Exercise: HTTP and AJAX

Problems for exercises and homework for the "JS Front-End" course @ SoftUni

Working with Remote Data

For the solution of some of the following tasks, you will need to use an up-to-date version of the **local REST service**, provided in the lesson's resources archive. You can read the documentation here.

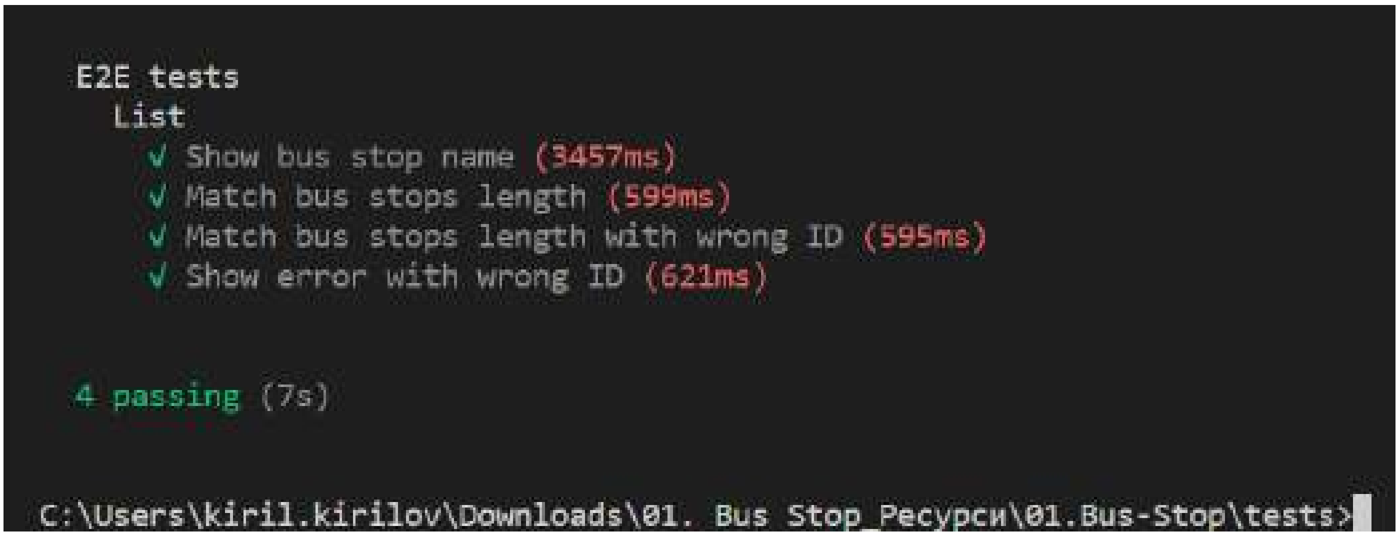
Requirements

For each task you need to install all dependencies using the "**npm install**" command

Then you can start the front-end application with the "**npm start**" command

You also must also start the **server.js** file in the server folder using the "**node server.js**" command in another console **(BOTH THE CLIENT AND THE SERVER MUST RUN AT THE SAME TIME)**

At any point, you can open up another console and run "**npm test**" inside the **tests subfolder** for the problem to test the **current state** of your application, it's preferable for **all of your test to pass locally** before you submit to the judge platform, like this:



```
  E2E tests
    List
      √ Show bus stop name (3457ms)
      √ Match bus stops length (599ms)
      √ Match bus stops length with wrong ID (595ms)
      √ Show error with wrong ID (621ms)


  4 passing (7s)


C:\Users\kiril.kirilov\Downloads\01. Bus Stop_Ресурси\01.Bus-Stop\tests>
```

1. Blog

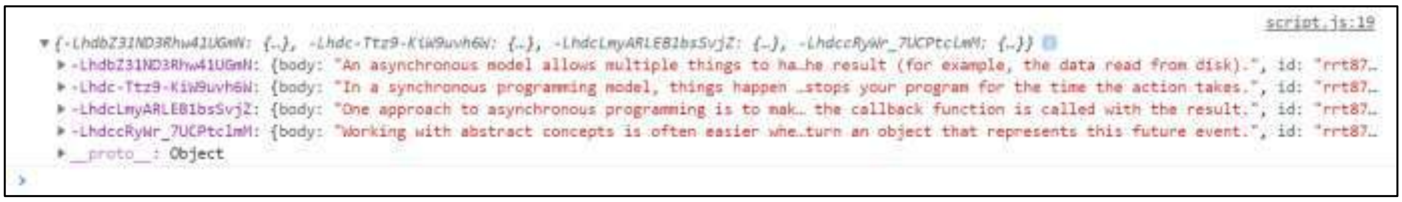
Write a program for reading blog content. It needs to make **requests** to the **server** and display **all blog posts** and their **comments**.
Request URL's:

Posts - **`http://localhost:3030/jsonstore/blog/posts`**

Comments - **`http://localhost:3030/jsonstore/blog/comments`**

Examples

The button with ID "**`btnLoadPosts`**" should make a **GET** request to "**`/posts`**". The **response** from the **server** will be an **Object of objects.**

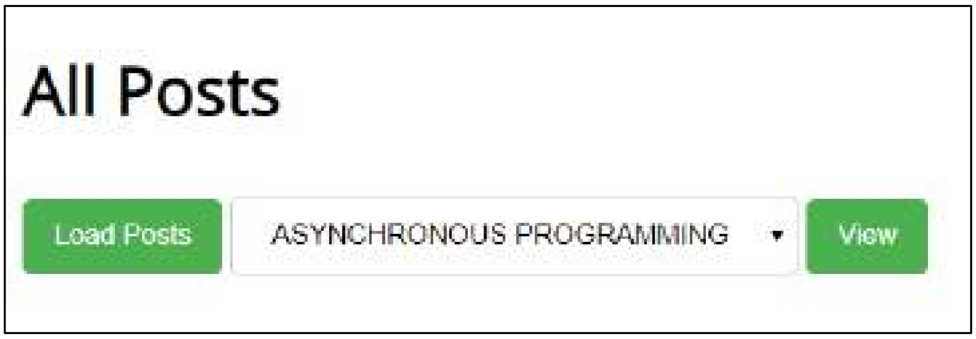


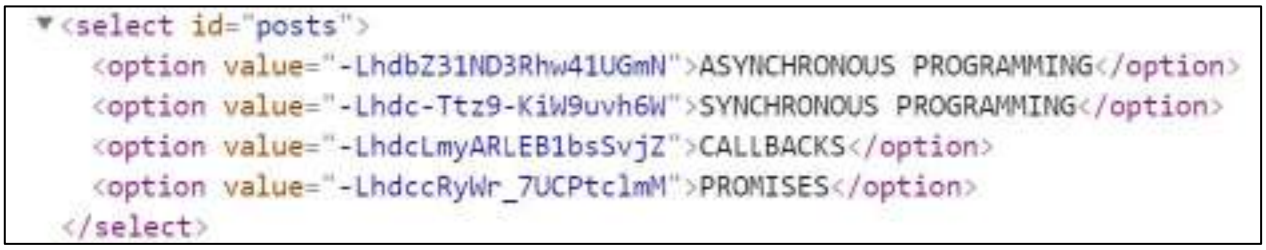
```
                                                                                                      script.js:19
▼{-LhdbZ31ND3Rhw41UGmN: {…}, -Lhdc-Ttz9-KiW9uvh6W: {…}, -LhdcLmyARLEB1bsSvjZ: {…}, -LhdccRyWr_7UCPtclmM: {…}}
  ▶-LhdbZ31ND3Rhw41UGmN: {body: "An asynchronous model allows multiple things to ha…he result (for example, the data read from disk).", id: "rrt87…
  ▶-Lhdc-Ttz9-KiW9uvh6W: {body: "In a synchronous programming model, things happen …stops your program for the time the action takes.", id: "rrt87…
  ▶-LhdcLmyARLEB1bsSvjZ: {body: "One approach to asynchronous programming is to mak… the callback function is called with the result.", id: "rrt87…
  ▶-LhdccRyWr_7UCPtclmM: {body: "Working with abstract concepts is often easier whe…turn an object that represents this future event.", id: "rrt87…
  ▶__proto__: Object
>
```

Each object will be in the following format:

```
{
  body: {postBody},
  id: {postId},
  title: {postTitle}
}
```

Create an **`<option>`** for each post using its **object key** as value and **current object title property** as text inside the node with ID "**`posts`**".

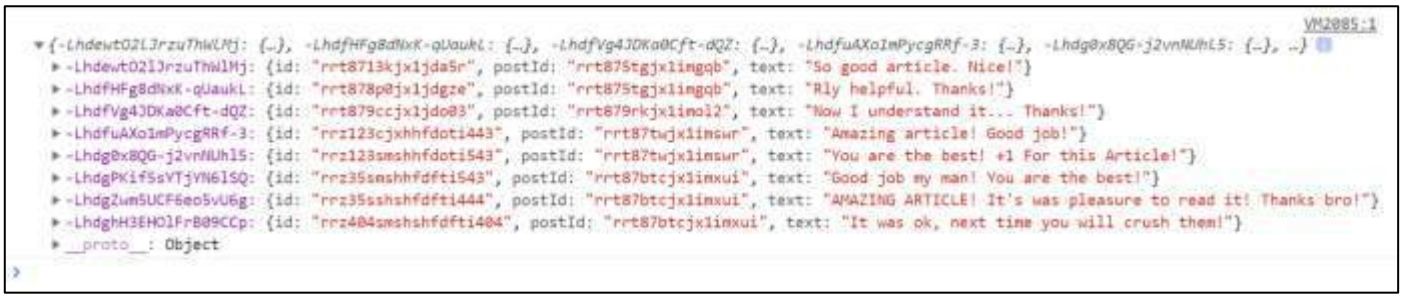




```
▼<select id="posts">
   <option value="-LhdbZ31ND3Rhw41UGmN">ASYNCHRONOUS PROGRAMMING</option>
   <option value="-Lhdc-Ttz9-KiW9uvh6W">SYNCHRONOUS PROGRAMMING</option>
   <option value="-LhdcLmyARLEB1bsSvjZ">CALLBACKS</option>
   <option value="-LhdccRyWr_7UCPtclmM">PROMISES</option>
 </select>
```

