Spring Data Workshop
Lab 2

1 Lab setup

Make sure you have the following items installed

- MySQL 8.x
- Latest LTS JDK version (at this point: JDK 21)
- Spring Tool Suite (STS) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder just holds the source code files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 Setting up a MySQL user account

Start up the MySQL server (if it has not already started up) and connect to it via the MySQL command line client.

Switch to the database `workshopdb` with:

```
USE workshopdb
```

Check whether the `developers` table exists with:

```
SHOW TABLES;
```

If it does, delete it with:

```
DROP TABLE developers;
```

We will now create a user account and password and grant full privileges to it for accessing this database:

```
CREATE USER 'spiderman'@'%' IDENTIFIED BY 'peterparker';
GRANT ALL ON workshopdb.* TO 'spiderman'@'%';
```

We can check that the user account has been created successfully with:

```
SELECT user FROM mysql.user;
```

We can check the privileges granted to this account with:

```
SHOW GRANTS FOR 'spiderman'@'%';
```

3 Loading the initial database tables

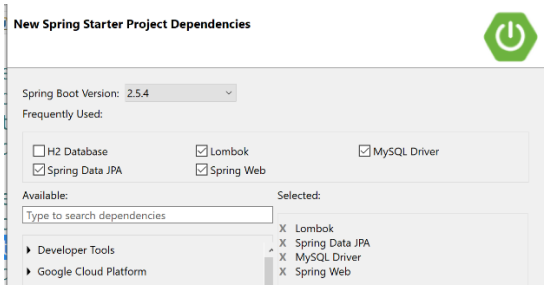
The main folder for this lab is `Data-JPA-Basic`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Other -> Spring Boot -> Spring Starter Project. Complete it with the following details:

Name: `DataJPABasic`
Group: `com.workshop.jpa`
Artifact: `DataJPABasic`
Version: `0.0.1-SNAPSHOT`
Description: `Demo setup and use of Spring Data JPA repo methods`
Package: `com.workshop.jpa`

Add the following dependencies:
```
Web -> Spring Web
SQL -> Spring Data JPA
SQL -> MySQL Driver
Developer Tools -> Project Lombok
Developer Tools -> Spring Boot Dev Tools
```



In the pom.xml of the newly generated project, notice that we now have additional dependencies for Spring Data JPA and the MySQL JDBC connector (in addition to the dependencies for Spring Web and Project Lombok)

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<scope>runtime</scope>
</dependency>

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

Other than MySQL, the Spring framework also provides support for other popular RDBMS such as IBM DB2, MS SQL Server, Oracle and PostgreSQL: all whose JDBC drivers can also be added as dependencies via the SQL section in a similar manner as the MySQL driver.

In `src/main/resources`, place the files

`application.properties`
`data.sql`
`schema.sql`

In `src/main/java` in the package `com.workshop.jpa`, place the files

`AppStartupRunner`

We can load an initial database using `*.sql` tables placed on the application classpath (typically in `src/main/resources`). The table creation statement and the data manipulation statements (to populate the tables with initial data) can be placed into separate files (`schema.sql` and `data.sql`).

The following needs to be configured in `application.properties`:

You need to disable Hibernate's DDL generation which takes priority over initialization from sql files:
`spring.jpa.hibernate.ddl-auto=none`

Also, to ensure that the data is loaded from these `*.sql` tables at application startup, you have to set this property:

```
spring.sql.init.mode=always
```

The remaining configuration in `application.properties` is to allow the app to connect to the specified database on a given MySQL server using a given username/password for an account (either root or normal user).

Run the app from the Boot dashboard in the usual way.
Back in the MySQL command line client, check that the `developers` table has been created and populated with sample data:

```
SHOW TABLES;

SELECT * FROM developers;
```

4 Using CrudRepository methods

In `src/main/resources`, make the following changes:

```
application.properties-v2
```

In `src/main/resources`, delete the following files:

```
data.sql
schema.sql
```

We have deleted the *.sql files used for initialization as the database table has already been created and for this round of execution, we intend to interact with it via standard CRUD methods.

