Robotic Factory Exercise 3
Microservices and APIs

Dominique Blouin, Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

For this third exercise, you will first implement a simple Hello World micro service using the Spring Boot framework. Next, you will modify the robotic factory simulator to create a microservice to run the simulation of the factory model on a remote server. The simulator user interface running on your local machine will then connect to the remote simulator to display the factory model being simulated.

Implementing a "Hello World!" Microservice

You will use the Spring Boot framework as introduced during the lecture to implement a simple "Hello World!" micro service.

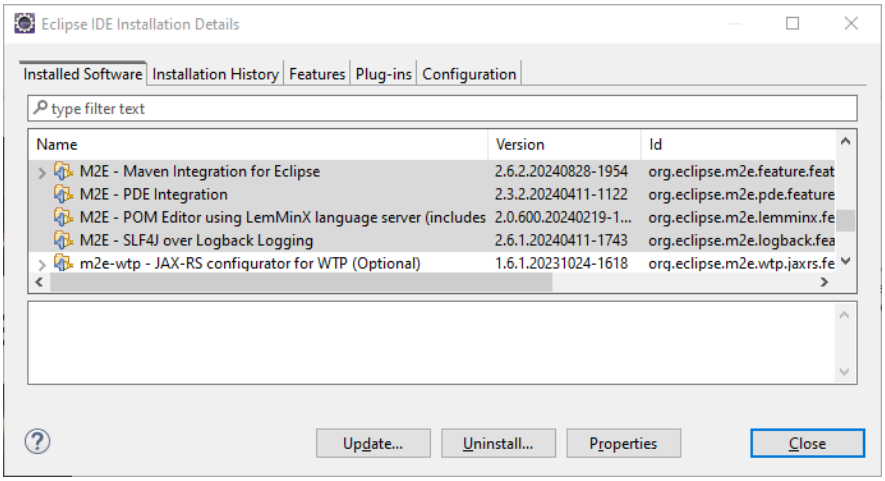
Installing the Required Tools

As seen during the lecture, Spring Boot relies on many other tools so the first step will be to install in your Eclipse IDE the required plugin extensions.

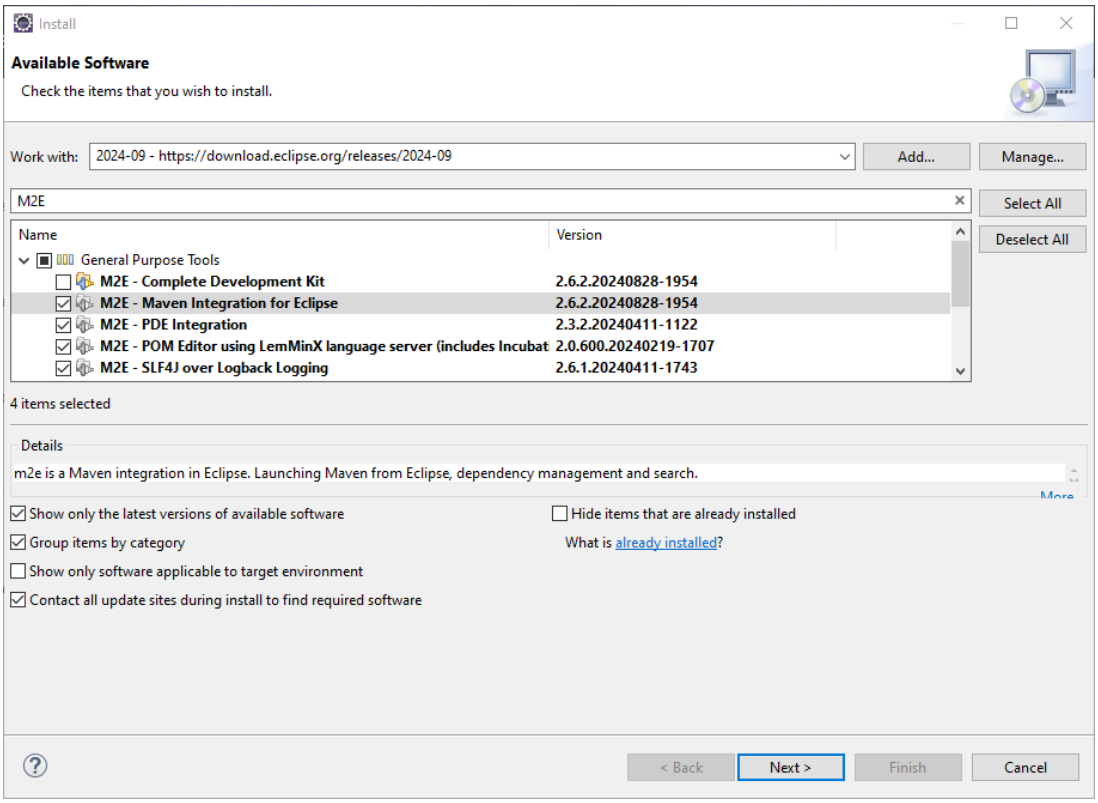
M2E (Maven)

So far you have been compiling your Java projects with the Eclipse JDT builder, which takes care of compiling and running your Java applications using the *javac* and *java* executables from the JRE. This works well for simple projects but for large applications, other tools such as Maven or Gradle are used. A main advantage of these tools is to automatically fetch the often-numerous external Java libraries required by your application code from a list of repositories.

