Exam Project

Frameworks And Architectures for the Web, Spring 2022

Course code: KSFRWAW1KU

Group 6

Elisabeth Berg Bøgebjerg (elib@itu.dk), Jonas Sylvain Jersild Balin (jbal@itu.dk)
Klara Krag (klak@itu.dk), Luna Marie Gymoese Berthelsen (lube@itu.dk)

This project is prepared by Elisabeth Berg Bøgebjerg, Jonas Sylvain Jersild Balin, Klara Krag and Luna Marie Gymoese Berthelsen as part of the course Frameworks and Architecture for the Web. It contains a client-side web application for online shopping, and a server-side RESTful web service, which supports the client-side application.

Web design of the client-side application

Our web shop enables customers to browse and shop for clothes. The products fit into the product categories: Men/Women and Tops/Bottoms. Figure 1 shows the product taxonomy of the webpage

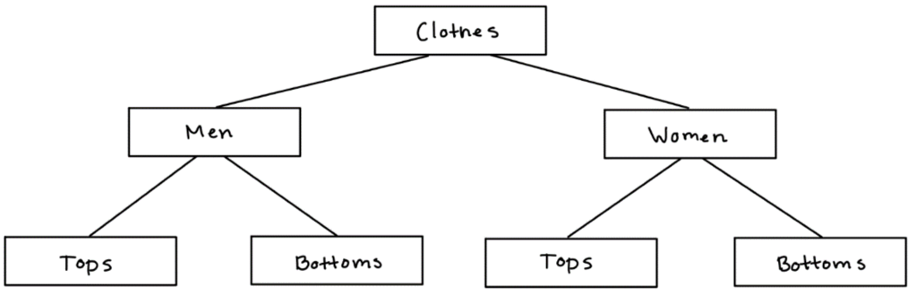


Figure 1: Product taxonomy.

All products are organized in a manner that allows the customer to access the desired product in as few clicks as possible. The customer can see more details of all the offered products, just as the customer can add a product to the basket from the product overview or product page, and products in the basket can the removed. If the customer creates a profile, the registered name of the profile is displayed on the navigation bar and in the basket. When a customer registers, the first name, last name, e-mail and password are validated (with simple requirements), and the customer can successfully continue shopping if the data is valid.

For this project, different Bootstrap components have been implemented on the website to make it responsive. These include a horizontal navigation bar with drop down elements, elements on the Profile Page, and the Product Overview that make use of Bootstrap cards as Product Cards. The navigation bar collapses to a burger menu when adjusting the screen size. The group decided to implement this functionality to accommodate different screen sizes (e.g., tablets or mobile devices).

Furthermore, the footer as well as the Product Overview and Basket were made responsive using the properties of Flexbox, as Flexbox offers Flex wrap and Grid display which was deemed necessary for the project.

All pages contain the navigation bar and the footer. This means that the user can access the Homepage, Product pages, their Profile or Login and the Basket from all pages.

Information architecture

The web shop is designed with a hybrid information structure, which allows the user to follow the site as a sequence, just as it supports the user in following their interests by having navigation possibilities on all pages. To support the chosen information structure, we decided to have a vertical navigation bar with drop down menus, displaying the product taxonomy and a footer. The footer does not have any navigational aspects now, but for further development this could be implemented.

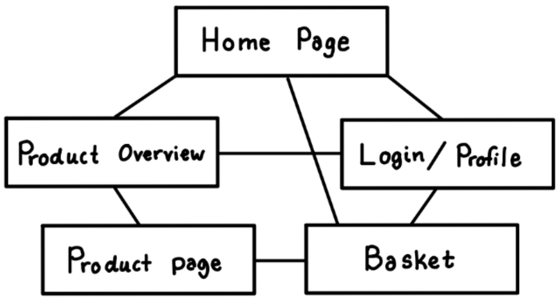


Figure 2: Information architecture for the web shop.

