2025 — 2026

1TI – 1ACS



Back-End Development

JPA — OneToOne and Testing with a database

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

CONTENTS

- OneToOne
- Testing with a Database
  - Unit Testing
  - Component Testing

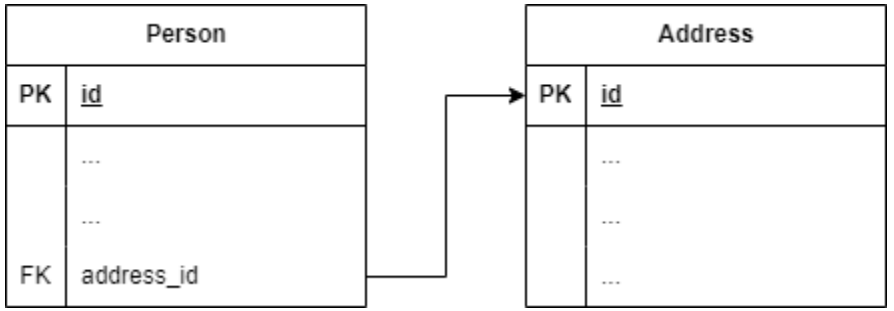
OneToOne

RELATIONSHIPS

- Mapping relationships for JPA is not easy
  - Relationships are managed differently in the database than in code
- Java
  - Relationships between classes are expressed through fields
  - A relationship can be unidirectional or bidirectional
- Relational Database
  - Relationships between tables are expressed through foreign keys
  - A relationship is always unidirectional

RELATIONSHIPS — DATABASE — ONE2ONE

- One row from a table can be linked to one row from a different table
- The table holding the foreign key is called the “owner” of the relationship



RELATIONSHIPS — JAVA & DATABASE

- The complexity of relationships lies in the difference between the database and code
  - In a database a relationship is mostly *unidirectional*
    - Only one table holds the foreign key
  - In code a relationship can be *unidirectional* or *bidirectional*
    - Both objects can hold references to each other

- It is the job of JPA to translate from one "world" to the other automatically
  - To do so, we need to configure the relationships correctly

UNIDIRECTIONAL ONETOONE — CONFIG

```java
@Entity
@Table(name = "person")
public class Person {

    @OneToOne
    @JoinColumn(name = "address_id")
    private Address address;
}
```

```java
@Entity
@Table(name = "address")
public class Address {

    // ...
}
```

- The field that maps to the column with the foreign key is marked with @JoinColumn
- The entity that holds the foreign key is called the “owning-side”

RELATIONSHIPS — HOW TO CODE

Code the correct configuration for the intended relationship in the entities.

- POST Requests, in the service:
    1. Get or create the necessary domain objects
        1. Via the requestbody or via the repositories
    2. Save the non-owning side via the repository (store result in variable!)
    3. Connect the participating domain objects
    4. Save the owning side via the repository

- DELETE requests, in the service:
    1. Get the necessary domain objects via the repositories
    2. Break the relationship between the participating domain objects
    3. Delete the requested entities via the repositories

- PUT requests, in the service:
    1. Make sure that relationships are taken into account when validating the new information

UNIDIRECTIONAL ONETOONE — POST EXAMPLE

```java
@Entity
public class Pony {

    // ...

    @OneToOne
    @JoinColumn(name = "owner_id")
    private Owner owner;

    // ...

    public void setOwner(Owner owner) {
        this.owner = owner;
    }

    public Owner getOwner() {
        return owner;
    }
}
```

```java
@Entity
public class Owner {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // For JPA, not public on purpose
    protected Owner() {}

    public Owner(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

UNIDIRECTIONAL ONETOONE — POST EXAMPLE

```
DROP TABLE IF EXISTS PONY;
DROP TABLE IF EXISTS OWNER;

CREATE TABLE OWNER(
    ID BIGINT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(255)
);

CREATE TABLE PONY(
    ID BIGINT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(255),
    AGE INT,
    OWNER_ID BIGINT,
    FOREIGN KEY(OWNER_ID) REFERENCES OWNER
);
```

- The database schema now includes a foreign key to model the relationship

UNIDIRECTIONAL ONETOONE — POST EXAMPLE

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

    // ...

    @PostMapping("/{name}/owner")
    public Pony addOwner(@PathVariable String name, @Valid @RequestBody Owner owner) {
        return ponyService.addOwner(name, owner);
    }
}
```

