# Table of Contents

1. Installation  ................................................................. 1  
   1.1. Requirements  ...................................................... 1  
   1.2. Get the APEX Source Code  ...................................... 2  
   1.3. Build APEX  ....................................................... 2  
   1.4. Install APEX  ..................................................... 4  
   1.5. Build from Source  ............................................... 6  
   1.6. Installation Layout  .............................................. 8  
   1.7. System Configuration  .......................................... 9  
   1.8. Verify the APEX Installation  .................................. 11  
   1.9. Installing WAR Applications  .................................. 15  
   1.10. Running APEX in Docker  .................................... 15  
2. APEX Configurations Explained  ..................................... 16  
   2.1. Introduction to APEX Configuration  ............................ 16  
   2.2. General Configuration Format  .................................. 17  
   2.3. Engine Service Parameters  .................................... 18  
   2.4. Input and Output Interfaces  ................................... 19  
   2.5. Executors  ....................................................... 21  
   2.6. $APEX_HOME/bin/apexEngine.sh [args]  ....................... 48  
   2.7. $APEX_HOME/bin/apexApps.sh engine [args]  ................. 48  
   2.8. $APEX_HOME/bin/apexCLIEditor.sh.sh [args]  ............... 49  
   2.9. $APEX_HOME/bin/apexApps.sh cli-editor [args]  .......... 49  
   2.10. $APEX_HOME/bin/apexRESTEditor.sh.sh [args] ............ 51  
   2.11. $APEX_HOME/bin/apexApps.sh rest-editor [args] ........... 51  
   2.12. $APEX_HOME/bin/apexApps.sh [args]  ....................... 55  
   2.13. $APEX_HOME/bin/apexApps.sh tpl-event-json [args] ....... 57  
   2.14. $APEX_HOME/bin/apexApps.sh model-2-cli [args] ......... 59  
   2.15. $APEX_HOME/bin/apexApps.sh ws-echo [args] ............... 60  
   2.16. $APEX_HOME/bin/apexApps.sh ws-console [args] ........... 60  
   2.17. $APEX_HOME/bin/apexApps.sh ws-echo [args] ............... 101  
   2.18. $APEX_HOME/bin/apexApps.sh ws-echo -p 42452 <1> ....... 101  
   2.19. $APEX_HOME/bin/apexApps.sh ws-console [args] ........... 102  
   2.20. $APEX_HOME/bin/apexApps.sh ws-console -p 42450 <1> .... 103
1. Installation

1.1. Requirements

APEX is 100% written in Java and runs on any platform that supports a JVM, e.g. Windows, Unix, Cygwin. Some APEX applications (such as the monitoring application) come as web archives, they do require a war-capable web server installed.

1.1.1. Installation Requirements

- Downloaded distribution: JAVA runtime environment (JRE, Java 8 or later, APEX is tested with the Oracle Java)
- Building from source: JAVA development kit (JDK, Java 8 or later, APEX is tested with the Oracle Java)
- A web archive capable webserver, for instance for the monitoring application
  - for instance Apache Tomcat
- Sufficient rights to install APEX on the system
- Installation tools depending on the installation method used:
  - ZIP to extract from a ZIP distribution
    - Windows for instance 7Zip
  - TAR and GZ to extract from that TAR.GZ distribution
    - Windows for instance 7Zip
  - RPM to install from the RPM distribution
    - Install: `sudo apt-get install rpm`
  - DEB to install from the DEB distribution
    - Install: `sudo apt-get install dpkg`

1.1.2. Feature Requirements

APEX supports a number of features that require extra software being installed.

- Apache Kafka to connect APEX to a Kafka message bus
- Hazelcast to use distributed hash maps for context
- Infinispan for distributed context and persistence
- Docker to run APEX inside a Docker container

1.1.3. Build (Install from Source) Requirements

Installation from source requires a few development tools

- GIT to retrieve the source code
- Java SDK, Java version 8 or later
• Apache Maven 3 (the APEX build environment)

1.2. Get the APEX Source Code

The first APEX source code was hosted on Github in January 2018. By the end of 2018, APEX was added as a project in the ONAP Policy Framework, released later in the ONAP Casablanca release.

The APEX source code is hosted in ONAP as project APEX. The current stable version is in the master branch. Simply clone the master branch from ONAP using HTTPS.

```bash
    git clone https://gerrit.onap.org/r/policy/apex-pdp
```

1.3. Build APEX

The examples in this document assume that the APEX source repositories are cloned to:

• Unix, Cygwin: `/usr/local/src/apex-pdp`
• Windows: `C:\dev\apex-pdp`
• Cygwin: `/cygdrive/c/dev/apex-pdp`

A Build requires ONAP Nexus
APEX has a dependency to ONAP parent projects. You might need to adjust your Maven M2 settings. The most current settings can be found in the ONAP parent repo: Settings.

A Build needs Space
Building APEX requires approximately 2-3 GB of hard disc space, 1 GB for the actual build with full distribution and 1-2 GB for the downloaded dependencies.

A Build requires Internet (for first build)
During the build, several (a lot) of Maven dependencies will be downloaded and stored in the configured local Maven repository. The first standard build (and any first specific build) requires Internet access to download those dependencies.

Building RPM distributions
RPM images are only build if the `rpm` package is installed (Unix). To install `rpm` run `sudo apt-get install rpm`, then build APEX.

Use Maven to for a standard build without any tests.

<table>
<thead>
<tr>
<th>Unix, Cygwin</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td># cd /usr/local/src/apex-pdp</td>
<td>&gt;c:</td>
</tr>
<tr>
<td># mvn clean install -DskipTests</td>
<td>&gt;cd \dev\apex-pdp</td>
</tr>
<tr>
<td></td>
<td>&gt;mvn clean install -DskipTests</td>
</tr>
</tbody>
</table>

The build takes 2-3 minutes on a standard development laptop. It should run through without errors, but with a lot of messages from the build process.

When Maven is finished with the build, the final screen should look similar to this (omitting some success lines):
The build will have created all artifacts required for an APEX installation. The following example show how to change to the target directory and how it should look like.

**Unix, Cygwin**

```bash
# cd packages/apex-pdp-package-full/target
# ls -l
```

- `-rwxrwx---- 1 esvevan Domain Users 772 Sep  3 11:55 apex-pdp-package-full_2.0.0~SNAPSHOT_all.changes`
- `-rwxrwx---- 1 esvevan Domain Users 146328882 Sep  3 11:55 apex-pdp-package-full-2.0.0-SNAPSHOT.deb`
- `-rwxrwx---- 1 esvevan Domain Users 15633 Sep  3 11:54 apex-pdp-package-full-2.0.0-SNAPSHOT.jar`
- `-rwxrwx---- 1 esvevan Domain Users 146296819 Sep  3 11:55 apex-pdp-package-full-2.0.0-SNAPSHOT-tarball.tar.gz`
- `drwxrwx---+ 1 esvevan Domain Users 0 Sep  3 11:54 archive-tmp/`
- `-rwxrwx---- 1 esvevan Domain Users 89 Sep  3 11:54 checkstyle-cachefile`
- `-rwxrwx---- 1 esvevan Domain Users 10621 Sep  3 11:54 checkstyle-checker.xml`
- `-rwxrwx---- 1 esvevan Domain Users 584 Sep  3 11:54 checkstyle-header.txt`
- `-rwxrwx---- 1 esvevan Domain Users 86 Sep  3 11:54 checkstyle-result.xml`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:54 classes/`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:54 dependency-maven-plugin-markers/`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:54 etc/`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:54 examples/`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:55 install_hierarchy/`
- `drwxrwx---- 1 esvevan Domain Users 0 Sep  3 11:54 maven-archiver/`

**Windows**

```bash
>cd packages\apex-pdp-package-full\target
>dir
```
1.4. Install APEX

APEX can be installed in different ways:

- Unix: automatically using `rpm` or `dpkg` from `.rpm` or `.deb` archive
- Windows, Unix, Cygwin: manually from a `.tar.gz` archive
- Windows, Unix, Cygwin: build from source using Maven, then install manually

1.4.1. Install with RPM and DPKG

The install distributions of APEX automatically install the system. The installation directory is `/opt/app/policy/apex-pdp`. Log files are located in `/var/log/onap/policy/apex-pdp`. The latest APEX version will be available as `/opt/app/policy/apex-pdp/apex-pdp`.

For the installation, a new user `apexuser` and a new group `apexuser` will be created. This user owns the installation directories and the log file location. The user is also used by the standard APEX start scripts to run APEX with this user's permissions.

RPM Installation

```
# sudo rpm -i apex-pdp-package-full-2.1.0-SNAPSHOT.rpm
******************* preinst *******************
arguments 1
************************************************
creating group apexuser ...
creating user apexuser ...
************* postinst *************
arguments 1
************************************************
```
DPKG Installation

```bash
# sudo dpkg -i apex-pdp-package-full-2.1.0-SNAPSHOT.deb
Selecting previously unselected package apex-uservice.
(Reading database ... 288458 files and directories currently installed.)
Preparing to unpack apex-pdp-package-full-2.1.0-SNAPSHOT.deb ...
********************preinst********************
arguments install
********************postinst********************
```

Once the installation is finished, APEX is fully installed and ready to run.

### 1.4.2. Install Manually from Archive (Unix, Cygwin)

Download a [tar.gz](#) archive. Create a directory where APEX should be installed. Extract the tar archive. The following example shows how to install APEX in `/opt/apex` and create a link to `/opt/apex/apex` for the most recent installation.

```bash
# cd /opt
# mkdir apex
# cd apex
# mkdir apex-full-2.1.0-SNAPSHOT
# tar xvfz ~/Downloads/apex-pdp-package-full-2.1.0-SNAPSHOT.tar.gz -C apex-full-2.1.0-SNAPSHOT
# ln -s apex apex-pdp-package-full-2.1.0-SNAPSHOT
```

### 1.4.3. Install Manually from Archive (Windows, 7Zip, GUI)

Download a [tar.gz](#) archive and copy the file into the install folder (in this example `C:\apex`). Assuming you are using 7Zip, right click on the file and extract the tar archive. Note: the screenshots might show an older version than you have.
The right-click on the new created TAR file and extract the actual APEX distribution.

Inside the new APEX folder you see the main directories: `bin`, `etc`, `examples`, `lib`, and `war`.

Once extracted, please rename the created folder to `apex-full-2.1.0-SNAPSHOT`. This will keep the directory name in line with the rest of this documentation.

### 1.4.4. Install Manually from Archive (Windows, 7Zip, CMD)

Download a `.tar.gz` archive and copy the file into the install folder (in this example `C:\apex`). Start `cmd`, for instance typing `Windows+R` and then `cmd` in the dialog. Assuming `7Zip` is installed in the standard folder, simply run the following commands (for APEX version 2.1.0-SNAPSHOT full distribution):

```bash
>c:
>cd \apex
>"\Program Files\7-Zip\7z.exe" x apex-pdp-package-full-2.1.0-SNAPSHOT.tar.gz -so | "\Program Files\7-Zip\7z.exe" x -aoa -si -ttar -o"apex-full-2.1.0-SNAPSHOT"
```

APEX is now installed in the folder `C:\apex\apex-full-2.1.0-SNAPSHOT`.

### 1.5. Build from Source

#### 1.5.1. Build and Install Manually (Unix, Windows, Cygwin)

Clone the APEX GIT repositories into a directory. Go to that directory. Use Maven to build APEX (all details on building APEX from source can be found in APEX HowTo: Build). Install from the created artifacts (`rpm`, `deb`, `tar.gz`, or copying manually).

**Building RPM distributions**

RPM images are only build if the `rpm` package is installed (Unix). To install `rpm` run `sudo apt-get install rpm`, then build APEX.

The following example shows how to build the APEX system, without tests (`-DskipTests`) to save some time. It assumes that the APX GIT repositories are cloned to:

- Unix, Cygwin: `/usr/local/src/apex`
The build takes about 2 minutes without test and about 4-5 minutes with tests on a standard development laptop. It should run through without errors, but with a lot of messages from the build process. If build with tests (i.e. without -DskipTests), there will be error messages and stack trace prints from some tests. This is normal, as long as the build finishes successful.

When Maven is finished with the build, the final screen should look similar to this (omitting some success lines):

```
[INFO] tools .............................................. SUCCESS [  0.248 s]
[INFO] tools-common ..................................... SUCCESS [  0.784 s]
[INFO] simple-wsclient .................................. SUCCESS [  3.303 s]
[INFO] model-generator ................................... SUCCESS [  0.644 s]
[INFO] packages .......................................... SUCCESS [  0.336 s]
[INFO] apex-pdp-package-full ............................. SUCCESS [ 10.307 s]
```

The build will have created all artifacts required for an APEX installation. The following example show how to change to the target directory and how it should look like.

Unix, Cygwin

```bash
# cd packages/apex-pdp-package-full/target
# ls -l
```

```
-rwxrwx---+ 1 esvevan Domain Users       772 Sep  3 11:55 apex-pdp-package-
full_2.0.0-SNAPSHOT_all.changes*
-rwxrwx---+ 1 esvevan Domain Users 146328082 Sep  3 11:55 apex-pdp-package-full-2.0.0-SNAPSHOT.deb*
-rwxrwx---+ 1 esvevan Domain Users 15633 Sep  3 11:54 apex-pdp-package-full-2.0.0-SNAPSHOT.jar*
-rwxrwx---+ 1 esvevan Domain Users 146296819 Sep  3 11:55 apex-pdp-package-full-2.0.0-SNAPSHOT-
tarball.tar.gz*
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 archive-tmp/
-rwxrwx---+ 1 esvevan Domain Users         89 Sep  3 11:54 checkstyle-cachefile*
-rwxrwx---+ 1 esvevan Domain Users 18621 Sep  3 11:54 checkstyle-checker.xml*
-rwxrwx---+ 1 esvevan Domain Users         584 Sep  3 11:54 checkstyle-header.txt*
-rwxrwx---+ 1 esvevan Domain Users         86 Sep  3 11:54 checkstyle-result.xml*
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 classes/
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 dependency-maven-plugin-markers/
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 etc/
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 examples/
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:55 install_hierarchy/
drwxrwx---+ 1 esvevan Domain Users         0 Sep  3 11:54 maven-archiver/
```
Now, take the `.deb` or the `.tar.gz` file and install APEX. Alternatively, copy the content of the folder `install_hierarchy` to your APEX directory.

## 1.6. Installation Layout

A full installation of APEX comes with the following layout.

```
$APEX_HOME
  └───bin ①
  └───etc ②
      ├───editor
      │   └───hazelcast
      │       └───infinispan
      │           └───META-INF
      └───examples ③
          ├───config ④
          │   └───docker
          ├───events ④
          ├───html ④
          └───models ④
          └───scripts ④
  └───lib ⑩
      └───applications ⑪
  └───war ⑫
```

① binaries, mainly scripts (bash and bat) to start the APEX engine and applications
② configuration files, such as logback (logging) and third party library configurations
③ example policy models to get started
④ configurations for the examples (with sub directories for individual examples)
Docker files and additional Docker instructions for the examples
example events for the examples (with sub directories for individual examples)
HTML files for some examples, e.g. the Decisionmaker example
the policy models, generated for each example (with sub directories for individual examples)
additional scripts for the examples (with sub directories for individual examples)
library folder with all Java JAR files
applications, also known as jar with dependencies (or fat jars), individually deployable
WAR files for web applications

1.7. System Configuration

Once APEX is installed, a few configurations need to be done:

- Create an APEX user and an APEX group (optional, if not installed using RPM and DPKG)
- Create environment settings for APEX_HOME and APEX_USER, required by the start scripts
- Change settings of the logging framework (optional)
- Create directories for logging, required (execution might fail if directories do not exist or cannot be created)

1.7.1. APEX User and Group

On smaller installations and test systems, APEX can run as any user or group.

However, if APEX is installed in production, we strongly recommend you set up a dedicated user for running APEX. This will isolate the execution of APEX to that user. We recommend you use the userid apexuser but you may use any user you choose.

The following example, for UNIX, creates a group called apexuser, an APEX user called apexuser, adds the group to the user, and changes ownership of the APEX installation to the user. Substitute <apex-dir> with the directory where APEX is installed.

```
# sudo groupadd apexuser
# sudo useradd -g apexuser apexuser
# sudo chwon -R apexuser:apexuser <apex-dir>
```

For other operating systems please consult your manual or system administrator.

1.7.2. Environment Settings: APEX_HOME and APEX_USER

The provided start scripts for APEX require two environment variables being set:

- APEX_USER with the user under whos name and permission APEX should be started (Unix only)
- APEX_HOME with the directory where APEX is installed (Unix, Windows, Cygwin)

The first row in the following table shows how to set these environment variables temporary (assuming the user is apexuser). The second row shows how to verify the settings. The last row explains how to set those variables permanently.
Making Environment Settings Permanent (Unix, Cygwin)

For a per-user setting, edit the a user's `bash` or `tcsh` settings in `~/.bashrc` or `~/.tcshrc`. For system-wide settings, edit `/etc/profiles` (requires permissions).

Making Environment Settings Permanent (Windows)

On Windows 7 do

- Click on the **Start** Menu
- Right click on **Computer**
- Select **Properties**

On Windows 8/10 do

- Click on the **Start** Menu
- Select **System**

Then do the following

- Select **Advanced System Settings**
- On the **Advanced** tab, click the **Environment Variables** button
- Edit an existing variable, or create a new System variable: 'Variable name'="APEX_HOME", 'Variable value'="C:\apex\apex-full-2.1.0-SNAPSHOT"

For the settings to take effect, an application needs to be restarted (e.g. any open cmd window).

1.7.3. Edit the APEX Logging Settings

Configure the APEX logging settings to your requirements, for instance:

- change the directory where logs are written to, or
- change the log levels

Edit the file `$APEX_HOME/etc/logback.xml` for any required changes. To change the log directory change the line

```xml
<property name="VAR_LOG" value="/var/log/onap/policy/apex-pdp/" />
```
<property name="VAR_LOG" value="/PATH/TO/LOG/DIRECTORY/" />

On Windows, it is recommended to change the log directory to:

<property name="VAR_LOG" value="C:/apex/apex-full-2.1.0-SNAPSHOT/logs" />

Note: Be careful about when to use \\ vs. / as the path separator!

1.7.4. Create Directories for Logging

Make sure that the log directory exists. This is important when APEX was installed manually or when the log directory was changed in the settings (see above).

<table>
<thead>
<tr>
<th>Unix, Cygwin</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mkdir -p /var/log/onap/policy/apex-pdp</code>&lt;br&gt;<code>chown -R apexuser:apexuser /var/log/onap/policy/apex-pdp</code></td>
<td><code>&gt;mkdir C:\apex\apex-full-2.1.0-SNAPSHOT\logs</code></td>
</tr>
</tbody>
</table>

1.8. Verify the APEX Installation

When APEX is installed and all settings are realized, the installation can be verified.