We will use Maven to build our applications. First check if the M2E extension is installed into your IDE. To do so, select menu *Help>>About*. From the window that pops up, click button *Installation Details*. In the dialog box that shows up, check that the following M2E features are installed as illustrated by the screenshot below.



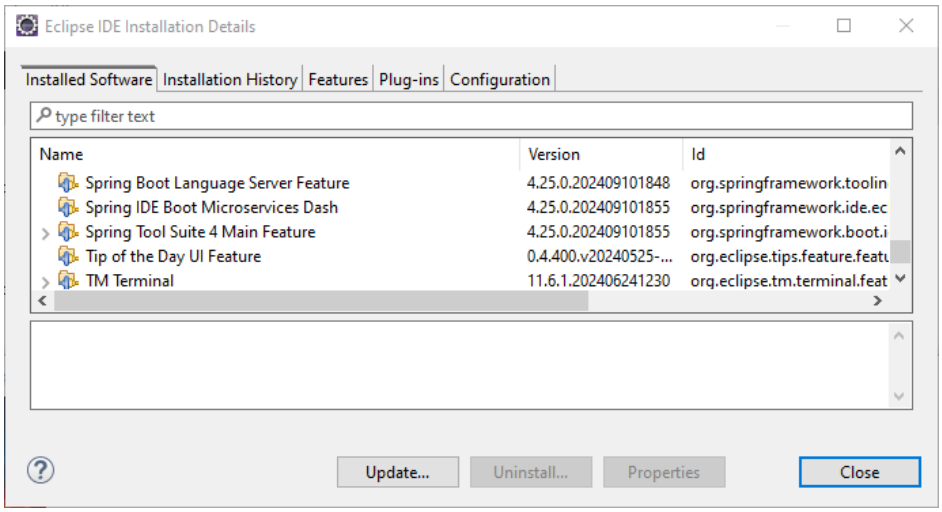
If M2E is not installed, click menu *Help>>Install New Software...*. From the dialog box that pops up, select the update site of your Eclipse release and check the M2E features as illustrated below.



Spring Tools 4

Next, install Spring Tools 4 (ST 4). For this, first select menu *Help>>Eclipse Market Place...* In the window that pops up, search for *Spring*. On the *Spring Tools 4* element, click the button *Install*, accept the defaults and restart Eclipse to install Spring Tools.

Next, check that Spring Tools was properly installed again by selecting menu *Help>>About*. You will also notice that the Spring Boot Microservices extension has also been installed.



Creating and Using Spring Boot Microservices with the Spring Boot Tutorials

In this part of the exercise, you will follow three simple tutorials to get acquainted with Spring Boot and microservices. Those are:

- Spring Quickstart Guide
- Building a RESTful Web Service
- Consuming a RESTful Web Service

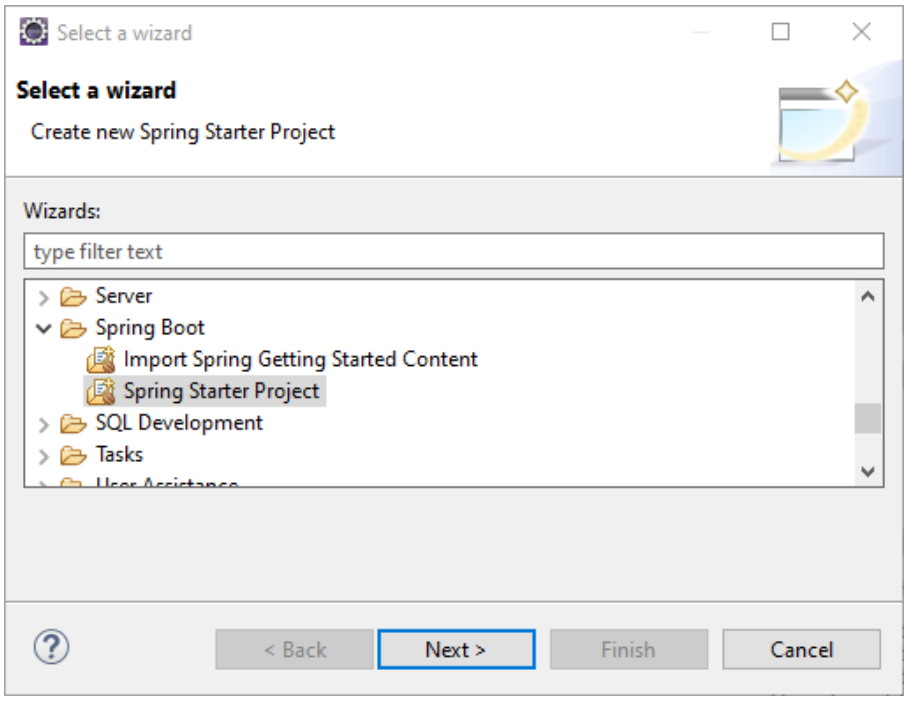
Note that the instructions for these tutorials are given independently of any IDE. If you are using Eclipse Spring Tools, here are ways to perform some of the tutorial tasks directly from the IDE, making them easier to perform.

