Maven Commerce Reference Store

Introduction

This document provides an overview of the Oracle ATG 11.2 Commerce Reference Store (CRS) configured to be built and deployed from maven.

Prerequisites

1. Oracle ATG Commerce 11.2 CRS has already been configured and installed using the instructions in the product manuals. This assumes ATG, WebLogic, Oracle database, and java are all installed and properly configured.
2. Maven is installed. This setup was testing with maven 3.2.5
3. Install ATG libraries to your maven repository. This is a one-time step.
    a. install-atglibs-maven.sh
    b. Note – the environment variable ATG_ROOT must be set, and point to the root of your ATG installation.
    c. Once libraries are added to your maven repository, there is no need to do it again unless they are deleted for some reason, or you change/add libraries.
4. Install a fake hotfix. This is a one-time step.
    a. install-fake-hotfix.sh
    b. This jar contains a single text file, and is used in BuildEar to demonstrate adding hotfixes from your maven repository.
5. Install WebLogic maven plugin to your maven repository if you want to be able to deploy an EAR file. This is a one-time step.
    a. install-wls-maven.sh
    b. Note – the MW_HOME environment variable must be set, and point to your middleware home.
    c. This install file, and the POM included in DeployEar are for WebLogic 12.1.2. If you are using a different version of WebLogic, consult the WebLogic docs for installing and using the WebLogic maven plugin specific to that version.
    d. Once libraries are added to your maven repository, there is no need to do it again unless they are deleted for some reason, or you change/add libraries.
6. Install the atg-maven-plugin if you want to be able to build EAR files. This is a one-time step.
    a. Download from here: https://github.com/oracle/atg-maven-plugin
    b. Instructions are included. All you do is run mvn install.
7. Copy the Commerce Reference Store files needed for building into the Maven build tree. A script is provided that will copy the correct files to the correct location, and change the module name in the appropriate files.
    a. From the directory you extracted this sample to, execute ./copy-CRS-Files.sh
8. This setup assumes you have a working knowledge of maven.

Usage

The setup demonstrates how to build code, build an ear, and deploy an ear using a modified CRS with maven.

This setup does not install the CRS. You must do that following the product manuals.

Building all code

Change to the top level Store directory and execute “mvn install”

The top level parent pom includes all the submodules in the proper order, and will build all of the code.

The build process also places the built code into your ATG_ROOT, under the MavenStore directory, in an ATG module format. Note that MavenStore is the root for this customized version of the OOTB CRS. The OOTB CRS is installed under CommerceReferenceStore.

Depending on if you have used maven on the machine you run this on, and what you have used it for, there may be a large amount of data downloaded the first time you use this. Maven needs to pull all its dependencies to your local repository if they do not already exist.

Building specific modules

The layout of this maven setup groups everything into various submodule layers, which mimics the ANT layout used by the CRS that ships with Commerce, but in a maven format.

The EStore module includes submodules EStore.International, EStore.Versioned, and the EStore.Versioned j2ee app (as well as some others, but we will use those as an example)

If you want to build EStore and all of its submodules, you would cd to Store/EStore and execute mvn install. That will build everything in the proper order.

If you want to build EStore.Versioned and its submodules, cd to Store/EStore/Versioned and execute mvn install.

If you only want to build the EStore.Versioned j2ee app, cd to Store/EStore/Versioned/J2EE-APP and execute mvn install.

Everything follows this module/submodule layout so you can build groups of modules, or individual modules.

Dependency management

All module versions are handled in the top level parent pom, either by explicit definition, or a property value.

While a submodule may include dependencies, they will not call out the version to use. The versions are located in a single pom to allow easier management, and to keep the build cleaner. This will also help prevent ending up with a build using several versions of the same library.

During the setup this document outlines, you added ATG libraries to your maven repository.

This allows code to be built with a common set of libraries, and allows builds to occur with an actual ATG installation in place. You will not be able to build and EAR without the full install, but you can compile and unit test code with something like junit with this setup as a standalone.

You can also import each maven module into eclipse (or another IDE that supports maven) without any special configuration. Eclipse will automatically use the libraries installed to your maven repository since they are defined in the pom files.

In a distributed environment with many developers, this helps setup will also help ensure everyone is using the same library versions. If you install the libs to a centralized maven repository, and modify the poms to use that repository, then all developers and build tools will always pull the exact same libraries. This is also useful to enforce which hotfixes and patches are deployed with your EAR.

A sample fake hotfix is included for demonstration purposes.

ATG dependencies

The version of ATG dependencies that are used is controlled by a single property in the top level pom. This allows you to easily run the same build against multiple versions of Commerce, as long as you have the libraries added to your maven repository.

Taglibs and 3rd party libs

In many build setups, taglibs and 3rd party libraries are checked in to source control. This build setup does not do that.

Taglibs are added to the maven repository, and declared in the top level pom. J2EE-APP modules that need taglibs have entries in their pom that copy the taglib from the maven repo to the appropriate location under in your war file at build time.

The same is true of 3rd party libraries. You will see an example of commons-codec used in this build. It is declared in the top level pom, and entries are in the EStore module pom that pull it from the maven repo at build time.

