scheduled for the citizen.

Displaying a message

Each message has a title and an icon. In addition, the **To-Dos** and **Notifications** messages have an effective date and time that specifies when the message is displayed. Usually the effective date of a message is set to the current date, but you can set the effective date by configuration settings.

For example, you might not want to display a message immediately if a service is scheduled in the future. You can configure the message to display a specified number of days before the start date of the service. The system uses the number of days to populate the effective date.

Messages from remote systems are displayed based on the effective date that is specified in the web service.

Prioritization and ordering

You can assign a priority to a message so that it is displayed at the top of the **Meeting** listing.

You can also configure the order of messages types in the administration system. For example, you can configure payment messages to be displayed first and meeting messages to be displayed second.

Message duration

The message type determines the length of time that the message is displayed. You can set the message duration by start and end dates or by replacing one message with another.

Where items have start and end dates, you can use them to specify the duration that message is displayed. For example, service messages are displayed until the start date of the service.

In other cases, you can replace a message with another message. Use a configuration setting to determine whether to:

- Specify the duration for when a message is replaced.
- Specify the number of days after which the message is removed.

The duration of messages from remote systems is based on the expiry date that is defined in the web service.

System messages

Agencies can use system messages to send a message to everyone who has a citizen account. For example, to provide information and help line numbers to clients for a natural disaster, such as a flood. You can configure system messages in the Administration application by using the **New System Message** page.

The **Title** and **Message** fields define the title of the message and the message body that is displayed to a client in the **My Messages** pane. You can assign a priority to the message by using the **Priority** field, which means that it displays at the top of the messages listing.

The **Effective Date and Time** field defines an effective date for the message, such as when the message is displayed in the dashboard page. The **Expiry Date and Time** field defines an expiry date for the message, for instance, when the message no longer is to be displayed in the dashboard.

Messages are saved with a status of **In-Edit**. Messages must be published before they display in the citizen account. When it is published, the message is active and is displayed in the citizen account based on the effective and expiry dates.

Predictive Response Manager

The Predictive Response Manager (PRM) is the infrastructure that is used to build and then generate and display messages on the Citizen Account home page.

A number of default messages are provided and are described in this information along with their associated configurations

For more information about configuring messages, see [“Customizing specific message types” on page 207](#).

Verifications messages

Verifications messages are displayed on the **To-Dos** pane. The messages are removed from the list when documents for all verifiable data items are supplied.

Verifications are grouped by person or case, either application case or integrated case, rather than as individual notifications. A case reference number is provided where appropriate. The verifiable data items are displayed in a list.

The 'Your benefits' page

When a citizen is logged in, they can see all of their benefits applications and the application status on the **Your benefits** page.

The **Your benefits** page displays applications that can be in one of the following states:

- **Application in progress.** The application is in progress but is not yet submitted. Citizens can either continue or delete applications in this category.
- **Pending decision.** The application is awaiting a decision from the case worker. Citizens can either download or withdraw applications in this category.
- **Active.** The caseworker has authorized the application.
- **Denied** The caseworker has rejected the application.
- **Authorization failed.** Citizens can download applications in this state.
- **Withdrawn.** Citizens can withdraw an application if it is in **Pending decision** or **Denied** status.

If a submitted application is approved by the caseworker and a product delivery case is created for that application, the application is also displayed on the **Your benefits** page.

The 'Your documents' page

When a citizen who is a linked user is logged in, they see a **Your documents** page that provides a consolidated view of their verifications and submitted documents. Citizens can see what information they need to provide documentation for, information for which they have submitted documentation, and verifications that were done in the past.

The verification items are organized based on their status:

Documents required

Verifications that require citizens to submit documentation so that their information can be verified by a caseworker.

Documents received

Verifications for which citizens submitted documents for information to be verified by a caseworker.

Documents accepted

Verifications for which a citizen submitted documents, and their information was verified by a caseworker.

Verifications

Each item of information for verification is shown. If needed, you can customize how information is presented for individual verifications, see [“Customizing how verification information is presented” on page 233](#).

Verifications show the following summary information:

- The information to be verified, which can be a single verifiable data item, or a group of verifiable data items related to an evidence record.
- The status of the verification:
 - **Not yet submitted.** One or more documents are required to verify this information but were not yet submitted.
 - **Documentation submitted.** The caseworker is reviewing the submitted documents or has verified some of the required documents for multiple verifiable data items.
 - **Verified** A caseworker has successfully verified this item of information with the submitted documents.
- The person for which information needs to be verified, that is, the case participant. For example **For James Smith**.
- The due date, that is, the date by which the documents are to be submitted by the citizen. For example, **Due 26 Sept**. By default, this is the same date as the date that the information needs to be verified by the caseworker. If needed, your organization can configure a lead time to the due date so that documents are submitted earlier to give caseworkers enough time to verify the evidence and process the application.
- The names of any programs that are associated with an application case, or product deliveries that are associated with an integrated case, depending on whether a citizen is applying for or receiving benefits.

Application for ... <program> is shown for application cases, for example, **Application for Rent Assistance**. The program name is shown for product delivery cases, for example **Food Assistance**.

Verification details

The following verification details are shown:

Provide documents that show

The details of the information that needs to be verified, which consists of the verifiable data items, and a description of the evidence provided and what the verification was raised against.

Eligible documents

A list of the documents that can be provided to prove the information. For example:

- Paid Medical Invoice
- Prescription Receipts
- Doctor's Letter

Add or reuse documents

To prove information is correct, you can add documents or reuse documents that you have already submitted.

Your submitted documents

If documents were previously submitted for the validation, the documents are listed here. You can download your previously submitted documents to see them in detail. A message indicates any documents that are no longer valid.

Related concepts

Viewing verifications

Citizens who are linked users can upload and submit documents to prove information that they provide. For example, they can submit a birth certificate to prove a date of birth and a caseworker can then verify the evidence with the submitted document.

The Notices page

When a citizen is logged in, they can see all communications that are relevant to them on the **Notices** page, with sent, received, or normal status indicated. Notices are typically formal written communications that are issued to meet legal, regulatory, or state requirements, which are created by using letterhead templates.

For example, online appeal requests are shown on the **Notices** page.

By default, communications are listed where the logged-in citizen is the concern or is a correspondent on the communication, in other words, for linked users.

Citizens can see the communication description and any attachment in the expanded view. They can view or save attachments by clicking the **View attachment** link.

Citizens can request that notices are sent to them by mail. The system logs the request to send the communication to the citizen. The request includes communication (ID), date, time, and status. After a citizen requests a notice by mail, the **Request this notice by mail** link is disabled.

What can you configure or customize?

You can configure the number of communications that are listed. You can also create a custom implementation to change what communications are shown, such as showing communications for other family members.

The processing of requests for communications by mail is customizable, so customers can add their own logic to deal with these requests.

Related concepts

[Customizing the Notices page](#)

By default, the notices relevant to the linked user are listed on the **Notices** page. You can replace the default CitizenCommunicationsStrategy implementation with your own custom implementation.

Related tasks

[Configuring communications on the Notices page](#)

You can configure the maximum number of communications that are displayed on the **Notices** page. By default, up to 20 communications are displayed.

The Profile page

When a citizen is logged in, they can see their information, including contact information, on their **Profile** page.

Citizen information

Citizens can see profile information, such as their contact information. Their contact information can include information like their address, phone number, and email address. A configuration setting determines whether the citizen's contact information is displayed on the citizen account. For example, an agency can set the `curam.citizenaccount.contactinformation.show.client.details` property to `false` to disable citizen contact information. For more information, see [“Configuring contact information” on page 54](#).

Tell us what has changed

Citizens can submit updates to their profile information and contact details. For more information, see [“Update” on page 20](#).

Related concepts

[Configuring contact information](#)

Configure contact information for citizens and caseworkers.

Chapter 4. Configuring IBM Cúram Universal Access

System administrators use the following configuration options to configure and maintain IBM Cúram Universal Access features such as applications and online categories.

Configuring the browser

To access the application, you must enable cookies and JavaScript in the browser by configuring the appropriate browser preferences.

The following table lists the browser preferences that you must configure for the application to work, and shows the errors that are displayed if the prerequisites are not met.

Table 1. Information messages for browser preferences	
Browser preference	Information message
When cookies are disabled.	Cookies are currently disabled and are required for the application to work. Please enable cookies and retry.
When JavaScript is disabled.	JavaScript is currently disabled and is required for the application to work. Please enable JavaScript and retry.
When cookies and JavaScript are disabled.	Cookies and JavaScript are currently disabled and are required for the application to work. Please enable and try again.

Configuring service areas and PDF forms

You can define a service area by configuring the counties or ZIP codes that are associated with the service area. You can also specify a PDF form that citizens can use to apply for programs.

Configuring service areas

Service areas are defined in the **Service Areas** section of the administration application. When defining a service area, you must specify a service area name. You can associate counties and zip codes with the service area, these represent the areas covered by the service area. Service areas can be associated with a local office which represents the office that services the service areas associated with it. Local offices identify where citizens can apply in person for a program or where they can send an application. For more information on associating service areas with local offices where a citizen can apply in person for a program, see *Defining local offices for a program*.

Configuring PDF forms

PDF forms are defined in the **PDF Forms** section of the administration application. When defining a PDF form, you must specify a name and language. You can also add a version of the form for each language that is configured. The forms are accessible from the **Print and Mail Form** page.

You can associate a local office with a PDF form. Associating a local office with a PDF form allows an administrator to define the local office and associated service areas where citizens can send their completed application.

Enabling citizens to search for a local office

A search page allows citizens to search for a local office. Citizens can either search by county or by zip code. The system property `curam.citizenworkspace.page.location.search.type` determines how the search works. If you set `curam.citizenworkspace.page.location.search.type` to **Zip**, citizens can search for a local office using a zip code. If you set this property to **County**, citizens can select from a list of counties to get a list of local offices.

Related concepts

Defining local offices for a program

Citizens might be able to apply for a program in person at a local office. A local office must be first defined in the *LocalOffice* code table in system administration.

Configuring programs

You can configure different types of programs. To configure a program, you configure display and system processing information, local offices, mappings to PDFs, and evidence types.

Configured programs can be associated with screenings and applications. The main aspects to configuring a program are as follows:

- Configure programs and associated display and system processing information.
- Configure local offices where an application for a program can be sent.
- Configure mappings that allow information gathered during application intake to be mapped to a PDF form.
- Configuring evidence types that allows for expedited authorization of programs that may need to be processed before other programs within a multi-program application.

Configuring a Program

Programs are configured on the administration **New Program** page. Details and specifications of the program are required to be defined when the program is created.

Defining a name and reference

The name that you define is displayed in the administration application.

Define a name and reference when creating a new program. The name that is defined is displayed both to the citizen and in the internal application. The reference is used to reference the program in code.

Defining an intake processing system

Define an intake processing system for each program.

Two options are available:

- **Cúram**
- Select from the list of preconfigured remote systems.

If intake is managed by IBM Cúram Social Program Management, select **Cúram**. If intake is managed by an external system, the program application is sent to the remote system by using the `ProcessApplicationService` web service, select a remote system.

If **Cúram** is specified as the intake system, an application case type must be selected. An application case of the specified type is created in response to a submission of an application for the program. An indicator is provided which dictates whether a **Reopen** action is enabled on the programs list on an application case for denied and withdrawn programs of a particular type. A workflow can be specified that is initiated when the program is reopened. For more information on configuring application cases, see *Cúram Intake overview*.

When an application case type is selected, the program can be added manually to that type of application case by a worker in the internal application as part of intake processing. A configuration setting specifies whether the program is a coverage type. Coverage types are automatically evaluated by program group

rules in the context of healthcare reform applications, such as insurance affordability. Coverage types cannot be applied for directly by a citizen or manually added to an application case by a worker and authorized. If the program is a coverage type, select **Yes**. The program is filtered out of the list of programs available to be added to online and internal applications in administration and the list of programs available to be manually added to an application case by a worker. If the program is not a coverage type, select **No**. The program will be available to be manually added to online and internal applications in administration and to an application case by a worker.

A remote system must be configured in the administration application before it can be selected as the case processing system. For more information about remote systems, see *Configuring Remote Systems*.

Related information

[Cúram Intake overview](#)

Defining case processing details

Define a case processing system for each program.

Two options are available:

- **Cúram**
- Select from remote systems.

If the program eligibility is determined and managed by using a Cúram-based system, select **Cúram**. If eligibility is determined and managed by an external system, select a remote system.

If you select **Cúram** as the case processing system, more options are available to allow you to configure program level authorization. Program level authorization means that if an application case contains multiple programs, each program can be authorized individually, and a separate case is used to manage the citizens on an ongoing basis.

Defining the integrated case strategy

Define the integrated case strategy so that the system can identify whether a new or existing integrated case is used when program authorization is successful.

The integrated case strategy identifies whether a new or existing integrated case is used when program authorization is successful. The integrated case hosts any product deliveries created as a result of the authorization. If a new integrated case is created, all of the application case clients are added as case participants to the integrated case. If an existing integrated case is used, any additional clients on the application case are added as case participants to the integrated case. Any evidence captured on the application case that is also required on the integrated case is copied to the integrated case upon successful authorization. The configuration options for the integrated case strategy are as follows:

New

A new integrated case of the specified type is always created when authorization of the program is successful.

Existing (Exact Client Match)

If an integrated case of the specified type exists with the same citizens as those cases present on the application case, the existing case is used automatically. If multiple integrated cases that meet these criteria exist, the caseworker is presented with a list of the cases and must select one to proceed with the authorization. If no existing cases match the criteria, a new integrated case is created.

Existing (Exact Client Match) or New

If one or more integrated cases of the specified type exist with the same citizens as those cases present on the application case, the caseworker is presented with the option to select an existing case to use as the ongoing case, or to create a new integrated case. If no existing cases match the criteria, a new integrated case is created.

Existing (Any Client Match) or New

If one or more integrated cases of the specified type exist, where any of the clients of the application case are case participants, the caseworker is presented with the option to select one of the existing cases to use as the ongoing case, or to create a new integrated case. If no existing cases match the criteria, a new integrated case is created.

Specifying the Integrated Case Type

The administrator must specify the type of integrated case to be created or used upon successful program authorization as defined by the Integrated Case strategy listed.

Specifying a client selection strategy

Specify a client selection strategy to define how clients are added from the application case to the product delivery.

The client selection strategy defines how clients are added from the application case to the product delivery created as a result of authorization of a program. If a product delivery type is specified, a client selection strategy must be selected. The configuration options are as follows:

All Clients

All of the application clients are added to the product delivery case. The application case primary client is set as the product delivery primary client. All other clients are added to the product delivery as members of the case members group.

Rules

A rule set determines the clients to be added to the product delivery if a product delivery is configured. At least one client must be determined by the rules for authorization to proceed.

User Selection

The user selects the clients who are added to the product delivery. The caseworker must select both the primary client and any other clients to be added to the case member group on the product delivery.

Specifying a Client Selection Ruleset

A Client Selection Ruleset must be selected when the Client Selection Strategy is **Rules**.

Specifying a product delivery type

Specify a product delivery type.

The **Product Delivery Type** drop-down specifies the product delivery that is used to make a payment to citizens in respect of a program. **Product Delivery Type** displays all active products configured on the system.

Note: This field applies to both program and application authorization processing. That is, program and application authorization can result in the creation of the product delivery type that is specified.

Submitting a product delivery automatically

The **Submit Product Delivery** indicator specifies if the product delivery created as a result of program authorization should be submitted automatically for approval. If selected, the product delivery created as a result of authorization of this program is submitted automatically to a supervisor for approval.

Note: This field applies to both program and application authorization processing. That is, program and application authorization can result in the automatic submission of a product delivery.

Configuring timers

Agencies can impose time limits within which an application for a program must be processed. You can configure application timers for each of these programs.

For example, an agency might want to specify that food assistance applications are authorized within 30 business days of the date of application.

The following configuration options are available, including the duration of the timer, whether the timer is based on business or calendar days, a warning period, and timer extension and approval.

Duration

The length of the timer in days. This value, along with the fields **Start Date** and **Use Business Days** (and the configured business hours for the organization) calculate the expiry date for the timer. This value is used as a number of business days if **Use Business Days** is set. If **Use Business Days** is not set, this value is used as calendar days.

Start Date

Specifies whether the timer starts on the application date or the program addition date. The options available are **Application Date** and **Program Addition Date**.

Note: In most cases, these dates are the same. That is, the programs are added at the same time as the application is made. However, when a program is added later to the application, after initial submission, the dates differ.

Warning Days

Specifies a number of warning days to warn citizens that the timer deadline is approaching. If configured, the **Warning Reached** workflow is enabled when the warning date is reached and the timer is still running (for example, the program is not completed).

End Date Extension Allowed

Specifies whether citizens can extend the timer by a number of days.

Extension Approval Required

Specifies whether a timer extension requires approval from a supervisor. If approval is required, the supervisor either approves or rejects the extension. After the extension is approved, or if approval is not required, the timer expiry date is updated to reflect the extension.

Use Business Days

Specifies if the timer should not decrement on non-working days. If this indicator is set, the system uses the **Working Pattern Hours** for the organization to determine the non-working days when it is calculating the expiry date for the timer.

Resume Timer

Specifies whether the program timer must be resumed when the program is reopened.

Resume From

If a timer is resumed, the **Resume From** field specifies the dates from which a program can be resumed. The values include the date that the program was completed, denied, or withdrawn, and the date that the program was reopened.

Timer Start

Specifies a workflow that is started when the timer starts.

Warning Reached

Specifies a workflow that is started when the warning period is reached.

Deadline Not Achieved

Specifies a workflow that is enacted if the timer deadline is not achieved; that is, the program is not being withdrawn, denied, or approved by the timer expiry date.

Configuring multiple applications

Configure multiple applications so that citizens can apply for a program while they have a previous application pending.

The **Multiple Applications** indicator dictates if citizens can apply for a program while they have a previous application pending. If set to true, citizens can have multiple pending applications for the given program. That is, citizens can submit an application for this program while they already have a pending application in the system. If it is set to false, this program is not offered if logged in citizens have pending applications for this program.

This configuration is not applicable to Health Care Reform Applications.

Defining a PDF form

Defining a PDF form for a program enables citizens to print an application for that program and either post it to the agency or bring it to a local office. When a PDF Form is specified for a program, the PDF form is displayed on the **Print Out and Mail** section of the **Here's what you might get** page that is displayed when citizens complete a screening. PDF forms must be defined before they can be associated with a program. When they are defined, they are displayed on the **Print and Mail Application Form** page.

Defining a URL

If a URL is defined, a **More Info** link is displayed with the program name so that citizens can find out more information about the selected program.

Defining description and summary information

When a program is displayed on the **Select Programs** page, a description can be displayed which gives a description of the program. The **Online Program Description** field defines this description.

A description summary of the program can also be defined using the **Online Program Summary** field. The field is a high-level description of the program displayed on the **Here's what you might get** page that is displayed when citizens complete a screening.

Defining local office application details

Citizens can apply for programs at a local office. If this is the case, the **Citizen Can Apply At Local Office** indicator indicates that local office information is displayed for a program.

Additional information can also be defined, for example, citizens might need to bring proof of identity if they want to apply at the local office. An administrator can define this information in the **Local Office Application Information** field.

Defining local offices for a program

Citizens might be able to apply for a program in person at a local office. A local office must be first defined in the *LocalOffice* code table in system administration.

Associating a local office with a program allows an administrator to define the local offices and their associated service areas where a particular program can be applied for in person. This information is displayed on the **Here's what you might get** page that is displayed to citizens when they complete a screening. Service areas must be defined before they can be associated with a local office.

Defining PDF mappings for a program

The information that citizens enter during an application can be mapped to a PDF form which citizens can then print.

To map the application data to the PDF Form for all programs a citizen is applying for, there must be a mapping configuration of type *PDF Form Creation* for each of the programs. The PDF Form is the form specified for the Online Application the program is associated with.

Defining program evidence types

Associate evidence types with a program.

Evidence types can support applications for multiple programs where a program must be authorized more quickly than other programs for which citizens might have applied. Using this type of configuration, only the evidence required for the program to be authorized is used and copied to the ongoing cases. This allows benefits for the authorized program to be delivered to citizens, while the caseworker continues to gather the evidence required for the other programs applied for.

Configuring screenings

Define different types of screenings that citizens can complete to identify programs that they might be eligible to receive.

For each screening, you can configure the available programs and eligibility requirements. You can then configure the script, rules, and data schema to collect and process citizen information, and define what information is displayed to citizens.

Once defined, citizens can perform a screening to identify programs that they may be eligible to receive. There are four main aspects to configuring a screening:

- Configuring information about a screening to be displayed to citizens.

- Configuring the script, rules and schema used to collect and process the information specified by citizens to identify their eligibility.
- Configuring the programs for which citizens can check their eligibility when performing a screening.
- Configuring additional screening system properties.

Configuring a new screening

Screenings are configured on the **New Screening** page.

The screening configurations are as follows.

Defining a name

You must define a name must be defined when creating a screening. The name defined is the name of the screening displayed to citizens in the IBM Cúram Universal Access portal.

Defining program selection

The **Program Selection** indicator defines whether citizens can select specific programs that they want to screen for, or whether they are brought directly into a screening script. If citizens are brought to a script, they are screened for all programs associated with the screening.

Defining a More Info URL

If a More Info URL is defined, a **More Info** link is displayed.

Allowing re-screening

The **Allow Rescreening** indicator defines whether citizens can re-screen when they have completed a screening.

Defining an icon for a screening

If you want an icon displayed with a screening, select an icon from the **Icon** selection box.

Note: Alternatively, you could modify the `img src` attribute of the icon directly on the screening HTML page, for example

```

```

Configuring eligibility and screening details

Configure details for eligibility screening or filtered screening

Two types of screening are supported - filtered screening and eligibility screening. Eligibility screening collects answers to a set of questions, stores this information and processes it to identify eligibility. Filtered screening reduces the number of programs that a citizen might screen for by asking a short set of questions and using the answers to filter out the programs that they would not be eligible for.

Configuring eligibility screening details

Specify an IEG script for the screening to collect the answers to a set of questions. You must also specify a data store schema to store the data entered in the script. On saving the screening, the system creates an empty template for both the script and schema based on the Question Script and schema that you specified. You can update these templates from the **Screening** tab by selecting hyperlinks provided on the page. Clicking the **Question Script** link starts the IEG editor that allows you to edit the question script. Click the schema link to start the Datastore Editor, you can then edit the schema.

You must specify a CER rule set to process the data in the data store and to produce an eligibility result. When specified on creation of the screening, the system creates an empty rules template. You can then update the ruleset from the **Screenings** tab by selecting the hyperlink provided on the page. Clicking the

link starts the CER Editor, which allows you to edit the ruleset. For more information about writing screening rule sets, see [“Writing Rule Sets For Screening” on page 40](#)

Configuring filtered screening details

Specify filtered screening details for a screening so that filtered screening is available before citizens perform eligibility screening. As with eligibility screening, you must define a Filter Script (IEG) and associated data store schema to collect and store the answers to questions. You must also specify a Filter Rules (CER rule set) to process the data and produce a filtered screening result. When specified on the **New Online Screening** page, the system automatically creates an empty template for the scripts and ruleset that can be subsequently updated by selecting the associated hyperlinks on the **Screening** page.

Reusing rule sets across screenings

Use the system property *curam.citizenworkspace.screening.ruleset.reuse.enabled* to specify:

- Whether CER rule sets can be reused across different screenings.
- Whether the same rule set can be used for eligibility and filtered screening.

If *curam.citizenworkspace.screening.ruleset.reuse.enabled* is enabled, you cannot reuse rule sets, if it is disabled you can reuse rule sets. You cannot use the *ScreeningRulesLinkDAO.readActiveByRuleSet* method when *curam.citizenworkspace.screening.ruleset.reuse.enabled* property is enabled.

Configuring screening display information

You can configure the screening information display fields for each screening.

Summary information

Define a high-level description of the screening.

Here's what you might get text

Define the text to be displayed on the **Here's what you might get** page, which is displayed to show citizens the results of a completed screening.

Description

Define a description of the screening to be displayed.

How to apply text

Allows an administrator to define the text displayed on the **Here's what you might get** page.

Defining programs for a screening

You must associate programs with a screening so that citizens can screen for those programs.

You can associate any program that is described in *Configuring Programs* with a screening. When associating programs with a screening, you can assign an order that sets the display order of the selected program relative to other programs associated with the screening.

Related concepts

[Configuring programs](#)

You can configure different types of programs. To configure a program, you configure display and system processing information, local offices, mappings to PDFs, and evidence types.

The screening auto-save property

Use the screening **curam.citizenworkspace.auto.save.screening** property to set whether screenings are automatically saved for authenticated citizens.

By default, **curam.citizenworkspace.auto.save.screening** is set to true. All screenings, irrespective of type, are automatically saved for authenticated citizens. Each screening is automatically saved when citizens click **Next** to progress through an IEG script. If **curam.citizenworkspace.auto.save.screening** is set to false, screenings are not automatically saved.

Configuring rescreening

Configure whether citizens can change and resubmit their screenings.

About this task

In the administration application, you can configure whether to allow citizens to change and re-submit their screening. If so, citizens can rescreen from the **Check what you might get** page or from the **Here's what you might get** page. If not, citizens who want to rescreen must delete their screenings and start again.

Procedure

1. Log in to IBM Cúram Social Program Management as an administrator.
2. Select **Administration Workspace > Shortcuts**.
3. Select **Universal Access > Screenings**.
4. Select the screening that you want to change.
5. Select **... > Edit**.
6. Select the **Allow Rescreening** checkbox to enable rescreening and click **Save**.

Pre-populating the screening script

When citizens screen from within a citizen account, you can pre-populate information already known about the citizen performing the screening.

Use the system property *curam.citizenaccount.prepopulate.screening* to set whether the IEG script is pre-populated. The default value of this property is true, which means that the script is pre-populated with information that already known about the citizen.

Related concepts

[Authenticated screening](#)

Citizens who are logged in to Universal Access can complete an authenticated screening.

Resetting data captured from a previous screening

Determine whether starting an intake application resets data captured by a previously completed screening.

Determines whether starting an intake application resets datastore data captured by a previously completed screening

Use the system property **curam.citizenworkspace.intake.resets.screening.results** to determine whether starting an intake application resets datastore data that was captured by a previously completed screening.

Setting **curam.citizenworkspace.intake.resets.screening.results** to true means that starting an intake application resets datastore data captured by a previously completed screening.

Setting **curam.citizenworkspace.intake.resets.screening.results** to false means that starting an intake application does not reset datastore data captured by a previously completed screening.

Writing Rule Sets For Screening

Develop screening rule sets.

Addin a data store schema

Create a new data store schema for use with screening and intake intelligent evidence gathering (IEG) scripts. However, some constraints exist on the format of these schemas. In some cases, requirements dictate that citizens can screen for a program and then follow that screening by applying for benefits.

In many cases, applications are processed by IBM Cúram Social Program Management and are mapped to Cúram cases and evidence by using the Cúram Data Mapping Engine (CDME). In these circumstances, use `CitizenPortal.xsd` as a basis for the schema for screening. This process is used because the same data store schema also needs to be used for intake. In particular, the CDME features do not work correctly if a schema is used that removes or changes the data type of any of the attributes or entities in the `CitizenPortal.xsd` schema.

All schema that follows the pattern of the `CitizenPortal.xsd` schema are safe for later releases. This assurance means that upgrades do not add any new mandatory attributes or entities. Upgrades do not change any existing attributes or entities that currently are required to support existing Cúram data mapping engine functions.

The screening rules interface

All screening rule sets must use the screening rules interface so that they can be executed within IBM Cúram Universal Access.

The ruleset interface is detailed in the following XML example:

```
<?xml version="1.0" encoding="UTF-8"?>

<RuleSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.curamsoftware.com/
  CreoleRulesSchema.xsd"
  name="ScreeningInterfaceRuleSet">

  <!-- This class must be extended by all rule sets invoked by
    the Citizen Portal screening results processing. -->
  <Class name="AbstractScreeningResult" abstract="true">

    <Initialization>
      <Attribute name="calculationDate">
        <type>
          <javaclass name="curam.util.type.Date"/>
        </type>
      </Attribute>
    </Initialization>

    <!-- The programs supported by this Screening Ruleset. -->
    <Attribute name="programs">
      <type>
        <javaclass name="List">
          <ruleclass name="AbstractProgram"/>
        </javaclass>
      </type>

      <derivation>
        <!-- Subclasses of AbstractScreeningResult must override
          this attribute to create a list of the Programs
          supported by the rule set. -->
        <abstract/>
      </derivation>
    </Attribute>
  </Class>

  <!-- This class must be extended by all programs supported
    in the rule set. -->
  <Class name="AbstractProgram" abstract="true">
```

```

<!-- Identifies the program as configured in the Citizen
Portal administration application. -->
<Attribute name="programTypeReference">
  <type>
    <javaclass name="String"/>
  </type>
  <derivation>
    <abstract/>
  </derivation>
</Attribute>

<!-- Whether the claimant is eligible for this program. -->
<Attribute name="eligible">
  <type>
    <javaclass name="Boolean"/>
  </type>
  <derivation>
    <abstract/>
  </derivation>
</Attribute>

<!-- The localizable explanation as to why the claimant is
or is not eligible for this program. May contain HTML
formatting/hyperlinks/etc. -->
<Attribute name="explanation">
  <type>
    <javaclass name="curam.creole.value.Message"/>
  </type>
  <derivation>
    <abstract/>
  </derivation>
</Attribute>
</Class>

</RuleSet>

```

Screening rule sets must include a class that extends the `AbstractScreeningResult` rule class outlined .

Using the `AbstractScreeningResult` rule class guarantees that the required attributes are available when the rules are executed.

Configuring applications

Use IBM Cúram Social Program Management administration and system administration applications to define the applications that are available for citizens. For each application, you can configure the available programs and an application script and data schema. You must also configure the remaining applications details, such as application withdrawal reasons.

Configuring applications in the administration application

Use the IBM Cúram Social Program Management administration application to configure an online application.

Procedure

1. Log in to Social Program Management as an administrator.
2. Select **Administration Workspace > Shortcuts > Universal Access > Applications**.
3. Click **New**. The **New Online Application** page opens.
4. Enter the required information. For more information, see *Configuring application information and display information*, *Configuring scripts*, and *Defining a PDF form*.

Configuring application information and display information

Configure the following information on the **New Online Application** page.

Name

The name of the application that is displayed in the online portal.

Program selection

Indicates whether citizens can select specific programs to apply for or whether they are brought directly into an application script. That is, citizens can apply for all programs associated with the application.

More Info URL

If a URL is defined, a **More Info** link is displayed with the application name so that citizens can find out more information about the selected application.

Client registration

Determines whether citizens are registered as prospect persons or persons.

To determine whether to register citizens as prospect persons or persons, the system checks the client registration configuration in the following two scenarios:

- If **Person Search and Match** is configured, and no match can be found for the citizen.
- If **Person Search and Match** is not configured, that is, citizens on an application are always registered without the system automatically searching and matching them.

If **Client Registration** is not set, the system checks the system property **curam.intake.registerAsProspect** to identify whether citizens are registered as a prospect person or a person.

Submit on Completion Only

Determines whether citizens can submit the application to the agency before completing the intake script.

Defining an icon for an application

If you want an icon displayed with an application, select an icon from the **Icon** selection box.

Note: Alternatively, you could modify the `img src` attribute of the icon directly on the application HTML page, for example

```

```

Summary

A high-level description of the application.

Description

An overview description of the application.

Submission Confirmation Page Details

A more detailed description of the application. Use the **Title** and **Text** fields to define a title and text to be displayed on the **Submission Confirmation** page.

Configuring scripts

Configure an IEG application script to collect the answers to the application questions. Then, configure a submission script for an application so that citizens can submit applications.

Application scripts

Specify a script name in the **Question Script** field. Specify a data store schema in the **Schema** field to store the data entered in the script. On saving the application, an empty template for both the script and schema is created by the system based on the question script and schema specified. You can update these templates from the **Application** tab by selecting the hyperlinks provided on the page. Click the **Question Script** link to start the IEG editor so you can edit the question script. Click the **Schema** link to start the Datastore Editor and edit the schema.

Submission scripts

Configure an IEG submission script in the **Submission Script**. The script defines additional information that does not form part of the application script to be captured, for example, a TANF typically requires information regarding the citizen's ability to attend an interview.

On saving the application, an empty template for the submission script is created by the system based on the Submission Script that you specify. You can update this from the **Application** tab by

selecting the hyperlink on the page. Clicking the link starts the IEG editor that you use to edit the question script.

Defining a PDF form

Define a PDF form that is displayed when citizens complete an online application.

The data that is collected during the online application is copied by the system into a PDF form, which citizens can print. Select the PDF form from the **PDF Forms** drop down menu. If a PDF form is not specified for an application, a default generic PDF form can be used. You can get the default template from the **XSL Templates** section of the system administration application.

The data passed to the XSL template reads from the data store. Instead of displaying the datastore labels in the PDF, define a property file to specify user-friendly names for entities and attributes and to hide entities and attributes that you do not want to display in the PDF. For more information, see *XSL Templates*.

Upload the property file to **Application Resources** in the **Intelligent Evidence Gathering** section of the administration application. For more information, see *Working with Intelligent Evidence Gathering*.

Name the property file using the following convention: <application schema name>PDFProps. The contents of the property file is as follows:

Name an entity

<Entity Name=<Name To Be Displayed in the PDF>, for example, *Application=Intake Application*

Hide an entity

<Entity Name.hidden=true, for example, *ScreeningType.hidden=true*

Hide an attribute

<Entity Name.Attribute Name.hidden=true, for example, *Application.user Name.hidden=true*

Specify a label for an attribute

<Entity Name.Attribute Name=PDF Label, for example, *Submission.dig FirstName=First Name*

Related information

[XSL Templates](#)

[Working with Intelligent Evidence Gathering](#)

Configuring applications in system administration

Use the IBM Cúram Social Program Management system administration application to configure an online application.

Procedure

1. Log in to Social Program Management as a system administrator.
2. Select **System Configurations > Shortcuts > Application Data**.
3. Enter the name of the application property that you want to configure in the **Name** field and select **Search**.
4. Select ... > **Edit Value**.
5. Change the property setting, for example change **YES** to **NO** and **Save** your changes.

Application properties

Configure application properties for an application.

Using *curam.citizenworkspace.authenticated.intake* to mandate authentication before you can apply

If this property set to **YES**, citizens must create an account or log in before they start an application. If this property set to **NO**, citizens are taken directly to the application selection page.

If *curam.citizenworkspace.authenticated.intake* is set to **YES**, citizens are brought to the following components:

- The **Apply for benefits** page.

- The login page when citizens select **Apply**.

Using *curam.citizenworkspace.intake.allow.login* to set Optional authenticated application

If this property is turned on, citizens can choose to log in before they start an application. If this property is turned off, citizens are taken directly to the application selection page.

Using *curam.citizenworkspace.display.confirm.quit.intake* to display a confirmation page to citizens when they quit the application process

If this property set to **YES**, a confirmation page is displayed when citizens quit during the application process. If the system property is set to **NO**, a confirmation page is not displayed when citizens quit an application. This property is only used when the property *curam.citizenworkspace.intake.allow.login* is set to **NO**.

Using *curam.citizenworkspace.intake.enabled* to indicate whether citizens can start the application process from the organization Home page

If this property is set to **YES**, the **Apply For Benefits** link is displayed on the organization **Home** page. If this property is set to **NO**, the applications link is not displayed.

Using *curam.citizenworkspace.intake.submit.intake.mandatory.login* to indicate that citizens must log in before they submit an application

If this property is set to **YES** on, citizens must create an account or log in before they can submit an application. If this property is set to **NO**, citizens can submit an application without logging in.

Using *curam.citizenaccount.prepopulate.intake* to prepopulate the application with information already known about authenticated citizens

The default value of this property is **true**, which means that the script is prepopulated.

Using *Auto-save intake* to mandate whether applications are auto-saved in the citizen account.

Each application is auto-saved when citizens click **Next** as they progress through the IEG script. By default, this property is set to **true**. If this property is set to **false**, applications are not automatically saved in the citizen account.

Configuring other application settings

You can associate programs with an application, define mappings for an application, and configure withdrawal reasons.

Associating programs with applications

Any program that is described in **Configuring Programs** can be associated with an application. When you associate programs with an application, you can set the display order of the selected program relative to other programs associated with the application. For more information, see *Configuring programs*.

Defining mappings for an application

Applications can be processed by IBM Cúram Social Program Management or a remote system.

If the application is processed by IBM Cúram Social Program Management, the entered information is mapped to the evidence tables that are associated with the application case that is defined for the programs that are associated with the application. The mappings are configured for an application by creating a mapping with the Data Mapping Editor. A mapping configuration must be specified in order for the appropriate evidence entities to be created and populated in response to an online application submission.

For more information about the Data Mapping Editor, see *Configuring with the Data Mapping Editor*.

Configuring withdrawal reasons

Citizens can withdraw the application for all or any one of the programs for which they applied.

When they withdraw an application, citizens must specify a withdrawal reason. You can define withdrawal reasons for an application in the **Intake Application** section of the administration application. Before you associate a withdrawal reason with an application, you must define withdrawal reasons in the *WithdrawalRequestReason* code table. For more information, see *Intake Application*.

Related concepts

[Configuring programs](#)

You can configure different types of programs. To configure a program, you configure display and system processing information, local offices, mappings to PDFs, and evidence types.

Related information

[Intake Application](#)

[Configuring with the Data Mapping Editor](#)

Configuring online categories

Online categories group different types of applications or screenings together to make it easier for citizens to find the ones that they need. You must define online categories for screenings and applications to be displayed. After you define online categories, you must associate each screening and application to a category.

Defining online categories

When defining an online category a name and URL must be defined. If a URL is defined a **More Info** link is displayed with the name of the online category allowing citizens to find out more information about the selected category. An order can be assigned to a category which dictates the display order of the selected category relative to other categories.

Associating screenings and applications

Applications and screenings must be associated with an online category so they can be displayed in the application. When associating a screening with an online category, an order can be applied which dictates the display order of the screening relative to other screenings within the same category. When associating an application with an online category an order can be applied which dictates the display order of the application relative to other applications within the same category.

Configuring the citizen account

Although customization is required to modify some citizen account information, you can configure information on the citizen account and the **Contact Information** tab.

Messages can originate as a result of transactions in IBM Cúram Social Program Management or a remote system. Most of the configuration options apply to all messages but there are some configuration options that do not apply to messages originating from a remote system.

Configuring messages

The **Messages** pane on the organization **Home** page displays messages to logged-in citizens. For example, a message that informs citizens when their next benefit payment is due or the amount of the last payment. You can configure a number of items in the **Messages** plane.

Messages can be displayed which relate to meetings, activities, and application acknowledgments. Messages can be displayed as a result of transactions in IBM Cúram Social Program Management or they can originate from remote systems through a web service.

Account messages

Adding an account message or changing a dynamic element of an account message requires customization. You can update the text in the default account messages by using a set of properties for each type of message.

The following properties are available:

- `CitizenMessageMyPayments` - messages about payments.
- `CitizenMessageApplicationAcknowledgement` - messages about application acknowledgments.

- CitizenMessageVerificationMessages - messages about verification messages.
- CitizenMessageMeetingMessages - messages about meetings.
- CitizenMessagesReferral.properties - messages about referrals.
- CitizenMessagesServiceDelivery - messages about service deliveries.
- OnlineAppealRequestMessage - messages about appeal requests.

The properties are in the **Application Resources** section of the administration application. To update the message, each file needs to be downloaded, updated, and uploaded again. The icons that are displayed in the citizen account for each type of message can be configured in the **Account Messages** section of the **administration** application.

Adding a message that originates from a remote system requires that a code table entry is added to the ParticipantMessageType code table and an associated entry in the **Account Messages** listing in the administration application. Messages can then be sent by the ExternalCitizenMessageWS web service.

Creating appeal request acknowledgment or appeal rejection messages

Create messages to acknowledge an appeal request or to reject an appeal request.

<i>Table 2. Appeal request acknowledgment</i>	
Message Area	Description
Title	Appeal Request Acknowledgment
Message	We have received your [Appeal Request - hyperlink to the appeal request on the My Appeals page] and it is currently under review. We will contact you shortly to confirm the next steps.
Effective Date	Current Date.
Duration	This value is defined in the Num.Days.To.Expiry=7 property in the OnlineAppealRequestMessage properties file and used in the implementation to set the attribute expiry date time. The default value is 7.
Notes	None.

<i>Table 3. Appeal rejection</i>	
Message Area	Description
Title	Appeal Request Disallowed
Message	We have reviewed your appeal request and determined it to be an invalid appeal. We will send you written notice of this, including further details.
Effective Date	Current Date.
Duration	This value is defined in the Num.Days.To.Expiry=7 property in the OnlineAppealRequestMessage properties file and used in the implementation to set the attribute expiry date time. The default value is 7.

Table 3. Appeal rejection (continued)	
Message Area	Description
Notes	None.

Creating application acknowledgments

Create messages to acknowledge an application.

Table 4. Application acknowledgment	
Message Area	Description
Title	<Icon> TANF Application Acknowledgment
Message	We have received your TANF Application form. The status of this application is pending. We will contact you when the application has been processed.
Effective Date	Current® date
Duration	An administrator can use a configuration setting to define the number of days (from the effective date) that the message is displayed.
Notes	None.

Creating meeting messages

Create messages for a meeting invitation, a meeting cancellation, and a meeting update. An administrator can use a configuration setting to set the number of days (from the effective date) that the meeting messages are displayed.

Table 5. Meeting invite	
Message Area	Description
Title	<Icon> Meeting Invitation - Meeting with Case Worker
Message 1 (Not an all day meeting and the meeting start and end date are on the same day)	You are invited to attend a meeting from 9.00AM until 5.00PM on 12/04/2010 in Meeting Room 1, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 2 (All day meeting for one day only)	You are invited to attend an all day meeting on 12/04/2010 in Meeting Room 1, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 3 (All day meeting for multiple days)	You are invited to attend an all day meeting each day from 12/04/2010 until 15/04/2010 in Meeting Room 1, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.

Table 5. Meeting invite (continued)

Message Area	Description
Message 4 (Non-all day meeting for multiple days)	You are invited to attend a meeting from 9.00AM until 5.00PM from 12/04/2010 to the 13/04/2010 in Meeting Room 1, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Notes	When the case worker is setting up a meeting, the location is an optional field. Therefore, if a meeting location is not specified, the preceding messages are displayed without a location. Also, the meeting organizer's contact details are optional. Therefore, if no contact details are found, the preceding message is displayed without the organizer's contact details.

Table 6. Meeting cancellation

Message Area	Description
Title	<Icon> Cancellation - Meeting with Case Worker
Message 1 (Not an all day meeting and the meeting start and end date are on the same day)	The meeting that you were scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 is canceled. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information.
Message 2 (All day meeting for one day only)	The all day meeting that you were scheduled to attend on 12/04/2010 is canceled. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information.
Message 3 (All day meeting for multiple days)	The all day meeting that you were scheduled to attend from 12/04/2010 until 15/04/2010 is canceled. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information.
Effective Date	Current Date.
Notes	The meeting organizer's contact details link opens a page that shows the organizer's contact details.

Table 7. Meeting update

Message Area	Description
Title	<Icon> Cancellation - Meeting with Case Worker

Table 7. Meeting update (continued)	
Message Area	Description
Message 1 (Date and Time change of a non-all-day meeting)	The meeting that you were scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 is rescheduled to 3.00PM until 7.00 PM on 13/04/2010 in Meeting Room 1, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 2 (Location change of a non-all-day meeting)	The location of the meeting you are scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 is changed. This meeting is now scheduled for Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 3 (Date, time, and location change of non-all-day meeting)	The meeting that you were scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 is rescheduled to 3.00PM until 7.00 PM on 13/04/2010. It is rescheduled for Meeting Room 2, Block C. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 4 (Date change of all day meetings for multiple days)	The all day meeting that you are scheduled to attend from 12/04/2010 until 15/04/2010 is rescheduled. This meeting will now take place from 13/04/2010 until 16/04/2010. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 5 (Location change for all day meeting for multiple days)	The location of the all day meeting you are scheduled to attend from 12/04/2010 until 15/04/2010 is changed. This meeting is rescheduled for Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 6 (Date and location change for all-day meeting for multiple days)	The all day meeting that you are scheduled to attend from 12/04/2010 until 15/04/2010 is rescheduled. This meeting will now take place from 13/04/2010 until 16/04/2010 in Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.

Table 7. Meeting update (continued)

Message Area	Description
Message 7 (Date change for an all-day meeting)	The all day meeting that you are scheduled to attend on 12/04/2010 is rescheduled. This meeting will now take place on 13/04/2010. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 8 (Location change for an all-day meeting)	The location of the all day meeting you are scheduled to attend on 12/04/2010 is changed. This meeting is rescheduled for Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 9 (Date and location change for an all-day meeting)	The all day meeting that you are scheduled to attend on 12/04/2010 is rescheduled. This meeting is rescheduled for 13/04/2010 in Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 10 (Date and time change of a non-all-day meeting for multiple days)	The meeting that you are scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 until 15/04/2010 is rescheduled. This meeting is rescheduled for 2.00PM until 6.00 PM on 13/04/2010 until 16/04/2010. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 11 (Location change of a non-all-day meeting for multiple days)	The location of the meeting you are scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 until 15/04/2010 is changed. This meeting is rescheduled for Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.
Message 12 (Date, time, and, location change of non-all-day meeting for multiple days)	The meeting that you are scheduled to attend from 2.00PM until 6.00 PM on 12/04/2010 until 15/04/2010 is rescheduled. This meeting is rescheduled for 2.00PM until 6.00 PM on 13/04/2010 until 16/04/2010 in Meeting Room 1, Block D. Please contact Joe Bloggs at 014567832 or joe@SemAgency.com if you need more information or cannot attend.

Table 7. Meeting update (continued)	
Message Area	Description
Notes	When the case worker is setting up a meeting, the location is an optional field. Therefore, if a meeting location is not specified, the preceding messages are displayed without a location. Also, the meeting organizer's contact details are optional. Therefore, if no contact details are found, the preceding message is displayed without the organizer's contact details.

Creating payment messages

Create messages for an issued payment, a canceled payment, a due payment, a stopped payment, an unsuspended payment, an issued overpayment, and an issued underpayment. An administrator can use a configuration setting to set the number of days (from the effective date) that the payment messages are displayed.

Table 8. Payment issued	
Message Area	Description
Title	<Icon> Latest Payment
Message 1	Your latest payment of \$22.00 was due on 22/07/2009. Click here to view the payment details. Your next payment is due on 29/07/2009. Click My Payments to view your payment history.
Message 2 (Payment previously canceled)	Your latest payment of \$22.00 was due on 22/07/2009. Click here to view the payment details. This payment was originally canceled on 23/07/2009. Click here to view details of the canceled payment. Your next payment is due on 29/07/2009. Click My Payments to view your payment history.
Effective Date	Current Date.
Notes	<p>A payment can be issued, then canceled, and then reissued. The here hyper link opens a page that shows payment details. The My Payments link opens the My Payments page in the Citizen Account.</p> <p>Note: If no more payments are due, the Your next payment is due on 29/07/2009 part of the messages is not displayed.</p>

Table 9. Payment canceled	
Message Area	Description
Title	<Icon> Payment Canceled

Table 9. Payment canceled (continued)

Message Area	Description
Message	Your payment of \$22.00, due on 22/07/2009, has been canceled. Click here to view the details. Click Contact Information to contact your caseworker if you need more information. Your next payment is due on 29/07/2009. Click My Payments to view your payment history.
Effective Date	Current Date.
Notes	If no more payments are due, the Your next payment is due on 29/07/2009 part of the message is not displayed. The Contact Information link opens the Contact Information tab in the citizen account. The My Payments link opens the My Payments page in the Citizen Account.

Table 10. Payment due

Message Area	Description
Title	<Icon> Next Payment Due
Message	Your next Cash Assistance payment is due on 29/07/2011.
Effective Date	Current Date.
Notes	This message is appropriate when it is the first payment that a citizen receives.

Table 11. Case suspended

Message Area	Description
Title	<Icon> Payments Stopped
Message	Your Cash Assistance payments have been stopped from 29/07/2009. Click Contact Information to contact your caseworker if you need more information.
Effective Date	Current Date.
Notes	The Contact Information link opens the Contact Information tab in the Citizen Account.

Table 12. Case unsuspended

Message Area	Description
Title	<Icon> Payments Unsuspended

Table 12. Case unsuspended (continued)	
Message Area	Description
Message	Your Cash Assistance payment suspension has been lifted from 29/07/2009. Your next payment is due on 31/07/2009.
Effective Date	Current Date.
Notes	None.

System messages

Agencies use system messages to send messages to citizens who have a citizen account. For example, an agency might want to provide information and helpline numbers to citizens who are affected by a natural disaster. You can configure system messages in the administration application on the **New System Message** page.

Use the **Title** and **Message** fields to define the title of the message and the message body that is displayed in the **My Messages** pane. If you define the message as a priority with the **Priority** field, the message appears first in the messages listing.

Note: If multiple priority messages exist, the effective date of the message and the message type determines the message order. For more information, see *Ordering and Enabling/Disabling Messages*.

Use the **Effective Date and Time** to define an effective date for the message, such as when the message is displayed in the citizen account. Use the **Expiry Date and Time** field to define an expiry date for the message, for example, when to remove the message from the Citizen Account.

The message is saved with a status of **In-Edit**. Before the message is displayed in the Citizen Account, it must be published. After it is published, the message is active and is displayed in the Citizen Account based on the effective and expiry dates defined.

Configuring message duration

System properties set the length of time a type of message is displayed in the citizen account. For example, a payment message can be configured to be displayed for 10 days. These configuration options apply only to messages that originate as a result of transactions on IBM Cúram Social Program Management.

The following system properties are provided:

- `curam.citizenaccount.payment.message.expiry.days` - sets the number of days from the effective date that a payment message is displayed in the citizen account. A payment message is displayed for this duration unless another payment message is created which replaces it. The default value is 10.
- `curam.citizenaccount.intake.application.acknowledgement.message.expiry.days` - sets the number of days from the effective date that an application acknowledgment message is displayed in the citizen account. An acknowledgment message is displayed for this duration unless another acknowledgment message is created which replaces it. The default value is 10.
- `curam.citizenaccount.meeting.message.effective.days` - sets the number of days from the effective date that a meeting message is displayed. A meeting message is displayed for this duration unless another meeting message is created which replaces it. The default value is 10.

Switching off messages

An agency might not want to display messages in the Citizen Account. To cater for this choice, the system property `curam.citizenaccount.generate.messages` enables an agency to switch all messages *on* or *off*. The default value is *true*, which means that messages are generated and displayed in the Citizen Account.

Configuring last logged in information

The text displayed in the welcome message and last logged on information can be updated using the properties that are stored in the `CitizenAccountHome` properties file stored in the **Application Resource** section of the Administration Application.

The following properties are provided:

- `citizenaccount.welcome.caption` - updates the welcome message.
- `citizenaccount.lastloggedon.caption` - updates the last logged on message.
- `citizenaccount.lastloggedon.date.time.text` - updates the date and time message.

Configuring contact information

Configure contact information for citizens and caseworkers.

Contact information displayed in the citizen account displays contact details (phone numbers, addresses and email addresses) stored for the logged in citizen and also caseworker contact details (business phone number, mobile phone number, pager, fax and email) of the case owners of cases associated with the logged in citizen in IBM Cúram Social Program Management and on remote systems.

Citizen contact information

The following system property is provided that sets whether contact information is displayed to a citizen.

`curam.citizenaccount.contactinformation.show.client.details`

If the property is set to `true`, citizens' address, phone number, and email address are displayed. If this property is set to `false`, contact information is not displayed. The default value for this property is `true`.

Caseworker

The following system properties are provided to set whether agency worker contact information is displayed to a citizen, and if displayed, additional system properties are provided to dictate the type of contact information displayed:

`curam.citizenaccount.contactinformation.show.caseworker.details`

Sets whether worker contact details are displayed in the citizen account. If this property is set to `true`, worker contact details of cases associated with the logged in citizen are displayed. If this property is set to `false`, worker contact information is not displayed. The default value for this property is `true`.

`curam.citizenaccount.contactinformation.show.businessphone`

Sets whether the worker's business phone number is displayed. The default value of this property is `true`.

`curam.citizenaccount.contactinformation.show.mobilephone`

Sets whether the worker's mobile number is displayed. The default value of this property is `true`.

`curam.citizenaccount.contactinformation.show.emailaddress`

Sets whether the worker's email address is displayed. The default value of this property is `true`.

`curam.citizenaccount.contactinformation.show.faxnumber`

Sets whether the worker's fax number is displayed. The default value of this property is `true`.

`curam.citizenaccount.contactinformation.show.pagernumber`

Sets whether the worker's pager is displayed. The default value of this property is `true`.

`curam.citizenaccount.contactinformation.show.casemember.cases`

Sets whether the worker's contact information is displayed for cases where the citizen is a case member. If this property is set to `true`, cases where the citizen is a case member are displayed. If this property is set to `false`, then only cases where the citizen is the primary client are displayed. Note: this property only applies to cases originating from IBM Cúram Social Program Management. The types of product deliveries and integrated cases to be displayed can be configured in the Product section of the Administration Application. For more information on administering this see the *Cúram Integrated Case Management Configuration Guide*.

Configuring user session timeout

Configure the user session timeout modal in the System Administration application and the Universal Access Responsive Web Application so that citizens know when their session is about to expire.

If a user session is inactive for a while, citizens can continue their current session by clicking **Stay logged in** so that they don't lose information that they entered on the current page. Citizens can also continue the current session by navigating away from the **Stay logged in** button.

If citizens do not continue their session, they are logged out automatically after a configurable period of time to secure their personal information.

Use the following properties to configure the session timeout:

curam.environment.enable.timeout.warning.modal

You can enable or disable the session timeout feature. For more information, see *Customizing the session timeout warning in Universal Access*.

curam.environment.timeout.warning.modal.time

You can configure the maximum time that the **Stay logged in** dialog is displayed to citizens. For more information, see *Customizing the session timeout warning in Universal Access*.

REACT_APP_SESSION_INACTIVITY_TIMEOUT

In the Universal Access Responsive Web Application, use the **REACT_APP_SESSION_INACTIVITY_TIMEOUT** environment variable to configure the time in seconds before a user session expires. You can set the environment variable in the `.env` or `.env.development` files in the root of your application. The value must match the session timeout that is configured on the server, by default, 30 minutes or 1800 seconds.

Configuring the dialog box text

To configure the dialog box title, informational text, or button text for the Universal Access Responsive Web Application, use the `SessionTimeoutDialogComponentMessages.js` file that accompanies the source files. For more information, see *Customizing the IBM Cúram Universal Access server*.

Configuring the login page to notify citizens when their session times out

Use the **sessionCountdownTimerEnd** property on the router location state to update a customized login page with a message to notify citizens when their session times out. For more information about routing, see [“Developing with routes”](#) on page 88.

An example of the **sessionCountdownTimerEnd** is shown:

```
if (location.state.sessionCountdownTimerEnd) {  
  <Alert .../>  
}
```

This notification message is configured by default when a citizen's session times out.

Related concepts

[Developing with routes](#)

Routes define the valid endpoints for navigation in your application. Your application consists of a network of routes that can be traversed by your users to access the application's pages.

Related tasks

[Customizing IBM Cúram Universal Access](#)

Use this information to customize Universal Access for your organization.

Related information

[Customizing the session timeout warning in Universal Access](#)

Configuring appeal requests

Complete the following steps to enable a citizen to request an appeal from their citizen account.

Procedure

1. Create a custom IEG script and data store schema to capture the appeal information.
2. Set the values of the `curam.citizenworkspace.appeals.datastore.schema` and the `curam.citizenworkspace.appeals.datastore.script.id` properties to the values of the script and data store schema that you created.
3. Create an XSL template to generate a PDF of the appeal information.

Related tasks

[Customizing IEG forms in the Universal Access Responsive Web Application](#)

Universal Access can use forms to gather information about citizens, such as applying for benefits or screening for programs. Where you need to save customer data as evidence, you can create forms in Intelligent Evidence Gathering (IEG). IEG is a framework for creating dynamic and conditional questionnaires and saving the input data as evidence. You can customize IEG forms for your organization in the Universal Access Responsive Web Application. The default connectivity handling helps to prevent citizens losing data in IEG forms by preventing them from leaving pages with unsaved changes.

[Customizing appeals](#)

You can customize appeals to suit your organization. You can integrate with an appeals system of your choice. If you are licensed for the IBM Cúram Appeals application module, the IBM Cúram Social Program Management appeals functionality is available on installation.

Configuring communications on the Notices page

You can configure the maximum number of communications that are displayed on the **Notices** page. By default, up to 20 communications are displayed.

Procedure

Edit the `curam.citizenaccount.max.communication` system property and specify the maximum number of communications to display.

What to do next

You can further customize the underlying communications implementation if needed. For more information, see [“Customizing the Notices page” on page 210](#).

Related concepts

[The Notices page](#)

When a citizen is logged in, they can see all communications that are relevant to them on the **Notices** page, with sent, received, or normal status indicated. Notices are typically formal written communications that are issued to meet legal, regulatory, or state requirements, which are created by using letterhead templates.

[Customizing the Notices page](#)

By default, the notices relevant to the linked user are listed on the **Notices** page. You can replace the default `CitizenCommunicationsStrategy` implementation with your own custom implementation.

Configuring life events

For each life event, you must define how information is collected, stored, and displayed. You can configure life event information categories, mappings to dynamic evidence, and information sharing with internal and external sources.

Life events are displayed in the citizen account to allow citizens to submit information to the agency. Life events can also provide citizens with useful information and resources. Life events can be made available

in other channels. For example, they can be submitted online by an agency worker in the internal application. Configuration settings allow different information to be displayed depending on where the life event is initiated from. For example, the **Having a Baby** life event question script that is displayed to citizens can be different from the **Having a Baby** life event question script that is displayed to an agency worker.

Configuring a life event

You can configure a life event in the administration application on the **New Life Event** page.

Defining a name

Specify a name that uniquely identifies the life event. This name is only displayed in the administration application. You must specify a schema if the life event enables citizens to submit information to the agency. The schema defines where the information submitted by a citizen or user in the life event script is stored. For more information about defining data store schemas, see *Working With Intelligent Evidence Gathering*.

Defining a channel type

The channel type defines the channel in which a life event is used, for example, 'Online' or 'Internal'.

Defining a display name

The display name represents the name of the life event that appears citizens or agency workers. For example, a change of job life event might be displayed as **Lost My Job** to citizens but **Client Loses Job** to caseworkers.

Displaying question and answer scripts

Question script is the name of the life event script. Answer script gathers answers to life event questions.

Defining a schema

The name of the data store schema used by the life event script to capture data. Select a schema from the **Schema** menu.

Defining the display ruleset

Define the ruleset that determines which recommendations are displayed to citizens when a life event is submitted.

Enabling citizen consent

For certain life events, a citizen's consent might be needed before information is sent to a remote system or agency. The **Citizen Consent Enabled** selection box allows an administrator to specify whether a citizen's consent is needed. This provision means that citizens can select the agencies that they would like to send their life event information to.

If this indicator is specified, a list of remote systems is displayed on completion of the life event script. If this indicator is not specified, the citizen is not presented with the list. If only one remote system is associated with the life event, the **Citizen Consent If One Choice Only** field is provided to determine whether the citizen is presented with the remote systems list. The citizen must specify their consent to send information to this remote system by selecting it on completion of the question script.

Defining the channel

The channel that this life event applies to, either online or internal.

Defining a display description

A description of the life event. This description is displayed on the cards on the citizen's profile page. Rich text is supported.

Defining additional information

Additional information related to the life event can be specified. For example, you can display links to useful websites or information that the agency deems relevant to a particular life event.

