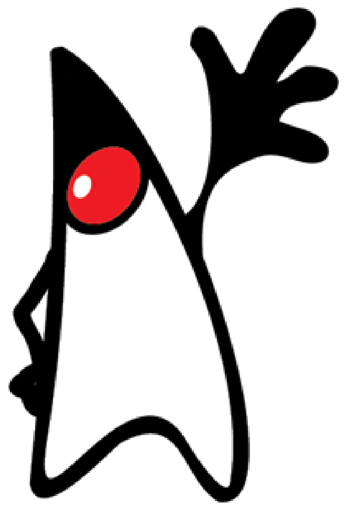
JPA overview



Inhoud

What is "Persistence"

- retention of application data for later use
  - Maximum scope of objects = application. When the application is closed, all objects are gone (and the values of their attributes)
  - => capture (=make persistent) all attribute values of all objects in a database
- Why ?
  - *“data lives longer than any application does.”*
- How?
  - store data in a relational database such as MySQL, Oracle, ...

Persistence: problems

- “Impedance mismatch”
  - = difference in approach between object-oriented and relational data structure https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
- Granularity
  - RDBMS has limited number of data types
  - Fine-meshed data structure not desirable in RDBMS based on performance
- Inheritance and Polymorphism
  - Not provided in RDBMS
- Identity of an object
  - Database : rows are equal if primary key is equal
  - OO
    - “Object identity” : object1 == object2
    - “Object equality” : object1.equals(object2)

Persistence: problems

- Associations between objects
  - Relationships between database tables are by definition
    - Unidirectional (via foreign keys)
    - 1 to many(1---*) or 1 to 1 (1---1)
    - *----* is not possible in a relational database but is possible in an OO Class Diagram

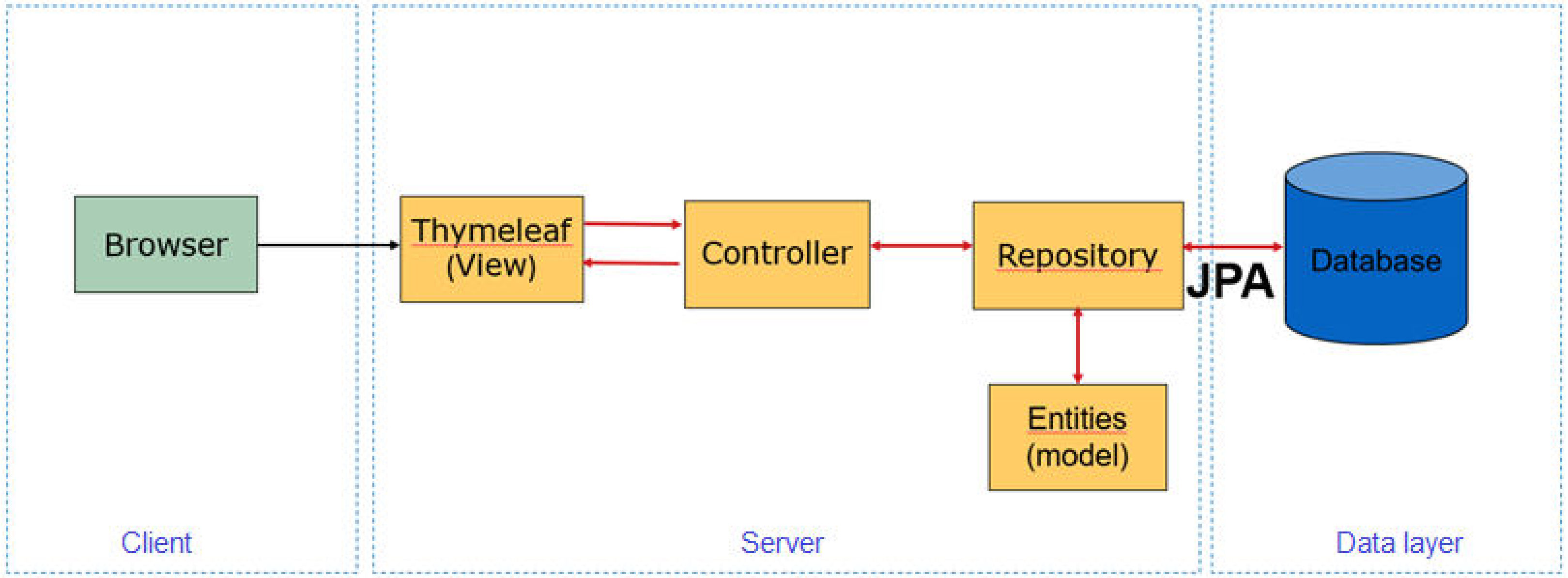
- Data retrieval
  - In OO: navigating from Object to Object
  - In database: retrieve as few tables as possible per query

Persistence

- We want to solve the difference between the relational model and the class diagram with
  - Rules that describe how to map an object to a row in a table (**object relational mapping = ORM**)
  - An easy way to make objects persistent (CRUD operations)

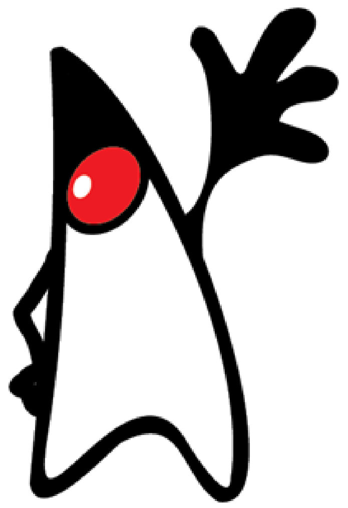
JPA

- Java Persistence API (JPA) is the standard object/relational mapping interface that allows us to make Java objects persistent in a relational database.



- In the remainder of this presentation we explain the above based on an example project

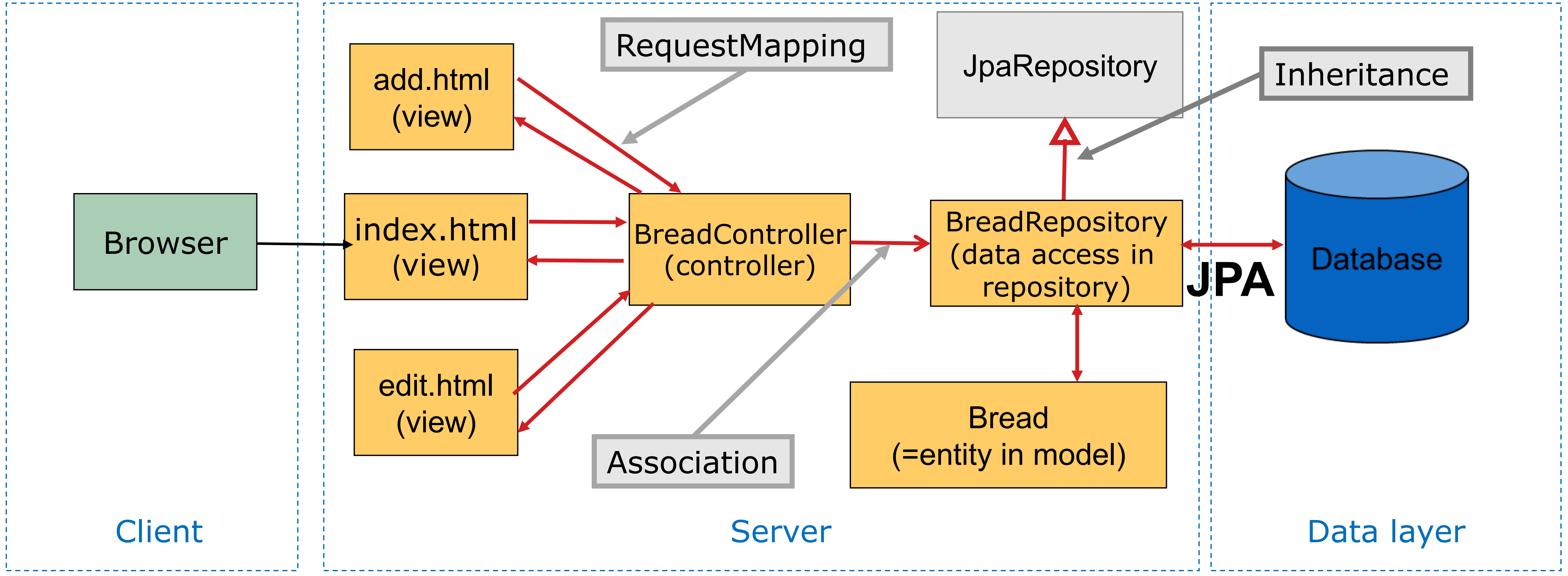
Creating a JPA example project



Contents

- This presentation will guide you through the process of creating a JPA project in IntelliJ.
- Follow these instructions carefully and you will learn what to do and how to do it. You must also be able to do this for the exam.
- Note: this example will not work if you have not installed the database correctly with the settings we use in these lessons. So go through the steps **in advance** in the document "Installation of MySQL 2022" on Canvas>Java advanced > Start
- The project that came with this item only serves in case you fail to make it yourself. It is best to use the (tele)coaching if you fail to make the project...
- Important tip: in the construction of the project it is possible that you have to (re)compile and (re)run several times: run File > Invalidate Caches/Restart...
- Good luck!

JPA example project

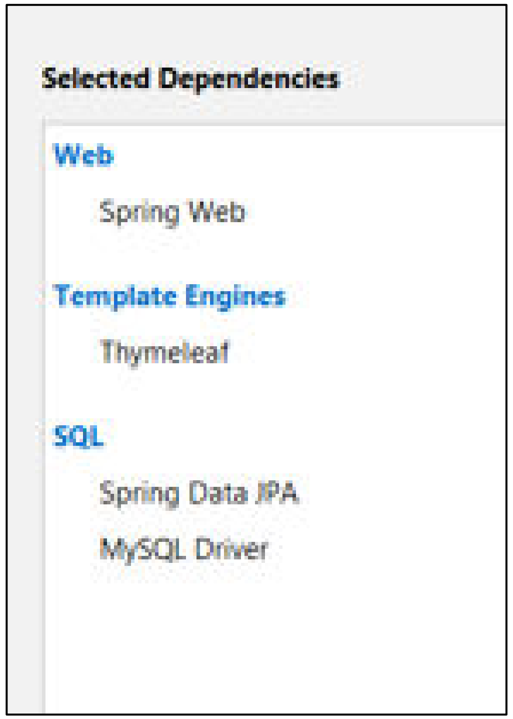


Creating a JPA example project in IntelliJ

- Step 1: Creating a Web project using the JPA Framework
- Step 2: Creating an entity "Bread" in the model package
- Step 3: Creating a repository class related to the "Bread" entity
- Step 4: Creating the BreadController
- Step 5: Creating the user interface

Step 1: Creating a new project

- Create a new project in IntelliJ (as you've learned so far) and select the additional dependencies:
  - SQL>Spring Data JPA
  - SQL>MySQL Driver
- See also the document on Canvas "Create project in IntelliJ‟
  - B. Web application with Thymeleaf and JPA



Selected Dependencies

Web

Spring Web

Template Engines

Thymeleaf

SQL

Spring Data JPA

MySQL Driver

Step 2: Creating an Entity 'Bread'

- Some characteristics of an Entity:
  - Can become persistent in the relational database:
    - Each Entity corresponds to a table in the database, and an object of this class corresponds to a record in this table
  - Supports transactions
  - Supports inheritance
  - Is identified by a *persistence id*
    - = the primary key of the table
    - preferably generated by the database
  - Attributes match the fields in a table
    - getters & setters for all attributes are needed to retrieve and save the data
  - How do we turn a class into an Entity? Via annotations. Each annotation:
    - Starts with @
    - Relates to the line of code immediately after the annotation

Step 2: Creating an Entity 'Bread'

Annotations:

@Entity

- annotation that makes it clear that this is not just a class, but a class whose objects can be made persistent in the database
- Convention = if you do not give details about the table to be made, then the table gets the same name as the class starting with a lowercase letter. The columns also get the names of the attributes.
- => When you create an Entity Bread, a Bread object will be stored in the bread table.

@Id

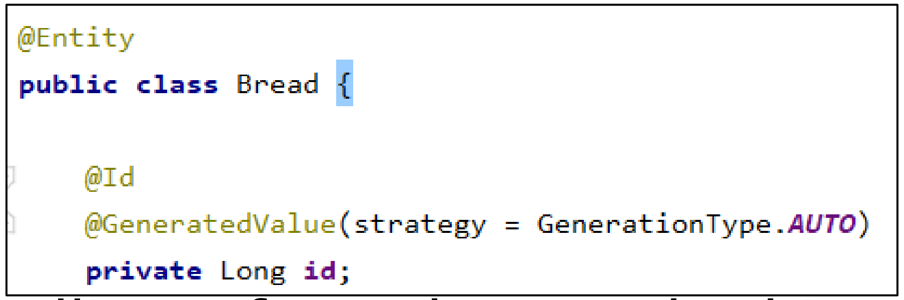
- This annotation indicates that the attribute corresponds to the primary key of the record

@GeneratedValue(strategy = GenerationType.AUTO)

- This annotation indicates that a newly created object will have the primary key generated by the database

Step 2: Creating an Entity 'Bread'

- First create the *model* package and create the *Bread* class.
- Complete the Bread class with the following code

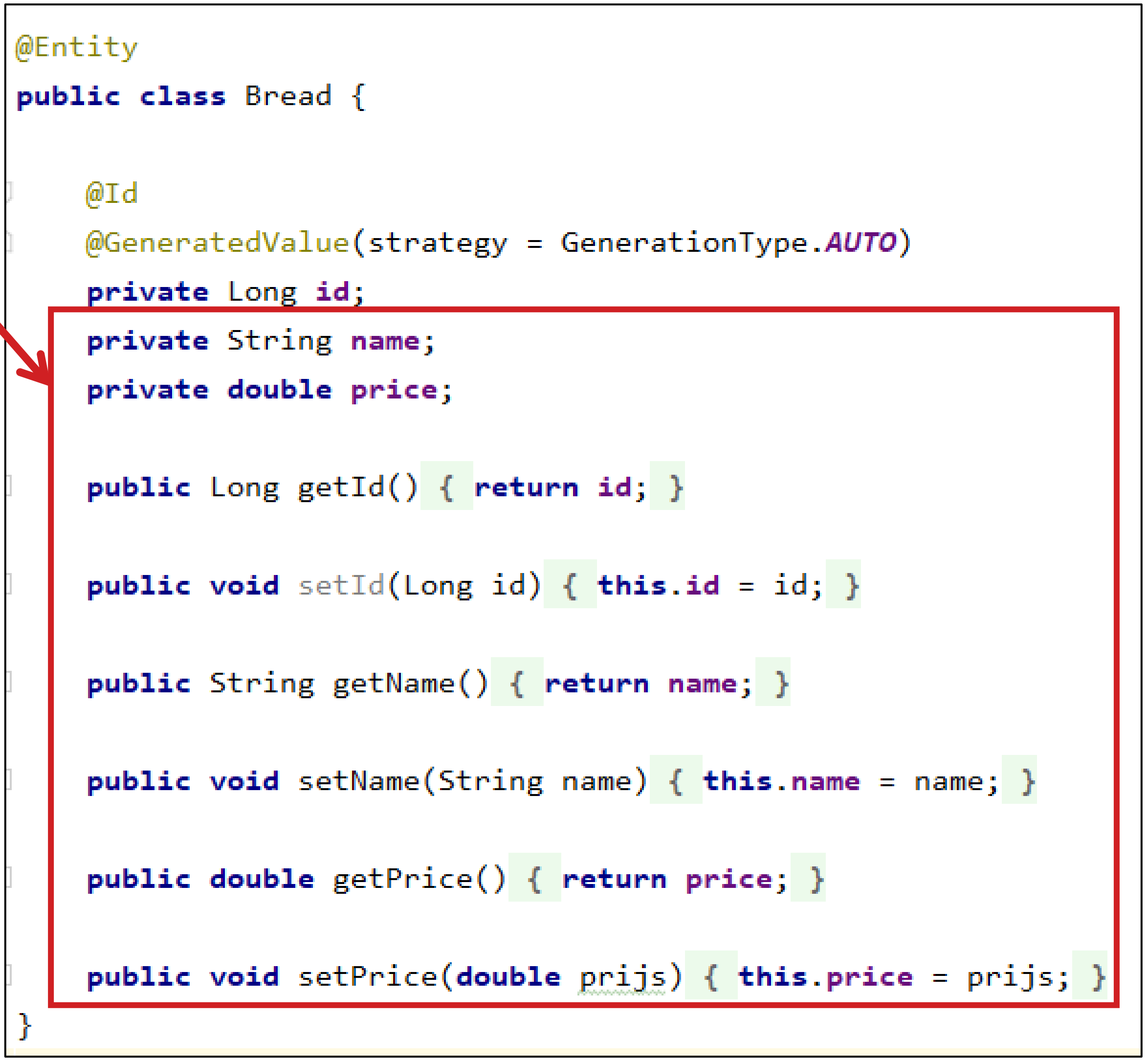


```
@Entity
public class Bread {

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

- And let IntelliJ perform the standard imports that go with it
- Then complete the entity with the attributes:
  - name: String
  - price: double
- Click right in the code and choose insert code...
  - Generate no-arg constructor: you can do this but it’s not strictly necessary. If you do not have a constructor, Java will call the default constructor (see lesson 3 Classes and objects of 1ITF Java)
  - Generate getters and setters...

Step 2: Creating an Entity 'Bread'

- Newly added code



