University of Wollongong Dubai

CSIT128 - Introduction to Web Technology

Practical Assignment Report

Authors:

Bilal Sajid | 8105807

Mahad Amer | 7795178

Malik Sami | 7774175

Mahad Umair | 7779823

Professor:

Dr. Haitham Yaish

June 26, 2023

Contents

1 Introduction

The conventional method of having clothing manufactured to order is frequently cumbersome, inconvenient, and impersonal. Customers must physically visit tailoring businesses, wait while measurements are collected, and then wait further until the finished product is ready. This procedure might not fit into the hectic schedules of contemporary consumers and can be frustrating. There is a need for a tailoring-on-demand website that offers customers looking for custom apparel a practical, tailored, and effective solution in order to address these problems.

In general, the website for on-demand tailoring aims to develop a user-friendly platform that transforms the standard tailoring procedure. The website will provide users with a convenient and personalized way to order high-quality, custom-made apparel while also enabling talented tailors to access a larger consumer base and grow their business by addressing the aforementioned issues.

2 HTML Files

The Embedded Javascript (EJS) view engine was used for this project as it is a popular template engine that allows for the creation of dynamic webpages with plain javascipt code embedded. EJS also provides features such as template inheritance, partials, and other custom functions which further enhance its flexibility and usefulness in web development projects.

This project consists of 7 HTML files that makes up the website. ”Project.ejs” is the main file where the user will be taken to after successful login, and ”services.ejs” shows our service offerings and ”contact.ejs” allows the user to contact us with their queries. ”Register.ejs”, and ”login.ejs” fulfill the needs for user registration, and ”services.ejs”, ”checkout.ejs” and ”thankyou.ejs” are used to fulfill the requirements for the website’s shopping infrastructure.

The register.ejs contains within a container, a registration form which is defined with fields for name, email, email confirmation, password, and password confirmation. There is also a ”Register” button to submit these details so that they can be stored in the server. A link to the login page is also present for users who are already registered.

Likewise, the login.ejs page also has a container. Within that a login form is defined with fields for email and password, and there is also a ”Sign In” button and a link to the registration page for users who are not registered yet.

Furthermore, the checkout.ejs page starts off with an <h1> heading displaying the services selected by that specific user. Next, an unordered list with the id ”services-list” is defined. It is populated dynamically based on the services chosen by the user which are displayed using a table which contains the selected services, their quantities, and their costs. The table is populated using a loop (services.forEach) to iterate over the services array. Once the user picks a service, the price of that service is obtained through an SQL query to the database and then displayed to the user below the quantity box.

Once the user has added as many services as they require to the cart, they can proceed to the checkout page by pressing the ”Proceed to Payment” button at the top. On the payment page, the user can see the services they have selected and the option to book an appointment date followed by a form to fill out their card details.

3 JavaScript

```
1  document.addEventListener('DOMContentLoaded', () => {
2    const logoutBtn = document.getElementById('logout-button');
3
4    logoutBtn.addEventListener('click', () => {
5      fetch('/logout', {
6        method: 'GET',
7        headers: {
8          'Content-Type': 'application/json'
9        },
10       redirect: 'follow'
11     })
12     .then((response) => response.text())
13     .then((result) => {
14       console.log(result); // Optional: Display the logout response in the console
15       window.location.href = '/'; // Redirect to the registration page
16     })
17     .catch((error) => {
18       console.error('Error logging out: ', error);
19     });
20   });
21 });
```

Code snippet of an event listener and its functions | from ”project.js”

The code above is an event listener that waits for the DOMContentLoaded event to be fired, which indicates that the HTML document has finished loading. Once the event is triggered, the code within the event listener is executed.

Lines 2 to 4 retrieve the HTML element with the id ”logout-button” and assign it to the variable logoutBtn. This allows for easier referencing of the element later on. At line 4, the logoutBtn event listener is set up.

Once the user presses the button, an HTTP GET request is made to the ”/logout” endpoint and the user is redirected to the ”/logout” page. Then within the index.js file, the ”app.get(’/logout’)” element is triggered which destroys the saved session of the user and force redirects them to the root page. When the user is sent to the root page, the routers within ”index.js” checks if the user is logged in or not, and since the loggedIn session was destroyed, the user is presented with the registration page.

Furthermore, if any errors occur during the HTTP request, they are captured and logged to the console using the console.error() function at line 17. This ensures that any potential errors during the request process are properly handled and can be identified for debugging or troubleshooting purposes

```
1  var swiper = new Swiper(".services-slider", {
2    slidesPerView: 1,
3    spaceBetween: 20,
4    centeredSlides: true,
5    loop: true,
6    grabCursor: true,
7    autoplay: {
8      delay: 3000,
9      disableOnInteraction: false,
10   },
11   pagination: {
12     el: ".swiper-pagination",
13     clickable: true,
14   },
15   breakpoints: {
16     0: {
17       slidesPerView: 1,
18     },
19     768: {
20       slidesPerView: 2,
21     },
22     991: {
23       slidesPerView: 3,
24     },
25   },
26 });
```

Code snippet of a swiper instance of the Swiper library | from ”project.js”

The above code initializes a new instance of the Swiper library and assigns it to the var ’swiper’. The initialized slider ”.services-slider” at line 1 is an HTML element with the class ”swiper services-slider” within the ”project.html” file. The Swiper library enables for sliders and carousels that can be interactively controlled with touch. Lines from 2 to 25 configure various options to customize the behavior and appearance of the slider.