Defining the submission text

Configure the text to be displayed to a citizen after they submit a life event. If a rule set was defined, the following default text is displayed:

Your information has been submitted. Based on the information you have given us, we have identified services and programs that may be of use to you. View your results.

Defining an icon

You cannot define an icon when first configuring a life event. Instead, you must save the life event and then take the following steps:

1. Select the ellipses ... icon for the new life event and then select **New Image**.
2. Select **Browse**, and select an image file from your local drive.

Note: Only .png or .gif images are supported. Image files must not be animated.

3. Specify an image name and alt text and select **Save**.

Related information

[Working with Intelligent Evidence Gathering](#)

Mapping life event information to evidence entities

Information that is gathered in the life event script is stored in the data store schema that is defined for the life event.

To pass information gathered in the life event script into IBM Cúram Social Program Management, it must be mapped to dynamic evidence entities. Dynamic evidence entities must first be defined in the **Rules and Evidence** section of the administration application. When defined, you must specify these entities as **Social Record Evidence Types** in the administration application. An indicator is also provided to set if a particular evidence type is visible to citizens. When the social record evidence entities are defined, use the Data Mapping Editor to map the data from the data store to the appropriate evidence entities. You can access the Data Mapping Editor from the **Mappings** tab on the life event.

When citizens submit a life event, the information that is gathered is mapped to evidence entities that are associated with a new case type called a social record case. The evidence broker can then be used to pass the information from this case to the appropriate ongoing cases.

For more information about dynamic evidence, see the *Configuring dynamic evidence* related link. For more information about data mapping, see the *Configuring with the data mapping editor* related link. For more information about sharing evidence, see the *Sharing evidence with the evidence broker* related link.

Related information

[Configuring dynamic evidence](#)

[Configuring with the Data Mapping Editor](#)

[Sharing evidence with the evidence broker](#)

Defining a question script, answer script, and schema

You must define an IEG script for the life event if the life event allows citizens or users to submit information to the agency.

The IEG script that you define collects the answers to a set of questions related to the life event. Specify a script name in the **Question Script** field. You must also specify a schema if the life event allows citizens or users to submit information to the agency. The schema defines where the information submitted in the life event script is stored. Specify a schema in the **Schema** field. You must specify an answer script to allow citizens to review the answers they have provided to the questions during submission of the life event. Specify an answer script in the **Answer Script** field.

When you save the life event, empty template scripts and a schema are created by the system based on the Question Script, Answer Script and Schema specified. You can then update these from the **Life Event** tab by selecting the hyperlinks provided on the page. Clicking on the **Question Script** and **Answer Script** links launch the IEG Editor. Clicking on the **Schema** link starts the Datastore Editor. Existing schema, question scripts and answer scripts can be used by selecting them on the **Edit Life Event** page.

Note: If a life event has been configured to send information to remote systems, set the **Finish Page** field in the script properties (accessed by selecting **Edit > Configure Script Properties** in the IEG Editor) to `cw/DisplayRemoteSystems.jspx`.

For more information on defining IEG scripts and schema, see *Working with Intelligent Evidence Gathering Guide*.

Related information

[Working with Intelligent Evidence Gathering](#)

Categorizing life events

Life event administration allows you to categorize or group together similar life events, for example, changing jobs, changing address and changing income life events could be categorized within an employment category.

Categorizing life events makes it easier for citizens or users to find the life event they need. You define categories in life event administration and then associate them with a life event. When defining a category, you must specify a name and description. Life events can then be associated with that category.

Defining Remote Systems

Life event information can be submitted to remote or external systems. You must associate a remote system with a life event so that life event information can be sent to that system.

The remote system must have the **Life Event Service** web service associated with it. This is used to transmit life event information to the remote system. Remote Systems can be configured in the Remote Systems section of the administration application.

Chapter 5. Developing with the Universal Access Responsive Web Application

Find out how to use the provided development resources to customize the IBM Universal Access Responsive Web Application reference application and build your custom application.

Installing the IBM Universal Access Responsive Web Application development environment

The IBM Universal Access Responsive Web Application requires a React JavaScript development environment in addition to the IBM Cúram Social Program Management Java development environment.

Note: The IBM Universal Access Responsive Web Application installation includes the IBM Social Program Management Design System so you don't need to install the design system separately.

Prerequisites and supported software

Before you install or upgrade, review the prerequisites and supported software to ensure compatibility.

IBM Cúram Social Program Management Platform and the IBM Cúram Universal Access application module

The IBM Universal Access Responsive Web Application client asset is released at more frequent intervals and requires specific IBM Cúram Social Program Management versions to benefit from server-side enhancements, security updates, and defect fixes.

Note:

- From IBM Universal Access Responsive Web Application 2.7.0 onwards, new features, and server-side enhancements and defect fixes, are supported only in the most recent IBM Cúram Social Program Management version line, that is, 7.0.10 or later.
- IBM Universal Access Responsive Web Application 2.6.0 continues to be supported only for security updates and critical defect fixes on older compatible versions of IBM Cúram Social Program Management, that is, 7.0.9 or earlier.

For more information about the support strategy, see [“Citizen Engagement support strategy”](#) on page 247.

Table 13. Compatibility with IBM Cúram Social Program Management	
IBM Universal Access Responsive Web Application asset	IBM Cúram Social Program Management Platform and IBM Cúram Universal Access application module
2.7.0	<ul style="list-style-type: none">• 7.0.10 for all new features, enhancements, and defect fixes in 2.7.0.

Table 14. Compatibility with IBM Cúram Social Program Management

IBM Universal Access Responsive Web Application asset	IBM Cúram Social Program Management Platform and IBM Cúram Universal Access application module
2.6.0	<ul style="list-style-type: none"> • 7.0.9 Refresh Pack iFix 1 for all new features, enhancements, and defect fixes in 2.6.0. • 7.0.8 Refresh Pack iFix 1 for security updates and critical defect fixes only. • 7.0.4.0 Refresh Pack and 7.0.4.4 Fix Pack iFix 1 for security updates and critical defect fixes only.
2.5.1	<ul style="list-style-type: none"> • 7.0.9 Refresh Pack iFix 1 for all new features, enhancements, and defect fixes in 2.5.1. • 7.0.8 Refresh Pack iFix 1 for security updates and critical defect fixes only. • 7.0.4.0 Refresh Pack and 7.0.4.4 Fix Pack iFix 1 for security updates and critical defect fixes only.
2.5.0	<ul style="list-style-type: none"> • 7.0.9 Refresh Pack iFix 1 for all new features, enhancements, and defect fixes in 2.5.0. • 7.0.8 Refresh Pack iFix 1 for security updates and critical defect fixes only. • 7.0.4.0 Refresh Pack and 7.0.4.4 Fix Pack iFix 1 for security updates and critical defect fixes only.
2.4.1	<ul style="list-style-type: none"> • 7.0.8 Refresh Pack iFix 1 for all new features, enhancements, and defect fixes in 2.4.1. • 7.0.4.0 Refresh Pack and 7.0.4.4 Fix Pack for security updates and critical defect fixes only.

Note: Universal Access does not support the dual deployment of the classic client application and the IBM Universal Access Responsive Web Application client against the same instance of the IBM Cúram Social Program Management server. You can build and deploy your server without the classic client application as described in [Alternative Targets for IBM WebSphere® Application Server](#) or [Multiple EAR files for Oracle WebLogic Server](#). Alternatively, you must use another strategy to block access to the classic client application URLs to ensure that users cannot concurrently access both clients.

Platforms

There is no dependency on specific hardware platforms, but some minimum hardware requirements apply:

- Desktop devices that meet Microsoft Windows 10 specifications.
-

Development tools

Node.js is a prerequisite for installing the IBM Universal Access Responsive Web Application and for developing and deploying your web applications

Supported software	Version	Prerequisite minimum	Operating system restrictions
Node.js	10 LTS and future fix packs	10.13.0 LTS (Dubnium)	No
	12 LTS and future fix packs	12.13.0 LTS (Erbium)	No

Interactive Development Environment (IDE)

The Universal Access Responsive Web Application does not depend on a specific IDE, you can choose your own. There are many IDEs that you can choose, for example Microsoft Visual Studio Code, Atom, and Sublime. However, IBM uses Microsoft Visual Studio Code to develop the reference application, it supports many plugins that make development faster and easier, for example it supports the following tools:

- Linting tools (ESLint)
- Code formatters (Prettier)
- Debugging tools (Debugger for Chrome)
- Documentation tools (JSDoc)

IBM does not own, develop, or support these tools.

Application server, web server, and DBMS

Deploying the Universal Access Responsive Web Application requires a web server in the IBM Cúram Social Program Management topology. The following application server, web server, and DBMS combinations are supported for developing and deploying your custom application.

- IBM WebSphere Application Server, IBM HTTP Server/Apache HTTP Server, and IBM Db2®
- IBM WebSphere Application Server, IBM HTTP Server/Apache HTTP Server, and Oracle Database
- Oracle WebLogic Server, Oracle HTTP Server/Apache HTTP Server, and Oracle Database

For more information about installing an application server for IBM Cúram Social Program Management, see [Installing an enterprise application server](#).

HTTP servers

These HTTP servers are supported for deployment.

Supported software	Version	Prerequisite minimum	Operating system restrictions
IBM HTTP Server	9.0	9.0.0.5	No
	8.5.5	8.5.5.9	No
Oracle HTTP Server	12.2.1.3.0 and future fix packs	12.2.1.3.190808	No
Apache HTTP Server	2.4 and future patches	2.4	No

Web browsers

IBM Cúram Universal Access with the Universal Access Responsive Web Application is developed for public-facing applications. Every effort was made to ensure that the application pages use standard web technologies and formats, which should be compatible with all browsers that are listed. However, the browsers that are listed in the following table are the only browsers that are officially supported.

Note: The browser **Back** and **Forward** buttons, and browser refresh, are now supported on IEG pages in the Universal Access Responsive Web Application. Information that is entered in IEG forms is now retained when the citizen clicks **Next** or goes back or forward through a form.

Chrome, Firefox, Edge, and Safari release new versions more frequently than Internet Explorer, and they install updates automatically by default. Universal Access Responsive Web Application releases are tested on the latest browser versions that are available at the start of the IBM development cycle.

Note: Only stable Chrome releases are tested.

If no issues result from the tests, IBM certifies the browser version.

For each new product release, the prerequisites list the version that is certified. If IBM cannot certify that version for any reason, you might need to revert to a previous version that is fully certified. While IBM supports customers who use newer versions of these browsers than the last certified version, customers must understand that the versions are not fully tested.

Supported software	Version	Operating system restrictions
Apple Safari	13 and future fix packs	No
Google Chrome	80 and future fix packs	No
Microsoft Edge	44 and future fix packs	No
Microsoft Internet Explorer	11 and future fix packs	No
Mozilla Firefox	73 and future fix packs	No

Accessibility

This accessibility software is supported.

Supported software	Version	Prerequisite minimum	Operating system restrictions	Browser
Freedom Scientific JAWS screen reader	2019 and future fix packs	2019	No	Microsoft Internet Explorer 11
Apple VoiceOver	13.2.2 and future fix packs	13.2.2	No	Table accessibility is certified on iOS 13.2.2 with Chrome 80

Note: The combination of Internet Explorer 11 and JAWS 2019 is the only certified screen reader and browser combination.

Previous versions

To see the prerequisites and supported software for previous versions, see the [IBM Cúram Social Program Management PDF library](#).

Installing the IBM Cúram Universal Access development environment

You can install a lightweight or a full development environment. The IBM Social Program Management Design System is installed as part of the IBM Universal Access Responsive Web Application installation and doesn't need a separate installation.

Before you begin

Lightweight development environment

For quick and easy installation, install the Universal Access React application plus the design system and Universal Access packages. Then use the `universal-access-mocks` package to provide mock data specific to Universal Access business scenarios for testing purposes. `universal-access-mocks` is consumed by the mock server to provide mock APIs in the development environment so that you do not have to host an IBM Cúram Social Program Management server during development.

Full development environment

Install the Universal Access React application plus the design system and Universal Access packages. Then, install the SPM Java Application Development Environment (ADE) to develop and test your APIs instead of using the `universal-access-mocks` package. For more information about installing the SPM Java ADE, see [Installing a development environment](#).

Note: If you are working with a non-English version, you must ensure that the appropriate language is installed on IBM Cúram Social Program Management.

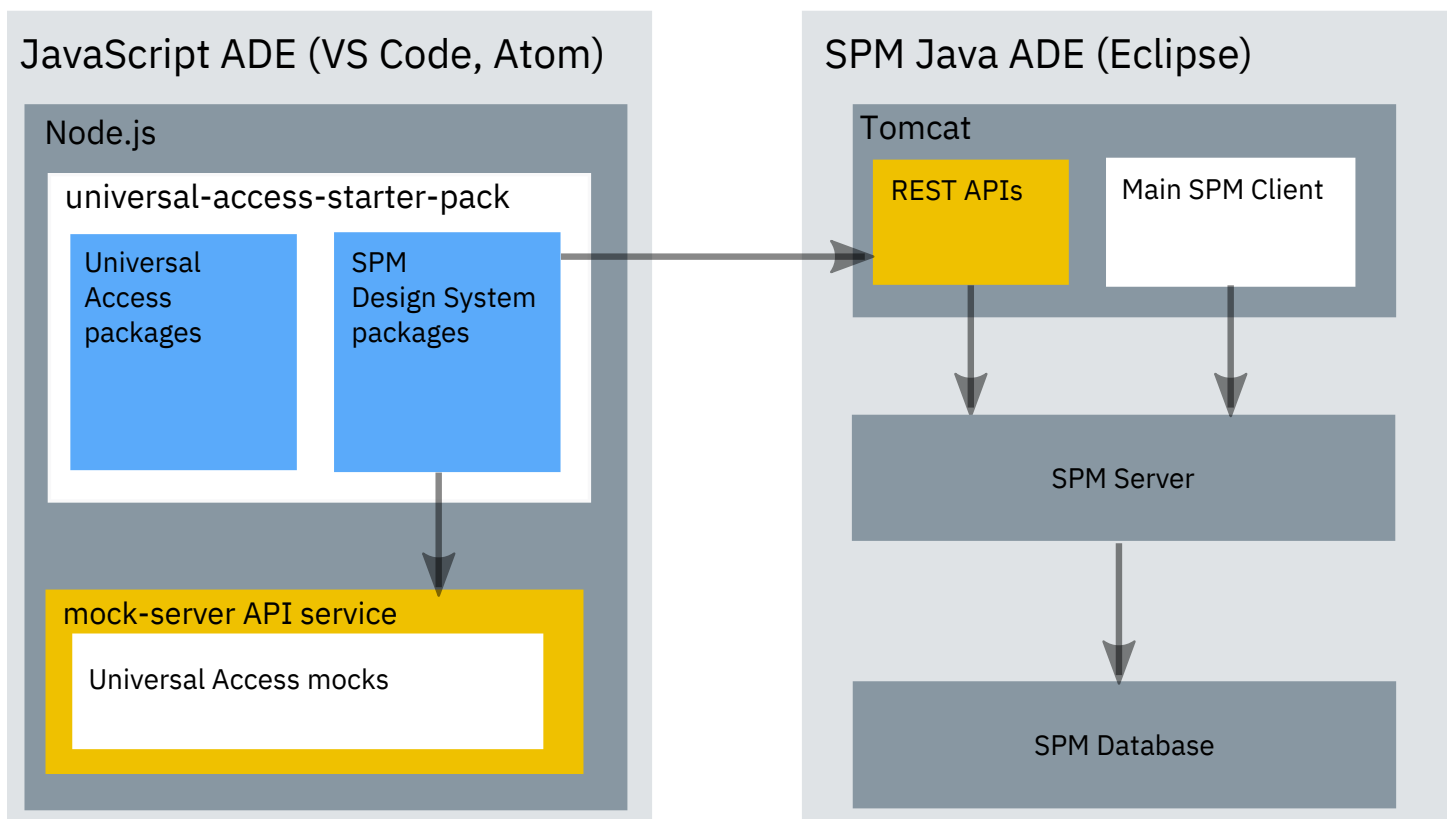


Figure 3. SPM React and Java application development environments

About this task

For the IBM Universal Access Responsive Web Application, first extract the `spm-universal-access-starter-pack` React starter application. Then install all of the IBM Social Program Management Design System and IBM Universal Access Responsive Web Application Node packages into the starter application.



Attention: When working with npm packages, it is important that you familiarize yourself with the npm ecosystem and how package dependencies work, so that you can adopt a suitable security strategy for your project needs.

Procedure

1. Download the Universal Access Responsive Web Application and IBM Social Program Management Design System Node packages.
 - a) Open IBM Fix Central, select **Cúram Social Program Management**, select your installed version and platform, and click **Continue**.
 - b) Ensure that **Browse for fixes** is selected, and click **Continue**.
 - c) Select the check boxes for IBMUniversalAccessResponsiveWebApplication and IBMSocialProgramManagementDesignSystem and click **Continue**.
 - d) Only versions that are compatible with your IBM Cúram Social Program Management version are shown. Download SPM_DS_<version>.zip and UA_Web_App_<version>.zip and extract the packages in the archive files to any directory.

2. Extract the spm-universal-access-starter-pack_<version>.tgz file.

The extracted package directory forms the React starter application. You must install all of the other packages into this directory.

3. Rename the extracted package directory to reflect your project. For example, universal-access-custom-app.
4. From your custom application directory, install the IBM Social Program Management Design System Node packages by entering the following commands:

```
npm install <path>/govhhs-govhhs-design-system-core-<version>.tgz
npm install <path>/govhhs-govhhs-design-system-react-<version>.tgz
npm install <path>/spm-core-<version>.tgz
npm install <path>/spm-core-ui-<version>.tgz
npm install <path>/spm-core-ui-locales-<version>.tgz
npm install <path>/spm-intelligent-evidence-gathering-<version>.tgz
npm install <path>/spm-intelligent-evidence-gathering-locales-<version>.tgz
```

Where <path> is the download path and <version> is the package version.

Enter the remaining commands in this order:

```
npm install <path>/spm-eslint-config-<version>.tgz
npm install <path>/spm-test-framework-<version>.tgz
npm install <path>/spm-web-dev-accelerator-scripts-<version>.tgz
npm install <path>/spm-web-dev-accelerator-<version>.tgz
```

Note: Ignore any Node package dependency warnings for now. If needed, you can resolve them later.

5. From your custom application directory, install the IBM Universal Access Responsive Web Application Node packages by entering the following commands. Ignore any warnings for now.

```
npm install <path>/spm-universal-access-<version>.tgz
npm install <path>/spm-universal-access-ui-<version>.tgz
npm install <path>/spm-universal-access-ui-locales-<version>.tgz
npm install <path>/spm-mock-server-<version>.tgz
npm install <path>/spm-universal-access-mocks-<version>.tgz
```

Where <path> is the download path and <version> is the package version.

6. Run the following command to install the package dependencies.

```
npm install
```

7. You can run the Universal Access starter application by entering the following command from your application directory.

```
npm start
```

If the local host does not start automatically, browse to <http://localhost:3000/> to see the running application.

Results

You can now start to customize the Universal Access reference application for your organization.

Upgrading the IBM Universal Access Responsive Web Application

You can upgrade your custom React application with the latest versions of the IBM Universal Access Responsive Web Application and IBM Social Program Management Design System Node packages to benefit from the most recent updates.

Before you begin

Before you upgrade, ensure that you review your custom application for any potential upgrade impacts. For more information, see [“Developing compliantly” on page 71](#).

Procedure

1. Download the IBM Universal Access Responsive Web Application and IBM Social Program Management Design System Node packages.
 - a) Open [IBM Fix Central](#), select **Cúram Social Program Management**, select your installed version and platform, and click **Continue**.
 - b) Ensure that **Browse for fixes** is selected, and click **Continue**.
 - c) Select the check boxes for `IBMUniversalAccessResponsiveWebApplication` and `IBMSocialProgramManagementDesignSystem` and click **Continue**.
 - d) Only versions that are compatible with your IBM Cúram Social Program Management version are shown. Download `SPM_DS_<version>.zip` and `UA_Web_App_<version>.zip` and extract the packages in the archive files to any directory.
2. Read all relevant [Chapter 2, “IBM Cúram Universal Access release notes,” on page 2](#) to review the changes between your current version and the new version .
3. Extract the `universal-access-starter-pack` package to a temporary directory and compare it to your working custom application directory. Apply any differences you find to your custom application directory.
4. From your custom application directory, install the IBM Social Program Management Design System Node packages by entering the following commands:

```
npm install <path>/govhhs-govhhs-design-system-core-<version>.tgz
npm install <path>/govhhs-govhhs-design-system-react-<version>.tgz
npm install <path>/spm-core-<version>.tgz
npm install <path>/spm-core-ui-<version>.tgz
npm install <path>/spm-core-ui-locales-<version>.tgz
npm install <path>/spm-intelligent-evidence-gathering-<version>.tgz
npm install <path>/spm-intelligent-evidence-gathering-locales-<version>.tgz
```

Where `<path>` is the download path and `<version>` is the package version.

Enter the remaining commands in this order:

```
npm install <path>/spm-eslint-config-<version>.tgz
npm install <path>/spm-test-framework-<version>.tgz
npm install <path>/spm-web-dev-accelerator-scripts-<version>.tgz
npm install <path>/spm-web-dev-accelerator-<version>.tgz
```

Note: Ignore any Node package dependency warnings for now. If needed, you can resolve them later.

5. From your custom application directory, install the IBM Universal Access Responsive Web Application Node packages by entering the following commands. Ignore any warnings for now.

```
npm install <path>/spm-universal-access-<version>.tgz
npm install <path>/spm-universal-access-ui-<version>.tgz
npm install <path>/spm-universal-access-ui-locales-<version>.tgz
npm install <path>/spm-mock-server-<version>.tgz
npm install <path>/spm-universal-access-mocks-<version>.tgz
```

Where *<path>* is the download path and *<version>* is the package version.

6. Run the following command to install the package dependencies.

```
npm install
```

What to do next

Note: After an upgrade, the `react-scripts@^3.0.0` package can display an error when building the application with `npm run build` or starting `webpack-dev-server` with `npm run start`. This error is due to optional package installation checks. To avoid this issue with `react-scripts`, use one of the following options:

- Set the `SKIP_PREFLIGHT_CHECK=true` environment variable in the `.env` file.
- Run `npm update --no-save babel-eslint babel-jest babel-loader eslint jest webpack webpack-dev-server` to update the packages respecting the semver, and then run `npm dedupe`.

For more information, see the `create-react-app` issue at <https://github.com/facebook/create-react-app/issues/4167>.

Related tasks

Installing the IBM Cúram Universal Access development environment

You can install a lightweight or a full development environment. The IBM Social Program Management Design System is installed as part of the IBM Universal Access Responsive Web Application installation and doesn't need a separate installation.

Starter pack and packages

Using the IBM Universal Access Responsive Web Application starter pack and packages, and Social Program Management Design System packages, as your starting point, you can customize Universal Access for your organization.

Each package includes a `package-lock.json.sample` file, which lists the packages and versions that the release was built with. This file is for reference only and is not to be used directly for building.

universal-access-starter-pack

This package contains a development environment and a fully functional and deployable reference application. The starter application uses the other provided modules to provide an external web application for Universal Access.

The starter pack demonstrates how a modern and responsive Universal Access client can be built by using React, Redux, and the IBM Social Program Management Design System. It includes a sample feature that demonstrates coding conventions and the correct usage of the IBM Social Program Management Web Development Accelerator tool to help you to get started with developing your own custom features, see [“The sampleApplication feature” on page 74](#). You can rename, modify, and extend the starter application to suit the needs of your organization.

universal-access

This package contains a module that connects the Universal Access Responsive Web Application to the IBM Cúram Social Program Management server. `universal-access` makes HTTP requests to the server to allow the web application to interact with the IBM Cúram Universal Access installation. Redux is the storage mechanism for requests and responses. For more information, see [“Redux in Universal Access”](#) on page 77 and [“Universal Access Redux modules”](#) on page 79. This module does not render content, it depends on `universal-access-ui` to render the content.

universal-access-ui

This package contains a set of Universal Access Responsive Web Application features that presents views to the user, it depends on `universal-access` to provide the data that it needs for those views.

universal-access-ui-locales

This package contains translated UI artifacts for the `universal-access-ui` package.

universal-access-mocks

This package contains a module that provides mock data specific to Universal Access business scenarios for testing purposes. It is used by the mock server to provide mock APIs in the development environment so you don't need to host an IBM Cúram Social Program Management server during development.

mock-server

This package contains a lightweight server that can serve HTTP requests and return mock data as a response. You can use `mock-server` during client development as a substitute for a real server to test features.

core

This package provides JavaScript utilities to help you develop your application. For example, use the *RETSERVICE* utility to connect to a IBM Cúram Social Program Management server-side REST API. Use *IntlUtils* to format numbers and dates for globalization.

For more information about the core package utilities, see the JSDoc API documentation in `spm/core/doc`.

core-ui

This package provides common React UI components to help you develop your application. For example, use the *AppSpinner* component to display a spinning animation while a page loads, or use the *Toaster* component to display notifications to the user.

For more information about the `core-ui` components, see the JSDoc API documentation in `spm/core-ui/doc`.

core-ui-locales

This package provides translated artifacts for the `core-ui` components.

intelligent-evidence-gathering

This package enables IEG scripts that are configured in the IBM Cúram Social Program Management application to run in your application. An API is provided to call the IEG scripts.

For more information, see the API documentation in `spm/intelligent-evidence-gathering/doc`.

intelligent-evidence-gathering-locales

This package contains translated artifacts for the `intelligent-evidence-gathering` package.

spm-web-dev-accelerator

This package contains the Social Program Management Web Development Accelerator rapid feature development tool, which generates Redux modules to handle the communication between your application and IBM Cúram Social Program Management REST APIs.

spm-web-dev-accelerator-scripts

This package contains a Swagger parser to retrieve information from IBM Cúram Social Program Management REST APIs, and scripts to generate the features and modules code from configuration information in the spm-web-dev-accelerator package.

spm-test-framework

This package contains a number of reusable files to help you to set up a test environment for testing with Test Cafe, Jest, and Enzyme. You can use the provided helper files to help you to develop and write end-to-end tests, unit tests, or snapshot tests for your project.

spm-eslint-config

This package contains an ESLint configuration with predefined coding style rules and an EditorConfig configuration file.

Related information

[Design system packages](#)

Sample application project structure

The project structure is based on the Facebook create-react-app.

For more information about create-react-app, see [create-react-app](#).

```
.
├── mock
├── node_modules
├── package.json
├── public
├── src
│   ├── App.js
│   ├── Config
│   ├── css
│   ├── Features
│   │   ├── sampleApplication
│   │   │   ├── confirmation
│   │   │   │   └── SampleApplicationConfirmation.js
│   │   │   ├── form
│   │   │   │   └── SampleApplicationForm.js
│   │   │   └── overview
│   │   │       └── SampleApplicationOverview.js
│   ├── index.js
│   ├── intl
│   ├── modules
│   ├── paths.js
│   ├── redux
│   ├── routes.js
│   ├── routesMessages.js
│   ├── /sass
│   └── serviceWorker.js
├── tests
├── .env
└── .env.development
```

The main files in the project are as follows:

package.json

The `package.json` file is customized to support the Universal Access starter application. For more information on standard `package.json`, see [package.json](#).

/mock

`/mock` contains the wiring that is needed to interact with the `mock-server` module. The mock server replicates the SPM APIs, providing the mocked end points that are used by the sample application.

For more information about the mock server, see [“The mock server API service” on page 91](#).

/public

`/public` is part of the `create-react-app` boilerplate. For more information, see [create-react-app](#).

/src

`/src` is your working folder. The starter pack provides the basic infrastructure that interacts with the universal-access modules that are the platform for your development effort. `/src` contains the following components:

- `/src/index.js` Initiates the application and adds the following capabilities:
 - Connection to a Redux store by using the `react-redux` module `Provider` component.
 - Globalization is added by using `react-intl` and the `LanguageProvider` component.
 - The universal-access module has a limited set of configurations that can be modified by using the `AppConfig` component.
- `src/App.js` is launched from the `index.js` file and wraps the main application in the `react-router`.
- `src/css` contains the compiled CSS styles.
- `src/config` contains the `intl` configuration files.
- `src/features` contains a sample feature to demonstrate how to implement a simple version of the **Apply for benefits** feature, see [“The sampleApplication feature” on page 74](#).
- `src/redux` contains the configuration for Redux reducers and the store.
- `src/intl` handles React-Intl Initialization.
- `src/routes.js` provides a point of customization for adding, replacing, or removing routes in your application.
- `src/paths.js` provides access the URLs that are mapped to each page by the route configuration.
- `src/routesMessages.js` contains the text Routes to be displayed on the window's title.
- `src/appconfig.sample.json` allows parts of `universal-access` to be customized, for example, specifying the default and other supported languages.
- `src/sass/styles.scss` contains the SCSS style definitions.
- `src/sass/custom_variables.css` provides a configuration point for CSS variables.

.env and .env.development

The `.env` file contains the environment variables for production. The `.env.development` file supersedes the environment variables in `.env` and sets specific environment variables for development. For more information about environment variables, see the [“React environment variable reference” on page 173](#).

Developing compliantly

Follow these guidelines to protect your project from making customization changes that are incompatible with the base product, or have the potential to incur upgrade impacts.

Never use undocumented APIs

JavaScript does not have access modifiers such as `public`, `private`, or `protected`. It is possible to call functions in SPM modules that are not intended for public use. Calling these functions is not supported as those APIs can change in a future release and break your code.

The only JavaScript APIs that are intended for public use are documented in the `docs` folder of the SPM `node_modules`. For example, `node_modules/@spm/core/docs/index.html`.

Observe the Redux reducer namespace

If you use Redux, your Reducer names must not infringe on the namespace for Universal Access reducers. All Universal Access reducers are prefixed with `UA`, for example, `UABenefitSelection`. When Universal Access and custom reducers are combined, clashing names override the Universal Access reducers. Customizing `universal-access` reducers is not supported.

Don't modify the starter application files

While you can modify the starter application files in place, it is better to copy the files and change the copy. Your upgrades will then be easier as you can compare files between the current and previous version of the product without the added complexity of your customization changes. Where upgrade changes are needed, manually apply the changes to your custom version.

Don't modify or source control any code that is generated

The IBM Social Program Management Web Development Accelerator tool generates code from the metadata in the `modules_config.json` file, which is the only file that you need to source control. The code is generated each time that you click **Generate** in the tool, or run the `npm install`, `npm run build`, or `npm run wda-generate` commands.

Enforce code style with ESLint

To help you to run static code analysis on your code, the `spm-eslint-config` package contains an ESLint configuration with predefined coding style rules and an `EditorConfig` configuration file.

The ESLint configuration is in the `./node_modules/@spm/eslint-config/index.js` file.

The `.editorconfig` `EditorConfig` setup file is in the root directory of the sample application.

Running ESLint

To check the code for ESLint violations, run the following command in the starter application root directory. Errors are listed in the console.

```
npm run lint
```

Fixing ESLint violations

Run the following command for ESLint to fix syntactic problems automatically:

```
npm run lint -- --fix
```

You must manually fix any violations that can't be resolved automatically.

The first time that you run a static code analyzer on your code, particularly if coding style was not previously enforced, you might see numerous errors. Don't get discouraged, while it might take to fix all of the violations, ensuring that your team uses a consistent coding style has significant long-term benefits.

ESLint plug-ins for code editors

Most code editors support plug-ins for linting. [ESLint plugin](#) is a useful plug-in for Microsoft Visual Studio Code. [ESLint plugins](#) are also available for Atom.

If you use a plug-in, errors are highlighted in the code editor and can be seen and fixed during development. When all the developers use a plug-in, it is easier to maintain a consistent code style.

Automation

If you have a CD/CI pipeline, you can include linting as part of the testing phase. It is a good idea to correct code with linting issues before you merge it into the codebase.

EditorConfig

The included [editor config](#) configuration file ensures consistent coding style when it comes to indentation, spacing, and quotation types.

EditorConfig works without a plug-in for Microsoft Visual Studio Code. If you use other editors, like Atom or Sublime Text, you need a plug-in. For more information about available plug-ins, see [EditorConfig website](#).

Universal Access UI coding conventions

The `universal-access-ui` package is responsible for the presentation of the UI in the application. Coding conventions ensure that the UI code is separated by responsibilities, which gives benefits such as easier maintenance. Features, Components, and Messages are coded to render each page of the application.

Each page represents a business process function along a specific URL route. It is presented by using individual IBM Social Program Management Design System components, embedded with localizable messages, and connected to the Redux store, in the `universal-access` package, to access and manage data in the application state (where applicable).

Features

A *feature* is an intangible concept of individual business functionality that is translated into a view navigable by a route.

A feature maps a particular business process or functionality, such as showing a user their payments, and makes it visible to the user in a collection of files that work together and are navigable by a URL route. For example `/payments`.

Multiple features can be used to implement a larger or more encompassing business process, such as Life Events, depending on how many separate views or business process functions are required.

Features are mainly defined through a path, a `Routes.js` entry, and a directory that references the feature's top-level React component.

Paths.js

A simple JavaScript file that exports a JSON object that contains the properties with each navigable path a user might traverse to in the application.

For a feature, the first step is to declare the appropriate navigable route here, for example:

```
const PATHS = {  
  ...  
  USER_ENROLMENT: '/user_enrolment'
```

```
} ...
```

Routes.js entry

At a high-level, the `Routes.js` file in `universal-access-ui` (not the customizable `Routes.js` file in the sample application) renders the feature's top-level React component (which is exported from the feature's `index.js` file) depending on the current URL route.

`react-loadable` is used for component-centric code splitting. The feature's top-level React component is dynamically imported.

```
// UserEnrolmentContainer exported by /features/UserEnrolment/index.js
const UserEnrolment = Loadable({
  loader: () =>
    import(/* webpackChunkName: "SomeFeature" */ "../features/UserEnrolment"),
  loading: LoadingPage
});
```

Declare the route within the `render()` function, either as a `TitledRoute` or an `AuthenticatedRoute`. Those familiar with [React-Router](#) might recognize some of the props.

```
...
render() {
  return (
    ...
    <TitledRoute
      component={UserEnrolment}
      exact
      path={PATHS.USER_ENROLMENT}
      title={localisableRoutesMessageFile.userEnrolmentTitle}
    />
    ...
  )
}
```

This effectively wires up the feature's route to the feature's React components in the internal `Routes.js` file.

Adding features, or customizing existing features, for example overriding the FAQs, require some implementation in your `sample-app/src/routes.js` file. You must add the new feature or redirect a route of an existing feature to your custom feature. For information about implementing similar routing in your custom application, see [Developing with routes](#).

Directory reference

The location of the feature in the file system. Each feature in `universal-access-ui` is a directory within `/universal-access-ui/src/features`. The directory is named after the business process function. It contains the files responsible for rendering the actual view to the user. A single React component, typically the Container, is exported by the feature's `index.js` to represent the feature at higher levels, for example `Routes.js`.

The `universal-access-ui` package does contain other high-level directories that are responsible for other functionality, but these are separate or complementary to the base feature concept.

Components

A *component* is a React component whose responsibility is to manage the data concerns for the piece of business functionality and render the user's view of the business functionality by using the data passed as props, text defined in Messages, and components from the IBM Social Program Management Design System.

Components are typically the highest-level React component that are exported from a feature (and act as the starting renderable component) as generally every business process function requires some type of data to retrieve, manipulate, and display. There are a few exceptions to this rule when the feature is only an informational or static text view.

Components render the view of the business process function to the user.

By default, layouts, HTML elements, and more complex UI widgets (like buttons, cards, badges, panels, sections, headers, etc.) are taken from the IBM Social Program Management Design System. This provides a standardized theme to the look-and-feel of all our features and benefit from common concerns, such as accessibility and differing screen size layouts. We reference text defined in a separate Messages file to render any text content.

Messages

Messages files define a JSON object that contains individual properties for each portion of text that is used by a component and exported as a parameter to an API of the `react-intl` library.

Typically, every component renders text as part of its view. Each portion of text must be translatable depending on the user's language. Universal Access uses the `react-intl` library to help manage the text content for translation.

For each component, there is a similarly created messages file, which contains the text that is wrapped in the `react-intl` `defineMessages()` API. For example, `UserEnrolmentComponentMessages.js`.

```
import { defineMessages } from 'react-intl';

export default defineMessages({
  userEnrolmentTitle: {
    id: 'UserEnrolment_Title',
    defaultMessage: 'User Enrolment',
  },
  userEnrolmentDescription: {
    id: 'UserEnrolment_Description',
    defaultMessage: "You can enrol in our user's program.",
  },
  userEnrolmentButtonLabel: {
    id: 'UserEnrolment_Button',
    defaultMessage: 'Continue',
  },
  ...
});
```

The sampleApplication feature

The sample feature illustrates the principles, tools, and technologies for developing features in the application. It implements a simple **Apply for Benefits** workflow that complies with the coding conventions.

The IBM Social Program Management Web Development Accelerator tool significantly speeds up the development of the Redux modules that connect the application to the REST APIs. The `BaseFormContainer` component is used to implement EG forms. The test framework speeds up the development of tests with less code. Where possible, replacing React containers with standard and custom React hooks can reduce complexity and further speeds up development.

Apply for Benefits workflow

Landing page

The `/sample-application` page shows a list of application types, which were obtained by using an API call. The code for that API call was generated with the Social Program Management Web Development Accelerator tool. Select an application type to go the **Overview** page. When you select the application type, the type is stored in a custom Redux store object that was also configured with the tool.

Overview page

The `/sample-application/overview` page describes the benefit and provides the option to start the application. Applying for the benefit starts an IEG script with a script ID that is obtained from an API call. This API call is configured by using the Social Program Management Web Development Accelerator.

The Apply for Benefits form

The form is rendered from the IEG script by using the `BaseFormContainer` component. Enter the required values to complete the form. When the form is complete, the confirmation page opens.

Confirmation page

The `/sample-application/confirmation` page summarizes the information that you entered.

Looking at the `SampleModule` module

To review the Redux module for the sample feature in Social Program Management Web Development Accelerator, start the tool by running `npm run wda`. From the home page, select **View Modules** and then **Edit** on `SampleModule` module. On the **APIs** tab, you can see the two APIs for the Apply for Benefits workflow.

- The `v1/ua/online_categories` API returns a list of online categories where each online category includes details of applications that a user can apply for. This API is used on the landing page.
- The `v1/ua/application_form` API is used to start a new application form for the logged in user. The `selectedApplicationType` value is defined when you click an application type on the landing page and is then used on subsequent pages.

On the **Store** tab, you can see the selector and action for the `selectedApplicationType`.

Overview of the sample application code

`SampleApplicationComponent.js`

Displays a list of benefit application types. This component shows you how to do the following tasks:

- To generate a temporary user if the current user is not logged-in by using the `useGeneratedUserIfNotLoggedIn` React hook.
- To retrieve information from the API and Redux store state by using the `SampleModuleHooks.useFetchOnlineCategories` hook.
- To verify whether the Rest API is still fetching information by using the `SampleModuleHooks.useFetchOnlineCategories` hook.
- To wrap the complete component with a React Higher Order Component (HOC). In this case, the `withErrorBoundary` error boundary HOC.

`SampleApplicationConfirmation.js`

A confirmation page with the identifier of the application submitted.

`SampleApplicationFormComponent.js`

This component handles the application IEG scripts, general IEG rendering and handling is delegated to the `BaseFormContainer` component.

`SampleApplicationOverviewComponent.js`

This component gives an end-to-end view of the application process to the user, along with a summary of the application type and program types that they are applying for. This component shows how to dispatch an action and create an application form by calling the `useCallback` hook that is associated with a button `onClick` handler.

Related concepts

Generating custom hooks

To abstract the complexity of working with Redux, the IBM Social Program Management Web Development Accelerator automatically generates React Hooks to be imported directly into React

components. You don't need to know about Redux to use these hooks for state management in the application.

Manage state with React Hooks

React Hooks enable you to use state, execute effects, and other React features without writing a class. You can use hooks to subscribe to the Redux store and dispatch actions, without having to wrap your components in `connect()`.

If you use containers, you need to:

- Use a React Class Component.
- Implement `mapDispatchToProps` to have access to the dispatch object to call actions.
- Implement `mapStateToProps` to have access to the state object to call selectors.
- Use the `connect` higher-order component when you export the component to wire it with Redux.

For example:

```
class SampleContainer extends Component {
  componentDidMount() {
    //Initializations
    //Calling Action
    this.props.sampleAction();
  }
  ...
  render() {
    //Calling selector
    const selectorValue = this.props.sampleSelector();

    return <Component body/>;
  }
  ...
}

// We need to implement this function to have access to the `dispatch` object
const mapDispatchToProps = dispatch => ({
  // Call actions using the dispatch object
  sampleAction: () => SampleModuleActions.actionName(dispatch);
})

// We need to implement this function to have access to the `state` object
const mapStateToProps = state => ({
  // Call selectors using the state object
  sampleSelector: () => SampleModuleSelectors.selectorName(state);
})

// To do the wiring with redux, we need to use the `connect` HOC passing the two functions:
`mapStateToProps` and `mapDispatchToProps`
export connect(mapStateToProps, mapDispatchToProps)(SampleContainer);
```

To do the same with hooks:

- You don't use class components.
- You don't need to use `connect`, `mapStateToProps` or `mapDispatchToProps`.
- Use `useDispatch` to get the dispatch objects and call the actions.
- Use `useSelector` to get the state object and call the selectors.
- Use `useEffect` to simulate the life cycle events, for example `componentDidMount`

For example:

```
const SampleComponent = props => {
  //Get the dispatch object to call actions
  const dispatch = useDispatch();
```

```
// Initializations - The same as componentDidMount
useEffect(() => {
  //Calling action
  SampleActions.actionName(dispatch);
}, [])

//To call the selectors you do:
const selectorValue = useSelector(state => SampleSelectors.selectorName(state));

return (<>Component body<>);
}
```

In addition to the reduced code, you can create custom hooks to further reduce the amount of code.

Custom hooks

The following custom hooks are provided:

- **useGeneratedUserIfNotLoggedIn:** On mounting a component, checks whether the user is logged in. If not, calls REST APIs to create a temporary user and automatically authenticate the user. This is useful for anonymous IEG forms.
- **usePublicCitizenIfNotLoggedIn:** On mounting a component, checks whether the user is logged in. If not, automatically authenticate the user as a publicCitizen. For example, this is useful for landing pages that need to call REST APIs to populate lists.

It is not possible to implement these two custom functions without hooks, as a utility JavaScript file for example, because they need to modify the React component state.

Redux in Universal Access

Redux is used as a client-side store to store data that is retrieved by IBM Cúram Social Program Management APIs and data that is used to present a consistent user experience.

What is Redux?

Redux is a client-side store that provides a mechanism for holding data in the browser.

- The store is typically used to manage state in the client application. State can include the following types of data:
 - System data that is returned from an API request.
 - User input data that is collected before it is posted to APIs.
 - Application data that is not sent from or to the server, but is created and maintained to control how the application works. For example, transient user selections like hiding or showing a side pane.
- Redux uses a unidirectional architecture, which simplifies the process of managing state.
- Redux can be used as a caching mechanism to avoid unnecessary network round-trips, although consider this usage carefully to ensure the data that is presented is always current.
- Redux proves to be beneficial as your application grows and becomes more complex. By centralizing state management and offering tools that give a holistic view of the application state, development can scale more easily.

Note: This topic assumes that you are familiar with Redux and using Redux with React components. If you are not familiar with these technologies and how they work together, you should complete tutorials from the official sources for these technologies.

How is Redux used in Universal Access?

IBM Cúram Universal Access uses Redux to store the data that is retrieved by the IBM Cúram Social Program Management APIs.

Each GET API used by Universal Access has an associated 'store slice' where the response of the API is stored. React components can monitor the store for updates relevant to them and automatically update as data changes. The store is also used for collecting user input, such as user information that is requested while users sign up. This data can then be retrieved from the store and posted to the IBM Cúram Social Program Management server.

Other parts of the store are not tied to IBM Cúram Social Program Management APIs, and track data that is used to present a consistent user experience.

Creating a Redux store

By default, the Universal Access starter pack is configured to use a Redux store. This configuration is needed to allow it to use the `universal-access` and `universal-access-ui` packages. The store configuration is initiated from the `src/redux/ReduxInit.js` file in the starter pack.

```
...  
import configureStore from './store';  
...  
  
// =====  
// 1. Create the store and initialize the universal-access module.  
// =====  
  
// Create a Redux store  
// This is optional, if you don't want to create your own Redux store you can remove this,  
const appStore = configureStore();  
  
// Configure the UA package  
// 1. If you are using your own store, you must share it with UA  
UAReduxStore.configureStore(appStore);  
  
...
```

For more information on Redux, see <https://redux.js.org/>.

Configuring the store

Configure the store in the `src/redux/store.js` file, which exports the `configureStore` function that can be called to create a new Redux store. The configure store function can be modified to:

- Add Redux 'middleware'.
- Provide a custom set of reducers.

Note: To work with the `universal-access` packages, the store must use the reducers that are exported from the `universal-access` package.

Clearing Redux store data

The Redux store is a JavaScript object that is stored in the global object for the browser window. The content of the store is visible through inspection, either programmatically or by browser plug-in tools, such as the developer tools. It is critical that the store is cleared for the current user when they log out to ensure that no sensitive user data is left on the device for malicious actors. The log-out feature that is provided by the starter app triggers a Redux action that clears the store.

Adding reducers

If you decide to use Redux with your custom React components, you must create custom reducers and add them to the store. All Universal Access reducers are prefixed with **UA**, for example `UAPaymentsReducer`. The `intelligent-evidence-gathering` package also exposes `IEGReduxReducers` reducers, prefixed with **IEG**. When adding custom reducers, you can combine your custom reducers with existing reducers. Do not use the UA or IEG prefixes in custom reducers to avoid overriding existing reducers. Overriding reducers is not supported, see “Developing compliantly” on page 71.

The `src/redux/rootReducer.js` file defines the set of reducers for the store, and combines them into a single *root reducer* that can be passed to the `configureStore` function in the `src/redux/store.js` file.

For convenience, the file defines an `AppReducers` object where you can add custom reducers. The custom reducers that are defined in the `AppReducers` object are combined with the `UAReducers` imported from the `universal-access` package, and the superset of reducers is returned.

The following code excerpt shows the `rootReducer` function that returns the combination of Universal Access reducers and custom reducers.

```
const AppReducers = {
  // Add custom reducers here...
  // customReducer: (state, action) => state,
};

/**
 * Combines the App reducers with those provided by the universal-access package
 */
const appReducer = combineReducers({
  ...AppReducers,
  ...UAReducers,
});

/**
 * Returns the rootReducer for the Redux store.
 * @param {*} state
 * @param {*} action
 */
const rootReducer = (state, action = { type: 'unknown' }) => {
  ...
  return appReducer(state, action);
};
```

Universal Access Redux modules

Modules in the Universal Access Responsive Web Application communicate between the application and the IBM Cúram Social Program Management REST APIs and manage data for the API in the Redux store.

This design allows the React components to focus on presentation and reduces the complexity of the code in the presentation layer. Modules manage the communication between the client application and the IBM Cúram Social Program Management REST APIs, including authentication, locale management, asynchronous communication, error handling, Redux store management and more.

Modules typically follow the [re-ducks pattern](#) for scaling with Redux

Modules and APIs

Modules consist of collection of artifacts that work together to communicate with IBM Cúram Social Program Management REST APIs and manage the storage and retrieval of the response in the application state. For example, the Payments module is responsible for communicating with the `/v1/ua/payments` API. For more information about IBM Cúram Social Program Management APIs, see *Connecting to a Cúram REST API*.

Models

The `models.js` file is your data representation of the response from the API. It must map the JSON response properties to an object that can be referenced within your web application.

```
class UserProfile {
  constructor(json = null) {
    if (json) {
      this.personFirstName = json.personFirstName;
      this.personMiddleName = json.personMiddleName;
      this.personSurname = json.personSurname;
      this.personDOB = json.personDOB;
      this.userName = json.userName;
      this.userType = json.userType;
      ...
    }
  }
}
```

```

    }
  }
}

export default UserProfile;

```

Utils

The `utils.js` file is responsible for the actual communication to the required API. On successful contact with the API, it constructs the model with the response. For simple GET calls, you can use `RESTService.get` to handle the API call. For more information, see the `RESTService` utility.

```

import { RESTService } from "@spm/core";
import UserProfile from "../models";

const fetchUserProfileUtil = callback => {
  const url = `${process.env.REACT_APP_REST_URL}/user_profile`;
  RESTService.get(url, (success, response) => {
    const modelledResponse = new UserProfile(response);
    callback(success, modelledResponse);
  });
};

export { fetchUserProfileUtil };

```

ActionTypes and Actions

Module actions are used to modify the Redux store, like inserting, modifying, or deleting data from the store. For example, the `PaymentsActions` action modifies the payments slice of the store.

Some actions include calls to APIs. For example, `PaymentsActions.getData` action calls the `v1/ua/payments` API and dispatches the result to the payments slice of the store, or sets an error if the API call fails.

The `actionTypes.js` file represents the type of action that is being performed. At its core, they are simple string types. For more information, see the [Redux Glossary](#).

```

const FETCH_USER_PROFILE = "UA-CUSTOM/USER_PROFILE/FETCH_USER_PROFILE";

export { FETCH_USER_PROFILE };

```

The `actions.js` file contains the Redux actions, which are objects that represent an intention to change the application state. They are exported to be accessible to call from a Container component.

The following example is a representation of the action that calls the API and attaches the response to the dispatch, but you might further improve by adding fallback behavior.

```

import { FETCH_USER_PROFILE } from "../actionTypes";
import { fetchUserProfileUtil } from "../utils";

export default class actions {
  static fetchUserProfile = dispatch => {
    fetchUserProfileUtil((success, payload) => {
      if (success) {
        dispatch({
          type: FETCH_USER_PROFILE,
          payload: payload
        });
      }
    });
  };
}

```

Reducer

The `reducers.js` file contains the [Redux Reducers](#). Redux Reducers are just functions that take the existing state and current actions and calculate a new state, thus updating the application state.

The following example represents a data reducer that updates the state based on the API result. You can implement more complex reducers based on the action to represent API errors or failures or if the API is awaiting a response, like an `isFetchingUserProfile` reducer.

Reducers aren't called from Container components.

```
import { combineReducers } from "redux";
import { FETCH_USER_PROFILE } from "../actionTypes";

const fetchUserProfileReducer = (state = {}, action) => {
  if (action.type === FETCH_USER_PROFILE) {
    return { ...state, payload: action.payload };
  } else {
    return state;
  }
};

const reducers = combineReducers({
  fetchUserProfile: fetchUserProfileReducer
  // room for more reducers!
});

export default { reducers };
```

Selectors

Module selectors are used to query the Redux store. They provide the response to predefined store queries. For example, the `PaymentsSelector.selectData` selector returns the `/payments/data` slice from the store, and the `PaymentsSelector.selectError` selector returns the value of the `/payments/error` slice of the store.

The `selectors.js` file is responsible for retrieving the data from the application state for use in the Container component (and likely passed as props to the Presentational component). It selects information from the state by using the state's 'slice' identifier.

```
export default class selectors {
  static moduleIdentifier = "UACustomUserProfile";

  static fetchUserProfile = state =>
    state[selectors.moduleIdentifier].fetchUserProfile.payload;
}
```

Index

You must export the parts of a module that must be accessible. Instead of creating an `index.js` per module, create one in the module directory that exports the Actions, Model, and Selectors of each custom module. These classes or functions are the only ones that need to be accessed from the container components.

```
// Modules
export { default as UserProfileActions } from "../UserProfile/actions";
export { default as UserProfileSelectors } from "../UserProfile/selectors";
export { default as UserProfileModels } from "../UserProfile/models";
```

Blackbox

Modules are blackbox so are not open to customization or extension. The modules expose actions and selectors to interact with the module. The actions and selectors are APIs that are documented in the `<your-project-root>/node_modules/@spm/universal-access/docs/index.html` file.

Reusing Universal Access modules in your custom components

You can use the actions and selectors from the universal-access package to connect your custom components to existing IBM Cúram Social Program Management APIs and the Redux store. You can use the `react-redux` module to connect your components. Examples of this technique can be found in the `universal-access-ui` features.

For example, the following code is from the `PaymentsContainer` file in the Payments feature. The code shows how the actions and selectors from the Payments module are connected to the properties of the Payments component.

This pattern is documented extensively in the official Redux documentation.

```
import { connect } from 'react-redux';
import React, { Component } from 'react';

...

/**
 * Retrieves data from the Redux store.
 *
 * @param state the redux store state
 * @memberof PaymentsContainer
 */
const mapStateToProps = state => ({
  payments: PaymentsSelectors.selectData(state),
  isFetchingPayments: PaymentsSelectors.isProcessing(state),
  paymentsError: PaymentsSelectors.selectError(state),
});
/**
 * Retrieve data from related rest APIs and updates the Redux store.
 *
 * @export
 * @param {*} dispatch the dispatch function
 * @returns {Object} the mappings.
 * @memberof PaymentsContainer
 */
export const mapDispatchToProps = dispatch => ({
  loadPayments: () => PaymentsActions.getData(dispatch),
  resetError: () => PaymentsActions.resetError(dispatch),
});
/**
 * PaymentsContainer initiates the rendering the payments list.
 * This component holds the user's payment details list.
 * @export
 * @namespace
 * @memberof PaymentsContainer
 */
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(PaymentsContainer);
```

Related information

[Connecting to a Cúram REST API](#)

Social Program Management Web Development Accelerator

The IBM Social Program Management Web Development Accelerator is a tool that automatically generates code for application state, such as custom hooks and Redux modules. Select and configure Social Program Management REST APIs and automatically generate all of the hook or module code.



[Click here for a video presentation that gives an overview of the Social Program Management Web Development Accelerator.](#)



[Click here for a video series that gives a comprehensive view of the Social Program Management Web Development Accelerator and other recent technologies.](#)

How it works

1. Add a module and provide a module name and description.
2. Click **Add API** and select and configure the Social Program Management REST APIs that are required for the module.
3. Select the **Store the API response in Redux?** checkbox if you want to store the API response in the Redux store. If you select this option, hooks cache the responses of APIs.
4. Save the module. Your configuration is saved as metadata in a JSON file, which is the only code that you need to source control.
5. Generate the code. The module and hooks code is generated from the metadata and placed into a specified directory in the project.

6. Review the generated code in the Code preview tab. The tab contains the following sub-tabs, Hooks, Actions, ActionTypes, Utils, Models, Reducers, and Selectors.
7. Import the module or hook into your React components.

Note: You don't need to source control the generated code. The code is generated each time that you click **Generate** in the tool, or when you run `npm install`, `npm run build` or `npm run wda-generate`.

Generating custom hooks

To abstract the complexity of working with Redux, the IBM Social Program Management Web Development Accelerator automatically generates React Hooks to be imported directly into React components. You don't need to know about Redux to use these hooks for state management in the application.

The generated hooks are responsible for the integration with the Redux modules, and handle all the caching that is needed to avoid unnecessary HTTP calls to REST APIs.

React Hooks are generated for each API and store objects that are added to the modules by using the IBM Social Program Management Web Development Accelerator. The hooks provide all data and functions that are needed by the React components. There is no need to interact directly with the Redux files, that is, Actions or Selectors.

For each API, a hook is generated with the following structure:

GET APIs:

```
const { data, isFetching, error, reFetchData, resetData } = useActionName([params], deps = []);
```

For DELETE, POST, PUT APIs:

```
const { actionName, data, isFetching, error } = useActionName();
```

Hooks are also generated for custom store objects that are defined in the **Store** tab of the IBM Social Program Management Web Development Accelerator:

```
const { object, setObject } = useSetObject();
```

For more information about how to interact with the React Hooks, see the Sample Application Feature at: `/src/features/sampleApplication`

Related concepts

[The sampleApplication feature](#)

The sample feature illustrates the principles, tools, and technologies for developing features in the application. It implements a simple **Apply for Benefits** workflow that complies with the coding conventions.

Generating Universal Access Redux modules

In the IBM Social Program Management Web Development Accelerator, you can create a module, select and configure your REST APIs, and generate all of the code that is needed to handle the API requests and manage your application state with Redux.

Before you begin

Check that the Social Program Management Web Development Accelerator environment variables are set correctly, see [“React environment variable reference” on page 173](#).

Procedure

1. In the root directory of the `universal-access-starter-app`, run the command:

```
npm run wda
```

The Social Program Management Web Development Accelerator opens locally at `http://localhost:3000/`.

2. On the home page, click **View modules**.
3. Click **Add module** or click an existing module to edit the module.
4. To add APIs, select the **APIs** tab and click **Add API**.
5. From the list of available APIs that is defined by the Swagger specification in the **WDA_SPM_SWAGGER** environment variable, select the APIs that you need.
The APIs are added to the model metadata JSON file that is specified in the **WDA_MODULES_CONFIG** environmental variable.
6. You can customize the default Action functions, Selectors, and Reducers for an API by changing their names, or by specifying whether the API response is stored in Redux.
 - a) By default, function names are defined by a convention based on the API URI and verb. Click a function name to rename the function.
 - b) By default, each REST API response is cached in the Redux store. If you don't want to store the API response, clear the **Store the API response in Redux?** check box. The corresponding functions are removed from the model.

The APIs are defined in the model.

7. To create a custom store object to cache JavaScript objects, select the **Store** tab, click **Add Store**, enter a name for the store object, and click **Confirm**.
8. You can preview the code to be generated from the modules metadata by selecting the **Code Preview** tab.
9. You can generate the code as follows:
 - a) From the **Modules** page, click **Generate**
 - b) By using npm, run the command:

```
npm run wda-generate
```

The code is also generated each time that the project is installed or built by running `npm run start` or `npm run build`.

The modules and the generated code are written directly to the directory that is defined in the **WDA_MODULES_OUTPUT** environment variable.

Error handling with a React higher-order component (HOC)

You can use the `withErrorBoundary` function as a higher-order component (HOC) to handle API errors on features. You can then focus on implementing components and delegate the error handling to the function. Additionally, this approach reduces the amount of code that is needed to implement the component and its tests.

The `withErrorBoundary` function is provided in the `@spm/universal-access-ui` package and provides the following functions:

- Retrieves the list of errors from the Redux Store. You can use the default `ReduxUtils.generateGlobalErrorSelector` invoking error selector, or provide a customer selector.
- For any errors, the `withErrorBoundary` function throws a JavaScript exception that is caught by the nearest `ErrorBoundary`.
- Wraps a component in an `ErrorBoundary`.
- Clears the errors from the Redux Store when the component is unmounted.

Table 15. The `withErrorBoundary` parameters

Parameter	Mandatory	Details
wrappedComponent	Yes	The component or container to wrap.
errorSelector	No	The selector to get the errors. If you don't provide an error selector, <code>ReduxUtils.generateGlobalErrorSelector</code> is used.
resetErrorAction	No	The action to reset the errors.

Examples

Exporting a component with the `withErrorBoundary` function.

Default values

```
import withErrorBoundary from '@spm/universal-access-ui';

class Container extends Component {
  ...
}

export default withErrorBoundary(Container);
```

With parameters

```
import withErrorBoundary from '@spm/universal-access-ui';
import { CustomSelectors, CustomActions } from '@spm/universal-access';

class Container extends Component {
  ...
}

export default withErrorBoundary(Container, CustomSelectors.selectError ,
CustomActions.resetError)
);
```

Connectivity handling

By default, a connectivity handler prevents data loss in IEG forms and provides offline detection for the application. You see a warning message when you are disconnected, giving you a chance to check your internet connectivity. You see a success message when you recover connectivity.

If internet connectivity is lost or the service becomes unavailable when you are in the application, a warning message is displayed.

