Jakarta EE in Practice

Filipe Araújo
Nuno Laranjeiro

October 2021

Jakarta EE in Practice

December 2, 2021

Contents

Acronyms

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

EAR Enterprise Application Repository.

EE Enterprise Edition.

EJB Enterprise JavaBean.

IDE Integrated Development Environment.

JAX-WS Java API for XML Web Services.

JDBC Java Database Connectivity.

JEE Jakarta Enterprise Edition.

JMS Java Message Service.

JNDI Java Naming and Directory Interface.

JPA Java Persistence API.

JSP Java Server Page.

JVM Java Virtual Machine.

MOM Message-Oriented Middleware.

ORM Object-Relational Mapping.

REST Representational State Transfer.

SQL Structured Query Language.

URL Uniform Resource Locator.

Preface

Java is currently one of the most important programming languages in use. According to the TIOBE index it has been on top positions for the last two decades. Java knew considerable evolution since its early days and remains as one of the most relevant languages, especially for for web and enterprise development.

For more than a decade we have been teaching students Java EE (now renamed to "Jakarta EE"). One of the most important problems we had to solve in our classes was the lack of good documentation. While having a vibrant community of Java developers, frameworks, and platforms is seemingly good, this fragmentation is also a shortcoming, because we never know exactly to which version of the language/framework/platform a web post refers to. One of the problems for the non-initiated is making a simple hello world application, because doing the apparently simple configuration is often far from being simple. This sort of problem prompted the creation of the blog "Enterprise Application Integration", to make up for the lack of a practical approach that could guide the developer. The blog targets WildFly, a very popular Application Server. The blog, however, grows organically and, like other documentation that we have prepared for students, didn't have a systematic, thorough approach of Java EE WildFly configurations.

In this book we try to cover the limitations of the blog and other available documentation. We go through the most important topics of Jakarta EE (plus Kafka) and present configurations and "Hello World"-kind examples. This book specifically targets students, but also developers having the first contact with the technology.

The book covers the pivotal topics in Jakarta EE: Java Persistence API (JPA), Enterprise JavaBeans (EJBs), SOAP and Representational State Transfer (REST) web services, as well as web applications created with Java Server Pages (JSPs). We also cover messaging with Java Message Service (JMS) and Apache Kafka. The latter is actually not part of Jakarta EE, but seems to be very important these days, it also includes powerful features related to streaming. For this reason, we found it worth of inclusion in this book. We use these technologies to create a complete demo application by the end of the book.

Filipe Araújo

Nuno Laranjeiro

Chapter 1

Getting Started

Goals
– Prepare the local environment to go through this book's examples.
– Setup a container-based environment to go through this book's examples.
– Learn the basic concepts in building software using Maven.

In this chapter, we briefly explain the basic setup needed to go through the various parts of this book, going through the following two alternatives:

1. Local installation on your operating system;

2. Composing Docker containers to create the intended setup.

In case you select option 1 (local installation), please move on to Section 1.1 for the installation procedure. In case you select option 2 (container installation), please go directly to Section 1.2 to for the setup details. This chapter concludes with Section 1.3, which holds basic background concepts regarding Maven, the build tool selected to build and package the projects presented throughout this book.

1.1 Local installation

Java Development Kit

We should begin by installing the Java platform in your system. In our case, we will be using **Java SE Development Kit version 16**. It is highly recommended that the environment variable `JAVA_HOME` is set to the full path of the Java installation directory (e.g., `C:\Program files\Java\jdk-16` in Windows). You should also be able to run commands like `javac` (the java compiler tool) from the command line. For this, your `PATH` variable should include `%JAVA_HOME%\bin` in Windows, or $JAVA_HOME/bin if using macOS or Linux. By the end you should be able to run `javac -version` from a terminal window and this should print out the version of Java you just installed.

Code Editor

You should download a Integrated Development Environment (IDE), which will help you in organizing your code and detect some types of mistakes. Popular IDEs include Eclipse (be sure to download an Enterprise Edition version that should have a name similar to *Eclipse IDE for Enterprise Java and Web Developers*, available at `https://www.eclipse.org/downloads/packages`), Apache NetBeans, or IntelliJ IDEA.

Application Server

The book is centred around Java Enterprise Edition specifications, namely **Jakarta 8**, which you can find implemented in several application servers, like WildFly or Glassfish. In this book we will be using **WildFly 24** to deploy our projects and provide the necessary services. WildFly is an Application Server that supports Jakarta Enterprise Edition (JEE) applications, which one may use to create web applications, web services, and message-oriented applications to name a few possibilities. For this, it supports EJBs, JPA, deployment of Enterprise Application Repository (EAR) and other JEE standards, some of which we cover in this book.

Make sure you install the server in a directory without any special characters like spaces or symbols. We can start up the server by opening a terminal window, moving to the installation directory of WildFly and then executing the following command :

```
bin/standalone.sh -server-config=standalone-full.xml
```

In Windows systems, the `.sh` is replaced by `.bat`. and you should use the backslash (\) instead.

WildFly is modular and can start under different configurations, with more or less components, thus trading speed for support. One may configure WildFly using a number of different options, including direct edition of its files, the use of its web management interface in port 9990, usually in the `localhost`, and a set of command-line tools, which include `jboss-cli.sh`.

Database

Most complex applications have a database to hold their data. We will resort to **PostgreSQL 13** in our examples. Please download and install, making sure you remember your installation password. The installation should be able to conclude without displaying any errors. The PostgreSQL installer includes PgAdmin (an SQL editor), which will allow you to execute queries and visualize objects in the database. In case you already have a PostgreSQL installation and also want to match your password with the one used throughout this book, you may execute the following command in PgAdmin's query tool, which sets the user named `postgres` with the password `MyPass01`. Notice that you may keep your old credentials and quickly modify the examples instead.

```
ALTER USER postgres WITH PASSWORD 'MyPass01'
```

You may be interested in installing the portable version of **PostgreSQL 13**, available at the

site for macOS operating systems. However, in this case you will need to install PgAdmin separately.

Messaging Software

Messaging applications rely on a provider, which allows message exchange between peers in various configurations. In this book we will use ActiveMQ, which is already bundled with WildFly 24 Ẇe also will illustrate how to use Apache Kafka to exchange messages and process streams.

Building and packaging

Finally, to build and package the various examples we will be using **Apache Maven 3**. Indeed, all examples presented in this book are independent from the IDE you choose to implement your code and can be compiled, packaged, and deployed with simple maven commands. All examples can be compiled and packaged using `mvn clean package`, with some cases (where the application is to be deployed on the application server) requiring an additional property, which specifies the destination to where the packaged files should be copied (e.g., `mvn clean package -Doutput=$WILDFLY_HOME/standalone/deployments`). You can now move on to Section 1.3 for a brief description on how Maven works.

Maven can be downloaded from its website. The download simply needs to be unzipped and its executable file (i.e., the *mvn* command) made accessible via the command line, which you can accomplish by configuring your `PATH` environment variable. macOS users may easily install maven via homebrew, by using the commadn `brew install maven`.

1.2 Containerized Installation

Manually installing software often runs into troubles when someone else tries to replicate the same setup. In many cases the installation cannot complete with success on a different laptop for some obscure reason. Furthermore, the process of installing and configuring is usually cumbersome and not prone to generalization, as one goes through multiple menus taking different options, downloading drivers and so on. Hence, in alternative to install everything on your own computer, you may rather use Docker and manage a few containers. We present that alternative in this section.

In this section we present a Docker compose set up with three containers:

- The PostgreSQL database.
- The WildFly application server.
- And a command line with Maven.

We could discard one of the containers somehow, e.g., by mounting a volume with WildFly in the Maven container, but since we already need two containers, adding a third ones comes nearly for free, without any further inconveniences. For this approach to work one has to install Docker.

1.2.1 The `docker-compose.yml`

The general idea is to define a `yml` compose file, `docker-compose.yml`:

```
version: '3.4'

services:
  database:
    image: postgres:13
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: postgres # The PostgreSQL user (useful to connect to the
          database)
      POSTGRES_PASSWORD: My01pass # The PostgreSQL password (useful to connect to
          the database)
      POSTGRES_DB: school # The PostgreSQL default database (automatically
          created at first launch)
  wildfly:
    build:
      context: .
      dockerfile: Dockerfile
    links:
      - database
    ports:
      - 8080:8080
      - 8443:8443
      - 9990:9990
      - 9993:9993
  command-line:
    build:
     context: .
     dockerfile: Dockerfile.2
    command: tail -f /dev/null
    links:
      - database
      - wildfly
    ports:
      - 2181:2181
      - 9092:9092
    volumes:
      - ..:/workspace
```

Listing 1.1: The `docker-compose.yml` file

We can see the three just mentioned services. One of them has the tag `image`, which contains a direct reference to the Docker hub container to download (`image` and `maven`). The other

two have more elaborate Docker files (tag `dockerfile`) that build on basic images to build more elaborate environments, e.g., with z shell configurations in the case of maven or driver and data source configurations in the case of WildFly. These build on a `Dockerfile` and `Dockerfile.2` available in the same directory as this yaml file (the "." part). More about the Docker files ahead. To prevent the `maven` container from finishing, we run an interminable script to read from `/dev/null`.

One can also see that each one of the containers links to the ones before: `wildfly` can access the database container using the network name `database`, while the command-line service (`maven`) can access the previous two. We also added port mappings for the `database` and `wildfly`, as we want to reach the database, and mainly, the WildFly server from our machine browser on the very same ports 8080 and 9990.

In this example, we are setting the environment of the database directly, which is not perfect, because this configuration file provides direct knowledge of the password. It could be better to use environment variables or refer to an external file with the environment configuration. Finally, notice the volume we are mounting in the command-line service: the `..` directory will show up inside the container as `/workspace`; whatever you write starting on the previous directory of the computer will show up in the `/workspace` and vice-versa. The main idea is that one can compile inside the `/workspace` in the container, using Maven, while keeping everything, including the results outside the container. The container may disappear, but the results will stay in one's file system.

1.2.2 The Docker files

We need a special `Dockerfile` because the base `wildfly` container does not do everything we need. We must set up an administrator account and a data source. The administrator part is easy, but the data source not so much. The challenge is that we must first start WildFly, make sure that it is running and—only then—install the data source. Hence, we may resort to a shell script called `wildfly-init-config.sh` for that purpose. The reader may also look in the documentation of the base container for an alternative solution.

```
FROM jboss/base:latest

ARG USER_NAME="main"
ARG USER_PASSWORD="main"

ENV USER_NAME $USER_NAME
ENV USER_PASSWORD $USER_PASSWORD
ENV CONTAINER_IMAGE_VER=v1.0.0

RUN echo $USER_NAME
RUN echo $USER_PASSWORD
RUN echo $CONTAINER_IMAGE_VER

USER root

RUN yum install -y epel-release \
```

```
    && yum install -y java-latest-openjdk \
    curl \
    git-core \
    less \
    passwd \
    sudo \
    vim \
    wget \
    zsh \
    # add a user (--disabled-password: the user won't be able to use the account
        until the password is set)
    && adduser -p ${USER_PASSWORD} --shell /bin/zsh --home /home/$USER_NAME -c
        "User" ${USER_NAME} \
    # update the password
    && echo ${USER_NAME}:${USER_PASSWORD} | chpasswd \
    && usermod -aG wheel ${USER_NAME}

# install WildFly
RUN curl -LO
    https://download.jboss.org/wildfly/24.0.1.Final/wildfly-24.0.1.Final.tar.gz \
 && tar zxvf wildfly-24.0.1.Final.tar.gz \
 && mv wildfly-24.0.1.Final /opt/jboss/wildfly \
 && rm wildfly-24.0.1.Final.tar.gz
RUN /opt/jboss/wildfly/bin/add-user.sh admin admin#7rules --silent
RUN /opt/jboss/wildfly/bin/add-user.sh -a -u ejbclient 'ejb01acceS' --silent -g
    'guest'
RUN /opt/jboss/wildfly/bin/add-user.sh -a -u john '!1secret' --silent -g 'guest'
ADD --chmod=0755 wildfly-init-config.sh /opt/jboss/wildfly/bin
ADD --chmod=0755 configure.cli /opt/jboss/wildfly/bin

USER jboss

CMD [ "/opt/jboss/wildfly/bin/wildfly-init-config.sh" ]
```

Listing 1.2: `Dockerfile`

We opted for a second Docker file for the Maven container to have a more elaborate shell configuration, named Oh my Zsh, based on Z shell and on a user named `main` with sudo capabilities.

```
FROM maven:3-openjdk-16

#From https://gist.github.com/MichalZalecki/4a87880bbe7a3a5428b5aebebaa5cd97
ARG USER_NAME="main"
ARG USER_PASSWORD="main"

ENV USER_NAME $USER_NAME
```

```
ENV USER_PASSWORD $USER_PASSWORD
ENV CONTAINER_IMAGE_VER=v1.0.0

RUN echo $USER_NAME
RUN echo $USER_PASSWORD
RUN echo $CONTAINER_IMAGE_VER

USER root

RUN set -eux \
 && microdnf install curl \
  git-core \
  less \
  passwd \
  sudo \
  vim \
  wget \
  zip \
  zsh \
  && microdnf clean all \
  # add a user (--disabled-password: the user won't be able to use the account
      until the password is set)
  && adduser -p ${USER_PASSWORD} --shell /bin/zsh --home /home/$USER_NAME -c
      "User" ${USER_NAME} \
  # update the password
  && echo ${USER_NAME}:${USER_PASSWORD} | chpasswd \
  && usermod -aG wheel ${USER_NAME}


# install Kafka
RUN curl -O https://archive.apache.org/dist/kafka/2.8.1/kafka_2.13-2.8.1.tgz
RUN tar -xzf kafka_2.13-2.8.1.tgz
RUN mv kafka_2.13-2.8.1 /opt/
RUN chown -R ${USER_NAME}.${USER_NAME} /opt/kafka_2.13-2.8.1/
RUN rm kafka_2.13-2.8.1.tgz

# install jax-ws (wsimport.sh)
RUN curl -LO
    https://repo1.maven.org/maven2/com/sun/xml/ws/jaxws-ri/3.0.2/jaxws-ri-3.0.2.zip
    \
  && unzip jaxws-ri-3.0.2.zip
ENV PATH="/jaxws-ri/bin/:${PATH}"

# the user we're applying this too (otherwise it most likely install for root)
USER $USER_NAME
# terminal colors with xterm
```

```
ENV TERM xterm
# set the zsh theme
ENV ZSH_THEME robbyrussell

# run the installation script
#  RUN wget https://github.com/robbyrussell/oh-my-zsh/raw/master/tools/install.sh
    -O - | zsh || true
RUN sh -c "$(curl -fsSL
    https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"


# start zsh
CMD [ "zsh" ]
```

Listing 1.3: `Dockerfile2`

This Docker file includes the installation of Apache Kafka, which becomes available in the Maven container under the directory `/opt/kafka_version_number`.

1.2.3 The `wildfly-init-config.sh` Script

```
#! /bin/sh

#start wildfly
/opt/jboss/wildfly/bin/standalone.sh -c standalone-full.xml -b 0.0.0.0
    -bmanagement 0.0.0.0 &

#install the driver and the data source
#the challenge is to have the server ready---it may take a while
RC=1
count=0
while [ $RC -ne 0 ] && [ $count -lt 20 ]
do
    sleep 5
    /opt/jboss/wildfly/bin/jboss-cli.sh --connect ""
    RC=$?
    let count++
done

if [ $RC -eq 0 ]
then
    /opt/jboss/wildfly/bin/jboss-cli.sh --connect
        --file=/opt/jboss/wildfly/bin/configure.cli
    #keep running forever
    tail -f /dev/null
fi
```

Listing 1.4: The `wildfly-init-config.sh` file

This script starts WildFly in background making sure that the sockets bind to address 0.0.0.0, to enable connection from our own machine. Then, it checks a limited number of times whether WildFly is running, by connecting to WildFly and doing nothing with `jboss-cli.sh`. Once it succeeds it invokes the same shell script, `jboss-cli.sh`, passing it the `configure.cli` command line interface file, with a number of instructions. These deploy the PostgreSQL Java Database Connectivity (JDBC) driver and install a data source. In addition, we set up security to enable remote connections to EJBs. In the end we just hang the script to prevent the container from finishing using no CPU in the process.

```
# Batch script to enable elytron for the quickstart application in the JBoss EAP
    server and a few more things
# Adapted from https://github.com/wildfly/quickstart/tree/11.x/ejb-security

# Start batching commands
batch

# Deploy the driver
deploy --force --url=https://jdbc.postgresql.org/download/postgresql-42.2.24.jar

# Deploy the data source
data-source add --name=PostgresDS --driver-name=postgresql-42.2.24.jar
    --driver-class=org.postgresql.Driver --jndi-name=java:/PostgresDS
    --connection-url=jdbc:postgresql://database:5432/school  --user-name=postgres
    --password=My01pass

# Add security domain mapping in the EJB3 subsystem to enable elytron for the
    quickstart EJBs
/subsystem=ejb3/application-security-domain=
    other:add(security-domain=ApplicationDomain)

# Update the http-remoting-connector to use the application-sasl-authentication
    factory
/subsystem=remoting/http-connector= http-remoting-connector:write-attribute(name=
    sasl-authentication-factory,value=application-sasl-authentication)

# JMS
jms-queue add --queue-address=playQueue
    --entries=java:jboss/exported/jms/queue/PlayQueue
jms-topic add --topic-address=playTopic
    --entries=java:jboss/exported/jms/topic/playTopic
/subsystem=messaging-activemq/server=
    default/security-setting=#/role=guest:remove()
/subsystem=messaging-activemq/server=
```