Notice as well that this property in `application.properties` has been changed to allow Hibernate to change the contents of the database table:
`spring.jpa.hibernate.ddl-auto=update`

In `src/main/java` in the package `com.workshop.jpa`, place the files

```
Developer
BasicRepository
RandomRecordGeneratorService
```

In `src/main/java` in the package `com.workshop.jpa`, make these changes

```
AppStartupRunner-v2
```

Developer is part of the domain model. It is annotated with appropriate JPA annotations (@Entity, @Table, @Id, @GeneratedValue) to allow Hibernate to persist it to the Developers table that we just initialized previously. Notice the name and type of its fields correspond exactly to the names and types of the columns of the Developers table. The class is also annotated with relevant Lombok methods (@Setter, @Getter, etc) to minimize the boilerplate code for constructors, setters and getters.

The RandomRecordGeneratorService provides a method to generate a list of Developer objects with random values. These values are provided via the `options.languages` and `options.names` properties in the `application.properties`

Note that when creating the new Developer object, it is important to set the `id` value to null since it is marked as @GeneratedValue. This signals to Hibernate that you're creating a new entity, and it will generate new id value when storing the entity object. Otherwise, if you put in a valid value here, it will attempt to find the value in the target table and if it can't, it will throw an error message.

The BasicRepository interface extends the standard CrudRepository to make all the methods in this interface available for use. Spring Data JPA will automatically create a proxy implementation for this interface when the app is run.

Both BasicRepository and RandomRecordGeneratorService are made available to AppStartupRunner as fields annotated via @Autowired.

Run the app from the Boot dashboard in the usual way. Step through the app and check the log output to verify that the results of the various CRUD operations performed on the Developers table via the CrudRepository methods are as expected.

You will be able to see the Hibernate / JDBC statements that are executed on the table corresponding to the CrudRepository method invocations. This is due to the property `spring.jpa.show-sql=true` in `application.properties`. Comment out this property if you do not wish to see these statements.

You can also verify the results of the CRUD operations using standard SQL commands in the MySQL command line shell.

An alternative way to initialize the database table (other than loading from *.sql tables as demonstrated earlier) is to generate the tables in memory (for e.g. based on the code in RandomRecordGeneratorService) and then persisting them using the CrudRepository `save` method.

Notice that newly added rows do not take on the ids of previous existent records that are now deleted. The new ids are always generated by incrementing from the last id generated. As the id numbering of the rows are now longer consecutive, this may produce unexpected results in future reruns of this app.

You may therefore choose to delete the database table from inside the MySQL command line shell:

```
DROP TABLE developers;
```

And then reload the table with its original contents specified in `devtable.sql` in the `sqltables` folder. Open a command prompt in this folder (or navigate to it), then type:

```
mysql -u root -p workshopdb < devtable.sql
```

5 Using PagingAndSortingRepository methods

In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

`BasicRepository-v3`
`AppStartupRunner-v3`

The BasicRepository interface now extends the standard PagingAndSortingRepository to make all the methods in this interface available for use. Spring Data JPA will automatically create a proxy implementation for this interface when the app is run.

This interface provides an additional two methods for the purposes of pagination (i.e. returning only a subset of the full results returned from a `findAll` method call) and sorting. Pagination works on the concept of page number and page size. The result set returned can be is divided into groups of page size records: each group is known as a page. We can then specify a specific page to be returned, with paging index starting from 0. For e.g. consider a result set of 30 records. If we specify a page size of 10 records, then we have a total of 3 pages (page 0, 1, 2). If we retrieve page #0, we would obtain records 1 - 10, page #1 would give us records 11 - 20 while page #2 would give us records 21 - 30.

We can also specify a sorting order (including a secondary sorting order if necessary) on specific columns in the table. Then, we can impose a pagination on the sorted results if desired.

Run the app from the Boot dashboard in the usual way.
Step through the app and check the log output to verify that the results of the various pagination and sorting operations performed on the Developers table via the PagingAndSortingRepository methods are as expected.

