howest
hogeschool

Object Oriented Architectures and Secure Development

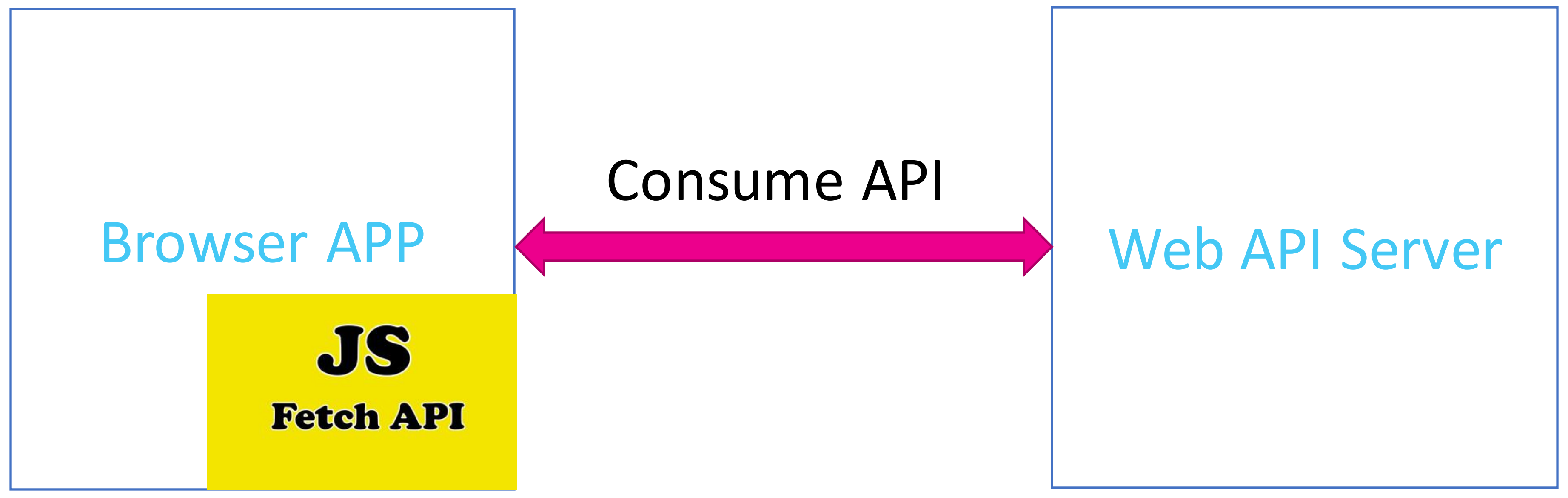
VERT.X - Futures & Promises

Class taught topic for the Analysis and development project

Matthias Blomme
Mattias De Wael
Frédéric Vlummens

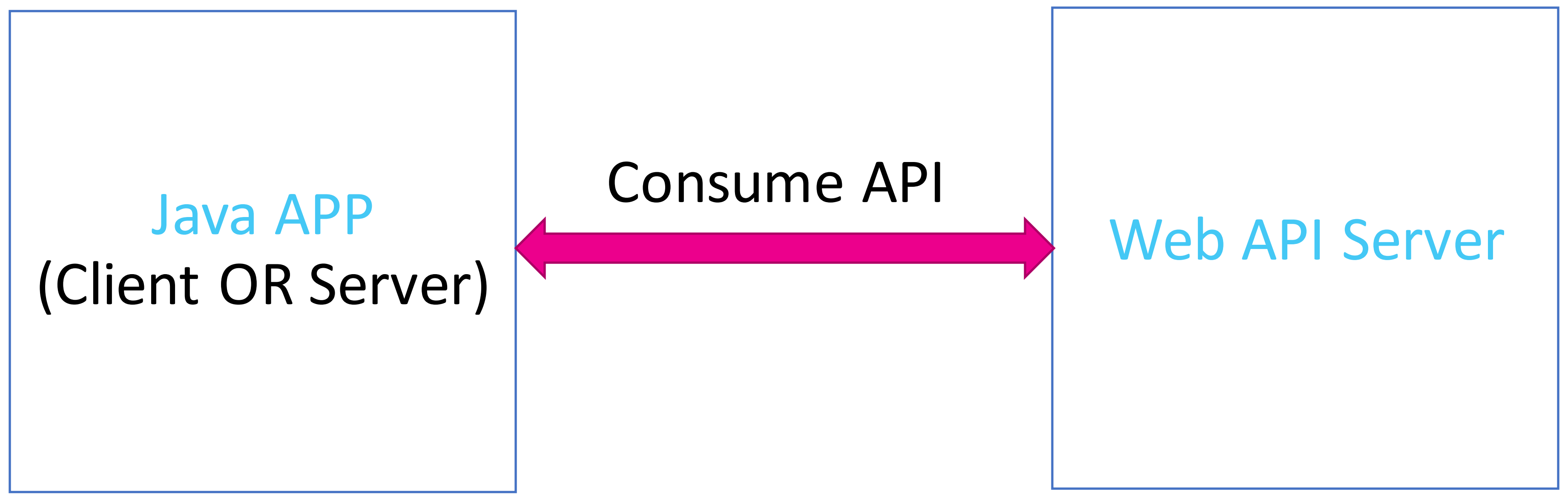
What do you already know?