Figure 1.1: Welcome message on well configured WildFly

```
    default/security-setting=#/role=guest:add(consume=true,
    create-durable-queue=true,delete-durable-queue=true,
    create-non-durable-queue=true,delete-non-durable-queue=true,send=true)

# Run the batch commands
run-batch

# Reload the server configuration
reload
```

Listing 1.5: The `configure.cli` file

`Adding Everything Up` We now need to run a couple of commands on the command line. One to build the images and run the containers:

```
docker-compose up --build
```

This may take a while depending on your network speed. Once this finishes, we may try the addresses localhost:8080 and localhost:9990 on the browser, to get the following results, starting from the 8080 port:

The port 9990 will allow us to see the driver:

And the data source that passes the connection test to the PostgreSQL database:

The next command is optional and allows us to enter the container that has Maven, such that we can run Maven itself. In the end, this was what we were looking for, a way of

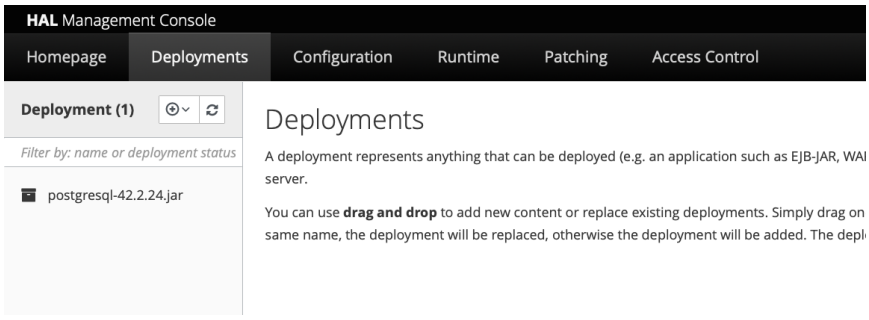

Figure 1.2: The PostgreSQL driver in WildFly

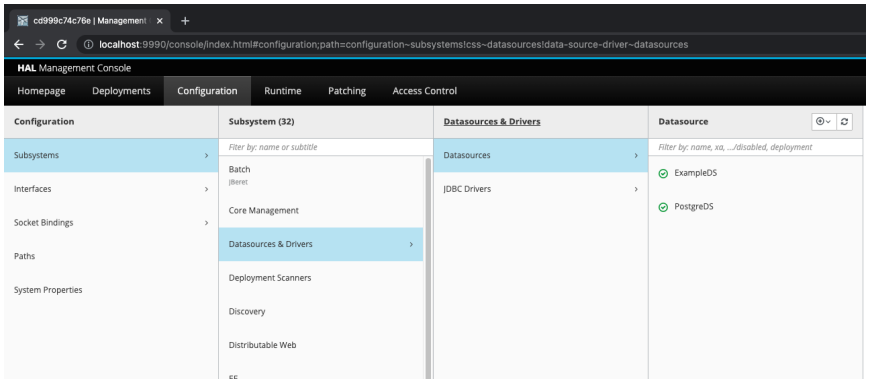

Figure 1.3: The data source in WildFly

running Maven with minimal installation of software on our own operating system. To do this, we start by looking for the identifiers of the running containers:

```
docker ps
```

Then, we look for the identifier of the maven container, for example `4a229ca55f65`. One has to replace this identifier by the real one provided by the `ps` command, to enter the container:

```
docker exec -it 4a229ca55f65 /bin/sh
```

We may now run Maven having a view of the parent directory `..` inside this container under the name `workspace`:

```
sh-4.4# ls workspace/
EJB-client  EJB-server  JMS  JPA-standalone  KafkaStreams  LICENSE  README.md
    application  kafka  local-maven-repo
```

Another very important feature is to look at the logs, specially of WildFly, as these often give precious indications of what is wrong. For example, a shortcoming of this setup concerns Java versions, as the `maven` container may be ready for a Java version that is different from the WildFly Java version. In a more complex example, Maven may output some ambiguous error message, that the logs immediately expose. To look at the logs we may use the following command:

```
docker logs d7882574eb38
```

being `d7882574eb38` the WildFly Docker container.

1.3 Maven Fundamentals

Maven allows you to manage your project so that you can specify what should happen during typical stages like compilation, deployment, testing, or documentation generation. Among several other aspects, the tool relies on best practices like separating test code to a parallel structure in the project or using naming conventions to locate tests. It also enforces guidelines for laying out the project structure and placing certain resources in particular locations, by convention. Among its features, we highlight the dependency management, where project dependencies (e.g., libraries) are identified by group, name, and version and are not kept part of the project itself.

1.3.1 Project Lifecycles

Maven is centred around three build lifecycles: clean, default, and site. Each lifecycle is composed of several phases. For instance, clean is composed of pre-clean, clean, and post-clean (at each phase different tasks are executed). A build phase is composed of plugin goals, which represent specific tasks to be carried out that allow completing the phase. The default lifecycle includes the following main phases (among others):

- **compile**: the source code is compiled.

- **test**: unit tests are executed against the compiled code.
- **package**: the compiled code is packaged, e.g., in a JAR or WAR file.
- **install**: the package is installed into a local repository, to be used by other projects.
- **deploy**: the package is copied to a remote repository.

At the command line, we can run `mvn compile` to compile source code, or `mvn package` to package compiled code. We may also target phases of different lifecycles, such as in `mvn clean package`, which will run all phases of the *clean lifecycle* (i.e., it will run pre-clean, clean, and post-clean), and will then run all phases of the *default lifecycle* up until the package phase. It is also possible to run a specific goal of a given build phase.

1.3.2 Creating a simple project

We can use the `mvn` command at the terminal to create a basic structure for a project. For this purpose, depending on the type of project we wish to create, we use a Maven Archetype, which is essentially a model that represents projects of a certain type and. In this sense it can be seen as a project template. The following command will use a Maven archetype to generate the directory structure for a simple Java project.

```
mvn archetype:generate  -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-simple -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=MySimpleApp -Dversion=1.0-SNAPSHOT
    -DinteractiveMode=false
```

Listing 1.6: Maven command for creating a simple Java project.

In practice, we must define the groupId, which identifies the project (e.g., like `org.apache.maven`, or `book` in this case). We must also define the artifactId, which will serve as base name for packaging the project (e.g., it will be used to name the respective JAR or WAR file), and a version number which will also be used (by default) in the package name. In the case of this example, the packaged project will be named `MySimpleApp-1.0-SNAPSHOT.jar`.

The project creation command in Listing 1.6 generates a default `pom.xml` which holds the following contents.

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>book</groupId>
  <artifactId>MySimpleApp</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
  <name>MySimpleApp</name>
  <description>A simple MySimpleApp.</description>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.7.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-project-info-reports-plugin</artifactId>
          <version>3.0.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
```

```
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>2.5.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
          <version>2.8.2</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <reporting>
    <plugins>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Listing 1.7: The generated project descriptor file.

The generated `pom.xml` file has a few main parts, of which we highlight the project identifier (i.e., groupId), artifact, and version identification. We can also see properties that specify file encoding or compiler configuration. Next we find project dependencies, which, in this case, is only junit (which will be used to run unit tests). The element `build` includes several plugins, which can also hold specific configuration. For instance, the jar-plugin can be configured to include or exclude certain content, or to include a `MANIFEST.MF` file holding specific content. You may find detailed information at the Maven project website.

The project creation command in Listing 1.6 will also generate a basic project directory structure, similar to the following one.

The root of the project holds the `pom.xml` and a single directory `src` before compilation. The build process will also generate a `target` directory (at the root of the project), which will hold all output of the build. Under `src`, we find `main`, `site`, and `test` respectively for the main project artifacts, the project site documentation, and test artifacts. Under `src/main` we will place source code (in the `java` directory); resource files in `resources`; filters that should be applied to the resource files in `filters`; and web application files like JSP, HTML, or JavaScript files under `webapp`, in case the project is a web application. The `src/test` directory mimics the `src/main` directory contents and is intended to hold code for testing the artifacts present under `java` (e.g., unit tests at `src/test/java` or test resource files at `src/test/resources`. A simple application will hold at least the `pom` file and

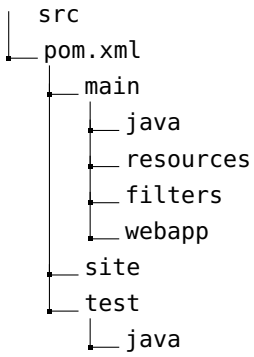

Figure 1.4: Simplified maven project directory structure.

a `src/main/java` directory, which will include the project source code.

Chapter 2

Java Persistence API

Goals
Learn how to use the Java Persistence API to handle storage requirements.

JPA is a standard that enables Java programs to persist objects into relational databases. In simple terms, it maps Java objects to their corresponding entries in relational database tables, thus persisting objects in the database. JPA is able to keep track of object references in memory, by means of some specific annotations in the source code.

To exemplify the use of JPA, we will use a football team and respective players. Each team has a name, address, and a chairman. Teams have players with a name, a birth date, and height. This is a one-to-many relation, as teams have many players and each player plays in a single team (if a player could also play in more than one team, the relation would become many-to-many). We use JPA annotations to express this kind of relationship, but, since JPA is only an Application Programming Interface (API), which does not actually implement the persistence functionality, we use Hibernate for the implementation. Hibernate is a framework for data persistence, which complies with the JPA specification and also provides access to custom functionality, usually via Hibernate specific annotations. Nonetheless, in this book, we use JPA annotations as the interface and relegate Hibernate to the implementation layer. The database we use in this book is PostgreSQL. One should notice that the programs we develop in this chapter are stand-alone, as they have a `main` method allowing them to run alone and have direct interaction with users.

2.1 Source Code

We begin by creating a project structure with the following command.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
    -DgroupId=jpaprimer -DartifactId=JPA-standalone -Dversion=1
    -DinteractiveMode=false
```

Listing 2.1: Maven archetype for a stand-alone application.

Replace the generated `pom.xml` with the following one:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.
    apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>book</groupId>
  <artifactId>JPA-standalone</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.3.20.Final</version>
    </dependency>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.2.24</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
```

```
            <phase>prepare-package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${project.build.directory}/lib</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <classpathPrefix>lib/</classpathPrefix>
              <mainClass>jpaprimer.WriteData</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 2.2: `pom.xml`.

The Team Entity can go like this:

```
package jpaprimer.data;


import java.io.Serializable;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Team implements Serializable {
  private static final long serialVersionUID = 1L;
```

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
int id;

private String name;

private String address;

private String chairman;

@OneToMany(mappedBy="team")
private List<Player> players;


public Team() {
  super();
}

public Team(String name, String address, String presidentname) {
  super();
  this.name = name;
  this.address = address;
  this.chairman = presidentname;
}

public String getName() {
  return name;
}
public void setName(String name) {
  this.name = name;
}
public String getAddress() {
  return address;
}
public void setAddress(String address) {
  this.address = address;
}
public String getPresidentname() {
  return chairman;
}
public void setPresidentname(String presidentname) {
  this.chairman = presidentname;
}
public List<Player> getPlayers() {
  return players;
```

```
  }
  public void setPlayers(List<Player> players) {
    this.players = players;
  }

}
```

Listing 2.3: `Team.java`.

Next, the Player Entity:

```
package jpaprimer.data;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

/**
 * Entity implementation class for Entity: Player
 *
 */
@Entity
public class Player implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int id;

  private String name;

  @Temporal(TemporalType.DATE)
  private Date birth;

  private float height;

  @ManyToOne
  private Team team;

  public Player() {
```

```
    super();
  }

  public Player(String name, Date birth, float height, Team team) {
    super();
    this.name = name;
    this.birth = birth;
    this.height = height;
    this.team = team;
  }


  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Date getBirth() {
    return birth;
  }

  public void setBirth(Date birth) {
    this.birth = birth;
  }

  public float getHeight() {
    return height;
  }

  public void setHeight(float height) {
    this.height = height;
  }

  public Team getTeam() {
    return team;
```

```
  }

  public void setTeam(Team team) {
    this.team = team;
  }

  public static long getSerialversionuid() {
    return serialVersionUID;
  }

  @Override
  public String toString() {
    return this.name + " id = " + this.id + ", " + this.height + " plays for " +
        this.team.getName() + ". Born on " + this.birth;
  }

}
```

Listing 2.4: `Player.java`.

Before putting the whole thing to run, we need a couple more things. A `persistence.xml` file that must go to the `src/main/resources/META-INF` directory. Just create this directory tree if you don't have any resources directory. Don't forget to adjust the name of the database, the user name and the password. Also to notice is that you will need to set the right host in the `javax.persistence.jdbc.url` property. Currently `localhost` is set, but if you are using a containerized installation, you must replace `localhost` with `database`. If you are using another database, like MySQL, you will need to adjust a number of parameters—nothing exceedingly complicated. The file looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="Players">
    <properties>
      <property name="javax.persistence.jdbc.user"
        value="postgres" />
      <property name="javax.persistence.jdbc.password"
        value="My01pass" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.PostgreSQLDialect" />
      <!-- table generation policies: validate, update, create, create-drop -->
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="javax.persistence.jdbc.driver"
```

```
        value="org.postgresql.Driver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:postgresql://localhost/playersandteams" />
    </properties>
  </persistence-unit>
</persistence>
```

Listing 2.5: `persistence.xml`.

To see the whole thing in action, we need a dummy main application that creates and persists the objects into the database. When one uses JPA on the server side, this application is not necessary, and the EJBs take care of using the entities, persisting them, doing queries, etc. One has to realize that when used in the context of a server, the configuration file `persistence.xml` must change. Refer to Chapter 8 for an example.

```
package jpaprimer;

import java.util.Calendar;
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

import jpaprimer.data.Player;
import jpaprimer.data.Team;

public class WriteData {

  public static Date getDate(int day, int month, int year) {
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR, year);
    cal.set(Calendar.MONTH, month - 1);
    cal.set(Calendar.DAY_OF_MONTH, day);

    Date d = cal.getTime();
    return d;
  }

  public static void main(String[] args) {
    Team[] teams = { new Team("Sporting", "Alvalade", "Silva"), new
        Team("Academica", "Coimbra", "Santos"),
        new Team("Porto", "Dragao", "Moreira"), new Team("Benfica", "Luz",
            "Martins") };
    Player[] players = { new Player("Albino", getDate(23, 4, 1987), 1.87f,
        teams[0]),
```

```
        new Player("Bernardo", getDate(11, 4, 1987), 1.81f, teams[0]),
        new Player("Cesar", getDate(12, 5, 1983), 1.74f, teams[0]),
        new Player("Dionisio", getDate(3, 12, 1992), 1.67f, teams[0]),
        new Player("Eduardo", getDate(31, 8, 1985), 1.89f, teams[0]),
        new Player("Franco", getDate(6, 1, 1989), 1.95f, teams[1]),
        new Player("Gil", getDate(7, 12, 1986), 1.8f, teams[1]),
        new Player("Helder", getDate(14, 5, 1987), 1.81f, teams[1]),
        new Player("Ilidio", getDate(13, 6, 1991), 1.82f, teams[1]),
        new Player("Jacare", getDate(4, 2, 1993), 1.83f, teams[1]),
        new Player("Leandro", getDate(4, 10, 1984), 1.81f, teams[2]),
        new Player("Mauricio", getDate(3, 6, 1984), 1.8f, teams[2]),
        new Player("Nilton", getDate(11, 3, 1985), 1.88f, teams[2]),
        new Player("Oseias", getDate(23, 11, 1990), 1.74f, teams[2]),
        new Player("Paulino", getDate(14, 9, 1986), 1.75f, teams[2]),
        new Player("Quevedo", getDate(10, 10, 1987), 1.77f, teams[2]),
        new Player("Renato", getDate(7, 7, 1991), 1.71f, teams[3]),
        new Player("Saul", getDate(13, 7, 1992), 1.86f, teams[3]),
        new Player("Telmo", getDate(4, 1, 1981), 1.88f, teams[3]),
        new Player("Ulisses", getDate(29, 8, 1988), 1.84f, teams[3]),
        new Player("Vasco", getDate(16, 5, 1988), 1.83f, teams[3]),
        new Player("X", getDate(8, 12, 1990), 1.82f, teams[3]),
        new Player("Ze", getDate(13, 5, 1987), 1.93f, teams[3]), };

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("Players");
    EntityManager em = emf.createEntityManager();
    EntityTransaction trx = em.getTransaction();

    trx.begin();
    for (Team t : teams)
    {
      em.persist(t);
    }

    for (Player p : players)
    {
      em.persist(p);
    }
    trx.commit();

    em.close();
    emf.close();
  }

}
```

Listing 2.6: `WriteData.java`.

2.2 Filling the Database

We must create the database and make it ready to receive the data from our program. We assume that the database is already up and running and that the administrator has the ability to log into the database. To access the database one may use any client from a very large set of available programs, including `pgAdmin`, which is bundled with the PostgreSQL installation. In this case, we may use the graphical user interface of `pgAdmin`, to create a database named `playersandteams`, prior to running the example of this chapter. Note the name `playersandteams`, which matches the suffix of the connection Uniform Resource Locator (URL) in the `persistence.xml` file, `jdbc:postgresql://database/playersandteams`.

On the other hand, you can also use the command line. In the following example, we assume that PostgreSQL is running inside a container (as described in Chapter 1, where we also explain how to enter the container). In this case, the command line client is available in the same container as follows (`postgres` is the name of the user):

```
psql -U postgres
```

Listing 2.7: Running PostgreSQL client.

Once connected to the server, the interaction goes as follows:

```
    postgres=# create database playersandteams;
    CREATE DATABASE
    postgres=# \c playersandteams;
    You are now connected to database "playersandteams" as user "postgres".
    playersandteams=# \dt
    Did not find any relations.
```

Listing 2.8: Interaction with PostgreSQL to create the database `playersandteams` connect to it and list the tables.

Coming back to the Java context, running the `WriteData` class may be faster and easier within the Integrated Development Environment (IDE), however the command line steps are, as follows. Start by compiling and packaging the code with:

```
mvn clean package
```

Listing 2.9: Create executable package.

then run:

```
java -jar target/JPA-standalone.jar
```

Listing 2.10: Executing the application.

The final name of the `jar` file would be `JPA-standalone-0.0.1-SNAPSHOT-jar-with-dependencies.jar` by default, but our `pom.xml` file specifies that the name of the file resulting from the packaging process should consist only in the artifact identification (see element `<finalName>` in Listing 2.2).

The output of executing the `WriteData` class has a lot of lines, as the Hibernate implementation is printing out the Structured Query Language (SQL):

```
Hibernate: create table Player (id integer not null, birth date, height
float(24) not null, name varchar(255), team_id integer, primary key (id))
Hibernate: create table Team (id integer not null, address varchar(255),
chairman varchar(255), name varchar(255), primary key (id))
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate: alter table if exists Player add constraint
FKqfn7q18rx1dwkwui2tyl30e08 foreign key (team_id) references Team
Hibernate: select nextval('hibernate_sequence')
Hibernate: select nextval('hibernate_sequence')
...
Hibernate: select nextval('hibernate_sequence')
Hibernate: insert into Team (address, chairman, name, id) values (?, ?, ?, ?)
Hibernate: insert into Team (address, chairman, name, id) values (?, ?, ?, ?)
Hibernate: insert into Team (address, chairman, name, id) values (?, ?, ?, ?)
Hibernate: insert into Team (address, chairman, name, id) values (?, ?, ?, ?)
Hibernate: insert into Player (birth, height, name, team_id, id) values (?,
?, ?, ?, ?)
Hibernate: insert into Player (birth, height, name, team_id, id) values (?,
?, ?, ?, ?)
Hibernate: insert into Player (birth, height, name, team_id, id) values (?,
?, ?, ?, ?)
...
```

Listing 2.11: SQL output of executing the `WriteData` program.

It is now interesting to take a look at the players and teams in the database. In particular, note the foreign key pointing to the team in the players:

```
playersandteams=# \dt
        List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | player | table | postgres
 public | team   | table | postgres
(2 rows)

playersandteams=# select * from team;
 id | address   | chairman |   name
----+-----------+----------+-----------
  1 | Alvalade  | Silva    | Sporting
  2 | Coimbra   | Santos   | Academica
  3 | Dragao    | Moreira  | Porto
  4 | Luz       | Martins  | Benfica
(4 rows)
```

```
playersandteams=# select * from player;
 id |   birth    | height |   name   | team_id
----+------------+--------+----------+---------
  5 | 1987-04-23 |   1.87 | Albino   |       1
  6 | 1987-04-11 |   1.81 | Bernardo |       1
  7 | 1983-05-12 |   1.74 | Cesar    |       1
  8 | 1992-12-03 |   1.67 | Dionisio |       1
  9 | 1985-08-31 |   1.89 | Eduardo  |       1
 10 | 1989-01-06 |   1.95 | Franco   |       2
 11 | 1986-12-07 |    1.8 | Gil      |       2
 12 | 1987-05-14 |   1.81 | Helder   |       2
 13 | 1991-06-13 |   1.82 | Ilidio   |       2
 14 | 1993-02-04 |   1.83 | Jacare   |       2
 15 | 1984-10-04 |   1.81 | Leandro  |       3
 16 | 1984-06-03 |    1.8 | Mauricio |       3
 17 | 1985-03-11 |   1.88 | Nilton   |       3
 18 | 1990-11-23 |   1.74 | Oseias   |       3
 19 | 1986-09-14 |   1.75 | Paulino  |       3
 20 | 1987-10-10 |   1.77 | Quevedo  |       3
 21 | 1991-07-07 |   1.71 | Renato   |       4
 22 | 1992-07-13 |   1.86 | Saul     |       4
 23 | 1981-01-04 |   1.88 | Telmo    |       4
 24 | 1988-08-29 |   1.84 | Ulisses  |       4
 25 | 1988-05-16 |   1.83 | Vasco    |       4
 26 | 1990-12-08 |   1.82 | X        |       4
 27 | 1987-05-13 |   1.93 | Ze       |       4
(23 rows)
```

Listing 2.12: Contents of the playersandteams database.

2.3 Querying the Database

Can we now use the data to do a query that lists players taller than some height? This is straightforward, as we need to only slightly refine a query to get all the players. The query `from Player p` would return the entire list of players. The `select` in the beginning of the query is implicit and refers to the entire object `p`. We could refine this query to provide, say, the name of the player in the `select`, by doing `select p.name ...`. To restrict the list based on their height, we must do as follows:

```
package jpaprimer;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import jpaprimer.data.Player;

public class QueryData {

  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("Players");
    EntityManager em = emf.createEntityManager();
    TypedQuery<Player> q = em.createQuery("select p from Player p where p.height
        > :height", Player.class);
    q.setParameter("height", 1.85f);

    List<Player> players = q.getResultList();
    if (players.size() > 0)
      for (Player p : players)
        System.out.println(p);

    em.close();
    emf.close();
  }
}
```

Listing 2.13: `QueryData.java` file.

Despite not being in the manifest, running this program is still possible with `java -cp target/JPA-standalone.jar jpaprimer.QueryData`. Besides the usual output of the SQL commands, we should get the following list:

```
Albino id = 5, 1.87 plays for Sporting. Born on 1987-04-23
Eduardo id = 9, 1.89 plays for Sporting. Born on 1985-08-31
Franco id = 10, 1.95 plays for Academica. Born on 1989-01-06
Nilton id = 17, 1.88 plays for Porto. Born on 1985-03-11
Saul id = 22, 1.86 plays for Benfica. Born on 1992-07-13
Telmo id = 23, 1.88 plays for Benfica. Born on 1981-01-04
Ze id = 27, 1.93 plays for Benfica. Born on 1987-05-13
```

Listing 2.14: Players taller than 1.85 meter

Let us now list the players from our favorite team, Academica. The interesting thing to note in this solution is that we are taking advantage of the Object-Relational Mapping (ORM), to pick the team from the database and then just exploring the players field from the Team object. Rather simple! We run this program just as we ran the previous one, with `java -cp target/JPA-standalone.jar jpaprimer.FavoriteTeam`:

```
package jpaprimer;
```

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import jpaprimer.data.Player;
import jpaprimer.data.Team;

public class FavoriteTeam {

  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("Players");
    EntityManager em = emf.createEntityManager();
    TypedQuery<Team> q = em.createQuery("select t from Team t where t.name = :t",
        Team.class);
    q.setParameter("t", "Academica");

    List<Team> resultteams = q.getResultList();
    if (resultteams.size() > 0)
      for (Player p : resultteams.get(0).getPlayers()) {
        System.out.println(p.getName());
      }

    em.close();
    emf.close();
  }

}
```

Listing 2.15: `FavoriteTeam.java` file.

```
    Franco
    Gil
    Helder
    Ilidio
    Jacare
```

Listing 2.16: Favorite team's players.

Chapter 3

Enterprise JavaBeans

Goals
- Learn how to create an Enterprise Java Bean on WildFly.

In this chapter, we will write a basic session EJB server component to run a calculator on the server. This calculator is capable of doing only a single operation: adding two numbers. Not very impressive, indeed, but a good starting point for greater feats. Once we have the server ready, we write a client that can use the server.

3.1 A Session Enterprise JavaBean for WildFly

```
mvn archetype:generate -DarchetypeGroupId=org.codehaus.mojo.archetypes
-DarchetypeArtifactId=ejb-javaee6  -DarchetypeVersion=1.4 -DgroupId=book
-DartifactId=EJB-server -Dversion=1 -DinteractiveMode=false
```

Listing 3.1: Maven archetype for an Enterprise JavaBeans application.

We start by creating and exposing a session EJB. One can do this in a number of different ways, but in this example we will do it through a remote interface, because we are aiming at the simplest possible example, with a stand-alone client. In many cases, we don't need to expose our EJB as remote, with the corresponding security issues, unless client and server actually run in different Java Virtual Machines (JVMs). The remote interface is thus:

```
package ejb;

public interface ICalculator {
  public int add(int a, int b);
}
```

Listing 3.2: The remote interface of our simple calculator (`ICalculator.java`).

To this interface, corresponds the following class:

```java
package ejb;

import javax.ejb.Stateless;
import javax.ejb.Remote;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.jboss.ejb3.annotation.SecurityDomain;
import javax.annotation.security.RolesAllowed;


/**
* Session Bean implementation class Calculator
*/
@Stateless
@Remote(ICalculator.class)
@RolesAllowed({ "guest" })
@SecurityDomain("other")
public class Calculator implements ICalculator {
  Logger logger = LoggerFactory.getLogger(Calculator.class);

  public Calculator() {
    logger.info("Created Calculator");
    logger.debug("Debug!! Created Calculator");
  }

  @Override
  public int add(int a, int b) {
    logger.info("adding " + a + " + " + b);
    logger.debug("Debug!! Adding " + a + " + " + b);
    return a + b;
  }

}
```

Listing 3.3: The calculator `Calculator.java` file.

Since both files belong to the `ejb` package they should be inside a directory with the same name. Notice that you will need the `@RolesAllowed` and `@SecurityDomain` annotations, in case you are running client and server in separate machines or if you are using the docker container configuration. If not, you should now remove these two annotations from the `Calculator.java` file.

For illustration purposes, we added logging to the code, using the "simple logging facade for Java" library. The logger starts at class creation time and runs on construction and inside the `add` method, using two different levels, `info` and `debug`. The latter has finer detail. The

pom.xml file should be replaced by the following one.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/
    maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>book</groupId>
    <artifactId>EJB-server</artifactId>
    <version>1</version>
    <packaging>ejb</packaging>
    <name>EJB-server</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <wildfly-plugin-version>2.1.0.Beta1</wildfly-plugin-version>
    <maven.compiler.release>16</maven.compiler.release>
    </properties>
    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupId>
            <artifactId>jakarta.jakartaee-api</artifactId>
            <version>8.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.jboss.ejb3</groupId>
            <artifactId>jboss-ejb3-ext-api</artifactId>
            <version>2.3.0.Final</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>org.wildfly.plugins</groupId>
            <artifactId>wildfly-maven-plugin</artifactId>
            <version>${wildfly-plugin-version}</version>
        </dependency>
```

```
    </dependencies>
    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>${wildfly-plugin-version}</version>
                <configuration>
                    <skip>false</skip>
                    <hostname>wildfly</hostname>
                    <port>9990</port>
                    <username>admin</username>
                    <password>admin#7rules</password>
                    <filename>${project.artifactId}.jar</filename>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-ejb-plugin</artifactId>
                <version>3.1.0</version>
                <configuration>
                  <outputDirectory>${output}</outputDirectory>
                    <ejbVersion>3.1</ejbVersion>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Listing 3.4: `pom.xml`

Our code is not a stand-alone application, because the `Calculator` class has no `main` method. Therefore, we need to deploy this class on an application server, like WildFly, by using one of the following two options. The first option is to use the following command, where `$WILDFLY_HOME` should be replaced to correspond to the location of your server on your file system.

```
mvn clean package -Doutput=$WILDFLY_HOME/standalone/deployments
```

Listing 3.5: Packaging the project for deployment on WildFly.

The abovementioned command will result in a jar being packaged at the WildFly deployment directory. You can now start the server by moving to the server's installation directory and running the following command.

```
bin/standalone.sh --server-config=standalone-full.xml
```

Listing 3.6: Starting the WildFly server.

In case you are following the configuration that uses docker containers, you may use the second option, which is to deploy the project using the wildfly plugin, as follows.

```
mvn wildfly:deploy
```

Listing 3.7: Deploying the server EJB on WildFly.

The goals `mvn wildfly:deploy` or also `mvn wildfly:undeploy`, respectively deploy and undeploy the jar in WildFly. For this to work, the `wildfly-maven-plugin` must have been set in the `pom.xml` file, as shown previously in Listing 3.4. Furthermore, we must start WildFly before doing the deployment, otherwise the deployment will fail. We opted for having the WildFly administrator password in the `pom.xml` for convenience, but this does seems to be a very questionable idea for any serious utilization.

Once we deploy the project, we should see the following lines in WildFly's console:

```
    INFO  [org.jboss.as.repository] (management-handler-thread - 1) WFLYDR0001:
    Content added at location
    /opt/wildfly-24.0.1.Final/standalone/data/content/...
    INFO  [org.jboss.as.server.deployment] (MSC service thread 1-8) WFLYSRV0027:
    Starting deployment of "EJB-server.jar" (runtime-name: "EJB-server.jar")
    INFO  [org.jboss.weld.deployer] (MSC service thread 1-3) WFLYWELD0003:
    Processing weld deployment EJB-server.jar
    INFO  [org.jboss.as.ejb3.deployment] (MSC service thread 1-3) WFLYEJB0473:
    JNDI bindings for session bean named 'Calculator' in deployment unit
    'deployment "EJB-server.jar"' are as follows:

        java:global/EJB-server/Calculator!ejb.CalculatorRemote
        java:app/EJB-server/Calculator!ejb.CalculatorRemote
        java:module/Calculator!ejb.CalculatorRemote
        java:jboss/exported/EJB-server/Calculator!ejb.CalculatorRemote
        ejb:/EJB-server/Calculator!ejb.CalculatorRemote
        java:global/EJB-server/Calculator
        java:app/EJB-server/Calculator
        java:module/Calculator

    INFO  [io.smallrye.metrics] (MSC service thread 1-8) MicroProfile: Metrics
    activated (SmallRye Metrics version: 2.4.2)
    INFO  [org.jboss.as.server] (management-handler-thread - 1) WFLYSRV0010:
    Deployed "EJB-server.jar" (runtime-name : "EJB-server.jar")
```

Listing 3.8: "Console logging of WildFly showing the deployed calculator."

You may also check the deployment by accessing WildFly's console, as shown in Figure 3.1.

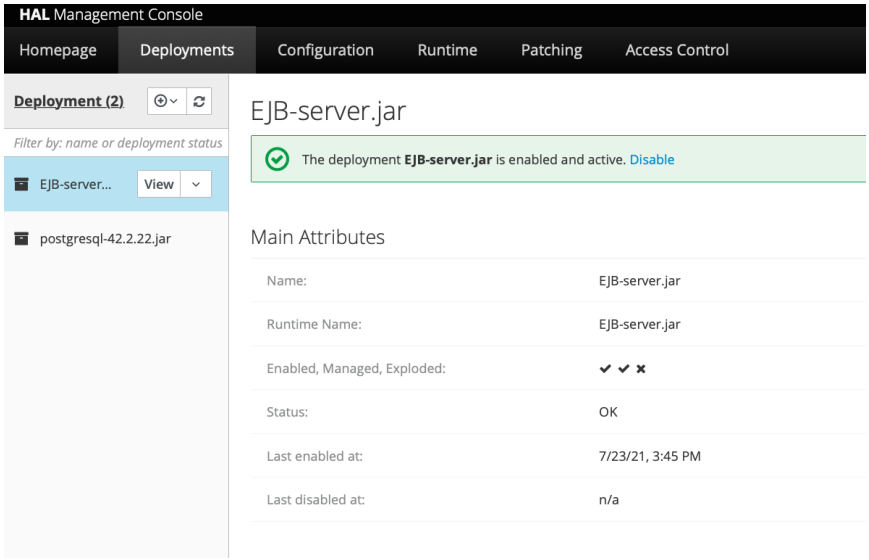

Figure 3.1: View of the Enterprise JavaBean deployed in WildFly.

3.2 A Client to Invoke the Enterprise JavaBean

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=EJB-client -Dversion=1 -DinteractiveMode=false
```

Listing 3.9: Maven archetype for a Enterprise JavaBeans application.

We are now going to create a small test program that will look for and invoke the EJB we have just deployed in the server. This program resorts to a Java Naming and Directory Interface (JNDI) client that looks up for a Calculator remote reference, before doing the invocation itself:

```
package ejbprimer;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import ejb.ICalculator;


public class TestCalculator {
```

```
  public static void main(String[] args) throws NamingException {
    ICalculator cr =
        InitialContext.doLookup("EJB-server/Calculator!ejb.ICalculator");
    System.out.println(cr.add(5, 3));
  }

}
```

Listing 3.10: Invoking the calculator. The`TestCalculator.java` file.

Notice that we make reference to the ICalculator file, which is actually an artifact that should be shared across projects. To simplify the demonstration, you may copy the file from the server project to this one. When copying be sure that you use the same destination package (i.e., the `ejb` package) and no not change the name of the class or contents.

The lookup operation depends on a number of properties that can be set in a file named `jndi.properties`, which names a factory class and the URL of the server. This file needs to stay in a resources directory, which is `src/main/resources`. The precise format of this file depends on the configuration of the server side. In this book we are considering two options: a direct installation of all the software in the operating system or a containerized approach using docker. In the former case, and as a first try, we are likely to run the client from the same host as where the server is running. For this solution the `jndi.properties` file looks as follows:

```
java.naming.factory.initial=org.wildfly.naming.client.WildFlyInitialContextFactory
java.naming.provider.url=remote+http://localhost:8080
jboss.naming.client.ejb.context=false
```

Listing 3.11: JNDI client configuration. The `jndi.properties` file for the case where client and server are on the same machine - the localhost.

This is about everything for this simplest case, where client and server reside on the same machine. This will not work, however, if client and server run on different machines or docker containers. In such case, we need to use the following `jndi.properties` file[1]:

```
java.naming.factory.initial=org.wildfly.naming.client.WildFlyInitialContextFactory
java.naming.provider.url=remote+http://wildfly:8080
jboss.naming.client.ejb.context=false
```

Listing 3.12: The `jndi.properties` file for the case where the server is in a container or in a different machine than the client.

Furthermore, when following the docker containers configuration, we need two additional files, `jboss-ejb-client.properties` and `wildfly-config.xml` in the `src/main/resources/META-INF` directory:

[1]We recommend the following GitHub project for additional details: https://github.com/wildfly/quickstart.git. For this case, look in the `ejb-security` example.

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=default

remote.connection.default.host=wildfly
remote.connection.default.port = 8080
remote.connection.default.username = ejbclient
remote.connection.default.password = ejb01acceS
remote.connection.default.connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS
    = JBOSS-LOCAL-USER
```

Listing 3.13: Client configuration. The `jboss-ejb-client.properties` file.

```
<configuration>
    <authentication-client xmlns="urn:elytron:1.0">
        <authentication-rules>
            <rule use-configuration="default-config"/>
        </authentication-rules>
        <authentication-configurations>
            <configuration name="default-config">
                <set-user-name name="ejbclient"/>
                <credentials>
                    <clear-password password="ejb01acceS"/>
                </credentials>
                <sasl-mechanism-selector selector="DIGEST-MD5"/>
                <providers>
                    <use-service-loader />
                </providers>
            </configuration>
        </authentication-configurations>
    </authentication-client>
</configuration>
```

Listing 3.14: Client configuration. The `wildfly-config.xml` file.