6 Using derived query methods

In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

`BasicRepository-v4`
`AppStartupRunner-v4`

Derived query methods are simply declared in an interface that either extends CrudRepository or PagingAndSortingRepository. Query method names have two main parts separated by the first By keyword. The first part (find, query, count, etc) is called the subject (or introducer) and the second part is called the predicate (or criteria).

Spring Data JPA will automatically translate these query method declarations into appropriate implementations for this interface when the app is run. If there is an error in translation, a translation is thrown when the app is booted up.

Run the app from the Boot dashboard in the usual way.
Step through the app and check the log output to verify that the results of the various operations performed on the Developers table via the various derived query methods are as expected.

7 Using custom queries with @Query

In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

`BasicRepository-v5`

`AppStartupRunner-v5`

Spring Data JPA provides support for making queries using normal / native SQL for the database engine connected to, as well as Jakarta Query Persistence Language (JPQL). JPQL is based on the Hibernate Query Language (HQL), an earlier non-standard query language included in the Hibernate ORM library.

Writing queries in native SQL is the most performant, since Hibernate will not need to perform ORM to generate the query. However, this means the queries will have to be rewritten if the underlying database engine connected to is changed as SQL queries are not normally portable across different RDMBS (e.g. Oracle, MySQL, MS SQL Server, etc) which have their own native dialects of SQL. Queries in JQPL are not as performant as Hibernate will need to perform ORM to generate the query, but these queries are portable across different RDBMS.

The interface BasicRepository demonstrates a mixture of native SQL queries and JPQL queries. Some key points to note:

- Native SQL queries must have the additional attribute `nativeQuery = true`
- The methods that the SQL / JPQL queries are mapped to (`getAllTheOldLonelyDevs`, `whoIsYoungestDev`, etc) can be any random names (unlike derived query methods)
- The return type from the method must match that from the specified query. This is typically a List, but can also be an int, String or some other basic data type.
- Both native SQL / JPQL queries support pagination by having their mapped methods accept an additional argument of type Pageable. Additionally, native SQL query must also define a `countQuery` for this to work correctly

Run the app from the Boot dashboard in the usual way.
Step through the app and check the log output to verify that the results of the various native SQL / JPQL queries performed on the Developers table are as expected.

8 Working with an in-memory database H2

The main folder for this lab is `Data-JPA-H2`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Other -> Spring Boot -> Spring Starter Project. Complete it with the following details:

Name: `DataJPAH2`
Group: `com.workshop.jpa`
Artifact: `DataJPAH2`
Version: `0.0.1-SNAPSHOT`
Description: `Demo setup and use of in-memory database H2`
Package: `com.workshop.jpa`

Add the following dependencies:
```
Web -> Spring Web
SQL -> Spring Data JPA
SQL -> H2 Database
Developer Tools -> Lombok
```

`Developer Tools -> Spring Boot DevTools`

In addition to working with databases that persist data on disk (such as MySQL, MS SQL Server, Oracle, etc), we can also work with in-memory databases whose contents are persisted in memory and lost when the application is terminated. Spring Data JPA provides support for several of these types of databases: H2, HSQLDB (HyperSQL Database) and Apache Derby. These can all be added on as dependencies to a Spring Boot project.

Notice that we now have additional dependencies for the H2 database (in addition to the dependencies for Spring Web, Spring Data JPA and Project Lombok)

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

8.1 Loading the initial database tables

In `src/main/resources`, place the files

`application.properties`
`data-h2.sql`
`schema-h2.sql`

In `src/main/java` in the package `com.workshop.jpa`, place the files

`H2StartupRunner`

Just as for the case of MySQL, we can also load an initial database in H2 using `*.sql` tables placed on the application classpath (typically in `src/main/resources`). The table creation statements and the data manipulation statements (to populate the tables with initial data) can be placed into separate files (`schema-h2.sql` and `data-h2.sql`).

