SPRING BOOT: TESTING & WEBFLUX

DAY 3

COURSE OVERVIEW

TODAY'S TOPICS

- Testing
  - @WebMvcTest
  - jUnit
  - Mocking services and repositories with Mockito
  - @DataJpaTest
- Webflux
  - Reactive programming
  - RESTful service
  - Kafka and webflux

WE'RE GOING TO EXPLORE…

- Unit tests: running in isolation
  - MockBean
  - WebMvcTest
- Integration test: bootstrap Spring context
  - SpringBootTest
  - TestConfiguration
  - DataJpaTest

DEMO UNIT TESTING

EXERCISE

- Create a Spring Boot app
- Add Junit 5
- Add a simple method to do some basic calculations: public double
- doCalcStuff(String operation, double d1, double d2)
- Write 4 tests for the method (simple cases)
- Also write a test to test what happens on division by 0

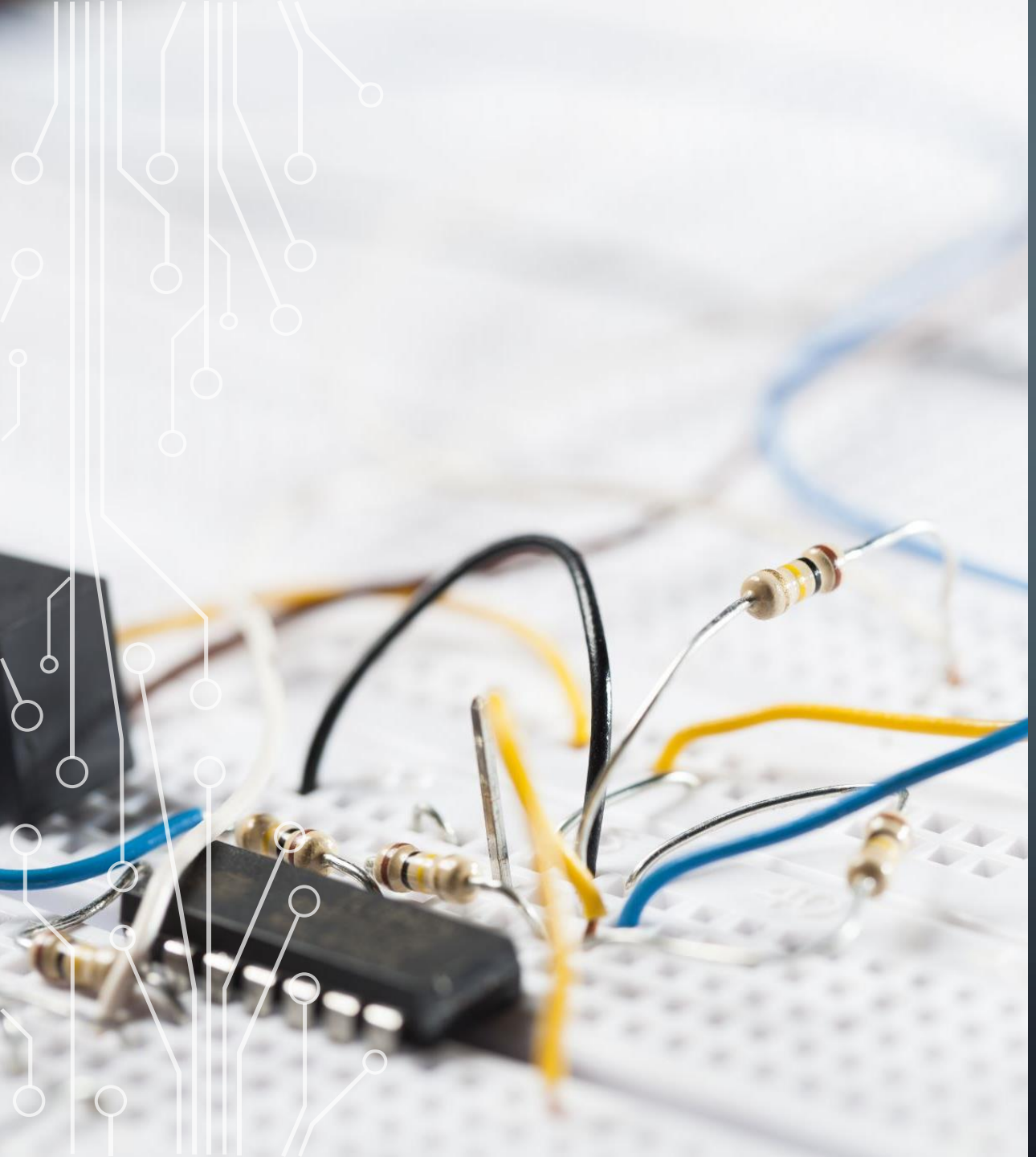
TESTING CONTROLLERS

- @WebMvcTest slice annotation to set up the application context needed to test the controller layer
- Autoconfigures Spring MVC and sets up the essentials: request mappings, controller advice, and more
- Test the controller as if it was called by a real user, but without running a full web server

TESTING AND MOCKING WITH MOCKMVC

- If you're controller relies on components, @WebMvcTest can automatically provide mock versions of these components
- Built-in MockMvc allows to simulate HTTP requests to controllers and asserts the responses
- @WebMvcTest can be combined with JUnit and Mockito
- @WithMockUser(username = "user", roles = "USER") can be used to simulate an authenticated user for tests that require security context

DEMO TESTING CONTROLLER

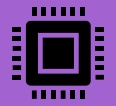


EXERCISE

- Create an Employee entity
- Create a Job entity (name and list of responsibilities)
- Give the Employee the following properties: name, employeeNr and a job (of type Job)
- Add controllers, services, repositories implementing CRUD functionality and connect the app to a MySQL db.
- Write the tests for the controllers.

