Spring Framework

About Me

- Freelance since 2022
- Software developer since 2018
- 2018 Master’s degree at UCLouvain

- Playing with Spring Boot since 2018

- Stack: Spring Boot + Angular + AWS

- navez.jerome@gmail.com

Java OOP - reminder

- Object-Oriented Programming
- Programming paradigm
- Define interaction of “bricks” called “objects”
- Represent concepts, ideas, or real life physical entity
  - A book, a car, the page of a book, authentication information, etc
- OOP represent the objects and their relations

Relation example – Class Diagram

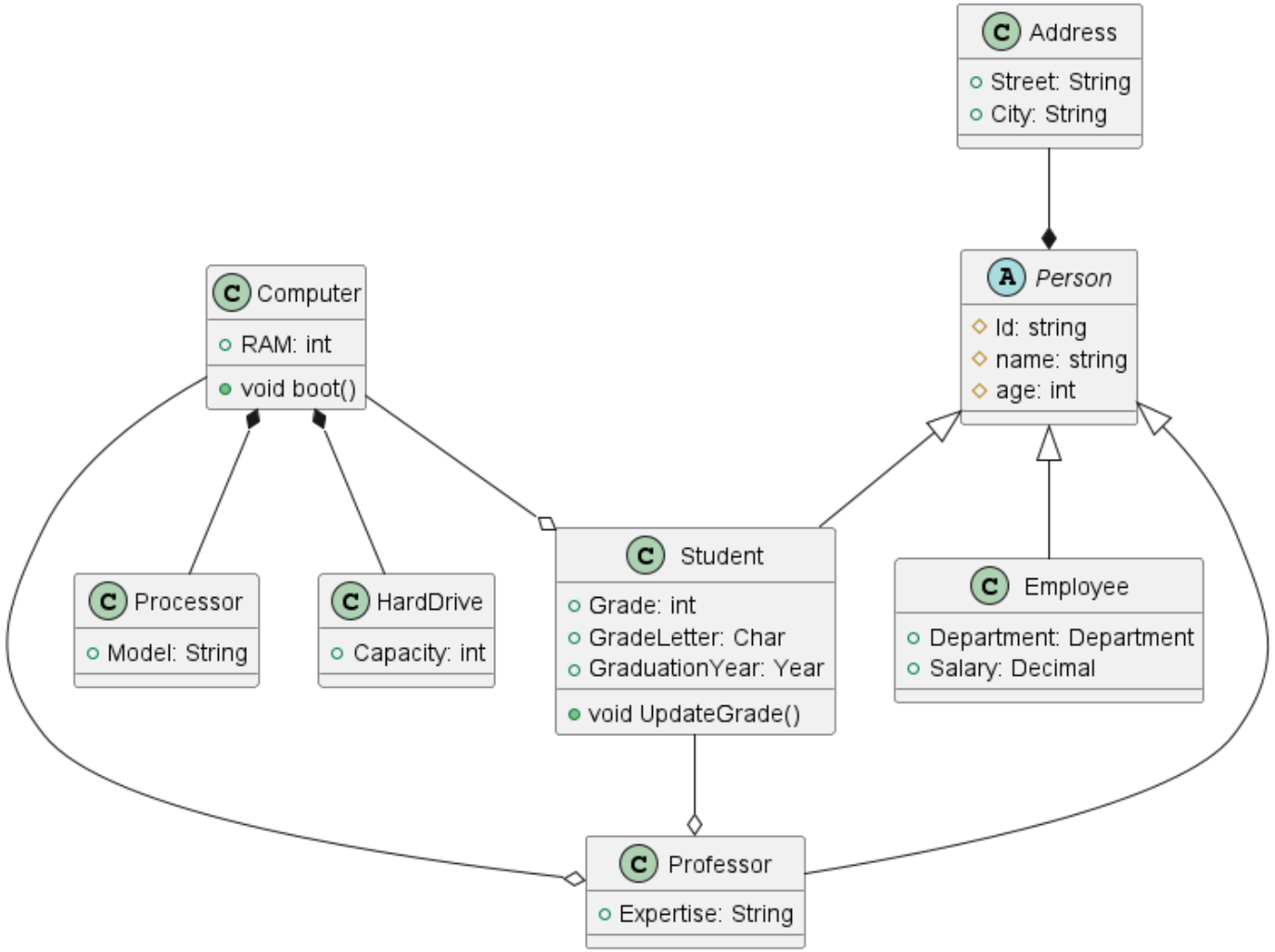


Diagram representing the relations between your classes.

Objects and Classes

- A Class is a definition/template for Objects
  - The definition of a car
  - The definition of a Student

- An object is an instantiation of the Class
  - The blue Ford Mustang
  - John Doe being in 4th grade, having 26 years old and living in brussels.

