Western Norway University of Applied Sciences

~~EXAM~~

Solution

Exam code: DAT152

Course name: Advanced Web Applications

Date: November 28, 2024

Type of examination: Written exam

Time: 4 hours (0900 -1300)

Number of questions: 6

Number of pages: 26 (including this page and appendices)

Appendices: The last 3 pages

Exams aids: Bilingual dictionary

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

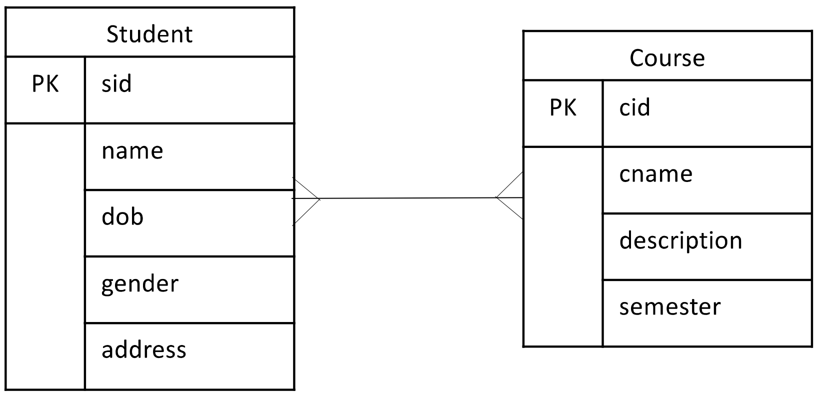
External sensor: Kent Inge Fagerland Simonsen

Notes: None

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

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

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



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

a) Build a REST API endpoints for this service. The REST API should provide the following services:

- Student
  - i. Register/create a profile
  - ii. View details about own profile
  - iii. Update profile information
  - iv. Register for a new course
  - v. Withdraw from a course
  - vi. List all registered courses
  - vii. View details about a registered course
- Staff
  - i. List all courses
  - ii. List all students registered for a course
  - iii. Create a new course
  - iv. Update a course
  - v. Delete a course
  - vi. List all students

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

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

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

Solution

Service	API Method	Resource Endpoint (URI Path)	HTTP Method	Role
StudentController				
Create a new profile	registerProfile	/students	POST	USER
List all students	getStudentsProfiles	/students	GET	ADMIN
View details about own profile	getStudentProfile	/students/{sid}	GET	USER
Update profile information	updateStudentProfile	/students/{sid}	PUT	USER
Register for a new course	registerForCourse	/students/{sid}/courses	POST	USER
Withdraw from a course	withdrawFromCourse	/students/{sid}/courses/{cid}	DELETE	USER
List all registered courses for a student	getAllCoursesByStudentId	/students/{sid}/courses	GET	USER
View details about a registered course	getACourseByStudentId	/students/{sid}/courses/{cid}	GET	USER