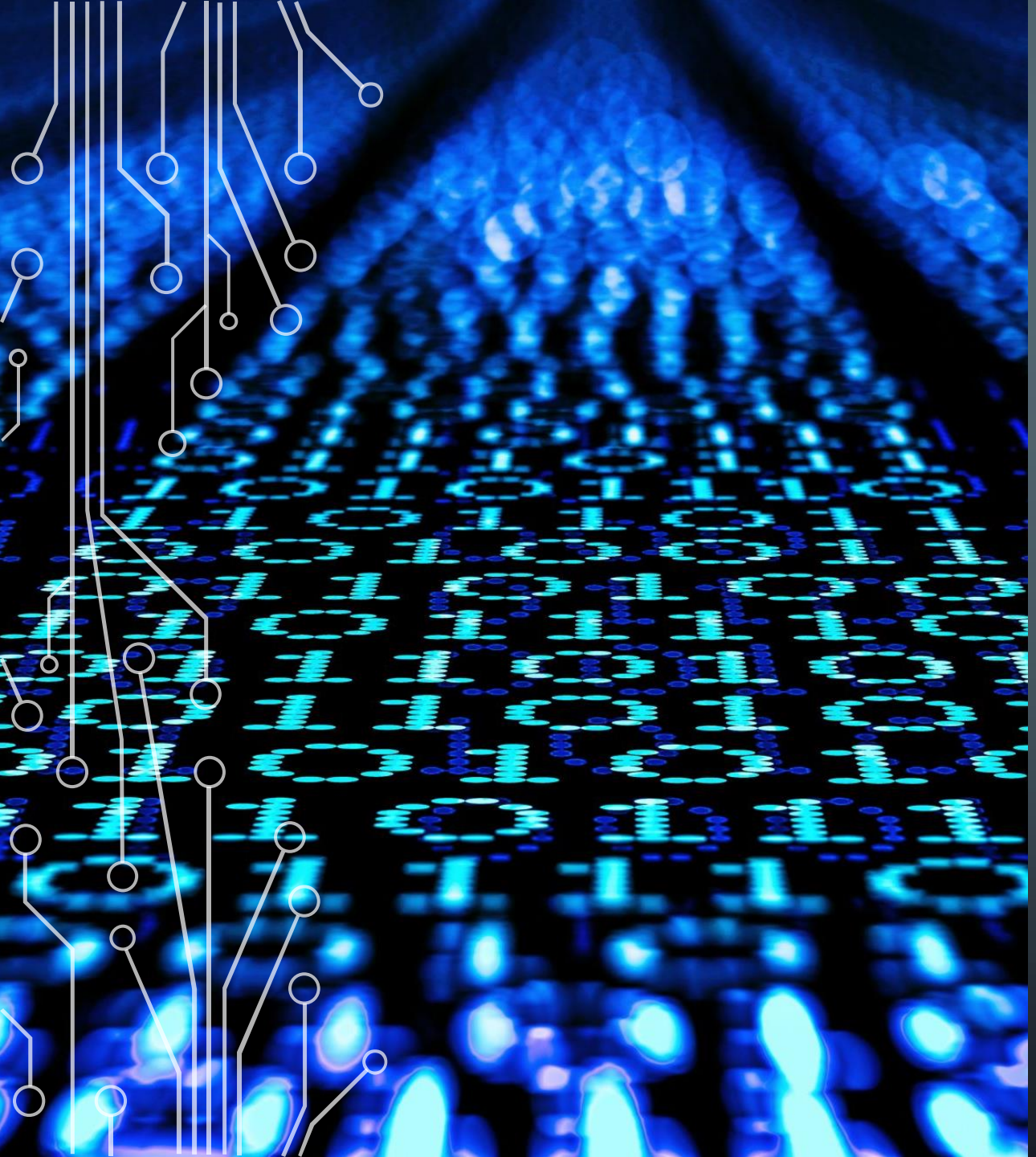
DEMO TESTING DATA LAYER

@DataJpaTest is used to test the repository layer and sets up embedded database for testing purposes (you can configure it another database if needed)



It doesn't load other parts of your application, like controllers or services, making the tests run faster and ensuring they're focused on the database interactions.

Tests annotated with @DataJpaTest are transactional, and each test is rolled back at the end. This means changes made to the database in a test won't affect other tests.



DEMO @DATAJPATEST

EXERCISE

- Continue from the last exercise and test the data layers

INTEGRATION TEST

- Covering multiple units
- Covering multiple layers
- Covering flow of a functionality
- This is what @SpringBootTest is for
- Configures the application context before running the tests, so more than a unit can be tested

TRADITIONAL SPRING MVC

- Built on a servlet API
- Uses a blocking I/O model: for each request, a thread is allocated from the server's thread pool, and this thread is held up until the response is ready and sent back to the client.
- This model is simple and works well for many applications, especially those with low to moderate traffic and straightforward CRUD operations.

BUT….

- In scenarios with high concurrency or long-duration requests, this model can become inefficient, as it might exhaust the thread pool or require a significantly larger amount of resources to handle peak loads effectively.

SPRING WEBFLUX

- Introduced with Spring 5 to build non-blocking, reactive applications
- It operates on a different model, designed to handle concurrency with a small number of threads and to optimize resource utilization (especially in microservices and cloud environments)
- Threads can be freed to handle other tasks, instead of waiting for a single request to complete