You're disconnected. Check your internet connection or wait while we try to reconnect.

To prevent you from losing information that you enter in IEG forms, you are now prevented from leaving IEG pages with unsubmitted changes when you are disconnected. An error message is displayed and you remain on the page where you entered the information. You must check your internet and get online or wait for the service to become available before you can continue. If internet connectivity or the service remains unavailable, the error message persists.

You're still disconnected. Check your internet connection or try again later.

When you recover connectivity or the service becomes available, a success message is displayed and you can save your changes.

You are now connected.

You can choose to prevent data loss in pages outside IEG forms by implementing the connectivity handler for other pages in the application.

Implementing a connectivity handler

You can implement loss-of-connectivity handling to improve the resilience of the application. Users are notified when they lose or recover internet connectivity or access to the service. In addition, you can prevent user actions when they are offline to avoid errors or data loss. By default, user actions are prevented in IEG forms to prevent information loss if users go offline when in IEG forms.

Before you begin

The implementation consists of the following components.

The **ConnectivityHandler** component

Use the **ConnectivityHandler** React component in `packages/core-ui/src/Connectivity/ConnectivityHandler` to detect offline and online events and to notify the user. The **ConnectivityHandler** component wraps the application with the logic to call the appropriate messages.

The **ConnectivityHandler** component provides the following functions:

- Registers browser offline and online events and runs the corresponding callback functions. Internet connectivity is checked by making a periodic request to the server, by default every 5 seconds.
 - If the server request fails, it triggers the offline event callback. A dismissable warning message is displayed:
You are offline, check your internet connection and try again.
 - After connection is lost, a successful server request triggers the online event callback. A dismissable success message that expires after 7 seconds is displayed:
You are back online.
- Reads the `preventActionOnOffline` Redux global state variable that is managed by the **Connectivity** Redux module. When this state is `true` and a user tries to interact with a page when they are offline, an error message is displayed:
You are still offline, check your internet connection and try again.

The **Connectivity** Redux module

Use the **Connectivity** Redux module in `packages/core/src/modules/Connectivity` to manage the connectivity global state of the application, and to provide an option to prevent unwanted actions.

The **Connectivity** module has two selectors and two actions:

- The `setOnline` action sets the connectivity state of the application. Set to `true` for online and `false` for offline.
- The `getOnline` selector returns the connectivity state of the application.
- The `setPreventActionOnOffline` action sets the prevent action state of the application. Set `true` to prevent user actions while offline and `false` to allow user actions.
- The `getPreventActionOnOffline` selector returns the prevent action state of the application.

The URL that is pinged for connectivity and the frequency of the ping request are set by the connection handler environment variables. For more information, see [“React environment variable reference” on page 173](#).

Procedure

1. Wrap your application in the **ConnectivityHandler** component.

Place the `ConnectivityHandler` component between the `ErrorBoundary` and `SSOVerifier` components in the application tree as follows:

```
...
  <ReduxInit>
    <IntlInit>
      <Router basename={process.env.PUBLIC_URL}>
        <ScrollToTop>
          <ErrorBoundary
            footer={<ApplicationFooter />}
            header={<ApplicationHeaderComponent hasErrorBeenCaught />}
            isHeaderBoundary
          >
            <ConnectivityHandler>
              <SSOVerifier
                placeholder={
                  <ApplicationHeaderComponent
                    isALinkedUser={() => false}
                    isAppealsEnabled={false}
                    isEmpty
                  />
                }
              >
                { /* Rest of the application tree */ }
              </SSOVerifier>
            </ConnectivityHandler>
          </ErrorBoundary>
        </ScrollToTop>
      </Router>
    </IntlInit>
  </ReduxInit>
...
```

The application now notifies users of online and offline events.

2. Optional: Prevent user actions when they are offline with the `Connectivity` Redux module. You can display a danger message to tell users that they are offline and that they need to check their internet connection.
 - a) Use the `getOnline` selector to read the connectivity global state of the application.
 - b) Use the `setPreventActionOnOffline` action to notify `ConnectivityHandler` to display the prevent action message.

For example:

```
preventOffline = callback => {
  const { isOnline, preventActionOnOffline } = this.props;
  return isOnline === false ? preventActionOnOffline() : callback;
};
```

The example function uses `{ connect }` from `'react-redux'`. Its implementation helps you to inject the selectors and actions from the connectivity global state of the application as props. `isOnline` is the value that is returned by `getOnline`, and `preventActionOnOffline()` calls `getPreventActionOnOffline(true)`.

The `preventOffline` function receives the function to run or prevent when online or offline as a callback.

The online status is received from props and the `preventActionOnOffline` function displays the message.

When `isOnline` is false, call the function to display the message, otherwise call the callback function.

Replacing routes

You can replace existing paths from the Universal Access module's Routes component with your preferred component.

Wrap your routes in a `<Switch>` component

You can replace existing paths from the Routes component with your preferred component. To achieve this, you must first wrap your routes in a `<Switch>` component from react-router. This action ensures that the first match of the requested path that is found in your application is used to resolve the path. For more information on Switch, see <https://reacttraining.com/react-router/web/guides/philosophy>.

Add a route with the same path

When you have wrapped in *Switch*, you add a route with the same path as the page you are overriding.

Note: This route must come before the `<Routes/>` component to ensure it is matched first.

The following code example shows a replacement route to *MyHomePageComponent* enclosed in a `<Switch>`:

```
import { BrowserRouter, Route, Switch } from 'react-router-dom';
...
<BrowserRouter>
  <div className="app">
    <div className="my-header-navigation">
      <a href="/">Home</a> | <a href="/my-new-page">New Page</a>
    </div>
    <Switch>
      <Route path="/" component={MyHomePageComponent} />
      <Routes />
      <Route path="/my-new-page" component={MyNewPageComponent} />
    </Switch>
  </div>
</BrowserRouter>
```

Redirecting routes

You can redirect existing paths by using the react-router Redirect component.

Redirecting a route

The following code example imports the *Redirect* component and redirects the path `'/bring-me-home'` to `'/'`.

```
import { BrowserRouter, Route, Switch, Redirect } from 'react-router-dom';
...
<BrowserRouter>
  <div className="app">
    <div className="my-header-navigation">
      <a href="/">Home</a> | <a href="/my-new-page">New Page</a>
    </div>
    <Switch>
      <Route path="/" component={MyHomePageComponent} />
      <Redirect path="/bring-me-home" to="/" />
      <Routes />
      <Route path="/my-new-page" component={MyNewPageComponent} />
    </Switch>
  </div>
</BrowserRouter>
```

Removing routes

You can remove unwanted routes from IBM Cúram Universal Access.

You might want to reuse some but not all of the Universal Access `<Routes/>`. For those routes that you want to remove instead of replacing, use the react-router `<Redirect>` component to send users to a '404' style page, or some other valid end point.

You must declare the redirect before the `<Routes/>` component. You must also wrap the redirect in a `<Switch>` component. The following code example removes the route to "FAQ" by redirecting to a 404 page:

```
<BrowserRouter>
  <div className="app">
    <div className="my-header-navigation">
      <a href="/">Home</a> | <a href="/faq">FAQ</a>
    </div>
    <Switch>
      <Redirect path="/faq" to="/404page" />
      <Routes />
    </Switch>
  </div>
</BrowserRouter>
```

Advanced routing

IBM Cúram Universal Access is now code-split based on routes.

Code splitting

Code-split based on routes is achieved using *react-loadable* and the `@spm/universal-access-ui` package that is in the default *LoadingPage* component. For more information, see <https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#code-splitting> and <https://github.com/jamiebuilds/react-loadable>. The following example shows how to achieve the same split with the routes that you added:

```
import { LoadingPage } from '@spm/universal-access-ui';
...
const MyNewPageComponent = Loadable({
  loader: () => import(/* webpackChunkName: "MyNewPageComponent" */ '../features/
MyNewPageComponent'),
  loading: LoadingPage,
});
...
<Route
  component={MyNewPageComponent}
  exact
  path='/my-new-page'
/>
```

Titled routes

Accessibility rules require that a web page should have a descriptive title. You can implement a descriptive title using the *TitledRoute* component of the `@spm/universal-access-ui` package. To localize the title, *TitledRoute* exposes a title prop that accepts a *react-intl message ()* and can be used with or without code-split routes as shown in the following example:

```
import { TitledRoute } from '@spm/universal-access-ui';
import { defineMessages } from 'react-intl';
...
const titles = defineMessages({
  myNewPage: {
    id: 'app.titles.myNewPage',
    defaultMessage: 'My New Page',
  },
});
...
<TitledRoute
  component={MyNewPageComponent}
  exact
  path='/my-new-page'
  title={titles.myNewPage}
/>
```

Authenticated routes

You can protect parts of the application in two ways:

1. On access, handle authentication failures to a REST API and redirect to a login page.
2. Block access to specific routes to avoid any cost in running the REST API.

The following example shows how to block access to specific routes. The `@spm/universal-access-ui` package provides an `AuthenticatedRoute` component that accepts an `authUserTypes` array prop of the allowed user types to access this route. `AuthenticatedRoute` also wraps `TitledRoute` and therefore offers a `title` prop. The following is an example of using `AuthenticatedRoute`:

```
import { AuthenticatedRoute } from '@spm/universal-access-ui';
import { Authentication } from '@spm/universal-access';
import { defineMessages } from 'react-intl';
...
const titles = defineMessages({
  myNewPage: {
    id: 'app.titles.myNewPage',
    defaultMessage: 'My New Page',
  },
});
...
<AuthenticatedRoute
  authUserTypes={[Authentication.USER_TYPES.STANDARD, Authentication.USER_TYPES.LINKED]}
  component={MyNewPageComponent}
  exact
  path='/my-new-page'
  title={titles.myNewPage}
/>
```

The example blocks access to the `/my-new-page` routes for all users who are not of type `STANDARD` or `LINKED`, these users are redirected to the `/login` route.

Connecting to Universal Access REST APIs

You must connect your web application to IBM Cúram Social Program Management Universal Access REST APIs. You can use the mock server API service and the `RESTService` utility to help you to develop and test your REST API connections.

The mock server API service

The mock server is a mock API service that is provided to aid rapid development. The mock server serves APIs that simulate calling real web APIs. When you are developing your application, the mock server provides a lightweight environment against which the React components can be tested communicating with the services that provide their data.

Configuring the mock server

Configure the mock server location through the following properties in the `.env.development` file. You can change these values to suit your needs.

- `REACT_APP_REST_URL=http://localhost:3080`
- `MOCK_SERVER_PORT=3080`

Running the mock server

Run the mock server by using the following command from the root directory of your project:

```
npm run start:mock-server
```

However, when you are developing locally, you can use the following command that starts both the mock server and the client:

```
npm run start
```

See the `package.json` file in your project for the full list of commands.

Adding mock APIs

The universal-access project includes a number of mock APIs that simulate calling the SPM Universal Access APIs. These mock APIs support running some basic scenarios in development mode for the existing set of features.

As you develop your application, you typically create new APIs that you also want to mock. When the mock server starts, it looks to import the `/mock/apis/mockapis` file relative to the folder the command was started from. In this file, the mock-server expects to find three objects, GET, POST, and DELETE, that it can query to serve API requests for those HTTP methods.

The format of the mock definition is a relative URL that is assigned a JavaScript object. For example, the following code assigns the object `user` to the URL `/user`, and the object `payments.json`, which is read from a file, to the `/payments` URL.

```
const user = {
  'firstname': 'James',
  'surname': 'Smith',
  'gender': 'male',
  ...
}

const mockAPIsGET = {
  // ADD YOUR GET MOCKS HERE

  // Example of providing mock data in response to an API request in
  // the format uri:mockobject
  '/user': user,
  '/payments': readFile('./payments/payments.json')
};
```

If you use mocking extensively, it is better to use separate files and folders to structure your mocks.

Using universal-access mock APIs

The `mockapis.js` file is preconfigured to import and use mock APIs defined and exported by the universal-access package. This allows your project to reuse and extend the set of universal-access mock APIs.

```
const mockAPIs = require('@spm/universal-access-mocks');

// Extract the existing universal access GET,POST and DELETE mocks for merging.
const UAMockAPIsGET = mockAPIs.GET;
const UAMockAPIsPOST = mockAPIs.POST;
const UAMockAPIsDELETE = mockAPIs.DELETE;

...

//create custom mocks

...

// Merge UA mocks with custom mocks
const GET = Object.assign({}, UAMockAPIsGET, mockAPIsGET);
const POST = Object.assign({}, UAMockAPIsPOST, mockAPIsPOST);
const DELETE = Object.assign({}, UAMockAPIsDELETE, mockAPIsDELETE);

module.exports = { GET, POST, DELETE };
```

Where the same URL is used by a custom mock that was previously assigned to a universal-access package mock, the custom mock replaces the universal access version.

The RESTService utility

The @spm/core package provides the RESTService utility, which you can use to connect your application to a REST API. You can fetch resources with alternatives such as Fetch API, SuperAgent, or Axios. However, the RESTService utility provides some useful functions for connecting to SPM REST APIs.

The RESTService utility supports the GET, POST, and DELETE HTTP methods through the following JavaScript methods:

- `RESTService.get(url, callback, params)`
- `RESTService.post(url, data, callback)`
- `RESTService.del(url, callback)`

The full RESTService class documentation is in the doc folder in the @spm/core package.

The RESTService utility hides details of calls, such as passing credentials, language, and errors. The callback that is passed to the GET, POST, or DELETE methods is started after the API calls return. API calls are asynchronous, so write your code to expect and handle a delay in receiving a response.

The RESTService utility provides the following functions during communications.

Authentication

Authentication of the user is handled transparently by the RESTService utility. After a user is authenticated, the REST APIs automatically send the needed 'credentials', that is, the authentication cookies, with each request. For information about how authentication is handled for REST, see [Cúram REST API security](#).

If a user's session is invalidated before a new request is made to a REST API, then the '401 unauthorized' response is returned by the server. The RESTService utility relays the response to the callback function passed by the caller.

Handling responses

The RESTService utility formats the response from the server to ensure that callbacks receive the response in a consistent manner.

Each GET, POST, and DELETE method accepts a callback function from the caller. When called by the RESTService utility, the callback function receives a Boolean value that indicates the success or failure of the API call and the response. The callback function can then deal with the result. For example, a failure can be used to trigger your code to throw an error with the response data that can be used to trigger an error boundary. For more information about the callback function parameters, see the API documentation for the RESTService utility.

User Language

The 'Accept-Language' HTTP header is automatically set by the RESTService utility based on the user's selected language, which the user can select with the language picker in the application. This approach lets the server respond in the correct locale where locale sensitive information is being handled on the server.

The locale that is passed in the header is set in the transaction that is initiated by that REST request, and is used for the duration of that transaction. For more on transactions, see [Transaction control](#).

Handling timeouts

The RESTService utility can manage unresponsive calls to the server. You can set environment variables in the .env files to set thresholds for timeouts.

- `REACT_APP_RESPONSE_TIMEOUT=10` Wait 10 seconds for the server to start sending.
- `REACT_APP_RESPONSE_DEADLINE=60` but allow 1 minute for the file to finish loading.

Simulating slow responses

During development, it is important to test that your application continues to operate in an acceptable way even when network responses are slow. You can simulate a slow network connection by setting a property in the `.env.development` file in the root of your project.

For example, set `REACT_APP_DELAY_REST_API=2` to delay the response from all GET requests for 2 seconds. The value can be set to any positive integer to adjust the delay.

Related concepts

Authentication

The `universal-access` package exports the `Authentication` module, which can be used to log in and out of the application and to inspect the details of the current user. The login service is passed a username and password, and optionally a callback function that is called when the authentication request is completed.

Related reference

React environment variable reference

A full list of Universal Access React environment variables categorized by function. You can set environmental variables in `.env` files in the root directory of your application. If you omit environment variables, either they are not set or the default values apply.

Implementing file security and validation for the POST files API

You must have appropriate file security and validation for any documents that are uploaded to your system by the Document Service file upload API `POST /v1/dbs/files`. If you don't manage these functions elsewhere, such as in content management system or a gateway, then you can use the provided `RESTFileValidation` hook point. You can override the provided dummy implementation with your own custom implementation.

About this task

The `RESTFileValidation` interface consists of a single `validateFile()` method that must do all the required virus scans and file checks, such as the following:

- Scanning for viruses.
- Validating the size of the file.
- Validating the file extension.
- Validating that the file name does not contain paths.

The dummy implementation of the interface logs a warning to ensure that this function is implemented.

You can use Guice binding to overwrite the dummy implementation. For more information about using Guice, see [Configure Guice](#).

Procedure

1. Use a module class in a custom component. The component extends the `com.google.inject.AbstractModule` class.
2. Create a custom implementation of the `RESTFileValidation` interface. If there is a problem with a file, ensure that an `AppException` with the appropriate error message is thrown. The REST infrastructure manages the `AppException`, and returns the error message in the API response body of the JSON and the correct HTTP error code.
3. Bind the new custom implementation to the interface in the `configure()` method. For example, the `configure()` method can include an entry in the format:

```
bind(RESTFileValidation.class).to(RESTFileValidationCustomImpl.class);
```

Related tasks

Enabling or disabling verifications

The verifications feature is disabled by default. Use the **REACT_APP_FEATURE_VERIFICATIONS_ENABLED** environment variable to enable or disable the **Your documents** page and options in your application. When you enable verifications, it is available only to linked users.

Customizing file formats and size limits for file uploads

You can specify which file formats to allow users to upload and a size limit for uploaded files by setting environment variables. By default, the allowed file formats are JPG, JPEG, PNG, TIFF, and PDF and the file size limit is 5 MB.

Enabling the POST files API for file uploads

By default, the Document Service POST `/v1/dbs/files` API is disabled for security purposes. Before you can safely upload documents, you must enable the API in the IBM Cúram Social Program Management system administration application.

Before you begin

Before you enable the POST `/v1/dbs/files` API, ensure that you have implemented file security and validation for document uploads with the provided hook point. For more information, see [“Implementing file security and validation for the POST files API” on page 94](#).

Procedure

1. Log in to Social Program Management as a system administrator.
2. Select **System Configurations > Shortcuts > Application Data > Property Administration**.
3. Type `curam.rest.docservice.fileupload.enabled` in the **Name** field and click **Search**.
4. Select **... > Edit Value**.
5. Set the value to **TRUE** and click **Save** to save your changes.
6. Click **Publish** for your changes to take effect.

Related tasks

Enabling or disabling verifications

The verifications feature is disabled by default. Use the **REACT_APP_FEATURE_VERIFICATIONS_ENABLED** environment variable to enable or disable the **Your documents** page and options in your application. When you enable verifications, it is available only to linked users.

Customizing file formats and size limits for file uploads

You can specify which file formats to allow users to upload and a size limit for uploaded files by setting environment variables. By default, the allowed file formats are JPG, JPEG, PNG, TIFF, and PDF and the file size limit is 5 MB.

Adding metadata to file uploads

You can add metadata to files that you upload with the Document Service APIs by using the X-IBM-Curam-File-MetaData attribute. That metadata can then be used, for example, as search criteria in a content management system.

When a user wants to upload a file by using the Document Service API, a request is made to the following API endpoint:

```
POST /v1/dbs/files
```

The body of the request is the file that you want to upload in binary format. The Content-Type of the request is set to `application/octet-stream`.

```
// file data JSON object
const fileData = {
  caseReference,
  loggedInUserName,
  loggedInUserId,
  relatedPersonId,
```

```

        relatedPersonName,
        filename,
        classification,
        dateOfUpload: new Date(),
    }

    // Stringify the file data
    const metadata = JSON.stringify(fileData);

    // Set headers
    const headers = {
        'X-IBM-Curam-File-MetaData': encodeURIComponent(metadata),
    };

```

To add metadata that relates to the file being uploaded, the X-IBM-Curam-File-MetaData header is set to a URI encoded JSON string. The URI encoded JSON string is decoded on the server so it is expected to be URI encoded before sending the request.

```

// POST request
RESTService.post(
    '/v1/dbs/files',
    // File being uploaded
    data,
    (success, body, header) => {
        callback(success, body, header);
    },
    // Set request body format
    'application/octet-stream',
    null,
    // Headers including metadata string
    headers
);

```

The headers are then included in the POST request to /v1/dbs/files. These headers are not parsed or used in the default POST /v1/dbs/files API endpoint.

For more information, see [encodeURIComponent\(\)](#).

Universal Access REST APIs

The following IBM Cúram Social Program Management REST APIs are relevant to the key business functions of IBM Universal Access Responsive Web Application.

For the full list of supported Social Program Management APIs, see the Swagger specification, which is available from a running Social Program Management instance at <http://<hostname>:<port>/Rest/api/definitions/v1>.

Appeals

```
POST /v1/ua/appeals_form
```

Starts a new IEG execution for an appeal.

```
POST /v1/ua/appeals_form/exit
```

Exits the Appeals IEG Form.

```
GET /v1/ua/appeals
```

Returns the list of appeals for the logged in user.

```
GET /v1/ua/appeals/{online_appeal_request_id}/attachment
```

Returns the attachment document for an appeal request.

Applications

```
GET /ua/online_categories
```

Returns a list of Online Categories. Each category includes details of the applications that a user can apply for.

```
GET /ua/submitted_applications
```

Returns a list of applications that were previously submitted by the logged in user.

```
POST /ua/submitted_applications/{application_id}/application_programs/{application_program_id}/withdrawal_request
```

Creates a withdrawal request for the specified program in a submitted application. The application can be withdrawn only if it has a status of pending, and if there is not already a pending withdrawal request for this application. For each program associated with the submitted application, a separate withdrawal request must be created. Either a `withdrawalReason` or `reasonText` value must be supplied, but not both. See `/withdrawal_request_reasons` for the list of possible withdrawal reasons that were configured for the associated application type.

```
GET /ua/submitted_applications/{application_id}/application_programs/{application_program_id}/withdrawal_request_reasons
```

Returns a list of possible withdrawal reasons that a user can choose when they withdraw an application.

```
GET /ua/application_types
```

Returns details of the application type definition of the specified draft or submitted intake application.

```
GET /ua/application_types/{application_type_id}
```

Returns details of the specified application type.

```
GET /ua/application_submission_message
```

Returns details of an application submission message.

```
GET /ua/application_confirmation_message
```

Returns details of an application confirmation message. The details are configurable by an administrator, by updating the details for the associated application type definition.

```
GET /ua/draft_applications
```

Returns a list of draft applications that are currently in-progress for the logged in user.

```
GET /ua/submitted_applications/{application_id}/attachment
```

Returns the attachment for the specified submitted application.

```
GET /ua/form_details/{application_form_id}
```

Gets details of a form instance.

```
POST /ua/application_form
```

Starts a new intake application form for the logged in user. Under the hood, a new datastore is created to store the data provided in the application form, for later use for when the user is ready to submit their intake application.

```
DELETE /ua/application_form/{application_form_id}
```

Cancels the specified intake application form without saving the details, which means the application form cannot be retrieved or resumed at a later stage.

```
POST /ua/submission_form
```

Starts a submission form for the logged in user, which is used in association with the specified intake application form.

```
GET /ua/submission_form/{submission_form_id}
```

Gets details of a submission form instance.

```
GET /ua/submission_form/{submission_form_id}/page_details
```

Returns details of questions for a single page of the specified form. If the page query parameter has a value of 'next', or is left empty, then the questions that are returned are for the next unanswered page in the application, or the first page if no answers were submitted yet. If the page query parameter has a value of 'previous', the questions that are returned are for the page previous to the last answered page. In this way, it is possible to navigate through the pages of a form, however it is not permitted to jump directly to a specific page.

```
POST /ua/applications
```

Creates an intake application based on the data that was previously supplied in the specified intake application and submission forms.

Document service

```
POST /v1/dbs/files
```

Uploads a file and returns the URL for the file. This API is disabled by default for security purposes, so you must implement file security and validation and enable the API before you can use upload files to your system. For more information, see [“Implementing file security and validation for the POST files API” on page 94](#) and [“Enabling the POST files API for file uploads” on page 95](#).

```
GET /v1/dbs/files/{file_id}
```

Retrieves a specified file.

```
DELETE /v1/dbs/files/{fileId}
```

Deletes a specified file. This API is not currently used by the Universal Access Responsive Web Application.

Life events

```
GET /ua/life_event_categories
```

Gets the list of life event categories and the life event contexts (of type Citizen/Online) that are contained inside those categories, and the life event contexts that are not associated with any category.

```
POST /ua/life_events_form
```

Get the formId given the lifeEventsContextId.

```
GET /ua/life_events_form/{formId}
```

Gets the Life Event Context record based on the IEG form.

```
POST /ua/life_events_form/exit
```

Submits the Life Event IEG form.

```
GET /ua/life_events_history
```

Get the life event history.

```
GET /ua/life_event_remote_systems/{formId}
```

Gets the list of Remote Systems that are associated with the Life Event Context of the specified Life Event Form.

```
POST /ua/life_event_remote_systems/{formId}
```

Sends the Life Event data to the selected Remote Systems.

Messages

```
GET /ua/messages
```

Returns a list of system messages and account messages applicable for the logged in user.

Notices

```
POST /v1/ua/communications/{communication_id}/mark_send_by_post
```

Mark a communication to be sent by mail. An attribute on the return of the API indicates whether a send by mail request exists for the communication.

```
GET /v1/ua/communications/
```

Returns the list of communications for the logged in user.

```
GET /v1/ua/communications/{communication_id}
```

Returns a communication.

```
GET /v1/ua/communications/{communication_id}/attachments/{attachment_id}
```

Returns the communication attachment details.

Organization

```
GET /ua/organisation
```

Returns the details of the organization.

```
GET /ua/local_offices
```

Returns a list of local offices. The list can be filtered either by county or by ZIP/postal code.

Payments

```
GET /ua/payments
```

Returns a list of payments for the logged in user. The returned list is ordered by payment date, with the most recent payment listed first. The list can be filtered to return a single payment by supplying both query parameters of `payment_id` and `isPaymentByExternalParty`.

```
GET /ua/payment_messages
```

Returns details of the user's next payment.

System

```
GET /ua/system_configurations
```

Returns a list of system properties. The list can be filtered to return a single system property by supplying the property ID.

```
GET /ua/app_image_resource
```

Returns the requested image resource.

```
GET /ua/icons/{icon_id}
```

Returns the requested icon.

User

```
GET /v1/ua/user
```

Returns information that is related to the current user, such as user permissions.

```
GET /ua/profile
```

Returns details of the logged in user.

```
GET /ua/profile_image/{image_id}
```

Returns the requested profile photo. This photo must belong to the logged in user. See /profile for retrieving the details for the value to use for {image_id}.

```
POST /ua/user_account
```

Creates a user account.

```
POST /ua/email_password_reset
```

Sends a reset password email to the email address registered for the user.

```
POST /ua/secret_question_password_reset
```

Resets the user's password, with the new password specified. The user's secret question and answer must be valid, in order for the password to be successfully reset.

```
POST /ua/password_reset
```

Resets the user's password, with the new password specified. The specified existing password must be valid, in order for the password to be successfully reset.

```
POST /ua/generated_user_accounts
```

Generates a temporary user account to be used to log in to the Social Program Management system under the hood, when the citizen user has not logged in or created their own user account. This account temporarily stores the details of the citizen user, for example any intake applications or benefits they start, and transfers these details to a permanent user account if the user signs up or logs in with their own account at a later stage.

```
POST /ua/application_form_ownership
```

Changes the ownership of the specified intake application form to the currently logged in user. This action can be completed only if the previous owner of the intake application form is a system-generated user, it is not permissible to use this API to change the ownership from one citizen account user to another.

```
GET /ua/user_account_login
```

Retrieve the users last successful login date time.

```
GET /ua/case_contacts
```

Returns a list of contact information for the caseworkers that are related to the logged in user's cases.

Screening

```
GET /ua/screening_form
```

List all screening forms for the current user.

```
POST /ua/screening_form
```

Starts a new IEG execution based on the Screening Type.

```
GET /ua/screening_form/{formId}
```

Gets the Screening Type and Program selection for a specified Screening Form.

```
POST /ua/screening_form/{formId}
```

Updates the Program selection for specified Screening Form.

```
DELETE /ua/screening_form/{formId}
```

Delete a screening form.

```
POST /ua/filter_screening_form
```

Starts a new Filter Screening IEG execution based on the Screening Type.

```
POST /ua/screening_form/exit
```

Exits the Screening IEG Form.

```
GET /ua/screening_form/{formId}/results
```

Get the Screening Results.

Verifications

```
POST /v1/ua/verifications/link_file
```

Links an existing file on the system to a specified verification. A link record is created to link the file and the verification.

```
GET /v1/ua/verifications
```

Returns details for all verifications for a specified person or case.

Universal Access passes in the ConcernRoleID for the primary participant, which returns verifications for all case participants on all active cases where the person is the primary participant.

```
GET /v1/ua/verifications/{verificationId}
```

Returns details for a specified verification.

```
DELETE /v1/ua/verifications/{verificationId}/file_links/{link_id}
```

Removes a link between a file and a verification without deleting the file. The link record is deleted. You can delete the file by using the document service `DELETE /v1/dbs/files/{fileId}` API.

Developing with headers and footers

IBM Cúram Universal Access contains a predefined header and footer. The header and footer contain content that is found in the header and footer of an application, such as links, **log in**, and **sign up** buttons, and menus for logged in users.

Headers and footers

You can customize your application headers and footers by replacing the sample components with your own custom versions.

The App.js file in the *universal-access-sample-app* module, reuses the sample *ApplicationHeader* and *ApplicationFooter* components that are provided by the universal-access module by placing them above and below the main content of the application:

App.js

```
<BrowserRouter>
  <ScrollToTop>
    <div className="app">
      <a className="wds-c-skipnav" href="main-content">
        {formatMessage(translations.appSkipLink)}
      </a>

      <Route path="/" component={ApplicationHeader} />
      <main id="main-content" className="main-content">
        <Content>{routes}</Content>
      </main>

      <ApplicationFooter />
    </div>
  </ScrollToTop>
</BrowserRouter>
```

Header

Typically, an application header has two views. One view has items relevant to users who are not logged in or signed up, for example a **Sign Up** button. The second view shows items that are relevant to users who are signed up and logged in, for example an **Update your profile** button.

To facilitate the separate views, use a react-router-dom *Route* component. The App.js sample demonstrates wrapping the *ApplicationHeader* component in a *Route* component and passing *Route* information to the *ApplicationHeader*. This allows the *ApplicationHeader* to query the *Route* properties and decide what to display based on the current location in the application. For example, you might want to show a different view for the login page route (*'my-app-domain/login'*) from the application home page route (*'my-app-domain/'*).

The following code sample shows how the *ApplicationHeader* queries its location property to find out what page the application is displaying. The sample code then uses this information to decide what to show in the header.

```
get isOnLoginPage() {
  return this.props.location.pathname === '/login';
}

render() {
  return (
    <Header
      title={this.pageTitle}
      type="scrollable"
      logo={<img src={logo}
        alt="agency"
        id={this.props.loggedInUser} />}>
    <PrimaryNavigation type="scrollable">
      <TabList scrollable>
        <Tab
          id="tab1"
```



```

      href="/"
      text={
        this.props.intl.formatMessage(translations.headerHomeLabel)}>
        <Tab
          id="tab2"
          href="/my-applications"
          text={this.props.intl.formatMessage(
            translations.headerBenefitsLabel)}>
        </Tab>
      </PrimaryNavigation>
    <SecondaryNavigation type="Scrollable">

      {/* Show signed out menu */}
      {!this.isOnLoginPage &&
        this.props.loggedInUser === null &&
        !this.isUserProfileLoaded &&
        this.signInMenu}

      {/* Show signed in menu */}
      {this.props.loggedInUser &&
        this.isUserProfileLoaded &&
        this.profileMenu}
    </SecondaryNavigation>
  </Header>
);
}

```

Login and sign up in the header

If you are building your own customer header, you must identify which page you are currently displaying the Header on, you must also differentiate between logged in and logged out users. Whether a user is logged in or out can be determined by using the authentication API provided by the universal-access module. The Authentication API provides functions to allow you to log in and out of the application, and also allows you to query if a user is logged in and who that user is. For more information, see the Authentication API documentation.

The following code sample shows how the *ApplicationHeader* uses the Authentication API. In this function, a check is made to see whether a user is logged in before it loads that user's profile. The user's profile is needed to display the user's full name in the header.

```

fetchProfile() {
  if (Authentication.isLoggedIn() && !this.isUserProfileLoaded) {
    this.props.loadProfile();
  }
}

```

Footer

You can add a footer to the bottom of the application page in the same way as you add the header to the top of the page. The universal-access module provides a sample application footer that is used in the universal-access-sample-app, see the App.js sample. The sample footer is static and does not change based on the location or the authentication state, however the footer can be made dynamic by following the example from the header.

Adding images, fonts, and files

As the Universal Access Responsive Web Application is based on create-react-app, you can follow one of their standard approaches for adding images, fonts, fonts and files, depending on whether you are adding images for IEG scripts.

For the application in general, you can co-locate the image or file with the component that requires the resource, then import this resource within the component as follows:

```

import React from 'react';
import image from './image.png';

const Component = () => {

```

```

    return <img src={logo} alt="Logo" />;
  };

  export default Component;

```

For more information, see [Adding Images, Fonts, and Files](#) in the create-react-app documentation.

Adding images for IEG scripts

Some IEG <Text> elements support rich text content that might include HTML tags. If you need to add an image as part of the text, the URL of the image must target to a resource in the public folder of the application, for example:

- Create an `img` folder in the public directory of your application. The relative path should look like this `universal-access-sample-app/public/img`.
- Store the image in the `img` folder, for example `universal-access-sample-app/public/img/image.png`.
- Define an IEG text element in the script, for example `<display-text id="DisplayText.Image"/>`.
- Define the content of the property as an HTML image tag in the property file :

```
DisplayText.Image=
```

Where the `src` path points to the folder created on the public folder.

Images added in this way are not sized to device screen sizes, therefore take a mobile-first approach when adding images to IEG Scripts.

For more information about adding resources to the public folder, see [Using the Public Folder](#) in the create-react-app documentation.

Customizing the color scheme or typography

You can customize the color scheme to display different colors and typography by using Sass. You do not modify CSS files directly, however, you can use CSS in the Sass files if you prefer.

Using Sass

The design system uses the [Sass](#) CSS preprocessor. You can use Sass to declare variables in CSS. You can define variables for colors, spacing, and typography in a single place and then reuse the variables throughout the design system stylesheets. To see the variables that are defined, view the `node_modules/@govhhs/govhhs-design-system-core/src/stylesheets/core/_variables.scss` file in your application. The Sass files are compiled into CSS at build time and your application uses the compiled CSS.

The file structure of the starter pack

The starter pack is configured to use Sass, the relevant files are located in a `css` folder and a `sass` folder under the `src` folder in the file structure.

```

.
├── src
│   ├── css
│   │   └── styles.css
│   └── sass
│       ├── customVariables.scss
│       └── styles.scss

```

The `css` folder contains the styles that your application uses.

Note: The contents of the `css` folder are generated at build time. Don't directly edit any files inside the `css` folder. For more information, you can view the **build-css** script in the project's `package.json` file.

You must edit the Sass files to make changes. If you don't want to use Sass features or if you don't have previous experience of SaaS, you can still write regular CSS into these files. By default, the sass folder contains two files:

- `styles.scss`. Use this file to import the design system stylesheets and all other styles that the app might use.
- `custom-variables.scss`. Customize the file by overriding the design system variables values with your intended values.

Other than changing the design system variables, do not add styling. However, if you want to add extra styling for your application, create a file in the sass folder. Then import the file in the `styles.scss` file as follows:

```
@import "my-custom-styles.scss";
```

Changing the color palette

When you select a color scheme for your site, ensure that color contrast is satisfactory. For users with low vision, low-contrast text is difficult or impossible to read. For more information about color contrast, see the [Text elements must have sufficient color contrast against the background](#). The color-related variables are in the color section of the design system's variables file, that is, `node_modules/@govhhs/govhhs-design-system-core/src/stylesheets/core/_variables.scss`.

```
// node_modules/@govhhs/govhhs-wds-design-system-core/src/core/_variables.scss
//-----
// □ Color
//-----
$color-primary:           color('blue', 50) !default;
$color-primary-darker:    color-shade($color-primary, 10) !default;
$color-primary-darkest:   color-shade($color-primary, 20) !default;
$color-primary-light:     color-tint($color-primary, 10) !default;
$color-primary-lightest:  color-tint($color-primary, 50) !default;
.....
```

The `-darker`, the `darkest`, the `light`, and the `lightest` variants are derived from the base `color-primary`. To obtain the derived color values, use the Sass `lighten` and `darken` utilities. Alternatively, use hardcoded values. To customize, override the values for the variables.

Example

This example shows how to update both the color scheme, that is the primary, secondary, link colors, and the typography of the application.

The example updates the application with the following color scheme:

- `#051380` as the primary color, that is, used on page headers, primary buttons, and hover states.
- `#37056b` as the application's secondary color, that is, used for avatar backgrounds.
- `#2b4380` for the link colors, `#0535d2` for the link hover color, and `#7834bc` for visited links.

The example updates the application with this typography:

- 20px font size with a 33px line height for the body text with a 400 font weight
- 16px font size with a 26px line height for small text with a 400 font weight

1. To start the application, enter the following command from your application. The application is accessible on your local host.

```
npm start
```

2. Edit the `sass/custom-variables.scss`.
3. Add the intended value to the primary color:

```
$color-primary: #051380;
```

4. Define the `-darker`, the `-darkest`, the `light` and the `lightest` variants by using the `lighten` or the `darken` utilities.

```
$color-primary-darker: darken($color-primary, 10%);
$color-primary-darkest: darken($color-primary, 20%);
$color-primary-light: lighten($color-primary, 10%);
$color-primary-lightest: lighten($color-primary, 50%);
```

5. Define the secondary colors.

```
$color-secondary: #37056b;
$color-secondary-dark: darken($color-secondary, 10%);
$color-secondary-darkest: darken($color-secondary, 20%);
$color-secondary-light: lighten($color-secondary, 10%);
$color-secondary-lightest: lighten($color-secondary, 50%);
```

6. Save the file. The app is reloaded in the browser so you can see your changes.

7. To define the link colors, use the proceeding `color-link`, the `color-link-hover`, and the `color-visited` variables.

```
$color-link: #2b4380;
$color-link-hover: #0535d2;
$color-visited: #7834bc;
```

8. To change the typography, override the `body-font` and `small-font` variables.

```
// Body font
$body-font: (
  'font-size': 20px,
  'line-height': 33px,
  'font-weight': 400
);

//Small Font
$small-font: (
  'font-size': 16px,
  'line-height': 26px,
  'font-weight': 400
);
```

9. Save the file to see your changes.

The final `custom-variables.scss` file for the example is shown.

```
$icon-path: "~@govhhs/govhhs-design-system-core/dist/icons";
$image-path: "~@govhhs/govhhs-design-system-core/dist/img";

.success-icon-color {
  fill: #3dc06e !important;
}

// Primary color
$color-primary: #051380;
$color-primary-darker: darken($color-primary, 10%);
$color-primary-darkest: darken($color-primary, 20%);
$color-primary-light: lighten($color-primary, 10%);
$color-primary-lightest: lighten($color-primary, 50%);

// Secondary color
$color-secondary: #37056b;
$color-secondary-dark: darken($color-secondary, 10%);
$color-secondary-darkest: darken($color-secondary, 20%);
$color-secondary-light: lighten($color-secondary, 10%);
$color-secondary-lightest: lighten($color-secondary, 50%);

// Link colors
$color-link: #2b4380;
$color-link-hover: #0535d2;
$color-visited: #7834bc;

// Body font
$body-font: (
  'font-size': 20px,
  'line-height': 33px,
  'font-weight': 400
);
```

```
);
//Small Font
$small-font: (
  'font-size': 16px,
  'line-height': 26px,
  'font-weight': 400
);
```

Developing toast notifications

A toast as a computing term refers to a graphical control element that communicates certain events to the user without forcing them to react to the notification immediately. In IBM Curam Universal Access, we use the web design system Alert component as a base to represent our toast notifications and allow capability to display these notifications independent of the main display content in any function within the application.

The <Toaster> component

The exposed <Toaster> component is used in App.js and is responsible for rendering toast notifications retrieved directly from the Redux store. These notifications are displayed independent of page content. This means that a deeply nested function can be used to display a notification without regard to the current component render and/or functionality that is used to navigate to different pages.

The <Toaster> component handles the retrieving of toast slice within the store, and in passing functionality to remove toast notifications once they have been dismissed.

The <Toast> component

The exposed <Toast> is the preferred component to display toast notifications. It accepts properties as defined by the web design system Alert component, without requiring the need to specify the component as an Alert and the properties 'banner', 'center', and 'toast'. It also requires a 'text' property to be defined.

The Toaster module

Any component that intends to display a toast notification within its processing must use the Toaster module action fillToaster function. This can be either passed to component as a property, or connected to the Redux store and defining the action as a property. For more information, see [“Universal Access Redux modules” on page 79](#).

An example of a page that implements the Toaster module action fillToaster and a service unavailable toast notification is shown.

```
import React from 'react';
import { connect } from 'react-redux';
import { ToasterActions } from '@spm/universal-access';
import { Toast } from '@spm/universal-access-ui';

...

/**
 * Updates the Toast slice of Redux store
 * @param {*} dispatch the dispatch function
 */
export function mapDispatchToProps(dispatch) {
  return {
    fillToaster: data => {
      ToasterActions.fillToaster(dispatch, data);
    },
  };
}

class MyComponent extends React.Component {

  ...

  doSomething({ success }) {
    if (success) {
```

```

    ...
  }
  else {
    this.props.fillToaster(
      <Toast
        dismissable={false}
        expireAfter={5}
        text="This service is currently unavailable"
        type="danger"
      />
    );
  }
}
...

export default connect(
  null,
  mapDispatchToProps
)(MyComponent);

```

Localization

You can add languages to the application, and apply regional settings for calendar and date formats, and for currencies.

Related information

[Developing for Regional Support](#)

Selecting a language

Citizens can select a preferred language from the **language** drop-down in the footer of the application. When citizens select a preferred language, the application is displayed in that language. The application retains the preferred language setting based on a cached value in the browser.

Note: The **language** drop-down only appears when more than one language is configured for the application.

Note: A citizen's language preference is not saved if the browser is configured to block access to its local storage, the application reverts to the default language (English) when the page is reloaded.

Configuring languages in the application

You can add languages to the application or change the default language. You must create a `src/config/intl.config.js` file to be read by the `src/intl/IntlInit.js` component, which handles storage of the configuration and creates the *react-intl IntlProvider*.

About this task

Review the `src/config/intl.config.js.sample.md` file, which contains the `intl.config.js` object schema and an example `src/config/intl.config.js` file.

Translated messages for the default supported languages are provided in the following locations:

- For universal-access-ui components, in the universal-access-ui-locales package at `/node_modules/@spm/universal-access-ui-locales`.
- For the core-ui components, in the core-ui-locales package at `/node_modules/@spm/core-ui-locales`.
- For the intelligent-evidence-gathering components, in the intelligent-evidence-gathering-locales package at `/node_modules/@spm/intelligent-evidence-gathering-locales`.

Procedure

Create a `src/config/intl.config.js` file with reference to the following example from the `src/config/intl.config.js.sample.md` file.

```

export default {
  defaultLocale: 'en',
  locales: [
    {
      locale: 'en',
      displayName: 'English',
      localeData: require('react-intl/locale-data/en')
    },
    {
      locale: 'de',
      displayName: 'German',
      localeData: require('react-intl/locale-data/de'),
      messages: {
        ...require('@spm/core-ui-locales/data/messages_de'),
        ...require('@spm/intelligent-evidence-gathering-locales/data/messages_de'),
        ...require('@spm/universal-access-ui-locales/data/messages_de')
      }
    },
    {
      locale: 'ar',
      displayName: 'Arabic',
      direction: 'rtl',
      localeData: require('react-intl/locale-data/ar'),
      messages: {
        ...require('@spm/core-ui-locales/data/messages_ar'),
        ...require('@spm/intelligent-evidence-gathering-locales/data/messages_ar'),
        ...require('@spm/universal-access-ui-locales/data/messages_ar')
      }
    },
    {
      locale: 'ht',
      displayName: 'Haitian',
      /*
       Custom locale data
       Where the locale you need to support is not found in the react-intl locale data you can
       create your own locale data to handle this.
       Simply create an object with the locale property. You must include at a minimum the
       pluralRuleFunction.
       See https://github.com/yahoo/react-intl/issues/1050
      */
      localeData: {
        locale: 'ht',
        pluralRuleFunction(arg1, arg2) {
          return arg1 && arg2 === 1 ? 'one' : 'other';
        }
      },
      messages: require('../locale/messages_ht')
    }
  ]
};

```

Translating your application

Use *react-intl* and *babel-plugin-react-intl* to extract text from your application. You can then translate the text into another language and include that translation in the application.

Extracting translatable content

During development, IBM used *react-intl* (<https://github.com/yahoo/react-intl>) and *babel-plugin-react-intl* (<https://github.com/yahoo/babel-plugin-react-intl>) to globalize IBM Cúram Universal Access.

About this task

Follow the same method as used by IBM during development to extract the translatable content from your application.

Note: *react-intl* provides react components and an API to format dates, numbers, and strings, including pluralization, and handling translations. *babel-plugin-react-intl* extracts string messages from React components that use *react-intl*.

Procedure

1. Use the *react-intl* `defineMessages` API to define the default message string entry within the application.

2. Add *babel-plugin-react-intl* and its dependencies *babel-cli* and *babel-preset-react-app* to the application's *devDependencies*.
3. Add a *.babelrc* file in the root of your project. Use *.babelrc* to configure the settings for the *babel-plugin-react-intl*. The following is an example *.babelrc* file:

```
{
  "presets": ["react-app"],
  "plugins": [
    [
      "react-intl", {
        "messagesDir": "translations/messages",
      }
    ]
  ]
}
```

4. Add the following line to your package.json "scripts":

UNIX:

```
"extractTranslations": "NODE_ENV=production babel ./src >/dev/null"
```

Windows:

```
"extractTranslations": "set NODE_ENV=production&&babel ./src > NUL"
```

5. Run the extraction command: **npm run extractTranslations**.

Results

This procedure extracts all translations to the *translations/messages* directory as specified in the *.babelrc* configuration.

The content of *translations/messages* along with the JSON content under the locale directories of the @spm/core-ui, @spm/universal-access-ui, and @spm/intelligent-evidence-gathering directory form what should be sent for translation.

Including translated content in your application

IBM Cúram Universal Access exposes a *src/intl/IntlInit* component. This component reads the configuration provided in the custom *src/config/intl.config.js* to seed your application with messages for all the languages you want your application to support.

About this task

Procedure

1. Translations must be returned for use in your product in the format of a single JSON file per locale. This JSON file should be in the format expected by *react-intl*, which is `{[id: string]: string}`, as shown in the following example:

```
{
  "label1": "Translated text1",
  "label2": "Translated text2",
}
```

Where *id* is the id that is used in your *defineMessages* entry and subsequent extracted message id.

Note: The id in this file format `{[id: string]: string}` must match the id that you define in your code as in the *defineMessages* structure. For more information, see <https://github.com/yahoo/react-intl/wiki/API#definemessages>.

This single file and its location within the application forms the entry to the messages value with the `intl.config.js` for your configured locale, for example:

```
{
  locale: "de",
  displayName: "German",
  localeData: require("react-intl/locale-data/de"),
  messages: require("../locale/messages_de")
},
```

2. *react-intl* also requires that its own locale configuration (`localeData`) is provided to support some of its internal functions. For more information, see <https://github.com/yahoo/react-intl/wiki#loading-locale-data>.

Results

When you have configured it correctly with the `src/config/intl.config.js` file, the *ApplicationFooter* language selection drop-down should expose your new locale selection, it should also load and apply the configured translation messages to the application.

Note: If your application does not find messages for the currently selected language at run time, *react-intl* defaults to the text of the *defaultMessage* entry that was used when the message was defined in the source code.

Regional settings

The universal-access module and its components respect the regional settings that are defined in IBM Cúram Social Program Management to ensure your application is synchronized with the Social Program Management instance on which it depends.

Regional settings for currencies, and for calendar and date formats in the user interface, are defined in Social Program Management. For more information, see *Developing for Regional Support*.

Related information

[Developing for Regional Support](#)

Development scenarios

As a developer, use these simple scenarios to learn how customize the IBM Universal Access Responsive Web Application.

The first scenario shows how to change default text on the **My Details** page. Each subsequent scenario adds to the previous one to build out new content in your application.

Note: Follow the scenarios in sequence. If you start in the middle of the scenario list, you might have to go back through previous scenarios.

Changing text in the application

You can change the default text in the application by providing custom text that overrides the default text for any language. In this scenario, an English language message is changed.

Before you begin

You can find text in the application components and in IEG forms.

- Text in IEG scripts. For more information, see [Changing Static Text](#).
- Text in IEG configuration settings. For more information, see [Using configuration properties to customize IEG pages](#).

About this task

Message or text strings in the application use the `react-intl` package, which supports globalization of React applications. `react-intl` allows the messages to be extracted and translated to other supported languages, it also adds placeholders for data, for example.

To change the existing text of any of the languages that are provided by IBM, you must provide a custom version of the message that is mapped to the same message `id`.

Procedure

1. To change an English language message, find the ID of the message you want to replace. In your project, go to `/node_modules/@spm/universal-access-ui/locale`.
 - a) The `locale` folder contains message files for each supported locale. For your chosen language, search the appropriate `message_xx.json` for the text string that you want to replace. For example, to change the English text **Apply for a benefit**, search `messages_en.json` for that string as shown in the following example. If there is more than one instance of the string, you must find the correct message ID for the text you want to change. The simplest way to find the correct instance is to try replacing each ID one by one, reloading the page each time to see whether the new string is displayed.

```
"System_Messages_Alert_Description": "System messages alert description",

"Payments_NoPaymentMessages": "No payment messages",

"Payments_ApplyForABenefitLink": "Apply for a benefit",

"TODO_NoTODOMessages": "No to-dos",

"TODO_CaseworkerMessage": "Your caseworkers can create to-dos for you.",

"Meetings_NoMessages": "No meetings",
```

- b) For the **Apply for a benefit** string, use the associated ID `"Payments_ApplyForABenefitLink"` to override the message in your custom `messages_en.json`.
2. Create a custom message file by creating a `messages_en.json` file in the `src/locale` folder. Custom messages are injected into the application at application start. For more information, see [“Localization” on page 108](#). By default, the starter application provides a `locale` folder from where custom messages files are automatically loaded. You can add your custom file to this folder: `src/locale`.
 3. To replace the message, create a new `id:message` mapping in your custom message file by using the same ID value as shown in the following example.

```
"Payments_ApplyForABenefitLink": "Click here to apply for a benefit",
```

4. Update the `src/config/intl.config.js` file in the English locale to point to the custom messages file.

```
// [...]
{
  locale: 'en',
  displayName: 'English',
  localeData: require('react-intl/locale-data/en'),
  messages: require('../locale/messages_en'),
},
// [...]
```

Related concepts

[Localization](#)

You can add languages to the application, and apply regional settings for calendar and date formats, and for currencies.

Adding content to the application

Build on the text change scenario from *Changing application text* to add a route. You also add content that is displayed when the route is loaded.

Before you begin

If you are not familiar with React and React Router, you must take a basic course in building a web application with React and React Router.

The term "feature" refers to the content that is displayed when a route is loaded, this content is what citizens see on the user interface. A feature is an abstraction that includes all the content that comes together to create the user experience. A feature can be a collection of JavaScript files, JSON files, and APIs that work together to generate the user experience. The term "feature" can be referred to as a page, view, or component in other application environments.

This scenario adds a feature that presents a logged-in person's details in the main content area when a / person URL is loaded. This scenario is built on in later scenarios by calling APIs, by using client-side stores, error handling, or globalization.

About this task

When you extend the IBM Cúram Universal Access reference application, you might want to introduce new content that is displayed when citizens click a link.

Procedure

1. Create the content for the feature, take the following steps:
 - a) Create a folder called `features` under the `/src` folder in your project
 - b) Create a person subfolder and create `PersonComponent.js` in the folder.

```
src/features/Person/PersonComponent.js
```

- c) Add some HTML to display when the component is loaded. The following example displays some data that is returned by an API:

```
import React from 'react';

const Person = () => { return (
  <div>
    <h1>James Smith</h1>
    <h2>Gender: Male</h2>
    <h2>Born: April 1st 1996</h2>
  </div>
)};
export default Person;
```

2. Add a route to link to your feature, take the following steps:
 - a) Declare an associated URI for each feature in the application. The URI allows React to present the feature when the URI is requested in the browser. This technique is standard 'React Routing' for displaying features. For more information about routes, see ["Developing with routes" on page 88](#). Add a simple component that displays when the route is loaded:
 - 1) Open `routes.js` in your project.
 - 2) Import a `Person` component from the folder `features/person`.
 - 3) Add a `"/person"` route that loads the `Person` component as shown in the following example:

```
import React from 'react';
import { Route, Switch } from 'react-router-dom';
import { Routes as UARoutes } from '@spm/universal-access-ui';
```

```
import Person from './features/PersonComponent'

export default (
  <Switch>
    <Route path="/person" component={Person} />
    <UARoutes />
  </Switch>
);
```

3. Load the new feature by using the route, take the following steps:

a) Run your application, enter the following command:

```
npm run start
```

b) Start a browser and enter the full URL for the feature, for example: <http://localhost:8888/person>

Results

When the application loads, the person details are displayed in the main content area.

Related concepts

Developing with routes

Routes define the valid endpoints for navigation in your application. Your application consists of a network of routes that can be traversed by your users to access the application's pages.

Styling content with the Social Program Management Design System

Build on the route and person content scenario that you added in *Adding content to the application* by styling the content of a person's details.

Before you begin

The IBM Social Program Management Design System is a design framework that helps you to build a cohesive and consistent application. By selecting components from a design catalog and applying design principles, design and development is faster and user experience is improved.

About this task

The full catalog of Social Program Management Design System components, including descriptions of when and where to use them, is documented in the *govhhs-design-system-react* package. You can access these packages through `index.html` file in `/node_modules/@govhhs/govhhs-design-system-react/docs`. This scenario uses a number of Social Program Management Design System components to improve the person feature.

Procedure

1. Import contents from the Social Program Management Design System. Enter the following command to import the *Avatar* and *MediaObject* components from the package `@govhhs/govhhs-design-system-react`:

```
import {Avatar, MediaObject} from '@govhhs/govhhs-design-system-react'
```

2. Update `PersonComponent.js` to use the `Grid`, `Column`, `Card`, `MediaObject`, `Avatar`, and `List` components to display the person's details. You can also include an address in a separate card.

Use the following code to replace the previous `PersonComponent.js`:

```
import React from 'react';
import {Grid, Column, Card,CardBody,CardHeader, List, ListItem, Avatar, MediaObject } from '@govhhs/govhhs-design-system-react'

const avatarMediaJames = <Avatar initials="JS" size="medium" tooltip="profile photo" />;
const Person = () => {
  return (
    <Grid className="wds-u-p--medium">
      <Column width="1/2">
        <Card>
```

```

        <MediaObject media={avatarMediaJames} title="James Smith">
          <List>
            <ListItem>Gender: Male</ListItem>
            <ListItem>Born: April 1st 1996</ListItem>
          </List>
        </MediaObject>
      </Card>
    </Column>
    <Column width="1/2">
      <Card title="Address">
        <CardHeader title="Address"/>
        <CardBody>
          <List>
            <ListItem>1074, Park Terrace</ListItem>
            <ListItem>Fairfield</ListItem>
            <ListItem>Midway</ListItem>
            <ListItem>Utah 12345</ListItem>
          </List>
        </CardBody>
      </Card>
    </Column>
  </Grid>
});
export default Person;

```

3. Save `PersonComponent.js`.

Results

When you reload the application, you see the updated application style.

Changing the application header or footer

Build on the styling scenario from *Using the Social Program Management Design System to style content* by adding a link to the application header or footer.

Before you begin

For more information about the application header and footer, see *Developing with headers and footers*.

To customize the header, you must create your own custom version. To keep this scenario brief, work on the header only and copy the existing header from `universal-access-ui`. Make some small changes to the header to show how it can be customized. Alternatively, completely replace the header or footer with your own version.

About this task

Change the application header to include a new link that to take you to the **My Details** page.

Procedure

1. Copy the Universal Access header by copying the `node_modules/@spm/universal-access-ui/src/features/ApplicationHeader` folder to `src/features`.
2. Fix any broken imports. Take the following steps:
 - a) Use ESLint or a similar linting tool to find any errors where imports are not found.

Note: If you do not use a linting tool, you get build errors.
 - b) Errors are generated because the `universal-access-ui` uses relative paths when it imports dependencies from its own project. For imports that are within the `universal-access-ui` module, but outside the `features/ApplicationHeader` folder, you must change the imports to reference the official exported version of those dependencies from the `universal-access-ui` node module.
 - c) For each import that is not resolved, find the equivalent export in the `universal-access-ui` package. Inspect `node_modules/@spm/universal-access-ui/src/index.js` to find the list of exported artifacts and their exported names.

The Paths module is referenced in the ApplicationHeader by using the default import from a relative path as shown in the following example: `import PATHS from '.././router/Paths'` Amend module as shown in the following example: `import { Paths } from 'universal-access-ui'`

- d) Repeat this procedure for all the files in the ApplicationHeader folder, some of the imports of 'Paths', and for some other references such as 'ErrorBoundary' and 'AppSpinner'.
3. Replace the existing header with your custom version, take the following steps:
 - a) Open `src/App.js` file and remove the imported ApplicationHeader from `universal-access-ui`.
 - b) Import your cloned version from `./features/ApplicationHeader` as shown in the following example:
`import ApplicationHeader from './features/ApplicationHeader';`
Import ApplicationHeader as a default import, without curly brackets, rather than a named import. Alternatively, you can add a named export to your ApplicationHeader feature.
4. Update the header feature to include a tab that loads the /person page take the following steps:
 - a) Open `constants.js` in `src/features/ApplicationHeader/components`. `constants.js` defines an object that represents a navigation item for the header.
 - b) Add an entry for the new page **My Details** as shown in the following example:

```
/**
 * Application navigation header tabs.
 */
const NAVIGATION_HEADER_TABS = {
  ...
  PROFILE: { NAME: 'PROFILE', ID: 'navigation-profile' },
  CHANGE_PASSWORD: { NAME: 'CHANGE_PASSWORD', ID: 'navigation-change-password' },
  MYDETAILS: { NAME: 'MYDETAILS', ID: 'my-details' },
};
```

- c) Open `ApplicationHeaderLogic.js`. `ApplicationHeaderLogic.js` contains the logic that tracks which tabs are selected so they can be highlighted as active.
- d) Update the `isTabActiveForUrlPathname` function to include the new **My Details** page in the **Your Account** section. For brevity, the value is hardcoded in the following example. However, you can replicate the pattern that is used by the `universal-access` code to add it to Paths.

```
const isTabActiveForUrlPathname = (urlPathname, navigationTabName) => {
  ...
  switch (navigationTabName) {
    case FIND_HELP.NAME:
      return (
        urlPathname === Paths.HOME ||
        urlPathname === Paths.APPLY ||
        urlPathname === Paths.BENEFIT_SELECTION ||
        urlPathname === Paths.APPLICATION_OVERVIEW
      );
    case YOUR_ACCOUNT.NAME:
      return (
        urlPathname === Paths.ACCOUNT ||
        urlPathname === Paths.BENEFITS ||
        urlPathname === Paths.PAYMENTS.ROOT ||
        urlPathname === Paths.PAYMENTS.DETAILS ||
        urlPathname === '/person'
      );
  }
};
```

Open `ApplicationHeaderComponent.js`, which renders the header, and find the `PrimaryNavigation` component.