Consuming Web APIs from a browser application

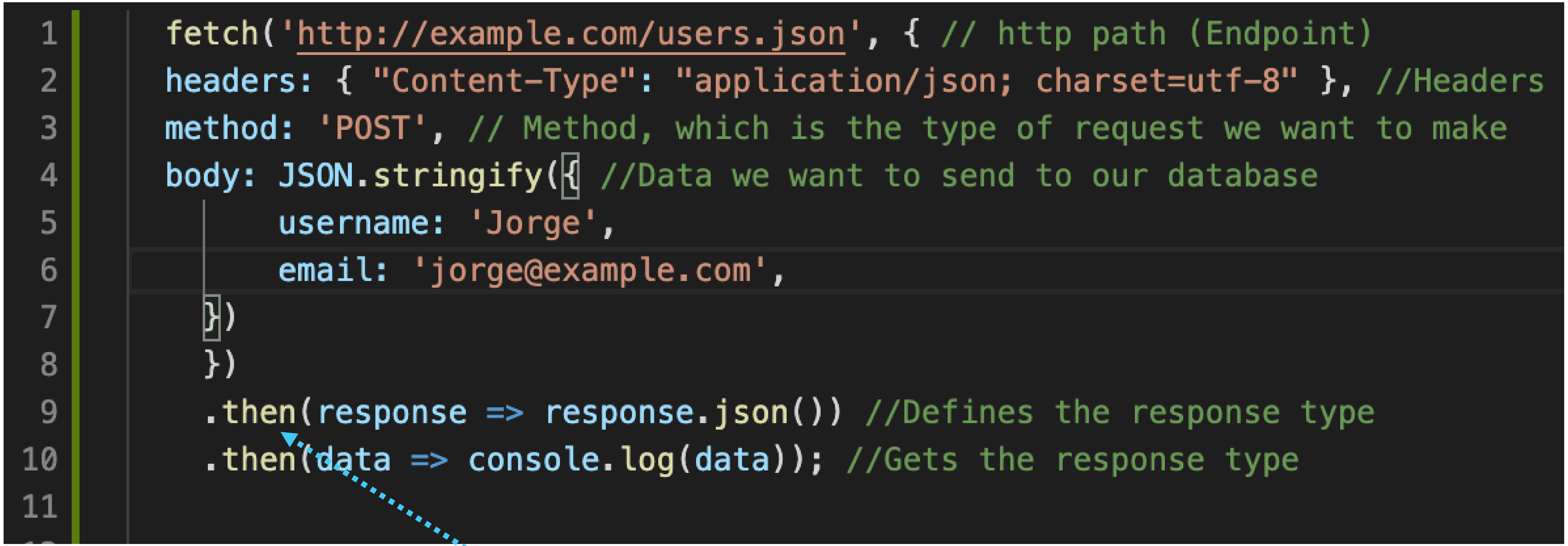


Goal

Consuming Web APIs from a Java application



JS Fetch API - Recap



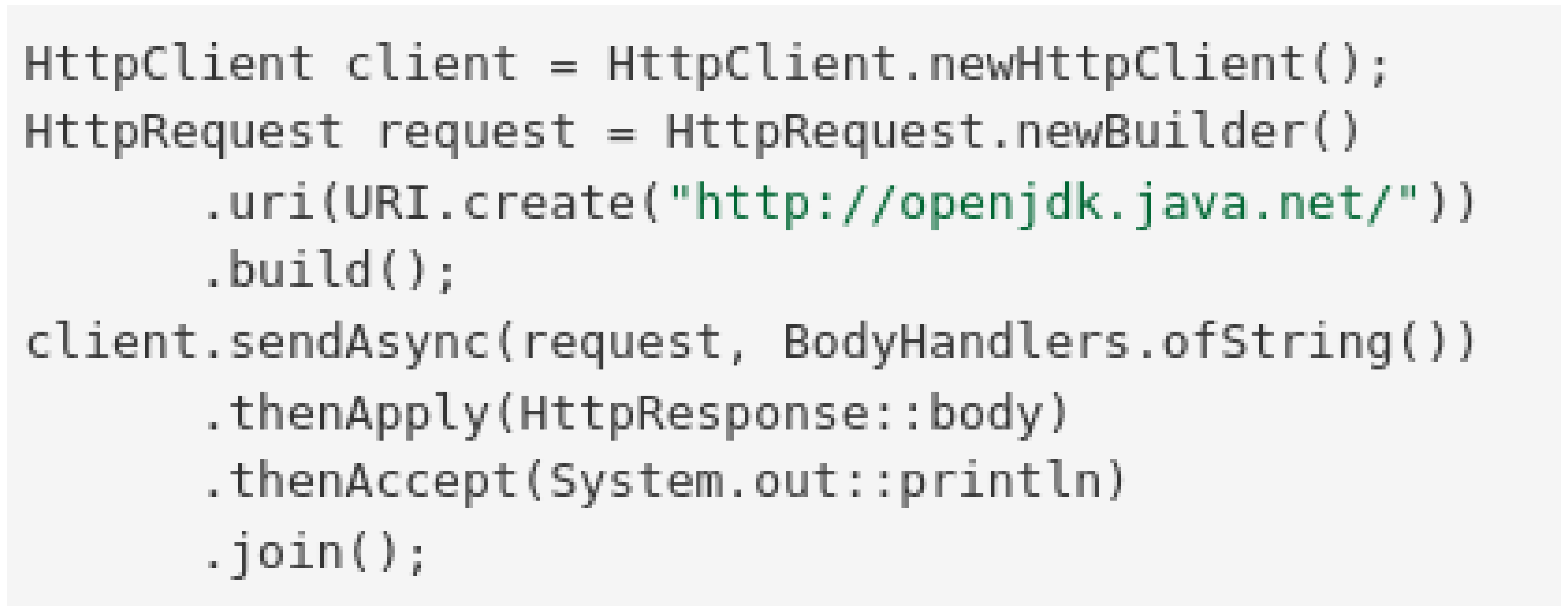
```
fetch('http://example.com/users.json', { // http path (Endpoint)
headers: { "Content-Type": "application/json; charset=utf-8" }, //Headers
method: 'POST', // Method, which is the type of request we want to make
body: JSON.stringify({ //Data we want to send to our database
        username: 'Jorge',
        email: 'jorge@example.com',
    })
    })
    .then(response => response.json()) //Defines the response type
    .then(data => console.log(data)); //Gets the response type
```

