2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

Spring Boot — HTTP API Exception Handling and Input Validation

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- HTTP API
  - Exception Handling
  - Input Validation
  - Domain Validation
  - HTTP — Java
- Layered Architecture
- Spring Boot
  - Dependency Injection

HTTP API

- Exception Handling
- Input Validation
- Domain Validation
- HTTP — Java

EXCEPTION HANDLING

```
@RestController
@RequestMapping("/pony")
public class PonyController {

    // ...

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler({RuntimeException.class})
    public Map<String, String> handleRuntimeException(RuntimeException ex) {
        Map<String, String> errors = new HashMap<>();
        errors.put("error", ex.getMessage());
        return errors;
    }
}
```

- Requests will sometimes ask for invalid things that will be rejected by the domain model or the service with an exception
- These exceptions can be translated to an HTTP response with a suitable status code

EXCEPTION HANDLING

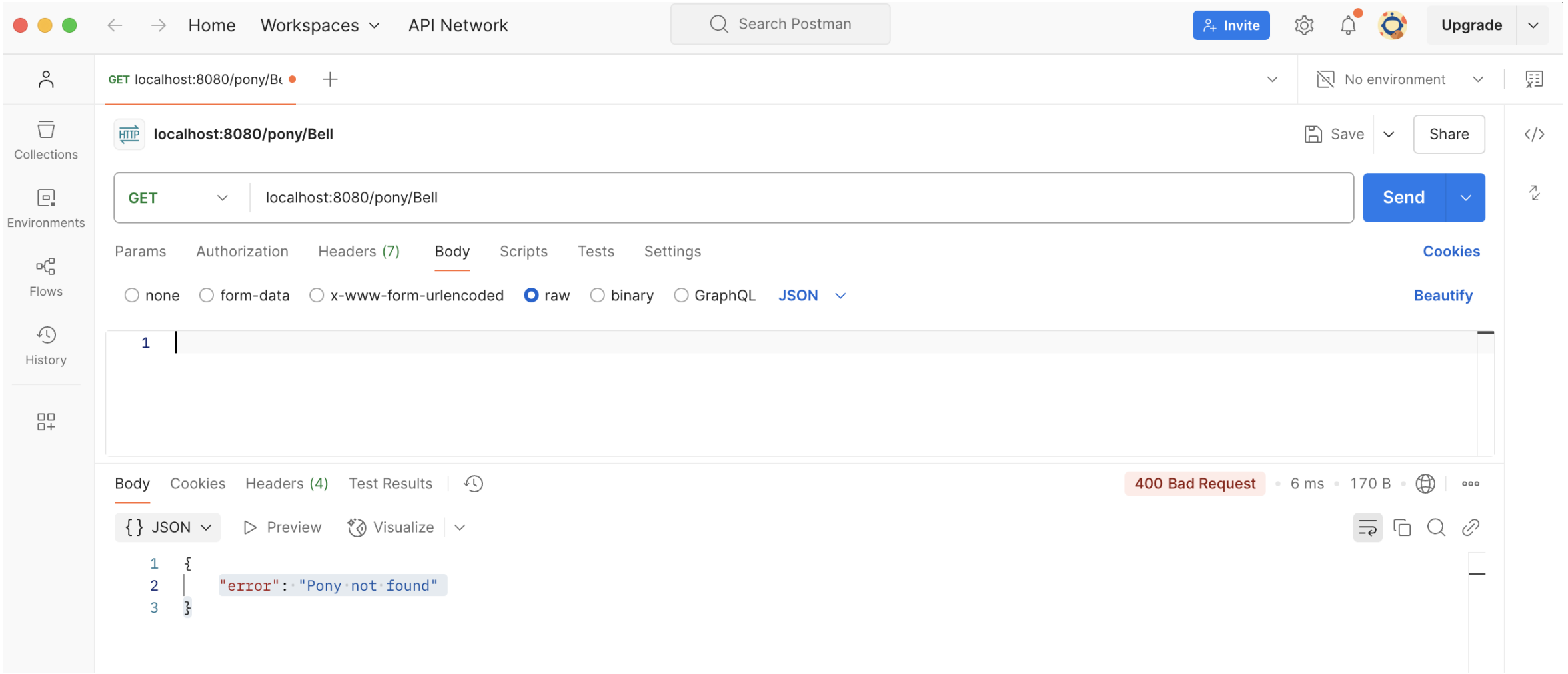
```
@RestController
@RequestMapping("/pony")
public class PonyController {

    // ...

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler({RuntimeException.class})
    public Map<String, String> handleRuntimeException(RuntimeException ex) {
        Map<String, String> errors = new HashMap<>();
        errors.put(”error", ex.getMessage());
        return errors;
    }
}
```