```
@Entity
public class Bread {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private double price;

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public double getPrice() { return price; }

    public void setPrice(double prijs) { this.price = prijs; }
}
```

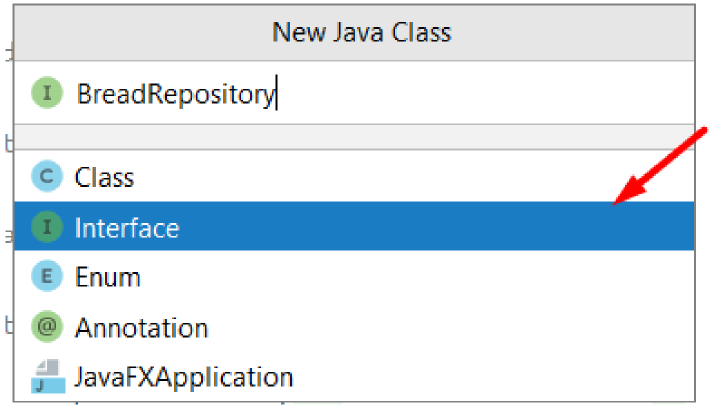
Step 3: Creating BreadRepository

A *Repository* in Spring Boot

- is a mechanism to get access to the database (CRUD operations). This is done through annotations and inheritance.
- in other words, the BreadRepository Interface that we are going to create will make the values stored in an object of the Entity class Bread persistent in the database and can also retrieve the corresponding data from the database and modify it.
- the BreadRepository inherits from another class which in turn inherits from another etc... This mechanism ensures that you can use a lot of interesting "pre-programmed" methods via this interface such as:
  - Retrieve all records from the "Bread" table and fill a List<Bread> with it
  - Add a new record in the "Bread" table based on the attribute values of a newly created object
  - Customize the fields of an existing record in the "Bread" table based on an object's modified attribute value
  - Delete a record from the "Bread" table when the corresponding object is deleted

Step 3: Creating BreadRepository

- First create the package repository and create a New.. Java class after which you choose Interface in the list and then enter the name BreadRepository:



New Java Class

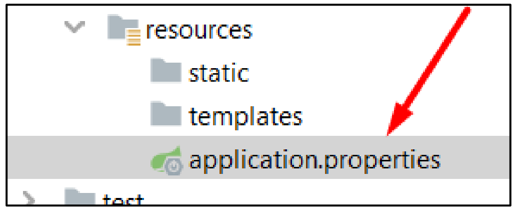
BreadRepository

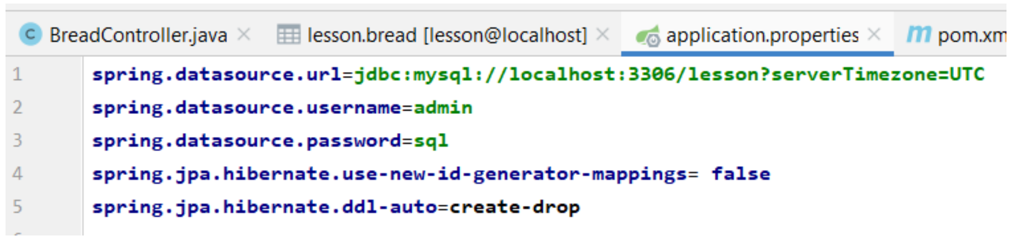
- Complete this interface with the following code:

```
@Repository
public interface BreadRepository extends JpaRepository<Bread, Long> {
}
```

- And let IntelliJ perform the standard imports that go with it

Step 3: Creating BreadRepository

- To create the link to the database, you must also provide the necessary settings in the application.properties file. Open this file:
- 



```
spring.datasource.url=jdbc:mysql://localhost:3306/lesson?serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=sql
spring.jpa.hibernate.use-new-id-generator-mappings= false
spring.jpa.hibernate.ddl-auto=create-drop
```

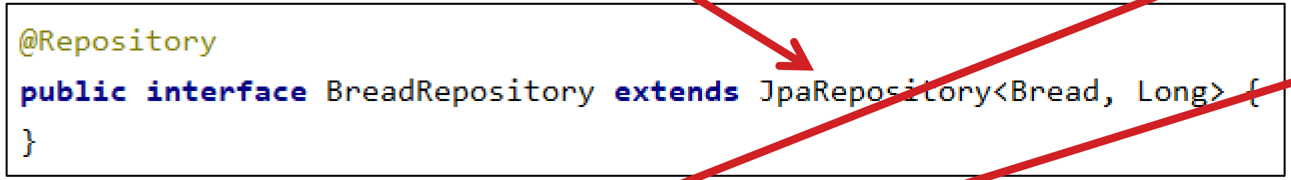
- Copy and paste the following code here:

```
spring.datasource.url=jdbc:mysql://localhost:3306/lesson?serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=sql
spring.jpa.hibernate.use-new-id-generator-mappings= false
spring.jpa.hibernate.ddl-auto=create-drop
```

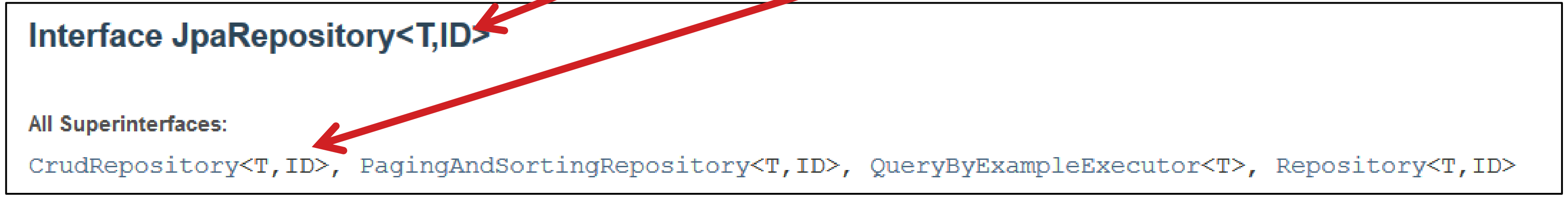
- This code provides the link to your lesson database lesson and when you run your application, the database will first be emptied, so that you always start "with a clean slate"...

Step 3: Creating BreadRepository

- BreadRepository inherits from JpaRepository<T, ID> which in turn also inherits from other super interfaces



```
@Repository
public interface BreadRepository extends JpaRepository<Bread, Long> {
}
```



Interface JpaRepository<T,ID>

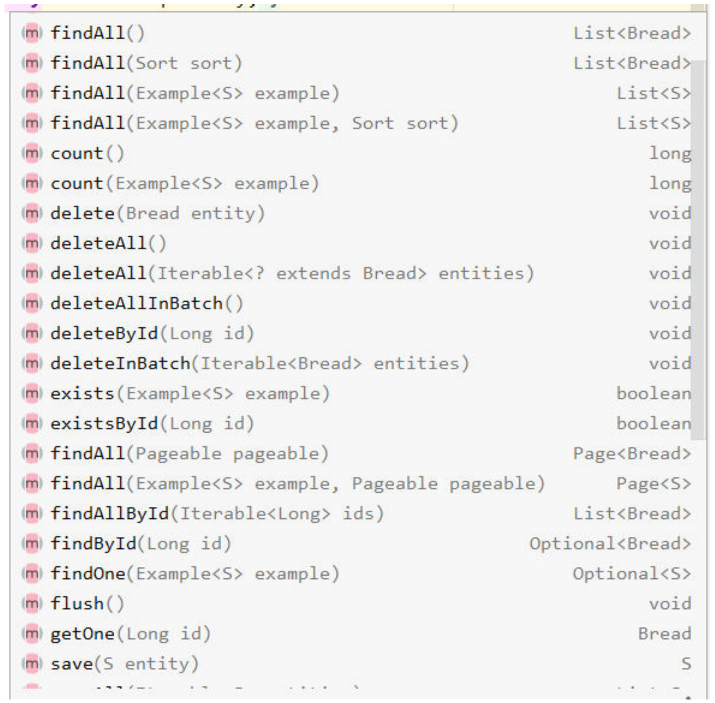
All Superinterfaces:

`CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>, Repository<T,ID>`

- JpaRepository is a generic interface:
  - the interpretation of which entity and what type of ID will be used to perform CRUD operations, has yet to be given
  - in this example: <T, ID> => <Bread, Long>

Step 3: Creating BreadRepository

- In this way you can use a lot of interesting methods to do CRUD operations on the database such as:



```
m findAll()                                                List<Bread>
m findAll(Sort sort)                                       List<Bread>
m findAll(Example<S> example)                                  List<S>
m findAll(Example<S> example, Sort sort)                       List<S>
m count()                                                         long
m count(Example<S> example)                                       long
m delete(Bread entity)                                            void
m deleteAll()                                                     void
m deleteAll(Iterable<? extends Bread> entities)                   void
m deleteAllInBatch()                                              void
m deleteById(Long id)                                             void
m deleteInBatch(Iterable<Bread> entities)                         void
m exists(Example<S> example)                                   boolean
m existsById(Long id)                                          boolean
m findAll(Pageable pageable)                               Page<Bread>
m findAll(Example<S> example, Pageable pageable)               Page<S>
m findAllById(Iterable<Long> ids)                          List<Bread>
m findById(Long id)                                    Optional<Bread>
m findOne(Example<S> example)                              Optional<S>
m flush()                                                         void
m getOne(Long id)                                                Bread
m save(S entity)                                                     S
```

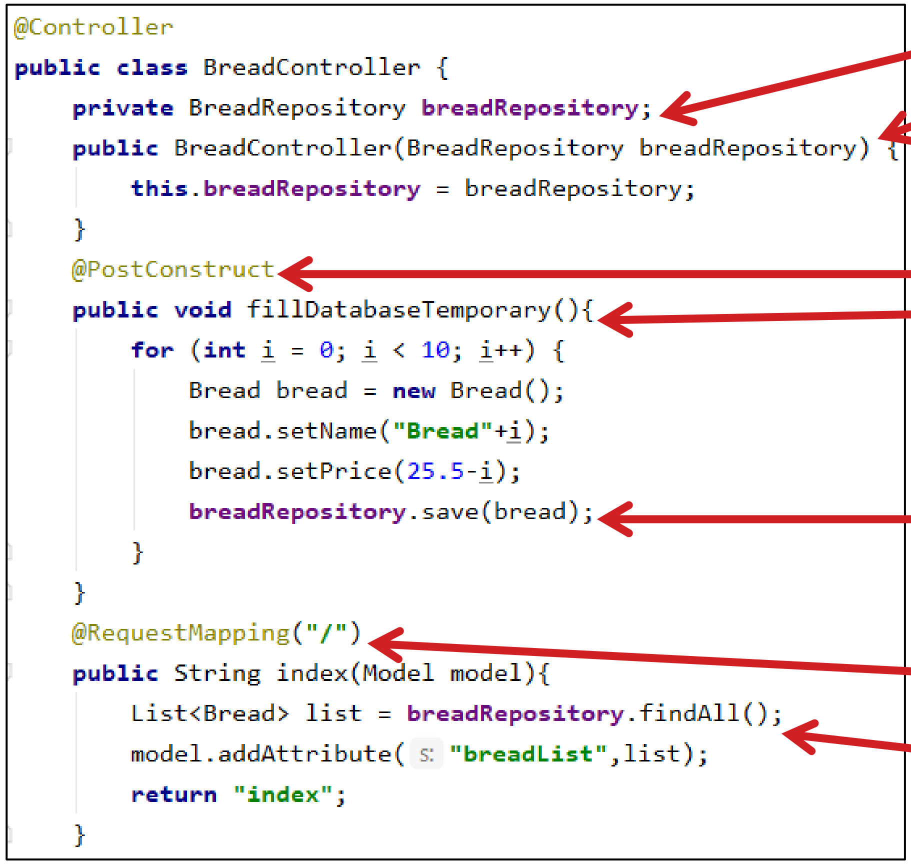
- For the implementation of these CRUD methods, the JPA framework is used

Step 3: Creating BreadRepository

- Interesting CRUD Methods:
  - List<Bread> findAll()
    - creates an object of the Bread class for each record in the bread table, and sets the attribute values to the values in the table
  - long count()
    - specifies the number of records in the bread table
  - void delete(Bread entity)
    - deletes the corresponding record in the bread table
  - Optional<Bread> findById(Long id)
    - this method returns an optional : value that can also be null if this method does not find a record with the given id. With the .get() method of the return value, you can retrieve the found bread object.
  - S save (S entity)
    - saves the object in the database: if no record exists yet, a new one is created, when the record already exists, the corresponding values are adjusted with the new values of the attributes
    - ….

Step 4: Creating BreadController

- First, create the package “controller” and then create the BreadController class inside this package.
- Complete the BreadController class with the following code (and do the required imports)