- The controller has a new endpoint to add an owner to a pony

UNIDIRECTIONAL ONETOONE — POST EXAMPLE

```java
@Service
public class PonyService {

    // ...

    public Pony addOwner(String name, Owner owner) {
        Pony pony = findPonyByName(name);
        owner = ownerRepository.save(owner);
        pony.setOwner(owner);
        return ponyRepository.save(pony);
    }
}
```

```java
public interface OwnerRepository extends JpaRepository<Owner, Long> {}
```

- The service retrieves the pony from the database, saves non-owning side, connects the objects and saves the owning side
- If necessary, business rules are checked in the service as usual

EXAMPLES WITH TESTS

- Adding a relationship also has impact on existing endpoints!
  - Retrieving a pony will also return information about its owner.
  - When you delete a pony, what do you do with the owner?
  - When you add a pony, can you add an owner in the same endpoint?
  - …
- The tests need to check more cases to cover the increase in complexity
- More examples with corresponding component tests can be found in the sample code

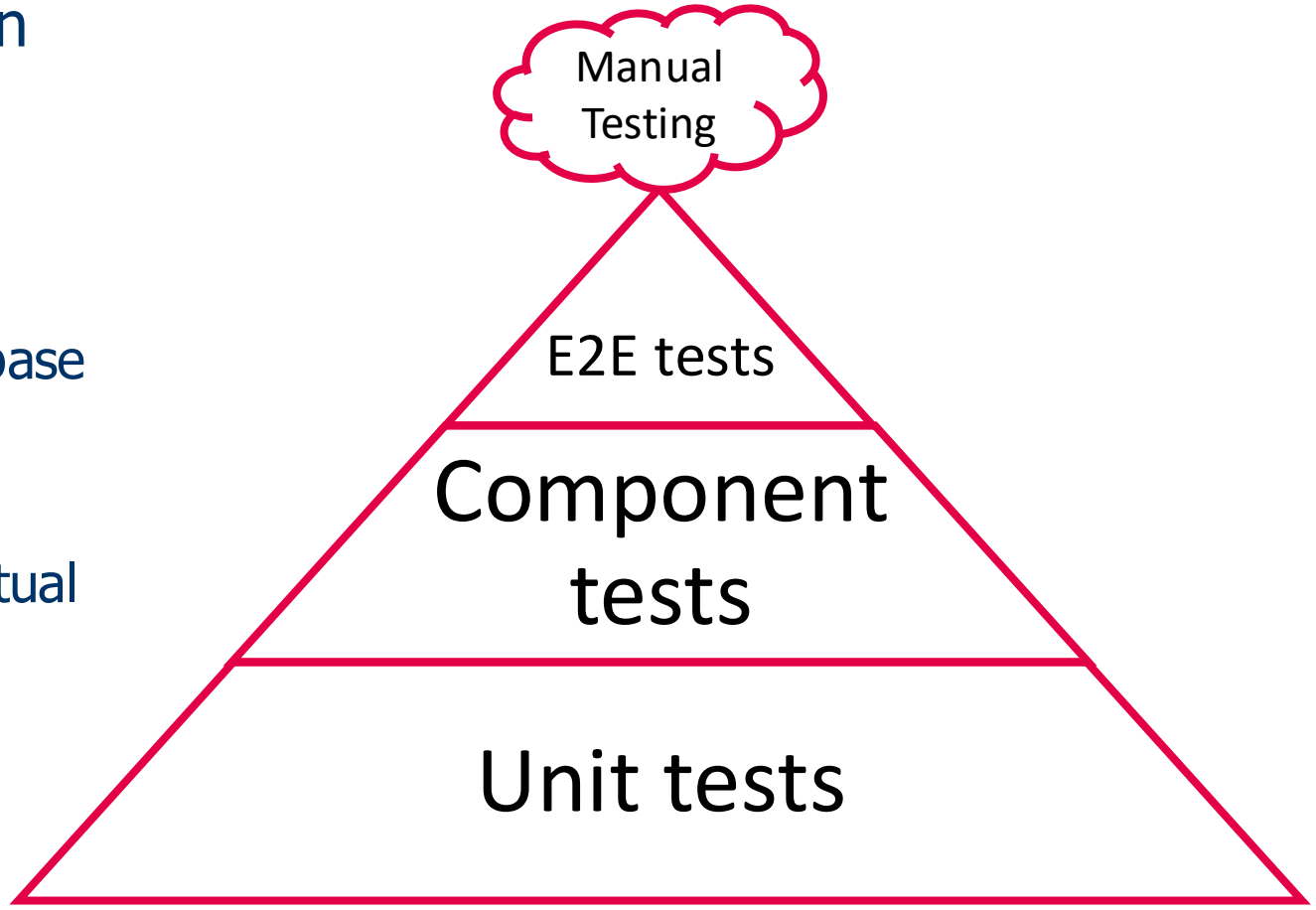
Testing with a database

- Unit Testing
- Component Testing

TESTING WITH A DATABASE

- There is impact on our tests when working with a database

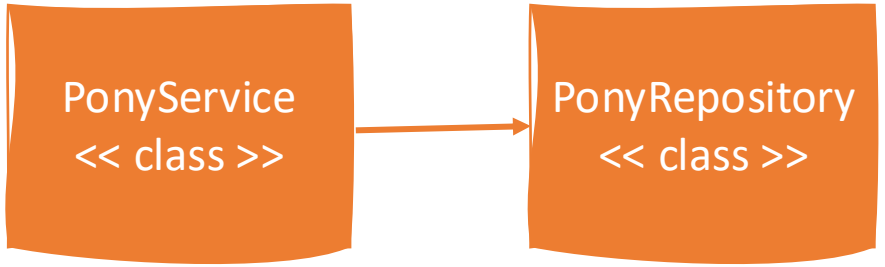
- Our component tests