When the button with ID "**`btnViewPost`**" is clicked, a **GET** request should be made to:

- "**`/comments`**" **-** to obtain all comments. The request will **return** a **Object** of **objects**.



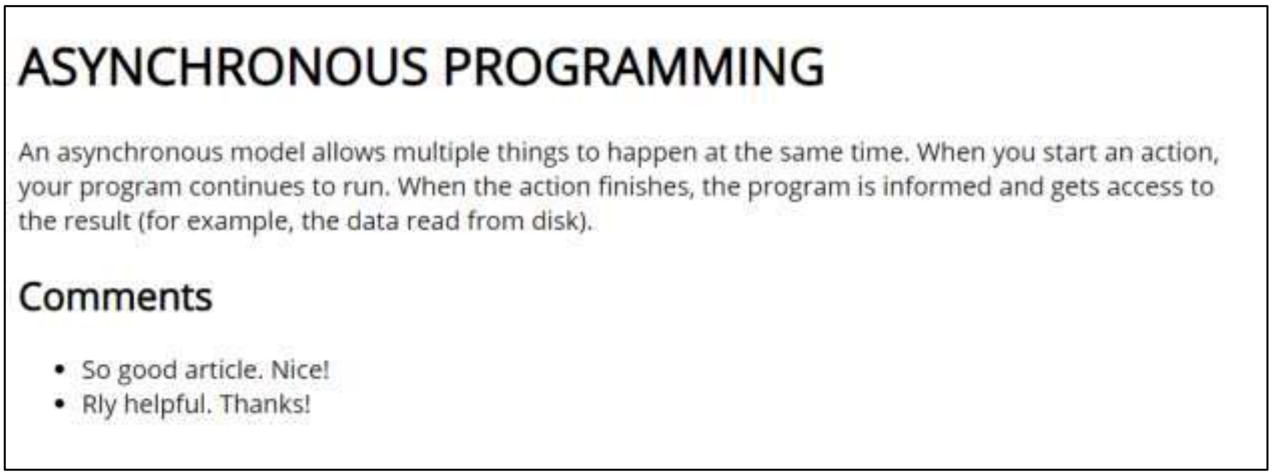
```
                                                                                                      VM2085:1
▼{-LhdewtO2L3rzuThWlMj: {…}, -LhdfHFg8dNxK-qUaukL: {…}, -LhdfVg4JDKa0Cft-dQZ: {…}, -LhdfuAXoImPycgRRf-3: {…}, -Lhdg0x8QG-j2vnNUhl5: {…}, …}
  ▶-LhdewtO2l3rzuThWlMj: {id: "rrt8713kjx1jda5r", postId: "rrt875tgjx1imgqb", text: "So good article. Nice!"}
  ▶-LhdfHFg8dNxK-qUaukL: {id: "rrt878p0jx1jdgze", postId: "rrt875tgjx1imgqb", text: "Rly helpful. Thanks!"}
  ▶-LhdfVg4JDKa0Cft-dQZ: {id: "rrt879ccjx1jdo03", postId: "rrt879rkjx1imol2", text: "Now I understand it... Thanks!"}
  ▶-LhdfuAXoImPycgRRf-3: {id: "rrz123cjxhhfdoti443", postId: "rrt87twjx1imswr", text: "Amazing article! Good job!"}
  ▶-Lhdg0x8QG-j2vnNUhl5: {id: "rrz123smshhfdoti543", postId: "rrt87twjx1imswr", text: "You are the best! +1 For this Article!"}
  ▶-LhdgPKif5sVTjYN6lSQ: {id: "rrz35smshhfdfti543", postId: "rrt87btcjx1imxui", text: "Good job my man! You are the best!"}
  ▶-LhdgZwm5UCF6eo5vU6g: {id: "rrz35sshshfdfti444", postId: "rrt87btcjx1imxui", text: "AMAZING ARTICLE! It's was pleasure to read it! Thanks bro!"}
  ▶-LhdgH3EHOlFrB09CCp: {id: "rrz404smshshfdfti404", postId: "rrt87btcjx1imxui", text: "It was ok, next time you will crush them!"}
  ▶__proto__: Object
>
```

Each object will be in the following format:

```
{
  id: {commentId},
  postId: {postId},
  text: {commentText}
}
```

You must find this comments that are for the current post (check the postId property)

Display the post title inside **h1** with ID "**`post-title`**" and the post content inside **p** with ID "**`post-body`**". Display **each comment** as a **`<li>`** inside **ul** with ID "**`post-comments`**". Do not forget to clear its content beforehand.