- The @ResponseStatus annotation governs the HTTP status code of the response
- The method its return value will be translated to JSON like usual

EXCEPTION HANDLING



EXCEPTION HANDLING

- You can add as many exception handlers as you want
  - This enables translation of each exception to a suitable HTTP response

- If an unhandled exception passes the controller it will be translated to a default response with a status code of 500

INPUT VALIDATION

- It is common practice to use two kinds of validation
  - *Domain validation*: custom written validation in the model and services for complex cases
  - Use of a validation framework to do *input validation* at the boundary of the backend

- Input validation checks that requests are well formatted
  - Are dates in the correct format?
  - Are numbers in the expected ranges?
  - Do all Strings have the expected length?

- JSR 380 is the Java specification that governs input validation
  - We will perform input validation via the "Hibernate Validator" library, an implementation of JSR 380

INPUT VALIDATION

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

- You can include Hibernate Validator via the dependency above

INPUT VALIDATION

```
// ...
import jakarta.validation.Valid;
// ...

@RestController
@RequestMapping("/pony")
public class PonyController {

    // ...

    @PostMapping
    public Pony addPony(@Valid @RequestBody Pony pony) {
        return ponyService.addPony(pony);
    }

    @PutMapping("/{name}")
    public Pony updatePony(@PathVariable String name, @Valid @RequestBody Pony pony) {
        return ponyService.updatePony(name, pony);
    }
}
```

- You can enable Hibernate Validator for a specific operation by adding @Valid before a parameter
- If the validation fails a MethodArgumentNotValidException is thrown

INPUT VALIDATION

```
public class Pony {

    @NotBlank(message = "name may not be empty")
    private String name;

    @Positive(message = "age must be positive")
    private int age;

    @Min(value = 40, message = "pony must be larger than 40 cm")
    @Max(value = 147, message = "pony must be smaller than 147 cm")
    private int size;

    @Valid
    private Saddle saddle;

    public Pony(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ...

}
```

- Hibernate Validator works by adding restrictions to the fields of a class
- These are checked before the controller method is entered if @Valid is present

INPUT VALIDATION

```
public class Pony {

    @NotBlank(message = "name may not be empty")
    private String name;

    @Positive(message = "age must be positive")
    private int age;

    @Min(value = 40, message = "pony must be larger than 40 cm")
    @Max(value = 147, message = "pony must be smaller than 147 cm")
    private int size;

    @Valid
    private Saddle saddle;

    public Pony(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ...

}
```