  - Should start and maintain the database when they run
- Our service unit tests
  - Should run independent from an actual database, they should test code **in isolation**



TESTING WITH A DATABASE — UNIT — BEFORE JPA

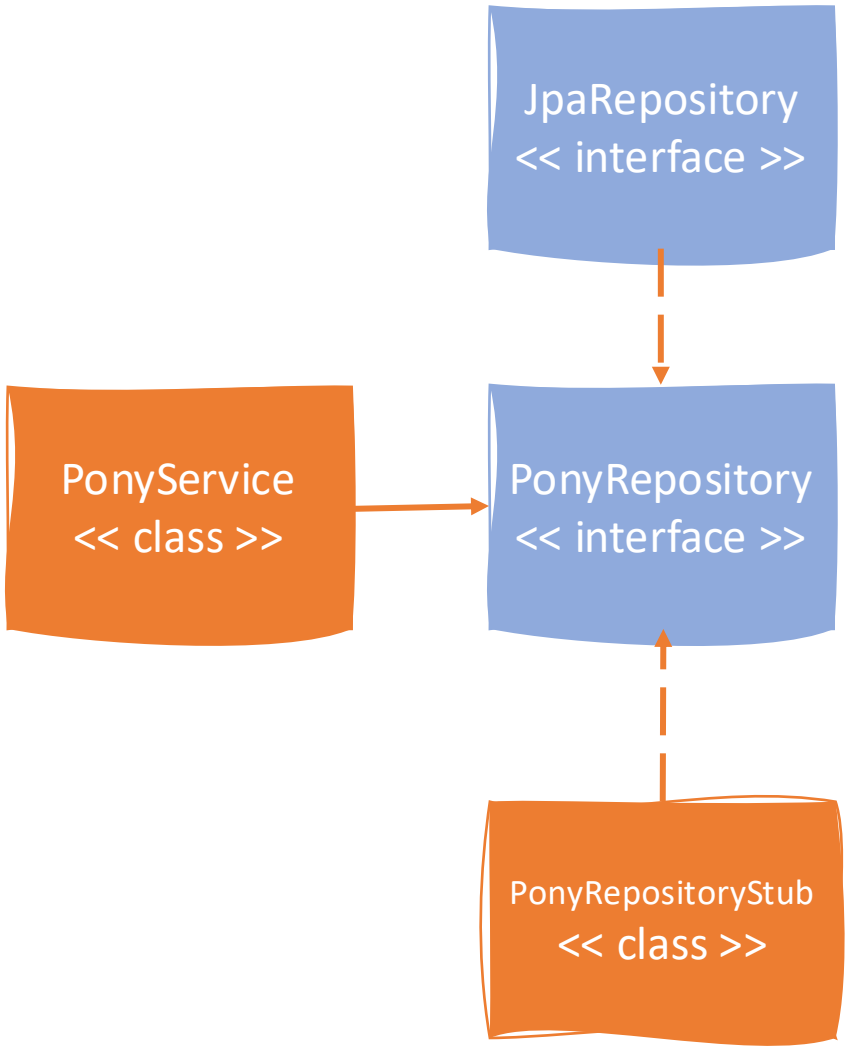
```
@Test
public void givenFindPonyByName_whenPonyExists_thenItIsReturned() {
    PonyRepository repository = new PonyRepository();
    PonyService service = new PonyService(repository);

    Pony foundPony = service.findPonyByName("Bella");
    Assertions.assertEquals("Bella", foundPony.getName());
}
```



- We test the service layer by instantiating a service and a repository in our unit test
  - The repository was a simple class with in-memory data

TESTING WITH A DATABASE — UNIT — WITH JPA



- The Spring Data Repository implementation will not be generated in unit tests since Spring is not used
- For unit testing we can keep instantiating the service with our "old" in-memory implementation, as long as it implements the Spring Data interface.
  - This pattern is called a **Stub**, the stub simulates real objects with the minimum methods needed for a test
  - The repository stub need to adapt to the new contract, you will need to override a lot of methods
    - Only override the methods your service uses!
    - Try to mimic the behaviour of the real repository in your stub implementation.