1.8.1. Verify Installation - run Engine

A simple verification of an APEX installation can be done by simply starting the APEX engine without any configuration. On Unix (or Cygwin) start the engine using `$APEX_HOME/bin/apexEngine.sh`. On Windows start the engine using `$APEX_HOME\bin\apexEngine.bat`. The engine will fail to fully start. However, if the output looks similar to the following line, the APEX installation is realized.

```
Starting Apex service with parameters [] . . .
start of Apex service failed: Apex configuration file was not specified as an argument
org.onap.policy.apex.model.basicmodel.concepts.ApexException: Apex configuration file was not specified as an argument
    at org.onap.policy.apex.service.engine.main.ApexMain.<init>(ApexMain.java:68)
at org.onap.policy.apex.service.engine.main.ApexMain.<init>(ApexMain.java:68)
at org.onap.policy.apex.service.engine.main.ApexMain.main(ApexMain.java:165)
usage: org.onap.policy.apex.service.engine.main.ApexMain [options...]
options
-c,--config-file <CONFIG_FILE> the full path to the configuration file to use, the configuration file must be a Json file
-h,--help outputs the usage of this command
-m,--model-file <MODEL_FILE> the full path to the model file to use, if set it overrides the model file set in the configuration file
-v,--version outputs the version of Apex
```
1.8.2. Verify Installation - run an Example

A full APEX installation comes with several examples. Here, we can fully verify the installation by running one of the examples.

We use the example called *SampleDomain* and configure the engine to use standard in and standard out for events. Run the engine with the provided configuration. Note: Cygwin executes scripts as Unix scripts but runs Java as a Windows application, thus the configuration file must be given as a Windows path.

```
# $APEX_HOME/bin/apexEngine.sh -c $APEX_HOME/examples/config/SampleDomain/Stdin2StdoutJsonEventJava.json
①
# $APEX_HOME/bin/apexEngine.sh -c C:/apex/apex-full-2.1.0-SNAPSHOT/examples/config/SampleDomain/Stdin2StdoutJsonEventJava.json ②
> \%APEX_HOME\%\bin\apexEngine.bat -c \%APEX_HOME\%\examples\config\SampleDomain\Stdin2StdoutJsonEventJava.json
③
```

① UNIX
② Cygwin
③ Windows

The engine should start successfully. Assuming the logging levels are not change (default level is *info*), the output should look similar to this (last few lines):

```
2018-09-05 15:16:43,963 Apex [main] INFO o.o.p.a.s.e.e.EngDepMessagingService - engine<--deployment messaging started
2018-09-05 15:16:45,120 Apex [main] INFO o.o.p.a.s.e.r.impl.EngineServiceImpl - Added the action listener to the engine
Started Apex service
```

Important are the last two line, stating that APEX has added the final action listener to the engine and that the engine is started.

The engine is configured to read events from standard input and write produced events to standard output. The policy model is a very simple policy.

The following table shows an input event in the left column and an output event in the right column. Past the input
Terminate APEX by simply using **CTRL+C** in the console.

### 1.8.3. Verify a Full Installation - REST Editor

APEX has a REST application for viewing policy models. The application can also be used to create new policy models close to the engine native policy language. Start the REST editor as follows.

```bash
# $APEX_HOME/bin/apexApps.sh rest-editor
```

The script will start a simple web server (**Grizzly**) and deploy a **war** web archive in it. Once the editor is started, it will be available on **localhost:18989**. The last few line of the messages should be:


APEX Editor REST endpoint (ApexEditorMain: Config=[ApexEditorParameters: URI=http://localhost:18989/apexservices/, TTL=-1sec], State=RUNNING) started at http://localhost:18989/apexservices/

Now open a browser (Firefox, Chrome, Opera, Internet Explorer) and use the URL **http://localhost:18989/**. This will connect the browser to the started REST editor. The start screen should be as follows.
Now load a policy model by clicking the menu File and then Open. In the opened dialog, go to the directory where APEX is installed, then examples, models, SampleDomain, and there select the file SamplePolicyModelJAVA.json. This will load the policy model used to verify the policy engine (see above). Once loaded, the screen should look as follows.

Now you can use the REST editor. To finish this verification, simply terminate your browser (or the tab), and then use **CTRL+C** in the console where you started the REST editor.
1.9. Installing WAR Applications

APEX comes with a set of WAR files. These are complete applications that can be installed and run in an application server. All of these applications are realized as servlets. You can find the WAR applications in $APEX_HOME/war (UNIX, Cygwin) or %APEX_HOME%\war (Windows).

Installing and using the WAR applications requires a web server that can execute war web archives. We recommend to use Apache Tomcat, however other web servers can be used as well.

Install Apache Tomcat including the Manager App, see V9.0 Docs for details. Start the Tomcat service, or make sure that Tomcat is running.

There are multiple ways to install the APEX WAR applications:

- copy the .war file into the Tomcat webapps folder
- use the Tomcat Manager App to deploy via the web interface
- deploy using a REST call to Tomcat

For details on how to install war files please consult the Tomcat Documentation or the Manager App HOW-TO. Once you installed an APEX WAR application (and wait for sufficient time for Tomcat to finalize the installation), open the Manager App in Tomcat. You should see the APEX WAR application being installed and running.

In case of errors, examine the log files in the Tomcat log directory. In a conventional install, those log files are in the logs directory where Tomcat is installed.

The current APEX version provides the following WAR applications:

- client-deployment-2.1.0-SNAPSHOT.war - a client to deploy new policy models to a running engine
- client-editor-2.1.0-SNAPSHOT.war - the standard policy REST editor GUI
- client-monitoring-2.1.0-SNAPSHOT.war - a client for monitoring a running APEX engine
- client-full-2.1.0-SNAPSHOT.war - a full client with a one-stop-access to deployment, monitoring, and REST editor
- examples-servlet-2.1.0-SNAPSHOT.war - an example APEX servlet

1.10. Running APEX in Docker

Since APEX is in ONAP, we provide a full virtualization environment for the engine.

1.10.1. Run in ONAP

Running APEX from the ONAP docker repository only requires 2 commands:

1. Log into the ONAP docker repo

   `docker login -u docker -p docker nexus3.onap.org:10003`

2. Run the APEX docker image

   `docker run -it --rm nexus3.onap.org:10003/onap/policy-apex-pdp:latest`
1.10.2. Build a Docker Image

Alternatively, one can use the Dockerfile defined in the Docker package to build an image.

APEX Dockerfile

```
# Docker file to build an image that runs APEX on Java 8 in Ubuntu
#
FROM ubuntu:16.04

RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get install -y zip unzip curl wget iproute2 iputils-ping vim && \
    apt-get install -y software-properties-common && \
    add-apt-repository ppa:openjdk-r/ppa -y && \
    apt-get update && \
    apt-get install -y openjdk-8-jdk

# Create apex user and group
RUN groupadd apexuser
RUN useradd --create-home -g apexuser apexuser

# Add Apex-specific directories and set ownership as the Apex admin user
RUN mkdir -p /opt/app/policy/apex-pdp
RUN mkdir -p /var/log/onap/policy/apex-pdp
RUN chown -R apexuser:apexuser /var/log/onap/policy/apex-pdp

# Unpack the tarball
RUN mkdir /packages
COPY apex-pdp-package-full.tar.gz /packages
RUN tar xvfz /packages/apex-pdp-package-full.tar.gz --directory /opt/app/policy/apex-pdp
RUN rm /packages/apex-pdp-package-full.tar.gz

# Ensure everything has the correct permissions
RUN find /opt/app -type d -perm 755
RUN find /opt/app -type f -perm 644
RUN chmod a+x /opt/app/policy/apex-pdp/bin/*

# Copy examples to Apex user area
RUN cp -pr /opt/app/policy/apex-pdp/examples /home/apexuser

RUN apt-get clean
RUN chown -R apexuser:apexuser /home/apexuser/

USER apexuser
ENV PATH /opt/app/policy/apex-pdp/bin:$PATH
WORKDIR /home/apexuser
```

2. APEX Configurations Explained

2.1. Introduction to APEX Configuration

An APEX engine can be configured to use various combinations of event input handlers, event output handlers, event protocols, context handlers, and logic executors. The system is built using a plugin architecture. Each configuration option is realized by a plugin, which can be loaded and configured when the engine is started. New plugins can be added to the system at any time, though to benefit from a new plugin an engine will need to be restarted.
The APEX distribution already comes with a number of plugins. The figure above shows the provided plugins. Any combination of input, output, event protocol, context handlers, and executors is possible.

### 2.2. General Configuration Format

The APEX configuration file is a JSON file containing a few main blocks for different parts of the configuration. Each block then holds the configuration details. The following code shows the main blocks:
2.3. Engine Service Parameters

The configuration provides a number of parameters to configure the engine. An example configuration with explanations of all options is shown below.
"engineServiceParameters": {
    "name": "AADMApexEngine", ①
    "version": "0.0.1", ②
    "id": 45, ③
    "instanceCount": 4, ④
    "deploymentPort": 12345, ⑤
    "policyModelFileName": "examples/models/VPN/VPNPolicyModelJava.json", ⑥
    "periodicEventPeriod": 1000, ⑦
    "engineParameters": {
        "engineParameters": {}, ⑧
        "contextParameters": {} ⑨
    } ⑩
}

① a name for the engine. The engine name is used to create a key in a runtime engine. An name matching the following regular expression can be used here: [A-Za-z0-9\-\_\.]+ ②
② a version of the engine, use semantic versioning as explained here: Semantic Versioning. This version is used in a runtime engine to create a version of the engine. For that reason, the version must match the following regular expression [A-Z0-9.]+ ③
③ a numeric identifier for the engine
④ the number of threads (policy instances executed in parallel) the engine should use, use 1 for single threaded engines
⑤ the port for the deployment Websocket connection to the engine
⑥ the model file to load into the engine on startup (optional)
⑦ an optional timer for periodic policies, in milliseconds (a defined periodic policy will be executed every X milliseconds), not used of not set or 0
⑧ engine parameters for plugin configurations (execution environments and context handling)
⑨ engine specific parameters, mainly for executor plugins
⑩ context specific parameters, e.g. for context schemas, persistence, etc.

The model file is optional, it can also be specified via command line. In any case, make sure all execution and other required plug-ins for the loaded model are loaded as required.

2.4. Input and Output Interfaces

An APEX engine has two main interfaces:

• An input interface to receive events: also known as ingress interface or consumer, receiving (consuming) events commonly named triggers, and

• An output interface to publish produced events: also known as egress interface or producer, sending (publishing) events commonly named actions or action events.

The input and output interface is configured in terms of inputs and outputs, respectively. Each input and output is a combination of a carrier technology and an event protocol. Carrier technologies and event protocols are provided by plugins, each with its own specific configuration. Most carrier technologies can be configured for input as well as output. Most event protocols can be used for all carrier technologies. One exception is the JMS object event protocol, which can only be used for the JMS carrier technology. Some further restrictions apply (for instance for carrier technologies using bi- or uni-directional modes).

Input and output interface can be configured separately, in isolation, with any number of carrier technologies. The resulting general configuration options are:
• Input interface with one or more inputs
  ◦ each input with a carrier technology and an event protocol
  ◦ some inputs with optional synchronous mode
  ◦ some event protocols with additional parameters

• Output interface with one or more outputs
  ◦ each output with a carrier technology and an event encoding
  ◦ some outputs with optional synchronous mode
  ◦ some event protocols with additional parameters

The configuration for input and output is contained in `eventInputParameters` and `eventOutputParameters`, respectively. Inside here, one can configure any number of inputs and outputs. Each of them needs to have a unique identifier (name), the content of the name is free form. The example below shows a configuration for two inputs and two outputs.

```
"eventInputParameters": { ①
  "FirstConsumer": { ②
    "carrierTechnologyParameters" : {...}, ③
    "eventProtocolParameters":{...}, ④
    ... ⑤
  },
  "SecondConsumer": { ⑥
    "carrierTechnologyParameters" : {...}, ⑦
    "eventProtocolParameters":{...}, ⑧
    ... ⑨
  },
},
"eventOutputParameters": { ⑩
  "FirstProducer": { ⑪
    "carrierTechnologyParameters":{...}, ⑫
    "eventProtocolParameters":{...}, ⑬
    ... ⑭
  },
  "SecondProducer": { ⑮
    "carrierTechnologyParameters":{...}, ⑯
    "eventProtocolParameters":{...}, ⑰
    ... ⑱
  }
}
```

① input interface configuration, APEX input plugins
② first input called `FirstConsumer`
③ carrier technology for plugin
④ event protocol for plugin
⑤ any other input configuration (e.g. event name filter, see below)
⑥ second input called `SecondConsumer`
⑦ carrier technology for plugin
⑧ event protocol for plugin
⑨ any other plugin configuration
⑩ output interface configuration, APEX output plugins
⑪ first output called `FirstProducer`
2.4.1. Event Filters

APEX will always send an event after a policy execution is finished. For a successful execution, the event sent is the output event created by the policy. In case the policy does not create an output event, APEX will create a new event with all input event fields plus an additional field `exceptionMessage` with an exception message.

There are situations in which this auto-generated error event might not be required or wanted:

- when a policy failing should not result in an event send out via an output interface
- when the auto-generated event goes back in an APEX engine (or the same APEX engine), this can create endless loops
- the auto-generated event should go to a special output interface or channel

All of these situations are supported by a filter option using a wildcard (regular expression) configuration on APEX I/O interfaces. The parameter is called `eventNameFilter` and the value are Java regular expressions (a tutorial). The following code shows some examples:

```json
"eventInputParameters": {
  "Input1": {
    "carrierTechnologyParameters": {...},
    "eventProtocolParameters": {...},
    "eventNameFilter": "^E[Vv][Ee][Nn][Tt][0-9]+$" ①
  }
},
"eventOutputParameters": {
  "Output1": {
    "carrierTechnologyParameters": {...},
    "eventProtocolParameters": {...},
    "eventNameFilter": "^E[Vv][Ee][Nn][Tt][0-9]+$" ②
  }
}
```

2.5. Executors

Executors are plugins that realize the execution of logic contained in a policy model. Logic can be in a task selector, a task, and a state finalizer. Using plugins for execution environments makes APEX very flexible to support virtually any executable logic expressions.

APEX 2.1.0-SNAPSHOT supports the following executors:

- Java, for Java implemented logic
  - This executor requires logic implemented using the APEX Java interfaces.
  - Generated JAR files must be in the classpath of the APEX engine at start time.
• Javascript
• JRuby,
• Jython,
• MVEL
  ◦ This executor uses the latest version of the MVEL engine, which can be very hard to debug and can produce unwanted side effects during execution

2.5.1. Configure the Javascript Executor

The Javascript executor is added to the configuration as follows:

```json
"engineServiceParameters": {
  "engineParameters": {
    "executorParameters": {
      "JAVASCRIPT": {
        "parameterClassName": "org.onap.policy.apex.plugins.executor.javascript.JavascriptExecutorParameters"
      }
    }
  }
}
```

2.5.2. Configure the Jython Executor

The Jython plugin allows you to use extra Python packages installed with `pip` or at startup using the `setup.py` or `build_py.py` configuration files. Extra modules must be checked by developers prior to installation to ensure that they are not malicious and do not exploit the Python Path Traversal vulnerability. The Jython plugin does NOT check extra modules for security vulnerabilities.

The Jython executor is added to the configuration as follows:

```json
"engineServiceParameters": {
  "engineParameters": {
    "executorParameters": {
      "PYTHONT";
      "org.onap.policy.apex.plugins.executor.jython.JythonExecutorParameters"
    }
  }
}
```
"engineServiceParameters": {
  "engineParameters": {
    "executorParameters": {
      "JYTHON": {
        "parameterClassName": "org.onap.policy.apex.plugins.executor.jruby.JrubyExecutorParameters"
      }
    }
  }
}

"engineServiceParameters": {
  "engineParameters": {
    "executorParameters": {
      "JRUBY": {
        "parameterClassName": "org.onap.policy.apex.plugins.executor.jruby.JrubyExecutorParameters"
      }
    }
  }
}

:leveloffset!:  
:leveloffset: +2

// == Configure the Java Executor
// The Java executor is added to the configuration as follows:
[source=\n]
== Configure the MVEL Executor

The MVEL executor is added to the configuration as follows:

```json
"engineServiceParameters": {
  "engineParameters": {
    "executorParameters": {
      "MVEL": {
        "parameterClassName": "org.onap.policy.apex.plugins.executor.mvel.MVELExecutorParameters"
      }
    }
  }
}
```
== Context Handlers

Context handlers are responsible for all context processing. There are the following main areas:
- Context schema: use schema handlers other than Java class (supported by default without configuration)
- Context distribution: distribute context across multiple APEX engines
- Context locking: mechanisms to lock context elements for read/write
- Context persistence: mechanisms to persist context

APEX provides plugins for each of the main areas.

== Configure AVRO Schema Handler

The AVRO schema handler is added to the configuration as follows:

```json
"engineServiceParameters": {
  "engineParameters": {
    "contextParameters": {
      "parameterClassName": "org.onap.policy.apex.context.parameters.ContextParameters",
      "schemaParameters": {
        "Avro": {
          "parameterClassName": "org.onap.policy.apex.plugins.context.schema.avro.AvroSchemaHelperParameters"
        }
      }
    }
  }
}
```

Using the AVRO schema handler has one limitation: AVRO only supports field names that represent valid Java class names. This means only letters and the character `_` are supported. Characters commonly used in field names, such as `.` and `-`, are not supported by AVRO.

For more information see link: https://avro.apache.org/docs/1.8.1/spec.html#names[Avro Spec: Names].
To work with this limitation, the APEX Avro plugin will parse a given AVRO definition and replace _all_ occurrences of `. ` and `-` with a `_`. This means that

- In a policy model, if the AVRO schema defined a field as `my-name` the policy logic should access it as `my_name`
- In a policy model, if the AVRO schema defined a field as `my.name` the policy logic should access it as `my_name`
- There should be no field names that convert to the same internal name
  ** For instance the simultaneous use of `my_name`, `my.name`, and `my-name` should be avoided
  ** If not avoided, the event processing might create unwanted side effects
- If field names use any other not-supported character, the AVRO plugin will reject it
  ** Since AVRO uses lazy initialization, this rejection might only become visible at runtime

---

// == Carrier Technologies

Carrier technologies define how APEX receives (input) and sends (output) events. They can be used in any combination, using asynchronous or synchronous mode. There can also be any number of carrier technologies for the input (consume) and the output (produce) interface.

Supported _input_ technologies are:

- Standard input, read events from the standard input (console), not suitable for APEX background servers
- File input, read events from a file
- Kafka, read events from a Kafka system
- Websockets, read events from a Websocket
- JMS,
- REST (synchronous and asynchronous), additionally as client or server
- Event Requestor, allows reading of events that have been looped back into APEX

Supported _output_ technologies are:

- Standard output, write events to the standard output (console), not suitable for APEX background servers
- File output, write events to a file
- Kafka, write events to a Kafka system
- Websockets, write events to a Websocket
- JMS
- REST (synchronous and asynchronous), additionally as client or server
- Event Requestor, allows events to be looped back into APEX

New carrier technologies can be added as plugins to APEX or developed outside APEX and added to an APEX deployment.
Standard IO does not require a specific plugin, it is supported by default.