Site design

The wireframes are schematic representations of the most important screens in the application. We have six screens in our wireframes: Home, Product Overview, Product Page, Login, Profile, and Basket. Our application is responsive, and we have therefore created wireframes for two screen sizes. Figure 3 shows the wireframe for a computer and tablet; in this schematic representation we use lines to show the flow of the application. Figure 4 shows the wireframe for a mobile; in this we have not shown the flow of the application because it is the same as in figure 3. In the mobile version, the navigation bar is collapsed and can be accessed via the burger icon.

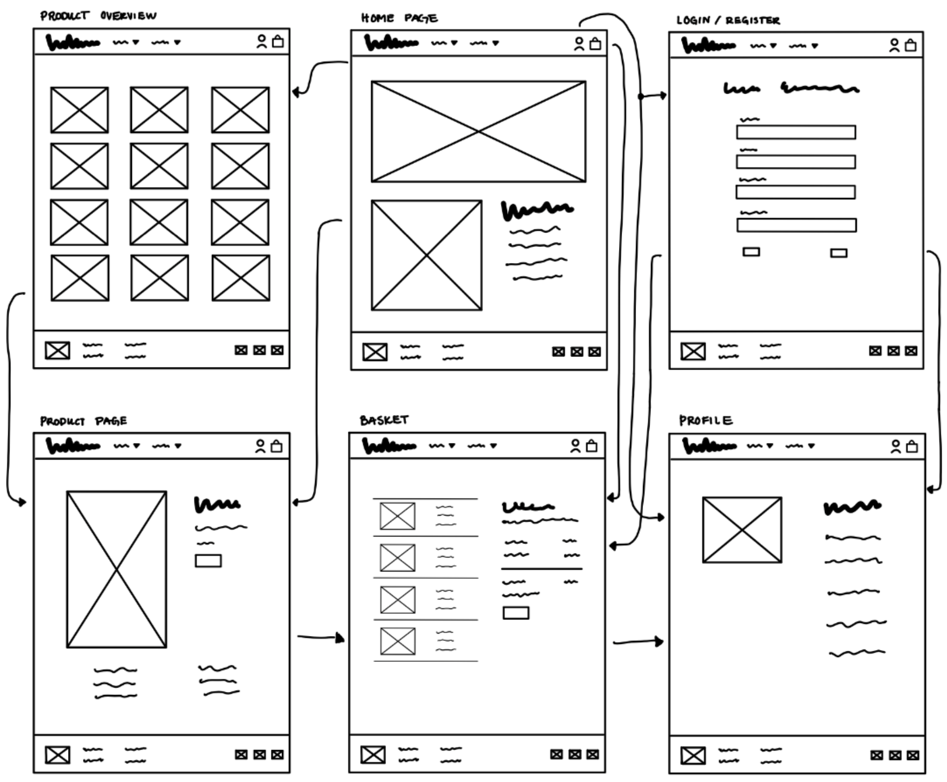


Figure 3: Wireframe for computer and tablet.

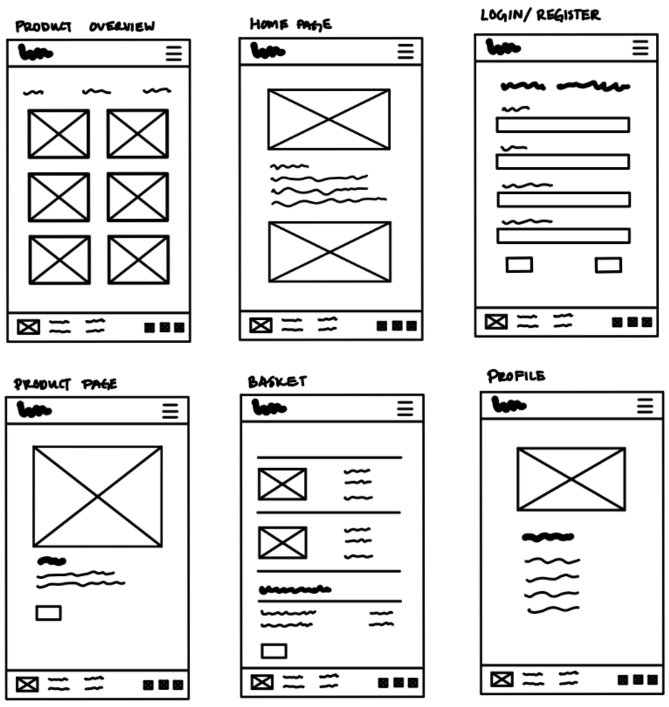


Figure 4: Wireframe for mobile.