- There is multiple objects for the same class

Objects and Classes – Java Example

```java
public class Car {
    String brand;
    String model;
    String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public void drive() {
        // ...
    }
}
```

```java
new Car("Ford", "Mustang", "blue");

new Car("Renaud", "Clio", "red");

Car car = new Car("Mercedes", "CLA", "white");
car.drive();
```

Encapsulation

```java
public class Car {
    String brand;
    String model;
    String color;

    List<Wheel> wheels;
    Motor motor;
}
```

```java
public class Motor {
    int size;
    int horsePower;
    String fuelType;
}
```

```java
public class Wheel {
    String brand;
    double size;
}
```

Inheritance & Polymorphism

```
public abstract class Vehicle {
    String brand;
    String model;
    String color;
    List<Wheel> wheels;

    public void drive() {
        // ...
    }
}
```

```
Car car = new Car();
Bike bike = new Bike();

car.drive();
bike.drive();
```

```
public class Car extends Vehicle {
    Motor motor;
}
```

```
public class Bike extends Vehicle {
    String pedals;
}
```

Abstract Classes

- A class that cannot be instantiated
  - Only non abstract children can be used to create objects of this type
- Can contains abstract methods with no implementation
- Abstract classes are used to create polymorphism

Interfaces

- Similar to abstract classes
- You cannot create objects of the type of the interface
  - Even from the children
- You can still use polymorphism by implementing an interface
- A class can implement multiple interfaces

Interfaces – Example

```
public interface Cleanable {
    void clean();
}
```

```
public class Boat implements Cleanable {
    @Override
    public void clean() {
        // ...
    }
}
```

```
public class Room implements Cleanable {
    @Override
    public void clean() {
        // ...
    }
}
```

What is Spring?

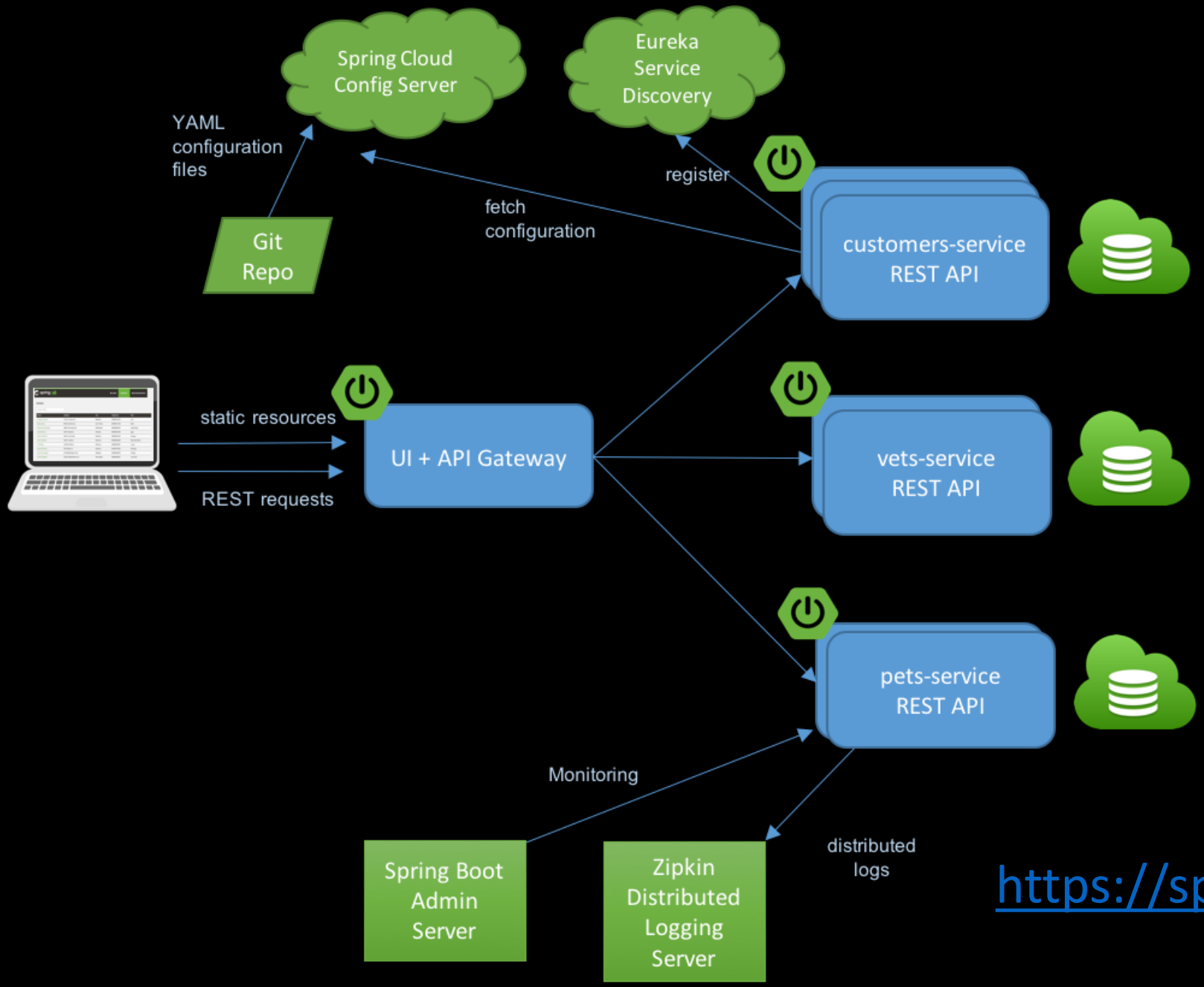
- Java Framework for Java EE
- Lightweight infrastructure
- Used in any layer of your application (from view to repository)
- Enforce loose coupling of dependencies
- Comes with various modules (web, data, security, test, etc)

What is Spring Boot?

