

# EXAM

**Exam code: DAT152**

**Course name: Advanced Web Applications**

**Date: November 28, 2024**

---

Type of examination: Written exam

Time: 4 hours (0900 -1300)

Number of questions: 6

Number of pages: 15 (including this page and appendices)

Appendices: The last 3 pages

Exams aids: Bilingual dictionary

Academic coordinator: Bjarte Kileng (909 97 348), Tosin Daniel Oyetoyan (405 70 403)

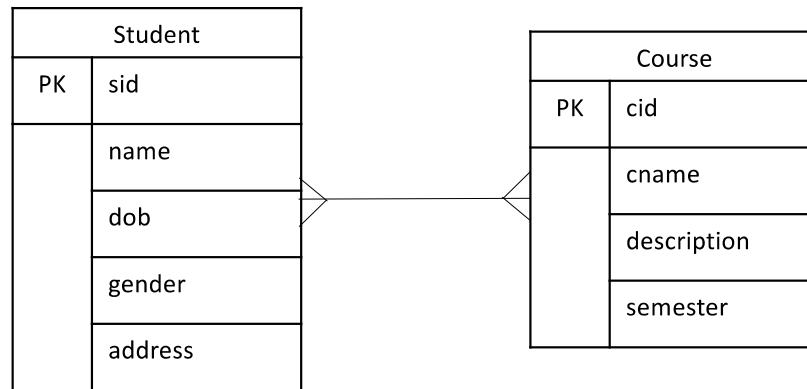
External sensor: Kent Inge Fagerland Simonsen

Notes: None

**Important information:** When answering questions in Wiseflow, give the document or attachment a name and then save before you start to answer the questions. This will start automatic saving of the document.

If you do not follow this procedure, you will lose your work if Wiseflow hangs or fails.

## Question 1 – Web Frameworks and Services (30% ~ 72minutes)



The above shows a simple student course entity model for building a studapp REST service. The student-course model has many-to-many relationship. A student can register for many courses and a course can contain many students. The REST services will allow for the management of students and courses per semester. For example, students will be able to create a profile, update their information, register for courses, withdraw from courses, update their courses, or list the courses they have registered for. Staff will be able to list all the students registered for a course, create a new course, update a course information, or delete courses.

- a) Build a REST API endpoints for this service. The REST API should provide the following services:
- Student
    - i. Register/create a profile
    - ii. View details about own profile
    - iii. Update profile information
    - iv. Register for a new course
    - v. Withdraw from a course
    - vi. List all registered courses
    - vii. View details about a registered course
  - Staff
    - i. List all courses
    - ii. List all students registered for a course
    - iii. Create a new course
    - iv. Update a course
    - v. Delete a course
    - vi. List all students

Use the table below to structure your answers (example in the first row).

Note that you can have up to four levels (e.g., collection/identifier/collection/identifier)

Service	API Method	Endpoint (URI)	HTTP Method
Get or view details about own profile	getStudentProfile	students/{sid}	GET
...	...	...	...

- b) Assume that you have the **Student Service** class, and the **Course Service** class implemented in Spring Framework as below.

```
@Service
public class StudentService {
    @Autowired
    private StudentRepository studentRepository;
    public Student saveStudentProfile(Student student) {
        return studentRepository.save(student);
    }
    public List<Student> findAll(){
        return (List<Student>) studentRepository.findAll();
    }
    public Student findById(Long sid) throws StudentNotFoundException {
        Student student = studentRepository.findById(sid);
        return student;
    }
    public Student updateStudentProfile(Student student, Long sid) throws StudentNotFoundException {
        findById(sid);
        ...
        return studentRepository.save(student);
    }
    public Set<Course> saveCourse(Long sid, Course course) throws StudentNotFoundException{
        Student student = findById(sid);
        student = save(student, course);
        return student.getCourses();
    }
    public Set<Course> findCourses(Long sid) throws StudentNotFoundException{
        Student student = findById(sid);
        return student.getCourses();
    }
    public Course findCourse(Long sid, Long cid) throws StudentNotFoundException, CourseNotFoundException{
        Student student = findById(sid);
        Course course = find(student, cid);
        return course;
    }
    public Set<Course> deleteCourse(Long sid, Long cid) throws StudentNotFoundException, CourseNotFoundException{
        Student student = findById(sid);
        Course course = find(student, cid);
        courses = delete(courses, course);
        return courses;
    }
}
```

```

@Service
public class CourseService {
    @Autowired
    private CourseRepository courseRepository;
    public Course saveCourse(Course course) {
        return courseRepository.save(course);
    }
    public Course updateCourse(Course course, Long cid) throws CourseNotFoundException {
        findCourseById(cid);
        course.setId(cid);
        course = courseRepository.save(course);
        return course;
    }
    public List<Course> findAll(){
        return (List<Course>) courseRepository.findAll();
    }
    public Set<Student> findStudentsByCid(Long cid) throws CourseNotFoundException {
        Course course = findCourseById(cid);
        return course.getStudents();
    }
    public void deleteById(Long cid) throws CourseNotFoundException {
        findCourseById(cid);
        courseRepository.deleteById(cid);
    }
}