```
@Controller
public class BreadController {
    private BreadRepository breadRepository;
    public BreadController(BreadRepository breadRepository) {
        this.breadRepository = breadRepository;
    }
    @PostConstruct
    public void fillDatabaseTemporary(){
        for (int i = 0; i < 10; i++) {
            Bread bread = new Bread();
            bread.setName("Bread"+i);
            bread.setPrice(25.5-i);
            breadRepository.save(bread);
        }
    }
    @RequestMapping("/")
    public String index(Model model){
        List<Bread> list = breadRepository.findAll();
        model.addAttribute( s: "breadList",list);
        return "index";
    }
```

Association with BreadRepository set via the constructor

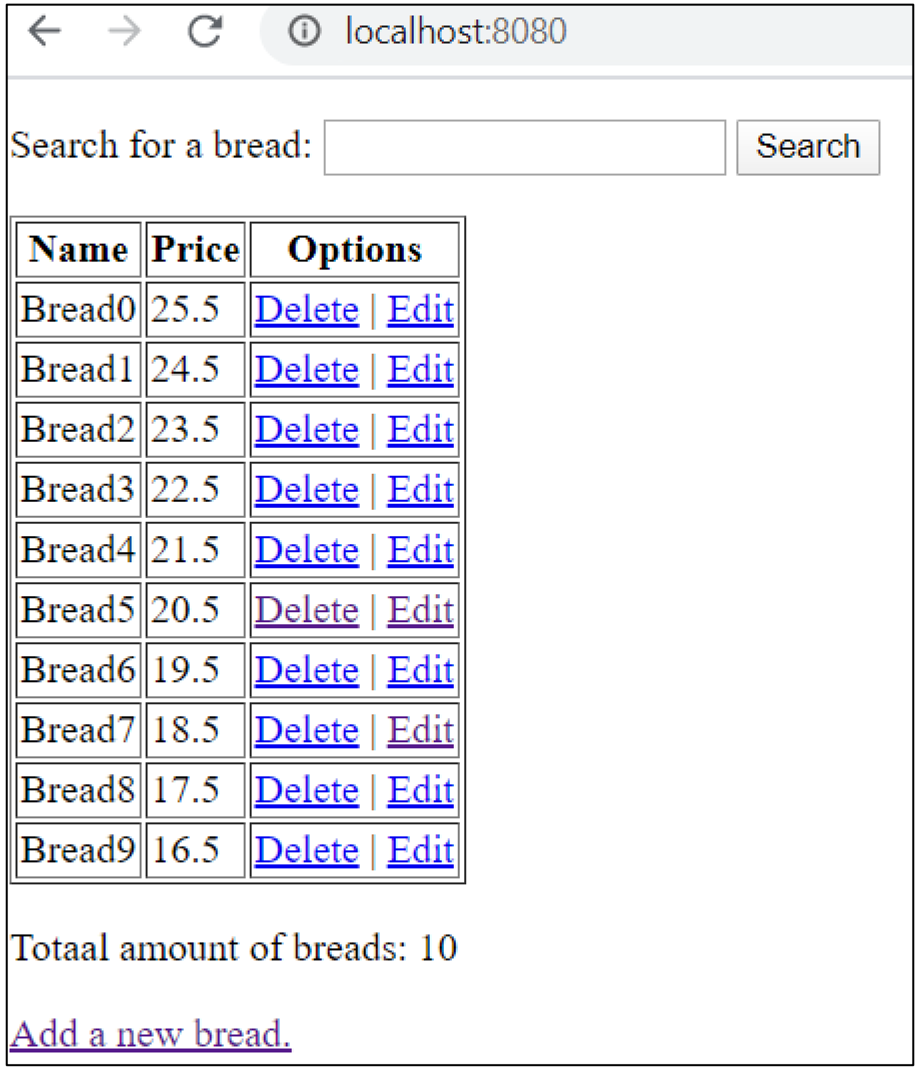
@PostConstruct ensures that the fillDatabaseTemporary() method runs after the BreadController constructor has been called and that the index.html is displayed.

The created Bread object is stored in a table “bread”.

If you surf to localhost:8080, this method will be performed first.
All records in the bread table are converted to Bread objects and placed in a list.

Step 5: Creating the Userinterface

- index.xhtml



localhost:8080

Search for a bread: Search

Name	Price	Options
Bread0	25.5	Delete | Edit
Bread1	24.5	Delete | Edit
Bread2	23.5	Delete | Edit
Bread3	22.5	Delete | Edit
Bread4	21.5	Delete | Edit
Bread5	20.5	Delete | Edit
Bread6	19.5	Delete | Edit
Bread7	18.5	Delete | Edit
Bread8	17.5	Delete | Edit
Bread9	16.5	Delete | Edit

Totaal amount of breads: 10

Add a new bread.

Step 5: Creating the Userinterface

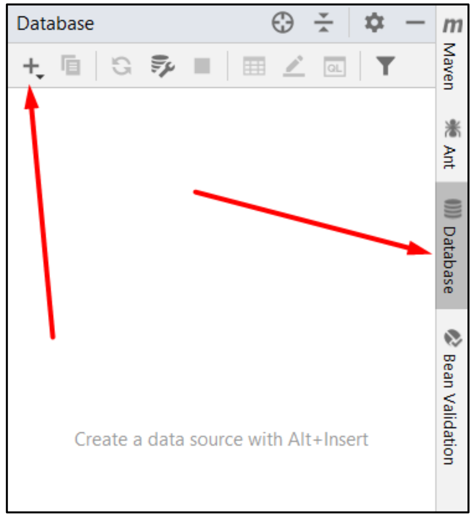
- Create a Thymeleaf HTML file index in the package templates and complete the "body" with the following code:

```
<form action="/search" method="post">
    <p>
        <label for="searchstring">Search for a bread: </label>
        <input type="text" name="searchstring" id="searchstring">
        <input type="submit" value="Search" name="searchname">
    </p>
</form>

<table border="1">
    <tr>
        <th>Name</th>
        <th>Price</th>
        <th>Options</th>
    </tr>
    <tr th:each="bread : ${breadList}">
        <td th:text="${bread.getName()}"></td>
        <td th:text="${bread.getPrice()}"></td>
        <td>
            <a th:href="@{/delete(breadId=${bread.getId()})}">Delete</a> |
            <a th:href="@{/edit(breadId=${bread.getId()})}">Edit</a>
        </td>
    </tr>
</table>

<p th:text="'Totaal amount of breads: ' + ${breadList.size()}"></p>
<a href="/add">Add a new bread.</a>
```

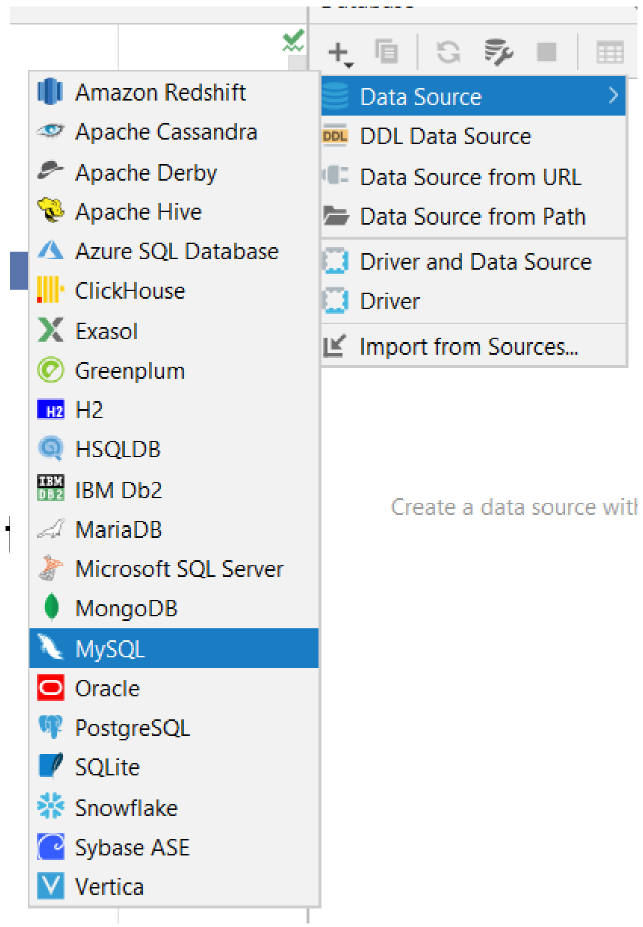
- Run your project. If everything goes well, you will get the screen that was in the previous slide...
- The links and buttons don't work yet, because we haven't written the corresponding mapping methods in the BreadController yet...

Linking the database…

- To see the result of your code in the database, you can open MySQL Workbench or work via IntelliJ.
- Here's how:



Create a data source with Alt+Insert

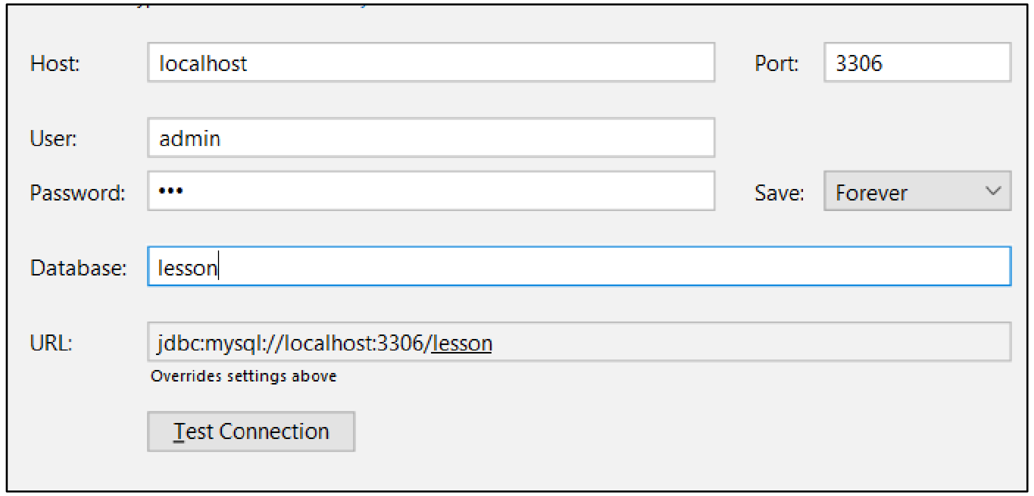


Create a data source witl

- Click on Database in the left margin and t source>MySQL

Linking the database…

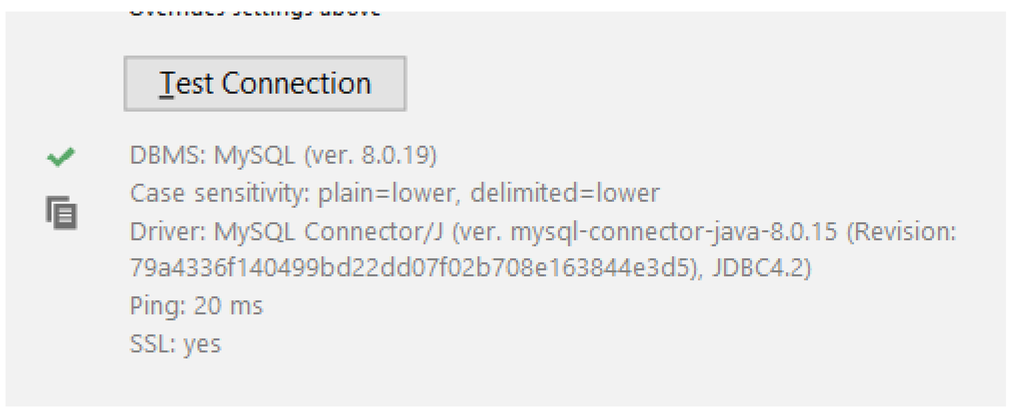
- Enter the connection values (as in the application.properties-file):



Host:	localhost	Port:	3306
User:	admin		
Password:	•••	Save:	Forever
Database:	lesson		
URL:	jdbc:mysql://localhost:3306/lesson		

Overrides settings above

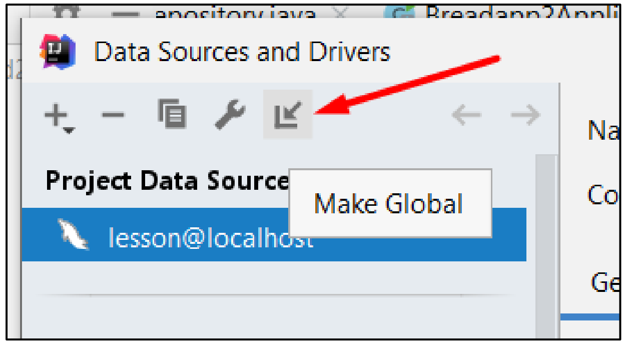
- Click on "Test Connection" and if everything goes well you will get this result:



```
DBMS: MySQL (ver. 8.0.19)
Case sensitivity: plain=lower, delimited=lower
Driver: MySQL Connector/J (ver. mysql-connector-java-8.0.15 (Revision:
79a4336f140499bd22dd07f02b708e163844e3d5), JDBC4.2)
Ping: 20 ms
SSL: yes
```

Linking the database…

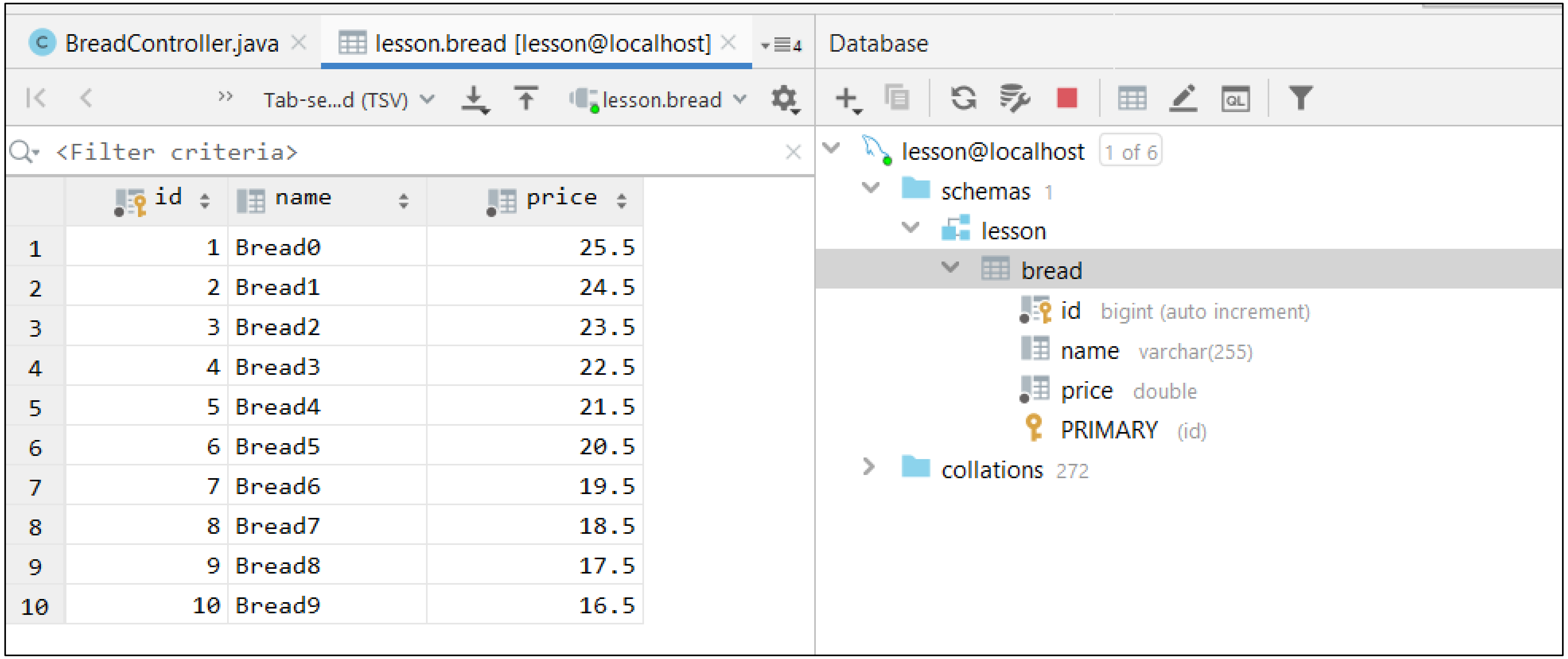
- Then click on the icon at the top left of the properties window:



- You now make this database connection "global" for your entire IntelliJ environment. Clicking on your database tab in the right margin will immediately connect you to this database...
- Click OK so that the properties window closes.

Linking the database…

- To see the full contents of a table, <u>double-click the table</u>...



BreadController.java | lesson.bread [lesson@localhost]

Tab-se...d (TSV) | lesson.bread

<Filter criteria>

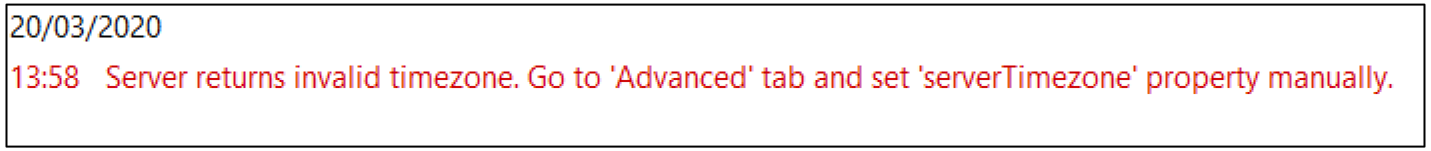
	id	name	price
1	1	Bread0	25.5
2	2	Bread1	24.5
3	3	Bread2	23.5
4	4	Bread3	22.5
5	5	Bread4	21.5
6	6	Bread5	20.5
7	7	Bread6	19.5
8	8	Bread7	18.5
9	9	Bread8	17.5
10	10	Bread9	16.5

Database