Creating a Spring Boot Project

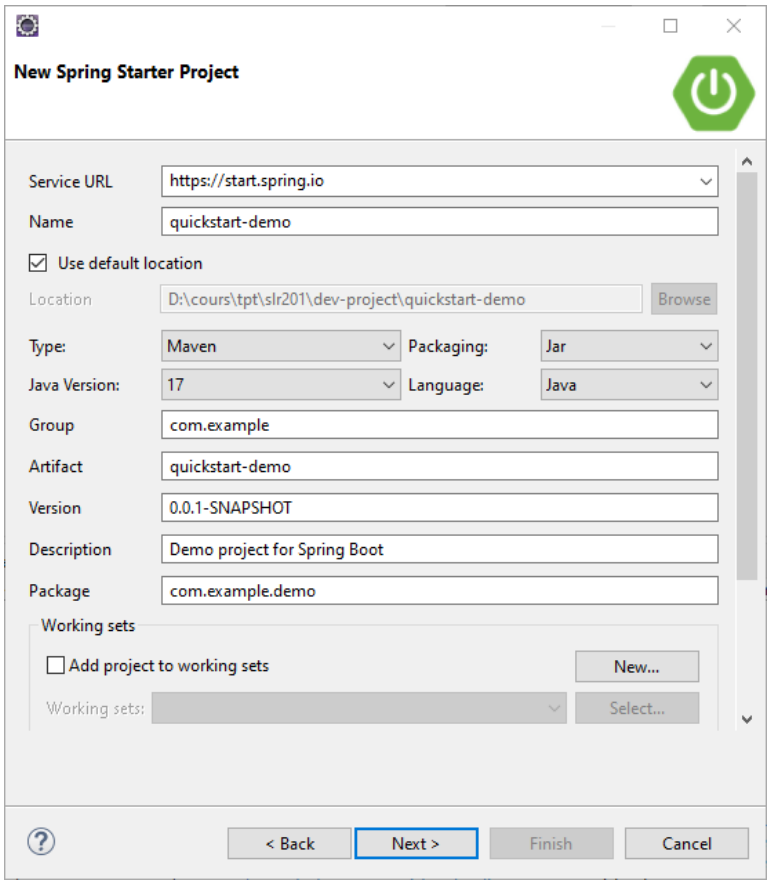
As opposed to standard Java projects that you have used so far, Spring Boot projects are not simple. This is because they require a lot of components such as the Spring libraries themselves, a Java application server such as Tomcat, JSON libraries for the textual format to transfer objects, Jackson to parse / unparsed JSON objects, Jakarta for JPA persistence, etc.

Luckily, the Eclipse ST 4 extension provides a wizard to create these complex projects directly within the IDE instead of having to use the online *spring initializr* tool and then importing the project into the IDE.

To create a Spring Boot project, select menu *File>>New>>Other*. From the dialog box that appears, select *Spring Starter Project* under the *Spring Boot* branch as illustrated in the screenshot below.