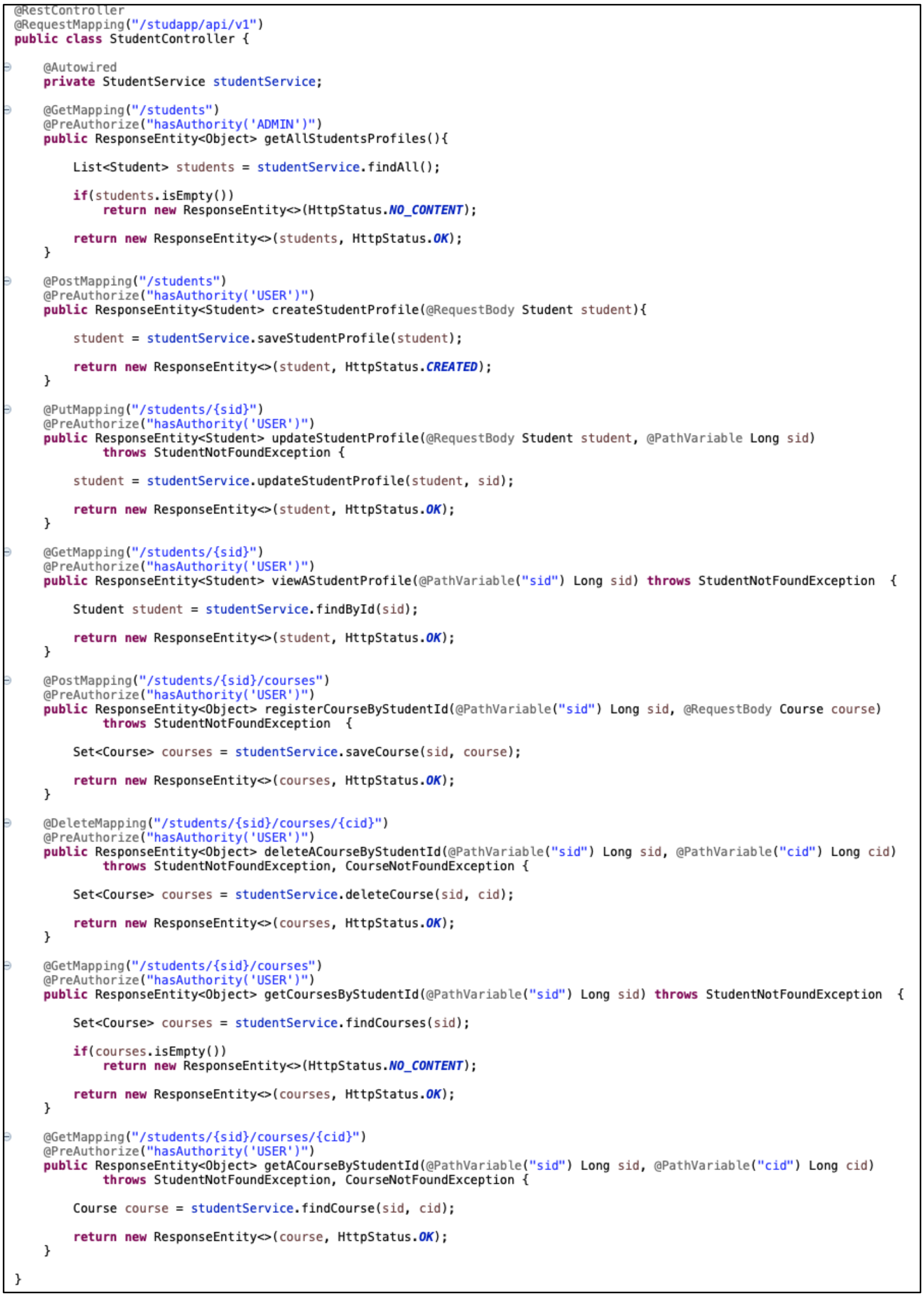
Service	API Method	Resource Endpoint (URI Path)	HTTP Method	Role
CourseController				
List all courses	getCourses	/courses	GET	USER
List all students registered for a course	getAllStudentsReg Course	/courses/{cid}/students	GET	ADMIN
Create a new course	createCourse	/courses	POST	ADMIN
Update a course	updateCourse	/courses/{sid}	PUT	ADMIN
Delete a course	deleteCourse	/courses/{sid}	DELETE	ADMIN

