OZONE Store

Configuration Guide

August 22, 2014

Publication/Revision History

Release	Date
Revised – OZONE Store 7.16.0 FOSS	August 22, 2014
Revised – OZONE Store 7.14.2 GOSS	May 27, 2014
Revised – OZONE Store 7.15.1	May 15, 2014
Revised – OZONE Store 7.15.0	April 15, 2014
Revised – OZONE Store 7.14.0	April 1, 2014
Revised – OZONE Store 7.13.0	March 18, 2014
Revised – OZONE Store 7.12.0	March 4, 2014
Revised – OZONE Store 7.11.0	February 19, 2014
Revised – OZONE Store 7.10.0	February 4, 2014
Revised – OZONE Store 7.9.0	January 21, 2014
Revised – OZONE Store 7.8.0	January 7, 2014
Revised – OZONE Store 7.7.0	December 24, 2013
Revised – OZONE Store 7.6.0	December 10, 2013
Revised – OZONE Store 7.5.0	November 26, 2013
Revised – OZONE Store 7.4.0	October 25, 2013
Revised – OZONE Store 7.3	October 8, 2013
Revised – OZONE Store 7.3	July 31, 2013
Revised – Marketplace 7.2	June 11, 2013
Revised – Marketplace 7.1	April 26, 2013

Note: This revision history addresses the Marketplace project which served as predecessor to the OZONE Store.

Contents

Tables

Figures

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.

Note: Throughout this guide, there are instances where the swapping of image files is discussed. A .png file can be replaced with .jpgs, .gifs or other standard image files.

1.3 Related Documents

Table 1: Related Documents

Document	Purpose
Store User's Guide	Searching, Creating and Editing Listings, Adding Comments, Ratings, Navigating a Store, Scorecards Explanation of Store Elements
Store Administrator's Guide	Importing / Exporting Data, Adding Affiliate Store, Approving Listings, Creating Types, States, Categories, and Custom Fields
Store Configuration Guide	Modifying Default Settings, Security, Database Settings, Store Upgrades
Store Quick Start Guide	Setting Up and Integrating the Store into OWF

2 Overview

2.1 Purpose

The Store is a thin-client registry of applications and services akin to a commercial storefront. Like the Apple Store or Android Market, users can create and interact with web components or listings such as OZONE Apps, App Components, plugins, REST & SOAP services, Web Apps, and more.

2.2 Basic Architecture

The Store consists of a number of components that were designed to be independently deployed or located on the same server. The simplest deployment scenario places them all on the same physical machine. These components are shown in detail, below, in Figure 1: Basic System Architecture Diagram.

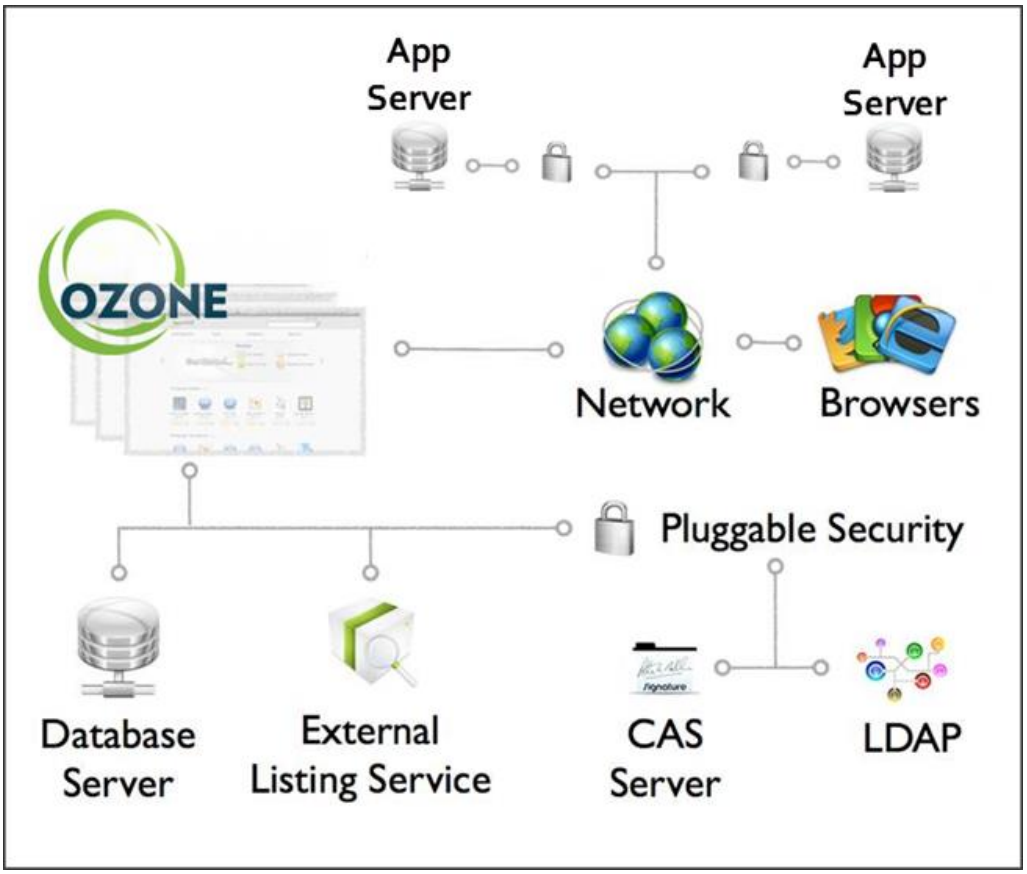

Figure 1: Basic System Architecture Diagram

2.3 Components

2.3.1 Apache Tomcat

Tomcat is a Web application server configured to host a Store.

2.3.2 In-memory HSQLDB

The Store ships with an in-memory HyperSQL database for testing and development purposes. A disk-based RDBMS such as Oracle or MySQL should be used in production.

2.3.3 Store Web Application

`marketplace.war`

– is the Store Web application archive

2.3.4 Pluggable Security

The Store does not provide a security module suitable for a production environment as delivered. Instead, a flexible, pluggable security framework, based on Spring Security, has been included to allow each organization to design and develop its own custom security solution. Find detailed instructions in section 3.5.2: Security Setup.

2.3.5 Store Security Project

`ozone-security-project.zip` – This bundle (located in **`/ozone-security`**) contains the source code, configuration files and library files needed to build the security files which are used by the Store. Additionally, an Apache ANT build script is included for building and creating the sample security project in the target directory, called **`ozone-security.jar`**. This file contains all six security samples which are used by the security .xml files. For more details see the **`README.TXT`**, found in the **`.zip`** file.

3 Installation

3.1 Dependencies

Listed below are the dependencies:

- Java 1.6 or higher.
- A Relational Database Management System (RDBMS). The Store currently ship with an in-memory HyperSQL database for testing and development purposes, but it is expected that a live deployment will use a more robust RDBMS such as Oracle or MySQL.

3.2 Supported Browsers

The Store requires JRE 1.6 or higher.

The Store is tested against Internet Explorer 7 and higher and Firefox 17 and higher. It is fully-supported against the following browsers:

Table 2: Tested Browsers

Browsers	Versions
Internet Explorer	7 & 9
Firefox	17
Chrome	33

3.3 Toolkit Description

The distribution of the Store consists of a **`.zip`** file containing the necessary components to set up and run a Store in a development environment. Go to https://www.owfgoss.org/downloads/AML to download the latest version of the bundle.

The bundle contains the following:

- Apache Tomcat (Java Web container)
- Sample PKI Certificates for SSL (both user certificate and server certificate)

The Store Web application (**`marketplace.war`**)

- Externalized Security Configurations

- Tomcat start scripts (**`start.sh`** and **`start.bat`** – runs Tomcat configured in SSL mode)
- Configurable and externalized properties files:
  - **`MarketplaceConfig.groovy`**
  - **`mp-override-log4j.xml`**
  - **`OzoneConfig.properties`**
  - **`Ehcache.xml`**

3.3.1 Deploying Store On Windows Environments

The following shows how to copy, unzip, and start a Store when deploying the bundle on Windows operating systems:

1) Create a **`C:\Store\`** directory. This can be done using the *Windows* UI or the command prompt.
2) Copy **`marketplace-bundle-7.16.0.zip`** to **`C:\ Store\`**.
3) Navigate to **`C:\ Store\`**.
4) Right-click the**`marketplace-bundle-7.16.0.zip`** , then select "Open," "Explore," or the command for the system's default zip/unzip program.
5) Unzip/unpack the bundle into **`C:\ Store\`**.
6) Navigate to **`apache-tomcat`**.
7) Execute **`start.bat`**.

Note: Instructions regarding running a Store as a Windows service are outside the scope of this document. Please refer to Web container-specific resources such as the Apache Tomcat Documentation.

3.3.2 Deploying a Store on Linux/Unix/*nix Environments

The following shows how to deploy the bundle on Linux/Unix/***nix** operating systems. Use the following code to copy, unzip and launch a Store by executing **`/apache-tomcat/start.sh`**:

```
mkdir /opt/Store
cp marketplace-bundle-7.16.0.zip /opt/Store
cd /opt/Store
unzip marketplace-bundle-7.16.0.zip
cd apache-tomcat/
./start.sh
```

3.4 Default Installation

Running a Store, using the included Tomcat Web Server with the default values, requires minimal installation. In addition to unzipping the bundle, certificates will need to be installed into client browsers.

The default certificates contained in the Store Bundle only function for localhost communications. The Store will *not* function correctly if accessed from a remote machine while using the bundle-included certificates. See section 3.5.2.2.3: Custom Security Logout for more details.

The use of the bundled deployment archive provides all of the necessary mechanisms to run the Tomcat Web container on any Java 1.6+ enabled system.

NOTE: To access a Store from any location other than localhost, a new server certificate needs to be generated. Otherwise, attempts to login from remote locations will fail.

3.4.1 Installing User PKI Certificates

By default, the security infrastructure of the Store bundle is configured to use client certificates. In order to identify themselves using certificates, clients need to install a PKI certificate into their Web browser. The client certificate, included with the Store Bundle, is recognized immediately and can be used in the default security configuration. The certificate is located in the **`apache-tomcat\certs`** directory of the Store Bundle.

Note: The included certificates should ONLY be used for testing. They are not to be used in a production environment.

The default client certificate is used by importing the included *testAdmin1.p12* or *testUser1.p12* certificate into the user's browser. In Internet Explorer, client certificates are added by selecting Tools → Internet Options → Content → Certificates → Personal and then Import.