This setup is especially useful when you want to upgrade a library, or you patch ATG. You will not need to go through your source repository and make sure you upgrade all the taglibs.

Anatomy of the maven module

In the root of a module is the pom.xml.

The directory tree is laid out as follows:

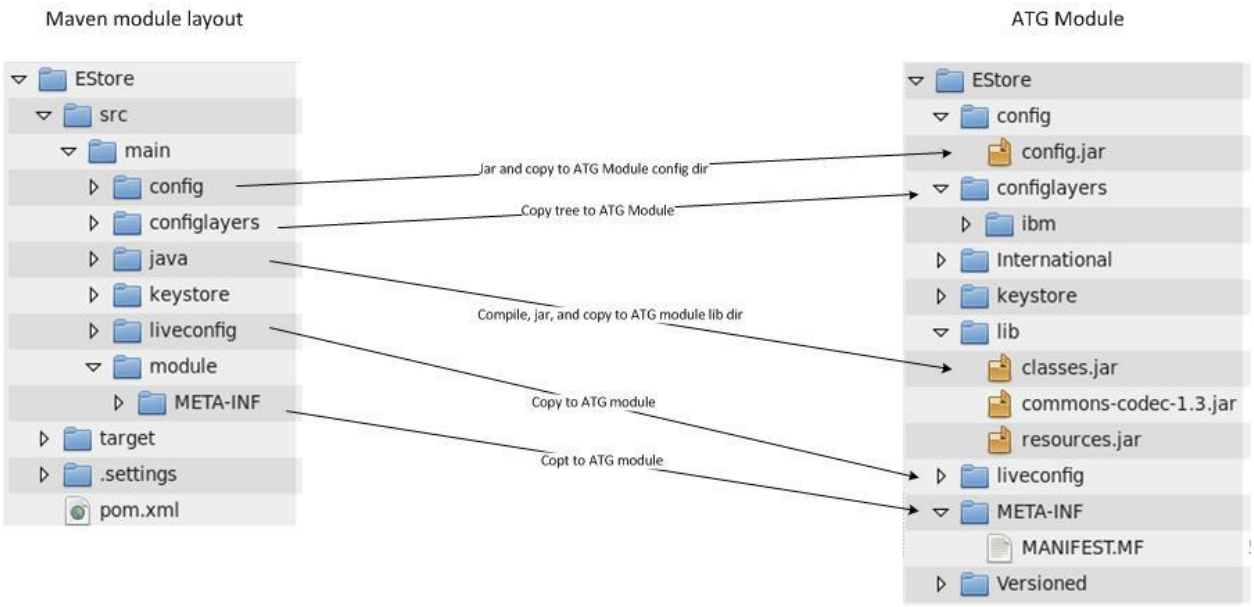
- src/main – this is the root of all other folders
    - module – contains the META-INF/MANFIEST.MF used by ATG modules
    - config - contains the ATG module configs, which will be jar'd and copied to your module's config directory in ATG_ROOT
    - configlayers – contains config layers that runAssembler will pull in if they are specified
    - java – contains source code, which will be compile, then jar'd, then the jar is copied to your ATG_ROOT/module's lib directory
    - liveconfig – contains properties for your ATG liveconfig layer
    - j2ee-apps – These trees contain war files

All directories are optional.

The top level pom defines how to handle each type of directory. If the build finds it, it will process it based on the rules in the parent pom.

There are two special directories, under EStore. They are keystore, and admin. Keystore is used for a cryptography module included in the CRS, and admin contains JHTML admin pages. You can see the special handling in the EStore pom that just copies these to the ATG module tree.

The following is a diagram showing how some of the maven directories map to an ATG module's directory structure.



The location that maven moves all the files to is controlled in each submodule pom, by the module.dir.name property.

The resources.jar being created under the lib directory is not actually used by the CRS. This is created for demonstration purposed to show separating class files from properties files if desired.

Special handling for clean

Because we are doing mapping of a maven layout to an ATG module layout, there is some special handling for executing mvn clean.

Normally, clean only handles files maven knows about under each module. Since we are moving files outside of maven's structure, and mapping things to an ATG module format, we have to add some features to the clean command if we want the ATG module trees cleaned as well.

You will see in each modules pom that specific clean plugins are added, and some declare file exclusions.

This is to handle a case like EStore and EStore.Versioned. When I clean the main EStore module, I do not want to delete the Versioned tree under the ATG module since this is handled by a separate maven build. So in my clean configuration for EStore, I specifically exclude the Versioned tree. If I want to clean the Versioned ATG module, I run clean either inside the EStore/Versioned maven module, OR, in the top level EStore main pom.

File Filtering

Several examples of file filtering are provided.

A timestamp, and build number is added to the MANIFEST.MF indicating when code was built.

A timestamp example is also added when we compile and jar source code. All classes.jar files will have a build.time with a timestamp embedded in the MANIFEST of classes.jar.