- An extension of Spring Framework
- Used for rapid application development (All-in one)
- Embedded Tomcat server
- Starter dependencies (plug and play dependencies)
- Automated configuration

Brief History of Spring

- Spring Framework first release: 2002 
- Spring Boot first release: 2014 
- Currently
  - Spring 6
  - Spring Boot 3
  - Moved to Jakarta EE

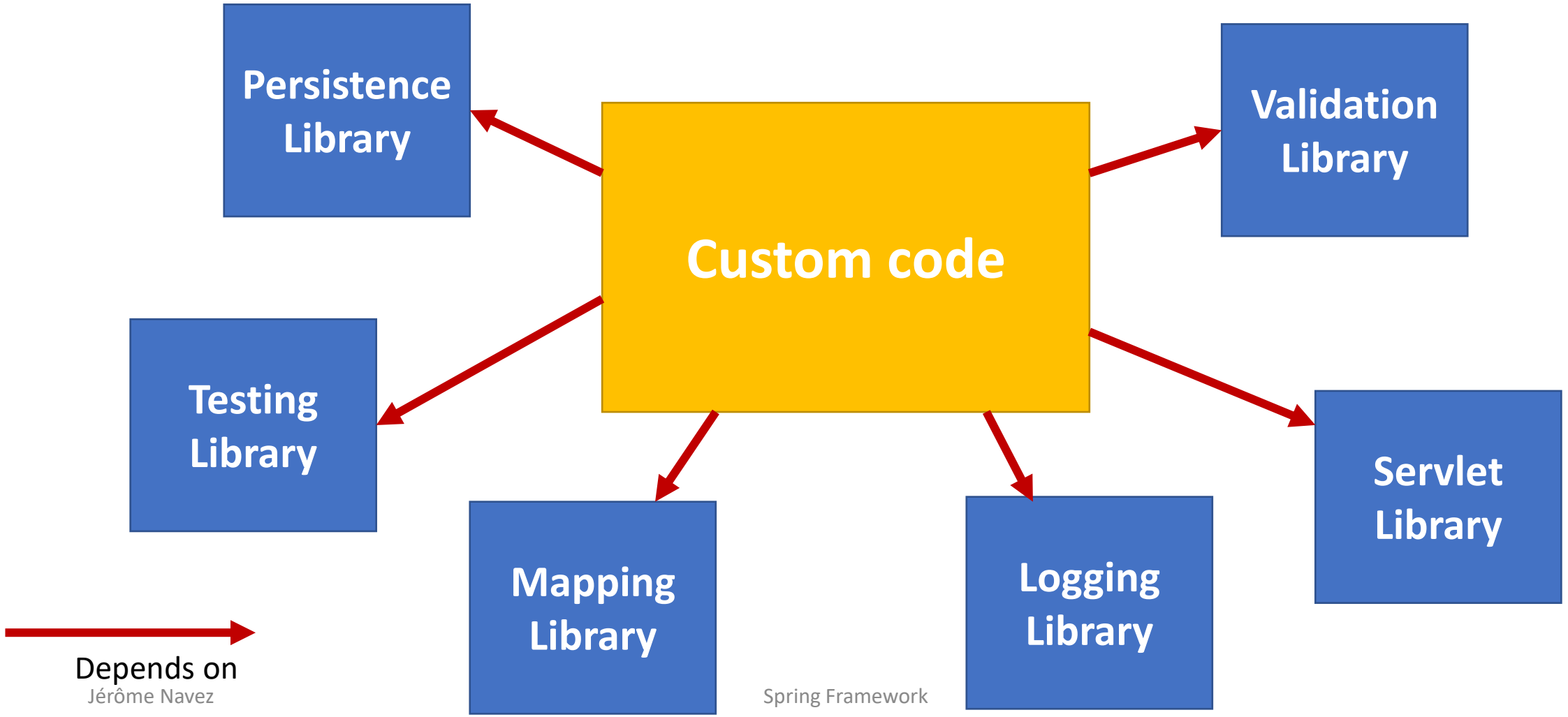


https://spring-petclinic.github.io/

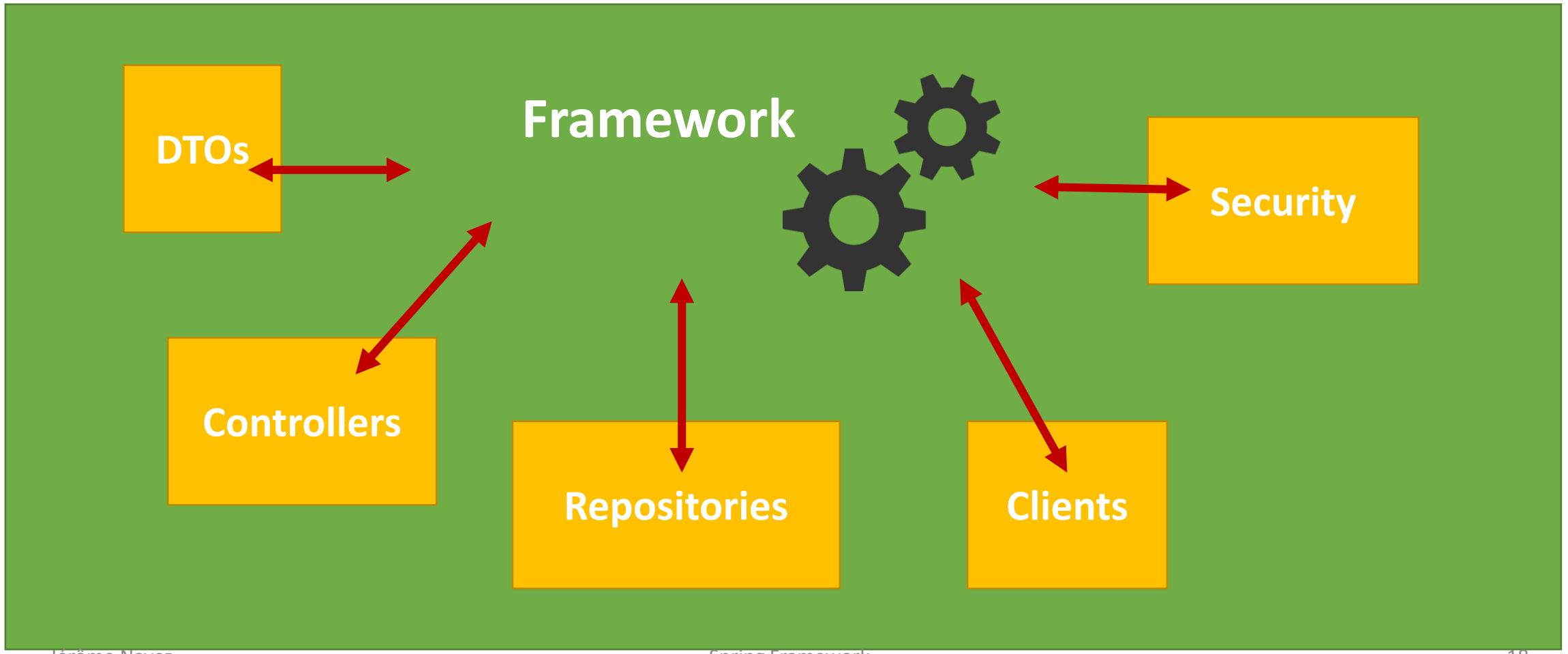
Inversion of Control

- The “natural” way of coding:
  - Custom code calls libraries
- Inversion of Control:
  - Custom code is called by the framework
- Related to the Dependency Inversion principle
  - But it’s not the only mechanism

When Using Libraries



Inversion of Control



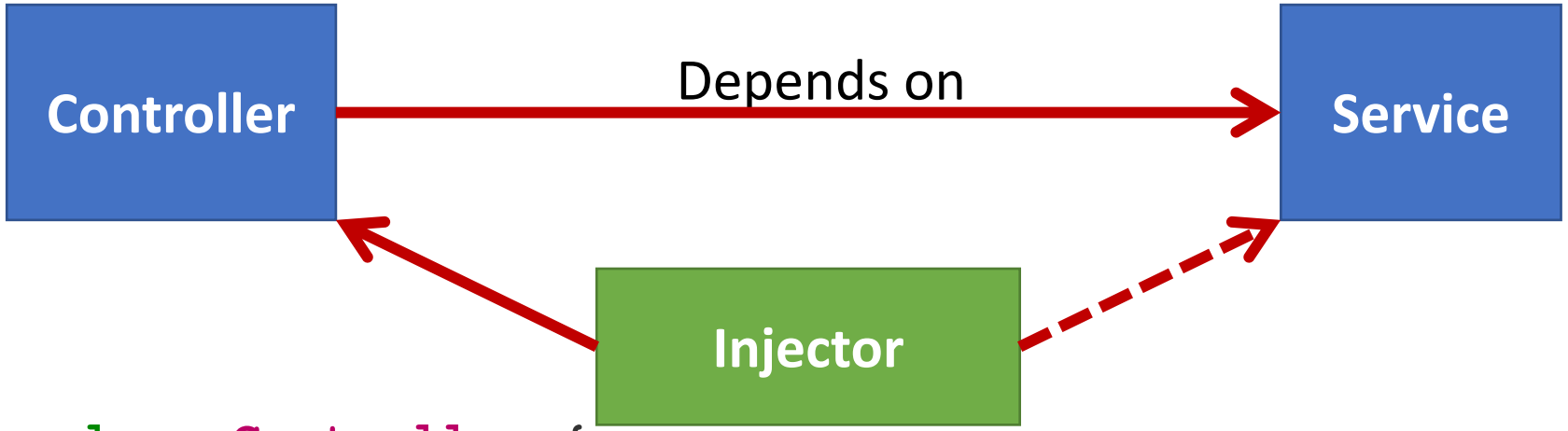
Spring Framework

- Spring Web
- Spring Security
- Spring Data
- Spring Cloud
- And many more

Dependency Injection – Before


```
public class Controller {

    private Service service;

    public Controller() {
        this.service = new Service();
    }
}
```

