Western Norway University of Applied Sciences

SOLUTION

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: 20 (including this page and appendices)

Appendices: The last 2 pages

Exams aids: Bilingual dictionary

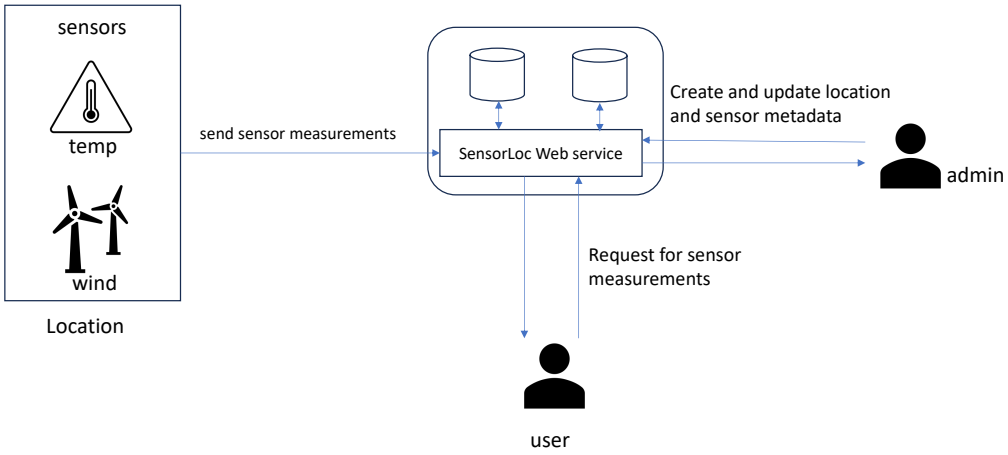
Academic coordinator: Bjarte Kileng (909 97 348), Tosin Daniel Oyetoyan (405 70 403)

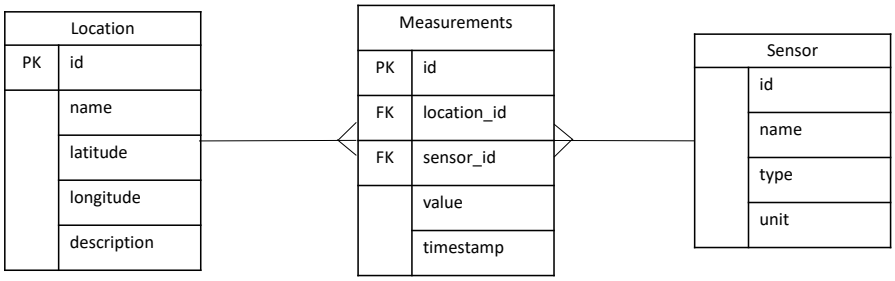
Notes: None

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

…	…	…	…

Solution

Service	API Method	Resource Endpoint (URI Path)	HTTP Method
LocationController (for location metadata)			
List all locations	getLocations	/locations	GET
Details of each location	getLocation	/locations/{id}	GET
Create a new location	createLocation	/locations	POST
Update a location	updateLocation	/locations/{id}	PUT
Delete a location	deleteLocation	/locations/{id}	DELETE
SensorController (for sensor metadata)			
Create a new sensor	createSensor	locations/{id}/sensors or /sensors	POST
List all sensors	getSensors	locations/{id}/sensors or /sensors	GET
Details of a sensor	getSensor	locations/{id}/sensors/{id} or /sensors/{id}	GET
Update a sensor	updateSensor	locations/{id}/sensors/{id} or /sensors/{id}	PUT
Delete a sensor	deleteSensor	locations/{id}/sensors/{id} or /sensors/{id}	DELETE
SensorDataController (for sensor measurement)			
List all measurements for all sensors in a location	getSensorsLocData	/locations/{id}/measurements	GET
List measurements for a sensor in a location	getSensorLocData	/locations/{id}/measurements/{sensor_id}	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

