Western Norway
University of
Applied Sciences

~~EXAM~~

Solution

Exam code: DAT152

Course name: Advanced Web Applications

Date: November 28, 2022

Type of examination: Written exam

Time: 4 hours (0900-1300)

Number of questions: 6

Number of pages: 19 (including this page and appendices):

Appendices: The last 7 pages

Exams aids: Bilingual dictionary

Academic coordinator: Bjarte Kileng (909 97 348), Atle Geitung (482 42 851), Tosin Daniel Oyetoyan (405 70 403), Lasse Jenssen (922 33 948)

Notes: None

Question 1 – Globalization (15% ~ 36minutes)

a) Explain, in your own words, the terms globalization (g11n), internationalization (i18n) and localization (l10n). Also explain the relationship between the concepts.

Solution:

A short definition of the three terms. The answer should contain more details.

- The globalization concept is to make an application available globally (g11n)
- Internationalization means to make the application ready for multiple languages (locales) without having to make changes in design (i18n)
- Localization means to make the application available in a particular language (locale) (l10n)

Given the following jsp:

```
<body>
     November 15, 2022, the human population passed
     8,000,000,000 humans.<br>
     The population is expected to grow by 30% until
     it peaks in the 2080s.<br>
</body>
```

b) Internationalize the jsp and localize it to English and one other language. You must add and replace code to the jsp. Also, you need to write the properties files for English and the other language.

Solution (this is one of several solutions, should use Calendar instead of Date):

The jsp:

```
<body>
  <fmt:bundle basename="no.hvl.dat152.eks22.Message">
    <p>
    <fmt:message key="population">
      <fmt:param>
        <%pageContext.setAttribute("date", new java.util.Date(122, 10, 
15));%>
        <fmt:formatDate value="${date}" dateStyle="long" />
      </fmt:param>
      <fmt:param>
        <fmt:formatNumber value="8000000000" type="number" pattern="#,##0"
/>
      </fmt:param>
    </fmt:message>
  </p>
  <p>
    <fmt:message key="grow">
      <fmt:param>
```

```
        <fmt:formatNumber value="0.30" type="percent" maxIntegerDigits="2"
/>
      </fmt:param>
      <fmt:param value="2080" />
    </fmt:message>
  </p>
  </fmt:bundle>
</body>
```

Message_no.properties:

```
population={0}, passerte jordens befolkning {1} mennesker.
grow=Befolkningen er forventet å øke med {0} inntil den når toppen i
{1}-tallet.
```

Message_en.properties:

```
population={0}, the human population passed {1} humans.
grow=The population is expected to grow by {0} until it peaks in the
{1}s.
```

Question 2 – Custom tags (10% ~ 24minutes)

We want to have a tag called “formattedText”. The purpose is to create a formatted text like this:

Before (without the tag):

```
    <p><b>Type: Written digital school exam</b></p>
```

Using the tag:

```
    <p><dat152:formattedText name="Type">
        Written digital school exam
    </dat152:formattedText></p>
```

Both should give the following result on the web page:

Type: Written digital school exam

a) Implement the “formattedText” tag using SimpleTagSupport in Java. You do not need to write xml-code for the tld-file. You do not need import-sentences.

   You will need to override and implement this:

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

Solution:

```
private String name;
```

```
/**
 * @return the name
 */
public String getName() {
    return name;
}

/**
 * @param name the name to set
 */
public void setName(String name) {
    this.name = name;
}

@Override
public final void doTag() throws JspException, IOException {
    PageContext pageContext = (PageContext) getJspContext();
    JspWriter out = pageContext.getOut();

    // this code could have been simplified
    StringWriter stringWriter = new StringWriter();
    JspFragment body = getJspBody();
    body.invoke(stringWriter);
    String bodyText = stringWriter.getBuffer().toString();
    out.println("<b>" + name + ": " + bodyText + "</b>");
}
```

b) Implement the “formattedText” tag using a tag-file.

A tag-file starts with this line:

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

Solution:

```
<%@ tag language="java" pageEncoding="UTF-8"%>
<%@ attribute name="name" type="String"%>

<b>${name}: <jsp:doBody /></b>
```

c) Add the attribute “type” to the “formattedText” tag. The “type” attribute can be either “normal”, “bold” or “italic” and tells how to show the name. Default should be “bold”. Examples (jsp and result on web page):

```
<p><dat152:formattedText name="Type">
    Written digital school exam
</dat152:formattedText></p>
<p><dat152:formattedText type="bold" name="Type">
    Written digital school exam
</dat152:formattedText></p>
<p><dat152:formattedText type="italic" name="Type">
    Written digital school exam
</dat152:formattedText></p>
<p><dat152:formattedText type="normal" name="Type">
    Written digital school exam
</dat152:formattedText></p>
```

Result on the web page:

Type: Written digital school exam

Type: Written digital school exam

Type: Written digital school exam

Type: Written digital school exam

Solution (only one of the two required):

SimpleTagSupport:

```
private String name;
private String type = "bold";

/**
 * @return the name
 */
public String getName() {
    return name;
}

/**
 * @param name the name to set
 */
public void setName(String name) {
    this.name = name;
}

/**
 * @return the type
 */
public String getType() {
    return type;
}

/**
 * @param type the type to set
 */
public void setType(String type) {
    this.type = type;
}
```

