Maven, Pizza Project

Using maven to build a prototype setup for pizza1

From tutorial from [https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html](https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html)

mvn archetype:generate
-DgroupId=com.mycompany.app
-DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4
-DinteractiveMode=false

Edited to use pizza1 artifactId and cs636.pizza groupId--

mvn archetype:generate
-DgroupId=cs636.pizza
-DartifactId=pizza1
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4
-DinteractiveMode=false

 We run this command to build the directory system that Maven wants us to have. Maven is "opinionated", i.e., has specific ideas on how to set up the project.
 This does make it easier to understand new project once you get used to it.

Resulting project, available at $cs636/maven-ex

Try out this project: mvn = maven tool

mvn package    compiles files, runs unit tests, builds jar file

Try out this project: mvn = maven tool

mvn package    compiles files, runs unit tests, builds jar file

Put's results in target directory: let's see what files are there...

-classes generated-test-sources maven-surefire-reports
generated-sources maven-archiver pizza1-1.0-SNAPSHOT.jar test-classes
classes contains the main class directory tree
test-classes contains the test class dir tree
pizza1-1.0-SNAPSHOT.jar is the jar file we can use to execute the app

Runttime app: put the jar on the classpath and execute cs636.pizza.App within it:
java -cp target/pizza1-1.0-SNAPSHOT.jar cs636.pizza.App

Hello World!

Look in the jar file

META-INF/MANIFEST.MF
META-INF/maven/...
classpath/csd636/pizza/csd636.pizza/pizza1/pom.properties
META-INF/maven/cs636/pizza/pizza1/pom.xml

From last class: Pizza1 directories/packages

Here's the directory system for the pizza1 sources (using "tree" command in Windows), looking down from the top of the source directory tree, itself at src/main/java in the project's directory system.

C:\cs636\pizza1\src\main\java

Folder PATH listing
Volume serial number is 385D-9017

C:
  └───cs636
      └───pizza
        ├───config
        │    └───for package cs636.pizza.config
        ├───dao
        │    └───for package cs636.pizza.dao, etc.
        └───domain
            ├───presentation
            └───service

How do we build this from scratch?

We run this command to build the directory system that Maven wants us to have. Maven is "opinionated", i.e., has specific ideas on how to set up the project.

This does make it easier to understand new project once you get used to it.

We run this command to build the directory system that Maven wants us to have. Maven is “opinionated” , i.e., has specific ideas on how to set up the project.

This does make it easier to understand new project once you get used to it.

This sets up a standard Maven project, so the sources root is at src/main/java relative to the base directory of the project.

Using Maven, we are expected to put the test sources in a separate tree from the regular sources -- they are rooted at src/test/java, with the same tree structure for the packages.

To finish this for the real project, we need to cd down to the pizza directory and create directories for presentation, service, dao, etc. But for now, just try out App.java…

Run the app: put the jar on the classpath and execute cs636.pizza.App within it:
java -cp target/pizza1-1.0-SNAPSHOT.jar cs636.pizza.App

Hello World!

Look in the jar file

META-INF/MANIFEST.MF
META-INF/maven/...
classpath/csd636/pizza/csd636.pizza/pizza1/pom.properties
META-INF/maven/cs636/pizza/pizza1/pom.xml

• This sets up a standard Maven project, so the sources root is at src/main/java relative to the base directory of the project.
• Using Maven, we are expected to put the test sources in a separate tree from the regular sources -- they are rooted at src/test/java, with the same tree structure for the packages.
• To finish this for the real project, we need to cd down to the pizza directory and create directories for presentation, service, dao, etc. But for now, just try out App.java…

java -cp target/pizza1-1.0-SNAPSHOT.jar cs636.pizza.App

Hello World!
Maven and running the program

We see that out of the box, maven doesn't support running the main class with a `mvn` command. It can be made to do so. See https://stackoverflow.com/questions/1089285/maven.

There is a `mvn` command for running unit tests: `mvn test`. To do this, maven must use junit.jar, but it's nowhere to be found in our project directories. It's in maven's local repository.

We'll have these entries to ensure current-enough sample, and test execution support.

maven clean

Another important maven command: `mvn clean`. It deletes the whole target directory, where the classes and jars were built. Use `mvn clean package` to do a clean build.

Pizza1 with the actual sources

The first step is studying the supplied Pizza1 project, the example to guide our own development project.

Also the pizza project "Project Description", linked to the class web page.

We use the layered architecture discussed in the first class. We see packages for presentation, service, and data, corresponding to the layers.

The package `domain` has classes PizzaOrder, PizzaTopping, and PizzaOrder, Menu and MenuTopping. These represent the needed domain objects, the subjects of the application: so the domain package does not have its own layer, though is associated closely with the service layer.

The domain objects are stored in the data in the service layer, the heart of the application. They also are used in the DAO layer, in a more technical way, to allow the data to be persisted in the database.

Now we see we have box packages for each layer: presentation, service, data, domain package objects for PizzaOrder, etc. coding package: setting up the system.