```
lesson@localhost  1 of 6
  schemas  1
    lesson
      bread
        id  bigint (auto increment)
        name  varchar(255)
        price  double
        PRIMARY  (id)
  collations  272
```

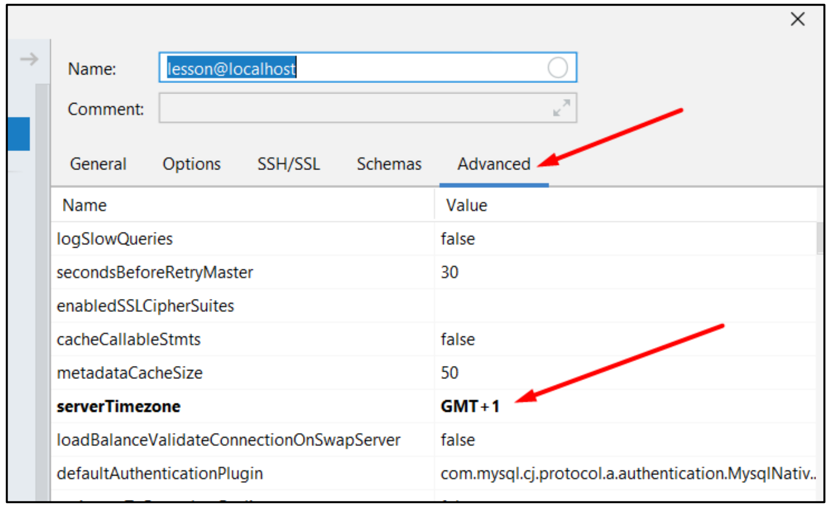
Linking the database...

- If you don't see the content, but get this error:



```
20/03/2020
13:58   Server returns invalid timezone. Go to 'Advanced' tab and set 'serverTimezone' property manually.
```

- Then go (back) to the properties window of your database click on tab Advanced where you have to set the setting "serverTimezone" like this:



Name: lesson@localhost

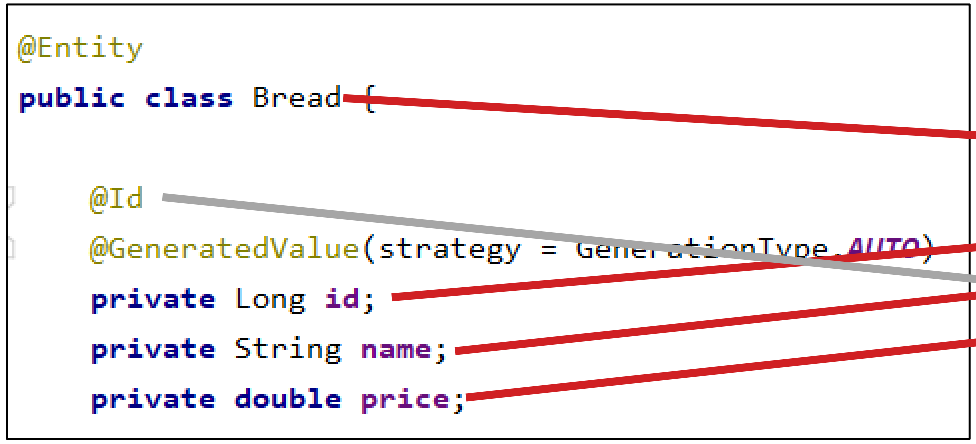
Comment:

Name	Value
logSlowQueries	false
secondsBeforeRetryMaster	30
enabledSSLCipherSuites	
cacheCallableStmts	false
metadataCacheSize	50
serverTimezone	**GMT+1**
loadBalanceValidateConnectionOnSwapServer	false
defaultAuthenticationPlugin	com.mysql.cj.protocol.a.authentication.MysqlNativ..

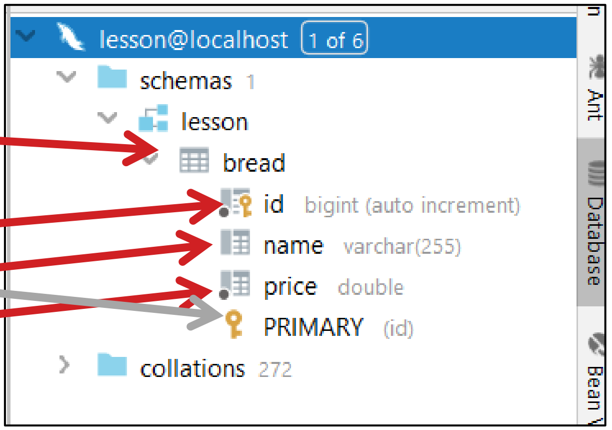
- After that, click Apply and OK and try again...

Linking the database…

- You will then see your lesson database with the "bread" table in it:

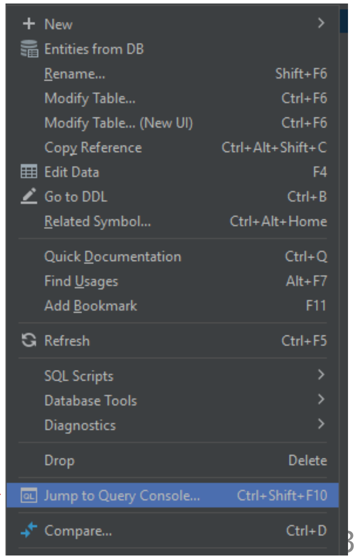


```java
@Entity
public class Bread {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private double price;
```



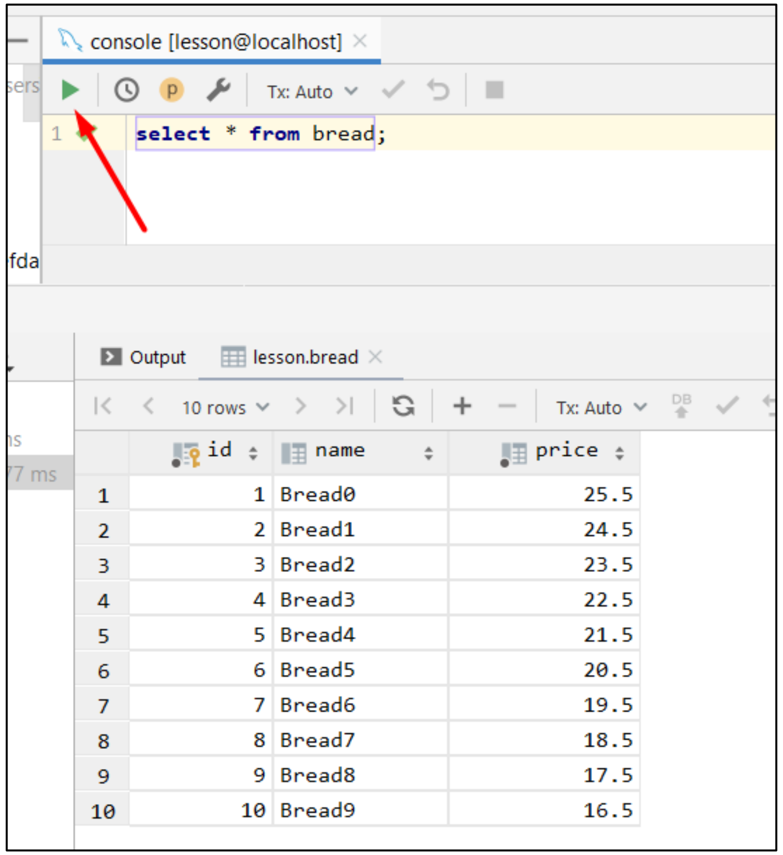
```
lesson@localhost  1 of 6
  schemas 1
    lesson
      bread
        id  bigint (auto increment)
        name  varchar(255)
        price  double
        PRIMARY  (id)
  collations 272
```

- Right-click the bread table and click “Jump to Query Console” to run queries and see the results.



Linking the database…

- Type query and click on the green triangle: this gives the result of your query at the bottom of the screen:

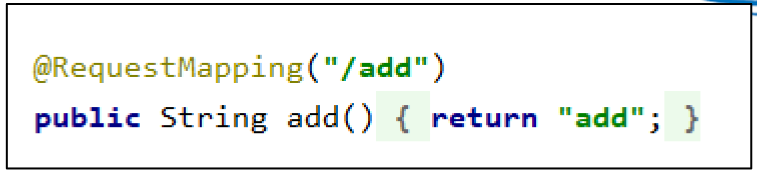


```
select * from bread;
```

	id	name	price
1	1	Bread0	25.5
2	2	Bread1	24.5
3	3	Bread2	23.5
4	4	Bread3	22.5
5	5	Bread4	21.5
6	6	Bread5	20.5
7	7	Bread6	19.5
8	8	Bread7	18.5
9	9	Bread8	17.5
10	10	Bread9	16.5

Completing the BreadController and user interface

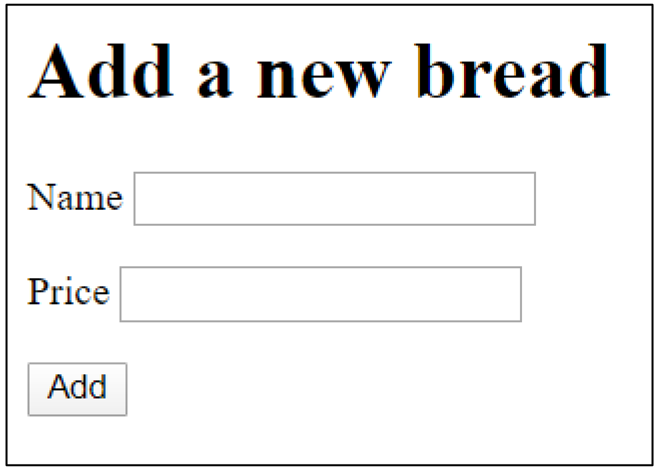
- Add the following method in BreadController:



```
@RequestMapping("/add")
public String add() { return "add"; }
```

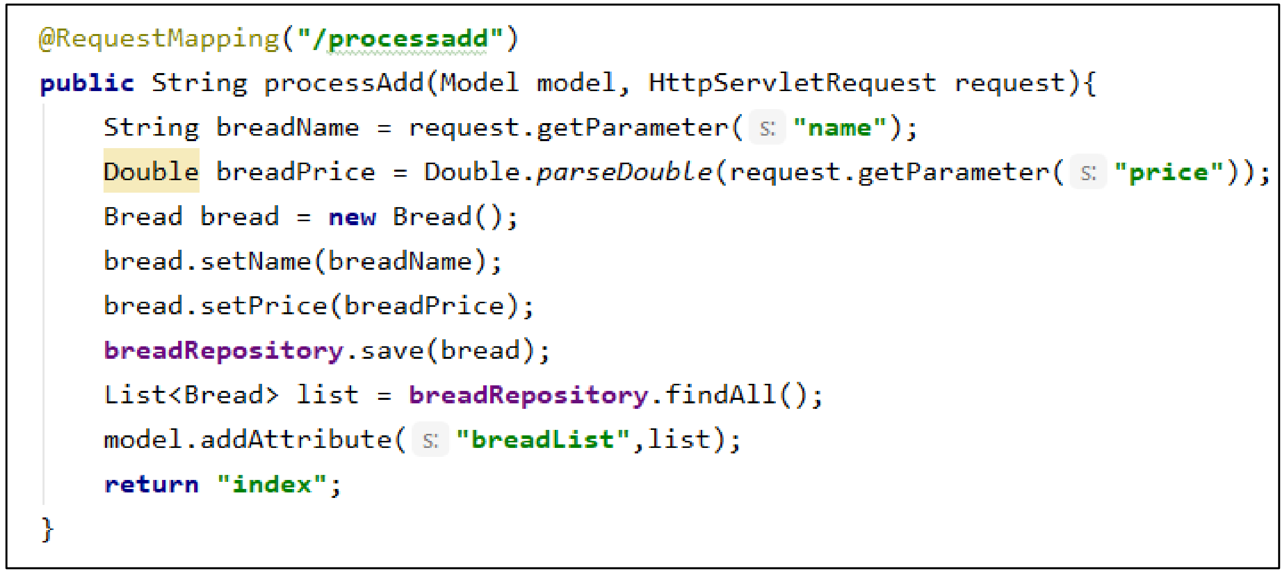
- Create a new Thymeleaf page, *add.html,* and insert the following code in the "body":
- `<h1>Add a new bread</h1>`

```
<form action="/processadd" method="post">
    <p>
        <label for="name">Name</label>
        <input type="text" name="name" id="name">
    </p>
    <p>
        <label for="price">Price</label>
        <input type="text" name="price" id="price">
    </p>
    <p>
        <input type="submit" value="Add" name="add">
    </p>
</form>
```



Completing the BreadController and user interface

- Add the following method to the BreadController:



```
@RequestMapping("/processadd")
public String processAdd(Model model, HttpServletRequest request){
    String breadName = request.getParameter( s: "name");
    Double breadPrice = Double.parseDouble(request.getParameter( s: "price"));
    Bread bread = new Bread();
    bread.setName(breadName);
    bread.setPrice(breadPrice);
    breadRepository.save(bread);
    List<Bread> list = breadRepository.findAll();
    model.addAttribute( s: "breadList",list);
    return "index";
}
```

- Run your project and test the link "add a new bread"...

Completing the BreadController and user interface

- We are now also completing the code behind the links "edit" and "delete".
- Copy the code below to your BreadController (if you copy first to notepad and then to IntelliJ, the strange ppt characters will disappear...):

```
@RequestMapping("/edit")
public String edit(Model model, HttpServletRequest request){
    Long breadId = Long.valueOf(request.getParameter("breadId"));
    //breadRepository.findById(breadId) returns an Optional, a value that ca
retrieve the actual value.
    Bread bread = breadRepository.findById(breadId).get();
    model.addAttribute("bread", bread);
    return "edit";
}

@RequestMapping("/processedit")
public String processEdit(Model model, HttpServletRequest request){
    Long breadId = Long.valueOf(request.getParameter("breadId"));
    String breadName = request.getParameter("name");
    Double breadPrice = Double.parseDouble(request.getParameter("price"));
    Bread bread = breadRepository.findById(breadId).get();
    bread.setName(breadName);
    bread.setPrice(breadPrice);
    breadRepository.save(bread);
    List<Bread> list = breadRepository.findAll();
    model.addAttribute("breadList",list);
    return "index";
}

@RequestMapping("/delete")
public String delete(Model model, HttpServletRequest request){
    Long breadId = Long.valueOf(request.getParameter("breadId"));
    breadRepository.deleteById(breadId);
    List<Bread> list = breadRepository.findAll();
    model.addAttribute("breadList",list);
    return "index";
}
```

We can retrieve an object from the "id" in the table. This "id" is of the Type Long, so we first have to convert it from String to Long in order to then use the "findById" or "deleteById" method.

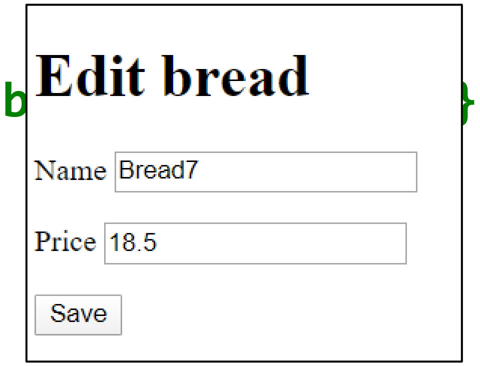
Note: we also need to call the .get() method of the findById return value to get the object back itself.

Completing the BreadController and user interface

- Finally, create a final Thymeleaf HTML page: edit and copy this code into the body:

breadId will simply be retrievable via the request.getParameter method, just like the other form elements. You have to use the post-method for this...