}
```

Solution I) & c (II)

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

	@Autowired
	private SensorService sensorService;

	@GetMapping("/sensors")
	@PreAuthorize("hasAuthority('USER')")
	public ResponseEntity<Object> getAllSensors(){

		List<Sensor> sensors = sensorService.findAll();

		if(sensors.isEmpty())
			return new ResponseEntity<>(HttpStatus.NO_CONTENT);

		return new ResponseEntity<>(sensors, HttpStatus.OK);
	}

	@GetMapping("/sensors/{id}")
	@PreAuthorize("hasAuthority('USER')")
	public ResponseEntity<Sensor> getSensor(@PathVariable("id") Long id) throws
SensorNotFoundException {

		Sensor sensor = sensorService.findById(id);
		if(sensor != null)
			return new ResponseEntity<>(sensor, HttpStatus.OK);
		else
			return new ResponseEntity<>(HttpStatus.NOT_FOUND);

	}

	@DeleteMapping("/sensors/{id}")
	@PreAuthorize("hasAuthority('ADMIN')")
	public ResponseEntity<Object> deleteSensorById(@PathVariable("id") Long id) throws
SensorNotFoundException {

		sensorService.deleteById(id);
		Sensor sensor = sensorService.findById(id);

		if(sensor == null)
			return new ResponseEntity<>(HttpStatus.OK);

		return new ResponseEntity<>(HttpStatus.NOT_MODIFIED);
	}

	@PostMapping("/sensors")
	@PreAuthorize("hasAuthority('ADMIN')")
	public ResponseEntity<Sensor> createSensor(@RequestBody Sensor sensor){

		sensor = sensorService.saveSensor(sensor);

		return new ResponseEntity<>(sensor, HttpStatus.CREATED);
	}

	@PutMapping("/sensors/{id}")
	@PreAuthorize("hasAuthority('ADMIN')")
```

```
        public ResponseEntity<Sensor> updateSensor(@RequestBody Sensor sensor, @PathVariable
int id)
                        throws SensorNotFoundException{

                sensor = sensorService.updateSensor(sensor, id);

                if(sensor != null)
                        return new ResponseEntity<>(sensor, HttpStatus.OK);
                else
                        return new ResponseEntity<>(HttpStatus.NOT_MODIFIED);
        }

}
```

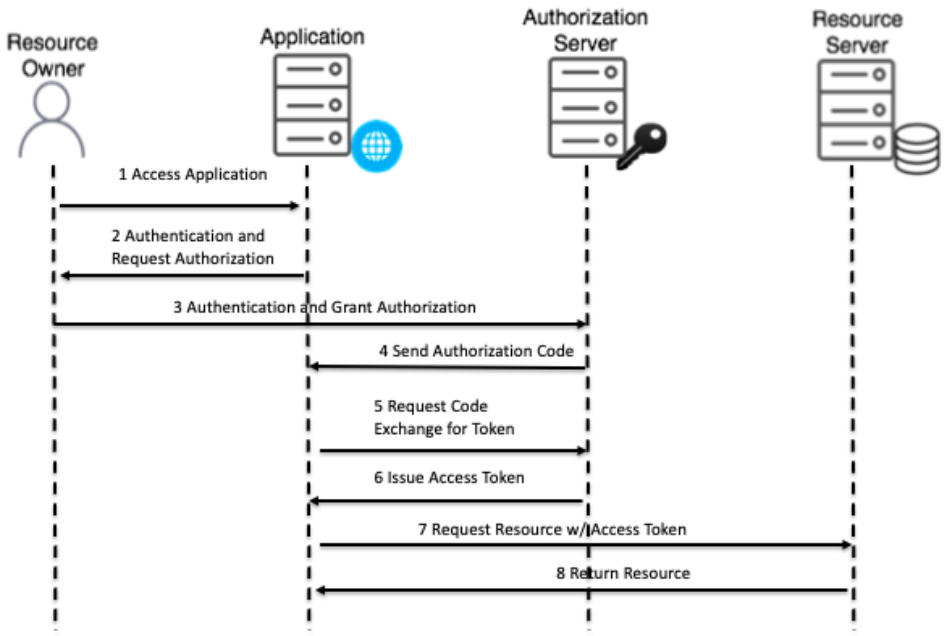
II)

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

        @Autowired
        private MeasurementService dataService;

        @GetMapping("/locations/{id}/measurements")
       @PreAuthorize("hasAuthority('USER')")
        public ResponseEntity<Object>
getAllSensorsMeasurementsByLocation(@PathVariable("id") Long id){