The client's `pom.xml` file should hold the following contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven
    -4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ejbprimer</groupId>
  <artifactId>EJB-client</artifactId>
```

```
<version>1</version>
<name>EJB-client</name>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>16</maven.compiler.release>
</properties>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.wildfly</groupId>
    <artifactId>wildfly-client-all</artifactId>
    <version>24.0.1.Final</version>
  </dependency>
</dependencies>
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>prepare-package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>${project.build.directory}/lib</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
```

```
            <classpathPrefix>lib/</classpathPrefix>
            <mainClass>ejbprimer.TestCalculator</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
              <version>3.8.0</version>
          </plugin>
  </plugins>
 </build>
</project>
```

Listing 3.15: Client's `pom.xml`

We may now compile the client with the following command.

```
mvn clean package
```

Listing 3.16: Compiling and packaging the client

Executing the client can be performed by running the following command:

```
java -jar target/EJB-client.jar
```

Listing 3.17: Running the client

The execution of the above command will get the expected `8` as response.

Chapter 4

SOAP Web Services

Goals

– Learn how to develop and deploy a SOAP web service.
– Learn how to create a SOAP client and invoke remote service operations.

Requirements

- wsimport tool bundled with the JAX-WS Reference Implementation
https://repo1.maven.org/maven2/com/sun/xml/ws/jaxws-ri/3.0.1/jaxws-ri-3.0.1.zip

SOAP web services have been designed to allow heterogeneous systems (i.e., clients and services) to inter-operate. A web service interface is described using a Web Services Description Language (WSDL), which is an XML-based specification that allows such representation to be made in a platform-independent manner. It allows describing the name of the service operations, which arguments each operation accepts, the data types of the arguments and response types, among several other aspects.

The information present in a WSDL file can be used by clients to write messages in a format that complies with the service interface, which are specified according to the Simple Object Access Protocol (SOAP) and typically transported via HTTP. SOAP is an XML-based protocol that allows, among several other aspects, to specify values for invoking service operations. A SOAP services environment also supports the presence of a registry, which allows providers to publish services using Universal Description, Discovery, and Integration (UDDI), which can then be discovered by clients.

In this chapter we use the Java API for XML Web Services (JAX-WS) 3.0.1 to create and deploy a SOAP web service and its respective client.

4.1 Building a SOAP Web Service

There are two main ways of creating a web service: top-down (also known as contract-first) and bottom-up (also known as code-first). The former involves writing the WSDL file first and using it to generate code that will support the service, while the latter requires the developer to write the code first and then using it to generate the WSDL. In this section we

will follow a code-first approach, which comprises the following basic steps: i) creating the project that will hold the web service code; ii) creating the web service code; iii) deploying the service in an application server.

We will start by creating the project that will hold all server-side components. The project can be a simple web application project that will hold the web service business logic in form of Java code. In this configuration, the web service is internally supported by a Servlet that takes care of receiving incoming HTTP requests and passing them on to the SOAP middleware involved (in our case, JAX-WS) for processing. You can create a web application project by executing the following command.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=soapws -Dversion=1 -DinteractiveMode=false
```

Listing 4.1: Maven soap web service project creation.

After creating the project structure be sure to delete the generated `web.xml` file, which is a deployment descriptor that can be used for configuring certain aspects of the web application (e.g., servlet mapping to certain URLs) and that we will not be using in this example. Replace the contents of the generated `pom.xml` file with the following.

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>book</groupId>
  <artifactId>soapws</artifactId>
  <version>1</version>
  <packaging>war</packaging>
  <name>soapws Maven Webapp</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>

  <dependencies>
    <dependency>
      <groupId>javax.jws</groupId>
      <artifactId>javax.jws-api</artifactId>
      <version>1.1</version>
    </dependency>
  </dependencies>
```

```
  <build>
    <finalName>${project.artifactId}</finalName>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <version>${wildfly-plugin-version}</version>
          <configuration>
            <skip>false</skip>
            <hostname>wildfly</hostname>
            <port>9990</port>
            <username>admin</username>
            <password>admin#7rules</password>
            <filename>${project.artifactId}.war</filename>
          </configuration>
        </plugin>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>3.2.2</version>
          <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
            <outputDirectory>${output}</outputDirectory>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 4.2: Soap web service project descriptor.

Now use your IDE to create a package named book, under the directory `src/main/java` and create the following new Java Class. The class will represent our service and will allow clients to perform a few arithmetic operations.

```
package book;

import javax.jws.WebService;

@WebService
public class CalculatorWS {

  public int add(int a, int b) {
    System.out.println("Calculating (" + a + " plus " + b + ") at the server...");
    return a + b;
  }

  public int subtract(int a, int b) {
    System.out.println("Calculating (" + a + " minus " + b + ")at the server...");
    return a - b;
  }
}
```

Listing 4.3: SOAP web service code.

The @WebService annotation on top of the class is a marker that tells the JAX-WS middleware that this class should be deployed as a SOAP web service. If no method-level annotations are present, all public methods are exported as operations of the service (in this case, add and subtract), thus being available for client invocations. Private methods will not be available to clients. If there is a need to export just a subset of the public methods, or customize the method exportation, we can mark methods with `@WebMethod`, which is an indication that the marked method should be exported as a web service operation (the remaining ones will not be exported). The `@WebMethod` annotation also allows us to specify a custom name for the operation, thus bypassing its automatic definition. It is also possible to set specific names for the operation arguments by preceeding each argument type with @WebParam(name="newName").

It is mandatory to place the service class under a package so that the JAX-WS runtime can instantiate it. In case the interface of a certain method uses complex parameters (i.e., not just basic types like integers or strings), they must comply with the JavaBeans specification. In particular, the classes involved must be placed under a package, there must be a constructor with no arguments (either implicit or explicit), attributes that are to be exported as part of the service must have corresponding getters and setters. In some cases where the attribute should not be exported but still the application requires that getters and setters exist, we can instruct JAX-WS to ignore such attribute when creating the WSDL file by marking the attribute or its getter with the `@XmlTransient` annotation.

Now it's time to package the service, which can be done easily by running the following command, where `$WILDFLY_HOME` should be replaced to correspond to the location of your

server on your file system.

```
mvn clean package -Doutput=$WILDFLY_HOME/standalone/deployments
```

Listing 4.4: Packaging the project for deployment on WildFly.

In case you are following the containerized configuration, you may deploy the project using the following command.

```
mvn wildfly:deploy
```

Listing 4.5: Using the wildfly pluing for deploying the project on WildFly.

Both options for deployment will result in packaging the whole application in a WAR file, which is basically a zip file holding the compiled contents of the project and complying with a specific structure. As you can see in the `pom.xml` file being used (see Listing 4.2), the maven-war-plugin is configured to set the assembly target directory to a custom local location, which is being specified at the commandline using `-Doutput` and that you should configure to fit your system. This location will be the final destination of the WAR file.

Once the maven command concludes, you can startup the application server. The server should confirm the correct deployment and it should announce it in the console with a set of messages like:

```
INFO  [org.jboss.ws.cxf.metadata] (MSC service thread 1-7) JBWS024061: Adding
    service endpoint metadata: id=book.CalculatorWS
 address=http://localhost:8080/soapws/CalculatorWS
 implementor=book.CalculatorWS
 serviceName={http://book/}CalculatorWSService
 portName={http://book/}CalculatorWSPort
 annotationWsdlLocation=null
 wsdlLocationOverride=null
 mtomEnabled=false
...
INFO  [org.apache.cxf.wsdl.service.factory.ReflectionServiceFactoryBean] (MSC
    service thread 1-7) Creating Service {http://book/}CalculatorWSService from
    class book.CalculatorWS
...
INFO  [org.apache.cxf.endpoint.ServerImpl] (MSC service thread 1-7) Setting the
    server's publish address to be http://localhost:8080/soapws/CalculatorWS
INFO  [org.jboss.ws.cxf.deployment] (MSC service thread 1-7) JBWS024074: WSDL
    published to:
    file:.../wildfly/standalone/data/wsdl/soapws.war/CalculatorWSService.wsdl
INFO  [org.jboss.as.webservices] (MSC service thread 1-8) WFLYWS0003: Starting
    service jboss.ws.endpoint."soapws.war"."book.CalculatorWS"
INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 86)
    WFLYUT0021: Registered web context: '/soapws' for server 'default-server'
INFO  [org.jboss.as.server] (ServerService Thread Pool -- 48) WFLYSRV0010:
    Deployed "soapws.war" (runtime-name : "soapws.war")
```

Listing 4.6: WildFly deploying the SOAP service.

The messages in Listing 4.6 carry essentially two important information. First, the web application has been correctly deployed, which you can confirm by inspecting the final message in Listing 4.6. This means that the server was able to generate a WSDL file for the service. Second, you can now discover the service address, which you can find by inspecting the first message in Listing 4.6. You can use a browser navigate to the announced address appended with `?wsdl`. In our case this would be `http://localhost:8080/soapws/CalculatorWS?wsdl`. If you can observe a well-formed WSDL file, you should now be able to create a client to invoke the service operations.

4.2 Creating the SOAP client

We begin by creating a simple java project that will hold the client code. Run the following command to create the project structure.

```
mvn archetype:generate  -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-simple -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=soapclient -Dversion=1 -DinteractiveMode=false
```

Listing 4.7: Maven simple java project creation.

We now need to add a few dependencies to the project so that the client. You can replace the generated pom by the following one.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>book</groupId>
  <artifactId>soapclient</artifactId>
  <version>1</version>
  <name>soapclient</name>
  <description>A simple soapclient.</description>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>
  <dependencies>
    <dependency>
      <groupId>jakarta.xml.ws</groupId>
      <artifactId>jakarta.xml.ws-api</artifactId>
      <version>3.0.1</version>
      <scope>compile</scope>
```

```
    </dependency>
    <dependency>
      <groupId>com.sun.xml.ws</groupId>
      <artifactId>jaxws-rt</artifactId>
      <version>3.0.1</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${project.build.directory}/lib</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
```

```
            <classpathPrefix>lib/</classpathPrefix>
            <mainClass>book.App</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

Listing 4.8: File pom.xml for the client.

The pom above essentially includes two dependencies: the Jakarta XML Web Services API and the JAX-WS Reference Implementation Runtime), which will allow you to compile and execute the client.

The easiest way of creating the client code is to use the wsimport tool that comes bundled with the reference implementation of JAX-WS (already included in the Docker containerized version). The tool will generate code that will allow invoking the web service operations in an easy manner. Assuming you have placed the JAX-WS distribution in your soapclient project (you can find the link for JAX-WS in the requirements at the beginning of this chapter), once you step into your project root, you may run the following command:

```
jaxws-ri/bin/wsimport.sh -keep -d src/main/java -p artifact
    "http://localhost:8080/soapws/CalculatorWS?wsdl"
```

Listing 4.9: Client-side artifact generation.

The -keep flag indicates that the generated code should be kept (and not only the compiled classes), the -p flag indicates the name of package where the code should be placed, the -d option indicates the final destination of the generated source code, and the final argument specifies where the wsdl file can be found. The wsimport tool should display a message mentioning it is parsing the WSDL, generating code, and compiling code. No error message should be observed in the console during the execution of these steps and you should be able to a new set of files (i.e., the generated code) under the artifact package in the project. The next step is to write the client code, which should be the following.

```
package book;

import artifact.CalculatorWS;
import artifact.CalculatorWSService;

public class App
{
    public static void main( String[] args )
    {
      int result;
      CalculatorWSService service = new CalculatorWSService();
```

```
        CalculatorWS calc = service.getCalculatorWSPort();
        result = calc.add(2, 3);
        System.out.println("Adding... result = " + result);
    }
}
```

Listing 4.10: SOAP client code.

The above code instantiates a Service object which is then used to retrieve a proxy to the service (i.e., the calc object). This proxy allows invoking the service operations in an easy manner and you should be able to inspect all available service operations (i.e., `add` and `subtract`) by using your IDE to run a code completion command over the `calc` object.

The final step is to compile and execute. Run `mvn clean package` and maven will prepare a jar file accompanied by the necessary dependencies, which you can now run with `java -jar target/soapclient.jar`. Run the created jar file and you should be able to see the result in the client console. You can also check the server console and confirm that it executed the print instruction that we had written in the add operation.

Chapter 5

REST Web Services

Goals

– Learn how to develop and deploy a REST web service and create a client for it.

– Understand the Jakarta REST Web Services API and gain experience with the RESTEasy project.

Requirements

– Java SE Development Kit

– WildFly

– Apache Maven

REST is an architectural style of providing services that builds on a set of principles, of which we highlight the following. In a REST architecture a `resource` represents information (e.g., a book in a library information system is a resource) and has a `global identifier` (usually a URI nowadays). A client may act on resources by specifying the resource identifier and the type of operation or `action` required, which nowadays usually corresponds to one of the HTTP verbs (e.g., GET, PUT, POST, DELETE). When a client asks for a certain resource, the server responds with its `representation`, which captures the current or intended state of the resource. This may be, for example, a document representing a certain book in a library). The format of the document may be specified by the client, but currently tends to be JavaScript Object Notation - JSON. Regardless of the specific format, the exchanged messages should be `self-descriptive`, in a way that allow the client to further act on them.

5.1 Building a REST web service

In this section we will create a project that will support our service, implement the service code, and finally deploy the project in the server. We begin by creating a simple web application project by running the following command.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
    -DgroupId=pt.uc.dei -DartifactId=restws -Dversion=1 -DinteractiveMode=false
```

Listing 5.1: Maven web application project creation.

Now replace the generated `pom.xml` file, by the following one.

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>pt.uc.dei</groupId>
  <artifactId>restws</artifactId>
  <version>1</version>
  <packaging>war</packaging>
  <name>soapws Maven Webapp</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>
  <dependencies>
    <dependency>
      <groupId>javax.ws.rs</groupId>
      <artifactId>javax.ws.rs-api</artifactId>
      <version>2.1.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>${project.artifactId}</finalName>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
```

```
          <artifactId>maven-war-plugin</artifactId>
          <version>3.2.2</version>
          <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
            <outputDirectory>${output}</outputDirectory>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <version>${wildfly-plugin-version}</version>
          <configuration>
            <skip>false</skip>
            <hostname>wildfly</hostname>
            <port>9990</port>
            <username>admin</username>
            <password>admin#7rules</password>
            <filename>${project.artifactId}.war</filename>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 5.2: Project descriptor file for the REST service.

Let's now move to the Java code. The following class is used to define the components of your JAX-RS application. The fact that it is currently empty (i.e., it does not override any methods) is an indicator that the server should scan the whole WAR deployment for JAX-RS annotation resources and providers. Otherwise, we should be able to specify specific resources or providers. The ApplicationPath annotation is used to specify that the REST resources will be available under a certain servlet context path, visible in the code below.

```
package book;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest")
public class ApplicationConfig extends Application
{

}
```

Listing 5.3: Definition of components and path of the REST service.

Let's now create the following `Person` class that will be used to represent fictitious data that is transferred between client and server.

```
package pojo;

public class Person {

  private String name;
  private int age;

  public Person() {
    // empty
  }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @Override
  public String toString() {
    return "Person:" + this.name + ", " + this.age;
  }
}
```

Listing 5.4: A class representing a person.

We now move on to the creation of the service itself. In our case, we will create a class with several methods (which will form the service operations), which has the sole purpose of illustrating different interface-level cases that may occur when creating REST services. As such, the different operations are simply accessible via numbered paths (e.g., person1,

person2), otherwise a different design should apply.

