2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

JPA — Basics

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

CONTENTS

- Java — Database
- H2
- JPA
  - What is JPA
  - Entity
  - Spring Data Repository
  - Creating the DB Schema

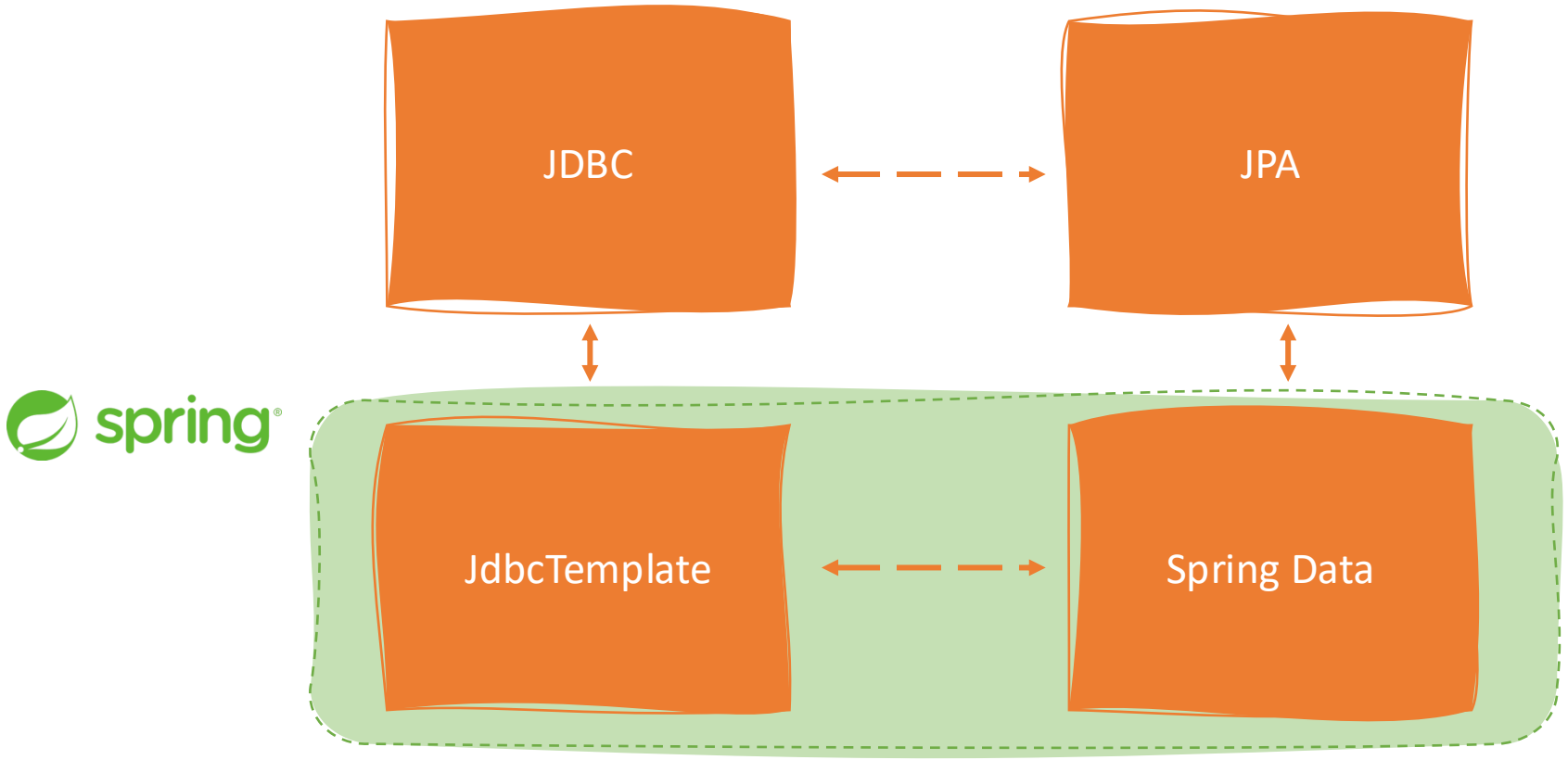
JAVA — DATABASE

- How can you connect with a database from Java?