```
<h1>Edit bread</h1>

<form th:action="@{/processedit(breadId=${bread.getId()})}" method="post">
    <p>
        <label for="name">Name</label>
        <input type="text" name="name" id="name" th:value="${bread.getName()}">
    </p>
    <p>
        <label for="price">Price</label>
        <input type="text" name="price" id="price" th:value="${b                 }">
    </p>
    <p>
        <input type="submit" value="Save" name="save">
    </p>
```



Edit bread

Name Bread7

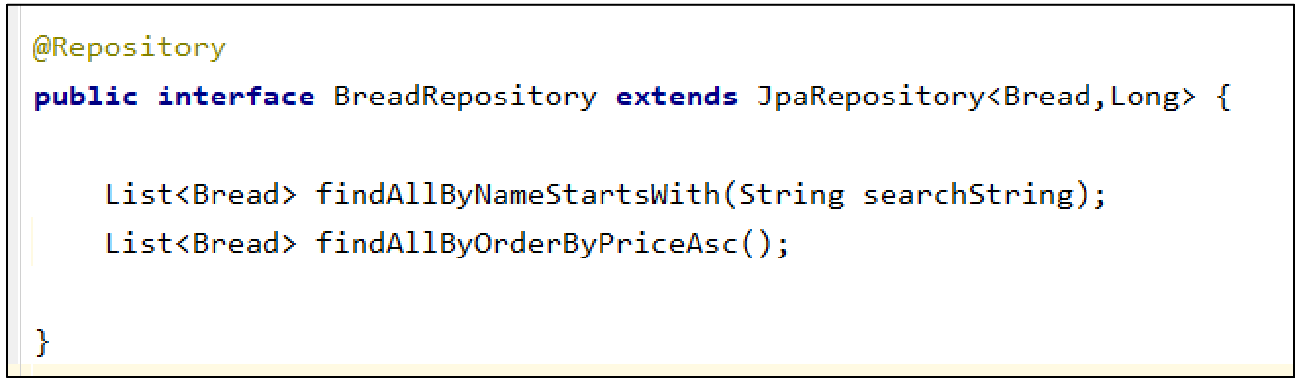
Price 18.5

Save

- Test your code. Everything should work except the search function...

Create Keyword query method for searching

- the standard crud methods are often not sufficient
  - For example, you cannot search for a (part of) a value of a certain attribute...
- Open the BreadRepository Interface and add the following "Keyword queries":



```
@Repository
public interface BreadRepository extends JpaRepository<Bread,Long> {

    List<Bread> findAllByNameStartsWith(String searchString);
    List<Bread> findAllByOrderByPriceAsc();

}
```

- You'll notice that IntelliJ automatically completes it...
  - These are extra pre-programmed methods that you can call and that can be built depending on the attributes of Bread.
  - The keyword query runs the following native SQL code in the background :
    - findAllBy => select * from bread
    - ByXXX => where ...

Keyword query methods

- LIKE Query Methods
  - The equivalent of the following MySQL query
    - `SELECT * FROM movie WHERE title LIKE '%in%';`
  - Can be converted into keyword query methods:
    - `List<Movie> findByTitleContaining(String title);`
    - `List<Movie> findByTitleContains(String title);`
    - `List<Movie> findByTitleIsContaining(String title);`
  - All 3 methods give the same result…
  - The equivalent of the following MySQL query
    - `SELECT * FROM movie WHERE title LIKE 'in%';`
  - Can be converted into keyword query methods:
    - `List<Movie> findByTitleStartsWith(String title);`
  - The equivalent of the following MySQL query
    - `SELECT * FROM movie WHERE title LIKE ‘%in’;`
  - Can be converted into keyword query methods:
    - `List<Movie> findByTitleEndsWith(String title);`

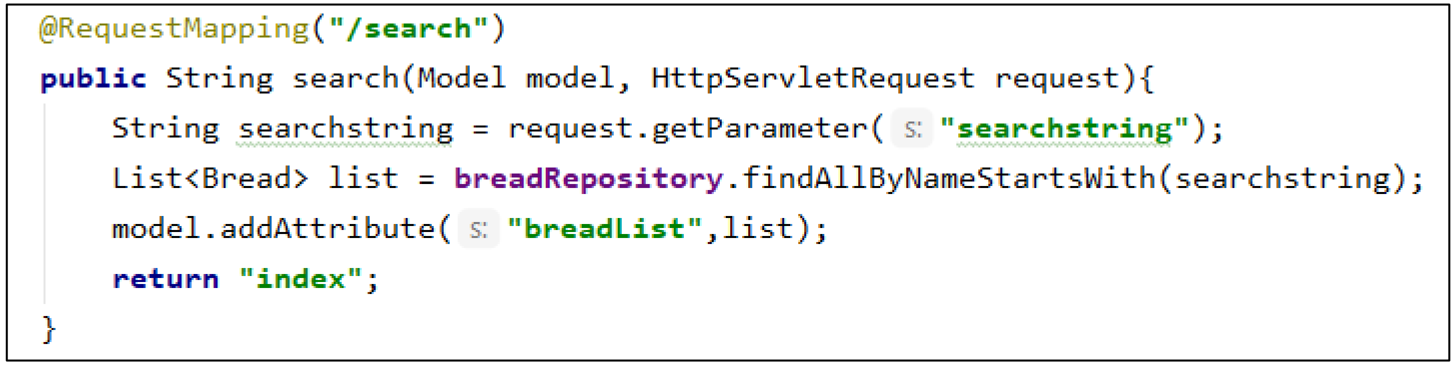
Keyword query methods

- Examples of keyword queries with Order By
  - `List<Person> findByLastnameOrderByFirstnameAsc(String lastname);`
    - `List<Person> findByLastnameOrderByFirstnameDesc(String lastname);`
- Attention! If you want to sort a list (without a where clause) you still have to repeat "By":
  - `List<Person> findAllByOrderByFirstnameDesc();`
- Tip! Use ctrl spacebar to build your keyword queries
- Other possibilities see:
  - https://docs.spring.io/spring-data/jpa/docs/1.4.3.RELEASE/reference/html/repositories.html
  - https://docs.spring.io/spring-data/jpa/docs/1.4.3.RELEASE/reference/html/repository-query-keywords.html

- If you have written the Keyword query methods in the Repository Interface, you can use them in the Controller class...

Create Keyword query method for searching

- Open the BreadController and add the following method:

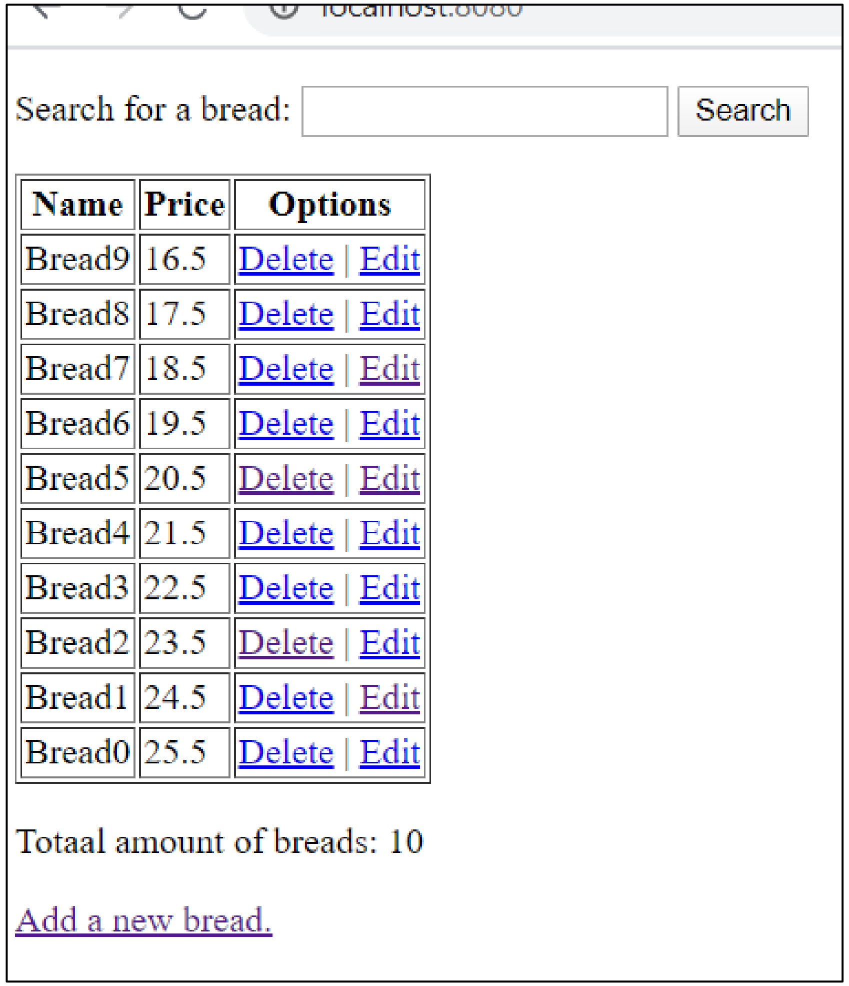


```
@RequestMapping("/search")
public String search(Model model, HttpServletRequest request){
    String searchstring = request.getParameter( s: "searchstring");
    List<Bread> list = breadRepository.findAllByNameStartsWith(searchstring);
    model.addAttribute( s: "breadList",list);
    return "index";
}
```

- Then replace the "findAll()" methods with the "findAllByOrderByPriceAsc()" in the following RequestMapping methods:
  - index(Model model)
  - processAdd (Model model, HttpServletRequest request)
  - processEdit-methodes (Model model, HttpServletRequest request)

Test your project

- Start your project...
- The loaves of bread are now sorted in ascending order of price...
- Add some loaves of bread
- Also test the Edit and Delete links...
- Also check what the result in the database looks like
- When errors occur, read what it says in the output window and try to fix your error(s) based on this...



Search for a bread: Search

Name	Price	Options
Bread9	16.5	Delete | Edit
Bread8	17.5	Delete | Edit
Bread7	18.5	Delete | Edit
Bread6	19.5	Delete | Edit
Bread5	20.5	Delete | Edit
Bread4	21.5	Delete | Edit
Bread3	22.5	Delete | Edit
Bread2	23.5	Delete | Edit
Bread1	24.5	Delete | Edit
Bread0	25.5	Delete | Edit

Totaal amount of breads: 10

Add a new bread.

JPQL

- Keyword queries have their limitations. You can run simple queries with this, but if you want to run a more complex query, JPQL is recommended.
- JPQL = **JP**A **Q**uery **L**anguage: JPA's query language is very similar to SQL
- JPQL is **Java code** (and NOT SQL) and therefore:
  - the queries are completely OO, with possibilities such as inheritance, polymorphism, association. You don't have to write joins often when you work OO
  - we use the names of the attributes (of the Entity) and not the column names (from the corresponding table) making the query database independent
  - The JPA framework provides:
    - the translation from JPQL to SQL
    - converting the records (with different field values) from a table to Java objects with corresponding attribute values
  - The syntax in the queries is not case-sensitive:
    - seLEct or SELECT or select or Select is identical
  - The names of the classes and the attributes used in the queries are CASE-sensitive!
    - Bread is not the same as BREAD or bread!
    - Bread.name's not the same as Bread.name!

Simple query

- As an alternative to List<Bread> findAllByOrderByPriceAsc(); you can also use JPQL:
  - SELECT b from Bread b order by b.price asc
    - You select all instances of the entity Bread
    - In the background, "Select * from bread order by price asc" is executed, and all records from the bread table become objects of the Bread class
    - You are **required to create an alias** for each entity. This alias is used to name an instance/object of the entity
      - One chooses to use one lowercase letter for the alias
- SELECT b from Bread b WHERE b.name = ‘white bread'
  - name refers to an attribute of the class
  - so don't write
    - SELECT b from Bread b WHERE b.NAME = 'white bread'
    - Error message "could not resolve property"

Operators for WHERE

- Comparing values

= <> > >= < <=

- Link multiple conditions

AND OR

- Use parentheses if necessary
- Mathematical operators

  + - * /

- [NOT] BETWEEN

  E.g.: WHERE b.price BETWEEN 1.15 AND 2.75

- [NOT] LIKE

  _ %

  E.g.: WHERE b.name LIKE '%tarwe%'

Operators for WHERE

- [NOT] IN
  - E.g.: WHERE b.name NOT IN (‘sugar bread', ‘raisin bread')
- IS [NOT] EMPTY
  - Do you use if one of the properties is a collection class eg: WHERE x.members IS EMPTY
- IS [NOT] NULL
  - E.g.: WHERE b.name is null
- NOT
  - To ignore a condition
- [NOT] EXISTS (subquery)
  - exists returns true if the subquery returns values
- {ALL|ANY|SOME} (subquery)
  - See below

JPQL functions

- CONCAT
- SUBSTRING
- TRIM
- LOWER
- UPPER
- LENGTH
- LOCATE

- ABS
- SQRT
- MOD
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

Find out more:
http://download.oracle.com/docs/cd/E14571_01/apirefs.1111/e13946/ejb3_overview_query.html

Aggregations

- COUNT returns a Long
- MAX, MIN
- AVG returns a Double
- SUM
  - returns a Double if you work with decimal numbers
  - returns a Long if you work with integers
- You can also use DISTINCT in combination with the previous

Even more

- ORDER BY
  - Standard ascending
  - E.g.: SELECT b FROM Bread b ORDER BY b.price desc, b.name asc
- GROUP BY
- HAVING

- Note: in where, order by or group by you may not use calculations
  - So NOT:
    ORDER BY SUM(leden.price) DESC
    WHERE AVG(leden.gewicht) > 65

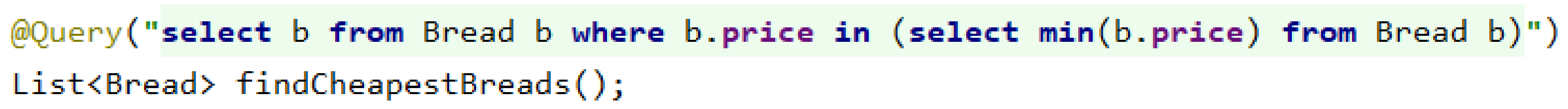
Use JPQL in the example project

- Open BreadRepository
- In the current version of your project, the keyword query was used:
  - `List<Bread> findAllByOrderByPriceAsc();`
- You didn't have to program anything for this
- You also get the same result if you solve this with JPQL.
  - Now delete the method above (or put it in comment) and add the following code in your BreadRepository. NB You can/may call the method differently what we have also done to prove this ... for example:

    -
      ```
      @Query("select b from Bread b order by b.price ASC")
      List<Bread> giveListOfAllBreadsOrderedByPrice();
      ```
  - After the @Query you still have to have the import done by IntelliJ
  - In the BreadController you must of course also replace every use of the findAllByOrderByPriceAsc() method with giveListOfAllBreadsOrderedByPrice()

Use JPQL in the example project

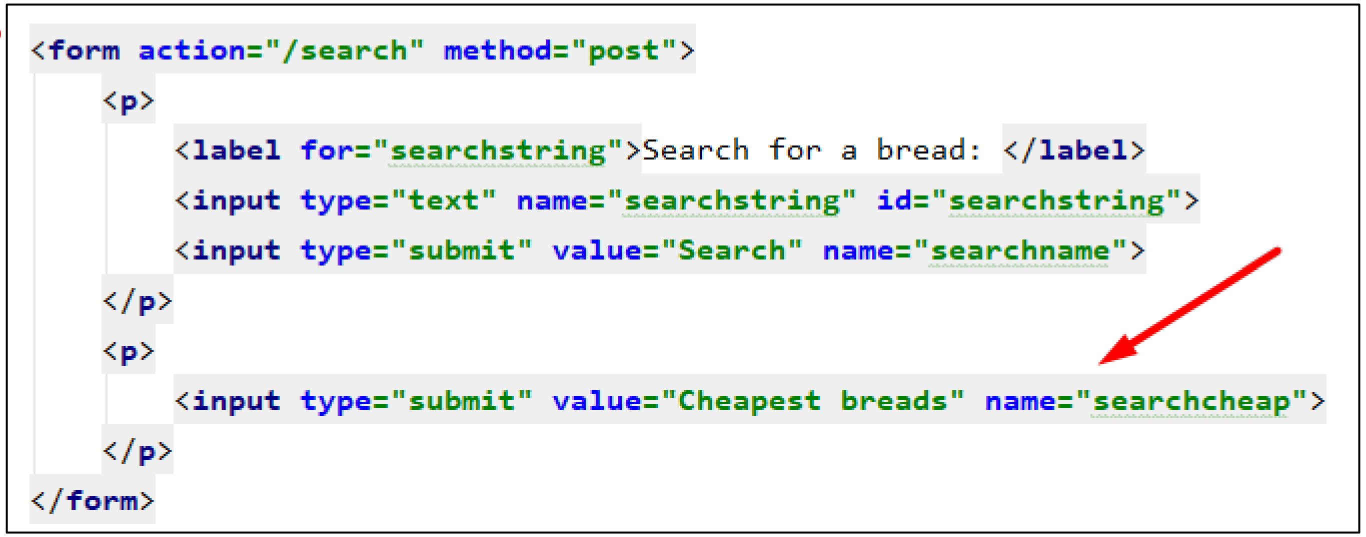
- We add another functionality:
- Suppose you want to get the cheapest Bread from the list. To solve this, you must first look for the lowest price `(select min(b.price) from Bread b).`
- Then you will select those breads whose price is equal to the lowest price you found via the previous query: `select b from Bread b where b.price in (select min(b.price) from Bread b)`
- We add this JPQL syntax in the BreadRepository as follows:



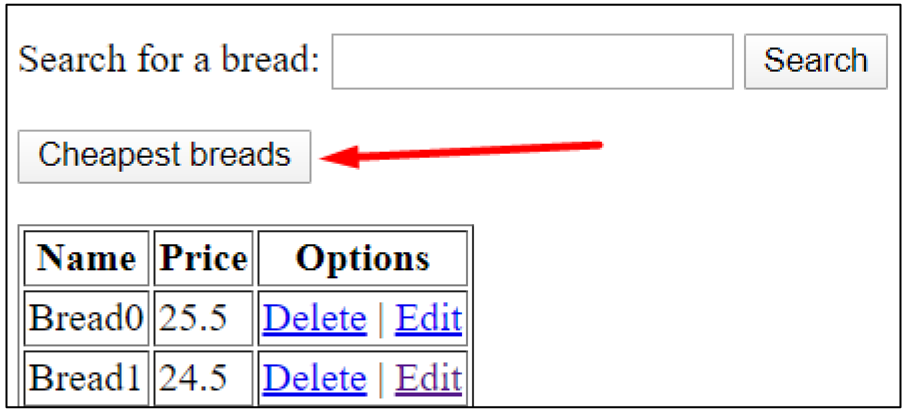
```
@Query("select b from Bread b where b.price in (select min(b.price) from Bread b)")
List<Bread> findCheapestBreads();
```

Use JPQL in the example project

- In the index we add a button at the top
-



