Java Programming 3

Week 7 - JDBC - JdbcTemplate - Profiles

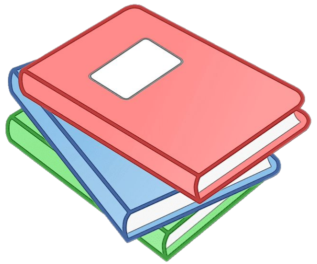
KdG

Agenda this week

JDBC
Implementing the repository
JdbcTemplate
Spring profiles

Tutorials

- JDBC: http://tutorials.jenkov.com/jdbc/index.html
- JDBC and Spring: https://www.baeldung.com/spring-jdbc-jdbctemplate
- Spring JDBC Reference: https://docs.spring.io/spring-framework/docs/1.2.2/reference/jdbc.html
- Spring Profiles: https://www.baeldung.com/spring-profiles
- Spring JDBC Guide: https://spring.io/guides/gs/relational-data-access/

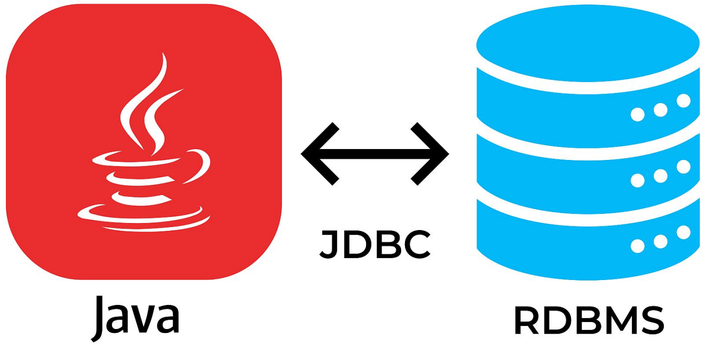


Agenda this week

JDBC
Implementing the repository
JdbcTemplate
Spring profiles

JDBC

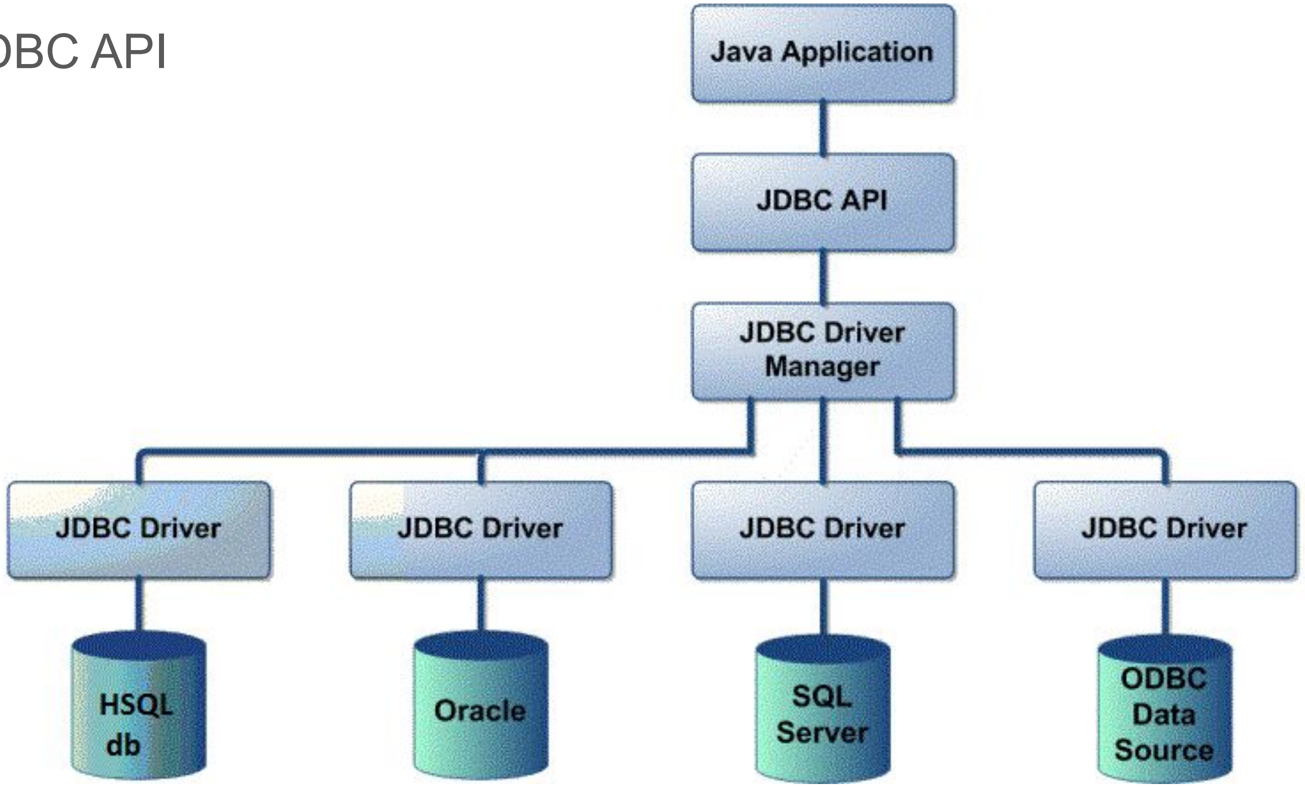
- API that allows a Java program to communicate with a relational database
- Independent of the vendor
- Access using SQL Queries
- Part of standard Java (java.sql.*)