```

- I). Implement the controller class and the methods for all the **Student** rest api endpoints you identified in a) (example of a template for StudentController is given below). Note that your method must return the appropriate HttpStatus code. You can also assume that you have two exceptions – StudentNotFoundException and CourseNotFoundException that you can use in your code.

```

@RestController
@RequestMapping("/studapp/api/v1")
public class StudentController {

    // Write your controller methods here

}

```

- II). Implement the controller class and the methods for the **Course** rest api endpoints you identified in a).

```

@RestController
@RequestMapping("/studapp/api/v1")
public class CourseController {

    // Write your controller methods here

}

```

- c) Assume the studapp service is using a 3<sup>rd</sup> party identity provider (IdP) for its users based on OAuth2 protocol.
- I). Briefly explain the authorization flow using OAuth2 “**authorization code grant type**” scheme between the resource owner and the IdP.
  - II). Assume that there are two roles ‘ADMIN’ and ‘USER’ in the JWT access token where the ADMIN and USER roles are assigned to the staff, and only the USER role is assigned to

the students. Protect the REST API endpoints in the StudentController and the CourseController classes using Spring relevant annotations.

```
@RestController
@RequestMapping("/studapp/api/v1")
public class StudentController {

    // update your controller methods in (b) with the appropriate roles and Spring
    // authorization annotations

}

@RestController
@RequestMapping("/studapp/api/v1")
public class CourseController {

    // update your controller methods in (b) with the appropriate roles and Spring authorization
    // annotations

}
```

## Question 2 – Globalization (10% ~ 24minutes)

- In your own words, briefly explain what are globalization, internationalization, and localization?
- Given the following jsp code below, internationalize the page and localize it to English and one other language of your choice. Your solution must include the ‘properties’ files for English and the other language you chose.

```
<body>
<div> If you live in Bergen, you should expect rain between 200 and 240 days out of 365 days. This is equivalent to
roughly 55% of rain in a year. </div>
</body>
```

## Question 3 – Custom tags (10% ~ 24minutes)

The tasks require to create a custom tag (fahrToCelsius) to convert a temperature in Fahrenheit to Celsius. As shown below, the tag will take an attribute ‘fahr’ and convert the value to celsius.

The formula for converting fahrenheit to celsius:  $celsius = 5.0/9.0 * (fahr - 32)$ ;

```
<body>
<dat152:fahrToCelsius fahr="80"/>
</body>
```

The above tag will produce the result below on a jsp page where it is used:

“26,7C”

- Implement the fahrToCelsius tag using SimpleTagSupport class. You need to override the doTag method and implement your solution. Note that you do not need to write the TLD xml file.

```
@Override
public void doTag() throws JspException, IOException {...}
```

- Implement the tag using a tag-file. A tag file starts with the line below:

```
<%@ tag language="java" pageEncoding="UTF-8"%>
```

## Question 4 – Universal Design (5% ~ 12minutes)

- a) What is Universal Design?
- b) Give 3 principles of UD
- c) Briefly discuss the WCAG principle #2 – “Operable” and the guidelines for this principle.