- **J**ava **D**ata**B**ase **C**onnectivity (JDBC)
  - Low-level API: manage connection yourself and write queries manually
  - Spring offers JdbcTemplate to work with JDBC more easily

- **J**ava **P**ersistence **A**pi (JPA)
  - High-level API: developer performs operations on objects. The JPA implementation translates these operations to queries behind the scenes.
  - Spring offers Spring Data JPA to work with JPA more easily

JAVA — DATABASE



H2

H2

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

- H2 is an in-memory database
- Runs **in-process**, sharing an OS process with the Java back-end
  - Starts and stops together with the Java application
- Runs **in-memory**, losing all data when the application is stopped
- Configured automatically by Spring Boot when H2 and spring-data dependencies are included

H2 — CONFIGURATION

```
spring.datasource.url=jdbc:h2:mem:pony
spring.datasource.username=user
spring.datasource.password=password
```

→ Configure credentials for Database

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

→ Configure H2 UI

- Overwrite configuration by defining properties in ***application.properties***

H2 — CONFIGURATION – YAML

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:pony
    username: user
    password: password
  h2:
    console:
      enabled: true
      path: /h2
```

- Alternatively, you can define the configuration in *yaml* format in ***application.yaml***
- Both formats achieve the same, only the syntax is different
  - yaml nests properties hierarchically
  - In a normal properties file every key is independent

H2 — CONSOLE

- How can you view your H2 database?
  - Connect with any DB client using the credentials while your back-end is running
  - Use the built-in DB client

- Built-in DB Client: H2 Console
  - Web interface
  - Available on the URL specified in the configuration (*spring.h2.console.path*)
    - Default is /h2-console

H2 — DATABASE SCHEMA

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

- We will let the backend create the schema in H2 on startup
  - Creating your schema on startup can be done by defining a file “schema.sql” in “src > main > resources”
- Our database schema should match the configuration in the entity
  - AUTO_INCREMENT is necessary for the generation of the primary key by the database
  - Some words are reserved by the database and cannot be used as table or column name without escaping (eg, ‘user’)

H2 — DATABASE SCHEMA

Java	H2
java.lang.String	VARCHAR
boolean, java.lang.Boolean	BOOLEAN
int, java.lang.Integer	INT
long, java.lang.Long	BIGINT
double, java.long.Double	DOUBLE
java.time.LocalDate	DATE
java.time.Year	INT

- The table above shows the corresponding H2 type for some common Java classes and primitive types

H2 — DATABASE SCHEMA

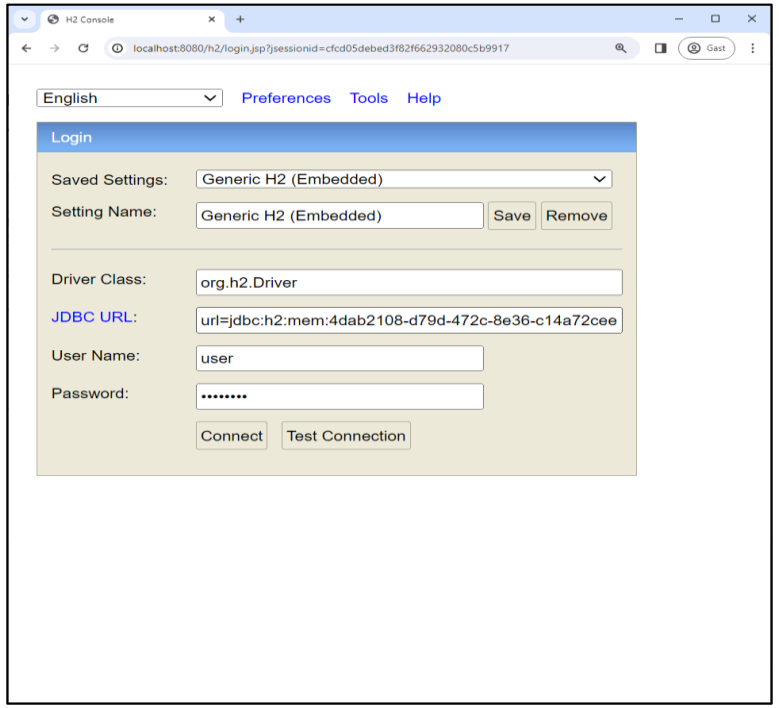
```
# ...