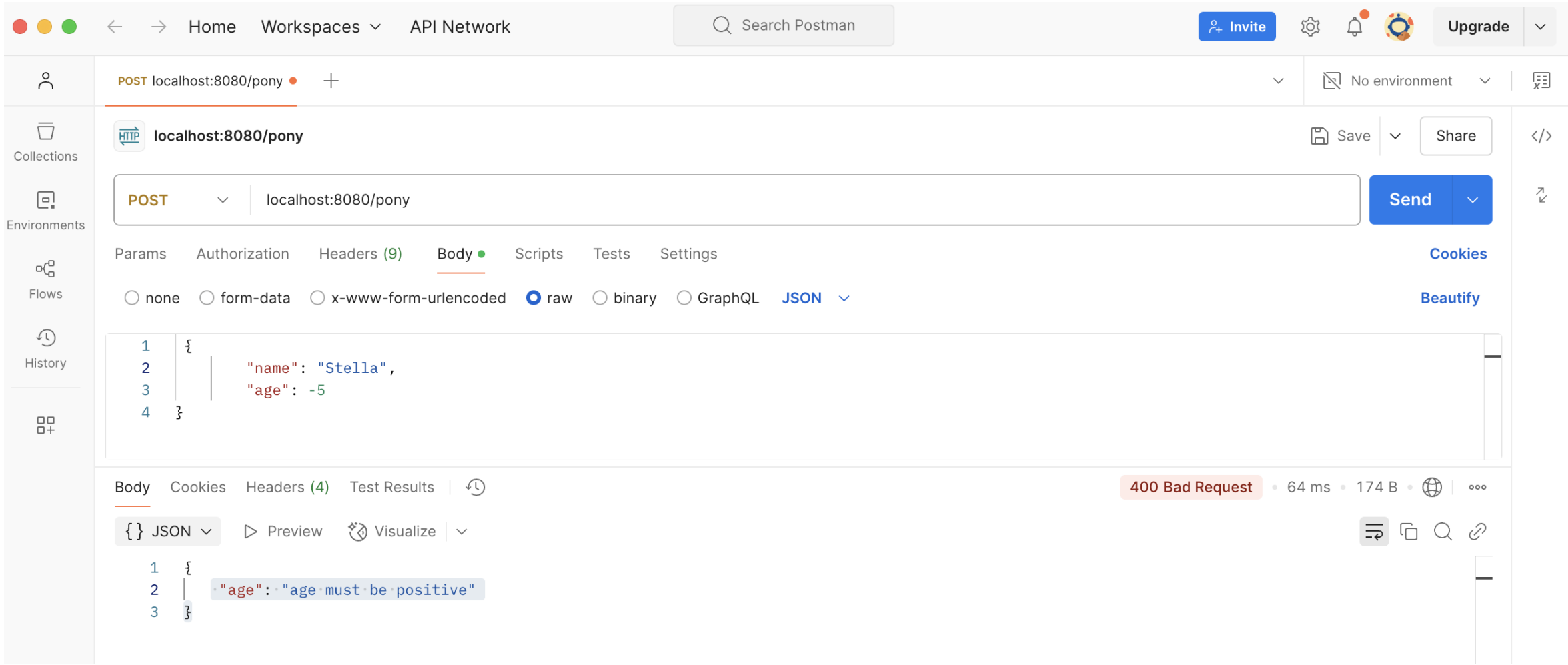
- Hibernate Validator works by adding restrictions to the fields of a class
- You can find more possible restrictions here

INPUT VALIDATION

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler({MethodArgumentNotValidException.class})
public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    for (FieldError error : ex.getFieldErrors()) {
        String fieldName = error.getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    }
    return errors;
}
```

- The controller then needs an ExceptionHandler to translate the exception thrown by Hibernate Validator to an HTTP response

INPUT VALIDATION



INPUT VALIDATION – DESERIALISATION

```java
public class Pony {

    @NotBlank(message = "name may not be empty")
    private String name;

    @Positive(message = "age must be positive")
    private int age;

    @Min(value = 40, message = "pony must be larger than 40 cm")
    @Max(value = 147, message = "pony must be smaller than 147 cm")
    private int size;

    @Valid
    private Saddle saddle;

    public Pony(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ...

}
```

```java
new Pony("", -5); // Ok, does not trigger input validation
```

- Note that input validation is triggered during deserialisation!
  - These checks will only be done when JSON is translated to an object
  - When constructing an object via the constructor this validation is not performed

INPUT VALIDATION – TESTING

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.ValidatorFactory;
import jakarta.validation.Validator;
// ...

public class PonyTest {

    private static ValidatorFactory validatorFactory;
    private static Validator validator;

    @BeforeAll
    public static void createValidator() {
        validatorFactory = Validation.buildDefaultValidatorFactory();
        validator = validatorFactory.getValidator();
    }

    // ...

    @AfterAll
    public static void close() {
        validatorFactory.close();
    }
}
```

- You can test your Hibernate Validator configuration in unit tests
- @BeforeAll runs before all tests in the file and creates the validator
- @AfterAll runs after all the tests in the file and cleans up the validator