Real pizza1 pom.xml

We see, compared to the prototype pom.xml:

- Similar elements at the start of the file, identifying the project
- Many more dependencies:
  - One for each of our databases, for the needed drivers
  - It needs an entry in the repositories because the driver is not in the standard Maven repository
  - JUnit5 instead of JUnit4: this is to get ready for Spring Boot 2, which assumes v5 unless you override it
- Different plugins, that is, tools to build or execute things
  - Two-plugin version specs from the proto, for maven-compiler and maven-surefire (for Junit tests)
- There's an additional plugin spec for building a "fat jar", that is, a jar with all the dependencies in it.
Dependencies: see structure of multiple dependencies inside one collection element:

From https://maven.apache.org/pom.html#Quick_Overview: The cornerstone of the POM is its dependency list. Most projects depend on others to build and run correctly. If all Maven does for you is manage this list, you have gained a lot.

This project needs to use the JUnit library version 4.12 for unit testing, a "dependency". JUnit in turn needs other libraries, but Maven takes care of that for us.

Pizza1: The layers and their APIs

- Recall that the layers are "call-down" layers: the presentation layer code calls the service API and thus into the service layer. The service layer code calls the DAO API, and the DAO code does JDBC for pizza1.
- The presentation layer code never calls the DAO directly.
- And nothing ever calls methods of a higher layer. So this is strict layering. That helps with understanding the system and debugging it.
- The service layer API is the most important one, defining the actions of the application.
- We can extract the API by using the fact that all these methods are public, and very little else in the Java source is public. We just cd to the directory of the service package and search the files using "find" or "grep".

Continued: StudentService (full list)

--- STUDENTSERVICE.java

```java
public class StudentService {
    public StudentService(PizzaOrderDAO poDAO, MenuDAO mDAO, AdminDAO adDAO) {
        // constructor
    }
    public void setSizeNames(String name) throws ServiceException {
        // set size names
    }
    public void setToppingNames(String name) throws ServiceException {
        // set topping names
    }
    public Set<String> getSizeNames() throws ServiceException {
        // get size names
    }
    public Set<String> getToppingNames() throws ServiceException {
        // get topping names
    }
    public void makeOrder(int roomNum, String sizeName, Set<String> toppingNames) {
        // make order
    }
    public List<PizzaOrderData> getOrderStatus(int roomNumber) throws ServiceException {
        // get order status
    }
    public void receiveOrders(int roomNumber) throws ServiceException {
        // receive orders
    }
}
```

The method names (in bold) give names for the actions here, the student-related actions of the app.
Understanding the layers in use
Answer: in the presentation layer; in the presentation package.
How does makeOrder do its work, that is, get an order into the database?
Answer: by calling methods in the DAO layer, i.e., calling the DAO API
How can we find the DAO API?
Answer: by cd ../dao, search again for “public”

The layers in use
Answer: by calling methods in the DAO layer, i.e., calling the DAO API
How can we find the DAO API?
Answer: by cd ../dao, search again for “public”

The types seen in the service API
- public Set<String> getSizeNames() throws ServiceException
- public Set<String> getToppingNames() throws ServiceException
- public void makeOrder(int roomNum, String sizeName, Set<String> toppingNames)
- public List<
  PizzaOrderData>
  getOrderByStatus(int roomNumber) throws ServiceException
- public void receiveOrders(int roomNumber) throws ServiceException

- The topping names and size names are provided to the presentation layer as simple Strings. That way, no domain objects are “leaked” into the UI code in presentation.
- The PizzaOrder information is provided using PizzaOrderData objects created from PizzaOrder objects just for the purpose of carrying the data to the presentation code.
- Each method throws ServiceException, an application-defined subclass of Exception—we should explore this further soon.

More on call-down layers
Why would you want to call upwards?
• For notifications. Consider when the pizza is ready: the admin action “this pizza is ready” calls down and changes the database. If this change triggered a notification to the student user, that would need an upcall to the presentation layer.
• Such notifications can be done in say a Swing application [Java GUI program running on the user’s machine], because the Swing app owns the screen. It can pop up a little note on the screen.
• But we are designing for an (eventual) web app, where the service layer runs in the server and doesn’t own the clients’ screens. The clients use HTTP to ask questions and make commands.
• So the client has to explicitly ask “is the pizza done?” and this request is handled in a call-down manner.

Web apps using HTTP are request-driven
• That’s the simple web app way. We could explore ways to notify [WebSockets for example], but it is quite difficult and requires app code on the client side, usually Javascript. We need to keep things simple to get where we want to go in one form.
• Consider that from the server’s standpoint, the clients are often hidden behind firewalls—how can you send them information at some arbitrary moment?
• They don’t have well-known network addresses! The chance to send them information is when they send a request in to you (the server) and you get to reply to it.
• Therefore, we will stick to the simple call-down layers.
• Next time—look at database for pizza1.

Eclipse and Maven
• When you create a new eclipse project, say with Open Project from File System, it recognizes the Maven directory structure and pom.xml
• You see a little M on the project, as well as a little J for Java
• Note that every time we edit pom.xml, we need to tell eclipse by right-clicking the project’s icon in Project Explorer, selecting Maven and then Update Project.
• Eclipse has a special UI for the pom, showing the dependency hierarchy and the effective pom, as well as the plain source file
• Eclipse uses the local repository of jar files for populating the classpath for the project