KdG

JDBC Driver

- Implements the JDBC API
- Vendor specific

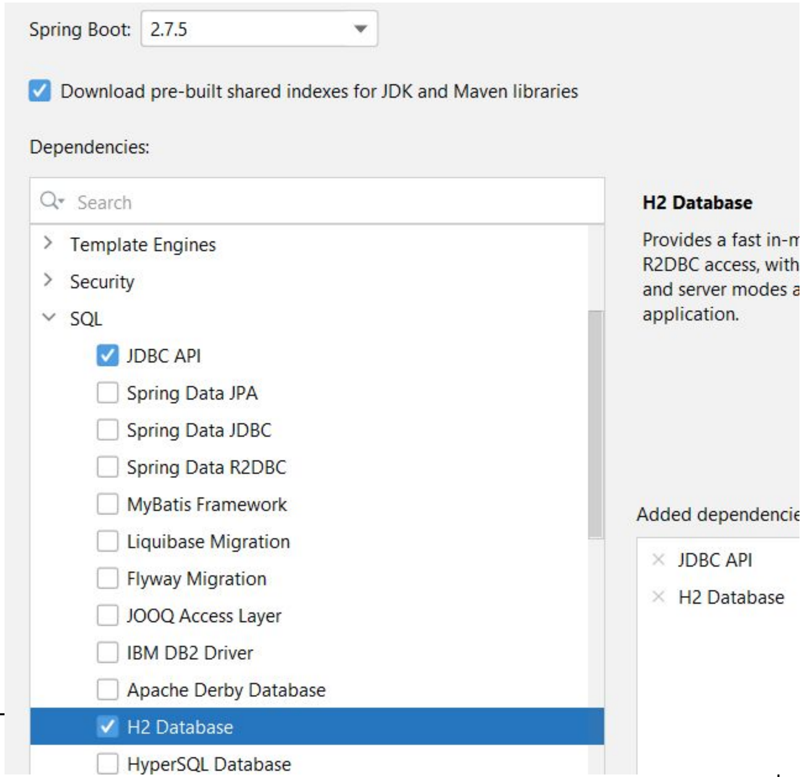


H2Database

- We will use H2 DB in the examples
  - Lightweight, small, fast, in memory
  - Perfect for development and testing
  - Not suitable for production: you will use PostgreSQL instead
- We will have to add the driver to the gradle dependency!

Create new Spring project

- Add 2 dependencies
  - JDBC API
  - H2 Database


Spring Boot: 2.7.5

[✓] Download pre-built shared indexes for JDK and Maven libraries

Dependencies:

- [✓] JDBC API
- [ ] Spring Data JPA
- [ ] Spring Data JDBC
- [ ] Spring Data R2DBC
- [ ] MyBatis Framework
- [ ] Liquibase Migration
- [ ] Flyway Migration
- [ ] JOOQ Access Layer
- [ ] IBM DB2 Driver
- [ ] Apache Derby Database
- [✓] H2 Database
- [ ] HyperSQL Database

H2 Database

Provides a fast in-m
R2DBC access, with
and server modes a
application.

Added dependencie

- JDBC API
- H2 Database

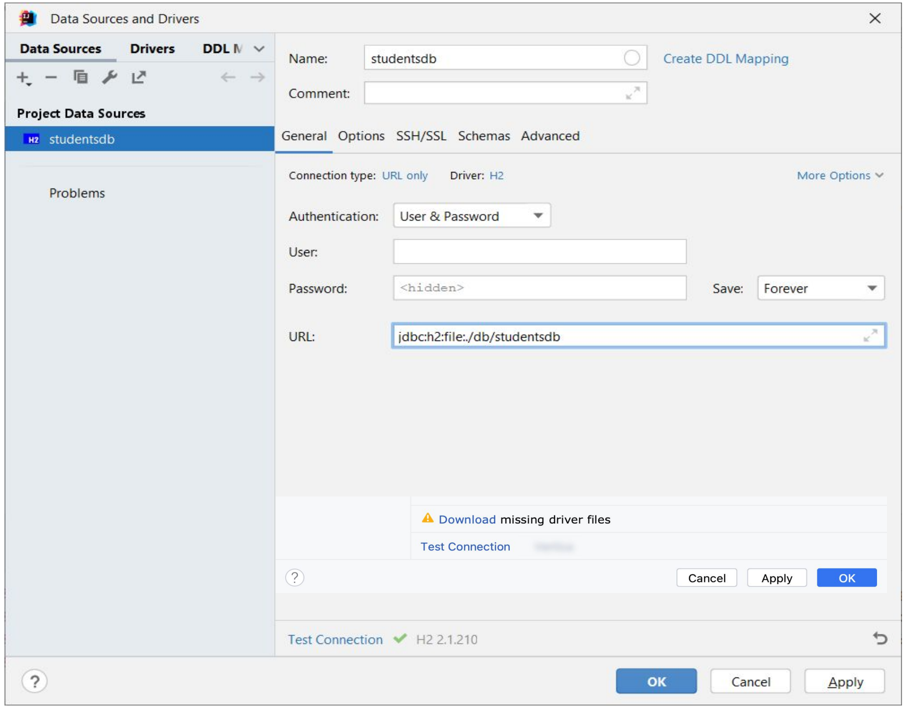
```
dependencies {
   //…
   implementation ("org.springframework.boot:spring-boot-starter-jdbc")
   runtimeOnly ("com.h2database:h2")
}
```

Create DB first: IntelliJ Database View

- Connection Type: URL Only
- URL: **jdbc:h2:file:./db/studentsdb**

  → Creates DB in ./db folder

  - If intellij proposes to download missing driver files, do so.



studentsdb

Name: studentsdb

Comment:

Connection type: URL only Driver: H2

Authentication: User & Password

User:

