Haaga-Helia

Server Programming

ORM/JPA, Databases

Minna Pellikka, Juha Hinkula and Jukka Juslin

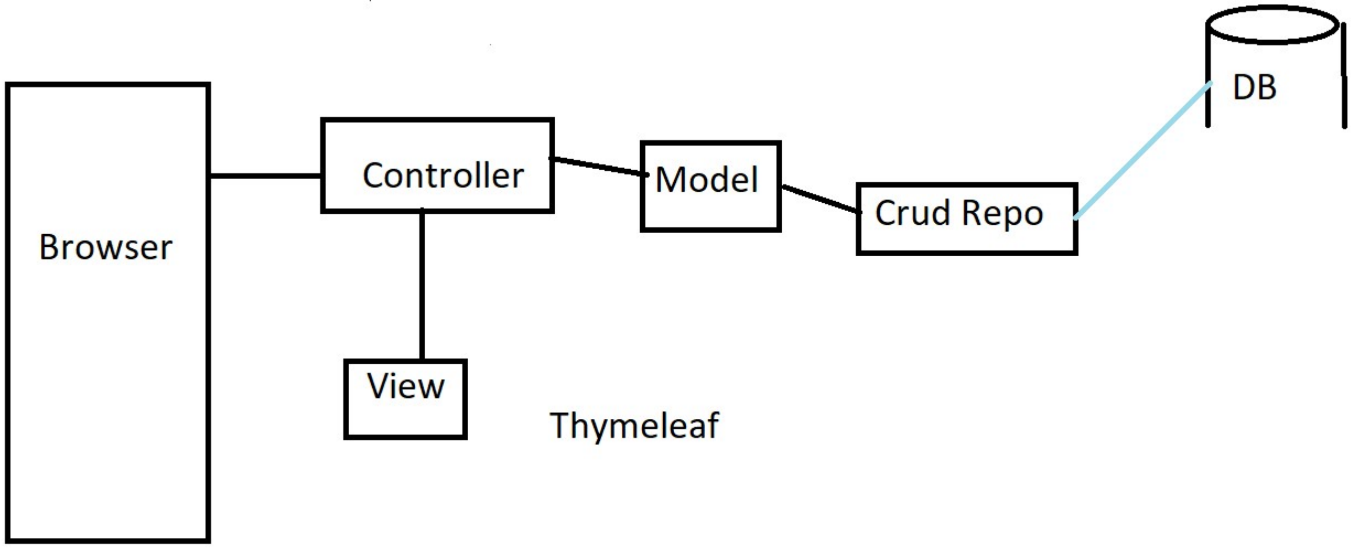
Haaga-Helia

Spring Boot: JPA

- JPA (Java Persistence API) is a collection of classes to persistently store data into a database
- JPA provides object – relational mapping for managing relational data in JAVA applications (ORM)
- There is lot of implementations of the JPA (like Hibernate)
- Dependency

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

Architecture of this lesson setup



Haaga-Helia

H2 database

- H2 is open source Java based SQL database
- www.h2database.com
- Embedded and server modes: in-memory databases
- Good database for prototyping, testing etc.
- Dependency

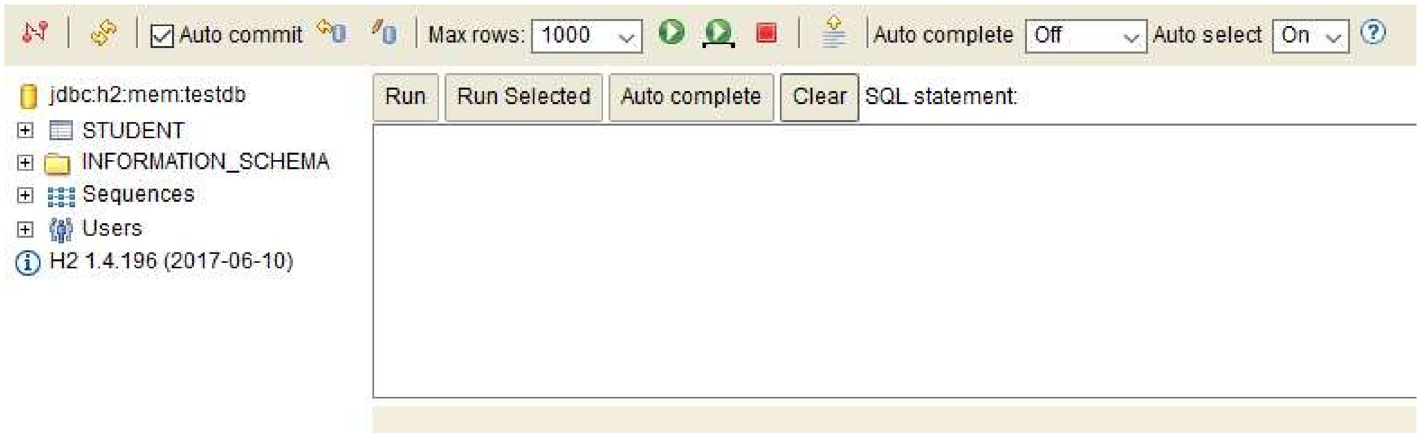
```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

H2 database

- H2 provides a web based console
- Add following lines to the *application.properties* file

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.datasource.url=jdbc:h2:mem:testdb
```

