Frameworks and Architectures for the Web

Project 3 - PokéShop

Group 11

Nanna Munk Berg (namb)
Camilla Krogh Dalsgaard (cdal)
Johannes Hermann Palbøl (jhpa)
Sophia Conrad (soco)

May 26, 2023

Contents

1 Web design of the client-side application

1.1 Site definition

Our website is called PokéShop and is set up like a webshop where you can browse and shop for a selection of pokémon. Thus our product is fictional but our intention is for the website to appear as a real online pet shop.

1.2 Information architecture

The advantage of building a webshop with pokémon as products is that they come with predetermined properties and taxonomies. For our purposes, we have elected to group our pokémon products according to the type and size that they have in the online Pokémon Database [1]. We have selected 15 different pokémon, ensuring they cover a range of different types and sizes. Additionally, we also attach a price to each pokémon. The users will be able to sort the products on the three categories of size, type and price, each falling on the same taxonomic level. To make it easier for the customer to find pokémon by size, we put the continuous size values (e.g. 0.6m) into distinct size categories (e.g. `s` for small). Our product taxonomy is depicted in figure 1.

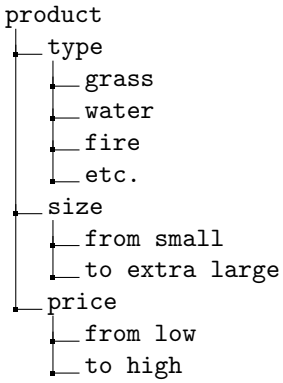

Figure 1: Product taxonomies used in PokéShop.

Since our webshop is quite simple and consists of a few layers, we have decided to go for a flat hierarchical site structure as shown in figure 2. To be clear, our app is a single-page React app but to the user, it will appear to have this simple hierarchical structure. The top layer consists of the front page which leads down to every other level. The only exception is the product detail pages which are found on the level below the products page. This has the advantage of being a conventional structure for webshops, meaning we put less strain on the user because a user will not need to put in the effort to learn new structures specific to our site.

[1] `https://pokemondb.net/pokedex/all`

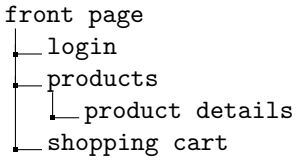

Figure 2: Site structure.

1.3 Site design

In this section, we present wireframes of the PokéShop website.

The main way to navigate around our webshop is by clicking on the different links in the top navigation bar which is shared by all pages (figures 3-7). Here the user can navigate to the products page by clicking "Products", to the login page by clicking "Login", to the shopping cart page by clicking on the shopping cart symbol and, finally, to the front page by clicking on "PokéShop" in the top left.

This is a simple and intuitive way to implement navigation, which is in line with Jakob Nielsen's 4th principle "Consistency and Standards".[2]. The header with the navigation functionality and the footer that contains the company's information are React components that are used on all of the pages.

To make the system status immediately visible to the user (Jakob Nielsen's 1st principle), we implement a counter next to the "Shopping cart" link in the navbar. This shows how many items are in the cart and will update whenever the user adds a new item.

The colour scheme of our webshop is set up to match that of a Pokéball, which has three main colours: white, grey (rgb(55, 55, 55)) and red (rgb(233,74, 74)). The colours are introduced by the large background/welcome image on the front page and are used throughout the site for a consistent and on-theme look. White is used as the background colour, while grey is used in our navigation and menu elements. The red colour is mainly used for buttons. We also have a lighter grey which we use to separate smaller containers from the background, such as on the products page where each product gets a separate container.