## Question 5 – Web security (25% ~ 60 minutes)

- a) You have saved a session token as a cookie in a web browser storage. What are the secure ways to protect the cookie from XSS vulnerability and session hijacking attack?
- b) A logout controller of a web application contains the code below.
  - I). What major session vulnerability is present in this implementation?
  - II). Write a mitigation for this vulnerability.

```
@WebServlet("/logout")
public class LogoutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        request.getSession().removeAttribute("user");
        request.getRequestDispatcher("index.jsp").forward(request, response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        doGet(request, response);
    }
}
```

- c) Given the 2 SQL queries below:

### Implementation 1:

```
public AppUser getAuthUser(String username, String password){
    ...
    PreparedStatement stmt = conn.prepareStatement("SELECT * FROM SecOblig.AppUser
    WHERE USERNAME = "+username+" and PASSWORD = "+password);
    stmt.executeUpdate();
    ...
}
```

### Implementation 2:

```
public List<String> getUsernames(){
    ...
    Statement stmt = conn.createStatement();
    stmt.executeQuery("SELECT username FROM SecOblig.AppUser");
    ...
}
```

- I). Which of the two implementations can be vulnerable to SQL injection vulnerability?
- II). Write the mitigation code for the implementation that you identified as vulnerable.

- d) Password can be stored securely by using 1) cryptographically secure hash, 2) slow hash algorithm, 3) unique per user salt, and 4) pepper. **Explain** the security benefits of using these four password storage features.
- e) The jjwt-api library that we have used in Oblig4 contains two methods for parsing the JWT token. Method #1 – `parse(token)` and Method #2 – `parseClaimsJws(token)`. Assume that the JWT has been **modified** as below:

I). JWT #1

```

HEADER:
{
  "alg": "none"
}
PAYLOAD:
{
  "iat": 1730803010",
  "role": "ADMIN",
  "user": "Bob"
}
Encoded JWT =
eyJhbGciOiJub251Iiwia2lkIjoicEp3W.eyJleHAiOjE2OTg5MTA4OTQsImhhdCI6MTY5ODg3NDg5NC.

```

II). JWT #2

```

HEADER:
{
  "alg": "RS256"
}
PAYLOAD:
{
  "iat": 1730803010",
  "role": "ADMIN",
  "user": "Bob"
}
Encoded JWT =
eyJhbGciOiJSUzI1NiIsImtpZCI6InBKd.eyJleHAiOjE2OTg5MTA4OTQsImhhdCI6MTY5AADg3NDgAOP.

```

The results of the code below for parsing JWT #1 and JWT #2 can be any of:

- (1) *throw an `InvalidTokenException` in line 12.*
- (2) *invoke the method “`deleteAllUsers()`” in line 7.*
- (3) *logs the error (“`You are not an admin user`”) in line 9.*

Using the above information and your knowledge from Oblig4, discuss the results the code will produce for:

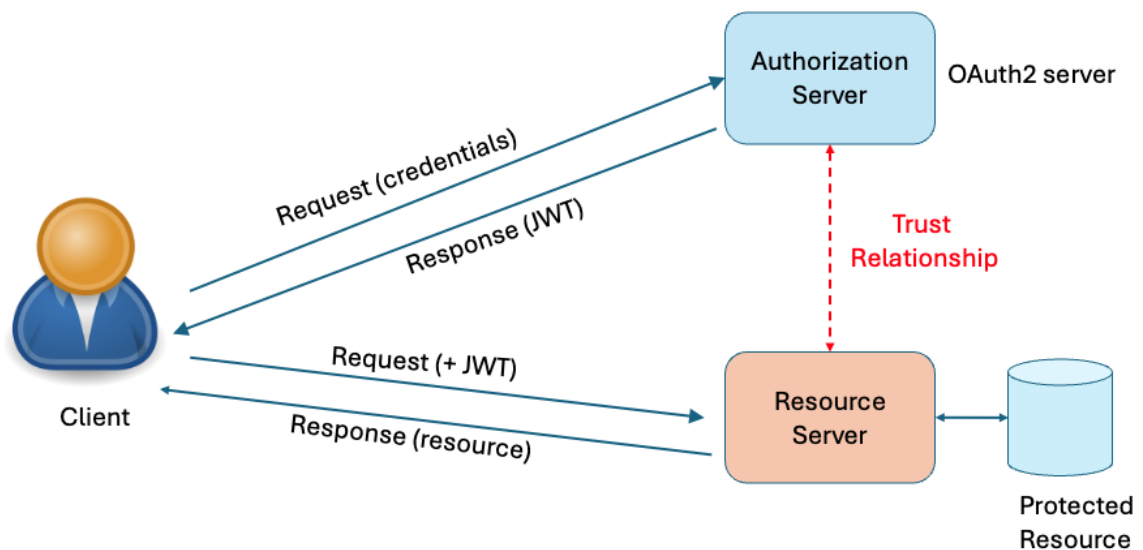
- I) JWT #1
- II) JWT #2.

```

1. try {
2.   Jwt jwt = Jwts.parser().setSigningKey(KEY).parseClaimsJws(accessToken);
3.   Claims claims = (Claims) jwt.getBody();
4.   String user = (String) claims.get("user");
5.   String role = (String) claims.get("role");
6.   if (role.equals("ADMIN")) {
7.     deleteAllUsers();
8.   } else {
9.     log.error("You are not an admin user");
10.  }
11. } catch (JwtException e) {
12.   throw new InvalidTokenException(e);
13. }

```

- f) In the figure below, there is a trust relationship between the Authorization Server and the Resource Server. Describe briefly how this ‘trust’ relationship between the Authorization Server and the Resource Server is established.
- g) In the same figure below, a client can access a protected resource by sending a request together with a JWT token to the “Resource Server”.  
**Describe** the security actions that the “**Authorization Server**” must take before sending the **JWT** token in a response to the client.



## Question 6 – JavaScript (20% ~ 48 minutes)

- a) JavaScript theory.

A shadow DOM can be created by the code below:

```
const shadow = this.attachShadow({ mode: 'closed' });
```

*Code snippet 1: Creating a shadow DOM*

- What is a shadow DOM? Your answer must outline the differences between a “normal” DOM and a shadow DOM.
- In the above example of Code snippet 1, a shadow DOM is created where property *mode* is set to *closed*.
  - What are the consequences if *mode* is set to *open*?
  - Show with a code example how to utilize that *mode* is set to *open* for a shadow DOM component.



iii. The HTML tag `<SLOT>` can be useful when making a web component.

1. What is the purpose of the tag `<SLOT>`?
2. Demonstrate with code examples how to use the `<SLOT>` tag.

A component with shadow DOM and *mode* set to *closed* should produce the view represented by the HTML code below:

```
<P>Course is <SPAN>DAT152</SPAN></P>
```

*Code snippet 2: View produced by a shadow DOM component*

The course code is decided by the parent component.

iv. Discuss different approaches for the parent component to set the course code to be displayed by the child component.

The code below is found in a web application:

```
const result = await storagemanager.remove(id);
```

*Code snippet 3: Code of web application*

- v. What kind of object must be returned by the call to *storagemanager.remove(id)*? Describe this kind of object.
- b) In this task you are asked to write the JavaScript code of a GUI component **ShoppingList** that can manage the display of a shopping list.

This component is part of a web application where the user also can add items to the shopping list, and the shopping items can be e.g. stored in a database on the web server. This task though is concerned only with display of the shopping list.

The **ShoppingList** component is based on the following HTML templates:

```
const template = document.createElement("template");
template.innerHTML = `
  <table id="shoppinglist">
    <thead>
      <tr><th>Item</th><th>Amount</th><th>Deadline</th></tr>
    </thead>
    <tbody></tbody>
  </table>`;
```

*Code snippet 4: Template for the HTML table of component ShoppingList*

```
const itemrow = document.createElement("template");
itemrow.innerHTML = `
  <tr>
    <td></td>
    <td></td>
    <td></td>
    <td><button type="button">Remove</button></td>
  </tr>`;
```

*Code snippet 5: Template for an HTML table row of a shopping*

The illustration below shows a possible view of the **ShoppingList** component.