- e) Add a tab called **'My Details'** with a link to the person feature inside `ApplicationHeaderComponent.js`. For brevity, the example is hardcoded values, but you can replace these values with variables. If you want, you can also globalize the tab.

```
..
<PrimaryNavigation>
  <Tabs>
    ...
    <Tab
      id={NAVIGATION_HEADER_TABS.YOUR_BENEFITS.ID}
      href={HASH_SYMBOL + LOCATIONS.BENEFITS}
      label={formatMessage(translations.headerYourBenefitsLabel)}
    />
    <Tab
      id="person_tab"
      href="/person"
      label="My Details"
    />
  </Tabs>
  ...
</PrimaryNavigation>
...
```

5. Save your file and restart the application.
6. You can modify the application footer in the same way by replacing the `universal-access-ui` version in `src/App.js` with your own custom version.

Results

Go to the home page. A new tab that is called **My Details** is in the primary navigation area. When you select **My Details**, the person feature is loaded in the main content area.

Related reference

[Developing with headers and footers](#)

IBM Cúram Universal Access contains a predefined header and footer. The header and footer contain content that is found in the header and footer of an application, such as links, **log in**, and **sign up** buttons, and menus for logged in users.

Creating an IBM Cúram Social Program Management REST API

Build on the scenario from *Changing the application header or footer*, use a REST API to get data to your application.

About this task

The most common way to get data to your application is to use a REST API to receive the requested data as a JSON string that your application then parses and renders. IBM Cúram Social Program Management provides development tools and the runtime infrastructure that you can use to build and deploy a REST API with your IBM Cúram Social Program Management server. The REST API can be called by using standard HTTP verbs such as GET, POST, and DELETE. The REST API returns data as a JSON string in the response body. For more information, see [Developing Cúram REST APIs](#).

Related information

[Developing Cúram REST APIs](#)

Connecting to REST APIs from the application

Build on the IBM Cúram Social Program Management REST API that you created in the scenario *Creating an IBM Cúram Social Program Management REST API* by calling it from your application.

About this task

Features in your application rely on passing data to and from the IBM Cúram Social Program Management server or another service. The reference application already consumes a number of Universal Access APIs to support business features.

This scenario updates the person feature to read the data from an API instead of just displaying hardcoded values. The scenario shows you how to create and use the following items:

- Use the RESTService utility to help you call APIs.
- Use the mock server to show you how to create a mock API so you can quickly develop your feature without spending time building and deploying the real API that it eventually uses.
- Connect your application to a IBM Cúram Social Program Management development environment that hosts the APIs by using Tomcat to enable real integration testing in the development environment.

Procedure

1. Create a mock API by completing the following steps:

- a) In your project, open `/mock/apis/mockAPIs.js`.

The mock server consumes `mockAPIs.js`, it contains the mappings from APIs to the mock data. The mock server uses this information to provide the correct data when an API call is made in development mode. `mockAPIs.js` also contains an import from the `universal-access-ui` package and assignments for GET, POST, and DELETE APIs as shown in the following example:

```
const mockAPIs = require('@spm/universal-access-mocks');

// Extract the existing universal access GET,POST and DELETE mocks for merging.
const UAMockAPIsGET = mockAPIs.GET;
const UAMockAPIsPOST = mockAPIs.POST;
const UAMockAPIsDELETE = mockAPIs.DELETE;
```

Use these APIs to test the Universal Access application. For more information, see [“The mock server API service”](#) on page 91.

- b) To add more mock data, add your mocks to the placeholders provided. This scenario adds the person data for a person 'James Smith' that is returned when the `/person` path is loaded.
- c) Add an object in `mockAPIs.js` to represent James Smith. For simplicity, do not normalize the dates, or use code tables, later scenarios show you how to globalize and handle code tables.

```
const user = {
  firstname: 'James',
  surname: 'Smith',
  dob: 'April 1st 1996',
  gender: 'male',
  address: {
    addr1: '1074, Park Terrace',
    addr2: 'Fairfield',
    addr3: 'Midway',
    addr4: 'Utah 12345',
  }
}
```

- d) Include a value for the URI `/user` in the `mockAPIsGET` object to return the mock object as shown in the following example:

```
const mockAPIsGET = {
  '/user': user,
}
```


The new `/user` mock API is merged with the mocks from `universal-access-ui` and is deployed by the mock server on port 3080.

- e) Test that the new API is working, start the application by using `npm start`.
 - f) Using the browser, load the `/person` URL: <http://localhost:3080/person>. If successful, the browser displays the response.
2. Use the `RESTService` utility from the core package to make an Ajax call to the API.

You can use many agents to make Ajax calls. The `RESTService` utility uses `Superagent`. The `RESTService` utility handles the following functions:

- Authentication credentials are automatically handled for each call, and users are redirect to log in when appropriate.
- The user's locale is passed to ensure that the response is in the correct locale.
- Timeouts are managed with environment variables in the `.env` file.
- Errors are captured and thrown in a standard fashion so that the error handling infrastructure is invoked.

For more information about the `RESTService` utility, see [“The RESTService utility”](#) on page 93.

3. Open `PersonComponent.js` file. Make the following changes, checking that your application still displays the page after each step:

- a) To enable lifecycle methods that are required to manage the API calls, convert the old stateless component to a stateful `React.Component` class:

Old stateless Person component

```
const Person = () => {  
  return (  
    <JSX code here>  
  );  
}
```

Updated stateful Person component

```
class Person extends Component {  
  render(){  
    return (  
      <JSX code here>  
    );  
  }  
}
```

- b) Create local state to hold the API data.

The local state stores the values returned by the API that drive the render function. Whenever the state is updated, the component re-renders to reflect the state change. For this scenario, hardcode the values for the state in your class constructor so that something is displayed on the page. To differentiate between this temporary default data and the API data, change the `firstName` to `'Roger'`. Later, when you introduce the API, the data for `'James'` is returned from the API and not the default state as shown in the following example:

```
constructor(props) {  
  super(props);  
  this.state= {  
    user : {  
      firstName: 'Roger',  
      surname: 'Smith',  
      dob: 'April 1st 1996',  
      gender: 'Male',  
      address: {  
        addr1: '1074, Park Terrace',  
        addr2: 'Fairfield',  
        addr3: 'Midway',  
        addr4: 'Utah 12345',  
      }  
    }  
  }  
}
```

```

    }
  }
}

```

- c) Convert all hardcoded references to use the values from the state.

Now that you have a state object, replace all hardcoded values with references to the state. Replace each hardcoded piece of data with a state reference `{this.state.user.X}`. Examples are as follows:

```

...
class Person extends Component {
  render(){
    return (
      ...
      <Card>
        <MediaObject media={avatarMedia} title={this.state.user.firstName}>
          <List>
            <ListItem>Gender: {this.state.user.gender}</ListItem>
            <ListItem>{this.state.user.gender}</ListItem>
          </List>
        </MediaObject>
      </Card>
    );
  }
}
...

```

- d) Import the RESTService utility.

To call an API, you must invoke one of the methods of the RESTService utility. First you must import it from the core package: `import { RESTService } from '@spm/core'`

- e) Create a componentDidMount method to invoke the API call.

When your component is mounted by React, the `componentDidMount` function is invoked. In `componentDidMount` the API call can be made to populate the component state. Update your constructor to set the user values to blank when initializing, this setting ensure that your data is being loaded from the API. Then, add the following code to your Person component. The root location of the API is taken from the values set in your `.env.development` file when in development mode. In production mode, it is taken from the `.env` file.

The `.env.development` file specifies the mock server URL as `REACT_APP_REST_URL`, which has the value `http://localhost:3080/` where the mock server is deployed. You can use this environment variable to prepend the `/user` API.

The RESTService API accepts a URL and a callback function as parameters. In the following code, the callback function is passed as an anonymous function in the second parameter. The 'success' is checked, before the state is updated with the response.

Note: Error scenarios are not handled in this code. The [“Handling failures in the application” on page 122](#) scenario contains details about failure responses, 'Error Boundaries', and failure handling.

```

componentDidMount() {

  const url = `${process.env.REACT_APP_REST_URL}/v1/user;
  const user = RESTService.get(url, (success, response) => {
    if (success) {
      this.setState((user: response));
    }
  });
}

```

Results

Start your application, log in and select the **My Details** tab. The tab loads using data that is pulled from the `/user` API.

The `REACT_APP_REST_URL` environment variable that is defined in the `.env` and `.env.development` files determines where the API is served. In development mode, the API calls the mock server. In production mode, the API calls the SPM server that hosts the application REST APIs. You can seamlessly switch between development and production, assuming the contract remains the same between your mock and real APIs. That is, that the JSON structure matches in both.

Related reference

[Handling failures in the application](#)

Handle any failures you find when you performed integration testing in the *Developing with IBM Cúram Social Program Management APIs by using Tomcat* scenario.

Testing REST API connections with Tomcat

Build on the scenario in *Calling an API from the application*. Do your integration testing with the real IBM Cúram Social Program Management APIs instead of the mock APIs in your Universal Access client.

Before you begin

You must be familiar with the IBM Cúram Social Program Management development environment, the development of REST APIs, and the IBM Cúram Universal Access development environment.

This scenario uses IP address 192.1.1.1 to represent the development computer for the IBM Cúram Social Program Management server, and 192.9.9.9 for the computer that hosts the Universal Access client. However, you can use the same computer with the same IP address. Replace this address with the IP address of your development computer.

About this task

The mock server is hosted on the same domain as the application during development `http://localhost`. However, when your APIs are served from a different domain, you might encounter Cross Origin Resource Sharing (CORS) issues. You can use Tomcat to configure your Universal Access client and IBM Cúram Universal Access server to allow Cross Origin requests. To overcome the CORS issues, the REST toolkit uses a filter that provides the required HTTP headers to allow browsers to accept responses from a different domain. In this scenario, the domain is where the REST application is deployed.

Procedure

1. Configure the IBM Cúram Social Program Management server, take the following steps:
 - a) In your development environment, add the following properties to `Bootstrap.properties` and set the hostname/ipaddress of the computer where the Universal Access client is to be deployed:
 - `curam.rest.refererDomains = 192.9.9.9`
 - `curam.rest.allowedOrigins = 192.9.9.9`

Note: If you develop the server and client on the same computer, you can use "localhost".

The property `curam.rest.allowedOrigins` is the `Origin` value in the CORS headers. Both properties can have comma-delimited domain names, for example, `curam.rest.allowedOrigins = 192.9.9.9, 192.9.9.8, mymachine.mycorp.com` to allow multiple domains to access the IBM Cúram Social Program Management application.
 - b) Set the `CATALINA_HOME` environment variable to the location of your Tomcat installation. For example, on Windows set the following variable: `'set CATALINA_HOME=C:\DevEnv\7.0.1\tomcat'`
 - c) Build IBM Cúram Social Program Management by using the appbuild server, database, client, and other components.

- d) Run an extra target **appbuild rest** to create the REST project in your `EJBServer\build\RestProject\devApp` directory.
- e) Copy `Rest.xml` into your Tomcat `conf/localhost` folder. For more information about building Cúram APIs, see *Developing Cúram REST APIs*.
- f) Start the server, RMILoginClient, and Tomcat in the normal way for IBM Cúram Social Program Management.

The REST client starts automatically. When the client is running, the APIs are accessible in the / Rest base path, for example: `http://192.1.1.1:9080/Rest/<myapi>`.

2. Configure the Universal Access client by completing the following steps:

- a) Modify the following environment variable in the `.env.development` file in the root of the application to point to the REST URL on Eclipse/Tomcat as shown in the following example:

```
REACT_APP_REST_URL=http://192.1.1.1:9080/Rest
```

Note: If you develop the server and client on the same computer, you can use "localhost".

If you want to connect to an application on WebSphere Application Server, you must change "http" to "https" and update to the correct port. 9044 is the default port.

- b) Build the application, enter the following command: **npm run build**.
- c) Start the application, enter the following command: **npm run start**.

Results

Your Universal Access client application now communicates with the REST API that is deployed on Eclipse with Tomcat.

Note: Run the application in debug mode so it stops at breakpoints in the application code.

Related information

[Developing Cúram REST APIs](#)

Handling failures in the application

Handle any failures you find when you performed integration testing in the *Developing with IBM Cúram Social Program Management APIs by using Tomcat* scenario.

Before you begin

You should build fault-tolerant web applications because, for example, web services such as a REST API are never fully reliable. When handling the expected response, the application must also allow for failures, such as network outages, timed out responses, internal server errors, or software bugs.

Universal Access ErrorBoundary component

According to React, "Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed."

An error boundary component is a React component that implements the `componentDidCatch` lifecycle method. For more information about error boundaries, see <https://reactjs.org/>

The `universal-access-ui` package exports a reusable `ErrorBoundary` component. The component has a default behavior to handle error scenarios by replacing the failing component with a generic message.

Note: Authentication errors have a specific handler in the `ErrorBoundary` component. If the error object that is received by the `componentDidCatch` method contains a status attribute with a value of '401' (Unauthorized error), then the client forces a log-out in the client application. Citizens are automatically redirected to the **Log in** page, so they can re validate and return to the page they were previously on. This

situation typically happens if the session times out or has been invalidated on the server. The source code for the `ErrorBoundary` component is available in the `universal-access-ui` package.

This scenario shows API error handling in the **My Details** page where the API call fails. This scenario also shows how to use the Universal Access `ErrorBoundary` component to provide a better user experience when failures occur.

Error boundaries in the Universal Access application

The Universal Access starter pack contains the following two error boundaries:

- The first wraps the entire application to capture errors that might occur when loading the header or footer.
- The second wraps the main content to capture errors that are raised from components that are loaded in the main content section.

The error boundaries are shown in the following example:

```
/**
 * App component entry point.
 */
const App = () => (
  <BrowserRouter>
    <ScrollToTop>
      <ErrorBoundary>
        <ApplicationHeader />
        <ErrorBoundary>
          <Main pushFooter className="wds-u-bg--page">
            {routes}
          </Main>
        </ErrorBoundary>
        <ApplicationFooter />
      </ErrorBoundary>
    </ScrollToTop>
  </BrowserRouter>
);
```

The error boundary on the main section allows the application context to be retained. That is, the header and footer continue to be displayed when the error is raised from the main section. This continuity provides a better user experience.

You can replace these error boundaries with your own error boundaries.

Faking an API error

This API failure scenario uses a 404 response as the error, you trigger this failure by temporarily changing the API call to a non-existent API.

Take the following steps:

1. Open `PersonComponent.js`
2. Update the API to call in the `componentDidMount` method to the non-existent `/user1` as shown in the following example:

```
componentDidMount() {
  const url = `${process.env.REACT_APP_REST_URL}/v1/user1;
  RESTService.get(url, (success, response) => {
    if (success) {
      this.setState({user: response});
    }
  });
}
```

3. Save your code and wait for the application to reload.

Provided you followed the previous scenarios, when the application reloads it displays the person and address cards but with no details. The values default to be the values that are created in the constructor of the `PersonComponent.js` file. Use the developer tools in your browser to verify the status of the

network call that is made for the '/user1' API. You should see that the response status is a 404 indicating that the network call failed.

Catching an API failure

Using the failure scenario *Faking an API error*, you can modify the code to cater for this failure. The API call is asynchronous, and the callback runs outside the context of the Component tree. This execution mode means that the error that is thrown in the call-back function is not caught by the `componentDidCatch` method of the `ErrorBoundary`. Therefore, instead of throwing an error in the callback, you update the state of the component. You can then use the lifecycle methods of the React component to react to the updated state when it arrives. Use a state attribute 'apiCallFailed' to hold the response.

In the `componentDidMount` method, add a branch to the callback passed to the `RestService.get` method. The failure branch sets the `apiCallFailed` value to the response value returned by the API as shown in the following example.

```
componentDidMount() {
  const url = `${process.env.REACT_APP_REST_URL}/v1/user1`;
  RestService.get(url, (success, response) => {
    if (success) {
      this.setState({user: response});
    } else {
      this.setState({apiCallFailed: response});
    }
  });
}
```

When the response is returned it updates the state, and triggers a rerender of the application. You can validate that the state was updated by printing the value in the console from the render method. An example response is as follows:

```
render() {
  console.log('state -> ${this.state.apiCallFailed}');
  return (
    ...
  );
}
```

The render method should print the following error in the console: `state -> Error: cannot GET http://localhost:3080/user1 (404)`

Throwing an error

Now that you have control of the failure, throw an error with an appropriate value for the `ErrorBoundary` component to catch. As indicated, the API call is asynchronous, so you cannot throw the error from the `componentDidMount`. The throw could be placed in the render function which will execute when the state updates, but this pollutes the rendering method with code that is not dedicated to rendering. Instead, use the `componentDidUpdate` lifecycle method. This method is called when the state is updated, which happens when the callback updates the 'apiCallFailed' value.

The error object thrown can be anything that you choose so that the error is as useful as possible to the citizen. In this instance, throw the string object that is returned by the response because it describes the issue.

Implementing a loading mask

Building on the previous scenarios, use a loading mask to indicate that the application is working on rendering a page.

About this task

Response times vary for REST APIs over a network. In many cases, the time it takes to receive the response is longer than the time it takes for React to render for the first time. This delay leads to a poor user experience when the page draws the components, but the data is missing.

To avoid poor user experience, use a loading mask to tell users that the application is working on rendering their page.

This scenario uses the `AppSpinner` component from the `universal-access-ui` package to include a loading mask for the **My Details** page to demonstrate how your components can handle slow response times.

API response delay

During development, you must often replicate real world response times for APIs. You can configure the `RestService` to set a delay by using the `env.development` file in your environment. By default this value is set to 2 seconds. Note this delay in the application when you are in development mode, where you see spinners while components wait for the data to be returned from the mock server by way of the `RestService` module. You can increase or decrease this value to meet your application's needs.

The `AppSpinner` component

The `universal-access-ui` package includes the `AppSpinner` component, which you can reuse in your project. The `AppSpinner` component wraps the `Spinner` component from the `govhhs-design-system-react` package and includes a label for accessibility purposes. You can also create your own loading mask in the same manner. You can view the source code for `AppSpinner` in the `universal-access-ui` package.

Procedure

1. Waiting for the API.

The `AppSpinner` is displayed while the application waits for the API to respond, so you need a mechanism to notify you when the data is, and is not loaded. Use the state to indicate when data is loaded and when it is not. Take the following steps:

- Open the `PersonComponent.js` file.
- In the constructor, add an attribute called `loading` to the state, with a value of `true`.

```
...
constructor(props) {
  super(props);
  this.state = {
    user: {
      firstName: "",
      surname: "",
      dob: "",
      gender: "",
      address: {
        addr1: "",
        addr2: "",
        addr3: "",
        addr4: ""
      }
    },
    loading: true,
  };
}
```

2. Display the loading mask.

Now you have a value that indicates whether the data is loading, take the following steps to display the loading mask based on the value:

- a) Import the AppSpinner loading mask from universal-access-ui:

```
import {AppSpinner} from '@spm/universal-access-ui';
```

- b) In the render function, add a check that renders the AppSpinner if the loading value is true:

```
render() {
  if (this.state.loading){
    return <AppSpinner/>
  }
  return (
    <Grid className="wds-u-p--medium">
      <Column width="1/2">

        ...

      </Column>
    </Grid>
  )
}
```

When you save and reload the application, you see the spinner in the main section area. However, the spinner continues to display after the data is returned.

3. Remove the loading mask.

When the data is returned from the API, remove the mask by updating the state to indicate that loading is finished. Take the following steps:

- a) In the componentDidMount function, update the state to set the loading value to false when a successful response is returned as shown in the following example:

```
componentDidMount() {
  const url = `${process.env.REACT_APP_REST_URL}/v1/user1;
  RESTService.get(url, (success, response) => {
    this.setState({loading: false});
    if (success) {
      this.setState({user: response});
    } else {
      this.setState({apiCallFailed: response});
    }
  });
}
```

- b) Save and reload the application. Now, when the API response is received, the loading mask is removed and the user's data is displayed.

Reusing existing features

The reference application that is available when you install IBM Cúram Universal Access satisfies a number of general business scenarios such as creating an account, logging in, and applying for benefits. The scenarios are provided both as working software and as examples of how to construct the product. You can clone and modify existing features in the application.

Before you begin

The universal-access-ui package is structured by feature. Typically, each feature is mapped to a single route. For example, when the /profile route is loaded, the Profile feature is displayed. The feature folder is a collection of files that work together to present that feature. An example from the Profile feature is shown.

```
/universal-access-ui
--/src
----/Feature
```



```

-----/Profile
-----/components
-----/ContactInformationComponent.js
-----/PersonalInformationComponent.js
-----/ProfileComponent.js
-----/ProfileComponentMessages.js
-----/index.js
-----/ProfileContainer.js

```

The feature uses a commonly used pattern to move the data retrieval and management into a container component, and the rendering logic into stateless presentation components. This pattern is widely documented and used extensively when you work with React and Redux. The pattern is not covered in detail here, but you can see how features are structured.

About this task

You can copy the entire code base for a feature into your custom project and replace the route that served that feature with your version. You can then modify the code base to create your own custom feature.

Note: After you reuse a feature, you now have full ownership of the custom feature. On upgrade of the `universal-access-ui` package, you do not receive any changes to the product version of the feature and must manually apply any updates that you need.

Note: Most features in the `universal-access-ui` package depend on the modules in the `universal-access` package for their data. On upgrade, you must validate that your feature was not affected by any changes to modules that the feature depends on. See [“Universal Access Redux modules” on page 79](#).

Procedure

1. Find the feature that you want to replace in the `universal-access-ui` package.
 - a) Inspect the URL end point that you want to change and note the path.
For example, the path to the `faqs` feature is `/myapp/faqs` so the path is `faqs`.
 - b) Open the `/node_modules/@spm/universal-access-ui/src/router/Path.js` file. Search for the path string literal, in this case `'/faqs'` is assigned to the `Paths.FAQS` variable.

```

const Paths = {
  HOME: '/',
  ...
  FAQS: '/faqs',
  ...
  SIGNUP: '/signup',
  ...
};
export default Paths;

```

- c) Open the `/node_modules/@spm/universal-access-ui/src/router/Routes.js` file. Search for `Paths.FAQS` to find the route that the variable is being used in. Use the component value of the route to find the associated feature.
For example, the `FAQ` route component is imported from `'../features/FAQ'`.

```

...
import FAQ from '../features/FAQ';
...
export default () => (
  <Switch>
    ...
    <Route component={FAQ} exact path={PATHS.FAQS} />
    ...
  </Switch>
);

```

2. Copy the entire feature folder into your custom application.

For example, copy the `/node_modules/@spm/universal-access-ui/src/features/FAQ` directory to `<myapp>/src/features/FAQ`.

3. Replace the route with your custom version.

a) In your project, open the `src/routes.js` file.

b) Add a route at any point before the `UARoutes` entry to ensure that your path supersedes the same path in `UARoutes`.

```
import React from 'react';
import { Switch, Route } from 'react-router-dom';
import { Routes as UARoutes } from '@spm/universal-access-ui';
import FAQ from './features/FAQ';

export default (
  <Switch>
    <Route component={FAQ} exact path='/faqs' />
    <UARoutes />
  </Switch>
);
```

4. You can now verify whether your custom version of the feature is being used. Make an obvious change to the feature and reload the application to see whether the change is picked up and displayed.

5. Change the code to customize the feature.

Implementing page view analytics

You can implement page view analytics in your application to collect citizen page views for analysis. Using the included page view JavaScript functions, you can start tracking page views by implementing a callback to send tracking data to a library of your choice for analysis. In this example, the data is sent to the Google global site tag (gtag.js) JavaScript tagging framework.

Before you begin

The `registerPageViewCallback` and `pageView` functions are available for you to implement tracking in your custom application.

registerPageViewCallback

This function takes a callback, which you must define, as an argument. You must call the `registerPageViewCallback` function before the application is rendered.

pageView

This function calls the registered page view callback where present. If the page view callback is not registered, it is not called.

For IEG pages, `pageView` passes an object with the following properties as a parameter to the callback:

- `pageType` ('IEG')
- `pageID` (the current IEG page ID)
- `scriptID` (the IEG script ID)

For non-IEG pages, `pageView` passes an object with the following properties as a parameter to the callback:

- `title`
- `location`
- `path`

About this task

To track page views, you must initialize the tracking library, register the callback, and implement the callback to send tracking data to a library for analysis.

When you define your own custom routes, you must use the `TitledRoute` component so that the pages can be tracked. If the route corresponds to an IEG script, you must set the `isIEG` property for the `TitledRoute` component.

Procedure

1. The `index.html` file is a good place to initialize the library. Insert this snippet, which is as provided by Google except for the tracking call.

```
<!-- Global site tag (gtag.js) - Google Analytics -->
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-TRACKINGID"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
</script>
```

2. Also in the `index.html` file, you must update the Content Security Policy to allow the Google script to run:

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-eval' 'unsafe-inline' https://www.googletagmanager.com/ http://www.google-analytics.com/" />
```

3. Implement the callback function.

The callback handles both IEG and non-IEG pages based on the `pageType` prop.

```
export default function customCallback(props) {
  const gtagProps = {};
  if (props.pageType && props.pageType === 'IEG') {
    // IEG pages
    gtagProps.page_title = `${props.scriptID} ${props.pageID}`;
    gtagProps.page_path = `/apply/${props.pageID}`;
  } else {
    // Non-IEG pages
    gtagProps.page_title = props.title;
    gtagProps.page_location = props.location;
    gtagProps.page_path = props.path;
  }
  window.gtag('config', 'UA-TRACKINGID', gtagProps);
}
```

4. In `index.js`, register the callback before the application renders.

```
registerPageViewCallback(customCallback);
ReactDOM.render(<App />, document.getElementById('root'));
```

5. For your own custom routes, you must use the `TitledRoute` component so that the pages can be tracked. If the route corresponds to an IEG script, you must set the `isIEG` property for the `TitledRoute` component. For more information, see [“Advanced routing” on page 90](#).

Implementing a test environment

Use the `test-framework` package to set up your IBM Universal Access Responsive Web Application test environment for testing with Test Cafe, Jest, and Enzyme. Then, use this guidance and the provided helper files to write end-to-end tests, unit tests, or snapshot tests for your project. You can configure the default test environment to suit your project requirements as needed.

End-to-end test environment

The `test-framework` package contains reusable files to help you set up a test environment with TestCafe, and to help you to develop end-to-end test scripts.

End-to-end test helper files

The end-to-end test helper files in `test-framework` are designed to operate best within a page object framework structure for your end-to-end automation suite.

Browser.js

The `Browser.js` module simulates interactions a user can have with their browser during an automated test, such as:

- Retrieving the current URL for the current page displayed in the remote browser.
- Clicking the browsers back button to navigate to the previous page.
- Clicking the browsers forward button to advance to the next page.

Page.js

The `Page.js` module simulates common interactions that a user can have with a web page in an application. A large variety of prebuilt methods are provided in this file, which help you to execute many user interactions, such as:

- Clearing text and typing new text into an input field.
- Clicking an element.
- Clicking an element only if it is displayed.
- Retrieving the value of an input field.
- Retrieving the text content of an element.
- Waiting for an element to be displayed.
- Plus many more as described in the JS documentation for this package.

In addition, the `Page` module contains two methods to help you with developing and debugging your end-to-end test scripts:

- The `wait` method pauses a test for a specified time (in milliseconds).
- The `debug` method physically stops the currently executing test script. You can then interact with the page that is displayed in the remote browser in its current state. You can resume the test script again at any time.

PageObject.js

The `PageObject.js` file acts as a base class from which you can build your own custom page objects for use with end-to-end tests for any application. This class provides a lot of built-in functionality to help you with your page object development tasks. For more information, see the JS documentation for this package and the `PageObject` class documentation.

Verify.js

The `Verify.js` module provides a number of assertion methods for verifying the results from your automated test scripts. This module allows you to execute verifications such as:

- Verifying whether an element is displayed in the UI.
- Verifying whether two values are equal (or not).
- Verifying whether a specified value is true or false.

End-to-end test initial setup and configuration

Create your directory structure and `index.js` file.

Project directory structure

Using the suggested directory structure for your end-to-end test framework helps you to get the best out of the `test-framework` package during test development. It also helps you to keep things clean and maintainable as your test framework scales in size.

```
├── tests
│   └── e2e
│       ├── config
│       ├── data
│       ├── page-objects
│       └── scripts
```

- The `config` directory contains a single `index.js` file that serves as the configuration file for all of the modules and page objects that are going to be used by your test scripts.
- The `data` directory contains any additional data that is used by the test scripts such as user data or routes data for your application.
- The `page-objects` directory is where you build the page objects that are required to test each individual page of your application.
- The `scripts` directory is where you place the test scripts to be ran by `testcafe`.

Initial config directory setup

The first step in building your end to end framework is to create an `index.js` file in the `config` directory as shown:

```
├── tests
│   └── e2e
│       └── config
│           └── index.js
```

This file is where you are import all of the modules from the `test-framework` package that you want to reuse in your test scripts. You also configure and export your page objects from this configuration file. This approach improves your framework's long-term maintainability as everything that is used by your test scripts is located in and exported from this single file. If something does change, the configuration file is all that needs to be updated and your scripts automatically inherit all of the changes without the need to refactor them.

Import the `test-framework` helper files and export them for use in your test scripts. Initially your `index.js` file contains the following code:

```
import { Browser, Page, Verify } from '@spm/test-framework';
export { Browser, Page, Verify };
```

If you set up your test directory structure as suggested, then importing each of these modules into your test scripts follows this pattern:

```
import { Browser, Page, Verify } from '../config';
```

Page object development and best practices

The page object model design pattern for building UI automation frameworks is our recommended practice. A page object is an object-oriented class that is built to represent the individual pages in the application under test. These representations offer an interface from which your test scripts can interact with any UI element that is associated with that page similarly to how a user would interact with them.

For example, the page object for the LoginPage in your application might include a `login()` method where you specify the user name and password credentials as parameters. This method then provides the automated steps that are required for logging a user in to your application. This page object can then be reused by any test script that requires a logged in user, with each test suite calling that `login()` method without needing to copy and paste the individual steps each time.

The benefits to the page object model extend far beyond simply reducing code duplication. Further benefits include:

- The API of your chosen automation framework is completely abstracted away from your test scripts. This makes tests easier to read, write and review.
- Element selectors are isolated in the page object that requires them.
- Since you are referencing page objects in your test scripts, the scenarios executed by the scripts document themselves as you write them. Managers and new team members alike will find these test scripts much clearer and easier to understand. For example: it is much easier to read and instantly know the meaning of `loginPage.goto()`; followed by `loginPage.login()`; as opposed to trying to make sense of a group of API calls.
- Suppose that an update completely changes the behavior for something that previously exists in one of your page objects. You need to update only the affected individual page object function to work with the new behavior and all of your test scripts automatically inherit the changes. You won't need to go back and change anything in any of your scripts.

Best practices

Best practices for the development of page objects in your automation framework.

Use CSS selectors to locate your UI elements

Use CSS selectors when trying to locate your UI elements. While you can use XPath for this purpose, CSS selectors are the highly recommended practice due to their sheer simplicity, not to mention the overall speed and performance advantages they have over their XPath equivalents. To get the best out of CSS selectors, assign some attribute to your UI elements to make them unique from all other elements. For example, set the `id`, `name`, or perhaps a custom `data-testid` attribute with some unique identifier for that element.

Keep assertions out of page objects

One of the golden rules for building end-to-end test scripts is that you should aim to include just one main assertion or set of assertions per test script. It is therefore equally important that you do not place assertions in any of the functions provided by your page objects. It can be very tempting to add assertions to a page object function because it always provides an assertion for you every time that method is invoked.

For example, suppose that a message is briefly displayed to the user to confirm that they have successfully logged in. You also have a scenario to automate that verifies that this message is displayed after a user has logged in. You might add the assertions for this as the final steps of the `login()` method in your LoginPage page object so that this verification is always made every time any page object invokes that `login()` function.

While it can look like a good idea to do this and also promotes the idea that you are getting something of a free verification for your login behavior in all of your other scripts, this is not a recommended practice because:

- First, you are losing a lot of clarity in your test scripts by adding verifications to your page object functions. Seeing `loginPage.login()` in your script does not clearly imply that this method also includes a verification therefore the intention of the test script will also be unclear as a result.
- Adding assertions to page objects adds too much ambiguity to your test suites. Your scripts will automatically inherit multiple assertions, any of which can fail, which may result in the conclusion of your scripts never being reached and their intended main verification(s) never taking place. Going back to our `login()` example, suppose a bug is introduced whereby the login message is not displayed to the user after successful login. Now all of your test scripts which invoke that `login()`

method will fail since you added the verifications to confirm the presence of the message even though only one test in your entire suite should realistically be verifying this.

- Developers that may have to debug a failing test will be forced to dig deep into your page object framework in order to find what verifications have actually taken place during the test execution. This will be even more complex a task if you are importing and reusing page objects that have been developed in a separate framework.
- Verifications aren't as free as you might think. In fact, they can be very expensive for time. Having multiple verifications taking place throughout your page object functions can slow your test script execution times down by a significant amount.

The `PageObject` class

The `PageObject.js` file in the `test-framework` package provides an interface from which you can easily create page objects for use in your end-to-end framework. When you create page objects, you can use `PageObject` constructor parameters to automatically generate methods that are commonly used by page objects during automation.

Import this class into your page object file directly and extend from it to inherit all of its behavior, for example:

```
import { PageObject } from '@spm/test-framework';

export default class MyPageObject extends PageObject {
  // ...
}
```

You can use the `PageObject` class to set a URL for the web page that is represented by your page object. It also has a number of parameters to automatically generate methods that are commonly used by page objects during automation. Alternatively, you can call the `super` method in the constructor to extend from this class without setting any of the parameters.

The `PageObject` constructor parameters

The `PageObject` class provides a number of constructor parameters that you can use to build your page objects. The sample code shows how to invoke the `PageObject` constructor and lists all of the parameters that are accepted:

```
export default class MyPageObject extends PageObject {
  /* Invokes the PageObject constructor - the following is the complete list of parameters
  supported in their correct order */
  constructor() {
    super(
      url,
      clickList,
      clickIfDisplayedList,
      clearAndTypeTextList,
      typeTextList,
      selectList,
      getValueList,
      getIsSelectedList,
      getDropdownSelectionList,
      gettextContentList,
      getIsReadOnlyList
    );
  }
}
```

@param {JSON} `clickList` parameter

The `clickList` parameter specifies a list of CSS selectors in JSON, all of which correspond to elements in the UI to be clicked during your test execution. For example, these two CSS selectors correspond to two different buttons in your UI:

```
const submitButton = 'input[id="submit"]';
const exitButton = 'button[id="exit"]';
```

Instead of declaring them as the individual variables as shown, declare them as the `clickList` parameter as follows:

```
const clickList= {
  exitButton: 'button[id="exit"]',
  submitButton: 'input[id="submit"]'
};

export default class MyPageObject extends PageObject {
  /* For this example we are only setting the URL and clickList parameters - all other
  parameters are left undefined */
  constructor() {
    super('http://www.ibm.com', clickList);
  }
}
```

By specifying your UI elements that are to be clicked in this way, you now have access to `click` methods for each of the selectors in the `clickList` after you create an instance of your page object. These `click` methods are automatically generated when your page object is created. So for the previous example, the following code sample demonstrates exactly what methods become available when you create an instance of the `MyPageObject` class:

```
/* First create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to both of these click methods
*/
await myPageObject.clickExitButton();
await myPageObject.clickSubmitButton();
```

The `click` method name is derived from the word `click` followed by the title of the key that you assigned to your selector. Therefore, if you declared a `myCustomSelector` key in the JSON that provided to the `clickList` parameter, the `click` method for that selector is `clickMyCustomSelector()`.

Note: As all of these method names are derived from keys, be careful with your spelling. Any spelling mistakes in keys are reproduced in the subsequent `click` method name.

@param {JSON} `clickIfDisplayedList` parameter

Specifying CSS selectors in the `clickIfDisplayedList` parameter automatically generates a method for each selector when the page object instance is created.

Each of the generated methods attempts to click the UI element corresponding to your specified selector only if that selector is displayed in the UI. If the UI element is not displayed, the method exits cleanly and allows your test script to continue running.

The naming convention for this method follows the format `click_XXX_IfDisplayed` where `_XXX_` is the title that you assigned to each of your keys.

The methods that are generated in this instance are as follows:

```
const clickIfDisplayedList= {
  exitButton: 'button[id="exit"]',
  submitButton: 'input[id="submit"]'
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL and clickIfDisplayedList parameters - all
  other parameters are left undefined */
  constructor() {
    super('http://www.ibm.com', undefined, clickIfDisplayedList);
  }
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to both of these
clickIfDisplayed methods */
```



```
await myPageObject.clickExitButtonIfDisplayed();
await myPageObject.clickSubmitButtonIfDisplayed();
```

@param {JSON} clearAndTypeTextList parameter and @param {JSON} typeTextList parameter

Both of these parameters automatically generate type methods. However, the functionality of the methods that are generated for each of the element selectors that are specified in either list is slightly different:

- If you add selectors to the `clearAndTypeTextList` parameter, then the methods clear all previous text that was entered into the corresponding UI element before you type new text into that element.
- Any selectors added to the `typeTextList` parameter generate methods that type text into the UI element. No previous text is cleared, so the text is appended to the existing text.

While the functionality varies depending on which list that you add your selectors to, the actual method names that are generated follow the very same naming convention. In both cases the method name follows the format `type_XXX`, where `_XXX` is the title that you assigned to each of your keys. This `type_XXX` method also accepts a `string` parameter where you can specify the exact text that you want to type into that element.

The methods that are generated in this instance are as follows:

```
const clearAndTypeTextList = {
  firstName: 'input[id="first_name"]',
  lastName: 'input[id="last_name"]'
};
const typeTextList = {
  addressLine1: 'input[id="address_line_1"]',
  addressLine2: 'input[id="address_line_2"]'
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL and both type text parameters - all other
  parameters are left undefined */
  constructor() {
    super(
      'http://www.ibm.com',
      undefined,
      undefined,
      clearAndTypeTextList,
      typeTextList
    );
  }
}

/* Now create an instance of your page object */
const myPageObject = new MyPageObject();

/* After creating the page object instance, you will have access to all of these type methods */
await myPageObject.typeFirstName('Michael');
await myPageObject.typeLastName('Myers');
await myPageObject.typeAddressLine1('Haddonfield');
await myPageObject.typeAddressLine2('Illinois');
```

@param {JSON} selectList parameter

You can use the `selectList` parameter to specify a list of element selectors that correspond to `<select>` elements in your UI. Any element selector that specified in this parameter has a method automatically generated for it when the page object instance is created. The naming convention for the generated methods follows the format `select_XXX`, where `_XXX` is the title that you assigned to each of your keys. This `select_XXX` method also accepts a `string` parameter where you can specify the exact option that is to be chosen from the list of options in that `<select>` element.

The following example shows the methods that are generated for element selectors that are specified in the `selectList` parameter:

```
const selectList = {
  company: 'select[id="company"]',
  county: 'select[id="county"]'
};
```

```

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL and the selectList parameter - all other
  parameters are left undefined */
  constructor() {
    super(
      'http://www.ibm.com',
      undefined,
      undefined,
      undefined,
      undefined,
      selectList
    );
  }
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to these select methods */
await myPageObject.selectCompany('IBM');
await myPageObject.selectCounty('Dublin');

```

@param {JSON} getValueList parameter

For verification purposes in your test scripts, you can retrieve the text value of an <input> field by adding element selectors to the `getValueList` parameter.

The naming convention for the methods follows the format `get_XXX_Value`, where `_XXX_` is the title that you assigned to each of your keys. When you invoke this method in your test script, it retrieves the current string value of the <input> element corresponding to the CSS selector you specified.

The following example shows how you might combine a `type_XXX` method action with a `get_XXX_Value` method action to enter text into an <input> field and then retrieve its value again:

```

const clearAndTypeTextList = {
  firstName: 'input[id="first_name"]',
  lastName: 'input[id="last_name"]'
};
/* We can re-use both of the existing selectors for the purpose of this list - there's no need
to declare them again */
const getValueList = {
  firstName: clearAndTypeTextList.firstName,
  lastName: clearAndTypeTextList.lastName
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL, the clearAndTypeTextList parameter and the
  getValueList parameter - all other parameters are left undefined */
  constructor() {
    super(
      'http://www.ibm.com',
      undefined,
      undefined,
      clearAndTypeTextList,
      undefined,
      undefined,
      undefined,
      getValueList
    );
  }
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to all of these methods */
await myPageObject.typeFirstName('Jack');
await myPageObject.typeLastName('Bauer');
const firstName = await myPageObject.getFirstNameValue();
const lastName = await myPageObject.getLastNameValue();

```

@param {JSON} getIsSelectedList parameter

During test execution, you can verify whether a specific checkbox or set of checkboxes were selected or cleared with the `getIsSelectedList` parameter. The naming convention for the generated methods

follows the format `is_XXX_Selected`, where `_XXX_` is the title that you assigned to each of your keys. When you invoke this method in your test script, it returns a Boolean `true` or `false` value that depends on whether the checkbox element corresponding to the CSS selector that you specified is checked or not.

The following example shows how you might combine a `click_XXX` method action with an `is_XXX_Selected` method action to select a checkbox and then determine whether it was checked:

```
const clickList= {
  agreeTermsAndConditions: 'input[type="checkbox"][id="terms_and_conditions"]'
};
/* We can re-use this existing selector for the purpose of this list - there's no need to
declare it again */
const isSelectedList= {
  agreeTermsAndConditions: clickList.agreeTermsAndConditions
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL, the clickList parameter and the
  isSelectedList parameter - all other parameters are left undefined */
  constructor() {
    super(
      'http://www.ibm.com',
      clickList,
      undefined,
      clearAndTypeTextList,
      undefined,
      undefined,
      undefined,
      isSelectedList
    );
  }
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* The first check for whether the checkbox is selected or not will return false */
let isChecked = awaitmyPageObject.isAgreeTermsAndConditionsSelected();

/* Now lets click on the checkbox and re-run our previous method - this time it will return
true */
await myPageObject.clickAgreeTermsAndConditions();
isChecked = await myPageObject.isAgreeTermsAndConditionsSelected();
```

@param {JSON} getDropdownSelectionList parameter

You can retrieve the selected option from a `<select>` element during test execution by adding selectors to the `getDropdownSelectionList` parameter. For example, if your test script already has a value for a `<select>` element in your UI and you want to verify whether the value is correct and retained after some other actions are executed.

The naming convention for the generated methods follows the format `get_XXX_Selection`, where `_XXX_` is the title that you assigned to each of your keys. When you invoke this method in your test script, it retrieves the `string` value of the currently selected option in the `<select>` element corresponding to the CSS selector you specified.

The following example shows how you might combine a `select_XXX` method action with an `get_XXX_Selection` method action to select an option in a `<select>` element and then retrieve the currently selected option from that `<select>` element again:

```
const selectList= {
  company: 'select[id="company"]',
  county: 'select[id="county"]'
};
/* We can re-use these existing selectors for the purpose of this list - there's no need to
declare them again */
const getDropdownSelectionList= {
  company: selectList.company,
  county: selectList.county
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL, the selectList parameter and the
```

```

getDropdownSelectionList parameter - all other parameters are left undefined */
constructor() {
    super(
        'http://www.ibm.com',
        undefined,
        undefined,
        undefined,
        undefined,
        selectList,
        undefined,
        undefined,
        getDropdownSelectionList
    );
}
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to these all of these methods */
await myPageObject.selectCompany('IBM');
await myPageObject.selectCounty('Dublin');
const companySelection = awaitmyPageObject.getCompanySelection();
const countySelection = awaitmyPageObject.getCountySelection();

```

@param {JSON} getTextContentList parameter

You can use the `getTextContentList` parameter to specify a list of selectors from which you want to retrieve text content.

The naming convention for the generated methods follows the format `get_XXX_TextContent` where `_XXX_` is the title that you assigned to each of your keys. When you invoke this method in your test script, it retrieves the string value of the text content for the UI element corresponding to the CSS selector that you specified.

The following example shows the methods that are generated for element selectors that are specified in the `getTextContentList` parameter:

```

const getTextContentList= {
    title: 'h1[id="main_heading"]'
};

class MyPageObject extends PageObject {
    /* For this example we are only setting the URL and the getTextContentList parameter - all
    other parameters are left undefined */
    constructor() {
        super(
            'http://www.ibm.com',
            undefined,
            undefined,
            undefined,
            undefined,
            undefined,
            undefined,
            undefined,
            undefined,
            undefined,
            getTextContentList
        );
    }
}

/* Now create an instance of your page object */
const myPageObject = newMyPageObject();

/* After creating the page object instance, you will have access to these methods */
const titleText = await myPageObject.getTitleTextContent();

```

@param {JSON} getIsReadOnlyList parameter

During test execution, you can verify whether a specific input field or set of input elements are marked as read only with the `getIsReadOnlyList` parameter.

The naming convention for the generated methods follows the format `is_XXX_ReadOnly`, where `_XXX_` is the title that you assigned to each of your keys. When you invoke this method in your test script, it returns a Boolean `true` or `false` value that depends on whether the input element corresponding to the CSS selector you specified is a read-only input field or not.

The following example shows the methods that are generated for the element selectors that are specified in the `getIsReadOnlyList` parameter:

```
const getIsReadOnlyList = {
  firstName: input[id="first-name"]
};

class MyPageObject extends PageObject {
  /* For this example we are only setting the URL and the getIsReadOnlyList parameter - all
  other parameters are left undefined */
  constructor() {
    super(
      'http://www.ibm.com',
      undefined,
      undefined,
      undefined,
      undefined,
      undefined,
      undefined,
      undefined,
      undefined,
      undefined,
      getIsReadOnlyList
    );
  }
}

/* Now create an instance of your page object */
const myPageObject = new MyPageObject();

/* After creating the page object instance, you will have access to these methods */
const isReadOnly = await myPageObject.isFirstNameReadOnly();
```

Adding custom behavior to your page objects

You can add custom behavior to your page objects. For example, a specific click action, or a specific series of instructions to run for an automated task in your end-to-end test scripts.

As a further example, a web page might render some dynamic content and you need to wait for a specific element to be visible in the UI before you continue.

The `test-framework` package provides a `PageObject` class from which you can take advantage of the automatically generated methods that are provided. You can add your own custom behavior to your page objects too.

Sample page object with custom behavior

In this example, you add a simple `waitForPageLoad()` method to your page object. It is assumed that your application is rendering some dynamic content, such as a timeline, and that a `See More` button is rendered at the foot of the dynamic content.

```
import { Page, PageObject } from '@spm/test-framework';

const url = 'http://www.ibm.com';

/* Now let's define some other selectors that we are going to use to define our custom behaviour */
const seeMoreButton = 'input[type="button"][id="see_more"]';

export default class MyPageObject extends PageObject {
  constructor() {
    /* For this example we will only define the URL - we don't need to define the other lists */
    super(url);
  }

  /* Now let's add our custom behaviour to our page object */
  async waitForPageLoad() {
    await this.waitForElementToBeDisplayed(seeMoreButton);
  }
}
```

```

/* Now create an instance of your page object */constmyPageObject=newMyPageObject();

/* Lets navigate to the URL defined in our page object and then wait for the page to load */
awaitmyPageObject.goto();
awaitmyPageObject.waitForPageLoad();

```

Building, exporting and configuring your page objects

Build, export, and configure your page objects so you can import and use them in your end-to-end test scripts.

Building your page objects

It is best to build each of your page objects by extending from the `PageObject` class in the `test-framework` package. Then, save each of your page object files in the `page-objects` folder in your test framework directory structure. The naming convention for page objects is to use the title of the application web page that the page object represents, for example `HomePage.js` or `LoginPage.js`.

```

.
├── tests
│   └── e2e
│       └── page-objects
│           ├── HomePage.js
│           └── LoginPage.js

```

Exporting your page objects from `page-objects/index.js`

After you create your page objects, you must export them from the `page-objects` directory to import them into your test scripts. Create an `index.js` file in the `page-objects` folder to enable all of your page object files to be exported from this single location. As you scale your page object framework, you can have many page objects to export from this folder.

```

.
├── tests
│   └── e2e
│       └── page-objects
│           ├── HomePage.js
│           ├── LoginPage.js
│           └── index.js

```

With the `index.js` file in place, export your page objects by using this file as shown in the example:

```

export { defaultasHomePage } from'./HomePage';
export { defaultasLoginPage } from'./LoginPage';

```

Configuring your page objects

You can now import page objects into your project's `config/index.js` file for reuse with your end-to-end test scripts. Before you continue, ensure that your test directory structure looks like this structure:

```

.
├── tests
│   └── e2e
│       ├── config
│       │   └── index.js
│       └── page-objects
│           ├── HomePage.js
│           ├── LoginPage.js
│           └── index.js

```

The following sample code shows your `config/index.js` file after you add your page object configuration to the file. In the sample code, you are importing each of your custom page objects from your `page-objects` folder, instantiating each page object and then exporting each instantiated page object from the file:

```

import { Browser, Page, Verify } from'@spm/test-framework';
import {

```

```

    HomePage,
    LoginPage// ... also import any other page objects that you require ...
} from'../page-objects';

/* Instantiate all of the page objects to be used during the e2e tests */
consthomePage=newHomePage();
constloginPage=newLoginPage();
// ... also instantiate any other page objects that you imported ...export {
    Browser,
    Page,
    Verify,
    homePage,
    loginPage// ... export all other instantiated page objects ...
};

```

With your page objects configured, you can now easily import and use your page objects in your end-to-end test scripts.

Writing end-to-end scripts

Now that your page objects are developed and your end-to-end framework is configured to use the test-framework package, you are ready to start writing test scripts that bring everything together. The code samples are developed with testcafe as the leading framework.

The sample code assumes that your framework directory structure is as shown.

```

|
| tests
| |
| | e2e
| | |
| | | config
| | | |
| | | | index.js
| | | |
| | | | page-objects
| | | | |
| | | | | HomePage.js
| | | | | LoginPage.js
| | | | | index.js
| | | |
| | | | scripts
| | | | |
| | | | | *.e2e.test.js

```

Scenario 1: Logging in redirects the user to the home page

You can write a test script for the following sample scenario based on the provided directory structure:

1. Open the application and go to the log-in page.
2. Enter the credentials of a valid user into the username and password fields and click **Log in**.
3. After you log in, verify that you were redirected to the user's account page.

Now to look at a test script for this scenario that incorporates your page objects and is driven by testcafe. Comments with each line of code further describe exactly what's happening at each step.

```

/* Firstly import all relevant page objects and test helper files by importing them from the
config/index.js file */
import { Browser, homePage, loginPage, Verify } from '../config';

fixture('Login e2e').page(loginPage.getUrl()); // Set the initial page to be opened as the
login page

test('Verify that the user is redirected to the home page on successful login', async () => {
    /* Log in as a valid user by re-using the page objects login method */
    await loginPage.login();

    // Re-use the Browser test helper file to get the current URL from the remote browser */
    const currentUrl = await Browser.getCurrentUrl();

    // Finally verify that the current URL in the remote browser matches the expected URL for the
    home page
    // It should be noted that every page object has a `getUrl()` method which allows you to
    easily retrieve the expected URL for the page it represents
    // Also note that this test is re-using the Verify test helper file to do its verifications
    await Verify.equal(
        currentUrl,
        homePage.getUrl(),
        'User was not redirected to the home page after successfully logging in'
    );
});

```

```
    );  
  });
```

Save this test into your `scripts` directory as `LoginPage.e2e.test.js`. Ensure that you save all other test scripts for your end-to-end framework in this directory.

Running end-to-end tests

It is straightforward to run your tests with `testcafe` by using a single `npm` script and a number of custom-set options.

For example, this `npm` script runs the specified test scripts by using `testcafe`. The tests run in Google Chrome with headless mode enabled and in incognito mode:

```
"testcafe": "testcafe \"chrome:headless -incognito\" tests/e2e/scripts/*.e2e.test.js",
```

To add this script to your project, copy and paste the `npm` script into the `package.json` file of the project that contains your end-to-end test framework. From the root of the project, run the script from the command line as follows:

```
npm run testcafe
```

You can watch your test suites run in headless mode from your command line.

You can disable headless mode by removing the `:headless` section of the script:

```
"testcafe": "testcafe \"chrome -incognito\" tests/e2e/scripts/*.e2e.test.js",
```

Now, when you run your test suites, you can see a physical remote browser open on the desktop of your local computer and you can watch the test execution as it happens.

For more information about the full list of supported browsers and all of the command line switches available for running scripts, see the [TestCafe documentation](#).

Jest and Enzyme test environment

The `test-framework` package contains reusable files to help you set up a test environment with Jest and Enzyme, and to help you to develop unit and snapshot test scripts.

Unit and snapshot test initial setup and configuration

Use the provided files to easily configure a default Jest and Enzyme test environment that you can use to start writing your unit and snapshot tests.

Project directory structure

By default, the Jest files expect a certain folder structure for your unit and snapshot test framework. Create the following folder structure in your environment.

```
.  
├── tests  
│   ├── config  
│   │   ├── setup-tests.js  
│   │   ├── snapshot.config.js  
│   │   ├── test-mapper.js  
│   │   └── unit.config.js  
│   ├── snapshots  
│   │   └── *.snap.test.js  
│   └── unit  
│       └── *.unit.test.js
```


Configuring the setup-tests.js file

Add the following code to the setup-tests.js file to configure Jest to work with enzyme-adapter-react-16 and to configure the snapshot serializer for use with the snapshot tests:

```
import Enzyme from 'enzyme';
import { createSerializer } from 'enzyme-to-json';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: newAdapter() });

/* Setup snapshot serializer */
expect.addSnapshotSerializer(createSerializer({ noKey:true, mode:'deep' }));
```

Mocking the Redux store

Some Jest tests mount components that access a Redux store by using the getState method. You can configure a mock store with the relevant Redux methods by adding this code to the setup-tests.js file.

```
global.mockStore = {
  getState: jest.fn(),
  dispatch: jest.fn(),
  subscribe: jest.fn()
};
```

You can then call the mock store from any component in a Jest test script with the following code:

```
const myComponent = IntlEnzymeTestHelper.mountWithIntlWithStore(
  <MyComponent />,
  global.mockStore
);
```

Mocking the Redux store with custom mock state

Some unit tests might need access to a mock Redux store with a specific mock state and with custom data.

- Add the mock state to the setup-tests.js file as follows:

```
const mockState = {
  // Add all of your mock data keys and values here
};
```

- Set the mock getState function to return the mock state when it is called during unit tests:

```
global.mockStore = {
  getState: jest.fn(() => mockState)
};
```

Configuring the test-mapper.js file

Jest cannot process data from CSS or image files and throws an error to the console if these files are referenced by any React component. Jest is designed to test the behavior of the component code and distances itself from any styling or images that are applied to that component.

To cleanly bypass any of these imports, add the following code to the test-mapper.js file.

```
module.exports= {};
```

Configuring the unit.config.js and snap.config.js files

These files are designed to configure the unit tests and snapshot tests for a project. You can use the default Jest configuration by adding the following content to both files:

```
// unit.config.js
const { getUnitTestConfig } = require('@spm/test-framework');
```

```
module.exports = getUnitTestConfig();
```

```
// snapshot.config.js
const { getSnapshotTestConfig } = require('@spm/test-framework');

module.exports = getSnapshotTestConfig();
```

Setting custom jest configurations

You can customize the default Jest configuration.

For example, you can set more project-specific folders to be ignored by the Jest coverage collection statistics as follows:

```
const { getUnitTestConfig } = require('@spm/test-framework');

const unitTestConfig = getUnitTestConfig();
unitTestConfig.coveragePathIgnorePatterns.push('<rootDir>/path/to/my/folder1');
unitTestConfig.coveragePathIgnorePatterns.push('<rootDir>/path/to/my/folder2');

module.exports = unitTestConfig;
```

Unit and snapshot test helper files

The test-framework provides the IntlEnzymeTestHelper.js and TestUtils.js helper files to help you to write unit and snapshot tests.

IntlEnzymeTestHelper.js

React components that use the react-intl module need access to the intl context, which is not available when you mount single components with Enzyme. You can use the IntlEnzymeTestHelper.js class to wrap a valid English-locale intl context around a component under test.

TestUtils.js

The TestUtils.js class is a utility class for testing React components with Redux modules.

To use the helper files in your Jest tests

Import any of the Jest helper files directly from the test-framework package as follows:

```
import { IntlEnzymeTestHelper, TestUtils } from '@spm/test-framework';
```

You can then call any of the class functions from your Jest test scripts as shown in the following examples:

```
describe('Test suite', () => {
  it('verifies something', () => {
    // ...
    const wrapper = IntlEnzymeTestHelper.mountWithIntl(
      <MyComponentUnderTest />
    );
    // ...
  });

  it('verifies something else', () => {
    const mockData = {
      // mock JSON data
    };
    const mockUtilFunction = TestUtils.mockActionsCallbackFxn([true, mockData]);
    // ...
  });
});
```

Guidelines for writing unit test scripts

The following guidance might be useful when you write Jest tests for both unit and snapshot testing.

To unit test or to snapshot test?

The first question that you must answer is whether to write a unit test or snapshot test.

Unit tests

Unit tests act as documentation for the project code or React component that you are testing. They include individual verifications for every piece of behavior in the code. Anyone must be able to read the verifications in the unit test suite and fully understand which behavior is being triggered and under which circumstances. Unit tests must be clear and concise and are a perfect indicator of code coverage within the overall project. These tests are the primary form of testing for the project code so you must write unit tests for all code in the project.

Snapshot tests

Snapshot tests can verify only that the DOM output for a React component in the provided state is correct. Don't use snapshot tests to test React component functionality, but use them as a complement to your unit tests to verify that the DOM output for a React component is correct. After unit tests verify the code behavior, snapshot tests can verify that everything is correct from an HTML markup perspective when the component is output to the DOM.

Snapshot test verifications are far too vague to offer any form of clear documentation for a component that is being tested. Because of the vague nature of their verifications, snapshot tests are also a poor indicator of code coverage so don't use them to collect code coverage statistics. It is much more beneficial for the project to collect code coverage statistics solely for behavioral based verifications, such as unit tests.

Collecting code coverage statistics for snapshot tests can provide a number of false positives. Code coverage might increase due to the presence of snapshot tests. However, the functionality of the code is not tested and verified as correct. You might read a high code coverage percentage in the coverage report and incorrectly assume that all of the component behavior is tested.

It can be beneficial to write both unit test and snapshot test suites for a project. However, unit tests must always be your priority given that they directly test the functionality of all of the code. A project can manage without snapshot tests. However, it can never survive without a thorough suite of unit tests.

Decide what must be tested

For each a new function or React component, you must decide what to test. Read through the code for the function or component and highlight the key behaviors and when they occur.

Create the unit test suite to test all of the identified functionality. After all of the behaviors are captured and tested in the unit test suite, then you can write snapshot tests to capture the DOM output for any new React components.

Your goal is to test all of the available functionality and cover 100% of the code. If there is code that is unreachable for any reason, then that code must be highlighted by the unit tests and refactored.

Ensure that all tests can be ran independently

All tests must be able to run independently of one another. A test that depends on the completion of another test is difficult to maintain and can be a direct cause of many avoidable consistency and reliability problems with your test suites.

- If the first test fails, the dependency can trigger a false negative by causing a dependent test to fail. Jest tests run concurrently by default so avoid creating tests that depend on each other.
- If a test is finishing work that started in another test, the dependency can significantly reduce the clarity of what each test is doing.
- If a test fails, the dependency significantly hinders debugging. You need to be able to isolate failing tests so you can rerun the failing test only. Test preconditions must be automatically included when you run the failed test independently on your local computer. If a failing test depends on another test, you

must find and run the other test before you can run the failing test. If several tests are chained in a sequence, you must find and run all preceding tests.

If you need reusable piece of test code for use in multiple test scripts, put the code inside one of the Jest test hooks, such as `beforeAll`, `beforeEach`, `afterAll`, or `afterEach`.

Use clear test descriptions

Each unit test in a project verifies some behavior of the code. Therefore, the description of the unit test must clearly indicate exactly what is being tested and under what circumstances.

There are essentially two ways to declare a unit test description:

- You can use a behavior-driven development (BDD) style description. For example:

```
it('given MyComponent, when the submit button is clicked, then the dialog is rendered', () => {
  // ...
});
```

- You can use a plain English sentence beginning with `verifies` `that...` to state exactly what is being verified. For example:

```
it('verifies that the dialog is rendered when the submit button is clicked', () => {
  // ...
});
```

Minimize the number of assertions for each test

Ideally each test script has one main assertion or `expect` statement that verifies the behavior that is being tested. It can be tempting to place multiple `expect` statements into a single test script, but avoid this practice. If any of the preceding `expect` statements in the script fail, then none of the subsequent `expect` statements will run.

If multiple `expect` statements are required, you can create a test suite that triggers the behavior to be tested in a `beforeAll` or `beforeEach` test hook. You can then write multiple test scripts that capture and verify each expected behavior individually. For example:

```
describe('MyComponent onClick() method behaviour', () => {
  let myComponent;
  let onClick;

  beforeAll(() => {
    onClick = jest.fn();
    myComponent = mount(<MyComponent onClick={onClick} />);
    /* Click the submit button to fire the onClick behaviour */
    myComponent.find('button').prop('onClick')();
  });

  it('verifies that the wds-u-hidden class of the dialog has been removed', () => {
    expect(myComponent.find('Dialog').hasClass('wds-u-hidden')).toBeFalsy();
  });

  it('verifies that the onClick functionality was invoked', () => {
    expect(onClick.mock.calls).toHaveLength(1);
  });
});
```

Running Jest and Enzyme tests

If you are using the default Jest configuration, you can run the Jest and Enzyme tests by adding scripts to the `package.json` file.

Procedure

- Add the following scripts to the `package.json` file.