This slider is displayed on the home page and on the registration page to give a preview of the services that is offered to the user before they create an account. Options on lines 2 to 6 specify the number of slides to be visible at a time (slidesPerView: 1), spaces between each slide in pixels (spaceBetween: 20), if they should be centered (centeredSlides: true) and create an endless loop (loop: true), and if the cursor should change to a grabbing hand icon when hovering over the slides (grabCursor: ture). Furthermore, the additional options from lines 7 to 24 add autoplay, pagination and breakpoint functionality to the slides along with additional options to customize the settings as required.

```
1  const express = require('express'); {
2  const mysql = require('mysql');
3  const path = require('path');
4  const app = express();
5  const session = require('express-session');
6
7  // Create connection to MySQL
8  const connection = mysql.createConnection({ {
9    host: 'localhost',
10   user: 'root',
11   password: 'MB$4amer',
12   database: 'csit128' {
13 });
14
15 // Connect to MySQL
16 connection.connect((err) => { {
17   if (err) {
18     console.error('Error connecting to database: ', err); {
19     return; {
20   }
21   console.log('Connected to database');
22 });
```

Code snippet of MySQL database connection | from "index.js"

The Express framework is imported on Line 1, along with other required modules such as mysql and express-session. The app variable represents the Express application instance.

Lines 8 to 12, a connection to the MySQL database is created using the connection details specified (host, user, password, and database name). The connection.connect() function at line 16 to 19 establishes the actual connection, and if successful, a message is logged to the console on line 18.

This code serves as the foundation for developing a web application with Express and MySQL, enabling the handling of HTTP requests and database interactions for data storage and retrieval.

```
1  app.post('/login', (req, res) => {
2    const { email, password } = req.body;
3    const sql = 'SELECT * FROM accounts WHERE email = ? AND password = ?';
4    connection.query(sql, [email, password], (err, results) => {
5      if (err) {
6        console.error('Error fetching user: ', err);
7        const errorMessage = 'Error fetching user';
8        // This code below sends a javascript alert with the error and then refreshes the
             page
9        res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
10       return;
11     }
12     if (results.length > 0) {
13       req.session.loggedIn = true;
14       req.session.name = results[0].name;
15       res.redirect('/home');
16     } else {
17       const errorMessage = 'Invalid credentials';
18       res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
19     }
20   });
21 });
22
23 app.post('/register', (req, res) => {  {
24   const { name, email, password } = req.body;
25   const sql = 'INSERT INTO accounts (name, email, password) VALUES (?, ?, ?)';
26   connection.query(sql, [name, email, password], (err) => {
27     if (err) {
28       console.error('Error saving data: ', err);
29       res.send('Error saving data');
30       return;
31     }
32     res.redirect('/login');
33   });
34 });
```

Code snippet of login and registration routes handling user authentication and database operations | from "index.js"

The code snippet includes the declaration of a route (/login) that handles a POST request for user login.It retrieves the email and password from the request body. A SQL query is executed to select the user from the accounts table in the database based on the provided email and password. If there is an error during the query execution, an error message is logged, and an alert is displayed to the user with an error message. If the query is successful and returns at least one result, the user session is marked as logged in, and the user's name is stored in the session. Finally, the user is redirected to the /home page. If the query does not return any results, an alert is displayed to the user indicating invalid credentials.

At line 23, the /register route handles a POST request for user registration. The name, email, and password are extracted from the request body. A SQL query is executed to insert the user's data into the accounts table. If there is an error during the query execution, an error message is logged, and an appropriate response is sent back to the client indicating the error. If the query is successful, the user is redirected to the /login page.

```
1 <% services.slice(0, 5).forEach(function(service) { %>
2   <option value="<%= service.service_id %>" data-price="<%= service.price %>"><%=
        service.service %></option>
3 <% }); %>
```

Code snippet of Looping and Generating <option> Elements in HTML Using JavaScript | from ”services.ejs”

Line 1 begins a JavaScript code block that loops through a subset of the services array. The slice(0, 5) part specifies that we want to loop through the first five elements of the services array. The forEach function is then used to iterate over each element

It generates an HTML <option> element for each service in the loop. The value attribute is set to the service_id property of the current service object, while the data-price attribute is set to the price property.

The text content of the <option> element is set to the service property. <% }); %>: This line marks the end of the JavaScript code block and closes the forEach loop. In summary, this looping code takes a subset of the services array and generates a set of <option> elements within the dropdown menu for each service section. The loop iterates over the specified range of services and dynamically populates the options with the corresponding service ID, price, and service name. This allows users to select a specific sub-service from the dropdown menu when interacting with the website.

For example, the dropdown menu for the ”Repairs” service is generated. The dropdown has an id attribute of ”service1” and a data-section attribute set to ”1”. The onchange event is set to call the updatePrice() function with the argument ”service1” when the selection changes. The options for the dropdown menu are populated using the services.slice(0, 5).forEach() loop. It iterates over the first five services in the services array and generates an <option> element for each service.

4 Form Validation

There are 2 different types of validations used in this project. The first one is for forms validation, and the second one is for session validations.

Listed below are some examples of the two types of validations.

All the code snippets are taken as excerpts from the actual HTML code

4.1 Form validation

1. Input Types:

```
1    <input type="email">
2    <input type="password">
3    <input type="text">
```

Line 1 specifies an email input field, which can trigger built-in email validation in modern browsers such as Chrome and Firefox. Line 2 Represents a password input field, where the entered text is masked, and line 3 represents a generic text input field.

2. Checkbox Validation:

```
1    <input type="checkbox">
```

This defines a checkbox input, which can be used for terms and conditions acceptance. Users can be required to check the checkbox to proceed.

3. Required Fields:

```
1    var email = emailInput.value;
2    var password = passwordInput.value;
3
4    if (email && password) {
5      alert("Login Successful!");
6
7      emailInput.value = "";
8      passwordInput.value = "";
9
10     loginFormContainer.classList.remove("show");
11   } else {
12     alert("Please enter your email and password.");
13   }
```

Adding required attributes to the input field ensures that the user must fill in that field before submitting the form. These are attributes such as Names and contact details. Moreover there is consistent use of “if” statements in the code to ensure a smooth flow of commands and to make sure that the program runs as intended even when encountered with erroneous or invalid data. Above lines 1 to 13 is a code snippet indicating the use of an if statement through an email and password checker which will validate the credentials of a user on the login page.