APEX will take events from its standard input. This carrier is good for testing, but certainly not for a use case where APEX runs as a server. The configuration is as follows:

```
"carrierTechnologyParameters": { "carrierTechnology": "FILE", 
                               "parameters": { "standardIO": true } }
```

APEX will send events to its standard output. This carrier is good for testing, but certainly not for a use case where APEX runs as a server. The configuration is as follows:

```
"carrierTechnologyParameters": { "carrierTechnology": "FILE", 
                               "parameters": { "standardIO": true } }
```
== File IO

File IO does not require a specific plugin, it is supported by default.

=== File Input

APEX will take events from a file.
The same file should not be used as an output.
The configuration is as follows:

[source%nowrap,json]

"carrierTechnologyParameters" : { "carrierTechnology" : "FILE", // <1> "parameters" : { "fileName" : "examples/events/SampleDomain/EventsIn.xmlfile" // <2> } }

<1> set file input
<2> the name of the file to read events from

=== File Output

APEX will write events to a file.
The same file should not be used as an input.
The configuration is as follows:

[source%nowrap,json]

"carrierTechnologyParameters" : { "carrierTechnology" : "FILE", // <1> "parameters" : { "fileName" : "examples/events/SampleDomain/EventsOut.xmlfile" // <2> } }
Event Requestor IO does not require a specific plugin, it is supported by default. It should only be used with the APEX event protocol.

APEX will take events from APEX.

```
"carrierTechnologyParameters": { "carrierTechnology": "EVENT_REQUESTOR" // <1> }
```

APEX will write events to APEX.

```
"carrierTechnologyParameters": { "carrierTechnology": "EVENT_REQUESTOR" // <1> }
```

When using event requestors, they need to be peered. This means an event requestor output needs to be peered (associated) with an event requestor input.

The following example shows the use of an event requestor with the APEX event protocol and the peering of output and input.

```
```
event requestor on a consumer
with APEX event protocol
optional filter (best to use a filter to prevent unwanted events on the consumer side)
activate requestor mode
the peer to the output (must match the output carrier)
an optional timeout in milliseconds

event requestor on a producer
with APEX event protocol
optional filter (best to use a filter to prevent unwanted events on the consumer side)
activate requestor mode
the peer to the output (must match the input carrier)
an optional timeout in milliseconds

-- Kafka IO

Kafka IO is supported by the APEX Kafka plugin.
The configurations below are examples.
APEX will take any configuration inside the parameter object and forward it to Kafka.
More information on Kafka specific configuration parameters can be found in the Kafka documentation:

* link:https://kafka.apache.org/090/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html[Kafka Consumer Class]
* link:https://kafka.apache.org/090/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html[Kafka Producer Class]

== Kafka Input

APEX will receive events from the Apache Kafka messaging system.
The input is uni-directional, an engine will only receive events from the input but not send any event to the input.

```json
```
APEX will send events to the Apache Kafka messaging system. The output is uni-directional, an engine will send events to the output but not receive any event from the output.

```
```
APEX supports the Java Messaging Service (JMS) as input as well as output.

JMS IO is supported by the APEX JMS plugin. Input and output support an event encoding as text (JSON string) or object (serialized object). The input configuration is the same for both encodings, the output configuration differs.

--- JMS Input

APEX will receive events from a JMS messaging system. The input is uni-directional, an engine will only receive events from the input but not send any event to the input.

```json
```
set JMS as carrier technology
set all JMS specific parameters
the context factory, in this case from JBOSS (it requires the dependency org.jboss:jboss-remote-naming:2.0.4.Final or a different version to be in the directory `$APEX_HOME/lib` or `%APEX_HOME%\lib` a connection factory for the JMS connection
URL with host and port of the JMS provider
access credentials, user name
access credentials, user password
the JMS topic to listen to

APEX engine send events to a JMS messaging system.
The output is uni-directional, an engine will send events to the output but not receive any event from output.

```json
```
set JMS as carrier technology
set all JMS specific parameters
the context factory, in this case from JBOSS (it requires the dependency org.jboss:jboss-remote-naming:2.0.4.Final or a different version to be in the directory `$APEX_HOME/lib` or `%APEX_HOME%\lib`
a connection factory for the JMS connection
URL with host and port of the JMS provider
access credentials, user name
access credentials, user password
the JMS topic to write to
set object messaging to `false` means it sends JSON text

== JMS Output with Object

To configure APEX for JMS objects on the output interface use the same configuration as above (for output).
Simply change the `objectMessageSending` parameter to `true`.

APEX supports the Websockets as input as well as output.
WS IO is supported by the APEX Websocket plugin.
This carrier technology does only support uni-directional communication.
APEX will not send events to a Websocket input and any event sent to a Websocket output will result in an error log.

The input can be configured as client (APEX connects to an existing Websocket server) or server (APEX starts a Websocket server).
The same applies to the output.
Input and output can both use a client or a server configuration, or separate configurations (input as client and output as server, input as server and output as client).
Each configuration should use its own dedicated port to avoid any communication loops.
The configuration of a Websocket client is the same for input and output.
The configuration of a Websocket server is the same for input and output.

APEX will connect to a given Websocket server.
As input, it will receive events from the server but not send any events.
As output, it will send events to the server and any event received from the server will result in an error log.

```
"carrierTechnologyParameters" : {
  "carrierTechnology" : "WEB_SOCKET",
  "parameterClassName" : "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
  "parameters" : {
"carrierTechnologyParameters" : {
  "carrierTechnology" : "WEB_SOCKET",
  "parameterClassName" : "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
  "parameters" : {

set Websocket as carrier technology
the host name on which a Websocket server is running
the port of that Websocket server

APEX will start a Websocket server, which will accept any Websocket clients to connect.
As input, it will receive events from the server but not send any events.
As output, it will send events to the server and any event received from the server will result in an error log.

APEX can act as REST client on the input as well as on the output interface.
The media type is `application/json`, so this plugin does only work with the JSON Event protocol.

APEX will connect to a given URL to receive events, but not send any events.
The server is polled, i.e. APEX will do an HTTP GET, take the result, and then do the next GET.
Any required timing needs to be handled by the server configured via the URL.
For instance, the server could support a wait timeout via the URL as `?timeout=100ms`.
APEX will connect to a given URL to send events, but not receive any events.
The default HTTP operation is POST (no configuration required).
To change it to PUT simply add the configuration parameter (as shown in the example below).

```json
```
APEX supports a REST server for input and output.

The REST server plugin always uses a synchronous mode. A client does a HTTP GET on the APEX REST server with the input event and receives the generated output event in the server reply. This means that for the REST server there has to always be an input with an associated output. Input or output only are not permitted.

The plugin will start a Grizzly server as REST server for a normal APEX engine. If the APEX engine is executed as a servlet, for instance inside Tomcat, then Tomcat will be used as REST server (this case requires configuration on Tomcat as well).

Some configuration restrictions apply for all scenarios:

- Minimum port: 1024
- Maximum port: 65535
- The media type is `application/json`, so this plugin does only work with the JSON Event protocol.

The URL the client calls is created using

- the configured host and port, e.g. `http://localhost:12345`
- the standard path, e.g. `/apex/`
- the name of the input/output, e.g. `FirstConsumer/`
- the input or output name, e.g. `EventIn`.

The examples above lead to the URL `http://localhost:12345/apex/FirstConsumer/EventIn`.

A client can also get status information of the REST server using `/Status`, e.g. `http://localhost:12345/apex/FirstConsumer/Status`.

--- REST Server Stand-alone

We need to configure a REST server input and a REST server output. Input and output are associated with each other via there name.

Timeouts for REST calls need to be set carefully. If they are too short, the call might timeout before a policy finished creating an event.

The following example configures the input named as `MyConsumer` and associates an output named `MyProducer` with it.

```json
[source=json]
```
"eventInputParameters": { "MyConsumer": { "carrierTechnologyParameters": { "carrierTechnology": "RESTSERVER", "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.restserver.RESTServerCarrierTechnologyParameters", "parameters": { "standalone": true, "host": "localhost", "port": 12345 } }, "eventProtocolParameters": { "eventProtocol": "JSON" }, "synchronousMode": true, "synchronousPeer": "MyProducer", "synchronousTimeout": 500 } }

1> set REST server as carrier technology
2> set the server as stand-alone
3> set the server host
4> set the server listen port
5> use JSON event protocol
6> activate synchronous mode
7> associate an output `MyProducer`
8> set a timeout of 500 milliseconds

The following example configures the output named as `MyProducer` and associates the input `MyConsumer` with it. Note that for the output there are no more parameters (such as host or port), since they are already configured in the associated input

[source%nowrap,json]

"eventOutputParameters": { "MyProducer": { "carrierTechnologyParameters": { "carrierTechnology": "RESTSERVER", "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.restserver.RESTServerCarrierTechnologyParameters" }, "eventProtocolParameters": { "eventProtocol": "JSON" }, "synchronousMode": true, "synchronousPeer": "MyConsumer", "synchronousTimeout": 500 } }

=== REST Server Stand-alone, multi input

Any number of input/output pairs for REST servers can be configured. For instance, we can configure an input `FirstConsumer` with output `FirstProducer` and an input `SecondConsumer` with output `SecondProducer`. Important is that there is always one pair of input/output.

=== REST Server Stand-alone in Servlet

If APEX is executed as a servlet, e.g. inside Tomcat, the configuration becomes easier since the plugin can now use Tomcat as the REST server. In this scenario, there are not parameters (port, host, etc.) and the key `standalone` must not be used (or set to false).

For the Tomcat configuration, we need to add the REST server plugin, e.g.

[source%nowrap,xml]

<servlet> ... <init-param> ... <param-value>org.onap.policy.apex.plugins.event.carrier.restserver</param-value> </init-param> ... </servlet>
APEX can act as REST requestor on the input as well as on the output interface. The media type is `application/json`, so this plugin does only work with the JSON Event protocol.

=== REST Requestor Input
APEX will connect to a given URL to request a input.

```
"carrierTechnologyParameters": { "carrierTechnology": "RESTREQUESTOR", "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.restrequestor.RESTRequestorCarrierTechnologyParameters", "parameters": { "url": "http://localhost:54321/some/path/to/rest/resource", "httpMethod": "POST", "restRequestTimeout": 2000 } },
```

1. set REST requestor as carrier technology
2. the URL of the HTTP server for events
3. use HTTP PUT (remove this line to use HTTP POST)
4. request timeout in milliseconds

Further settings are required on the consumer to define the event that is requested, for example:

```
"eventName": "GuardResponseEvent", "eventNameFilter": "GuardResponseEvent", "requestorMode": true, "requestorPeer": "GuardRequestorProducer", "requestorTimeout": 500
```

1. the event name
2. a filter on the event
3. the mode of the requestor
4. a peer for the requestor
5. a general request timeout

=== REST Requestor Output
APEX will connect to a given URL to send events, but not receive any events.

```
"carrierTechnologyParameters": { "carrierTechnology": "RESTREQUESTOR", "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.restrequestor.RESTRequestorCarrierTechnologyParameters" },
```
Further settings are required on the consumer to define the event that is requested, for example:

```
"eventNameFilter": "GuardRequestEvent",  
"requestorMode": true,  
"requestorPeer": "GuardRequestorConsumer",  
"requestorTimeout": 500
```

- a filter on the event
- the mode of the requestor
- a peer for the requestor
- a general request timeout

---

Event protocols define what event formats APEX can receive (input) and should send (output). They can be used in any combination for input and output, unless further restricted by a carrier technology plugin (for instance for JMS output). There can only be 1 event protocol per event plugin.

Supported _input_ event protocols are:
- JSON, the event as a JSON string
- APEX, an APEX event
- JMS object, the event as a JMS object,
- JMS text, the event as a JMS text,
- XML, the event as an XML string,
- YAML, the event as YAML text

Supported _output_ event protocols are:
- JSON, the event as a JSON string
- APEX, an APEX event
- JMS object, the event as a JMS object,
- JMS text, the event as a JMS text,
- XML, the event as an XML string,
- YAML, the event as YAML text

New event protocols can be added as plugins to APEX or developed outside APEX and added to an APEX deployment.
== JSON Event

The event protocol for JSON encoding does not require a specific plugin, it is supported by default. Furthermore, there is no difference in the configuration for the input and output interface.

For an input, APEX requires a well-formed JSON string. Well-formed here means according to the definitions of a policy. Any JSON string that is not defined as a trigger event (consume) will not be consumed (errors will be thrown). For output JSON events, APEX will always produce valid JSON strings according to the definition in the policy model.

The following JSON shows the configuration.

[source%nowrap,json]
```
"eventProtocolParameters":{ "eventProtocol" : "JSON" }
```

For JSON events, there are a few more optional parameters, which allow to define a mapping for standard event fields. An APEX event must have the fields `name`, `version`, `source`, and `target` defined. Sometimes it is not possible to configure a trigger or actioning system to use those fields. However, they might be in an event generated outside APEX (or used outside APEX) just with different names. To configure APEX to map between the different event names, simply add the following parameters to a JSON event:

[source%nowrap,json]
```
"eventProtocolParameters":{ "eventProtocol" : "JSON", "nameAlias" : "policyName", <1> "versionAlias" : "policyVersion", <2> "sourceAlias" : "from", <3> "targetAlias" : "to", <4> "nameSpaceAlias" : "my.name.space" <5> }
```
The event protocol for APEX events does not require a specific plugin, it is supported by default. Furthermore, there is no difference in the configuration for the input and output interface.

For input and output APEX uses APEX events.

The following JSON shows the configuration.

```json
"eventProtocolParameters":{ "eventProtocol": "APEX" }
```
== JMS Event

The event protocol for JMS is provided by the APEX JMS plugin. The plugin supports encoding as JSON text or object. There is no difference in the configuration for the input and output interface.

=== JMS Text

If used as input, APEX will take a JMS message and extract a JSON string, then proceed as if a JSON event was received.
If used as output, APEX will take the event produced by a policy, create a JSON string, and then wrap it into a JMS message.

The configuration for JMS text is as follows:

[source%nowrap, json]

"eventProtocolParameters": { "eventProtocol" : "JMSTEXT", "parameterClassName" : "org.onap.policy.apex.plugins.event.protocol.jms.JMSTextEventProtocolParameters" }
The event protocol for YAML is provided by the APEX YAML plugin. There is no difference in the configuration for the input and output interface.

If used as input, APEX will consume events as YAML and map them to policy trigger events. Not well-formed YAML and not understood trigger events will be rejected.

If used as output, APEX produce YAML encoded events from the event a policy produces. Those events will always be well-formed according to the definition in the policy model.

The following code shows the configuration.

```json
"eventProtocolParameters": {  "eventProtocol" : "XML",  "parameterClassName" : "org.onap.policy.apex.plugins.event.protocol.yaml.YamlEventProtocolParameters" }
```
The event protocol for XML is provided by the APEX XML plugin. There is no difference in the configuration for the input and output interface.

If used as input, APEX will consume events as XML and map them to policy trigger events. Not well-formed XML and not understood trigger events will be rejected. If used as output, APEX produce XML encoded events from the event a policy produces. Those events will always be well-formed according to the definition in the policy model.

The following code shows the configuration.

```json
"eventProtocolParameters": {  "eventProtocol": "XML",  "parameterClassName": "org.onap.policy.apex.plugins.event.protocol.xml.XMLEventProtocolParameters" }
```

== A configuration example

The following example loads all available plug-ins. Events are consumed from a Websocket, APEX as client. Consumed event format is JSON. Events are produced to Kafka. Produced event format is XML.
== Engine and Applications of the APEX System

== Introduction to APEX Engine and Applications

The core of APEX is the APEX Engine, also known as the APEX Policy Engine or the APEX PDP (since it is in fact a Policy Decision Point).

Beside this engine, an APEX system comes with a few applications intended to help with policy authoring, deployment, and execution.

The engine itself and most applications are started from the command line with command line arguments. This is called a Command Line Interface (CLI).

Some applications require an installation on a webserver, as for instance the REST Editor. Those applications can be accessed via a web browser.

You can also use the available APEX APIs and applications to develop other applications as required. This includes policy languages (and associated parsers and compilers / interpreters), GUIs to access APEX or to define policies, clients to connect to APEX, etc.

For this documentation, we assume an installation of APEX as a full system based on a current ONAP release.
A note on APEX CLI applications: all applications and the engine itself have been deployed and tested on different operating systems: Red Hat, Ubuntu, Debian, Mac OSX, Windows, Cygwin. Each operating system comes with its own way of configuring and executing Java.

The main items here are:

- For UNIX systems (RHL, Ubuntu, Debian, Mac OSX), the provided bash scripts work as expected with absolute paths (e.g. `/opt/app/policy/apex-pdp/apex-pdp-{release-version}/examples`), indirect and linked paths (e.g. `..\apex\apex`), and path substitutions using environment settings (e.g. `$APEX_HOME/bin/`)
- For Windows systems, the provided batch files (`.bat`) work as expected with absolute paths (e.g. `C:\apex\apex-{release-version}\examples`), and path substitutions using environment settings (e.g. `%APEX_HOME%\bin\`)
- For Cygwin system we assume a standard Cygwin installation with standard tools (mainly bash) using a Windows Java installation.

This means that the bash scripts can be used as in UNIX, however any argument pointing to files and directories need to use either a DOS path (e.g. `C:\apex\apex-{release-version}\examples\config\...`) or the command `cygpath` with a mixed option.

The reason for that is: Cygwin executes Java using UNIX paths but then runs Java as a DOS/WINDOWS process, which requires DOS paths for file access.

== The APEX Engine

The APEX engine can be started in different ways, depending your requirements. All scripts are located in the APEX __bin__ directory

On UNIX and Cygwin systems use:

- `apexEngine.sh` - this script will
  ** Test if `$APEX_USER` is set and if the user exists, terminate with an error otherwise
  ** Test if `$APEX_HOME` is set. If not set, it will use the default setting as `/opt/app/policy/apex-pdp/apex-pdp`.
  Then the set directory is tested to exist, the script will terminate if not.
  ** When all tests are passed successfully, the script will call `apexApps.sh` with arguments to start the APEX engine.
- `apexApps.sh engine` - this is the general APEX application launcher, which will
  ** Start the engine with the argument `engine`
  ** Test if `$APEX_HOME` is set and points to an existing directory. If not set or directory does not exist, script terminates.
  ** Not test for any settings of `$APEX_USER`.

On Windows systems use `apexEngine.bat` and `apexApps.bat engine` respectively. Note: none of the windows batch files will test for `\APEX_USER\`.

Summary of alternatives to start the APEX Engine:

| width="100X",options="header",cols="5a,5a" |
2.6. `$APEX_HOME/bin/apexEngine.sh` [args]

2.7. `$APEX_HOME/bin/apexApps.sh` `engine` [args]

---

The APEX engine comes with a few CLI arguments for setting configuration and policy model. The configuration file is always required. The policy model file is only required if no model file is specified in the configuration, or if the specified model file should be overwritten. The option `-h` prints a help screen.