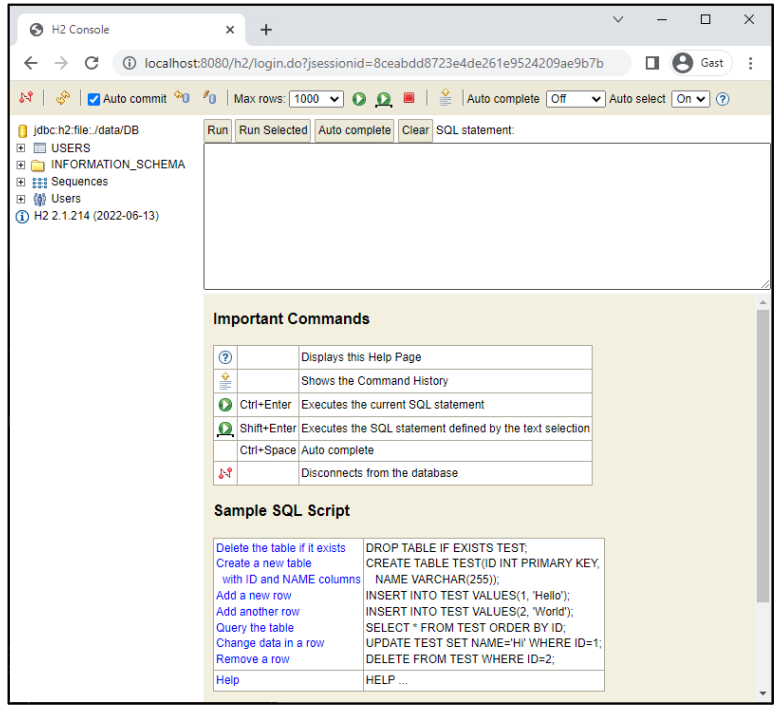
spring.sql.init.mode=always

# ...
```

- To execute the “schema.sql” file add the above property to application.properties

H2 — CONSOLE



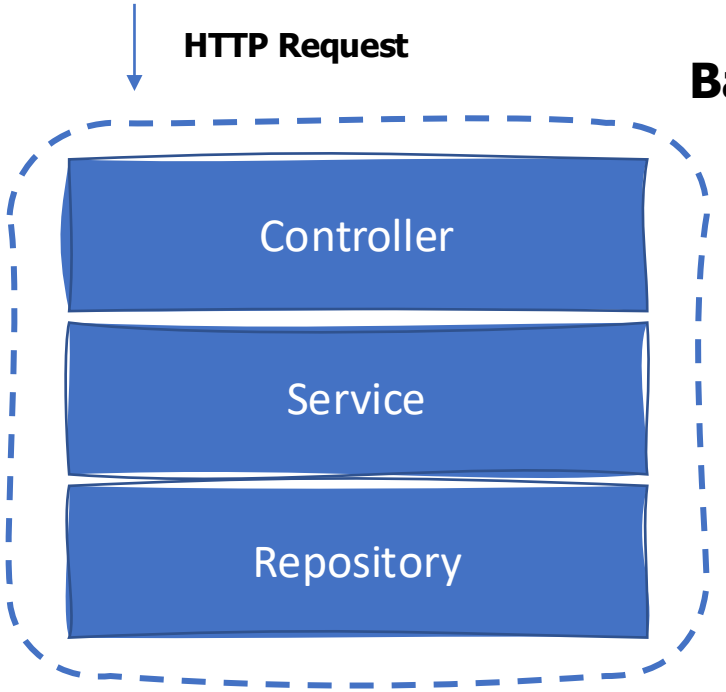


- The JDBC URL is printed in the application logs in terminal when your application starts
- If the property *spring.datasource.url* is filled in, the JDBC url is equal to the value of that property

JPA

- What is JPA
- Entity
- Spring Data Repository
- Creating the DB Schema

LAYERED ARCHITECTURE



Translates HTTP request to Java and translates Java back to a HTTP response

Guards application rules and gives commands to repositories to store, retrieve data

Communicates with the database and provides an API to use the database from within Java code

- The repository contains the interaction with the database
- Rewriting the repository to work with a database will not impact other classes since they rely on the repository for their database operations (the repository *encapsulates* the database access)

WHAT IS JPA — ORM

- The mapping of Java objects to database tables, and vice-versa is called **O**bject-**R**elational **M**apping (ORM)
  - An ORM library works as
    - a **bridge** between

      a **relational database (tables and records)**

      and

      a **Java application (classes and objects)**

- The purpose of an ORM library is to automatically sync operations done on Java objects to the database
  - For example, updating fields of an object leads to an automatic update query behind the scenes

WHAT IS JPA — HIBERNATE

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

- **J**ava **P**ersistence **A**PI (JPA) is a Java specification for managing relational data in Java applications
  - Specifies how an ORM for Java should work
  - Implemented by *Hibernate* (other implementations exist)
- Through JPA, we manipulate Java objects and the changes are pushed to the database transparently
  - No need to write SQL directly!

WHAT IS JPA — CONFIGURATION

```
# ...

