Western Norway University of Applied Sciences

# EXAM

**Exam code: DAT152**

**Course name: Advanced Web Applications**

**Date: December 11, 2023**

---

Type of examination: Written exam

Time: 4 hours (0900-1300)

Number of questions: 6

Number of pages: 14 (including this page and appendices):

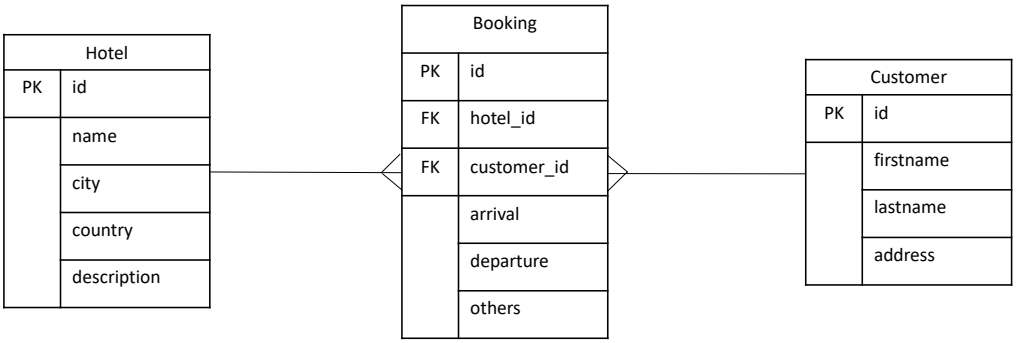
Appendices: The last 3 pages

Exams aids: Bilingual dictionary

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

Notes: None

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




A 3rd party company (HotelOnline) provides online hotel booking service for customers worldwide. The above model shows the customer hotel-booking model.

a) Build a REST API endpoints for this service. The REST API should provide the following services:
   - Produce a list of all the hotels.
   - Provide details of a hotel.
   - Provide support for creating, updating, and deleting a hotel.
   - Produce a list of all the customers.
   - Provide details of each customer.
   - Provide the support for creating, updating, and deleting a customer.
   - Produce a list of all bookings.
   - Provide details of a specific booking.
   - Provide support for customers to book a hotel, update a booking and cancel their booking.
   - Produce a list of all bookings made by a customer.

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

| **Service** | **API Method** | **Endpoint (URI)** | **HTTP Method** |
|---|---|---|---|
| List all hotels | getHotels | /hotels | GET |
| … | … | … | … |

b) You are provided with the Customer model class that aggregates the Booking objects. Also, you are provided with the repository class for Customer (CustomerRepository) containing the CRUD methods required to write the service and controller classes. The implementation is using Spring Framework.

```
@Entity
public class Customer {
        ...
    private Long id;

        private Set<Booking> bookings = new HashSet<>();
        public Set<Booking> getBookings() {
          return bookings;
        }
        public void addBooking(Booking booking) {
          bookings.add(booking);
        }
        public void removeBooking(Booking booking) {
          bookings.remove(booking);
        }
```

```
    ...
}
```

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

        List<Customer> findAll();
        Optional<Customer> findById(Long id);
        Customer save(Customer customer);
        void delete(Customer customer);
}
```

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

```
@RestController
@RequestMapping("/hotelonline/api/v1")
public class CustomerController {

        // Write your controller methods here

}
```

II). Write the service class and the methods for the corresponding controller class in I). An example of a template for CustomerService class is given below.

```
@Service
public class CustomerService {

        // Write your service methods here
}
```

c) Assume HotelOnline is using a 3rd party identity provider (IdP) for its customers based on OAuth2 protocol.

I). Briefly explain the authorization flow using OAuth2 "resource owner credentials/password grant" 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 HotelOnline staff, and only the USER role is assigned to customers. Protect the REST API endpoints in the CustomerController class using Spring relevant annotations.