```
usage: org.onap.policy.apex.service.engine.main.ApexMain [options...] options -c,--config-file <CONFIG_FILE> the full path to the configuration file to use, the configuration file must be a Json file containing the Apex configuration parameters -h,--help outputs the usage of this command -m,--model-file <MODEL_FILE> the full path to the model file to use, if set it overrides the model file set in the configuration file -v,--version outputs the version of Apex
```
The CLI Editor allows to define policies from the command line. The application uses a simple language and supports all elements of an APEX policy. It can be used in two different ways:

- non-interactive, specifying a file with the commands to create a policy
- interactive, using the editors CLI to create a policy

When a policy is fully specified, the editor will generate the APEX core policy specification in JSON. This core specification is called the policy model in the APEX engine and can be used directly with the APEX engine.

On UNIX and Cygwin systems use:

- `apexCLIEditor.sh` - simply starts the CLI editor, arguments to the script determine the mode of the editor
- `apexApps.sh cli-editor` - simply starts the CLI editor, arguments to the script determine the mode of the editor

On Windows systems use:

- `apexCLIEditor.bat` - simply starts the CLI editor, arguments to the script determine the mode of the editor
- `apexApps.bat cli-editor` - simply starts the CLI editor, arguments to the script determine the mode of the editor

Summary of alternatives to start the APEX CLI Editor:

<table>
<thead>
<tr>
<th>Unix, Cygwin</th>
<th>Windows</th>
</tr>
</thead>
</table>

```bash
2.8. $APEX_HOME/bin/apexCLIEditor.sh.sh [args]

2.9. $APEX_HOME/bin/apexApps.sh cli-editor [args]
```
usage: org.onap.policy.auth.clieditor.ApexCLIEditorMain [options...] options -a,--model-props-file <MODEL_PROPS_FILE> name of the apex model properties file to use -c,--command-file <COMMAND_FILE> name of a file containing editor commands to run into the editor -h,--help outputs the usage of this command -i,--input-model-file <INPUT_MODEL_FILE> name of a file that contains an input model for the editor -if,--ignore-failures <IGNORE_FAILURES_FLAG> true or false, ignore failures of commands in command files and continue executing the command file -l,--log-file <LOG_FILE> name of a file that will contain command logs from the editor, will log to standard output if not specified or suppressed with "-nl" flag -m,--metadata-file <CMD_METADATA_FILE> name of the command metadata file to use -nl,--no-log if specified, no logging or output of commands to standard output or log file is carried out -nm,--no-model-output if specified, no output of a model to standard output or model output file is carried out, the user can use the "save" command in a script to save a model -o,--output-model-file <OUTPUT_MODEL_FILE> name of a file that will contain the output model for the editor, will output model to standard output if not specified or suppressed with "-nm" flag -wd,--working-directory <WORKING_DIRECTORY> the working directory that is the root for the CLI editor and is the root from which to look for included macro files

The option `-h` provides a help screen with all command line arguments.

[source

```bash
usage: org.onap.policy.auth.clieditor.ApexCLIEditorMain [options...] options -a,--model-props-file <MODEL_PROPS_FILE> name of the apex model properties file to use -c,--command-file <COMMAND_FILE> name of a file containing editor commands to run into the editor -h,--help outputs the usage of this command -i,--input-model-file <INPUT_MODEL_FILE> name of a file that contains an input model for the editor -if,--ignore-failures <IGNORE_FAILURES_FLAG> true or false, ignore failures of commands in command files and continue executing the command file -l,--log-file <LOG_FILE> name of a file that will contain command logs from the editor, will log to standard output if not specified or suppressed with "-nl" flag -m,--metadata-file <CMD_METADATA_FILE> name of the command metadata file to use -nl,--no-log if specified, no logging or output of commands to standard output or log file is carried out -nm,--no-model-output if specified, no output of a model to standard output or model output file is carried out, the user can use the "save" command in a script to save a model -o,--output-model-file <OUTPUT_MODEL_FILE> name of a file that will contain the output model for the editor, will output model to standard output if not specified or suppressed with "-nm" flag -wd,--working-directory <WORKING_DIRECTORY> the working directory that is the root for the CLI editor and is the root from which to look for included macro files
```
== The APEX REST Editor

The standard way to use the APEX REST Editor is via an installation of the __war__ file on a webserver. However, the REST editor can also be started via command line. This will start a Grizzly webserver with the __war__ deployed. Access to the REST Editor is then via the provided URL

On UNIX and Cygwin systems use:

- `$apexRESTEditor.sh` - simply starts the webserver with the REST editor
- `$apexApps.sh rest-editor` - simply starts the webserver with the REST editor

On Windows systems use:

- `apexRESTEditor.bat` - simply starts the webserver with the REST editor
- `apexApps.bat rest-editor` - simply starts the webserver with the REST editor

Summary of alternatives to start the APEX REST Editor:

```bash
| Width="100%",options="header",cols="5a,5a"
| Unix, Cygwin | Windows
|
| source%nowrap,sh]
```

2.10. `$APEX_HOME/bin/apexRESTEditor.sh [args]`

2.11. `$APEX_HOME/bin/apexApps.sh rest-editor [args]`

```bash
| %APEX_HOME%\bin\apexRESTEditor.bat [args] %APEX_HOME%\bin\apexApps.bat rest-editor [args]
```

======================

The option `-h` provides a help screen with all command line arguments.

[source%nowrap,sh]
usage: org.onap.policy.apex.client.editor.rest.ApexEditorMain [options...] -h,--help outputs the usage of this command -l,--listen <ADDRESS> the IP address to listen on. Default value is localhost to restrict access to the local machine only. -p,--port <PORT> port to use for the Apex RESTful editor REST calls. -t,--time-to-live <TIME_TO_LIVE> the amount of time in seconds that the server will run for before terminating. Default value is -1 to run indefinitely.

If the REST Editor is started without any arguments the final messages will look similar to this:

```source%nowrap,sh```

Apex Editor REST endpoint (ApexEditorMain: Config=[ApexEditorParameters: URI=http://localhost:18989/apexservices/, TTL=-1sec], State=READY) starting at http://localhost:18989/apexservices/...


The last line states the URL on which the REST Editor can be accessed.
The example above stated `http://0.0.0.0:18989/apex/`.
In a web browser use the URL `http://localhost:18989` and the REST Editor will start.

```leveloffset: 2```
```leveloffset: +1```
```// ============LICENSE_START==================================================================================================```
```// Copyright (C) 2016-2018 Ericsson. All rights reserved.```
```// ============LICENSE_END==================================================================================================```
```// @author Sven van der Meer (sven.van.der.meer@ericsson.com)```
```// == The APEX Monitoring Client```
```The standard way to use the APEX Monitoring Client is via an installation of the __war__ file on a webserver.```
```However, the Monitoring Client can also be started via command line. This will start a Grizzly webserver with the __war__ deployed. Access to the Monitoring Client is then via the provided URL```
```On UNIX and Cygwin systems use:```
```  - `apexApps.sh eng-monitoring` - simply starts the webserver with the Monitoring Client```
```On Windows systems use:```
```  - `apexApps.bat eng-monitoring` - simply starts the webserver with the Monitoring Client```
```The option `^-h` provides a help screen with all command line arguments.```
```[source%nowrap,sh]```

usage: org.onap.policy.apex.client.monitoring.rest.ApexMonitoringRestMain [options...] -h,--help outputs the usage of this command -p,--port <PORT> port to use for the Apex Services REST calls -t,--time-to-live <TIME_TO_LIVE> the
amount of time in seconds that the server will run for before terminating

If the Monitoring Client is started without any arguments the final messages will look similar to this:

```sh
[source%nowrap,sh]
```


The last line states the URL on which the Monitoring Client can be accessed.
The example above stated `http://localhost:18989/apexservices`
In a web browser use the URL `http://localhost:18989`.

```
usage: org.onap.policy.apex.client.deployment.rest.ApexDeploymentRestMain [options...] -h,--help outputs the usage of this command -p,--port <PORT> port to use for the Apex Services REST calls -t,--time-to-live <TIME_TO_LIVE> the amount of time in seconds that the server will run for before terminating
```
If the Deployment Client is started without any arguments the final messages will look similar to this:

```
```

The last line states the URL on which the Deployment Client can be accessed. The example above stated `http://localhost:18989/apexservices`. In a web browser use the URL `http://localhost:18989`.

```
usage: org.onap.policy.apex.client.full.rest.ApexServicesRestMain [options...] -h,--help outputs the usage of this command -p,--port <PORT> port to use for the Apex Services REST calls -t,--time-to-live <TIME_TO_LIVE> the amount of time in seconds that the server will run for before terminating
```
If the Full Client is started without any arguments the final messages will look similar to this:


The last line states the URL on which the Monitoring Client can be accessed. The example above stated `http://localhost:18989/apexservices`. In a web browser use the URL `http://localhost:18989`.

<p>|====================|   |</p>
<table>
<thead>
<tr>
<th>Unix, Cygwin</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.12. $APEX_HOME/bin/apexApps.sh [args]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[source%nowrap,sh,leveloffset: +1,leveloffset: 2]</td>
</tr>
</tbody>
</table>

== The APEX Application Launcher
The standard applications (Engine, CLI Editor, REST Editor) come with dedicated start scripts. For all other APEX applications, we provide an application launcher.

On UNIX and Cygwin systems use:
- `apexApps.sh` - simply starts the application launcher

On Windows systems use:
- `apexApps.bat` - simply starts the application launcher

Summary of alternatives to start the APEX application launcher:
The option `-h` provides a help screen with all launcher command line arguments.

```
Usage: apexApps.sh [options] | [<application> [<application options>]]
```

Options

```
-d <app>  - describes an application
-l        - lists all applications supported by this script
-h        - this help screen
```

Using `-l` lists all known applications the launcher can start.

```
apexApps.sh: supported applications: ws-echo engine eng-monitoring full-client eng-deployment tpl-event-json model-2-cli rest-editor cli-editor ws-console
```

Using the `-d <name>` option describes the named application, for instance for the `ws-console`:

```
apexApps.sh: application 'ws-console' -
```

Launching an application is done by calling the script with only the application name and any CLI arguments for the application.

For instance, starting the `ws-echo` application with port `8888`:

```
apexApps.sh ws-echo -p 8888
```
This application takes a policy model (JSON or XML encoded) and generates templates for events in JSON format. This can help when a policy defines rather complex trigger or action events or complex events between states. The application can produce events for the types: stimuli (policy trigger events), internal (events between policy states), and response (action events).

2.13. $APEX_HOME/bin/apexApps.sh tpl-event-json [args]

The option `-h` provides a help screen.

The created templates are not valid events, instead they use some markup for values one will need to change to actual values.

For instance, running the tool with the __Sample Domain__ policy model as:
apexApps.sh tpl-event-json -m $APEX_HOME/examples/models/SampleDomain/SamplePolicyModelJAVA.json -t stimuli

will produce the following status messages:

```
[source%nowrap,sh]
gen-model2event: starting Event generator  →  model file: examples/models/SampleDomain/SamplePolicyModelJAVA.json → type: stimuli
```

and then run the generator application producing two event templates. The first template is called `Event0000`.

```
[source%nowrap,json]
{
  "name" : "Event0000",
  "nameSpace" : "org.onap.policy.apex.sample.events",
  "version" : "0.0.1",
  "source" : "Outside",
  "target" : "Match",
  "TestTemperature" : double: 0.0,
  "TestTimestamp" : long: 0,
  "TestMatchCase" : integer: 0,
  "TestSlogan" : "string"
}
```

The values for the keys are marked with `###` and the expected type of the value. To create an actual stimuli event, all these markers need to be change to actual values, for instance:

```
[source%nowrap,json]
{
  "name" : "Event0000",
  "nameSpace" : "org.onap.policy.apex.sample.events",
  "version" : "0.0.1",
  "source" : "Outside",
  "target" : "Match",
  "TestTemperature" : 25,
  "TestTimestamp" : 123456789123456789,
  "TestMatchCase" : 1,
  "TestSlogan" : "Testing the Match Case with Temperature 25"
}
```

---

```
leveloffset: 2

leveloffset: +1

//
// ===========LICENSE_START=============================================
// Copyright (C) 2016-2018 Ericsson. All rights reserved.
// ====================================================================
// This file is licensed under the CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE
// Full license text at https://creativecommons.org/licenses/by/4.0/legalcode
//
// SPDX-License-Identifier: CC-BY-4.0
// ===========LICENSE_END==============================================
//
// @author Sven van der Meer (sven.van.der.meer@ericsson.com)
//
== Application: Convert a Policy Model to CLI Editor Commands

**Status: Experimental**

This application takes a policy model (JSON or XML encoded) and generates commands for the APEX CLI Editor. This effectively reverses a policy specification realized with the CLI Editor.

```
The option `-h` provides a help screen.

usage: gen-model2cli -h,--help prints this help and usage screen -m,-model <MODEL-FILE> set the input policy model file -sv,-skip-validation switch of validation of the input file -v,-version prints the application version

For instance, running the tool with the __Sample Domain__ policy model as:

`apexApps.sh model-2-cli -m $APEX_HOME/examples/models/SampleDomain/SamplePolicyModelJAVA.json`

will produce the following status messages:

`gen-model2cli: starting CLI generator -- model file: examples/models/SampleDomain/SamplePolicyModelJAVA.json`
and then run the generator application producing all CLI Editor commands and printing them to standard out.

```plaintext
# == Application: Websocket Clients (Echo and Console)
**Status: Production**

The application launcher also provides a Websocket echo client and a Websocket console client. The echo client connects to APEX and prints all events it receives from APEX. The console client connects to APEX, reads input from the command line, and sends this input as events to APEX.

<table>
<thead>
<tr>
<th>Unix, Cygwin</th>
<th>Windows</th>
</tr>
</thead>
</table>

2.15. `$APEX_HOME/bin/apexApps.sh ws-echo [args]`

2.16. `$APEX_HOME/bin/apexApps.sh ws-console [args]`

The arguments are the same for both applications:

- `-p` defines the Websocket port to connect to (defaults to `8887`)
- `-s` defines the host on which a Websocket server is running (defaults to `localhost`)

A discussion on how to use these two applications to build an APEX system is detailed HowTo-Websockets.
== Introduction

Consider a scenario where a supermarket chain called _HyperM_ controls how it sells items in a policy-based manner. Each time an item is processed by _HyperM_'s point-of-sale (PoS) system an event is generated and published about that item of stock being sold. This event can then be used to update stock levels, etc.

__HyperM__ want to extend this approach to allow some checks to be performed before the sale can be completed. This can be achieved by requesting a policy-controlled decision as each item is processed by for sale by each PoS system. The decision process is integrated with __HyperM__'s other IT systems that manage stock control, sourcing and purchasing, personnel systems, etc.

In this document we will show how APEX and APEX Policies can be used to achieve this, starting with a simple policy, building up to more complicated policy that demonstrates the features of APEX.

== Data Models

=== Sales Input Event

Each time a PoS system processes a sales item an event with the following format is emitted:

<table>
<thead>
<tr>
<th>Event</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALE_INPUT</td>
<td>time, sale_ID, amount, item_ID, quantity, assistant_ID, branch_ID, notes, ...</td>
<td>Event indicating a sale of an item is occurring</td>
</tr>
</tbody>
</table>
In each `SALE_INPUT` event the `sale_ID` field is a unique ID generated by the PoS system. A timestamp for the event is stored in the `time` field. The `amount` field refers to the value of the item(s) to be sold (in cents). The `item_ID` field is a unique identifier for each item type, and can be used to retrieve more information about the item from _HyperM_'s stock control system. The `quantity` field refers to the quantity of the item to be sold. The `assistant_ID` field is a unique identifier for the PoS operator, and can be used to retrieve more information about the operator from the _HyperM_'s personnel system. Since _HyperM_ has many branches the `branch_ID` identifies the shop. The `notes` field contains arbitrary notes about the sale.

### Sales Decision Event

After a `SALE_INPUT` event is emitted by the PoS system _HyperM_'s policy-based controlled sales checking system emits a Sale Authorization Event indicating whether the sale is authorized or denied. The PoS system can then listen for this event before continuing with the sale.

<table>
<thead>
<tr>
<th>Event</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALE_AUTH</td>
<td>sale_ID, time, authorized, amount, item_ID, quantity, assistant_ID, branch_ID, notes, message...</td>
<td>Event indicating a sale of an item is authorized or denied</td>
</tr>
</tbody>
</table>

In each `SALE_AUTH` event the `sale_ID` field is copied from the `SALE_INPUT` event that trigger the decision request. The `SALE_AUTH` event is also timestamped using the `time` field, and a field called `authorised` is set to `true` or `false` depending on whether the sale is authorized or denied. The `message` field carries an optional message about why a sale was not authorized. The other fields from the `SALE_INPUT` event are also included for completeness.

### Stock Control: Items

_HyperM_ maintains information about each item for sale in a database table called `ITEMS`.

<table>
<thead>
<tr>
<th>Table</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEMS</td>
<td>item_ID, description, cost_price, barcode, supplier_ID, category, ...</td>
<td>Database table describing each item for sale</td>
</tr>
</tbody>
</table>

The database table `ITEMS` has a row for each items that _HyperM_ sells. Each item is identified by an `item_ID` value. The `description` field stores a description of the item. The cost price of the item is given in `cost_price`. The barcode of the item is encoded in `barcode`, while the item supplier is identified by `supplier_ID`. Items may also be classified into categories using the `category` field. Useful categories might include: `soft drinks`, `alcoholic drinks`, `cigarettes`, `knives`, `confectionery`, `bakery`, `fruit&vegetables`, `meat`, etc..

### Personnel System: Assistants

_HyperM_ maintains information about each sales assistant employed by _HyperM_. Each assistant is identified by an `assistant_ID` value, with their name given in the `firstname`,
`middlename` and `surname` fields.
The assistant's age in years is given in `age`, while their phone number is contained in the `phone_number` field.
The assistant's grade is encoded in `grade`.
Useful values for `grade` might include: `trainee`, `operator`, `supervisor`, etc..

--- Locations: Branches

<table>
<thead>
<tr>
<th>Table</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCHES</td>
<td>branch_ID, branch_Name, category, street, city, country, postcode, ...</td>
<td>Database table describing each <em>HyperM</em> branch</td>
</tr>
</tbody>
</table>

_HyperM_ operates a number of branches. Each branch is described in the `BRANCHES` database table. Each branch is identified by a `branch_ID`, with a branch name given in `branch_Name`. The address for the branch is encoded in `street`, `city`, `country` and `postcode`. The branch category is given in the `category` field. Useful values for `category` might include: `Small`, `Large`, `Super`, `Hyper`, etc..

--- Policy Step 1

--- Scenario
For the first version of our policy, let's start with something simple. Let us assume that there exists some restriction that alcohol products cannot be sold before 11:30am. In this section we will go through the necessary steps to define a policy that can enforce this for _HyperM_.

* Alcohol cannot be sold before 11:30am.
=== Create the an empty Policy Model `MyFirstPolicyModel`  

Since an organisation like _HyperM_ may have many policies covering many different domains, policies should be grouped into policy sets. In order to edit or deploy a policy, or policy set, the definition of the policy(ies) and all required events, tasks, states, etc., are grouped together into a 'Policy Model'. An organization might define many Policy Models, each containing a different set of policies.

So the first step is to create a new empty Policy Model called `MyFirstPolicyModel`. Using the APEX Policy Editor, click on the 'File' menus and select 'New'. Then define our new policy model called `MyFirstPolicyModel`. Use the 'Generate UUID' button to create a new unique ID for the policy model, and fill in a description for the policy model. Press the 'Submit' button to save your changes.