INPUT VALIDATION – TESTING

```
public class PonyTest {

    // ...

    @Test
    void givenNegativeAge_whenCreatingPony_thenAgeViolationIsThrown() {
        Pony tornado = new Pony("Tornado", -5);

        Set<ConstraintViolation<Pony>> violations = validator.validate(tornado);

        assertEquals(1, violations.size());
        ConstraintViolation<Pony> violation = violations.iterator().next();
        assertEquals("age must be positive", violation.getMessage());
    }
}
```

- You can invoke the validation manually and then check the results

DOMAIN VALIDATION

- Input validation checks that requests are well formatted
- But what with other cases?
  - A well formatted pony is added twice…
  - The data is formatted nicely but is nonsensical…
- Enter domain validation!

- "Complex" validation rules are enforced by the service and the domain
  - The domain is the owner and should contain as much validation as possible
    - For instance, the creation of a nonsensical object should be blocked by the constructur
  - The service should perform validation that spans multiple objects
    - For instance, the pony class cannot check whether a pony with the same identifier already exists, so this should be handled in the service

WHICH VALIDATION?

- We use the following rules for validation in Back-End Development:
  - If a validation can be coded as input validation, then use input validation
  - If input validation is not possible, then code the validation as validation in the domain model if possible. Use the service layer only if coding in the domain model is not possible.

- Always add an Exception Handler for Domain Validation (*RuntimeException*) and for Input Validation (*MethodArgumentNotValidException*) in every controller