- Navigate to `localhost:8080/h2-console`
  - JDBC URL = `jdbc:h2:mem:testdb`
  - Leave password field empty



Spring Boot: JPA

- You can add following property to *application.properties* file. This enables the logging of SQL statements

```
spring.jpa.show-sql=true
```

Spring Boot: JPA

- ***Entity***
  - An entity represents a table in relational database
  - Entity class must be annotated with @Entity annotation (jakarta.persistence.Entity)
  - By default, the table name is the name of the entity class. It can be changed by using @Table annotation

```
import jakarta.persistence.Entity;

@Entity
public class Student {
        // More code..
```

Haaga-Helia

Spring Boot: JPA

- ***@Id*** annotation is used for creating id column of the table
- ***@GeneratedValue*** annotation generates automatically a unique primary key for every new entity object (GenerationType.Auto)

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

Spring Boot: JPA

- The other properties can be left unannotated. Then these properites are mapped to columns that share the same name as properties itself
- ***@Column*** annotation can be used to specify mapped column. Example: `@Column(name=”address”)`

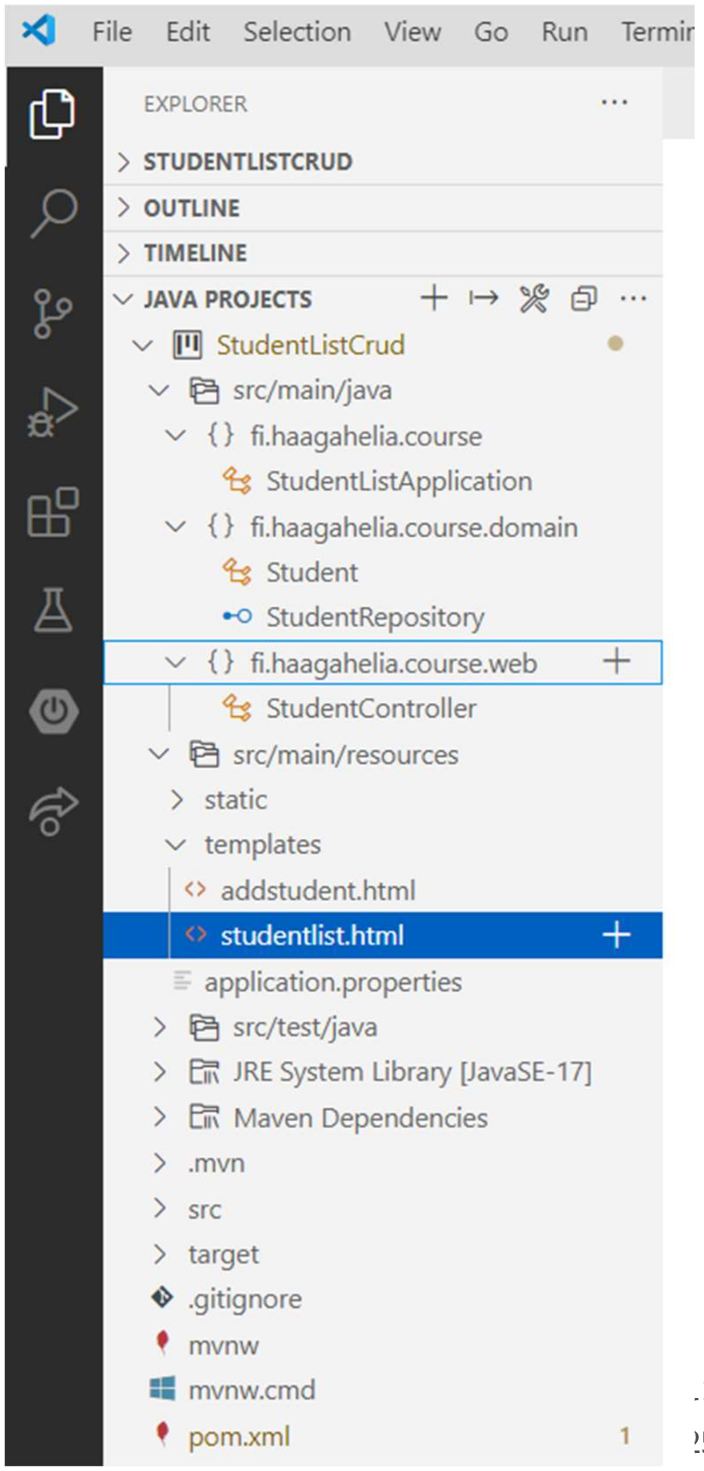
```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String firstName, lastName, email;
...getters and setters
```