.Create a new Policy Model 1/2
image::mfp/MyFirstPolicy_P1_newPolicyModel1.png[File > New to create a new Policy Model]

.Create a new Policy Model 2/2
image::mfp/MyFirstPolicy_P1_newPolicyModel2.png[Create a new Policy Model]

---

=== Create the input event `SALE_INPUT` and the output event `SALE_AUTH`  

Using the APEX Policy Editor, click on the 'Events' tab. In the 'Events' pane, right click and select 'New':

.Create a new Event type
image::mfp/MyFirstPolicy_P1_newEvent1.png[Right click to create a new event]

Create a new event type called `SALE_INPUT`. Use the 'Generate UUID' button to create a new unique ID for the event type, and fill in a description for the event. Add a namespace, e.g. `com.hyperm`. We can add hard-coded strings for the `Source` and `Target`, e.g. `POS` and `APEX`. At this stage we will not add any parameter fields, we will leave this until later. Use the `Submit` button to create the event.

.Populate the `SALE_INPUT` event
image::mfp/MyFirstPolicy_P1_newEvent2.png["Fill in the necessary information for the 'SALE_INPUT' event and click 'Submit'"]

Repeat the same steps for a new event type called `SALE_AUTH`. Just use `APEX` as source and `POS` as target, since this is the output event coming from APEX going to the sales point.

Before we can add parameter fields to an event we must first define APEX Context Item Schemas that can be used by those fields.
To create new item schemas, click on the 'Context Item Schemas' tab. In that 'Context Item Schemas' pane, right click and select 'Create new ContextSchema'.

Create item schemas with the following characteristics, each with its own unique UUID:

<table>
<thead>
<tr>
<th>Name</th>
<th>Schema Flavour</th>
<th>Schema Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>time</code> values</td>
</tr>
<tr>
<td>sale_ID_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>sale_ID</code> values</td>
</tr>
<tr>
<td>price_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>price</code> values</td>
</tr>
<tr>
<td>item_ID_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>item_ID</code> values</td>
</tr>
<tr>
<td>assistant_ID_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>assistant_ID</code> values</td>
</tr>
<tr>
<td>quantity_type</td>
<td>Java</td>
<td>java.lang.Integer</td>
<td>A type for <code>quantity</code> values</td>
</tr>
<tr>
<td>branch_ID_type</td>
<td>Java</td>
<td>java.lang.Long</td>
<td>A type for <code>branch_ID</code> values</td>
</tr>
<tr>
<td>notes_type</td>
<td>Java</td>
<td>java.lang.String</td>
<td>A type for <code>notes</code> values</td>
</tr>
<tr>
<td>authorised_type</td>
<td>Java</td>
<td>java.lang.Boolean</td>
<td>A type for <code>authorised</code> values</td>
</tr>
<tr>
<td>message_type</td>
<td>Java</td>
<td>java.lang.String</td>
<td>A type for <code>message</code> values</td>
</tr>
</tbody>
</table>

The item schemas can now be seen on the 'Context Item Schemas' tab, and can be updated at any time by right-clicking on the item schemas on the 'Context Item Schemas' tab. Now we can go back to the event definitions for `SALE_INPUT` and `SALE_AUTH` and add some parameter fields.

[TIP]

Field Schema types
================================
APEX natively supports schema definitions in `Java` and `Avro`.

`Java` schema definitions are simply the name of a Java Class. There are some restrictions:

* the class must be instantiatable, i.e. not an Java interface or abstract class
* primitive types are not supported, i.e. use `java.lang.Integer` instead of `int`, etc.
* it must be possible to find the class, i.e. the class must be contained in the Java classpath.

`Avro` schema definitions can be any valid https://avro.apache.org/docs/current/spec.html[Avro] schema. For events using fields defined with `Avro` schemas, any incoming event containing that field must contain a value that conforms to the Avro schema.

Click on the 'Events' tab, then right click the `SALE_INPUT` row and select 'Edit Event `SALE_INPUT`'. To add a new event parameter use the 'Add Event Parameter' button at the bottom of the screen. For the `SALE_INPUT` event add the following event parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>timestamp_type</td>
<td>no</td>
</tr>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
<td>no</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
<td>no</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
<td>yes</td>
</tr>
</tbody>
</table>
Remember to click the 'Submit' button at the bottom of the event definition pane.

[TIP]
.Optional Fields in APEX Events
================================
Parameter fields can be _optional_ in events. If a parameter is not marked as _optional_ then by default it is _mandatory_, so it must appear in any input event passed to APEX. If an _optional_ field is not set for an output event then value will be set to `null`.
================================

.Add typed parameter fields to an event
image::mfp/MyFirstPolicy_P1_newEvent3.png["Add new event parameters to an event"]
Select the `SALE_AUTH` event and add the following event parameters:

.Event Parameter Fields for the `SALE_AUTH` Event
[width="100\%",options="header"]
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>time</td>
<td>timestamp_type</td>
<td>no</td>
</tr>
<tr>
<td>authorised</td>
<td>authorised_type</td>
<td>no</td>
</tr>
<tr>
<td>message</td>
<td>message_type</td>
<td><em>yes</em></td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
<td>no</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
<td>no</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
<td>no</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
<td><em>yes</em></td>
</tr>
</tbody>
</table>

Remember to click the 'Submit' button at the bottom of the event definition pane.

The events for our policy are now defined.

APEX policies are defined using a state-machine model. Each policy comprises one or more _states_ that can be individually executed. Where there is more than one _state_ the states are chained together to form a [Directed Acyclic Graph (DAG)](https://en.wikipedia.org/wiki/Directed_acyclic_graph) of states. A _state_ is triggered by passing it a single input (or 'trigger') event and once executed each state then emits an output event. For each _state_ the logic for the _state_ is embedded in one or more _tasks_. Each _task_ contains specific _task logic_ that is executed by the APEX execution environment each time the _task_ is invoked. Where there is more than one _task_ in a _state_ then the _state_ also defines some _task selection logic_ to select an appropriate task each time the _state_ is executed.

Therefore, to create a new policy we must first define one or more tasks.

66
To create a new Task click on the 'Tasks' tab. In the 'Tasks' pane, right click and select 'Create new Task'. Create a new Task called `MorningBoozeCheck`. Use the 'Generate UUID' button to create a new unique ID for the task, and fill in a description for the task.

![Right click to create a new task](image:mfp/MyFirstPolicy_P1_newTask1.png)

Tasks are configured with a set of _input fields_ and a set of _output fields_. To add new input/output fields for a task use the 'Add Task Input Field' and 'Add Task Output Field' button. The list of input and output fields to add for the `MorningBoozeCheck` task are given below. The input fields are drawn from the parameters in the state's input event, and the task's output fields are used to populate the state's output event. The task's input and output fields must be a subset of the event parameters defined for the input and output events for any state that uses that task. (You may have noticed that the input and output fields for the `MorningBoozeCheck` task have the exact same names and reuse the item schemas that we used for the parameters in the `SALE_INPUT` and `SALE_AUTH` events respectively).

### Input fields for `MorningBoozeCheck` task

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>timestamp_type</td>
</tr>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
</tr>
</tbody>
</table>

### Output fields for `MorningBoozeCheck` task

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
</tr>
<tr>
<td>time</td>
<td>timestamp_type</td>
</tr>
<tr>
<td>authorised</td>
<td>authorised_type</td>
</tr>
<tr>
<td>message</td>
<td>message_type</td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
</tr>
</tbody>
</table>

### Add input and out fields for the Task

![Add input and out fields for the task](image:mfp/MyFirstPolicy_P1_newTask2.png)

Each task must include some 'Task Logic' that implements the behaviour for the task. Task logic can be defined in a number of different ways using a choice of languages. For this task we will author the logic using the Java-like scripting language called https://en.wikipedia.org/wiki/MVEL[`MVEL`].

For simplicity use the following code for the task logic. Paste the script text into the 'Task Logic' box, and use "MVEL" as the 'Task Logic Type / Flavour'.

```java
This logic assumes that all items with `item_ID` between 1000 and 2000 contain alcohol, which is not very realistic, but we will see a better approach for this later. It also uses the standard `Java` time utilities to check if the current time is between `00:00:00 GMT` and `11:30:00 GMT`. For a detailed guide to how to write your own logic in https://en.wikipedia.org/wiki/JavaScript[`JavaScript`], https://en.wikipedia.org/wiki/MVEL[`MVEL`] or one of the other supported languages please refer to APEX Programmers Guide.

`MVEL` code for the `MorningBoozeCheck` task

[source,java,options="nowrap"]
logger.info("Task Execution: ":subject.id". Input Fields: ":inFields"");

outFields.put("amount", inFields.get("amount")); outFields.put("assistant_ID", inFields.get("assistant_ID"));
outFields.put("notes", inFields.get("notes"); outFields.put("quantity", inFields.get("quantity");
outFields.put("branch_ID", inFields.get("branch_ID"); outFields.put("item_ID", inFields.get("item_ID");
outFields.put("time", inFields.get("time"); outFields.put("sale_ID", inFields.get("sale_ID");

item_id = inFields.get("item_ID");

gmt = TimeZone.getTimeZone("GMT"); timenow = Calendar.getInstance(gmt); df = new SimpleDateFormat("HH:mm:ss z"); df.setTimeZone(gmt); timenow.setTimeInMillis(inFields.get("time");

midnight = timenow.clone(); midnight.set( timenow.get(Calendar.YEAR),timenow.get(Calendar.MONTH),
timenow.get(Calendar.DATE),0,0,0); eleven30 = timenow.clone(); eleven30.set( timenow.get(Calendar.YEAR),timenow.get(Calendar.MONTH), timenow.get(Calendar.DATE),11,30,0);

itemisalcohol = false; if(item_id != null && item_id >=1000 && item_id < 2000) itemisalcohol = true;

if( itemisalcohol && timenow.after(midnight) && timenow.before(eleven30)) { outFields.put("authorised", false); outFields.put("message", "Sale not authorised by policy task ":subject.taskName" for time "df.format(timenow.getTime()) ". Alcohol can not be sold between "df.format(midnight.getTime()) " and "df.format(eleven30.getTime())"; return true; } else{ outFields.put("authorised", true); outFields.put("message", "Sale authorised by policy task "+subject.taskName " for time "+df.format(timenow.getTime())); return true; }

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 GMT and 11:30:00 GMT then the sale is not authorised. Otherwise the sale is authorised. In this implementation we assume that items with item_ID value between 1000 and 2000 are all alcoholic drinks :-) */

.Add Task Logic the Task
image::mfp/MyFirstPolicy_P1_newTask3.png["Add task logic the task"]

An alternative version of the same logic is available in JavaScript.
Just use "JAVASCRIPT" as the 'Task Logic Type / Flavour' instead.

.Javascript alternative for the `MorningBoozeCheck` task
[source,javascript,options="nowrap"]
```javascript
var returnValueType = Java.type("java.lang.Boolean"); var returnValue = new returnValueType(true);
load("nashorn:mozilla_compat.js"); importPackage(java.text); importClass(java.text.SimpleDateFormat);
executor.logger.info("Task Execution: "; executor.subject.id". Input Fields: "; executor.inFields""");
executor.outFields.put("amount" , executor.inFields.get("amount")); executor.outFields.put("assistant_ID", executor.inFields.get("assistant_ID")); executor.outFields.put("notes" , executor.inFields.get("notes"));
executor.outFields.put("quantity" , executor.inFields.get("quantity")); executor.outFields.put("branch_ID" , executor.inFields.get("branch_ID")); executor.outFields.put("item_ID" , executor.inFields.get("item_ID"));
executor.outFields.put("time" , executor.inFields.get("time")); executor.outFields.put("sale_ID" , executor.inFields.get("sale_ID"));
item_id = executor.inFields.get("item_ID");
var timenow_gmt = new Date(Number(executor.inFields.get("time")));
var midnight_gmt = new Date(Number(executor.inFields.get("time"))); midnight_gmt.setUTCHours(0,0,0,0);
var eleven30_gmt = new Date(Number(executor.inFields.get("time"))); eleven30_gmt.setUTCHours(11,30,0,0);
var timeformatter = new java.text.SimpleDateFormat("HH:mm:ss z");
var itemisalcohol = false; if(item_id != null & item_id >=1000 & item_id < 2000) itemisalcohol = true;
if( itemisalcohol && timenow_gmt.getTime() >= midnight_gmt.getTime() && timenow_gmt.getTime() < eleven30_gmt.getTime()) {
    executor.outFields.put("authorised", false);
    executor.outFields.put("message", "Sale not authorised by policy task "+
    executor.subject.taskName+ " for time "+ timeformatter.format(timenow_gmt.getTime()) +
    ". Alcohol can not be sold between "+ timeformatter.format(midnight_gmt.getTime()) +
    " and "+ timeformatter.format(eleven30_gmt.getTime()));
}
else{
    executor.outFields.put("authorised", true);
    executor.outFields.put("message", "Sale authorised by policy task "+
    executor.subject.taskName+ " for time "+ timeformatter.format(timenow_gmt.getTime()));
}
```

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 GMT and 11:30:00 GMT then the sale is not authorised. Otherwise the sale is authorised. In this implementation we assume that items with item_ID value between 1000 and 2000 are all alcoholic drinks :-) */

The task definition is now complete so click the 'Submit' button to save the task. The task can now be seen on the 'Tasks' tab, and can be updated at any time by right-clicking on the task on the 'Task' tab. Now that we have created our task, we can create a policy that uses that task.

To create a new Policy click on the 'Policies' tab. In the 'Policies' pane, right click and select 'Create new Policy'.

Create a new Policy called 'MyFirstPolicy'. Use the 'Generate UUID' button to create a new unique ID for the policy, and fill in a description for the policy. Use 'FREEFORM' as the 'Policy Flavour'.

Each policy must have at least one state. Since this is 'freeform' policy we can add as many states as we
wish. Let's start with one state. Add a new state called 'BoozeAuthDecide' to this 'MyFirstPolicy' policy using the 'Add new State' button after filling in the name of our new state.

_Create a new Policy_

[Create a new policy]

Each state must use one input event type. For this new state select the `SALE_INPUT` event as the input event.

Each policy must define a 'First State' and a 'Policy Trigger Event'. The 'Policy Trigger Event' is the input event for the policy as a whole. This event is then passed to the first state in the chain of states in the policy, therefore the 'Policy Trigger Event' will be the input event for the first state. Each policy can only have one 'First State'. For our 'MyFirstPolicy' policy, select 'BoozeAuthDecide' as the 'First State'. This will automatically select 'SALE_INPUT' as the 'Policy Trigger Event' for our policy.

_Create a new State_

[Create a state]

In this case we will create a reference to the pre-existing `MorningBoozeCheck` task that we defined above using the 'Add New Task' button. Select the `MorningBoozeCheck` task, and use the name of the task as the 'Local Name' for the task.

_in the case where a state references more than one task, a 'Default Task' must be selected for the state and some logic ('Task Selection Logic') must be specified to select the appropriate task at execution time. Since our new state 'BoozeAuthDecide' only has one task the default task is automatically selected and no 'Task Selection Logic' is required._

_Note_

_State Output Mappings_

In a 'Policy' 'State' a 'State Output Mapping' has 3 roles: 1) Select which 'State' should be executed next, 2) Select the type of the state's 'Outgoing Event', and 3) Populate the state's 'Outgoing Event'. This is how states are chained together to form a (https://en.wikipedia.org/wiki/Directed_acyclic_graph[Directed Acyclic Graph (DAG)]) of states. The final state(s) of a policy are those that do not select any 'next' state. Since a 'State' can only accept a single type of event, the type of the event emitted by a previous 'State' must be match the incoming event type of the next 'State'. This is also how the last state(s) in a policy can emit events of different types. The 'State Output Mapping' is also responsible for taking the fields that are output by the task executed in the state and populating the state's output event before it is emitted.

Each 'Task' referenced in 'State' must have a defined 'Output Mapping' to take the output of the task, select an 'Outgoing Event' type for the state, populate the state's outgoing event, and then select the next state to be executed (if any).

There are 2 basic types of output mappings:

* _Direct Output Mappings_ have a single value for 'Next State' and a single value for 'State Output Event'. The outgoing event for the state is automatically created, any outgoing event parameters that were present in the incoming event are copied into the outgoing event, then any task output fields that have the same name and type as parameters in the outgoing event are automatically copied into the outgoing event.

* _Logic-based State Output Mappings / Finalizers_ have some logic defined that dynamically selects and creates the 'State Outgoing Event', manages the population of the outgoing event parameters (perhaps changing or adding to the outputs from the task), and then dynamically selects the next state to be executed (if any).

Each task reference must also have an associated 'Output State Mapping' so we need an 'Output State Mapping' for the 'BoozeAuthDecide' state to use when the 'MorningBoozeCheck' task is executed. The simplest type of output mapping is a 'Direct Output Mapping'.

Create a new 'Direct Output Mapping' for the state called 'MorningBoozeCheck_Output_Direct' using the 'Add New Direct State Output Mapping' button. Select 'SALE_AUTH' as the output event and select 'None' for the next state value. We can then select this output mapping for use when the the 'MorningBoozeCheck' task is executed. Since there is only state, and only one task for that state, this output mapping ensures that the 'BoozeAuthDecide' state is the only state executed and the state (and the policy) can only emit events
of type 'SALE_AUTH'. (You may remember that the output fields for the 'MorningBoozeCheck' task have the same
exact same names and reuse the item schemas that we used for the parameters in 'SALE_AUTH' event. The
'MorningBoozeCheck_Output_Direct' direct output mapping can now automatically copy the values from the
'MorningBoozeCheck' task directly into outgoing 'SALE_AUTH' events.)

Add a Task and Output Mapping

Click the 'Submit' button to complete the definition of our 'MyFirstPolicy' policy. The policy
'MyFirstPolicy' can now be seen in the list of policies on the 'Policies' tab, and can be updated at any
time by right-clicking on the policy on the 'Policies' tab.

The 'MyFirstPolicyModel', including our 'MyFirstPolicy' policy can now be checked for errors. Click on the
'Model' menu and select 'Validate'. The model should validate without any 'Warning' or 'Error' messages.
If you see any 'Error' or 'Warning' messages, carefully read the message as a hint to find where you might
have made a mistake when defining some aspect of your policy model.

Validate a Policy Model

Congratulations, you have now completed your first APEX policy. The policy model containing our new policy
can now be exported from the editor and saved. Click on the 'File' menu and select 'Download' to save the
policy model in JSON format.
The exported policy model is then available in the directory you selected, for instance
'$APEX_HOME/examples/models/MyFirstPolicy/1/MyFirstPolicyModel_0.0.1.json'.
The exported policy can now be loaded into the APEX Policy Engine, or can be re-loaded and edited by the
APEX Policy Editor.

Export a Policy Model

To start a new APEX Engine you can use the following configuration.
In a full APEX installation you can find this configuration in
'$APEX_HOME/examples/config/MyFirstPolicy/1/MyFirstPolicyConfigStdin2StdoutJsonEvent.json'.
This configuration expects incoming events to be in `JSON` format and to be passed into the APEX Engine
from `stdin`, and result events will be printed in `JSON` format to `stdout`. This configuration loads the
policy model stored in the file 'MyFirstPolicyModel_0.0.1.json' as exported from the APEX Editor. Note,
you may need to edit this file to provide the full path to wherever you stored the exported policy model
file.

`JSON` to load and execute _My First Policy_, read input JSON events from `stdin`, and emit output events
to `stdout`
[source,json,options="nowrap"]
To test the policy try paste the following events into the console as the APEX engine executes:

<table>
<thead>
<tr>
<th>Input Event (JSON)</th>
<th>Output Event (JSON)</th>
<th>comment</th>
</tr>
</thead>
</table>
| ```json
{ "nameSpace": "com.hyperm", "name": "SALE_INPUT", "version": "0.0.1", "time": 1483351989000, "sale_ID": 99999991, "amount": 299, "item_ID": 5123, "quantity": 1, "assistant_ID": 23, "branch_ID": 1, "notes": "Special Offer!!" }
``` | ```json
{ "name": "SALE_AUTH", "version": "0.0.1", "nameSpace": "com.hyperm", "source": ",", "target": ",", "amount": 299, "assistant_ID": 23, "authorised": true, "branch_ID": 1, "item_ID": 5123, "message": "Sale authorised by policy task MorningBoozeCheck for time 10:13:09 GMT", "notes": "Special Offer!!", "quantity": 1, "sale_ID": 99999991, "time": 1483351989000 }
``` | Request to buy a non-alcoholic item (`item_ID=5123`) at _10:13:09_ on _Tuesday, 10 January 2017_. Sale is authorized. |
| ```json
{ "nameSpace": "com.hyperm", "name": "SALE_INPUT", "version": "0.0.1", "time": 1483346466000, "sale_ID": 99999992, "amount": 1249, "item_ID": 1012, "quantity": 1, "assistant_ID": 12, "branch_ID": 2 }
``` | ```json
{ "name": "SALE_AUTH", "version": "0.0.1", "nameSpace": "com.hyperm", "source": ",", "target": ",", "amount": 1249, "assistant_ID": 12, "authorised": false, "branch_ID": 2, "item_ID": 1012, "message": "Sale not authorised by policy task MorningBoozeCheck for time 08:41:06 GMT. Alcohol can not be sold between 00:00:00 GMT and 11:30:00 GMT", "notes": null, "quantity": 1, "sale_ID": 99999992, "time": 1483346466000 }
``` | Request to buy a non-alcoholic item (`item_ID=1012`) at _08:41:06_ on _Monday, 9 January 2017_. Sale is not authorized. |
| Request to buy alcohol item (`item_ID=1249`) at _08:41:06_ on _Monday, 02 January 2017_.

Sale is not authorized.