REACTIVE PROGRAMMING

DECLARATIVE PROGRAMMING PARADIGM

FOCUSSED ON DATA STREAMS AND CHANGE PROPAGATION

FEATURES OF REACTIVE PROGRAMMING

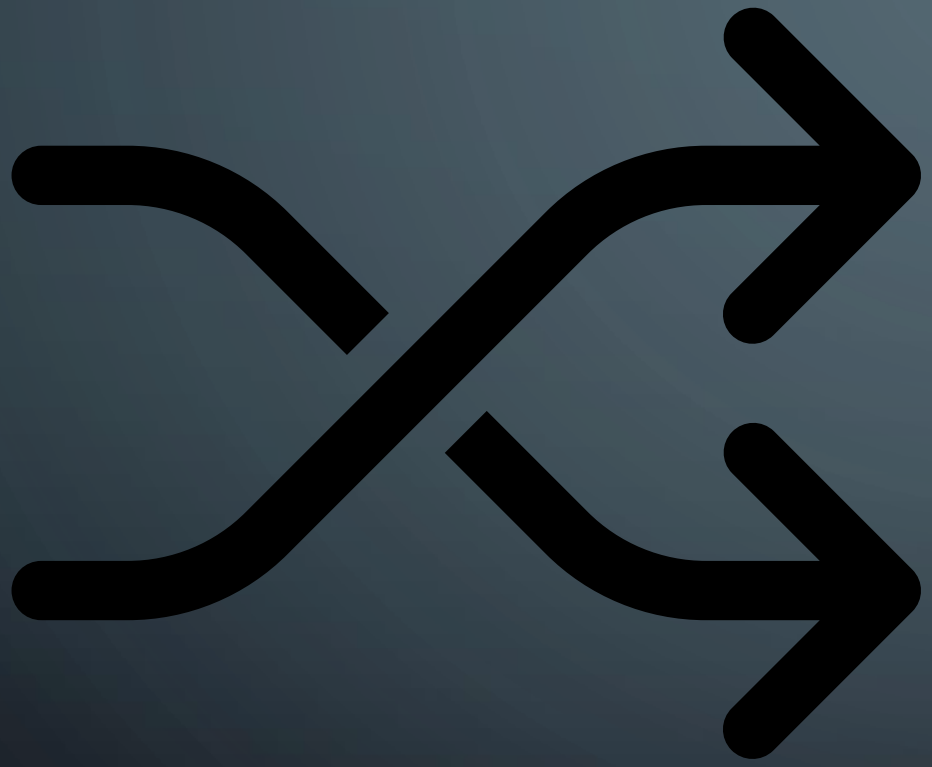
Non-blocking

Asynchronous

Funtional / declarative

NON-BLOCKING

- A non-blocking call responds immediately with the data currently available.