4. **Form Submission:**

```
1    <form>
2    <input type="submit">
```

Line 1 defines the form element that encapsulates the input fields and form-related elements, and line 2 represents the submit button that triggers the form submission.

5. **Regexp:**

```
1 <label for="card-no">Card Number:</label>
2 <input type="text" id="card-no" name="card-no" pattern="[0-9]{1,16}" required><br>
3
4 <label for="cvv">CVV:</label>
5 <input type="text" id="cvv" name="cvv" pattern="[0-9]{1,3}" required><br>
```

The addition of regular expressions (regex) to the form validation process enhances the overall validation of user input. By incorporating regex patterns into specific form fields, such as card number and CVV, we can enforce stricter rules and prevent users from submitting invalid data.

In the case of card number validation, the regex pattern [0-9]1,16 ensures that the card number can only consist of digits and must be between 1 and 16 characters in length. This prevents users from entering longer card numbers, which could be erroneous or indicate potential issues.

Similarly, for the CVV field, the regex pattern [0-9]1,3 limits the input to a minimum of 1 and a maximum of 3 digits. By enforcing this constraint, we ensure that the CVV is within the expected range and prevent users from entering excessive or incorrect values.

By incorporating these regex patterns into the form, we provide real-time validation feedback to users. When they enter card numbers or CVV that exceed the allowed length, the browser's built-in form validation mechanism will display an error message, alerting users to correct their inputs. This helps improve the user experience by guiding them towards providing valid and properly formatted data, reducing the likelihood of errors and improving the overall integrity of the submitted form data.

4.2 Session validation

Session validations have been extensively utilized within the "index.js" file, which is considered to be the root of the tree and is responsible for validating and storing sessions so that other scripts can utilize them. When the user proceeds to checkout, the "index.js" file will ensure that they are logged in and only then will it allow them to proceed Parallel to this, the name of the user will also be saved so that other scripts can access it. The code below is an example of validating the "loggedIn" session, and only allowing the user to proceed if it is true.

```
1 app.get('/checkout', checkLoggedIn, (req, res) => {
2   const name = req.session.name; // Retrieve the user's name from the session
```

The below code destroys all of the sessions once the user calls the \logout page and redirects them to the root page, or the registration page. This essentially loggs the user out and does not allow them to proceed as their "loggedIn" session is not destroyed and not valid.

```
1   app.get('/logout', (req, res) => {
2     req.session.destroy(); // Destroy the session on logout
3     res.redirect('/');
4   });
```

5 CSS Files

```css
.login-form-container {
  position: fixed;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  padding: 20px;
  background-color: rgba(0, 0, 0, 0.9);
  border: 3px solid #00ff00;
  border-radius: 10px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
  backdrop-filter: blur(5px);
  z-index: 9999;
  display: none;
}

.register-form-container {
  position: fixed;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  width: 450px;
  padding: 20px;
  background-color: #ffffff;
  border: 2px solid #00ff00;
  border-radius: 10px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
  z-index: 9999;
  display: none;
}

.header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 20px;
  background-color: #000;
}

.banner {
  text-align: center;
  padding: 50px 0;
  background-color: #000;
}

.services .services-slider .box {
  position: relative;
  text-align: center;
  width: 350px;
  height: 350px;
  border: 2px solid #ffffff;
  border-radius: 10px;
  overflow: hidden;
```

```
53    margin: 10px;
54    display: inline-block;
55  }
56
57
58  .menu {
59    display: flex;
60    justify-content: flex-end;
61    align-items: center;
62  }
63
64  .footer {
65    background-color: #00ff00;
66    color: #fff;
67    padding: 20px;
68    display: flex;
69    justify-content: space-between;
70    align-items: center;
71  }
72
73  .logo {
74    width: 50px;
75  }
76
77  .social-media a {
78    display: inline-block;
79    margin-right: 10px;
80    color: #fff;
81    text-decoration: none;
82    transition: color 0.3s ease;
83  }
```

Sample code snippet of the CSS style for various different containers | from ”project.css”

CSS files have been linked with all the Ejs pages to aesthetically beautify them and to provide a suitable and interactive user interface.

The code above from lines 1 to 83 is CSS code, from the ”project.css” file, that styles for various elements on a web page. It includes styles for a login form, register form, header, banner, services section, and footer. Lines from 1 to 14 is the login-form-container HTML class element, and its respective JavaScript code can be found in section 3.

The login form and register form containers, from lines 1 to 29, are positioned in the center of the screen using the ”position: fixed” and ”transform” properties. The forms have a box shadow and a border radius to give them a 3D effect. The register form has a white background color, while the login form has a semi-transparent black background color.

The header, lines 31 to 37, has a black background color and contains a logo and a menu. The menu is a horizontal list of links, with a dropdown menu for additional links. The header also has a button with a green background color. The next bunch of lines contains containers for a banner, services, menu, footer, logo and social media information respectively.

The banner has a black background color and contains a heading, subheading, and a button with a green background color. The services section contains a slider with boxes that have images and content. The content is hidden until the box is hovered over, and the footer has a green background color and also contains a logo, contact details,

and social media links.

```
1   body {
2     font-family: Arial, sans-serif;
3     text-align: center;
4   }
5
6   h1 {
7     font-size: 24px;
8     margin-bottom: 20px;
9   }
10
11  table {
12    width: 25%;
13    border-collapse: collapse;
14    margin-bottom: 20px;
15    margin-left: auto;
16    margin-right: auto;
17  }
18
19  #services-list {
20    list-style: none;
21    padding: 0;
22  }
23
24  input[type="submit"] {
25    background-color: rgba(0, 128, 0, 0.8);
26    color: white;
27    cursor: pointer;
28  }
```

Sample code snippet of the CSS styles | from "checkout.css"

This CSS page contains styling rules for various elements in the associated HTML document. The body element is styled to be a flex container, centered both horizontally and vertically, with a specific height, font, background color, and padding. The h1 element has a defined font size and margin. The table is given a specific width, border-collapse, and margin at lines 1 - 9.