```
"test-snapshots": "jest --config ./tests/config/snapshot.config.js",
"test-snapshots-update": "npm run test-snapshots -- -u",
```

```
"test-unit": "jest --config ./tests/config/unit.config.js",  
"test-unit-coverage": "npm run test-unit -- --collectCoverage",
```

Security, authentication, and authorization

IBM Cúram Universal Access gives citizens access to their most sensitive personal data over the internet. Security must be a primary concern in the development of citizen account customizations. All projects that are built on Universal Access must focus on delivering security from beginning to end.

It is recommended that all projects take at least the following steps to ensure the security of the project delivery:

- Ensure that the project team are familiar with the principles of secure application development, and common vulnerabilities such as the [OWASP Top Ten](#).
- Develop and apply a [Threat Model](#).
- Employ security experts to test everything from requirements to the finished deployment.
- Plan for how the application is used in public spaces like libraries and kiosks.

Build secure web apps with the Social Program Management Design System

When you develop with the IBM Social Program Management Design System and npm, you must consider your security attack plane, and take the appropriate steps to prevent or reduce the threat to your runtime application from malicious actors. npm is an impressive resource of open source software, but you must understand the threats that exist with this software stack and act responsibly to mitigate risk.

IBM Cúram Universal Access goes through rigorous security threat monitoring during development and before release. However, after the software is released there is an ongoing threat to your business:

- From your customization of the software.
- From new vulnerabilities that are discovered after release.

npm, which is the backbone of the Social Program Management Design System software development lifecycle strategy, promotes the reuse of third-party packages. Each of your packages uses other packages, which in turn use other packages, and so on. This generates a large tree of dependencies on software packages, many of which you might have little or no knowledge of. It is important to understand the NPM threat vectors and take the appropriate steps.

Protect yourself during development

You can use a number of strategies to reduce the possibility that you might release vulnerable software.

Review Dependencies

If you introduce new third-party dependencies in your custom code, try to choose packages that are established, well-maintained, and used widely. This strategy increases the chances that vulnerabilities are discovered quickly and resolved immediately by the package owners.

Audit Packages

Monitor the results of the `npm audit` command that is run after each installation. To reduce possible vulnerabilities, fix issues in your direct dependencies as they arise to ensure that your development environment is using the latest versions of packages in your dependency tree. While not all packages that are used during development get deployed in production, it is still good practice to minimize warnings as much as possible.

Lint in development

Use linting tools in your development environment to highlight security issues in your custom code and resolve or mitigate any reported issues before your release the code. For more information about configuring linting, see [“Enforce code style with ESLint” on page 71](#).

Lint in the continuous integration pipeline

Introduce linting to your integration pipeline to ensure that failures for security rules block the integration of new code.

Review code

Include security as a critical aspect of a code review, and educate your developers on how to spot scenarios where security is a concern, and how to identify coding errors.

Consider your authentication strategy

The reference application provides basic sign-up and log-in pages to demonstrate how the IBM Universal Access Responsive Web Application can be integrated with a IBM Cúram Social Program Management deployment. These features are provided as references, but are not intended to be directly deployed into production.

You must consider your authentication strategy and ensure it meets your organization's requirements. For example, you might need to integrate with an Identity Provider in your authentication flow, or use a captcha for your sign-up or log-in pages.

Secure your REST APIs

Ensure that you secure your REST APIs before deployment.

If you are using file uploads, see [“Implementing file security and validation for the POST files API” on page 94.](#)

Protect your production environment

You must understand how your production code is created, and use **npm audit** to identify security issues.

How is your deployed code created?

npm install

When you run `npm install` for your development environment, it typically installs packages that:

- Are used at run time and need to be included in the build for deployment.
- Are only required for development purposes or to create the deployment.

By convention, runtime or development-only packages are separated by the `dependencies` and `devDependencies` lists in the `package.json` file. However, you can have the packages in either list or mixed any way that you want. After an installation runs successfully you have all the code that is required to go to the next stage, which is to build your deployment and run your app. Adding packages to the `devDependencies` list does not ensure that they are not in your deployed code.

npm run build

When you run `npm run build`, the `dependencies` list in the `package.json` file is ignored as it is only relevant to the development environment. Instead, the `src/index.js` file that represents the root of the App is used to initiate a process of including only the files that are required to create a working deployment. It is possible that code from packages that are listed in your `devDependencies` ends up in your deployment bundle, if code from those packages is called from code in your runtime application.

Use npm audit to identify security issues

When you run `npm audit`, all packages that are listed in `package-lock.json` are analyzed for vulnerabilities. The audit is not sophisticated enough to know what is part of the deployment. That is, the contents of the build folder, and the packages from which they were included. For this reason, an `npm audit` is an *indicative* security check that helps to identify potential vulnerabilities with the packages that you are using. However, it is not an accurate picture of how vulnerable your running application is because:

- Code from packages with known vulnerabilities might not be included in your deployed code. Either because it is only used for development, or because the bundling algorithm calculates that the code is not required and doesn't add it to the deployment bundle.
- Vulnerabilities might not be discovered yet. So the code that you successfully audited yesterday might not pass an audit check today. No code did not change, but a new vulnerability was discovered and reported to NPM <https://www.npmjs.com/advisories>.

Therefore, `npm audit` is a good smoke test for vulnerabilities, but needs further analysis before action is taken.

When to run `npm audit`?

`npm audit` runs on each installation. In a development environment, it is obvious to the team when vulnerabilities arise as each time a developer installs the app, they are notified of vulnerabilities.

For a deployed environment, you can run `npm audit` daily against your deployed code to highlight any new vulnerabilities that were discovered since you deployed your application. You might be tempted to run it against your development codebase, but this code is not the same as your deployed code. The packages that are in a deployment might have been removed from your development repository since you last deployed and therefore not show up in an audit on your development codebase.

To monitor the code that is running in your deployed environment, you must run `npm audit` against the packages that were used to build that deployment. You need the `package.json` and `package-lock.json` from the codebase from which the deployment was built. You can create a simple automated job to run `npm audit` against a folder that contains these files and to report any failures. For example, you can run `npm audit --audit-level=high` from a folder that contains these files:

```
/my-current-deployment-packages
+ package.json
+ package-lock.json
```

Consider penetration testing

Penetration testing is an activity that is carried out on running software to find security holes. Penetration testing can be done with automated tools, such as IBM Security AppScan. To add a further strength test to your development process, you can hire third-party services that specialize in the penetration testing of apps, which includes both automated and manual testing. Typically, penetration testing is carried out periodically to provide an external health check on your applications security.

How to address security vulnerabilities

If you find that you are using a vulnerable package, you must analyze the threat to decide how to mitigate the risk.

Sometimes, an `npm audit` scan might report packages that are used in development but are not in your deployed application bundle and therefore not in your runtime application. The urgency of fixing issues with these packages is reduced.

For vulnerabilities that you discover through `npm audit` that are **High** or **Critical**, you must address them as soon as possible. For most, the fix is already available or to be provided within hours or days of registration of the vulnerability. You must redeploy your production code from a repository that was updated to the patched version of the vulnerable package. Typically, `npm audit` advises you what you need to do.

In some cases, the fix might require upgrading a package to a major version, which requires a manual upgrade as it might be a breaking change for your code. Where the package was included through your own custom code, you can do this upgrade yourself. In other cases, the fix is outside your control. For example, where the vulnerable package is a dependency of a package that you depend on, you need the owner of that package to fix their code. If IBM owns the package, you can open a support case for the issue.

For more information about how deal with security audits, see this [npm article](#).

User account types

IBM Cúram Universal Access has different account types to support both anonymous and registered citizens. As citizens use Universal Access, their account type can change.

IBM Cúram Universal Access has the following user types:

Public citizen account

When citizens view the organization **Home** page, they are automatically logged in under the *publiccitizen* account. This account has access only to the home page and the pages that allow citizens to enter or reset passwords.

Anonymous account

When the user clicks a link to start screening or applying for benefits, they are logged out as *publiccitizen* and logged back as an *anonymous* account with a random username. A principle of Universal Access is that users cannot access the data of other users. If all intakes and screenings used the same *publiccitizen* user account, a citizen might see data that was entered by another citizen.

Registered accounts

Registered accounts are standard accounts that are created by citizens. Citizens can create accounts when they first use the application, or during processes like applying for benefits. These accounts are different from anonymous accounts in that they allow citizens to continue previously saved applications, restart applications that were previously unfinished, and review or withdraw previously submitted applications.

Linked accounts

Linked accounts are accounts that are linked to a Concern Role ID for a Person entity. Organizations must implement their own linking functions. Universal Access APIs that allow a username to be linked to a Concern Role ID are available to support linking.

Citizens with linked accounts have access to detailed information about their benefits and cases by using their citizen account. Citizens with a linked account can submit life events such as getting married or losing their job. They also have access to information about benefit payments. Because of the sensitivity of this information, customers must ensure that they have a robust process for creating linked user accounts.

The following scenarios are examples. The actual processes for linking are unique to each organization.

- A citizen creates a user account in Universal Access and submits an intake application:
 - An organization can implement an online mechanism to link a registered account. Citizens can be asked in the online application if they want to upgrade their account for access to more services. If they consent, their registered account is automatically linked.
 - Citizens are contacted by their caseworker who asks them if they want access to more services. The citizen agrees and presents themselves at their local office with personal identification, such as a passport. The caseworker can then link the registered account that they used to submit their intake application.
- A citizen requests a citizen account and is asked to present themselves at their local Social Welfare office with personal identification, such as drivers license. After they verify the citizen's identity, the caseworker uses custom functions to enter details for the new linked account.

In none of these scenarios does a caseworker have access to a citizen's password. The linking process can automatically upgrade their account, or can trigger a batch job that sends the password by letter to the citizen's home address.

User account authorization roles and groups

User account types are assigned different authorization roles, which limit the methods that can be called. Do not grant extra permissions to authorization roles, except for Linked Accounts, which use the LINKEDCITIZENROLE. If you add custom methods to the citizen account, extra permissions are required.

For more information about adding custom methods to the citizen account, see *Customizing the citizen account*.

If you use only a subset of IBM Cúram Universal Access functions, remove permission to call any unused methods from the database. For example, if the citizen account is not used, remove the unneeded LINKEDCITIZENROLE and other related authorization artifacts. Projects that don't use the citizen account must also consider the deployment implications. For more information, see *Customizing the citizen account*.

Configure authorization roles only for functions that are being used. Remove unused Security Identifiers (SIDs) removed from the database. For more information, see *Security and the Citizen Account*.

Proper use of authorization roles and groups ensures that no user can access functions for which they have no permission. However, it does not prevent users from using these functions to access data that belongs to other users, which is done with Data-based Security. Universal Access provides a Data-based Security framework for all customizations to use. For more information, see *Security and the Citizen Account*.

Related concepts

[Customizing the Citizen Account](#)

Users can use the Citizen Account to log in to a secure area where users can screen and apply for programs.

[Security and the Citizen Account](#)

Security must be a primary concern when you customize the citizen account customizations. All public-facing applications must be analyzed and tested before they are deployed. Users must contact IBM support to discuss unusual customizations that might have specific security issues.

Authentication

The universal-access package exports the Authentication module, which can be used to log in and out of the application and to inspect the details of the current user. The login service is passed a username and password, and optionally a callback function that is called when the authentication request is completed.

Authentication services

The Authentication API supports the following modes:

- JAAS Authentication (Default)
- Simple Authentication (Development mode)
- Single Sign-on (SSO) Authentication
- Custom authentication

JAAS authentication

By default, the login process uses the standard JAAS login module.

The JAAS login module is exposed through the Social Program Management Universal Access API at the `/j_security_check` endpoint, and authenticates the user against the SPM database of users. For more information about JAAS login in SPM, see [Authentication Architecture](#).

Simple authentication (Development Mode)

During development, you can use a simple authentication that does not require an SPM server. This simple authentication bypasses proper authentication (JAAS or SSO) and instead accepts the username

dev without any password. The login process runs and allows access to the 'user account' password protected pages.

This simple authentication is sufficient to do most client development work and avoids the need to configure your client application to communicate with an SPM server.

SSO authentication

The application supports single sign-on (SSO), which is a typical use case for many enterprises that serve multiple applications with a single username and password for their clients.

For more information about configuring your application to use SSO, see [“Configuring single sign-on” on page 154](#).

Automatically logging in to your SSO

The default `SSOVerifier` component wraps the whole application and checks for SSO status when the application is loaded for the first time. The `SSOVerifier` component verifies whether the user is already logged in with SSO and can be logged directly into the application. If the user is not yet authenticated, then they must authenticate as a public citizen so that we can access the system configuration.

The default implementation calls `Authenticate.login` and passes only a `preCheckResponse` callback that checks if the authentication method is SSO SP. If so, it calls `Authentication.performSPInitSSOAuthRequest` to get information about the SSO status.

If you need custom verification, you must create your own verifier component to replace the current `SSOVerifier` component. Then, add your custom verifier component to the call stack in your entry JavaScript file, for example `App.js`.

Custom authentication

You can implement your own custom authentication to suit your specific environment.

User account types

The Universal Access client supports three different user account types, Public, Generated, and Citizen. For more on user accounts and security, see [“User account types” on page 150](#). If you want to customize the log in and sign up process that is provided with the Universal Access starter pack, the `Authentication` module provides log in functions to support each of these three user account types.

```
Authentication.loginAsPublicCitizen
Authentication.loginWithGeneratedUser
Authentication.login
```

Tracking the logged in user

The Universal Access Responsive Web Application uses 'session storage' in the browser to store some basic details of the currently logged-in user after they are authenticated with the server. This session storage is typically used to inform the client application what views to present. For example, if no user is logged in, then the login and sign-up page buttons are displayed on the home page.

The `Authentication` module provides functions that query the current logged-in user and their account details, according to the session storage in the browser.

```
Authentication.getLoggedInUser
Authentication.getUserAccount
```

Logged in on the client or the server

Citizens can seem to be logged in on the client when they are not logged in on the server. This situation does not compromise the security of the application. The SPM server APIs use session tokens that are stored in cookies to determine whether the current user is authenticated. The cookies are transmitted with each API call, and only a valid token results in a successful response.

For example, if a user's session times out on the server, the next API request to the server results in a 401 unauthorized response, even if the user seems to be logged in to the client application. This behavior ensures that no matter what the client application says about the currently logged-in user, the server responds only to valid session tokens.

For more information, see [“Configuring user session timeout” on page 55](#).

Related concepts

The `RESTService` utility

The `@spm/core` package provides the `RESTService` utility, which you can use to connect your application to a REST API. You can fetch resources with alternatives such as `Fetch API`, `SuperAgent`, or `Axios`. However, the `RESTService` utility provides some useful functions for connecting to SPM REST APIs.

Customizing the authentication method

The default implementation for authentication is a Java™ Authentication and Authorization Service (JAAS) authentication method. If the JAAS authentication method does not suit, you can change to one of the other provided authentication methods, or implement your own custom authentication method.

About this task

The following authentication methods are available in the application. For more information, see [“Authentication” on page 151](#). To use any of the provided authentication methods, set the **REACT_APP_AUTH_METHOD** environmental variable in the appropriate `.env` file to one of the following options and set any related environmental variables. For example:

```
REACT_APP_AUTH_METHOD=SSOIDPAuthentication
```

- **JAASAuthentication**

(Default for production environments. That is, **npm start** with `.env.development`.)

No further environmental variables needed.

- **DevAuthentication**

(Default for development environments.)

Set the [“Simple authentication for development” on page 177](#) environmental variables.

- **SSOSPAAuthentication**

Service-provider (SP)-initiated SAML 2.0 web SSO.

- **SSOIDPAuthentication**

Identity provider (IdP)-initiated SAML 2.0 web SSO.

If you set SSO authentication, you must set the [“Single sign-on \(SSO\) authentication” on page 177](#) environmental variables.

For more information about environmental variables, see the [“React environment variable reference” on page 173](#).

If you want to use custom authentication, you must create a custom authentication method and register the new authentication method as follows:

Procedure

1. Create a custom authentication method, which consists of a normal class file that contains two static methods as follows:

- `static login = input => {}`

Where `input` is an object that contains one or more of these authentication properties: `username`, `password`, `callback`, `ssoPreCheck`, `ssoLogin`.

Implement the logic to authenticate the user in this method.

- `static logout = (callback, reportLogoutError) => {}`

Where:

- `callback` is a function that is called when logout completes.
- `reportLogoutError` can be used to define whether a message is shown.

Implement the logic to log out the user in this method.

2. Register your new authentication method in an entry point file such as `App.js` by using the `AuthenticationRegistry` component as shown in the following example:

- a) Import the `AuthenticationRegistry` component and the authentication class:

```
import { AuthenticationRegistry } from '@spm/core';
import CustomAuthentication from '<path_to_custom_method>';
```

- b) Use `AuthenticationRegistry.registerAuthenticationFunctions` to register the functions from the authentication class:

```
AuthenticationRegistry.registerAuthenticationFunctions(CustomAuthentication);
```

Authenticating with external security systems

By default, IBM Cúram Universal Access uses its own authentication system that is backed up by a database of registered users. However, Universal Access can also be configured to authenticate with external security systems.

As government agencies increasingly provide online services, there is a drive to ensure that citizens can be authenticated for any of these services by using a single set of credentials. This approach provides benefits for the government in streamlining the authentication process and also for the citizen because citizens do not have to remember user names and passwords.

This process, in turn, increases security for the following reasons:

- It makes it less likely that citizens write down their user names and passwords.
- It focuses security efforts on implementing best practice in a single enterprise security system.

You can deploy Universal Access in *Identity Only* mode for registered users so that creating accounts occurs externally and user accounts are authenticated externally. For more information, see *Identity Only Authentication*.

Related information

[Identity only authentication](#)

Configuring single sign-on

Single sign-on (SSO) authentication enables users to access multiple secure applications by authenticating once with a single user name and password. Federated single sign-on that uses SAML 2.0 browser profile, using either an IdP-initiated HTTP POST binding or an SP-initiated HTTP POST binding, can be implemented through the Universal Access Responsive Web Application.

If a user authenticates to an SSO system, they are no longer prompted for credentials when they access any of the other applications that are configured to work with the SSO system.

SSO systems usually maintain the user accounts on a lightweight directory application protocol (LDAP) server. If user accounts are stored in one location, it is easier for system administrators to safeguard the accounts. Also, it is easier for users to reset one account password for multiple applications.

The following information describes the scenario where IBM Cúram Social Program Management is deployed on WebSphere. However, a similar process applies if IBM Cúram Social Program Management is deployed on another supported application server, such as Oracle Weblogic.

Related information

[Oracle: Configuring SAML 2.0 Services](#)

SAML 2.0 single sign-on initiation and flow in Universal Access

For single sign-on, the SAML response, by HTTP POSTs, is interpreted and controlled by logic in Universal Access.

In all SAML web SSO profile flows, the binding defines the mechanism that is used to send information through assertions between the identity provider (IdP) and the service provider (SP). WebSphere supports HTTP POST binding for sending web SSO profiles. The browser sends an HTTP POST request, whose POST body contains a SAML response document. The SAML response document is an XML document that contains certain data about the user and the assertion, some of which is optional.

Browser-based single sign-on (SSO) through SAML v2.0 works well with many web applications where the SAML flow is controlled by HTTP redirects between the identity provider (IdP) and the service provider (SP). The user is guided seamlessly from login screens to SP landing pages by HTTP redirects and hidden forms that use the browser to POST received information to either the IdP or the SP.

In a single-page application, all the screens are contained within the application and dynamic content is expected to be passed only in JSON messages through XMLHttpRequests. Therefore, the rendering of HTML content for login pages and the automatic posting of hidden forms in HTML content is more difficult. If the SP processes the content in the same way, it would be necessary to leave the application and hand back control to either the user agent or the browser, in which case the application state would be lost.

IdP-initiated use case

The IdP can send an assertion request to the service provider ACS in one of two ways:

- The IdP sends a URL link in a response to a successful authentication request. The user must click on the URL link to post the SAML response to the service provider ACS.
- The IdP sends an auto-submit form to the browser that automatically posts the SAML response to the service provider ACS.

The ACS then validates the assertion, creates a JAAS subject, and redirects the user to the SP resource.

SP-initiated use case

When an unauthenticated user first accesses an application through an SP, the SP directs the user's browser to the IdP to authenticate. To be SAML specification compliant, the flow requires the generation of a SAML AuthnRequest from the SP to the IdP. The IdP receives the AuthnRequest, validates that the request has come from a registered SP, and then authenticates the user. After the user has been authenticated, the IdP directs the browser to the Assertion Consumer Service (ACS) application that is specified in the AuthnRequest that was received from the SP.

Assertions and the SAML Response document

To prove the authenticity of the information, the assertion in the SAML response is almost always digitally signed. To protect the confidentiality of parts of the assertion, the payload can be digitally encrypted. A typical SAML response contains information that can be sent only through a login by a POST parameter. After login, an alternative mechanism is typically used to maintain the logged-in security context. Most systems use some cookie-based, server-specific mechanism, such as a specific security cookie, or the server's cookie tied to the user's HTTP session.

IdP-initiated flow in Universal Access

When Universal Access is configured with an IdP initiated web SSO flow, any attempt to connect to a protected resource without first authenticating through IdP results in a 403 HTTP response from IBM

Cúram Social Program Management web API. Any authentication requests that are initiated through SP result in a 403 HTTP response, and the application redirects the user to the login page that is contained in Universal Access.

The following figure illustrates the IdP initiated flow that is supported by Universal Access in a default installation.

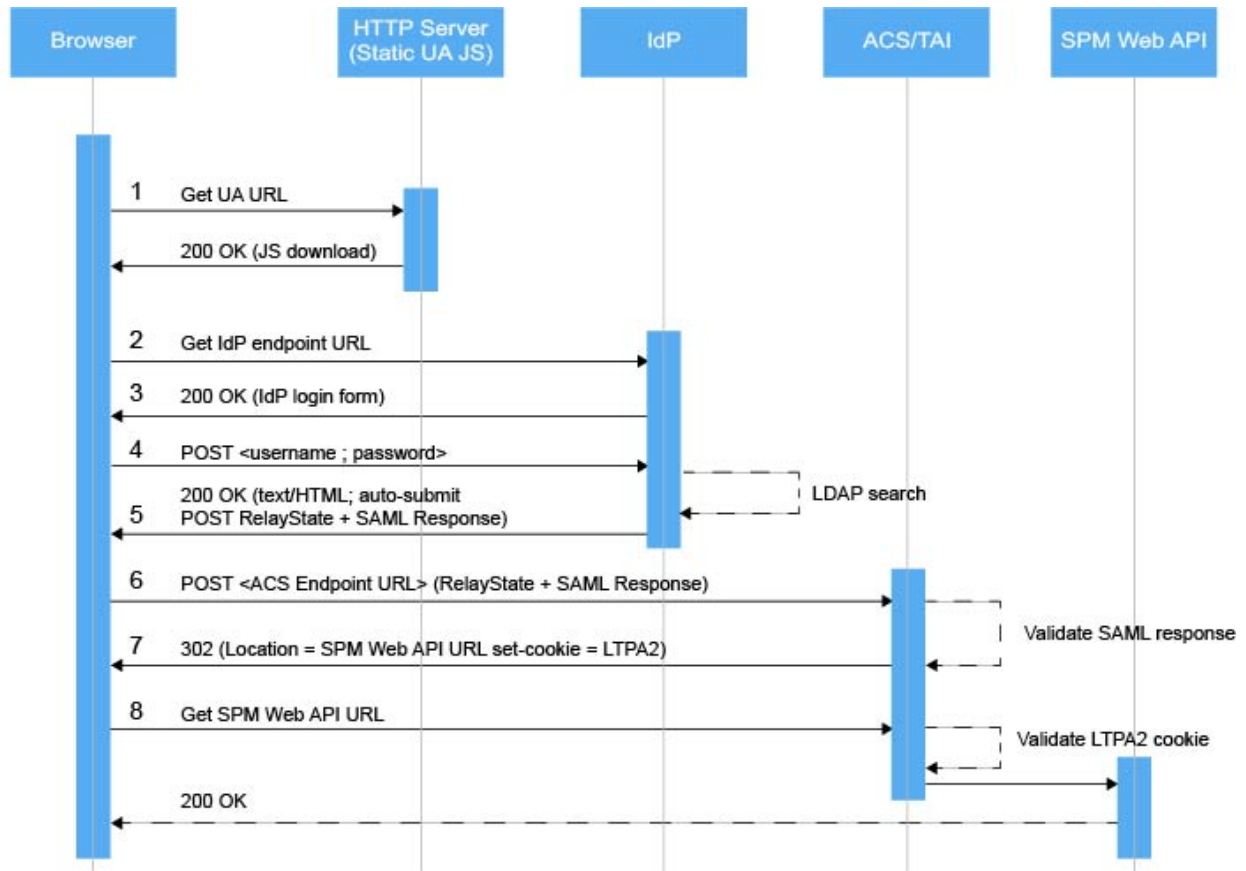


Figure 4. IdP-initiated flow in Universal Access

1. A user browses to the HTTP server that contains Universal Access.
2. The user can browse as normal by interacting with IBM Cúram Social Program Management as either a public or a generated user (which is not shown in the diagram). The user then opens the login page to access protected content, which triggers an initial request to the IdP endpoint.
3. In most IdP configurations, an HTML login form responds to the request. Universal Access ignores the response.
4. To authenticate, the user completes the login form and clicks **Submit**. The form submission triggers an HTTP POST request that contains login credentials to the IdP.
5. After successful validation of the user credentials at the IdP, the IdP populates the SAML Response and returns it in an HTML form that contains hidden input fields. Several redirects might occur before the 200 OK HTTP response that contains the SAML information is received. Universal Access does not respond to the redirects.
6. Universal Access extracts the RelayState and SAMLResponse values, and inserts them in a new POST request to the application server Assertion Consumer Service (ACS).
7. The application server ACS validates the signature that is contained in the SAML Response. WebSphere Application Server also ensures that the originator is a Trusted Authentication Realm. If the validation is successful, the ACS sends an HTTP redirect that points to the configured IBM Cúram Social Program Management target landing page, along with an LTPA2 Cookie that will be used in any subsequent communication.

8. Universal Access begins its standard user setup by requesting account and profile information from the relevant web API endpoints.

SP-initiated flow in Universal Access

When Universal Access is configured with an SP-initiated web SSO flow, any attempt to connect to a protected resource without first authenticating results in a 401 HTTP response from the application server Assertion Consumer Service's Trust Association Interceptor, and the generation of the SAML AuthnRequest message to be sent to the IdP.

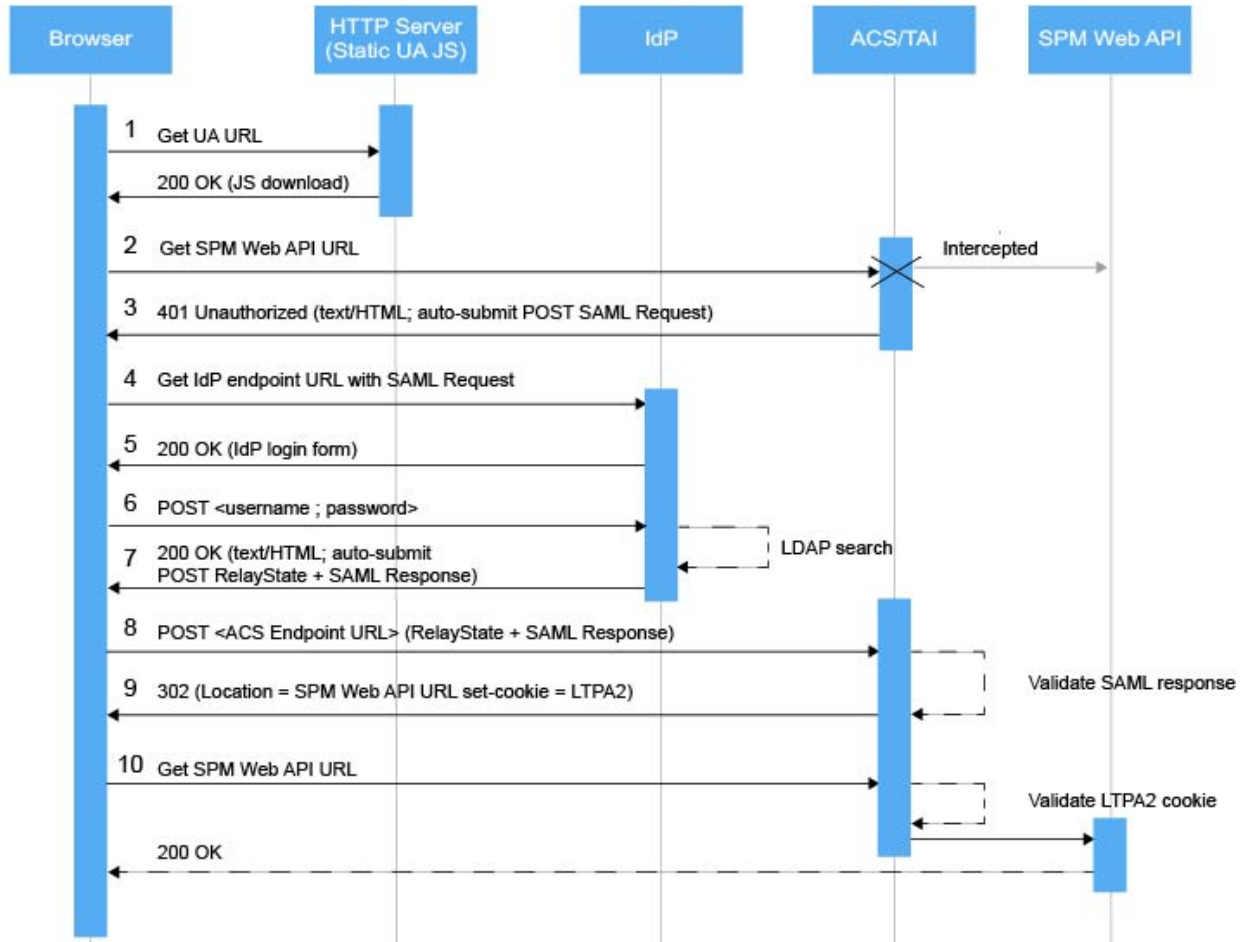


Figure 5. SP-initiated flow in Universal Access

1. A user browses to the HTTP server that contains Universal Access.
2. The user can browse as normal by interacting with IBM Cúram Social Program Management as either a public or a generated user (which is not shown in the diagram). The user then accesses protected content in the application, which is intercepted by the Assertion Consumer Service Trust Association Interceptor (TAI).
3. The TAI triggers an 401 HTTP response with the SAML request message to be sent to the IdP.
4. Universal Access then directs the SAML Request to the IdP SAML endpoint.
5. In most IdP configurations, an HTML login form responds to the request. Universal Access extracts a hidden authentication token in the login form if present, ignoring the rest of the response.
6. To authenticate, the user completes the login form and clicks **Submit**. The form submission triggers an HTTP POST request that contains login credentials to the IdP, along with the token extracted in the previous step if present.
7. After successful validation of the user credentials at the IdP, the IdP populates the SAML Response and returns it in an HTML form that contains hidden input fields. Several redirects might occur before

the 200 OK HTTP response that contains the SAML information is received. Universal Access does not respond to the redirects.

8. Universal Access extracts the RelayState and SAMLResponse values, and inserts them in a new POST request to the application server Assertion Consumer Service (ACS).
9. The application server ACS validates the signature that is contained in the SAML Response. WebSphere Application Server also ensures that the originator is a Trusted Authentication Realm. If the validation is successful, the ACS sends an HTTP redirect that points to the configured IBM Cúram Social Program Management target landing page, along with an LTPA2 Cookie that will be used in any subsequent communication.
10. The browser automatically sends a new request to the target URL, but Universal Access does not respond to the request. Universal Access begins its standard user setup by requesting account and profile information from the relevant web API endpoints.

Related information

[Oasis: SAML 2.0 Technical Overview](#)

[Oracle: JAAS Authorization Tutorial](#)

Configuring single sign-on properties

To enable IBM Cúram Universal Access to work with SAML single sign-on (SSO), configure the appropriate properties in the `.env` environment variable file in the root of the application and rebuild the application.

About this task

- The `<IdP_URL>` consists of three parts: the HTTPS protocol, the IdP hostname or IP address, and the listener port number. For example, `https://192.168.0.1:12443`.
- The `<ACS_URL>` consists of three parts: the HTTPS protocol, the Assertion Consumer Service (ACS) hostname or IP address, and the listener port number. For example, `https://192.168.0.2:443`.

Procedure

1. Set the authentication method to SSO, see [“Customizing the authentication method” on page 153](#).
2. Set the related environment variables for your SSO environment, see [“Single sign-on \(SSO\) authentication” on page 177](#). These properties are applicable to both identity provider (IdP)-initiated and service-provider (SP)-initiated SAML 2.0 web SSO unless otherwise stated.

Configuring cross-origin resource sharing

For security reasons, browsers restrict cross-origin HTTP requests, including XMLHttpRequest HTTP requests, that are initiated inside IBM Cúram Universal Access. When the Universal Access application and the Universal Access web API are deployed on different hosts, extra configuration is required.

About this task

Universal Access can request HTTP resources only from the same domain that the application was loaded from, which is the domain that contains the static JavaScript. To enable Universal Access to support cross-origin resource sharing (CORS), enable the use of CORS headers.

Procedure

1. Log on to the IBM Cúram Social Program Management application as a system administrator, and click **System Configurations**.
2. In the Shortcuts panel, click **Application Data > Property Administration**.
3. Configure the `curam.rest.allowedOrigins` property with the values of either the host names or the IP addresses of the IdP server and the web server on which Universal Access is deployed.

Related information

[Cúram REST configuration properties](#)

Single sign-on configuration example

The example outlines a single sign-on (SSO) configuration for IBM Cúram Universal Access that uses IBM Security Access Manager to implement federated single sign-on by using the SAML 2.0 Browser POST profile. The example applies to both IdP-initiated and SP-initiated flows. Some additional steps are required to configure SP-initiated flows.

Universal Access SSO configuration components

The following figure shows the components that are included in a Universal Access SSO configuration.

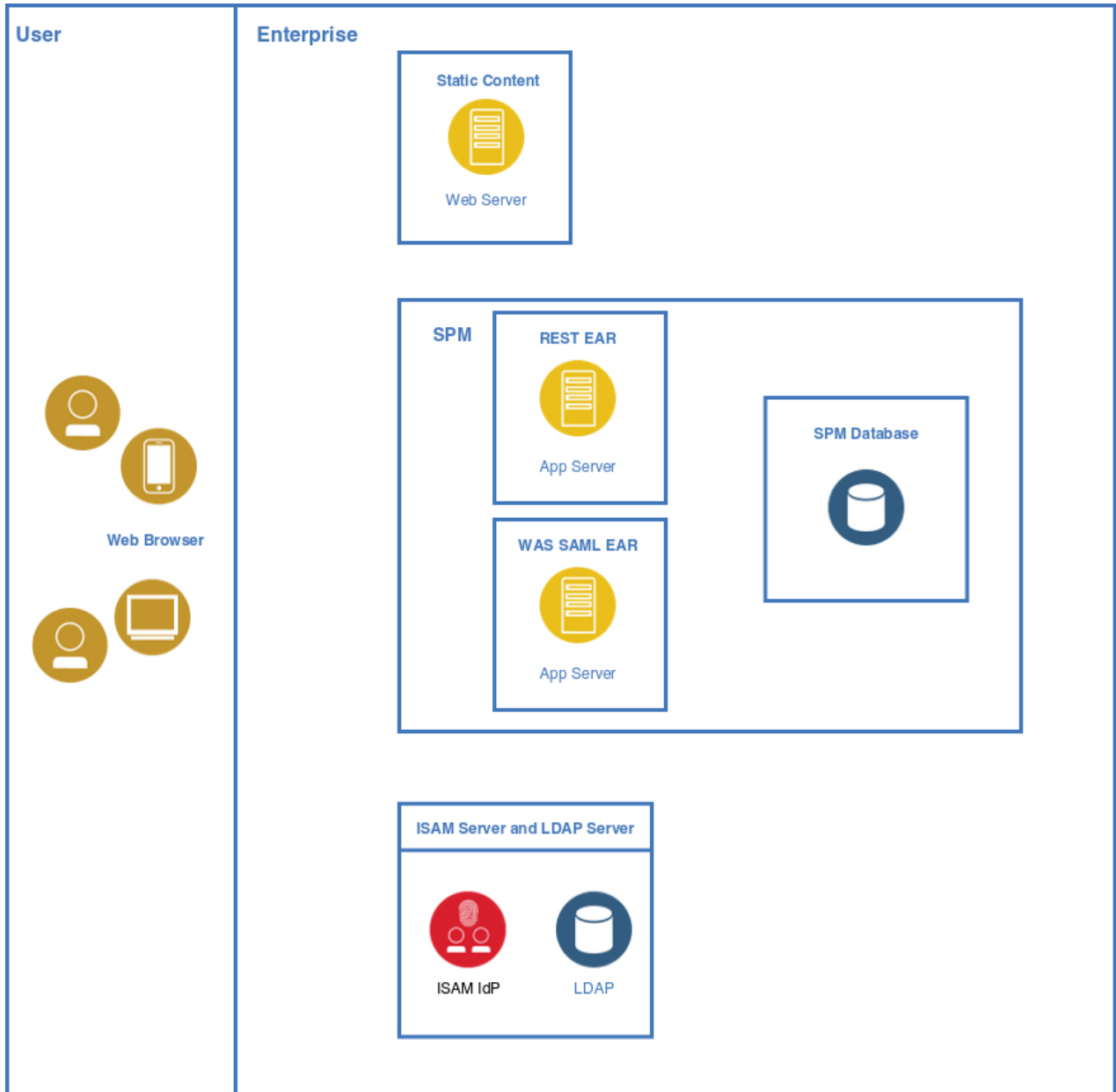


Figure 6. Universal Access SSO configuration components

Web browser

A user sends requests from their web browser for applications in the SSO environment.

Web server

The Universal Access ReactJS static content is deployed on a web server, such as IBM HTTP Server, or Apache HTTP Server.

IBM Security Access Manager (ISAM) server

The IBM Security Access Manager server includes the identity provider (IdP).

LDAP server (user directory)

Among other items, the LDAP server contains the user name and password of all the valid users in the SSO environment.

IBM WebSphere Application Server

Among other applications, WebSphere Application Server contains the deployed IBM Cúram Social Program Management, Citizen WorkSpace, and REST enterprise applications.

WebSphere Application Server SAML EAR

A WebSphere package that contains the packages to run the SAML Assertion Consumer Service (ACS).

SPM Database

Data storage for the IBM Cúram Social Program Management, Citizen WorkSpace, and REST enterprise applications.

Configuring single sign-on through IBM Security Access Manager

Use the IBM Security Access Manager management console to configure single sign-on (SSO) in IBM Cúram Universal Access.

Before you begin

1. Start IBM Security Access Manager.
2. In the management console, log on as an administrator.
3. Accept the services agreement.
4. If required, change the administrative password.

About this task

In the IBM Security Access Manager management console, complete the following steps, with reference to the *IBM Security Access Manager 9 Federation Cookbook*.

Procedure

1. Configure the IBM Security Access Manager database:
 - a) In the top menu, click **Home Appliance Dashboard > Database Configuration**.
 - b) Enter the database configuration details, such as **Database Type, Address, Port**, and so on, and click **Save**.
 - c) When the **Deploy Pending Changes** window opens, click **Deploy**.
2. To install all the required product licenses, complete the steps in section 4.3 *Product Activation* from the *IBM Security Access Manager 9 Federation Cookbook*.
3. Configure the LDAP SSL database by completing section 25.1.1 *Load Federation Runtime SSL certificate into pdsrv trust store* from [IBM Security Access Manager Federation Cookbook](#).
4. Configure the runtime component by completing 4.6 *Configure ISAM Runtime Component on the Appliance* from [IBM Security Access Manager Federation Cookbook](#).

Configuring IBM Security Access Manager as an IdP

To configure IBM Security Access Manager as an identity provider (IdP), see the IBM Security Access Manager 9.0 Federation Cookbook that is available from IBM Developer Works.

Before you begin

Download the IBM Security Access Manager 9.0 Federation Cookbook from IBM Developer Works, as shown in the related link. Also download the mapping files that are provided with the cookbook.

About this task

To set up the example environment, complete the specified sections in the IBM Security Access Manager 9.0 Federation Cookbook.

Procedure

1. Complete *Section 5, Create Reverse Proxy instance*.

2. Complete *Section 6, Create SAML 2.0 Identity Provider federation*.

In Section 6.1, if you are using the ISAM docker deployment, it is possible to re-use the existing keystore that is included in the container instead of creating a new keystore. It is important to reflect this change in subsequent sections where the myidpkeys certificate database is referenced.

3. Complete *Section 8.1, ISAM Configuration for the IdP*.

In Section 8.1, use the host name of the IdP federation.

4. Optional: After completing Section 8.1.1, if you require ACLs to be defined to allow and restrict access to the IdP junction, then follow the instructions in *Section 25.1.3, Configure ACL policy for IdP*.

5. Complete *Section 9.1, Configuring Partner for the IdP*.

The export from Websphere does not contain all the relevant data. Therefore, in Section 9.1, after you complete configuring partner for the IdP, you must click **Edit configuration** and complete the remaining advanced configuration.

Related information

[IBM Security Access Manager 9.0 Federation Cookbook](#)

Configuring WebSphere Application Server

The procedure outlines the high-level steps that are required to configure IBM WebSphere Application Server as a SAML service provider.

About this task

For more information, see the related link to the WebSphere Application Server documentation.

Procedure

1. Deploy the WebSphereSam1SP.ear file.

Note: So that SAML Assertion Consumer Service (ACS) works with cross-origin resource sharing (CORS) security requirements during redirections, you must map its modules to the same virtual host used for the REST target application (that is, *client_host*).

The WebSphereSam1SP.ear file is available as an installable package. Choose one of the following methods:

- Log on to the WebSphere Application Server administrative console, and install the app_server_root/installableApps/WebSphereSam1SP.ear file to your application server or cluster.
- Install the SAML ACS application by using a Python script. In the app_server_root/bin directory, enter the following command to run the installSam1ACS.py script:

```
wsadmin -f installSam1ACS.py install nodeName serverName
```

Where *nodeName* is the node name of the target application server, and *serverName* is the server name of the target application server. When you complete this step, you must map the modules to the REST application, for more information see: [Mapping virtual hosts for web modules](#).

2. Configure the ACS trust association interceptor:

- a) In the WebSphere Application Server administrative console, click **Global security > Trust association > Interceptors > New**.

- b) For **Interceptor class name**, enter `com.ibm.ws.security.web.saml.ACSTrustAssociationInterceptor`.
- c) Under custom properties, enter the values that are shown in the following table:

In a standard WebSphere Application Server configuration, you would also define a value for the `login.error.page` custom property. However, the preferred method is to log on to the IdP first. Therefore, if you do not define a value for `login.error.page`, WebSphere Application Server returns a 403 error if a user logs on without first logging on to the identity provider (IdP).

Table 16. ACS trust association interceptor custom properties	
Custom property name	Value
sso_1.sp.acsUrl	<code>https://WAS_host_name:ssl port//samlsp/acs</code>
sso_1.idp_1.EntityID	<code>https://isam_hostname:isam_port//URL of ISAM/ISAM Junction/IdP endpoint/federation name/saml20</code>
sso_1.idp_1.SingleSignOnUrl	<code>https:// isam_hostname:isam_port//URL of ISAM/ISAM Junction/IdP endpoint/federation name/saml20/login</code>
sso_1.sp.targetUrl	<code>https://WAS_host_name:WAS_port/Rest</code>
sso_1.idp_1.certAlias	<code>isam-conf</code>
sso_1.sp.filter	<code>request-url^=/Rest;request-url!=/Rest/j_security_check</code>
sso_1.sp.enforceTaiCookie	<code>false</code>

3. Add the IdP federation partner data. The following substeps describe how to add the IdP data by using the WebSphere Application Server administrative console.
 - a) To add the IdP host name or IP address as a trusted realm, click **Global security > Trusted authentication realms - inbound > Add External Realm**.
 - b) Enter either the IBM Security Access Manager host name or IP address.
 - c) To load the IdP certificate from IBM Security Access Manager, click **Security > SSL certificate and key management > Key stores and certificates > NodeDefaultTrustStore > Signer certificates > Retrieve from port**
 - d) Enter the IBM Security Access Manager IP address and listener port, for example, 12443, alias = `isam-conf`.

Note: When the browser first attempts to connect to the IBM Cúram Social Program Management web API, an LTPA2 cookie is sent as part of the request. If the WebSphere Application Server **`com.ibm.ws.security.web.logoutOnHTTPSessionExpire`** property is set to true, which is the default configuration in IBM Cúram Social Program Management, then authentication fails because an HTTP session does not exist on the application server. By setting the property to false, the check for a valid HTTP session is not completed and when the LTPA2 token is valid, authentication succeeds.

To configure the property in the WebSphere Application Server administrative console, click **Security > Global security > Custom properties**, and set the value of **`com.ibm.ws.security.web.logoutOnHTTPSessionExpire`** to false.

4. Implement cross-origin resource sharing (CORS) from the HTTP server to the WebSphere Application Server SAML ACS.
 - a) To add a CORS header, configure a servlet filter for the `WebSphereSam1SP.ear` file that is deployed by a Trust Association Interceptor (TAI). The servlet filter adds a CORS HTTP header to HTTP responses. You can archive the implemented servlet filter as a jar file, and then store it in the `WebSphereSam1SP.ear\WebSphereSam1SPWeb.war\WEB-INF\lib` directory that is in the `installedApps` directory of your project in WebSphere Application Server.

See the following example of how to implement a servlet filter:

```
public class SampleFilter implements Filter {

    @Override
    public void doFilter(ServletRequest arg0, ServletResponse servletResponse,
        FilterChain arg2) throws IOException, ServletException {

        HttpServletResponse response = (HttpServletResponse) servletResponse;
        HttpServletRequest request = (HttpServletRequest) arg0;

        response.setHeader("Access-Control-Allow-Origin",
            "http://dubxpcvm156.mul.ie.ibm.com:9880");    <hostname or IP address of IBM UA
server>
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.setHeader("Access-Control-Allow-Headers", "x-requested-with, Content-Type,
origin, authorization, accept, client-security-token");
        response.setHeader("Access-Control-Expose-Headers", "content-length");
        arg2.doFilter(request, response);
    }
}
```

- b) Configure the `web.xml` file for the deployed TAI EAR file to use the servlet filter for all the requests. Add the filter element that is shown in the following sample to the `web.xml` file, with the actual fully qualified name of the filter.

You can add the filter element as a sibling to any existing element in the `web.xml` file, such as `<servlet>`. The `web.xml` file is in the `WebSphereSam1SP.ear\WebSphereSam1SPWeb.war\WEB-INF\lib` directory, which is in the `installedApps` directory of your project in WebSphere Application Server.

```
<filter>
  <filter-name> SampleFilter </filter-name>
  <filter-class> SampleFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name> SampleFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Related information

[Enabling WebSphere Application Server to use the SAML web SSO feature](#)

Add and enable the users in LDAP

Complete the following steps to add the users from LDAP and enable them in ISAM.

Procedure

1. To create LDAP and IBM Security Access Manager runtime users, create an `ldif` file that can be used to populate OpenLdap, as shown in the following sample:

```
# cat UA_usersCreate_ISAM.ldif
dn: dc=watson-health,secAuthority=Default
objectclass: top
objectclass: domain
dc: watson-health

dn: c=ie,dc=watson-health,secAuthority=Default
objectclass: top
objectclass: country
c: ie

dn: o=curam,c=ie,dc=watson-health,secAuthority=Default
objectclass: top
objectclass: organization
o: curam

dn: ou=curamint,o=curam,c=ie,dc=watson-health,secAuthority=Default
objectclass: top
objectclass: organizationalUnit
ou: curamint

dn: cn=caseworker,ou=curamint,o=curam,c=ie,dc=watson-health,secAuthority=Default
```

```

objectclass: person
objectclass: inetOrgPerson
objectclass: top
objectclass: organizationalPerson
objectclass: ePerson
cn: caseworker
sn: caseworkersurname
uid: caseworker
mail: caseworker@curam.com
userpassword: Passw0rd

dn: ou=curamext,o=curam,c=ie,dc=watson-health,secAuthority=Default
objectclass: top
objectclass: organizationalUnit
ou: curamext

dn: cn=jamesmith,ou=curamext,o=curam,c=ie,dc=watson-health,secAuthority=Default
objectclass: person
objectclass: inetOrgPerson
objectclass: top
objectclass: organizationalPerson
objectclass: ePerson
cn: jamesmith
sn: Smith
uid: jamesmith
mail: jamesmith@curamexternal.com
userpassword: Passw0rd

```

2. Add users to the OpenLDAP database:

- a) On the host server that is running the docker containers, enter the following command:

```
docker cp UA_usersCreate_ISAM.ldif idpisam9040_isam-ldap_1:/tmp
```

- b) To log on to the OpenLDAP container, enter the following command:

```
docker exec -ti idpisam9040_isam-ldap_1 bash
```

- c) To add the users to OpenLDAP, enter the following command:

```
ldapadd -H ldaps://127.0.0.1:636 -D cn=root,secAuthority=default -f /tmp/
Curam_usersCreate_ISAM.ldif
```

3. Import the users into IBM Security Access Manager:

- a) To log on to the IBM Security Access Manager command line interface, enter the following commands:

```
docker exec -ti idpisam9040_isam-webseal_1 isam_cli
isam_cli> isam admin
pdadmin> login -a sec_master -p <password>
```

- b) To import the users into IBM Security Access Manager, enter the following commands:

```
pdadmin sec_master> user import caseworker
cn=caseworker,ou=curamint,o=curam,c=ie,dc=watson-health,secAuthority=Default
pdadmin sec_master> user modify caseworker account-valid yes
pdadmin sec_master> user import jamesmith
cn=jamesmith,ou=curamext,o=curam,c=ie,dc=watson-health,secAuthority=Default
pdadmin sec_master> user modify jamesmith account-valid yes
```

4. To test the identity provider (IdP) flow, enter the following URL in a browser:

```
https://ISAM login initial URL?RequestBinding=HTTPPost
&PartnerId=webspherehostname:9443/samlsp/acs&NameIdFormat=Email
&Target=WAS hostname:WAS port/Rest/v1
```

Replace the following values in the URL with the appropriate values for your configuration:

- *IBM Security Access Manager login initial URL*
- *WebSphere host name*
- *WebSphere Application Server host name*

- *WebSphere Application Server port*; in IBM Cúram Social Program Management the default value is 9044

When the IBM Security Access Manager docker container starts, the IdP endpoints are initialized only when the first connection request is received. However, if the first connection request is triggered by IBM Cúram Universal Access, an XHR timeout occurs before the initialization finishes. Therefore, this test step is required to ensure that the initialization of the IdP endpoints is completed.

5. In a browser, go to the home page and log in.

Test IdP-initiated SAML SSO infrastructure

When the IBM Security Access Manager docker container starts, the IdP endpoints are initialized only when the first connection request is received. However, if the first connection request is triggered by Universal Access, an XHR timeout occurs before the initialization finishes. This test step is required to ensure that the initialization of the IdP endpoints is completed.

Procedure

To test the identity provider (IdP) flow, enter the following URL in a browser:

```
https://<isam_url>/isam/sps/saml20idp/saml20/logininitial?
RequestBinding=HTTPPost&PartnerId=https://<was_url>/samlsp/acs&NameIdFormat=Email&Target=<
was_url>/Rest/api/definitions
```

where:

- **<isam_url>** - The URL for IBM Security Access Manager. It consists of the IBM Security Access Manager host name, and port number, for example, `https:// 192.168.0.1:12443`.
- **<junction_name>** - The junction name that is used during the federation configuration in reverse proxy. The default value is `isam`.
- **<idp_endpoint>** - The endpoint that is configured for the IDP federation. The default value is `sps`.
- **<federation_name>** - The name that was used when creating the federation.
- WebSphere host name
- WebSphere Application Server host name
- WebSphere Application Server port. The default value is 9044 for IBM Cúram Social Program Management.

SP-Initiated only: Implementing the SAML AuthnRequest functionality in WebSphere Application Server
WebSphere Application Server does not support SP-initiated SAML web SSO by default. In addition to the previous steps, you must also implement the provided `com.ibm.wsspi.security.web.saml.AuthnRequestProvider` interface to handle the AuthnRequest functionality that is needed in the service provider.

About this task

For more information, see [Enabling SAML SP-Initiated web single sign-on \(SSO\)](#) in the WebSphere Application Server documentation.

Procedure

1. Implement the `AuthnRequestProvider` interface as in the following example. Note that in the `getAuthnRequest` method, the `ssoUrl` variable is set to the value of the `ACSTrustAssociationInterceptor` interceptor property `sso_1.idp_1.SingleSignOnUrl`, while `acsUrl` is set to the value of the `sso_1.sp.acsUrl` property.

```
package curam.sso;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Base64;
import java.util.Date;
import java.util.HashMap;
```

```

import java.util.TimeZone;
import javax.servlet.http.HttpServletRequest;
import com.ibm.websphere.security.NotImplementedException;
import com.ibm.wsspi.security.web.saml.AuthnRequestProvider;
public class SPInitTAI implements AuthnRequestProvider {
    @Override
    public String getIdentityProviderOrErrorURL(HttpServletRequest arg0, String arg1, String
arg2,
        ArrayList<String> arg3) throws NotImplementedException {

        return null;
    }
    @Override
    public HashMap<String, String> getAuthnRequest(HttpServletRequest arg0, String arg1,
String arg2,
        ArrayList<String> paramArrayList) throws NotImplementedException {

        //create map with following keys
        HashMap <String, String> map = new HashMap <String, String>();

        String ssoUrl = "https://<isam_hostname>:<isam_port>/<URL of ISAM>/<ISAM Junction>/
<IdP endpoint>/<federation name>/saml20/login";
        String acsUrl = "https://<WAS_host_name>:<ssl port>/samlsp/acs";
        String issuer = acsUrl;
        String destination = ssoUrl;

        map.put(AuthnRequestProvider.SSO_URL, ssoUrl);
        map.put(AuthnRequestProvider.RELAY_STATE, acsUrl);
        String requestID = "Test" + Double.toString(Math.random());
        map.put(AuthnRequestProvider.REQUEST_ID, requestID);

        String authnMessageNew = "<samlp:AuthnRequest xmlns:samlp=
\"urn:oasis:names:tc:SAML:2.0:protocol\" "
            + "ID=\""+requestID+"\" "
            + "Version=\"2.0\" "
            + "IssueInstant=\""+getDateTime()+"\" ForceAuthn=\"false\" IsPassive=\"false
\" "
            + "ProtocolBinding=\"urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST\" "
            + "AssertionConsumerServiceURL=\""+acsUrl+"\" "
            + "Destination=\""+destination+"\"> "
            + "<saml:Issuer xmlns:saml=\"urn:oasis:names:tc:SAML:2.0:assertion\">"+issuer
            + "</saml:Issuer> <samlp:NameIDPolicy Format=
\"urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress\" AllowCreate=\"true\" />
            + "<samlp:RequestedAuthnContext Comparison=\"exact\">
<saml:AuthnContextClassRef xmlns:saml=\"urn:oasis:names:tc:SAML:2.0:assertion\">
            + \"urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport</
saml:AuthnContextClassRef></samlp:RequestedAuthnContext> </samlp:AuthnRequest>";

        String encodedAuth = Base64.getEncoder().encodeToString(authnMessageNew.getBytes());

        map.put(AuthnRequestProvider.AUTHN_REQUEST, encodedAuth);

        return map;
    }

    private String getDateTime() {
        // e.g 2018-11-11T23:52:45Z
        String pattern = "yyyy-MM-dd'T'HH:mm:ss'Z'";
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern);
        simpleDateFormat.setTimeZone(TimeZone.getTimeZone("Zulu"));
        String date = simpleDateFormat.format(new Date());
        return date;
    }
}

```

2. Pack your AuthnRequestProvider implementation in a JAR, and place it in WAS_HOME/lib/ext.
3. Ensure that your AuthnRequestProvider implementation class is added to the ACSTrustAssociationInterceptor custom property sso_1.sp.login.error.page so that it can handle errors.
 - a) In the WebSphere Application Server admin console, go to **Security > Global Security > Web and Sip Security > Trust association > Interceptors > com.ibm.ws.security.web.saml.ACSTrustAssociationInterceptor**.
 - b) Set the sso_1.sp.login.error.page custom property to the value curam.sso.SPInitTAI.

- c) Click **OK** and save the configuration.
4. You might need to restart the application server for the changes to take effect.

SP-Initiated only: Test SP-initiated SAML SSO infrastructure

Complete the following steps to test the SP-initiated SAML SSO infrastructure.

Procedure

1. Open your browser, with network devtools, and load a protected REST URL like this example:

```
<was_url>/Rest/api/definitions
```

where `<was_url>` is the WebSphere URL, for example `https:// 192.168.0.1`.

2. You are redirected to the ISAM log-in page. Log in with the credentials that were used to set the reverse proxy instance as outlined in [“Configuring IBM Security Access Manager as an IdP” on page 160](#).
3. You should be redirected to the definitions page that you opened in step 1.

External security authentication example

Ensure that citizens can be authenticated for any of your services by using a single set of credentials, which provides the benefits of a streamlined authorization process for both governments and citizens. An example outlines the implementation of a set of customization requirements for a team that is deploying Universal Access.

Universal Access, by default, authenticates against a set registered users that are stored on the Cúram database. You can also configure the system to integrate with external security systems. You can improve security by enabling the use of a single set of credentials, because citizens do not have to remember lists of user names and passwords and, hence, are less likely to write down their user names and passwords. Also, security efforts are focused on implementing best practice in a single Enterprise Security System.