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

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

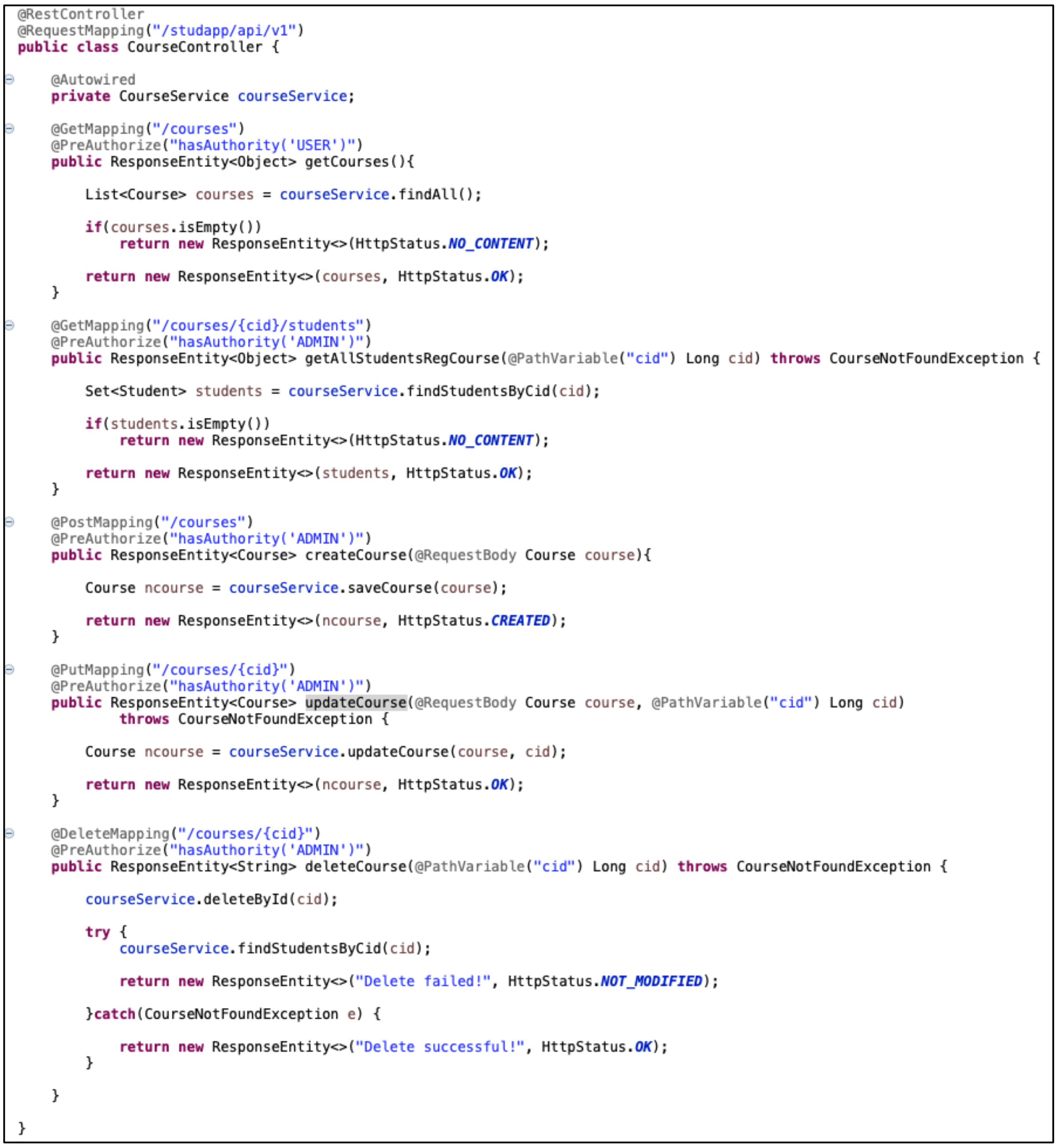
```
@Service
public class CourseService {
    @Autowired
    private CourseRepository courseRepository;
    public Course saveCourse(Course course) {
       return courseRepository.save(course);
     }
     public Course updateCourse(Course course, Long cid) throws CourseNotFoundException {
        findCourseById(cid);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

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



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

    @Autowired
    private StudentService studentService;

    @GetMapping("/students")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity<Object> getAllStudentsProfiles(){

        List<Student> students = studentService.findAll();

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

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

    @PostMapping("/students")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Student> createStudentProfile(@RequestBody Student student){

        student = studentService.saveStudentProfile(student);

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

    @PutMapping("/students/{sid}")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Student> updateStudentProfile(@RequestBody Student student, @PathVariable Long sid)
            throws StudentNotFoundException {

        student = studentService.updateStudentProfile(student, sid);

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

    @GetMapping("/students/{sid}")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Student> viewAStudentProfile(@PathVariable("sid") Long sid) throws StudentNotFoundException  {

        Student student = studentService.findById(sid);

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

    @PostMapping("/students/{sid}/courses")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Object> registerCourseByStudentId(@PathVariable("sid") Long sid, @RequestBody Course course)
            throws StudentNotFoundException  {

        Set<Course> courses = studentService.saveCourse(sid, course);

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

    @DeleteMapping("/students/{sid}/courses/{cid}")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Object> deleteACourseByStudentId(@PathVariable("sid") Long sid, @PathVariable("cid") Long cid)
            throws StudentNotFoundException, CourseNotFoundException {

        Set<Course> courses = studentService.deleteCourse(sid, cid);

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

    @GetMapping("/students/{sid}/courses")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Object> getCoursesByStudentId(@PathVariable("sid") Long sid) throws StudentNotFoundException  {

        Set<Course> courses = studentService.findCourses(sid);

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

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

    @GetMapping("/students/{sid}/courses/{cid}")
    @PreAuthorize("hasAuthority('USER')")
    public ResponseEntity<Object> getACourseByStudentId(@PathVariable("sid") Long sid, @PathVariable("cid") Long cid)
            throws StudentNotFoundException, CourseNotFoundException {

        Course course = studentService.findCourse(sid, cid);

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

}
```