The form element has an outline with a solid dark green border of 2 pixels, giving it a rectangular shape. The border-radius property is explicitly defined for the form which gives the corners a rounded shape, at lines 11 - 22. Moreover the background is given a light green hue, at lines 24 - 28. This coupled with the dark green border and 'pay' button makes the page a whole lot more pleasing to look at and gives a professional and safe feeling, lines

6 Images / Logos

Every image used in the layout of the website comes from reliable sources, referenced in section 12. They not only improve the aesthetic appeal of the website overall, but they also give customers who need it a visual aid and speed up the decision-making process.

The images have been encased in their separate divs to allow for smoother transitioning between the images in the carousel included on the webpage. The below code snippet also shows that alternative texts to the images have also been provided in case the images fail to load for any reason.

```
<section class="services" id="services">
  <h1 class="heading">Services</h1>
  <div class="swiper services-slider">
    <div class="swiper-wrapper">
      <div class="swiper-slide">
        <div class="box">
          <img src="/content/Repairs_pic.jpg" alt="">
          <div class="content">
            <h3>Repairs</h3>
          </div>
        </div>
      </div>
      <div class="swiper-slide">
        <div class="box">
          <img src="/content/Alterations_pic.jpg" alt="">
          <div class="content">
            <h3>Alterations</h3>
          </div>
        </div>
      </div>
```

The element with the class ”swiper-slide” shows that it is a single slide within the slideshow component. The slideshow structure is the carousel referred above. Moreover, the images included have been properly sized to blend perfectly within the webpage with no abnormalities in either dimensions as verified by the below code snippet.

```
<section id="clothing-section" class="service-section">
  <div class="service-content">
    <div class="service-image">
      <img src="/content/Clothing_pic.jpg" alt="clothing Image" style="width: 200px; height:
          auto;">
    </div>
```

The above code implements the alt text within the images in the ”services.html” file. The design philosophy of the logo is inline with the company values. The minimalistic look of the logo also enhances the look of the website.

7 User Interface Design and Flow

User Interface (UI) design plays a crucial role in creating a pleasant user experience. Here are some aspects of the code which help the UI to be well-designed and easy to use:

1. **Intuitive and User-Friendly:**

   - The login and registration forms have clear labels and placeholders for input fields, making it easy for users to understand what information is required.

   - The navigation menu is straightforward, with clear buttons for different sections.

2. **Improved Usability:**

   - The login and registration forms have logical input fields and a submit button, allowing users to perform the desired actions easily.

   - The ”Contact” and ”Login” buttons in the menu provide quick access to the corresponding sections.

3. **Consistency:**

   - The logo and brand name ”CustomThreads” are consistently used throughout the website, reinforcing brand identity.

   - The close buttons for the login and registration forms have the ”×” symbol, a commonly recognized visual cue for closing or dismissing elements.

4. **Visual Appeal:**

   - The banner section with its headings and ”Explore” button is visually appealing, grabbing the users’ attention.

   - The services section uses attractive images and headings to showcase different service offerings.

5. **Clear Communication:**

   - The login and registration forms have clear labels and placeholders for input fields, providing guidance to users.

   - The contact details and address in the footer section communicate important information to users.

6. **Accessibility:**

   - The HTML code explicitly includes accessibility features, such as alt text for images. Adding these features improve the accessibility in case the images fail to load.

7. **Brand Consistency:**

   - The logo and brand name ”CustomThreads” appear consistently in the header and footer sections.

8. **Faster Task Completion:**

   - The ”Login” button in the menu provides a quick way for users to access the login form without navigating through multiple pages.

9. **Error Prevention and Handling:**

   - The HTML code includes specific error prevention and handling features, such as form validation and error messages. Incorporating these features enhances the user experience as they are less bound to enter wrong or invalid data. Moreover it also makes the website secure by disallowing any unauthorized access other than by the user as they have to validate themselves by their password.

10. Mobile and Responsive Design:

- The referenced CSS files (‘project.css‘ and ‘swiper-bundle.min.css‘) handle responsive styling to make the pages look much more appealing.

Upon opening the website, users will land on the registration page where they can create an account. After providing the necessary information, they can submit the registration form. Once registered, users can proceed to the login page to enter their email and password.

It is important to note that only logged-in users can access the home screen and enjoy the full range of features. If a user attempts to access the home screen without logging in, they will be redirected back to the registration page. This security measure ensures that only authorized individuals can interact with the website’s functionalities and protects user data.

After a successful login, users will be directed to their personalized ”home” page, which displays a friendly greeting with their name. From there, they can explore various services and add desired items to their cart. To pay for and finalize their selected services, users can proceed to the checkout page.

If users wish to end their session and log out, they can simply click on the ”logout” option. This will destroy the session and redirect them back to the registration page.

The website also offers a ”contact” page, where users can find relevant information or fill out a form to communicate with the website administrators. There is also a profile page for the user to edit their account details in case they wish to change their email, username or password.

Throughout the user interface design, the focus is on providing a user-friendly experience and ensuring the security of user data. By utilizing a database, the website efficiently stores and retrieves user details and service information, enabling seamless communication between users and the platform.

Overall, the website offers a smooth and enjoyable user experience, allowing individuals to easily register, log in, explore available services, manage their cart, and complete transactions. The emphasis on user-friendliness and data security ensures that users can navigate the website with confidence and peace of mind.