Password: <hidden> Save: Forever

URL: jdbc:h2:file:./db/studentsdb

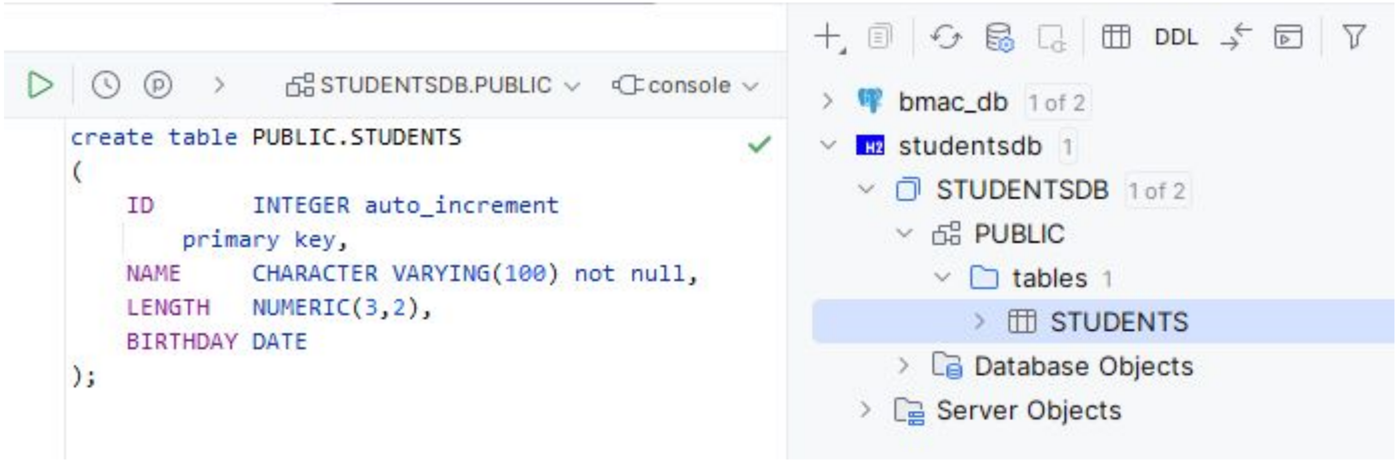
⚠ Download missing driver files

Test Connection ✓ H2 2.1.210

KdG

Create table STUDENT

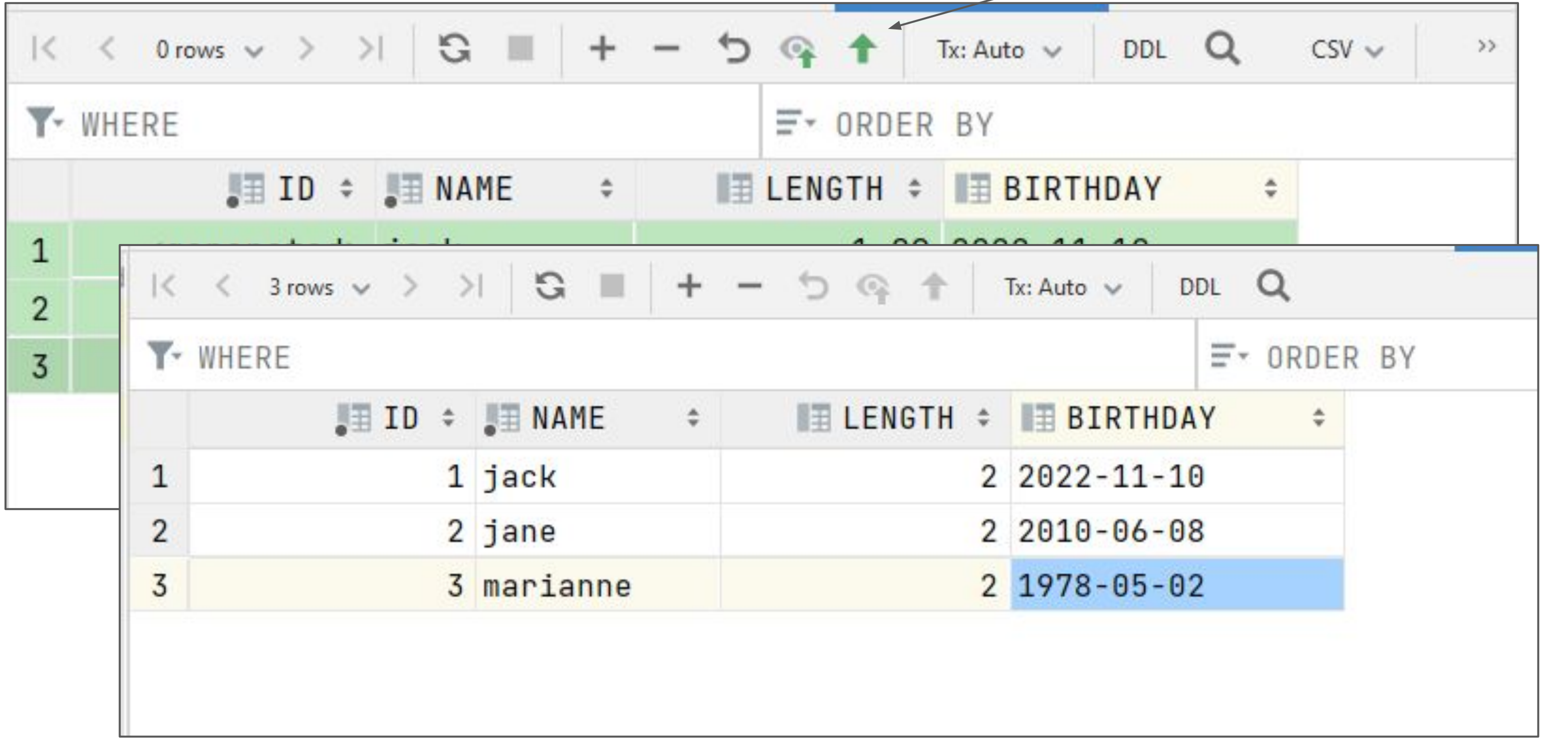
- Use the DB View
- Create columns ID (PK, auto_increment), NAME, LENGTH, BIRTHDAY



```
create table PUBLIC.STUDENTS
(
    ID       INTEGER auto_increment
        primary key,
    NAME     CHARACTER VARYING(100) not null,
    LENGTH   NUMERIC(3,2),
    BIRTHDAY DATE
);
```

Add some records…

Don't forget to push Submit…



	ID	NAME	LENGTH	BIRTHDAY
1	1	jack	2	2022-11-10
2	2	jane	2	2010-06-08
3	3	marianne	2	1978-05-02

KdG

JDBC: basic steps

1. Add JDBC Driver
2. Create a `Connection` to the database
3. Create a `Statement` (or `PreparedStatement`)
4. Perform query
5. Process the result (`ResultSet`)
6. Close `ResultSet` - `Statement` - `Connection`

Slides code

Create a CommandLineRunner

```
When a bean implementing
CommandLineRunner is loaded,
It's run method is executed
```

```java
@SpringBootApplication
public class JdbcApplication implements CommandLineRunner{

 public static void main(String[] args) {
  SpringApplication.run(JdbcApplication.class, args);
 }

 @Override
 public void run(String... args) throws Exception {
  Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
  Statement statement = connection.createStatement();
  ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS");
  while (resultSet.next()) {
   System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
  }
  resultSet.close();
  statement.close();
  connection.close();
 }
}
```

```
1 jack
2 jane
3 marianne
```

```
A file can only be opened by one process, deactivate the
database in your IntelliJ tool window before running this.
(This problem does not occur with network databases)
```

Connection

- You create a connection to the Database via **`DriverManager.`*`getConnection(..)`***
  - You can provide username and password as extra parameters

```
public void run(String... args) throws Exception {
   Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
   Statement statement = connection.createStatement();
   ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS");
   while (resultSet.next()) {
       System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
   }
   resultSet.close();
   statement.close();
   connection.close();
}
```

Closing the connection

- You should close the connection if it is no longer used!
  - Typically the DB can only serve a limited number of connections at the same time…
- Use try (*with resources)* to be sure → will automatically close connection
  - You can open multiple resources in one try (with; resources)

```
public void run(String... args) throws Exception {
 try (Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
      Statement statement = connection.createStatement();
 ) {
  try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
   while (resultSet.next()) {
    System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
   }
  }
 }
}
```

Statement

- You create a Statement to perform a query on the database
- Statements should also be closed (eg using try with resources)!

```
public void run(String... args) throws Exception {
 try (Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
      Statement statement = connection.createStatement();
 ) {
  try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
   while (resultSet.next()) {
    System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
   }
  }
 }
}
```

ResultSet

- The executeQuery method returns a ResultSet. Via the ResultSet you can retrieve the rows returned by the query…
- ResultSet should also be closed after use…

```java
public void run(String... args) throws Exception {
 try (Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
      Statement statement = connection.createStatement();
 ) {
  try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
   while (resultSet.next()) {
    System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
   }
  }
 }
}
```

ResultSet

- The next() method moves the cursor to the next row
- It returns true if there is a new row, false otherwise
- Perfect in combination with a while loop…

```
public void run(String... args) throws Exception {
 try (Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb");
      Statement statement = connection.createStatement();
 ) {
  try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
   while (resultSet.next()) {
    System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
   }
  }
 }
}
```

ResultSet

- Once you are on a row, you can retrieve values of different columns using the column name or the column index (starts from 1)
    - Using name is preferred over index…

```
public void run(String... args) throws Exception {
   try (Connection connection = DriverManager.getConnection("jdbc:h2:file:./db/studentsdb")){
       try (Statement statement = connection.createStatement()) {
           try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
              while (resultSet.next()) {
                  System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
                  System.out.println(resultSet.getString(1) + " " + resultSet.getString(2));
              }
           }
       }
   }
}
```

ResultSet get…: what SQL types?