Consider an example analysis of requirements to integrate with an external security system. Any analysis of requirements for external security integration should consider the following minimum questions:

- Does your deployment support anonymous screening, anonymous intake, or both?
- Is account management supported in IBM Cúram Universal Access or in the external security system?
- Is single sign-on (SSO) required?

Example customization requirements

The topics in this section describe the configuration and development tasks to implement the following set of customization requirements for a team that is deploying Universal Access. The topics refer to the requirements as appropriate.

1. Users can access Universal Access and perform anonymous screening or intake.
2. Users who want to access their saved screening or intake information must first create an account on a system called CentralID.
3. Users who log in to Universal Access can use their CentralID username and password to authenticate.
4. Users perform all of their account management using an external system that is named CentralID, for example, resetting a password, creating a new account, changing account details.
5. CentralID stores all user records in a secure LDAP server.
6. Because all account management is now performed in CentralID, the account creation screens and password reset screens are to be removed from Universal Access.
7. Users should be able to log in as soon as they have registered with CentralID, and there should be no delay while waiting for an ID to propagate to Universal Access.

Configuring an alternative login ID

By default, you cannot change user names after they are created. However, you can configure an alternative login ID that can be updated.

For information about configuring alternative login IDs, see *Alternate login IDs*. If you configure an alternative login ID for a user name that is case-sensitive, then the alternative login ID is also case-sensitive.

Related information

[Alternate Login IDs](#)

Deploying in identity-only mode for registered users

You must configure the application server to use LDAP for authentication if a user is in Identity-Only mode. Also, configure the necessary properties to deploy in identity-only mode for registered users.

Configuring the application server to use LDAP for authentication in Identity-Only mode

If a user is in Identity-Only mode, it is necessary to match the login IDs that are stored in LDAP with the login IDs that are stored in the ExtendedUsersInfo table.

For information about how to configure your application server to use LDAP for authentication, see the relevant application server documentation.

Configuring properties to deploy in identity-only mode for registered users

Add the following properties to the *AppServer.properties* file:

```
curam.security.check.identity.only=true
curam.security.user.registry.disabled.types=EXT_AUTO,EXT_GEN
curam.citizenworkspace.enable.usertypes.for.temporary.users=true
public.user.type=EXT_AUTO
```

To reconfigure the application server, run the following command:

```
appbuild configure
```

The `curam.security.check.identity.only` property ensures that application security is set to work in Identity Only mode. For more information about Identity Only authentication mode, see either *Deployment Guide for WebSphere* or *Deployment Guide for WLS*. In Identity Only mode, authentication uses only the internal user table to check for the existence of the user. The validation of the password is left to a subsequent module, either a JAAS module (Oracle WebLogic) or the User Registry (IBM WebSphere).

Take the example of a user, "johnsmith", who has been registered with the CentralID LDAP server. For John Smith to be able to use Universal Access, there must also be a "johnsmith" entry in the ExternalUser table. When John Smith logs in, his authentication request is passed to the Cúram JAAS Login Module. The Cúram JAAS Login Module checks that the user johnsmith exists in the Cúram ExternalUser table but does not check the password. The authentication then proceeds to the User Registry (WebSphere) or LDAP JAAS Module (WebLogic) where the user name and password are checked against the contents of the CentralID LDAP server. For the authentication to work correctly, it is necessary to configure the application server with the connection details for the secure LDAP server.

The Identity Only configuration allows the application to defer to an external security system such as an LDAP-based directory service for the authentication of user credentials. However, when an anonymous user accesses the organization **Home** page for the first time, the user is automatically logged in as a publiccitizen user. Subsequently, if the user chooses to screen themselves or to perform an intake, Universal Access creates a new "generated" anonymous user. Each generated user is unique, which ensures that the data that belongs to that user is kept confidential. Public citizen users and generated users are not inserted into the LDAP directory, so they cannot be authenticated by using the Identity Only

mechanism. The following line ensures that users with the user type EXT_AUTO (public citizen users) and EXT_GEN (generated users) are authenticated against the External User table:

```
curam.security.user.registry.disabled.types=EXT_AUTO,EXT_GEN
```

After the previous configuration has been applied to the server and the server has been started, perform the following configuration steps:

1. Log in as sysadmin.
2. **Select Application Data > Property Administration.**
3. Select category **Citizen Account - Configuration.**
4. Set the property **curam.citizenaccount.public.included.user** to EXT_AUTO.
5. Set the property **curam.citizenaccount.anonymous.included.user** to EXT_GEN.
6. Set the property **curam.citizenworkspace.enable.usertypes.for.temporary.users** to TRUE.
7. **Publish** the property changes.

You need another configuration entry so that Universal Access operates correctly with respect to authentication as shown in the following steps:

8. Select **Select Application Data > Property Administration.**
9. Select category **Infrastructure – Security parameters.**
10. Set **curam.custom.externalaccess.implementation** to **curam.citizenworkspace.security.impl.CitizenWorkspacePublicAccessSecurity.**
11. **Publish** the property changes.
12. Log out and restart the server.

Disabling the Create Account screens

Configure the necessary properties to disable the screens for creating an account that Universal Access provides by default. Requirement 4 in the example requirements indicates that all account management functions are handled by the external system, CentralID, including the creation of a new account and performing a password reset.

Configure Universal Access to disable the screens that are related to account management:

1. Log in as sysadmin.
2. Select **Application Data > Property Administration.**
3. Select Category **Citizen Portal - Configuration.**
4. Set the property *curam.citizenworkspace.enable.account.creation* to **NO**.
5. **Publish** the property changes.

The previous steps remove references to **Account Creation** pages from Universal Access. The Login screen still contains a link to a page for changing passwords. In this example, the implementation team can use the following steps to retain the link but change it to open a new browser window on the CentralID password reset page:

1. Log in as sysadmin.
2. Select **Application Data > Property Administration.**
3. Select Category **Citizen Portal - Configuration.**
4. Set the property *curam.citizenworkspace.forgot.password.url* to , for example **http://www.centralid.gov/resetpassword**
5. **Publish** the property changes.

To completely remove the reset password link, use the following steps:

1. Log in as sysadmin.
2. Select **Application Data > Property Administration.**

3. Select Category **Citizen Portal - Configuration**.
4. Set the property *curam.citizenworkspace.display.forgot.password.link* to **NO**.
5. **Publish** the property changes.

Redirecting users to register with an external system

Replace the message that is displayed in the log in page so that non-registered users are directed to the CentralID page for registration.

Universal Access invites users to log in with a log in message. You can replace the message so that the log in page displays a message that is similar to the following example:

```
"<p>If you are registered with CentralID enter your user name
and password to log in. To register, go to
<a href="http://www.centralid.gov/register"> The CentralID
registration page.</a></p>"
```

The properties for controlling the login page message are contained in the <CURAM_DIR>/EJBServer/components/Data_Manager/Initial_Data/blob/prop/Logon.properties file.

Enabling users to log on immediately after registration with CentralID

Users should be able to log in as soon as they have registered with CentralID. Some configuration is required to prevent a delay in the propagation of a user's ID to other systems.

To function correctly, each user must have an entry in the ExternalUser table. The customer could build a batch process to import users from the LDAP directory into the ExternalUser table. However, requirement 7 in the example requirements would not be satisfied, which states that users must be able to register with CentralID, and then immediately use Universal Access. Another option would be to build a web service or similar mechanism that would be launched when a new user registers with CentralID. The implementation of the web service would create the appropriate entry in the ExternalUser table.

A simpler option is to override the default log-in behavior to create new accounts as needed, after the completion of checks to ensure that the relevant entry exists in the LDAP server. You can override the default log-in behavior in Universal Access by extending the *curam.citizenworkspace.security.impl.AuthenticateWithPasswordStrategy* class and overriding the *authenticate()* method. The following code outlines how to use the *AuthenticateWithPasswordStrategy* and other security APIs to meet the previous requirements:

```
public class CustomSecurityStrategy extends AuthenticateWithPasswordStrategy {
    @Inject
    private CitizenWorkspaceAccountManager cwAccountManager;
    ...
    @Override
    public String authenticate(final String username,
        final String password)
        throws AppException, InformationalException {
        final String retval = null;
        if (username.equals(PUBLIC_CITIZEN)) {
            return super.authenticate(username, password);
        }
        // Authenticate generated accounts as normal
        if (cwAccountManager.isGeneratedAccount(username)) {
            return super.authenticate(username, password);
        }
        // Check that the user exists in LDAP
        // This prevents hackers from registering many bogus
        // accounts that exist in Curam but not in LDAP
        if (!isUserInLDAP(username)) {
            return SECURITYSTATUS.BADUSER;
        }
        // If there's no account for this user
        if (!cwAccountManager.hasAccount(username)) {
            createUserAccount(username);
        }
        return SECURITYSTATUS.LOGIN;
    }
    private void createUserAccount(final String username)
        throws AppException, InformationalException {
        final CreateAccountDetails newAcctDetails;
        ...
        cwAccountManager.createStandardAccount(newAcctDetails);
    }
}
```

```
}
}
```

This code checks to see whether the user is logging in is a public citizen user or a generated account. In both cases, authentication logic is delegated to the default `AuthenticateWithPasswordStrategy` API. In the case of a registered user, the Strategy checks the LDAP directory to ensure that the user exists in the LDAP directory. If the user exists in the LDAP directory and does not exist yet in Universal Access, then a new user account is created. Note, the custom code does not need to authenticate the user against LDAP since the authentication is handled by the User Registry in WebSphere or the LDAP JAAS Module in WebSphere. It is important to note that the password parameter of the `authenticate()` method is passed in clear text.

To install the `CustomSecurityStrategy` class, it must be bound in place of the Default Security Strategy class. Use a Guice Module to bind the implementation:

```
public class CustomModule extends AbstractModule {
    @Override
    protected void configure() {
        binder().bind(SecurityStrategy.class).to(
            CustomSecurityStrategy.class);
    }
}
```

You must configure the `CustomModule` at startup by adding a DMX file to the custom component as shown in the following example:

```
<CURAM_DIR>/EJBServer/custom/data/initial/MODULECLASSNAME.dmx

<?xml version="1.0" encoding="UTF-8"?>
<table name="MODULECLASSNAME">
  <column name="moduleName" type="text" />
  <row>
    <attribute name="moduleName">
      <value>gov.myorg.CustomModule</value>
    </attribute>
  </row>
</table>
```

Customizing account creation and management

You can customize account creation and management.

Account management configurations

Use the following configuration properties to define the behavior of password validations for citizen accounts.

Table 17. Account configurations	
Property	Description
<code>curam.citizenworkspace.username.min.length</code>	Minimum number of characters in the username.
<code>curam.citizenworkspace.username.max.length</code>	Maximum number of characters in the username.
<code>curam.citizenworkspace.password.min.length</code>	Minimum number of characters in the password.
<code>curam.citizenworkspace.password.max.length</code>	Maximum number of characters in the password.
<code>curam.citizenworkspace.password.min.special.chars</code>	Minimum number of special characters and/or numbers in the password.

To update these properties, log in as System Administrator (sysadmin), select **Application Data > Property Administration**, and search for the property.

Account management events

Events are raised at key points during account processing. The events can be used to add custom validations to the account management process.

For more information about adding custom validations to the account management process, see the *Cúram Server Developer* section. The following table shows the events that are in the `curam.citizenworkspace.security.impl.CitizenWorkspaceAccountEvents` class:

Table 18. Account events	
Event Interface	Description
<code>CitizenWorkspaceCreateAccountEvents</code>	Events raised around account creation. For more information, see the related Javadoc information in the <code>WorkspaceServices</code> component.
<code>CitizenWorkspacePasswordChangedEvent</code>	Event raised when a user is changing their password. For more information, see the related Javadoc information in the <code>WorkspaceServices</code> component.
<code>CitizenWorkspaceAccountAssociations</code>	Events raised when a user is linked or unlinked from an associated Person Participant. For more information, see the related Javadoc information in the <code>WorkspaceServices</code> component.

Related information

[Cúram Server Developer](#)

PasswordReuseStrategy API

Use the `curam.citizenworkspace.security.impl.PasswordReuseStrategy` API to add your own password change validations.

As part of the password reset function, there is a default validation that prevents a user from entering a new password that is the same as the user's current password. Using the `PasswordReuseStrategy` API, custom validations can be added to restrict users from changing their passwords to current or previous values if required. For example, a customer might want to implement a password reuse strategy that prevents users from reusing a previous password until after six password changes.

For further details, see the API Javadoc.

CitizenWorkspaceAccountManager API

Use the `curam.citizenworkspace.security.impl.CitizenWorkspaceAccountManager` API to create and link citizen accounts. Use the API to build out custom functionality to support caseworkers who want to link accounts and create accounts on behalf of the citizen.

The API offers the following methods:

- Creating standard accounts
- Creating linked accounts
- Removing links between participants and accounts.
- Retrieving account information

For more information, see the API Javadoc.

Data caching

Minimize the risk of citizens accessing each others' data from browser and server data caches. Cached data can be accessed when citizens use the browser back button or browser history to retrieve data entered by other users, or when PDF files are cached locally on the computer that was used to make the application.

Server caching

HTTP servers like Apache can set cache-control response headers to not store a cache. Use this approach to prevent access to data using the browser back button or history.

Browser caching

Browsers can be configured not to cache content. If citizens can access the web portal in a "kiosk", then the browser should be configured never to cache content.

Advise citizens to clear their cache and close all browser windows they have used when they are finished using the web portal. Also tell citizens to remove PDF documents that they download from the browser's temporary internet files.

React environment variable reference

A full list of Universal Access React environment variables categorized by function. You can set environmental variables in `.env` files in the root directory of your application. If you omit environment variables, either they are not set or the default values apply.

The starter pack provides the `.env` and the `.env.development` files to get you started. For more information about using `.env` files in react-scripts, see *Adding Development Environment Variables In .env* in the [Create React App documentation](#).

- [“REST API” on page 173](#)
- [“Locale” on page 174](#)
- [“Feature toggles” on page 174](#)
- [“Connectivity handler” on page 175](#)
- [“User account status polling” on page 175](#)
- [“Document uploads” on page 175](#)
- [“Social Program Management Web Development Accelerator” on page 176](#)
- [“Application authentication” on page 177](#)
- [“Simple authentication for development” on page 177](#)
- [“Single sign-on \(SSO\) authentication” on page 177](#)
- [“User session” on page 179](#)
- [“Intelligent Evidence Gathering \(IEG\)” on page 179](#)

REST API

REACT_APP_REST_URL

Specifies the path to REST services. You must set this variable as it is needed by the Authentication service. When you specify a path, it can be a URL to a server, or a relative path in the local deployment server if you are using a proxy. For the Universal Access application, it is `http{s}://<ServerHostName>:<Port>/Rest`. For example:

```
REACT_APP_REST_URL=https://192.0.2.4:9044/Rest
```

Where `<ServerHostName>` and `<Port>` are the hostname and port number of the server where the REST services are deployed.

This variable is also used by the default Redux modules and modules that are generated by IBM Social Program Management Web Development Accelerator to call REST APIs. For example:

```
RestService.get(`${REACT_APP_REST_URL}/v1/users)
```

For development with the mock server, you can use local host without `/Rest`. For example:

```
REACT_APP_REST_URL=http://localhost:3080
```

For more information, see [“The mock server API service” on page 91](#).

MOCK_SERVER_PORT

Specifies the port to serve mock APIs. For example:

```
MOCK_SERVER_PORT=3080
```

For more information, see [“The mock server API service” on page 91](#).

REACT_APP_RESPONSE_TIMEOUT

Specifies the maximum time in seconds to wait for the first byte to arrive from the server, by default 10, but does not limit how long the entire download can take. Set the response timeout to be a few seconds longer than the actual time it takes the server to respond. The lengthened response allows for time to make DNS lookups, TCP/IP, and TLS connections. For example:

```
REACT_APP_RESPONSE_TIMEOUT=10
```

For more information, see [“The RESTService utility” on page 93](#).

REACT_APP_RESPONSE_DEADLINE

Specifies the maximum time in seconds for the entire request, including all redirects, to complete. If the response is not fully downloaded within REACT_APP_RESPONSE_DEADLINE, the request is canceled. The default value is 60. For example:

```
REACT_APP_RESPONSE_DEADLINE=60
```

For more information, see [“The RESTService utility” on page 93](#).

REACT_APP_DELAY_REST_API

(Development only) Specifies a time in seconds to simulate a delay in the response from the API. For example:

```
REACT_APP_DELAY_REST_API=2
```

The value can be set to any positive integer to adjust the delay. For more information, see [“The RESTService utility” on page 93](#).

Locale

REACT_APP_INTL_LOCALE

Specifies a locale to set the correct regional format for dates and numbers in the application. The value must align with the `curam.environment.default.locale` value that is set in your regional settings on the server, see [The Application.prx file](#).

The format of the locale is `xx-XX`, for example. `en-US`, rather than `en_US`, which is the format on the server. For example, to set the US locale:

```
REACT_APP_INTL_LOCALE=en-US
```

Feature toggles

You can enable the display of specific features in the application.

REACT_APP_FEATURE_LIFE_EVENTS_ENABLED

Specifies whether to display the Life Events feature in the application with a Boolean value. It is enabled by default. For example, to enable Life Events:

```
REACT_APP_FEATURE_LIFE_EVENTS_ENABLED=true
```

For more information, see [“Enabling and disabling life events” on page 214](#).

REACT_APP_FEATURE_APEALS_ENABLED

Specifies whether to display the Appeals feature in the application with a Boolean value. It is disabled by default. For example:

```
REACT_APP_FEATURE_APEALS_ENABLED=false
```

For more information, see [“Enabling and disabling appeals” on page 238](#).

REACT_APP_FEATURE_VERIFICATIONS_ENABLED

Specifies whether to display the Citizen Verifications feature in the application with a Boolean value. It is disabled by default. For example, to enable Citizen Verifications:

```
REACT_APP_FEATURE_VERIFICATIONS_ENABLED=true
```

For more information, see [“Enabling or disabling verifications” on page 231](#).

Connectivity handler

REACT_APP_CONNECTIVITY_URL

Specifies a URL to ping to check internet connectivity. By default, the REST API server is used.

```
REACT_APP_CONNECTIVITY_URL=<REACT_APP_REST_URL>/v1/ua/system_configurations
```

REACT_APP_CONNECTIVITY_INTERVAL

Specifies the interval between pings to check internet connectivity. By default, a connectivity check request is sent to the server URL every 5000 milliseconds.

```
REACT_APP_CONNECTIVITY_INTERVAL=5000
```

For more information, see [“Implementing a connectivity handler” on page 86](#).

User account status polling

To check whether a user has a standard or linked account when they submit an application, you can poll the user's account type to check for updates to their account status. This feature can be useful when an application is set up to automatically create linked accounts, such as in test or demonstration environments.

REACT_APP_USER_ACCOUNT_POLLING

For example:

```
REACT_APP_USER_ACCOUNT_POLLING={“api”: “/v1/ua/user_account_login”, “timeout”: “0”,  
“interval”: “1000”}
```

Where:

- **api** Specifies a URL to ping to check the user account type. By default, `/v1/ua/user_account_login`.
- **timeout** Specifies the timeout in milliseconds for the user account type ping duration. By default, the timeout is set to 0, which disables user account type polling. Five seconds is a sensible period to allow for asynchronous processing to finish while not polling indefinitely.
- **interval** Specifies the interval in milliseconds between pings to check the user account type. By default, 1 second.

Document uploads

You can specify the allowed file formats and maximum size for the documents that users can upload.

REACT_APP_DOC_UPLOAD_FILE_FORMATS

Specifies the file name extension, including the dot separator, of the allowed file types for document uploads in a comma-separated list. By default, if you do not set this environment variable, the allowed file types are JPG, JPEG, PNG, TIFF, and PDF. To change the default file types, set this environment variable. For example:

```
REACT_APP_DOC_UPLOAD_FILE_FORMATS=".png, .jpg, .pdf"
```

If you specify an invalid file extension string, all file types are denied.

For more information, see [“Customizing file formats and size limits for file uploads” on page 232](#).

REACT_APP_DOC_UPLOAD_SIZE_LIMIT

Specifies the maximum size limit for uploaded documents. By default, the maximum file size is 5 MB. To change the default file size, set this environment variable. For example:

```
REACT_APP_DOC_UPLOAD_SIZE_LIMIT=6
```

For more information, see [“Customizing file formats and size limits for file uploads” on page 232](#).

REACT_APP_DOC_UPLOAD_LEAD_DAYS

Specifies a lead time in days to subtract from due dates to give caseworkers time to process applications before the actual due dates. This earlier date is then displayed to citizens in the application.

The default value of REACT_APP_DOC_UPLOAD_LEAD_DAYS is 0 days. The value that you set is converted to its absolute value and subtracted from the verification due date. For example, -1 and 1 have the same effect.

For example:

```
REACT_APP_DOC_UPLOAD_LEAD_DAYS=-7
```

For more information, see [“Customizing a file upload lead time for verifications” on page 232](#).

Social Program Management Web Development Accelerator

For more information, see [“Generating Universal Access Redux modules” on page 83](#).

WDA_MODULES_OUTPUT

(Development only) Specifies the directory to place module files generated by the IBM Social Program Management Web Development Accelerator, by default `src/modules/generated`. For example:

```
WDA_MODULES_OUTPUT=src/modules/generated
```

WDA_MODULES_CONFIG

(Development only) Specifies a JSON file in which to save the module configuration that you define, by default `modules_config.json`. This file contains the metadata that is used to generate the code. For example:

```
WDA_MODULES_CONFIG=src/modules/modules_config.json
```

It is recommended that you add only this file to source control.

WDA_SPM_SWAGGER

(Development only) Specifies the location of a copy of the IBM Cúram Social Program Management Swagger specification that defines which REST APIs are available to the Social Program Management Web Development Accelerator. For example:

```
WDA_SPM_SWAGGER=spm_swagger.json
```

You can copy this file from a running IBM Cúram Social Program Management instance at `http://hostname:port/Rest/api/definitions/v1`.

Application authentication

The default implementation for authentication is a Java Authentication and Authorization Service (JAAS) authentication method. If the JAAS authentication method does not suit, you can change to another authentication method, such as Single sign-on (SSO). Ensure that you set any related environmental variables where needed. For more information, see [“Authentication” on page 151](#).

The following authentication methods are provided:

REACT_APP_AUTH_METHOD

- **JAASAuthentication**
(Default) No further environmental variables needed.
- **DevAuthentication**
(Development only) Specifies simple authentication during development that bypasses proper authentication (JAAS or SSO) and accepts the username dev without any password. The login process can run and allows access to the 'user account' password protected pages. If you specify simple authentication, you can set the optional user type environmental variable in [“Simple authentication for development” on page 177](#).
- **SSOSPAAuthentication or SSOIDPAuthentication**
Specifies service-provider (SP)-initiated or identity provider (IdP)-initiated SAML 2.0 web SSO. If you set SSO authentication, you must set the related SSO environmental variables in [“Single sign-on \(SSO\) authentication” on page 177](#).

For example:

```
REACT_APP_AUTH_METHOD=SSOIDPAuthentication
```

Simple authentication for development

(Development only) If you are using simple authentication for development, you can set the following environmental variable. For more information, see [“Customizing the authentication method” on page 153](#).

REACT_APP_SIMPLE_AUTH_USER_TYPE

(Development only) Specifies a user type during development so you can test functionality for those users.

- **PUBLIC**, a public citizen account user.
- **GENERATED**, an anonymous generated account user.
- **STANDARD**, a standard registered account user.
- **LINKED**, a linked account user.
- **null**, no user type.

For more information about user types, see [“User account types” on page 150](#).

For example, to test the application for a linked user:

```
REACT_APP_SIMPLE_AUTH_USER_TYPE=LINKED
```

Single sign-on (SSO) authentication

If you are using SSO authentication, you must set the following environmental variables. For more information, see [“Customizing the authentication method” on page 153](#) and [“Configuring single sign-on properties” on page 158](#).

- The <IdP_URL> consists of three parts: the HTTPS protocol, the IdP hostname or IP address, and the listener port number. For example, `https://192.168.0.1:12443`.
- The <ACS_URL> consists of three parts: the HTTPS protocol, the Assertion Consumer Service (ACS) hostname or IP address, and the listener port number. For example, `https://192.168.0.2:443`.

REACT_APP_SAMLSSO_ENABLED

Specifies whether SSO authentication is used in the application. By default, the IdP-initiated flow of the SAML SSO browser profile is used. A Boolean value is accepted. For example, to handle the SAML SSO browser profile in the application:

```
REACT_APP_SAMLSSO_ENABLED=true
```

REACT_APP_SAMLSSO_SP_MODE

(SP-initiated flow only) Specifies whether to use the SP-initiated flow of the SAML SSO Browser profile. By default, the default IdP-initiated flow of the SAML SSO Browser profile and this setting overrides it. A Boolean value is accepted. For example:

```
REACT_APP_SAMLSSO_SP_MODE=true
```

REACT_APP_SAMLSSO_USERLOGIN_URL

Specifies the IdP login page URL, that is, the URL where the application sends the user login credentials. For example:

```
REACT_APP_SAMLSSO_USERLOGIN_URL=<IdP_URL>/pkmslogin.form
```

REACT_APP_SAMLSSO_SP_ACS_URL

Specifies the ACS application server URL, that is, the service provider URL where the application sends the SAML response. For example:

```
REACT_APP_SAMLSSO_SP_ACS_URL=<ACS_URL>/samlsp/acs
```

REACT_APP_SAMLSSO_USERLOGOUT_URL

Specifies the IdP logout page URL, that is, the URL where the application sends the user logout request. For example:

```
REACT_APP_SAMLSSO_USERLOGOUT_URL=<IdP_URL>/pkmslogout
```

REACT_APP_SAMLSSO_IDP_LOGININITIAL_URL

(IdP-initiated flow only) Specifies the initial URL to which the application sends the initial login request to the identity provider. Refer to the identity provider documentation for the correct URL and values. For example:

```
REACT_APP_SAMLSSO_IDP_LOGININITIAL_URL=<IdP_URL>/isam/sps/saml20idp/saml20/logininitial?
RequestBinding=
HTTPPost&PartnerId=<ACS_URL>/samlsp/acs&NameIdFormat=Email
```

REACT_APP_SAMLSSO_IDP_SSOLOGIN_URL

(SP-initiated flow only) Specifies the identity provider URL where the application sends the SAML request. Refer to the identity provider documentation for the URL. For example

```
REACT_APP_SAMLSSO_IDP_SSOLOGIN_URL=<IdP_URL>/isam/sps/saml20idp/saml20/login
```

User session

REACT_APP_SESSION_INACTIVITY_TIMEOUT

Specifies the time in seconds before a user session expires. The value must match the session timeout that is configured on the server, by default, 30 minutes, or 1800 seconds.

```
REACT_APP_SESSION_INACTIVITY_TIMEOUT=1800
```

For more information, see [“Configuring user session timeout” on page 55](#).

REACT_APP_SESSION_PING_INTERVAL

Specifies the time in seconds between each time that the user’s current session is checked to see whether they are actively using the application or not. By default, the value is 60. For example:

```
REACT_APP_SESSION_PING_INTERVAL=60
```

Intelligent Evidence Gathering (IEG)

For more information, see Chapter 6, [“Customizing IEG forms in the Universal Access Responsive Web Application,” on page 181](#).

REACT_APP_DISPLAY_REQUIRED_LABEL

Specifies whether to indicate the required form fields or the optional form fields. As most questions in a typical form are required, indicating the optional questions rather than the required questions typically results in a less cluttered form. By default, optional fields are highlighted in IEG forms. For example, to display labels for required fields only:

```
REACT_APP_DISPLAY_REQUIRED_LABEL=true
```

REACT_APP_DATE_FORMAT

Specifies the date format for form fields, by default, MM/DD/YYYY. The valid values are dd-mm-yyyy and mm-dd-yyyy. If you omit the environment variable or set an invalid value, the default date format is used. For example, to change the date format to DD/MM/YYYY:

```
REACT_APP_DATE_FORMAT=dd-mm-yyyy
```

REACT_APP_PHONE_MASK_FORMAT

Specifies a phone number mask for a form field in a question. The value must be in ISO 3166-1 alpha-2 code format, for example, US | CA | GB | DE. In your IEG script, you must add the `wds-js-input-mask-phone` class name to the question.

```
REACT_APP_PHONE_MASK_FORMAT=US
```

Where country is the locale that you want to use.

REACT_APP_PHONE_MASK_DELIMITER

Specifies a custom delimiter for phone numbers. For example, to convert 1 636 5600 5600 to 1-636-5600-5600:

```
REACT_APP_PHONE_MASK_DELIMITER=-
```

REACT_APP_PHONE_MASK_LEFT_ADDON

Specifies a fixed country code for phone number fields. For example, to convert 1-636-5600-5600 to +1-636-5600-5600:

```
REACT_APP_PHONE_MASK_LEFT_ADDON=+
```

REACT_APP_CURRENCY_MASK_LEFT_ADDON

Specifies a currency symbol to display before the amount. If you omit this value, US dollars are displayed by default. For example, to specify Euro:

```
REACT_APP_CURRENCY_MASK_LEFT_ADDON=$
```

REACT_APP_CURRENCY_MASK_RIGHT_ADDON

Specifies a currency symbol to display after the amount. If both left and right values are set, left takes precedence. For example, to specify Euro for Luxemburg:

```
REACT_APP_CURRENCY_MASK_RIGHT_ADDON=€
```

Chapter 6. Customizing IEG forms in the Universal Access Responsive Web Application

Universal Access can use forms to gather information about citizens, such as applying for benefits or screening for programs. Where you need to save customer data as evidence, you can create forms in Intelligent Evidence Gathering (IEG). IEG is a framework for creating dynamic and conditional questionnaires and saving the input data as evidence. You can customize IEG forms for your organization in the Universal Access Responsive Web Application. The default connectivity handling helps to prevent citizens losing data in IEG forms by preventing them from leaving pages with unsaved changes.

Before you begin

If you are not familiar with IEG, you must familiarize yourself with how to author IEG scripts and include them in the application. For more information about IEG, see [Authoring Intelligent Evidence Gathering scripts](#) and [Working with Intelligent Evidence Gathering](#).

About this task

Universal Access forms that gather data as evidence are implemented in IEG, as in the classic Universal Access citizen application. However, forms are now rendered in the browser by IEG React components from the design system, which replace the IEG player, and in some cases, the IEG behavior has changed.

Due to the technology and user interface changes, your existing IEG scripts must be tested before use, and in most cases, at least some minor changes are needed for existing scripts to work in the new application.

For the best user experience, always disable the **Back** button on the first page of IEG forms. The **Back** button goes back one page in the script, not in the application, so you don't need one on the first page.

For more information about data loss prevention in IEG, see [“Connectivity handling” on page 85](#).

Related tasks

[Configuring appeal requests](#)

Complete the following steps to enable a citizen to request an appeal from their citizen account.

IEG elements and attributes specific to the design system and Universal Access Responsive Web Application

The following IEG elements and attributes apply to the design system and Universal Access Responsive Web Application only.

Display elements and attributes

- The `combo-box` element, which is a child element of the `question` element.
- The `explainer` element, which is a child element of the `cluster`, `question-page`, and `relationship-page` elements.
- The `hint-text` element, which is a child element of the `container`, `list-question`, and `question` elements.
- The `next-button-label` element, which is a child element of the `question-page`, `relationship-page`, and `summary-page` elements.
- The `relationship-detail-header` element, which is a child element of the `relationship-summary-list` element.

Display element attributes

- The `grouping-id` attribute of the `cluster` element.

Meta-display elements

- The `class-names` element, which is a child element of the layout element.
- The `date-picker` value for the type child element of the layout element.

For more information about IEG elements, see the [IEG script element reference](#).

IEG configuration not currently supported for the Universal Access Responsive Web Application

The following IEG configuration is not currently supported by the design system and the Universal Access Universal Access Responsive Web Application.

Question matrices

Question matrices display a list of questions that are based on a code table and, for each of the code table values and each entity, a check box is displayed for you to select the values that apply to a particular entity.

Three-field date picker

The three-field date picker is no longer supported. Dates either default to a single-field date input field or can be configured with a date picker component by using the layout element.

Grouping individual question help at cluster level

Cluster-level help is supported, however, the `compile.cluster.help` property, which groups the help text for each of the questions in a cluster into the cluster help panel is not supported.

Display elements and attributes

- The `custom-output` element, which renders custom HTML on summary pages only.
- The `show-page-elements` attribute on the `edit-link` element for editing specific clusters.
- The `footer-field` element, which displays values that are calculated from expressions in the `footer-row` element of a list.
- The `footer-row` element, which adds an extra row at the end of a list to display total or summary information.
- The `help-text` element, which displays help text, is not supported for pages.
- The `label-alignment` element, which is used in the layout element for a cluster to control the text alignment of the labels in the cluster.
- The `label-width` element, which is used in the layout element for a cluster to control the width of the labels in the cluster.
- The `num-cols` element, which is used in the layout element for a cluster to control the number of columns in the cluster.
- The `type` element, which is used in the layout element for a cluster to control the layout of labels in relation to input controls.
- The `width` element, which is used in the layout element for a cluster to control the width of the cluster on the page.
- The `legislation` element, which creates legislation links at page and question level to point to relevant legislative information.
- The `policy` element, which creates policy links at page and question level to point to relevant policy information
- The `skip-field` element, which enables a more flexible layout of elements within clusters or footer rows in lists where no visible display element is needed.
- The `row-help` element, which specifies help for rows in a list.
- The `set-focus` attribute of the `question-page` element, which sets focus for a page.

Meta-display elements

- The `codetable-hierarchy-layout` element, which is used in questions with a code table hierarchy type to control different aspects of the layout.

Structural, administrative, and other elements and attributes

- The `hide-for-control-question` attribute on the `ieg-script` element, which hides the label and value of control questions for loops when the loop is entered.
- The `highlight-validation` attribute on the `ieg-script` element. Validations are now always displayed with the failing input field.
- The `show-progress-bar` attribute on the `ieg-script` element. Progress through sections is now indicated by text and the section title. For example, **STEP 2 OF 4 • HOUSEHOLD**.

For more information about IEG elements, see the [IEG script element reference](#).

Configuring section navigation for forms

If you are developing scripts in IEG, you can enable section navigation to guide people through forms.

About this task

You can use section navigation on any forms, but it is particularly useful for longer forms. If you enable section navigation, it is a good idea to use section summary pages so that users can review their changes regularly.

Procedure

In your IEG script, add the `show-section` element to the `ieg-script` element.

Configuring progress information for forms

If you are developing pages in IEG, you can show progress text and the section title so citizens can see where they are in the script, for example, **STEP 2 OF 4 • HOUSEHOLD**.

Add the following IEG configuration property to the `ieg-config.properties` file to configure the text. The section title is added automatically.

```
# Text progress bar indicator
progress.bar.indicator.text=Step %1s of %2s
```

Where `%1s` is the current step number and the `%2s` is the total number of steps on the script. The message is calculated based on the total number of sections and the current section.

The `IEGPageMetadata(JSON)` component contains all of the metadata for each IEG form. The text progress indicator is displayed if `IEGPageMetadata` finds the `metadata['ieg-config']['progress-indicator']` element in the JSON.

Configuring dynamic titles on forms

If you are developing pages in IEG, you can configure the relationship pages with more relevant titles that are based on the user's responses.

The relationship page title accepts an ICU message template.

For more information about the ICU messaging format, see <http://icu-project.org/apiref/icu4j/com/ibm/icu/text/MessageFormat.html>. Page titles and subtitles accept a specific formatting syntax based on ICU. It should be used in loops and will give more context to the users.

These six keywords are defined:

- index
- innerIndex
- outerIndex
- ordinal
- innerOrdinal
- outerOrdinal

You can use `index` and `ordinal` in simple non-nested loops. If they are used in a nested loop, it is synonymous to `outerIndex` and `outerOrdinal`.

Refer to these examples.

"Add {ordinal} member" displays **Add first member, Add second member, ...**

"Add the {innerOrdinal} income for the {outerOrdinal} member" displays **Add the first income for the first member ...**

"{index, select, 0 {Add your {innerOrdinal} income} other {Add %1s's {innerOrdinal} income}}" displays **Add your first income** or **Add Jane's first income** depending on the value of `index` (this is equal to `ordinal - 1`).

"Ajouter la {ordinal}#feminine# personne" displays **Ajouter la première personne.**

"Ajouter la {innerOrdinal}#feminine# recette du {outerOrdinal}#%spellout-ordinal-masculine# membre" displays **Ajouter la première recette du premier membre.**

You can define the title as follows:

```
{index, select, 0 {Your relationships} other {{personName}'s relationships}}
```

The outcome of this message template on the first relationship question page is **Your relationships**. On the following relationship question pages, it shows **[personName]'s relationships**. The reserved word `personName` displays the person's first name on the title of the page.

Configuring rich text on forms

You can configure rich text to display with a number of IEG display elements in IEG forms. You can also configure external links in rich text to open in a new tab or window.

About this task

Rich text is supported in the following IEG display elements that support text:

- `cluster` title, help, and description
- `container` title, help, and description
- `display-text`
- `divider`
- `list` title, help, and description
- question label and help
- `subtitle`

For more information about IEG elements, see [Display elements](#).

Configuring external links to open in a new tab or window

You can configure external links to open in a new tab or window in IEG forms. By default, links open in the current tab.

About this task

For security reasons, HTML in rich text is sanitized to remove certain attributes before display, including the HTML target attribute. You must configure the rich text to leave the target attribute in the sanitized content so that the link opens in a new tab or window.

For example, the `my link` link in rich text opens in the current tab as intended. The `my link` link is intended to open in a separate tab or window. However, because the rich text is sanitized with DOMPurify before display, the target attribute is removed and the link opens in the current tab by default.

To configure DOMPurify to leave specific attributes, you must add `dompurify` to the dependencies and specify a DOMPurify persistent configuration in any JavaScript or JSX code that runs when the app is loaded. For example, `App.js`. For more information about DOMPurify, see <https://github.com/cure53/DOMPurify#persistent-configuration>.

Only one active configuration at a time is allowed. After you set the configuration, any extra configuration parameters that are passed to `DOMPurify.sanitize` are ignored. The DOMPurify configuration persists until the next call to `DOMPurify.setConfig`, or until `DOMPurify.clearConfig` is called to reset it.

Procedure

1. Add `dompurify` to the dependencies in the `package.json` file.

```
npm install dompurify
```

2. To configure DOMPurify to leave the target attribute, specify the following DOMPurify persistent configuration in any JavaScript or JSX code that runs when the app is loaded.

```
import DOMPurify from 'dompurify';
DOMPurify.setConfig({ ADD_ATTR: ['target'] });
```

Configuring hint text for forms

You can use short sentences of hint text to explain the expected input format or content in IEG forms. For example, you can explain the expected format for a telephone number.

About this task

Hint text is suitable for short sentences and does not support HTML tags. If you want to add more text or format text with HTML tags, use the `help-text` or `explainer` elements instead. For more information, see [explainer](#), [hint-text](#), and [help-text](#).

Hint text is suitable for short sentences and does not support HTML tags. If you want to add more text or format text with HTML tags, use the `help-text` element instead. For more information, see [hint-text](#) and [help-text](#).

Procedure

In your IEG script, you can add the `hint-text` element to any container, question or list-question element.

For example:

- Container

```
<container show-container-help="true">
  <title id="primaryPhoneNumber">primaryPhoneNumber</title>
```

```

    <hint-text id="PhoneNumber.Hint">PhoneNumber.Hint</hint-text>
    <help-text id="PhoneNumber.Help">Telephone number must only contain numbers, parentheses,
or dashes and be 10 digits. For example, (212) 555-0010 or 2125550010.</help-text>
    <question id="primaryPhoneType" mandatory="true">
        <help-text id="PhoneNumber.Help">Telephone number must only contain numbers,
parentheses, or dashes and be 10 digits. For example, (212) 555-0010 or 2125550010.</help-
text>
        <label id="PrimaryPhoneType.Label">Primary Phone Type</label>
    </question>
</container>

```

- Question

```

<question id="firstName" mandatory="true">
    <hint-text id="FirstName.Hint">FirstName.Hint</hint-text>
    <label id="FirstName.Label">First Name</label>
</question>

```

- List question

```

<list-question entity="Person" id="currentlyWorking" mandatory="false">
    <label id="CurrentlyWorking.Label">Please select the people that have a job:</label>
    <hint-text id="CurrentlyWorking.Hint">CurrentlyWorking.Hint</hint-text>
    <item-label>
        <label-element attribute-id="firstName" />
    </item-label>
</list-question>

```

Configuring explainer text for forms

You can use the `explainer` element to provide extra text in IEG forms that is initially hidden and that can be expanded to show further explanation. For example, you can provide background information that a user can choose to expand only if needed.

About this task

You can use the `explainer` element to provide a large amount of text without cluttering up the form. For more information about showing text in IEG forms, see [explainer](#), [hint-text](#), and [help-text](#).

Procedure

In your IEG script, add the `explainer` element to any `cluster`, `question-page`, or `relationship-page` element.

For example:

- cluster

```

<cluster>
    <explainer>
        <title id="ExplainerCluster.Title">Why do we ask for your Social Security Number?</title>
        <description id="Explainer.Description">Your Social Security Number ensures that
your application is unique to you and reduces processing time.</description>
    </explainer>
    <question control-question="false" id="isSSN" mandatory="true" multi-select="false"
show-field-help="false">
        <label id="IsSSN.Label">What is your Social Security Number?</label>
    </question>
</cluster>

```

- question-page

```

<question-page>
    <explainer>
        <title id="ExplainerSSN.Title">Why do we ask for your Social Security Number?</title>
        <description id="ExplainerSSN.Description">Your Social Security Number ensures that your
application is unique to you and reduces processing time.</description>
    </explainer>
    <question id="isSSN" mandatory="true" multi-select="false">
        <label id="IsSSN.Label">What is your Social Security Number?</label>
    </question>
</question-page>

```

```
</explainer>
</question-page>
```

- relationship-page

```
<relationship-page>
<explainer>
  <title id="ExplainerSSN.Title">Why do we ask for your Social Security Number?</title>
  <description id="ExplainerSSN.Description">Your Social Security Number ensures that your
application is unique to you and reduces processing time.</description>
</explainer>
</relationship-page>
```

Configuring the 'Help' label for forms

You can change or remove the 'Help' label from the help icon for input controls in the application by overriding the default text. To remove the label, override the default text with a single space character in a custom messages file.

Procedure

1. To remove the 'Help' label, create a `src/locale/messages_en.json` messages file with a single space character as the value for the help label message ID, `WidgetHelp_helpToggleText`.

```
{
  "WidgetHelp_helpToggleText": " "
}
```

2. Update the `src/config/intl.config.js` file in the English locale to point to the custom messages file.

```
// [...]
{
  locale: 'en',
  displayName: 'English',
  localeData: require('react-intl/locale-data/en'),
  messages: require('../locale/messages_en'),
},
// [...]
```

3. Rebuild and deploy the application to see your changes.

Related tasks

[Changing text in the application](#)

You can change the default text in the application by providing custom text that overrides the default text for any language. In this scenario, an English language message is changed.

Configuring required or optional labels for form fields

You can choose whether to indicate the required fields or the optional fields in IEG forms. As the majority of questions in a typical form should be required, indicating the optional questions rather than the required questions typically results in a less cluttered form. By default, optional fields are highlighted in IEG forms.

About this task

By default, fields that are not configured as required in the IEG script are labeled as **Optional** and required fields are not labeled. If you choose to indicate required fields instead, fields that are configured as required in the script are labeled **Required** and optional fields are not labeled.

Procedure

Show labels for required questions only by adding the `REACT_APP_DISPLAY_REQUIRED_LABEL` environment variable to your `.env` file with a value of `true`.

For example:

```
REACT_APP_DISPLAY_REQUIRED_LABEL=true
```

Configuring input formats and constraints for form fields

You can customize field inputs and constraints for phone numbers, social security numbers (SSN), dates, currencies, and percentages on IEG forms. You can also adjust the width of form fields to match the length of the expected input, and choose to use a date picker for dates where appropriate.

About this task

To avoid confusion about expected inputs, always match the width of form fields to the expected input. For example, use a form field that matches the length of the SSN.

Masked input fields increase input field readability by formatting or constraining typed data. You can apply input masks with the IEG `class-names` element, which is a child element of the `layout` element. The `class-names` element adds the content of the element to the HTML that is generated for the component, this element accepts multiple values that are separated by a space.

For date questions, in addition to the masked input, you can choose to add a date picker for dates by setting the value of the `type` child element of the `layout` element to `date-picker`. For those questions, you can then use the calendar or type the date. By default, date questions are displayed with the masked input field if no `layout type` is specified.

For more information about the `layout` element, see [layout](#).

If the class name matches any of the reserved input mask class names, that class name is applied to the HTML control input. If the class name does not match a reserved input mask class name, the class name is applied to the `<div>` element that contains the HTML element (cluster, question, or list-question). You can use the following design system CSS classes as input masks to format and constrain input values for questions:

- `wds-js-input-mask-currency`

Masks input for currency questions. The character limit is 21 characters. You can also set optional environmental variables for currency symbols, see [“Configuring currency symbols” on page 190](#).

- `wds-js-input-mask-ssn`

Masks input for social security numbers.

- `wds-js-input-mask-percentage`

Masks input for percentage characters.

- `wds-js-input-mask-phone`

Masks input for phone number fields according to the defined locale for the application. Configuring the phone number input mask requires some additional steps and you can also set optional environmental variables for delimiters and country codes, see [“Configuring phone numbers” on page 189](#).

- `wds-js-input-layout-size-field_size`

Adjusts the width of form fields to match the length of the expected input. Where *field_size* is one of the following sizes:

x-small

Use for 2 - 3 characters, such as DD, MM, or title.

small

Use for 4 - 6 characters, such as ZIP code, postal code, or CVV number.

medium

Use for around 8 characters, such as SSN or DD/MM/YYYY.

large

Use for around 16 characters, such as credit card numbers.

x-large

Use for around 24 characters, such as email addresses.

Procedure

In your IEG script, add the appropriate CSS classes to the question. For example:

```
<question id="ssn" mandatory="true">
  <label id="SSN.Label">SSN</label>
  <layout>
    <class-names>custom-css-class1 wds-js-input-mask-ssn wds-js-input-layout-size--medium
    </class-names>
  </layout>
</question>
```

Configuring phone numbers

You can configure an input mask class name to format phone number fields in IEG forms according to the defined locale for the application. You can also configure a phone number delimiter or a country prefix if needed.

Procedure

1. Add `cleave.js` as a dependency in your `package.json` file.

```
"cleave.js": "<version>"
```

Where *version* is the version that you want to use.

2. Import the region-specific `.js` file in your initializing `.js` file.

For example:

```
import 'cleave.js/dist/addons/cleave-phone.[country]';
```

Where *country* is the locale that you want to use.

3. Add a `REACT_APP_PHONE_MASK_FORMAT` environment variable to your `.env` file.

```
REACT_APP_PHONE_MASK_FORMAT=[country]
```

Where *country* is the locale that you want to use.

4. In your IEG script, add the `wds-js-input-mask-phone` class name to the question. For example:

```
<question id="primaryPhoneNumber" mandatory="true" show-field-help="true">
  <layout>
    <class-names>wds-js-input-mask-phone</class-names>
  </layout> <label id="PrimaryPhoneNumber.Label">Primary Phone Number</label>
</question>
```

5. Optional: You can set a custom delimiter for phone numbers by adding the `REACT_APP_PHONE_MASK_DELIMITER` environment variable to your `.env` file. For example, to convert 1 636 5600 5600 to 1-636-5600-5600, set the environment variable as follows:

```
REACT_APP_PHONE_MASK_DELIMITER=-
```

6. Optional: You can set a fixed country code for phone numbers by adding the `REACT_APP_PHONE_MASK_LEFT_ADDON` environment variable to your `.env` file. For example, to convert 1-636-5600-5600 to +1-636-5600-5600, set the environment variable as follows:

```
REACT_APP_PHONE_MASK_LEFT_ADDON=+
```

Configuring date formats

You can configure the date format in IEG forms by setting the `REACT_APP_DATE_FORMAT` environment variable.

About this task

By default, the date format is `MM/DD/YYYY` if you do not set a value for the `REACT_APP_DATE_FORMAT` environment variable.

The valid values are:

```
dd-mm-yyyy  
mm-dd-yyyy
```

If you set an invalid value, the default date format is used.

Procedure

Change the date format by adding the `REACT_APP_DATE_FORMAT` environment variable to your `.env` file. For example, to change the date format to `DD/MM/YYYY`, set the environment variable as follows:

```
REACT_APP_DATE_FORMAT=dd-mm-yyyy
```

Configuring currency symbols

You can configure currency symbols for currency fields in IEG forms by setting the `REACT_APP_CURRENCY_MASK_LEFT_ADDON` or `REACT_APP_CURRENCY_MASK_RIGHT_ADDON` environment variables.

About this task

Use the appropriate variable to set the currency symbol either before or after the value. If both environment variables are set, `REACT_APP_CURRENCY_MASK_LEFT_ADDON` takes precedence.

Procedure

Add a currency symbol for currency fields by adding the `REACT_APP_CURRENCY_MASK_LEFT_ADDON` or `REACT_APP_CURRENCY_MASK_RIGHT_ADDON` environment variables to your `.env` file. For example, to set the currency symbol for US dollars, set the environment variable as follows:

```
REACT_APP_CURRENCY_MASK_LEFT_ADDON=$
```

Configuring code-table hierarchies for form fields

You can use code-table hierarchies to add two related questions in IEG forms. When you answer the first question, the second question is enabled.

About this task

Any question where the data type is defined as a code table hierarchy is displayed as two separate questions in vertically aligned drop-down menus. The first question menu corresponds to the root code table in the hierarchy, and displays the label that is specified for the question. The second question menu corresponds to the second-level code table in the hierarchy, and displays a label that corresponds to the code table display name. The second menu is disabled until a selection is made in the first menu. Summary pages display both questions.

Displaying a code-table hierarchy value in a list, or the `codetable-hierarchy-layout` options, are not supported.

Procedure

To ensure that the label is displayed correctly for the second question, you must ensure that, for each code table name element, there is a corresponding `locale` element within the `displaynames` element in your code-table definition.

For example, see the following code-table definition.

```
<codetables package="curam.codetable" hierarchy_name="CountyCityHierarchy">
  <!-- Parent codetable - County -->
  <codetable java_identifier="COUNTYCODE" name="CountyCode">
    <displaynames>
      <name language="en">County</name>
      <locale language="en">County</name>
    </displaynames>
    <!-- code items... -->
  </codetable>
  <!-- Child codetable - City -->
  <codetable java_identifier="CITYCODE" name="CityCode" parent_codetable="CountyCode">
    <displaynames>
      <name language="en">City</name>
      <locale language="en">City</name>
    </displaynames>
    <!-- code items... -->
  </codetable>
</codetables>
```

Implementing a combo box for form fields

You can implement a combo box question with an auto-complete search function to help you to complete form fields in IEG forms as you type. For example, known address fields can be automatically selected when you enter an address. You can implement the option to add new items if they are not found, for example, add an address.

About this task

You must implement a search function in the Universal Access Responsive Web Application and register the search function with `IEGRegistry`. The search function can point to an internal or external search service to provide the information. Then, update the datastore schema definition and your IEG script.



[Click here for a video presentation that gives an overview of the combo box.](#)

Implementing search functions for ComboBox components

You can implement the `ComboBox` component to search external data sources as you type in a form field, with a built-in filter function. Implement a search function and associated error handling, and make that search function available to the IEG form. If needed, you can implement an **Add New** option so that users can add an item if it is not found.

Procedure

1. Implement the search function. A search function is a JavaScript function that receives one parameter that contains the value of the `ComboBox`, and returns an array of items to be displayed by the `ComboBox`.

The response of `search-function` is an array of items, `{items}`. Each item is an object with the following structure:

```
{
  id: "key"
  value: "value"
```

```

    item: { "attribute1": "value1", "attribute2": "value2" },
  }

```

Where:

- `id` is a mandatory attribute to store the ID in the data store.
- `value` is the value of the question to store in the data store and to render in the list of options of the `ComboBox`.
- `item` is an optional complex object with the structure of the `formData` to be populated if that element is selected in the `ComboBox` component.

The structure of the `item` object must match the `formData` of the target entity. The following simple example populates the `ResidentialAddress` entity:

```

{
  'street1': 'street1',
  'street2': 'street2',
  'city': 'city',
  'zipCode': 'zipCode',
  'state': 'state',
}

```

2. Register the search function with the `IEGRegistry` object. `IEGForm` has access to `IEGRegistry` and all registered functions. `IEGForm` reads the custom functions from `IEGRegistry` and stores them on its `formContext` so `IEGForm` can call custom functions.
 - a. Implement the JavaScript function in any `.js` file.
 - b. Import `IEGRegistry` in a JavaScript initial file, such as `App.js`, and add the custom function to the registry. For example:

```

import { IEGRegistry } from '@spm/core';
import { searchCity, customFunction } from './examples/playground/customFunctions';
...

const App = () => {
  IEGRegistry.registerComboBoxSearchFunctions({ searchCity, customFunction });
  ....
};

```

Add New option

If you want to render an **Add New** option in the menu that is displayed by the `ComboBox`, the response of the JavaScript function must follow the structure:

```

{
  newItem: { id: '-1', label: 'Add New', value: ' ', position: 'top' },
  items,
}

```

Where:

- `newItem` is a complex object with the definition of the **Add New** option.
- `id` is the ID of the new option.
- `label` is the label of the new option.
- `value` is the value of the new option.
- `position` is the position where the new option renders. The possible values are `bottom` and `top`.

Error messages

The search function must implement its own logic to handle errors if an error needs to be displayed on the UI, the response of the search function must be:

```

{errorMessage: 'Controlled Error Message'}

```

The error message is displayed underneath the ComboBox.

Configuring combo box scripts and schemas

Add the combo-box element to a question in your IEG script and configure the combo-box element attributes. Add a cluster after the question to display the information to the user when they select a menu item. Update the schema definition with the appropriate elements.

About this task

The question schema type must be a string. You cannot use a question with a combo-box child element as a control question.

You can review the design system usage guidance for the ComboBox component. In your development environment, open the Social Program Management Design System Storybook documentation at `<path>@govhhs/govhhs-design-system-react/doc/index.html` and search for ComboBox.

For more information about the IEG combo-box element, see [combo-box](#).

Procedure

1. Add the combo-box child element to the question element. For example:

```
<question-page id="AboutTheApplicant_GB" read-only="false" set-focus="false" show-back-button="false" show-exit-button="true" show-next-button="true" show-person-tabs="false" show-save-exit-button="true" entity="Person" >

<!-- ComboBox -->
<cluster entity="SearchAddress">
  <title id="SearchAddress.Title">Your address</title>
  <question id="fullAddress" mandatory="true" show-field-help="false">
    <label id="FullAddress.Label">Search for your address</label>
    <combo-box key="id" search-function="searchAddress" target-entity="ResidentialAddress"
      filter-items="true" />
  </question>
</cluster>
</question-page>
```

Where:

- **key** is the id to be stored in the data store and renders as a hidden widget on the front end. It is mandatory and the entity must define this property in the schema definition. The key schema type must be a string.
- **search-function** is the name of the JavaScript search function to be called on each keydown event.
- **target-entity** is an optional attribute to show information to the user when they select a combo box menu item. In **target-entity**, specify the cluster entity to be populated with the value of the search-function result item attribute. Update the script to display the cluster entity on the page, the target entity must be shown on the same page as the combo box. If more than one cluster on the page is related to the same entity name, the first cluster that matches the entity attribute value with the target-entity value is populated.
- **filter-items** is an optional attribute that, if true, filters the items as you type with the built-in filter. By default, it is false.

2. Add a cluster to display the target-entity information when a user selects a menu item.

```
<question-page id="AboutTheApplicant_GB" read-only="false" set-focus="false" show-back-button="false" show-exit-button="true" show-next-button="true" show-person-tabs="false" show-save-exit-button="true" entity="Person" >

<!-- ComboBox -->
<cluster entity="SearchAddress">
  <title id="SearchAddress.Title">Your address</title>
  <question id="fullAddress" mandatory="true" show-field-help="false">
    <label id="FullAddress.Label">Search for your address</label>
    <combo-box key="id" search-function="searchAddress" target-entity="ResidentialAddress"
      filter-items="true" />
  </question>
</cluster>
</question-page>
```

```

</cluster>

<!-- ComboBox -->
<cluster entity="ResidentialAddress">
  <title id="Address.Title">Enter address</title>
  <help-text id="ADHelp">You must enter the address in which you physically reside
(residential address).</help-text>
  <question control-question="false" id="street1" mandatory="true" multi-
select="false" show-field-help="false">
    <label id="Street1.Label">Street 1</label>
  </question>
  <question control-question="false" id="street2" mandatory="false" multi-
select="false" show-field-help="false">
    <label id="Street2.Label">Street 2</label>
  </question>
  <question control-question="false" id="city" mandatory="false" multi-select="false"
show-field-help="false">
    <label id="City.Label">City</label>
  </question>
  <question control-question="false" id="zipCode" mandatory="false" multi-
select="false" show-field-help="false">
    <label id="Zipcode.Label">ZIP code</label>
  </question>
</cluster>
</question-page>

```

3. Edit the schema definition and add an element for the combo box and the target entity, for example:

```

<!-- ComboBox -->
<xs:element name="SearchAddress">
  <xs:complexType>
    <xs:attribute name="id" type="IEG_STRING" />
    <xs:attribute name="fullAddress" type="IEG_STRING" />
  </xs:complexType>
</xs:element>
<!-- Target Entity -->
<xs:element name="ResidentialAddress">
  <xs:complexType>
    <xs:attribute name="street1" type="IEG_STRING"/>
    <xs:attribute name="street2" type="IEG_STRING"/>
    <xs:attribute name="city" type="IEG_STRING"/>
    <xs:attribute name="zipCode" type="IEG_STRING"/>
  </xs:complexType>
</xs:element>
2. Associate that new element to a Person entity.
<xs:element name="Person">
  <xs:complexType>
    <xs:sequence minOccurs="0">
      <xs:element ref="SearchAddress" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="ResidentialAddress" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Customizing script behavior with BaseFormContainer

The behavior of scripts in the application is controlled by the `BaseFormContainer.js` container component. Each form calls this container component, which controls script behavior such as whether partial submission is allowed, or where to go on exiting the script. You can customize the behavior for individual scripts by modifying `BaseFormContainer` properties.

About this task

The following `BaseFormContainer` properties are available:

- `iegFormId`. (Mandatory) This property corresponds to the IEG execution ID that is obtained from one of the following options:
 - An API that starts the script, by creating the execution with the necessary script ID and data store schema.
 - Existing executions that can be resumed.

Note: Later, the ID is used on the server to ensure that the current user matches the user who is associated with the execution in the `CitizenScriptInfo` table. The ID also ensures that the execution is not completed.

- **title.** (Mandatory) The title to be displayed in the header. You can translate the property by using the `formatMessage` for `react-intl`.
- **isLoginOrSignupAllowed.** If the property is `true` when **Save and exit** is clicked and the user is not logged in, the log-in screen is displayed. The default value is **True**.
- **isPartialSubmissionAllowed.** Specifies that partially completed scripts can be submitted. The corresponding option must be added to the header. The default value is **False**.
- **onExit.** Specifies what happens when a user exits the script without saving. By default, it goes to the home page.
- **onFinish.** Specifies what happens when the last page of the script is submitted. By default, it goes to the home page.
- **onPartialSubmission.** Specifies what happens when a partial script is submitted. By default, it saves the current page and then starts the `OnFinish` handler.
- **onSaveAndExit.** Specifies what happens when a user saves and exits the script. By default, it saves the current page and determines what page to go to. If the user is not logged in, the log-in page is displayed. If the user is logged in, the dashboard is displayed.
- **onRef.** A function that receives the instance of the current `BaseFormContainer` to provide access to its defined functions and props. You can use this function to customize the default `BaseFormContainer` functions. For an example of using the `onRef` function to customize the behavior of **Save and exit**, see the `SampleApplicationFormComponent` in the `sampleApplication`.

Procedure

1. To modify the behavior for an existing feature, from the root of your application, open / `node_modules/@spm/universal-access-ui/src/features/Forms`. Directories for each feature with IEG forms are displayed.
2. Open the directory and edit the `<feature>FormContainer.js` file. For example, `Eligibility/EligibilityFormContainer.js`.
3. In the `render()` function, modify the properties as needed.

Merging clusters with the `cluster` element `grouping-id` attribute

If you are developing pages in IEG, you can merge several clusters on summary pages by using the **cluster** element **grouping-id** attribute. The **grouping-id** attribute is not supported for standard IBM Cúram Social Program Management web applications.

Related data fields can be defined within different clusters under the following conditions. You can use the **grouping-id** attribute to merge these related data fields into a single cluster on IEG pages.

- Data is defined within different schema entities but a single cluster can be defined for a single entity only.
- Data is defined within a conditional cluster but it must be included in a non-conditional cluster when the condition is met.

All clusters with a specific **grouping-id** attribute are merged into the first cluster with that **grouping-id** attribute. Aside from the questions, the cluster elements are shown as defined by the first cluster. Ensure that the other cluster elements in the first cluster, such as the title or buttons, are suitable for the merged cluster.

Where possible, do not have a conditional cluster as the first cluster if you are merging conditional and non-conditional clusters. If the first cluster is conditional and the condition is not met, then the merged cluster is not displayed. If a conditional cluster must be positioned before non-conditional clusters in a

merged cluster, then add a non-conditional cluster with no questions as the first cluster with the **grouping-id**.

This sample XML snippet merges three clusters into a single cluster with the **grouping-id** attribute. The three clusters have data fields from three different entities and the last cluster is conditional.