8 Dynamic web pages with MySQL integration

```
app.get('/', (req, res) => {
  res.render('register', { loggedIn: req.session.loggedIn });
});

app.post('/register', (req, res) => {
  const { name, email, password } = req.body;
  const sql = 'INSERT INTO accounts (name, email, password) VALUES (?, ?, ?)';
  connection.query(sql, [name, email, password], (err) => {
    if (err) {
      console.error('Error saving data: ', err);
      res.send('Error saving data');
      return;
    }
    res.redirect('/login');
  });
});
```

Code snippet of get and post request | from "index.js"

When the GET request is made to the root route ('/'), the server responds by rendering the 'register' ejs template. The second argument to the res.render function is an object containing data to be passed to the template. In this case, it includes a property called loggedIn which is set to the value of req.session.loggedIn.

When the POST request is made to the '/register' route, the server handles the registration process. It expects the client to send data in the body of the request, including the name, email, and password. Name, Email and Password are then extracted from the req.body object using destructuring assignment. It then forms an SQL query string to insert the extracted data into a table called 'accounts' in the SQL database. The question marks in the query string are placeholders for the actual values.

The code uses a database connection object to execute the SQL query. The values for the placeholders in the query are provided as an array as the second argument to the connection.query function.

If there is an error during the execution of the query (duplicate entry), a message is logged to the console, "Error saving data". If the query is successful and no error occurs, the server redirects the client to the '/login' route.

```
app.get('/login', (req, res) => {
  res.render('login');
});

app.post('/login', (req, res) => {
  const { email, password } = req.body;
  const sql = 'SELECT * FROM accounts WHERE email = ? AND password = ?';
  connection.query(sql, [email, password], (err, results) => {
    if (err) {
      console.error('Error fetching user: ', err);
      const errorMessage = 'Error fetching user';
      res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
      return;
    }
    if (results.length > 0) {
      req.session.loggedIn = true;
      req.session.name = results[0].name;
      res.redirect('/home');
    } else {
      const errorMessage = 'Invalid credentials';
      res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
    }
  });
});
```

Code snippet of get and post request | from "index.js"

When the GET request is made to the '/login' route, the server responds by rendering the 'login' EJS file. This file is responsible for displaying the login form to the user.

When the POST request is made to the '/login' route (user submitting the login form), the server processes the login request. The code extracts the email and password fields from the req.body object using destructuring assignment. These fields contain the user's login credentials.

It then forms an SQL query string to retrieve(SELECT) users information from the 'accounts' table based on the provided email and password. The query searches for a row in the table where the email and password match the provided values.

The code uses a database connection object to execute the SQL query. The values for the placeholders in the query are provided as an array as the second argument to the connection.query function.

If there is an error during the execution of the query (duplicate entry), a message is logged to the console, "Error saving data". If the query is successful and no error occurs, the server redirects the client to the '/login' route.

If there is an error during the execution of the query, a message is logged to the console, "Error fetching user: ", indicating that there was an error fetching the user. The user is then redirected back to the previous page instead of the '/home' route.

If the query is successful and no error occurs, the code checks if the query results contain any rows (results.length > 0). If there is at least one row, it means that the user with the provided credentials was found in the 'accounts' table.

In that case, the code sets the loggedIn property in the req.session object to true to indicate that the user is logged in. It also stores the user's name from the first row of the results in the req.session.name property and is displayed on the home page welcoming the user. If the query results are empty (user does not exist in the table), the code sends an alert message to the client's browser indicating 'invalid credentials'. The user is then redirected to the register page.

```
app.get('/services', checkLoggedIn, (req, res) => {
  const sql = 'SELECT * FROM services';
  connection.query(sql, (err, results) => {
    if (err) {
      console.error('Error retrieving services: ', err);
      res.send('Error retrieving services');
      return;
    }
    const services = results;
    res.render('services', { services });
  });
});
```

Code snippet of get request | from "index.js"

When the GET request is made to the '/services' route, the server invokes the route function. The 'checkLoggedIn' function is a parameter function that is executed before the main route function. It is responsible for checking if the user is logged in. The purpose of this parameter is to restrict access to the '/services' page only to logged in users. If the user is not logged in, the middleware may redirect them to a login page.

The code then constructs an SQL query string to select all rows from the 'services' table. It uses the database connection object 'connection' to execute the SQL query. The query is executed by passing the SQL query string as the first argument to the 'connection.query' function.

The results of the query are passed to a callback function that handles the response from the database. The callback function is executed when the database operation is completed. If an error occurs during the execution of the query, an error message is logged to the console, and the server responds with the message "Error retrieving services".

If the query is successful and no error occurs, the callback function receives the query results in the 'results' parameter. The code assigns the 'results' to a variable named 'services'. Finally, the code renders the 'services' EJS file and passes the 'services' variable as data to be used in the EJS file. The rendered EJS content is sent as the response to the client.

```
app.get('/addToCart', checkLoggedIn, (req, res) => {
  const { serviceId, quantity } = req.query;
  const name = req.session.name;

  const order = {
    account_name: name,
    service_id: serviceId,
    quantity: quantity
  };

  const sql = 'INSERT INTO orders SET ?';
  connection.query(sql, order, (err) => {
    if (err) {
      console.error('Error inserting order: ', err);
      res.send('Error inserting order');
      return;
    }
    res.redirect('/services');
  });
});
```

Code snippet of get request | from "index.js"

When a GET request is made to the '/addToCart' route, the server invokes the route function. The 'checkLoggedIn' function is a parameter function that is executed before the main route function. It is responsible for checking if the user is logged in. If the user is not logged in, the function may redirect them to a login page.

The route retrieves the serviceId and quantity parameters from the query string along with the username from the session. An SQL query is then executed to insert the order into the 'orders' table. The connection.query() function executes the SQL query using the connection to the MySQL database.