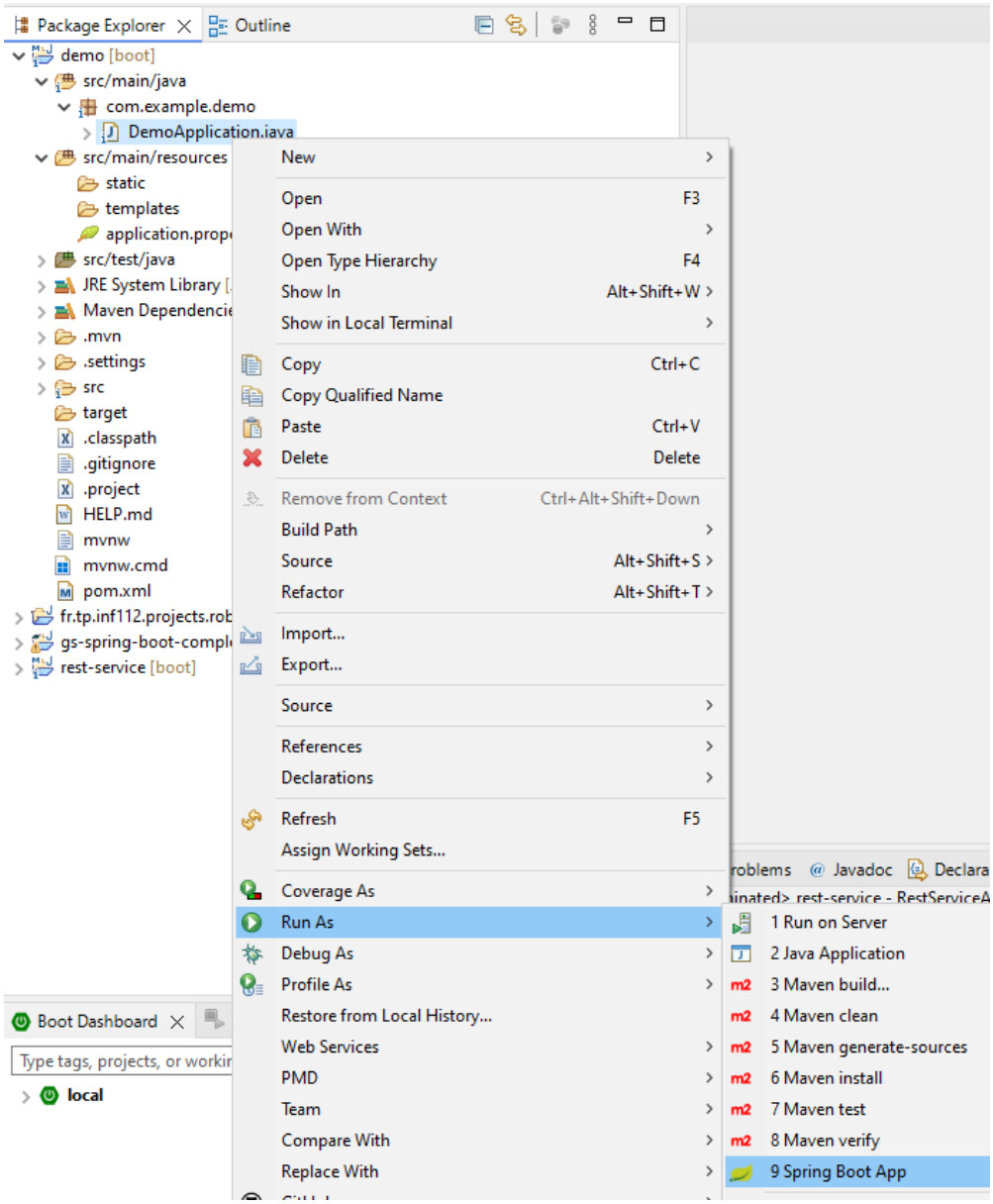
Click *Next* and make sure that the *Maven* builder is selected and check that other options such as the *Java Version* and programming *Language* are correct as shown below.



Click *Next* and make sure that *Spring Web* is selected as project dependency and click *Finish*.

Starting and Stopping the Server

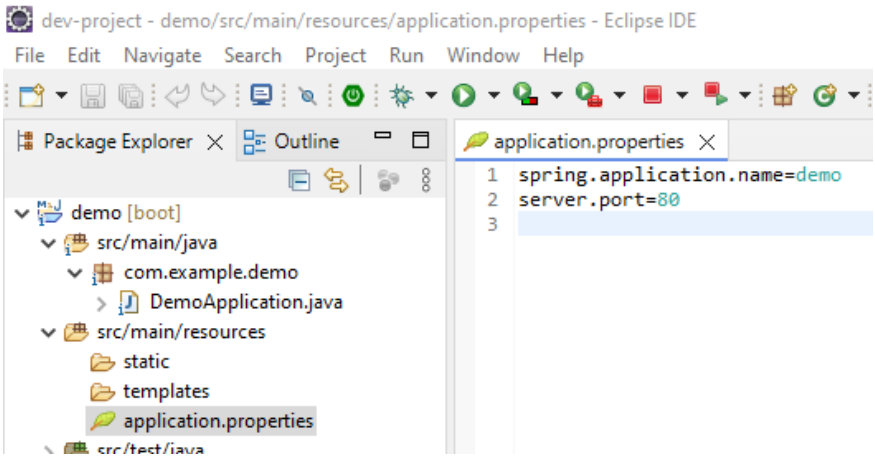
The server onto which the microservices are deployed is embedded into Eclipse, making server applications easier to debug. To start the server, simply select the application class and right-click menu *Run As>>Spring Boot App*.



Check that the server is running properly by looking at the console as shown below. The server can be stopped by clicking the red square button in the upper right corner of the *Console* view.

Changing the Default Spring Boot and Server Configuration Parameters

By default, the Server will be listening for client requests on port 8080. It is possible to change this parameter and many other ones such as the application name by setting their value in a file named *application.properties*, as illustrated in the screenshot below. You can use the CTRL – Space keys to obtain parameter names suggestions via autocompletion.