- getString(...): for character based types (char, varchar, varchar2, …)
  - Works on any type (does a toString)
- getInt(...), getLong(...): integer types (smallint, int, …)
- getFloat(...), getDouble(...): decimal types (decimal, real, double precision, …)
- getBoolean(...): boolean types
- getDate(...): for dates
  - Returns java.sql.Date object → convert it to LocalDate for use in Java…
- …

```
Date birthDay = resultSet.getDate("BIRTHDAY");
LocalDate localDate = birthDay.toLocalDate();
```

Insert or delete data?

```
try (Connection connection = DriverManagergetConnection("jdbc:h2:file:./db/studentsdb")) {
   try (Statement statement = connection.createStatement()) {
       int result = statement.executeUpdate("INSERT INTO STUDENTS(NAME, LENGTH, BIRTHDAY) " +
                                              "VALUES('AN',1.65,'1967-3-12')");
       Sy
   }
}
```

```
try (Connection connection = DriverManagergetConnection("jdbc:h2:file:./db/studentsdb")) {
   try (Statement statement = connection.createStatement()) {
       int result = statement.executeUpdate("DELETE FROM STUDENTS WHERE NAME = 'AN'");
       System.out.println(result + "row(s) updated");
   }
}
```

- Use executeUpdate method to do updates (inserts, deletes) to the database
- Returns number of rows affected (1 in the above cases)

Exercise:

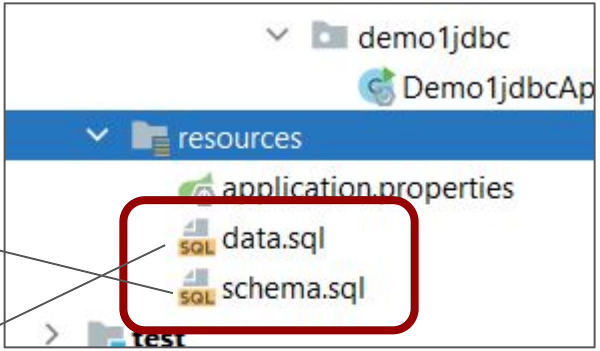
- Create new Spring application
  - Add JDBC and H2 dependencies
- Use the Database view in IntelliJ to:
  - Create table STUDENTS: a user has an id, name, length and birthday
  - Add some data
- Write a small application (commandlinerunner) that shows all STUDENTS sorted by birthday
- Perform some updates and deletes on the data
- What happens if you keep the connection open on the IntelliJ Database tool?

Creating the database tables and loading initial data?

- You can do this in database tool, or using Java code
- Or: you can add a `schema.sql` and `data.sql` to the resources folder
- Spring will run the .sql files once when the application starts...

```sql
DROP TABLE IF EXISTS PERSONS; -- not needed for in memory DB

CREATE TABLE PERSONS(
   ID INTEGER AUTO_INCREMENT PRIMARY KEY,
   NAME CHARACTER VARYING(100) NOT NULL,
   FIRSTNAME CHARACTER VARYING(50)  NOT NULL,
   REMARK CHARACTER VARYING(256)
);
```

```sql
INSERT INTO PERSONS(NAME, FIRSTNAME,REMARK)
VALUES ('JONES', 'JACK','of all trades'),
      ('POTTER', 'JACK','Lilly''s dad'),
      ('POTTER', 'MIA','Lilly''s mum'),
      ('REED', 'JACK','union');
```



Configure the datasource in Spring...

- Spring needs to know where to connect: you can add this info in the `application.properties`

```
spring.datasource.url=jdbc:h2:mem:personsdb
spring.datasource.username=sa
spring.datasource.password=
spring.sql.init.mode=always
```

We use a H2 memory database now: it will only exist in memory. Ideal for developing and testing…

If not using a memory database Spring does not run the schema.sql and data.sql, you need to add this last line...

[Slides code: persons_datasource](#) DataSourceRunner

Exercise:

- Create a small Spring application (no Spring MVC, just a commandlinerunner) that loads 10 students (name, length, birthday) into a H2 memory database at startup:
  - Configure in the application.properties
  - use schema.sql and data.sql to create the table and load the data
- The application asks for the name via the console
- The application shows all records that match the name
- Use this query: `"SELECT * FROM STUDENTS WHERE NAME = '" + name + "'"`

Exercise:

- Create a small Spring application (no Spring MVC, just a commandlinerunner) that loads 10 students (name, length, birthday) into a H2 memory database at startup:
  - Configure in the application.properties
  - use schema.sql and data.sql to create the table and load the data
- The application asks for the name via the console
- The application shows all records that match the name
- Use this query: `"SELECT * FROM STUDENTS WHERE NAME = '" + name + "'"`

→ Try using `JACK' OR '1'='1` as name, what happens?

→ Example of SQL Injection!