If an error occurs during the execution, it logs the error and sends an error message as the response. If the query is successful, the user is redirected back to the '/services' page

This code allows a logged-in user to add services to their cart by inserting an order record into the 'orders' table in the MySQL database. The service ID, quantity, and user's name are used to create the order, which is then inserted into the database. After a successful insertion, the user is redirected back to the 'services' page.

```
app.get('/checkout', checkLoggedIn, (req, res) => {
  const name = req.session.name;
  const sql = `
  SELECT services.service, orders.quantity, services.price
  FROM orders
  INNER JOIN services ON orders.service_id = services.service_id
  WHERE orders.account_name = ?
  `;
  connection.query(sql, [name], (err, results) => {
    if (err) {
      console.error('Error retrieving services: ', err);
      res.send('Error retrieving services');
      return;
    }
    const services = results;
    let totalCost = 0;
    services.forEach((service) => {
      totalCost += service.quantity * service.price;
    });
    res.render('checkout', { name, services, totalCost });
  });
});
```

Code snippet of get request | from "index.js"

When a GET request is made to the '/checkout' route, the server invokes the route function. The 'checkLoggedIn' function is a parameter function that is executed before the route function. It is responsible for checking if the user is logged in. The purpose of this parameter is to restrict access to the '/checkout' page only to logged in users. If the user is not logged in, the GET request redirects them to a login page.

The code retrieves the username from the req.session object. It constructs an SQL query string which joins the 'orders' table with the 'services' table using the 'service_id' field. It selects the service name, quantity, and price from the joined tables for the given account name.

The username is passed as a parameter in the SQL query by using a placeholder '?' and providing the value as an array in the subsequent query execution. It uses the database connection object 'connection' to execute the SQL query. The query is executed by passing the SQL query string as the first argument and the user's name as the second argument to the 'connection.query' function.

The results of the query are passed to a callback function that handles the response from the database. The callback function is executed when the database operation is completed. If an error occurs during the execution of the query, an error message is logged to the console, and the server responds with the message "Error retrieving services". If the query is successful and no error occurs, the callback function receives the query results in the 'results' parameter.

The code assigns the 'results' to a variable named 'services'. It assumes that the query results represent the services related to the users account retrieved from the database. The code calculates the total cost of the services by iterating over the 'services' array and multiplying the quantity of each service by its price. Finally, the code renders the 'checkout' EJS file and passes the 'name', 'services', and 'totalCost' variables. The rendered EJS content is sent as the response to the client.

9 MySQL Database

```sql
create database csit128;
use csit128;

CREATE TABLE `accounts` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
);

CREATE TABLE `services` (
  `service_id` int NOT NULL AUTO_INCREMENT,
  `service` varchar(255) DEFAULT NULL,
  `price` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`service_id`)
);

CREATE TABLE `orders` (
  `account_name` varchar(255) DEFAULT NULL,
  `service_id` int DEFAULT NULL,
  `quantity` int DEFAULT NULL,
  KEY `service_id` (`service_id`),
  CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`service_id`) REFERENCES `services` (`service_id`)
);
```

Code snippet of SQL statements for creating the "csit128" database and tables | from "csit128.sql"

The database csit128 is created with three tables: accounts, services, and orders. The accounts table stores user information with columns for id, name, email, and password. The services table contains information about the available services, including service_id, service, and price. The orders table represents user orders with columns for account_name, service_id, and quantity. The service_id column in the orders table is a foreign key referencing the service_id column in the services table.

The queries used in the project includes SELECT to retrieve data from the services table, INSERT to add orders to the orders table, JOIN to combine data from multiple tables, WHERE to filter results based on specific conditions, and INSERT to register new user accounts in the accounts table. These queries enable the application to fetch services, store user selections, calculate total costs, authenticate users, and manage user registrations. The SQL queries play a crucial role in accessing and manipulating data, facilitating the functionality of the project.

```
// Set up view engine
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// Middleware for handling form data
app.use(express.urlencoded({ extended: false }));

// Serve static files from the 'public' directory
app.use(express.static(path.join(__dirname, 'public')));

// Middleware to check if the user is logged in
const checkLoggedIn = (req, res, next) => {
  if (req.session.loggedIn) {
    next(); // User is logged in, proceed to the next middleware or route handler
  } else {
    res.redirect('/'); // User is not logged in, redirect to the registration page
  }
};
```

Code snippet of SQL statements for creating the "csit128" database and tables | from "csit128.sql"

The code connects to the MySQL database using the provided credentials. It also includes session middleware for managing user sessions and sets up the view engine as EJS.The application defines several routes for different pages such as registration, login, home, services, checkout, and more. It includes middleware to check if a user is logged in before accessing certain routes. The routes interact with the database to retrieve and manipulate data.

```
app.post('/login', (req, res) => {
  const { email, password } = req.body;
  const sql = 'SELECT * FROM accounts WHERE email = ? AND password = ?';
  connection.query(sql, [email, password], (err, results) => {
    if (err) {
      console.error('Error fetching user: ', err);
      const errorMessage = 'Error fetching user';
      // This code below sends a javascript alert with the error and then refreshes the page
      res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
      return;
    }
    if (results.length > 0) {
      req.session.loggedIn = true;
      req.session.name = results[0].name;
      res.redirect('/home');
    } else {
      const errorMessage = 'Invalid credentials';
      // This code below also sends a javascript alert with the error and then refreshes the
          page
      res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
    }
  });
});

app.post('/register', (req, res) => {
  const { name, email, password } = req.body;
  const sql = 'INSERT INTO accounts (name, email, password) VALUES (?, ?, ?)';
  connection.query(sql, [name, email, password], (err) => {
    if (err) {
      console.error('Error saving data: ', err);
      res.send('Error saving data');
      return;
    }
    res.redirect('/login');
  });
});
```

Code snippet of login and registration route handlers for handling form submissions and interacting with the database | from "index.js"