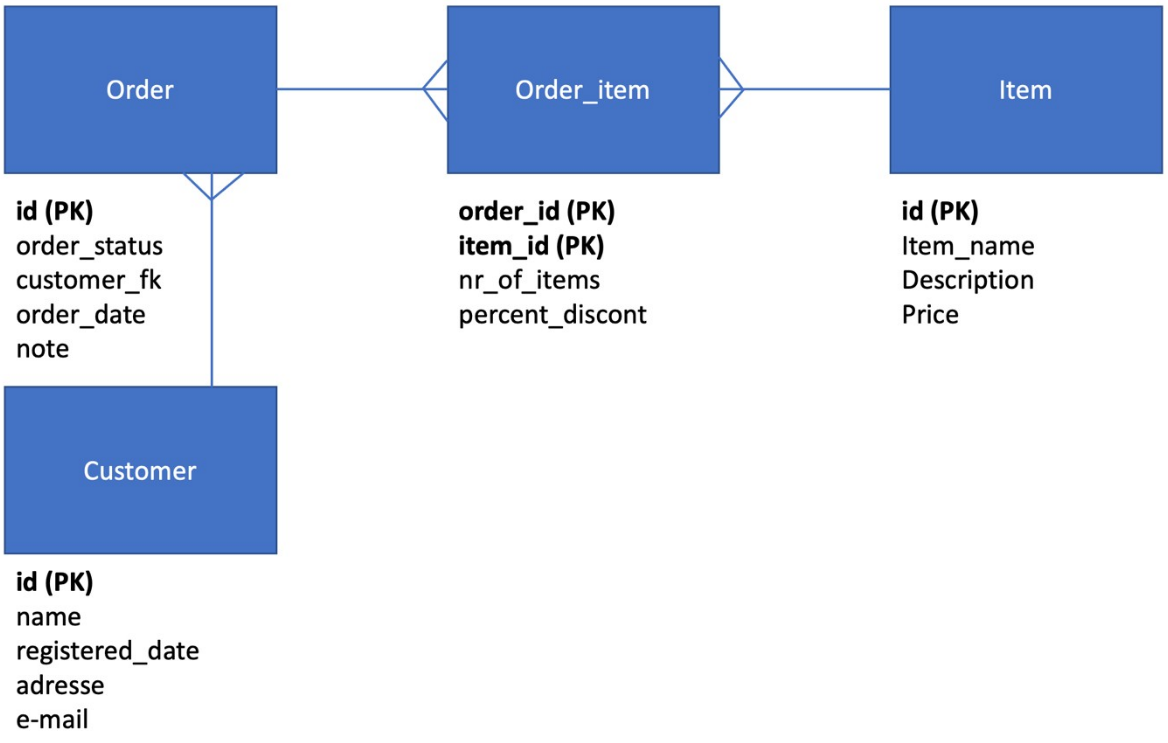
```java
@Override
public final void doTag() throws JspException, IOException {
    PageContext pageContext = (PageContext) getJspContext();
    JspWriter out = pageContext.getOut();

    // this code could have been simplified
    StringWriter stringWriter = new StringWriter();
    JspFragment body = getJspBody();
    body.invoke(stringWriter);
    String bodyText = stringWriter.getBuffer().toString();
    switch(type) {
        case "normal":
            out.println(name + ": " + bodyText);
            break;
        case "bold":
            out.println("<b>" + name + ": " + bodyText + "</b>");
            break;
        case "italic":
            out.println("<i>" + name + ": " + bodyText + "</i>");
            break;
    }
}
```

Tag-file (can be solved using if, and…)

```jsp
<%@ tag language="java" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ attribute name="name" type="String"%>
<%@ attribute name="type" type="String"%>
<c:choose>
  <c:when test = "${type == 'normal'}">
    ${name}: <jsp:doBody />
  </c:when>
  <c:when test="${type =='italic'}">
    <i>${name}: <jsp:doBody /></i>
  </c:when>
  <c:otherwise>
    <b>${name}: <jsp:doBody /></b>
  </c:otherwise>
</c:choose>
```

Question 3 – Web APIs and Framework (20% ~ 48minutes)



a) The data model above is designed for a startup web store.
The web applications to be built expects two types of users (Actors):

- Customers: Registering (creating a customer account), Ordering items
- Company employees: Handling customers, items and orders

Task: Create the URI path design for the needed REST web services for both types of users (actors). The URI path design should include:

- **HttpMethod**
- **URI (from root, shown in example above)**
- **A short description of the REST web service**

Notes!

- **You do NOT need to add filters (except for the path variables such as {id} shown in the example below).**
- **The URI path design should follow level 2 in the Richardson Maturity model.**

Examples:

Method	URI	Description
GET	/items	List all item in the store

GET	/items/{id}	Get an item in the store (given by the item id)

Solution 3a:

Method	URI	Description
GET	/items	List all items in the store
GET	/items/{id}	Get item in the store (given by the item id)
POST	/items	Create a new item
PUT	/items/{id}	Update an item
DELETE	/items/{id}	Delete an item
GET	/customers	Get all customers
GET	/customers/{id}	Get a customer given by an customer id
POST	/customers	Create a new customer
PUT	/customers/{id}	Update a customer
DELETE	/customers/{id}	Delete a customer (given by customer id)
GET	/customers/{id)/orders	Get all orders for a customer
GET	/customers/{id}/orders?status={status}	Get all orders for a customer with a give status (NOT ASKED TO ADD FILTERS LIKE THIS)
GET	/customers/{id}/orders/{oid}	Get an order for a given customer (given by customer id and order id)
POST	/customers/{id}/orders	Create a new order for a customer
PUT	/customers/{id}/orders/{oid}	Update an order for a customer
DELETE	/customers/{id}/orders/{oid}	Delete an order for a customer
GET	/orders	Get all orders
GET	/orders/{id}	Get an order (given by id) including items
GET	/orders?status={status}	Get all orders with a given status (NOT ASKED TO ADD FILTERS LIKE THIS)
POST	/orders	Create a new order
PUT	/orders/{id}	Update an order
DELETE	/orders/{id}	Delete an order