PreparedStatement

The values are inserted at the ? in the PreparedStatement.
The prepared statement can execute the SAME statement multiple times with DIFFERENT parameters

- You can *prepare* an SQL statement:
  - It will be precompiled → you can prepare it beforehand
  - You can change the parameters
  - It is a good protection against SQL injection...

```
try (PreparedStatement statement = connection.prepareStatement("INSERT INTO STUDENTS(NAME, LENGTH, BIRTHDAY)
VALUES(?,?,?)")) {
   for (int i=0;i<10;i++) {
       statement.setString(1, "An" + i);
       statement.setDouble(2, 1.78);
       statement.setDate(3, Date.valueOf(LocalDate.of(1987, 1 + i, 23)));
       int result = statement.executeUpdate();
       System.out.println(result);
   }
}
```

Slides code: preparedStatement

Exercise:

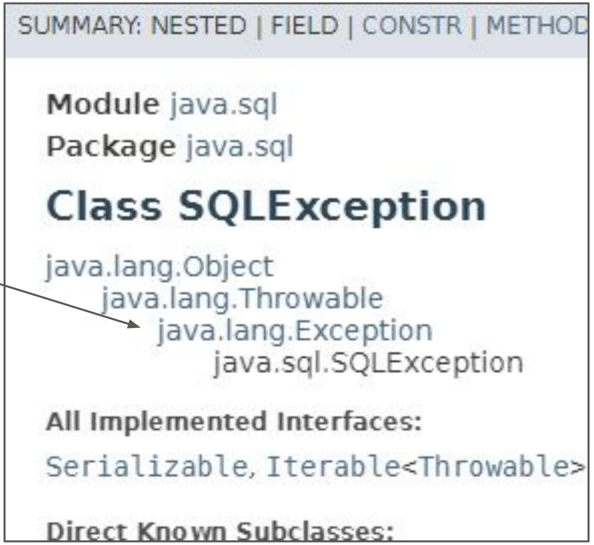
- Use a PreparedStatement in the previous exercise: does it solve the SQL Injection problem?

KdG

SQLExceptions

- If something goes wrong, JDBC throws an SQLException
  - Example: DB offline, wrong table, column does not exist, …
  - It is a *checked* exception: you have to write exception handling code for it…
  - Not very fine-grained: you get an SQLException for all kinds of problems

```
Extract from the Java 17 API Documentation:
SQLException inherits from Exception so it
is a checked exception…
```



SUMMARY: NESTED | FIELD | CONSTR | METHOD

Module java.sql

Package java.sql

Class SQLException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.sql.SQLException
```

All Implemented Interfaces:

`Serializable`, `Iterable<Throwable>`

Direct Known Subclasses:

Transactions in JDBC

- Sometimes you need to perform more than one query together.
- Example: transfer money to other account
  - Query one: remove money from the first account
  - Query two: add money to the second account
    - If for some reason the second query does not succeed, the first one should roll back!
- In a relational database you use transactions for this
- You can also perform transactions using JDBC

Transactions in JDBC

```
try (Connection connection = DriverManager.getConnection("jdbc:h2:mem:personsdb","sa","")) {
 connection.setAutoCommit(false);
 try (Statement statement = connection.createStatement()) {
  try {
   statement.executeUpdate("""
    INSERT INTO PERSONS(NAME, FIRSTNAME, REMARK)
    VALUES('Truus','Trampoline','Lastly through a hogshead of real fire!')
    """);
   statement.executeUpdate("DELETE FROM PERSONS WHERE FIRSTNAME LIKE '%ER'");
   if (new Random().nextBoolean()) throw new SQLException("Problem!");
   System.out.println("No problem, inserting and deleting...");
   connection.commit();
  } catch (SQLException e) {
   System.out.println("Problem, rolling back delete and insert!");
   connection.rollback();
  }
}
```

We mimic an SQLException here, just for the demo. If the exception is thrown, the transaction will rollback and BOTH the insert and delete will not happen!

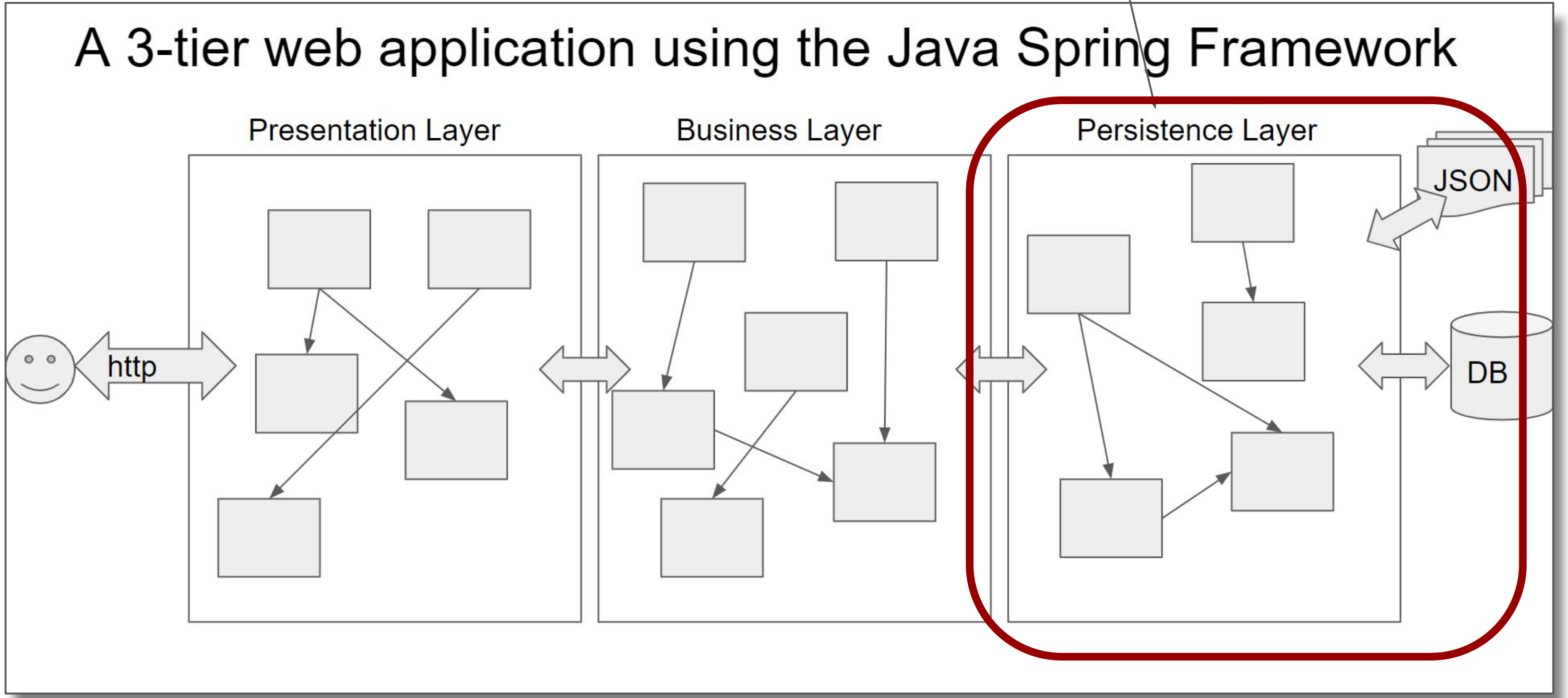
Slides code: persons_datasource TransactionRunner

Agenda this week

JDBC
Implementing the repository
JdbcTemplate
Spring profiles

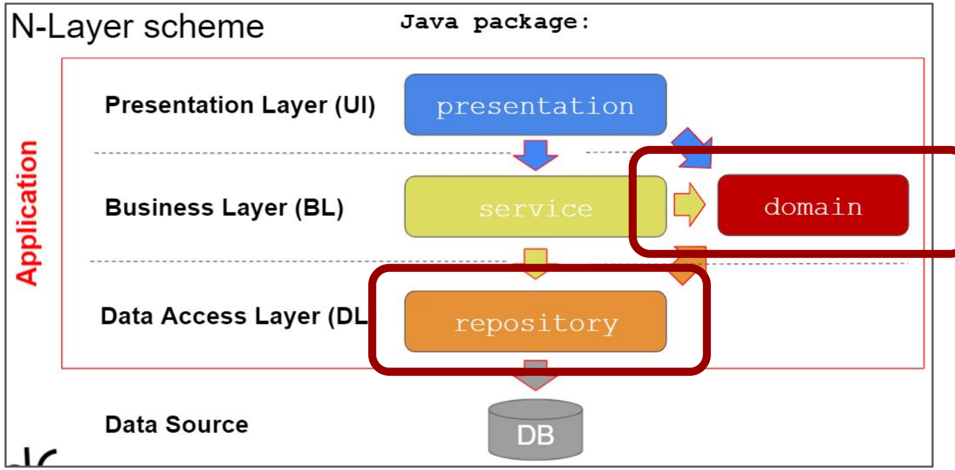
The persistence layer

We will use JDBC in the Persistence Layer.
We will create Repository classes for it...



Remember: Domain and Repository

- Domain classes map to database tables (“entities”)
- They have an ID to contain the PK of the record
- We create Repository classes for each entity.
- Annotate with @Repository (= a Spring Component)
- The Repository classes will contain the JDBC code
- Repository has methods to
  - Query entities (findBy…)
  - Create entities (create…)
  - Update entities (update…)
  - Delete entities (delete…)



KdG

Exercise: create the StudentJdbcRepository(1/6)

- Create the domain class Student (id, name, length, birthday)
- Configure your project with a database with some Students
  - Use application.properties - schema.sql - data.sql
- Create the StudentRepository interface:

```
public interface StudentRepository {
   List<Student> findAll();
   Student createStudent(Student student);
   void updateStudent(Student student);
   void deleteStudent(int id);
}
```

KdG

Exercise: create the StudentJdbcRepository(2/6)

- Create the StudentJdbcRepository(implements StudentRepository)
- Implement the findAll() method
  - You need the database URL, username and password to create the connection → values from the application.properties files can be retrieved as follows:

```java
public StudentJdbcRepository(@Value("${spring.datasource.url}") String dbURL,
 @Value("${spring.datasource.username}") String user,
 @Value("${spring.datasource.password}:''") String password) {
 this.dbURL = dbURL;
 this.user = user;
 this.password = password;
}
```

```
:'' default value
(if property is not found)
is an empty string
```

Exercise: create the StudentJdbcRepository(3/6)

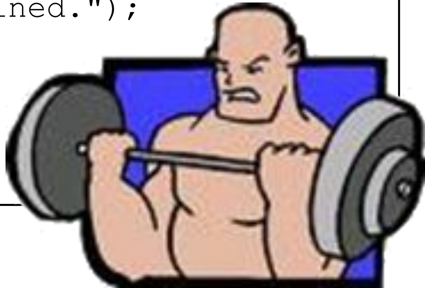
- Create the presentation class StudentMenu
  - The StudentRepository is injected (*we will skip the service layer for this exercise…*)
  - We will show a small console menu like this:
- Test findAll!

```
Welcome to the Student Management System
========================================
1) List students
2) Add student
3) Update student
4) Delete student
Make a choice:
```

Exercise: create the StudentJdbcRepository(4/6)

- Implement the createStudent() method
- You need the ID of the newly created Student. How?

```
try (PreparedStatement statement
  = connection.prepareStatement("INSERT INTO STUDENTS(..)"= Statement.RETURN_GENERATED_KEYS)){
   //…
   int result = statement.executeUpdate();
   if (result != 0) {
       try (ResultSet generatedKeys = statement.getGeneratedKeys()) {
           if (generatedKeys.next()) {
               createdStudent.setId(generatedKeys.getInt(1));
           }
           else {
               throw new SQLException("Creating student failed, no ID obtained.");
           }
       }
   }
}
```



Exercise: create the StudentJdbcRepository(5/6)

- Implement the updateStudent() and deleteStudent() methods
- Example run:

```
Welcome to the Student Management System
========================================
1) List students
2) Add student
3) Update student
4) Delete student
Make a choice:3
Student{id=1, name='jane', length=1.87, birthday=2010-06-08}
Student{id=2, name='marianne', length=1.23, birthday=1978-05-02}
Student{id=3, name='Truus', length=2.0, birthday=1954-03-12}
Student{id=4, name='An', length=1.67, birthday=1987-02-23}
Which student (id)?4
new name:Jef
new length:1.87
new birthday (mm-dd-yyyy):02-23-1987
```

Exercise: create the StudentJdbcRepository(6/6)

- What about those SQLExceptions?
- We don’t want the repository to throw SQLExceptions to the higher layers
    - Create an unchecked exception class DatabaseException
    - Wrap the SQLException in it and throw to higher layers
    - Catch in the presentation layer and show a message…

JdbcTemplate: removes the boilerplate code

```
@Component
public class JdbcTemplateRunner implements CommandLineRunner {
 private JdbcTemplate jdbcTemplate;

 public JdbcTemplateRunner(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
 }

 @Override
 public void run(String... args) throws Exception {
  //Query the database
  jdbcTemplate.query("SELECT * FROM PERSONS",
   (RowCallbackHandler) rs ->
    System.out.println(rs.getString("ID") + " " + rs.getString("NAME")));
 }
}
```

Spring provides JdbcTemplate.
It handles connections, statement, closing, exceptions etc behind the scenes,
taking parameters (url, user, password) from application.properties

RowCallbackHandler is executed for each resultset row

Old JDBC Code:

```
public void run(String... args) throws Exception {
   try (Connection connection = DriverManager.getConnection("jdbc:h2:mem:peronsdb","sa","")){
       try (Statement statement = connection.createStatement()) {
           try (ResultSet resultSet = statement.executeQuery("SELECT * FROM STUDENTS")) {
               while (resultSet.next()) {
                   System.out.println(resultSet.getString("ID") + " " + resultSet.getString("NAME"));
               }
           }
```

Slides code: persons_datasource JdbcTemplateRunner

Other JdbcTemplate methods

- `query`: execute a query. A lot of overloaded methods here, see docs.
- `queryForObject`: executes a Query, returns a domain object.
- `update`: perform an update
- `batchUpdate`: perform a batch of updates in one go

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

Example: convert ResultSet to List of domain objects