Dependency Injection

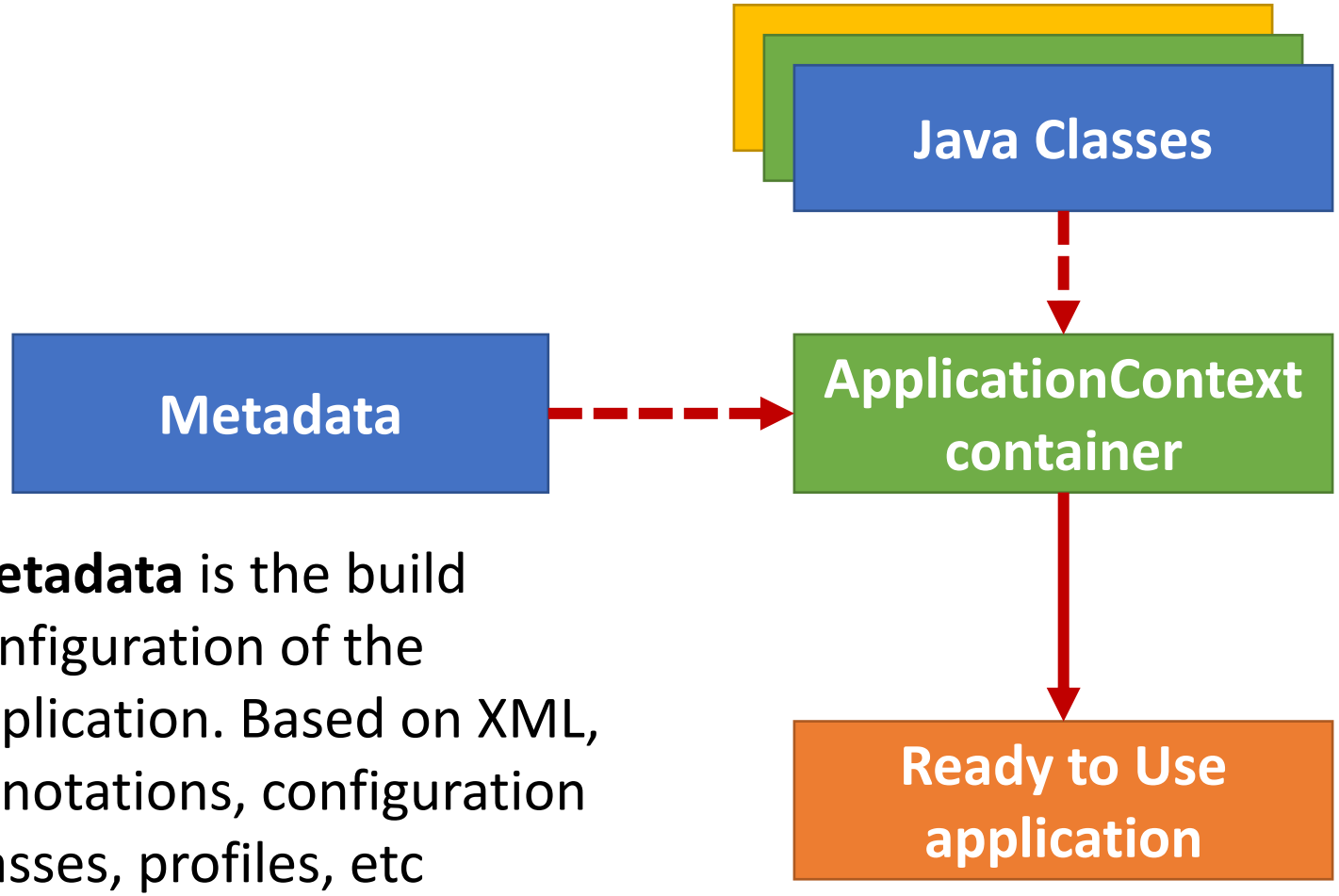



```
public class Controller {
    private Service service;

    public Controller(Service service) {
        this.service = service;
    }
}
```

The injector inject the dependency B in A

- **Via constructor**
- Via method
- Via annotation

Spring ApplicationContext Container



Metadata is the build configuration of the application. Based on XML, annotations, configuration classes, profiles, etc

ApplicationContext is the Injector. It build the application by wiring the dependencies based on configurations (metadata) and Java classes.

Spring dependencies – Beans

- In Spring, **`Controller`** and **`Service`** are called **Beans** and are singletons
- Spring can inject its own dependencies to yours
- Beans are building blocks of our application
  - Custom or provided by Spring
- Spring scans classes and creates Beans at start up
  - Using reflection

Spring Framework – Some Types of Bean

- **`@Controllers`**
  - Beans receiving parsed HTTP requests
- **`@Services`**
  - Beans being dependency of your application
- **`@Repositories`**
  - Beans used to persist data

Stereotype Annotations being sub-types of **`@Component`**

Creating a Bean – Using Annotation

```java
@Service
public class Kitchen {
    private final Oven oven;

    public Kitchen(Oven oven) {
        this.oven = oven;
    }

    public void cook() {
        // ...
        oven.bake();
        //...
    }
}
```

```java
@Service
public class Oven {
    public void bake() {
        // ...
    }
}
```

`ApplicationContext` will build (wire) the application by looking for the classes having the annotation **`@Service`**.

Creating a Bean – Using Configuration Class

```
@Configuration
public class HomeConfiguration {

    @Bean
    public Kitchen kitchen(Oven oven) {
        return new Kitchen(oven);
    }

    @Bean
    public Oven oven() {
        return new Oven();
    }
}
```

`@Configuration` classes are used to define beans by building them manually.

Useful to create beans from classes that are out of our control.

Allow us to have a better control over the configuration of our beans

Creating a Bean – Using XML configuration

Less and less used

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="oven" class="com.woodchopper.tuto.onion.Oven"/>
    <bean id="kitchen" class="com.woodchopper.tuto.onion.Kitchen">
        <constructor-arg ref="oven" />
    </bean>
</beans>
```

Coding Session 1 – Set up your first project

- Java 17
- SpringBoot 3
- Maven
- Use https://start.spring.io/

Spring Web

- Spring module used to handle HTTP requests

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

GetMapping

http://domain.com/items/123 -> 123 is the **id**

```
@RequestMapping("items")
@RestController
public class ItemController {
    @GetMapping("{id}")
    public ItemDto getItem(@PathVariable String id) {
        // ...
    }
}
```

PostMapping

`itemDto` Is the body of the request

```
@RequestMapping("items")
@RestController
public class ItemController {
    @PostMapping
    public ItemDto addItem(@RequestBody ItemDto itemDto) {
        // ...
    }
}
```

@RequestParam

http://domain.com/items?type=italian&rating=4