GET	/orders/{id}/items	Get items in order (NOT REALLY NEEDED, items could be included in /orders/{id})
GET	/orders/{id}/items/{iid}	Get details for an item in an given order
POST	/orders/{id}/items	Add an item to an order
PUT	/orders/{id}/items/{iid}	Update an item in an order
DELETE	/orders/{id}/items/{iid}	Delete an item from an order

b) A skeleton for a REST controller class for handling "Customers" is already started on (see below). The project is using the Spring Framework with both Spring Boot and Spring Web MVC.

```
@RestController
@RequestMapping(CustomerController.BASE_URL)
public class CustomerController {

  final static String BASE_URL = "/customers";

  @Autowired
  private CustomerRepositoryImpl repository;

  <you need to write the controller methods below>
}
```

The controller class is using a "CustomerRepositoryImpl" class for accessing data. The repository class implements the following interface:

```
public interface CustomerRepository {
  List<Customer> findAll();
  Optional<Customer> findById(Long id);
  Customer save(Customer customer);
  void delete(Customer customer);
  List<Order> findCustomerOrdersByStatus(
                        Long customerId, String status);
}
```

Task: Write the controller methods for the functionalities described in the list below. The methods should return a "ResponseEntity" and include proper HTTP statuses.

1) **Create a new customer.**
   Statuses: Status OK (200) or Bad Request (400).
2) **Get a customer given by the customer id (primary key).**
   Statuses: OK (200) or Not Found (404).
3) **Get all orders with a given status for a given customer.**
   Statuses: OK (200) or Not Found (404).

Solution 3b: (suggestion for what is expected):

1. Create a new customer.

```
@PostMapping
public ResponseEntity<Customer> new(Customer customer) {
   URI location = null;
   Customer newCustomer = null;
   try {
      newCustomer = repository.save(customer);
      location = new URI("/customers/" + newCustomer.getId());
   } catch (Exception e) {
      return ResponseEntity.badRequest().build();
   }
   return ResponseEntity.created(location).body(newCustomer);
}
```

2. Get a customer given by the customer id (primary key).

```
@GetMapping("/{id}")
public ResponseEntity<Customer> getCustomer(@PathVariable("id") Long id) {
   Optional<Customer> customer = repository.findById(id);
   return customer
.map( c -> ResponseEntity.ok().body(c) )
.orElseGet( () -> ResponseEntity.notFound().build() );
}
```

3. Get all orders with a given status for a given customer.

```
@GetMapping("/{id}/orders")
public ResponseEntity<List<Order>> getCustomerOrdersByStatus (
                            @PathVariable("id") Long id,
                            @RequestParam("status") String status) {

   List<Order> orders = null;

   if (status != null && !"".equals(status)) {
      orders = repository.findCustomerOrdersByStatus(id, status);
   }
   if (orders == null || orders.size() == 0) {
return ResponseEntity.notFound().build();
   }

   return ResponseEntity.ok().body(orders);
}
```

Question 4 – Universal Design (10% ~ 24 minutes)

a) Who is WCAG 2.1 (Web Content Accessibility Guidelines) meant for and why?

Solution:

WCAG is a stable, referenceable technical standard that is primarily intended for (importance in order):

- Web page/content developers (page authors, site designers, etc.)
- Web authoring tool developers
- Web accessibility evaluation tool developers
- Others who want or need a standard for web accessibility

WCAG is developed through the W3C process in cooperation with individuals and organizations around the world, with a goal of proving a single shared standard for web content accessibility that meets the needs of individuals, organizations, and governments internationally. The WCAG documents explain how to make web content more accessible to people with disabilities.

b) What does WCAG 2.1 cover?

Solution:

It is organized so that it covers for four principles: Perceivable, Operable, Understandable and Robust – with Guidelines given for each at three (A, AA, AAA) levels.

c) One of the five principles in WCAG 2.1 is “Operable” (mulig å betjene). Discuss briefly how to meet this principle. Also, give and explain an examples of failure to adhere to this principle.

Solution:

See: https://www.w3.org/TR/WCAG21/#operable

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

a) **Part 1 -** Multiple choice (Some questions may contain multiple correct answers. In this case, choose all the correct options)

1) Can it be a serious security risk to rely only on client-side validation of critical functions?
   1. No, because we can make the validation strong enough and enable all security policies
   2. Yes, because it is possible to inject untrusted data from the client-side
   3. No, because the client can validate the response from the server
   4. Yes, because anything coming from the client should not be trusted

2) A user has requested to reset his/her password and received the password in plaintext by email. In what possible way was the user's password stored in the database?
   1. stored in plaintext
   2. stored using a two-way encryption
   3. stored using a one-way hash function

3) In a single sign on (SSO) solution:
   1. [✓ An identity provider is responsible for issuing and managing authentication tokens]
   2. A service provider is responsible for issuing and managing authentication tokens
   3. [✓ Service providers must use the certificate from the identity provider to verify authentication tokens]
   4. A service provider must issue an authentication token to the identity provider which is verified by the identity provider

4) A web service uses a combination of access and refresh tokens. The access token has expired. Is it possible to use any refresh token to request for a new access token?
   1. Yes, if there is an association between the access and refresh tokens
   2. [✓ No, if there is an association between the access and refresh tokens]
   3. No, if there is no association between the access and refresh tokens
   4. [✓ Yes, if there is no association between the access and refresh tokens]

5) A Json Web Token (JWT) contains a header, payload and signature. Which part of the token is usually signed?
   1. The header and signature
   2. The payload
   3. The header, payload, and signature
   4. [✓ The header and payload]

Part 2

b) Consider the code snippet below. The sql query is vulnerable to SQL injection. The ***asckey*** corresponds to either ‘shoppingitem’ or ‘datetime’ column in the ShoppingCart table. Write a mitigation for this code.

```
public List<Cart> doSearch(String userid, String asckey) throws SQLException, NoSuchAlgorithmException {

    List<Cart> items = new ArrayList<Cart>();

     try {

        String sqlQuery = "select * from ShoppingCart where userid=’"+userid+"’ ORDER BY “+asckey+” ASC";

        Connection conn = Db.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sqlQuery);

        if (!rs.next()) {
            throw new SecurityException("UserId incorrect");
        } else {
          // proceed and store result in items
        }

      } finally {
         try {
           conn.close();
         } catch (SQLException x) {
            //
        }
    }
    return items;
}
```