- The current process doesn't need to wait for the non-blocking call to return the result



ASYNCHRONOUS

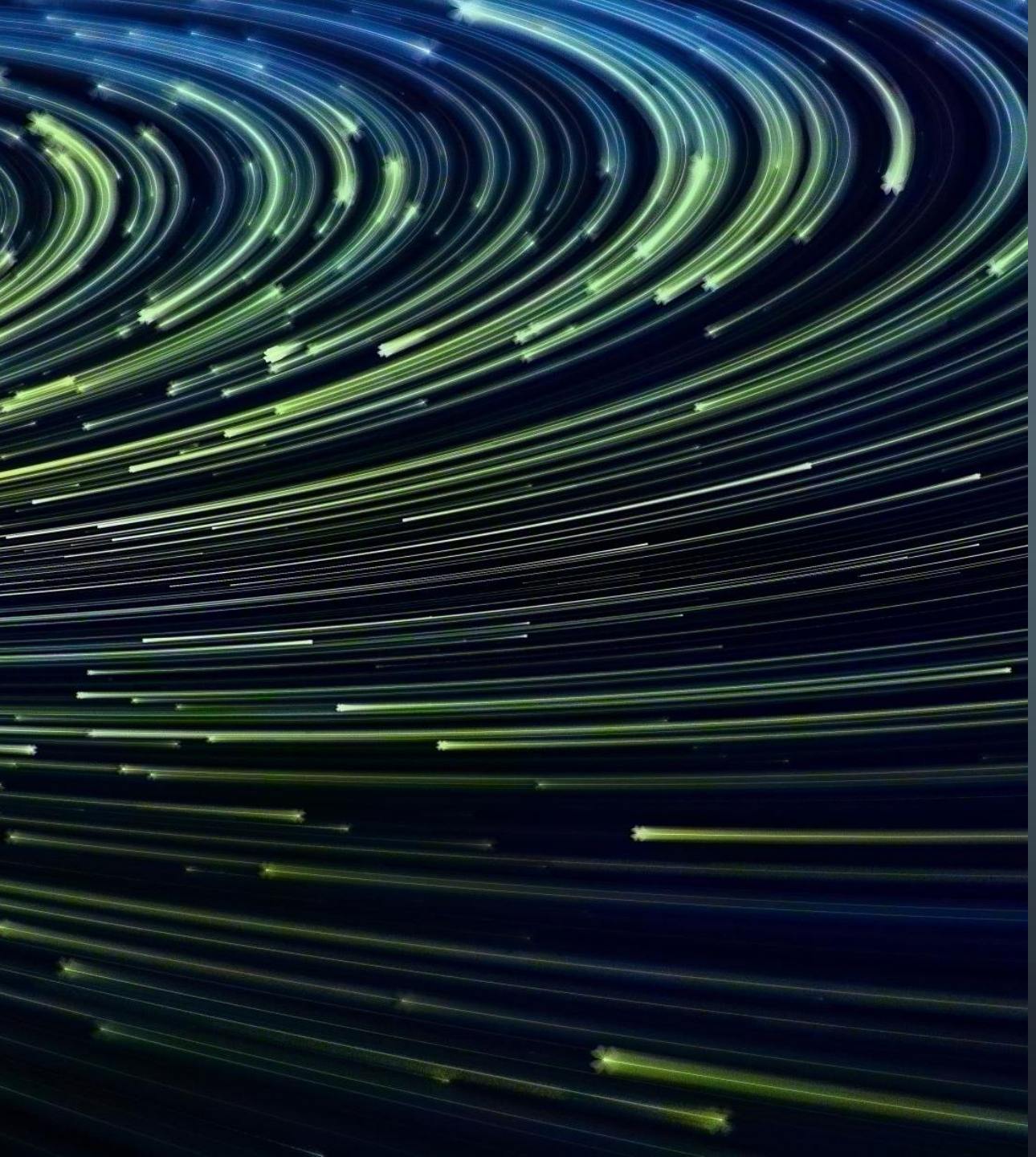
- Parallel programming
- No need to wait with the next task for the execution of the current task to be done
- Multiple things happening at the same time