```
package book;

import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

import pojo.Person;

@Path("/myservice")
@Produces(MediaType.APPLICATION_JSON)
public class MyService {

  @GET
  @Path("/person1")
  public Person method1() {
    System.out.println("M1 executing....");

    return new Person("John", 10);
  }

  @GET
  @Path("/person2")
  public Person method2(@QueryParam("name") String value) {
    System.out.println("M2 executing... args=" + value);

    return new Person(value, 20);
  }

  @GET
  @Path("/person3/{name}")
  public Person method3(@PathParam("name") String value) {
    System.out.println("M3 executing... args=" + value);
    return new Person(value, 30);
  }
```

```
  @POST
  @Path("/person4")
  @Consumes(MediaType.APPLICATION_JSON)
  public Response method4(Person person) {
    System.out.println("M4 executing....");
    String str = "Person received : " + person.getName() + " " + person.getAge();
    return Response.status(Status.OK).entity(str).build();
  }

  @POST
  @Path("/person5")
  @Consumes(MediaType.APPLICATION_JSON)
  public Response method5(Person person) {
    System.out.println("M5 executing....");
    person.setName("No Name");
    person.setAge(person.getAge() + 1);
    return Response.status(Status.OK).entity(person).build();
  }

  @GET
  @Path("/person6")
  @Produces(MediaType.APPLICATION_JSON)
  public Response method6() {
    System.out.println("M6 executing....");
    Person p1 = new Person("James", 60);
    Person p2 = new Person("Kate", 61);
    Person p3 = new Person("Claire", 62);
    List<Person> personList = List.of(p1, p2, p3);
    return Response.ok().entity(personList).build();
  }

}
```

Listing 5.5: Class holding the implementation of the operations of the REST service.

The first aspect to notice is the `@Path` annotation that is used to specify another part of the URI that will be used to access the operations. This annotation is used at the class level and also at the method level to specify partial paths that will compose the final URI through which a certain resource may be accessed. In general, a certain REST resource will be identified by the following different parts:

`web context / application path / class-level path / method-level path /`

As an example, and knowing the project will be deployed with the name `restws`, method1 could be invoked using an HTTP `GET` request sent to the following address.

`http://localhost:8080/restws/rest/myservice/person1`

The operations that compose our service perform the following functions:

- method1: receives a simple GET request and serializes a JSON object that is sent back to the client as payload of an HTTP response.
- method2: receives a GET request with a name query parameter. The value is used to set the name of the Person object that is sent back to the client.
- method3: receives a GET request with a path parameter. The value in the path is used to set the name of the Person object that is sent back to the client.
- method4: receives a POST request with a JSON payload that is deserialized to a Person object. The response payload is text and the response status code is explicitly set to 200 OK.
- method5: receives a POST request with a JSON payload that is deserialized to a Person object. The response payload is JSON and the response status code is explicitly set to 200 OK.
- method6: receives a GET request and builds a JSON response with status 200 OK and holding a list of persons.

As we can see in the service code, the different HTTP verbs necessary to reach a certain operation are also available in JAX-RS via annotations with the same names (e.g.,GET, POST, PUT, DELETE). We are using only two verbs, for illustrative purposes. The methods can also be configured to produce a certain type of media, such as JSON or even XML. This is configured with the Produces annotation, which can be set at the class level, or at the method level. In this latter case, the method level configuration will take precedence and overried the class level configuration. The Consumes annotation indicates which the media types a resource can accept as input and again can be applied at class and method level.

You can deploy the service in wildfly by running the following command, where `$WILDFLY_HOME` should be replaced with the location of your server on your file system.

```
mvn clean package -Doutput=$WILDFLY_HOME/standalone/deployments
```

Listing 5.6: Packaging the project for deployment on WildFly.

In case you are following the containerized configuration, you may deploy the project using the following command.

```
mvn wildfly:deploy
```

Listing 5.7: Using the wildfly pluing for deploying the project on WildFly.

You should be able to see the initialization of the class that extends javax.ws.rs.core.Application, the registration of the web context, and also the successful deployment of the application, as follows.

```
INFO  [org.jboss.resteasy.resteasy_jaxrs.i18n] (ServerService Thread Pool -- 85)
    RESTEASY002225: Deploying javax.ws.rs.core.Application: class
    restws.ApplicationConfig
...
INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 85) WFLYUT0021:
    Registered web context: '/restws' for server 'default-server'
INFO  [org.jboss.as.server] (ServerService Thread Pool -- 48) WFLYSRV0010: Deployed
    "restws.war" (runtime-name : "restws.war")
```

Listing 5.8: Application server deploying the REST service and registering the web context.

Which of the service methods can you invoke directly from your browser (i.e., using the address bar)? Try to invoke them. You can use a tool like Postman (postman.com) to invoke the remaining methods.

5.2 Creating the REST client

Let us now create a simple Java project to programmatically invoke the operations of the REST service. Run the following command to create the project structure.

```
mvn archetype:generate  -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-simple -DarchetypeVersion=1.4
    -DgroupId=pt.uc.dei -DartifactId=restclient -Dversion=1 -DinteractiveMode=false
```

Listing 5.9: Maven simple Java project creation.

We can use the following `pom` that will take care of packaging and adding the necessary libraries, which are the RESTEasy client-side libraries and Jackson that will take care of marshalling and unmarshaling JSON.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>pt.uc.dei</groupId>
  <artifactId>restclient</artifactId>
  <version>1</version>
  <name>restclient</name>
  <description>A simple restclient.</description>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>
  <dependencies>

    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-client</artifactId>
      <version>3.13.2.Final</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jackson2-provider</artifactId>
      <version>3.13.2.Final</version>
    </dependency>

  </dependencies>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
```

```
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${project.build.directory}/lib</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <classpathPrefix>lib/</classpathPrefix>
              <mainClass>book.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 5.10: Maven project descriptor for the REST client.

The following code shows how to programatically invoke the different service operations. You can copy the Person class to the client side, to make the example compile.

```
package book;

import java.util.List;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.Response;

import pojo.Person;

public class App {
  public static void main(String[] args) {
    Client client = ClientBuilder.newClient();

    WebTarget target =
        client.target("http://localhost:8080/restws/rest/myservice/person1");
    Response response = target.request().get();
    String value = response.readEntity(String.class);
    System.out.println("RESPONSE1: " + value);
    response.close();

    target = client.target("http://localhost:8080/restws/rest/myservice/person2");
    target = target.queryParam("name", "xpto");
    response = target.request().get();
    value = response.readEntity(String.class);
    System.out.println("RESPONSE2: " + value);
    response.close();

    target = client.target("http://localhost:8080/restws/rest/myservice/person3/xpto");
    response = target.request().get();
    value = response.readEntity(String.class);
    System.out.println("RESPONSE3: " + value);
    response.close();

    target = client.target("http://localhost:8080/restws/rest/myservice/person4");
    Person p = new Person("Peter", 40);
    Entity<Person> input = Entity.entity(p, MediaType.APPLICATION_JSON);
    response = target.request().post(input);
    value = response.readEntity(String.class);
    System.out.println("RESPONSE4: " + value);
    response.close();

    target = client.target("http://localhost:8080/restws/rest/myservice/person5");
    response = target.request().post(input);
    p = response.readEntity(Person.class);
    System.out.println("RESPONSE5: " + p.getName() + " " + p.getAge());
    response.close();

    target = client.target("http://localhost:8080/restws/rest/myservice/person6");
    response = target.request().get();
    List<Person> personList = response.readEntity(new GenericType<List<Person>>(){});
    System.out.println("RESPONSE6: " + personList);
    response.close();
  }
}
```

Listing 5.11: REST client code.

Finally, we can run the client application by executing java -jar targetrestclient.jar. We should be able to see the result of invoking the different operations available at the server.

Chapter 6

Web Applications

Goals
– Learn how to create a basic web application using Servlets, Java Server Pages (JSP) and JSTL.
– Understand how to create a filter to intercept requests to resources.

In this Chapter, we will create a basic web application making use of Servlets, JSPs and JSTL. The application will hold minimum business logic and will follow the Model-View-Controller architecture (MVC). In MVC the model mostly represents the data, the view is concerned with the presentation aspects and will be mostly handled by JSP and JSTL, and the controller, represented in our case by a simple Servlet, handles user input and interacts with the data model, being the link between the user and the system.

6.1 Creating a basic web application

We begin by creating a web application project that will hold the code. Running the following command will create the project structure, be sure to delete the `web.xml` file after the command finishes.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4 -DgroupId=book
    -DartifactId=simpleweb -Dversion=1 -DinteractiveMode=false
```

Listing 6.1: Maven web application project creation.

You can use the following `pom.xml` as a replacement for the generated one. The project descriptor includes the servlet-api dependency, which is marked with scope `provided` as the application will run within the context of the server that will provide the necessary dependencies at runtime. We also include the Java Standard Tag Lib (JSTL) as a provided dependency, so that we can compile the project without further references to the set of Jakarta EE libraries provided by the application server.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>
<groupId>book</groupId>
<artifactId>simpleweb</artifactId>
<version>1</version>
<packaging>war</packaging>
<name>simpleweb Maven Webapp</name>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>16</maven.compiler.release>
</properties>
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <finalName>${project.artifactId}</finalName>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
          <outputDirectory>${output}</outputDirectory>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <version>${wildfly-plugin-version}</version>
        <configuration>
          <skip>false</skip>
          <hostname>wildfly</hostname>
          <port>9990</port>
          <username>admin</username>
          <password>admin#7rules</password>
          <filename>${project.artifactId}.war</filename>
        </configuration>
```

```
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 6.2: Web application project descriptor.

The following code creates a servlet, more specifically an HTTPServlet, which is basically a component that is able to process HTTP requests and is able to build the respective responses.

```
package servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/greeting")
public class GreetingServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
    response.getWriter().println("Hello World!!!!!");
  }
}
```

Listing 6.3: A basic HTTP servlet

Important elements in this Servlet are:

- `@WebServlet("/Greeting")` – Indicates the partial URL which will be associated with the servlet. This is an indication to the application server that means that requests that map to the specified pattern in this annotation should be forwarded to the servlet.
- doGet(HttpServletRequest request, HttpServletResponse response) – This method will be called when an HTTP GET request set to the above URL is received at the server. We can generate a GET request using a browser, or using telnet. The method includes two arguments (request and response) that represent the request received by the server and the response that will be sent to the client. We can use these objects to read information sent by the client and to manipulate the response that will be sent. In the case of the example we are using a Writer over the response object to send a String to the browser.

Notice that, in the case of this example, we will be able to process HTTP GET requests only, as there is a `doGet` implementation (and no `doPost`, for instance). Let us now deploy the project by running `mvn clean package`. We should see the web context being registered and the deployment finishing successfully, as follows.

```
INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 86) WFLYUT0021:
    Registered web context: '/simpleweb' for server 'default-server'
INFO  [org.jboss.as.server] (ServerService Thread Pool -- 48) WFLYSRV0010: Deployed
    "simpleweb.war" (runtime-name : "simpleweb.war")
```

Listing 6.4: Wildfly successfully deploying the web application and registering the web context.

Use a browser to access `http://localhost:8080/simpleweb/greeting`. This will trigger the servlet and you should be see the hello world message being printed at the browser, as follows.

Figure 6.1: Hello world message generated by a servlet.

We could change the servlet to respond with HTML instead of plain text, but let us try to forward the flow to a Java Server Pages (JSP) file, which can take care of the presentation aspects in an easier manner. JSP allows to include dynamic content brought in from the Java side within static content (i.e., typically HTML), thus this is a first step for separating the business logic from presentation aspects. The following code retrieves a `RequestDispatcher`, which is an object that is able to pass the request to another resource (e.g., a JSP file, another servlet). Change the contents of the `doGet` method to the following.

```
request.getRequestDispatcher("/display.jsp").forward(request, response);
```

Listing 6.5: Request dispatcher forwarding the request to a JSP file.

Now, create a `display.jsp` file under `src/main/webapp`. This file, which for now holds only static content, will be responsible for taking care of all presentation aspects.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
  <strong>Hi there!</strong>
  <div>Greetings from your JSP page!</div>
</body>
</html>
```

Listing 6.6: A JSP file with static content.

Deploy the WAR file again and use the browser to check the result.

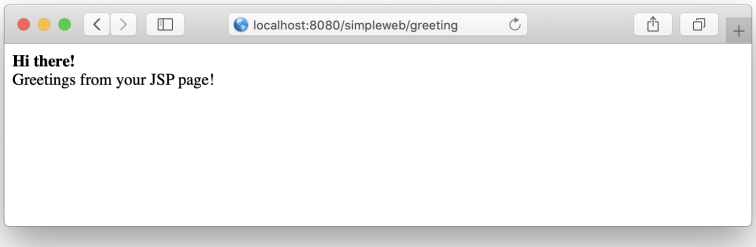

Figure 6.2: The output of a JSP file.

Let's now add some dynamic content at the servlet and render it at the JSP page. Replace the contents of the doGet method by the following.

```
request.setAttribute("today", new Date());
request.getRequestDispatcher("/display.jsp").forward(request, response);
```

Listing 6.7: Adding an object to the request.

As you can see we are adding an object to the request (it is a date object) and associating it with a name. At the JSP file we will retrieve it using JSTL. Replace your previous JSP page with the following (notice the inclusion of the JSTL taglib and a reference to the `today` name).

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
  <strong>Hi there!</strong>
  <div>Greetings from your JSP page!</div>
  <br />
  <div>Current date: ${today}</div>
</body>
</html>
```

Listing 6.8: Basic usage of JSTL in a JSP file

Let us again try to access the same URL as before and we should see the date being printed, as follows.

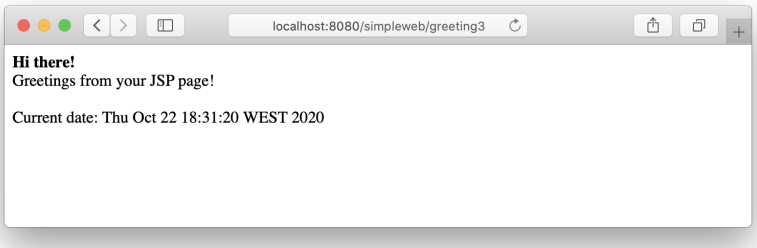

Figure 6.3: Dynamic content being produced by the JSP file.

JSTL allows to perform more complex operations, such as iterating over a collection of values. Replace your servlet `doGet` method with the following code.

```
List<String> myList = new ArrayList<String>();
myList.add("One...");
myList.add("Two...");
myList.add("Three...");
request.setAttribute("myListOfNumbers", myList);
    request.getRequestDispatcher("/display.jsp").forward(request, response);
```

Listing 6.9: Adding a list of strings at the servlet.

Now let us try to display the list at the JSP page. Use the following code inside the `<body></body>` tags.

```
<strong>A list of items...</strong>

<c:forEach var="item" items="${myListOfNumbers}">
    <div>Content is ${item}</div>
</c:forEach>
```

Listing 6.10: JSTL code for iterating over a list of Strings.

The previous code allows iterating over a list, in this case a list of Strings. The list to be iterated is referenced by the name `myListOfNumbers` and each element will be associated with the `item` variable. Should the list be composed of complex objects (i.e., JavaBeans), we could easily access each JavaBean property or attribute by using a dot followed by the name of the attribute, e.g., `item.propertyName`. Let's now try to deploy the WAR file again and check the result at the browser, which should be similar to the following output.

Let's now add a conditional instruction to the JSP file. In the case of JSTL, we can use the `if` tag to specify simple cases (e.g., `<c:if test="condition"></c:if>`, or the `choose` tag along with `/when` and also `otherwise` for more advanced control, as in the following example.

```
  <strong>
    <c:choose>
      <c:when test="${empty myListOfNumbers}">No items to be shown...</c:when>
      <c:when test="${myListOfNumbers.size()==1}">A single item...</c:when>
      <c:otherwise>Too many items...</c:otherwise>
```

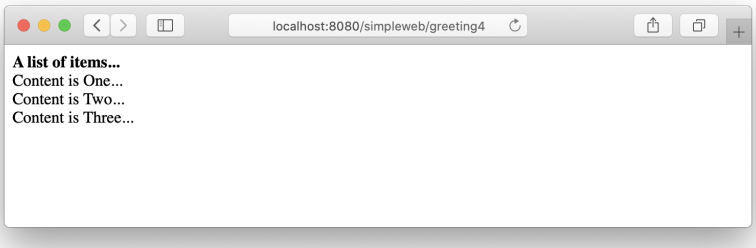

Figure 6.4: The iteration resulting output.

```
    </c:choose>
  </strong>
```

Listing 6.11: JSTL code illustrating conditional instructions.

In the case of this example 6.11, we are using a basic if-then-else block, where the first test in the **choose** statement uses the JSTL `empty` operator, the second test shows that it is also possible to use a Java-like syntax and the last item corresponds to the `else` case, which will be executed when none of the previous conditions apply. You can test this, by replacing the code shown in Listing 6.10 within the `<strong>` tag with the code shown in Listing 6.11 and by manually setting the number of items in the list being manipulated in the `GreetingServlet` and observe the different results. Naturally, the expression language is much more powerful. Overall, we should be careful with keeping the presentation layer concerned with presentation aspects only.

6.2 Intercepting requests with Filters

The Servlet specification includes a component named Filter, whose purpose is to dynamically intercept requests and perform some processing of a certain request before it reaches its destination (e.g., a servlet or a JSP file). It is also able to intercept responses and can again be used for performing some processing or manipulation of the response, before it is delivered to the client. It is usual that filters perform functions like logging or are used to authorize access to certain server resources. In this subsection we show a simple case of resource protection mostly carried out by a filter, used along with a simple case of authentication.

We now create a new web application project as before (you can create a copy of your previous one and name it `simpleweb1`), which will have the structure represented in Figure 6.5.