[2] `https://www.nngroup.com/articles/ten-usability-heuristics/`

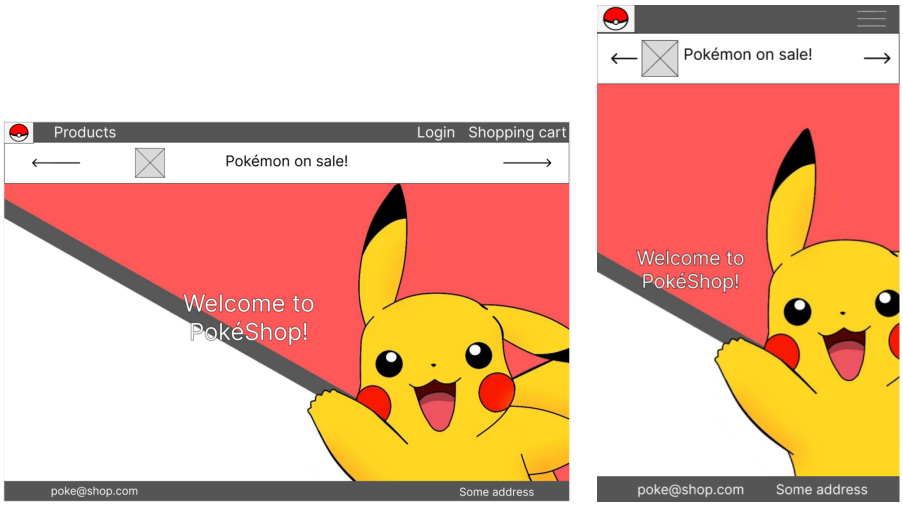

Figure 3: Frontpage wireframe. Left: desktop. Right: mobile

For the starting page (figure 3) we decided to divide the page horizontally into two parts: a carousel of current products on sale at the top and then a large background picture of Pikachu filling out the bulk of the page with a welcome message in the centre. If the user is logged in, their name is displayed in the welcome message. The current offers are displayed one at a time using a bootstrap carousel so that users can see multiple different offers without being overwhelmed by too many pictures at a time. We decided to have the offers displayed towards the top of the page so that no scrolling would be needed.

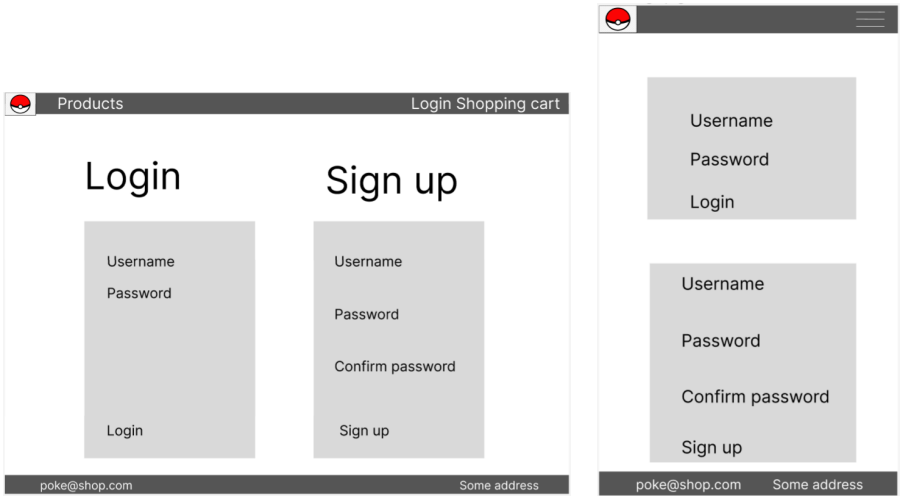

Figure 4: Login wireframe. Left: desktop. Right: mobile

The login page (figure 4) has two different containers. One for logging in with an already existing user, and another for signing up if there is no preexisting user. Note the wireframe has different input form fields than the final forms. The form for signing up is implemented with data validation. The user will be warned if they have not entered a first or a last name, or if they have not entered an email address in the correct format. The login form validates the email and checks if the entered email corresponds to an already existing customer. If the entered data is valid, the user's name will be displayed on the front page and the shopping cart page.