```
@RequestMapping("items")
@RestController
public class ItemController {
    @GetMapping
    public ItemDto findItem(@RequestParam("type") String type,
                            @RequestParam("rating") Integer rating) {
        // ...
    }
}
```

Spring Data JPA

- Spring dependency
- Providing auto-configuration classes and beans
- An implementation of JPA using Hibernate

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Java Persistence API

- JPA for short
- The Object-Relational Mapping of Java (ORM)
- Used to represent the database into an OO model
- Composed of
  - Annotations
  - JPQL (Java Persistence Query Language)

Hibernate

- A JPA implementation
- An ORM implementation
- By Red Hat

H2 – In-memory Database

- Light in-memory database
- Alternative to MySQL, PostgreSQL
- Ideal for testing
- Maven dependency

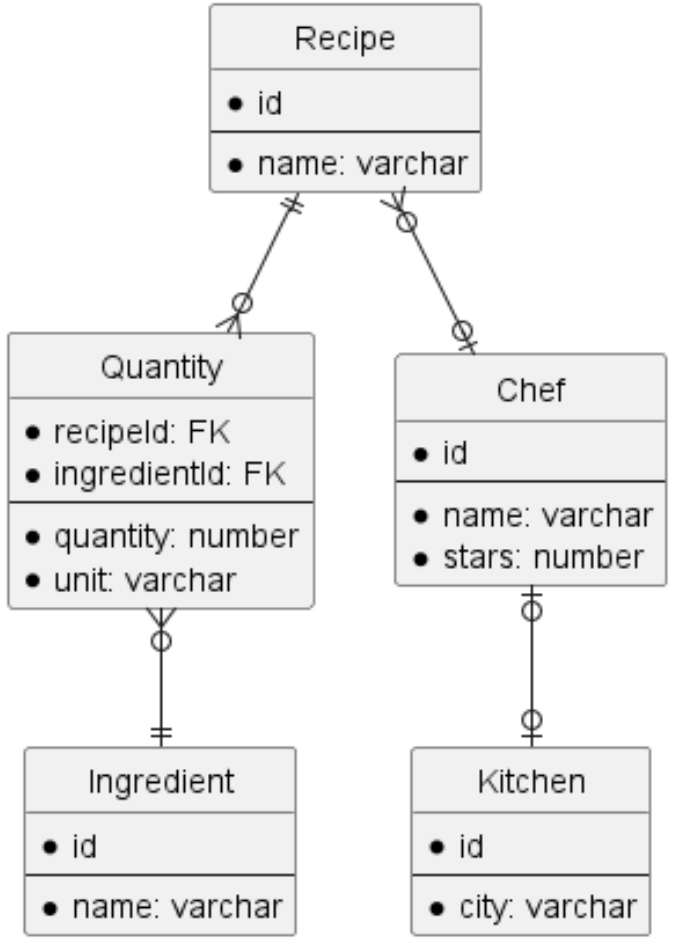
H2 – Set Up

```xml
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

```yaml
#application.yaml
spring:
  datasource:
    url: jdbc:h2:file:./db
    driverClassName: org.h2.Driver
    username: sa
    password: password
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
  h2:
    console:
      enabled: true
```

Chefs & Recipes

- Chefs are able to cook recipes
- Chefs have their own kitchen
- A recipe is composed of ingredients with a quantity



Recipe Entity – ManyToOne

```java
import jakarta.persistence.*;

@Entity
@Table(name = "recipe")
public class Recipe {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column
    private String name;

    @ManyToOne
    private Chef chef;

    @OneToMany(mappedBy = "recipe")
    List<QuantityEntity> quantities;

}
```

Chef Entity – OneToMany – OneToOne

```java
@Entity
@Table(name = "chef")
public class Chef {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false)
    private String name;

    @OneToOne
    private Kitchen kitchen;

    @OneToMany(mappedBy = "chef")
    private List<Recipe> recipes;
}
```

Kitchen Entity – OneToOne

```java
@Entity
@Table(name = "kitchen")
public class Kitchen {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false)
    private String city;

    @OneToOne
    private Chef chef;
}
```

Ingredient Entity

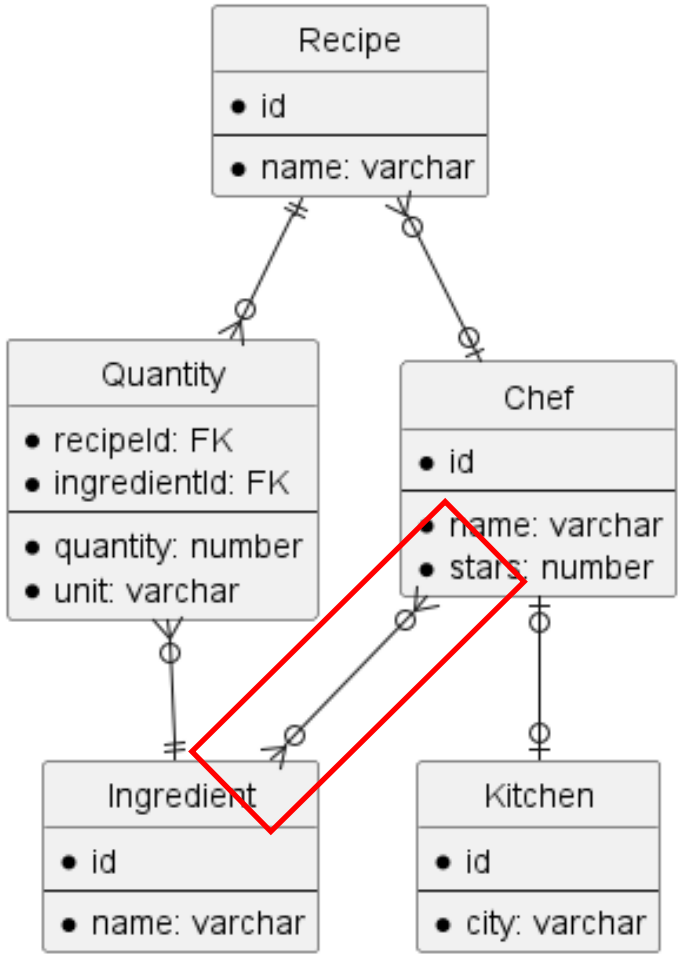
```
@Entity
@Table(name = "ingredient")
public class Ingredient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "ingredient", cascade = CascadeType.ALL)
    private List<QuantityEntity> quantities = new ArrayList<>();


}
```