Design of the RESTful API

The server-side of the application is a RESTful API which consists of endpoints that support the client-side application. The API is developed in JavaScript using Node.js and Express framework and structures data following the JSON data format, for the purposes of communicating data back and forth. Data about customer profiles, offered products, product categories, and the registered customers' baskets are stored on the web server.

The web service operations implemented in the RESTful API make sure that the correct data is shown on the client-side. This includes getting information about all products, getting categories, getting products based on the category/categories, and getting details about a specific product. Furthermore, a new customer can be added to the web server, and by default have a basket created for that specific customer, where products can be added and removed from that specific basket for the specific customer.

The API features basic checks on invalid data, for example when using the POST operation on a customer with an existing ID or adding a basket item to a non-existent customer.

RESTful API specifications

Resource path	**POST**	**GET**	**PUT**	**DELETE**
/customers	Create new customer	Get list of all customers	Error	Error
/customers/:id	Error	Retrieve existing customer	Update existing customer	Delete existing customer
/customers/:id/basket	Error	Retrieve products in basket for specific customer	Error	Error
/customers/:id /basket/:productId	Error	Error	Add specific product to specific customer basket	Delete specific product from basket for specific customer

/products	Error	Get a list of all products	Error	Error
/products/:id	Error	Get specific product	Error	Error
/products/categories/category	Error	Get list of available categories	Error	Error
/products/category/:category	Error	Get list of products by categories	Error	Error
/products/category/:category/type/:type	Error	Get list of products in specific category by type	Error	Error

The detailed operation specifications can be found in Appendix.

Software architecture

Logical & physical architecture

On the client-side, the single-page application for this project is built on a layered architecture pattern with React as framework. This type of architecture allows us to decompose the system into horizontal layers, where each layer holds different kinds of functionality and performs a specific role within the application. The application consists of a Presentation layer, Application layer and Data layer. The Presentation layer is responsible for handling the user interface and formats the data for display on the screen. The Application layer runs the logic in our application by processing commands and data between the Presentation layer and Data Layer. The Data layer stores all the data about the products and customers in an array in JSON format. By using HTTP protocol, the data can be retrieved by the Application layer for processing and eventually up to the Presentation layer.

The server-side for the application is implemented by using the Express framework, which is based on the Model-View-Controller pattern. The View component is represented by HTTP requests and responses, which contains data in JSON format, these requests and responses are processed by the Controller. The Controller is also responsible for executing functions to the access data layer after an event has been triggered.

In terms of tiered architecture, we argue that the application is an example of a typical two-tier client-server architectural pattern. Practically, given that the server currently necessitates that requests are sent to localhost, one could argue that it is one-tiered, all located on one machine. Theoretically, however, the future intention of this application would be to separate the client and server machine such that n-amount of decentralized client machines can send requests to a limited number of dedicated centralized servers.