SPRING REACTIVE FRAMEWORK (WEBFLUX)

- Spring framework using reactive principles
- Alternative, not a replacement, for Spring MVC stack
- Runs on top of Reactive Streams

BENEFITS & WHEN TO USE SPRING WEBFLUX

- Better utilization of resources because of non-blocking event-driven nature (less threads, less memory)
- Complexity of handling concurrent operations is abstracted away in the reactive programming
- When to use: real time data, microservices and distributed systems, highly scalable systems



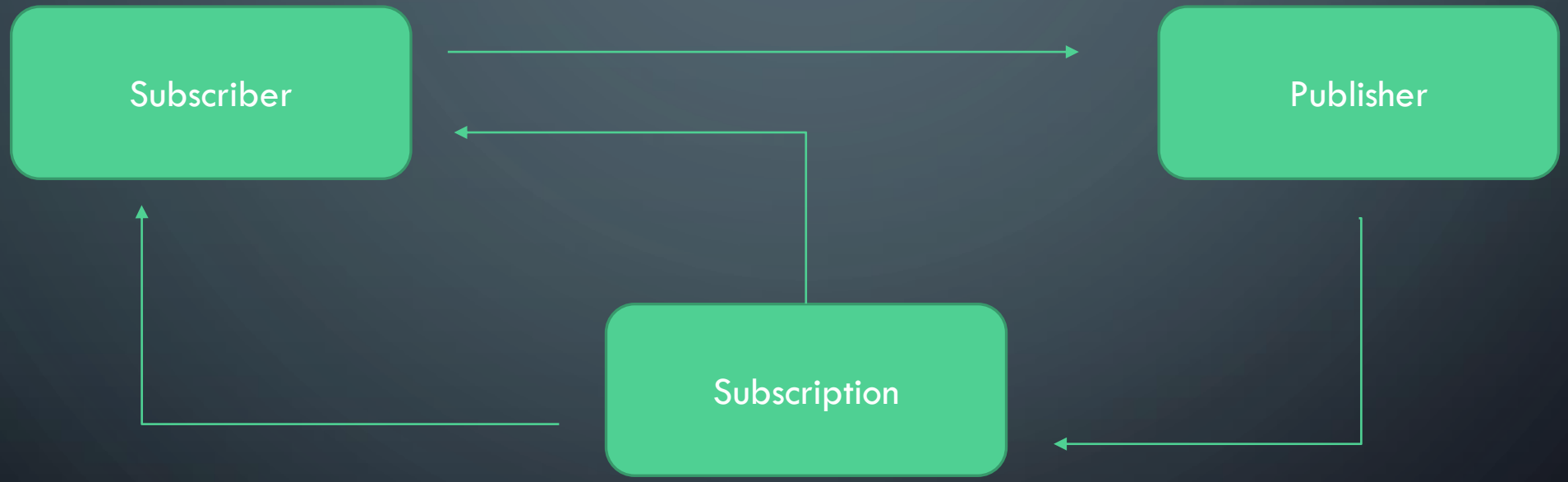
REACTIVE STREAMS

- Contract for:
  - Async
  - Non-blocking with backpressure
- Standard, not an implementation

INTERFACES OF REACTIVE STREAMS

- Publisher
- Subscriber
- Subscription
- Processor

PUBLISHER, SUBSCRIBER, SUBSCRIPTION



PROJECT REACTOR

The default implementation for reactive programming for webflux

Two implementation interfaces for publisher: mono and flux

MONO AND FLUX

- Mono: for 0 or 1 (void or 1) object (like Optional)
- Flux: for 0 to n objects (for lists)

MONO

- Sequence of 0 or 1 element
- Models asynchronous operations that produce a single result or possibly no result
- You can think of Mono as a promise or future that will eventually either produce a result or complete without producing any result
- Useful for HTTP GET requests that retrieve a single resource or operations that may or may not result in a single value (e.g., finding a specific entity in the database)

```
Mono<String> monoString = Mono.just("Hello,
WebFlux!");

monoString.subscribe(System.out::println); //
Outputs: Hello, WebFlux!
```

FLUX

- Sequence of 0 to N elements.
- Used to handle streams of data (zero, one, or multiple elements)