Spring Boot: JPA

```
// Example Entity
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
      private Long id;
      private String firstName, lastName, email;

      public Student() {}

      public Student(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }

    @Override
    public String toString() {
        return "Student id=" + id + ", firstName=" + firstName + ",lastName=" + lastName;
       }
}
```

Haaga-Helia

Wanted directory structure


File Edit Selection View Go Run Termir
EXPLORER
> STUDENTLISTCRUD
> OUTLINE
> TIMELINE
∨ JAVA PROJECTS
∨ StudentListCrud
  ∨ src/main/java
    ∨ {} fi.haagahelia.course
      StudentListApplication
    ∨ {} fi.haagahelia.course.domain
      Student
      StudentRepository
    ∨ {} fi.haagahelia.course.web
      StudentController
  ∨ src/main/resources
    > static
    ∨ templates
      addstudent.html
      studentlist.html
    application.properties
  > src/test/java
  > JRE System Library [JavaSE-17]
  > Maven Dependencies
  > .mvn
  > src
  > target
  .gitignore
  mvnw
  mvnw.cmd
  pom.xml 1

Spring Boot: JPA

- The ***CrudRepository*** provides CRUD functionality for the entity class that is being managed.
- How to use repositories
  1. Declare an interface extending Repository
  2. Declare query methods on the interface
  3. Get the repository instance injected and use it

Spring Boot: JPA

1.) Declare an interface extending Repository (Create a new class which extends CrudRepository)

By extending *CrudRepository* the StudentRepository inherits methods for working with Student persistence, including methods for saving, deleting, and finding Student entities.

```
import org.springframework.data.repository.CrudRepository;

public interface StudentRepository extends CrudRepository<Student, Long>
{
        ...
}
```

Haaga-Helia

Spring Boot: JPA

2.) Declare query methods on the interface

Note! In typical Java application you should write an class that implements StudentRepository. Spring Data JPA creates this automatically when you run the application.

```
import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface StudentRepository extends CrudRepository<Student, Long> {
    List<Student> findByLastName(String lastName);
}
```

Spring Boot: JPA

3.) Get the repository instance injected and use it

Constructor Injection annotation bring repository class into the context, and will inject an instance of the service into the YourAppClass class. Works only if there is only one constructor! Otherwise you need to use @Autowired annotation

```java
@Controller
public class StudentController {
    private StudentRepository repository;
    // constructor injection. Can only be one constructor then.
    public StudentController(StudentRepository repository) {
        this.repository = repository;

    }
    @RequestMapping(value= {"/", "/studentlist"})
    public String studentList(Model model) {
        model.addAttribute("students", repository.findAll());
        return "studentlist";
    }
}
```

Spring Boot: JPA

- Some useful CrudRepository methods

`long count()`	Returns the number of entities available.
`void deleteById(ID id)`	Deletes the entity with the given id
`void delete(T Entity)`	Deletes given entity
`deleteAll()`	Deletes all entities managed by the repository
`Iterable<T> findAll()`	Returns all entities
`Optional<T> findById(ID id)`	Retrieves an entity by its id
`<S extends T> S  save(S entity)`	Saves a given entity

Spring Boot: CommandLineRunner

- If you need to run some specific code when the SpringApplication has started, you can implement the `CommandLineRunner` interfaces. This is good place to add some demo data to your apllication

```
…
@Bean
public CommandLineRunner demo(StudentRepository repository) {
        return (args) -> {
          // Your code...add some demo data to db
        };
}
…
```