Implementing a Remote Robotic Factory Simulator as a Microservice

In this part of the exercise, you will modify the robotic factory simulator to distribute the simulation process to a remote server by deploying it into a microservice. As already mentioned during the first exercise, the simulator implements a Model-View Controller design pattern. To distribute the simulation to a remote server, you will need to create a new remote simulator controller class.

The Local Simulator Controller

The simulator controller class is named *SimulatorController* and is in package *fr.tp.inf112.projects.robotsim.app*. It implements the *CanvasViewerController* interface, which provides methods corresponding to the factory viewer menu items to start and stop the simulation and to obtain a factory model to be displayed in the viewer. When the user asks to start the animation from the canvas viewer, this calls the *startAnimation* method on the controller, which in turns calls the *startSimulation* method on the factory model.

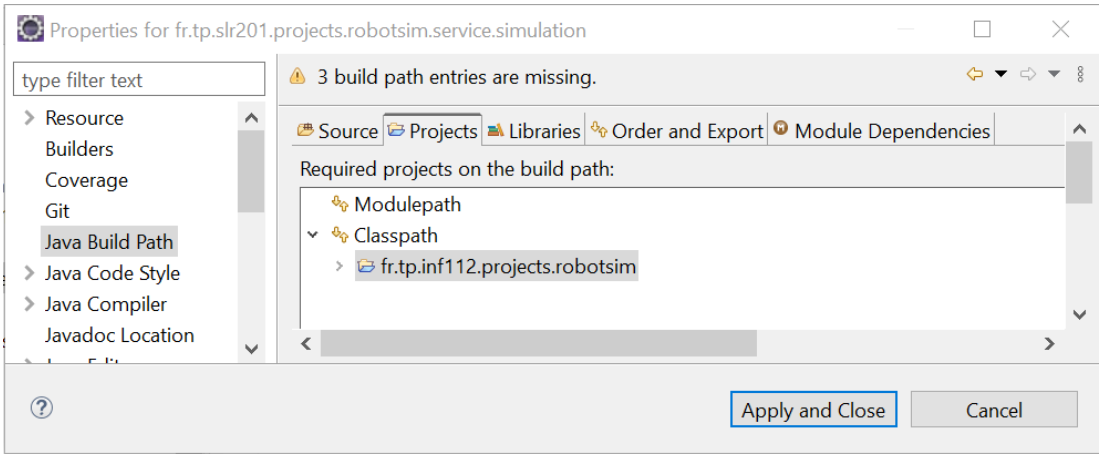
As explained before, the way the viewer knows that the model data has changed and that it needs to refresh its display via the observable-observer design pattern. For this, the viewer first registers itself to the model. Later, whenever the data of the model is changed by calling its modification (setters) methods, this will also place a call to notify the observers (the viewer) that the model data has been changed. In this way, the viewer is notified that it should refresh its display.

Implementing a Simulation Microservice

Inspired by the Spring Boot tutorials that you have followed during the first part of this exercise, create a microservice that will run the simulation on a server.

First create a Spring Boot project named *fr.tp.slr201.projects.robotsim.service.simulation*. Next, as shown in the screenshot below, add the robot factory simulator project to the *Java*

Build Path of your microservice project so that it can use the robotic factory model classes for the simulation.



Create a microservice controller class that will provide the following methods to its clients:

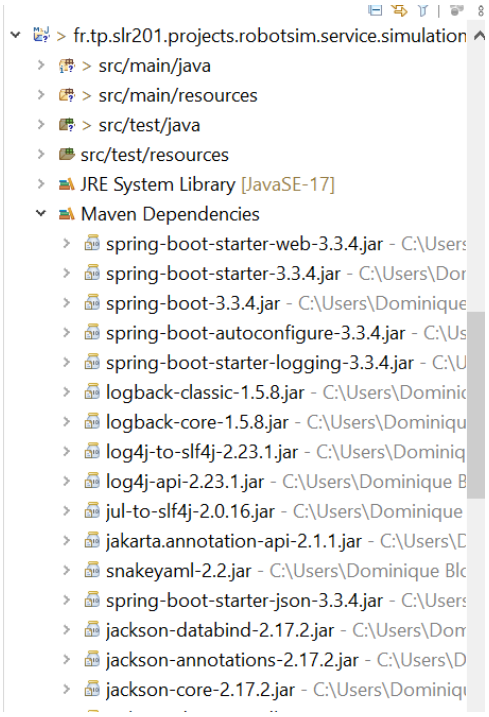
1. Start simulating a factory model as identified by its ID.
    a. The service will call the persistence web server that you created in a previous exercise to read the factory model from it.
    b. It will store this model in a list of models being currently simulated by the simulation server.
    c. It will start the simulation of this factory model.
    d. It will return true if everything went well and false otherwise.
2. Retrieve a factory model currently being simulated as identified by its ID passed as a parameter. This method will be used later by the factory viewer to obtain the simulated model at a given period to be displayed by the viewer showing the fresh model.
3. Stop the simulation of a robotic factory model as identified by its ID passed as parameter.