You can add other fields here. This is meant to show how you can mark builds with version numbers and timestamps, which is useful if you want to look at a running module and know what version is being run. In larger environments, and those with continuous integration setups, adding timestamps to the build can be very helpful so others can quickly see what is running at any given time. By adding entries to a module MANIFEST.MF, you can see data in the /dyn/admin interface – so there is no need to get on a physical server and look inside an EAR/JAR to know when it was built. Look at the what's running section of /dyn/admin to see the build number data.

You can also add a custom property field, text file, html file, or any other text based file/field this way.

Building and deploying the EAR

Building the EAR file, and deploying the EAR file to Weblogic is not part of the default build.

To build the EAR, all code must already be built with the main Store module.

Once this is built, execute mvn install in the Store/BuildEar directory.

This will create an EAR file using the atg-maven-plugin. Note that the modules line passed to runAssembler uses MavenStore instead of Store. Store is what the OOTB CRS uses. Our version with maven has been renamed to MavenStore.

Once you have an EAR created, you can also deploy it to Weblogic using maven.

Execute mvn install in the Store/DeployEar directory.

This uses the Weblogic maven plugin that ships with Weblogic. You should have installed this plugin as part of the initial setup process these instructions outline.

To deploy the EAR to Weblogic with this build, your AdminServer must be running. This setup also assumes your WebLogic server name is ATGProduction.

Note that if the ATGProduction instance is not running, you will see a message similar to the following:

java.rmi.RemoteException: [Deployer:149145]Unable to contact "ATGProduction". Deployment is deferred until "ATGProduction" becomes available.

This is not an error. It just means the ear file is deployed through the admin instance, and will fire up the next time ATGProduction is started.

The way the pom is setup assumes you are running Weblogic on the same host you are building/deploying code on. You can edit the pom under DeployEar to change where the code is deployed to, the Weblogic admin user/password, as well as other options. Consult the Weblogic documentation for details on other options the Weblogic maven plugin supports.

Before you deploy using this build process the first time, you should go in to your Weblogic admin console and delete the ATGProduction.ear deployment. This should already exist from running the initial CRS installation with CIM.

You don’t want to try to start your instance with the default CRS app, and this version installed. It will not work.

You can automatically build and deploy the EAR file as part of the larger build if desired. There are multiple ways to accomplish this, including shell aliases, maven profiles, and adding to the parent pom. There are additional ways as well. You would need to decide what works best for your environment/setup.

Other documentation

The pom files themselves contain many comments explaining how things are setup.

Sample starter module

There is a second application called ATeam.

This is a skeleton application to show a very simply starting point for a new application/ATG module.

To build this application, change directory to the ATeam folder, and execute “mvn clean install”.

This top level pom will build ATeam/ATeamCommon and ATeam/WebApp, and place the compiled code in an ATG module located at ATG_ROOT/ATeamSample

Note that this is a unique application from the MavenStore module created in the first portion of this document.

This skeleton application has a single droplet, and single main JSP page under the context root /ateam

ATeamCommon

The ATeamCommon folder becomes the ATG module that holds custom code and configurations. This is where the sample droplet is located.

WebApp

The WebApp folder is where JSP pages go.

Included is a base web.xml that will intercept 404, 409 and 500 errors, and redirect the user to the corresponding page under the /global folder in the j2ee-app.

There is also a shared taglib include file. The purpose of includes/taglibs.jspf is to automatically give any JSP page access to tag libraries. This page is injected into every JSP by the include-prelude parameter in web.xml.

Building and Deploying the starter module

This starter module can be deployed either as a standalone application (with the Maven CRS), or with the Maven CRS. This is to demonstrate how you can have multiple top level projects and still deploy them together in the same EAR.

Standalone application EAR

After doing a build of the code (mvn clean install at the top level ATeam directory), change to the BuildStandalone directory and execute mvn install. This will create an ear file at /tmp/ATeam.ear

This EAR contains only the start application.

Starter app and MavenStore together

After doing a build of the code (mvn clean install at the top level ATeam directory), change to the BuildWithMavenCRS directory and execute mvn install. This will create an ear file at /tmp/ATeamWithMavenCRS.ear

This EAR contains the start application and the Maven CRS.

Note that you must have already built the MavenStore module described in the first portion of this document.

If you compare the pom files between BuildStandalone and BuildWithMavenCRS, the key difference is which modules are being passed to runAssembler when building the EAR.

Deploying the standalone application

You can manually install ATeam.ear into WebLogic, or use the pom provided in DeployStandalone (run mvn install in this directory).

Make sure you have removed any existing deployments bound to the ATGProduction instance from WebLogic before deploying this new EAR. Otherwise, you WebLogic instance will attempt to start multiple EAR deployments at the same time, and you will likely have problems.

Deploying the starter application and MavenStore together

You can manually install ATeamWithMavenCRS.ear into WebLogic, or use the pom provided in DeployWithMavenCRS (run mvn install in this directory).

Make sure you have removed any existing deployments bound to the ATGProduction instance from WebLogic before deploying this new EAR. Otherwise, you WebLogic instance will attempt to start multiple EAR deployments at the same time, and you will likely have problems.