The file `login.html` is a simple entry point to the application, allowing a `name` and `key` parameter to reach the MainServlet. These two parameters represent authentication information.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"><head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <strong>Authentication example</strong>
```

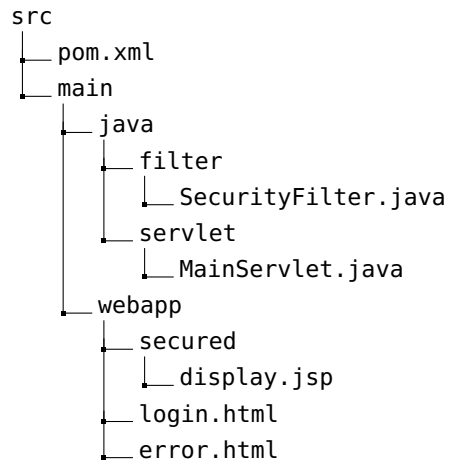

Figure 6.5: Project directory structure.

```
  <form action="main" method="get">
    <input name="name" type="text" placeholder="username..." />
    <input name="key" type="password" placeholder="password..." />
    <input type="submit">
  </form>
</body>
</html>
```

Listing 6.12: Authentication HTML page.

`MainServlet.java` receives the `name` and `key` parameters by `HTTP GET` and checks if they math the hardcoded credentials `john, 1`. In a real situation it would be likely that the server would delegate the authentication procedure to an Enterprise Java Bean. In case the credentials match, the servlet forwards the request to the `display.jsp` file, otherwise it removes any possible existing occurrence of the `auth` name from the session.

```
package servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/main")
public class MainServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
```

```
    String name = request.getParameter("name");
    String key = request.getParameter("key");
    String destination = "/error.html";

    if (name != null && key != null) {
      boolean auth = name.equals("john") && key.equals("11");
      if (auth) {
        request.getSession(true).setAttribute("auth", name);
        destination = "/secured/display.jsp";
      } else {
        request.getSession(true).removeAttribute("auth");
      }
    }

    request.getRequestDispatcher(destination).forward(request, response);
  }
}
```

Listing 6.13: Authentication servlet.

The `display.jsp` file will be simply displaying the value of the `auth` parameter, as follows.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
  <strong>Current logged user is ${auth}</strong>
</body>
</html>
```

Listing 6.14: Authentication servlet.

`SecurityFilter.java` is a filter that will be responsible for intercepting all requests to resources we want to secure and that should be available to authenticated users only. In this case, we decided to keep all protected resources under directory `secured` and, as you can see in the following code, the filter is set to be invoked upon each access to path under. This is specified by the class annotation `@WebFilter("/secured/*")`.

```
package filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
```

```
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;


@WebFilter("/secured/*")
public class SecurityFilter implements Filter {

  public void doFilter(ServletRequest request, ServletResponse response, FilterChain
      chain) throws IOException, ServletException {
    System.out.println("Accessing the filter...");
    HttpServletRequest httpReq = (HttpServletRequest) request;
    HttpSession session = httpReq.getSession(false);

    if (session != null && session.getAttribute("auth") != null)
    {
      System.out.println("Verified authentication token...");
      chain.doFilter(request, response);
    }
    else
    {
      request.getRequestDispatcher("/error.html").forward(request, response);
    }
  }
}
```

Listing 6.15: Filter example.

Upon each invocation, the filter checks for any existing session. If the session exists and holds a token named 'auth' (that is set when the user successfully authenticates) then the filter allows the flow to proceed to the secured destination, otherwise it simply forwards the request to `error.html`, which is a simple html file with an error message (e.g., Unauthorized access), as follows.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <strong>Unauthorized access.</strong>
</body>
</html>
```

Listing 6.16: Error message file.

You can run `mvn clean package` and try to first access a secured resource like the following.

```
http://localhost:8080/simpleweb1/secured/display.jsp
```

Listing 6.17: URL access for a secured resource.

The access should have you forwarded to the error page. Now try to access the login page to try out the authentication.

```
http://localhost:8080/simpleweb1/login.html
```

Listing 6.18: Login page URL.

In case you use the wrong credentials you should again be forwarded to the error page, otherwise you should see `John` displayed as the current logged user.

Chapter 7

Messaging

Goals
– Learn how to use the Java Message Service (JMS) 2.0 API.
– Learn how to use Apache Kafka.

In this Chapter, we will learn how to use messaging and stream processing. In Section 7.1 we will learn how to use JMS 2.0 supported by Apache ActiveMQ Artemis in WildFly. In Section 7.2, we exemplify the use of Kafka. Kafka is also a Message-Oriented Middleware (MOM) that has gained of lot of traction in the industry, so covering it is worthwhile. Furthermore, it supports reactive programming via the streams library, a fault-tolerant and scalable way of writing applications.

7.1 Java Message Service

In this section we will create three applications that will exchange messages of different types and using different configurations: a message sender, a synchronous message receiver, and an asynchronous message receiver and will go go through durable subscriptions and temporary queues. Notice that interchangeably we will also use the terms producer and consumer as synonyms of sender and receiver, respectively.

We begin by configuring WildFly for supporting our messaging needs. Use a text editor to open the following file:

`standalone/configuration/standalone-full.xml`

Find the `DLQ` queue and add another named `playQueue`, so that you get: We begin by configuring WildFly for supporting our messaging needs. If you are following the Docker configuration path, you do not need to do these configurations. Open `standalone/configuration/standalone-full.xml`, find the `DLQ` queue and add another named `playQueue`, so that you get:

```
<jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
<jms-queue name="playQueue" entries="java:jboss/exported/jms/queue/playQueue"/>
```

Listing 7.1: Configuration required for a Queue named playQueue.

Add a new user to WildFly by executing the `add-user` command under the `$WILDFLY_HOME/bin` directory.

```
athena:wildfly-10.1.0.Final user$ bin/add-user.sh
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): b
Enter the details of the new user to add.
Using realm 'ApplicationRealm' as discovered from the existing property files.
Username : john
Password recommendations are listed below. To modify these restrictions edit the
    add-user.properties configuration file.
- The password should be different from the username
- The password should not be one of the following restricted values {root, admin,
    administrator}
- The password should contain at least 8 characters, 1 alphabetic character(s), 1
    digit(s), 1 non-
alphanumeric symbol(s)
Password : !1secret
Re-enter Password : !1secret
What groups do you want this user to be\texttt{Long} to? (Please enter a comma separated
    list, or leave blank for none)[ ]: guest
About to add user 'john' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'john' to file '/Users/nuno/Documents/tech/wildfly-
    10.1.0.Final/standalone/configuration/application-users.properties'
Added user 'john' to file
    '/Users/nuno/Documents/tech/wildfly-10.1.0.Final/domain/configuration/application-
    users.properties'
Added user 'john' with groups guest to file '/Users/nuno/Documents/tech/wildfly-
    10.1.0.Final/standalone/configuration/application-roles.properties'
Added user 'john' with groups guest to file '/Users/nuno/Documents/tech/wildfly-
    10.1.0.Final/domain/configuration/application-roles.properties'
Is this new user going to be used for one AS process to connect to another AS process?
e.g. for a slave host controller connecting to the master or for a Remoting connection
    for server to server EJB calls.
yes/no? no
```

Listing 7.2: Adding a new user to WildFly.

Let us now create a simple Java project to try a few basic messaging operations. Run the following command to create the project structure.

```
mvn archetype:generate  -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-simple -DarchetypeVersion=1.4 -DgroupId=book
    -DartifactId=jmstutorial -Dversion=1 -DinteractiveMode=false
```

Listing 7.3: Maven simple Java project creation.

Use the following `pom.xml` that will take care of packaging and especially adding the necessary client-side library (which is also available under the `$WILDFLY_HOME/bin/client` directory).

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>book</groupId>
  <artifactId>jmstutorial</artifactId>
  <version>1</version>
  <name>jmstutorial</name>
  <description>A simple jms client.</description>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.wildfly</groupId>
      <artifactId>wildfly-jms-client-bom</artifactId>
      <version>21.0.2.Final</version>
      <type>pom</type>
    </dependency>
  </dependencies>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${project.build.directory}/lib</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <archive>
            <manifest>
```

```
            <addClasspath>true</addClasspath>
            <classpathPrefix>lib/</classpathPrefix>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

Listing 7.4: Maven project descriptor for the JMS client.

Now create a `jndi.properties` file under the `src/main/resources` folder of your project, with the following content, which will allow you the JMS program to connect to the JNDI provider:

```
java.naming.factory.initial=org.wildfly.naming.client.WildFlyInitialContextFactory
java.naming.provider.url=remote+http://localhost:8080
jboss.naming.client.ejb.context=false
```

Listing 7.5: Configuration of the jndi.properties file.

In case you are using the containerized setup, make sure you replace `localhost` with `wildfly` in the `jndi.properties` file. We are now ready to create the code and execute. Use the following code to create a **message producer** application.

```
package book;

import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSContext;
import javax.jms.JMSProducer;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Sender {

  private ConnectionFactory connectionFactory;
  private Destination destination;

  public Sender() throws NamingException {
    this.connectionFactory = InitialContext.doLookup("jms/RemoteConnectionFactory");
    this.destination = InitialContext.doLookup("jms/queue/playQueue");
  }

  private void send(String text) {
    try (JMSContext context = connectionFactory.createContext("john", "!1secret");) {
      JMSProducer messageProducer = context.createProducer();
      messageProducer.send(destination, text);
    } catch (Exception re) {
      re.printStackTrace();
    }
  }
```

```
  public static void main(String[] args) throws NamingException {
    Sender sender = new Sender();
    sender.send("Hello Receiver!");
    System.out.println("Finished sender...");
  }
}
```

Listing 7.6: Code for a message producer.

You may have noticed the following less usual code:

```
try (JMSContext jcontext = connectionFactory.createContext("john", "!1secret");) {
    JMSProducer messageProducer = context.createProducer();
        messageProducer.send(destination, text);
}
catch (Exception re) {
    re.printStackTrace();
}
```

Listing 7.7: A try-with-resources block.

The `try-with-resources` is a feature first introduced with Java 7. The main idea is that you will not need to clean up the resources when you finish the operation. In this case, you will not need to explicitly close the JMSContext object. Anyway, in some cases you may need to use the regular try-catch, but make sure that you close the context when it is no longer needed.

Use the following code to create a **synchronous consumer** application.

```
package book;

import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.JMSRuntimeException;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Receiver {

  private ConnectionFactory connectionFactory;
  private Destination destination;

  public Receiver() throws NamingException {
    this.connectionFactory = InitialContext.doLookup("jms/RemoteConnectionFactory");
    this.destination = InitialContext.doLookup("jms/queue/playQueue");
  }

  private String receive() {
    String msg = null;
    try (JMSContext context = connectionFactory.createContext("john", "!1secret");) {
      JMSConsumer mc = context.createConsumer(destination);
      msg = mc.receiveBody(String.class);
```

```
    } catch (JMSRuntimeException re) {
      re.printStackTrace();
    }
    return msg;
  }

  public static void main(String[] args) throws NamingException {
    Receiver receiver = new Receiver();
    String msg = receiver.receive();
    System.out.println("Message: " + msg);
  }
}
```

Listing 7.8: Code for a synchronous message consumer.

Start WildFly and run `mvn clean package` to compile and package the project. As we will be running several classes written in the same project, you can resort to the following command to run a specific class (in this case, the Receiver class).

```
java -cp target/jmstutorial.jar book.Receiver
```

Listing 7.9: Running class Sender.

Try to execute the receiver and then the sender (by replacing `Receiver` with `Sender` in the previous command) and you should see a message being exchanged. Does the order of execution make a difference?

The following code represents an **asynchronous consumer**, create a new class with it and try to execute the sender and the asynchronous consumer.

```
package book;

import java.io.IOException;

import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSConsumer;
import javax.jms.JMSContext;
import javax.jms.JMSException;
import javax.jms.JMSRuntimeException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class AsyncReceiver implements MessageListener {
  private ConnectionFactory connectionFactory;
  private Destination destination;

  public AsyncReceiver() throws NamingException {
    this.connectionFactory = InitialContext.doLookup("jms/RemoteConnectionFactory");
    this.destination = InitialContext.doLookup("jms/queue/playQueue");
```

```
  }

  @Override
  public void onMessage(Message msg) {
    TextMessage textMsg = (TextMessage) msg;
    try {
      System.out.println("Got message: " + textMsg.getText());
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }

  public void launch_and_wait() {
    try (JMSContext context = connectionFactory.createContext("john", "!1secret");) {
      JMSConsumer consumer = context.createConsumer(destination);
      consumer.setMessageListener(this);
      System.out.println("Press enter to finish...");
      System.in.read();
    } catch (JMSRuntimeException | IOException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) throws NamingException, IOException {
    AsyncReceiver asyncReceiver = new AsyncReceiver();
    asyncReceiver.launch_and_wait();
  }
}
```

Listing 7.10: Code for an asynchronous message consumer.

Let's now change the configuration to create a Topic at the server. Open `standalone-full.xml` (always shut the server down before editing this file) and add the following Topic configuration, below the previously created queue.

```
<jms-topic name="playTopic" entries="java:jboss/exported/jms/topic/playTopic"/>
```

Listing 7.11: Topic configuration.

Let's also give permission to the guest users to create durable queues. For this, find the following line in `standalone-full.xml`:

```
<role name="guest" send="true" consume="true" create-non-durable-queue="true"
    delete-non-durable- queue="true"/>
```

Listing 7.12: Default guest user queue permissions.

and replace it with:

```
<role name="guest" send="true" consume="true" create-non-durable-queue="true"
    delete-non-durable-queue="true" create-durable-queue="true"
    delete-durable-queue="true"/>
```

Listing 7.13: Guest user configuration for allowing creating and deleting durable queues.

Change your sender and receiver classes to lookup your topic:

```
this.destination = InitialContext.doLookup("jms/topic/playTopic");
```

Listing 7.14: Topic lookup code.

Now try to run two receivers and a sender. How many applications receive the message? What happens if you change the order of execution? What if a subscriber crashes, will it get a published message after reconnecting?

As we already have the configuration in place, we can create a **durable consumer** so that messages are not lost if the subscriber goes offline. For this you only need to change the line that creates the consumer to the following.

```
context.setClientID("someId");
JMSConsumer mc = context.createDurableConsumer((Topic) destination, "mySubscription");
```

Listing 7.15: Durable consumer creation.

The purpose of the above client identifier is to allow the provider to maintain the necessary state on behalf of particular clients (necessary to support the durable subscription). The name `mySubscription` identifies the subscription. A crashed consumer must use both (after restart) to retrieve any messages meanwhile sent to the topic.

Many times, we will need to create **temporary queues** in order to receive a reply from the other endpoint. Change your sender application to use the following code that creates a TextMessage (which types of Message can you send over the wire?) and sends it to the other endpoint. At the same time the code is setting a reference to a temporary Destination in the message, so that the receiver knows where to send a reply.

```
JMSProducer messageProducer = context.createProducer();
TextMessage msg = context.createTextMessage();
Destination tmp = context.createTemporaryQueue();
msg.setJMSReplyTo(tmp);
msg.setText("Hello my friend!");
messageProducer.send(destination, msg);
JMSConsumer cons = context.createConsumer(tmp);
String str = cons.receiveBody(String.class);
System.out.println("I received the reply sent to the temporary queue: " + str);
```

Listing 7.16: Temporary queue usage at the **Sender**.

The following code reflects the needed changes at the receiver application (try to fix the resulting errors in the code).

```
JMSConsumer consumer = context.createConsumer(destination);
TextMessage msg = (TextMessage) consumer.receive();
System.out.println("Message received:" + msg.getText());
JMSProducer producer = context.createProducer();
TextMessage reply = context.createTextMessage();
reply.setText("Goodbye!");
```

```
producer.send(msg.getJMSReplyTo(), reply);
System.out.println("Sent reply to " + msg.getJMSReplyTo());
```

Listing 7.17: Temporary queue usage at the **Receiver**.

Finally, in an Enterprise Edition (EE) container environment, a MessageDriven Bean is the typical solution to read messages from a Destination. The following code, illustrates how to define a simple bean that will be listening for messages arriving at your `playQueue`, which you can put in practice once you go through Chapter 8.

```
package beans;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destination", propertyValue =
        "jms/queue/playQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
        "javax.jms.Queue") })
public class MyMessageDrivenBean implements MessageListener {

  public void onMessage(Message message) {
    try {
      System.out.println(">>>>>>>>>>" + message.getBody(String.class));
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 7.18: Code for a MessageDrivenBean.

Also, in a container-managed environment it will be possible to use dependency injection to inject a JMSContext and a destination in a Session Bean, as follows.

```
@Inject
private JMSContext context;

@Resource(mappedName = "java:jboss/exported/jms/queue/playOut")
private Destination destination;
```

Listing 7.19: JMSContext and Destination injection in a EE managed environment.

The Session Bean may then be used to send messages to the destination.

7.2 Apache Kafka

Use the following `maven` command to create a project for the Kafka and Kafka Streams examples:

```
    mvn archetype:generate  -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-simple -DarchetypeVersion=1.4 -DgroupId=book
    -DartifactId=KafkaStreams -Dversion=1.0-SNAPSHOT -DinteractiveMode=false
```

Listing 7.20: Maven command for creating a simple Java project.