Uses **PROMISES**.
The arrow expressions are executed **ASYNC**!

Consuming Web APIs in Java

Consuming Web APIs in an asynchronous manner is built into Java

- Classes:
  - Completable Future
  - HttpClient (> Java 11)
  - HttpRequest



```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create("http://openjdk.java.net/"))
      .build();
client.sendAsync(request, BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();
```

- **We will not use these built-in classes!**
  Instead we will make use of the **Vert.X toolkit.**

What is Vert.X ?

- Vert.X is NOT a framework but a **toolkit**.
- Designed for **asynchronous** communications.
- Can be written in many languages.
- Was used in the **programming project**: creating an Open API.
- Needed in the **Analysis and Development Project**.

Vert.X - Asynchronous communications

Vert.X uses the concept of **Futures & Promises**
to create all kinds of **asynchronous** applications.

Vert.X - Futures & Promises

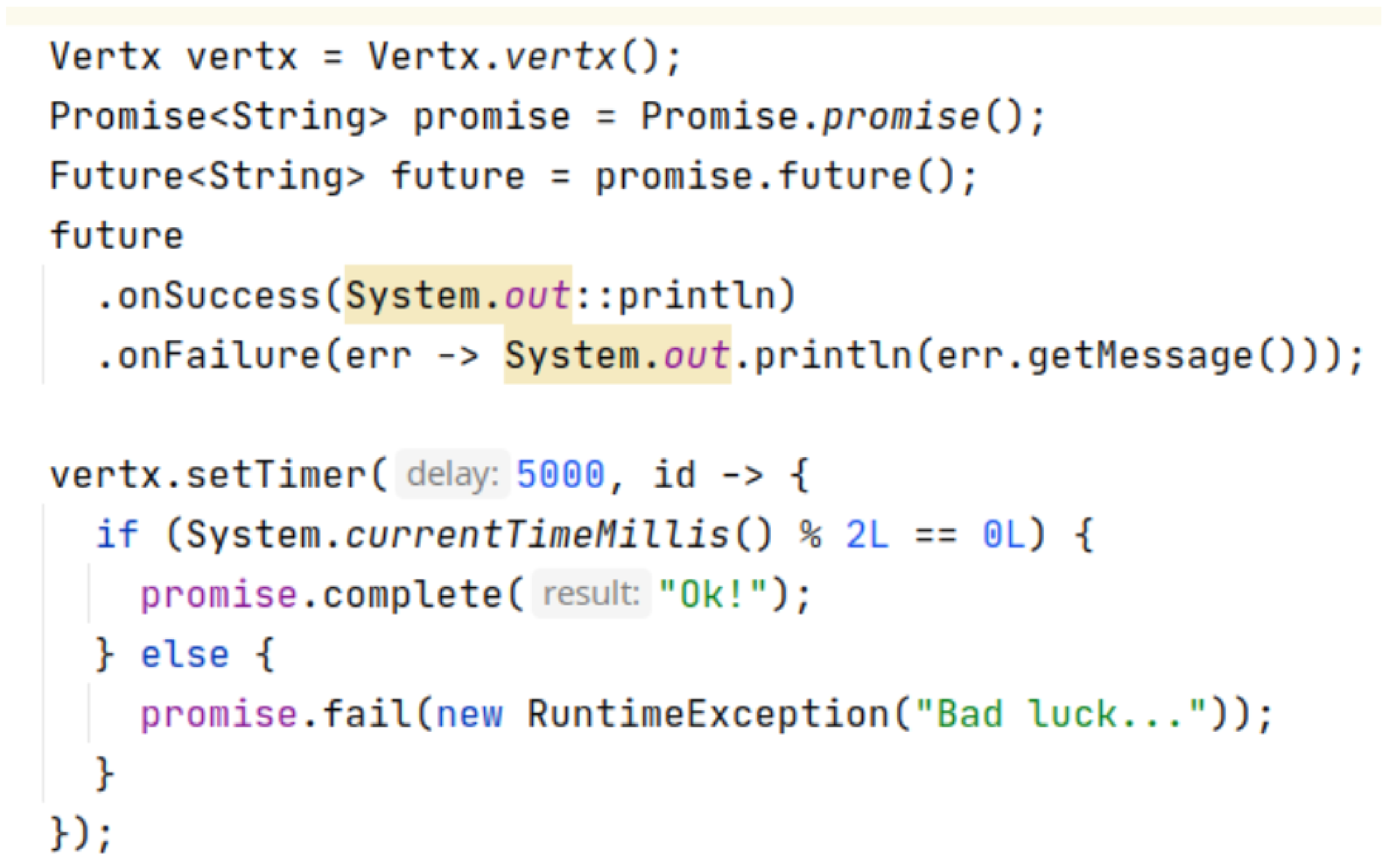
- A promise holds the value of some computation for which there is **no value right now.**
- A promise is **eventually** completed with a **result** value or an **error**.
- When the **promise is completed**, the **future** object is **notified**.
- In turn, a future allows you to read a value that will **eventually** be available

