| *Exercise 1: Mocking and Stubbing* |
| --- |

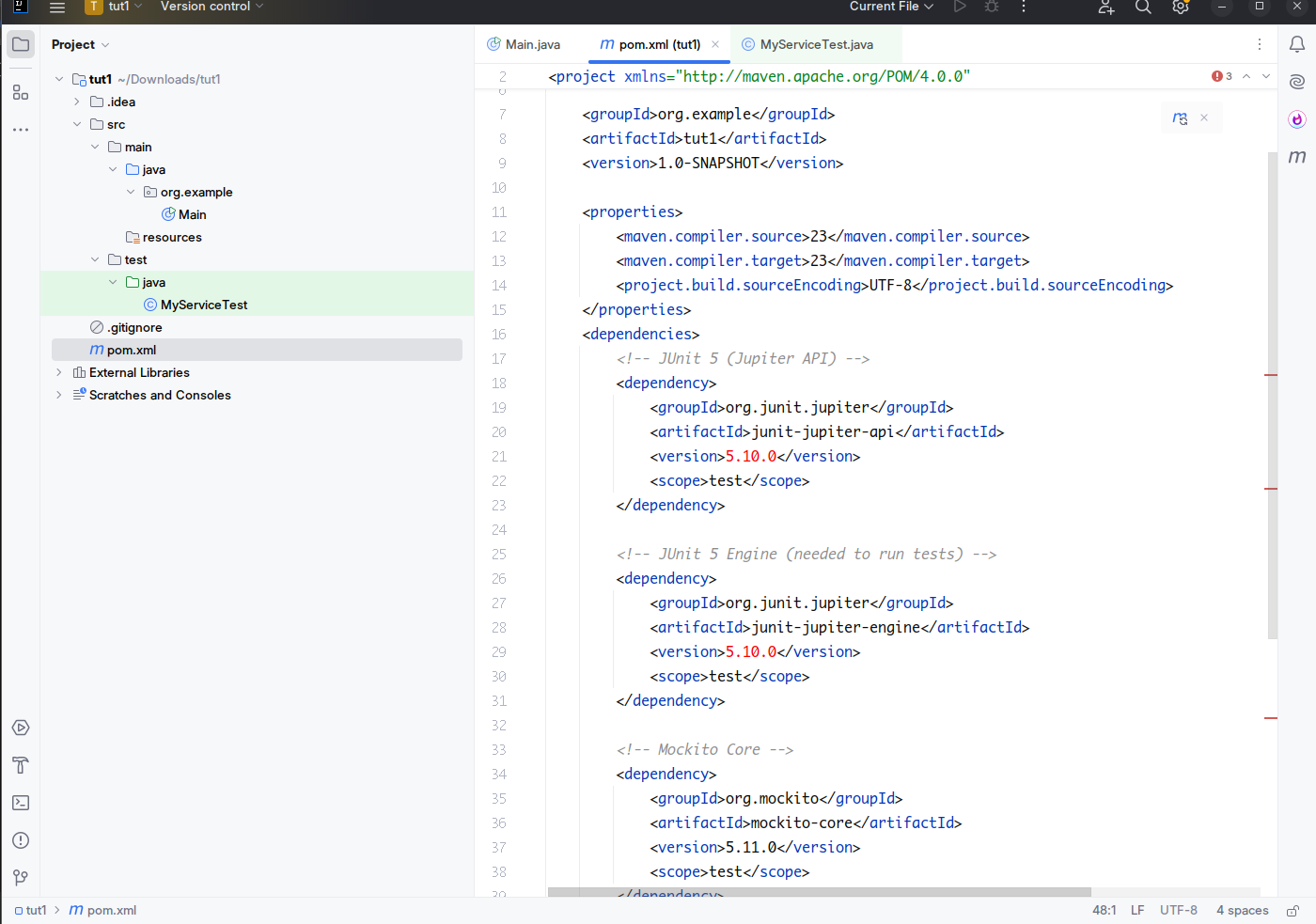
## 

### **1. Understand the scenario**

We are writing unit tests for a service class that depends on an external API. Calling the real external API during tests is not ideal because it can be slow, unstable, or produce inconsistent results. Instead, we are going to use **Mockito**, a popular Java mocking framework, to create a **mock** version of the external API. This allows us to simulate the behavior of the real API without actually calling it, ensuring our tests remain fast, stable, and focused only on our own logic.

### **2. Add Mockito to the project**

To use Mockito, we need to add its dependency to our project. If we are using Maven, we add the following block to our pom.xml:



If we are using Gradle, we include:

testImplementation 'org.mockito:mockito-core:5.11.0'

This ensures Mockito is available during test execution.

### **3. Create a mock object**

In the test class, we create a mock object for the external dependency using Mockito.mock(). Suppose the dependency is an interface named ExternalApi. The code to create its mock is:

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

Now mockApi behaves like ExternalApi, but without any real implementation. We are free to define how its methods behave.