```
<form action="/search" method="post">
    <p>
        <label for="searchstring">Search for a bread: </label>
        <input type="text" name="searchstring" id="searchstring">
        <input type="submit" value="Search" name="searchname">
    </p>
    <p>
        <input type="submit" value="Cheapest breads" name="searchcheap">
    </p>
</form>
```



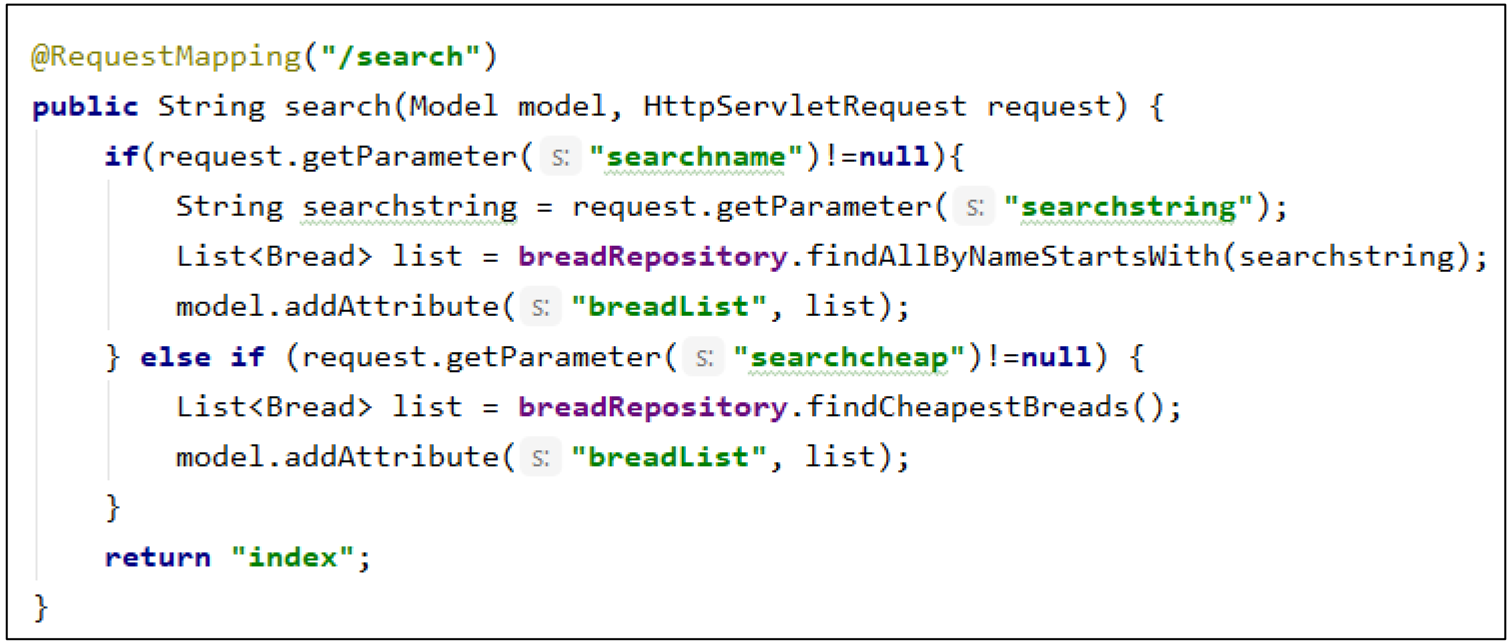
Search for a bread: [ ] Search

Cheapest breads

Name	Price	Options
Bread0	25.5	Delete | Edit
Bread1	24.5	Delete | Edit

Use JPQL in the example project

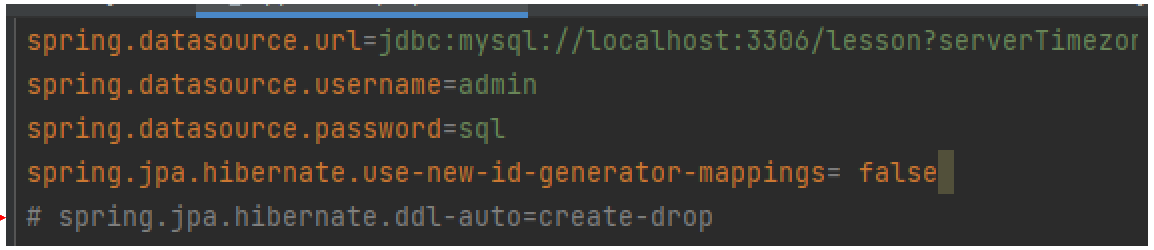
- in BreadController we modify the method that comes after the RequestMapping("/search")



```
@RequestMapping("/search")
public String search(Model model, HttpServletRequest request) {
    if(request.getParameter( s: "searchname")!=null){
        String searchstring = request.getParameter( s: "searchstring");
        List<Bread> list = breadRepository.findAllByNameStartsWith(searchstring);
        model.addAttribute( s: "breadList", list);
    } else if (request.getParameter( s: "searchcheap")!=null) {
        List<Bread> list = breadRepository.findCheapestBreads();
        model.addAttribute( s: "breadList", list);
    }
    return "index";
}
```

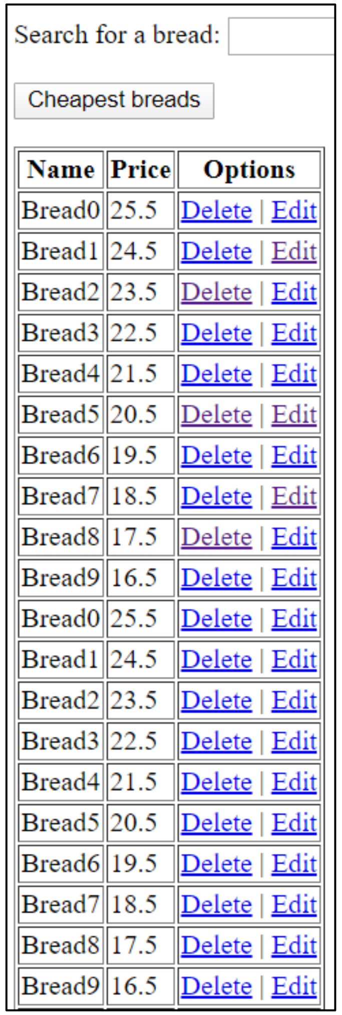
Use JPQL in the example project

- To try out the test of this method, we are going to put the following code in the application.properties-file in comments



```
spring.datasource.url=jdbc:mysql://localhost:3306/lesson?serverTimezor
spring.datasource.username=admin
spring.datasource.password=sql
spring.jpa.hibernate.use-new-id-generator-mappings= false
# spring.jpa.hibernate.ddl-auto=create-drop
```

- Every time you run your application now, 10 new records will be added to your table because we no longer start "from a clean slate" every time but simply supplement with what is in the @PostConstruct method ...
-



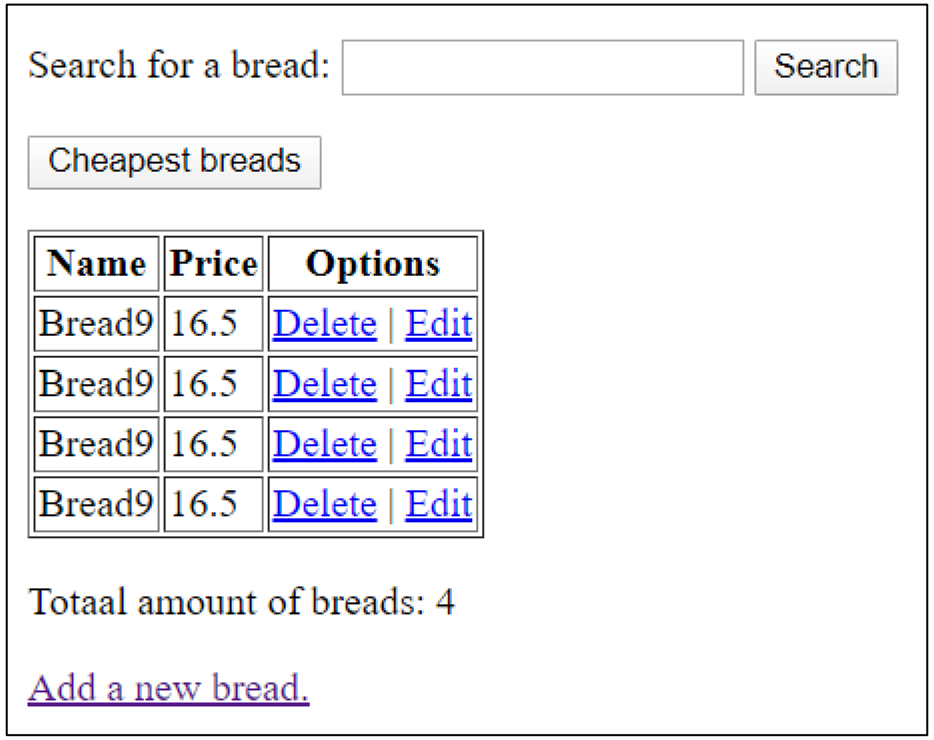
Search for a bread:

Cheapest breads

Name	Price	Options
Bread0	25.5	Delete | Edit
Bread1	24.5	Delete | Edit
Bread2	23.5	Delete | Edit
Bread3	22.5	Delete | Edit
Bread4	21.5	Delete | Edit
Bread5	20.5	Delete | Edit
Bread6	19.5	Delete | Edit
Bread7	18.5	Delete | Edit
Bread8	17.5	Delete | Edit
Bread9	16.5	Delete | Edit
Bread0	25.5	Delete | Edit
Bread1	24.5	Delete | Edit
Bread2	23.5	Delete | Edit
Bread3	22.5	Delete | Edit
Bread4	21.5	Delete | Edit
Bread5	20.5	Delete | Edit
Bread6	19.5	Delete | Edit
Bread7	18.5	Delete | Edit
Bread8	17.5	Delete | Edit
Bread9	16.5	Delete | Edit

Use JPQL in the example project

- After running the application again for 4 times, clicking on the "Cheapest breads" button will give the following result:


Search for a bread:

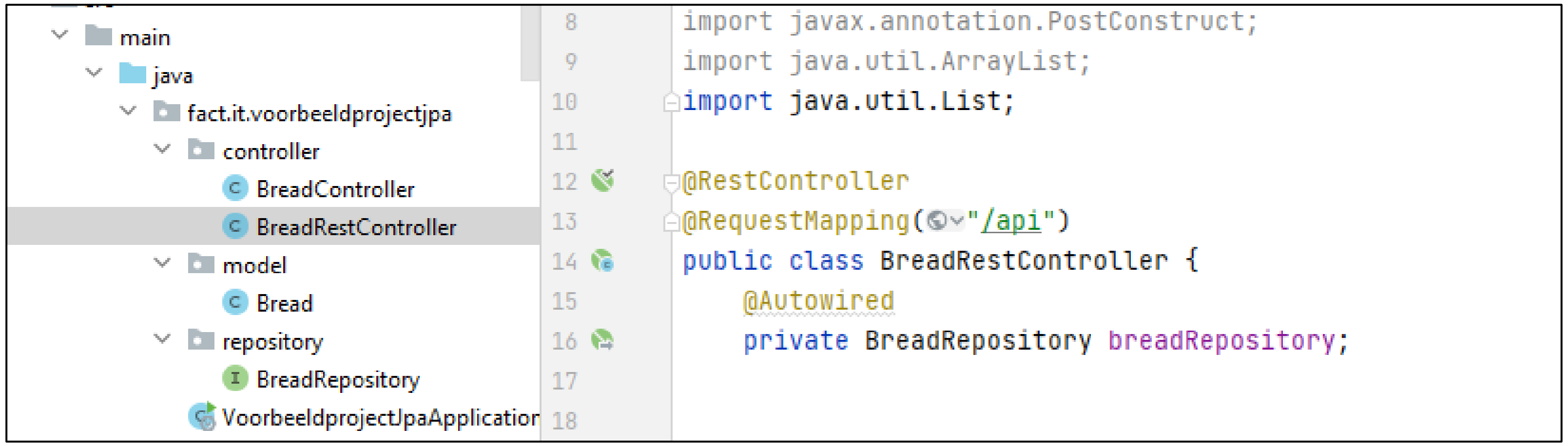
Name	Price	Options
Bread9	16.5	Delete | Edit
Bread9	16.5	Delete | Edit
Bread9	16.5	Delete | Edit
Bread9	16.5	Delete | Edit

Totaal amount of breads: 4

Add a new bread.

Exposing the API

- The same "backend" that you have created can now also be offered in an easy way to other frontend applications via an API.
- Create a RestController for that (in the “controller” package)
- Also make an association with your BreadRepository class
  - you can "autowire" it or have it instantiated via the constructor with a parameter



```
main
  java
    fact.it.voorbeeldprojectjpa
      controller
        BreadController
        BreadRestController
      model
        Bread
      repository
        BreadRepository
      VoorbeeldprojectJpaApplicatior
```

```java
import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/api")
public class BreadRestController {
    @Autowired
    private BreadRepository breadRepository;


```

Exposing API

- The services that you can easily offer are:
  - a service to retrieve all bread records from the database, sorted by price
  - a service to retrieve all bread records from the database where the name starts with a particular name part
  - a service to retrieve the cheapest bread records from the database
  - a service to modify a bread record in the database. This service returns the custom bread object
  - a service to remove a bread record from the database. This service does not give anything back

Exposing API: BreadRestController

```
//a service to retrieve all bread records from the database, sorted by price
@GetMapping("/breads")
public List<Bread> getBreads(){
    return breadRepository.giveListOfAllBreadsOrderedByPrice();
}

// a service to query all bread records from the database where the name starts with a certain
letter combination
@GetMapping("/breads/search")
public List<Bread> getBreadsNamePart(@RequestBody String namePart){
    return breadRepository.findAllByNameStartsWith(namePart);
}