```
public List<Person> findByName(String name){
 return jdbcTemplate.query("SELECT * FROM PERSONS WHERE NAME = ?",
   (rs, rowNum) -> new Person(rs.getInt("id"),
     rs.getString("name"),
     rs.getString("firstname"),
     rs.getString("remark")),
   name);
}
```

The second parameter is a RowMapper implementation (here a lambda). It provides code to convert the ResultSet into a domain Object. JdbcTemplate will apply it to all records in the ResultSet and returns a List!

Subsequent parameters are for the PreparedStatement...

The `RowMapper`

- You can pass a `RowMapper` to the `query` or `queryForObject` methods. It maps a row in the `ResultSet` to the Domain object. It can be done in different ways:
  - Implement a `RowMapper` (using class, a lambda expression, referencing a method from the repository…)
  - Using a `BeanPropertyRowMapper` to generate the `RowMapper` based on the Domain object

`The domain object needs a default constructor for this to work…!`

`Slides code: persons_datasource` `RowMapperRunner`

Example: queryForObject

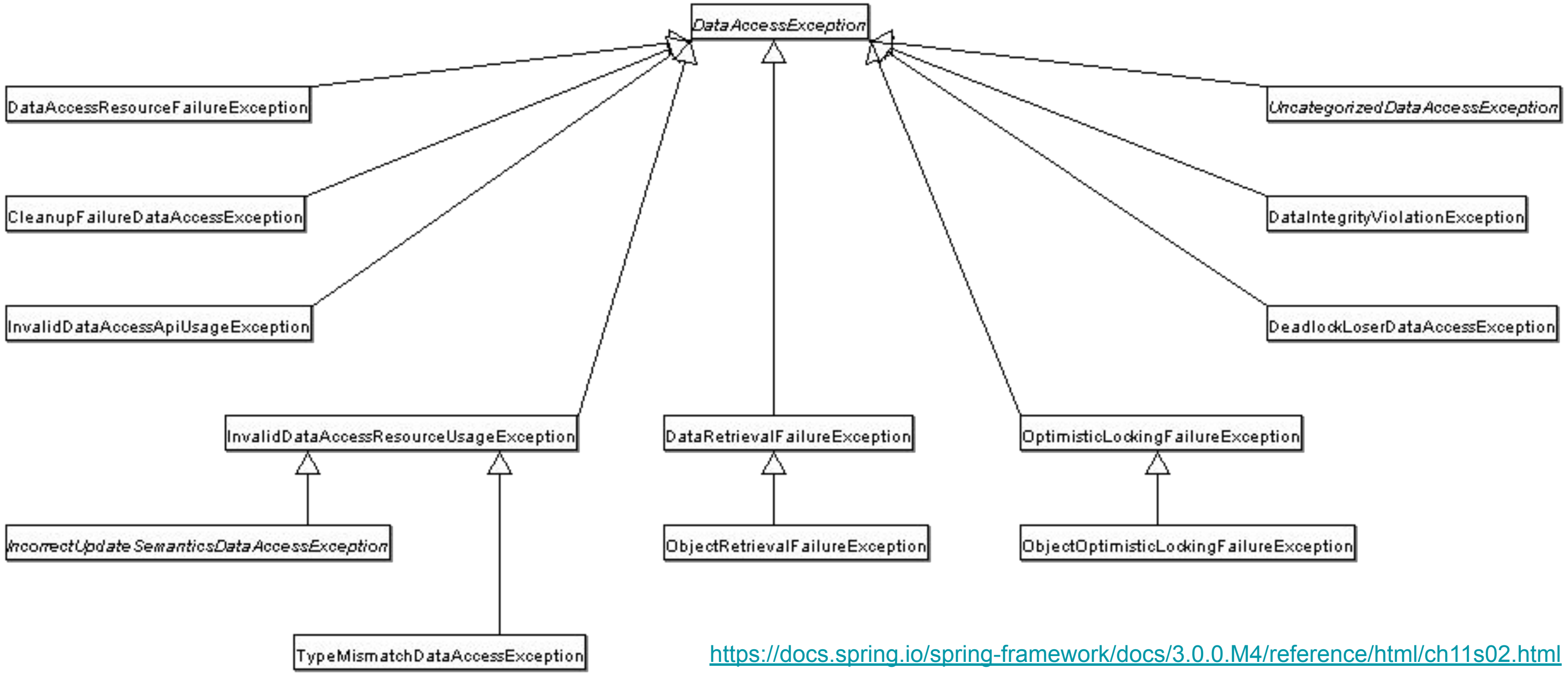
```
public Person findById(int id) {
 return jdbcTemplate.queryForObject("SELECT * FROM PERSONS WHERE ID = ?",
   this::mapRow,
   id);
}

private Person mapRow(ResultSet rs, int i) throws SQLException {
 return new Person(rs.getInt("id"),
   rs.getString("name"),
   rs.getString("firstname"),
   rs.getString("remark"));
}
```

The mapRow method reference does the same as the lambda in the previous example, but can be reused in other queries returning Person(s)

And the SQLException?

Spring provides more fine-grained exceptions so you can write better exception handling.
The exceptions are unchecked: you don't need to handle them!

- It is replaced by a hierarchy of `DataAccessExceptions`


DataAccessException
DataAccessResourceFailureException
CleanupFailureDataAccessException
InvalidDataAccessApiUsageException
InvalidDataAccessResourceUsageException
DataRetrievalFailureException
OptimisticLockingFailureException
UncategorizedDataAccessException
DataIntegrityViolationException
DeadlockLoserDataAccessException
IncorrectUpdateSemanticsDataAccessException
TypeMismatchDataAccessException
ObjectRetrievalFailureException
ObjectOptimisticLockingFailureException

https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch11s02.html

KdG

Exercise: walk through this spring.io guide

- This small tutorial walks you through using JdbcTemplate to access a relational database: https://spring.io/guides/gs/relational-data-access/

KdG

Saving entities to the database

- How is the retrieving of the ID implemented?