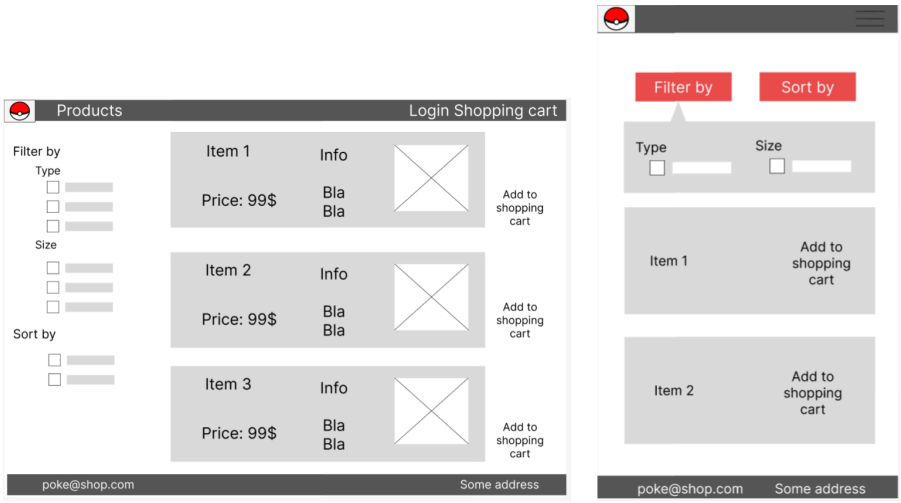

Figure 5: Products page wireframe. Left: desktop. Right: mobile

The products page (figure 5) lists all products. Each product can be added directly to the shopping cart from the products page, or the user can go to the detailed products page which displays more information about the product and add to the cart from there.

The products page also has a sidebar for filtering and sorting the pokémon based on the taxonomy described in section 1.2. This means that the user can filter on type and size.

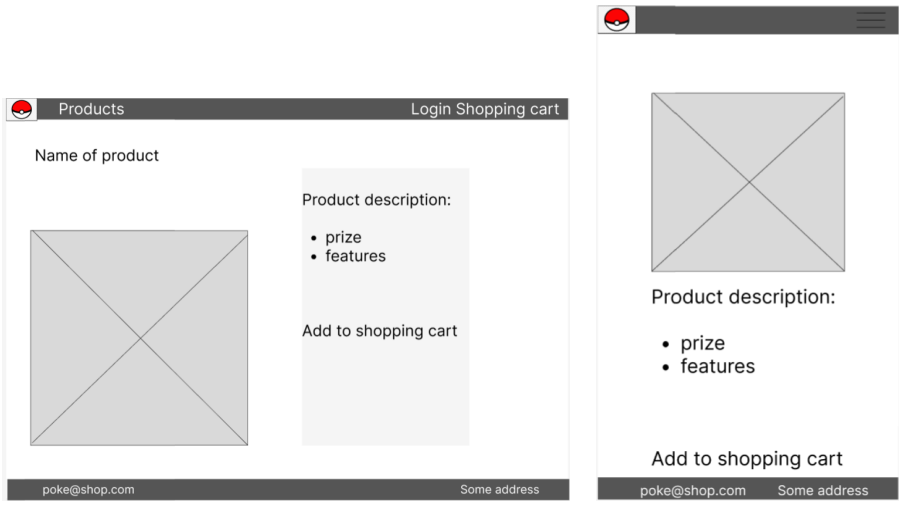

Figure 6: Product details wireframe. Left: desktop. Right: mobile

The detailed product page (figure 6) displays a large image of the pokémon as well as more in-depth information about the product. The user can add to the shopping cart from this page.