In Firefox this menu is accessed by clicking:

Tools → Options → Advanced → Encryption → View Certificates →Your Certificates and then Import.

Note: Depending on the browser, importing certificates may cause warning messages to appear before accessing the Store. Web browsers will allow exceptions to be added to permit usage of these certificates the first time they are accessed.

3.5 Custom Installation

Customize the Store to run in a variety of environments. The following sections detail how to change default ports and security setups. For information about customizing database installations, see section 5.1.1: Database Setup. For additional information

about setting up custom security, see section 3.5.2: Security Setup. For administrators or developers who wish to create custom security implementations for production environments, the sample security package at **\ozone-security\ozone-security-project.zip** can be used as a starting point. The **.zip** file contains source code complete with build scripts.

3.5.1 Running the Store From Different Ports

The initial Store configuration is set up so that Tomcat can be run from a local installation. Throughout this document, "**servername:port**" is used to reference the server name and port number. For example purposes, this document uses "**localhost:8080**" or "**localhost:8443**." The example below shows how to set up a Store so that it can be used on ports 5050/5443 through the default security module. To enable ports other than 8080/8443 while using Spring Security options, the desired port numbers need to be explicitly named in the following files:

- **\apache-tomcat\lib\OzoneConfig.properties**
- Web server configuration file **\apache-tomcat\conf\server.xml**

Note: In the event that a Store is running on a server where a port number is already in use, it must run from a different port number and use a different shutdown port. One server cannot run two applications from the same port.

- Shut off the Tomcat Web Server via the appropriate stop command found in **\apache-tomcat\bin**.
- Open **\apache-tomcat\lib\OzoneConfig.properties** and change the port to the new port mapping (This file is referenced from other configuration files.).
- Save the file.
- Change the port numbers in the Tomcat Web Server configuration file **\apache-tomcat\conf\server.xml**:

```
…
<Connector port="5050" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="5443" />
…
<Connector port="5443" protocol="HTTP/1.1" SSLEnabled="true"
               maxThreads="150" scheme="https" secure="true"
               keystoreFile="conf/keystore.jks" keystorePass="changeit"
               clientAuth="want" sslProtocol="TLS" />
…
```

  a. Ports 5050 and 5443 are just examples and can be changed to whatever is needed. Ensure that the port value used in the Web server configuration file match the port value used in the **OzoneConfig.properties** file. In

the following example, the port numbers in **`OzoneConfig.properties`** were updated:

```
ozone.host = localhost
ozone.port = 5443
ozone.unsecurePort = 5050
```

b. If a Store is running on a server where a port number was already in use, the Store **`SHUTDOWN`** port must also be changed. To do this, change the port number in the Tomcat Web server configuration file **`\apache-tomcat\conf\server.xml`** to another port, in the following example the default shutdown port was changed from 8005 to 8006:

```
<Server port="8006" shutdown="SHUTDOWN">
```

- Save **`server.xml`** and start the Tomcat Web Server with the appropriate start command, found in **`\apache-tomcat`**.

3.5.2 Security Setup

The Store provides a modular security approach (security plugin) based on Spring Security. All provided security options supply an **`.xml`** file (located in the **`\ozone-security`** directory) that can be customized to perform authentication and authorization. **These example security options act only as samples and should in no way be used in a production environment.** However, the following files (which house common configurations information and bean definitions that are used by multiple plugins) can be used as the basis for updating a production environment:

- **`\ozone-security\ozone-security-beans\MPCasBeans.xml`**
- **`\ozone-security\ozone-security-beans\MPLogInOutBeans.xml`**
- **`\ozone-security\ozone-security-beans\LdapBeans.xml`**
- **`\ozone-security\ozone-security-beans\ListenerBeans.xml`**
- **`\ozone-security\ozone-security-beans\SessionManagementBeans.xml`**
- **`\ozone-security\ozone-security-beans\UserServiceBeans.xml`**

*Note: In most of the pluggable security samples, user authentication is configured through an external text file like **`users.properties`** or LDAP.*

Included in the **`\ozone-security`** directory is the **`users.properties`** file which can be used for authentication and authorization for local development installations. Also included are the following seven sample security files used to help developers start custom development of a pluggable security implementation.

- **`MPsecurityContext.xml`**—Contains the default security implementation and uses a PKI certificate for authentication. If a PKI certificate is not provided for authentication, the HTTP-BASIC authentication method will be used, prompting the user for a username and password.

  NOTE: As of version 7.5.0 of the OZONE Store, the default configuration has changed. In prior versions, the default was the configuration which is now stored in `MPsecurityContext_x509_CAS.xml`.

- **`MPsecurityContext_x509_CAS.xml`**—Contains an example of how the CAS and x509 authentication mechanisms can be combined. This configuration first attempts to authenticate the user using a PKI certificate. If a certificate is not provided, this configuration will redirect the user to a CAS server for authentication via whatever mechanism that CAS server is configured to use.
- **`MPsecurityContext_cert_only.xml`**—Contains the X509-only security implementation and uses a PKI certificate for authentication. If a valid certificate is not provided, the user will be denied access to the Store.
- **`MPsecurityContext_CAS_only.xml`**—Contains the CAS-only security implementation and requires successful sign in through the CAS server. If a valid username and password are not provided, the user will be denied access to the Store.
- **`MPSecurityContect_cert_ldap.xml`**—Contains an X509/LDAP security implementation that uses X509 for authentication and then performs an LDAP-based lookup to determine the user's authorization.
- **`MPsecurityContext_basic_ldap.xml`**—Contains a sample configuration for the security plugin that demonstrates how to use an LDAP-based lookup in conjunction with a simple username/password login page.
- **`MPsecurityContext_BasicSpringLogin.xml`** – An example of a very simple security plugin configuration using a form-based login page and inline user information.

The **`ozone-security-project.zip`** project (briefly mentioned in section 2.3.5: Store Security Project) stores the specific Java source code and configuration files necessary for the example security plugins and for interoperation with the OWF and the Store applications. In addition, this project contains the following support resource files:

- ***`src\main\resources\conf\apache-ds-server.xml`***, a sample **`.xml`** file used by Apache Directory Server (ApacheDS, an open-source LDAP v3 compliant embeddable directory server) that sets up the initial directory service partitions for the test data.

- ***src\main\resources\conf\testUsers.ldif*** an LDAP Data Interchange Format test file that can be imported to set up test entries that match the certificates bundled with the bundle.
- ***src\main\resources\conf\sample-log4j.xml,*** a sample **.xml** configuration file that demonstrates how to enable logging for Store Security.
- ***lib\spring-core-3.0.3.RELEASE.jar,*** a file which provides LDAP functionality used by the Spring Security LDAP security plugin.

The **ozone-security-project.zip** also contains an Apache ANT build script that can be used to generate the **.jar** file that contains the sample security implementation. The build script allows for a rebuild of the OWF security **.jar** file for customization.

3.5.2.1 Requirements for Customizing Security

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.

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

1) **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***.

2) **`ROLE_USER` granted to all users**

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

3) **`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.

4) **`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, in **`MPsecurityContext.xml`** and **`MPLogInOutBeans.xml`**.

5) **Session management configurations must be present**

These configurations include the **`concurrencyFilter`** bean, the **`concurrentSessionControlStrategy`** bean, the session **`Registry`** bean as well as a **`<session-management>`** element and a **`<custom-filter>`** element which references the **`concurrencyFilter`**.
For examples of the required settings for these elements, see **`MPSecurityContext.xml`** and **`ozone-security-beans\SessionManagementBeans.xml`**. It is important not to change the **`id`** of the **`concurrentSessionControlStrategy`**, as it is referenced by **`id`** from within the application.
Note: The ***`maximumSessions`*** setting contained in the ***`xml`*** configuration will be overwritten at runtime, since the maximum number of sessions is configured in the Application Configuration UI.

3.5.2.2 Installing the security module

When using the default security module in a testing environment, the user must present a valid certificate (like X509), or a valid login (like CAS), to gain access to the Store. While CAS should not be used in a production environment, if you're using it in staging, some of the CAS fields can be customized using the **`OzoneConfig.properties`**. See section 3.5.2.2.1.1: Customizing CAS.

For each available security option, there is a specific **`.xml`** file located in the **`ozone-security`** directory.

Changing the authentication or authorization method is accomplished in just a few easy steps:

1) Stop the Tomcat Web Server.

2) Delete any security related **.xml** configuration files from your classpath. When running Tomcat, these files are typically located in the **\apache-tomcat\lib** directory.
3) Place the appropriate **.xml** file from the **\ozone-security** folder onto the application server's classpath. When running Tomcat, the **.xml** file can be placed in **\apache-tomcat\lib**.
4) Restart the application server by clicking the **\apache-tomcat\start.bat** file.

Note: Some of the pluggable security modules provided use an externalized **users.properties** *file for the user store. This file can be placed anywhere on the application server's classpath. When running Tomcat, this file can be found in the* **\apache-tomcat\lib** *directory.*

3.5.2.2.1 Replacing Pluggable Security Files

To change the Store Security file, **MPsecurityContext.xml,** to a different security sample, replace the default security **.xml** file with the pluggable security **.xml** file (For example, replace **\apache-tomcat\lib\MPsecurityContext.xml** with the **\ozone-security\MPsecurityContext_cert_only.xml**).

Some security examples do not use certificates authentication. To shut off certificate authentication in the container edit the **server.xml** file. For example, the bundle's Tomcat instance is set up to ask for certificate authentication, but not require it. To eliminate the certificate prompt, edit the **\apache-tomcat\conf\server.xml** file and change the **clientAuth** property to **false**:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
           maxThreads="150" scheme="https" secure="true"
           keystoreFile="certs/keystore.jks" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"
/>
```

3.5.2.2.1.1 Customizing CAS

OzoneConfig.properties proxies some of the CAS properties that can be customized by specific organizations. To change where the CAS server, CAS login and CAS logout point, update the corresponding values in **OzoneConfig.properties**. For additional CAS instructions, see CAS documentation.

The Store and the Ozone Widget Framework (OWF) use some files that have the same name but different properties. **OzoneConfig.properties** is one of those files. If the Store is deployed in the same application server as an instance of OWF, the server can only use one **OzoneConfig.properties** file. The file that is used must include CAS properties for both OWF and the Store.

To update **`OzoneConfig.properties`** using the Store version of the file:

1. From the **`apache-tomcat\lib`** folders in the Store Bundle and OWF, open the **`OzoneConfig.properties`** file in both products.
2. From the OWF **`OzoneConfig.properties`** file, copy:

```
ozone.cas.owf.serverSecureReceptorLocation=owf/secure/receptor
ozone.cas.owf.jSpringCasSecurityCheckLocation=owf/j_spring_cas_security_check
```

Note: In both instances in the configuration block above, the locations should be renamed to "owf/..." if only being used for a Store installation.

3. Paste the two lines of code into the **`OzoneConfig.properties`** file.
4. Copy the updated **`OzoneConfig.properties`** file onto the classpath of the server that is hosting both applications.
5. Restart the server.

3.5.2.2.2 Custom Security

Because security is based on Spring Security, 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 custom security module must contain a **`securityContext.xml`** file in the META-INF directory conforming to the following naming convention **`*MPsecurityContext*.xml`**.

Note: 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.

3.5.2.2.3 Custom Security Logout

The sample security plugins can perform single sign out if the user logged in using CAS authentication. PKI authentication is handled by the browser and requires that the user close the browser to completely sign out. To sign out from LDAP or a custom authentication, the system administrator must implement their own single sign out or instruct the user to close the browser after logout. Use the following line in the security context file to invoke CAS's single sign out process.