```
@RestController
@RequestMapping("/hotelonline/api/v1")
public class CustomerController {

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

}
```

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

a) In your own words, briefly explain what are globalization, internationalization, and localization?

b) 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>
 <h1>DAT152 Exam info</h1>
 <div> This course has between 50 and 100 registered students</div>
 <table border="1">
  <tr>
   <td><b>Date:</b>11.12.2023</td>
  </tr>
  <tr>
   <td><b>Total duration:</b>4 hours</td>
  </tr>
  <tr>
   <td><b>Type:</b>Written exam</td>
  </tr>
  <tr>
   <td><b>Pass mark:</b>You need more than 39.9% correct answers to pass</td>
  </tr>
 </table>
<body>
```

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

The tasks require to create a custom tag to calculate the exponent of a given number. As shown below, the tag will take a number and an exponent.

```
<body>
   <b>Math.pow(5, 3) = </b><dat152:expFunction number="5" exponent="3"/>
</body>
```

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

**Math.pow(5, 3)** = 125.0

The Math.pow function below can be used for the questions:

**double** java.lang.Math.pow(**double** number, **double** exponent);

a) Implement the "expFunction" 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 {...}
```

b) Implement the "expFunction" 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 Web Content Accessibility Guidelines (WCAG)?
b) Briefly discuss WCAG principle #2 – "Operable" and the guidelines for this principle.

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

a) A software development lifecycle consists of requirement, design, implementation, testing, and deployment/release phases. Mention one approach to identify security defects in each phase of the software development lifecycle.

b) Given the SQL query below where the name and email are user supplied data:

```
String sqlQuery = "INSERT INTO USERS (id, name, email) VALUES ('"+id+"','"+name+"','"+email+"')";
PreparedStatement stmt = conn.prepareStatement(sqlQuery);
stmt.executeUpdate();
```

I). Briefly explain why this SQL query can be vulnerable to injection.
II). Mitigate the SQL injection vulnerability by writing the mitigation code.

c) Briefly explain the difference between XSS and CSRF
I). Describe how output encoding works to mitigate XSS.
II). Describe the synchronizer token pattern for mitigating CSRF vulnerability.

d) Given the code snippet below: What major **session** vulnerability can you see in the authentication controller?

```
1.  protected void doPost(HttpServletRequest request,
2.  HttpServletResponse response) throws ServletException, IOException {
3.      boolean successfulLogin = false;
4.      String username = validate(request.getParameter("username"));
5.      String password = validate(request.getParameter("password"));
6.      if (username != null && password != null) {
7.              AppUser authUser = getAuthenticatedUser(username, password);
8.              if (authUser != null) {
9.                successfulLogin = true;
10.               request.getSession().setAttribute("user", authUser);
11.             }
12.     }
13.     // do authenticated stuff...
14. }
```

e) A Json Web Token (JWT) consists of three parts: Header.Claims.Signature
I). Describe the two different ways to derive the signature part.
II). Assume the JWT token is signed with the secret key "victory". Discuss the major threat to this token.

f) 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). Given that the modified JWTs below:

I). JWT #1

```
HEADER:
{
    "alg": "none"
}
PAYLOAD:
{
    "iat": 1572038943”,
    "admin": "true",
    "user": "Jerry”
}
Encoded JWT =
eyJhbGciOiJub25lIiwia2lkIjoicEp3W.eyJleHAiOjE2OTg5MTA4OTQsImlhdCI6MTY5ODg3NDg5NC.
```

II). JWT #2

```
HEADER:
{
    "alg": "RS256"
}
PAYLOAD:
{
    "iat": 1572038943”,
    "admin": "true",
    "user": "Jerry”
}
Encoded JWT =
eyJhbGciOiJSUzI1NiIsImtpZCI6InBKd.eyJleHAiOjE2OTg5MTA4OTQsImlhdCI6MTY5ODg3NDg5NC.
```

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 “removeAllUsers()”* 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.      boolean isAdmin = Boolean.valueOf((String) claims.get("admin"));
6.      if (isAdmin) {
7.          removeAllUsers();
8.      } else {
9.          log.error("You are not an admin user");
10.     }
11. } catch (JwtException e) {
12.     throw new InvalidTokenException(e);
13. }
```

g) Password can be stored securely by using 1) cryptographically secure hash, 2) slow hash algorithm, 3) unique per user salt, and 4) pepper. Discuss the security benefits of using these four password storage features.

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

a) Asynchronous code and JavaScript:

i. Give examples on situations where JavaScript code will be run asynchronously. Observe, you are not asked to write any JavaScript code.