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



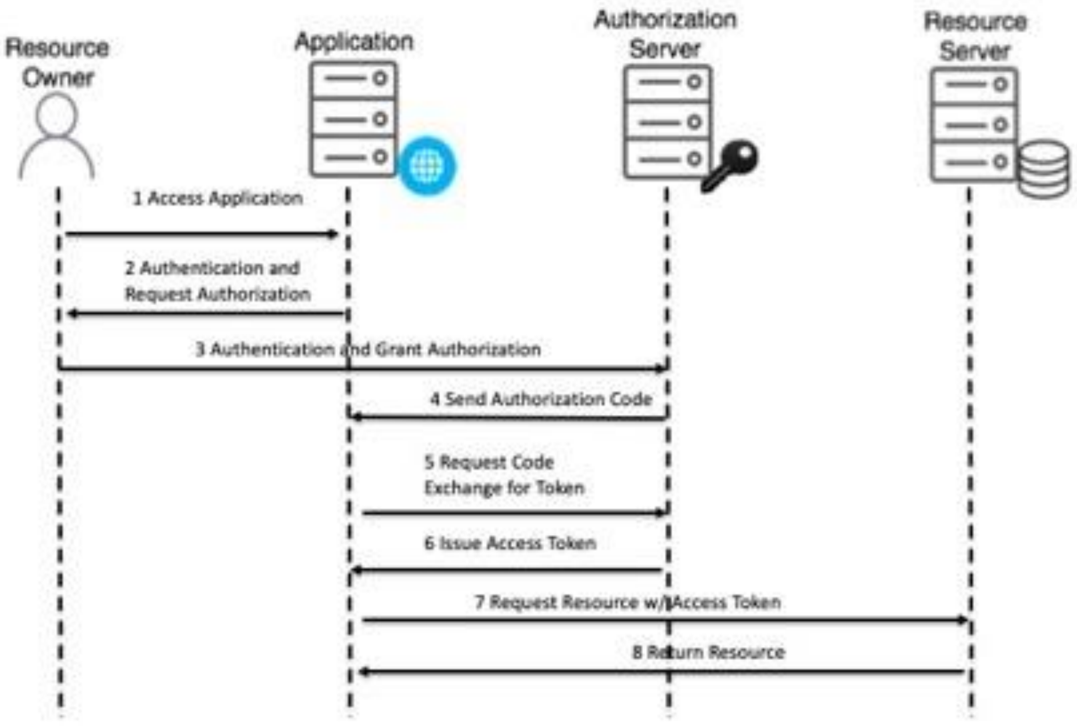
```java
@RestController
@RequestMapping("/studapp/api/v1")
public class CourseController {

    @Autowired
    private CourseService courseService;

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

        List<Course> courses = courseService.findAll();

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

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

    @GetMapping("/courses/{cid}/students")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity<Object> getAllStudentsRegCourse(@PathVariable("cid") Long cid) throws CourseNotFoundException {

        Set<Student> students = courseService.findStudentsByCid(cid);

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

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

    @PostMapping("/courses")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity<Course> createCourse(@RequestBody Course course){

        Course ncourse = courseService.saveCourse(course);

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

    @PutMapping("/courses/{cid}")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity<Course> updateCourse(@RequestBody Course course, @PathVariable("cid") Long cid)
            throws CourseNotFoundException {

        Course ncourse = courseService.updateCourse(course, cid);

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

    @DeleteMapping("/courses/{cid}")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity<String> deleteCourse(@PathVariable("cid") Long cid) throws CourseNotFoundException {

        courseService.deleteById(cid);

        try {
            courseService.findStudentsByCid(cid);

            return new ResponseEntity<>("Delete failed!", HttpStatus.NOT_MODIFIED);

        }catch(CourseNotFoundException e) {

            return new ResponseEntity<>("Delete successful!", HttpStatus.OK);
        }

    }
}
```

c) Assume the studapp 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

Authorization code grant flow


Resource Owner
Application
Authorization Server
Resource Server
1 Access Application
2 Authentication and Request Authorization
3 Authentication and Grant Authorization
4 Send Authorization Code
5 Request Code Exchange for Token
6 Issue Access Token
7 Request Resource w/ Access Token
8 Return Resource

II). Assume that there are two roles 'ADMIN' and 'USER' in the JWT access token where the ADMIN and USER roles are assigned to the staff, and only the USER role is assigned to the students. Protect the REST API endpoints in the StudentController and the CourseController classes using Spring relevant annotations.

Solution

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

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

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

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

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

}

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

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

Question 2 – Globalization (10% ~ 24minutes)

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

Solution

Smple definitions below (Explanations should describe each of the concepts clearly without ambiguity)

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 when implementing a new language.

Localization: the process to adapt application for specific language and regional cultural context, called locale and it also includes translation of text into the target language.

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.

Solution

Property files (Resource bundles)

Message_en_EN.properties
msg=If you live in Bergen, you should expect rain between {0} and {1} days out of {2} days. This is equivalent to roughly {3} of rain in a year.