```
<cluster entity="ResidentialAddress" grouping-id="100">
  <title id="Address.Title">Address</title>
  <edit-link
    skip-to-summary="false"
    start-page="AboutTheApplicant_GB"
  />
  <layout>
    <type>flow</type>
    <num-cols>2</num-cols>
    <label-alignment>left</label-alignment>
  </layout>
  <question
    id="street1"
  >
    <label id="Street1.Label">Street 1:</label>
  </question>
  ...
</cluster>
<cluster entity="Person" grouping-id="100">
  <question
    id="applyToMailingAddress"
  >
    <label id="ApplyToMailingAddress.Label">Mail to Same Address?</label>
  </question>
</cluster>
<condition expression="Person.applyToMailingAddress==&quot;N20ITYN2&quot;">
  <cluster entity="MailingAddress" grouping-id="100">
    <question
      id="street1"
    >
      <label id="Street1.Label">Street 1:</label>
    </question>
    ...
  </cluster>
</condition>
</cluster>
```

Configuring relationship pages questions

If you are developing pages in IEG, you can configure the text of the relationship questions on relationship pages.

By default, the question label is dynamic, in the first relationship question page, it renders as “What is [Name and Age of the Person related] to you?”. On the following relationship question pages, it renders “What is [Name and Age of the Person related] to [Name and Age of the Person]?”

The attribute name for the start date must be `startDate`.

To show age in the relationship question label, you must populate the date of birth, which is defined as the `dateOfBirth` attribute of the `Person` entity.

You can use the following IEG configuration property to configure the default text.

```
# relationship question label on relationship page
relationship.question.label={index, select, 0 {What is %2s to you?} other {What is %2s to %1s?}}
```

The example ICU template does the following:

In the first iteration:

```
What is %2s to you?
```

Where `%2s` is the related person in the first iteration.

From the second iteration until the end:

```
What is %2s to %1s?
```

Where %1s is the new main person in the iteration and %2s is the related person in the iteration.

Configuring relationship starting dates on relationship summary pages

If you are developing pages in IEG, you can configure the start date of relationships for relationship summary pages. For example, Married since Jun 12, 2014.

You can use the following IEG configuration property to configure the default text.

```
# relationship type and start date label.  
relationship.type.date.label=%1s since %2s
```

Where %1s is the relationship type and %2s is the relationship start date.

Chapter 7. Customizing IBM Cúram Universal Access

Use this information to customize Universal Access for your organization.

Customizing screening

Use the supported classes and APIs to customize screening.

For information on setting up and configuring screening, see [Configuring screening](#).

Related concepts

[Configuring screenings](#)

Define different types of screenings that citizens can complete to identify programs that they might be eligible to receive.

Track the volume, quality, and results of screenings

Use the *curam.citizenworkspace.impl.CWScreeningEvents* class to access the events that are started for screening events.

You can use *curam.citizenworkspace.impl.CWScreeningEvents* to track the volume or results of screening for reporting purposes. For more information, see to the API Javadoc for *CWScreeningEvents* in `<CURAM_DIR>/EJBServer/components/CitizenWorkspace/doc`

Populating a custom screening results page

Use the *performScreening* that is contained in the *curam.citizenworkspace.security.impl.UserSession* API to populate a custom **Screening Results** page.

The **Screening Results** page is displayed when an IEG screening script is run. The operation runs the configured rule set for the selected screening type. The results of the screening, that is, the list of eligible and undecided programs, are stored against the user's session.

The *screeningResultsForDisplay* return type of the operation allows access to three objects. These objects contain the information that is required to populate either the default or custom **Screening Results** page:

ScreeningType

The screening type that the user selected.

List<Program>

A list of the programs that the user was screened for. The *ScreeningResultsOrderingStrategy* sets the order of the programs listed.

Map<String, ProgramType>

A *join.util.map* that contains a mapping of strings to *ProgramTypes* where the string contains the unique reference for that *ProgramType*.

The following is a sample usage:

```
UserSession userSession = userSessionDAO.get(sessionID);
ScreeningResultsForDisplay screeningResultsForDisplay =
    userSession.performScreening();
```

The following is a sample interface definition:

```
/**
 * Executes the Screening rule set associated with the current screening IEG
 * script data. The return object, {@link ScreeningResultsForDisplay},
 * contains all of the information required to be displayed on the
 * Screening Results page.
 */
```


For more information, see the API Javadoc for the `curam.citizenworkspace.security.impl.UserSession` in `<CURAM_DIR>/EJBServer/components/CitizenWorkspace/doc`.

Use customization points, for example, customizing the generic PDF for processed applications, to customize the application intake process when an intake application is submitted.

Review a summary of the intake application workflow in a flowchart.



This automatic activity creates a PDF document based on the content of the application. For more information, see “Customizing the generic PDF for processed applications” on page 201.

InvokeLegacySystemProcessing

This automatic activity sends applications to legacy systems by using web services. This path is taken only if there are legacy systems that are associated with at least one of the programs on the application.

CreateParticipantsAndCasesDeferAES

This automatic activity creates participants for the submitted application and then creates one or more cases for the programs that are associated with the application. In most cases, application cases are created.

This path is taken only if the configuration property `curam.intake.use.resilience` is set to `true`. For compatibility with previous versions, this property is `false` by default. However, it is recommended that all production systems set this value to `true`. For more information about the implications of setting this value to `true`, see [“Using events to extend intake application processing” on page 201](#).

This activity also defers the evidence sharing that occurs when case members are added to an application case to a later point in the workflow.

Note: For customizations to this workflow or for custom intake workflows, evidence sharing can be deferred by using the `curam.aes.sl.observe.impl.AESShareSetPullManualEnactment.process()` API. For more information, see the associated Javadoc.

Mapping

This automatic activity uses the Cúram Data Mapping Engine (CDME) to map data collected in the application script into Case Evidence. If a validation issue occurs with the mapped evidence, this activity is automatically retried. During the retry, if there is a single Application Case, the validations are disabled and a WDO flag `IntakeCaseDetails.mappingValidInd` set to `false`.

EvidenceCorrections

This manual task is called if the Mapping activity fails due to a validation error. That is, the `IntakeCaseDetails.mappingValidInd` set to `false`. The assignment of this task is configurable. For more information, see [Evidence Issues Ownership Strategy](#). The caseworker or operator must resolve the evidence validation issues and resubmit the application.

Subflow: Evidence Sharing Pull Workflow

If there is evidence to share, the evidence sharing pull workflow is enacted to trigger any sharing activities that were deferred in `CreateParticipantsAndCasesDeferAES`.

If there is no evidence to share, the workflow bypasses this step.

PostMapping

This automatic activity starts the next stage of application processing by calling the `IntakeApplication.IntakeApplicationEvents.postMapDataToCuram()` event.

CreateParticipantsCasesAndMapEvidence

This path is followed when the configuration property `curam.intake.use.resilience` is set to `false`. This automatic activity behaves identically to the older non-resilient workflow. It creates cases and participants and completes all evidence mapping in a single transaction, which results in a less resilient process if a failure occurs.

Customers can customize the workflow compliantly as described in the [Developing Compliantly with IBM Cúram Social Program Management](#) and [Cúram Workflow Reference](#). Customers must not change the enactment structs that are used by these workflows.

Related concepts

[Customizing the generic PDF for processed applications](#)

Use IBM Cúram Universal Access to map all intake applications to a generic PDF that records the values of all the information that the user enters.

[Using events to extend intake application processing](#)

The interface `IntakeApplication.IntakeApplicationEvents` contains events that are invoked when citizens submit an intake application for processing.

Related information

[Evidence Issues Ownership Strategy](#)

Customizing the generic PDF for processed applications

Use IBM Cúram Universal Access to map all intake applications to a generic PDF that records the values of all the information that the user enters.

This PDF is rendered by the XML Server. Customers can override the default formatting of the generic PDF as follows:

1. Copy `CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/XSLTEMPLATEINST.dmx` to `CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData`.
2. Edit `project\config\datamanager_config.xml`, replace the entry for: `CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/XSLTEMPLATEINST.dmx` with an entry for: `CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData/XSLTEMPLATEINST.dmx`
3. Copy `CURAM_DIR/EJBServer/components/Workspaceservices/Data_Manager/InitialData/blob/WSXSLTEMPLATEINST001` to: `CURAM_DIR/EJBServer/components/custom/Data_Manager/InitialData/blob`.
4. Edit `WSXSLTEMPLATEINST001` to suit the needs of the project.

Using events to extend intake application processing

The interface `IntakeApplication.IntakeApplicationEvents` contains events that are invoked when citizens submit an intake application for processing.

Use these events to change the way that intake applications are handled, for example supplement or replace the standard CDME mapping or perform an action after an application has been sent to a remote system using web services. For more information, see the API Javadoc information for `IntakeApplication.IntakeApplicationEvents` in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

The interface `IntakeProgramApplication.IntakeProgramApplicationEvents` contains events that are invoked at key stages during the processing of an application for a particular program. For information, see the API Javadoc information for `IntakeProgramApplication.IntakeProgramApplicationEvents` in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

Customizing the concern role mapping process

The `curam.workspaceservices.applicationprocessing.impl` package contains a `ConcernRoleMappingStrategy` API that provides a customization point into the online application process.

Use the `ConcernRoleMappingStrategy` API to implement custom behavior following the creation of each new concern role that is added to an application. For example, customers who have customized the prospect person entity might want to store information on that entity that cannot be mapped using the default CDME processing.

Enable the ConcernRoleMappingStrategy API

In the IBM Cúram Social Program Management system administration application, enable the `ConcernRoleMappingStrategy` API by setting the `Enable Custom Concern Role Mapping` property to true.

Procedure

1. Log in to the Social Program Management application as system administrator.
2. Click **System Configurations > Application Data > Property Administration**.

3. In the **Application - Intake Settings** category, search for the property *curam.intake.enableCustomConcernRoleMapping*.
4. Edit the property to set its value to `true`.
5. Save the property.
6. Select **Publish**.

Use the ConcernRoleMappingStrategy API

Use the enabled ConcernRoleMappingStrategy API to implement a strategy for mapping information to a custom concern role.

About this task

The `curam.workspaceservices.applicationprocessing.impl` package contains the ConcernRoleMappingStrategy API.

Procedure

1. Provide an implementation of the customization point.
2. Bind your custom implementation by creating or extending your custom mapping module as follows:

```
package com.myorg.custom;
class MyModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(ConcernRoleMappingStrategy.class).to(
            MyCustomConcernRoleMapping.class);
    }
}
```

3. If you did not already add your MyModule class to the ModuleClassName table by using an appropriate DMX file, add your MyModule class.

How to send applications to remote systems for processing

Use the Citizen Workspace to send applications to remote systems that use web services for processing.

An event `ReceiveApplicationEvents.receiveApplication` is raised before the application is sent to the remote system. The event can be used to edit the contents of the data store that is used to gather application data before transmission. For more information, refer to the API Javadoc for `ReceiveApplicationEvents`, which is in `<CURAM_DIR>/EJBServer/components/WorkspaceServices/doc`.

Customizing the Citizen Account

Users can use the Citizen Account to log in to a secure area where users can screen and apply for programs.

Users also use the Citizen Account to view information relevant to them, including individually tailored messages, system-wide announcements, updates on their payments, contact information for agency staff and outreach campaigns that might be relevant to them. The Citizen Account also provides a framework for customers to build their own pages or override the existing pages.

Security and the Citizen Account

Security must be a primary concern when you customize the citizen account customizations. All public-facing applications must be analyzed and tested before they are deployed. Users must contact IBM support to discuss unusual customizations that might have specific security issues.

Permission to call the server facade methods that serve data to citizen account pages is managed by the standard authorization model. For more information, see the *Server Developer* documentation. In addition to the standard authorization checks, each facade method that is called by a **Citizen Account** page must

complete the following security checks to ensure the user who is associated with the transaction (the currently logged in user) has permission to access the data they are requesting:

- Ensure that the currently logged in user is of the correct type. They must be an external user with an applicationCode of CITWSAPP, and have an account of type Linked.
- Ensure that the currently logged in user has permission to access the specific records that they are reading. For instance, validate any page parameters that are passed in to ensure that the records requested are related to the currently logged in user in some way.

Ensure that the currently logged in user is the correct type

The `curam.citizenaccount.security.impl.CitizenAccountSecurity` API offers a method `performDefaultSecurityChecks` that ensures that the user is the correct type. This method checks the user type, and if not acceptable, writes a message to the logs and fails the transaction.

Note: This API needs to be called in the first line of every custom facade method before any processing or further validation takes place:

```
public CitizenPaymentInstDetailsList listCitizenPayments()
    throws ApplicationException, InformationalException {

    // perform security checks
    citizenAccountSecurity.performDefaultSecurityChecks();

    // validate any page parameters (none in this case)

    // invoke business logic
    return citizenPayments.listPayments();
}
```

Ensure that the logged in user has access to the requested records

A malicious user who is logged in to a valid linked account might send requests to the system to request other users' data. To prevent this intrusion from happening, all page parameters must be validated to ensure that they are somehow traceable back to the currently logged in user. How this conclusion is determined is different for each type of record.

For example, a **Payment** can be traced back to the **Participant** by way of the **Case** on which it was entered.

The `curam.citizenaccount.security.impl.CitizenAccountSecurity` application programming interface (API) offers methods to complete these checks for the types of records that are served to citizens by the initially configured pages. For specific information, review the Javadoc of this API. For custom pages that serve different types of data, additional checks must be implemented to validate the page parameters.

This process needs to be added to a custom security API and called by the facade methods in question. The methods need to check to see whether the record requested can be traced back to the currently logged in user, and if not, it needs to log the user name, method name, and other data. If these conditions are not met, the transaction needs to be failed immediately (as opposed to adding the issue to the validation helper and allowing the transaction to proceed):

```
if (paymentInstrument.getConcernRole().getID()
    != citizenWorkspaceAccountManager
        .getLoggedInUserConcernRoleID().getID()) {

    /**
     * the payment instrument passed in is not related
     * to the logged in user log the user name of the
     * current user, the method invoked and any other
     * pertinent data
     */

    // throw a generic message
    throw PUBLICUSERSECURITYExceptionCreator
        .ERR_CITIZEN_WORKSPACE_UNAUTHORISED_METHOD_INVOKATION();
}
```

While as much information as possible regarding the infraction needs to be logged, it is important to ensure that the exceptions thrown do not display any information that might be useful to malicious users. A generic exception needs to be thrown that does not contain any information that relates to what went wrong. The `curam.citizenaccount.security.impl.CitizenAccountSecurity` API throws a generic message that states You are not privileged to access this page.

Messages

When a linked citizen logs in, messages are gathered from the system and from remote systems for display.

The `curam.citizenmessages.impl.CitizenMessageController` API gathers and displays messages. The API reads persisted messages by participant from the `ParticipantMessage` database table. The API also raises the `CitizenMessagesEvent.userRequestsMessages` event, inviting listeners to add messages to a list that is passed as part of the event parameter. The messages that are gathered from each source are sorted, turned into XML, and returned to the citizen for display.

Configuring citizen messages

Global configurations are included that can be specified for **Citizen Messages**, such as enabling certain types and configuring their display order. The different types of messages also include their own configuration points. Specific information about how to customize the various message types is provided later.

The textual content of a message type also can be configured. Each message type has a related properties file that includes the localizable text entries for the various messages displayed for that type. These properties also include placeholders that are substituted for real values related to the citizen at run time.

The wording of this text can be customized, by inserting a different version of the properties file into the resource store. The following table defines which properties file need to be changed for each type of message:

Table 19. Message properties files	
Message type	Property file name
Payments	<code>CitizenMessageMyPayments.properties</code>
Application Acknowledgment	<code>CitizenMessageApplicationAcknowledgement.properties</code>
Verifications	<code>CitizenMessageVerificationMessages.properties</code>
Meetings	<code>CitizenMessageMeetingMessages.properties</code>
Referral	<code>CitizenMessagesReferral.properties</code>
Service Delivery	<code>CitizenMessagesServiceDelivery.properties</code>

You can also remove placeholders (which are populated with live data at run time) from the properties. However, there is currently no means to add further placeholders to existing messages. A custom type of message must be implemented in this situation.

Adding a new type of citizen message

Messages are gathered by the controller in two ways: the controller reads messages that were persisted to the database by using the `curam.citizenmessages.persistence.impl.ParticipantMessage` API, and also gathers them by raising the `curam.participantmessages.events.impl.CitizenMessagesEvent`

A decision needs to be made regarding whether to 'push' the messages to the database, or else have them generated dynamically by a listener that listens for the event that is raised when the citizen logs in. The specific requirements of the message type need to be considered, along with the benefits and drawbacks of each option.

Persisted messages

In this scenario, when something takes place in the system that might be of interest to the citizen, a message is persisted to the database. For example, when a meeting invitation is created, an event is fired. The initially configured meeting messages function listens for this event. If the meeting invitee is a participant with a linked account, a message is written to the ParticipantMessage table that informs the citizen that they are invited to the meeting.

One benefit of this approach is that little processing is done when the citizen logs in to see this message: the message is read from the database and displayed, as opposed to calculation that takes place that would determine whether the message was required. However, the implementation also needs to handle any changes to the underlying data that might invalidate or change the message, and take appropriate action.

For example, the meeting message function also listens for changes to meetings to ensure the meeting time, location, and similar, are up to date, and to send a new message to the citizen to inform the citizen that the location or time was changed.

Dynamic messages

These messages are generated when the citizen logs in, by event listeners that listen for the `curam.participantmessages.events.impl.CitizenMessagesEvent.userRequestsMessages` event.

Because the message is generated at runtime, code is not required to manage change over time. The message is generated based on the data within the system each time the citizen logs in. If some underlying data changes, the next time the citizen logs in, they will get the correct message.

A drawback to this approach is that significant processing might be required at run time to generate the message. Care must be taken to ensure that this processing does not adversely affect the load time of the **Citizen Account** dashboard.

Performance considerations must be evaluated against the requirements of the specific message type and the effort that is required to manage change to the data that the message is related to over time. For example, the initially configured verification message is dynamic. When a citizen logs in, it checks to see whether any outstanding verifications exist for that citizen. This process is a relatively simple database read, whereas it would be complicated to listen for various events in the Verification Engine and ensure that an up-to-date message was stored in the database related to the participants' outstanding verifications. Alternatively, the meeting messages need to inform the citizen of changes to their meetings, so functionality had to be written to manage changes to the meeting record and its related message over time.

Implementing a new message type

Organizations can implement a dynamic message or a persisted message.

To implement a new message type, regardless of whether the message is persisted or is generated dynamically, complete the following steps.

Common tasks

- In the administration system, add an entry to the CT_ParticipantMessageType code table to represent the new message type.
- Add a DMX entry for the ParticipantMessageConfig database table. This entry stores the type and sort order of the new message type and is used for administration. For example:

```
<row>
  <attribute name="PARTICIPANTMESSAGECONFIGID">
    <value>2110</value>
  </attribute>
  <attribute name="PARTICIPANTMESSAGETYPE">
    <value>PMT2001</value>
  </attribute>
  <attribute name="ENABLEDIND">
    <value>1</value>
  </attribute>
```

```

    <attribute name="SORTORDER">
      <value>5</value>
    </attribute>
    <attribute name="VERSIONNO">
      <value>1</value>
    </attribute>
  </row>

```

- Add a properties file to the App Resource store that contains the text properties and image reference for the message.
- Add an image for this message type to the resource store.

Implementing a dynamic message

To implement a dynamic style message, an event listener must be implemented to listen for the `CitizenMessagesEvent.userRequestsMessages` event. This event argument contains a reference to the Participant and a list, to which the listener adds `curam.participantmessages.impl.ParticipantMessage` Java objects.

For more information, see the Javadoc API for `CitizenMessagesEvent` in the `<CURAM_DIR>/EJBServer/components/core/doc` directory. For a full explanation, see the Javadoc API for `curam.participantmessages.impl.ParticipantMessage` and `curam.participantmessages.impl.ParticipantMessages`.

The message text is stored in a properties file in the resource store. A dynamic listener retrieves the relevant properties from the resource store, and creates the `ParticipantMessage` object. The message text for a message can include placeholders. Values for placeholders are added to `ParticipantMessage` objects as parameters. The `CitizenMessagesController` resolves these placeholders, replacing them with the real values for the participant.

For example, look at this entry from the `CitizenMessageMyPayment.properties` file:

```

Message.First.Payment=
  Your next payment is due on {Payment.Due.Date}

```

The actual payment due date of the payment is added to the `ParticipantMessage` object as a parameter. The `CitizenMessagesController` then resolves the placeholders, populating the text with real values, and then turns the message into XML that is rendered on the citizen account. A public `CitizenMessageController` method also exists, which returns all messages for a citizen as a list, see the Javadoc.

From the `curam.participantmessages.impl.ParticipantMessage` API:

```

/**
 * Adds a parameter to the map. The paramReference
 * should be present in the message title or body so
 * it can be replaced by the paramValue before the message
 * is displayed.
 *
 * @param paramReference
 * a string place holder that is present in either the
 * message title or body. Used to indicate where the value
 * parameter should be positioned in a message.
 *
 * @param paramValue
 * the value to be substituted in place of the place holder
 */
public void addParameter(final String paramReference,
    final String paramValue) {

    parameters.put(paramReference, paramValue);
}

```

The call to the method would look like this:

```
participantMessage.addParameter("Payment.Due.Date", "1/1/2011");
```

Messages can also include links, which are also resolved at run time. Links can use placeholder values for the link text. A link is defined in a properties file as shown.

Click {link:here:paymentDetails} to view the payment details.

In this example, here is the text to display, and `paymentDetails` is the name of the link to be inserted at that point in the text. For more information, see the *Advisor Developer's Guide*. For a dynamic listener to populate this link with a target, it creates a `curam.participantmessages.impl.ParticipantMessageLink` object, specifying a target and a name for the link. The code would look like this example:

```
ParticipantMessageLink participantMessageLink =
    new ParticipantMessageLink(false,
        "CitizenAccount_listPayments", "paymentDetails");

participantMessage.addLink(participantMessageLink);
```

Before the dynamic listener composes the message, it must check to ensure that the message type in question is enabled. The `curam.participantmessages.configuration.impl.ParticipantMessageConfiguration` record for that message type is read, and the `isEnabled` method is used to determine whether this message type is enabled. If not, processing stops.

Note: You can separate the code that listens for the event and the code that composes a specific message to adhere to the philosophy of "doing one thing and doing it well".

Implementing a persisted message

To display a persisted message to the citizen, it must be written to the database with the `curam.citizenmessages.persistence.impl.ParticipantMessage` API. Message arguments are handled by persisting a `curam.advisor.impl.Parameter` record and associating it with the `ParticipantMessage` record. Links are handled by the `curam.advisor.impl.Link` API. Parameter names map to placeholders in the message text. Link names relate to the names of links that are specified in the message text. For more information, see the Javadoc for `curam.citizenmessages.persistence.impl.ParticipantMessage`, `curam.advisor.impl.Parameter`, and `curam.advisor.impl.Link`.

An expiry date time must be specified for each `ParticipantMessage`. After this date time, the message is no longer be displayed.

Messages can be removed from the database. If a message needs to be replaced with a modified version, or removed for another reason, use the `curam.citizenmessages.persistence.impl.ParticipantMessage` API.

Each message has a related ID and type that is used to track the record that the message is related to. For example, meeting messages store the Activity ID and a type of Meeting. Messages can be read by participant, related ID, and type by the `ParticipantMessageDAO`.

Before it persists the message, the dynamic listener checks to ensure that the message type in question is enabled. The `curam.participantmessages.configuration.impl.ParticipantMessageConfiguration` record for that message type is read, and the `isEnabled` method is used to determine whether this message type is enabled. If not, no further processing occurs.

Customizing specific message types

Organizations can customize the default message to create a referral message or a service delivery message.

Referral message

This message type creates messages related to referrals. This is a dynamic message. When the citizen logs in, a message will be created for each referral that exists for the citizen in the system, provided that referral has a referral date of today or in the future, and provided that a related Service Offering has been specified for this referral. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageReferral.properties` contains the properties for the referral message text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageReferral`. To

change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Service delivery message

This message type creates messages related to service deliveries. This is a dynamic message. When the citizen logs in, a message will be created for each service delivery that exists for the citizen in the system, provided that service delivery has a status of 'In Progress' or 'Not Started'. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageServiceDelivery.properties` contains the properties for the service delivery message text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageServiceDelivery`. To change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Payment messages

The payment message type creates messages based on the payments that are issued or canceled for a citizen.

The payment messages are persisted to the database. They replace each other, for example, if a payment is issued and then canceled, the payment issued message is replaced with a payment canceled message. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageMyPayments.properties` contains the properties for financial message text, message parameters, links, and images. This properties file is stored in the resource store. This resource is registered in the resource name `CitizenMessageMyPayments`. To change the message text of financial messages, or to remove placeholders or change links, upload a new version of this file to the resource store. The following table lists the messages that are created when events that are related to payments occur in the system, and the related property in `CitizenMessageMyPayments.properties`.

Table 20. Payment messages and related properties	
Payment event	Message Property
First payment issued on a case	Message.First.Payment
Latest payment issued	Message.Payment.Latest
Last payment issued	Message.Last.Payment
Payment canceled	Message.Cancelled.Payment
Payment reissued	Message.Reissue.Payment
Payment stopped (case suspended)	Message.Stopped.Payment
Payment / Case unsuspended	Message.Unsuspended.Payment

Customization of the payment messages expiry date

You can set the number of days that the payment message is displayed to the citizen with a system property. By default the property value is set to 10 days, but you can override this default from property administration.

Table 21. Payment message expiry property	
Name	Description
<code>curam.citizenaccount.payment.message.expiry.days</code>	The number of days that the payment message is displayed to the participant.

Meeting messages

The meeting message type creates messages based on meetings that citizens are invited to, provided that they are created by using the `curam.meetings.sl.impl.Meeting` API.

The API raises events that the meeting messages functionality consumes. There are other ways of creating Activity records without this API, but meetings created in these ways do not have related

messages created as the events are not raised. These messages are persisted to the database. They replace each other, for example, if a meeting is scheduled and then the location is changed, the initial invitation message is replaced with one informing the citizen of the location change. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageMeetingMessages.properties` contains the properties for the meeting messages text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered in the resource name `CitizenMessageMeetingMessages`. To change the message text of meeting messages, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store. Table 1 describes the messages created when various events related to meetings occur in the system, and the properties in `CitizenMessageMeetingMessages.properties` that relates to each message created. Different versions of the message text are displayed depending on whether the meeting is an all day meeting, whether a location has been specified, and whether the meeting organizer has contact details registered in the system. Accordingly, the property values in this table are approximations that relate to a range of properties within the properties file. Refer to the properties file for a full list of the message properties.

Table 22. Meeting messages	
Meeting event	Message Properties
Meeting invitation	Non.Allday.Meeting.Invitation.*, Allday.Meeting.Invitation.*
Meeting update	Non.Allday.Meeting.Update.*, Allday.Meeting.Update.*
Meeting canceled	Allday.Meeting.Update.*, Allday.Meeting.Cancellation.*

Customization of the meeting messages display date

The number of days before the meeting start date that the message should be displayed to the citizen can be configured using a system property. By default the property value is set to 10 days, however, this can be overridden from property administration.

The meeting message expires (it is no longer displayed to the citizen) at the end of the meeting, that is, the date time at which the meeting is scheduled to end.

Table 23. Meeting message display date property	
Name	Description
curam.citizenaccount.meeting.message.effective.days	The number of days before the meeting start date that the message should be displayed to the citizen.

Application acknowledgment message

The application acknowledgment message type creates a message when an application is submitted by a citizen.

The message is persisted to the database. The properties file `EJBServer\components\CitizenWorkspace\data\initial\blob\prop\CitizenMessageApplicationAcknowledgment.properties` contains the properties for the messages text, message parameters, links and images. This properties file is stored in the resource store. This resource is registered under the resource name `CitizenMessageApplicationAcknowledgment`. To change the message text of the message, or to remove placeholders or change links, a new version of this file must be uploaded into the resource store.

Customization of application acknowledgment message expiry date

The number of days the Application Acknowledgment message will be displayed to the citizen can be configured using a system property. By default the property value is set to 10 days, however, this can be overridden from property administration.

Table 24. Application acknowledgment message expiry property	
Name	Description
curam.citizenaccount.intake.application.acknowledgement.message.expiry.days	The number of days the application acknowledgment message will be displayed to the participant.

Customizing the Notices page

By default, the notices relevant to the linked user are listed on the **Notices** page. You can replace the default `CitizenCommunicationsStrategy` implementation with your own custom implementation.

For example, you can create a custom implementation to retrieve the communications of all of the household members of the logged-in citizen, instead of just the citizen.

Create an alternative implementation of the `curam.citizenaccount.impl.CitizenCommunicationsStrategy.listCitizenCommunications(ConcernRoleKey)` method for listing the citizen communication records.

In addition, a number of default hooks are available for custom implementations to customize the behavior of the communication processing module.

Related concepts

The Notices page

When a citizen is logged in, they can see all communications that are relevant to them on the **Notices** page, with sent, received, or normal status indicated. Notices are typically formal written communications that are issued to meet legal, regulatory, or state requirements, which are created by using letterhead templates.

Related tasks

Configuring communications on the Notices page

You can configure the maximum number of communications that are displayed on the **Notices** page. By default, up to 20 communications are displayed.

Communication processing hooks and events

How electronic notices are managed and supported in the Citizen Portal affects the communication processing module.

While the default implementation doesn't address or implement any of the impacts, the following default hooks are available for the custom implementation to customize the communication processing module.

curam.core.hook.impl.PreCreateCommunicationHook - can be used in customized scenarios for any kind of pre creation processing for communication records.

curam.core.hook.impl.PreModifyCommunicationHook - can be used in customized scenarios for any kind of pre modify processing for communication records.

For e.g.; in situations where create or modify operation is not applicable, this hook points can be used to redirect the user with customized messages before the creation or modification of communication records using custom exception handling.

curam.core.hook.impl.CommunicationInvocationStrategyHook - can be used as a toggle the above hooks i.e., `PreModifyCommunicationHook` and `PreCreateCommunicationHook` should be invoked or not.

The following communication processing methods have been updated by the pre creation and pre modification hooks that are mentioned above to enable further customization.

- `curam.core.facade.impl.Communication.modifyWordDocument(ModifyWordDocumentDetails)`
- `curam.core.facade.impl.Communication.modifyEmail(ModifyEmailCommDetails, ModifyEmailCommKey)`
- `curam.core.facade.impl.Communication.modifyRecordedCommunication1(ModifyRecordedCommKey, ModifyRecordedCommDetails1)`
- `curam.core.facade.impl.Communication.modifyProForma1(ModifyProFormaCommDetails1)`
- `curam.core.facade.impl.Communication.createEmailCommunication(CreateEmailCommDetails)`
- `curam.core.facade.impl.Communication.createEmail(CreateEmailCommDetails)`
- `curam.core.facade.impl.Communication.createMSWordCommunication1(CreateMSWordCommunicationDetails1)`
- `curam.core.facade.impl.Communication.createCaseMSWordCommunication1(CreateMSWordCommunicationDetails1)`

- `curam.core.facade.impl.Communication.createRecordedCommunication1(RecordedCommDetails1)`
- `curam.core.facade.impl.Communication.createProForma1(CreateProFormaCommDetails1)`
- `curam.core.facade.impl.Communication.createProFormaCommunication1(CreateProFormaCommDetails1)`

Communication events

`curam.core.events.CONCERNROLEACOMMUNICATION.INSERT_CONCERN_ROLE_COMMUNICATION`

`curam.core.events.CONCERNROLEACOMMUNICATION.MODIFY_CONCERN_ROLE_COMMUNICATION`

These are the events that are raised post-creation or post-modification of a communication record. Custom implementations can listen to these events for any kind of post processing requirements.

Customizing appeal request statuses

You can create an implementation to enable the display of appeal request status from an external appeals system in the citizen account by using the provided API.

About this task

The `curam.core.onlineappealrequest.impl.OnlineAppealRequestStatus` interface takes an appeal request as an input and passes back a code-table value. You can modify code-table entries as required.

- The appeal status text that you see in the application is hardcoded as <description> tags in two `CT_CitizenAppealRequestStatus.ctx` files.
 - The `EJBServer\components\core\codetable\CT_CitizenAppealRequestStatus.ctx` file contains the code table value for the **Appeal Request Submitted** status. This is so you can submit an appeal even if IBM Cúram Appeals is not installed and the `Appeals.jar` file is not present. You can modify the description for the **Appeal Request Submitted** status in this file.
 - When IBM Cúram Appeals is installed and the `Appeals.jar` is present, more appeal status values are available. You can modify the descriptions for the other code table status values in the `EJBServer\components\Appeal\codetable\CT_CitizenAppealRequestStatus.ctx` file.

For information about editing code tables, see [Customizing a code table file](#).

- The color of each appeal status is set by the Badge component in the Social Program Management Watson Design System. The `AppealRequestsComponent.js` file contains a `getBadgeDataByCodetable` function. The `getBadgeDataByCodetable` function is a map of code tables to badge type. For example, the `CARS1001` code table is mapped to the warning badge type so it is displayed in red. In your Web app development environment, you can see the badge colors by opening the Web Design System Storybook documentation at [@govhhs/govhhs-design-system-react/doc/index.html](#) and expanding to **Components > Badge**.

Procedure

1. Identify the appeal request ID from the caseworker application.
2. Use the appeal request ID to associate the appeal request status from the external system with the appeal request status in IBM Cúram Universal Access.
3. Implement the `curam.core.onlineappealrequest.impl.OnlineAppealRequestStatus` interface to return the appropriate code table value based on the `OnlineAppealRequest`.
For example, a custom implementation of this class might call a remote system and map the return value to an appropriate code table value.
4. Customize an appeal status message to display in the **Citizen Account**.
5. If you create a new status, you must map it to a badge type to specify a color to display.

Related tasks

[Customizing appeals](#)

You can customize appeals to suit your organization. You can integrate with an appeals system of your choice. If you are licensed for the IBM Cúram Appeals application module, the IBM Cúram Social Program Management appeals functionality is available on installation.

Error logging in the citizen account

When a citizen submits an application, when a citizen clicks **Submit** a deferred process starts. If a mapping failure occurs, an error is logged.

Application property

The application property *curam.workspaceservices.application.processing.logging.on* increases the level of detail of error messages.

When *curam.workspaceservices.application.processing.logging.on* is set to *true*, detailed error messages are written to the application log files if the submission process fails.

Error codes

Each error message is prepended with an error code. These error codes help to automatically scan application logs so that unexpected failures can be identified. The error codes that are returned by the application is defined in the code table file *CT_ApplicationProcessingError.ctx*.

The range of codes that are reserved for internal processing is **APROCER001 – APROCER500**. Customers can use the range **APROCER501 – APROCER999** to log errors in custom processing, for example error codes for extension-mapping handler class.

The list of error codes that are returned by the application, and a brief description of the problem, is listed in Table 1.

Table 25. Application error codes	
Code	Description
APROCER001	An error occurred creating a person.
APROCER002	An error occurred creating a prospect person.
APROCER003	A relationship error occurred creating a person.
APROCER004	An error occurred creating a case.
APROCER005	An error occurred while performing a "map-attribute" mapping.
APROCER006	An error occurred while performing a "set-attribute" mapping.
APROCER007	An error occurred while performing a "map-address" mapping.
APROCER008	General mapping failure.
APROCER009	Error creating evidence.
APROCER010	More than one PDF form is registered against the program type.
APROCER011	Error setting the alternate id type for a Prospect Person.
APROCER012	Invalid alternate ID value.
APROCER013	Error the Evidence Application Builder has not been correctly configured.
APROCER014	Evidence type not listed in the Mapping Configuration.
APROCER015	No parent evidence entity found.
APROCER016	An error occurred when trying to unmarshal the application XML.
APROCER017	An error occurred when trying to set a field value.

Table 25. Application error codes (continued)

Code	Description
APROCER018	An error occurred when trying to create the PDF document.
APROCER019	An error occurred when trying to create the PDF document. A form code could not be mapped to a codetable description.
APROCER020	An error occurred when trying a WorkspaceServices mapping extension handler.
APROCER021	Missing source attribute in datastore entity.
APROCER022	An attribute in an expression is not valid.
APROCER023	Application builder configuration error.
APROCER024	Failed creating <i>DataStoreMappingConfig</i> , no name specified.
APROCER025	Failed creating <i>DataStoreMappingConfig</i> , the name is not unique.
APROCER026	The mapping to datastore had to be abandoned because the schema is not registered.
APROCER027	There was a problem parsing the Mapping Specification.
APROCER028	General mapping error. Mapping XML included.
APROCER029	Cannot have multiple primary participants.
APROCER030	No programs have been applied for.
APROCER031	An error occurred while attempting to map to Person data.
APROCER032	An error occurred while attempting to map to Relationship data.
APROCER033	An error occurred while creating Cases.
APROCER034	An error occurred while creating evidence.
APROCER035	No programs have been applied for.
APROCER036	An error occurred reading data from the datastore.
APROCER037	Specified integrated case type does not exist.
APROCER038	Specified case type does not exist
APROCER039	Duplicate SSN entered for prospect person.
APROCER040	Duplicate SSN entered.
APROCER041	There was a problem with the workflow process.
APROCER042	No primary participant has been identified as part of the intake process.

Customizing life events

A description of the high-level architecture of life events and how to perform the analysis and development tasks in building a life event.

Many types of life events can be built by analysts, some require input from developers. This information will help analysts to understand how to perform the analysis for a new life event and how to determine whether input is needed from developers.

Enabling and disabling life events

The life events feature is enabled by default. When it is enabled, the life event feature is available for linked users only. You can use the **REACT_APP_FEATURE_LIFE_EVENTS_ENABLED** environment variable to enable or disable life events.

About this task

For more information about linked users, see [“User account types” on page 150](#).

The following life events functionality can be enabled or disabled:

- The **View your account** card on the **Home** page is updated to say **See your next payment, tell us of any changes in your circumstances, and more**.
- **Review your profile** card on the **Dashboard** page.
- **Tell us if any anything has changed** pane on the **Profile** page.
- Live event-related URLs, for example `/life-events/history`.

Procedure

1. Edit the `.env` file in the root of your application.
2. Set **REACT_APP_FEATURE_LIFE_EVENTS_ENABLED** to `true` or `false`. If you don't define the environment variable, the life event feature defaults to enabled.

How to build a life event

To design a life event for IBM Cúram Universal Access, you must undertake an analysis.

You can build life events for caseworkers, or use the life event infrastructure to drive other processes like certification, but that is outside the scope of this information. Java coding skills are not a prerequisite for developing all life events. Depending on requirements, many and in some cases all of the artifacts required can be developed by a Business Analyst. Business Analysts can use this information to determine whether Java developers are needed to complete the implementation of a life event.

Generally, there are two types of life events for citizens:

- Standard life events
- Round tripping life events

Standard life events allow citizens to enter new life event information and submit it to the agency. For example, a citizen logs in and submits a "Having a Baby" life event, which is new information on the system. If they submit incorrect information, such as the name of the obstetrician, they can start a new life event, reenter the information, and submit.

Round-tripping life events are more complex. The distinction between round-tripping or standard life events is whether the data that is pre-populated in the life event can be changed by the user. If a citizen is expected to update pre-populated information in addition to adding new information, the life event is considered a round-tripping life event. It's considerably harder to design scripts for this type of life event.

The primary artifacts that constitute a simple life event are:

- An IEG script and its associated data store schema
- An IEG script to review answers in a previously submitted life event (optional)
- A Cúram Data Mapping Engine specification that describes how to map data from the IEG script into evidence on the citizen's cases

All of these artifacts can be configured in the administrator application. For more information about configuring Simple life events, see [“Configuring life events” on page 56](#).

Information that is entered is processed by the life events system as follows:

1. If the user is linked to the local IBM Cúram Social Program Management case processing system, then the life events system can update related evidence in their cases.

2. If the user is linked to remote systems, then the life events system can send updates to the remote system by using web services.

If the life event is a round-tripping life event, or it is required to update the person's evidence in IBM Cúram Social Program Management, then some development work is needed. See the life events APIs needed to meet these requirements or indeed to supplement the standard life event behavior with more custom functionality.

Customizing advanced life events

To develop advanced life events, you must understand the difference between a simple life event and advanced life event.

When to use advanced life events

Advanced life events enable fully automated round-tripping of data. This means that evidence is read into the datastore for an IEG script. It is then updated by the citizen. When the life event is submitted, the original evidence that was read into the IEG script is updated. Advanced life events are only required when this level of automated round tripping of data is required. Under all other circumstances Simple life events are the recommended approach. Project Architects should consider carefully whether round tripping is required or whether the data entered can be treated as new evidence to be integrated into the citizen's cases.

Advanced life events cannot be configured through the administration user interface, they must be created by developers.

How to build a life event

Analysis

The distinction between standard life events and round tripping life events is whether citizens can change the data that is pre-populated into the life event. If citizens can update pre-populated information, rather than just adding new information, then use a round tripping life event. It's more difficult to develop this type of life event. The advanced life events subsystem is designed to cater for round tripping life events.

The following describes how to develop an advanced life event that supports round tripping:

The primary artifacts that constitute an advanced life event are:

- An IEG script and its associated data store schema.
- An IEG script to review answers in a previously submitted life event (optional).
- A Recommendations Ruleset that produces the set of recommendations based on the information that is entered in the IEG script (optional).

The life events system can take information that is entered by the user and update related evidence in any cases they have.

The life events system can do one of the following:

1. If the user is linked to the local IBM Cúram Social Program Management case processing system, then the life events system can update related evidence in any of their cases.
2. If the user is linked to remote systems, then the life events system can send updates to related remote systems through web services.

You can configure the life events system to ask a citizen's permission before life event information is sent to remote systems. A standard life event that just sends information to remote systems can be configured through the administration application. For more information, see *Defining Remote Systems*.

If the life event is a round tripping life event or is needed to update evidence in the local case processing system, then some development work is needed to configure the life event. Round tripping life events must be pre-populated. Pre-population of life events is only supported for users that are linked to the

local IBM Cúram Social Program Management case processing system by using a concern role. To read information from cases and update those cases, the life events system relies on the Citizen Data Hub subsystem. The following information outlines the work that is needed to configure the Citizen Data Hub.

The life event broker uses the Data Hub to get the data it needs to populate the life event, so you must configure the Data Hub to extract this data. The life event Broker also sends the updated data back through the Data Hub. The Data Hub must be configured to tell it what to do with this updated data.

You can use some of these artifacts to configure the Citizen Data Hub for reading information:

- Transform - converts data from the Holding Case into data store XML
- Filter Evidence Links - When you read Citizen Data, these links filter out only the evidence entities of interest when reading from the Holding Case.
- View Processors - Java classes for extracting non-evidence data into the data store XML

These are some of the artifacts that are used to configure the Citizen Data Hub for updating information:

- Transforms - Convert a data store XML Difference Description back into Holding Case Evidence
- Update Processors - Do other update tasks or update non-evidence data that relates to citizens

Considerations for life events analysis

The considerations that affect the complexity of developing a particular life event that must read from, or write to, an evidence or participant-related data store in IBM Cúram Social Program Management. These considerations inform any analysis of life events development and any resulting estimates.

1. Is the life event a standard life event or a round tripping life event
2. What information needs to be pre-populated into the IEG script?
3. What evidence data is read by the life event?
4. What evidence data is updated by the life event?
5. What non-evidence data is read/updated by the life event
6. How many programs or case types are affected by the life event
7. If a life event shares to multiple cases, will those case types also share evidence with each other using Evidence Broker?
8. Does a life event have associated recommendations? If so, do they relate to Community Services, Government Programs or both?

Of these items that deal with Non-Evidence Entities presents the greatest challenge. Any life event that updates non-evidence entities require developers with Java skills.

Building the components of a life event

How you build the component parts of a life event that uses the Citizen Data Hub. This information does not require any knowledge of Java.

Writing life event IEG Scripts

Writing a life event IEG script is similar to writing any other IEG script. However, there are special considerations for life event scripts. These considerations depend on whether the life event is a round tripping life event or a standard life event.

For a round tripping life event, citizen data is read into the data store that is used by the IEG script. This data can be modified by citizens as they progress from page to page in the life event script. For example, a citizen can modify income data that is read into the life event script before it's submitted. The life event broker ensures that when the citizen changes the income data the changes are detected and propagated correctly back to the income entity from which the data was read. The life event broker needs a way to track data from its origin in the income entity, through the life event script, and back to the same income entity. To facilitate this process, the IEG script designer needs to place a marker into the data store schema.

The following code block is an example of the definition of an income data store:

```
1 <xsd:element name="Income">
  <xsd:complexType>
    <xsd:attribute name="incomeType" type="INCOME_TYPE"
      default="" />
5    <xsd:attribute name="cgissIncomeType"
      type="CGISS_INCOME_TYPE" />
    <xsd:attribute name="incomeFrequency"
      type="INCOME_FREQUENCY" default="" />
    <xsd:attribute name="incomeAmount" type="IEG_MONEY"
10    default="0" />
    <xsd:attribute name="localID" type="IEG_STRING" />
  <xsd:complexType>
</xsd:element>
```

The life event broker uses the attribute `localID` to track the unique identity of the entity from which the income data was drawn. When this entity is changed and submitted, the life event broker can use the value of `localID` to locate the correct entity to update as a result of the changes in the life event. Other special markers exist that can be placed in the schema to aid with providing automatic updates to evidence entities.

When you design a script for a round tripping life event, you must account for the effects that pre-population of data can have on the flow of the script. One example of this situation is conditional clusters. Life event scripts need to avoid conditional clusters that are associated with pre-populated data. These clusters are common in intake scripts but don't work well when the data store was pre-populated. For example, for a life event involving the loss of a job, a Boolean flag on the Person entity, `hasJob` is used to indicate that person has a job. The IEG script presents the user with a question: Does anyone in your household have a job?. This question is used to drive the display of a conditional cluster that identifies which household members who have jobs.

However, if the data in the data store is repopulated, it's likely one or more Person entities with `hasJob` already be set to `true`. In the current implementation of IEG, it isn't possible to get the Does anyone in your household have a job? control question to default to `true` even when `hasJob` is `true` for one or more household members. For this reason, the rule needs to be to avoid control questions for conditional clusters such as when the fields they control are pre-populated.

Pre-populating a life event

A description of the artifacts that you must develop to pre-populate a life event script:

How the Data Hub works for reading

You can use the Data Hub to collect data about Citizens from different locations and return the data as an XML document in a datastore. You can use the Data Hub to hide the complexities of where data comes from and how it is represented in its original locations. For example, to drive a "Lost my Job" life event, you might need to gather information about a person's Income, Address, and Employment. These three pieces of information might be represented differently on the underlying system, they might be on one or more different systems. The caller doesn't need to know this detail. The Citizen Data Hub can get these pieces of information in one single operation. Operations of this type are named uniquely, each is called a "Data Hub Context". To animate the "Lost my Job" example, define a Data Hub Read Context called "CitizenLostJob" that enables the collection of Income, Address and Employment information in a single query.

One of the sources that the Data Hub can draw on is Evidence on Cases. In particular, Evidence on the Citizen's Holding Case. The Holding Case can use the Evidence Broker to gather data from many disparate Integrated Cases or even from other systems through web services. The Holding Case is a little different from other Cases. There is only one Holding Case per Citizen on a given IBM Cúram Social Program Management system. The Holding Case has an interface that allows all of the Evidence it contains to be extracted in XML format. This XML format is optimized for the description of Evidence in particular. Because it is optimized for the description of Evidence, it isn't necessarily in a format that is suitable for insertion into a data store. Fortunately it is relatively easy to translate data in one XML format into another format with XSLT. For more information about XSLT, see <http://www.w3.org/TR/xslt>.

Authoring Read Transforms

You can write XSLT Transforms for use in the Data Hub. To write Citizen Data Hub Transforms, you must understand the structure of the Holding Evidence XML that is the source data and the Data Store schema that is the target.

For example, a simple life event for Citizens who have bought a new car is associated with the Data Hub Context "CitizenBoughtCar". Look at the following fragment of Holding Evidence XML that is used to describe a Vehicle:

```
<?xml version="1.0" encoding="UTF-8"?>
<client-data
  xmlns="http://www.curamsoftware.com/schemas/ClientEvidence">
  <client localID="101" isPrimaryParticipant="true">
    <evidence>
      <entity localID="-416020015578349568" type="ET10081">
        <attribute name="vehicleMake">VM2</attribute>
        <attribute name="versionNo">2</attribute>
        <attribute name="startDate">20110301</attribute>
        <attribute name="usageCode">VU1</attribute>
        <attribute name="amountOwed">3,200.00</attribute>
        <attribute name="numberOfDoors">0</attribute>
        <attribute name="evidenceID">
          -5315936410157449216
        </attribute>
        <attribute name="monthlyPayment">0.00</attribute>
        <attribute name="vehicleModel">159</attribute>
        <attribute name="year">2008</attribute>
        <attribute name="equityValue">0.00</attribute>
        <attribute name="endDate">10101</attribute>
        <attribute name="fairMarketValue">17,000.00</attribute>
        <attribute name="curamEffectiveDate">20110301
        </attribute>
      </entity>
    </evidence>
  </client>
</client-data>
```

The client element represents data that belongs to the participant with concern role id 101. In Social Program Management demo data this is James Smith. The client contains a single evidence entity of type ET10081. In the Social Program Management Common Evidence layer, ET10081 is the Evidence Type identifier for Vehicle Evidence. The localID attribute plus the evidence type uniquely identifies the underlying evidence object for the Vehicle. This data must be mapped to data store XML so that it can be used to populate an IEG Script. Consider how the previous data is represented in data store XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Application>
  <Person localID="101" isPrimaryParticipant="true"
    hasVehicle="true">
    <Resource resourcePageCategory="RPC4001"
      localID="-416020015578349568" vehicleMake="VM2"
      versionNo="2" amountOwed="3,200.00" vehicleModel="159"
      year="2008" fairMarketValue="17,000.00"
      curamEffectiveDate="20110301">
      <Descriptor/>
    </Resource>
  </Person>
</Application>
```

This XML data must conform to the schema that is used to build the IEG script. Notice that the data store XML conforms to a schema that is a superset of the `CitizenPortal.xsd` schema. You can use the `CitizenPortal.xsd` schema as a starting point for the schemas used in Customer life events and add "marker" attributes that are needed for life events. These marker attributes include the use of localID. Datastore schemata for entities can also include the following special markers that are specialized for representing Evidence in the Holding Case:

- `curamEffectiveDate` - This maps to the effective date of a piece of Social Program Management Evidence

The following XSLT fragment shows how to transform Vehicle Holding Evidence into a corresponding Data Store Entity:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:x="http://www.curamsoftware.com/
  schemas/DifferenceCommand"
  xmlns:fn="http://www.w3.org/2006/xpath-functions"
  version="2.0">
  <xsl:output indent="yes"/>

  <xsl:strip-space elements="*" />

  <xsl:template match="update">
    <xsl:for-each select="./diff[@entityType='Application']">
      <xsl:element name="client-data">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:for-each>
  </xsl:template>

  <xsl:template match="diff[@entityType='Person']">
    <xsl:element name="client">
      <xsl:attribute name="localID">
        <xsl:value-of select="./@identifier"/>
      </xsl:attribute>
      <xsl:element name="evidence">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:element>
  </xsl:template>

  <xsl:template match="diff[@entityType='Resource']">
    <xsl:element name="entity">

      <xsl:attribute name="type">ET10081</xsl:attribute>
      <xsl:attribute name="action">
        <xsl:value-of select="./@diffType"/>
      </xsl:attribute>
      <xsl:attribute name="localID">
        <xsl:value-of select="./@identifier"/>
      </xsl:attribute>
      <xsl:for-each select="./attribute">
        <xsl:copy-of select="."/>
      </xsl:for-each>

    </xsl:element>
  </xsl:template>

  <xsl:template match="*">
    <!-- do nothing -->
  </xsl:template>
</xsl:stylesheet>
```

Adding this transform to your life event can turn Vehicle Evidence recorded on any Integrated Case into a Data Store format that can be displayed in an IEG script with all the information pre-populated from the Evidence Record.

Defining Filters for Evidence

When the Holding Case is called to return an XML representation of its evidence, by default it returns all evidence for the citizen concerned. This can be a very large query that returns more information than is required. For each Data Hub Context, use a Filter Evidence Link to define, which Evidence Types you need. Define a Filter Evidence Link by adding entries to a Filter Evidence Link dmx file. The following example shows a Filter Evidence Link dmx file that defines the information that to be returned for the "CitizenBoughtCar" life event:

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="FILTEREVIDENCELINK">
  <column name="FILTEREVLINKID" type="id" />
  <column name="FILTERNAME" type="text" />
  <column name="EVIDENCETYPECODE" type="text" />
</table>
```

```

        <attribute name="FILTEREVLINKID">
          <value>175</value>
        </attribute>
        <attribute name="FILTERNAME">
          <value>CitizenBoughtCar</value>
        </attribute>
        <attribute name="EVIDENCETYPECODE">
          <value>ET10081</value>
        </attribute>
      </row>
    </table>

```

Using Pre-Packaged View Processors

You now know how Transforms can be used to turn Evidence data into Data store XML for use in a life event Script. However, other important pieces of information are not represented as Evidence. In general, you must develop custom Java code to populate any information that is not represented as evidence. With Java, you can develop *View Processors* that can be used to extract non-evidence data and translate this data into data store XML. By associating these View Processors with the right Data Hub Context, they can add their information into the data store in addition to the data put there by the transforms. The Life Events Broker ships with some pre-packaged View Processors that are capable of inserting certain frequently used non-evidence Data.

- Household View Processor
- The Person Address View Processor

The Household View Processor finds all Persons that are related to the currently logged-in user and pulls them into the data store with information on how they are related to the logged-in Citizen. This information is based on the IBM Cúram Social Program Management Platform ConcernRoleRelationship entity.

The Person Address View Processor populates the most important details of the logged-in Citizen, such as name and Social Security Number. It also pulls in the Residential and Mailing addresses of the logged-in Citizen. Both the Household View processor and the Person Address View Processor can be used together in the same life event Context but the Person Address View Processor must be run after the Household View Processor. The following example shows how to configure these two View Processors to execute for the "CitizenBoughtCar" life event.

```

<?xml version="1.0" encoding="UTF-8"?>
<table name="VIEWPROCESSOR">
  <column name="VIEWPROCESSORID" type="id"/>
  <column name="LOGICALNAME" type="text" />
  <column name="CONTEXT" type="text" />
  <column name="VIEWPROCESSORFACTORY" type="text" />
  <column name="RECORDSTATUS" type="text"/>
  <column name="VERSIONNO" type="number"/>
  <row>
    <attribute name="VIEWPROCESSORID">
      <value>4</value>
    </attribute>
    <attribute name="LOGICALNAME">
      <value>CitizenLostJob0</value>
    </attribute>
    <attribute name="CONTEXT">
      <value>CitizenBoughtCar</value>
    </attribute>
    <attribute name="VIEWPROCESSORFACTORY">
      <value>
        curam.citizen.datahub.internal.impl.
        +HouseholdCustomViewProcessorFactory
      </value>
    </attribute>
    <attribute name="RECORDSTATUS">
      <value>RST1</value>
    </attribute>
    <attribute name="VERSIONNO">
      <value>1</value>
    </attribute>
  </row>
  <row>
    <attribute name="VIEWPROCESSORID">
      <value>5</value>
    </attribute>

```

```

</attribute>
<attribute name="LOGICALNAME">
  <value>CitizenLostJob1</value>
</attribute>
<attribute name="CONTEXT">
  <value>CitizenBoughtCar</value>
</attribute>
<attribute name="VIEWPROCESSORFACTORY">
  <value>
    curam.citizen.datahub.internal.impl.
    +CustomPersonAddressViewProcessorFactory
  </value>
</attribute>
<attribute name="RECORDSTATUS">
  <value>RST1</value>
</attribute>
<attribute name="VERSIONNO">
  <value>1</value>
</attribute>
</row>
</table>

```

The CONTEXT field links the ViewProcessor to the "CitizenBoughtCar" life event Context. This ensures that this ViewProcessor is called whenever the "CitizenBoughtCar" Data Hub Context is called. The use of a logicalName uniquely distinguishes each View Processor. View Processors for a Data Hub Context are executed in lexical order. A View Processor name with a logicalName of "AAA" for the Data Hub Context "CitizenBoughtCar" is executed before one with a logicalName of "AAB".

Driving updates from life events

A description of the artifacts that you need to develop to process the data submitted from a life event script.

How the Data Hub Works for Updating

The Citizen Data Hub also has Data Hub Contexts for updating. Life events typically use the same Data Hub Context name for the read and updates that are associated with the same life event. The example "CitizenBoughtCar" context describes a set of artifacts for pre-populating a "CitizenBoughtCar" life event script and also a set of artifacts for handling updates to the Citizen's data when the "CitizenBoughtCar" life event script is complete.

An update operation for a Citizen Data Hub Context can update multiple individual entities in a single transaction. The artifacts that are provided to a Data Hub following a script submission are:

- A Data Store root entity, which is the root of the data store that was updated by the life events IEG script.
- A Difference command, which is an entity that describes how this data store is different from the one that was passed to the IEG script before it was launched. In other words it describes how the user has changed the data as a result of executing the life event script. These differences are broken down into three basic types:
 - Creations - The user has created a data store entity as a result of running the IEG script
 - Updates - The user has updated an entity as a result of running the IEG script
 - Removals - The user has removed an entity as a result of running the IEG script

Of these three, Creations and Updates are the most common. Allowing users to remove items in life events scripts is generally considered bad practice. Standard life events tend to be characterized by a number of Creations whereas Round Tripping life events tend to be a mixture of Creations and Updates. The Difference Command is generated automatically by the life event Broker after a life event is submitted.