```
<sec:custom-filter ref="casSingleSignOutFilter" after="LOGOUT_FILTER"/>
```

Also, see **`MPCasBeans.xml`** which contains bean definitions:

```
<bean id="casSingleSignOutFilter" class="org.jasig.cas.client.session.SingleSignOutFilter"/>
<bean id="casSingleSignOutHttpSessionListener"
class="org.jasig.cas.client.session.SingleSignOutHttpSessionListener" />
```

3.5.2.3 Server Certificate Creation and Installation

Valid server certificates are needed for configuring the server to allow https authentication. See the steps below for generating and installing a self-signed server certificate.

3.5.2.3.1 Generating a New Self-Signed Server Certificate

The Java **`keytool`** utility can be used to generate the public/private key and signed certificate. Open a command prompt, navigate to the **`\apache-tomcat\certs`** directory and execute the following command:

```
keytool -genkey -alias servername -keyalg rsa -keystore servername.jks
```

Note: *Some systems do not default to the Java keytool. The keytool can be explicitly called by running the command directly from the JRE/bin directory.*

The keytool genkey command will prompt a series of questions. The questions are listed below with example entries matching a server with the name '*www.exampleserver.org*' and a keystore password of '*changeit*'.

Enter keystore password: *changeit*

What is your first and last name? [Unknown]: www.exampleserver.org
*Note: Make sure to enter the **fully qualified** server name. This needs to match the hostname of the machine exactly or the certificate will not work correctly.*

What is the name of your organizational unit? [Unknown]: *sample organization unit*

What is the name of your organization? [Unknown]: *sample organization*

What is the name of your City or Locality? [Unknown]: *sample city*

What is the name of your State or Province? [Unknown]: *sample state*

What is the two-letter country code for this unit? [Unknown]: *US*

Is CN= www.exampleserver.org, OU= sample organization unit, O= sample organization, L= sample city, ST= sample state, C=US correct? [no]: *yes*

***Note:** When using an IP address as the Common Name (CN), add an entry to the Subject Alternative Name entry in the certificate. The better alternative to using an IP address is to add a name/IP pair to the hosts file and register the name as the CN.*

3.5.2.3.2 Configuring for Different Truststore/Keystore

For server-to-server calls (Store-to-CAS communications, for example) the newly created self-signed certificate should be imported into the truststore.

1) Export the certificate from the keystore into a file:

```
keytool -export -file servername.crt -keystore servername.jks -alias servername
```

2) Import the file into the truststore:

```
keytool -import -alias servername –keystore mytruststore.jks  -file servername.crt
```

3) Modify the JVM Parameters to start the web-application server in order to use the truststore referenced in Step 2. If a Tomcat server is being used, the parameters can be found in **`setenv.sh`** or **`setenv.bat`** (located in **`\apache-tomcat\bin`**), inside of the unpacked **`marketplace-bundle-7.16.0.zip`**. If an application server other than Tomcat is being used, the parameters will need to be added to the JVM parameters which are loaded when the application server is started.

Table 3: Custom JVM Parameters

Parameter	Note
-Djavax.net.ssl.trustStore=certs/mytruststore.jks	Replace 'certs/keystore.jks' with the path and filename to the truststore
-Djavax.net.ssl.trustStorePassword=changeit	Replace 'changeit' with the truststore's password (if applicable).

Finally, modify the Web server configuration to use the new keystore in SSL. Shown below is the relevant section from the Tomcat configuration script found in **`\apache-tomcat\conf\server.xml`**:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
```

```
            maxThreads="150" scheme="https" secure="true"
            keystoreFile="certs/keystore.jks" keystorePass="changeit"
            clientAuth="want" sslProtocol="TLS"
/>
```

4 Configuration

4.1 Default Configuration

The bundle is configured to run by default on localhost with a predefined set of users. To modify the default configuration, override files are provided.

The override files are found in **`\apache-tomcat\lib`**. To use an override file, place it on the classpath of the server running the bundle. For example, if using the default Tomcat bundle, externalized configuration files should be placed in the folder **`\apache-tomcat\lib`**.

The override files are:

- **`MarketplaceConfig.groovy`**
- **`Mp-override-log4j.xml`**
- **`OzoneConfig.properties`**

Each of the override files is detailed in the sections that follow.

4.1.1 Adding Users and Assigning Roles

The addition of users and assignment of Roles in the Store varies and should be customized based on individual enterprise authentication and authorization infrastructure. The Store supports three Roles:

- **ROLE_USER** (standard Store user, with no administrator privileges)
- **ROLE_ADMIN** (Store administrator role)
- **ROLE_EXTERN_ADMIN** (Used by external application to allow greater administration privileges for listings that were generated externally)

The following example outlines the procedures for adding users and Roles to the default sample security module that ships with the Store Bundle.

Note: The sample security module is included as an example and should NOT be used in a production environment.

1) Edit **`\apache-tomcat\lib\users.properties`**:

```
…
testUser1=password,ROLE_USER,Test User 1,Test 1 Org,testUser1@nowhere.com
testAdmin1=password,ROLE_ADMIN,Test Admin1,Test Admin Org,testAdmin1@nowhere.com
testAdmin2=password,ROLE_EXTERN_ADMIN,External Admin, Test Admin Org, testAdmin2@nowhere.com
…
```

2) Add user/users to the file in accordance with format as shown:

a. Username=password, role, display name, organization, email

3) Save the file and restart the server.

Any user added to the **`users.properties`** file will be added to the database upon the user's initial login.

Note: If using a custom webserver along with the provided example security, copy the **`users.properties`** *file to any directory that is on the classpath of the webserver in use.*

4.1.1.1 Adding multiple roles to a single user

The Store allows for individual users to have multiple roles within the **`users.properties`** file.

1) Edit **`\apache-tomcat\lib\users.properties`**. For example:

```
…
testUser1=password,ROLE_USER:ROLE_EXTERN_ADMIN,Test User 1,Test 1 Org,testUser1@nowhere.com
…
```

2) Add user/users to the file in accordance with format as shown:

   Username=password, role1[:roleN:roleN+1], display name, organization, email

4.2 Creating and Updating User Accounts

Upon first sign in (and regardless of the security model in use), the Store automatically creates a user's profile. The profile contains a **`UserDetails`** object which designates the user's role and can also provide:

- display name
- email address
- last date/time sign in (which will update after every sign in)

Administrators can update a user's display name and email address from the Profile section of the Administration Interface (see the Administrators Guide for details). However, if the user's display name and email address are not updated in the **`apache-tomcat\lib\users.properties`** file, a subsequent sign in by the user will overwrite any changes the administrator manually entered from the Administrative Interface.

4.3 Custom Configuration

Two of the most common customizations that require configuration file modifications are as follows:

1) The first is encountered when the default security **`.jar`** included with the bundle is replaced.
2) The second is encountered when the Store is deployed to a non-localhost environment. When this is done, all three externalized configuration files must be modified. The changes that need to be made to each file, in order to deploy to a non-localhost environment are detailed in the individual sections for each file.

Learn more about the third most-common customization, custom database (such as Oracle or MySQL) in section 5: Database Information.

4.3.1 Configuring the MarketplaceConfig.groovy File

`MarketplaceConfig.groovy` is a configuration file. It can be placed anywhere on the classpath (which is a Web server-dependent setting) and must be on the same server where **`marketplace.war`** has been deployed.

Developers comfortable with the Groovy language and the Grails application framework should be comfortable writing additional code for this file.

The following sub-sections include specific instructions for modifying configurable settings that are housed within the **`MarketplaceConfig.groovy`** file. To change default settings, go to **`\apache-tomcat\lib\MarketplaceConfig.groovy`** and replace the existing values with the desired values. Once finished, restart the Store server.
Note: For Grails developers specifically; this file is merged with the base **`Config.groovy`**.

Listed below is the default **`MarketplaceConfig.groovy`** file:

```
import org.codehaus.groovy.grails.commons.ConfigurationHolder
import grails.util.*

println "Loading custom Marketplace configurations."

// Session timeout in minutes
//httpsession.timeout=60

//enables dynamic changes to gsp files in a running webapp
//grails.gsp.enable.reload = true

// Scheduled Import timeout value, in milliseconds. Applied to the connection to the remote
//   repository when reading listings. Default is 30 minutes.
//marketplace.importTimeout = 1800000

//Set the Default Theme
//Uncomment this to replace with your custom theme if you need to make it the default theme for the
application
//marketplace.defaultTheme = "cobalt"
```

```
environments {

    production {

        dataSource {
            dbCreate = "none"
            username = "sa"
            password = ""
            driverClassName = "org.h2.Driver"
            url = "jdbc:h2:file:mktplProdDb"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }

        notifications {
            xmpp {
                username = ''
                password = ''
                host = ''
                room = ''
                port = 5222
            }
            enabled = false
        }

        log4j = {
            appenders {
                rollingFile name: 'stacktrace',
                        maxFileSize: "10000KB",
                        maxBackupIndex: 10,
                        file: "logs/stacktrace.log",
                        layout: pattern(conversionPattern: '%d{dd MMM yyyy HH:mm:ss,SSS z} %m%n')
            }
            error stacktrace: "StackTrace"
        }

            elasticSearch {
                    client.mode = 'local'  //In a clustered environment this should be 'dataNode'
```

```
                    discovery.zen.ping.multicast.enabled=false
                    discovery.zen.ping.unicast.hosts = 'localhost:9300'  //Change this to a list of
valid nodes, comma delimited
```

```
                    replicas = 1

            //Example of how to specify a file system path for the embedded elasticSearch instance
            /*path.data = new File(
                "${userHome}/.grails/projects/${appName}/searchable-index/${GrailsUtil.environment}"
            ).absolutePath*/
            }

    }

}

// Variables used for CEF logging. These fields are combined to form the CEF prefix.
// For more information about CEF logging, check ArcSight documentation
cef {
    device {
        // The device vendor or organization
        vendor = "OZONE"
        // The device product name
        product = "OWF"
        // The device version
        version = "500-27_L2::1.3"
    }
    // The CEF version
    version = 0
}

println "MarketplaceConfig.groovy completed successfully."
```

4.3.1.1 Session Timeout

By default, Marketplace will allow for 120-minutes of inactivity before the session performs a timeout operation. However, an administrator can change the length of time a user may remain inactive prior to the session actually "timing out."