//a service to find the cheapest bread
@GetMapping("/breads/searchcheapest")
public List<Bread> getBreadsCheapest(){
    return breadRepository.findCheapestBreads();
}
```

Exposing API: BreadRestController

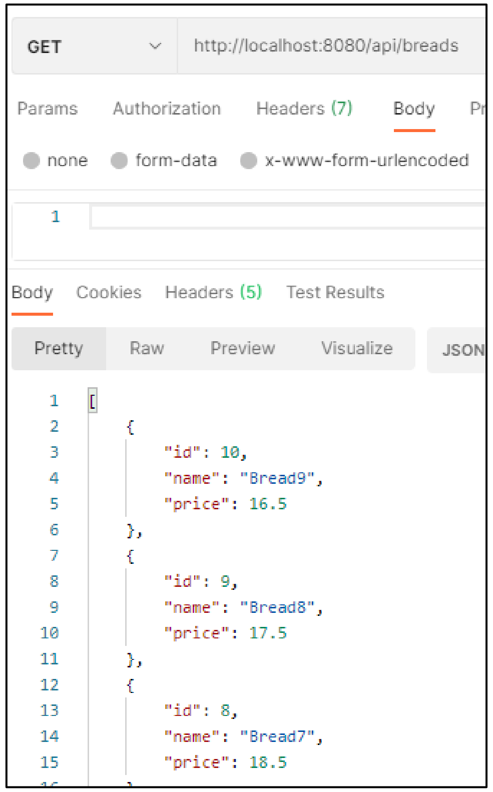
```
//a service to add a bread record in the database. This service also returns the added bread object
@PostMapping("/breads")
public Bread createBread(@RequestBody Bread bread){
    return breadRepository.save(bread);
}

//a service to modify a bread record in the database. This service  returns the modified bread-object if found, if not, it returns HTTP 4
@PutMapping("/breads/{id}")
public ResponseEntity<Bread> changeBread(@RequestBody Bread updateBread, @PathVariable long id){
    Optional<Bread> bread1 = breadRepository.findById(id);
    if (bread1.isPresent()) {
        Bread bread=bread1.get();
        bread.setName(updateBread.getName());
        bread.setPrice(updateBread.getPrice());
        breadRepository.save(bread);
        return new ResponseEntity<>(bread, HttpStatus.OK);
    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

Exposing API: BreadRestController

```
//a service to remove a bread record from the database. This service returns the number of breads in the database
@DeleteMapping("/breads/{id}")
public ResponseEntity<Integer> deleteBread(@PathVariable long id) {
    Optional<Bread> bread1 = breadRepository.findById(id);
    if (bread1.isPresent()) {
        Bread bread = bread1.get();
        breadRepository.delete(bread);
        return new ResponseEntity<>(breadRepository.findAll().size(), HttpStatus.OK);
    }
    return new ResponseEntity<>(breadRepository.findAll().size(), HttpStatus.NOT_FOUND);
```

Exposing API: Test the API

- Use Postman to test all your services...
-



GET http://localhost:8080/api/breads

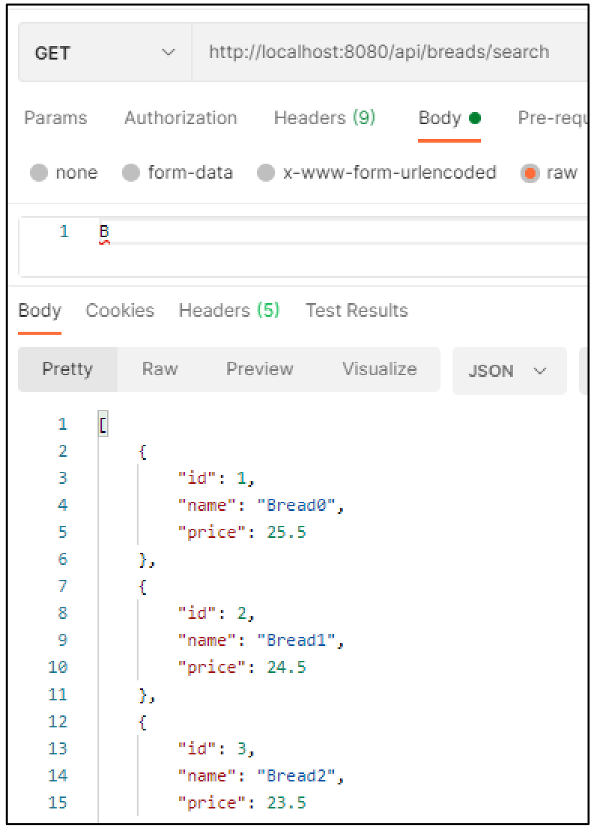
Params | Authorization | Headers (7) | Body | Pr

none | form-data | x-www-form-urlencoded

Body | Cookies | Headers (5) | Test Results

Pretty | Raw | Preview | Visualize | JSON

```
[
    {
        "id": 10,
        "name": "Bread9",
        "price": 16.5
    },
    {
        "id": 9,
        "name": "Bread8",
        "price": 17.5
    },
    {
        "id": 8,
        "name": "Bread7",
        "price": 18.5
```



GET http://localhost:8080/api/breads/search

Params | Authorization | Headers (9) | Body ● | Pre-requ

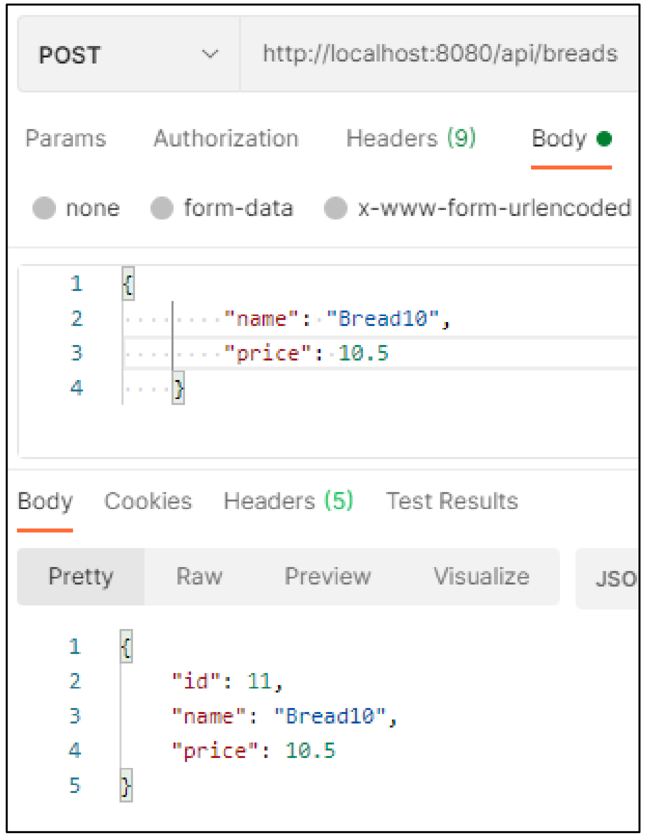
none | form-data | x-www-form-urlencoded | raw

```
B
```

Body | Cookies | Headers (5) | Test Results

Pretty | Raw | Preview | Visualize | JSON

```
[
    {
        "id": 1,
        "name": "Bread0",
        "price": 25.5
    },
    {
        "id": 2,
        "name": "Bread1",
        "price": 24.5
    },
    {
        "id": 3,
        "name": "Bread2",
        "price": 23.5
```



POST http://localhost:8080/api/breads

Params | Authorization | Headers (9) | Body ●

none | form-data | x-www-form-urlencoded

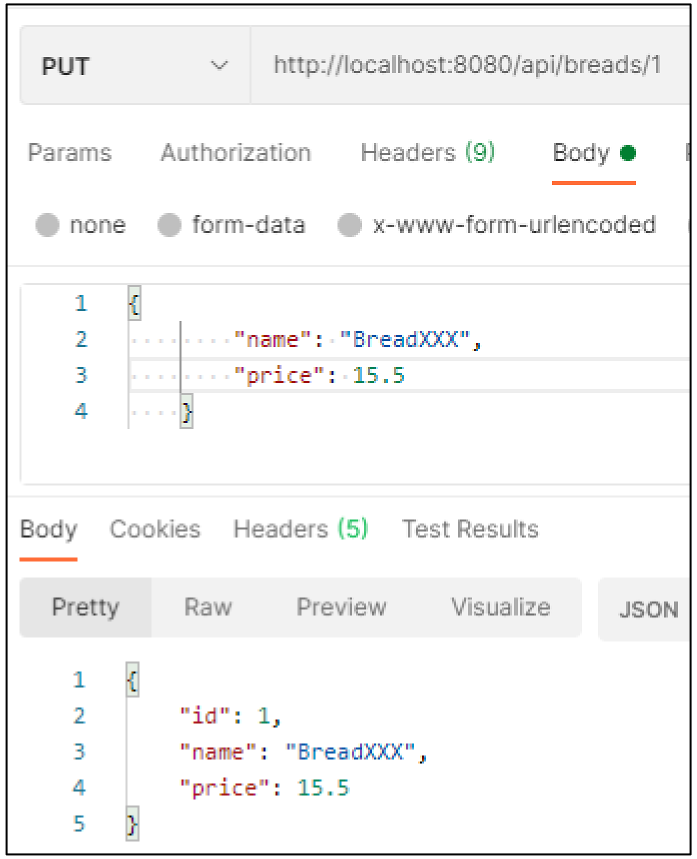
```
{
    "name": "Bread10",
    "price": 10.5
}
```

Body | Cookies | Headers (5) | Test Results

Pretty | Raw | Preview | Visualize | JSO

```
{
    "id": 11,
    "name": "Bread10",
    "price": 10.5
}
```

Exposing API: Test the API

- Use Postman to try out all your services...
- 

PUT http://localhost:8080/api/breads/1

Params Authorization Headers (9) Body ●

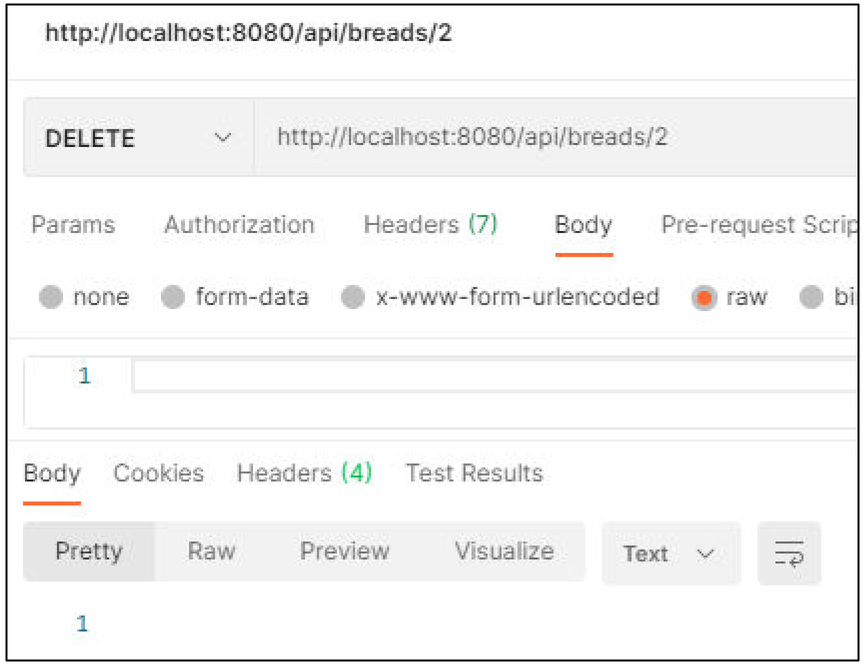
none form-data x-www-form-urlencoded

```
{
    "name": "BreadXXX",
    "price": 15.5
}
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "id": 1,
    "name": "BreadXXX",
    "price": 15.5
}
```



http://localhost:8080/api/breads/2

DELETE http://localhost:8080/api/breads/2

Params Authorization Headers (7) Body Pre-request Scrip

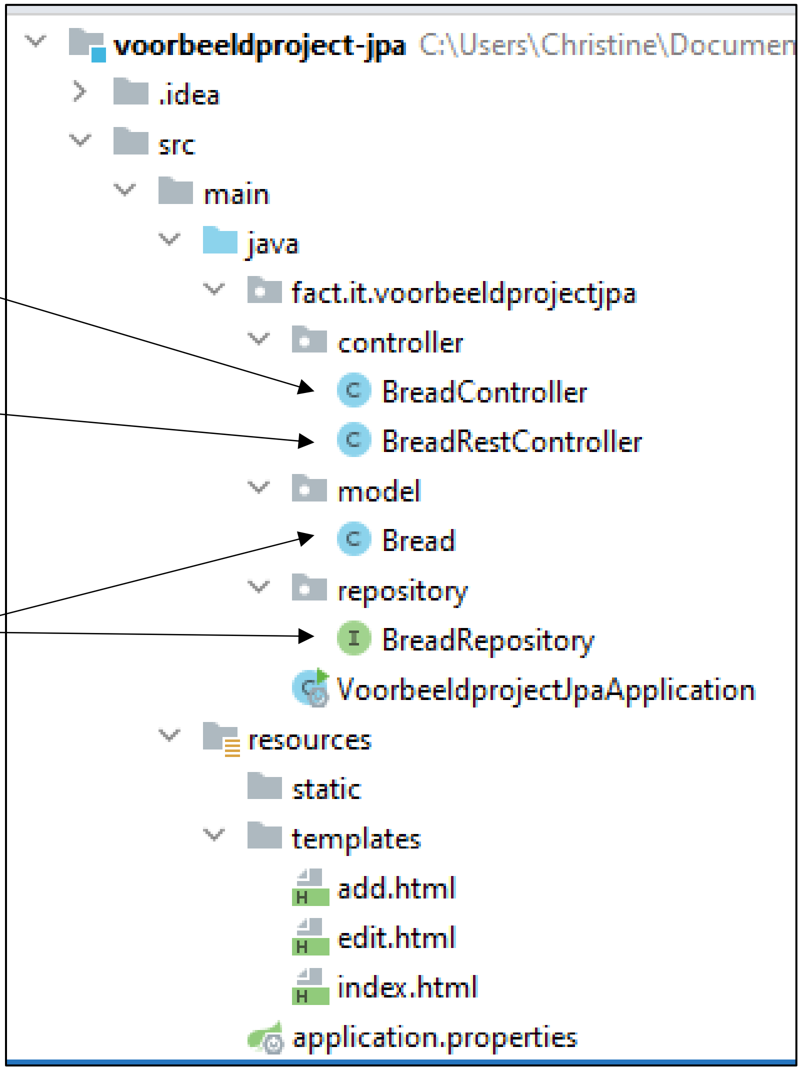
none form-data x-www-form-urlencoded raw bi

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize Text

Summarized

- The **BreadController** performs the same function as before: capturing html requests but now includes a "BreadRepository breadRepository" attribute to retrieve and write data
- The **BreadRestController** captures all requests from other frontend applications and provides answers/results in the form of a JSON or an XML.
- The **BreadRepository** provides access to the database. This allows you to use standard methods to perform CRUD operations or create new query methods using keyword queries or JPQL
- An **entity** object corresponds to a row in a table of the database. The attributes of an entity correspond to the columns of the table.