The '/login' and '/register' routes handle user authentication. The '/login' route retrieves the provided email and password, queries the accounts table using a SELECT query, and compares the credentials. If valid, the user's session is updated, and they are redirected to the home page. The '/register' route inserts a new user's information into the accounts table using an INSERT query upon successful registration and then redirects the user to the login page, which redirects the user to the home page upon successful login.

```
app.get('/addToCart', checkLoggedIn, (req, res) => {
  const { serviceId, quantity } = req.query;
  const name = req.session.name; // Retrieve the user's name from the session

  const order = {
    account_name: name,
    service_id: serviceId,
    quantity: quantity
  };

  const sql = 'INSERT INTO orders SET ?';
  connection.query(sql, order, (err) => {
    if (err) {
      console.error('Error inserting order: ', err);
      res.send('Error inserting order');
      return;
    }
    res.redirect('/services');
  });
});

app.get('/checkout', checkLoggedIn, (req, res) => {
  const name = req.session.name; // Retrieve the user's name from the session

  const sql = `
  SELECT services.service, orders.quantity, services.price
  FROM orders
  INNER JOIN services ON orders.service_id = services.service_id
  WHERE orders.account_name = ?
  `;
  connection.query(sql, [name], (err, results) => {
    if (err) {
      console.error('Error retrieving services: ', err);
      res.send('Error retrieving services');
      return;
    }

    const services = results;
    let totalCost = 0;
    services.forEach((service) => {
      totalCost += service.quantity * service.price;
    });

    res.render('checkout', { name, services, totalCost });
  });
});
```

Code snippet of the addToCart and checkout route handlersfrom | from "index.js"

The '/addToCart' route is responsible for processing user requests to add items to their cart. Once the user picks what service they want and the quantity, the serviced, according to the services table, and the quantity are sent as a query to the /addToCart route. It then retrieves the service ID and quantity from the request query parameters and the user's name from the session. Then, it constructs an order object and inserts it into the orders table using an

INSERT query. Upon successful insertion, the user is redirected back to the services page.

The '/checkout' route retrieves the user's name from the session and performs a JOIN operation between the orders and services tables. It fetches relevant data based on the user's account name using a SELECT query with a WHERE condition. The fetched data, along with the calculated total cost, is rendered on the 'checkout' page.

10 Node.js

```
1  const express = require('express');
2  const mysql = require('mysql');
3  const path = require('path');
4  const app = express();
5  const session = require('express-session');
6
7  // Create connection to MySQL
8  const connection = mysql.createConnection({
9    host: 'localhost',
10   user: 'root',
11   password: 'MB$4amer',
12   database: 'csit128'
13 });
14
15 // Connect to MySQL
16 connection.connect((err) => {
17   if (err) {
18     console.error('Error connecting to database: ', err);
19     return;
20   }
21   console.log('Connected to database');
22 });
23
24 // Start the server
25 const port = 3000;
26 app.listen(port, () => {
27   console.log(`Server started on port ${port}`);
28 });
```

Node.js and how it manages the files in the project | from "index.js"

The code above sets up a Node.js server using the Express.js framework and establishes a connection to a MySQL database. It also includes session management using the 'express-session' middleware.

The server listens on port 3000 and starts by connecting to the MySQL database on lines 16 - 22. If the connection is successful, it logs a message indicating the successful connection.

The server defines various routes for handling different HTTP requests:

```
1
2  // Routes
3  app.get('/', (req, res) => {
4    res.render('register', { loggedIn: req.session.loggedIn });
5  });
6
7  app.get('/login', (req, res) => {
8    res.render('login');
9  });
10
11 app.get('/thankyou', (req, res) => {
12   res.render('thankyou');
13 });
14
```

```
app.get('/home', checkLoggedIn, (req, res) => {
  const { name } = req.session;
  res.render('project', { name }); {
});

app.get('/logout', (req, res) => {
  req.session.destroy(); // Destroy the session on logout
  res.redirect('/');
});

app.get('/contact', (req, res) => {
  res.render('contact');
});

app.get('/services', checkLoggedIn, (req, res) => {
  const sql = 'SELECT * FROM services';
  connection.query(sql, (err, results) => {
    if (err) {
      console.error('Error retrieving services: ', err);
      res.send('Error retrieving services');
      return;
    }

    const services = results;
    res.render('services', { services });
  });
});

app.get('/addToCart', checkLoggedIn, (req, res) => {
  const { serviceId, quantity } = req.query;
  const name = req.session.name; // Retrieve the user's name from the session

  const order = {
    account_name: name,
    service_id: serviceId,
    quantity: quantity
  };

  const sql = 'INSERT INTO orders SET ?';
  connection.query(sql, order, (err) => {
    if (err) {
      console.error('Error inserting order: ', err);
      res.send('Error inserting order');
      return;
    }
    res.redirect('/services');
  });
});

app.get('/checkout', checkLoggedIn, (req, res) => {
  const name = req.session.name; // Retrieve the user's name from the session

  const sql = `
  SELECT services.service, orders.quantity, services.price
```

```
69    FROM orders
70    INNER JOIN services ON orders.service_id = services.service_id
71    WHERE orders.account_name = ?
72    `;
73    connection.query(sql, [name], (err, results) => {
74      if (err) {
75        console.error('Error retrieving services: ', err);
76        res.send('Error retrieving services');
77        return;
78      }
79
80      const services = results;
81      let totalCost = 0;
82      services.forEach((service) => {
83        totalCost += service.quantity * service.price;
84      });
85
86      res.render('checkout', { name, services, totalCost });
87    });
88  });
```