1) Remove the "//" marks associated with the "**`httpsession.timeout`**" property.
2) Set the "**`httpsession.timeout`**" value to the number of inactive minutes allowed before a session timeout occurs.
3) Save the changes to the file in **`\apache-tomcat\lib`**.
4) Restart the Store server.

```
// Session timeout in minutes
httpsession.timeout=60
```

4.3.1.1.1 Timeout Warning

The Store automatically warns users of a pending session timeout by launching a 120-second (image taken at the 60-second mark) countdown in warning window, as shown below:

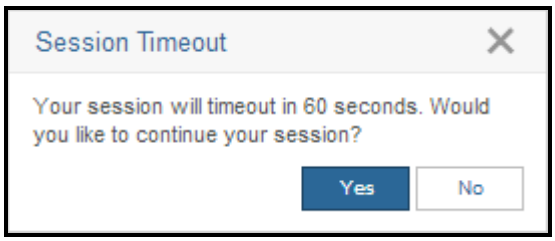

Figure 2: Timeout Warning Dialog

Two-minutes prior to a Store timeout, a Timeout Warning window will launch, notifying the user of the pending timeout. A user can restart the session timer by clicking the Yes button.

4.3.1.2 Import Timeout

By default, the Store does not restrict the amount of time an import is allowed to take. If an administrator wishes to set a time limit on an import, they can do so with the **`marketplace.importTimeout`** setting.

1) Remove the "//" marks associated with "**`marketplace.importTimeout`**".
2) Save the changes to the file in **`\apache-tomcat\lib`**.
3) Restart the Store server.
4) If a more custom timeout is required, the 180000 millisecond value can be modified to a higher or lower number, as needed:

```
// Scheduled Import timeout value, in milliseconds. Applied to the connection to the remote
//   repository when reading listings. Default is 30 minutes.
//marketplace.importTimeout = 1800000
```

4.3.1.3 Notifications

The OWF Notifications system can connect an OWF and Store to an XMPP server through which it can receive information about app components. This feature was originally designed to be used by administrators to monitor the changelogs' of Store listings, but may be used to allow any app components to communicate with OWF users. App components that connect to this XMPP server can use it to publish notification messages for consumption by OWF users. Messages published to the chat room must follow a specific format that designates both the message content and the OWF

usernames of the intended recipient users. The OWF server will then receive the messages from the chat room and distribute them to individual users.

To configure the Store to use notifications, follow these steps:

1. Create a room on an XMPP server that serves as a repository for alerts to/from OWF and the Store, respectively.
2. Create user accounts on the XMPP server for the Store and OWF service notifications.
3. Use the XMPP server information to populate the XMPP Settings in the Notifications Configuration files in OWF and the Store. Find OWF instructions in the OWF Configuration Guide. The Store fields are defined below:
   Note: Use the user accounts and their passwords from step two as the Username and Password fields in the Notifications fields in MarketplaceConfig.groovy and OwfConfig.groovy.
   - **XMPP Settings**
     - **username** – The domain name for the XMPP server that administers notifications.
     - **password** – The domain password for the XMPP server that administers notifications.
     - **host** – The hostname of the XMPP server.
     - **room** –The XMPP chat room that receives notifications.
     - **port** –The TCP port the XMPP server uses.
   - **Enabled** – Turn on or turn off notifications.

```
}

notifications {
    xmpp {
        username = ''
        password = ''
        host = ''
        room = ''
        port = 5222
    }
    enabled = false
}
```

4. Create a Public URL for the Store that will serve as the publicly accessible URL for the Store instance.

   *Note: To access the **Store Public URL** setting in the user interface: Click the drop-down user menu, click Configuration Pages, click Application Configuration and then click Additional Configurations.*

5. Notifications should be enabled and accessible to users.

4.3.1.4 ElasticSearch Index Location

The Store provides a robust search via a Lucene™-based index. The default location for the index directory and files (relative to the base Tomcat directory) is:

```
./data
```

If this location is unacceptable for a particular production environment, it can be changed as follows:

1) Edit the newly copied file and uncomment the **`elasticSearch`** tag and modify the **`path.data`** setting to the desired directory location. For example:

```
elasticSearch {
    /**
     * The location of the data files of each index / shard allocated on the node.
     */
    path.data = new File(
            "${userHome}/.grails/projects/${appName}/searchable-index/production"
    ).absolutePath
}
```

2) Re-start the server for this change to take effect. Once restarted, the data will be re-indexed to the specified location.

Note: To provide full-text-search functionality, Marketplace uses an embedded [elasticsearch] server that by default accepts HTTP requests on port 9200. If the port is taken, ElasticSearch will use the next available port in the 9200-9300 range.

4.3.1.5 CEF Logging—Additional Fields

Some customers use specific software to analyze their CEF logs. If your organization uses this software, add the following fields to your groovy file and complete them:

```
// Variables used for CEF logging. These fields are combined to form the CEF prefix.
// For more information about CEF logging, check ArcSight documentation
cef {
    device {
        // The device vendor or organization
        vendor = "OZONE"
        // The device product name
        product = "OWF"
        // The device version
        version = "500-27_L2::1.3"
    }
    // The CEF version
    version = 0
}
```

For more information about CEF logging, see section 4.4: Application Logging.

4.3.2 MarketplaceConfig.xml File

At start up, the store will look for configuration metadata in a file called **`MarketplaceConfig.xml`** and apply any custom settings there, overriding the defaults. This file can be placed anywhere on the class path (e.g. in the bundle this can be **`\apache-tomcat\lib`**). In general, configuring the application this way is not needed, and not recommended. As such the file is not included with the bundle. Dependency injection is an advanced topic, beyond the scope of this documentation. For more information, including example configuration files, please see: http://docs.spring.io/spring/docs/2.5.6/reference/beans.html.

A skeleton example looks like this:

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Insert bean configurations here -->

</beans>
```

4.3.3 MPSecurityContext.xml

The Store allows developers to customize the system behavior by supplying custom plugins. There are currently two places where custom plugins can be supplied. They are the Security Plugin and the custom header and footer. Administrators can customize the header and footer through the administration controls in the user interface; for more information, see the Store Administrator's Guide.

`MPSecurityContext.xml` (found in the **`\apache-tomcat\lib`** directory) is a Spring Framework bean configuration file. For the security changes to take effect, restart the Tomcat server. Please see 2.3.4: Pluggable Security for more details on changes to security.

4.3.3.1 Enable Synchronization with OWF

The synchronization feature allows **the Store** to automatically send app components and their subsequent updates to OWF. Synchronization is also necessary to push OWF applications to the Store. To use this feature, developers must do the following:

- Configure **`OwfConfig.groovy`** to accept synchronization messages from the Store.

- Configure `OWFsecurityContext.xml` and `MPSecurityContext.xml` (see below).
- Synchronize the OWF server with the Store through the Store's Administration Configuration pages.

To implement the synchronization feature with the OWF sample security plugin, configure the following OWF and Store files (found in the ***apache-tomcat-7.0.21*** → ***lib*** directory):

In OWF:

- **OWFsecurityContext.xml** must include the following in the top-level ***<beans>*** element:

  <sec:http pattern="/marketplace/sync/**" security="none" />

In the Store:

- **MPSecurityContext.xml** must include the following in the top-level ***<beans>*** element:

  <sec:http pattern="/public/descriptor/**" security="none" />

The Store will only synchronize with OWF servers linked to that Store. The Store will not synchronize with unspecified OWF systems or offline OWF systems. Synchronize OWF servers with the Store through the Store Administration Configuration pages. For more information, see the OZONE Store Administrator's Guide.

To use this feature:

1. From the drop-down User Menu, click Configuration Pages.
2. Links to reference data will appear, click Application Configuration.
3. In the left-panel, click Additional Configurations. To synchronize with an OWF instance, add its URL in the OWF Sync Servers section:
   Note: To connect to multiple OWFs, uses commas to separate their URLS.

Figure 3: OWF Sync Field

4. Listings that meet the OWF criteria described in the Configuring Listings to Appear in OWF section of the Store Administrator's Guide will appear in the Store in OWF (if listing approval is enabled in OWF) or directly in the administrator's App Component Manager in OWF if auto-approved is enabled.

4.3.3.2 Server Settings

All references to the Store must match the current installation settings. The table below shows the values of the settings and optional CAS settings. Based on the settings in **`OzoneConfig.properties`**, the variables (e.g. ${ozone.host}) are filled in at runtime.

Table 4: MPCasBeans.xml Server Settings

Bean ID/Property	Purpose	Value
casProcessingFilterEntryPoint/ loginUrl	Must point to the CAS login page.	https://${ozone.host}:${ozone.port}/ ${ozone.cas.serverLoginLocation} (e.g. https://servername:port/cas/login)
serviceProperties/ service	Must point to the Store web server.	https://${ozone.host}:${ozone.port}/ ${ozone.cas.marketplace.jSpringCasSecurityCheckLocation} (e.g. https://servername:port /marketplace/j_spring_cas_security_check)
ticketValidatorFactory/ casServiceUrl	Must point to the CAS server.	https://${ozone.host}:${ozone.port}/ ${ozone.cas.serverName} (e.g. https://servername:port/cas)
ticketValidatorFactory/ proxyCallbackUrl	Must point to the Store web server.	https://${ozone.host}:${ozone.port}/ ${ozone.cas.marketplace.serverSecureReceptorLocation} (e.g. https://servername:port/marketplace/secure/receptor)

Table 5: MPLogInOutBeans.xml Server Setting

Bean ID/Property	Purpose	Value
OzoneLogoutSuccessHandler/constructor-arg/index=1	Must point to the CAS logout page.	https://${ozone.host}:${ozone.port}/${ozone.cas.serverLogoutLocation} (e.g. https://servername:port/ cas/logout)

4.3.4 JVM Memory Settings

Adjusting a server's memory settings can increase performance or resolve PermGen **`OutOfMemory`** errors. To adjust memory settings:

Note: OWF will automatically run the JVM in server mode. For optimum performance, ensure that the JVM supports server mode.

1) In **`setenv.sh`** or **`setenv.bat`** (located in **`\apache-tomcat\bin`**), set the initial PermGen size to at least 256 MB. This can be accomplished by adding -XX:PermSize=256m to the Java options. If more memory is available on the server, increasing this PermGen size may increase performance.
2) Set the maximum PermGen size to at least 384 MB. This is accomplished by adding -XX:MaxPermSize=384m to the Java options. If more memory available on the server, increasing this PermGen size may increase performance.

4.4 Application Logging

Application Logging can be a useful troubleshoot and debugging tool. The bundle ships with Common Event Format (CEF) logging level turned on by default. However, due to the additional amount of data that is generated, using the more detailed settings for extended periods may slow performance. A system administrator can change the setting to stop recording detailed information that can be used to diagnose problems. They can also relocate where the files are stored.

The toggle controls for both CEF and Object Access auditing and relocation of files are found in the Store's Application configurations, found under the drop-down User Menu in the user interface. For more information, see the OZONE Store Administrator's Guide.

When enabled, CEF auditing records common user events:

- Log in and out (Log out - Marketplace only)
- Create, Read*, Edit and Delete
- Search
- Import and Export

*Note: Object Access auditing, is a separate CEF auditing feature that records users' Read events. Read events are logged when **both** the CEF auditing global flag and the Object Access flag are ON in the Store's Application configurations. If the Object Access auditing is ON and the CEF auditing is OFF, no Read events are logged with CEF auditing.*

The following are two log examples using CEF auditing.

CEF auditing from an object modification event:

```
26 Jun 2013 12:31:37,217 EDT CEF:0|COMPANY|STORE|500-27_L2::IC::1.3|FILEOBJ_MODIFY|Object was
updated|7|cat=FILEOBJ_MODIFY suid=MikePAdmin shost=127.0.0.1 requestMethod=USER_INITIATED
outcome=SUCCESS deviceFacility=0CCB5827C819E1A4AC9BE5BD4C6F9FE9.mp02 reason=UNKNOWN cs5=UNKNOWN
act=7.2 deviceExternalId=UNKNOWN dhost=example.hostname.owfgoss.org cs4=UNKNOWN start=06:26:2013
12:31:37 [.037]cs3=UNKNOWN fname=[CLASS:marketplace.OwfProperties, stackContext:,
stackDescriptor:] filePermission=UNKNOWN fileId=16 fsize=2 fileType=OBJECT
oldFilename=[CLASS:marketplace.OwfProperties, stackContext:null,
stackDescriptor:null]oldFilePermission=UNKNOWN oldFileId=16 oldFileSizesize=2 oldFileType=OBJECT
```

CEF auditing from a log on event:

```
26 Jun 2013 08:57:51,974 EDT CEF:0|COMPANY|STORE|500-27_L2::IC::1.3|LOGON|A logon event
occured.|7|cat=LOGON suid=MikePAdmin shost=127.0.0.1 requestMethod=USER_INITIATED outcome=SUCCESS
deviceFacility=8C24A08B7E9848C80F929791DA40F734.mp02 reason=UNKNOWN cs5=UNKNOWN act=7.2
deviceExternalId=UNKNOWN dhost=example.hostname.owfgoss.org cs4=UNKNOWN start=06:26:2013
08:57:51 [.051]cs3=UNKNOWN
```

4.5 Adding a SYSLOG

An administrator can modify configuration so that the Store writes to a syslog.

The included log4j syslog appender uses UDP. An administrator can configure their syslog daemon to accept UDP connections. See the example for CentOS below:

1) Configure syslog to allow UDP messages:

```
sudo /etc/sysconfig/sysconfig
SYSLOGD_OPTIONS="-m 0 \-r \-x"
```

2) Modify the mp-audit-log appender:

```
sudo vi /opt/omp/mp-tomcat-config/apache-tomcat-6.0.30/lib/mp-override-log4j.xml

<appender name="mp-audit-log" class="org.apache.log4j.net.SyslogAppender">
  <param name="syslogHost" value="localhost" />
  <param name="Facility" value="SYSLOG" />
  <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="%-5p %m%n" />
  </layout>
</appender>
```

3) Restart syslog:

```
sudo /sbin/service syslog restart
```

5 Database Information

5.1 Custom Database Installation

While the full extent of administering a database is outside the scope of this guide, the following sections provide information on how to configure Marketplace to work with various databases.

Note: The Development Team recommends using current database drivers. For further information, please reference external database documentation.

5.1.1 Database Setup

The Store ships with the following default database configuration found in **`/apache-tomcat/lib/MarketplaceConfig.groovy`**. This file points to the local HyperSQL that is shipped with the bundle. This database should not be used in a production environment. Use the instructions in the following sections to modify the following configuration to point to a production database.

```
        dataSource {
            dbCreate = "update" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL,
Postgresql or SQL Server
            username = "sa"
            password = ""
            driverClassName = "org.h2.Driver"
            url = "jdbc:h2:file:mktplProdDb"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

5.1.1.1 Oracle

1) Create an Oracle Database user for the Store. It is recommended that there be a dedicated user for the Store, in order to avoid database object name collisions.
2) The bundle does not provide a JDBC driver for Oracle. Download the appropriate JDBC driver and place it into the web server's classpath. For Tomcat, the driver can be placed in the **`/apache-tomcat/lib`** directory.
3) Open **`/apache-tomcat/lib/MarketplaceConfig.groovy`** and change the Environments → Production → dataSource section using the values that are appropriate for the Store environment. For example:

```
        dataSource {
            dbCreate = "none" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL, Postgresql
or SQL Server
            username = "aml_user"
            password = " aml_password"
            dialect="org.hibernate.dialect.Oracle10gDialect"
            driverClassName = "oracle.jdbc.driver.OracleDriver"
            url = "jdbc:oracle:thin:@DEVDB1:1521:XE"
            driverClassName = "org.h2.Driver"
            url = "jdbc:h2:file:mktplProdDb"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1 FROM DUAL"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

In the example above, an Oracle database-user named **aml_user** with a password of **aml_password** was created. There is a connection to an Oracle 10g database using a thin JDBC driver on host **`myhost.somewhere.org`** with a **SID** of **DEVDB1** on Oracle's default **port** of **1521**.

There are several different types of Oracle drivers (thin, OCI, kprb) and connection options (service, SID, TNSName) available. Please consult the Oracle DBA and Oracle's JDBC documentation to create the connection most appropriate for the installed environment.

4) Save the changes to **`MarketplaceConfig.groovy`**.

5) Create the database objects.
    - If you are upgrading an existing Store, refer to Upgrading .
    - Otherwise, run the **`dbscripts/OracleCreate.sql`** script. This script will create all the database objects.
      Note: In previous versions of the bundle, the system automatically generated database objects. This is no longer the case, developers must run this script.
    - Optional step: run **`OracleDefaultDataCreate.sql`** to create default data like categories, types, etc. Administrators can edit the default data from the user interface to customize it for their system.
6) Verify that your dbCreate setting in **`MarketplaceConfig.groovy`** is set to none.
7) Start the Store.

Note: The Store team is aware of a known issue with the Oracle Web-based admin Console returning truncated characters when dealing with large data sets. Accordingly, using SQLPlus or SQLDeveloper to run the script mentioned herein is recommended.

5.1.1.2 MySQL

The Store requires the use of transactional tables. The **`MySqlCreate.sql`** script creates all tables with the character set for ISO-8859-1 (informally known as latin1). The upgrade scripts also create any new tables with the character set for ISO-8859-1 (informally known as latin1).

1) Create a schema within MySQL for use with the Store. It is recommended that there be a dedicated schema for the Store to avoid database object name collisions.
2) Create a MySQL user with full access to the Store schema created above.
3) The bundle does not provide a JDBC driver for MySQL. Download the appropriate JDBC driver and place it into the Web server's classpath. In Tomcat, the driver can be placed in the **`/apache-tomcat/lib`** directory.
4) Open **`/apache-tomcat/lib/MarketplaceConfig.groovy`** and change the Environments → Production → dataSource section using the appropriate values. For example:

```
        dataSource {
            dbCreate = "none" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL, Postgresql
or SQL Server
            username = "aml_user"
            password = "aml_password"
            driverClassName = "com.mysql.jdbc.Driver"
            dialect  = "org.hibernate.dialect.MySQL5InnoDBDialect"
            url="jdbc:mysql://myhost.somewhere.org:3306/aml "
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

In the example above, a MySQL database-user named **aml_user** with a password of **aml_password** is used to connect to the **aml** database on host **myhost.somewhere.org**.

5) Save the modified **`MarketplaceConfig.groovy`**.

6) Create the database objects.

   - To upgrade an existing Store, refer to Upgrading .
   - Otherwise, run the **`dbscripts/MySqlCreate.sql`** script. This script will create all the database objects.
     Note: In previous versions of the bundle, the system automatically generated database objects. This is no longer the case, developers must run this script.
   - Optional step: run **`MySqlDefaultDataCreate.sql`** to create default data like categories, types, etc. Administrators can edit the default data from the user interface to customize it for their system.

7) Verify that the dbCreate setting in **`MarketplaceConfig.groovy`** is set to none.

8) Start the bundle.

*Note: Modify the **`.sql`** script (mentioned above) with the appropriate schema name. For example:*

```
use aml;
```

5.1.1.2.1 Configuring MySQL JDBC to use SSL

By default MySQL communicates over an unencrypted connection. However, in most cases, it can be configured to use SSL. This is somewhat implementation specific.

Note: This capability completely depends on how the implementation's MySQL binaries were compiled, packaged, etc.