Quantity Entity – Composite Key

```
@Entity
@Table(name = "quantity")
public class Quantity {
    @EmbeddedId
    private QuantityId quantityId;

    @Column(nullable = false)
    private int quantity;
    @Column
    private String unit;

    @MapsId("recipeId")
    @ManyToOne
    private Recipe recipe;

    @MapsId("ingredientId")
    @ManyToOne
    private Ingredient ingredient;
}
```

Chefs & Recipes – New Rule!

- Chefs have their preferences and can like ingredients



Chefs' Favourite Ingredients – ManyToMany

```java
@Entity
@Table(name = "chef")
public class Chef {
    // ...

    @ManyToMany
    @JoinTable(
    name = "favourite",
    joinColumns =
        @JoinColumn(name = "chef_id"),
    inverseJoinColumns =
        @JoinColumn(name = "ingredient_id"))
    private Set<Ingredient> favouriteIngredients;
}
```

```java
@Entity
@Table(name = "ingredient")
public class Ingredient {
    // ...

    @ManyToMany(mappedBy =
        "favouriteIngredients")
    private Set<Chef> chefsFavourites;
}
```

Executing queries

- JPQL queries
- Query methods
- Query Specifications
- **`JpaRepository<Entity, ID>`**

JPQL Queries

- Java Persistence Query Language
- Similar to SQL but using the naming of your classes

```
@Repository
public interface ChefRepository extends JpaRepository<Chef, Long> {
    @Query("SELECT c FROM Chef c WHERE c.name LIKE :name")
    Chef findByNameLike(@Param("name") String name);
}
```

Query Methods

- Interface methods being interpreted as queries
- Write the behaviour, spring does the rest

```
Chef findById(long id);

Chef findByNameLike(String name);

void deleteById();
```

Unit Tests – Why?

- Make sure that no one break your feature
- Facilitate refactoring
- Find bugs earlier
- Improve confidence on the code
- Easier debugging
- Provides documentation
- Ensure that a bug is fixed

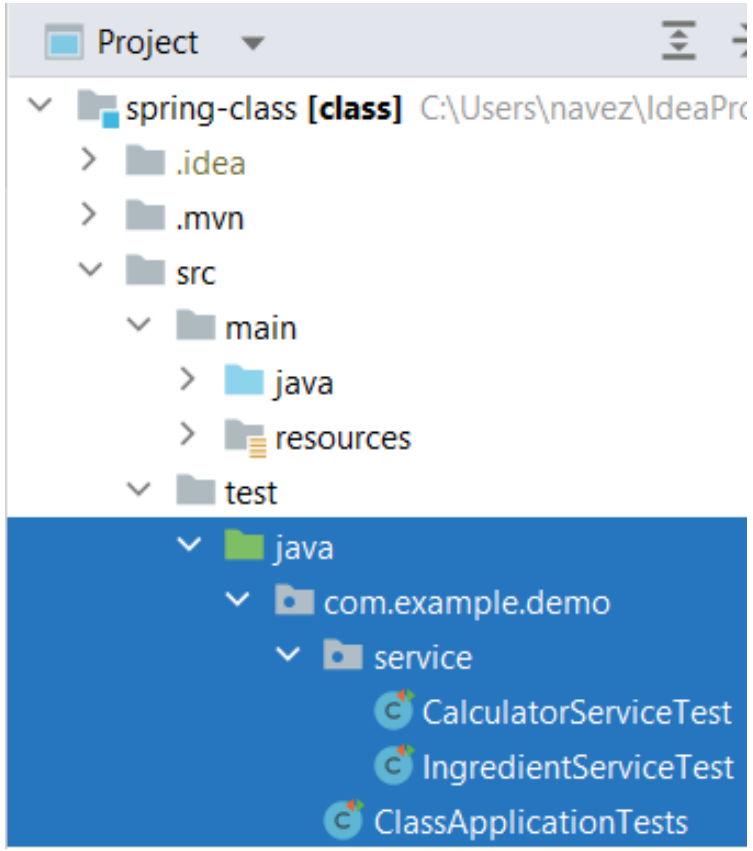
Unit Tests – Good practices

- One test for one purpose / One scenario per test
- Readable simple tests (AAA – Arrange, Act, Assert)
- Avoid real API calls
- You should be able to run all your tests at once
- Isolate your test by mocking the dependencies
- Deterministic tests