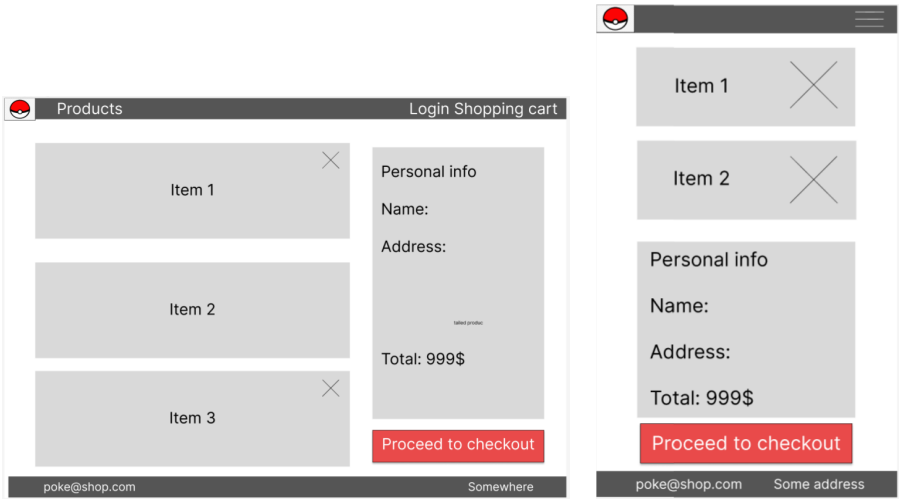

Figure 7: Shopping cart wireframe. Left: desktop. Right: mobile

The last wireframe shows the shopping cart page (figure 7). This page displays all the items that have been added to the shopping cart as well as their total price. Any item can be removed from the shopping cart by clicking on the cross in the top right of the product's container. The user can also add more of the

same product to the shopping cart by clicking on the plus or remove one clickling on the minus.

We have added a "Proceed to checkout"-button which does not have functionality. It is meant to give the impression that the user would be able to input their address and card info to actually purchase the products in the shopping cart.

2 Design of the RESTful API

The backend of our webshop is an app with a RESTful API. REST APIs are a common way of enabling two services to communicate with each other using a client-server architecture. In our case, it enables our frontend/client-side app to access resources that we store in the backend/server-side app.

The REST API (REpresentational State Transfer) relies on resources. Resources are identified by URI's (Uniform Resource Identifiers) which take the following form: https://pokeshop.com/products/{productId}. This example is taken from our own REST API and points to a resource being a product with some specific id. A full overview of resources and operations used in the backend can be found in table 1. Each URI supports a certain number of HTTP methods. The aforementioned URI supports the GET operation which, when invoked, transfers data about a specific product in JSON format. In REST terms, this data is called a representation of the state of the requested resource.

We have followed conventions in naming the URI's of the resources of our web app. This means:

1. basing URI's on nouns instead of verbs - e.g. /customers/{customerId}/baskets instead of /customers/{customerId}/make-baskets
2. organising resources into a hierarchy that isn't too deep

URI	POST	GET	PATCH	DELETE
/customers/{customerId} /baskets	Create shopping basket for customer	Retrieve basket content for customer	-	-
/customers/{customerId} /baskets/products /{productId}	-	-	Put product in the basket for customer	Remove product from basket
/products	-	Retrieve all products	-	-
/products/categories	-	Retrieve all categories	-	-
/products/categories /{genericCategory} /{specificCategory}	-	Retrieve information about products in category	-	-
/products/{productId}	-	Retrieve information about specific product	-	-
/customers	Add new customer	-	-	-
/customers/{email}	-	Get customer based on email	-	-
/customers/{email} /baskets	-	-	-	Clean basket

Table 1: REST API URI's and operations

3 Software Architecture

The code is organized in two different folders: the frontend and the backend folder. The backend and the frontend have to run at the same time but on two different servers in order to communicate with each other. The frontend has to display information about the customer and the products in their shopping basket that it fetches from the backend.

3.1 Frontend

The frontend folder contains folder `public` with the images of our pokémon, as well as a `src` folder containing all of our TypeScript files as well as specific CSS files for the single components. We have one .tsx file for static data, i.e. an object that maps pokémon types to a colour code (to make containers that have the correct background colour depending on the type). We have separate components for the footer and header, which are shown across the whole webpage. Some components correspond to the main content of a page e.g. the detailed product page. There are separate functional components called `LoginForm` and `SignUpForm` that are children of the `Forms` functional component.