The following procedure covers how to configure an SSL capable MySQL server to work with a Franchise Store bundle. The starting point for the server implementation used in this example is:

- Operating System: CentOS 6.4, 64 bit minimal installation (no updates were applied)
- MySQL Server v5.1.69 (achieved by installing the "mysql-server" package with yum)

This procedure was developed from the MySQL 5.1 documentation, specifically the following sections:

- 6.3.6. Using SSL for Secure Connections
- 20.3.5. Connector/J (JDBC) Reference

This procedure relies on self-signed certificates. It is for testing and demonstration purposes only. The following three sections explain the configuration steps:

5.1.1.2.1.1 Step 1: Creating MySQL Server's CA and Server

The following steps guide a developer through creating the CA and Server certificates for a MySQL server:

1) **Create the CA Key and Certificate**
   From a shell prompt on a MySQL server, type the following commands:

```
prompt> openssl genrsa 2048 > ca-key.pem
prompt> openssl req -new -x509 -nodes -days 365 -key ca-key.pem -out cacert.pem
```

   Note: After the second command, the system prompts the developer to provide basic identity information. For the purpose of this demonstration, it is not important what information the developer provides here. However, it will be necessary to provide the same information in the next step.

2) **Create the Server Certificate**
   From a shell prompt on your MySQL server, type the following commands. After the first command, the developer will again be prompted to provide identity information. The developer must provide the same information that they provided in Step 1. However, when prompted for the Common Name, provide the MySQL server's hostname (e.g. mysql.mydomain.com).

```
prompt> openssl req -newkey rsa:2048 -days 365 -nodes -keyout server-key.pem -out server-req.pem
prompt> openssl rsa -in server-key.pem -out server-key.pem
prompt> openssl x509 -req -in server-req.pem -days 365 -CA cacert.pem -CAkey ca-key.pem -set_serial 01
-out server-cert.pem
```

3) **Consolidate the Output from Steps 1 & 2**

Copy the following files (produced in the preceding two steps) to a location where at a minimum the MySQL user has read access.
In this example, it is **`/etc/ssl/certs/`**

- **`cacert.pem`**
- **`server-key.pem`**
- **`server-cert.pem`**

5.1.1.2.1.2 Step 2: Configure MySQL

1) **Edit my.cnf**
Edit the MySQL configuration file. For this example, the file is located at **`/etc/my.cnf`**. Add the lines shown in bold to the [mysqld] section of the file.

```
ssl-ca=/etc/ssl/certs/cacert.pem
ssl-cert=/etc/ssl/certs/server-cert.pem
ssl-key=/etc/ssl/certs/server-key.pem
```

2) **Restart MySQL**

```
sudo service mysqld restart
```

3) **Create the MySQL Franchise Store Schema**
At a minimum, create the schema and assign permissions as you would normally, following the steps in section 4.1.1: Adding Users and Assigning Roles. This will leave the choice of using an encrypted connection up to the connecting client (in this case the Franchise Store application).
To enforce the use of SSL from the database server side, add REQUIRE SSL to the end of the grant statement where user permissions to the Franchise Store schema are assigned. For example:

```
GRANT ALL ON franchise.* TO 'franchise'@'storehost.mydomain.com' IDENTIFIED BY 'franchise' REQUIRE
SSL;
```

5.1.1.2.1.3 Step 3: Configure the Franchise Store Bundle

1) **Modify MarketplaceConfig.groovy**
Add **`useSSL=true`** to the **`dataSource URL`** configuration. This can also be enforced from the client side with the **`requireSSL`** option. For example:

```
url = """jdbc:mysql://192.168.56.11/franchise?useSSL=true&
                    requireSSL=true&trustCertificateKeyStoreUrl=
                    file://${System.properties['javax.net.ssl.trustStore']}&
                    trustCertificateKeyStorePassword=changeit"""
```

2) **Modify the Application TrustStore**
Add the CA certificate, created above (**`cacert.pem`**) to the application's trust store. In the case of the franchise bundle, the trust store is a file called **`keystore.jks`** found in **`$BUNDLE_PATH/apache-tomcat/certs`**. Do this with the following command (assuming you have the JDK installed and JAVA_HOME/bin on your PATH, if they aren't there, add them first).

```
prompt> keytool -import -alias mysqlCAcert -file cacert.pem -keystore keystore.jks
```

3) **Start the Application**

```
prompt> ./start.sh
```

5.1.1.3 PostgreSQL

1) Create either a new login role or a new schema in order to avoid database object name collisions between the Store and other database applications.
2) Edit the user so that it can create database objects.
3) Create a new database. Use UTF8 as the encoding (default).
4) The bundle does not provide a JDBC driver for PostgreSQL. Download the appropriate JDBC driver and place it into the web server's classpath. In Tomcat, the driver can be placed in the **`/apache-tomcat/lib`** directory.
5) Open **`/apache-tomcat/lib/MarketplaceConfig.groovy`** and change the Environments → Production → dataSource section using the appropriate values. For example:

```
        dataSource {
            dbCreate = "none" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL, Postgresql
or SQL Server
            username = "aml_user"
            password = "aml_password"
            driverClassName = "org.postgresql.Driver"
            url = "jdbc:postgresql://myhost.somewhere.org:5432/aml"
            dialect="org.hibernate.dialect.PostgreSQLDialect"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
```

```
                validationQuery = "SELECT 1"
                validationInterval = 30000

                maxActive = 100

                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

In the example above, a PostgreSQL database user named **aml_user** was created with a password of **aml_password** is used to connect to the **aml** database on host **myhost.somewhere.org**.

6) Create the database objects. To upgrade an existing Store, refer to Appendix A Upgrading the StoreUpgrading .
7) Otherwise, run the **dbscripts/PostgresqlCreate.sql** script. This script will create all the database objects.
   Note: In previous versions of the bundle, the system automatically generated database objects. This is no longer the case, developers must run this script.
   a. Optional step: run **PostgresqlDefaultDataCreate.sql** to create default data like categories, types, etc. Administrators can edit the default data from the user interface to customize it for their system.
8) Verify that the dbCreate setting in **MarketplaceConfig.groovy** is set to none.
9) Start the Store.

5.1.1.4 SQL Server

1) Create a new SQL Server database for use with the Store.
2) Create a SQL Server user with full access to the Store database created above.
3) Run the following SQL command from the master database to enable row level versioning for the Store database:

```
ALTER DATABASE <name of database>
SET READ_COMMITTED_SNAPSHOT ON;
```

4) The bundle does not provide a JDBC driver for SQL Server. Download the appropriate JDBC driver and place it on the Web server's classpath. In Tomcat, the driver can be placed in the **/apache-tomcat/lib** directory.
5) Open **/apache-tomcat/lib/MarketplaceConfig.groovy** and edit the newly copied file and modify the Environments → Production → dataSource and

Environments → Production → Hibernate sections using the values that are appropriate for the Store environment. For example:

```
        dataSource {
            dbCreate = "none" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL, Postgresql
or SQL Server
            username = "aml_user"
            password = "aml"
            driverClassName = "net.sourceforge.jtds.jdbc.Driver"
            dialect="ozone.marketplace.domain.NSQLServerDialect"
            url = "jdbc:jtds:sqlserver://myhost.somewhere.org:1443/aml"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

In the example above, the SQL Server database-user named **aml_user** was created with a password of **aml**.

6) Create the database objects.
    a. To upgrade an existing Store, refer to Upgrading .
    b. Otherwise, run the **`dbscripts/SQLServerCreate.sql`** script. This script will create all the database objects.
    Note: In previous versions of the bundle the system automatically generated database objects. This is no longer the case, developers must run this script.
    c. Optional step: run **`SQLServerDefaultDataCreate.sql`** to create default data like categories, types, etc. Administrators can edit the default data from the user interface to customize it for their system.
7) Verify that the dbCreate setting in **`MarketplaceConfig.groovy`** is set to none.
8) Start the Store.

5.2 Custom Database Configurations

While the full extent of administering a database is outside the scope of this guide, the following sections provide information on how to configure the Store to work with various databases. See the sections on individual files to determine the configuration changes that can be made.
Database customization is one of the most common configuration file modifications to the Store. When a custom database (such as Oracle or MySQL) is used, it requires a change to the **`MarketplaceConfig.groovy`** file. For information about using a custom database see section 5.1.1: Database Setup. Information about other changes to the **`MarketplaceConfig.groovy`** file is found in section 4.3.1: Configuring the MarketplaceConfig.groovy File.

5.2.1 Database Settings

Listed below are the variable database elements that need to be modified as new databases are implemented. A detailed explanation of each field follows.

```
        dataSource {
            dbCreate = "update" // IMPORTANT! Set this parameter to 'none' for Oracle, MySQL, Postgresql
or SQL Server
            username = "sa"
            password = ""
            driverClassName = "org.h2.Driver"
            url = "jdbc:h2:file:mktplProdDb"
            pooled = true
            properties {
                minEvictableIdleTimeMillis = 180000
                timeBetweenEvictionRunsMillis = 180000
                testOnBorrow = true
                testWhileIdle = false
                testOnReturn = false
                validationQuery = "SELECT 1"
                validationInterval = 30000
                maxActive = 100
                initialSize = 10
                maxIdle = 50
                minIdle = 10
                removeAbandoned = false
                jdbcInterceptors = null
            }
        }
```

Table 6: Externalized Database Properties

Property	Purpose	Example
dbcreate	How database is created/updated.	none

Property	Purpose	Example
username	Username for database connections. *Note: This field is database specific.*	Admin
password	Password for database connections. *Note: This field is database specific.*	password
driverClassName	The JDBC driver. *Note: This field is database specific.*	org.postgresql.Driver
url	The Connection String. *Note: This field is database specific.*	jdbc:postgresql://myhost.somewhere.org:5432/aml
pooled	Enable database connection pooling when true	true
minEvictableIdleTimeMillis	Minimum amount of time in milliseconds an object can be idle in pool before eligible for eviction	"180000"
timeBetweenEvictionRunsMillis	Time in milliseconds to sleep between runs of the idle object evictor thread	"180000"
testOnBorrow	When true objects are validated before borrowed from the pool	true
testWhileIdle	When true, objects are validated by the idle object evictor thread	false
testOnReturn	When true, objects are validated before returned to the pool	true
validationQuery	Validation query to run on target database *Note: This field is database specific.*	SELECT 1
validationInterval	Frequency of validations on connections	3000
maxActive	Maximum number of active connections that can be allocated from this pool at the same time	100
initialSize	Initial number of connections that are created when the pool is started	10
maxIdle	Maximum number of connections that should be kept in the pool at all times.	50

Property	Purpose	Example
minIdle	Minimum number of established connections that should be kept in the pool at all times.	10
removeAbandoned	Ends connections if they exceed the removeAbandonedTime. (If set to true a connection is considered abandoned and eligible for removal if it has been in use longer than the removeAbandonedTimeout). *Note: Setting this to true can recover db connections from applications that fail to close a connection.*	false
jdbcInterceptors	A semicolon separated list of classnames extendingorg.apache.tomcat.jdbc.pool.JdbcInterceptor class.	null