Add proper logging instructions to your service so that you can easily see if your simulation is working when calling your REST methods from a web browser.

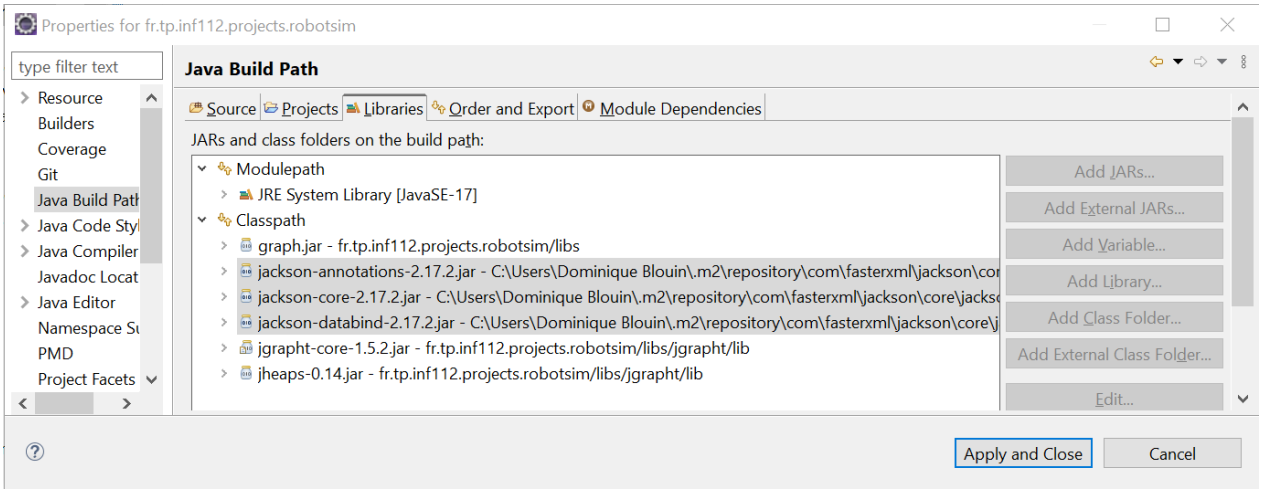
Configuring the Robotic Factory Simulator Project to use Jackson

In a Spring Boot project, when calling a microservice to read an object, parsing the textual JSON data received from the microservice to create the associated object is automatically done by the Spring framework. However, the robotic factory simulator that was given to you at the beginning of the course is not using Sprong Boot. Therefore, to call a microservice from this plain Java application, you will need to explicitly code the parsing of the textual JSON data returned by the microservice. This is done using the Jackson library, which needs to be added to the build path of your project.

To do so, first find out where the Jackson libraries are in your file system. This information can be obtained by looking under the *Maven Dependencies* library folder of your simulation microservice project as illustrated in the screenshot below.



Next, add these libraries to your simulator plain Java project using the *Add External JARs* button as shown below:



Preparing the Factory Model Classes for JSON Serialization

If you tried to serialize into JSON text the current *Factory* model by calling your microservice from a web page, you have certainly encountered several problems. Indeed, serializing into JSON text with Jackson is not as simple as it is with the Java built-in serialization mechanism. This section will help you to refactor the Robotic Factory model classes so that they can be serialized into JSON text.

Default Constructors and Making the Model more Robust

For Jackson serialization to work, every model class is required to have an explicit empty (no parameters) constructor. Therefore, review the model classes to add such constructor. The

constructor can simply call the existing constructor passing null or any default parameter values. In any case, such values will be set again automatically as Jackson parses the object's JSON text.

However, passing null values to the default constructor implies that some methods of the model may need to be modified to consider the fact that these values may be null. An example of this is given in the code snippet below, which has been modified to account for a potential *null* value:

```
@Override
public int getxCoordinate() {
    final PositionedShape shape = getPositionedShape();

    return shape == null ? -1 : shape.getxCoordinate();
}
```

Similarly, review the code of the factory model classes to ensure that no exception will occur when the empty constructors are used by Jackson.

Attributes to be Serialized

By default, Jackson will serialize all class attributes that have a public getter method. Review the attributes of the factory model classes that should be serialized to ensure a getter is available. If for some reason the getter method must be named differently, you can also annotate the getter attribute with the *JsonGetter* annotation as illustrated below:

```
@JsonGetter("simulationStarted")
public boolean getMySimulationStarted() {
    return simulationStarted;
}
```

Transient and Derived attributes that should not be Serialized

Some attributes of the robot factory model are declared to be *transient*, such as the list of observers of the model. Identify throughout all the factory model classes the attributes that are declared to be transient. Use the *JsonIgnore* annotation to tell Jackson not to serialize those attributes as illustrated in the code snippet below:

```
@JsonIgnore
private transient List<Observer> observers;
```

Derived attributes are those whose data is not maintained within the model, but instead computed from the values of other attributes. This means that those attribute values do not need to be serialized.