Answer:

1. perform input validation on the **asckey** to make sure it can only be either **shoppingitem or datetime** column

2. Rewrite the query such that the userid is parameterized and the validated column from (1) is included in the query.

1. Input validation (example – other correct validation techniques are accepted)

```
String sql = "SELECT * FROM ShoppingCart WHERE userid = '" + userid";
If(asckey.equals("shoppingitem") || asckey.equals("datetime")){
    sql = "SELECT * FROM ShoppingCart WHERE userid = ? ORDER BY asckey ASC";
} else {
  // log/throw exception – invalid input
}
```

2. Then use **preparedstatement**

```
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, userid);
ResultSet rs = pstmt.executeQuery();
```

c) Inspect the two code snippets (snippet1 and snippet2) below.

1. Are they vulnerable to XSS?
   Answer: YES
2. If no, justify your answer.
3. If yes, justify your answer and what is/are the type(s) of XSS?

Answer:

Both are vulnerable to XSS.

Snippet 1 is vulnerable to XSS because an untrusted input is stored in the browser's cookie. No validation is shown to have been performed to this data and no output encoding was performed at the sink (getWriter).

Snippet 2 is similarly vulnerable to XSS since the sink (getWriter) output untrusted data from the source (getParameter) with no validation and proper output encoding. Although, a mitigation is used to filters js <script> tags from the data, this mitigation is weak as other forms of injection attack vectors can bypass the control. Example is using <scr<script>ipt>

Types of XSS

Snippet 1: Stored XSS (Because the XSS persists in the local storage.)

Snippet 2: Reflected XSS

```
private void snippet1(HttpServletRequest request, HttpServletResponse response) throws Throwable
{
        String data = “”;

        Cookie cookies = request.getCookies();

        if (cookies != null)
            data = cookies[0].getValue();

        if (data != null)
        {
            response.getWriter().println("<br> data = " + data);
        }

}
```

```
private void snippet2(HttpServletRequest request, HttpServletResponse response) throws Throwable
{
        String data;

        data = request.getParameter("name");

        if (data != null)
        {
            response.getWriter().println("<br> data = " + data.replaceAll("(<script>)", ""));
        }

}
```

d) Describe the different approaches to store passwords.

Answer:

Passwords can be stored in the database as

1. Plaintext
2. Encrypted (two-way): Encrypt the plaintext password with an encryption algorithm with a secret key before storing it in the database.
3. Hashed (one-way): Compute a hash value of the plaintext using a cryptographic hash function and store the hash in the database.
   3.1 Hashed + salt: Append a cryptographic random salt to the password and hash both. Store the hashed password and salt in the database.

Slow (Hash) + salt: In addition to 3.1, slow hash (e.g., PBKDF, scrypt) introduces work factor (computational cost) and thus increases the length of time it takes to crack a given password.

1. Among the approaches you have discussed, which approach is the best to protect against various attacks and why?
   Answer:

   Best approach: Slow hash + salt provides the **best way to defend against password attacks** (brute-force, dictionary, and rainbow table).

   Why?

Hash function makes it harder to recover password. Salt increases the complexity and strength of the password and makes two similar passwords dissimilar. Slow hash aims to increase the computational time it takes to crack a given password.

2. What are the challenges with the other approaches?
   Answer:

   If the password database is stolen: for plaintext, there is no protection as all password is open for the attacker. With two-way encryption, the secret key used to encrypt the password has to be kept. Besides, the secret key is also known to a third party (e.g., the admin), thus exposing it to internal threat. One-way hashing without salt produces the same hash for users with the same passwords and make many users vulnerable to cracked password. Hashing without work factor reduces the time it takes for attackers to crack password.
3. Describe briefly the attack techniques against password?

   Answer:

   Brute-force attack: Check every possible character combination in the password search space.

   Dictionary attack: Check password against a pre-compiled list (dictionary) of common dictionary words, leaked passwords, and commonly used passwords

   Rainbow attack: Check the hash of password against a pre-computed list of hashes of a password dictionary.

e) In the web security obligatory assignment (Oblig3), the service provider (DAT152BlogApp) recieves an authentication token (id_token) from the identity provider (DAT152WebSearch). The id_token is a Json web token (see example below) with some claims and it is used to grant access to the blog website and determine the role (privilege) of the user.

HEADER:
```
{
  "alg": "RS256"
}
```
PAYLOAD:
```
{
  "iss": "http://localhost:9092/DAT152WebSearch",
  "sub": "47544B959729E79BDCA8766A87A8971C",
  "aud": "http://localhost:9091/blogapp/callback",
  "iat": 1632243868,
  "exp": 1633243868,
  "role": "user"
}
```
SIGNATURE:

Answer the following questions:

1. Explain clearly how to manipulate the id_token such that it can be exploited to elevate the privilege of the user.
   Answer:
   Two things must be done to manipulate the token when intercepted by a proxy or in a cookie (from the browser storage).

   1. Make the following changes in the decoded token:
   - algorithm must be changed to “none”
   - role must be changed to admin

```
HEADER:
{
  "alg": "none"
}
PAYLOAD:
{
  "iss": "http://localhost:9092/DAT152WebSearch",
  "sub": "47544B959729E79BDCA8766A87A8971C",
  "aud": "http://localhost:9091/blogapp/callback",
  "iat": 1632243868,
  "exp": 1633243868,
  "role": "admin"
}
```

2. Base64 encode the token and remove the signature part but keep the dot. For the attck to work, the new JWT will look like:

    JWT = Header.Payload.

2. If the attack in 1. would succeed, what vulnerability will be present in the service provider endpoint?

    Answer:
    For the vulnerability to succeed, it is either that
    - the SP does not explicitly verify the signature OR

    the SP uses (misuses) a JWT parser that allows unsigned token to be parsed. (e.g., using the Jwts – **parse**(jwt) instead of **parseClaimsJws**(jwt))

f) Describe briefly how the 'trust' relationship between an identity provider and a service provider is established.