6 Creating and Editing Themes

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)

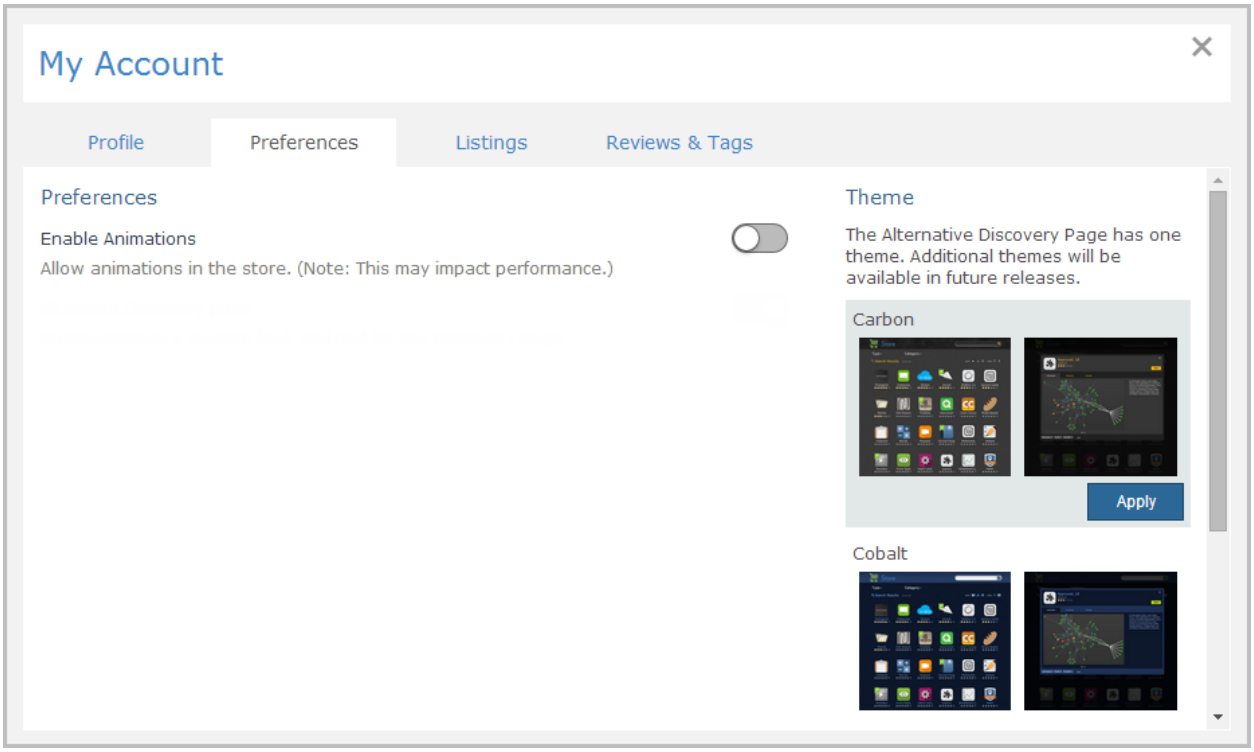

Figure 4: Themes Menu

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.

6.1 Changing the Default Theme

The default theme is named Cobalt. This is the theme that users will see when they first access the Store. The default theme can easily be changed to any available theme, such as a custom theme created for the store. To change the default theme, modify the **`/apache-tomcat/lib/MarketplaceConfig.groovy`** file as follows and restart the server:

1. In **`MarketplaceConfig.groovy`**, locate the following lines of code:

```
//Set the Default Theme
```

```
//Uncomment this to replace with your custom theme if you need to make it the default theme for
the application
//marketplace.defaultTheme = "cobalt"
```

2. Delete the two slashes ("//") in front of **`marketplace.defaultTheme`** and replace "cobalt" with a custom theme name that will become the default theme for the store.
3. For example, to change the default theme to the "Oxygen" theme, these lines would look like:

```
//Set the Default Theme
//Uncomment this to replace with your custom theme if you need to make it the default theme for
the application
marketplace.defaultTheme = "Oxygen"
```

4. Save the file.
5. Restart the Markeplace server.

6.2 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.

6.2.1 Changing the header tooltip

Figure 5: Header Tooltip

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

To update this text:

1) In the exploded **`.war`**, go to **`WEB-INF\grails-app\i18n`** folder and open the **`messages_overlay.properties`** file.
2) Change the **`tooltip.ompHeaderLogo=Marketplace`** line to the new name:

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

6.3 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/.

6.3.1 Prerequisites

To create and modify themes, the developer will need:

- Compass 0.11.7
- SASS 3.1.1 (required by Compass)
- Ruby 1.9.2 or higher (required by SASS and Compass)

Note: Developers must use these exact versions of Compass and SASS must be used in order to successfully compile the theme files.

To obtain these dependencies:

1) Install Ruby (http://www.ruby-lang.org/en/downloads/).
2) Use the included "gem" tool to install SASS and Compass by running **gem install compass -v 0.11.7**. as an administrator.
3) Confirm that SASS and Compass are on the system PATH.

6.3.2 Layout of Themes Directory

To locate the theme files, unzip the **`apache-tomcat/webapps/marketplace.war`** and open the **`themes`** directory. The following figure shows the layout of the theme files:

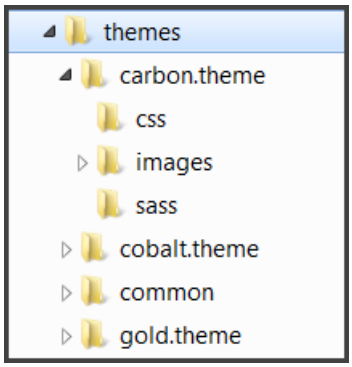

Figure 6: Themes Directory Structure

The following table offers a brief description of the theme folders and files which are found under **`marketplace.war/themes`**:

Table 7: Theme File and Folder Descriptions

File or folder name	Description
/compile_all_themes.bat or /compile_all_themes.sh	Shell scripts for Windows or UNIX that automate the process of compiling the themes.
/watch_all_themes.bat or /watch_all_themes.sh	Shell scripts for Windows or UNIX that start watch-processes on all themes, which will automatically recompile the stylesheets of the themes whenever a change is detected.
/cobalt.theme	Parent folder for the default theme, cobalt
/gold.theme	Parent folder for the gold theme
/carbon.theme	Parent folder for the carbon theme
/common	Parent directory for files that are likely to be used by most or all themes
/common/images	Directory for images that are common to many themes or can serve as defaults
/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 creating new stylesheets in the common folder, they should be included in this file
/common/stylesheets/variables/	Contains variable definitions for Store specific components
/common/stylesheets/variables/_constants.scss	Variables values that should generally not change
/common/stylesheets/variables/ *all other files	Variables 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 *.css. 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 folders that a theme touches. The files and folders are found under the directories of the specific theme that they modify. In the Store, the parent theme directories include **`.theme`** in their naming convention. Example: **`carbon.theme.`** The table below uses an example theme named **`example.theme`**—this example is not included in the bundle (however, look at the **`themes/template`** directory for a reference).

Table 8: Theme Conventions

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	The 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	Location of theme-specific images. The Store searches for images here first. If they are not found, it searches 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 **common/stylesheets/variables/** files. It defines the theme background and imports the desired files from **common/stylesheets**.
example.theme/sass/config.rb	This file sets ruby variables used by Compass to locate resources. Do not modify it.

6.4 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.
2) Copy the **`themes/cobalt.theme`** directory to **`themes/<theme_name>.theme`**; substitute the name created in step 1 for the theme for **`<theme_name>`**.
3) Delete the **`cobalt.css`** file in the **`themes/<theme_name>.theme/css`** directory.
4) Navigate into the **`<theme_name>.theme/images/preview`** directory and delete the contained files.
5) Navigate into the **`<theme_name>.theme`** directory.
6) Edit **`theme.json`**. Change every reference to "cobalt" to "**`<theme_name>`**".
   - The **`name`** attribute must use the **`<theme_name>`** chosen in step 1.
   - The **`display_name`** attribute should contain a user-friendly, readable name for the theme (It can include spaces and capital letters.) as it will appear in the theme picker.
   - The **`css`** attribute must use **`themes/<theme_name>.theme/css/<theme_name>.css`**
   - All URL properties are relative to the context root. 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.
   a) Set the **`$theme-name`** variable to the **`<theme_name>`** chosen in step 1.
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 numerious 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. 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 do a one-time compile of only the theme being referenced above, navigate to the **sass** directory and run **compass compile** from a terminal window. Passing in the **--force** option will force a recompile even when source files do not appear to be changed.
- To start a process that will continue to watch SCSS for changes and recompile as needed, navigate to the **sass** directory and run the "**compass watch**" from a terminal window.
- To do a one-time compile of ALL themes, navigate to the **themes** directory and run the appropriate script.
    a) On a Windows system, this is done by running **`compile_all_themes.bat`**
    b) On UNIX systems, this is done by running **sh compile_all_themes.sh**
    Note: The scripts assume that their current directory is the themes directory.
- To start watches on ALL themes, navigate to the themes directory and run the appropriate script. These scripts assume that the current directory is the **themes** directory.
    a) On a Windows system, run **watch_all_themes.bat**. This will open a new, minimized command prompt for each theme, in which **compass watch** will be running.
    b) On UNIX systems, run **`sh watch_all_themes.sh`** from a terminal window. This will start **`compass watch`** as a background process for each theme. The script will continue to run until the watches exit, but can be terminated with a CTRL-C.

12) Once the theme successfully compiles, verify that **`<theme_name>.css`** has been created in the **`css`** directory, 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 **`grails -Duser=testAdmin1 run-app -https`** from the top directory of the source tree.
  Note: the exact syntax of this command is Operating System specific. On Windows use one dash, on UNIX use two dashes.
- Build the Store and start the Tomcat server. To build the Store, run **ant** from the top directory of the source tree. To start the build server, run **`start.bat`** or **`start.sh`** located in **`/apache-tomcat`**.
  Note: before running ant, you may need to run 'ant init-build'.

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. Screenshots should be saved in the **`<theme_name>.theme/images/preview`** directory so that they show up in the theme picker. Lastly modify the **`<theme_name>.theme/theme.json`** file and add/modify the screenshot entries of the images which are stored in the **`/images/preview`** folder.