                List<Measurement> sloc = dataService.findAllSensorsByLocation(id);

                if(sloc.isEmpty())
                        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
                else
                        return new ResponseEntity<>(sloc, HttpStatus.OK);
        }

        @GetMapping("/locations/{id}/measurements/{sensor_id}")
       @PreAuthorize("hasAuthority('USER')")
        public ResponseEntity<Object> getSensorMeasurementsOfLocationId(@PathVariable("id")
Long lid, @PathVariable("sensor_id") Long sid) throws MeasurementNotFoundException{

                List<Measurement> sloc = dataService.findSensorByLocation(lid, sensor_id);

                if(sloc.isEmpty())
                        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
                else
                        return new ResponseEntity<>(sloc, HttpStatus.OK);

        }

}
```

c) Assume SensorLoc service is using a 3rd 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.

Solution



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.

Solution

See solution in b (I) where the annotations below are used.

```
@PreAuthorize("hasAuthority('ADMIN')")
@PreAuthorize("hasAuthority('USER')")
```

```
@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?

Solution

Globalization: the process to make an application available globally and it includes internationalization and localization.

Internationalization: the process to enable an application from technical perspective for multiple language support without having to make changes in the design/code.

Localization: the process to adapt application for specific languages or regions, called locale.

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>
```

Solution

Property files (Resource bundles)

Message_en_EN.properties

```
desc=The number of registered students is {0} more this year. The number
of registrations between {1} and {2} is within the range of {3} and {4}
students.
```

Message_no_NO.properties

```
desc=Antall registrerte studenter er {0} flere i år. Antall
registreringer mellom {1} og {2} er innenfor rekkevidden til {3} og {4}
elever.
```

jsp

```
<body>
  <jsp:useBean id="now" class="java.util.Date"></jsp:useBean>
  <fmt:setLocale value="no_NO"/>
  <fmt:bundle basename="no.hvl.dat152.i18n.Message">

     <fmt:message key="desc">
       <fmt:param>
       <fmt:formatNumber value="0.205" type="percent" maxIntegerDigits="2"
minFractionDigits="1"/>
       </fmt:param>
       <fmt:param>
          <c:set var="date1" value="01-05-2023"/>
          <fmt:parseDate pattern="dd-MM-yyyy" var="fdate1" value="${date1}" />
          <fmt:formatDate value="${fdate1}" pattern="dd.MM.yyyy"/>
       </fmt:param>
       <fmt:param>
          <c:set var="date2" value="30-08-2023"/>
          <fmt:parseDate pattern="dd-MM-yyyy" var="fdate2" value="${date2}" />
          <fmt:formatDate value="${fdate2}" pattern="dd.MM.yyyy"/>
       </fmt:param>
       <fmt:param>
          <fmt:formatNumber value="50" type="number"/>
       </fmt:param>
       <fmt:param>
           <fmt:formatNumber value="100" type="number"/>
       </fmt:param>
    </fmt:message>
 </fmt:bundle>
</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 {...}
```

Solution

```
public class LowerCase extends SimpleTagSupport {

    @Override
    public final void doTag() throws JspException, IOException {

        PageContext pageContext = (PageContext) getJspContext();
        JspWriter out = pageContext.getOut();

        StringWriter sw = new StringWriter();
        JspFragment body = getJspBody();
        body.invoke(sw);

        out.println(sw.toString().toLowerCase());
    }
}
```

b) Implement the "lowercase" tag using a tag-file. A tag file starts with the line below:

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

Solution

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

<jsp:doBody var="jspBody" />

<%
    String bc = (String) jspContext.getAttribute("jspBody");
%>

<%=bc.toLowerCase()%>
```

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.

Solution

a) **WCAG** is a standard that defines how to make web pages and content more accessible to a wider range of people with disabilities.

b) **Perceivable:** Information and user interface components must be presentable to users in ways they can perceive.

Perceivable has 4 guidelines (WCAG 2.1):

Text Alternatives: Provide text alternatives for non-text content.

Time-based Media: Provide captions and other alternatives for multimedia.

Adaptable: Create content that can be presented in different ways, including by assistive technologies, without losing meaning.

Distinguishable: Make it easier for users to see and hear content.

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.