Answer:

The service provider (SP) must know with certainty that the token or claims are issued by the identity provider (IdP) and has not been tampered with by an adversary.

This trust is established through **cryptography keys (Asymmetric or symmetric)** as follows:

1. The IdP signs the token or claims with its private key.
2. The SP uses the public key (certificate) of the IdP to verify both the authenticity and integrity of the token or claims.

If symmetric, then the secret key must be shared securely between the IdP and SP before the transactions.

Question 6 – JavaScript (20% ~ 48 minutes)

a) A web application includes a file *util.js* with the following code:

```
export default {
    // Returns a number, length in feet
    // Input parameter is a number, length in metres
    metrestofeet(metres) { /* JavaScript code */ },

    // Returns a number, length in metres
    // Input parameter is a number, length in feet
    feettometres(feet) { /* JavaScript code */ },

    // Returns a string, the name of unit in languge
    // Input parameters are strings
    // Parameter unit is a unit name in "en-US"
    unitInLanguage(unit, language) { /* JavaScript code */ }
}
```

Observe, you are not asked to implement any of the the code of *util.js*.

i. What are the consequences of the *export default* statement at the beginning of the file?

Solution:

Imports and exports lets JavaScript handle its dependencies.

A module can have zero or one *export default* and this tells that this will be imported from the module if we do not name/specify what to import.

ii. Demonstrate how to import *utils.js* from another JavaScript module, and how to call the method *metrestofeet* of *utils.js* from that module. You can choose the file folder structure as you wish.

Solution:

The folder structure for the application is chosen as:

```
webapp (root directory)
  ├─index.html (file)
  └─js (directory)
     ├─main.js (file)
     └─modules (diretory)
        └─utils.js (file)
```

The file *index.html* loads as a module *main.js* that loads *utils.js*. The code below demonstrates how *main.js* loads *utils.js* and calls *metrestofeet* of *utils.js*.

```
import utils from './modules/utils.js';

const metres = 10;
const feet = utils.metrestofeet(metres);
```

```
console.log(feet);
```

b) An HTML custom tag **LENGTH-READER** adds a GUI component to the web application that lets the user input a length value.

The custom tag is based on an HTML **TEMPLATE** element that is determined by the current browser language settings. The HTML code below shows the **TEMPLATE** elements to be used for the languages “en-GB” and “nb-NO”:

```
<template data-name="length-reader" data-language="en-GB">
  <fieldset>
    <legend>Length in <span data-units></span></legend>
    <input type="number" value="" min="0" placeholder="Enter length"/>
  </fieldset>
</template>

<template data-name="length-reader" data-language="nb-NO">
  <fieldset>
    <legend>Lengde i <span data-units></span> </legend>
    <input type="number" value="" min="0" placeholder="Angi lengde"/>
  </fieldset>
</template>
```

If no **TEMPLATE** element corresponds to the current browser language setting, the application should default to the first **TEMPLATE** element with attribute *data-name* set to “length-reader”. If the browser does not report a language, the language should default to “en-GB”.

The HTML code below demonstrates how to use the custom tag **LENGTH-READER**:

```
<length-reader data-units="foot"></length-reader>
```

For language “en-GB”, the HTML code above should produce the view as is shown in Figure 1.

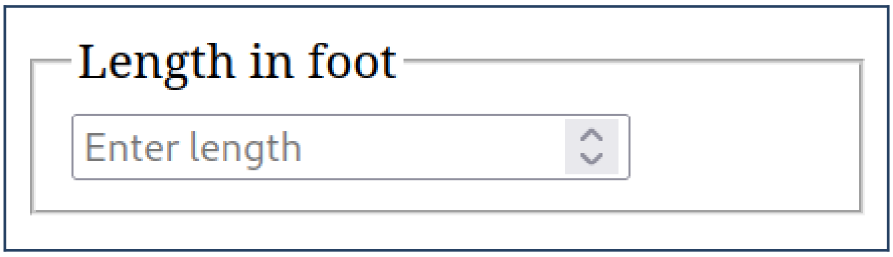


Figure 1: Length-reader in language “en-GB”

For language “nb-NO” the HTML code above with **LENGTH-READER** should produce the view as is shown in Figure 2.

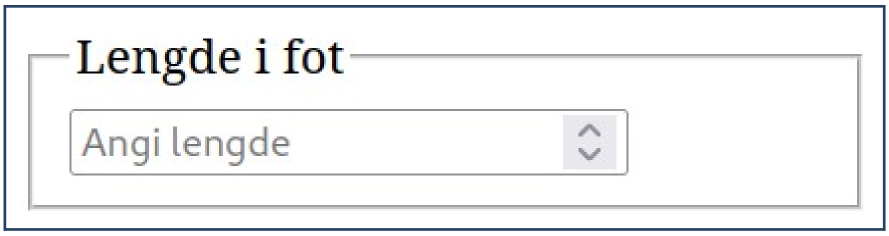


Figure 2: Length-reader in language “nb-NO”

The HTML element class of **LENGTH-READER** has two public methods only:

- *setValue(length)*: The method will set *length* as the value of the HTML **INPUT** element, i.e. display the value.

  Parameters:

  - In parameter: Number
  - Return value: None
- *addCallback(callback)*: The method will add a callback that is run at event *input* on the HTML **INPUT** element.

  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 at event *input*, it must be run with the current value of the HTML **INPUT** element as parameter.

i. What is shadow DOM, what are the consequences of using shadow DOM and what differentiates the *open* and *closed* modes of shadow DOM?

Solution:

A shadow DOM is independent of, and invisible from the main document. It let us isolate a GUI component DOM tree from the main document. Shadow DOM let us keep the markup structure, style, and behavior hidden and separate from other code on the page so that different parts do not clash.