Haaga-Helia

Spring Boot: JPA

- Adding List page to a Spring Boot application
  - 1.) Create template for list page (studentlist.html).

```
<table class="table table-striped">
  <tr>
   <th>Name</th>
   <th>Email</th>
  </tr>
     <tr th:each = "student : ${students}">
       <td th:text="${student.firstName} + ' ' + ${student.lastName}"></td>
       <td th:text="${student.email}"></td>
     </tr>
</table>
…
```

Haaga-Helia

Spring Boot: JPA

2.) Create method to your controller. All students are fetched from the database and added to the model attribute.

```
@Autowired
private StudentRepository repository;


…


@RequestMapping(value="/studentlist")
public String studentList(Model model) {
  model.addAttribute("students", repository.findAll());
  return "studentlist";
}
…
```

Haaga-Helia

Spring Boot: JPA

- Adding Create funtionality to a Spring Boot application

  1.) Create template for adding new entity (in this example addstudent.html). Download source code from the course site.

```
…
<h1>Add student</h1>
<div>
  <form th:object="${student}" th:action="@{save}" action="#" method="post">
     <label for="fname">Firstname</label>
     <input type="text" id="fname" th:field="*{firstName}" />
     <label for="lname">Lastname</label>
     <input type="text" id="lname" th:field="*{lastName}" />
     <label for="email">Email</label>
     <input type="text" th:field="*{email}" />
     <input type="submit" value="Save"></input>
  </form>
</div>
…
```