To get points on this task, you must clearly explain the asynchronous behaviour of your examples.

ii. The browser is running asynchronously a piece of JavaScript code that is blocking. How does this affect the browser and the application? You must explain your answer.

iii. The JavaScript code example below uses the *async* and *await* keywords:

```
async function myFunction() {
    try {
        const answer = await waitforSomething();
        console.log(`Resolved with value "${answer}"`);
    } catch (e) {
        console.log(`Rejected with "${e.message}"`);
    }
}
```

*Code snippet 1: JavaScript code with async and await*

What kind of ECMA object must the function *waitforSomething* return?

Write code for a possible *waitforSomething* function. The object returned by *waitforSomething* should include some asynchronous code.

b) In this task you are asked to write the JavaScript code of a GUI component **ExamManager** that is explained below.

A web application is made up of three GUI components:

- **ExamManager**, with custom tag *exam-manager*,
- **ExamInfo**, with custom tag *exam-info*,
- **ExamList**, with custom tag *exam-list*.

The component **ExamManager** is made up of the components **ExamInfo** and **ExamList** and is based on the following HTML template:

```
const template = document.createElement("template");
template.innerHTML = `
    <h1>Exam manager</h1>
    <div>
        <exam-info></exam-info>
        <exam-list></exam-list>
    </div>`;
```

*Code snippet 2: HTML template for component ExamManager*

The component **ExamList** can display a list of course exams with exam dates, and **ExamInfo** lets the user add a new course exam or modify the date of an existing exam.

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

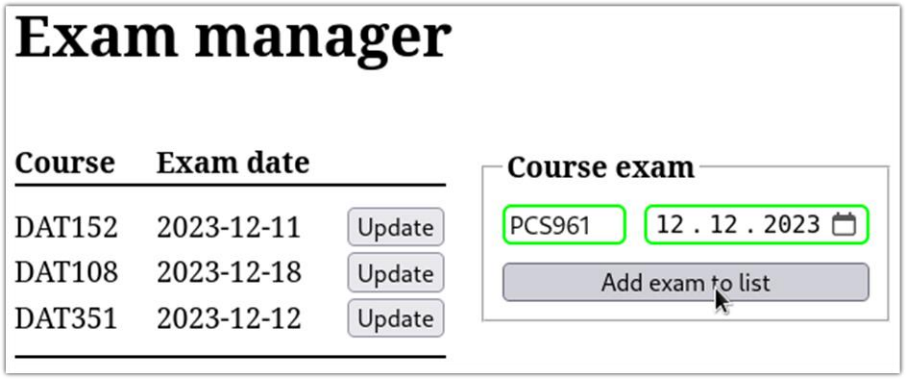



*Figure 1:* ExamManager *component view*

The right side of Figure 1 shows the **ExamInfo** component, and is also shown below:

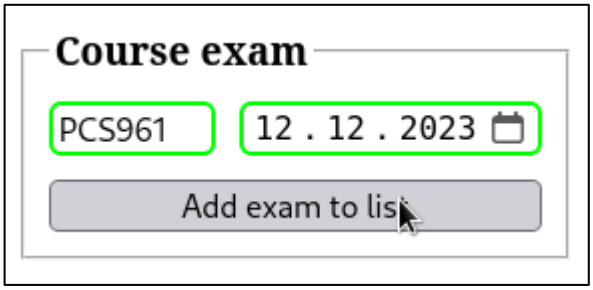



*Figure 2: View of ExamInfo component*

The left side of Figure 1 shows the **ExamList** component, and is also shown below:

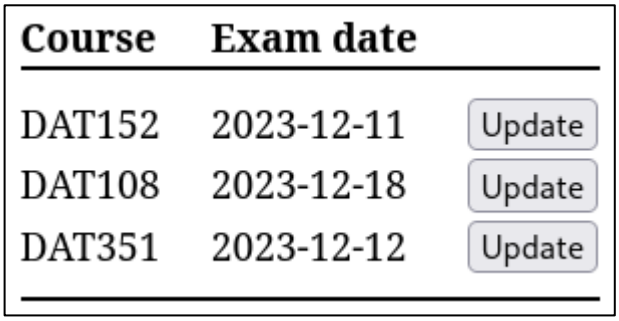



*Figure 3: View of ExamList compoenet*

Methods of both **ExamInfo** and **ExamList** uses a parameter *examdata*. This is an object with the following properties.