In open mode JavaScript written in the main page context can access the shadow DOM. This is not possible in closed mode. Close mode though does not guarantee that all access is denied from the outside. A public property into the closed shadow DOM will open access from the outside.

ii. Use shadow DOM in closed mode to implement a JavaScript HTML element for the custom tag **LENGTH-READER** as described above.

Solution:

```
import utils from '../../js/modules/utils.js';

class LengthReader extends HTMLElement {
    #shadow;
    #callbacks = new Map();
    #callbackId = 0;

    constructor() {
        super();

        let language = navigator.language;
        if (language === undefined) {
            language = "en-GB";
        }

        this.#shadow = this.attachShadow({ mode: 'closed' });
        let template = document.querySelector(
            `[data-name="length-reader"][data-language="${language}"]`
        );
        if (template === null) {
            template = document.querySelector('[data-name="length-reader"]');
        }
        if (template === null) return;
```

```
            const content = template.content.cloneNode(true);
            this.#shadow.appendChild(content);
            if (this.hasAttribute('data-units')) {
                let unit = this.getAttribute('data-units').trim();
                unit = utils.unitInLanguage(unit, language);
                const elmUnits = this.#shadow.querySelector("span[data-units]");
                elmUnits.textContent = unit;
            }

            this.#shadow.querySelector("input").addEventListener(
                "input", this.#oninput.bind(this)
            );
        }

        addCallback(callback) {
            this.#callbacks.set(this.#callbackId, callback);
            const prevId = this.#callbackId;
            ++this.#callbackId;
            return prevId;
        }

        #oninput() {
            const value = this.#shadow.querySelector("input").value;
            this.#callbacks.forEach(
                callback => { callback(value) }
            );
        }

        setValue(length) {
            this.#shadow.querySelector("input").value = length;
        }
    }
```

You can assume a method *unitInLanguage* of the JavaScript module *util.js*. This method lets you convert a unit name from language “en-US” to any language, e.g. “foot” to “fot” if “nb-NO”, “pied” if “fr-FR” or “πόδι” if “el-GR”.

Tip: See the appendix.

c) A GUI component has functionality to convert between lengths specified in metres and lengths specified in feet. The GUI component can be added to the web application with the HTML custom tag **LENGTH-CONVERTER**.

The custom tag **LENGTH-CONVERTER** is based on the HTML **TEMPLATE** element that is shown below:

```
<template data-name="converter">
    <length-reader data-units="meter"></length-reader>
    <length-reader data-units="foot"></length-reader>
</template>
```

The HTML code below demonstrates how to use the custom tag **LENGTH-CONVERTER**:

```
<length-converter></length-converter>
```

With the browser language configured as “en-GB”, and after user input for a length in feet, the HTML code above should produce the view as is shown in Figure 3.

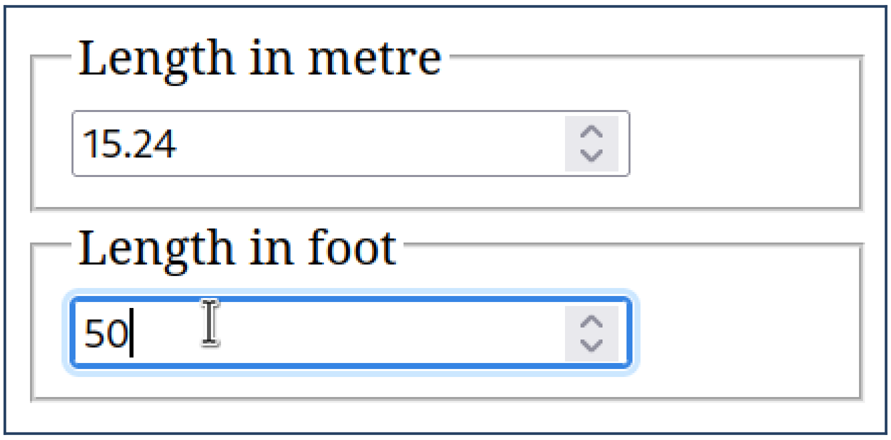


Figure 3: Metric-imperial length converter in language "en-GB" after user input for length in foot

Observe: On user input in one of the **LENGTH-READER** elements, the corresponding value should immediately be calculated and displayed in the other **LENGTH-READER** element. That is, if the user inputs a length i metres, the length in feet should immediately be shown. Similarly, an input in feet should immediately show the corresponding length in metres.

You can assume the methods *metrestofeet* and *feettometre* of the JavaScript module *util.js*. These methods let you convert between feet and metres.

i. Use shadow DOM in closed mode to implement a JavaScript HTML element for the custom tag **LENGTH-CONVERTER** as described above.

Solution:

```
import '../lengthreader/main.js';
import utils from '../../js/modules/utils.js';

class LengthConverter extends HTMLElement {
    #shadow;
    #metricElement;
    #imperialElement;

    constructor() {
        super();

        this.#shadow = this.attachShadow({ mode: 'closed' });
        const template = document.querySelector('[data-name="converter"]');
        if (template === null) return;

        const content = template.content.cloneNode(true);
        this.#shadow.appendChild(content);
        this.#metricElement = this.#shadow.querySelector(
            'length-reader[data-units="meter"]'
        );
        this.#metricElement.addCallback(this.#metricinput.bind(this));
        this.#imperialElement = this.#shadow.querySelector(
            'length-reader[data-units="foot"]'
        );
        this.#imperialElement.addCallback(this.#imperialinput.bind(this));
    }

    #metricinput(input) {
        const value = input.trim();

        if (value === "") {
                this.#imperialElement.value = "";
            } else {
                const meters = parseFloat(value);
                let feet = utils.metrestofeet(meters);

                /* Workaround for Firefox, something is weird with the number
                   input element for string value 0.000 */
                if (feet == 0) {
                    this.#imperialElement.setValue("0");
                } else {
                    this.#imperialElement.setValue(feet.toPrecision(4));
                }
            }
        }

        #imperialinput(input) {
            const value = input.trim();

            if (value === "") {
                this.#metricElement.value = "";
            } else {
                const floatvalue = parseFloat(value);
                let meters = utils.feettometres(floatvalue);

                /* Workaround for Firefox, something is weird with the number
                   input element for string value 0.000 */
                if (meters == 0) {
                    this.#metricElement.setValue("0");
                } else {
                    this.#metricElement.setValue(meters.toPrecision(4));
                }
            }
        }
    }
```

ii. The HTML file of the converter is named *index.html*, and all JavaScript code of the application is included as JavaScript modules, using *import* and *export* statements.