Haaga-Helia

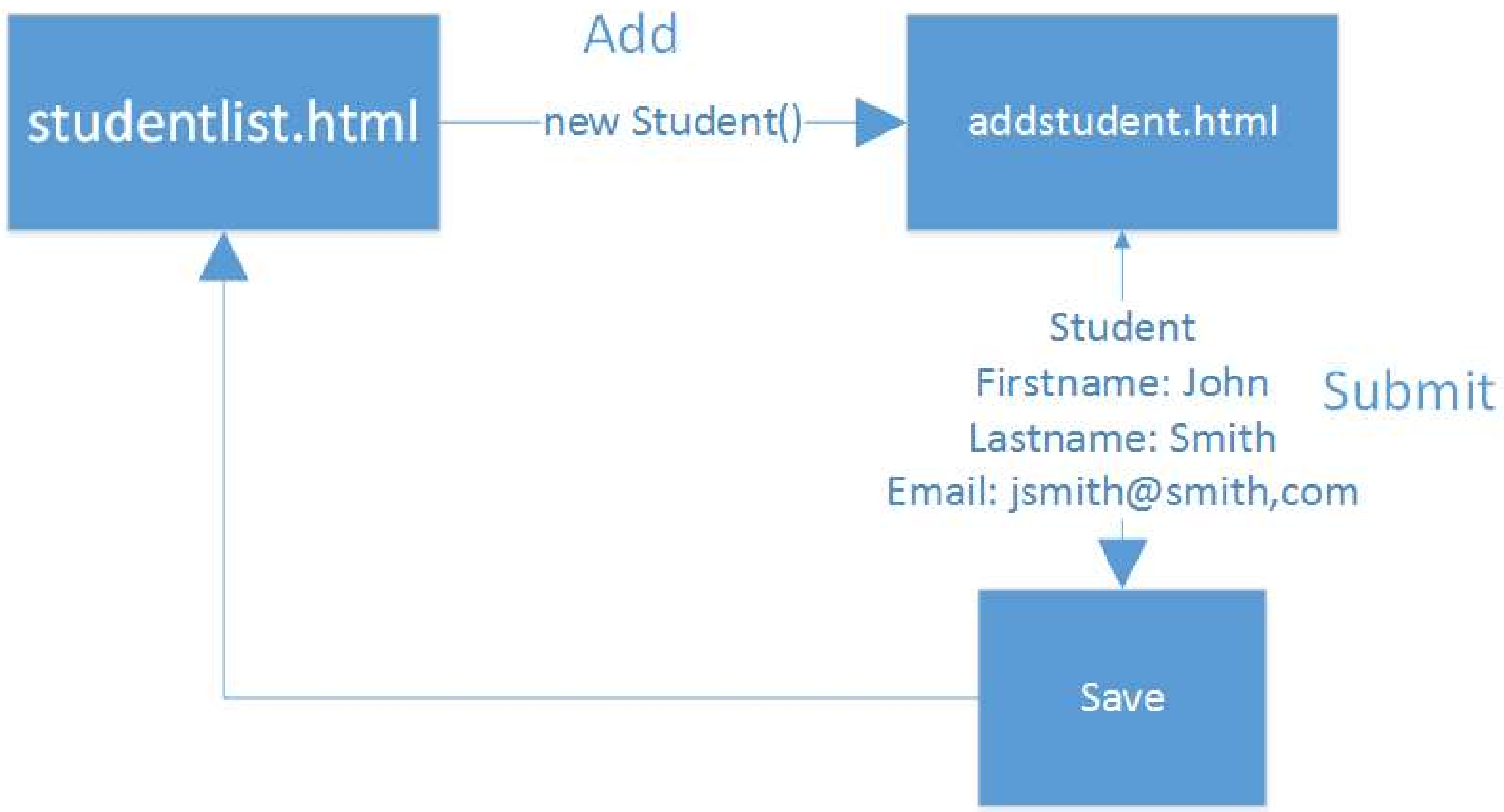
Spring Boot: JPA

2.) Create functionality to your controller

```
…
@RequestMapping(value = "/add")
public String addStudent(Model model){
    model.addAttribute("student", new Student());
    return "addstudent";
}

@RequestMapping(value = "/save", method = RequestMethod.POST)
public String save(Student student){
     repository.save(student);
     return "redirect:studentlist";
}
…
```

3.) Add link to create functionality

```
<a href="/add" class="btn btn-primary">Add Student</a>
```


Add
studentlist.html
new Student()
addstudent.html
Student
Firstname: John
Lastname: Smith
Email: jsmith@smith,com
Submit
Save

Haaga-Helia

Spring Boot: JPA

- Adding Delete funtionality to a Spring Boot application
  1.) Create functionality to the controller

```
…
@RequestMapping(value = "/delete/{id}", method = RequestMethod.GET)
public String deleteStudent(@PathVariable("id") Long studentId, Model model) {
   repository.deleteById(studentId);
   return "redirect:../studentlist";
}
…
```

- deleteStudent method listens */delete/studentid* endpoint
- By using `@PathVariable` annotation Spring extracts id from the URI
- For example, request http://*localhost:8080/delete/5*, the @PathVariable studentId will be 5.

Spring Boot: JPA

2.) Add link to delete functionality (for example in the listpage row)

```
…
<a th:href="@{/delete/{id}(id=${student.id})}">Delete</a>
…
```

Haaga-Helia

Spring Boot: CrudRepository

- *Query methods*: CrudRepository can derive the query from the method name
- Examples:

```
public interface StudentRepository extends CrudRepository<Student, Long> {
    List<Student> findByLastName(String lastName);

    List<Student> findByFirstNameAndLastName(String firstName, String lastName);

     // Enabling ignoring case
     List<Student> findByLastNameIgnoreCase(String lastName);


     // Enabling ORDER BY for a query
     List<Student> findByLastNameOrderByFirstNameAsc(String lastName);
}
```

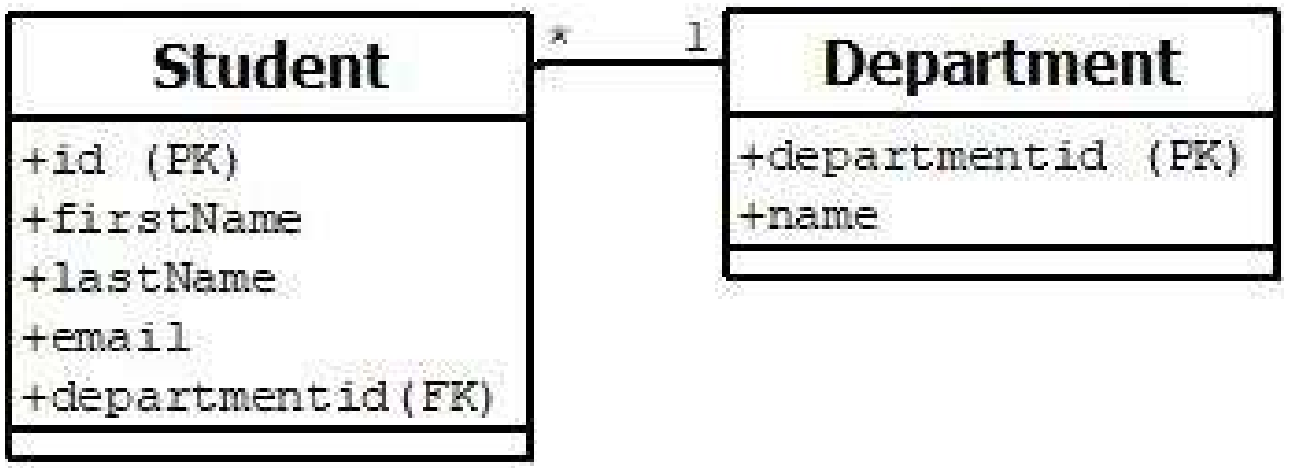
Spring Boot: JPA

- Relationships with JPA
- One-to-Many
  - @OneToMany and @ManyToOne annotations defines a one-to-many and many-to-one relationship between two entities
  - @JoinColumn annotation defines the owner of the relationship