EXERCISES @CLASS

- exercise1
- exercise2
- exercise3
- exercise4
- exercise5



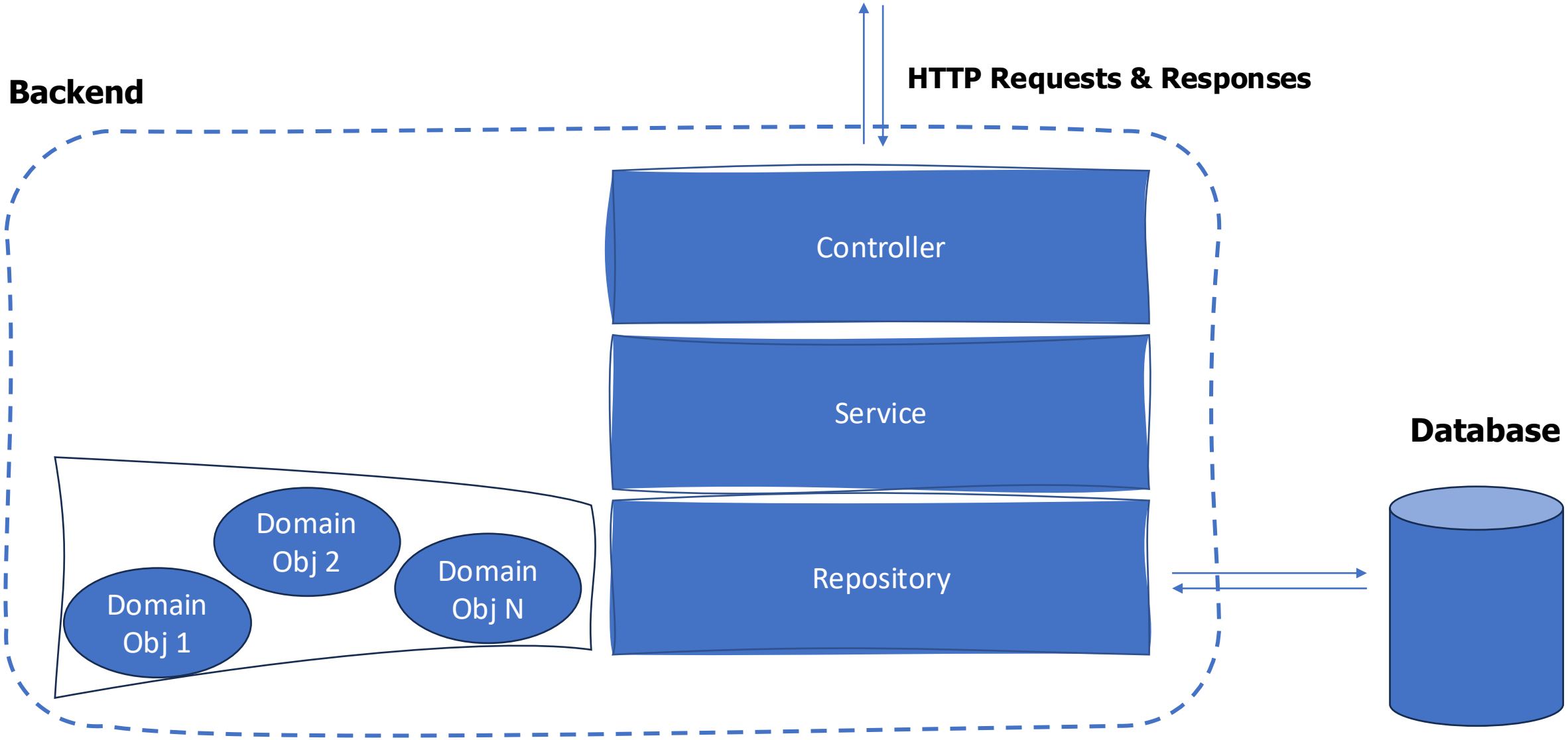
Layered Architecture

- General Request Flow

LAYERED ARCHITECTURE

- A good solution for a small backend is the *Layered Architecture* pattern
  - In a layered architecture, backend code belongs to one of three layers
  - These layers are managed by Spring
- Next to these layers and their responsabilities we have our classes that are application specific
  - These are called *domain objects* and live next to the three layers
  - They model the concepts that our application handles
  - They contain the application logic specific to this application
- The controllers, services and repositories work together with the domain model to implement functionality

GENERAL REQUEST FLOW



GENERAL REQUEST FLOW

- As an example, this is the flow for a call “Update Pony” made on the backend
  1. The Controller receives the HTTP call “Update Pony” and translates all parts of the HTTP message to Java
  2. The validator framework checks the call and does not find input validations
  3. The Controller invokes an operation “Update Pony” on the Service
  4. The Service uses a Repository to retrieve the Pony from storage
  5. The Service and the Pony check if all business constraints for updating are fulfilled
  6. The Service updates the fields of the Pony and instructs the Repository to save the updated Pony
  7. The Repository saves the Pony to storage
  8. The Service returns to the Controller that the operation succeeded
  9. The Controller constructs an HTTP message to return to the frontend indicating that the operation was successful

Spring Boot

- Dependency Injection

SPRING BOOT

- We have seen the annotations @Repository, @Service and @RestController
  - These are all specific versions of the general annotation @Component
- When Spring starts it instantiates all classes annotated with @Component
  - If they have dependencies that are injected via @Autowired these dependencies are searched for in the container and started and injected if found
- To make sure that Spring finds your @Component classes they should be defined in
  - The same package as the class annotated with @SpringBootApplication
  - A sub-package of the class annotated with @SpringBootApplication

SPRING BOOT

- The mechanism in play with @Component and @Autowired is called **D**ependency **I**njection (DI)
  - Spring *injects* the *dependencies* every class needs
  - This allows for loose coupling of components and moves the responsibility of managing components onto Spring

- Dependency Injection is a technique used to implement the principle of **I**nversion **o**f **C**ontrol (IoC)
  - IoC is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework
  - It is not the developer that is in control of the object graph (via instantiating classes manually) but the framework

SPRING BOOT

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

- Spring Boot calls these dependencies Spring Boot Starters
  - They include many dependencies (frameworks and libraries) at once
  - The versions are managed by the Spring Boot parent pom our project inherits from
- Spring Boot automatically configures all the included dependencies
  - For example, by including Spring Web, Jackson (the JSON to Java library) and Tomcat (the system behind receiving HTTP traffic) are automatically configured

SPRING BOOT

```
# Web properties
server.port=8081

# Additional properties
# ...
```

- This automatic configuration of libraries can be disabled or modified
- Modifying it happens via properties in a file called “application.properties” that resides in the directory “src > main > java > resources”
  - We will use this file in the next classes

EXERCISES @HOME

- exercise6
- exercise7