```
public Person save(Person person) {
 jdbcTemplate.update("INSERT INTO PERSONS (NAME, FIRSTNAME, REMARK) VALUES (?, ?, ?)",
  person.getName(),
  person.getFirstName(),
  person.getRemark());
 // set Person id??
 return person;
}
```

Using a SimpleJdbcInsert object

```
private SimpleJdbcInsert personInserter;

public PersonJdbcTemplateRepository(JdbcTemplate jdbcTemplate) {
 this.jdbcTemplate = jdbcTemplate;
 personInserter = new SimpleJdbcInsert(jdbcTemplate).withTableName("PERSONS")
  .usingGeneratedKeyColumns("ID");
}
//...
public Person save(Person person) {
 int personId = personInserter.executeAndReturnKey(Map.of(
  "NAME", person.getName(),
  "FIRSTNAME", person.getFirstName(),
  "REMARK", person.getRemark())
 ).intValue();
 person.setId(personId);
 return person;
}
```

Exercise:

- Create a second implementation of the StudentRepository, this time using JdbcTemplate!
  - Inject the JdbcTemplate
  - Use a private mapRow method to map a ResultSet entry to a Student
  - Use SimpleJdbcInsert to retrieve the id when creating a new Student
  - Think about the exception handling…
- Test this implementation…

Agenda this week

JDBC
Implementing the repository
JdbcTemplate
Spring profiles

Spring profiles

- https://www.baeldung.com/spring-profiles
- If you like to use another DB technology for development versus testing or production?
  - Spring provides the possibility to define *profiles*
    - You can select the active profile in `application.properties`:
      - `spring.profiles.active=prod`
    - You can create separate `application-dev.properties`, `application-prod.properties, …`
  - In your code: using the `@Profile` annotation you can map beans to different profiles

Development
Testing/QA
Production

KdG

Example: H2 for development - PostgreSQL for production

- Make sure `org.postgresql:postgresql` dependency is in build.gradle.kts
- Put PostgreSql attributes into application-prod.properties

```
spring.datasource.url=jdbc:postgresql:personsdb
spring.datasource.username=postgres
spring.datasource.password=student_1234
spring.sql.init.mode=always
spring.sql.init.schema-locations=classpath:schema-prod.sql
spring.sql.init.data-locations=classpath:data-prod.sql
```

Normally you will not reinitialise your production db, and only use the schema and data files for development.

- PostgreSql dialect may differ

```
DROP TABLE IF EXISTS PERSONS;
CREATE TABLE PERSONS(
   ID  SERIAL PRIMARY KEY,
   NAME CHARACTER VARYING(100) NOT NULL,
   FIRSTNAME CHARACTER VARYING(50)  NOT NULL,
   REMARK CHARACTER VARYING(256)
);
```

Or: Configure the database from code with @Profile

- Annotating beans with @Profile(“dev”) and @Profile(“prod”), will only activate them for these profiles
- Create two @Configuration classes H2DatabaseConfig and PostgreSqlDatabaseConfig

dev

```
@Bean
public DataSource dataSource(){
   DataSource dataSource = DataSourceBuilder
.create()
       .driverClassName("org.h2.Driver")
           .url("jdbc:h2:mem:studentdb")
           .username("sa")
           .password("")
           .build();
   return dataSource;
}
```

prod

```
@Bean
public DataSource dataSource(){
   DataSource dataSource = DataSourceBuilder
.create()

.driverClassName("org.postgresql.Driver")
           .url("jdbc:postgresql:pro3_db")
           .username("postgres")
           .password("student_1234")
           .build();
   return dataSource;
}
```

Slides code: profiles_code

Configuring @Profile from code

- Add `H2LoadData` and `PostgresLoadData` class (annotate with `@Component` and `@Profile`)
- Add `@PostConstruct` method in each class:

dev

```
@PostConstruct
public void loadData(){
  jdbcTemplate.update("DROP TABLE IF EXISTS
PERSONS");
  jdbcTemplate.update("""
    CREATE TABLE PERSONS(
      ID INTEGER AUTO_INCREMENT PRIMARY KEY,
      NAME CHARACTER VARYING(100) NOT NULL,
      FIRSTNAME CHARACTER VARYING(50)  NOT NULL,
      REMARK CHARACTER VARYING(256)
   );
""");
   jdbcTemplate.update("""
 INSERT INTO PERSONS(NAME, FIRSTNAME,REMARK)
VALUES ('JONES', 'JACK','of all trades'),
      ('POTTER', 'JACK','Lilly''s dad'),
      ('POTTER', 'MIA','Lilly''s mum'),
      ('REED', 'JACK','union');
      """);
}
```

prod

```
@PostConstruct
public void loadData(){
  jdbcTemplate.update("DROP TABLE IF EXISTS
PERSONS");
  jdbcTemplate.update("""
    CREATE TABLE PERSONS(
      ID INTEGER SERIALPRIMARY KEY,
      NAME CHARACTER VARYING(100) NOT NULL,
      FIRSTNAME CHARACTER VARYING(50)  NOT NULL,
      REMARK CHARACTER VARYING(256)
   );
""");
   jdbcTemplate.update("""
 INSERT INTO PERSONS(NAME, FIRSTNAME,REMARK)
VALUES ('JONES', 'JACK','of all trades'),
      ('POTTER', 'JACK','Lilly''s dad'),
      ('POTTER', 'MIA','Lilly''s mum'),
      ('REED', 'JACK','union');
      """);
}
```

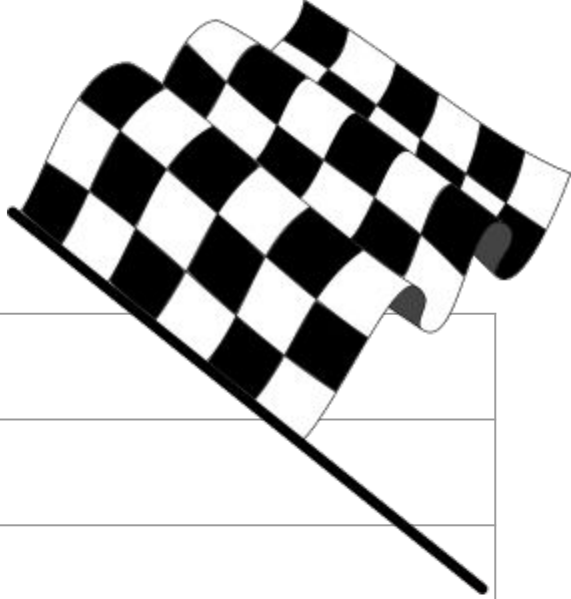
Profiles: there’s more…

- Profiles can be configured in many other ways and can be used for many other purposes, we only look at a small example
- For more info, check the tutorial: https://www.baeldung.com/spring-profiles

Exercise: using profiles

- Install the PostgreSQL database system on your platform
- Create a database using PostgreSQL tooling
- Try to connect using the IntelliJ Database tooling
- Walk through the previous slides and try to create 2 profiles, switch between the two and test your application.
    - Try to configure the databases using 2 application.properties files
    - Try to configure the databases from code using the @Profile annotation

Agenda this week



JDBC
Implementing the repository
JdbcTemplate
Spring profiles

KdG

Project

- Implement the Repository of your 2 main entities using Spring JDBC (JdbcTemplates) in combination with a H2B
- Use a schema.sql and data.sql to load initial data.
- Use profiles to be able to switch between the old implementation (using Java Collections) and the new implementation (using JDBC)