Solution

Cryptographically strong hash will ensure password is only verifiable and not recoverable.

Slow hash slows down attackers by making it longer to generate the hash of a password and thereby increasing the length of time it takes to hack a given password.

Salt will ensure that two passwords are different in the database and increase the length of the rainbow table. Salt can also increase the complexity of brute-forcing if stored on a separate database.

Pepper is a secret not stored in the database and that is combined with the password during login. It is meant to defeat offline dictionary and rainbow-table attacks.

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.

Solution

Challenge response pattern has been used which forces the application to request for a form of authentication (what you know) from the user/owner.

c) You inspect the html source of an online blog at *https://gossip.blog.com* and find *"<script>document.write('');</script>"* included in one user's comment.
Explain the types of vulnerability in this attack scenario.

Solution
This is a stored (persistent) XSS and the attacker aims to steal the session cookie of blog users.

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.

Solution

I)

Untrusted data (username and password) are bound to the sql query before passing it to the PreparedStatement. The PreparedStatement is ineffective in this instance.

II)

```
String sqlQuery = "SELECT * FROM ACCDB WHERE USERNAME = ? and PASSWORD = ?";
PreparedStatement stmt = conn.prepareStatement(sqlQuery);
stmt.setString(1, USERNAME);
stmt.setString(2, PASSWORD);
stmt.executeUpdate();
```

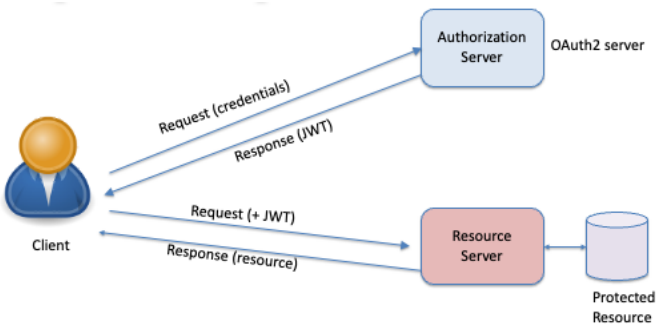
e) A Json Web Token (JWT) consists of three parts: Header.Claims.Signature
I). Describe the two different ways to calculate the signature part.
II). Assume the JWT token is signed with the secret key "bergen". Discuss the major threat to this token.

Solution

The signature can be calculated using 1) HMAC (symmetric) where a secretkey is hashed with the base64 encoding of header and claims. Or 2) Public key cryptography (asymmetric) by using the private key to encrypt base64 encoding of header and claims.

- HMAC: Signature = HMAC(secretKey, base64(header), base64(claims))
- Asymmetric: Signature = PrivateKey(base64(header), base64(claims))

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.



Solution

The "Resource Server" must perform the following validation

- Verify that the JWT access token has valid signature

        - Signature = HMAC(secretKey, base64(header), base64(claims))
        - Signature = PrivateKey(base64(header), base64(claims))
    - Verify that the JWT access token has not expired (Note that expired access token if not verified for expiration will always have valid signature)
    - Verify the issuer and the audience
    - Ensure that the JWT contains the necessary claims to access the protected resource

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?

Solution:

- Synchronization of view from state: Any change in state (Model) will update view.
- Synchronization of state from view: Any change in view will update state (**Model**).

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*.

Solution:

Each SPA state should have its own URL. Different URLs represent different states or views of the same page.

When the URL in address bar is changed, the view is modified by JavaScript. No new web page is loaded from web server. The new view can include data or components that are fetched from the server with Ajax.

If an URL contains a character "#" (the hash sign), no new page is loaded form the server if only the part starting with the character "#" is modified.

If the hash part of an URL is modified, the window event *hashchange* fires. The event handler can then update the state and view according to the new URL, i.e.routing has been implemented.

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

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

Solution:

```
Object.defineProperty(student,"surename", {"writable": false});
```

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

Solution:

An enumerable property will be encountered when looping through the object.

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.