```json
{
  "nameSpace": "com.hyperm",
  "name": "SALE_INPUT",
  "version": "0.0.1",
  "time": 148226503000,
  "sale_ID": 99999993,
  "amount": 4799,
  "item_ID": 1943,
  "quantity": 2,
  "assistant_ID": 9,
  "branch_ID": 3
}
```

| Request to buy alcohol (`item_ID=1943`) at _20:17:13_ on _Tuesday, 20 December 2016_.

Sale is authorized.

```apex
#Siemens License Start==================================================================
// Copyright © 2016-2018 Siemens. All rights reserved.
// This file is licensed under the CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE
// Full license text at https://creativecommons.org/licenses/by/4.0/legalcode
// SPDX-License-Identifier: CC-BY-4.0
// Siemen License End====================================================================
// @author Sven van der Meer (sven.van.der.meer@siemens.com)
#
== Policy 1 in CLI Editor ==

An equivalent version of the `MyFirstPolicyModel` policy model can again be generated using the APEX CLI editor.
A sample APEX CLI script is shown below:

APEX CLI Editor code for Policy 1
```
model create name=MyFirstPolicyModel version=0.0.1 uuid=540226fb-55ee-4f0e-a444-983a0494818e
description="This is my first Apex Policy Model."

schema create name=assistant_ID_type version=0.0.1 uuid=36df4c71-9616-4206-8b53-976a5cd4bd87
description="A type for 'assistant_ID' values" flavour=Java schema=java.lang.Long

schema create name=authorised_type version=0.0.1 uuid=d48b619e-d00d-4008-b884-02d76ea4350b
description="A type for 'authorised' values" flavour=Java schema=java.lang.Boolean

schema create name=branch_ID_type version=0.0.1 uuid=6468845f-4122-4128-8e49-0f52c26078b5
description="A type for 'branch_ID' values" flavour=Java schema=java.lang.Long

schema create name=item_ID_type version=0.0.1 uuid=4f227ff1-aae0-453a-b6b6-9a4b2e0da932
description="A type for 'item_ID' values" flavour=Java schema=java.lang.Long

schema create name=message_type version=0.0.1 uuid=ad1431bb-3155-4e73-b5a3-b89bee498749
description="A type for 'message' values" flavour=Java schema=java.lang.String

schema create name=notes_type version=0.0.1 uuid=eeecfde90-896c-4333-8f9c-2603ced9e2d
description="A type for 'notes' values" flavour=Java schema=java.lang.String

schema create name=price_type version=0.0.1 uuid=52c2fc45-fd8c-463c-bd6f-d91b0554ae7
description="A type for 'amount'/'price' values" flavour=Java schema=java.lang.Long

schema create name=quantity_type version=0.0.1 uuid=ac3d9842-80af-4a98-951c-bd79a431c613
description="A type for 'quantity' values" flavour=Java schema=java.lang.Integer

schema create name=sale_ID_type version=0.0.1 uuid=cca47d74-7754-4a61-b163-ca31f66b157b
description="A type for 'sale_ID' values" flavour=Java schema=java.lang.Long

schema create name=timestamp_type version=0.0.1 uuid=fdf594e88-411d-4a94-b2be-697b3a07adf
description="A type for 'time' values" flavour=Java schema=java.lang.Long

task create name=MorningBoozeCheck version=0.0.1 uuid=3351b0f4-cf06-4fa2-8823-edf67bd30223
description=LS
This task checks if the sales request is for an item that contains alcohol. If the local time is between 00:00 and 11:30:00 then the sale is not authorised. Otherwise the sale is authorised. In this implementation we assume that all items with item_ID values between 1000 and 2000 contain alcohol :-) LE task inputfield create
task create name=MorningBoozeCheck version=0.0.1 fieldName=sale_ID schemaName=sale_ID_type schemaVersion=0.0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=amount schemaName=price_type schemaVersion=0.0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=assistant_ID schemaName=assistant_ID_type schemaVersion=0.0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=item_ID schemaName=item_ID_type schemaVersion=0.0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=branch_ID schemaName=branch_ID_type schemaVersion=0.0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=time schemaName=timestamp_type schemaVersion=0.0.1 task outputfield create name=MorningBoozeCheck version=0.0.1 fieldName=amount schemaName=price_type schemaVersion=0.0.1 task outputfield create name=MorningBoozeCheck version=0.0.1 fieldName=assistant_ID schemaName=assistant_ID_type schemaVersion=0.0.1 task outputfield create
logger.info("Task Execution: " + subject.id + ". Input Fields: " + inFields + ")

outFields.put("amount", inFields.get("amount")); outFields.put("assistant_ID", inFields.get("assistant_ID"));
outFields.put("notes", inFields.get("notes")); outFields.put("quantity", inFields.get("quantity"));
outFields.put("branch_ID", inFields.get("branch_ID")); outFields.put("item_ID", inFields.get("item_ID"));
outFields.put("time", inFields.get("time")); outFields.put("sale_ID", inFields.get("sale_ID"));

item_id = inFields.get("item_ID");

gmt = TimeZone.getTimeZone("GMT"); timenow = Calendar.getInstance(gmt); df = new
SimpleDateFormat("HH:mm:ss z"); df.setTimeZone(gmt); timenow.setTimeInMillis(inFields.get("time"));

midnight = timenow.clone(); midnight.set(timenow.get(Calendar.YEAR), timenow.get(Calendar.MONTH),
timenow.get(Calendar.DAY_OF_MONTH),0,0,0); eleven30 = timenow.clone(); eleven30.set(timenow.get(Calendar.YEAR),
timenow.get(Calendar.MONTH), timenow.get(Calendar.DAY_OF_MONTH),11,30,0);

itemisalcohol = false; if(item_id != null && item_id >=1000 && item_id < 2000) itemisalcohol = true;

if( itemisalcohol && timenow.after(midnight) && timenow.before(eleven30)){
    outFields.put("authorised", false);
    outFields.put("message", "Sale not authorised by policy task "+subject.taskName+" for time "+df.format(timenow.getTime())+", Alcohol can not be sold between "+df.format(midnight.getTime())+" and "+df.format(eleven30.getTime())+"; return true; } else{
    outFields.put("authorised", true);
    outFields.put("message", "Sale authorised by policy task "+subject.taskName+" for time "+df.format(timenow.getTime())+"; return true; }"}

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 GMT
and 11:30:00 GMT then the sale is not authorised. Otherwise the sale is authorised. In this implementation we
assume that items with item_ID value between 1000 and 2000 are all alcoholic drinks ;-) */

event create name=SALE_AUTH version=0.0.1 uuid=c4500941-3f98-4080-a9cc-5b9753ed050b description="An event
emitted by the Policy to indicate whether the sale of an item has been authorised" nameSpace=com.hyperm
source="APEX" target="POS" event parameter create name=SALE_AUTH version=0.0.1 parName=amount
schemaName=price_type schemaVersion=0.0.1 event parameter create name=SALE_AUTH version=0.0.1 parName=assistant_ID
schemaName=assistant_ID_type schemaVersion=0.0.1 event parameter create name=SALE_AUTH version=0.0.1 parName=authorised
schemaName=authorised_type schemaVersion=0.0.1 event
parameter create name=SALE_AUTH version=0.0.1 parName=branch_ID schemaName=branch_ID_type schemaVersion=0.0.1 event parameter create name=SALE_AUTH version=0.0.1 parName=item_ID schemaName=item_ID_type schemaVersion=0.0.1 event parameter create name=SALE_AUTH version=0.0.1 parName=message schemaName=message_type schemaVersion=0.0.1 optional=true event parameter create name=SALE_AUTH version=0.0.1 parName=notes schemaName=notes_type schemaVersion=0.0.1 optional=true event parameter create name=SALE_AUTH version=0.0.1 parName=quantity schemaName=quantity_type schemaVersion=0.0.1 optional=true event parameter create name=SALE_AUTH version=0.0.1 parName=sale_ID schemaName=sale_ID_type schemaVersion=0.0.1 event parameter create name=SALE_AUTH version=0.0.1 parName=time schemaName=timestamp_type schemaVersion=0.0.1

event create name=SALE_INPUT version=0.0.1 uuid=4f04aa98-e917-4f4a-882a-c75ba5a99374 description="An event raised by the PoS system each time an item is scanned for purchase" nameSpace=com.hyperm source="POS" target="APEX" event parameter create name=SALE_INPUT version=0.0.1 parName=amount schemaName=price_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=assistant_ID schemaName=assistant_ID_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=branch_ID schemaName=branch_ID_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=item_ID schemaName=item_ID_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=notes schemaName=notes_type schemaVersion=0.0.1 optional=true event parameter create name=SALE_INPUT version=0.0.1 parName=quantity schemaName=quantity_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=sale_ID schemaName=sale_ID_type schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=time schemaName=timestamp_type schemaVersion=0.0.1

policy create name=MyFirstPolicy version=0.0.1 uuid=6c5e410f-489a-46ff-964e-982ce6e8b6d0 description="This is my first Apex policy. It checks if a sale should be authorised or not." template=FREEFORM firstState=BoozeAuthDecide policy state create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide triggerName=SALE_INPUT triggerVersion=0.0.1 defaultTaskName=MorningBoozeCheck defaultTaskVersion=0.0.1 policy state output create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide outputName=MorningBoozeCheck_Output_Direct eventName=SALE_AUTH eventVersion=0.0.1 nextState=NULL policy state taskref create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide taskLocalName=MorningBoozeCheck taskName=MorningBoozeCheck taskVersion=0.0.1 outputType=DIRECT outputName=MorningBoozeCheck_Output_Direct

== Policy Step 2

*** Scenario

_HyperM_ have just opened a new branch in a different country, but that country has different rules about when alcohol can be sold! In this section we will go through the necessary steps to extend our policy to enforce this for _HyperM_.

* In some branches alcohol cannot be sold before 1pm, and not at all on Sundays.
Although there are a number of ways to accomplish this the easiest approach for us is to define another task and then select which task is appropriate at runtime depending on the branch identifier in the incoming event.

To create a new Task click on the 'Tasks' tab. In the 'Tasks' pane, right click and select 'Create new Task':

Create a new Task called `MorningBoozeCheckAlt1`. Use the 'Generate UUID' button to create a new unique ID for the task, and fill in a description for the task. Select the same input and output fields that we used earlier when we defined the `MorningBoozeCheck` task earlier.

### Input fields for `MorningBoozeCheckAlt1` task

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>timestamp_type</td>
</tr>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
</tr>
</tbody>
</table>

### Output fields for `MorningBoozeCheckAlt1` task

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sale_ID</td>
<td>sale_ID_type</td>
</tr>
<tr>
<td>time</td>
<td>timestamp_type</td>
</tr>
<tr>
<td>authorised</td>
<td>authorised_type</td>
</tr>
<tr>
<td>message</td>
<td>message_type</td>
</tr>
<tr>
<td>amount</td>
<td>price_type</td>
</tr>
<tr>
<td>item_ID</td>
<td>item_ID_type</td>
</tr>
<tr>
<td>assistant_ID</td>
<td>assistant_ID_type</td>
</tr>
<tr>
<td>quantity</td>
<td>quantity_type</td>
</tr>
<tr>
<td>branch_ID</td>
<td>branch_ID_type</td>
</tr>
<tr>
<td>notes</td>
<td>notes_type</td>
</tr>
</tbody>
</table>

This task also requires some 'Task Logic' to implement the new behaviour for this task.

For simplicity use the following code for the task logic. It again assumes that all items with `item_ID`
between 1000 and 2000 contain alcohol. We again use the standard `Java` time utilities to check if the current time is between `00:00:00 CET` and `13:00:00 CET` or if it is `Sunday`.


MVEL code for the `MorningBoozeCheckAlt1` task
[source,java,options="nowrap"]

```java
/* * ============LICENSE_START======================================================= * Copyright © 2016-2018 Ericsson. All rights reserved. *================================================================================ * Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. * * SPDX-License-Identifier: Apache-2.0 * ============LICENSE_END======================================================== */

import java.util.Date;
import java.util.Calendar;
import java.util.TimeZone;
import java.text.SimpleDateFormat;

logger.info("Task Execution: "+subject.id". Input Event: "inFields";

outFields.put("amount", inFields.get("amount")); outFields.put("assistant_ID", inFields.get("assistant_ID"));
outFields.put("notes", inFields.get("notes")); outFields.put("quantity", inFields.get("quantity"));
outFields.put("branch_ID", inFields.get("branch_ID")); outFields.put("item_ID", inFields.get("item_ID"));
outFields.put("time", inFields.get("time")); outFields.put("sale_ID", inFields.get("sale_ID"));

item_id = inFields.get("item_ID");

cet = TimeZone.getTimeZone("CET"); timenow = Calendar.getInstance(cet); df = new SimpleDateFormat("HH:mm:ss z"); df.setTimeZone(cet); timenow.setTimeInMillis(inFields.get("time"));

midnight = timenow.clone(); midnight.set(timenow.get(Calendar.YEAR), timenow.get(Calendar.MONTH), timenow.get(Calendar.DATE), 0, 0, 0); onepm = timenow.clone(); onepm.set(timenow.get(Calendar.YEAR), timenow.get(Calendar.MONTH), timenow.get(Calendar.DATE), 13, 0, 0);

itemisalcohol = false; if(item_id != null && item_id >=1000 && item_id < 2000) itemisalcohol = true;

if( itemisalcohol && ( (timenow.after(midnight) && timenow.before(onepm)) || (timenow.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) ) ){
outFields.put("authorised", false);
outFields.put("message", "Sale not authorised by policy task "+subject.taskName " for time "+df.format(timenow.getTime()) ". Alcohol can not be sold between "+df.format(midnight.getTime()) " and "+df.format(onepm.getTime()) +" or on Sunday"); return true; } else{
outFields.put("authorised", true);
outFields.put("message", "Sale authorised by policy task "+subject.taskName " for time "+df.format(timenow.getTime())"); return true; }

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 CET and 13:00:00 CET then the sale is not authorised. Also alcohol sales are not allowed on Sundays. Otherwise the sale is authorised. In this implementation we assume that items with item_ID between 1000 and 2000 are all alcoholic drinks :-) */
```
Create a new Task

The task definition is now complete so click the 'Submit' button to save the task. Now that we have created our task, we can add this task to the single pre-existing state (`BoozeAuthDecide`) in our policy.

To edit the `BoozeAuthDecide` state in our policy click on the 'Policies' tab. In the 'Policies' pane, right click on our 'MyFirstPolicy' policy and select 'Edit'. Navigate to the `BoozeAuthDecide` state in the 'states' section at the bottom of the policy definition pane.

To add our new task `MorningBoozeCheckAlt1`, scroll down to the `BoozeAuthDecide` state in the 'States' section. In the 'State Tasks' section for `BoozeAuthDecide` use the 'Add new task' button. Select our new `MorningBoozeCheckAlt1` task, and use the name of the task as the 'Local Name' for the task. The `MorningBoozeCheckAlt1` task can reuse the same `MorningBoozeCheck_Output_Direct` 'Direct State Output Mapping' that we used for the `MorningBoozeCheck` task. (Recall that the role of the 'State Output Mapping' is to select the output event for the state, and select the next state to be executed. These both remain the same as before.)

Since our state has more than one task we must define some logic to determine which task should be used each time the state is executed. This task selection logic is defined in the state definition. For our `BoozeAuthDecide` state we want the choice of which task to use to be based on the 'branch_ID' from which the 'SALE_INPUT' event originated. For simplicity sake let us assume that branches with 'branch_ID' between '0' and '999' should use the `MorningBoozeCheck` task, and the branches with with 'branch_ID' between '1000' and '1999' should use the `MorningBoozeCheckAlt1` task.

This time, for variety, we will author the task selection logic using the https://en.wikipedia.org/wiki/JavaScript [JavaScript] scripting language. Sample task selection logic code (specified in https://en.wikipedia.org/wiki/JavaScript [JavaScript]) is given below. Paste the script text into the 'Task Selection Logic' box, and use "JAVASCRIPT" as the 'Task Selection Logic Type / Flavour'. It is necessary to mark one of the tasks as the 'Default Task' so that the task selection logic always has a fallback default option in cases where a particular task cannot be selected. In this case the `MorningBoozeCheck` task can be the default task.