- Great for working with sequences of data that are emitted over time, which could be infinite or finite
- Useful for scenarios where you expect multiple items, such as retrieving all records from a database or processing streams of events

```
Flux<String> fluxStrings = Flux.just("Hello",
"Reactive", "World", "with", "WebFlux");

fluxStrings.subscribe(System.out::println);

// Outputs:

// Hello

// Reactive

// World

// with

// WebFlux
```

DEMO

- Mono
- Flux

REACTIVE STREAMS AND OPERATORS

- Operators can be used to manipulate data streams
- Creation operators can be used to create a stream from various data sources: collectors, arrays, hard coded items
- Transformation operators: map, flatMap, buffer
- Filtering operators: filter, distinct
- Combining operators: merge, concat, zip
- Error handling operators: onErrorResume, retry
- Utility operators: doOnNext, delayElements

DEMO

- Map
- Flatmap
- Merge
- Concat
- Zip

EXERCISE

- Practice reactive programming operators with Project Reactor
- Java file with exercises, steps and hints below the class

REACTIVE REST API

Two options: annotated controllers and functional endpoints

The annotated controllers use the same annotations, but from webflux package

Return mono or flux with whatever object you'd need

DATA SOURCE

SPRING DATA R2DBC FOR SQL DB'S

SPRING DATA REACTIVE MONGODB AND OTHER NOSQL DATABASES

DEMO

- REST API
- SQL database

DEMO: CREATE A RESTFUL API WITH WEBFLUX

EXERCISE

Develop a RESTful API for a Personal Budget Tracker that supports creating, retrieving, updating, and deleting financial transactions (incomes and expenses).

The application should be reactive, utilizing Spring WebFlux and R2DBC for database interactions with MySQL.