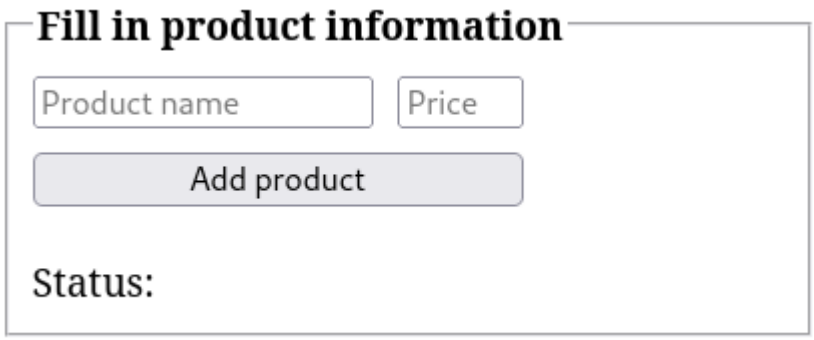


Figure 1:The ProductInfo component

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

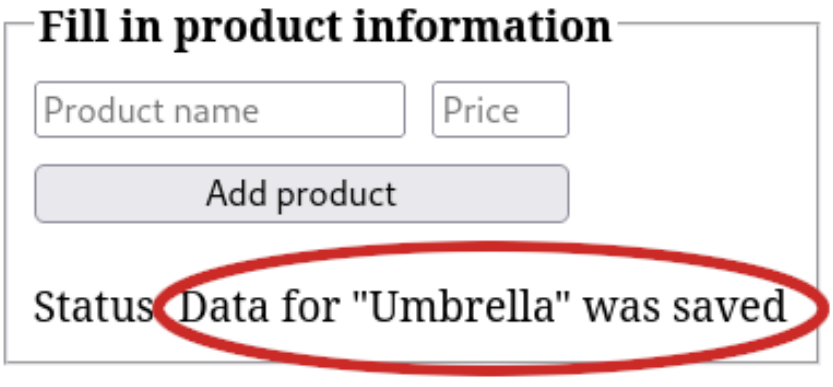


Figure 2: ProductInfo showing a status message

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.

Solution:

```
class ProductInfo extends HTMLElement {
    #callbacks = new Map();
    #iteminput;
    #priceinput;
    #messageElement;

    constructor() {
        super();

        const shadow = this.attachShadow({ mode: 'closed' });
        const content = template.content.cloneNode(true);
        shadow.append(content);
        shadow.querySelector("button").addEventListener("click",
            () => { this.#addproduct() });
        const fs = shadow.querySelector("fieldset");
        this.#iteminput = fs.querySelector("input[name='item']");
        this.#priceinput = fs.querySelector("input[name='price']");
        this.#messageElement = fs.querySelector(".message > span");
    }

    clear() {
        this.#iteminput.value = "";
        this.#priceinput.value = "";
        this.showmessage("");
    }

    setproductcallback(method) {
        const callbackid = Symbol("newexamCallbackID");
        this.#callbacks.set(callbackid, method);
        return callbackid;
    }

    showmessage(message) {
```

```
        console.log(message);
        this.#messageElement.textContent = message;
    }

    #addproduct() {
        const item = this.#iteminput.value.trim();
        const price = this.#priceinput.value.trim();

        if ((item === "") || (price === "")) {
            return;
        }

        const product = { 'item': item, 'price': price };
        this.#callbacks.forEach(callback => callback(product));
    }
}
```

c) The **ProcuctInfo** 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 **ProcuctInfo** 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.

Solution:

```
template.innerHTML = "<product-info></product-info>";

class StorageManager extends HTMLElement {
    #pinfo;

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

        this.#pinfo = shadow.querySelector("product-info");
```

```
        this.#pinfo.setproductcallback(
            (product) => { this.#saveproduct(product) }
        );
    }

    async #saveproduct(product) {
        try {
            const result = await this.#storeOnServer(product);
            if (result.reponsestatus) {
                this.#pinfo.clear();
                this.#pinfo.showmessage(
                    `Data for "${product.item}" was saved`
                );
            } else {
                this.#pinfo.showmessage(
                    `Failed to save data for "${product.item}"`
                );
            }
        } catch (error) {
            this.#pinfo.showmessage(
                `Failed to save data for "${product.item}"`
            );
        }
    }

    #storeOnServer(product) {
        /**
          * Stores product information on server.
          * The candidate was asked to not implement this method.
          **/
    }
}
```

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)

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.