You may use the following `pom.xml`:

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>book</groupId>
  <artifactId>KafkaStreams</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>KafkaStreams</name>
  <description>A simple KafkaStreams.</description>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.8.0</version>
    </dependency>

    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.12.3</version>
    </dependency>

    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-streams</artifactId>
      <version>2.8.0</version>
    </dependency>
```

```
</dependencies>

<build>
  <finalName>${project.artifactId}</finalName>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <version>3.7.1</version>
      </plugin>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>3.0.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.1</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-install-plugin</artifactId>
        <version>2.5.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-deploy-plugin</artifactId>
        <version>2.8.2</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<reporting>
  <plugins>
    <plugin>
      <artifactId>maven-project-info-reports-plugin</artifactId>
    </plugin>
```

```
    </plugins>
  </reporting>
</project>
```

Listing 7.21: Maven project descriptor for the Kafka client (`pom.xml`).

We will create a simple publisher and simple consumer. Let us start with the consumer. Please take notice of the package:

```
package kafka;

import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class SimpleConsumer {

  public static void main(String[] args) throws Exception{
    //Assign topicName to string variable
    String topicName = args[0].toString();
    // create instance for properties to access producer configs
    Properties props = new Properties();

    //Assign localhost id
    props.put("bootstrap.servers", "localhost:9092");
    //Set acknowledgements for producer requests.
    props.put("acks", "all");
    //If the request fails, the producer can automatically retry,
    props.put("retries", 0);
    //Specify buffer size in config
    props.put("batch.size", 16384);
    //Reduce the no of requests less than 0
    props.put("linger.ms", 1);
    //The buffer.memory controls the total amount of memory available to the producer for
        buffering.
    props.put("buffer.memory", 33554432);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
    props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
        "org.apache.kafka.common.serialization.LongDeserializer");

    Consumer<String, Long> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Collections.singletonList(topicName));

    while (true) {
      ConsumerRecords<String, Long> records = consumer.poll(Long.MAX_VALUE);
```

```
      for (ConsumerRecord<String, Long> record : records) {
        System.out.println(record.key() + " => " + record.value());
      }
    }
    //consumer.close();
  }
}
```

Listing 7.22: A consumer for a Kafka topic (`SimpleConsumer.java` file).

We now follow to the producer:

```
package kafka;

import java.util.Properties;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class SimpleProducer {

  public static void main(String[] args) throws Exception{
    //Assign topicName to string variable
    String topicName = args[0].toString();
    // create instance for properties to access producer configs
    Properties props = new Properties();

    //Assign localhost id
    props.put("bootstrap.servers", "localhost:9092");
    //Set acknowledgements for producer requests.
    props.put("acks", "all");
    //If the request fails, the producer can automatically retry,
    props.put("retries", 0);
    //Specify buffer size in config
    props.put("batch.size", 16384);
    //Reduce the no of requests less than 0
    props.put("linger.ms", 1);
    //The buffer.memory controls the total amount of memory available to the producer for
        buffering.
    props.put("buffer.memory", 33554432);
    props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer",
    "org.apache.kafka.common.serialization.LongSerializer");

    Producer<String, Long> producer = new KafkaProducer<>(props);

    for(int i = 0; i < 1000; i++) {
      producer.send(new ProducerRecord<String, Long>(topicName, Integer.toString(i),
          (long) i));
      if (i % 100 == 0)
        System.out.println("Sending message " + (i + 1) + " to topic " + topicName);
```

```
    }
    producer.close();
  }
}
```

Listing 7.23: A producer for a Kafka topic (`SimpleProducer.java` file).

To run the producer and the consumer, you should check the Apache Kafka documentation to see the exact details. Starting the server involves Zookeper, to keep state, and the server itself. To start Zookeeper, do as follows:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Listing 7.24: Starting Zookeper

To start the server, do as follows:

```
bin/kafka-server-start.sh config/server.properties
```

Listing 7.25: Starting the Kafka server

You are now able to run the consumer and the producer. As long as you pick the same topic for both, the consumer will see whatever the producer publishes to the topic regardless of the order you start them, even if the consumer is down for a while, as the server will delete the messages only after a predetermined period of time.

7.3 Kafka Streams

According to its own site, "Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters". The main idea is to see the data going through the Kafka topics as an endless stream, which we can simply filter, or join with other topics in a fault-tolerant, scalable, and simple way. For example, a service might be publishing data regarding the sales it is doing to a topic, while another service might be subscribed to the topic, filtering the data, to learn about the most popular item during the last hour. Another possible use of streams is to monitor distributed microservice applications that use the topics to publish their internal metrics.

In this section, we assume that you have been able to start Kafka and that it is running on your localhost on port 9092, together with Zookeeper, which is available on port 2181, as in the previous section. We will then use a stream of key-value pairs and use this stream to solve a couple of exercises:

- Counting the occurrences of a given key in a stream.
- Converting the output from `Long` to `String`.
- Doing a Reduce() to sum all the values for each key.
- Use materialized views to perform queries on data.
- Use time windows to restrict the queries on the streams.

Figure 7.1: Using Kafka Streams to process data (from the Kafka Streams online documentation).

7.3.1 Overview

In the end, we want to have an arrangement similar to that of Figure 7.1. Here, a producer is writing content to the topic, a dedicated application is reading the data from the topic (possibly in parallel with other applications) and outputting results to a second topic. In this second topic, we have a dedicated consumer waiting to get the results computed by the streams application (in the middle of the figure).

To reach this setting, we start from the right-side components. You could use our own consumer from the previous section. We can, however, also resort to a shell application provided with Kafka, to run a consumer waiting on the `resultstopic` topic, using the following command for that purpose:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic resultstopic
```

Listing 7.26: Maven archetype for an Enterprise JavaBeans application.

Note that the exact way of running this command depends on how you ran Kafka. Once you start the consumer, you may want to start a producer, just as we saw in the previous section and make it point to the topic `kstreamstopic`. Finally, we need to run the following streams processing code:

```
package streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;


public class SimpleStreamsExercisesa {

  public static void main(String[] args) throws InterruptedException, IOException {
    if (args.length != 2) {
      System.err.println("Wrong arguments. Please run the class as follows:");
      System.err.println(SimpleStreamsExercisesa.class.getName() + " input-topic
          output-topic");
      System.exit(1);
    }
    String topicName = args[0].toString();
    String outtopicname = args[1].toString();
```

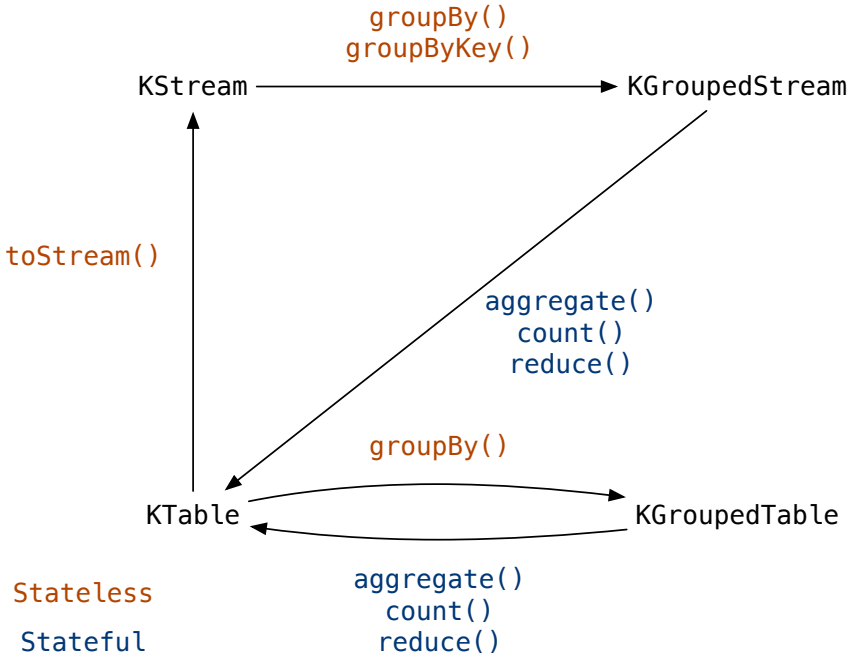

Figure 7.2: Using Kafka streams to process data (from the KafkaStreams online documentation).

```
    java.util.Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "exercises-application-a");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass());

    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, Long> lines = builder.stream(topicName);

    KTable<String, Long> outlines = lines.
    groupByKey().count();
    outlines.toStream().to(outtopicname);

    KafkaStreams streams = new KafkaStreams(builder.build(), props);
    streams.start();

    System.out.println("Reading stream from topic " + topicName);

  }
}
```

Listing 7.27: A Kafka streams consumer (`SimpleStreamsExercisesa.java` file).

This program picks the key-value pairs the producer writes to the `kstreamstopic` and processes them. To understand the kind of processing it does, we may analyze Figure 7.2, from the Kafka

Streams documentation.

The `groupByKey()` operation converts the stream to a `KGroupedStream`, by creating records of values indexed by the keys. In our case, since the producer will output 1000 different keys, each key will have a single record (a 0 for key 0, a 1 for key 1, a value 2 for key 2 and so on). Hence, the following `count()` will compute 1 for all keys and convert the `KGroupedStream` into a `KTable`, which is similar to a regular database table, having the key as the primary key. This table will have 1000 registers, with 1000 different keys, each one having the corresponding value of 1. To see the result, we convert the `KTable` back to a `KStream` using the `toStream()`.

To run the experiment, we need to specify the topic where the streams application will receive the data, as a command line argument. The same for the producer. A lack to do this will crash the programs. You may use the names of Figure 7.1. Just start the applications in this order:

1. First, `kafka-console-consumer.sh`.
2. Then, `SimpleStreamsExercises`.
3. Finally, the Producer.

Regarding the issue of the order at which applications start, Kafka keeps messages on the topic for a configurable amount of time, so we could always get the messages from the topic later, if necessary. You may also repeat the execution of the Producer as many times as you want, with only slight changes in the results (the value of the `count()` will keep increasing).

7.3.2 Converting from `Long` to `String`

If you run the previous setting you will get nothing that you could see on the `kafka-console-consumer.sh`, except 1000 empty lines. Why? Because we are outputting `Long`s instead of `String`s and, therefore, you will be looking at American Standard Code for Information Interchange (ASCII) character 1. Let us change our code slightly, to ensure that we can properly see the results of our operation:

```
package streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;


public class SimpleStreamsExercisesb {

  public static void main(String[] args) throws InterruptedException, IOException {
    if (args.length != 2) {
      System.err.println("Wrong arguments. Please run the class as follows:");
      System.err.println(SimpleStreamsExercisesb.class.getName() + " input-topic
          output-topic");
      System.exit(1);
```

```
    }
    String topicName = args[0].toString();
    String outtopicname = args[1].toString();

    java.util.Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "exercises-application-b");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass());

    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, Long> lines = builder.stream(topicName);

    KTable<String, Long> outlines = lines.groupByKey().count();

    outlines.mapValues(v -> "" + v).toStream().to(outtopicname,
        Produced.with(Serdes.String(), Serdes.String()));

    KafkaStreams streams = new KafkaStreams(builder.build(), props);
    streams.start();

  }
}
```

Listing 7.28: A modified Kafka streams consumer (`SimpleStreamsExercisesb.java` file).

What is new here? The `mapValues()`, which uses a lambda expression to transform the `Long` value `v` into a `String`. However, this change alone crashes the program, because we specified the `DEFAULT_VALUE_SERDES` to be a `Long`. Hence, the attempt to write a `String` on the `outtopicname` stream will not work. We need te explicitly tell the library that we are producing the output with a `String` format (the `Produced.with` in the end). In other words, the final stream has a format that is different from the initial stream and from the `KTable`. These two had `Long` values, while the final stream has a `String` value.

Now, you should see this in the Kafka-console-consumer shell:

```
3
3
3
3
3
...
```

or whatever number of times you ran the whole application, instead of 3 (e.g., I'm actually seeing a thousand 10s).

This is still not very handy, because we cannot see the keys. To see them we may change the lambda expression in the mapValues to become:

```
mapValues((k, v) -> k + " => " + v)
```

Listing 7.29: Displaying the key and the value

i.e., it receives the key-value pair and replaces the value (because the function is `mapValues()`) by the string with the key, the arrow and the value, which is much nicer (the 12 might be different in your case):

```
...
989 => 12
990 => 12
991 => 12
992 => 12
993 => 12
994 => 12
995 => 12
996 => 12
997 => 12
998 => 12
999 => 12
```

7.3.3 `Reduce()`

What if we want to perform some operation on the values of a given key, say adding them up? In this case, we may swap the `count()` by a `reduce()`:

```
KTable<String, Long> outlines = lines.
    groupByKey().
    reduce((oldval, newval) -> oldval + newval);
outlines.mapValues((k, v) -> k + " => " + v).toStream().to(outtopicname,
    Produced.with(Serdes.String(), Serdes.String()));
```

Listing 7.30: Adding the values for each key

The lambda expression in the `reduce()` keeps accumulating the new values that show up for the key. The reduce stores the result as it is stateful (mind the legend in the figure before, regarding the `reduce()`). For example, you might see the following output in `kafka-console-consumer.sh`:

```
985 => 3940
986 => 3944
987 => 3948
988 => 3952
989 => 3956
990 => 3960
991 => 3964
992 => 3968
993 => 3972
994 => 3976
995 => 3980
996 => 3984
997 => 3988
998 => 3992
999 => 3996
```

7.3.4 Materialized Views

Now, the case for a materialized view. Materialized views allow us to query the tables, either directly by reaching for the value of a key, or in ranges, as show in this case:

```
package streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.state.KeyValueIterator;
import org.apache.kafka.streams.state.QueryableStoreTypes;
import org.apache.kafka.streams.state.ReadOnlyKeyValueStore;


public class SimpleStreamsExercisesc {

    private static final String tablename = "exercises";

    public static void main(String[] args) throws InterruptedException, IOException {
        if (args.length != 2) {
            System.err.println("Wrong arguments. Please run the class as follows:");
            System.err.println(SimpleStreamsExercisesc.class.getName() + " input-topic
                output-topic");
            System.exit(1);
        }
        String topicName = args[0].toString();
        String outtopicname = args[1].toString();

        java.util.Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "exercises-application-c");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            Serdes.Long().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, Long> lines = builder.stream(topicName);

        KTable<String, Long> countlines = lines.
        groupByKey().
        reduce((oldval, newval) -> oldval + newval, Materialized.as(tablename));
        countlines.mapValues(v -> "" + v).toStream().to(outtopicname,
            Produced.with(Serdes.String(), Serdes.String()));
```

```
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();


        System.out.println("Press enter when ready...");
        System.in.read();
        while (true) {
            ReadOnlyKeyValueStore<String, Long> keyValueStore = streams.store(tablename,
                QueryableStoreTypes.keyValueStore());
            System.out.println("count for 355:" + keyValueStore.get("355"));
            System.out.println();
            // Get the values for a range of keys available in this application instance
            KeyValueIterator<String, Long> range = keyValueStore.range("960", "980");
            while (range.hasNext()) {
                KeyValue<String, Long> next = range.next();
                System.out.println("count for " + next.key + ": " + next.value);
            }
            range.close();
            Thread.sleep(30000);
        }
    }
}
```

Listing 7.31: A program using a materialized view (`SimpleStreamsExercisesc.java` file).

```
count for 355:5325

count for 960: 14400
count for 961: 14415
count for 962: 14430
count for 963: 14445
count for 964: 14460
count for 965: 14475
count for 966: 14490
count for 967: 14505
count for 968: 14520
count for 969: 14535
count for 97: 1455
count for 970: 14550
count for 971: 14565
count for 972: 14580
count for 973: 14595
count for 974: 14610
count for 975: 14625
count for 976: 14640
count for 977: 14655
count for 978: 14670
count for 979: 14685
count for 98: 1470
count for 980: 14700
```

This seems awkward, because the 98 shows up between the 979 and the 980, but one should keep

in mind that the keys are strings, not integer numbers.

7.3.5 Windowed streams

What if we want to restrict the results to the last couple of minutes? In this case we should do as follows:

```
package streams;

import java.io.IOException;
import java.util.Properties;
import java.util.concurrent.TimeUnit;

import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.kstream.TimeWindows;
import org.apache.kafka.streams.kstream.Windowed;


public class SimpleStreamsExercisesd {

    private static final String tablename = "exercises";

    public static void main(String[] args) throws InterruptedException, IOException {
        if (args.length != 2) {
            System.err.println("Wrong arguments. Please run the class as follows:");
            System.err.println(SimpleStreamsExercisesd.class.getName() + " input-topic
                output-topic");
            System.exit(1);
        }
        String topicName = args[0].toString();
        String outtopicname = args[1].toString();

        java.util.Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "exercises-application-d");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            Serdes.Long().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, Long> lines = builder.stream(topicName);

        KTable<Windowed<String>, Long> addvalues = lines.
```

```
          groupByKey().
          windowedBy(TimeWindows.of(TimeUnit.MINUTES.toMillis(1))).
          reduce((aggval, newval) -> aggval + newval, Materialized.as("lixo"));
        addvalues.toStream((wk, v) -> wk.key()).map((k, v) -> new KeyValue<>(k, "" + k +
            "-->" + v)).to(outtopicname, Produced.with(Serdes.String(),
            Serdes.String()));

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        System.out.println("Reading stream from topic " + topicName);
    }
}
```

Listing 7.32: A windowed stream (`SimpleStreamsExercisesd.java` file).

We are basically applying a window of 1 minute to the results, and therefore we may get:

```
988 => 988
989 => 989
990 => 990
991 => 991
992 => 992
993 => 993
994 => 994
995 => 995
996 => 996
997 => 997
998 => 998
999 => 999
```

In fact several variants of windows exist, but we will not cover them here.

Chapter 8

Enterprise Archives

Goals
– Create an enterprise archive application, composed of different types of modules. Namely, we will be integrating JPA, EJB, a REST service, and a web application.

In this Chapter, we will deploy an EAR application, with a JPA module, an EJB module, a web module using servlets, and a web module containing a rest service. While there is nothing completely new, which we did not cover in the previous chapters, creating the complete application bundle tends to be difficult, because we need to create the necessary project archetypes and appropriately configure their structure and respective maven files.