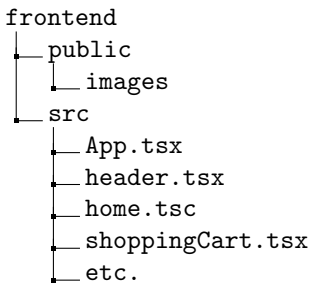

Figure 8: File structure of the frontend.

Pages are rendered conditionally depending on the HTTP path the user is on. If no path is specified, the carousel and the welcome banner are rendered in addition to the header and the footer and if they click on e.g. "Login", the `Forms` component will be rendered between the header and the footer. We use React Router in our project, where the routing renders the components based on the provided URI and parameters as shown in figure 9.

```
<Routes>
    <Route path="/" element={
        <UserContext.Provider value={newGetUserContext}>
            <Home />
        </UserContext.Provider>}/>
    <Route path="/signup" element={
        <SetUserContext.Provider value={newSetUserContext}>
            <Forms/>
        </SetUserContext.Provider>}/>
```

Figure 9: Part of the code for the conditional rendering.

Our components are made up of multiple bootstrap-react components such as Carousel, Card, or Container, in order to make the page responsive and to make it look consistent.

For the most part, we avoid prop drilling and lifting states and instead use the Context API to share information consistently across components. For example, we create a `UserContext` with an empty string as the user name and -1 for the user id that gets updated possibly multiple times. This is possible by using the context in combination with the `useState` hook, where we have the user and the user id as states. The first time the state and thus the context get updated is using the `useEffect` hook that calls the function `setDefaultUser` to fetch a default user from the backend that the user and user id states get set to using the set-methods returned from `useState`. They get updated once again when the user decides to log in to their already existing account or when creating a new account.

By wrapping components in `UserContext.Provider` and passing the user and id state variables as values they get provided with read access to the `UserContext`, that is e.g. needed to display a logged-in user's name on the front page (see figure 9).

On some occasions, we pass props, as in the case with components `DecButton` and `IncButton`. Both these components are passed callback functions to be executed when they are clicked. Since nobody else needs these functions and since there are no intermediaries and other components through which they pass (there is no actual prop drilling happening), there is no need to create contexts.

3.2 Backend

Our RESTful API backend is implemented in JavaScript using Node.js and the Express framework. This means that whenever the client-side of our app requests something from the server-side it passes through a Node server which hands over the request to an Express app. The Express app will process the request using middleware functions, return the result to the Node server, and then finally the result of the request will be transferred back to the client. This structure allows us to use the Model-View-Controller architecture for our app. This architecture has the advantages of being modifiable and making it easy to provide different views. Essentially, it allows us to add new views/components and update them without having to change the overall structure of the app. Since the View has to do with the frontend of our app, we will discuss the Controller and Model in the following paragraphs.

First off, it is important to note that the backend of our app is divided in two. This stems from the fact that our REST API is based on two main resources and thus URI's: `/customers` and `/products`. Thus, we have separate logic for routing `/customers` requests and `/products` requests. Both of these routers are built with the same architecture, as can be seen in Figure 10.

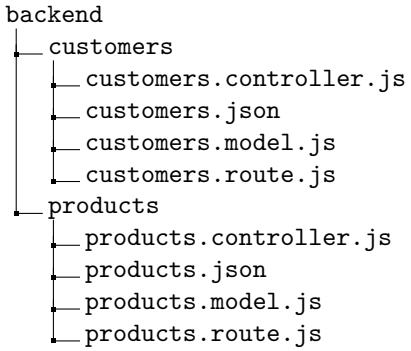

Figure 10: File structure of the backend. Note that it is essentially divided in two parts: customers and products.