Message_no_NO.properties
msg=Bor du i Bergen bør du regne med regn mellom {0} og {1} dager av {2} dager. Dette tilsvarer omtrent {3} av regn i løpet av et år

jsp

```
<body>
  <fmt:setLocale value="en_EN"/>
  <fmt:bundle basename="no.hvl.dat152.i18n.Message">
    <fmt:message key="msg">
          <fmt:param><fmt:formatNumber value="200" type="number"/></fmt:param>
          <fmt:param><fmt:formatNumber value="240" type="number"/></fmt:param>
          <fmt:param><fmt:formatNumber value="365" type="number"/></fmt:param>
          <fmt:param><fmt:formatNumber value="0.55" type="percent"/></fmt:param>
    </fmt:message>
  </fmt:bundle>
</body>
```

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

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

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

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

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

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

"26,7C"

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

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

Solution

```
public class FahrToCelcius extends SimpleTagSupport {

    private Double fahr;

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

        PageContext pageContext = (PageContext) this.getJspContext();
        JspWriter writer = pageContext.getOut();

        Double ctemp = 5.0/9.0 * (fahr - 32);
        String cstemp = String.format("%.2f", ctemp);

        writer.println(cstemp);

    }

    /**
     * @return the fahr
     */
    public Double getFahr() {
        return fahr;
    }

    /**
     * @param fahr the fahr to set
     */
    public void setFahr(Double fahr) {
        this.fahr = fahr;
    }

}
```

b) Implement the 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"%>

<%@ attribute name="fahr" type="Double"  %>

<%
    Double ctemp = 5.0/9.0 * (fahr - 32);
    String cstemp = String.format("%.2f", ctemp);
%>

<%=cstemp %>
```

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

a) What is Universal Design?

Solution

Universal Design is "the design of products and environments to be usable by all people, to the greatest extent possible, without the need for adaptation or specialized design."

b) Give 3 principles of UD

Solution

Any 3 from:

Equitable use, Flexibility in use, Simple and intuitive, Perceptible information, Tolerance for error, Low physical effort, Size and space for approach and use.

c) Briefly discuss the WCAG principle #2 – "Operable" and the guidelines for this principle.

Solution

Operable means that the user interface components and navigation must be operable. Users must be able to operate the interface and its controls. The interface cannot require actions that users cannot perform.

Operable has 5 guidelines (WCAG 2.1):

Keyboard Accessible: All functionalities should be available from a keyboard.

Enough Time: Provide users enough time to read and use content.

Seizures and Physical Reactions: Do not design content in a way that is known to cause seizures or physical reactions.

Navigable: Provide ways to help users navigate, find content, and determine where they are.

Input Modalities: Make it easier for users to operate functionality through various inputs beyond keyboard.

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

a) You have saved a session token as a cookie in a web browser storage. What are the secure ways to protect the cookie from XSS vulnerability and session hijacking attack?

Solution

Set HttpOnly flag (HttpOnly = True) to prevent javascript from accessing the cookie
Set Secure flag (Secure = True) to disallow cookie from being sent over non-encrypted communication. In the case of packet sniffing, the session token is then in an encrypted form and protected from unauthorized disclosure.

b) A logout controller of a web application contains the code below.

I). What major session vulnerability is present in this implementation?

Solution

The logout servlet failed to invalidate the session. Although the user attribute is removed from the session, there can still be other attributes that are still valid and can be leaked.

II). Write a mitigation for this vulnerability.

Solution

```
    // invalidate session
    request.getSession().invalidate();
    request.getRequestDispatcher("index.jsp").forward(request, response);
```

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

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

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

c) Given the 2 SQL queries below:
Implementation 1:

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

Implementation 2:

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

I). Which of the two implementations can be vulnerable to SQL injection vulnerability?

Solution
Only implementation 1 is vulnerable to SQLi. It uses a preparedstatement insecurely since untrusted data is added to the query before preparing the query. Implementation 2 is not vulnerable to SQLi because there is no untrusted data in the query.

II). Write the mitigation code for the implementation that you identified as vulnerable.
Solution
Mitigation for implementation 1

```
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM SecOblig.AppUser WHERE USERNAME = ? and PASSWORD = ?");
stmt.setString(1, username);
stmt.setString(2, password);
stmt.executeQuery();
```

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

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 crack 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.

e) The jjwt-api library that we have used in Oblig4 contains two methods for parsing the JWT token. Method #1 – parse(token) and Method #2 – parseClaimsJws(token). Assume that the JWT has been **modified** as below:

I). JWT #1

```
HEADER:
{
      "alg": "none"
}
PAYLOAD:
{
      "iat": 1730803010”,
      "role": "ADMIN",
      "user": "Bob”
}
Encoded JWT =
eyJhbGciOiJub25lIiwia2lkIjoicEp3W.eyJleHAiOjE2OTg5MTA4OTQsImlhdCI6MTY5ODg3NDg5NC.
```

II). JWT #2