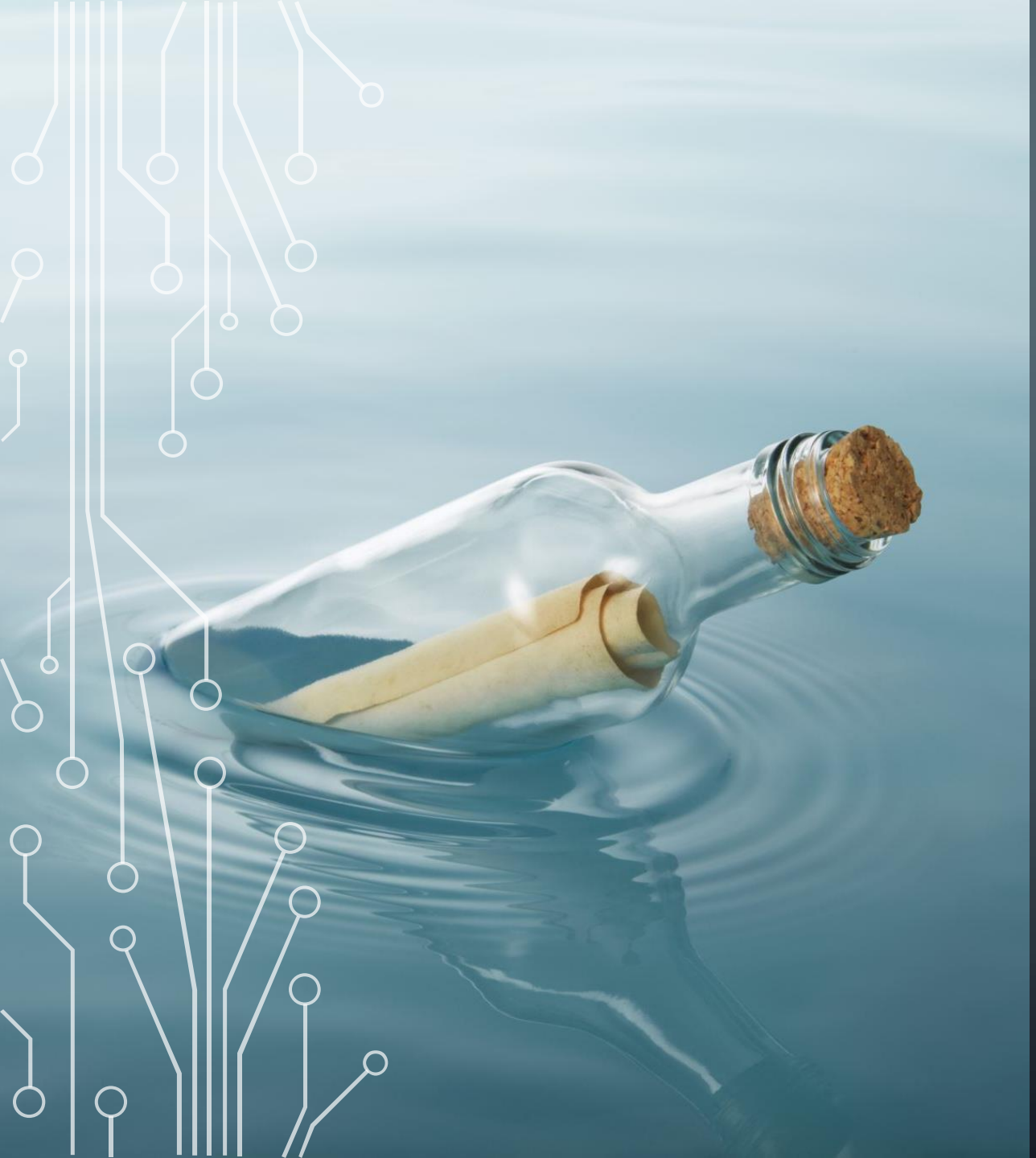
KAFKA & SPRING WEBFLUX

- Kafka is a distributed streaming platform that excels at handling high volumes of data in real-time
- Spring WebFlux provides a reactive programming model for building non-blocking and asynchronous web applications
- Combined, they enable the development of systems that can efficiently process and react to streams of events or data with backpressure support

DEMO: CREATE A DASHBOARD WITH REALTIME DATA

EXERCISE

You will create a system that simulates receiving real-time weather data (e.g., temperature, humidity) from multiple "stations" and streams this data to a web client dashboard in real-time using SSE. The system will use Kafka for message passing.



QUESTIONS?

- Maaike.vanputten@brightboost.nl
- Or Whatsapp: +31683982426
- Don’t hesitate to contact me, I love to help!