Testing a Spring Application

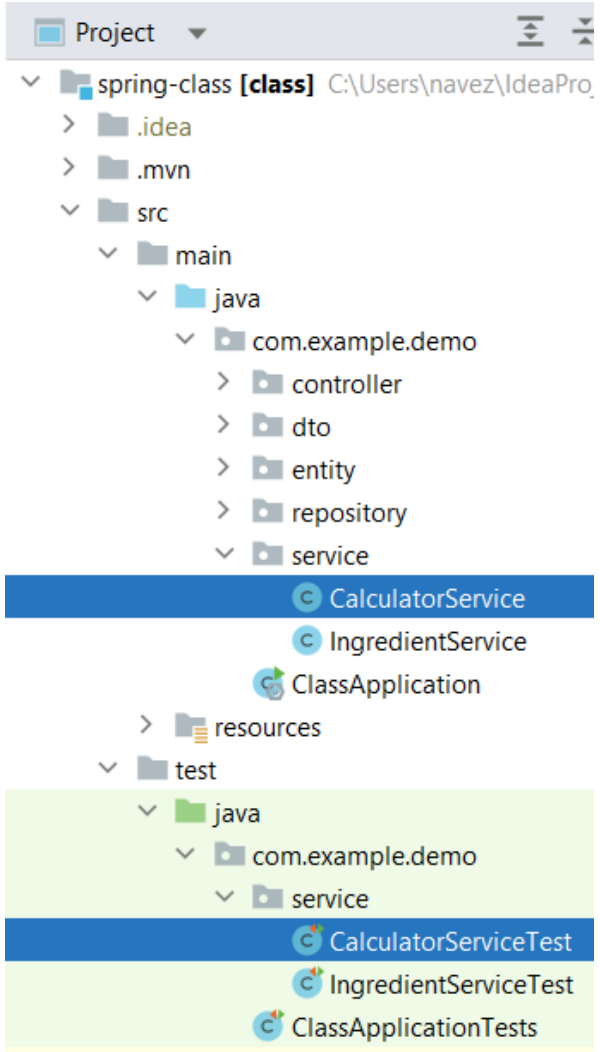
- JUnit 5
  - Testing framework
- MockMvc
  - Mock HTTP requests
- Mockito
  - Mock dependencies
- H2
  - In-memory database

Testing in a Maven Project



Tests are written in the **test** folder

Testing a Class



Each class has its own testing class

`CalculatorService` is tested by **`CalculatorServiceTest`**

Structure of a Unit Test

```java
class CalculatorServiceTest {

    private CalculatorService service = new CalculatorService();

    @Test
    void sumList() {
        // Arrange the data of the test
        List<Integer> integers = List.of(1, 2, 3, 4);

        // Act, execute the method call
        int sum = service.sumList(integers);

        // Assert the result
        assertEquals(10, sum);
    }
}
```

Mocking a dependency

```
class IngredientServiceTest {

    // We create a service using a mock instead of a real repository
    private IngredientRepository repository = Mockito.mock(IngredientRepository.class);
    private IngredientService service = new IngredientService(repository);

    @Test
    void getIngredient_isFound() {
        // Arrange
        IngredientEntity entity = new IngredientEntity("Egg");
        Mockito
            .when(repository.findById(anyInt())) // mock the method call
            .thenReturn(Optional.of(entity)); // fake the return value

        // Act
        String ingredient = service.getIngredient(1);

        // Assert
        assertEquals("Egg", ingredient);
    }
```

Verifying Calls to dependencies

```
@Test
void update() {
    // Arrange
    // ArgumentCaptor will capture the arguments sent to our mocked method
    ArgumentCaptor<IngredientEntity> captor = ArgumentCaptor.forClass(IngredientEntity.class);
    Mockito.when(repository.save(captor.capture())).thenReturn(new IngredientEntity());

    // Act
    service.update(5, "Mascarpone");

    // Assert that the repository has been called with the right id and name
    IngredientEntity captorValue = captor.getValue();
    assertEquals("Mascarpone", captorValue.getName());
    assertEquals(5, captorValue.getId());
    // captor.capture() will capture the argument
    // captor.getValue() returns the captured argument
}
```

Integration Tests

- Unit tests verify a single class/method of your application
- Integration tests verify the whole flow of your application
  - From controller to database
- Unit tests will try to cover all the edge cases of a specific method
- Integration tests will cover the happy flow and possible integration issues

Tools

- MockMvc will mock HTTP requests to your controller
- H2 will be used to create a temporary in-memory database

MockMvc – Set Up

```
@SpringBootTest
@AutoConfigureMockMvc // Allows us to import MockMvc
class IngredientControllerTest {
    // Test dependencies are imported using @Autowired
    @Autowired
    MockMvc mockMvc;

    // EntityManager is a low-level repository
    @Autowired
    EntityManager em;

    // Used to transform objects to JSON
    ObjectMapper objectMapper = new ObjectMapper();

    @Test
    void testAdd() throws Exception {
        // ...
    }

}
```

MockMvc – Usage

```
@Test
void testAdd() throws Exception {
    // Arrange
    IngredientRequestDto ingredientRequestDto = new IngredientRequestDto();
    ingredientRequestDto.setName("Mascarpone");
    String content = objectMapper.writeValueAsString(ingredientRequestDto);
    // Act
    byte[] result = mockMvc.perform(
            post("/ingredients")// We create a POST request on /ingredients
                .content(content)// The body is {"name": "Mascarpone"}
                .contentType(MediaType.APPLICATION_JSON)
        )
        .andReturn()// The id is returned in byte[]
        .getResponse()
        .getContentAsByteArray();
    // Assert
    int resultInt = Integer.valueOf(new String(result));// byte -> int
    // We verify that the entity is well saved using the EntityManager em
    IngredientEntity entity = em.find(IngredientEntity.class, resultInt);
    assertEquals("Mascarpone", entity.getName());
}
```