  (Table has a column with a foreign key to the referenced table)

Spring Boot: JPA

- Relationships with JPA
- One-to-Many



Spring Boot: JPA

- Student entity
  - Add new department attribute with @ManyToOne and @JoinColumn annotations

```
@ManyToOne
@JoinColumn(name = "departmentid")
private Department department;
```

- Add getters and setters for department
- Add department to constructor

Spring Boot: JPA

- Department entity
  - Add new students attribute with @OneToMany annotation

```
@OneToMany(cascade = CascadeType.ALL,mappedBy =
  "department”)
private List<Student> students;
```

- Add getters and setters

Spring Boot: JPA

- Add CrudRepository for Department entity

```
import org.springframework.data.repository.CrudRepository;

public interface DepartmentRepository extends CrudRepository<Department, Long> {
            ...
}
```

Haaga-Helia

Spring Boot: JPA

- Add department dropdown list to student form
  - You have to add new model attribute to controller method which shows student creation form. You also have to inject department repository to controller

```
@Autowired
private StudentRepository repository;
@Autowired
private DepartmentRepository drepository;

// Add new student
@RequestMapping(value = "/add")
public String addStudent(Model model){
  model.addAttribute("student", new Student());
  model.addAttribute("departments", drepository.findAll());
  return "addstudent";
}
```

Haaga-Helia

Spring Boot: JPA

- Add department dropdown list to student form
  - Departments can be now get from the model attribute in the template (departments attribute)
  - Select element shows department names (th:text) but the value will be departmentid (th:value)

```
<label for="deplist">Department</label>
<select id="deplist" th:field="*{department}" class="form-control">
  <option th:each="department: ${departments}"
            th:value="${department.departmentid}"
              th:text="${department.name}"></option>
</select>
```

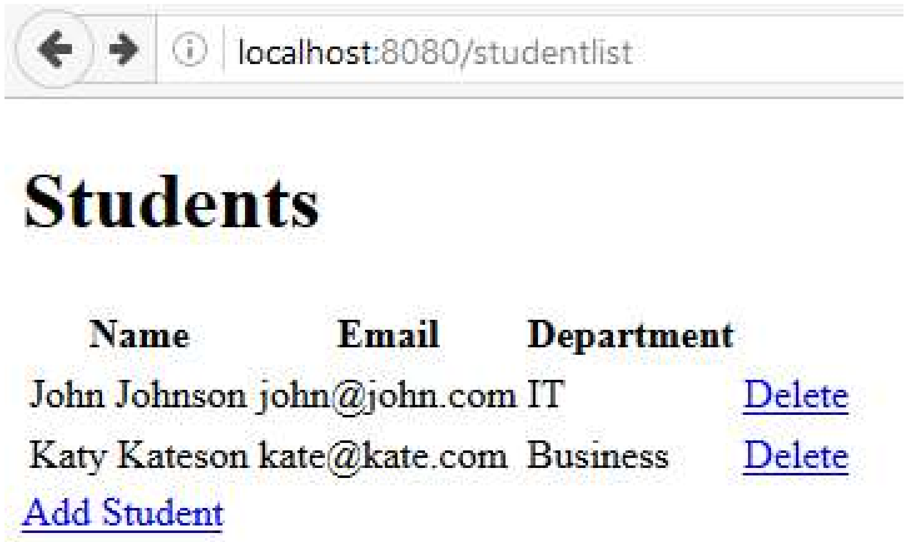
Spring Boot: JPA

- Show department name in the studentlist

```
<tr th:each = "student : ${students}">
   <td th:text="${student.firstName} + ' ' + ${student.lastName}"></td>
   <td th:text="${student.email}"></td>
   <td th:text="${student.department.name}"></td>
   <td><a th:href="@{/delete/{id}(id=${student.id})}">Delete</a></td>
</tr>
```