A **promise** is used to **write** an eventual value,

and a **future** is used to **read** it when it is available.

Simple example

1. Create a promise that will hold an **eventual** value. String in this case.
2. A promise gives back a future, where in the future the result will be available.
3. With the future we define **what** we want to do when the value is available.
   1. A promise can be successful or failed.
   2. Notice that we defined **what** to do with the result before even generating a result with a promise.
   3. A future is a Java object that can be passed around just like any other object.
4. vertx timer is created that will trigger after 5 seconds.
   1. If the current time in ms is even we complete the promise with the result "Ok!"
   2. Otherwise we say the promise fails with an exception.
   3. When **complete** or **fail** is called. The original future gets **automatically notified.**



```java
Vertx vertx = Vertx.vertx();
Promise<String> promise = Promise.promise();
Future<String> future = promise.future();
future
  .onSuccess(System.out::println)
  .onFailure(err -> System.out.println(err.getMessage()));

vertx.setTimer( delay: 5000, id -> {
  if (System.currentTimeMillis() % 2L == 0L) {
    promise.complete( result: "Ok!");
  } else {
    promise.fail(new RuntimeException("Bad luck..."));
  }
});
```

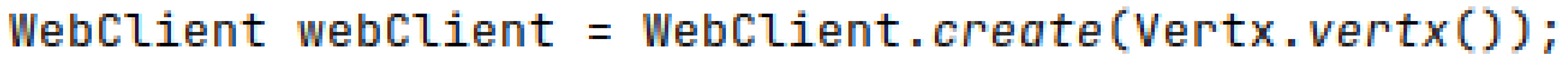
Vert.X - WebClient

Vert.X **WebClient** can be used to consume a **Web API** in an **asynchronous** manner.

The accomplish this Vert.X uses **Futures & Promises**

Demo

Vert.X - WebClient

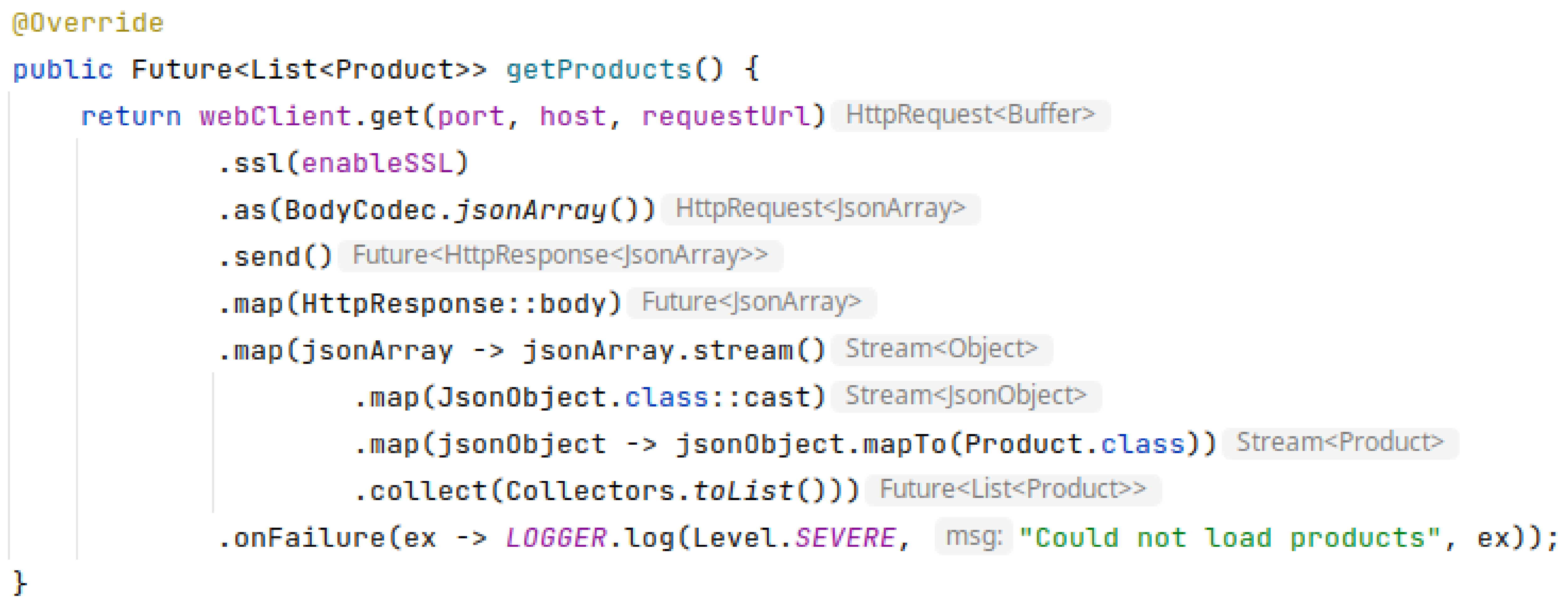


```
WebClient webClient = WebClient.create(Vertx.vertx());
```

Only **one** WebClient per application is needed!