As mentioned earlier, by default, Jackson automatically identifies a class attribute from the class's getter method, even if there is no corresponding attribute declaration in the class. An example of this is the *getxCoordinate()* method in the *Component* class whose code is given below:

```
@JsonIgnore
@Override
public int getxCoordinate() {
    final PositionedShape shape = getPositionedShape();
```

```
    return shape == null ? -1 : shape.getxCoordinate();
}
```

In this example, Jackson will automatically try to serialize a value for the *xCoordinate* of the component, even though its value will already have been serialized when serializing the *PositionShape* object of the component. To avoid this, add the *JsonIgnore* annotation to the getter method as illustrated in the example code above. Review all the factory model classes to identify the derived attributes and do the same to avoid serializing them.

Bidirectional Relationships

The factory model includes bi-directional relationships, which are relationships that are the inverse of each other. For instance, the *Factory* class includes a reference to a list of contained components. At the same time, the *Component* class includes a reference to its parent factory.

Bidirectional relationships need to be annotated for Jackson to properly manage these cyclic dependencies between classes. To do so, use the *JsonManagedReference* and *JsonBackReference* to annotate these attributes in their class as shown in the example below:

```
public class Factory extends Component implements Canvas, Observable {
    ...
    @JsonManagedReference
    private final List<Component> components;
    ...
}

public abstract class Component implements Figure, Serializable, Runnable {
    ...
    @JsonBackReference
    private final Factory factory;
    ...
}
```

References

See the following references for more information on Jackson serialization:

- Jackson Exceptions
- Jackson Ignore Properties on Serialization
- Jackson Bidirectional Relationships and Infinite Recursion
- Jackson Inheritance

Testing the Jackson JSON Serialization

Modifying the factory model so that it can be serialized into JSON text and testing that the serialization works fine requires some time. To simplify this task, it is worthed creating a dedicated class to test the serialization. It will be much faster to run this lightweight class (with JUnit for example) than running the fully distributed robotic factory simulator.

The created test class will first need to instantiate a Jackson object mapper, which will be responsible to serialize and deserialize the model objects as illustrated in the code below:

```
private final ObjectMapper objectMapper;

public TestRobotSimSerializationJSON() {
```

```
    objectMapper = new ObjectMapper();
    ...
```

However, another tricky aspect of Jackson serialization is class inheritance and polymorphism. For example, the list of components in the *Factory* class is declared to contain elements of the *Component* class. But concretely, because the *Component* class is abstract, the class of the contained components will always be one of the different classes that extend the *Component* class and that are not abstract, such as *Robot*, *Door*, *Room*, etc.

By default, the Jackson object mapper will not serialize the specific subclasses unless it is explicitly told to do so. This can be achieved by first creating a polymorphic type validator object specifying the names of the packages containing the allowed polymorphic subclasses, and the classes of the collection containing the attribute elements. Next, the object mapper must be told to use the validator previously created. This is achieved as illustrated by the code snippet below:

```
public TestRobotSimSerializationJSON() {
    objectMapper = new ObjectMapper();
    PolymorphicTypeValidator typeValidator = BasicPolymorphicTypeValidator.builder()
        .allowIfSubType(PositionedShape.class.getPackageName())
        .allowIfSubType(Component.class.getPackageName())
        .allowIfSubType(BasicVertex.class.getPackageName())
        .allowIfSubType(ArrayList.class.getName())
        .allowIfSubType(LinkedHashSet.class.getName())
        .build();
    objectMapper.activateDefaultTyping(typeValidator,
                                       ObjectMapper.DefaultTyping.NON_FINAL);
```

Then, in your test class, create a method that will first create a factory model object. To create do so, you can simply copy the code of the beginning of the *main()* method of the *SimulatorApplication* class into your test class.

Next, create a test method that will convert the factory model into JSON text, and then recreate the Factory object from this JSON text. This is illustrated by the code below:

```
@Test
public void testSerialization()
throws JsonProcessingException {
    final String factoryAsJsonString = objectMapper.writeValueAsString(myFactory);
    LOGGER.info(factoryAsJsonString);

    final Factory roundTrip = objectMapper.readValue(factoryAsJsonString,
                                                     Factory.class);
    LOGGER.info(roundTrip.toString());
    ...
}
```

Finally, run your test class with a JUnit launch configuration and check that the factory model is serialized correctly.

Implementing a Remote Simulator Controller

To use the simulation microservice that you have developed in the previous section, another controller will need to be implemented to replace the current controller of the robotism project. This remote controller will implement the start and stop simulation methods

differently to call the methods of the microservice you just developed instead of calling those methods directly on the *Factory* object.

