Apache Tomcat

CDI 2, JAX-RS and dependent libraries support

Introduction

CDI and JAX-RS are dependencies for many other APIs and libraries. This guide explains how to add support for them in Tomcat using two optional modules that are provided in the Tomcat sources.

CDI 2 support

CDI 2 support is provided by the modules/owb optional module. It packages the Apache OpenWebBeans project and allows adding CDI 2 support to the Tomcat container. The build process of the module uses Apache Maven, and is not available as a binary bundle as it is built using a number of publicly available JARs.

The process to build CDI support is the following.

```
cd $TOMCAT_SRC/modules/owb

mvn clean && mvn package
```

The resulting JAR at target/tomcat-owb-x.y.z.jar (where x.y.z depends on the Apache OpenWebBeans version used during the build) should be processed by the Tomcat migration tool for Jakarta EE, and then be placed into the lib folder of the Tomcat installation.
CDI support can then be enabled for all webapps in the container by adding the following listener in server.xml nested inside the Server element:

```
<Listener
className="org.apache.webbeans.web.tomcat.OpenWebBeansListener"
optional="true" startWithoutBeansXml="false" />
```

The listener will produce a non fatal error if the CDI container loading fails.
CDI support can also be enabled at the individual webapp level by adding the following listener to the webapp context.xml file nested inside
the Server element:

```
<Listener
className="org.apache.webbeans.web.tomcat.OpenWebBeansContextLifecycleListener" />
```

JAX-RS support

JAX-RS support is provided by the modules/cxf optional module. It packages the Apache CXF project and allows adding JAX-RS support to individual webapps. The build process of the module uses Apache Maven, and is not available as a binary bundle as it is built using a number of publicly available JARs. The support depends on CDI 2 support, which should have previously been installed at either the container or webapp level.

The process to build JAX-RS support is the following.

```
cd $TOMCAT_SRC/modules/cxf

mvn clean && mvn package
```

The resulting JAR at target/tomcat-cxf-x.y.z.jar (where x.y.z depends on the Apache CXF version used during the build) should then be placed into the /WEB-INF/lib folder of the desired web application.

If the CDI 2 support is available at the container level, the JAR can also be placed in the Tomcat lib folder, but in that case the CXF Servlet declaration must be individually added in each webapp as needed (it is normally loaded by the web fragment that is present in the JAR). The CXF Servlet class that should be used is org.apache.cxf.cdi.CXFCdiServlet and should be mapped to the desired root path where JAX-RS resources will be available.

The webapp as a whole should be processed by the Tomcat migration tool for Jakarta EE.

Eclipse Microprofile support

ASF artifacts are available that implement Eclipse Microprofile specifications using CDI 2 extensions. Once the CDI 2 and JAX-RS support is installed, they will be usable by individual webapps.

The following implementations are available (reference: org.apache.tomee.microprofile.TomEEMicroProfileListener) as Maven artifacts which must be added to the webapp /WEB-INF/lib folders:

- **Configuration**: Maven artifact: org.apache.geronimo.config:geronimo-config CDI extension
  class: org.apache.geronimo.config.cdi.ConfigExtension
- **Fault Tolerance**: Maven
  artifact: org.apache.geronimo.safeguard:safeguard-parent CDI extension
  class: org.apache.safeguard.impl.cdi.SafeguardExtension
- **Health**: Maven artifact: org.apache.geronimo:geronimo-health CDI extension
  class: org.apache.geronimo.microprofile.impl.health.cdi.GeronimoHealthExtension
- **Metrics**: Maven artifact: org.apache.geronimo:geronimo-metrics CDI extension
  class: org.apache.geronimo.microprofile.metrics.cdi.MetricsExtension
- **OpenTracing**: Maven artifact: org.apache.geronimo:geronimo-opentracing CDI extension
  class: org.apache.geronimo.microprofile.opentracing.microprofile.cdi.OpenTracingExtension
- **OpenAPI**: Maven artifact: org.apache.geronimo:geronimo-openapi CDI extension
  class: org.apache.geronimo.microprofile.openapi.cdi.GeronimoOpenAPIExtension
- **Rest client**: Maven artifact: org.apache.cxf:cxf-rt-rs-mp-client CDI extension
  class: org.apache.cxf.microprofile.client.cdi.RestClientExtension
- **JSON Web Tokens**: Note: Fore reference only, unusable outside Apache TomEE; Maven artifact: org.apache.tomee:mp-jwt CDI extension
  class: org.apache.tomee.microprofile.jwt.cdi.MPJWTCDIExtension

Ahead of Time compilation support

Introduction

Tomcat supports using the GraalVM/Mandrel Native Image tool to produce a native binary including the container. This documentation page describes the build process of such an image.

Setup

The native image tool is much easier to use with single JARs, as a result the process will use the Maven shade plugin JAR packaging. The idea is to produce a single JAR that contains all necessary classes from Tomcat, the webapps and all additional dependencies. Although Tomcat has received compatibility fixes to support native images, other libraries may not be compatible and may require replacement code (the GraalVM documentation has more details about this).

Download and install GraalVM or Mandrel.

Download the Tomcat Stuffed module from https://github.com/apache/tomcat/tree/main/modules/stuffed. For convinience, an env property can be set:

export TOMCAT_STUFFED=/absolute...path...to/stuffed

The build process now requires both Apache Ant and Maven.