```javascript
var returnValueType = Java.type("java.lang.Boolean"); var returnValue = new returnValueType(true);
executor.logger.info("Task Selection Execution: ", "executor.subject.id ", Input Event: "executor.inFields""
branchid = executor.inFields.get("branch_ID"); taskorig = executor.subject.getTaskKey("MorningBoozeCheck"); taskalt = executor.subject.getTaskKey("MorningBoozeCheckAlt1"); taskdef = executor.subject.getDefaultTaskKey();
if(branchid >=0 && branchid <1000){ taskorig.copyTo(executor.selectedTask); } else if (branchid >=1000 && branchid <2000){ taskalt.copyTo(executor.selectedTask); } else{ taskdef.copyTo(executor.selectedTask); }
```
/* This task selection logic selects task "MorningBoozeCheck" for branches with 0 <= branch_ID < 1000 and selects task "MorningBoozeCheckAlt1" for branches with 1000 <= branch_ID < 2000. Otherwise the default task is selected. In this case the default task is also "MorningBoozeCheck" */

.State definition with 2 Tasks and Task Selection Logic
image:mfp/MyFirstPolicy_P2_editState1.png

When complete don't forget to click the 'Submit' button at the bottom of 'Policies' pane for our 'MyFirstPolicy' policy after updating the 'BoozeAuthDecide' state.

Congratulations, you have now completed the second step towards your first APEX policy. The policy model containing our new policy can again be validated and exported from the editor and saved as shown in Step 1.

The exported policy model is then available in the directory you selected, as link:files/mfp-files/2/MyFirstPolicyModel_0.0.1.json[_MyFirstPolicyModel_0.0.1.json_]. The exported policy can now be loaded into the APEX Policy Engine, or can be re-loaded and edited by the APEX Policy Editor.

=== Test Policy Step 2

To start a new APEX Engine you can use the following configuration.

In a full APEX installation you can find this configuration in `$APEX_HOME/examples/config/MyFirstPolicy/2/MyFirstPolicyConfigStdin2StdoutJsonEvent.json`. Note, this has changed from the configuration file in Step 1 to enable the `JAVASCRIPT` executor for our new 'Task Selection Logic'.

To load and execute _My First Policy_, read input JSON events from `stdin`, and emit output events to `stdout`:

`[source, json, options="nowrap"]`

```json
{   "engineServiceParameters": { "name": "MyFirstPolicyApexEngine", "version": "0.0.1", "id": 102, "instanceCount": 4,   "deploymentPort": 12345,   "policyModelFileName": "examples/models/MyFirstPolicy/2/MyFirstPolicyModel_0.0.1.json", "engineParameters": { "executorParameters": { "MVEL": { "parameterClassName": "org.onap.policy.apex.plugins.executor.mvel.MVELExecutorParameters" }, "JAVASCRIPT": { "parameterClassName": "org.onap.policy.apex.plugins.executor.javascript.JavascriptExecutorParameters" } } },   "eventOutputParameters": { "FirstProducer": { "carrierTechnologyParameters": { "carrierTechnology": "FILE", "parameters": { "standardIO": true } }, "eventProtocolParameters": { "eventProtocol": "JSON" } } },   "eventInputParameters": { "FirstConsumer": { "carrierTechnologyParameters": { "carrierTechnology": "FILE", "parameters": { "standardIO": true } }, "eventProtocolParameters": { "eventProtocol": "JSON" } } }
```
To test the policy try paste the following events into the console as the APEX engine executes. Note, all tests from Step 1 will still work perfectly since none of those events originate from a branch with `branch_ID` between `1000` and `2000`. The 'Task Selection Logic' will therefore pick the 'MorningBoozeCheck' task as expected, and will therefore give the same results.

Inputs and Outputs when testing _My First Policy_

<table>
<thead>
<tr>
<th>Input Event (JSON)</th>
<th>Output Event (JSON)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ &quot;nameSpace&quot;: &quot;com.hyperm&quot;, &quot;name&quot;: &quot;SALE_INPUT&quot;, &quot;version&quot;: &quot;0.0.1&quot;, &quot;time&quot;: 148334646000, &quot;sale_ID&quot;: 99999992, &quot;amount&quot;: 1249, &quot;item_ID&quot;: 1012, &quot;quantity&quot;: 1, &quot;assistant_ID&quot;: 12, &quot;branch_ID&quot;: 2 }</code></td>
<td><code>{ &quot;nameSpace&quot;: &quot;com.hyperm&quot;, &quot;name&quot;: &quot;SALE_AUTH&quot;, &quot;version&quot;: &quot;0.0.1&quot;, &quot;source&quot;: &quot;&quot;, &quot;target&quot;: &quot;&quot;, &quot;amount&quot;: 1249, &quot;assistant_ID&quot;: 12, &quot;authorised&quot;: false, &quot;branch_ID&quot;: 2, &quot;item_ID&quot;: 1012, &quot;message&quot;: &quot;Sale not authorised by policy task MorningBoozeCheck for time 08:41:06 GMT. Alcohol can not be sold between 00:00:00 GMT and 11:30:00 GMT&quot;, &quot;notes&quot;: null, &quot;quantity&quot;: 1, &quot;sale_ID&quot;: 99999992, &quot;time&quot;: 148334646000 }</code></td>
<td>Request to buy alcohol item (<code>item_ID=1249</code>) at <em>08:41:06 GMT</em> on <em>Monday, 02 January 2017</em>. Sale is not authorized. Uses the 'MorningBoozeCheck' task. Note this test is copied from Step 1 above, and demonstrates that the original 'MorningBoozeCheck' task is executed.</td>
</tr>
<tr>
<td><code>{ &quot;nameSpace&quot;: &quot;com.hyperm&quot;, &quot;name&quot;: &quot;SALE_INPUT&quot;, &quot;version&quot;: &quot;0.0.1&quot;, &quot;time&quot;: 1482398073000, &quot;sale_ID&quot;: 99999981, &quot;amount&quot;: 299, &quot;item_ID&quot;: 1047, &quot;quantity&quot;: 1, &quot;assistant_ID&quot;: 1212, &quot;branch_ID&quot;: 1002 }</code></td>
<td><code>{ &quot;nameSpace&quot;: &quot;com.hyperm&quot;, &quot;name&quot;: &quot;SALE_AUTH&quot;, &quot;version&quot;: &quot;0.0.1&quot;, &quot;source&quot;: &quot;&quot;, &quot;target&quot;: &quot;&quot;, &quot;amount&quot;: 299, &quot;assistant_ID&quot;: 1212, &quot;notes&quot;: null, &quot;quantity&quot;: 1, &quot;branch_ID&quot;: 1002, &quot;item_ID&quot;: 1047, &quot;authorised&quot;: false, &quot;time&quot;: 1482398073000, &quot;message&quot;: &quot;Sale not authorised by policy task MorningBoozeCheckAlt1 for time 10:14:33 CET. Alcohol can not be sold between 00:00:00 CET and 13:00:00 CET or on Sunday&quot; }</code></td>
<td>Request to buy alcohol (<code>item_ID=1047</code>) at <em>10:14:33</em> on <em>Thursday, 22 December 2016</em>. Sale is not authorized. Uses the 'MorningBoozeCheckAlt1' task.</td>
</tr>
<tr>
<td><code>{ &quot;nameSpace&quot;: &quot;com.hyperm&quot;, &quot;name&quot;: &quot;SALE_INPUT&quot;, &quot;version&quot;: &quot;0.0.1&quot;, &quot;time&quot;: 1482077977000, &quot;sale_ID&quot;: 99999982, &quot;amount&quot;: 2199, &quot;item_ID&quot;: 1443, &quot;quantity&quot;: 12, &quot;assistant_ID&quot;: 94, &quot;branch_ID&quot;: 1003, &quot;notes&quot;: &quot;Buy 3, get 1 free!!&quot; }</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Request to buy alcohol (`item_ID=1443`) at _17:19:37_ on _Sunday, 18 December 2016_.
Sale is not authorized. Uses the `MorningBoozeCheckAlt1` task.

Request to buy alcohol (`item_ID=5321`) at _11:13:09_ on _Sunday, 25 December 2016_.
Sale is authorized by policy task `MorningBoozeCheckAlt1` for time 11:13:09 CET.
A request to buy a non-alcoholic item (item_ID=5321) at 11:13:09 on Monday, 2 January 2017. Sale is authorized. Uses the `MorningBoozeCheckAlt1` task.

---

An equivalent version of the `MyFirstPolicyModel` policy model can again be generated using the APEX CLI editor. A sample APEX CLI script is shown below:

```
model create name=MyFirstPolicyModel version=0.0.1 uuid=540226fb-55ee-4f0e-a444-983a0494818e
description="This is my first Apex Policy Model."
```

```
schema create name=assistant_ID_type version=0.0.1 uuid=36df4c71-9616-4206-8b53-976a5cd4bd87
description="A type for 'assistant_ID' values" flavour=Java schema=java.lang.Long
```

```
schema create name=authorised_type version=0.0.1 uuid=d48b619e-d00d-4008-b884-02d76ea4350b
description="A type for 'authorised' values" flavour=Java schema=java.lang.Boolean
```

```
schema create name=branch_ID_type version=0.0.1 uuid=6468845f-4122-4128-8e49-0f52c26078b5
description="A type for 'branch_ID' values" flavour=Java schema=java.lang.Long
```

```
schema create name=item_ID_type version=0.0.1 uuid=4f227ff1-aee0-453a-b6b6-9a4b2e0da932
description="A type for 'item_ID' values" flavour=Java schema=java.lang.Long
```
task create name=MorningBoozeCheck version=0.0.1 uid=3351b0f4-cf06-4fa2-8823-edf67bd30223 description=LS
This task checks if the sales request is for an item that contains alcohol. If the local time is between 00:00:00 and 11:30:00 then the sale is not authorised. Otherwise the sale is authorised. In this implementation we assume that all items with item_ID values between 1000 and 2000 contain alcohol :-) LE
task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=sale_ID schemaName=sale_ID_type schemaVersion=0.0.1
fieldName=amount schemaName=price_type schemaVersion=0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=assistant_ID schemaName=assistant_ID_type schemaVersion=0.1 optional=true task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=notes schemaName=notes_type schemaVersion=0.0.1
fieldName=quantity schemaName=quantity_type schemaVersion=0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=item_ID schemaName=item_ID_type schemaVersion=0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=branch_ID schemaName=branch_ID_type schemaVersion=0.1 task inputfield create name=MorningBoozeCheck version=0.0.1 fieldName=time schemaName=timestamp_type schemaVersion=0.1
fieldName=message schemaName=message_type schemaVersion=0.1 optional=true task logic create name=MorningBoozeCheck version=0.0.1 logicFlavour=MVEL logic=LS /* *
===============================================LICENSE START=======================================================================
* Copyright © 2016-2018 Ericsson. All rights reserved. *
*==============================================================================LICENSE END==================================================================================*/
outFields.put("amount", inFields.get("amount")); outFields.put("assistant_ID", inFields.get("assistant_ID"));
outFields.put("notes", inFields.get("notes")); outFields.put("quantity", inFields.get("quantity"));
outFields.put("branch_ID", inFields.get("branch_ID")); outFields.put("item_ID", inFields.get("item_ID"));
outFields.put("time", inFields.get("time")); outFields.put("sale_ID", inFields.get("sale_ID"));

item_id = inFields.get("item_ID");

gmt = TimeZone.getTimeZone("GMT");
timenow = Calendar.getInstance(gmt);
df = new SimpleDateFormat("HH:mm:ss z");
df.setTimeZone(gmt);
timenow.setTimeInMillis(inFields.get("time"));

midnight = timenow.clone(); midnight.set(timenow.get(Calendar.YEAR),timenow.get(Calendar.MONTH),
timenow.get(Calendar.DATE),0,0,0);

eleven30 = timenow.clone();

if(item_id != null && item_id >=1000 && item_id < 2000) itemisalcohol = true;

if( itemisalcohol && timenow.after(midnight) && timenow.before(eleven30)){
    outFields.put("authorised", false);
    outFields.put("message", "Sale not authorised by policy task " + subject.taskName + " for time " + df.format(timenow.getTime()) + ". Alcohol can not be sold between " + df.format(midnight.getTime()) + " and " + df.format(eleven30.getTime()) + "); return true;
} else{
    outFields.put("authorised", true);
    outFields.put("message", "Sale authorised by policy task " + subject.taskName + " for time " + df.format(timenow.getTime())); return true;
}

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 GMT and 11:30:00 GMT then the sale is not authorised. Otherwise the sale is authorised. In this implementation we assume that items with item_ID value between 1000 and 2000 are all alcoholic drinks :-) */
logger.info("Task Execution: ":subject.id"). Input Event: ":inFields"");
outFields.put("amount" , inFields.get("amount")); outFields.put("assistant_ID", inFields.get("assistant_ID"));
outFields.put("notes" , inFields.get("notes")); outFields.put("quantity" , inFields.get("quantity"));
outFields.put("branch_ID" , inFields.get("branch_ID")); outFields.put("item_ID" , inFields.get("item_ID"));
outFields.put("time" , inFields.get("time")); outFields.put("sale_ID" , inFields.get("sale_ID"));

item_id = inFields.get("item_ID");

cet = TimeZone.getTimeZone("CET"); timenow = Calendar.getInstance(cet); df = new SimpleDateFormat("HH:mm:ss z"); df.setTimeZone(cet); timenow.setTimeInMillis(inFields.get("time"));

midnight = timenow.clone(); midnight.set( timenow.get(Calendar.YEAR),timenow.get(Calendar.MONTH),
timenow.get(Calendar.DATE),0,0,0); onepm = timenow.clone(); onepm.set( timenow.get(Calendar.YEAR),timenow.get(Calendar.MONTH), timenow.get(Calendar.DATE),13,0,0);

itemisalcohol = false; if(item_id != null && item_id >=1000 && item_id < 2000) itemisalcohol = true;

if( itemisalcohol && ( (timenow.after(midnight) && timenow.before(onepm)) ||
(timenow.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) )){ outFields.put("authorised", false);
outFields.put("message", "Sale not authorised by policy task "subject.taskName " for time "+df.format(timenow.getTime()) +" or on Sunday"); return true; } else{ outFields.put("authorised", true);
outFields.put("message", "Sale authorised by policy task "+subject.taskName " for time "+df.format(timenow.getTime()) +" or on Sunday"); return true; }

/* This task checks if a sale request is for an item that is an alcoholic drink. If the local time is between 00:00:00 CET and 13:00:00 CET then the sale is not authorised. Also alcohol sales are not allowed on Sundays. Otherwise the sale is authorised. In this implementation we assume that items with item_ID between 1000 and 2000 are all alcoholic drinks :-) */
event create name=SALE_AUTH version=0.0.1 optional=true event parameter create name=SALE_INPUT version=0.0.1
parName=amount schemaName=price_type schemaVersion=0.0.1
event parameter create name=SALE_INPUT version=0.0.1 parName=assistant_ID schemaName=assistant_ID_type
schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=branch_ID schemaName=branch_ID_type
schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=item_ID schemaName=item_ID_type
schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=notes schemaName=notes_type
schemaVersion=0.0.1 optional=true event parameter create name=SALE_INPUT version=0.0.1 parName=sale_ID schemaName=sale_ID_type
schemaVersion=0.0.1 event parameter create name=SALE_INPUT version=0.0.1 parName=time schemaName=timestamp_type
schemaVersion=0.0.1

policy create name=MyFirstPolicy version=0.0.1 uuid=6c5e410f-489a-46ff-964e-982ce6e8b6d0 description="This is my first Apex policy. It checks if a sale should be authorised or not." template=FREEFORM
firstState=BoozeAuthDecide triggerName=SALE_INPUT policy state create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide
defaultTaskName=MorningBoozeCheck defaultTaskVersion=0.0.1 policy state output create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide
outputName=MorningBoozeCheck_Output_Direct eventName=SALE_AUTH nextState=NULL
nextState=UNL
policy state taskref create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide
outputType=DIRECT taskLocalName=MorningBoozeCheckAlt1 taskName=MorningBoozeCheckAlt1 taskVersion=0.0.1
outputType=DIRECT taskLocalName=MorningBoozeCheck taskName=MorningBoozeCheck taskVersion=0.0.1
outputType=DIRECT policy state selecttasklogic create name=MyFirstPolicy version=0.0.1 stateName=BoozeAuthDecide
logicFlavour=JAVASCRIPT logic=LS /* */

var returnValueType = Java.type("java.lang.Boolean"); var returnValue;
/* This task selection logic selects task "MorningBoozeCheck" for branches with 0 ≤ branch_ID<1000 and selects task "MorningBoozeCheckAlt1" for branches with 1000 ≤ branch_ID<2000. Otherwise the default task is selected. In this case the default task is also "MorningBoozeCheck" */

```java
:leveloffset: 2

== APEX Logging
:leveloffset: +1
```

// == LICENSE_START===================================================================================================
// Copyright (C) 2016-2018 Ericsson. All rights reserved.
// This file is licensed under the CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE
// Full license text at https://creativecommons.org/licenses/by/4.0/legalcode
// SPDX-License-Identifier: CC-BY-4.0
// == LICENSE_END=====================================================================================================

// @author Sven van der Meer (sven.van.der.meer@ericsson.com)

== Introduction to APEX Logging

All APEX components make extensive use of logging using the logging façade [SLF4J](https://www.slf4j.org/) with the backend [Logback](https://logback.qos.ch/). Both are used off-the-shelve, so the standard documentation and configuration apply to APEX logging. For details on how to work with logback please see the [logback manual](https://logback.qos.ch/manual/index.html).

The APEX applications is the logback configuration file `$APEX_HOME/etc/logback.xml` (Windows: `%APEX_HOME%\etc\logback.xml`). The logging backend is set to no debug, i.e. logs from the logging framework should be hidden at runtime.

The configurable log levels work as expected:

- `__ERROR__` (or `__ERROR__) is used for serious errors in the APEX runtime engine
- `__WARN__` (or `__WARN__) is used for warnings, which in general can be ignored but might indicate some deeper problems
- `__INFO__` (or `__INFO__) is used to provide generally interesting messages for startup and policy execution
- `__DEBUG__` (or `__DEBUG__) provides more details on startup and policy execution
- `__TRACE__` (or `__TRACE__) gives full details on every aspect of the APEX engine from start to end

The loggers can also be configured as expected. The standard configuration (after installing APEX) uses log level `__INFO__` on all APEX classes (components).

The applications and scripts in `$APEX_HOME/bin` (Windows: `%APEX_HOME%\bin`) are configured to use the logback configuration `$APEX_HOME/etc/logback.xml` (Windows: `%APEX_HOME%\etc\logback.xml`). There are multiple ways to use different logback configurations, for instance:

- Maintain multiple configurations in `etc`, for instance a `logback-debug.xml` for deep debugging and a `logback-production.xml` for APEX in production mode, then copy the required configuration file to the used `logback.xml` prior starting APEX
- Edit the scripts in `bin` to use a different logback configuration file (only recommended if you are familiar with editing bash scripts or windows batch files)

:leveloffset: 2
:leveloffset: +1
== Standard Logging Configuration

The standard logging configuration defines a context __APEX__, which is used in the standard output pattern.
The location for log files is defined in the property `VAR_LOG` and set to `/var/log/onap/policy/apex-pdp`.
The standard status listener is set to __NOP__ and the overall logback configuration is set to no debug.