Consume an API with the WebClient.

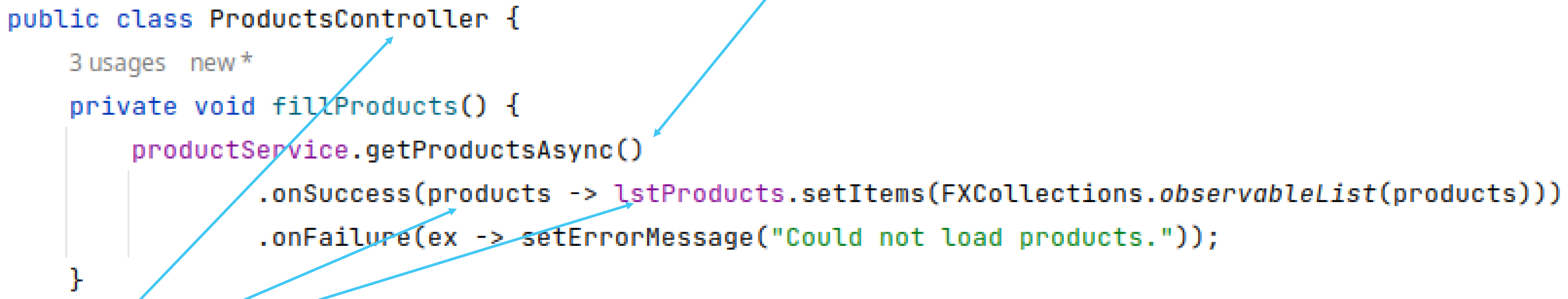
- **AS**: what is the format of the response body.
- **Send**: send the HTTP request.
- **HttpReponse::Body**: get the body from the response
- **Map** the result to the wanted Domain Logic (List<Product>)
- **OnFailure:** if something goes wrong, log the result



```
@Override
public Future<List<Product>> getProducts() {
    return webClient.get(port, host, requestUrl) HttpRequest<Buffer>
            .ssl(enableSSL)
            .as(BodyCodec.jsonArray()) HttpRequest<JsonArray>
            .send() Future<HttpResponse<JsonArray>>
            .map(HttpResponse::body) Future<JsonArray>
            .map(jsonArray -> jsonArray.stream() Stream<Object>
                    .map(JsonObject.class::cast) Stream<JsonObject>
                    .map(jsonObject -> jsonObject.mapTo(Product.class)) Stream<Product>
                    .collect(Collectors.toList())) Future<List<Product>>
            .onFailure(ex -> LOGGER.log(Level.SEVERE, msg: "Could not load products", ex));
}
```

Vert.X - handle the Future object

GetProductsAsync returns a **Future<List<Product>>**

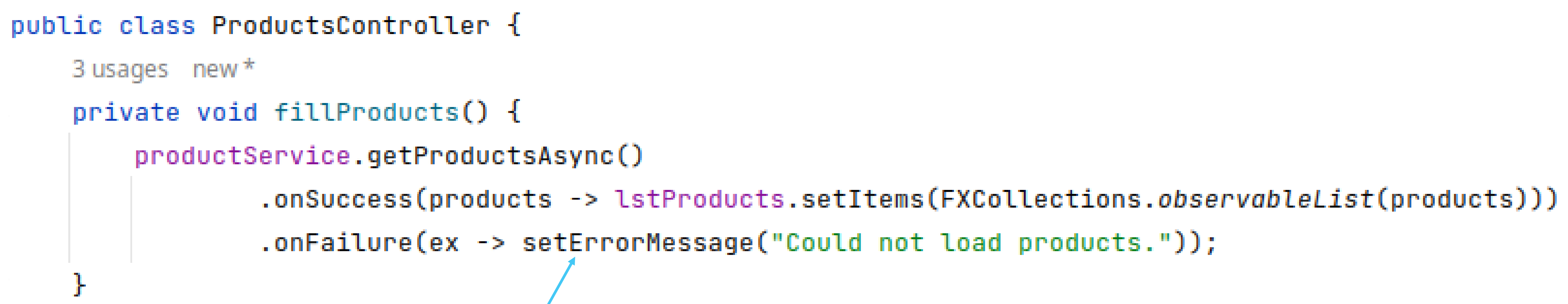


```
public class ProductsController {
    3 usages   new *
    private void fillProducts() {
        productService.getProductsAsync()
                .onSuccess(products -> lstProducts.setItems(FXCollections.observableList(products)))
                .onFailure(ex -> setErrorMessage("Could not load products."));
    }
```

Handle the **Future** object where it is needed/wanted!
In this case, it is the **ProductsController** which needs the **results** (List<Product>) from the **Future** object.