- *code*: The course code
- *date*: Date of the exam

The *date* property of *examdata* is a string formatted as YYYY-MM-DD. You can assume that *code* contains digits and capital letters only, as the **ExamInfo** component will capitalize all letters in the course code and only allow valid course codes.

The HTML element class of **ExamInfo** has two public methods only:

- *setInfo(examdata)*: The method will set exam information that will be displayed by the component.

  Parameters:

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

  With *courseinfo* representing an occurrence of **ExamInfo**, the view show in Figure 2 can be the result of the following use of *setInfo*:

```
courseinfo.setInfo({
   "code": "PCS961",
   "date": "2023-12-12"
});
```

*Code snippet 3: Using the method setInfo*

- *examinfoCallback (callback)*: The method will add a callback that is run when the user clicks the button *Add exam to list*.

  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 the button, the callback is run with a parameter *examdata* that contains the data of the HTML **INPUT** elements of **ExamInfo**.

The HTML element class of **ExamList** has two public methods only:

- *addExam (examdata)*: The method will add an exam to the list of exams displayed by the **ExamList** component.

  Parameters:

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

  The parameter *examdata* contains the information to be shown by the **ExamList** component.

  If the course code does not exist in the exam list, a new entry with the new exam is added to the end of the exam list.

  If the course code already exists in the exam list, the course is not added to the list, but the exam date is modified according to the new data.

- *examupdateCallback (callback)*: The method will add a callback that is run when the user clicks one of the buttons *Update*.

  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 one of the buttons, the callback is run with a parameter *examdata* that contains the data of the corresponding exam.

The HTML element class **ExamManager** includes the HTML template code of Code snippet 2, and also the following JavaScript code:

```
class ExamManager extends HTMLElement {
    #shadow;

    // Add the necessary private fields

    constructor() {
        super();

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

        // More code

        this.#shadow.append(content);

        // More code
    }

    // Add any additional private methods
 }
```

*Code snippet 4:JavaScript code of ExamManager*

**Task:** Fill in the missing code of Code snippet 4 above.

c) In this task you are asked to write the JavaScript code of **ExamList**. Observe that you are not asked to write the code of **ExamInfo** on this exam.

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

```
const template = document.createElement("template");
template.innerHTML = "<div id="examlist"></div>";
```

*Code snippet 5: Main HTML template for **ExamList***

```
const examtable = document.createElement("template");
examtable.innerHTML = `
    <table>
        <thead><tr><th>Course</th><th>Exam date</th></tr></thead>
        <tbody></tbody>
    </table>`;
```

*Code snippet 6: Template for the HTML Table of **ExamList***

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

*Code snippet 7: Template for the HTML Table row for the display of one exam*

See the previous task for the details of the **ExamList** public methods *addExam* and *examupdateCallback*.

**Task**: Write the JavaScript code of **ExamList**. You can assume that course codes are digits and capital letters only.

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

| | |
|---|---|
| **parseDate** | Parses the string representation of a date and/or time |

## Help for question 6 (JavaScript)

### EventTarget: addEventListener() method

The ***addEventListener()*** method of the **EventTarget** interface 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)
```

### Element: append() method

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

Syntax:

```
append(param1)
append(param1, param2)
append(param1, param2, /* …, */ paramN)
```

Parameters are **Node** or string objects to insert.

### Element: setAttribute() method

Sets the value of an attribute on the specified element. If the attribute already exists, the value is updated; otherwise a new attribute is added with the specified name and value.

Syntax:

```
setAttribute(name, value)
```

Parameter *name* is the name of the attribute, and *value* is the value to assign to the attribute.

### Element: getAttribute() method

The *getAttribute()* method of the **Element** interface returns the value of a specified attribute on the element.

Syntax:

```
getAttribute(attributeName)
```

### Element and Document: querySelector() method

The *querySelector()* method returns the first element that is a descendant of the element on which it is invoked that matches the specified group of CSS selectors. If no matches are found, null is returned.

Syntax:

```
querySelector(selectors)
```

## Element: closest() method

The *closest()* method of the **Element** interface traverses the element and its parents (heading toward the document root) until it finds a node that matches the specified CSS selector.

Syntax:

```
closest(selectors)
```

## Element: firstElementChild property

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

## Node: textContent property

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