1. **GET '/':** Renders the 'register' view on line 3, passing the 'loggedIn' property from the session to indicate if the user is logged in or not.
2. **GET '/login':** Renders the 'login' view on line 7
3. **GET '/thankyou':** Renders the 'thankyou' view on line 11
4. **GET '/home':** Renders the 'project' view on 17 if the user is logged in (checked using the 'checkLoggedIn' middleware on 15). The user's name is passed to the view for personalization
5. **GET '/logout':** Destroys the session on logout on 20 and redirects to the root route ('/') on line 53 21.
6. **GET '/contact':** Renders the 'contact' view on 25
7. **GET '/services':** Retrieves services data from the database on line 29 and renders the 'services' view on line 39, passing the retrieved services as a parameter
8. **GET '/addToCart':** Retrieves the service ID and quantity from the query parameters on line 43 and inserts an order into the database on line 54. Then redirects to '/services' on line 60.
9. **GET '/checkout':** Retrieves the user's name from the session on line 64 and fetches the services associated with the user's orders from the database on line 73. Calculates the total cost on lines 81 - 84, and renders the 'checkout' view on line 86, passing the user's name, services, and total cost.

```
1  app.post('/login', (req, res) => {
2    const { email, password } = req.body;
3    const sql = 'SELECT * FROM accounts WHERE email = ? AND password = ?';
4    connection.query(sql, [email, password], (err, results) => {
5      if (err) {
6        console.error('Error fetching user: ', err);
7        const errorMessage = 'Error fetching user';
```

```
8       // This code below sends a javascript alert with the error and then refreshes the page
9       res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
10      return;
11    }
12    if (results.length > 0) {
13      req.session.loggedIn = true;
14      req.session.name = results[0].name;
15      res.redirect('/home');
16    } else {
17      const errorMessage = 'Invalid credentials';
18      // This code below also sends a javascript alert with the error and then refreshes the
            page
19      res.send(`<script>alert('${errorMessage}'); window.history.back();</script>`);
20    }
21  });
22 });
23
24 app.post('/register', (req, res) => {
25   const { name, email, password } = req.body;
26   const sql = 'INSERT INTO accounts (name, email, password) VALUES (?, ?, ?)';
27   connection.query(sql, [name, email, password], (err) => {
28     if (err) {
29       console.error('Error saving data: ', err);
30       res.send('Error saving data');
31       return;
32     }
33     res.redirect('/login');
34   });
35 });
```

1. **POST '/login':** Retrieves the email and password from the request body on line 1, queries the database to find a matching user on line 3, and sets the 'loggedIn' property in the session if the credentials are valid on line 13. Then redirects to '/home' on line 15.

2. **POST '/register':** Retrieves the name, email, and password from the request body on line 24, inserts the user data into the database on line 26, and redirects to '/login' on line 33

```
1  const session = require('express-session');
2  app.use(
3    session({
4      secret: 'aVeryS3cr3tK3y123123',
5      resave: false,
6      saveUninitialized: true
7    })
8  );
9  const checkLoggedIn = (req, res, next) => {
10   if (req.session.loggedIn) {
11     next(); // User is logged in, proceed to the next middleware or route handler
12   } else {
13     res.redirect('/'); // User is not logged in, redirect to the registration page
14   }
15 };
```

This project also utilizes express-session for managing sessions, and also stores the secret key into the session.

Overall, this code sets up a server with routes for registration, login, logging out, accessing different views, and interacting with the database to handle user requests related to services, cart, and checkout.

11 Summary

The following elements have been implemented within this project:

1. **Registration page:**

   This section covers the functionality related to user registration. It allows users to create a new account by providing their necessary information, such as name, email, and password. The email is validated within the HTML form element before the user data is stored into the database.

2. **A login page:**

   The login section focuses on user authentication. It provides a login form where users can enter their credentials, email and password, to gain access to their account. Once the user logs in to their account, a ”loggedIn” session is stored as ”True”, which allows other scripts within the project to validate proper log in by the user before running.

3. **Home page:**

   Once logged in, the user is redirected to the home page. The home page also checks if the user’s ”loggedIn” session is valid or not, and if it is they are able to proceed else they are redirected back to the registration page. This page serves as the starting point for users and displays various options, such as the services offered by CustomThreads and buttons to navigate throughout the website.

4. **Profile page:**

   The profile section allows users to view and manage their personal information and account settings. Users can update their profile details, such as name, email, and password. Furthermore, the user can also view what services they have currently added to their cart.

5. **Contact page:**

   The contact section provides a means for users to get in touch with CustomThread’s staff team with their inquiries. It includes a contact form where the user can fill in their contact details, such as their name and email, and a text box where they can enter their message.

6. **Services page:**

   The services section showcases the available services or products offered by CustomThreads. The services are divided into 4 categories and each category has a box to pick the sub-service, the quantity and the option to add the service to their cart. Once the service is added to cart, the user is shown an alert displaying the quantity and the sub-service that they have selected.

7. **Checkout page:**

   The checkout section enables users to review and finalize their selected services for purchase or booking. It displays a summary of the chosen services, quantities, and costs. The user are also able to edit the quantity of the services that they have selected, or delete it all together from this place. Once the user is ready to purchase, they can select the appointment date most appropriate for them and enter their card details to purchase the services and confirm the booking.

8. **Thank You page:**

   Once the user has successfully entered their payment details and time for booking, the ’thank you’ page is displayed which confirms to the user that their order has been successfully confirmed and that the CustomThread’s team

will be in contact with the user as soon as possible. Furthermore, once the user completes their purchase, all of their orders are deleted to confirm that they have successfully purchased the items in their carts.

12 Bibliography

References

Arabesque, S. (2016), 'Traditional saudi dress'.
URL: *https://saudiarabesque.com/traditional-urban-men-s-dress-of-saudi-arabia/*

Britches, F. (2018), 'Menswear alterations: What can a custom tailor do for you?'.
URL: *https://www.familybritches.com/menswear-alterations-what-can-a-custom-tailor-do-for-you/*

Cleaners, E. D. (n.d.), 'Alterations, repair'.
URL: *https://www.eddysdrycleaners.com/alterations-repair*

TAILORING and REPAIRS (n.d.).
URL: *https://coucoudre.org/repairs/*