- What JavaScript files would should use for the application, and in what files would you put the *import* and *export* statements? Remember also to import the file *utils.js*.

  Solution:

  The files and folders of a possible solution is:

```
webapp (root directory)
  ├─index.html (file)
  ├─components (diretory)
  │   ├─lengthconverter (diretory)
  │   │   └─main.js (file)
  │   └─lengthreader (diretory)
  │       └─main.js (file)
  └─js (directory)
      ├─appcontroller.js (file)
      └─modules (diretory)
          └─util.js (file)
```

  I have chosen to define the custom tags in the corresponding component module. The component modules are therefore imported only for their side effects, and no export is required of the component classes.

  The file *index.html* loads *appcontroller.js* that loads the **LENGTH-CONVERTER** component.

```
import '../components/lengthconverter/main.js';
```

The **LENGTH-CONVERTER** component imports the **LENGTH-READER** component and the *utils.js* module.

```
import '../lengthreader/main.js';
import utils from '../../js/modules/utils.js';
class LengthConverter extends HTMLElement { ... }
```

Also the **LENGTH-READER** component needs the *utils.js* module.

```
import utils from '../../js/modules/utils.js';
class LengthReader extends HTMLElement { ... }
```

- Show the HTML **SCRIPT** tag(s) with all attributes and values that are necessary to include the JavaScript code of the application. Where in the file *index.html* would you put the HTML **SCRIPT** tag(s)? You must explain your answer.

  Solution:

  The SCRIPT tag of the *index.html* file loads *appcontroller.js* as a module. This is required for *appcontroller.js* to use the import statement:

  ```
  <script src="js/appcontroller.js" type="module"></script>
  ```

  The *type="module"* implies *defer*. The **SCRIPT** tag is therefore put in the **HTML** head to start the loading and parsing of the JavaScript sooner. If *defer*, the browser can download the HTML and JavaScript in separate I/O-threads, and build the DOM structure while downloading the JavaScript code.

- Demonstrate how to use the *customElements.define* method to create the custom tags **LENGTH-READER** and **LENGTH-CONVERTER**. In what files would you put the call to *customElements.define*? You must explain your answer.

  Solution:

  The are different options for where to put the *customElement.define* code.

  - It can be put together with the code that defines the component. Observe then that the export of the component class is not necessary.
  - The *customElement.define* code can also be put together with the code that uses the HTML tag. The same file that imports a component then assigns an HTML tag to the component.
  - Other JavaScript modules need access to the custom tags, not the component class of the tag. A third solution can then be to let the application controller JavaScript code import the components and define the custom tags.

    This solution can require that the tags are created in a certain order, since a custom tag must exist prior to being used.

  Custom tags are global constructs. They belong to the *window* object. Tags are never local to a module. This favors the last option above. The two first options though solve the difficulty of ordering the tag definitions. The component class must exist prior to the custom tag being used. The first two approaches above guarantees that this is fulfilled.

  Observe that a tag, the same or not, can only be defined once on the same component class. Therefor, if several modules import the same component, the code should check whether the tag is already defined. This was not covered by the course.

  With the first option above, we can actually skip the whole class definition if the tag is already defined. The first option is used for the code used of this solution.

  The component for **LENGTH-CONVERTER** is created by the module file *components/lengthconverter/main.js* that then assigns the tag:

  ```
  customElements.define('length-converter', LengthConverter);
  ```

The component for **LENGTH-READER** is created by the module *components/lengthreader/main.js* that then assigns the tag:

```
customElements.define('length-reader', LengthReader);
```

We should also check whether the tag is already defined, but that was not covered in the the course.

Tip: See the appendix.

Appendix

Help for question 1 (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 3

Package org.springframework.http

Class ResponseEntity<T>

java.lang.Object

org.springframework.http.HttpEntity<T>

org.springframework.http.ResponseEntity<T>

Type Parameters:

`T` - the body type

```
public class ResponseEntity<T> extends HttpEntity<T>
```

Extension of `HttpEntity` that adds an `HttpStatusCode` status code. Used in `RestTemplate` as well as in `@Controller` methods.

…

Constructor Details

ResponseEntity

`public ResponseEntity(HttpStatusCode status)`

Create a `ResponseEntity` with a status code only.

Parameters: `status` - the status code

ResponseEntity

`public ResponseEntity(@Nullable T body, HttpStatusCode status)`

Create a `ResponseEntity` with a body and status code.

Parameters: `body` - the entity body, `status` - the status code

ResponseEntity

`public ResponseEntity(MultiValueMap<String,String> headers, HttpStatusCode status)`

Create a `ResponseEntity` with headers and a status code.

Parameters:

`headers` - the entity headers
`status` - the status code

ResponseEntity

```
public ResponseEntity(@Nullable T body, @Nullable MultiValueMap<String,String> headers,
       HttpStatusCode status)
```

Create a `ResponseEntity` with a body, headers, and a status code.

Parameters:

`body` - the entity body
`headers` - the entity headers
`status` - the status code

ResponseEntity

```
public ResponseEntity(@Nullable T body, @Nullable MultiValueMap<String,String> headers,
       int rawStatus)
```

Create a `ResponseEntity` with a body, headers, and a raw status code.