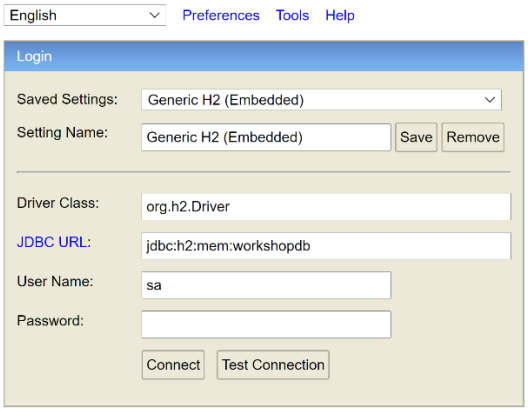
The following needs to be configured in `application.properties`:

You need to disable Hibernate's DDL generation which takes priority:
`spring.jpa.hibernate.ddl-auto=none`

Also, to ensure that the data is loaded from these `*.sql` tables at application startup, you have to set this property:
`spring.sql.init.mode=always`
To alert Spring that we are specifically working with h2 database, also set this:
`spring.sql.init.platform=h2`

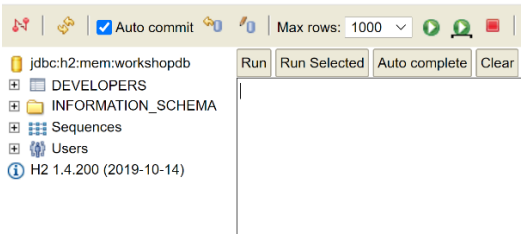
The remaining configuration in `application.properties` is to allow the app to connect to the specified database on the in-memory H2 database using a given username/password for an account.

Run the app from the Boot dashboard in the usual way. Once the server is up and running, check the H2 console at: http://localhost:8080/h2-console



Ensure the settings for the JDBC URL matches that specified in `application.properties` and type in the password provided in `application.properties`. Then click Connect

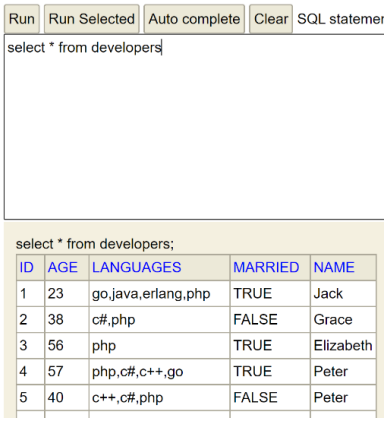
At the main H2 dashboard, you should be able to see the table DEVELOPERS in the left hand column under the database `workshopdb`:



In the SQL pane, type the SQL statement

```
select * from developers
```

and select Run. The entire contents of this table (30 initial rows) should be listed below. This verifies the table has been loaded correctly.



Stop the app from the Boot Dashboard in the usual manner.

8.2 Using CrudRepository methods

In `src/main/java` in the package `com.workshop.jpa`, place the files

```
Developer
BasicRepository
RandomRecordGeneratorService
```

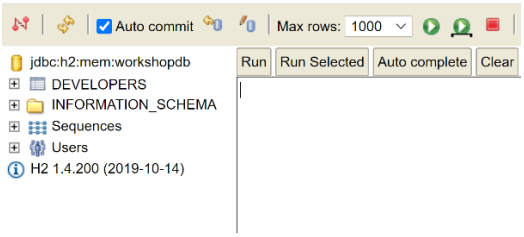
In `src/main/java` in the package `com.workshop.jpa`, make these changes

```
H2StartupRunner-v2
```

The functionality demonstrated here is exactly identical to that for the case of interacting with the MySQL table that we saw earlier. This illustrates that we can preserve the code base that uses standard CrudRepository functionality while swapping out the underlying database engines through the use of the Hibernate ORM abstraction.

Run the app from the Boot dashboard in the usual way. Step through the app and check the log output to verify that the results of the various CRUD operations performed on the Developers table via the CrudRepository methods are as expected.

You can also check the H2 console at: http://localhost:8080/h2-console
and login again at any point of time in the execution to see the table DEVELOPERS in the left hand column under the database `workshopdb`:



And check the latest content of this table with:

```
select * from developers
```

Stop the app from the Boot Dashboard in the usual manner.