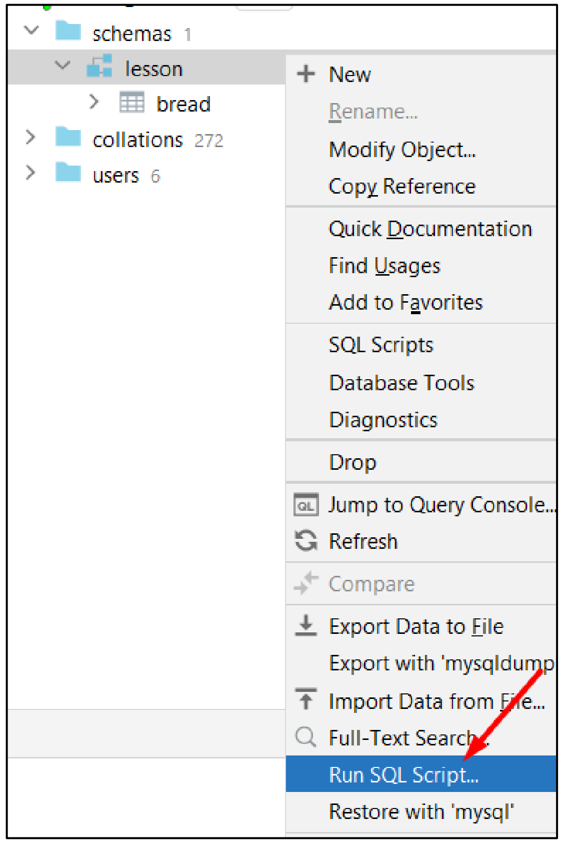
```
voorbeeldproject-jpa C:\Users\Christine\Documen
  .idea
  src
    main
      java
        fact.it.voorbeeldprojectjpa
          controller
            BreadController
            BreadRestController
          model
            Bread
          repository
            BreadRepository
          VoorbeeldprojectJpaApplication
      resources
        static
        templates
          add.html
          edit.html
          index.html
        application.properties
```

Testing the API via Postman testing script

- If you've successfully tested all your services through Postman, you can also run the Postman test script we've put in place (lesson_bread.sql). All the tests that are included in this should therefore be successful.
  - On the exam we will also make such a script available for verification...
- Necessary preparation:
  1. Run sql script on the database
  2. put the @Postconstruct method that serves to populate the database in comments and modify the application.properties file
  3. Load and run the Postman test script (Test breads.postman_collection.json) in Postman

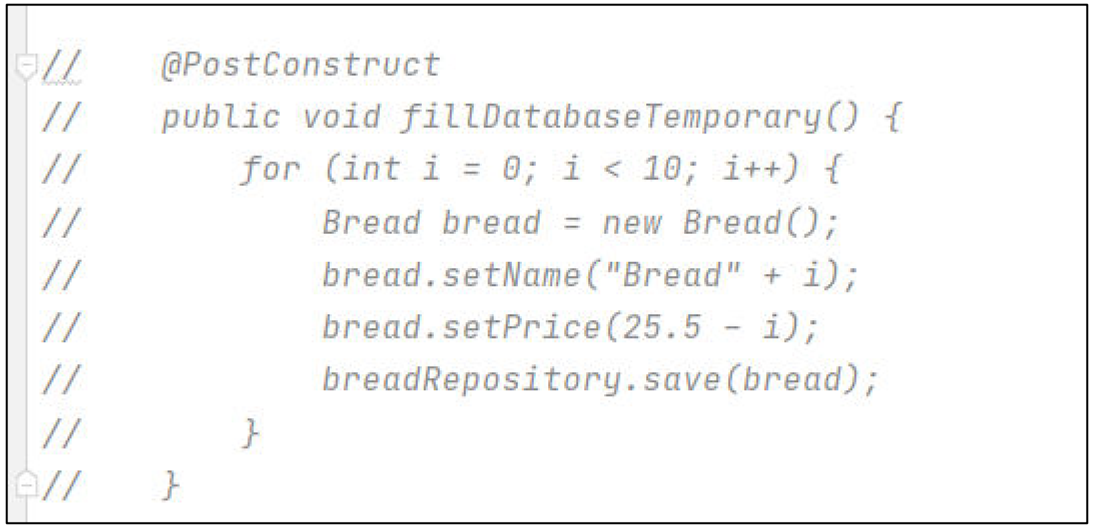
Run SQL script

- you can easily run the sql script via IntelliJ
- right-click on the database "lesson" and choose "Run SQL Script..."
  - then click and run the downloaded script lesson_bread.sql



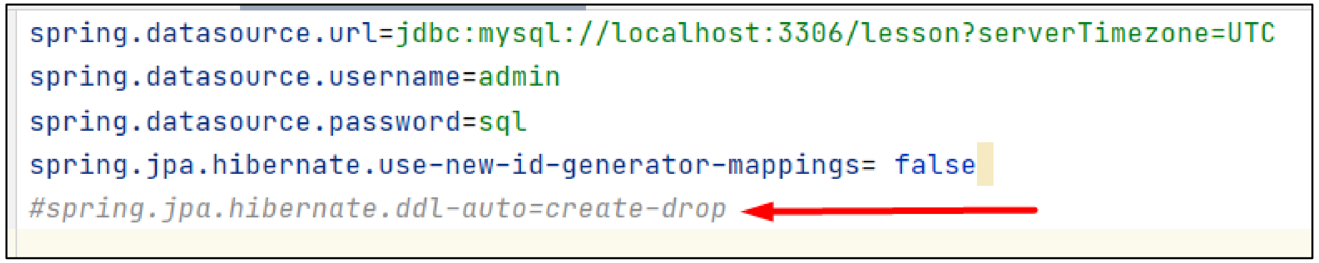
@PostConstruct method and properties-file

- In @Controller-class:



```
//    @PostConstruct
//    public void fillDatabaseTemporary() {
//        for (int i = 0; i < 10; i++) {
//            Bread bread = new Bread();
//            bread.setName("Bread" + i);
//            bread.setPrice(25.5 - i);
//            breadRepository.save(bread);
//        }
//    }
```

- In application.properties-file:

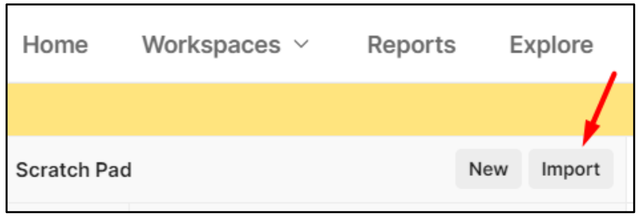


```
spring.datasource.url=jdbc:mysql://localhost:3306/lesson?serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=sql
spring.jpa.hibernate.use-new-id-generator-mappings= false
#spring.jpa.hibernate.ddl-auto=create-drop
```

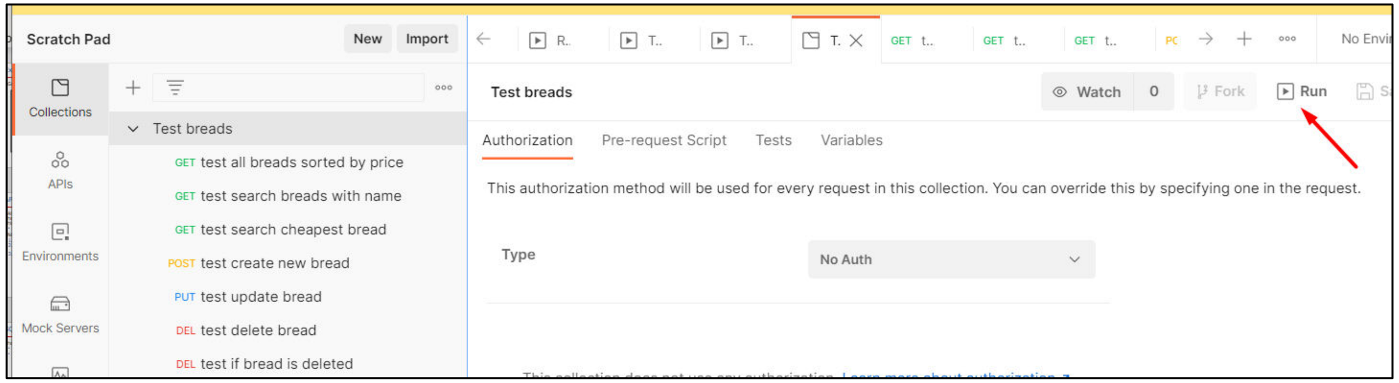
  - at the start of your application, nothing will be changed in the database
- **Now you run your project**

Load and run the Postman test script

- Download *Test breads.postman_collection.json*
- Start Postman and import this file via "import" (top left of your screen)



- Then click "Run":

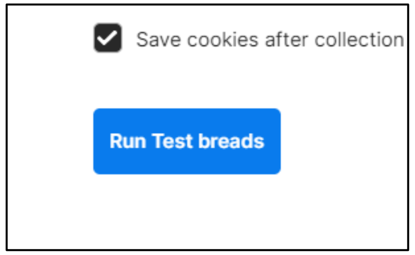


Test breads
- GET test all breads sorted by price
- GET test search breads with name
- GET test search cheapest bread
- POST test create new bread
- PUT test update bread
- DEL test delete bread
- DEL test if bread is deleted

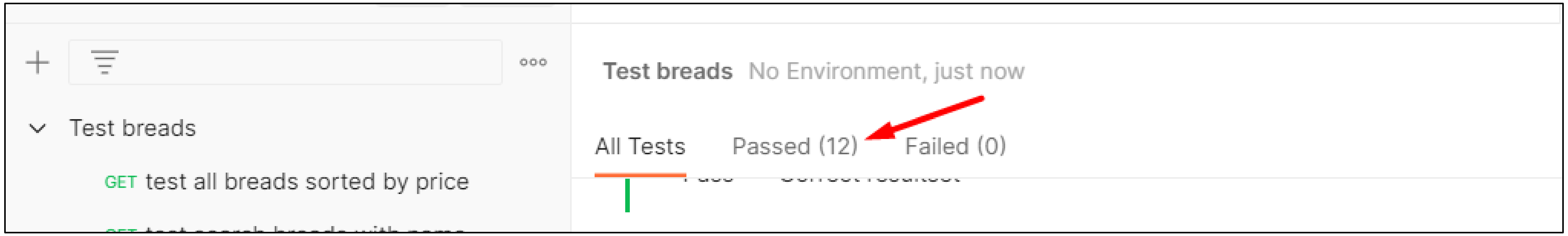
This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

Type: No Auth

- And finally, once again on



Result...



Test breads

GET test all breads sorted by price

Test breads No Environment, just now

- If you want to test the script again, you have to run the SQL script again...

More about how the Spring framework works

- Spring Component Scanning
- Inversion of Control IoC
- Dependency Injection
- see also "Explanation Spring Component Scan IoC DI.pdf" on Canvas

Spring Component Scanning

- At the start of the application, Spring framework scans your code for @-annotations and registers them. The @Component annotations and the annotations derived from them such as @Controller and @RestController and @Repository, will be detected.
- In a @Controller or @RestController, the scan will also register which @... Mappings are all there, and then register them as connected to that particular controller they're in.
- In the end, the framework has a complete register of all @Controller, @RestController, @Repository, ... classes etc. in your project.

Inversion of Control (IoC)

- For all classes or interfaces with @Component or with annotations derived from them such as @Controller, @RestController and @Repository Spring will perform **Inversion of Control**.
- In the past (without a framework) we always had control over the creation and management of instances of classes by creating instances/objects ourselves via **new** RecordController().
- Spring will **take that control out of your hands** and create one instance for the classes or interfaces with the above annotations. In other words:
    - At the startup (in the background) of your program, Spring will create **one instance**/object of each Component, Controller, RestController, Repository-, ...-class and put it **in the Spring Container**. While running the application, at any time the application needs the item, it will offer it
- Because only one instance of Components, Controllers and Repositories is created each time, these are **Singletons**.

Dependency Injection (DI)

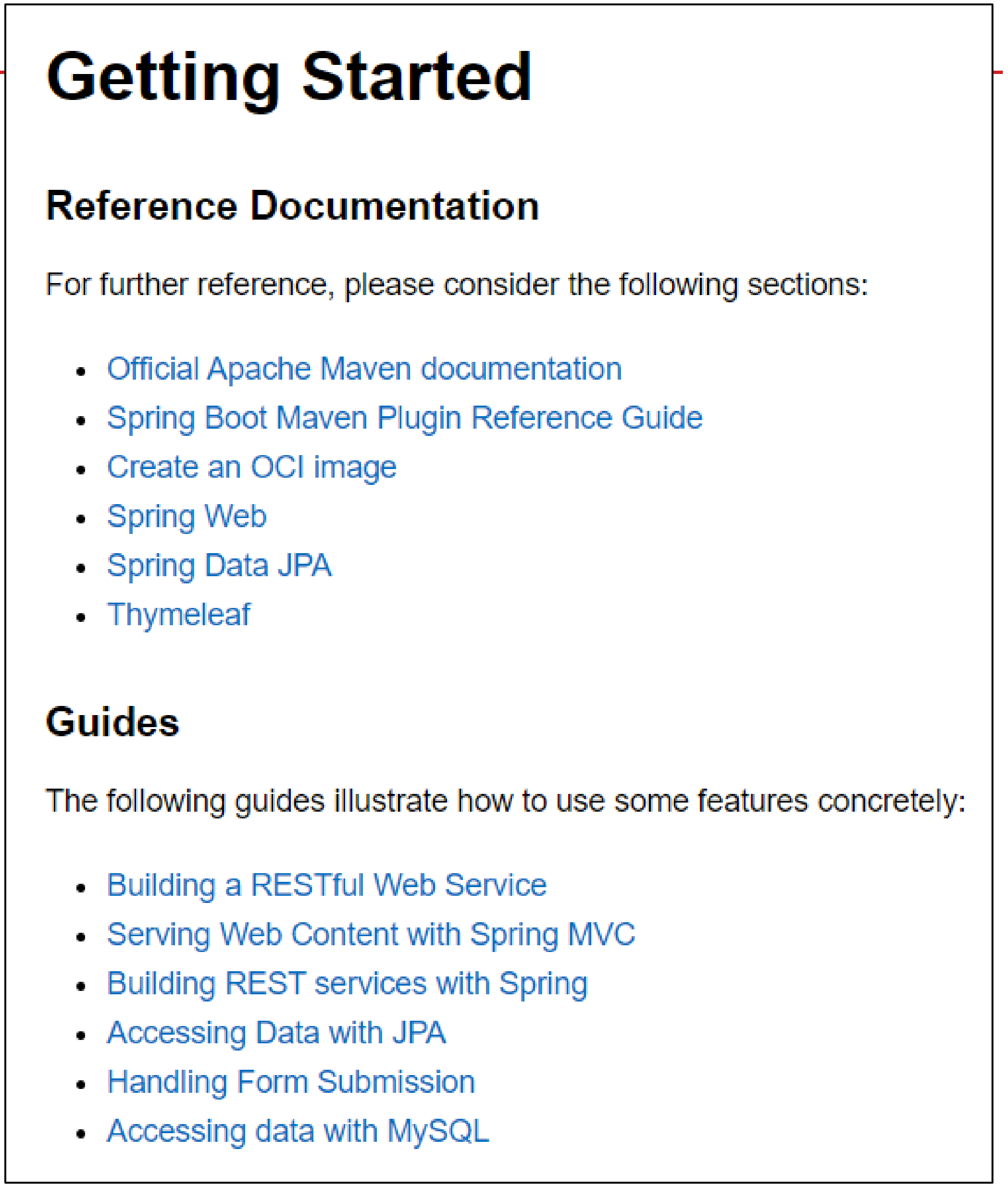
- In a Controller class that needs a Repository class, we do not create an object of this Repository class ourselves (e.g. with **new** RecordRepository().) It is much more efficient to use the one instance of RecordRepository that is in the Spring Container when your application is loaded...
- We can do this in several ways
  - 1 way is by using the Constructor to specify an instance of a Repository Class via a parameter
  - an other way is by putting the annotation @Autowired above the attribute (that of a class that contains the annotation @Component or one of the derived annotations such as @Controller and @Repository)
- In other words, we do not handle this **dependency** (= association between Controller class and RepositoryClass) by creating an instance in the class itself, but by **injecting** an instance into the class from the outside. Hence the name **Dependency Injection**

Dependency injection

- The video below explains very clearly what the term "Dependency injection" means in general outside the context of Spring Boot
  - Dependency injection - in general
- The video below explains how to do dependency injection in Spring Boot
  - Dependency injection – in spring boot

Documentation

- See HELP.md in your project:



Getting Started

Reference Documentation

For further reference, please consider the following sections:

- Official Apache Maven documentation
- Spring Boot Maven Plugin Reference Guide
- Create an OCI image
- Spring Web
- Spring Data JPA
- Thymeleaf

Guides

The following guides illustrate how to use some features concretely:

- Building a RESTful Web Service
- Serving Web Content with Spring MVC
- Building REST services with Spring
- Accessing Data with JPA
- Handling Form Submission
- Accessing data with MySQL