8.1 Setting Up the Project

Setting up the project is a multi-step work, as no Maven archetype will create the project structure exactly as we would like. We first set up a simple structure for an aggregator project named `jeeapp` using the `maven-archetype-j2ee-simple` archetype:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-j2ee-simple -DarchetypeVersion=1.4
-DgroupId=book -DartifactId=jeeapp  -Dversion=1 -DinteractiveMode=false
```

Listing 8.1: Maven archetype for a complete JEE application.

Now, step inside the generated `jeeapp` directory and run the following commands to create new additional modules named jpa, web, and rest.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=jpa -Dversion=1 -DinteractiveMode=false
```

Listing 8.2: Creating the jpa module with maven.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
    -DgroupId=book -DartifactId=web -Dversion=1 -DinteractiveMode=false
```

Listing 8.3: Creating the web module with maven.

```
    mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4 -DgroupId=book
    -DartifactId=rest -Dversion=1 -DinteractiveMode=false
```

Listing 8.4: Creating the rest module with maven.

We will now replace the main `pom.xml` file that maven generated at the root of the `jeeapp` project, with the following one:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>book</groupId>
  <artifactId>jeeapp</artifactId>
  <version>1</version>
  <packaging>pom</packaging>
  <name>Jakarta EE Application</name>
  <modules>
    <module>ejbs</module>
    <module>jpa</module>
    <module>ear</module>
    <module>rest</module>
    <module>web</module>
  </modules>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>16</maven.compiler.release>
    <wildfly-plugin-version>2.1.0.Beta1</wildfly-plugin-version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>book</groupId>
        <artifactId>ejbs</artifactId>
        <version>1</version>
        <type>ejb</type>
      </dependency>
      <dependency>
        <groupId>book</groupId>
        <artifactId>jpa</artifactId>
        <version>1</version>
        <type>jar</type>
      </dependency>
      <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
```

```
      <version>8.0.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<build>
<finalName>${project.artifactId}</finalName>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <version>${wildfly-plugin-version}</version>
        <configuration>
          <skip>false</skip>
          <hostname>wildfly</hostname>
          <port>9990</port>
          <filename>${project.artifactId}.ear</filename>
          <username>admin</username>
          <password>admin#7rules</password>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
    <plugin>
      <artifactId>maven-ear-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <!-- Use simple names for ear packaging -->
        <outputFileNameMapping>@{artifactId}@.@{extension}@</outputFileNameMapping>
        <defaultLibBundleDir>lib</defaultLibBundleDir>
        <outputDirectory>${output}</outputDirectory>
      </configuration>
    </plugin>
      <plugin>
        <artifactId>maven-ejb-plugin</artifactId>
        <version>3.0.1</version>
      </plugin>
      <plugin>
```

```
          <artifactId>maven-deploy-plugin</artifactId>
          <version>2.8.2</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 8.5: The main `pom.xml`

We can now delete the unnecessary directories of the modules that were generated by the `j2ee-simple` archetype, namely `jeeapp/primary-source`, `jeeapp/projects`, and `jeeapp/servlets`. Indeed we do not need the former two and we are now using a simpler layout for the two webapp modules generated (i.e., `rest` and `web`) and can remove the latter one also. Indeed, a major challenge of this example is precisely tuning maven! Notice that the root `pom.xml` now reflects all of these changes and references the right modules.

In the `dependencyManagement` section we changed the modules according to the changes in the structure, i.e., considering the inclusion/exclusion of the abovementioned modules. We also added a few configurations, which you should look for in the `pom`, like the compiler version used, the final build name (in tag `<finalName>`), the configuration of the `ear` plugin which references: an output directory, a directory for libraries that need to be shared accross the `ear`, and the use of simple names for the files to be included in the `ear`, and the configuration of the maven wildfly plugin, in case you wish to use it.

At this point, the ear module and the ejbs module still hold references to deleted dependencies, which we will correct in the next sections. For the time being, we should check if the project compiles, for which we need to eliminate unneeded dependencies, as follows. Open `ejbs/pom.xml` and delete the `primary-source` and `logging` dependencies. Now open the `ear/pom.xml` and delete dependencies `primary-source`, `logging`, and `servlet`. At the root of the jeeapp project, we should be able to now run `mvn clean package` and see success messages.

8.2 The Enterprise Application Archive Module

We now will configure the EAR module, which is mostly responsible for identifying the modules that will compose the EAR assembly that will be deployed on WildFly. Replace the generated `ear/pom.xml` by the following one:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>book</groupId>
    <artifactId>jeeapp</artifactId>
    <version>1</version>
  </parent>
  <artifactId>ear</artifactId>
  <packaging>ear</packaging>
  <name>ear assembly</name>
  <dependencies>
```

```
    <dependency>
      <groupId>book</groupId>
      <artifactId>jpa</artifactId>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>book</groupId>
      <artifactId>ejbs</artifactId>
      <type>ejb</type>
    </dependency>
    <dependency>
      <groupId>book</groupId>
      <artifactId>web</artifactId>
      <type>war</type>
      <version>1</version>
    </dependency>
    <dependency>
      <groupId>book</groupId>
      <artifactId>rest</artifactId>
      <version>1</version>
      <type>war</type>
    </dependency>
  </dependencies>
</project>
```

Listing 8.6: The EAR `pom.xml`

Notice that this pom file basically identifies the units that will be used to build the EAR file. Again, we may check if the project compiles correctly by running `mvn clean package`.

8.3 The JPA Module

In case you are not following the containerized setup, you will need to add the database driver to WildFly and define a datasource to be used by applications. Let's begin by downloading the PostgreSQL JDBC driver at `https://jdbc.postgresql.org/download.html`, make sure you download the right version for your configuration.

Go to the WildFly Home directory and find the `modules/system/layers/base/org` folder. Create two new folders, so you get the following structure:

`modules/system/layers/base/org/postgresql/main`

Now copy the PostgreSQL JDBC driver .jar file to the abovementioned `main/` folder. In this same folder create a file named `module.xml` with the following content (set in `<resource-root path="" >` the exact name of your .jar file):

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="org.postgresql">
  <resources>
    <resource-root path="postgresql-42.2.23.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
```

```
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

Listing 8.7: The PostgreSQL module.xml file in WildFly.

Edit WildFly's `standalone-full.xml` and add the following code between the `<drivers></drivers>` tags.

```
<driver name="postgresql" module="org.postgresql">
    <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-class>
</driver>
```

Listing 8.8: The PostgreSQL JDBC driver description WildFly.

Now open a command line window and go to the Home directory of WildFly. Start the server by executing `bin\standalone.bat --server-config=standalone-full.xml` in Windows systems or by executing `bin/standalone.sh - -server-config=standalone-full.xml` in Linux or macOS. Check if WildFly is loading the driver, the console should show something like:

```
INFO [org.jboss.as.connector.subsystems.datasources] (ServerService Thread Pool -- 47)
    WFLYJCA0005: Deploying non-JDBC-compliant driver class org.postgresql.Driver
    (version 42.2)
INFO [org.jboss.as.connector.deployers.jdbc] (MSC service thread 1-1) WFLYJCA0018:
    Started Driver service with driver-name = postgresql
```

Listing 8.9: WildFly deploying and starting the JDBC driver.

Let us now add a datasource in standalone-full.xml. Add the following code between the <datasources></datasources> tags, right after the pre-configured datasource that already exists in the file (set your connection-url, username and be sure to set the password you are using, jndi-name can be java:/postgresDS.

```
<datasource jndi-name="java:/postgresDS" pool-name="postgresDS" enabled="true" jta="true"
    use-java-context="true" use-ccm="false">
  <connection-url>jdbc:postgresql://localhost:5432/postgres</connection-url>
      <driver-class>org.postgresql.Driver</driver-class> <driver>postgresql</driver>
  <pool>
    <min-pool-size>2</min-pool-size>
    <max-pool-size>20</max-pool-size> </pool>
    <security>
      <user-name>postgres</user-name>
      <password>postgres</password>
    </security>
    <validation>
      <validate-on-match>false</validate-on-match>
          <background-validation>false</background-validation>
          <background-validation-millis>60000</background-validation-millis>
    </validation>
    <statement>
      <prepared-statement-cache-size>0</prepared-statement-cache-size>
      <share-prepared-statements>false</share-prepared-statements>
    </statement>
```

```
</datasource>
```

Listing 8.10: The definition of the datasource in WildFly.

If you restart the server you should now be able to see a message like:

```
INFO [org.jboss.as.connector.subsystems.datasources] (MSC service thread 1-3)
    WFLYJCA0001: Bound data source [java:/postgresDS]
```

Listing 8.11: WilfFly binding the new datasource.

Before writing the code, let us replace the contents of the `jpa/pom.xml` with the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>jeeapp</artifactId>
    <groupId>book</groupId>
    <version>1</version>
  </parent>
  <artifactId>jpa</artifactId>
  <name>jpa</name>
  <dependencies>
    <dependency>
      <groupId>jakarta.platform</groupId>
      <artifactId>jakarta.jakartaee-api</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 8.12: The jpa `pom.xml` file.

This project will contain a single entity, representing Students, defined as follows, and which will be placed in `src/main/java/data/Student.java`:

```java
package data;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 * Entity implementation class for Entity: Student
 *
 */
@Entity
public class Student implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;
  private String name;

  public Student() {
    super();
  }

  public Student(String name) {
    this.name = name;
  }


  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

}
```

Listing 8.13: The Student entity.

Finally, we also need to define the following `persistence.xml` file, which we will place under `jpa/src/main/resources/META-INF/`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="playAula" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>java:/postgresDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.PostgreSQLDialect" />
      <!-- table generation policies: validate, update, create, create-drop -->
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.transaction.jta.platform"
          value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform"
          />
    </properties>
  </persistence-unit>
</persistence>
```

Listing 8.14: Contents of the persistence.xml file.

Although it is not really functional yet, you may start WildFly and try to deploy the project. You will find the ear file in WildFly's deployment directory. Once deployment occurs, you should see a message announcing the initialization of your `playAula` persistence unit, similar to the following:

```
11:12:26,450 INFO  [org.jboss.as.jpa] (ServerService Thread Pool -- 85) WFLYJPA0010:
    Starting Persistence Unit (phase 1 of 2) Service 'jeeapp.ear#playAula'
11:12:26,480 INFO  [org.hibernate.jpa.internal.util.LogHelper] (ServerService Thread Pool
    -- 85) HHH000204: Processing PersistenceUnitInfo [
  name: playAula
  ...]
```

Listing 8.15: WilfFly starting the `playAula` persistence unit.

8.4 The Enterprise JavaBeans Module

Let us now define one EJBs that will expose two operations, namely adding students and retrieving the list of students in the database. We begin by replacing the contents of `ejbs/pom.xml` with the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <parent>
    <groupId>book</groupId>
    <artifactId>jeeapp</artifactId>
    <version>1</version>
  </parent>
  <artifactId>ejbs</artifactId>
  <packaging>ejb</packaging>
  <name>enterprise java beans</name>
  <dependencies>
    <dependency>
      <groupId>book</groupId>
      <artifactId>jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>jakarta.platform</groupId>
      <artifactId>jakarta.jakartaee-api</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 8.16: The EJB `pom.xml`.

Notice the dependency on the previously created JPA project. Now delete the `META-INF` directory, which is under `src/main/resources` and contains an unneeded EJB deployment descriptor file (i.e., `ejb-jar.xml`). We now move on to the actual code. Unlike the example of Chapter 3, where we needed a remote EJB, to make accesses possible from other hosts, in this case, the web and REST layers are accessing the bean from the same host. The EJB can, therefore, be local. We may also discard the need for an interface, as the public methods will be automatically exposed. The implementation will go like this:

We begin by creating the EJB Interface:

```
package beans;

import java.util.List;

import javax.ejb.Remote;

import data.Student;
```

```
@Remote
public interface IManageStudents {
  public void addStudent(String name);
  public List<Student> listStudents();
}
```

Listing 8.17: The EJB interface file.

```
package beans;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

import data.Student;

@Stateless
public class ManageStudents implements IManageStudents {

  @PersistenceContext(unitName = "playAula")
  EntityManager em;

  public void addStudent(String name) {
    System.out.println("Adding student " + name + "...");
    Student s = new Student(name);
    em.persist(s);
  }

  public List<Student> listStudents() {
    System.out.println("Retrieving students from the database...");
    TypedQuery<Student> q = em.createQuery("from Student s", Student.class);
    List<Student> list = q.getResultList();
    return list;
  }
}
```

Listing 8.18: The EJB implementation that offers the operations for managing students.

You can now test the deployment on WildFly by running `mvn clean package` and should be able to see the EJB JNDI names being announced in the console, as follows:

```
INFO  [org.jboss.as.ejb3.deployment] (MSC service thread 1-3) WFLYEJB0473: JNDI bindings
    for session bean named 'ManageStudents' in deployment unit 'subdeployment "ejbs.jar"
    of deployment "jeeapp.ear"' are as follows:

  java:global/jeeapp/ejbs/ManageStudents!beans.IManageStudents
  java:app/ejbs/ManageStudents!beans.IManageStudents
  java:module/ManageStudents!beans.IManageStudents
```

```
  java:jboss/exported/jeeapp/ejbs/ManageStudents!beans.IManageStudents
  ejb:jeeapp/ejbs/ManageStudents!beans.IManageStudents
  java:global/jeeapp/ejbs/ManageStudents
  java:app/ejbs/ManageStudents
  java:module/ManageStudents
```

Listing 8.19: WilfFly announcing the JNDI bindings for our session bean.

8.5 The REST module

We now move on to the rest module and begin by replacing the generated `rest/pom.xml` file with the following one:

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

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>book</groupId>
    <artifactId>jeeapp</artifactId>
    <version>1</version>
  </parent>
  <artifactId>rest</artifactId>
  <packaging>war</packaging>
  <name>rest services</name>
  <dependencies>
    <dependency>
      <groupId>jakarta.platform</groupId>
      <artifactId>jakarta.jakartaee-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>book</groupId>
      <artifactId>ejbs</artifactId>
      <version>1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
```

```
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 8.20: The rest module `pom.xml`.

We now create the code for the ApplicationConfig file, under `src/main/java/book`, as follows:

```
package book;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/services")
public class ApplicationConfig extends Application
{

}
```

Listing 8.21: The ApplicationConfig file.

You can safely delete the `WEB-INF/web.xml` file under `src/main/webapp/` and also the `index.jsp` file, which we will not be using. We will now create a service with a test operation, an operation allowing to add a student to the database, and an operation to list students. The code for the service is the following one:

```
package book;

import java.sql.Time;
import java.util.Calendar;
import java.util.List;

import javax.ejb.EJB;
import javax.enterprise.context.RequestScoped;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import beans.IManageStudents;
import data.Student;

@RequestScoped
@Path("/myservice")
@Produces(MediaType.APPLICATION_JSON)
public class MyService {

  @EJB
  private IManageStudents manageStudents;

  @GET
  @Path("/test")
```

```
  public String method1() {
    System.out.println("M1 executing....");

    return "M1 executed...";
  }

  @GET
  @Path("/add")
  public String method2() {
    System.out.println("M2 executing....");
    String name = "name_" + new Time(Calendar.getInstance().getTimeInMillis());
    manageStudents.addStudent(name);

    return name;
  }

  @GET
  @Path("/list")
  public List<Student> method3() {
    System.out.println("M3 executing....");
    List<Student> list = manageStudents.listStudents();

    return list;
  }
}
```

Listing 8.22: The REST service code file.

After deploying the EAR file and starting WildFly you may find the following information at the console:

```
INFO  [org.jboss.resteasy.resteasy_jaxrs.i18n] (ServerService Thread Pool -- 88)
    RESTEASY002225: Deploying javax.ws.rs.core.Application: class book.ApplicationConfig
INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 88) WFLYUT0021:
    Registered web context: '/rest' for server 'default-server'
```

Listing 8.23: WilfFly deploying the REST service and annoucing the registration of the /rest web context.

We can test the service operations by pointing a browser to the following URLs.

`http://localhost:8080/rest/services/myservice/test`

`http://localhost:8080/rest/services/myservice/add`

`http://localhost:8080/rest/services/myservice/list`

8.6 The web Module

We begin by replacing the generated `web/pom.xml` with the following one:

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>book</groupId>
    <artifactId>jeeapp</artifactId>
    <version>1</version>
  </parent>
  <artifactId>web</artifactId>
  <packaging>war</packaging>
  <name>webapp servlets</name>
  <dependencies>
    <dependency>
      <groupId>jakarta.platform</groupId>
      <artifactId>jakarta.jakartaee-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>book</groupId>
      <artifactId>ejbs</artifactId>
      <version>1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.wildfly.plugins</groupId>
          <artifactId>wildfly-maven-plugin</artifactId>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Listing 8.24: The web module `pom.xml`.

This module will contain a single servlet, placed under `src/main/java/servlet` as follows.

```
package servlet;

import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

import javax.ejb.EJB;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import beans.IManageStudents;
import data.Student;

@WebServlet("/webaccess")
public class MyServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  @EJB
  private IManageStudents manageStudents;

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

    List<String> field1List = manageStudents.listStudents().stream().
      map(Student::getName).collect(Collectors.toList());

    String result = "Students list: " + field1List;

    System.out.println(result);
    response.getWriter().print(result);
  }
}
```

Listing 8.25: A simple servlet.

Now delete the unneeded `webapp/WEB-INF/web.xml` file and deploy the EAR. You will see the web context being registered, as follows:

```
INFO  [org.wildfly.extension.undertow] (ServerService Thread Pool -- 93) WFLYUT0021:
    Registered web context: '/web' for server 'default-server'
```

Listing 8.26: WilfFly announcing the registration of the /web web context.

The servlet will be accessible via `GET` at: `http://localhost:8080/web/webaccess` and will print and return the list of student names currently in the database.