Item	Amount	Deadline	
Milk	5 litres	Friday, in the morning	<button>Remove</button>
Bananas	2 kg	Saturday, noon	<button>Remove</button>
Bread	1 pcs.	Monday, in the afternoon	<button>Remove</button>

Figure 1: The *ShoppingList* component

An item on the shopping list is represented as an object with the following properties:

- *id*: Integer, a unique identifier of the shopping item
- *name*: String, the name of the item
- *amount*: String, the amount of item
- *deadline*: Date, deadline to fulfil shopping of the item

As an example, the shopping item *Bananas* in Figure 1 can be represented by the following object:

```
const bananas = {
  "id": 1,
  "name": "Bananas",
  "amount": "2 kg",
  "deadline": new Date(2024, 10, 30, 12, 23)
};
```

Code snippet 6: Object representing the shopping item *Bananas*

The *deadline* property for the shopping item *Bananas* corresponds to the date Saturday November 30, 2024, at 12:23. The text in Figure 1 that represents the deadline is produced by the JavaScript **Intl** object, if language *en-GB*, as:

```
const deadlineText = item.deadline.toLocaleDateString(
  navigator.language,
  { weekday: "long", dayPeriod: "long" }
);
```

Code snippet 7: Text with *Weekday* and *day period* for deadline of item

The HTML element class of **ShoppingList** has three public methods:

- *additem(item)*: Adds an item to the shopping list.

This method updates the view.

**Observe:** The shopping list should be displayed sorted on the deadline, with the shortest deadline at the top of the list.

Parameters:

- In parameter: **Object**

- Return value: None

With *shoppinglist* an occurrence of **ShoppingList**, and *bananas* the object of Code snippet 6, the code below should add the shopping item for Bananas to the display:

```
shoppinglist.additem(bananas);
```

*Code snippet 8: Adding shopping item for Bananas to the display*

- *removeitem(id)*: Removes an item from the shopping list.

This method updates the view.

Parameters:

- In parameter: **Integer**
- Return value: None

The method removes the item with the given *id* from the shopping list.

With *shoppinglist* an occurrence of **ShoppingList**, the code below should remove the shopping item with unique identifier *id* equal to “1” from the display:

```
shoppinglist.removeitem(1);
```

*Code snippet 9: Removing item with id "1" from the display*

- *addremoveitemcallback(callback)*: The method will add a callback that is run when the user clicks a button *Remove* of **ShoppingList**, see Figure 1.

This method does not update the view.

Parameters:

- In parameter: **Function**
- Return value: Not required

The method can return a value to identify the callback, but that is not required.

When *callback* is run on a click at a button *Remove*, the callback must be run with the unique identifier *id* of the shopping item as argument.

The callback can be used e.g. to remove a shopping item from a database on the server. Only if the item was successfully removed should the item be removed from the display using the *removeitem* public method.

The HTML element class **ShoppingList** includes the HTML template code of Code snippet 5, Code snippet 4 and the JavaScript code of Code snippet 10 shown below:

```
class ShoppingList extends HTMLElement {
  // Add the necessary private fields

  constructor() {
    super();

    const shadow = this.attachShadow({ mode: 'closed' });
    const content = template.content.cloneNode(true);
    shadow.append(content);

    // More code, if necessary
  }

  addremoveitemcallback(callback) {
    // More code
  }

  additem(item) {
    // More code
  }

  removeitem(id) {
    // More code
  }

  // More code, if necessary
}

customElements.define('shopping-list', ShoppingList);
```