TESTING WITH A DATABASE — UNIT — WITH JPA

```
public interface PonyRepository extends JPARepository<Pony, Long> {


}
```

```
public class PonyRepositoryStub implements PonyRepository {

    private List<Pony> ponies;

    public PonyRepositoryStub() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    @Override
    public List<Pony> findAll() {
        return ponies;
    }

    @Override
    public <S extends Pony> S save(S pony) {
        ponies.add(pony);
        return pony;
    }

    // ...

}
```

- We can code an in-memory implementation for use in our unit tests
- This implementation will be very similar to our in-memory repository from before, it turns out that this was a very good testing implementation!

TESTING WITH A DATABASE — UNIT — WITH JPA

```
public interface PonyRepository extends JPARepository<Pony, Long> {


}
```

```
public class PonyRepositoryStub implements PonyRepository {

    private List<Pony> ponies;

    public PonyRepositoryStub() {
        ponies = new ArrayList<>();
        ponies.add(new Pony("Bella", 5));
        ponies.add(new Pony("Luna", 7));
        ponies.add(new Pony("Angel", 12));
    }

    @Override
    public List<Pony> findAll() {
        return ponies;
    }

    @Override
    public <S extends Pony> S save(S pony) {
        ponies.add(pony);
        return pony;
    }

    // ...

}
```

- The in-memory implementation should be stored under “src > test”
  - This file should not be used in the running application
  - It will be the default repository implementation when writing tests

TESTING WITH A DATABASE — UNIT — WITH JPA

```
public interface PonyRepository extends JPARepository<Pony, Long> {


}
```

```
public class PonyRepositoryStub implements PonyRepository {

  …

  @Override
   public <S extends Pony> List<S> saveAll(Iterable<S> entities) {
      return null;
   }

   @Override
   public Optional<Pony> findById(Long aLong) {
      return Optional.empty();
   }

@Override
   public Pony findByName(String name) {
      for (Pony pony : findAll()) {
         if (pony.getName().equals(name)) {
            return pony;
         }
      }
      return null;
   }
   …
}
```

- The repository stubs need to adapt to the new contract, you will need to override a lot of methods
  - Only override the methods your service uses! Try to mimic the behaviour of the real repository in your dummy implementation.

TESTING WITH A DATABASE — UNIT — WITH JPA

```
@Test
public void givenFindPonyByName_whenPonyExists_thenItIsReturned() {
    PonyRepository repository = new PonyRepositoryStub();
    PonyService service = new PonyService(repository);

    Pony foundPony = service.findPonyByName("Bella");
    Assertions.assertEquals("Bella", foundPony.getName());
}
```

- Using the in-memory implementation, we can test service methods in isolation
  - A dummy implementation for a dependency of the class under test is called a *stub*
- Every test should be changed so that it uses the in-memory implementation
  - This change should be small since the test implementation is very similar to the in-memory repository from before

TESTING WITH A DATABASE — COMPONENT — WITH JPA

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWebTestClient
@Sql("classpath:schema.sql")
public class HttpTest {

    private WebTestClient client;
    private PonyRepository repository;
    private DbInitializer dbInitializer;

    @Autowired
    public HttpTest(WebTestClient client, DbInitializer dbInitializer, PonyRepository ponyRepository) {
        // ...
    }

    @BeforeEach
    public void setup() {
        dbInitializer.initialize();
    }

    // ...
}
```

- Via @Sql the database is recreated in between each test
- Via @BeforeEach test-data is inserted before each test
- Your assertions should adapt to the changed implementations

SCHEMA.SQL

```
DROP TABLE IF EXISTS PONY;

CREATE TABLE PONY(
    ID BIGINT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(255),
    AGE  INT
);
```

- We drop tables before creating them to make sure the script can be ran multiple times

EXERCISES @CLASS

- exercise1
- exercise2
- exercise3
- exercise4
- exercise5



EXERCISES @HOME

- exercise6