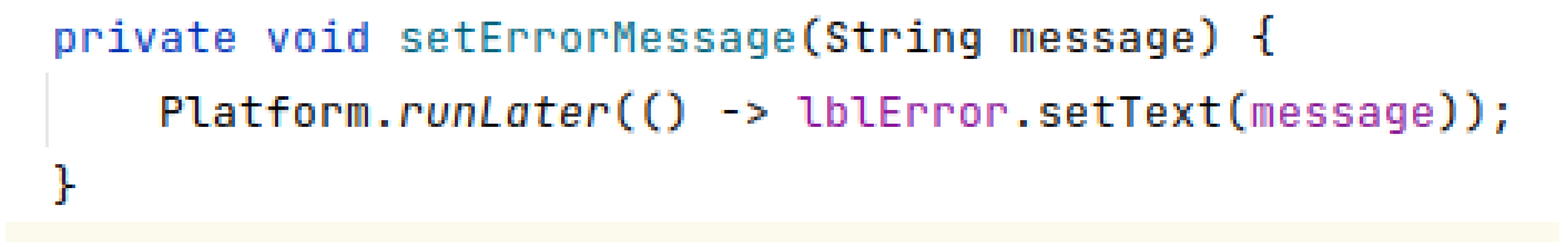
Remember, the function passed into the **onSuccess** method is only executed when an **actual result** is available (this is an unpredictable time after the original WebClient.send call).

Vert.X - JavaFX integration



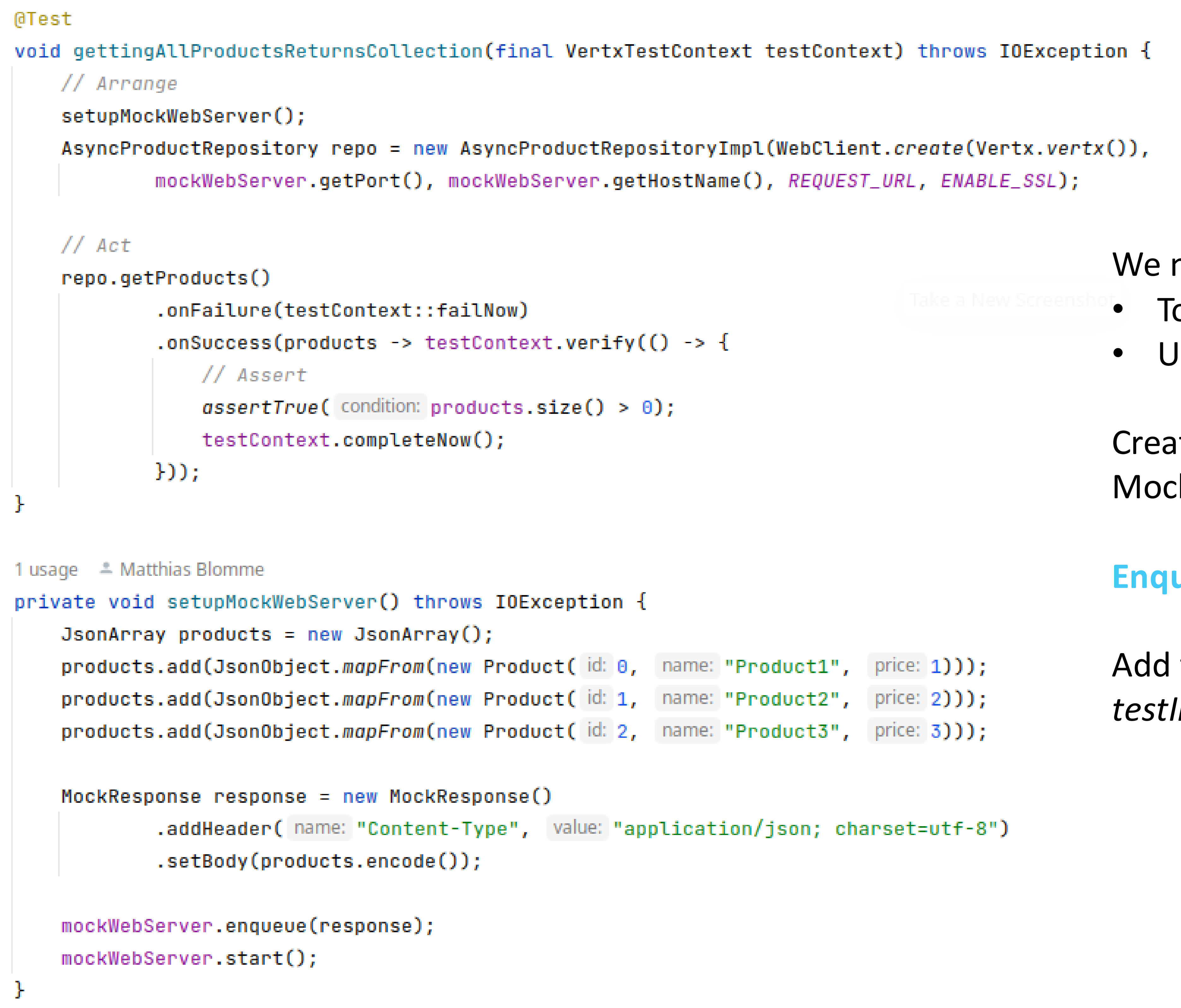
```java
public class ProductsController {
    3 usages   new *
    private void fillProducts() {
        productService.getProductsAsync()
                .onSuccess(products -> lstProducts.setItems(FXCollections.observableList(products)))
                .onFailure(ex -> setErrorMessage("Could not load products."));
    }
```