*Code snippet 10: JavaScript code of ShoppingList*

**Task:** Fill in the missing code Code snippet 10 above.

## Appendix

### Help for question 1 (REST API using Spring Framework)

org.springframework.web.bind.annotation.GetMapping
org.springframework.web.bind.annotation.PutMapping
org.springframework.web.bind.annotation.DeleteMapping
org.springframework.web.bind.annotation.PostMapping
org.springframework.http.ResponseEntity(HttpStatusCode status)
org.springframework.http.ResponseEntity(T body, HttpStatusCode status)
org.springframework.web.bind.annotation.PathVariable
org.springframework.web.bind.annotation.RequestBody
org.springframework.http.HttpStatus
HttpStatus.OK
HttpStatus.CREATED
HttpStatus.NO_CONTENT
HttpStatus.NOT_FOUND
org.springframework.web.bind.annotation.RequestParam
org.springframework.security.access.prepost.PreAuthorize
org.springframework.beans.factory.annotation.Autowired

### Help for question 2 (JSTL fmt)

Tag Summary	
<a href="#"><u>requestEncoding</u></a>	Sets the request character encoding
<a href="#"><u>setLocale</u></a>	Stores the given locale in the locale configuration variable
<a href="#"><u>timeZone</u></a>	Specifies the time zone for any time formatting or parsing actions nested in its body
<a href="#"><u>setTimeZone</u></a>	Stores the given time zone in the time zone configuration variable
<a href="#"><u>bundle</u></a>	Loads a resource bundle to be used by its tag body
<a href="#"><u>setBundle</u></a>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable
<a href="#"><u>message</u></a>	Maps key to localized message and performs parametric replacement
<a href="#"><u>param</u></a>	Supplies an argument for parametric replacement to a containing <message> tag
<a href="#"><u>formatNumber</u></a>	Formats a numeric value as a number, currency, or percentage
<a href="#"><u>parseNumber</u></a>	Parses the string representation of a number, currency, or percentage
<a href="#"><u>formatDate</u></a>	Formats a date and/or time using the supplied styles and pattern

<a href="#">parseDate</a>	Parses the string representation of a date and/or time
---------------------------	--

## Help for question 6 (JavaScript)

### EventTarget: *addEventListener()* method

The *addEventListener()* method of **EventTarget** sets up a function that will be called whenever the specified event is delivered to the target.

Common targets are **Element**, or its children, **Document**, and **Window**.

Syntax:

```
addEventListener(type, listener)
```

### Node: *textContent* property

The *textContent* property of **Node** represents the text content of the node and its descendants.

### Document: *getElementById()*

#### DocumentFragment: *getElementById()*

The *getElementById()* method returns an **Element** object representing the element whose id property matches the specified string.

### Document: *querySelector()* method

The **Document** method *querySelector()* returns the first **Element** within the document that matches the specified selector, or group of selectors. If no matches are found, *null* is returned.

With *tbody* representing an HTML TBODY of an HTML TABLE, the code below will return the first HTML TR element with an attribute *data-id* equal to *id*:

```
const row = body.querySelector(`tr[data-id="${id}"]`);
```

### Element: *getElementsByTagName()*

The *Element.getElementsByTagName()* method returns a live **HTMLCollection** of elements with the given tag name.

### Element: *firstElementChild*

The *Element.firstElementChild* read-only property returns an element's first child **Element**, or *null* if there are no child elements.

### HTMLTableElement: *rows*

#### HTMLTableSectionElement: *rows*

The read-only property *rows* return a live **HTMLCollection** of all the rows in the table or section, elements.

Examples of HTML TABLE sections are e.g.: <TBODY> and <THEAD>.

### HTMLTableRowElement: *cells*

The *cells* read-only property of **HTMLTableRowElement** returns a live **HTMLCollection** containing the cells in the row.

### Element: `append()`

The *Element.append()* method inserts a set of **Node** objects or strings after the last child of the **Element**. Strings are inserted as equivalent **Text** nodes.

### Element: `before()`

The *Element.before()* method inserts a set of **Node** objects or strings in the children list of this **Element's** parent, just before this **Element**. Strings are inserted as equivalent **Text** nodes.

### HTMLInputElement: Instance property *value*

A string that represents the current value of the control. If the user enters a value different from the value expected, this may return an empty string.

### Date: `getTime()`

The *getTime()* method of **Date** instances returns the number of milliseconds for this date since the epoch, which is defined as the midnight at the beginning of January 1, 1970, UTC.