```
<configuration debug="false">
  <statusListener class="ch.qos.logback.core.status.NopStatusListener" />

  <contextName>APEX</contextName>
  <property name="VAR_LOG" value="/var/log/onap/policy/apex-pdp/" />

  ...appenders
  ...loggers
</configuration>
```

The first appender defined is called `STDOUT` for logs to standard out.

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <Pattern>%d %contextName[%t] %level %logger{36} - %msg%n</Pattern>
  </encoder>
</appender>
```

The root level logger then is set to the level __info__ using the standard out appender.

```
<root level="info">
  <appender-ref ref="STDOUT" />
</root>
```

The second appender is called `FILE`.
It writes logs to a file `apex.log`.

```
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <file>${VAR_LOG}/apex.log
  <encoder>
    <pattern>%d %-5relative [procId=${processId}] [%thread] %-5level %logger{26} - %msg %n %ex{full}</pattern>
  </encoder>
</appender>
```
The third appender is called `CTXT_FILE`. It writes logs to a file `apex_ctxt.log`.

```xml
<appender name="CTXT_FILE" class="ch.qos.logback.core.FileAppender">  
  <file>${VAR_LOG}/apex_ctxt.log</file> 
  <encoder> 
    <pattern>%d %-5relative [procId=${processId}] [%thread] %-5level %logger{26} - %msg %n %ex{full}</pattern> 
  </encoder> 
</appender>
```

The last definitions are for specific loggers. The first logger captures all standard APEX classes. It is configured for log level __info__ and uses the standard output and file appenders. The second logger captures APEX context classes responsible for context monitoring. It is configured for log level __trace__ and uses the context file appender.

```xml
<logger name="org.onap.policy.apex" level="info" additivity="false"> 
  <appender-ref ref="STDOUT" /> 
  <appender-ref ref="FILE" /> 
</logger>
```

```xml
<logger name="org.onap.policy.apex.core.context.monitoring" level="TRACE" additivity="false"> 
  <appender-ref ref="CTXT_FILE" /> 
</logger>
```

---

== Adding Logback Status and Debug

To activate logback status messages change the status listener from 'NOP' to for instance console.

```xml
<statusListener class="ch.qos.logback.core.status.OnConsoleStatusListener"/>
```

To activate all logback debugging, for instance to debug a new logback configuration, activate the debug attribute in the configuration.

```xml
<configuration debug="true"> ... </configuration>
```
== Logging External Components

Logback can also be configured to log any other, external components APEX is using, if they are using the common logging framework.

For instance, the context component of APEX is using __Infinispan__ and one can add a logger for this external component.
The following example adds a logger for __Infinispan__ using the standard output appender.

[source%nowrap,xml]

```xml
<logger name="org.infinispan" level="INFO" additivity="false"> <appender-ref ref="STDOUT" /> </logger>
```

Another example is Apache Zookeeper.
The following example adds a logger for Zookeeper using the standard output appenders.

[source%nowrap,xml]

```xml
<logger name="org.apache.zookeeper.ClientCnxn" level="INFO" additivity="false"> <appender-ref ref="STDOUT" /> </logger>
```
== Configuring loggers for Policy Logic

The logging for the logic inside a policy (task logic, task selection logic, state finalizer logic) can be configured separate from standard logging.

The logger for policy logic is `org.onap.policy.apex.executionlogging`.

The following example defines
- a new appender for standard out using a very simple pattern (simply the actual message)
- a logger for policy logic to standard out using the new appender and the already described file appender.

[source%nowrap,xml]

```xml
<appender name="POLICY_APPENDER_STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
  <encoder>
    <pattern>policy: %msg\n</pattern>
  </encoder>
</appender>

<logger name="org.onap.policy.apex.executionlogging" level="info" additivity="false">  
  <appender-ref ref="POLICY_APPENDER_STDOUT" />
  <appender-ref ref="FILE" />
</logger>
```

It is also possible to use specific logging for parts of policy logic.

The following example defines a logger for task logic.

[source%nowrap,xml]

```xml
<logger name="org.onap.policy.apex.executionlogging.TaskExecutionLogging" level="TRACE" additivity="false">  
  <appender-ref ref="POLICY_APPENDER_STDOUT" />
</logger>
```
== Rolling File Appenders

Rolling file appenders are a good option for more complex logging of a production or complex testing APEX installation. The standard logback configuration can be used for these use cases. This section gives two examples for the standard logging and for context logging.

First the standard logging.
The following example defines a rolling file appender. The appender rolls over on a daily basis. It allows for a file size of 100 MB.

[source%nowrap,xml]
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
<file>${VAR_LOG}/apex.log</file>  
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"> <!-- rollover daily -->  
<fileNamePattern>xstream-%d{yyyy-MM-dd}.%i.txt</fileNamePattern>  
<maxHistory>4</maxHistory>  
<timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"> <!-- or whenever the file size reaches 100MB -->  
<maxFileSize>100MB</maxFileSize>  
</timeBasedFileNamingAndTriggeringPolicy> </rollingPolicy>  
<encoder>  
<pattern> %d %-5relative [procId=${processId}] [%thread] %-5level %logger{26} - %msg %ex{full} %n </pattern>  
</encoder> </appender>

A very similar configuration can be used for a rolling file appender logging APEX context.

[source%nowrap,xml]
<appender name="CTXT-FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
<file>${VAR_LOG}/apex_ctxt.log</file>  
<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
<fileNamePattern>${VAR_LOG}/apex_ctxt_%d{yyyy-MM-dd}.%i.log.gz</fileNamePattern>  
<maxHistory>4</maxHistory>  
<timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"> <!-- or whenever the file size reaches 100MB -->  
<maxFileSize>100MB</maxFileSize>  
</timeBasedFileNamingAndTriggeringPolicy> </rollingPolicy>  
<encoder>  
<pattern> %d %-5relative [procId=${processId}] [%thread] %-5level %logger{26} - %msg %ex{full} %n </pattern>  
</encoder> </appender>
== Example Configuration for Logging Logic

The following example shows a configuration that logs policy logic to standard out and a file (\_\_info\_\__). All other APEX components are logging to a file (\_\_debug\_\__).

This configuration can be used in a pre-production phase with the APEX engine still running in a separate terminal to monitor policy execution.

This logback configuration is in the APEX installation as `etc/logback-logic.xml`.

```
<configuration debug="false">
  <statusListener class="ch.qos.logback.core.status.NopStatusListener" />

  <contextName>Apex</contextName>
  <property name="VAR_LOG" value="/var/log/onap/policy/apex-pdp/" />

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <Pattern>%d %contextName [%t] %level %logger{36} - %msg%n</Pattern>
    </encoder>
  </appender>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${VAR_LOG}/apex.log</file>
    <encoder>
      <pattern>%d %-5relative [procId=${processId}] [%thread] %-5level%logger{26} - %msg %n %ex{full}</pattern>
    </encoder>
  </appender>

  <appender name="POLICY_APPENDER_STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>policy: %msg\n</pattern>
    </encoder>
  </appender>
</configuration>
```
== Example Configuration for a Production Server

The following example shows a configuration that logs all APEX components, including policy logic, to a file (`debug`).

This configuration can be used in a production phase with the APEX engine being executed as a service on a system without console output.

This logback configuration is in the APEX installation as `logback-server.xml`.

```xml
<configuration debug="false">
  <statusListener class="ch.qos.logback.core.status.NopStatusListener" />
  <contextName>Apex</contextName>
  <property name="VAR_LOG" value="/var/log/onap/policy/apex-pdp/" />
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>${VAR_LOG}/apex.log</file>
    <encoder>
      <pattern>
        %d %-5relative [procId=${processId}] [%thread] %-5level%logger{26} - %msg %n %ex{full}
      </pattern>
    </encoder>
  </appender>
</configuration>
```
== Building a System with Websocket Backend

== Websockets

Websocket is a protocol to run sockets of HTTP. Since it in essence a socket, the connection is realized between a server (waiting for connections) and a client (connecting to a server). Server/client separation is only important for connection establishment, once connected, everyone can send/receive on the same socket (as any standard socket would allow).

Standard Websocket implementations are simple, no publish/subscribe and no special event handling. Most servers simply send all incoming messages to all connections. There is a PubSub definition on top of Websocket called link:http://wamp-proto.org/[WAMP]. APEX does not support WAMP at the moment.
In Java, [JSR 356](http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html) defines the standard Websocket API. This JSR is part of Java EE 7 standard. For Java SE, several implementations exist in open source. Since Websockets are a stable standard and simple, most implementations are stable and ready to use. A lot of products support Websockets, like Spring, JBoss, Netty, … there are also Kafka extensions for Websockets.

== Websocket Example Code for Websocket clients (FOSS)

There are a lot of implementations and examples available on Github for Websocket clients. If one is using Java EE 7, then one can also use the native Websocket implementation. Good examples for clients using simply Java SE are here:

- [Websocket implementation](https://github.com/TooTallNate/Java-WebSocket)
- [Websocket sending client example, using AWT](https://github.com/TooTallNate/Java-WebSocket/blob/master/src/main/example/ChatClient.java)
- [Websocket receiving client example (simple echo client)](https://github.com/TooTallNate/Java-WebSocket/blob/master/src/main/example/ExampleClient.java)

For Java EE, the native Websocket API is explained here:

- [Oracle docs](http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html)
- [An example](http://www.programmingforliving.com/2013/08/jsr-356-java-api-for-websocket-client-api.html)

== BCP: Websocket Configuration

The probably best is to configure APEX for Websocket servers for input (ingress, consume) and output (egress, produce) interfaces. This means that APEX will start Websocket servers on named ports and wait for clients to connect.
Advantage: once APEX is running all connectivity infrastructure is running as well.
Consequence: if APEX is not running, everyone else is in the dark, too.

The best protocol to be used is JSON string.
Each event on any interface is then a string with a JSON encoding.
JSON string is a little bit slower than byte code, but we doubt that this will be noticeable.
A further advantage of JSON strings over Websockets with APEX starting the servers: it is very easy to
connect web browsers to such a system.
Simple connect the web browser to the APEX sockets and send/read JSON strings.

Once APEX is started you simply connect Websocket clients to it, and send/receive event.
When APEX is terminated, the Websocket servers go down, and the clients will be disconnected.
APEX does not (yet) support auto-client reconnect nor WAMP, so clients might need to be restarted or
reconnected manually after an APEX boot.

---

== Demo with VPN Policy Model

We assume that you have an APEX installation using the full package, i.e. APEX with all examples, of
version '0.5.6' or higher.
We will use the VPN policy from the APEX examples here.

Now, have the following ready to start the demo:
- 3 terminals on the host where APEX is running (we need 1 for APEX and 1 for each client)
- the events in the file `$/APEX_HOME/examples/events/VPN/SetupEvents.json` open in an editor (we need to
  send those events to APEX)
- the events in the file `$/APEX_HOME/examples/events/VPN/Link09Events.json` open in an editor (we need to
  send those events to APEX)

---
Create a new APEX configuration using the VPN policy model and configuring APEX as discussed above for Websockets.

Copy the following configuration into 
`$APEX_HOME/examples/config/VPN/Ws2WsServerAvroContextJsonEvent.json` (for Windows use `%APEX_HOME%\examples\config\VPN\Ws2WsServerAvroContextJsonEvent.json`):

```json
{
  "engineServiceParameters": {
    "name": "VPNApexEngine",
    "version": "0.0.1",
    "id": 45,
    "instanceCount": 1,
    "deploymentPort": 12345,
    "policyModelFileName": "examples/models/VPN/VPNPolicyModelAvro.json",
    "engineParameters": {
      "executorParameters": {
        "MVEL": {
          "parameterClassName": "org.onap.policy.apex.plugins.executor.mvel.MVELExecutorParameters"
        }
      },
      "contextParameters": {
        "parameterClassName": "org.onap.policy.apex.context.parameters.ContextParameters",
        "schemaParameters": {
          "Avro": {
            "parameterClassName": "org.onap.policy.apex.plugins.context.schema.avro.AvroSchemaHelperParameters"
          }
        }
      },
      "producerCarrierTechnologyParameters": {
        "carrierTechnology": "WEBSOCKET",
        "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
        "parameters": {
          "wsClient": false,
          "port": 42452
        }
      },
      "producerEventProtocolParameters": {
        "eventProtocol": "JSON"
      },
      "consumerCarrierTechnologyParameters": {
        "carrierTechnology": "WEBSOCKET",
        "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
        "parameters": {
          "wsClient": false,
          "port": 42450
        }
      },
      "consumerEventProtocolParameters": {
        "eventProtocol": "JSON"
      }
    }
  },
  "producerCarrierTechnologyParameters": {
    "carrierTechnology": "WEBSOCKET",
    "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
    "parameters": {
      "wsClient": false,
      "port": 42452
    }
  },
  "producerEventProtocolParameters": {
    "eventProtocol": "JSON"
  },
  "consumerCarrierTechnologyParameters": {
    "carrierTechnology": "WEBSOCKET",
    "parameterClassName": "org.onap.policy.apex.plugins.event.carrier.websocket.WEBSOCKETCarrierTechnologyParameters",
    "parameters": {
      "wsClient": false,
      "port": 42450
    }
  },
  "consumerEventProtocolParameters": {
    "eventProtocol": "JSON"
  }
}
```

In a new terminal, start APEX with the new configuration for Websocket-Server ingress/egress:

```bash
#$APEX_HOME/bin/apexEngine.sh -c $APEX_HOME/examples/config/VPN/Ws2WsServerAvroContextJsonEvent.json
```

Wait for APEX to start, it takes a while to create all Websocket servers (about 8 seconds on a standard laptop without cached binaries).

Depending on your log messages, you will see no (some, a lot) log messages. If APEX starts correctly, the last few messages you should see are:

```bash
#$ (APEX_HOME%bin/apexEngine.bat -c %APEX_HOME%\examples\config\VPN\Ws2WsServerAvroContextJsonEvent.json
```
APEX is running in the new terminal and will produce output when the policy is triggered/executed.

:leveloffset: 2

:leveloffset: +1

```
// ============LICENSE_START=======================================================
//  Copyright (C) 2016-2018 Ericsson. All rights reserved.
// ================================================================================
// This file is licensed under the CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE
// Full license text at https://creativecommons.org/licenses/by/4.0/legalcode
// ====
// SPDX-License-Identifier: CC-BY-4.0
// ============LICENSE_END==========================================================
// @author Sven van der Meer (sven.van.der.meer@ericsson.com)
//```

:leveloffset: +1

```
== Run the Websocket Echo Client

The echo client is included in an APEX full installation.
To run the client, open a new shell (Unix, Cygwin) or command prompt (`cmd` on Windows).
Then use the APEX application launcher to start the client.

[IMPORTANT]
APEX engine needs to run first
====
The example assumes that an APEX engine configured for __produce__ carrier technology Websocket and __JSON__ event protocol is executed first.
====
```
```
```
2.17. $APEX_HOME/bin/apexApps.sh ws-echo [args]

| %APEX_HOME%\bin\apexApps.bat ws-echo [args] |

|====================|

Use the following command line arguments for server and port of the Websocket server. The port should be the same as configured in the APEX engine. The server host should be the host on which the APEX engine is running.

- `-p` defines the Websocket port to connect to (defaults to `8887`)
- `-s` defines the host on which a Websocket server is running (defaults to `localhost`)

Let's assume that there is an APEX engine running, configured for produce Websocket carrier technology, as server, for port 42452, with produce event protocol JSON.

If we start the console client on the same host, we can omit the `-s` options. We start the console client as:

```bash
%APEX_HOME%\bin\apexApps.bat ws-echo -p 42452
```

2.18. $APEX_HOME/bin/apexApps.sh ws-echo -p 42452 <1>

| %APEX_HOME%\bin\apexApps.bat ws-echo -p 42452 <2> |

|<1> Start client on Unix or Cygwin |
|<2> Start client on Windows |

Once started successfully, the client will produce the following messages (assuming we used `-p 42452` and an APEX engine is running on `localhost` with the same port:

```
ws-simple-echo: starting simple event echo → server: localhost → port: 42452
```

Once started, the application will simply print out all received events to standard out. Each received event will be prefixed by `---` and suffixed by `====`

```
ws-simple-echo: opened connection to APEX (Web Socket Protocol Handshake)
```
== Run the Websocket Console Client

The console client is included in an APEX full installation.
To run the client, open a new shell (Unix, Cygwin) or command prompt (`cmd` on Windows).
Then use the APEX application launcher to start the client.

[IMPORTANT]
APEX engine needs to run first
====
The example assumes that an APEX engine configured for __consume__ carrier technology Websocket and
__JSON__ event protocol is executed first.
====

[width="100%",options="header",cols="5a,5a"]
|====================
| Unix, Cygwin | Windows |
| [source%nowrap,sh]

2.19. $APEX_HOME/bin/apexApps.sh ws-console [args]

| [source%nowrap,bat]

%APEX_HOME%\bin\apexApps.bat ws-console [args]
Use the following command line arguments for server and port of the Websocket server. The port should be the same as configured in the APEX engine. The server host should be the host on which the APEX engine is running.

- `-p` defines the Websocket port to connect to (defaults to `8887`)
- `-s` defines the host on which a Websocket server is running (defaults to `localhost`)

Let’s assume that there is an APEX engine running, configured for consume Websocket carrier technology, as server, for port 42450, with consume event protocol JSON.

If we start the console client on the same host, we can omit the `-s` options. We start the console client as:

```
2.20. $APEX_HOME/bin/apexApps.sh ws-console -p 42450 <1>
```

```bash
%APEX_HOME%/bin/apexApps.sh ws-console -p 42450 <2>
```

<1> Start client on Unix or Cygwin
<2> Start client on Windows

Once started successfully, the client will produce the following messages (assuming we used `-p 42450` and an APEX engine is running on `localhost` with the same port:

```
ws-simple-console: starting simple event console -→ server: localhost -→ port: 42450

• terminate the application typing 'exit<enter>' or using 'CTRL+C'
• events are created by a non-blank starting line and terminated by a blank line

ws-simple-console: opened connection to APEX (Web Socket Protocol Handshake)
```
Now you have the full system up and running:
- Terminal 1: APEX ready and loaded
- Terminal 2: an echo client, printing received messages produced by the VPN policy
- Terminal 2: a console client, waiting for input on the console (standard in) and sending text to APEX

We started the engine with the VPN policy example.
So all the events we are using now are located in files in the following example directory:

```bash
# $APEX_HOME/examples/events/VPN > %APEX_HOME%/examples/events/VPN
```

To send events, simply copy the content of the event files into Terminal 3 (the console client).
It will read multi-line JSON text and send the events.
So copy the content of `SetupEvents.json` into the client.
APEX will trigger a policy and produce some output, the echo client will also print some events created in the policy.
In Terminal 1 (APEX) you'll see some status messages from the policy as:

```json
```

In Terminal 2 (echo-client) you see the received events, the last two should look like:

```json
ws-simple-echo: received
```
Congratulations, you have triggered a policy in APEX using Websockets, the policy did run through, created events, picked up by the echo-client. Now you can send the Link 09 and Link 10 events, they will trigger the actual VPN policy and some calculations are made.

Let's take the Link 09 events from `Link09Events.json`, copy them all into Terminal 3 (the console). APEX will run the policy (with some status output), and the echo client will receive and print events.

To terminate the applications, simply press `CTRL+C` in Terminal 1 (APEX). This will also terminate the echo-client in Terminal 2. Then type `exit<enter>` in Terminal 3 (or `CTRL+C`) to terminate the console-client.