```
HEADER:
{
      "alg": "RS256"
}
PAYLOAD:
{
      "iat": 1730803010”,
      "role": "ADMIN",
      "user": "Bob”
}
Encoded JWT =
eyJhbGciOiJSUzI1NiIsImtpZCI6InBKd.eyJleHAiOjE2OTr7ATMQOTQsImlhdCI6MTYAADg3NDgAOP.
```

The results of the code below for parsing JWT #1 and JWT #2 can be any of:
(1) *throw an InvalidTokenException* in line 12.
(2) *invoke the method "deleteAllUsers()"* in line 7.
(3) *logs the error ("You are not an admin user")* in line 9.

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

Solution

throw an InvalidTokenException in line 12. Since the method parseClaimsJws will verify the signature part and the signature was removed in the encoded JWT.

I) JWT #1

Solution

throw an InvalidTokenException in line 12. Since the method parseClaimsJws will verify the signature part and the signature was removed in the encoded JWT.

II) JWT #2.

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

f) In the figure below, there is a trust relationship between the Authorization Server and the Resource Server. Describe briefly how this ‘trust’ relationship between the Authorization Server and the Resource Server is established.

Solution

The Resource Server (RS) must know with certainty that the token it received is issued by the Authorization Server (AS) and has not been tampered with by an adversary.

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

1. The AS signs the token with its private key.
2. The RS uses the public key (certificate) of the AS to verify both the authenticity and integrity of the token.

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

g) In the same figure below, a client can access a protected resource by sending a request together with a JWT token to the “Resource Server”.
Describe the security actions that the “**Authorization Server**” must take before sending the **JWT** token in a response to the client.

Solution

The following mandatory fields must be included by the Authorization Server (AS):

iss: Issuer Identifier for the Issuer of the response (i.e., the AS)

aud: Audience(s) that this ID Token is intended for. It MUST contain the OAuth 2.0 client_id of the Relying Party as an audience value.

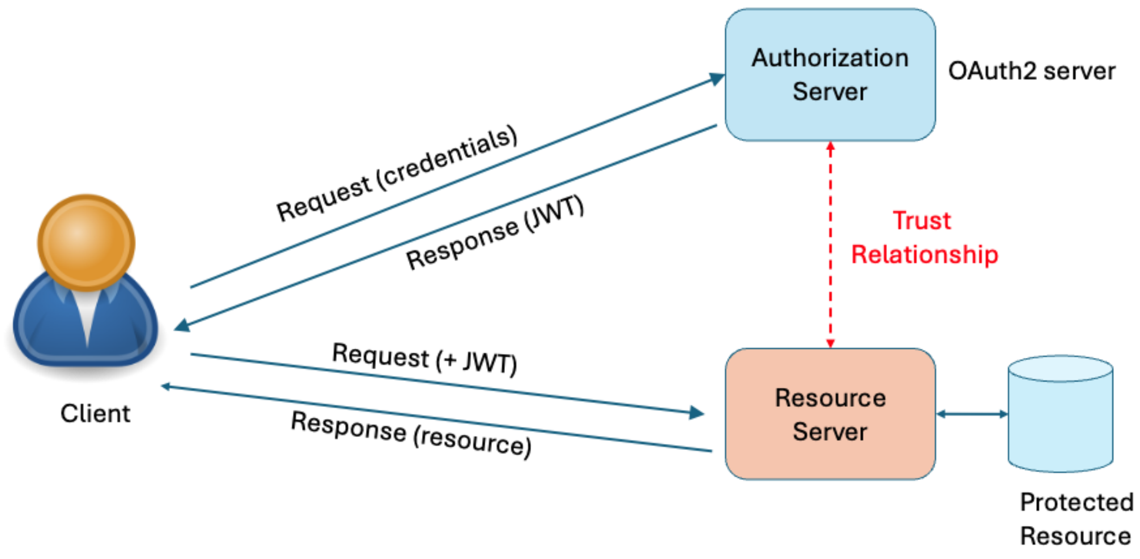
sub: Subject Identifier. A locally unique and never reassigned identifier within the Issuer for the End-User, which is intended to be consumed by the Client

exp: Expiration time on or after which the ID Token MUST NOT be accepted for processing.

iat: Time at which the JWT was issued.

claims: include all attributes required by the resource server (e.g., profiles, access levels, etc)

Signature: Lastly, the token MUST be **signed**.



Question 6 – JavaScript (20% ~ 48 minutes)

a) JavaScript theory.

A shadow DOM can be created by the code below:

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

Code snippet 1: Creating a shadow DOM

i. What is a shadow DOM? Your answer must outline the differences between a “normal” DOM and a shadow DOM.

A shadow DOM is a DOM tree that is a separate from the regular DOM, but attached to an element of the regular DOM.

Shadow DOM internals are hidden from, and invisible from main document.
- The main document CSS and JavaScript does not see the shadow DOM.
- The shadow DOM CSS and JavaScript does not affect the main document.

HTML IDs and styles in shadow DOM do not clash with main document.

ii. In the above example of Code snippet 1, a shadow DOM is created where property *mode* is set to *closed*.

1. What are the consequences if *mode* is set to *open*?

If mode is set to *open*, elements of the shadow DOM are accessible with JavaScript from outside the shadow DOM, using e.g. the property *shadowRoot* of the shadow DOM root element.

2. Show with a code example how to utilize that *mode* is set to *open* for a shadow DOM component.

Assuming a custom element <shopping-list> created as a component with shadow DOM, and mode *open*. If the component includes an HTML <table> element, the code below will delete the <table> element:

```
const element = document.querySelector("shopping-list");
const shadowElement = element.shadowRoot;
const tableElement = shadowElement.querySelector("table");
tableElement.remove();
```

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

1. What is the purpose of the tag <SLOT>?

The HTML <slot> element is a placeholder inside a GUI component that can be replaced with DOM structures.

2. Demonstrate with code examples how to use the <SLOT> tag.

Assume a <course-info> custom element that contains the following HTML:

```
<slot>
    <p>DAT152 exam date to be decided.</p>
```

```
</slot>
```

On use of the <course-info> custom element, the <slot> element will be replaced by the DOM content of the <course-info> element:

```
<course-info>
    <p>DAT152 exam is today.</p>
    <p>DAT151 exam to be decided.</p>
</course-info>
```

The paragraph of the <slot> element of the component will then be replaced by two paragraphs.

If the component has a multiple HTML <slot> elements, each <slot> element need a *name* attribute, corresponding to a *slot* attribute on the element that should replace the <slot>.

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

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

Code snippet 2: View produced by a shadow DOM component

The course code is decided by the parent component.

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

For the below demonstrations, the custom tag <course-info> is assumed for the component.

1. The <span> element with the course code can be put inside a <slot> tag, and the parent component can then set the course code by replacing the <slot> element with e.g. a <span> element that contains the course code.

   ```
   <course-info>
       <span slot="course">DAT152</span>
   </course-info>
   ```

2. The course code can be given as content of the custom tag:

   ```
   <course-info>DAT152</course-info>
   ```

   The component can get the course code using e.g. the HTML element property *textContent*:

   ```
   connectedCallback(){
       const spanElm = this.#shadow.querySelector("span");
       spanElm.textContent = this.textContent;
   };
   ```

3. The course code can be given as an HTML attribute when using the custom tag of the component:

   ```
   <course-info data-course="dat152"></course-info>
   ```

The component can get the course code using e.g. the element method *getAttribute*, or the *dataset* property:

```
connectedCallback(){
    const spanElm = this.#shadow.querySelector("span");
    spanElm.textContent = this.dataset.course;
};
```

4. The course component can have an API that lets the parent set the course code, e.g.:

```
setCourse(course) {
    const spanElm = this.#shadow.querySelector("span");
    spanElm.textContent = course;
}
```

The parent component can then use the API to set the course code:

```
const courseElm = document.querySelector("course-info");
courseElm.setCourse("DAT152");
```

The code below is found in a web application:

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

Code snippet 3: Code of web application

v. What kind of object must be returned by the call to *storagemanager.remove(id)*? Describe this kind of object.

The *await* keyword can only be used before a **Promise** (or a promise like object). If the method after *await* does not return a **Promise** by itself, the browser will encapsulate the return value in a **Promise** that fulfils immediately.

A **Promise** is usually used together with asynchronous code, code that eventually in the future will fulfil (resolve) of fail (reject).

Using *then*, a **Promise** can be given a method to be run if the **Promise** fulfils (resolve).:

```
storagemanager.remove(id).then((result) => {...})
```

Using *catch*, or adding a second method to *then*, the **Promise** can be given a method to run if the **Promise** fails (reject).

The async/await syntax can make the code easier to read. The code following the *await* will be the *then* code. If using try/catch, the catch part will be the **Promise** *catch* code.

If the **Promise** completes (resolve), the *await* keyword will return the resolved value, i.e. *result* above will be the resolved value from the **Promise**.

b) In this task you are asked to write the JavaScript code of a GUI component **ShoppingList** that can manage the display of a shopping list.

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

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

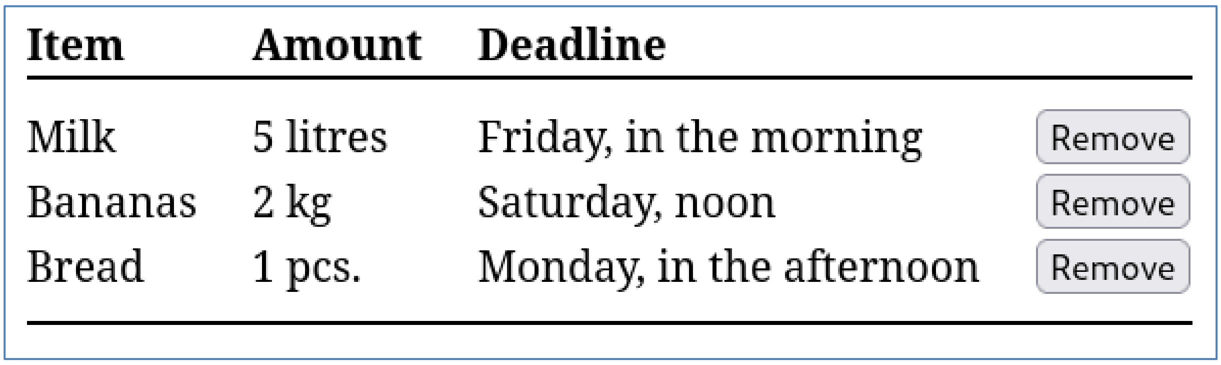
```
const template = document.createElement("template");
template.innerHTML = `
    <table id="shoppinglist">
        <thead>
            <tr><th>Item</th><th>Amount</th><th>Deadline</th></tr>
        </thead>
        <tbody></tbody>
    </table>`;
```

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

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

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

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



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

*Figure 1: The **ShoppingList** component*

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

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

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

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

Code snippet 6: Object representing the shopping item Bananas

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

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

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

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

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

  This method updates the view.

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

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

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

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

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

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

  This method updates the view.

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

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

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

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

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

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

  This method does not update the view.

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

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

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

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

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

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

    constructor() {
        super();

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

        // More code, if necessary
    }

    addremoveitemcallback(callback) {
        // More code
    }

    additem(item) {
        // More code

    }

    removeitem(id) {
        // More code
    }

    // More code, if necessary
}

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

Code snippet 10: JavaScript code of ShoppingList

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

The text in red as the code added Code snippet 10 to above.

```
class ShoppingList extends HTMLElement {
    #callbacks = new Map();
    #shoppinglist;

    constructor() {
        super();

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

        this.#shoppinglist = shadow
            .getElementById("shoppinglist").tBodies[0];
    }

    addremoveitemcallback(callback) {
        const callbackid = Symbol("additemcallback");
        this.#callbacks.set(callbackid, callback);
        return callbackid;
    }

    additem(item) {
        let row = this.#shoppinglist
            .querySelector(`tr[data-id="${item.id}"]`);
        if (row !== null) {
            // Item is already listed in table
            return;
        }

        const content = itemrow.content.cloneNode(true);
        row = content.firstElementChild;
        row.dataset.id = item.id;
        row.dataset.time = item.deadline.getTime();
        row.cells[0].textContent = item.name;
        row.cells[1].textContent = item.amount;
        const bt = row.cells[3]
            .getElementsByTagName("button")[0];
        bt.addEventListener("click",
           () => { this.#onremove(item.id) }
        );
        const deadlineText = item.deadline.toLocaleDateString(
            navigator.language,
            { weekday: "long", dayPeriod: "long" }
        );
        row.cells[2].textContent = deadlineText;

        const index = Array.from(this.#shoppinglist.rows)
            .findIndex(
                (element) =>
                element.dataset.time > row.dataset.time
            );
        const newrow = this.#shoppinglist.insertRow(index);
        newrow.replaceWith(row);
    }
    removeitem(id) {
        const row = this.#shoppinglist
            .querySelector(`tr[data-id="${id}"`);
        if (row !== null) {
            row.remove();
        }
    }

    #onremove(id) {
        this.#callbacks.forEach(callback => callback(id));
    }
}

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

Appendix

Help for question 1 (REST API using Spring Framework)

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

Help for question 2 (JSTL fmt)

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

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

Help for question 6 (JavaScript)

EventTarget: *addEventListener()* method

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

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

Syntax:

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

Node: *textContent* property

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

Document: getElementById()

DocumentFragment: getElementById()

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

Document: *querySelector()* method

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

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

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

Element: getElementsByTagName()

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

Element: firstElementChild

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

HTMLTableElement: rows

HTMLTableSectionElement: rows

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

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

HTMLTableRowElement: cells

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

Element: append()

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

Element: before()

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

HTMLInputElement: Instance property *value*

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

Date:getTime()

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