Due to some threading issues with JavaFX/Vert.X, the following fix is needed to set an actual Label



```java
private void setErrorMessage(String message) {
    Platform.runLater(() -> lblError.setText(message));
}
```

Unit test the Future/Promises + mock a web server



```
@Test
void gettingAllProductsReturnsCollection(final VertxTestContext testContext) throws IOException {
    // Arrange
    setupMockWebServer();
    AsyncProductRepository repo = new AsyncProductRepositoryImpl(WebClient.create(Vertx.vertx()),
            mockWebServer.getPort(), mockWebServer.getHostName(), REQUEST_URL, ENABLE_SSL);

    // Act
    repo.getProducts()
            .onFailure(testContext::failNow)
            .onSuccess(products -> testContext.verify(() -> {
                // Assert
                assertTrue( condition: products.size() > 0);
                testContext.completeNow();
            }));
}

1 usage   Matthias Blomme
private void setupMockWebServer() throws IOException {
    JsonArray products = new JsonArray();
    products.add(JsonObject.mapFrom(new Product( id: 0,  name: "Product1",  price: 1)));
    products.add(JsonObject.mapFrom(new Product( id: 1,  name: "Product2",  price: 2)));
    products.add(JsonObject.mapFrom(new Product( id: 2,  name: "Product3",  price: 3)));

    MockResponse response = new MockResponse()
            .addHeader( name: "Content-Type",  value: "application/json; charset=utf-8")
            .setBody(products.encode());

    mockWebServer.enqueue(response);
    mockWebServer.start();
}
```

We need to **mock** the external API:

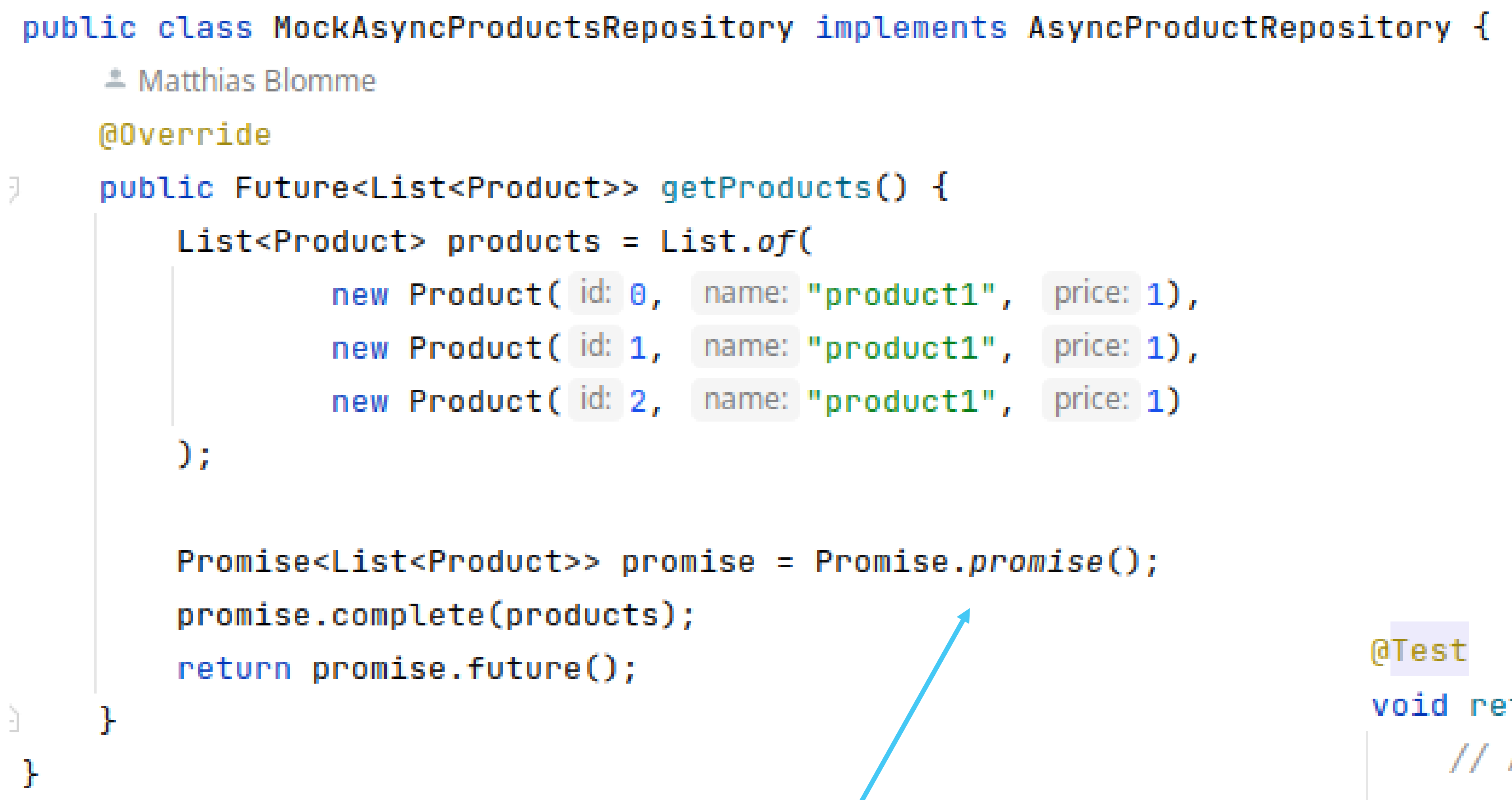
- Too slow
- Unreliable

Create a **MockWebServer**
MockWebServer mockWebServer = new MockWebServer();

Enqueue (mock) the **expected** HTTP Request and it's response.