Haaga-Helia

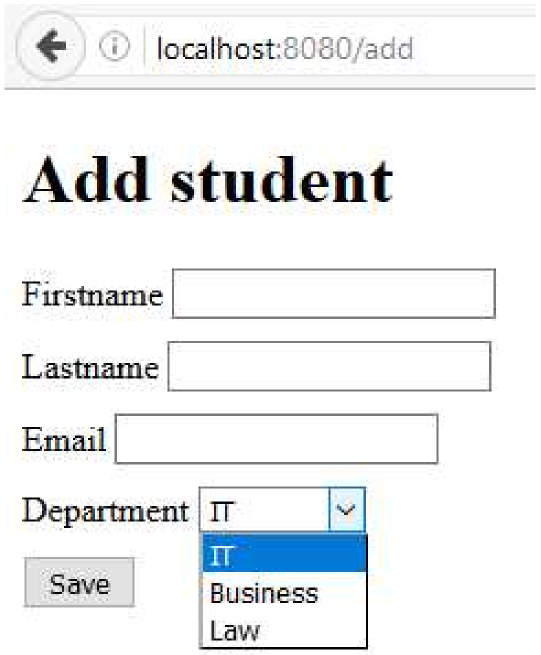
Spring Boot: JPA


localhost:8080/studentlist

Students

Name	Email	Department	
John Johnson	john@john.com	IT	Delete
Katy Kateson	kate@kate.com	Business	Delete

Add Student


localhost:8080/add

Add student

Firstname	
Lastname	
Email	
Department	IT

Spring Boot: JPA

- Edit functionality
  - Similar to Add functionality
  - Model contains now edited object instead of empty object (in the case of add)

```
// Edit student
@RequestMapping(value = "/edit/{id}")
public String showModStu(@PathVariable("id") Long studentId, Model model){
   model.addAttribute("student", repository.findById(studentId);
   model.addAttribute("departments", drepository.findAll());
   return ”editstudent";
}
```

Haaga-Helia

Spring Boot: JPA edit functionality continues

- In thymeleaf need to be careful of proper syntax. For example modifystudent.html:

```
…
<body>
  <h1>Modify student</h1>
  <div>
    <form th:object="${student}" th:action="@{../save}" action="#" method="post">
      <label for="id"></label>
      <input type="hidden" id="id" th:field="*{id}" readonly="readonly" />
      <div style="clear: both; display: block; height: 10px;"></div>
      <label for="studentNumber">Student number</label>
      <input type="text" id="author" th:field="*{studentNumber}" />
      <label for="catlist">Category</label>
      <select id="catlist" th:field="*{category.categoryid}" class="form-control">
      <option th:each="cat: ${categories}" th:value="${cat.categoryid}" th:text="${cat.name}"></option>
      </select>
      <div style="clear: both; display: block; height: 10px;"></div>
      <input class="btn btn-success" type="submit" value="Save"></input>
     </form>
</div>
</body>
</html>
```

Haaga-Helia

Spring Boot: JPA

- Edit functionality
  - Template for editing
  - Note! Id should be added otherwise new student is created

```
<form th:object="${student}" th:action="@{../save}" action="#" method="
post">
    <input type="hidden" th:field="*{id}" class="form-control"/>
    <input type="text" th:field="*{firstName}" />
    <input type="text" th:field="*{lastName}" />
    <input type="text" th:field="*{email}" />
    <input type="submit" class="btn btn-success" value="Save"></input>
</form>
```

Spring Boot: MariaDB

- Switching database from H2 to MySQL/MariaDB
  1. Remove H2 dependency from the pom.xml file
  2. Add MySQL/MariaDB dependency to the pom.xml file