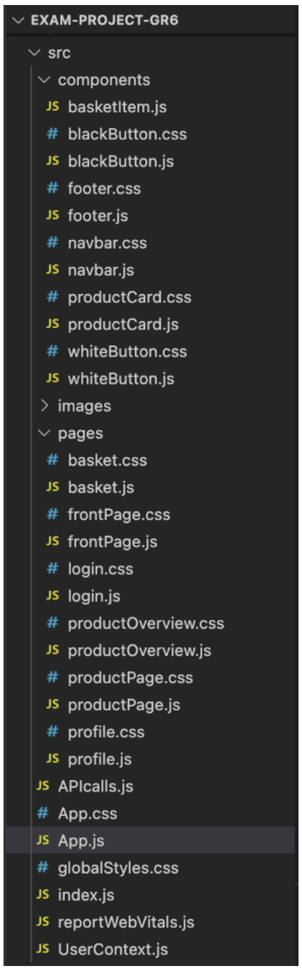
The architecture mentioned above results in different origins between the client-side and server-side. This is referred to as Cross-Origin Resource Sharing (CORS). CORS is used to restrict data coming from different domains, frequently implemented for security reasons. In this application, the two-tier structure entails that the server is running from a different port than the client. By default, CORS would reject such a connection. To accommodate this problem, the Cross-Origin Resource Sharing (CORS) mechanism has been implemented and our server has been configured to support CORS, to allow requests made on behalf of an authorized client. In this situation, this comes in the form of the server architecture specifically enabling CORS. Currently, it accepts requests from all domains. While not ideal for information security, in this instance it is acceptable due to the circumstances this software will be running under. In an actual environment, it would be advisable to restrict CORS to pre-approved domains to the extent possible. Further, CORS can also be used to limit the type of requests received, such as only allowing GET methods (for example, a weather API allowing other services to get current temperature information would perhaps be advised not to allow POST/PUT/DELETE as there would be no need for these).

Software style

The application for this project is developed in React using JavaScript. Below we list some important files of the client-side applications and briefly explain their purpose (see Figure 5 for an image of folder structure).

- APIcalls.js: Defines and exports functions for the purpose of calling the server via RESTful API.
- GlobalStyles.css: Defines global style elements for the site such as font and headings.
- UserContext.js: Stores information on the authentication of the user, and if a user has registered the name and id is stored.
- Pages (folder): This folder holds the different pages the user can navigate to. Each page consists of a .js and .css file, and represent the pages shown in the wireframe in the 'Site Design' section.
- Components (folder): This folder includes the components developed for the project for the purposes of reusability. Most components consist of a .js and .css file.

- Images (folder): Contains all the images used by the site.



```
EXAM-PROJECT-GR6
  src
    components
      basketItem.js
      blackButton.css
      blackButton.js
      footer.css
      footer.js
      navbar.css
      navbar.js
      productCard.css
      productCard.js
      whiteButton.css
      whiteButton.js
    images
    pages
      basket.css
      basket.js
      frontPage.css
      frontPage.js
      login.css
      login.js
      productOverview.css
      productOverview.js
      productPage.css
      productPage.js
      profile.css
      profile.js
    APIcalls.js
    App.css
    App.js
    globalStyles.css
    index.js
    reportWebVitals.js
    UserContext.js
```

Figure 5: Image of folder structure.

Several components were developed for this project. For example, the component BasketItem represents a single row of an item that a user could add to their basket, which would be repeated for however many items a user has in their basket. Some of the developed components make use of props in order to be able to send data to the components. Other components include BlackButton, Footer, Navbar, ProductCard and WhiteButton, which are used on most of the pages in the application.

Furthermore, some third-party components were used. For example, the navigation bar is implemented using the React Bootstrap components Navbar, Nav and NavDropdown, as well as the components FaUser and FaShoppingCart from Font Awesome. Another example is the navigational aspects of the site which are implemented using the BrowserRouter, Route, Routes and Link from React Router.

In order to share data between the different components, the Context API has been implemented as a technique to store the id, name and authentication of a registered user. The Context API stores the data at the top-level of the application and passes the data through the components.

Notes on the Project

Running the Project

We advise that the server is launched before the client-side application, as the API calls of the client-side application assumes that the server runs on localhost:3000.

Disclaimer

The application has two bugs. The first is when registering you need to click twice on the login button before the profile will appear. The second one is that the state will not persist if the user reloads the browser. This means if a user has logged in and reloads the page, the user will no longer be logged in – and are therefore again using the application without registration.

Appendix

Detailed operation specifications

Path: /customers

Method: POST

Summary: Create a new customer.

URL Params: Customer

Body: {“id”: 1, “firstname”: “John”, “surname”: ”Jameson”, “basket”: []}

Success Response:

> **Code:** 200 OK
> Body Content:

Error Response:

> **Code:** 400 BAD REQUEST
> Body Content: Customer with ID: ${customer.id} already exists in database

Sample Call:

```
let customer= {“id”: 1, “firstname”: “John”, “surname”: “Jameson”, “basket”: []}

let response = await fetch('/customers’,

        {method: 'POST',

          headers: {'Content-Type': 'application/json;charset=utf-8'},

          body: JSON.stringify(customer)

});
```

Path: /customers/:id

Method: GET

Summary: Retrieve details about a customer

URL Params: customerId: number

Body:

Success Response:

> **Code:** 200 OK
> Body Content: {“id”: 1, “firstname”: “John”, “surname”: “Jameson”, “basket”: []}

Error Response:

> **Code:** 404 NOT FOUND
>
> **Body Content:** Customer with ID:${customerId} doesn't exist

Sample Call:

```
let id = 1,

let response = await fetch('/customers/{$customerID}’,

        {method: ‘GET’,

                headers: {'Content-Type': 'application/json;charset=utf-8'},

          });
```

Path: /customers/:id

Method: PUT

Summary: Update details about a customer

URL Params: Customer

Body: {“id”: 1, “firstname”: “John”, “surname”: “Jameson”, “basket”: []}

Success Response:

> **Code:** 200 OK
>
> **Body Content:**

Error Response:

> **Code:** 400 BAD REQUEST
>
> **Body Content:** Customer with ID: ${customerId} doesn't exist

Sample Call:

```
let customer = {“id”: 1, “firstname”: “Jens”, “surname”: “Jameson”, “basket”: []};
let response = await fetch('/customers/{$customerID}’,
        {method: PUT,
        headers: {'Content-Type': 'application/json;charset=utf-8'},
});
```

Path: /customers/:Id

Method: DELETE

Summary: Delete customer

URL Params: customerId: number

Body:

Success Response:

Code: 200 OK

Body Content:

Error Response:

Code: 400 BAD REQUEST

Body Content: Customer with ID: ${customerId} doesn't exist

Sample Call:

```
let id = 1;
let response = await fetch('/customers/{$customerID}’,
        {method: ‘DELETE,
              headers: {'Content-Type': 'application/json;charset=utf-8'},
        });
```

Path: /customers/:customerID/basket

Method: GET

Summary: Get product(s) in basket for a specific customer

URL Params: customerId: number

Body:

Success Response:

Code: 200 OK

Body Content: [{"id": 5, "title": "Twill Car Coat", "description": "Car coat in twill. Collar, concealed buttons at front, long sleeves, and adjustable cuffs with tab and buttons. Diagonal front pockets, two inner pockets, and a vent at back. Lined. ", "price": 700, "image": "", "category": "men", "type": "tops", "sizes": ["S", "M", "L", "XL"]}, {}…]

Error Response:

Code: 400 BAD REQUEST

Body Content: Customer with ID: ${customerId} doesn't exist

Sample Call:

```
let id= 1,
let response = await fetch('/customers/{$customerID}/basket',
        {method: 'GET,
                headers: {'Content-Type': 'application/json;charset=utf-8'},
        });
```

Path: /customers/:customerID/basket/:productID

Method: POST

Summary: Add product to basket

URL Params: customerId: number, productId: number

Body:

Success Response:

Code: 200 OK

Body Content: {"id": 1, "firstname": "John", "surname": "Jameson", "basket": [{"id": 5, "title": "Twill Car Coat", "description": "Car coat in twill. Collar, concealed buttons at front, long sleeves, and adjustable cuffs with tab and buttons. Diagonal front pockets, two inner pockets, and a vent at back. Lined. ", "price": 700, "image": "", "category": "men", "type": "tops", "sizes": ["S", "M", "L", "XL"]},]}

Error Response:

Code: 400 BAD REQUEST

Body Content: {Product with ID: ${productID} dosen't exists}

Sample Call:

```
Let customerId = 1,
let productId = 5,

let response = await fetch('/customers/{$customerID}/basket/{$productID}',

        {method: 'POST,

                headers: {'Content-Type': 'application/json;charset=utf-8'},

                body: JSON.stringify(customerId)

        });
```

Path: /customers/:customerID/basket/:productID

Method: DELETE

Summary: Delete product from basket

URL Params: customerId: number, productID: number

Body:

Success Response:

> **Code:** 200 OK
>
> **Body Content:**

Error Response:

> **Code:** 400 BAD REQUEST
>
> **Body Content:** Basket for customer with ID: ${customerId} does not contain Product ID: ${ProductId}

Sample Call:

```
Let customerId = 1,
let productId = 5,

let response = await fetch('/customers/{$customerID}/basket/{$productID}',

        {method: 'DELETE,

                headers: {'Content-Type': 'application/json;charset=utf-8'},

        });
```

Path: /products

Method: GET

Summary: Get all products

URL Params:

Body:

Success Response:

Code: 200 OK

Body Content: [{“id”:1, “title”:”Cardigan with zip”, “description”:”Soft cardigan ...”, “price”:200, “image”:””, “category”:”men”, “type”:”tops”, “sizes”: ["S”, “M”]}, {}, {}, {} …]

Error Response:

Code: 400 BAD REQUEST

Body Content:

Sample Call:

```
let products = [];

let response = await fetch(/products,

        {method: ‘GET,

              headers: {'Content-Type': 'application/json;charset=utf-8'},

        });
```

Path: /products/:productID

Method: GET

Summary: Retrieve details about a product

URL Params: productID: number

Body:

Success Response:

> **Code:** 200 OK
>
> **Body Content:** {“id”:1, “title”:”Cardigan with zip”, “description”:”Soft cardigan ...”, “price”:200, “image”:””, “category”:”men”, “type”:”tops”, “sizes”: ["S”, “M”]}

Error Response:

> **Code:** 400 BAD REQUEST
>
> **Body Content:** Product with ID:{productId} doesn't exist

Sample Call:

```
let productID = 12
let response = await fetch(‘/products/{$productID}’,
        {method: ‘GET,
                headers: {'Content-Type': 'application/json;charset=utf-8'},
        });
```

Path: /products/categories/category

Method: GET

Summary: Get list of available categories

URL Params:

Body:

Success Response:

Code: 200 OK

Body Content: [“men”, “women”]

Error Response:

Code: 400 BAD REQUEST

Body Content:

Sample Call:

```
let response = await fetch('/products/categories/category’,
        {method: ‘GET,
                headers: {'Content-Type': 'application/json;charset=utf-8'},
        });
```

Path: /products/category/:category

Method: GET

Summary: Retrieve products from a category

URL Params: category: string

Body:

Success Response:

Code: 200 OK

Body Content: [{ "id": 11, "title": "Rib-knit wool-bend cardigan ", "description": " Cropped cardigan in a soft, rib-knit wool blend with a V-neck, buttons down the front and long, cuffed, raglan balloon sleeves.", "price": 450, "image": "", "category": " women", "type": "tops", "sizes": ["S", "M", "L", "XL"]}, {}, {}, {}]

Error Response:

Code: 400 BAD REQUEST

Body Content: Category: ${category} doesn’t exist

Sample Call:

```
let category = women,
let response = await fetch('/products/category/{$category}’,
        {method: ‘GET,
              headers: {'Content-Type': 'application/json;charset=utf-8'},
        });
```

Path: /products/category/:category/type/:type

Method: GET

Summary: Retrieve products by category and type

URL Params: category: string, type: string

Body:

Success Response:

Code: 200 OK

Body Content: [{"id": 16, "title": "Slacks", "description": "Ankle-length cigarette trousers in a stretch weave. Regular waist with concealed elastication, zip fly with a concealed hook-and-eye fastening, side pockets, fake back pockets and tapered legs.", "price": 200, "image": "", "category": "women", "type": "bottoms", "sizes": ["S", "M", "L", "XL"]}, {}, {}, {}]

Error Response:

Code: 400 BAD REQUEST

Body Content: Category: ${category} and Type: ${type} doesn't exist together

Sample Call:

```
Let category = women,
let type = bottoms,

let response = await fetch('/products/category/{$category}/type/{$:type}’,

        {method: ‘GET,

                headers: {'Content-Type': 'application/json;charset=utf-8'},

        });
```