To explain how the Model and Controller parts work, we can imagine an example where a user of our app makes an HTTP request. This occurs in the View part of the app since that is what the user interacts with. If the user made a GET request to the `/products` endpoint that would be handled by the products Controller (`products.controller.js`). The Controller will make a request to the Model. The Model then handles the actual logic of getting the data and returning it to the Controller. In the case of a /products GET request we have a function that makes an asynchronous call to a Model function that reads and returns all products from the JSON file where they are stored. The Controller then updates its response object with the `res.json()` method. Finally, the controller sends this response back to the View (in this particular case, to the function/component that handles the products page) where the products will then be rendered for the user to see.

As a final note on the backend, it should be mentioned that we use middleware to parse JSON and CORS. CORS (Cross-Origin Resource Sharing) enables the app to send HTTP requests between a client and server whose origins are not the same. By default, sending requests between two different origins is not allowed without authentication but it is needed for our app since we run the backend on a different port than the frontend.

4 Appendix

4.1 Run PokéShop

To run our webshop you need to execute the following commands:

```
$ unzip group11-final-project.zip
```

Backend (port 3005):

```
$ cd poke-app/backend
$ npm install
$ node pokemonAPI.js
```

Then, frontend (port 3000):

```
$ cd ../frontend
$ npm install
$ npm start
```

Navigate to https://localhost:3000 in your favourite browser and enjoy!

OBS. Try using `npm ci` instead of `npm install` if errors are encountered.

4.2 Operation specifications from 2nd project

POST to shopping basket

Path: **/customers/{customerId}/baskets**

Method: POST

Summary: Create a shopping basket for the specified customer.

URL Params: customerId: integer, required in path

Body: -

Success Response:

> **Code:** 201 CREATED
> **Body Content:** -

Error Response: For example:

> **Code:** 404 NOT FOUND
> **Body Content:** { error: "Customer with id 12 already has a basket." }

Sample Call:

```
let customerId = 3

let response = await fetch('/customers/{$customerId}/baskets,

                { method: 'POST',

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

GET shopping basket

Path: **/customers/{customerId}/baskets**

Method: GET

Summary: Retrieve the basket content for the specified customer.

URL Params:

customerId: integer, required in path

Body: -

Success Response:

Code: 200 OK
Body Content:

{ 'customerId': 3,

'basket': [

{'amount': 3,

'product': { 'id': 1,

'name': "Bulbasaur",

'type': ['grass', 'poison'],

'size': 0.7,

'sizeCategory': 'm',

'price': 40000,

'info': '"Your new best friend. A basic grass pokemon who will follow you through thick and thin"'

}},

…]}

Error Response: For example:

Code: 404 NOT FOUND
Body Content: {error: "Customer with id 12 does not have a basket"}

Sample Call:

```
let customerId = 3

let response = await fetch(´/customers/{$customerId}/baskets´,

                          { method: 'GET',

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

PATCH in shopping basket

Path: /customers/{customerId}/baskets/products/{productId}

Method: PUT

Summary: Put product in the shopping basket for a specified customer.

URL Params:

- customerId: integer, required in path
- productId: integer, required in path

Body: For example:

{‘amount’: 3 }

Success Response:

Code: 200 OK
Body Content:

{ 'customerId': 3,

‘basket’: [

{‘amount’: 3,

‘product’: { ‘id’: 1,

‘name’: "Bulbasaur",

‘type’: [‘grass’, ‘poison’],

‘size’: 0.7,

‘sizeCategory’: ‘m’,

‘price’: 40000,

‘info’: ‘"Your new best friend. A basic grass pokemon who will follow you through thick and thin"’

}},

…]}

Error Response: For example:

Code: 404 NOT FOUND
Body Content: {error: "Customer with id 12 does not have a basket"}

Sample Call:

```
let customerId = 3

let productId = 2

let amount = { amount: 3 }

let response = await fetch('/customers/{$customerId}/baskets/products/{$productId}

        {       method: 'PUT',

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

                body: JSON.stringify(amount)

        });
```