- A Data Hub Context Name.

To turn a Data Hub Update Operation into automatic updates to evidence entities on the Holding Case, specify a Data Hub Update Transform. Where there is a requirement to update non-evidence entities, you must develop an Update Processor. These Update Processors involve Java code development.

Writing Transforms for Updating

Update Transforms, like Read Transforms are specified by using a simple XSLT syntax. To write Update Transforms, the author must understand both the input XML, and the output Evidence XML format. The following examples are built around a "CitizenHavingABaby" life event. This life event allows the user to report that they are due to have a baby. They can enter a number of unborn children to indicate, for example, that they are expecting twins. The user can also enter a due date and they can nominate a father for the unborn child. The father can be an existing case participant or someone else entirely. In the latter case they must enter name, address, Social Security Number etc. This life event is not a "Round Tripping" life event, it is concerned with the creation of new Evidence rather than the update of existing Evidence. The input to an Update Transform is an XML-based description of the Data Store Difference Command. Example difference command XML for the "CitizenHavingABaby" is shown:

```
<update>
  <diff diffType="NONE" entityType="Application">
    <diff diffType="NONE" entityType="Person" identifier="102">
      <diff diffType="CREATE" entityType="Pregnancy">
        <attribute name="numChildren">1</attribute>
        <attribute name="dueDate">20110528</attribute>
        <attribute name="curamDataStoreUniqueID">385</attribute>
      </diff>
    </diff>
    <diff diffType="UPDATE" entityType="Person" identifier="101">
      <attribute name="isFatherToUnbornChild">true</attribute>
      <attribute name="curamDataStoreUniqueID">399</attribute>
    </diff>
  </diff>
</update>
```

The difference command XML corresponds node-for-node with the data store XML. Each *diff* node describes how the corresponding data store entity was modified by running the IEG script. The *curamDataStoreUniqueID* attribute identifies which data store entity has changed. The *diffType* attribute identifies the nature of the change, for example CREATE, UPDATE, NONE, or REMOVE. Attributes that changed or were added to each data store entity are listed. In the previous example, the user registered a pregnancy for Linda Smith (concern role ID 102) with one unborn child, due on 28 May 2011. The father is listed as being James Smith (concern role ID 101). For more information about the difference command XML, see the schema in the Difference Command XML Schema. There are a couple of additional attributes and elements to use when you update the XML as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<client-data>
  <client localID="102">
    <evidence>
      <entity type="ET10074" action="CREATE" localID="">
        <attribute name="numChildren">1</attribute>
        <attribute name="dueDate">20110528</attribute>
        <entity-data entity-data-type="role">
          <attribute type="LG"/>
          <attribute roleParticipantID="102"/>
          <attribute
            entityRoleIDFieldName="caseParticipantRoleID"/>
        </entity-data>
        <entity-data entity-data-type="role">
          <attribute type="FAT"/>
          <attribute roleParticipantID="101"/>
          <attribute participantType="RL7"/>
          <attribute
            entityRoleIDFieldName="fahCaseParticipantRoleID"/>
        </entity-data>
      <entity type="ET10125" action="CREATE">
        <attribute name="comments">Unborn child 1</attribute>
        <entity-data entity-data-type="role">
          <attribute type="UNB"/>
          <attribute roleParticipantID="102"/>
          <attribute
            entityRoleIDFieldName="caseParticipantRoleID"/>
        </entity-data>
      </entity>
    </evidence>
  </client>
</client-data>
```


Note the use of the action attribute, which describes the action to be taken on the underlying evidence. For example, to create the evidence or to update existing evidence. The next section discusses the meaning of the <entity-data> element. The following example shows the XSLT used to transform the previous difference XML into the previous Evidence XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This script plucks out and copies all resource-related -->
<!-- entities from output built by the XMLApplicationBuilder -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:x="http://www.curamsoftware.com/
    schemas/DifferenceCommand"
  xmlns:fn="http://www.w3.org/2006/xpath-functions"
  version="2.0">
  <xsl:output indent="yes"/>
  <xsl:strip-space elements="*" />
  <xsl:template match="update">
    <xsl:for-each select="./diff[@entityType='Application']">
      <xsl:element name="client-data">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:for-each>
  </xsl:template>
  <xsl:template match="diff[@entityType='Person']">
    <xsl:element name="client">
      <xsl:attribute name="localID">
        <xsl:value-of select="./@identifier"/>
      </xsl:attribute>
      <xsl:element name="evidence">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:element>
  </xsl:template>
  <xsl:template match="diff[@entityType='Pregnancy']">
    <xsl:element name="entity">
      <xsl:attribute name="type">ET10074</xsl:attribute>
      <xsl:attribute name="action">
        <xsl:value-of select="./@diffType"/>
      </xsl:attribute>
      <xsl:attribute name="localID">
        <xsl:value-of select="./@identifier"/>
      </xsl:attribute>
      <xsl:for-each select="./attribute">
        <xsl:copy-of select="."/>
      </xsl:for-each>
      <xsl:element name="entity-data">
        <xsl:attribute name="entity-data-type">
          role
        </xsl:attribute>
        <xsl:element name="attribute">
          <xsl:attribute name="type">LG</xsl:attribute>
        </xsl:element>
        <xsl:element name="attribute">
          <xsl:attribute name="roleParticipantID">
            <xsl:value-of select="./@identifier"/>
          </xsl:attribute>
        </xsl:element>
        <xsl:element name="attribute">
          <xsl:attribute name="entityRoleIDFieldName">
            caseParticipantRoleID
          </xsl:attribute>
        </xsl:element>
      </xsl:element>
      <xsl:element name="entity-data">
        <xsl:attribute name="entity-data-type">
          role
        </xsl:attribute>
        <xsl:element name="attribute">
          <xsl:attribute name="type">FAT</xsl:attribute>
        </xsl:element>
        <xsl:for-each select="
          './../diff[@entityType='Person']/attribute[
            @name='isFatherToUnbornChild'
            and ./text()='true']">
          <!-- Copy the participant id if a family -->
          <!-- member is the father -->
          <xsl:element name="attribute">
            <xsl:attribute name="roleParticipantID">
              <xsl:value-of select="
                ./@identifier"/>
            </xsl:attribute>
          </xsl:element>
        </xsl:for-each>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

```

        </xsl:element>
      </xsl:for-each>
      <!-- Copy details of absent parent -->
      <xsl:call-template name="absentFather"/>
      <xsl:element name="attribute">
        <xsl:attribute name="entityRoleIDFieldName">
          fahCaseParticipantRoleID
        </xsl:attribute>
      </xsl:element>
    </xsl:element>
    <xsl:variable name="numBabies">
      <xsl:value-of select="attribute[
        @name='numChildren'
      ]/text()"/>
    </xsl:variable>
    <xsl:call-template name="unbornChildren">
      <xsl:with-param name="count" select="$numBabies"/>
    </xsl:call-template>
  </xsl:element>
</xsl:template>

<xsl:template name="unbornChildren">
  <xsl:param name="count" select="1"/>
  <xsl:if test="$count > 0">
    <xsl:element name="entity">
      <xsl:attribute name="type">ET10125</xsl:attribute>
      <xsl:attribute name="action">
        <xsl:value-of select="./@diffType"/>
      </xsl:attribute>
      <xsl:element name="attribute">
        <xsl:attribute name="name">
          comments
        </xsl:attribute>
        Unborn child <xsl:value-of select="$count"/>
      </xsl:element>
      <xsl:element name="entity-data">
        <xsl:attribute name="entity-data-type">
          role
        </xsl:attribute>
        <xsl:element name="attribute">
          <xsl:attribute name="type">
            UNB
          </xsl:attribute>
        </xsl:element>
        <xsl:element name="attribute">
          <xsl:attribute name="
            roleParticipantID">
            <xsl:value-of select="
              ../@identifier"/>
          </xsl:attribute>
        </xsl:element>
        <xsl:element name="attribute">
          <xsl:attribute name="
            entityRoleIDFieldName">
            caseParticipantRoleID
          </xsl:attribute>
        </xsl:element>
      </xsl:element>
    </xsl:element>
    <xsl:call-template name="unbornChildren">
      <xsl:with-param name="count" select="$count - 1"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

<xsl:template name="absentFather">
  <xsl:element name="attribute">
    <xsl:attribute name="participantType">
      <xsl:text>RL7</xsl:text>
    </xsl:attribute>
  </xsl:element>

  <xsl:if test="attribute[@name='fahFirstName']">
    <xsl:element name="attribute">
      <xsl:attribute name="firstName">
        <xsl:value-of select="attribute[
          @name='fahFirstName'
        ]/text()"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:if>

```

```

<!-- etc. map other personal details such as -->
<!-- SSN, date of birth -->

<xsl:if test="diff[@entityType='ResidentialAddress']">
  <xsl:if test="diff[
    @entityType='ResidentialAddress']/attribute[
      @name='street1']">
    <xsl:element name="attribute">
      <xsl:attribute name="street1">
        <xsl:value-of select=
          "diff[
            @entityType='ResidentialAddress'
          ]/attribute[
            @name='street1']/text()"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:if>
  <!-- etc. map other parts of residential address -->
</xsl:if>
</xsl:template>

<xsl:template match="*">
  <!-- do nothing -->
</xsl:template>
</xsl:stylesheet>

```

Writing Transforms that create new case participants

Evidence Entities frequently refer to third parties. For example, Pregnancy evidence refers to the father through a Case Participant Role. The associated father can be a Person or a Prospect Person. Other evidence types, such as Student, can refer to a School that is entered as a Representative Case Participant Role.

The Evidence XML schema provides a generic element that is called `<entity-data>`, which can be used to provide special handling instructions to the Citizen Data Hub. The type of handling depends on the `<entity-data-type>` specified. IBM Cúram Social Program Management provides a special processor for the entity-data-type role. This role entity data processor can be used to create new Case Participant Roles or to reference existing Case Participant Roles for existing Case Participants. In the Evidence XML output listed previously, the attribute that is denoted by type is used to denote the Case Participant Role Type. For example, FAT for Father or UNB for Unborn Child. This value must be a code-table value from the CaseParticipantRoleType code table. The roleParticipantID denotes the ConcernRoleID of an existing participant on the system. If the roleParticipantID is supplied, the system does not attempt to create a new Case Participant, but reuses a case participant with this ID. The entityRoleIDFieldName is the field name in the corresponding Evidence Entity. For example, for the Pregnancy evidence entity, it is `fahCaseParticipantRoleID`. Where a new participant needs to be created, the following fields are supported by the Role Entity Data Processor.

- participantType - this is a code table entry from the ConcernRoleType code table. For example, use RL7 to create a new Prospect Person
- firstName
- middleInitial
- lastName
- SSN
- dateOfBirth
- lastName
- lastName
- street1
- city
- state
- zipCode

Updating Non-Evidence Entities

You can configure life events to automatically map updates through to Evidence Entities on multiple integrated cases. Sometimes life events must update non-Evidence entities such as a Residential Address, Employment, or other customer-specific non-evidence entities. Typically, these entities are shared across multiple cases. It is also typical that these entities do not follow the same controlled Life Cycle as evidence entities. Evidence has many advantages:

- It is temporal.
- It is case-specific, with sharing of updates between cases being controlled through the Evidence Broker.
- Caseworkers can veto acceptance of updates that come from external sources like IBM Cúram Universal Access.
- It has an in-edit and approval cycle.
- It has support for verifications.

Non-evidence entities have none of these advantages and safeguards. A decision by analysts to update non-evidence entities based on life events must be made with due care, especially if the changes can be applied simultaneously across multiple cases. You can update non-evidence entities but this approach always involves custom code. It is recommended that the design of such functionality includes safeguards to ensure that at least one Agency worker manually approves changes before they are applied to the system.

Configuring the evidence broker for use with the holding case

The Holding Case is only a holding area for Evidence before it is sent somewhere else. Normally, after data is updated on the Holding Case, the goal is to broker these updates to Integrated Cases so that caseworkers can evaluate the changes and apply them to the relevant cases.

For example, after the data is accepted on Integrated Cases, a user can see the positive impact of submitting a life event because the updated data has an impact on the user's benefits. The bridge between the Holding Case and the Integrated Cases is crossed only if the appropriate Evidence Broker configuration is defined. For more information about the Evidence Broker, see the *Evidence Broker Developers Guide*.

Configuring sharing from the Holding Case

An evidence configuration for sharing of Pregnancy evidence from the Holding Case to an Integrated Case is shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
  <table name="EVIDENCEBROKERCONFIG">
    <column name="EVIDENCEBROKERCONFIGID" type="id"/>
    <column name="SOURCETYPE" type="text" />
    <column name="SOURCEID" type="id" />
    <column name="TARGETTYPE" type="text" />
    <column name="TARGETID" type="id"/>
    <column name="SOURCEEVIDENCETYPE" type="text"/>
    <column name="TARGETEVIDENCETYPE" type="text"/>
    <column name="AUTOACCEPTIND" type="bool"/>
    <column name="WEBSERVICESIND" type="bool"/>
    <column name="SHAREDTYPE" type="text"/>
    <column name="RECORDSTATUS" type="text"/>
    <column name="VERSIONNO" type="number"/>
  </table>
  <row>
    <attribute name="EVIDENCEBROKERCONFIGID">
      <value>10003</value>
    </attribute>
    <attribute name="SOURCETYPE">
      <value>CT10301</value>
    </attribute>
    <attribute name="SOURCEID">
      <value>10330</value>
    </attribute>
    <attribute name="TARGETTYPE">
      <value>CT5</value>
    </attribute>
    <attribute name="TARGETID">
```

```

        <value>4</value>
      </attribute>
      <attribute name="SOURCEEVIDENCETYPE">
        <value>ET10000</value>
      </attribute>
      <attribute name="TARGETEVIDENCETYPE">
        <value>ET10074</value>
      </attribute>
      <attribute name="AUTOACCEPTIND">
        <value>0</value>
      </attribute>
      <attribute name="WEBSERVICESIND">
        <value>0</value>
      </attribute>
      <attribute name="SHAREDTYPE">
        <value>SET2002</value>
      </attribute>
      <attribute name="RECORDSTATUS">
        <value>RST1</value>
      </attribute>
      <attribute name="VERSIONNO">
        <value>1</value>
      </attribute>
    </row>
  </table>

```

When evidence is shared from the Holding Case to another Integrated Case, the source type needs to be CT10301 and the source ID needs to be set to 10330. The source evidence type needs to be set to ET10000, which is the code for all Evidence that is stored in Holding Cases. Evidence of this type is known as Holding Evidence. The target evidence type in this case is ET10074. In Cúram Common Evidence, this particular designation identifies Pregnancy Evidence. The evidence sharing type needs to be set to SET2002, which is the code for Non-Identical Sharing.

Note: The AUTOACCEPTIND is set to 0. Always set this value to 0 when it is shared from a Holding Case to an Integrated Case. This setting means that a caseworker always sees any changes that come from the citizen's Holding Case.

If the caseworker agrees with the changes, the **Incoming Evidence** link of the **Integrated Case Evidence** page can be used to synchronize the data from the Holding Case in the normal way.

To establish Evidence Broker Configuration for a custom component, a DMX file must be created that contains the configuration that follows the previous example, for example, %SERVER_DIR%\components\Custom\data\initial\EBROKER_CONFIG.dmx

In sharing Holding Evidence to a Standard Evidence Entity like a Pregnancy, the Evidence Broker copies the Holding Evidence that contains the Pregnancy data into a new Pregnancy Evidence Record in the target Integrated Case. Holding Evidence is not standard Evidence. Holding Evidence is stored in an XML representation, so while the Holding Evidence is copied to the Target Evidence type, the Evidence Broker converts the XML data into standard Evidence data. To assist with this conversion process, it is necessary to supply metadata. An example of this metadata is illustrated in the following code block:

```

<?xml version="1.0" encoding="UTF-8"?>
<data-hub-config>
  <evidence-config package="curam.holdingcase.evidence">
    <entity name="HoldingEvidence" ev-type-code="ET10000">
      <attribute name="entityStruct">
        curam.citizen.datahub.holdingcase.holdingevidence.struct.
        +HoldingEvidenceDtls
      </attribute>
    </entity>
    <entity name="Pregnancy" ev-type-code="ET10074">
      <attribute name="entityStruct">
        curam.evidence.entity.struct.PregnancyDtls
      </attribute>
      <related-entity>
        <case-participant-role>
          <attribute name="linkAttribute">
            fahCaseParticipantRoleID
          </attribute>
        </case-participant-role>
        <case-participant-role>
          <attribute name="linkAttribute">
            caseParticipantRoleID
          </attribute>
        </case-participant-role>
      </related-entity>
    </entity>
  </evidence-config>
</data-hub-config>

```

```

        </case-participant-role>
    </related-entity>
</entity>
</evidence-config>
</data-hub-config>

```

The metadata describes each of the entities that can be copied to and from the Holding Case and an Integrated Case. The metadata describes the dtls structs that are used to build the target evidence. It also describes which of the attributes in Case Evidence refer to case participant roles. This information ensures that when the Holding Evidence is copied, it does not blindly copy case participant role identifiers from Holding Evidence. Instead, it looks for the equivalent case participant role ID on the target case and, if it does not exist, creates one.

This metadata is stored in an AppResource resource store key. The resource store key is identified by the Environment Property `curam.workspaceservices.datahub.metadata`. The initially configured value for this variable defaults to the value `curam.workspaceservices.datahub.metadata`. This variable points to default Holding Evidence Data Hub metadata. You can use the following steps to replace the default Holding Evidence Data Hub metadata with a custom version to support all Evidence Types that need to be brokered from the Holding Case to all Integrated Cases:

- Copy the contents of `%SERVER_DIR%\components\WorkspaceServices\data\initial\clob\DataHubMetaData.xml` to `%SERVER_DIR%\components\Custom\data\initial\clob\CustomDataHubMetaData.xml`
- Edit the contents of `CustomDataHubMetaData.xml` to describe all the Evidence Entities that need to be updated by the Data Hub.
- Create a file `%SERVER_DIR%\components\Custom\data\initial\APP_RESOURCES.dmx`. Add an entry to this file as shown as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<table name="APPRESOURCE">
  <column name="resourceid" type="id" />
  <column name="localeIdentifier" type="text"/>
  <column name="name" type="text"/>
  <column name="contentType" type="text"/>
  <column name="contentDisposition" type="text"/>
  <column name="content" type="blob"/>
  <column name="internal" type="bool"/>
  <column name="lastWritten" type="timestamp"/>
  <column name="versionNo" type="number"/>
  <row>
    <attribute name="resourceID">
      <value>10700</value>
    </attribute>
    <attribute name="localeIdentifier"> <value/>
    </attribute>
    <attribute name="name">
      <value>custom.datahub.metadata</value>
    </attribute>
    <attribute name="contentType">
      <value>text/plain</value>
    </attribute>
    <attribute name="contentDisposition"> <value>inline</value>
    </attribute> <
    attribute name="content"> <value> ./Custom/data/initial/clob/CustomDataHubMetaData.xml </
value>
    </attribute> <attribute name="internal"> <value>0</value> </attribute>
    <attribute name="lastWritten"> <value>SYSTIME</value>
    </attribute> <attribute name="versionNo"> <value>1</value>
    </attribute>
  </row>
</table>

```

- Create or append to the file `%SERVER_DIR%\components\Custom\properties\Environment.xml` adding an entry along the following lines:

```

<environment>
  <type name="dynamic_properties">
    <section code="WSSVCS"
      name="Workspace Services - Configuration">
      <variable name="curam.workspaceservices.datahub.metadata"
        value="custom.datahub.metadata" onlyin="all"

```

```

        type="STRING">
        <comment>
            Identifies an AppResource used to configure DataHub
            meta-data.
        </comment>
    </variable>
</section>
</type>
</environment>

```

Round tripping and configuring sharing to the Holding Case

The previous section described how data is shared from the Holding Case to Integrated Cases. Analysts also might want to consider whether evidence needs to be transferred in the opposite direction - that is, from the Integrated Cases to the Holding Case. When sharing is configured from the Integrated Case to the Holding Case, changes made by the caseworker to selected evidence can be propagated back to the Holding Case. This process is essential for life events that need to pre-populate data from Evidence Entities in existing Integrated Cases. This example shows how to configure Pregnancy Evidence for Sharing to the Holding Case.

```

<?xml version="1.0" encoding="UTF-8"?>
<table name="EVIDENCEBROKERCONFIG">
  <column name="EVIDENCEBROKERCONFIGID" type="id" />
  <column name="SOURCETYPE" type="text" />
  <column name="SOURCEID" type="id" />
  <column name="TARGETTYPE" type="text" />
  <column name="TARGETID" type="id" />
  <column name="SOURCEEVIDENCETYPE" type="text" />
  <column name="TARGETEVIDENCETYPE" type="text" />
  <column name="AUTOACCEPTIND" type="bool" />
  <column name="WEBSERVICESIND" type="bool" />
  <column name="SHAREDTYPE" type="text" />
  <column name="RECORDSTATUS" type="text" />
  <column name="VERSIONNO" type="number" />
  <row>
    <attribute name="EVIDENCEBROKERCONFIGID">
      <value>2</value>
    </attribute>
    <attribute name="SOURCETYPE">
      <value>CT5</value>
    </attribute>
    <attribute name="SOURCEID">
      <value>4</value>
    </attribute>
    <attribute name="TARGETTYPE">
      <value>CT10301</value>
    </attribute>
    <attribute name="TARGETID">
      <value>10330</value>
    </attribute>
    <attribute name="SOURCEEVIDENCETYPE">
      <value>ET10074</value>
    </attribute>
    <attribute name="TARGETEVIDENCETYPE">
      <value>ET10000</value>
    </attribute>
    <attribute name="AUTOACCEPTIND">
      <value>1</value>
    </attribute>
    <attribute name="WEBSERVICESIND">
      <value>0</value>
    </attribute>
    <attribute name="SHAREDTYPE">
      <value>SET2002</value>
    </attribute>
    <attribute name="RECORDSTATUS">
      <value>RST1</value>
    </attribute>
    <attribute name="VERSIONNO">
      <value>1</value>
    </attribute>
  </row>
</table>

```

Note: Unlike Sharing from Holding Case to Integrated Case, the AUTOACCEPTIND is set to 1. This designation is because the target case is a Holding Case and Holding Cases are designed to operate unattended. It is not expected that caseworkers need to review items that are being shared onto the Holding Case as they come from an authoritative source, for instance, the Integrated Case.

Issues for consideration

With suitable configuration, It is possible to share data from the Holding Case to many different Integrated Cases. Take the example of two different Integrated Cases (cases A and B) that are configured to share information with a citizen's Holding Case (case H). Both cases A and B separately recorded an Income Evidence record for the citizen. In the citizen's Holding Case, this evidence record shows up as two separate Income Records. As far as cases A and B are concerned, they are two entirely separate records - A's view of the citizen's Income and B's view of the citizen's Income. However, to the citizen, this breakdown might not make much sense. The citizen has only one Income and is using one Portal to communicate with the Social Enterprise Management (SEM) agency or agencies concerned. Why does the citizen see two records for the same Income? In cases where there is sharing to multiple Integrated Cases from a single Holding Case, consideration needs to be given to creating another set of sharing relationships to be established from A to B and B to A. This consideration is an issue that requires proper consideration early on in the project lifecycle.

Putting it all together

Previous topics showed how to create the parts of a life event, this topic discusses how to join all these pieces together to make a completed life event.

New life events can be configured in administration application. You can create new life event types and life event channels, add rich text descriptions, and associate life events with IEG scripts and recommendation rule sets. When all of the required Entities are created, the data can be extracted into a set of DMX files that can be used as a basis for ongoing development. The following set of commands can be used to extract the relevant DMX files:

```
build extractdata -Dtablename=LifeEventType
build extractdata -Dtablename=LifeEventContext
build extractdata -Dtablename=LifeEventCategory
build extractdata -Dtablename=LifeEventCategoryLink
build extractdata -Dtablename=LocalizableText
build extractdata -Dtablename=TextTranslation
```

The LocalizableText and TextTranslation tables contain all of the life event descriptions, but they also contain text translations that don't relate to life events. Developers must audit these DMX files and remove any entries that do not correspond to the relevant life event descriptions before copying the DMX files to %SERVER_DIR%\components\Custom\data\initial\.

Event APIs for life events

The life event broker is instrumented with guice events. Developers can write listeners that can be bound to these events. The available events are:

- `PreCreateLifeEvent` - Invoked before launching a life event
- `PostCreateLifeEvent` - Invoked after the life event has been initialized. That is after the Data Hub Transform and View Processors have been executed.
- `PreSubmitLifeEvent` - Invoked after the life event has been submitted but before the Update Processors have been run.
- `PostSubmitLifeEvent` - Invoked after the life event has been submitted.

Note that both the Pre and Post SubmitLifeEvent events are executed from within a Deferred Process so the current user is expected to be SYSTEM. Life events should never attempt to change the contents of the life event. The code extract below shows how a Listener class, `MyPreCreateListener` can be bound to one of these life events:

```
Multibinder<LifeEventEvents.PreCreateLifeEvent>
    preCreateBinder =
```



```

Multibinder.newSetBinder(binder(),
    new TypeLiteral<LifeEventEvents.PreCreateLifeEvent>() { /**/
    });

preCreateBinder.addBinding().to(MyPreCreateListener.class);

```

Customizing verifications

If your organization includes the online submission of documents in their business process, citizens can upload and submit documents from the IBM Universal Access Responsive Web Application to prove information that they provided in their benefit applications. You can customize a number of aspects of the verifications functionality in the application.

Your organization can integrate a verifications system of your choice. If you use the IBM Cúram Verification Engine application module, the verifications functionality is available in the Universal Access Responsive Web Application after you set up your IBM Cúram Social Program Management verifications.

Related concepts

Verify

If your organization includes the online submission of documents in their business process, citizens are notified in the Universal Access Responsive Web Application when some of their information needs to be verified with supporting documentation. They can then provide that supporting documentation online. Both citizens and caseworkers receive notifications, alerting them to any steps to take. Case workers control the verification of evidence, ensuring adherence to agency standards.

Related information

[Verification Engine](#)

Enabling or disabling verifications

The verifications feature is disabled by default. Use the

REACT_APP_FEATURE_VERIFICATIONS_ENABLED environment variable to enable or disable the **Your documents** page and options in your application. When you enable verifications, it is available only to linked users.

Before you begin

Before you enable verifications, you must ensure that you have implemented the appropriate file security and validations for document uploads and enabled the file upload API.

About this task

For more information about linked users, see [“User account types” on page 150](#).

The following verifications functionality is enabled or disabled:

- The verifications-related URLs, the **Your documents** page at `/verifications` and the verification details page at `/verifications/details`.
- The verification Alert on the **Dashboard** page.
- Verifications messages in the **To-dos** pane on the dashboard.

For more information about environment variables, see the [“React environment variable reference” on page 173](#).

Procedure

1. Ensure that you have implemented the appropriate file security and validations for document uploads, see [“Implementing file security and validation for the POST files API” on page 94](#).
2. Enable the file upload API, see [“Enabling the POST files API for file uploads” on page 95](#).
3. Edit the `.env` file in the root of your application and set **REACT_APP_FEATURE_VERIFICATIONS_ENABLED** to `true`. If you don't define the environment variable, the verifications feature defaults to disabled.

Related tasks

[Enabling the POST files API for file uploads](#)

By default, the Document Service POST `/v1/dbs/files` API is disabled for security purposes. Before you can safely upload documents, you must enable the API in the IBM Cúram Social Program Management system administration application.

[Implementing file security and validation for the POST files API](#)

You must have appropriate file security and validation for any documents that are uploaded to your system by the Document Service file upload API POST `/v1/dbs/files`. If you don't manage these functions elsewhere, such as in content management system or a gateway, then you can use the provided `RESTFileValidation` hook point. You can override the provided dummy implementation with your own custom implementation.

Customizing file formats and size limits for file uploads

You can specify which file formats to allow users to upload and a size limit for uploaded files by setting environment variables. By default, the allowed file formats are JPG, JPEG, PNG, TIFF, and PDF and the file size limit is 5 MB.

Before you begin

You must ensure that you have implemented the appropriate file security and validations for document uploads and enabled the file upload API.

If you do not set the `REACT_APP_DOC_UPLOAD_FILE_FORMATS` environment variable, the default file formats are allowed. If you specify an invalid file extension string, all file types are denied.

If you do not set the `REACT_APP_DOC_UPLOAD_SIZE_LIMIT`, the default value applies.

Procedure

1. To change the allowed file formats for file uploads, set the `REACT_APP_DOC_UPLOAD_FILE_FORMATS` environment variable in your `.env` file. Specify the file name extension, including the dot separator, for each allowed file type in a comma-separated list.

For example:

```
REACT_APP_DOC_UPLOAD_FILE_FORMATS=".png,.jpg,.pdf"
```

2. To change the allowed file sizes for file uploads, set the `REACT_APP_DOC_UPLOAD_SIZE_LIMIT` environment variable in your `.env` file. Enter the maximum size in megabytes (MB).

For example:

```
REACT_APP_DOC_UPLOAD_SIZE_LIMIT=6
```

Related tasks

[Implementing file security and validation for the POST files API](#)

You must have appropriate file security and validation for any documents that are uploaded to your system by the Document Service file upload API POST `/v1/dbs/files`. If you don't manage these functions elsewhere, such as in content management system or a gateway, then you can use the provided `RESTFileValidation` hook point. You can override the provided dummy implementation with your own custom implementation.

[Enabling the POST files API for file uploads](#)

By default, the Document Service POST `/v1/dbs/files` API is disabled for security purposes. Before you can safely upload documents, you must enable the API in the IBM Cúram Social Program Management system administration application.

Customizing a file upload lead time for verifications

If needed, your organization can configure a lead time to the due date so that document are submitted earlier to give caseworkers enough time to verify the evidence. Use the

REACT_APP_DOC_UPLOAD_LEAD_DAYS environment variable to set how many days you want to subtract from the actual date. This earlier date is then displayed to citizens in the application.

About this task

By default, the due date is the date when the information needs to be verified by the caseworker. The default value of REACT_APP_DOC_UPLOAD_LEAD_DAYS is 0 days. The value you set is converted to its absolute value and subtracted from the verification due date. For example, -1 and 1 have the same affect.

Procedure

To set a lead time, set the REACT_APP_DOC_UPLOAD_LEAD_DAYS environment variable in your .env file.

For example:

```
REACT_APP_DOC_UPLOAD_LEAD_DAYS=-7
```

If the actual due date is 31 August, then 24 August is displayed in the application.

Customizing how verification information is presented

The information for the majority of verifications that are presented to the citizen is processed and grouped by evidence records to present meaningful and consumable information in the UI. However, you might have a number of evidence types that contain disparate information that would be more meaningfully displayed in separate verifications. You can customize how information from evidence records is grouped and displayed in the application.

To understand how information is grouped by default, take an evidence record called Medical Expenses, which is an instance of a Medical expenses evidence type. An evidence record can consist of one or more verifiable data items. For example, the amount entered for a medical expense and it's frequency. If a citizen had multiple expenses, it might look like this:

Medical Expense:

- For Diabetes
 - Amount: \$100
 - Frequency: Weekly
- For Asthma
 - Amount: \$125
 - Frequency: Monthly

To show related expenses that need documentation, verifiable data items are grouped into bigger verifications. By default, information is grouped by the following five criteria in order of importance:

- Evidence type
- Evidence record
- Case
- Person
- Due date

Anything different in the list results in a separate verification.

The text that is displayed on the verification is taken from the evidence descriptor of the evidence record for which the verification was raised, for example **Paid \$100 for Diabetes**.

Customizing verifiable data item grouping

You can customize how verifiable data items on an evidence record are grouped for display in verifications. For example, to separate out disparate items that are on the same evidence record but that are not suitable to show on the same verification.

About this task

To customize the grouping, you must override the method with your own custom implementation and import your custom implementation. You can use the following example as a reference.

Procedure

1. Create a custom module that implements your custom grouping.

For example:

```
src/features/Verification/GroupIdSelectorWrapper.js
```

2. In your custom module, import the original selectors as follows:

```
import { CustomVerificationsSelectors, VerificationsSelectors } from '@spm/universal-access';
```

3. Create a config function that overrides the default functions.

```
function config(){
  CustomVerificationsSelectors.fetchVerifications = state => {}
  CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {}
}
```

4. To group by ID, override `CustomVerificationsSelectors.fetchVerifications` by using the `VerificationsBuilder` and passing the `groupByVerificationId` function as a parameter to the builder.

```
function config(){
  CustomVerificationsSelectors.fetchVerifications = state => {
    const verificationsData = VerificationsSelectors.fetchVerifications(state);
    if (!verificationsData || !verificationsData.data) return null;
    // group by evidence type and due date
    const groupByVerificationId = verification => {
      const {
        relatedEvidence: {
          type: { value },
        },
        dueDate,
      } = verification;
      return `${value}-${dueDate}`;
    };
    const { VerificationsBuilder } = CustomVerificationsSelectors;
    return new VerificationsBuilder(verificationsData.data, groupByVerificationId)
      .withDueDate()
      .withRelatedPerson()
      .withRelatedEvidenceType()
      .withRelatedCase()
      .withAcceptedProofs()
      .withRequiredProofs()
      .withProvidedFiles()
      .withExpiryDate()
      .withStatus()
      .buildAsArray();
  };
  CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {}
}
```

5. Override the `CustomVerificationsSelectors.fetchVerificationsById` to use the custom function.

```
CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {
  const verificationsGroups = CustomVerificationsSelectors.fetchVerifications(state);
  // validate if exist verifications group
  if (!verificationsGroups || !verificationsGroups.length > 0) return null;
  // find group id
  return verificationsGroups.find(verificationDetailsItem => {
    return verificationDetailsItem.groupID === groupId;
  });
}
```

```
});
};
```

Your `GroupIdSelectorWrapper.js` file now looks like this:

```
import { CustomVerificationsSelectors, VerificationsSelectors } from '@spm/universal-access';
function config() {
  CustomVerificationsSelectors.fetchVerifications = state => {
    const verificationsData = VerificationsSelectors.fetchVerifications(state);
    if (!verificationsData || !verificationsData.data) return null;
    // group by evidence type and due date
    const groupByVerificationId = verification => {
      const {
        relatedEvidence: {
          type: { value },
        },
        dueDate,
      } = verification;
      return `${value}-${dueDate}`;
    };
  };
  const { VerificationsBuilder } = CustomVerificationsSelectors;
  return new VerificationsBuilder(verificationsData.data, groupByVerificationId)
    .withDueDate()
    .withRelatedPerson()
    .withRelatedEvidenceType()
    .withRelatedCase()
    .withAcceptedProofs()
    .withRequiredProofs()
    .withProvidedFiles()
    .withExpiryDate()
    .withStatus()
    .buildAsArray();
};
CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {
  const verificationsGroups = CustomVerificationsSelectors.fetchVerifications(state);
  // validate if exist verifications group
  if (!verificationsGroups || !verificationsGroups.length > 0) return null;
  // find group id
  return verificationsGroups.find(verificationDetailsItem => {
    return verificationDetailsItem.groupID === groupId;
  });
};
}
export default { config };
```

6. Update your `App.js` file or equivalent as follows:

```
import GroupByIdSelectorWrapper from '../features/Verification/GroupIdSelectorWrapper';
GroupIdSelectorWrapper.config();
```

Customizing verification names

You can customize the name of a verification if the name from the evidence type is not suitable.

Procedure

1. Create a custom module that implements your custom naming.

For example:

```
src/features/Verification/ChangeNameSelectorWrapper.js
```

2. In your custom module, import the original selectors as follows:

```
import { CustomVerificationsSelectors, VerificationsSelectors } from '@spm/universal-access';
```

3. Create a `config` function that overrides the default functions.

```
function config(){
  CustomVerificationsSelectors.fetchVerifications = state => {}
  CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {}
}
```

4. Override `CustomVerificationsSelectors.fetchVerifications` by using the `VerificationsBuilder` and use the `VerificationsBuilder.mapVerificationGroup` function to change the name.

```
function config() {    CustomVerificationsSelectors.fetchVerifications = state => {
    const verificationsData = VerificationsSelectors.fetchVerifications(state);
    if (!verificationsData || !verificationsData.data) return null;
    const { VerificationsBuilder } = CustomVerificationsSelectors;
    return new VerificationsBuilder(verificationsData.data)
        .withDueDate()
        .withRelatedPerson()
        .withRelatedEvidenceType()
        .withRelatedCase()
        .withAcceptedProofs()
        .withRequiredProofs()
        .withProvidedFiles()
        .withExpiryDate()
        .withStatus()
        .mapVerificationGroup(group => {
            const { relatedEvidenceType } = group;
            // if the code of the name is "DET106" change the description to "Health expenses"
            if (relatedEvidenceType && relatedEvidenceType.value === 'DET106') {
                // add description to "Health expenses"
                const newRelatedEvidenceType = { ...relatedEvidenceType, description: 'Health
expenses' };
                // return a group with the name modified
                return { ...group, relatedEvidenceType: newRelatedEvidenceType };
            }
            return group;
        })
        .buildAsArray();
    };    CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {}
}
```

5. Override the `CustomVerificationsSelectors.fetchVerificationsById` to use the custom function.

```
CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {
    const verificationsGroups = CustomVerificationsSelectors.fetchVerifications(state);
    // validate if exist verifications group
    if (!verificationsGroups || !verificationsGroups.length > 0) return null;
    // find group id
    return verificationsGroups.find(verificationDetailsItem => {
        return verificationDetailsItem.groupID === groupId;
    });
};
```

Your `ChangeNameSelectorWrapper.js` file now looks like this:

```
import { CustomVerificationsSelectors, VerificationsSelectors } from '@spm/universal-access';
function config() {
    // Override the CustomVerificationsSelectors.fetchVerifications method
    CustomVerificationsSelectors.fetchVerifications = state => {
        const verificationsData = VerificationsSelectors.fetchVerifications(state);
        if (!verificationsData || !verificationsData.data) return null;
        const { VerificationsBuilder } = CustomVerificationsSelectors;
        return new VerificationsBuilder(verificationsData.data)
            .withDueDate()
            .withRelatedPerson()
            .withRelatedEvidenceType()
            .withRelatedCase()
            .withAcceptedProofs()
            .withRequiredProofs()
            .withProvidedFiles()
            .withExpiryDate()
            .withStatus()
            .mapVerificationGroup(group => {
                const { relatedEvidenceType } = group;
                // if the code of the name is "DET106" change the description to "Health expenses"
                if (relatedEvidenceType && relatedEvidenceType.value === 'DET106') {
                    // add description to "Health expenses"
                    const newRelatedEvidenceType = { ...relatedEvidenceType, description: 'Health
expenses' };
                    // return a group with the name modified
                    return { ...group, relatedEvidenceType: newRelatedEvidenceType };
                }
                return group;
            })
    }
}
```

```

        .buildAsArray();
    };
    CustomVerificationsSelectors.fetchVerificationsById = (state, groupId) => {
        const verificationsGroups = CustomVerificationsSelectors.fetchVerifications(state);
        // validate if exist verifications group
        if (!verificationsGroups || !verificationsGroups.length > 0) return null;
        // find group id
        return verificationsGroups.find(verificationDetailsItem => {
            return verificationDetailsItem.groupID === groupId;
        });
    };
}
export default { config };

```

6. Update your App.js file or equivalent as follows:

```

import ChangeNameSelectorWrapper from './features/Verification/ChangeNameSelectorWrapper';
ChangeNameSelectorWrapper.config();

```

Customizing caseworker tasks

When a citizen submits a document for a verification, a task is generated for the caseworker. Tasks are displayed to the caseworker that is assigned to the citizen's case when they log in to the caseworker application. System administrators can configure the system to display a task each time a citizen provides all documents for an individual evidence record on a case, or to display the task only when a citizen has provided all documents for every evidence record on a case.

About this task

By default, the task is displayed only when all documents are uploaded for all evidence records on the case.

Procedure

1. Log on to the IBM Cúram Social Program Management application as a system administrator, and click **System Configurations**.
2. In the **Shortcuts** pane, click **Application Data > Property Administration**.
3. Configure the **curam.citizenworkspace.task.notifications.on.all.evidence.uploads** property.

Related concepts

Caseworker tasks

When documents are submitted for verification by a citizen, a task is generated for the caseworker that is assigned to the citizen's case.

Customizing appeals

You can customize appeals to suit your organization. You can integrate with an appeals system of your choice. If you are licensed for the IBM Cúram Appeals application module, the IBM Cúram Social Program Management appeals functionality is available on installation.

About this task

You can customize the following aspect of appeals:

- The **Your rights to appeal content** text on the dashboard.
- The **Your appeals** page. The **Appeals** page is shown only when a citizen has a case to appeal, otherwise it is not displayed.
- The Request an Appeal **Overview** page, from which you can start the **Request an Appeal** form.
- The **Request an Appeal** IEG script, in which you specify the contents of the form.
- The **Confirmation and next steps** page.

- The **Appeal** cards on the **Appeals** home page, which contain information about each appeal request that a user creates. Each card shows the status of the appeal request in a colored badge, with text such as **Appeal Request Submitted** or **Appeal Request Pending**. The color depends on the status. For example, **Appeal Request Submitted** is blue. You can customize the label text.

Procedure

1. The Appeals feature is unavailable by default. Enable Appeals in the application, see [“Enabling and disabling appeals”](#) on page 238.
2. Review the text on application pages. For more information about modifying text on pages, see [“Changing text in the application”](#) on page 111.
3. Review the **Request an Appeal** form. For more information, see [“Configuring appeal requests”](#) on page 56.
4. Review the **Appeal Request** cards on the **Your appeals** page, which show the appeals status. For more information about customizing the appeals statuses, see [“Customizing appeal request statuses”](#) on page 211.

Related concepts

Appeal

If your organization includes appeals in their business process, citizens can appeal decisions on their benefits online from their citizen accounts on their own devices. If your organization uses the IBM Cúram Appeals application module, your organization can process appeals through the full appeals life cycle that is provided by that solution.

Enabling and disabling appeals

The Appeals feature is disabled by default. Use the **REACT_APP_FEATURE_APPEALS_ENABLED** environment variable to enable or disable the Appeals pages and options in your application. When you enable Appeals, it is available only to linked users with an existing case that they can appeal.

About this task

For more information about linked users, see [“User account types”](#) on page 150.

The following Appeals functionality can be enabled or disabled:

- The **Appeals** tab on the home page.
- The **Appeals Request** page.
- **Your rights of appeal** message on the home page.
- Appeals-related URLs, for example /appeals.

Procedure

1. Edit the .env file in the root of your application.
2. Set **REACT_APP_FEATURE_APPEALS_ENABLED** to true or false. If you don't define the environment variable, the appeals feature defaults to enabled.

Artifacts with limited customization scope

A description of IBM Cúram Universal Access artifacts that have restrictions on their use. Customers that want to change these artifacts should consider alternatives or request an enhancement to Universal Access.

Model

Customers are not supported in making changes to any part of the Universal Access model. Changes in the model such as changing the data types of domains can cause failure of the Universal Access system and upgrade issues. This applies to the model files in the following packages:

- WorkspaceServices
- CitizenWorkspace
- CitizenWorkspaceAdmin

Code tables

See *Extending code tables* for a list of restricted code tables.

Related information

[Extending code tables](#)

Chapter 8. Deploying your web application to a web server

You can deploy your web application on a web server in a production-like environment as part of your development process. Deployment in a production environment is outside the scope of this documentation, but you can use the instructions in this section for guidance.

Building the Universal Access Responsive Web Application for deployment

Build Universal Access Responsive Web Application for deployment on an HTTP server.

Before you begin

For production builds, review all of the environment variables in your `.env` files, and check the order of the environment variables where you have multiple `.env` files. For more information about the priority of different `.env` files in `react-scripts`, see [What other .env files can be used?](#) in the Create React App documentation.

Procedure

1. To quickly configure the `universal-access-starter-pack` application for deployment, edit the `.env` configuration file in the root of your app, and modify the following environment variable to point to the server that hosts the REST services:

```
REACT_APP_REST_URL=<ServerHostName>:9044/Rest
```

Replace the `<ServerHostName>` and the port number with the hostname and port of the server where the REST services are deployed, for example:

```
REACT_APP_REST_URL=https://192.0.2.4:7002/Rest
```

2. Enter the following command to install dependent packages:

```
npm install
```

3. Enter the following command to build the application into a build folder in the `universal-access-starter-pack`:

```
npm run build
```

4. Copy the build folder to the HTTP Server and deploy. For more information about deploying the built application, see *Deploying your web application*.

Related information

[Deploying your application](#)

Install and configure IBM HTTP Server with WebSphere Application Server

Install and configure IBM HTTP Server either on the same server as WebSphere Application Server or on a remote server. To enable cross-origin resource sharing (CORS), you can set the **curam.rest.allowedOrigins** property for the REST application on your application server, or install the IBM HTTP Server plug-in for WebSphere Application Server.

Before you begin

WebSphere Application Server must be installed and configured.

Install IBM Installation Manager. For more information, see the [IBM Installation Manager documentation](#). You can download IBM Installation Manager from [Installation Manager and Packaging Utility download documents](#).

About this task

To enable cross-origin resource sharing (CORS), choose one of the following options:

- Set the **curam.rest.allowedOrigins** property for the REST application that is deployed on the application server. For more information about setting the **curam.rest.allowedOrigins** property, see [Cúram REST configuration properties](#).
- Install and configure the IBM HTTP Server plug-in for WebSphere Application Server to enable IBM HTTP Server to communicate with WebSphere Application Server. WebSphere Customization Toolbox is needed to configure the plug-in.

Procedure

1. Install IBM HTTP Server. For more information, see [Migrating and installing IBM HTTP Server](#).
2. Optional: If you don't set the **curam.rest.allowedOrigins** property, you must install the following software:
 - a) Install the IBM HTTP Server plug-in for WebSphere Application Server.
For more information, see [Installing and configuring web server plug-ins](#).
 - b) Install the WebSphere Customization Toolbox.
For more information, see [Installing and using the WebSphere Customization Toolbox](#).
3. Start IBM HTTP Server. For more information, see [Starting and stopping the IBM HTTP Server administration server](#).
4. To secure IBM HTTP Server, see [Securing IBM HTTP Server](#).

Generating an IBM HTTP Server plug-in configuration

This task is needed only if you install the IBM HTTP Server plug-in for WebSphere Application Server. Use WebSphere Customization Toolbox to generate a plug-in configuration.

Before you begin

Start WebSphere Application Server. For more information, see [Starting a WebSphere Application Server traditional server](#).

Procedure

To generate the IBM HTTP Server plug-in configuration, complete the steps at the [WebSphere Application Server Network Deployment plug-ins configuration topic](#).

Configuring the IBM HTTP Server plug-in

Configure the IBM HTTP Server plug-in for WebSphere Application Server and WebSphere Customization Toolbox. This task is necessary only if you have chosen to install the IBM HTTP Server plug-in, instead of setting the **curam.rest.allowedOrigins** property for the REST application that is deployed on the application server.

About this task

You can run the `configurewebserverplugin` target to complete the following tasks:

- Add the web server virtual hosts to the client hosts configuration in WebSphere Application Server.
- Propagate the plug-in key ring for the web server.
- Map the modules of any deployed applications to the web server.

Procedure

1. Start IBM HTTP Server.
For more information, see [Starting and stopping the IBM HTTP Server administration server](#).
2. On the remote WebSphere Application Server, run the following command.

```
build configurewebserverplugin -Dserver.name=server_name
```

The `configurewebserverplugin` target requires a mandatory `server.name` argument that specifies the name of the server when the target is invoked. For more information about the `configurewebserverplugin` target, see [Configuring a web server plug-in in WebSphere Application Server](#).

3. Consider adding extra aliases to the *client_host*, as shown in the following examples:

- For WebSphere Application Server, add port number 9044.
- For the default HTTP port, add port number 80.
- For HTTPS ports, add port number 433.

For more information about client host setup, see step 19 in the [WebSphere Application Server port access setup](#) topic.

4. To avoid port mapping issues from web applications, restart WebSphere Application Server and IBM HTTP Server.

For more information, see [Starting and stopping the IBM HTTP Server administration server](#).

Install and configure Oracle HTTP Server with Oracle WebLogic Server

Install and configure Oracle HTTP Server on either the same server as Oracle WebLogic Server or on a remote server.

Before you begin

Oracle WebLogic Server must be installed and configured. For more information, see [Installing and Configuring Oracle WebLogic Server and Coherence for Oracle HTTP Server 12.1.3](#), and [Installing and Configuring Oracle HTTP Server for Oracle HTTP Server 12.2.1.3](#).

Installing Oracle HTTP Server and its components

Install and configure Oracle HTTP Server in either a stand-alone domain, or in an Oracle WebLogic Server domain. You must install and configure an Oracle web server plug-in for proxying requests.

About this task

The Oracle web server plug-in allows requests to be proxied from Oracle HTTP Server to Oracle WebLogic Server. If you install and configure the Oracle web server plug-in, requests that are delegated to Oracle WebLogic Server still appear to originate from the Oracle HTTP Server, even if Oracle HTTP Server and Oracle WebLogic Server are hosted on two different servers.

Because of the web browser same-origin policy, cross-origin resource sharing (CORS) is restricted in many browsers by default. The web server plug-in enables CORS where Oracle HTTP Server and Oracle WebLogic Server are installed on different computers.

CORS enables an instance of your web application that is deployed on Oracle HTTP Server in one domain to request the REST services that are deployed on Oracle WebLogic Server in another domain.

Procedure

1. Install Oracle HTTP Server for Oracle WebLogic Server. For more information, see [Installing and Configuring Oracle HTTP Server for Oracle HTTP Server 12.1.3](#), and [Installing and Configuring Oracle HTTP Server for Oracle HTTP Server 12.2.1.3](#).

2. To configure Oracle HTTP Server, choose one of the following options:

- To configure Oracle HTTP Server in a stand-alone domain, follow the instructions at [Configuring Oracle HTTP Server in a Standalone Domain for Oracle HTTP Server 12.1.3](#), or [Configuring Oracle HTTP Server in a Standalone Domain for Oracle HTTP Server 12.2.1.3](#).
- To configure Oracle HTTP Server in an Oracle WebLogic Server domain, follow the instructions at [Configuring Oracle HTTP Server in a WebLogic Server Domain for Oracle HTTP Server 12.1.3](#), or [Configuring Oracle HTTP Server in a WebLogic Server Domain for Oracle HTTP Server 12.2.1.3](#).

3. If Oracle HTTP Server and Oracle WebLogic Server are installed in different domains, to enable CORS, install a web server plug-in.

For more information about configuring an Oracle WebLogic Server proxy plug-in, see [Configuring the Plug-In for Oracle HTTP Server for Oracle HTTP Server 12.1.3](#), or [Configuring the Plug-In for Oracle HTTP Server for Oracle HTTP Server 12.2.1.3](#).

4. To secure Oracle HTTP Server, follow the procedure at [Managing Application Security 12.1.3](#), or [Managing Application Security for Oracle HTTP Server 12.2.1.3](#).

Results

The Oracle HTTP Server instance is now ready to for you to deploy the application. The default location for deploying the application is `OHS_INSTANCE/config/fmwconfig/components/${COMPONENT_TYPE}/instances/${COMPONENT_NAME}/htdocs`. However, you can configure the default location value to a different location.

What to do next

Start Oracle HTTP Server. For more information, see [Next Steps After Configuring an Oracle HTTP Server Domain for Oracle HTTP Server 12.1.3](#), and [Next Steps After Configuring the Domain for Oracle HTTP Server 12.2.1.3](#).

Configuring the Oracle HTTP Server plug-in

If a web server such as Oracle HTTP Server is configured in the topology, you must configure a web server plug-in in Oracle WebLogic Server. The web server plug-in enables Oracle WebLogic Server to communicate with Oracle HTTP Server.

About this task

To enable an Oracle HTTP Server web server plug-in in Oracle WebLogic Server, you can run the `configurewebserverplugin` target.

Procedure

1. Ensure the remote Oracle WebLogic Server Oracle WebLogic Server is running.

For more information, see [Starting and stopping Web Logic servers](#) .

2. On the remote Oracle WebLogic Server, run the following command.

The `configurewebserverplugin` target requires a mandatory `server.name` argument that specifies the name of the server when the target is invoked.

```
build configurewebserverplugin -Dserver.name=server_name
```

For more information about the `configurewebserverplugin` target, see [Configuring a web server plug-in in Oracle WebLogic Server](#).

3. Restart the remote Oracle WebLogic Server.

For more information, see [Starting and stopping Web Logic servers](#) .

Installing and configuring Apache HTTP Server

Install and configure Apache HTTP Server on either the same server as the application server or on a remote server. To enable cross-origin resource sharing (CORS), you can set the **curam.rest.allowedOrigins** property for the REST application on your application server, or install the appropriate plug-in for your web server.

Before you begin

An application server must be installed and configured.

About this task

To enable cross-origin resource sharing (CORS), choose one of the following options:

- Set the **curam.rest.allowedOrigins** property for the REST application that is deployed on the application server. For more information about setting the **curam.rest.allowedOrigins** property, see [Cúram REST configuration properties](#).
- Install and configure the plug-in for your server.

Procedure

1. Install Apache HTTP Server. For more information, see [Compiling and Installing](#) in the Apache HTTP Server documentation.
2. Optional: If you don't set the **curam.rest.allowedOrigins** property, you must choose one of the following options:
 - WebSphere Application Server
Install the plug-in for WebSphere Application Server, see [Installing and configuring web server plug-ins](#).
Install the WebSphere Customization Toolbox, see [Installing and using the WebSphere Customization Toolbox](#).
To configure Apache HTTP Server with WebSphere Application Server, see [Configuring Apache HTTP Server](#).
 - Oracle WebLogic Server 12cR1 (12.1.3):
For more information about configuring an Oracle WebLogic Server proxy plug-in, see [Configuring the Plug-In for Oracle HTTP Server](#).
To configure Apache HTTP Server with Oracle WebLogic Server, see [Configuring the Plug-In for Apache HTTP Server](#).
 - Oracle WebLogic Server 12cR2 (12.2.1.3):
For more information about configuring an Oracle WebLogic Server proxy plug-in, see [Configuring the Plug-In for Oracle HTTP Server](#).
To configure Apache HTTP Server with Oracle WebLogic Server, see [Configuring the Plug-In for Apache HTTP Server](#).
3. Start Apache HTTP Server. For more information, see [Starting Apache](#) in the Apache HTTP Server documentation.
4. To secure Apache HTTP Server server, see [Security Tips](#) and [Apache SSL/TLS Encryption](#) in the Apache HTTP Server documentation.

Deploying your web application

To test your web application against an existing IBM Cúram Social Program Management application that is deployed on an enterprise application server, you can deploy the web application on IBM HTTP Server

or Oracle HTTP Server. Both web servers are based on Apache HTTP server so the deployment procedure is similar.

Before you begin

You must have built your application for deployment.

About this task

The built deliverable comes with a preconfigured `.htaccess` configuration file for the Content-Security-Policy (CSP) header. When you configure the CSP header in the web server, the `.htaccess` file is detected and executed by the web server to alter the web server configuration by enabling or disabling additional functionality. For more information about CSP, see the *Content Security Policy Quick Reference Guide* related link.

Procedure

1. Copy and paste the build directory contents to the appropriate directory for your HTTP server.

For more information about the `<directory>` directive, see the related links.

2. Configure the web server.

The preconfigured `.htaccess` file contains a comment section with the web server configuration requirements for both CSP and `.htaccess` enablement.

For more information about how to configure `.htaccess` files in a web server, see the *Apache HTTP Server Tutorial: .htaccess files* related link.

Related information

[GitHub documentation: npm run build](#)

[Content Security Policy Quick Reference Guide](#)

[Apache core features V2.0: <Directory> Directive](#)

[Apache core features V2.4: <Directory> Directive](#)

[Apache HTTP Server Tutorial: .htaccess files](#)

Chapter 9. Troubleshooting and support

Use this information to help you to troubleshoot issues with the IBM Cúram Universal Access Responsive Web Application or IBM Social Program Management Design System.

The IBM Cúram Social Program Management supported assets can be installed, customized, and deployed separately from IBM Cúram Social Program Management, before being integrated into the system.

When troubleshooting web applications that are integrated with IBM Cúram Social Program Management, use this troubleshooting information in conjunction with the troubleshooting information for IBM Cúram Social Program Management. For more information, see the *Troubleshooting and support* related link.

Citizen Engagement components and licensing

You can use and customize the IBM Universal Access Responsive Web Application for your organization, or develop your own custom web applications in addition to the standard IBM Cúram Social Program Management application. Use this information to understand the IBM Cúram Social Program Management components, supported assets, and licenses that you need.

Installable components

IBM Social Program Management Design System supported asset

The design system provides foundational packages for building accessible and responsive web applications. It consists of a React UI component library, React development resources, and a style guide for creating web applications.

IBM Universal Access Responsive Web Application supported asset

The IBM Universal Access Responsive Web Application provides a reference web application, which you can use and customize for your organization. The IBM Universal Access Responsive Web Application requires the IBM Social Program Management Design System and the Universal Access application module.

Universal Access application module

The Universal Access (UA) application module provides the Universal Access administrator application and the Universal Access REST APIs that expose interfaces to Universal Access functions for consumption by the IBM Universal Access Responsive Web Application. Universal Access requires the IBM Cúram Social Program Management Platform.

Licensing Universal Access

You can buy the Universal Access application module, which entitles the IBM Universal Access Responsive Web Application asset, and IBM Cúram Social Program Management Platform, which entitles the IBM Social Program Management Design System asset.

Alternatively, you can buy Citizen Engagement, which includes the Universal Access application module, the IBM Cúram Social Program Management Platform, and both assets.

Licensing the IBM Social Program Management Design System

To develop custom web applications to complement the IBM Cúram Social Program Management Platform, you can buy the IBM Cúram Social Program Management Platform, which entitles the IBM Social Program Management Design System asset.

Citizen Engagement support strategy

The Citizen Engagement assets are typically released monthly, and they can be upgraded independently of IBM Cúram Social Program Management (SPM) . Each release is a full release and not a delta release.

The assets are supported for the lifetime of the latest supported SPM version available at the time of the asset release.

- The main asset line is released monthly and contains new features, enhancements, security updates, defects, and support for the latest SPM version.
- IBM Universal Access Responsive Web Application 2.6.0 continues to be supported with security updates and critical defect fixes for older compatible SPM versions.

Although new features can be delivered in any asset release, they are typically delivered at the same time as the Universal Access application module release that contains the new APIs for those features. Where possible, Universal Access REST API changes are delivered in refresh pack or other impact-free releases that impose no forced upgrade impact.

Semantic versioning

The assets use semantic versioning. As a general guideline, this means:

- MAJOR version for incompatible API changes
- MINOR version for adding functionality in a backwards-compatible manner
- PATCH version for backwards-compatible bug fixes

Examining log files

Log files are a useful resource for troubleshooting problems.

Examining the browser console logs

For JavaScript applications, you can examine the browser console logs for errors that might be relevant to investigating problems. For the exact details about how to locate the console logs within the browser, see your browser documentation.

Note: When you are developing applications with the IBM Social Program Management Design System, console logging information might also be displayed in the console that runs the start process for the application.

Examining the HTTP Server log files

When you deploy a built application on an HTTP Server, the built application introduces a new point with which logging is captured in your system topology. The IBM HTTP Server, Oracle HTTP Server, and the Apache HTTP Server include comprehensive logging system and related information.

For more information about troubleshooting the IBM HTTP Server, see [Troubleshooting IBM HTTP Server](#).

For more information about troubleshooting the Oracle HTTP Server, see [Managing Oracle HTTP Server Logs](#).

For more information about troubleshooting the Apache HTTP Server, see [Log Files](#).

Notices

This information was developed for products and services offered in the United States.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Privacy Policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user's name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.



Part Number:

(1P) P/N: