Ozone Marketplace Developer's Guide

DOD GOSS

Exported on Oct 16, 2018

Table of Contents

1 Introduction ... 3
1.1 Objectives ... 3
1.2 Document Scope ... 3
1.3 Related Documents ... 3

2 Build ... 4
2.1 Overview ... 4
2.2 Build Tool Dependencies ... 4
2.3 Required Application Modules ... 4
2.3.1 Module Build Order ... 5
2.4 Compile and Run the Server ... 5
2.4.1 Development Mode ... 5
2.4.2 Production Mode ... 5
2.5 Build the Distribution Bundle ... 5

3 Themes ... 7
3.1 Overview ... 7
3.2 Changing the Default Theme ... 7
3.3 Changing the Default Logo ... 7
3.3.1 Changing the Store Header Tooltip ... 7
3.3.2 ... 7
3.4 Creating and Modifying Themes ... 8
3.4.1 Prerequisites ... 8
3.4.2 Compiling Themes ... 8
3.4.3 Layout of Themes Directory ... 8
3.4.4 Creating a New Theme ... 10
3.4.5 Customizable Theme Components ... 11
3.4.6 Minifying and Compressing Themes ... 13

4 Custom Field Types ... 14
4.1 Overview ... 14
4.2 Adding a Custom Field Type ... 14

5 Custom Security Modules ... 17
5.1 Overview ... 17
5.2 Requirements ... 17

6 OMP-DG: Appendix A Software Dependency Versions ... 19
6.1 Back-end ... 19
6.2 Front-end ... 19

1 Introduction

1.1 Objectives

This guide covers topics relevant to installing, configuring, and administering an OZONE Store. Similar to an online storefront like the Apple App Store or Google Play, the OZONE Store operates as a thin-client registry of applications and services. It enables users to create, browse, download and use a variety of applications or software components that are known as listings. Like commercial software stores, it offers quick and easy access to a variety of listings including—but not limited to—OZONE Apps, App Components, plugins, REST & SOAP services, Web Apps and desktop applications.

1.2 Document Scope

This guide is intended for system administrators of a Store and for Developers who wish to extend a Store beyond the default look. A System Administrator is someone who installs, optimizes and maintains the Store and sets up user authentications and authorizations. A Developer is understood as someone who is comfortable unpacking and packing WAR (.war) files, editing JavaScript (.js) files, Cascading Style Sheets (.css) and editing customized configuration files.

1.3 Related Documents

Document	Purpose
Quick Start Guide	Setting up and integrating the Store into OWF.
Configuration Guide	Modifying Default Settings, Security, Database Settings, Upgrading to a newer Store version.
User Guide	Searching, Creating and Editing Listings, Adding Comments, Ratings, Navigating a Store, Scorecards Explanation of Store Elements.
Administrator's Guide	Importing / Exporting Store Data, Adding Affiliated Stores, Approving Listings, Creating Types, States, Categories, and Custom Fields, Scorecard Configuration.
Developer's Guide	Custom Security Modules, Creating and Editing Themes
Release Notes	Major and minor changes for the current release.

2 Build

2.1 Overview

All build steps are run using the Gradle build system.

The instructions below use the Gradle wrapper scripts included in each project folder, which uses the recommended Gradle version.

To use a locally installed version of Gradle, replace the `gradlew` command with the `gradle` command.

Please note that only the Gradle version listed in the Build Tool Dependencies section has been fully tested.

2.2 Build Tool Dependencies

Application	Java (JDK)	Gradle	Groovy	Grails	Node.js	NPM
OMP 7.17.1.0	1.8	4.2.1	2.4.12	3.3.1	8.6.0	5.3.0

Obtain installation media and instructions for the various operating systems from the primary websites for each tool or trusted download source. The default locations are provided below. Also, install the tools in the order listed below. Once all tools have been installed, the following sections will describe how to configure the environment.

Application	Location
Java (JDK)	http://www.oracle.com/technetwork/java/javase/downloads/index.html
Gradle	https://gradle.org/releases/ Installation optional - a standalone Gradle wrapper utility, `gradlew`, is provided with the project.
Groovy	http://groovy-lang.org/ Installation optional - dependency managed by Gradle build system.
Grails	http://www.grails.org/ Installation optional - dependency managed by Gradle build system. Only required to use the `grails` command line utility.
Node.js	https://nodejs.org/en/
NPM	https://www.npmjs.com/ Installed by default as part of the Node.js distribution

2.3 Required Application Modules

Dependencies **MUST** be installed in the order listed below.

2.3.1 Module Build Order

Order	Module	Latest Branch	Command	Dependencies
1	ozone-classic-bom	release/v7.17.2.0-RC1	gradlew install	n/a
2	owf-appconfig	release/v0.9.1.0-RC1	gradlew install	ozone-classic-bom
2	owf-auditing	release/v1.3.2.0-RC1	gradlew install	ozone-classic-bom
2	owf-messaging	release/v1.19.1.0-RC1	gradlew install	ozone-classic-bom
2	owf-security	release/v4.0.4.0-RC1	gradlew install	ozone-classic-bom
2	owf-custom-tomcat	release/v1.2.3.0-RC1	gradlew install	ozone-classic-bom
3	omp-marketplace	release/v7.17.2.0-RC1	gradlew :bundle	ozone-classic-bom,owf-appconfig,owf-auditing,owf-messaging,owf-security,owf-custom-tomcat

2.4 Compile and Run the Server

2.4.1 Development Mode

The `:bootRun` task will build the project and start the Marketplace server in "development" mode.

```
omp-marketplace/> gradlew :bootRun
```

2.4.2 Production Mode

By providing the `-Dgrails.env=production` setting, the `:bootRun` task will build the project and start the Marketplace server in "production" mode.

The `-Dowf.db.init=true` option populates the database with the initial data, and is only required the first time the server is started (if the database is persisting changes).

```
omp-marketplace/> gradlew :bootRun -Dgrails.env=production -
Dowf.db.init=true
```

2.5 Build the Distribution Bundle

The `:bundle` task will build the Store distribution bundle .ZIP file, which includes the .WAR file, the custom Tomcat container, and all example configuration.

```
omp-marketplace/> gradlew :bundle
```

After the build has completed, the generated bundle can be found in the build directory: `/omp-marketplace/build/ozone-marketplace-7.17.1.0-RC1.zip.`

If the build fails (especially after pulling new changes), run a full clean of the project and then retry the build.

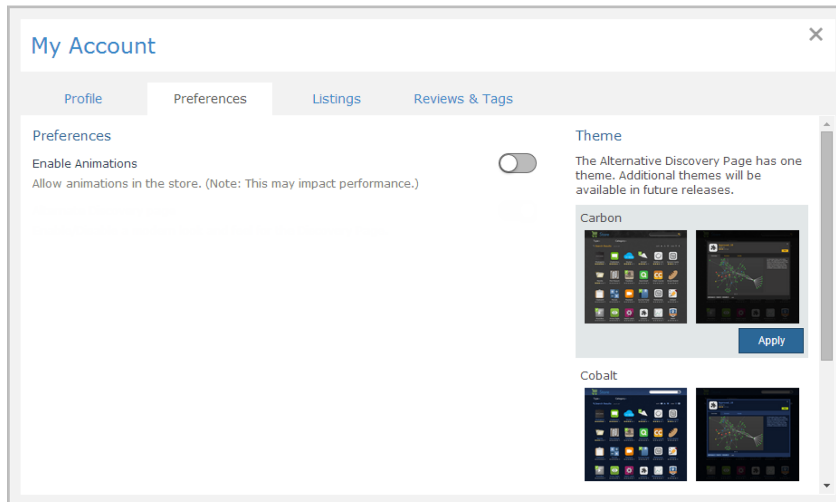
```
omp-marketplace/> gradlew clean
```

3 Themes

3.1 Overview

The Store includes three default themes: Oxygen, Cobalt and Carbon. Users can change themes from the Store Themes link under Preferences tab on the User Profile (accessible from the drop-down User Menu)

Figure 1 Theme Selection Menu



The Store allows developers to make changes to the images which are used in the interface by opening the marketplace.war file and replacing some of the default image files in addition to making modifications to the marketplace.css file.

3.2 Changing the Default Theme

3.3 Changing the Default Logo

Use the Application Configuration user interface to change the default store logo, rather than by manually replacing the logo on the server as was done in previous versions. See the Administration Guide for instructions.

3.3.1 Changing the Store Header Tooltip

3.3.2

A tooltip appears when a user hovers over the Store title in the header.

To update this text:

1. In the exploded .WAR file, open the `/WEB-INF/classes/messages_overlay.properties` file.
2. Update the `tooltip.ompHeaderLogo` property to set the new tooltip message:

```
marketplace.title=New Organization Name
```

3.4 Creating and Modifying Themes

The Store uses Compass, an open-source CSS stylesheet framework built on top of the SASS family of stylesheet languages. Two languages comprise SASS. The Store uses SCSS, the newer of the two languages. SCSS is a superset of CSS which compiles into CSS. Compass is a framework for managing large SASS projects as well as augmenting and managing the SASS compilation process. For more information on SASS and Compass, see http://compass-style.org/ and http://sass-lang.com/.

3.4.1 Prerequisites

Since v7.17.1.0, the Store uses Gradle and JRuby to automate the task of compiling the themes.

The required dependencies (JRuby, Compass, and SASS) are managed by the Gradle build environment, set to the versions listed below:

> These versions are out-of-date, but are tested and work with the current theme resources. Change at your own risk.

- JRuby – v1.7.27
- SASS (Ruby gem) – v3.1.3
- Compass (Ruby gem) – v0.11.7

3.4.2 Compiling Themes

The Gradle scripts for compiling the themes may be found in `/gradle/build-theme-resources.gradle`.

This script provides the `:buildTheme` task that will compile all the themes contained in `/src/main/resources/public/themes/`. Additionally, it will dynamically generate tasks for building individual themes that follow the specified theme directory naming conventions. For example, for the Cobalt theme found in `/src/main/resources/public/themes/cobalt.theme/`, it will generate the `:buildTheme_cobalt` task.

3.4.3 Layout of Themes Directory

The themes directory may be found in `/src/main/resources/public/themes/`.

The following table offers a brief description of the various theme directories and files provided in the distribution:

File or Directory	Description
/cobalt.theme/	Directory containing the default Cobalt theme.
/gold.theme/	Directory containing the Gold theme.
/carbon.theme/	Directory containing the Carbon theme.
/common/	Directory containing shared files that are likely to be used by most or all themes.
/common/images/	Directory containing images that are common to many themes or can serve as defaults.

File or Directory	Description
/common/lib/marketplace_utils.rb	Override a third-party stylesheet function to adjust it to our custom layout and image resolving setup (shouldn't need to be modified).
/common/stylesheets/	SCSS "partials" that build store themes.
/common/stylesheets/_marketplace_all.scss	Combines all the stylesheets into one. If new stylesheets are created in the common directory, they should be included in this file.
/common/stylesheets/variables/	Variable definitions for Store specific components.
/common/stylesheets/variables/_constants.scss	Variable definitions that (generally) should not change.
/common/stylesheets/variables/*	Variable definitions that control aspects of the stylesheet generation. The values of these variables may be overridden in a given theme
/template/	Directory containing a theme template. It contains every file listed above except the *.css files. These files are as complete as possible, without including properties that differentiates themes. The developer must enter data for the differences.

The following table explains the files and directories conventions for individual theme files. The files and folders are found under the directories of the specific theme that they modify. In the Store, the parent theme directory names end with ".theme". For example, `/carbon.theme/`. The table below uses an example theme named `example.theme`—this example is not included in the bundle (however, see the `/themes/template/` directory for reference).

Naming Convention	Description
/example.theme/theme.json	Contains the theme metadata description that tells the Store where to find the theme's files at runtime and provides information about the description, author, and display name.
/example.theme/css/example.css	Result of compiling a SCSS file from the SASS directory is stored here, in a file with the same name but with a .css extension instead of .scss.
/example.theme/images/	Directory containing images specific to the theme. The Store searches for images in this location first, and if they are not found, falls back to `/common/images/`.
/example.theme/images/preview/	Directory for theme screenshots.
/example.theme/sass/example.scss	The main .scss file for a theme – overrides any desired variables from the files in the `/common/stylesheets/variables/` directory. It defines the theme background and imports the desired files from the `/common/stylesheets/` directory.

Naming Convention	Description
/example.theme/sass/config.rb	Sets any Ruby variables used by Compass to locate resources. **Modify only at your own risk.**

3.4.4 Creating a New Theme

The easiest way to create a new theme is to copy an existing theme and then edit it. While it is possible to create a theme from scratch by using files from the template directory, it is not advisable.

Choose an existing theme that mirrors the overall contrast of your new theme. If the theme will feature dark text on light backgrounds, copy cobalt. If the theme will feature light text on dark backgrounds, copy carbon.

In the following instructions, the existing theme being copied is the cobalt theme.

1. Choose a theme name.
   The name should not have any spaces. It should be all lowercase. Words can be separated by hyphens.
   (Note: further references to this theme name will be denoted as $THEME_NAME; replace ALL occurences below with the chosen name.)
2. Copy the `/themes/cobalt.theme/` directory to `/themes/$THEME_NAME.theme/`
3. Delete the `cobalt.css` file in the `/themes/$THEME_NAME.theme/css/` directory.
4. Delete any files contained in the `/$THEME_NAME.theme/images/preview/` directory.
5. Navigate to the `/$THEME_NAME.theme/` directory.
6. Edit the `theme.json` file. Change every reference to "cobalt" to "$THEME_NAME".
   - The `name` attribute must be $THEME_NAME
   - The `display_name` attribute should contain a user-friendly, readable name for the theme (it may include spaces and capital letters). This will appear in the theme picker.
   - The `css` attribute must be `themes/$THEME_NAME.theme/css/$THEME_NAME.css`
   - All URL properties are relative to the context root.
     Note: For now, the thumb and screenshots fields will point to screenshot images that do not exist yet. You will add actual screenshots of your new theme when it is complete.
7. Rename `/sass/cobalt.scss` to `/sass/$THEME_NAME.scss`.
8. Edit `/sass/$THEME_NAME.scss`. **This is a mandatory step.**
   - Set the `$theme-name` variable to `$THEME_NAME`
9. Continue editing `/sass/$THEME_NAME.scss`. This is an optional step.
   This file is the primary place to create a custom theme by overriding variables. The files within themes/common/stylesheets/variables contain lists of variables that are available for overriding. Custom values for these variables usually should be defined directly below the $theme_name declaration (before importing variables/*). The lower part of this file imports all the SCSS partials that use the variable values to construct the stylesheets.
   After the import of the partials, you can add customizations to those partials. You will note that the included themes' SCSS files all contain numerous customizations. Since you copied an existing theme to create your new one, all you need to do is make any modifications to these customizations that are already present.
   Note: For complex customizations, import statements can be deleted if equivalent functionality is custom-implemented by the theme.
   Note: The default themes facilitate variable font sizes. When modifying these themes,

consider variable font sizes. Failure to do so may cause rendering inconsistencies at larger font sizes.

10. By default, a theme uses default images that are found in the `/common/images` directory. If custom images are going to be used, they can be placed in the `/$THEME_NAME.theme/images/` directory.
    To override an existing image from the common directory, the new images must have the same pathname as the image being overridden relative to the images directory.
    For example, `/common/images/table/bullet.png` would become `/$THEME_NAME.theme/images/table/bullet.png`.
11. Compile the theme. This can be done in several ways.
    - To compile a single theme, run the generated Gradle task :buildTheme_$THEME_NAME

    ```
    omp-marketplace/> gradlew :buildTheme_cobalt
    ```

    - To compile all the themes, run the Gradle task `:buildThemes`

    ```
    omp-marketplace/> gradlew :buildThemes
    ```

12. Once the theme successfully compiles, verify that the `/css/$THEME_NAME.css` file has been created, and that it does not contain any error messages. (These messages replace the entire normal output, so if errors exist they will be obvious).
13. Deploy the Store with the newly created theme. Do this one of two ways:
    - For a development instance, run the Gradle `:bootRun` task from the top directory of the source tree.

    ```
    omp-marketplace/> gradlew :bootRun
    ```

    - For a production instance, build the Store bundle and run it under the included Tomcat server. Run the Gradle `:bundle` task from the top directory of the source tree, extract the generated bundle .ZIP file, and start the server by using the start script found in the bundle's `/tomcat/` directory.

    ```
    omp-marketplace/> gradlew :bundle
    ```

14. Log into the Store as a user or admin, and open the user drop-down menu, located under the settings button on the toolbar.
15. Select and apply the new theme in the theme selector.
    Note: Currently, there are no new screenshots for the newly created theme.
16. Once the new theme is running, take screenshots, and add them to the preview window.
    a. Screenshots should be saved in the `/$THEME_NAME.theme/images/preview/` directory so that they show up in the theme picker.
    b. Modify the `/$THEME_NAME.theme/theme.json` file and add/modify the screenshot entries to include the new images.

3.4.5 Customizable Theme Components

This section lists the Store components which may (optionally) be themed.

All themeable components are located in the `/src/main/resources/public/themes/common/stylesheets/` directory as described in the following table:

File that can be overridden	Themeable Component(s)
variables/_aboutWindow.scss	About Window
variables/_actionMenu.scss	Action Menu
_admin.scss	Admin Home Page
_affiliated_search_results.scss	Affiliated Search Results carousel
_applicationConfiguration.scss variables/_applicationConfiguration.scss	Application Configuration Page
_bootstrap_about_window.scss	About window
_bootstrap_all.scss	Contains all the bootstrap files in one file
_buttons.scss	Buttons
_carousel.scss	Get Started Carousel
_carousel_component.scss	Other Carousels – Highest Rated and Newest Listing
_create_edit_listing.scss	Create/Edit listing modal window
_dialog.scss	Pop-up dialog windows
_dragAndDrop.scss	Drag and Drop
_error.scss	Error
_header.scss variables/_header.scss	Header
_landing.scss variables/_landing.scss	Landing Page – Getting started carousel, listing carousels
_list_group.scss	List styles – Change log and recent activity
_loadmask.scss variables/_loadmask.scss	Loading mask
_main.scss	Drop-down Menu Form Headers: Window, Panel and Taskbar Loadmask Message Box Progress Bar Tooltips
_marketplace_all.scss	Includes all bootstrap, including all overrides, in one file
_menubar.scss variables/_menubar.scss	Filter Menus
_profile.scss	User Profile modal window

File that can be overridden	Themeable Component(s)
_quickview.scss	Quick View modal window
variables/_scoreCard.scss	Scorecard
_settingsPanel.scss variables/_settingsPanel.scss	Settings panel on listing details page
_store_modal.scss	Base styles for modal windows
_tag.scss	Tag(s) on the Quick View window
_taglist.scss	Tags Page
_themeSwitcher.scss variables/_themeSwitcher.scss	Theme Switcher Window
_tutorial.scss variables/_tutorial.scss	Tutorials
_widget.scss *(deprecated)*	The Store in Widget Mode in OWF
_wizard.scss	Import/Export Wizard

3.4.6 Minifying and Compressing Themes

As for Marketplace v7.17.1.0, due to changes in the build processes, the compiled .CSS files and other theme assets are no longer minified or compressed.

This feature will be re-added as part of the automated build process in a future release.

4 Custom Field Types

4.1 Overview

The Store ships with several default custom field types. They include text, image URL, checkbox and drop-down custom field options. Use the following instructions to create additional custom field types. Creating new custom fields requires that a developer modify and recompile the Store source code, specifically:

- Adding new domain objects (with the accompanying database tables)
- Changing the existing JavaScript files
- Creating a new Groovy Server Page

To continue using the custom field type, a developer must apply these changes to the Store source code when upgrading to the next version of the product.

This method is different from the *Create a Custom Field Definition* process performed through the Store UI Administration pages.
More information on this process is found in the Store Administrator's Guide.

4.2 Adding a Custom Field Type

The following instructions explain how to add a custom field type named INTEGER (used to create custom fields for integer values):

1. In the Store's source code, navigate to \grails-app\domain and create a new domain class called marketplace.IntegerCustomField that extends the marketplace.CustomField class. The Store uses the class to store the values of ServiceItem fields associated with the new custom field type. For that purpose, the class marketplace.IntegerCustomField must have an instance variable of type int.
    a. Define the method void setValue(def value). The Store will call this method to set the value of a custom field for the new custom field type.
    b. Define the method String getFieldValueText(). The Store calls this method to retrieve the value for the new custom field type.
    c. Define the method asJSON() which should return a JSON representation of the custom field. This method must call its parent's asJSON method to obtain the generic representation of a custom field and add this field's information to it:

```
package marketplace


class IntegerCustomField extends CustomField {

    int value

    static constraints = {}
```

```
            void setValue(def val) {
                this.value = val?.toInteger()
            }

            String getFieldValueText() {
                return value.toString()
            }

            def asJSON() {
                def jsonObject = super.asJSON()
                jsonObject.putAll(id: id, value: value)
                return jsonObject
            }
        }
```

2. Create a new domain class called marketplace.IntegerCustomFieldDefinition extending marketplace.CustomFieldDefinition. This class represents the new custom field type.

    a. Define a null constructor that sets the styleType member variable to the enum that will be created in the next step:

```
        package marketplace

        class IntegerCustomFieldDefinition extends
        CustomFieldDefinition {
            static constraints = {}

            IntegerCustomFieldDefinition() {
                this.styleType =
        Constants.CustomFieldDefinitionStyleType.INTEGER
            }
        }
```

3. Open Constants.groovy and add the new field type to the CustomFieldDefinitionStyleType enum. Add the following code to the enum:

```
INTEGER("Integer", IntegerCustomFieldDefinition.class,
IntegerCustomField.class)
```

The enum needs to reference the classes from Steps 1 and 2.

4. Under web-app/js/customFields open js/customFields/standardFields.js and do the following:

    a. Define two new functions:

        i. CustomFields.createInteger: This function is used to create the form element in addListingForm.js that is used on the Create and Edit listing pages.

        ii. CustomFields.displayInteger: This function is used to display the value of the custom field on the specifications tab.

```
                CustomFields.createInteger =
                function(cfs, cf, pos) {
                    var cfLbl =
                cfs.label.cleanEscapeHTML();
```

```
    var cfReq = false;
    if (cfs.isRequired) {
        cfLbl = cfLbl.requiredLabel();
        cfReq = true;
    }
    return {
        xtype: 'numberfield',
        fieldLabel: cfLbl,
        name:
CustomFields.getCustomFieldName(pos),
        maxLength: 10,
        allowBlank: !cfReq,
        value: cf ? cf.value : null
    };
};

CustomFields.displayInteger =
function(cf) {
    return (cf ? cf.value : null);
}
```

b. Register the functions with the CustomFields object. This associates the create and display functions with the field type. Do this by adding the following at the end of the standardFields.js file:

```
CustomFields.addField("INTEGER", {create:
CustomFields.createInteger, display:
CustomFields.displayInteger});
```

5. Add a _INTEGER.gsp template file to the views/customFieldDefinition directory:

    a. When creating or editing the custom field definition, the Store's configurations use this template to display specific field-type form elements. For example, an administrator can use the integer custom field to set the upper and lower bounds for the entry value. In that case, the gsp would display the controls for such bounds. If the new custom field definition does not have specific field-type form elements, an empty gsp template (_INTEGER.gsp under grails-app/views/customFieldDefinition) must be supplied as you can see in the INTEGER example:

```
<tr class="customFieldAdmin INTEGER"></tr>.
```

6. Create database tables that correspond with the two new domain objects that were created in steps 1 and 2.

    a. To use the grails command "grails schema-export" to create the script for the required tables, run the script from the top level Grails project directory. This will generate the statements that are used in the next step to create the tables. It generates file target/ddl.sql containing the DDL for creation of the entire application schema. SQL statements for the new tables will be in the generated file (for MySQL in this example).

    b. Using an RDBMS administrative tool, execute the SQL statements from the generated file:

```
create table integer_custom_field (id bigint not null,
value integer not null, primary key (id))
ENGINE=InnoDB;
create table integer_custom_field_definition (id
bigint not null, primary key (id)) ENGINE=InnoDB;
```

5 Custom Security Modules

5.1 Overview

Because security is based on the Spring Security framework, a custom authentication and authorization method can also be implemented by following the same conventions. A more detailed discussion of Spring Security can be found here:

http://static.springframework.org/spring-security/site/index.html

The following should be considered when writing the custom security module:

The Store will query the Metadata of the UserDetails object returned by the custom security module for the presence of displayName, organization and email. If these attributes are present their values will be retrieved and used by the Store.

> The sample security modules included with the Store will need to be replaced by a custom security module that meets the security requirements of the organization.

Administrators or Developers can use the sample security package, found at \ozone-security\ozone-security-project.zip as a starting point when developing a custom security project to be used in production. The .zip file contains source code complete with build scripts.

5.2 Requirements

The Spring Security framework allows individual deployments to customize the Store's authentication and authorization mechanisms. Developers can use the security plugin to integrate with any available enterprise security solutions. When customizing the security plugin, it is important to remember OWF/Store requirements for the plugin. These requirements are described below.

> The OWF/Store requirements are in addition to any general Web application requirements relating to Spring Security.

- **User Principal implements the UserDetails interface and optionally the OWFUserDetails interface**

Like all Spring Security web applications, the Store expects its security plugin to provide a UserDetails object which represents the logged-in user. A custom plugin should set this object as the principal on the Authentication object stored within the active SecurityContext. Optionally, the provided object may also implement the OWFUserDetails interface. In addition to the fields supported by the UserDetails interface, the OWFUserDetails interface supports access to the user's OWF display name, organization and email. The source code for OWFUserDetails can be found in ozone-security-project.zip.

- **ROLE_USER granted to all users**

The user principal object's getAuthorities() method must return a collection that includes the ROLE_USER GrantedAuthority.

- **ROLE_ADMIN granted to Store administrators**

The user principal object's getAuthorities() method must return a collection that includes the ROLE_ADMIN GrantedAuthority if the user is to have administrative access.

- **OZONELOGIN cookie set when the user signs in and deleted on sign out**

The user interface performs a check for the existence of a cookie named OZONELOGIN during the page load. If the cookie does not exist, the interface will not load, but will instead present a message indicating that the user is not logged in. It is up to the security plugin to create this cookie when the user logs in, and to delete it when they log out.

This mechanism prevents users from logging out, and then pressing the browser's Back button to get back into an instance that cannot communicate with the server due to failed authentication. The sample security plug-in configurations contain filters that manage this process. It is recommended that custom configurations include this default implementation of the cookie behavior by using the same `ozoneCookieFilter` and OzoneLogoutCookieHandler beans that are included in the sample configuration, `security.xml`.

- **Session management configurations must be present**

These configurations include the `concurrentSessionControlAuthenticationStrategy` bean and the `sessionRegistry` bean in `session-control.xml`, as well as the `<sec:session-management>` and `<sec:custom-filter>` elements in `security.xml` which references the `concurrencyFilter`.

> The `max-sessions` setting (in `security.xml`) and the `maximumSessions` setting (in `session-control.xml`) will be overwritten at run-time, since the maximum number of sessions is configured in the Application Configuration UI.

6 OMP-DG: Appendix A Software Dependency Versions

6.1 Back-end

Component	OMP v7.17.1.0-RC1	Latest Version *as of 2018-01-31*	Notes
Groovy	2.4.13	2.4.13	
Grails	3.3.2	3.3.2	
Spring	4.3.12*	4.3.14	* Grails supported version
Spring Security	4.2.3	4.2.4*	* Update released after RC-1
Hibernate	5.1.9*	5.2.12	* Grails supported version
Liquibase	n/a*	3.5.3	* Removed for RC1, to be re-added before GA
JAX-RS/Jersey Plugin	n/a*		* Removed; library no longer supported
Elasticsearch Plugin	1.4.1*	2.4.0-RC2	* Custom in-house version of plugin which public v2.4.0-RC1 was based on
Elasticsearch	5.4.3*	6.1.2	* Elasticsearch Plugin supported version

6.2 Front-end

Component	OMP v7.17.1.0-RC1	Latest Version *as of 2018-06-25*	Notes
Backbone.js	1.0.0	1.3.3	
Require.js	2.1.9	2.3.5	
Bootstrap	2.3.2	4.1.0	
Dojo	1.2.3	1.13	
Handlebars	1.0.0	4.0.11	
jQuery	1.10.2	3.3.1	
lodash	1.3.1	4.17.10	
Moment	2.1.0	2.22.2	