### **4. Stub method behavior**

After creating the mock, we specify what it should return when certain methods are called. This is called **stubbing**. We use when(...).thenReturn(...) for this.

when(mockApi.getData()).thenReturn("Mock Data");

This means that when getData() is called on the mock object, it will return the string "Mock Data". We are controlling the behavior of the mock to test how our service reacts.

### **5. Inject the mock into the service**

We assume we have a class called MyService that depends on ExternalApi. Instead of passing a real implementation of the API, we inject the mock. This keeps the test independent from real-world data or network calls.

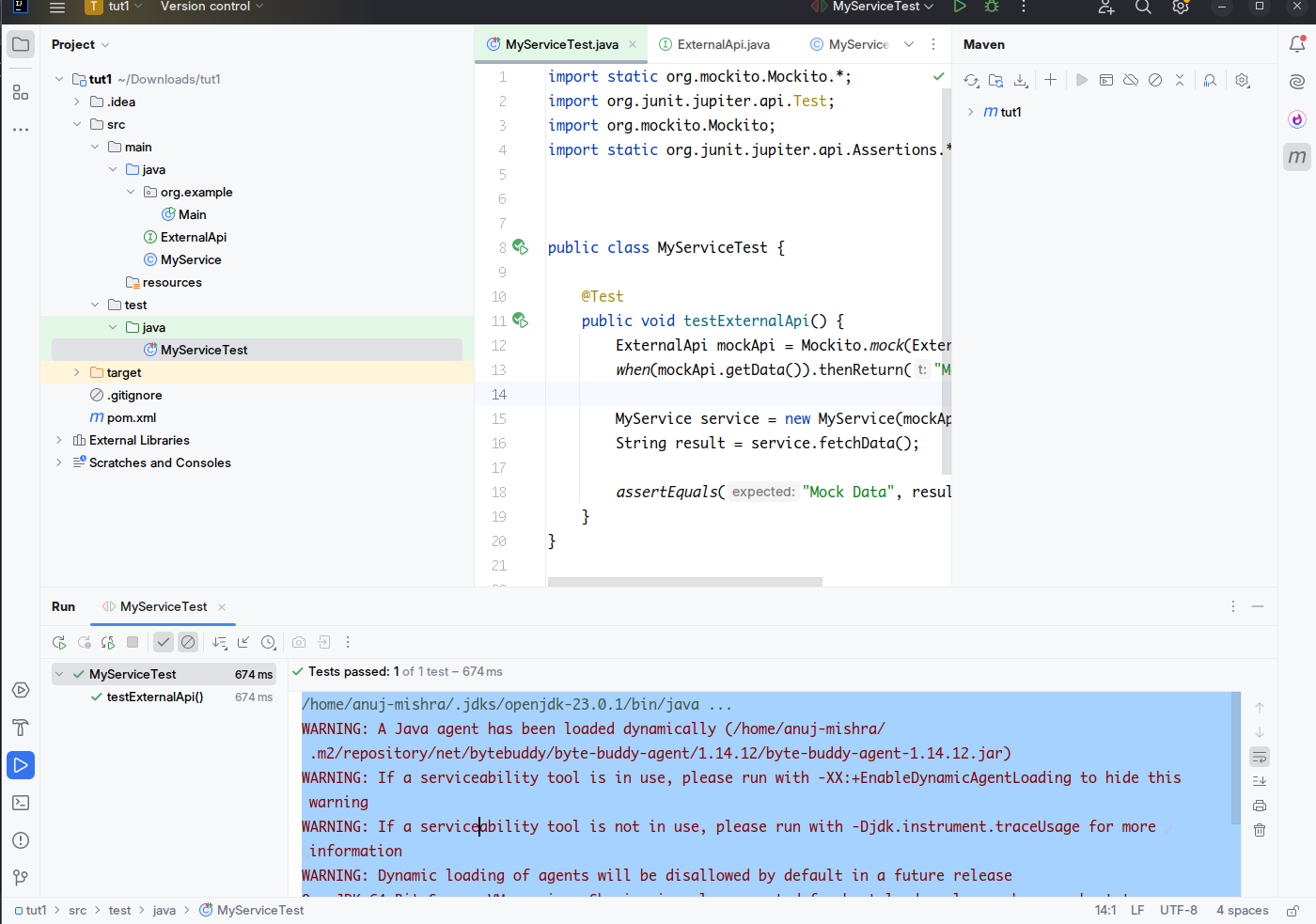
MyService service = new MyService(mockApi);

We are now testing MyService in isolation, using a predictable and controlled version of its dependency.

### 

### **6. Call the method and assert the result**

With the setup complete, we call the method from our service that internally uses the mock. We then use assertions to check whether the return value is what we expect based on our stubbed behavior.



We are verifying that our service uses the external API correctly and handles its response as intended.

### 

### **7. Complete solution code**