6.5 Customizable Theme Components

This section lists optional theme-able components. To apply a custom theme to any of the theme-able components see section 6.3: Creating and Modifying Themes.

All theme-able components are located in the **`marketplace.war/themes/common/stylesheets`** directory as described in the following table:

Table 9: Theme Components

File that can be overridden	Theming Component(s)
_aboutWindow.scss	About Window
_actionMenu.scss	Action Menu
_admin.scss	Admin Home Page
_affiliated_search_results.scss	Affiliated Search Results carousel
_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	Header
_helpModal.scss	Help Modal window
_landing.scss	Landing Page – Getting started carousel, listing carousels
_list_group.scss	List styles – Change log and recent activity
_listingDetail.scss	Listing Details page
_loadmask.scss	Loading mask

File that can be overridden	Theming Component(s)
_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	Filter Menus
_profile.scss	User Profile modal window
_quickview.scss	Quick View modal window
_scorecard.scss	Scorecard
_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	Theme Switcher Window
_tutorial.scss	Tutorials
_widget.scss (deprecated)	The Store in Widget Mode in OWF
_wizard.scss	Import/Export Wizard
\gidgets\affiliatedMarketplace\registry_registry.scss	Affiliated Store Registry Grid and Entry Window
\gidgets\affiliatedMarketplace\search_search.scss	Affiliated Store top search

File that can be overridden	Theming Component(s)
	and notification section & bottom affiliated results section on Search Results page
\gidgets\profile\bio_bio.scss	User Profile page Bio section
\gidgets\profile\contributions_contributions.scss	User Profile page Contributions section (Tab Panel)
gidgets\profile\extern_extern.scss	External User Profile page External System Section
gidgets\profile\view_view.scss	User Profile page Avatar section
gidgets\serviceItem\activity_activity.scss	Detailed Listing Page Change Log section (Tab Panel)
gidgets\serviceItem\list\user\active_active.scss	My Listings Page Active Listings section (Grid Panel)
gidgets\serviceItem\list\user\pending_pending.scss	My Listings Page Pending Listings section (Grid Panel)

6.6 Minifying and Compressing Files

Source JavaScript and CSS files are included in the **`marketplace.war`**. To modify any JavaScript or CSS file, change the original un-minified file found in the **`marketplace.war`**. Minify and compress the changed version using the naming conventions mentioned below in step 3.

The following example explains how to modify **`cobalt.css`**. Follow this example to change other JavaScript and CSS files in the **`.war`**:

1) From **`/web-app/themes/cobalt.theme/css`**, modify the original un-minified the Store version of the **`cobalt.css`** file.
2) To replace the minified and compressed version that shipped with the bundle with the updated file, the updated file should follow the naming convention of: marketplace_v5-28902.css (where 5-29802 is the version of the minified and compressed files).

3) Minify the new **`cobalt.css`** using a minification tool like a YUI Compressor Tool.

4) Use a gzip compression utility to compress this file to one that follows the naming convention of: marketplace_**v5-28902**.css (where 5-29802 is the version of the minified and compressed files).

Repackage the **`.war`** and deploy. Changes should appear after clearing the browser cache.

7 Creating Custom Field Types

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

Note: 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.

Note: This method 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.

7.1 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)
```

Note: 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:

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

- 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;
```

Appendix A Upgrading the Store

A.1 Upgrading

1) Backup Everything:

   Before starting the upgrade, backup the entire deployment of the Store and the corresponding database. Make sure all custom override configuration files have been included in the backup (In Tomcat they are normally located in the **`lib`** directory).

2) Unpack the bundle:

   The following example shows how an administrator might copy and unzip the Store from the bundle on **Windows** operating systems:

   a) Create a new directory from where the Store will be run. This can be done via the Windows UI or the command prompt.
   b) Copy the Bundle Zip File to the new directory created in step a.
   c) Right-Click on **`marketplace-bundle-7.16.0.zip`**, and select "open," "explore" or the command for the system's default zip/unzip program.
   d) Unzip/unpack the bundle into the new directory created in step a.

3) Configure `OWFsecurityContext.xml` and `MPSecurityContext.xml` (found in the **`apache-tomcat`** → **`lib`** directory):

   Update the security context file syntax:
   a) **`intercept-url`** elements which include the **`filters="none"`** attribute are no longer supported, instead use additional '**`http`**' elements which contain a **`security="none"`** attribute and a **`'pattern'`** attribute equivalent to the **`'pattern'`** attribute on the old **`'intercept-url'`** element.

   b) The **`xsi:schemaLocation`** attribute on the parent '**`beans`**' element must be updated to the following:

```
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org
/schema/beans/spring-beans-3.2.xsd
http://www.springframework.org/schema/security http://www.springframework.org/schema/security/
spring-security-3.2.xsd"
```

4) Upgrade Database:
   If you are upgrading from version 7.14.2, skip this step. If you are upgrading from version 5 to 7.3.0, you need to run the first upgrade script the version 5 to 7.3.0 found in the **`dbscript`** directory. Then, run the **`_v7.3.0_v7.15.0.sql`** script. If you are upgrading from 7.3.0, you only need to run the **`dbscripts\MySqlUpgrade_v7.3.0_v7.15.0.sql`** script.

Note: The Store must be at 7.3 before running the upgrade script. If the database is not at 7.3, look for upgrade instructions in downloadable bundles on how to bring the database up to 7.3.

5) Upgrade Database continued:
Run the following database script **`MySqlUpgrade_v7.14.2_v7.16.0.sql`**.

6) Reconfigure and Re-apply Customizations:

If any changes to the override files are in place, please **do not** copy the files from the previous version into the directory as they will most likely not work. Instead, apply any custom modifications to the **`Store 7.16.0`** files.

a) Reapply any changes which have been made to the following that are located in the **`\apache-tomcat\lib`** directory:
   - **`users.properties`**
   - **`MPSecurityContext.xml`**
   - **`OzoneConfig.properties.xml`**
   - **`MarketplaceConfig.groovy`**

b) If custom security modules were deployed, they will need to be re-deployed in **`Store 7.16.0`**.

c) Replace all occurrences of **`https://servername:port`** with the name of the server where the Store is running, for example, **`https://www.yourcompany.com:8443`**

Note: See section 3.5.1: Running the Store From Different Ports for new instructions on port changes.

d) If a custom keystore was deployed in a previous build, the keystore for **`Store 7.16.0`** will need to be manually configured in the same manner.

e) If custom changes were applied to the **`marketplace.war`** in version 5, those changes will need to be reapplied to the new **`.war`** file.

The upgrade is now complete.

Appendix B Custom Deployments

B.1 Deploying Tomcat outside the Bundle

The Store's development team uses Tomcat 7.0.54 for all internal testing. The following instructions explain how to deploy Tomcat outside the Store Bundle. To decrease confusion, the Tomcat Instance that is NOT part of the bundle will be referred to as the *new* instance in the following instructions.

1) Unzip the Markteplace Store bundle in a new directory called **`/AML-bundle`**. Some of the files from the bundle will be used in the *new* Tomcat installation.
2) Copy **`/AML-bundle/apache-tomcat/bin/setenv.bat`** into the corresponding folder in the *new* Tomcat installation.
3) Next, modify the *new* Tomcat installation's **`setenv.bat`** file to establish communication with the default security sample. To use a CAS-based security plugin, the **`javax.net.ssl.trustStore`** property must allow the Store and CAS to communicate.
    - To use a self-signed server certificate with a CAS-based security plugin: Add the server certificate to the trust store and set the **`Djavax.net.ssl.trustStore`** option to the file path of the trust store in **`setenv.bat`**.
      Then set the trust store password by adding the **`javax.net.ssl.trustStorePassword`** option to the Java options.

      For example:

      ```
      -Djavax.net.ssl.trustStorePassword=changeit.
      ```

    - To set up a local development or test environment, copy the **`/apache-tomcat/certs/keystore.jks`** to a folder accessible by Tomcat and set **`javax.net.ssl.trustStore`** accordingly.

      For example:

      ```
      -Djavax.net.ssl.trustStore=<new Tomcat directory>\certs\keystore.jks.
      ```

4) Continue working in setenv.bat to increase performance.
   Set the following properties in CATALINA_OPTS to:
   `-XX:PermSize=256m`
   `-XX:MaxPermSize=384m`
   *Note: The memory settings should reflect the memory available in the install environment. For an example of how to set the options see **`\apache-tomcat\bin\setenv.bat`**.*

5) Copy **`/AML-bundle/apache-tomcat/webapps/marketplace.war`** to the *new* Tomcat **`webapps`** directory.

6) To use the sample key store and client certificate in a local development or test environment, copy the **`/AML-bundle/apache-tomcat/certs`** directory to a *new* Tomcat **`certs`** directory (This may need to be created.).

7) Modify the *new* Tomcat **`/apache-tomcat/conf/server.xml`** file with the following configurations. Ensuring that the file contains the "**Connector**" configuration (In the example below, port 8443 has been used.):

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
maxThreads="150" scheme="https" secure="true"
clientAuth="want" sslProtocol="TLS"
keystoreFile="${catalina.home}/certs/keystore.jks" keystorePass="changeit"
truststoreFile="${catalina.home}/certs/keystore.jks"
truststorePass="changeit" />
```

The default security module uses the user's client certificate for authentication if it is available.

To use the default security module (PKI only security) OR a custom security module with a user's client certificate for authorization, set the clientAuth to **`want`** or **`true`**. True requires a client certificate like the PKI only security module. It should only be used if a fallback authentication scheme such as CAS is unnecessary.

Set the key store and trust store parameters to the values that are appropriate for the production environment. The example above is using the default key store shipped with the bundle. It is **only** appropriate for local development and testing environments.

8) Start Tomcat which allows it to unpack the WAR

9) Stop Tomcat

10) Copy the following from the **`apache-tomcat/lib`** directory in the distribution bundle to the *new* Tomcat **`webapps/marketplace/WEB-INF/classes`** directory:

- **`MarketplaceConfig.groovy`**
- **`MPsecurityContext.xml`**
- **`mp-override-log4j.xml`**
- **`OzoneConfig.properties`**
- **`Ehcache.xml`**

11) Add a copy of the **`/AML-bundle/apache-tomcat/lib/resources/images`** directory to the *new* Tomcat. Save it as the **`webapps/marketplace/WEB-INF/classes/resources/images`** directory. If the images directory is not included, the Store will not display the avatar images and default listing images.

12) If the sample security modules are in use, copy **`/AML-bundle/apache-tomcat/lib/users.properties`** to the *new* **`/lib`** directory.

13) Start Tomcat.