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.33</version>
</dependency>
```

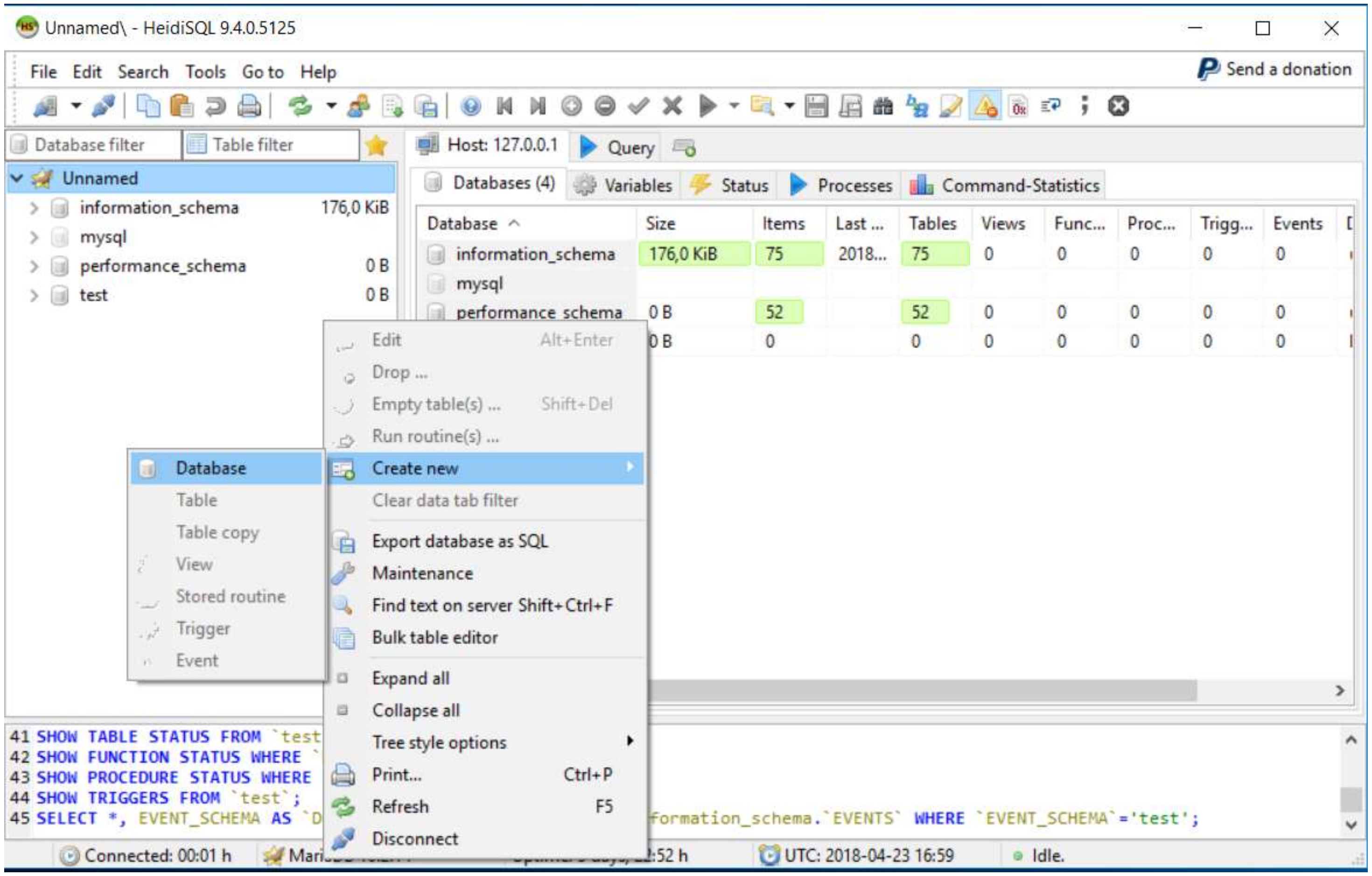
Add following database connection settings to the `application.properties` file:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/jusju?useUnicode=true&useJDBCCompli
antTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=jusju
spring.datasource.password=YOUR_PASSWORD
spring.datasource.initialization-mode=always
spring.batch.initialize-schema=always
```

Spring Boot: MySQL/MariaDB

- Switching database from H2 to MariaDB
  - For testing purposes you can install MariaDB locally and use `localhost` as server address. Change also password in the `application.properties` file (Password that you defined when installing MySQL/MariaDB).
  - Use HeidiSQL to create new database before running the application (see the next slide)
  - Set `DB_NAME` in the application.properties file to match name of the database you just created.
  - Run the application and check with HeidiSQL that database tables were created.

Spring Boot: MySQL/MariaDB



Database	Size	Items	Last ...	Tables	Views	Func...	Proc...	Trigg...	Events
information_schema	176,0 KiB	75	2018...	75	0	0	0	0	0
mysql									
performance_schema	0 B	52		52	0	0	0	0	0
	0 B	0		0	0	0	0	0	0

```
SHOW TABLE STATUS FROM `test
SHOW FUNCTION STATUS WHERE `
SHOW PROCEDURE STATUS WHERE
SHOW TRIGGERS FROM `test`;
SELECT *, EVENT_SCHEMA AS `D        formation_schema.`EVENTS` WHERE `EVENT_SCHEMA`='test';
```

Connected: 00:01 h   Mari   :52 h   UTC: 2018-04-23 16:59   Idle.