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true

# ...
```

- The value of the “spring.jpa.hibernate.ddl-auto” property (none) says that we will create our schema manually and let JPA connect with it

WHAT IS JPA — HOW TO USE

To use JPA, we need to do two things

1. Our domain objects need to be modeled as *entities*
   - An entity is an object that can be tracked and managed by JPA
   - An entity is an regular object that has JPA specific annotations added to it
2. Our repositories need to change so that they work with JPA

ENTITY

```
@Entity
public class Pony {

    // ...
}
```

- Writing @Entity on a class marks it as a JPA entity
  - JPA will search for a table that corresponds to this class
  - JPA will translate changes to entities directly to the database
- A class marked as @Entity needs
  - To have a constructor without arguments (can be public or protected)
  - To have a field marked as primary key

ENTITY

```java
@Entity
@Table(name = "my_ponies")
public class Pony {

    // ...

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

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

    // ...

}
```

- By default, JPA will search for a table that has the same name as the class
- This can be overriden with the @Table annotation

ENTITY

```java
@Entity
@Table(name = "my_ponies")
public class Pony {

    @Id
    private long id;

    // ...

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

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

    // ...

}
```

- An entity needs to have a field marked as primary key via `@Id`
- We always opt for a technical key instead of a natural key

ENTITY

```
@Entity
@Table(name = "my_ponies")
public class Pony {

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

    // ...

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

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

    // ...

}
```

- @GeneratedValue indicates that the database should generate the primary key
  - Strategy IDENTITY uses a database *identity_column* to generate the primary key when the row is created

ENTITY

```
@Entity
@Table(name = "my_ponies")
public class Pony {

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

    private String name;

    private int age;

    // ...
}
```

- By default, JPA maps each field to a column with the same name

ENTITY

```
@Entity
@Table(name = "my_ponies")
public class Pony {

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

    @Column(name = "pony_name")
    private String name;

    @Transient
    private int age;

    // ...
}
```

- @Column changes the column the field is mapped to
- @Transient says that this field should not be mapped

REPOSITORY

```
public interface PonyRepository extends JpaRepository<Pony, Long> {}
```

- A JPA based repository is an interface that inherits from *JpaRepository*
  - Two things should be specified
    - The class of the entity that the repository will handle
    - The class of the primary key of the entity
- By inheriting from JpaRepository default methods for querying, adding, updating and deleting the specified entity are already present
- Spring will generate an implementation for this interface when our application starts

JPAREPOSITORY

- Most relevant repository methods
  - findAll: returns a list of all entities
  - save: saves a given entity and returns the saved instance
  - findById: searches for an entity by id and returns the result as an optional
  - delete: deletes a given entity
  - count: returns the amount of stored entities in the database
  - …

- For more methods see: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

PONYREPOSITORY

```
@Service
public class PonyService {

    // ...

    public List<Pony> allPonies() {
        return ponyRepository.findAll();
    }

    public Pony findPonyByName(String name) {
        return ponyRepository.findByName(name);
    }

    public Pony addPony(Pony pony) {
        return ponyRepository.save(pony);
    }

    public Pony updatePony(String name, Pony newInformation) {
        Pony pony = ponyRepository.findByName(name);
        pony.updateNameAndAge(newInformation.getName(), newInformation.getAge());
        return ponyRepository.save(pony);
    }

    public void removePony(String name) {
        Pony pony = findPonyByName(name);
        ponyRepository.delete(pony);
    }
}
```

- The service should change so that it uses the methods of the JpaRepository interface

REPOSITORY — ADDITIONAL QUERIES

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

    Pony findByName(String name);
}
```

- What if you need functionality that is not provided by the default JpaRepository?
- In this case you can add your own methods using pre-defined building blocks
  - As long as the method name is valid; Spring will provide an implementation at start-up

REPOSITORY — ADDITIONAL QUERIES

- Some possible keywords are
  - find
  - count
  - top
  - first
  - startingWith
  - containing
  - equals
  - isNot
  - greaterThan
  - after
  - order
  - …
- You can find a full list here

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

    Pony findByName(String name);

    List<Pony> findByAgeAfter(int age);
}
```

TEST DATA

```
@Component
public class DbInitializer {

    private PonyRepository ponyRepository;

    @Autowired
    public DbInitializer(PonyRepository ponyRepository) {
        this.ponyRepository = ponyRepository;
    }

    @PostConstruct
    public void initialize() {
        ponyRepository.save(new Pony("Bella", 5));
        ponyRepository.save(new Pony("Luna", 7));
        ponyRepository.save(new Pony("Angel", 12));
    }
}
```

- It is useful to have some data in the database when the application starts
- We can achieve this via the construct above
  - @Component makes sure that the class is found and initialized by Spring
  - @PostConstruct executes the method once after the class is initialized

EXERCISES @CLASS

- exercise1
- exercise2
- exercise3
- exercise4



EXERCISES @HOME

- exercise5
- exercise6
- exercise7