Packaging and Building

Inside the $TOMCAT_STUFFED folder, the directory structure is the same as for regular Tomcat. The main configuration files are placed in the conf folder, and if using the default server.xml the webapps are placed in the webapps folder.

All the webapp classes need to be made available to the Maven shade plugin as well as the compiler during the JSP precompilation step. Any JARs that are present in /WEB-INF/lib need to be made available as Maven dependencies. The webapp-jspc.ant.xml script will copy classes from the /WEB-INF/classes folder of the webapp to the target/classes path that Maven uses as the compilation target, but if any of the JSP sources use them, then they need to be packaged as JARs instead.

The first step is to build the shaded Tomcat JAR with all dependencies. Any JSP in the webapp must all be precompiled and packaged (assuming that the webapps contains a $WEBAPPNAME webapp):

```
cd $TOMCAT_STUFFED

mvn package

ant -Dwebapp.name=$WEBAPPNAME -f webapp-jspc.ant.xml
```

Dependencies for the webapp should now be added to the main $TOMCAT_STUFFED/pom.xml, following by building the shaded JAR:

```
mvn package
```

As it is best to avoid using reflection whenever possible with Ahead of Time compilation, it can be a good idea to generate and compile Tomcat Embedded code out of the main server.xml configuration as well as the context.xml files used to configure the contexts.

```
$JAVA_HOME/bin/java\

        -Dcatalina.base=. -
Djava.util.logging.config.file=conf/logging.properties\

        -jar target/tomcat-stuffed-1.0.jar --catalina -generateCode
src/main/java
```

Then stop Tomcat and use the following command to include the generated embedded code:

```
mvn package
```

The rest of the process described here will assume this step was done and the --catalina -useGeneratedCode arguments are added to the command lines. If this was not the case, they should be removed.

Native image configuration

Native images do not support any form of dynamic classloading or reflection unless it is defined explicitly in descriptors. Generating them uses a tracing agent from the GraalVM, and needs additional manual configuration in some cases.

Run Tomcat using the GraalVM substrate VM and its trace agent:

```
$JAVA_HOME/bin/java\

        -agentlib:native-image-agent=config-output-
dir=$TOMCAT_STUFFED/target/\

        -Dorg.graalvm.nativeimage.imagecode=agent\

        -Dcatalina.base=. -
Djava.util.logging.config.file=conf/logging.properties\

        -jar target/tomcat-stuffed-1.0.jar --catalina -useGeneratedCode
```

Now all paths from the webapp that lead to dynamic classloading (ex: Servlet access, websockets, etc) need to be accessed using a script that will exercise the webapp. Servlets may be loaded on startup instead of needing an actual access. Listeners may also be used to load additional classes on startup. When that is done, Tomcat can be stopped.

The descriptors have now been generated in the agent output directory. At this point, further configuration must be made to add items that are not traced, including: base interfaces, resource bundles, BeanInfo based reflection, etc. Please refer to the Graal documentation for more information on this process.

Even though all classes that are used have to be complied AOT into the native image, webapps must still be left unchanged, and continue including all needed classes and JARs in the WEB-INF folder. Although these classes will not actually be run or loaded, access to them is required.

Building the native image

If everything has been done properly, the native image can now be built using the native-image tool.

```
$JAVA_HOME/bin/native-image --report-unsupported-elements-at-runtime\

        --enable-http --enable-https --enable-url-protocols=http,https,jar,jrt\
        --initialize-at-build-
time=org.eclipse.jdt,org.apache.el.parser.SimpleNode,jakarta.servlet.jsp.JspFac
tory,org.apache.jasper.servlet.JasperInitializer,org.apache.jasper.runtime.JspFac
toryImpl\

        -H:+UnlockExperimentalVMOptions\

        -H:+JNI -H:+ReportExceptionStackTraces\

        -H:ConfigurationFileDirectories=$TOMCAT_STUFFED/target/\

        -H:ReflectionConfigurationFiles=$TOMCAT_STUFFED/tomcat-
reflection.json\

        -H:ResourceConfigurationFiles=$TOMCAT_STUFFED/tomcat-
resource.json\

        -H:JNIConfigurationFiles=$TOMCAT_STUFFED/tomcat-jni.json\

        -jar $TOMCAT_STUFFED/target/tomcat-stuffed-1.0.jar
```

The additional --static parameter enables static linking of glibc, zlib and libstd++ in the generated binary.

Running the native image is then:

```
./tomcat-stuffed-1.0 -Dcatalina.base=. -
Djava.util.logging.config.file=conf/logging.properties --catalina -
useGeneratedCode
```

Compatibility

Servlets, JSPs, EL, websockets, the Tomcat container, tomcat-native, HTTP/2 are all supported out of the box in a native image.

At the time of writing this documentation, JULI is not supported as the log manager configuration property is not supported by Graal, in addition to some static initializer problems, and the regular java.util.logging loggers and implementation should be used instead.

If using the default server.xml file, some Server listeners have to be removed from the configuration as they are not compatible with native images, such as a

JMX listener (JMX is unsupported) and leak prevention listeners (use of internal code that does not exist in Graal).

Missing items for better Tomcat functionality:

- java.util.logging LogManager: Configuration through a system property is not implemented, so standard java.util.logging must be used instead of JULI
- Static linking configuration: tomcat-native cannot be statically linked