Parameters:

`body` - the entity body
`headers` - the entity headers
`rawStatus` - the status code value

Since: 5.3.2

Method Details

getStatusCode

`public HttpStatusCode getStatusCode()`

Return the HTTP status code of the response.

Returns: the HTTP status as an HttpStatus enum entry

getStatusCodeValue

`@Deprecated(since="6.0") public int getStatusCodeValue()`

Deprecated.

as of 6.0, in favor of `getStatusCode()`

Return the HTTP status code of the response.

Returns: the HTTP status as an int value

Since: 4.3

equals

```
public boolean equals(@Nullable

        Object other)
```

Overrides: `equals` in class `HttpEntity<T>`

hashCode

`public int hashCode()`

Overrides: `hashCode` in class `HttpEntity<T>`

toString

`public String toString()`

Overrides: `toString` in class `HttpEntity<T>`

status

`public static ResponseEntity.BodyBuilder status(HttpStatusCode status)`

Create a builder with the given status.

Parameters:

`status` - the response status

Returns: the created builder

Since: 4.1

status

`public static ResponseEntity.BodyBuilder status(int status)`

Create a builder with the given status.

Parameters:

`status` - the response status

Returns: the created builder

Since: 4.1

ok

`public static ResponseEntity.BodyBuilder ok()`

Create a builder with the status set to OK.

Returns: the created builder

Since: 4.1

ok

`public static <T> ResponseEntity<T> ok(@Nullable T body)`

A shortcut for creating a `ResponseEntity` with the given body and the status set to OK.

Parameters:

body - the body of the response entity (possibly empty)

Returns: the created `ResponseEntity`

Since: 4.1

of

`public static <T> ResponseEntity<T> of(Optional<T> body)`

A shortcut for creating a `ResponseEntity` with the given body and the OK status, or an empty body and a NOT FOUND status in case of an Optional.empty() parameter.

Returns: the created `ResponseEntity`

Since: 5.1

of

`public static ResponseEntity.HeadersBuilder<?> of(ProblemDetail body)`

Create a new `ResponseEntity.HeadersBuilder` with its status set to `ProblemDetail.getStatus()` and its body is set to `ProblemDetail`.

Note: If there are no headers to add, there is usually no need to create a `ResponseEntity` since `ProblemDetail` is also supported as a return value from controller methods.

Parameters: body - the problem detail to use

Returns: the created builder

Since: 6.0

created

`public static ResponseEntity.BodyBuilder created(URI location)`

Create a new builder with a CREATED status and a location header set to the given URI.

Parameters: `location` - the location URI

Returns: the created builder

Since: 4.1

accepted

`public static ResponseEntity.BodyBuilder accepted()`

Create a builder with an ACCEPTED status.

Returns:the created builder

Since: 4.1

noContent

`public static ResponseEntity.HeadersBuilder<?> noContent()`

Create a builder with a NO_CONTENT status.

Returns: the created builder

Since: 4.1

badRequest

`public static ResponseEntity.BodyBuilder badRequest()`

Create a builder with a BAD_REQUEST status.

Returns: the created builder

Since: 4.1

notFound

`public static ResponseEntity.HeadersBuilder<?> notFound()`

Create a builder with a NOT_FOUND status.

Returns: the created builder

Since: 4.1

Interface ResponseEntity.BodyBuilder

All Superinterfaces:

`ResponseEntity.HeadersBuilder<ResponseEntity.BodyBuilder>`

Enclosing class:

`ResponseEntity<T>`

Method Details

contentType

`ResponseEntity.BodyBuilder contentType(MediaType contentType)`

Set the media type of the body, as specified by the `Content-Type` header.

Parameters: `contentType` - the content type

Returns: this builder

body

`<T> ResponseEntity<T> body(@Nullable T body)`

Set the body of the response entity and returns it.

Type Parameters: `T` - the type of the body

Parameters: `body` - the body of the response entity

Returns: the built response entity

Package org.springframework.http

Interface HttpStatusCode

Method Summary

Modifier and Type	Method	Description
`boolean`	`is1xxInformational()`	Whether this status code is in the Informational class (1xx).
`boolean`	`is2xxSuccessful()`	Whether this status code is in the Successful class (2xx).
`boolean`	`is3xxRedirection()`	Whether this status code is in the Redirection class (3xx).
`boolean`	`is4xxClientError()`	Whether this status code is in the Client Error class (4xx).
`boolean`	`is5xxServerError()`	Whether this status code is in the Server Error class (5xx).
`boolean`	`isError()`	Whether this status code is in the Client or Server Error class

`int`	`value()`	Return the integer value of this status code

Enum Class HttpStatus

java.lang.Object

java.lang.Enum<HttpStatus>

org.springframework.http.HttpStatus

All Implemented Interfaces:

Serializable, Comparable<HttpStatus>, Constable, HttpStatusCode

Enum Constants

Enum Constant	Description
`ACCEPTED`	`202 Accepted.`
`BAD_REQUEST`	`400 Bad Request.`
`CREATED`	`201 Created.`
`FORBIDDEN`	`403 Forbidden.`
`FOUND`	`302 Found.`
`INTERNAL_SERVER_ERROR`	`500 Internal Server Error.`
`NO_CONTENT`	`204 No Content.`
`NOT_FOUND`	`404 Not Found.`
`OK`	`200 OK.`
`UNAUTHORIZED`	`401 Unauthorized.`

Help for question 6

The current language, e.g. “nb-NO” is available in the browser as *navigator.language*.

The code below demonstrates how to use an HTML **TEMPLATE** element for a GUI component with shadow DOM.

```
class MyElement extends HTMLElement {
    #shadow;

    constructor() {
        super();
        this.#shadow = this.attachShadow({ mode: 'closed' });
        const template = document.querySelector("template");
        const content = template.content.cloneNode(true);
        this.#shadow.appendChild(content);
    }
}
```

Good Luck!