DELETE from shopping basket

Path: /customers/{customerId}/baskets/products/{productId}

Method: DELETE

Summary: Remove a product from basket for the specified customer

URL Params:

- customerId: integer, required in path
- productId: integer, required in path

Body: For example: { ‘amount’: 3 }

Success Response:

Code: 200 OK
Body Content:

{ 'customerId': 3,

‘basket’: [

{‘amount’: 3,

‘product’: { ‘id’: 1,

‘name’: "Bulbasaur",

‘type’: [‘grass’, ‘poison’],

‘size’: 0.7,

‘sizeCategory’: ‘m’,

‘price’: 40000,

‘info’: ‘"Your new best friend. A basic grass pokemon who will follow you through thick and thin"’

}},

…]}

Error Response: For example:

> **Code:** 404 NOT FOUND
> **Body Content:** {error: "Customer with id 12 does not have a basket"}

Sample Call:

```
let customerId = 3

let productId = 2

let amount = { amount: 3 }

let response = await fetch('/customers/{$customerId}/baskets/products/{$productId}

        {       method: 'DELETE',

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

                body: JSON.stringify(amount) });
```

GET products

Path: **/products**

Method: GET

Summary: Retrieve the most important information about all products

URL Params: No params.

Body: -

Success Response:

Code: 200 OK
Body Content: [{‘name’: "Bulbasaur",

‘type’: [‘grass’, ‘poison’],

‘size’: ‘0.7’,

‘sizeCategory’: ‘m’,

‘price’: ‘40000’,

‘info’: ‘"Your new best friend. A basic grass pokemon who will follow you through thick and thin"’},

{...}, …]

Error Response: For example:

Code: 404 NOT FOUND
Body Content: { error: "Page not found"}

Sample Call:

```
let response = await fetch(´/products´,

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

GET product categories

Path: **/products/categories**

Method: GET

Summary: Retrieve all the product categories

URL Params: No params.

Body: -

Success Response:

Code: 200 OK
Body Content: { “type”: [‘grass’, ‘poison’, ‘fire’, …] ,

“sizeCategory”: [‘s’, ‘m’, ‘l’, ‘xl’, …]}

Error Response:

Code: 404 NOT FOUND
Body Content: {error: "Page not found"}

Sample Call:

```
let response = await fetch(´/products/categories´,

                { method: 'GET',

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

GET products of specific category

Path: **/products/categories/{genericCategory}/{specificCategory}**

Method: GET

Summary: Get most important information about products for a specific category.

URL Params: genericCategory: string, required in path; specificCategory: string, required in path;

Body: -

Success Response:

Code: 200 OK
Body Content: [

{‘name’: "Bulbasaur",

‘type’: [‘grass’, ‘poison’],

‘size’: 0.7,

‘sizeCategory’: ‘m’,

‘price’: 40000,

‘info’: ‘"Your new best friend. A basic grass pokemon who will follow you through thick and thin"’,

… }]

Error Response: For example:

Code: 404 NOT FOUND
Body Content: {error: "Category doesn't exist"}

Sample Call:

```
let genericCategory = ‘type’

let specificCategory = ‘water’

let response = await
fetch(´/products/categories/{$genericCategory}/{$specificCategory}´,
```

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

GET information of specific product

Path: **/products/{productId}**

Method: GET

Summary: Get all details about a specific product.

URL Params: productId: integer, required in path

Body: -

Success Response:

Code: 200 OK
Body Content:

{ ‘id’: 12,

‘name’: "Bulbasaur",

‘type’: [‘grass’, ‘poison’],

‘size’: 0.7,

‘sizeCategory’: ‘m’,

‘price’: 40000,

‘info’: ‘"Your new best friend. A basic grass pokemon who will follow you through thick and thin"’ }

Error Response: For example:

Code: 404 NOT FOUND
Body Content: { error: "Product not found"}

Sample Call:

```
const productId = 12

let response = await fetch(´/products/{$productId}´,

                { method: 'GET',

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