Add the following **dependency** to start using **MockWebServer**:
testImplementation 'com.squareup.okhttp3:mockwebserver:4.10.0'

Mock the AsyncRepository in the Service layer.

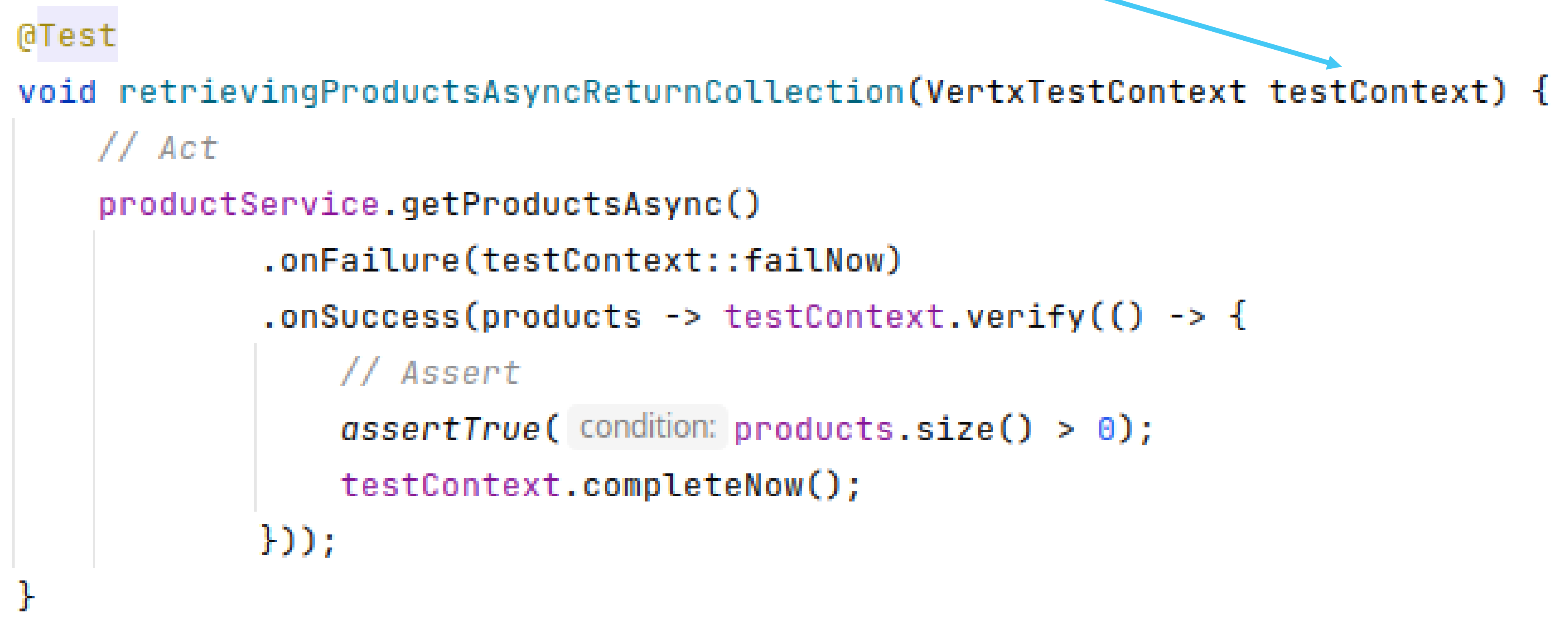


```
public class MockAsyncProductsRepository implements AsyncProductRepository {
    Matthias Blomme
    @Override
    public Future<List<Product>> getProducts() {
        List<Product> products = List.of(
                new Product( id: 0,  name: "product1",  price: 1),
                new Product( id: 1,  name: "product1",  price: 1),
                new Product( id: 2,  name: "product1",  price: 1)
        );

        Promise<List<Product>> promise = Promise.promise();
        promise.complete(products);
        return promise.future();
    }
}
```

Vert.X helper object for **Junit** 5.
Assert the result in the t**estContext.verify** method.

TestContext.completeNow() -> let the test **pass**.
TestContect.failNow(Ex) -> let the test **fail**.



```
@Test
void retrievingProductsAsyncReturnCollection(VertxTestContext testContext) {
    // Act
    productService.getProductsAsync()
            .onFailure(testContext::failNow)
            .onSuccess(products -> testContext.verify(() -> {
                // Assert
                assertTrue( condition: products.size() > 0);
                testContext.completeNow();
            }));
}
```

Create a **Promise**.
Call the **complete** method to **pass** the results and **notify** the Future object