8.3 Using PagingAndSortingRepository methods

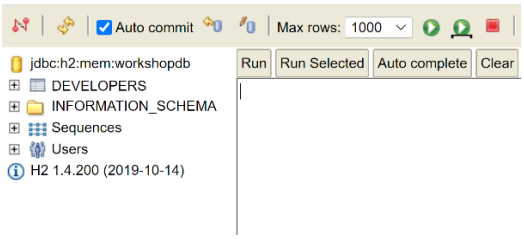
In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

```
BasicRepository-v3
H2StartupRunner-v3
```

The functionality demonstrated here is exactly identical to that for the case of interacting with the MySQL table that we saw earlier. This illustrates that we can preserve the code base that uses standard PagingAndSortingRepository functionality while swapping out the underlying database engines through the use of the Hibernate ORM abstraction.

Run the app from the Boot dashboard in the usual way. Step through the app and check the log output to verify that the results of the various pagination and sorting operations performed on the Developers table via the PagingAndSortingRepository methods are as expected.

You can also check the H2 console at: http://localhost:8080/h2-console
and login again at any point of time in the execution to see the table DEVELOPERS in the left hand column under the database `workshopdb`:



And check the latest content of this table with:

```
select * from developers
```

Stop the app from the Boot Dashboard in the usual manner.

8.4 Using derived query methods

In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

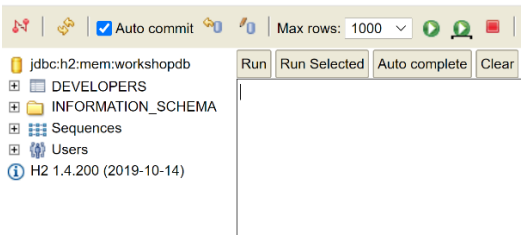
```
BasicRepository-v4
H2StartupRunner-v4
```

For the case of derived query methods, not all the functionality that we used in interacting with MySQL is preserved for the case of working with a H2 database. For e.g. all queries involving the keyword `Containing` are no longer producing the expected result here as the translated query by Hibernate does not achieve the expected result on the H2 table.

```
Hibernate: select developer0_.id as id1_0_, developer0_.age as age2_0_, developer0_.languages as language3_0_, developer0_.married as married4_0_, developer0_.name as name5_0_ from developers developer0_ where developer0_.languages like ? escape ?
```

Run the app from the Boot dashboard in the usual way. Step through the app and check the log output to verify that the results of the various operations performed on the Developers table via the various derived query methods are as expected with the exception of `Containing`

You can also check the H2 console at: http://localhost:8080/h2-console
and login again at any point of time in the execution to see the table DEVELOPERS in the left hand column under the database `workshopdb`:



And check the latest content of this table with:

```
select * from developers
```

Stop the app from the Boot Dashboard in the usual manner.

8.5 Using custom queries with @Query

In `src/main/java` in the package `com.workshop.jpa`, make the following changes:

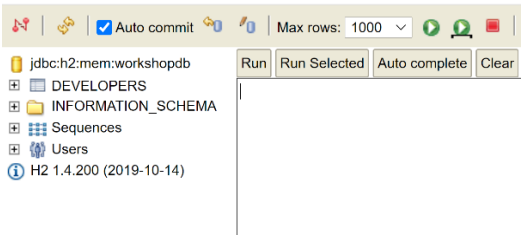
```
BasicRepository-v5
H2StartupRunner-v5
```

For the case of native queries, notice that there is much similarity between the SQL syntax for MySQL and H2. However, there is also some differences as well: for e.g. in MySQL to check whether a string is contained in the languages columns we would write (`INSTR(languages,:devlang) > 0`) whereas to achieve the same functionality in H2, we would write (`LOCATE(:devlang,languages) > 0`).

For the case of JPQL queries, the same queries that we used in MySQL achieve identical functionality here on H2. So, we have complete portability of queries here.

Run the app from the Boot dashboard in the usual way. Step through the app and check the log output to verify that the results of the various native SQL / JPQL queries performed on the Developers table are as expected.

You can also check the H2 console at: http://localhost:8080/h2-console
and login again at any point of time in the execution to see the table DEVELOPERS in the left hand column under the database `workshopdb`:



And check the latest content of this table with:

```
select * from developers
```

Stop the app from the Boot Dashboard in the usual manner.