Calling a Microservice from a Plain Java Application

To call a microservice from a plain Java application like the robotic factory simulator, you will need to use the HTTP classes provided in the *java.net.http* package of the JDK. The first step is to create a *HTTPClient* object that will be used to connect to the server of the microservice and to send REST requests to it. This can be done with the following code:

```
HttpClient httpClient = HttpClient.newHttpClient();
```

Next, create a URI for the service REST method as illustrated in the following code:

```
final URI uri = new URI("http", null, "localhost”, 8080, "/simulation/test.factory",
null, null);
```

This will automatically encode the file name of the factory model to replace the characters that are not allowed in URLs by acceptable characters.

Next, create a HTTP request. As an example, the code below allows to create a GET REST request.

```
HttpRequest.newBuilder().uri(uri).GET().build();
```

Finally, send the request to the microservice as shown below:

```
HttpResponse<String> response = httpClient.send(request,
    HttpResponse.BodyHandlers.ofString());
```

To parse the received JSON string, instantiate an object mapper as described previously and use it as illustrated below:

```
ObjectMapper objectMapper = new ObjectMapper();
Factory factory = objectMapper.readValue(response.body(), Factory.class);
```

If needed, see https://www.youtube.com/watch?v=MAw5Ku1OVFA for a detailed explanation on how to call a REST API from a plain Java application (JDK client).

Creating a Remote Simulator Controller Class

Finally, create a new *RemoteSimulatorController* class for the simulator controller which will call the microservice to control the simulation. The class can extend the previous *SimulatorController* class and redefine the *startAnimation()* and *stopAnimation()* methods so that they use the Java HTTP classes previously introduced to call the REST methods of the simulation microservice.

As explained before, the simulator implements a MVC design pattern so that every time the model data is changed, the viewer is called to refresh its data display. Now that the model is on a remote server, such mechanism can no longer work. Therefore, you will need to implement a method in the controller class that will periodically query the simulation microservice to retrieve from the factory ID a fresh factory model being simulated. An example of such code is given below:

```
private void updateViewer()
throws InterruptedException, URISyntaxException, IOException {
    while (getCanvas().isSimulationStarted()) {
```

```
        final Factory remoteFactoryModel = getFactory();
        setCanvas(remoteFactoryModel);

        Thread.sleep(100);
    }
}
```

The *setCanvas()* method can then be redefined to notify the canvas viewer that its model has changed so that it refreshes its display. This is illustrated in the example code below:

```
/**
 * {@inheritDoc}
 */
@Override
public void setCanvas(final Canvas canvasModel) {
    final List<Observer> observers = getCanvas().getObservers();

    super.setCanvas(canvasModel);

    for (final Observer observer : observers) {
        getCanvas().addObserver(observer);
    }

    getCanvas().notifyObservers();
}
```

Note that you will have to change the visibility of the *getObservers()* and *notifyObservers()* methods of the Factory class for this code to compile.

Testing the Simulator

Two major changes have been applied to the simulator code. The first one consisted in modifying to the factory model classes so that the model can be serialized into JSON text with Jackson. The second one consisted of creating a microservice to simulate the model and modifying the simulator controller so that it calls this service instead of using the model directly.

Testing the Monolithic Simulator with the Modified Factory Model Classes

To make testing easy, first ensure that the monolithic simulator still runs correctly using the modified model classes. For this, make sure that the original local persistence manager and simulator controller classes are used in the *SimulatorApplication* class. Launch the simulator and ensure that it still works correctly as before modifying the model classes for JSON.

Testing the Monolithic Simulator with the Roundtrip (Parsed / Unparsed) Factory Model

Create a method in your JSON serialization test class so that it returns a factory model that will have first been serialized into JSON text and then deserialized into a factory model from the JSON text. Launch the simulator with this model and ensure it still works correctly.

Testing the Distributed Simulator

Next, make sure that remote persistence manager and simulator controller classes that you created are used in the *SimulatorApplication* class.

Next, you need to customize the Jackson object mapper that is used by Spring in the microservice project so that it is configured the same way as for the serialization tests using the polymorphic type validator. To do so, create a Spring configuration bean and provide the mapper as shown in the code below:

```
@Configuration
public class SimulationRegisterModuleConfig {

    @Bean
    @Primary
    public ObjectMapper objectMapper() {
        final PolymorphicTypeValidator typeValidator =
            BasicPolymorphicTypeValidator.builder()
                .allowIfSubType(PositionedShape.class.getPackageName())
                .allowIfSubType(Component.class.getPackageName())
                .allowIfSubType(BasicVertex.class.getPackageName())
                .allowIfSubType(ArrayList.class.getName())
                .allowIfSubType(LinkedHashSet.class.getName())
                .build();

        final ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.activateDefaultTyping(typeValidator,
                                           ObjectMapper.DefaultTyping.NON_FINAL);

        return objectMapper;
    }
}
```

Finally, launch the simulator and ensure that the distributed simulator still works as well as the monolithic one.