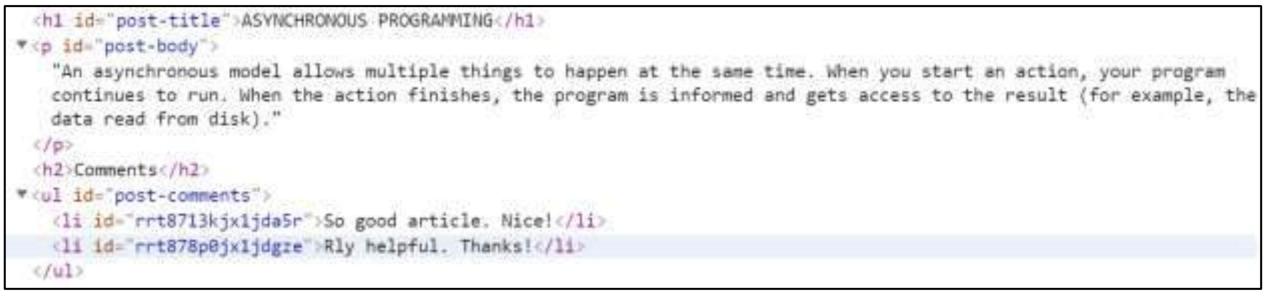


ASYNCHRONOUS PROGRAMMING

An asynchronous model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

Comments

- So good article. Nice!
- Rly helpful. Thanks!



```
  <h1 id="post-title">ASYNCHRONOUS PROGRAMMING</h1>
▼<p id="post-body">
   "An asynchronous model allows multiple things to happen at the same time. When you start an action, your program
   continues to run. When the action finishes, the program is informed and gets access to the result (for example, the
   data read from disk)."
  </p>
  <h2>Comments</h2>
▼<ul id="post-comments">
    <li id="rrt8713kjx1jda5r">So good article. Nice!</li>
    <li id="rrt878p0jx1jdgze">Rly helpful. Thanks!</li>
  </ul>
```

2. Messenger

Write a JS program that records and displays messages. The user can post a message, supplying a name and content and retrieve all currently recorded messages.

The url for the requests - **http://localhost:3030/jsonstore/messenger**

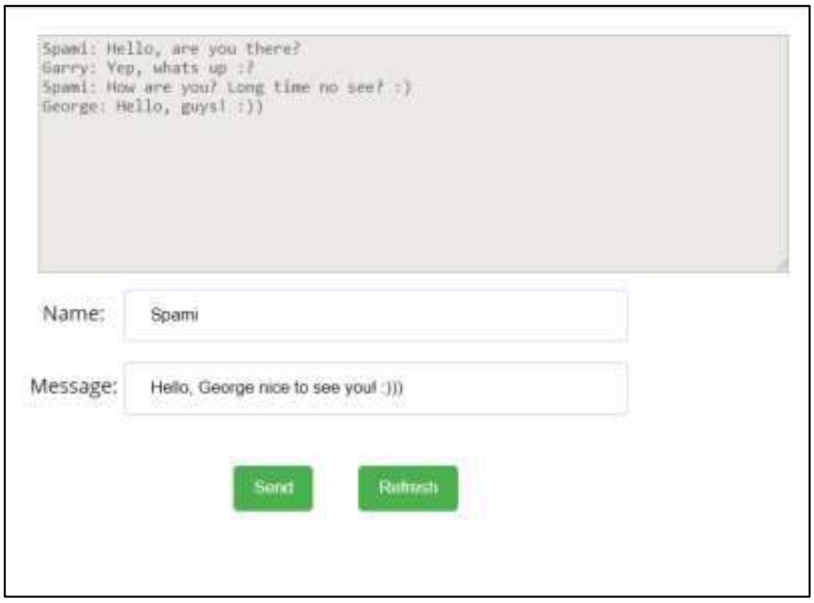
When [**Send**] **button** is clicked you should create a **new object** and send a **post request** to the given url. Use the following message structure:

```
{
  author: authorName,
  content: msgText,
}
```

If you click over [**`Refresh`**] **button** you should **get all** messages with **GET request** and display them into the textarea. Use the following message format:

"**`{author}: {message}`**"

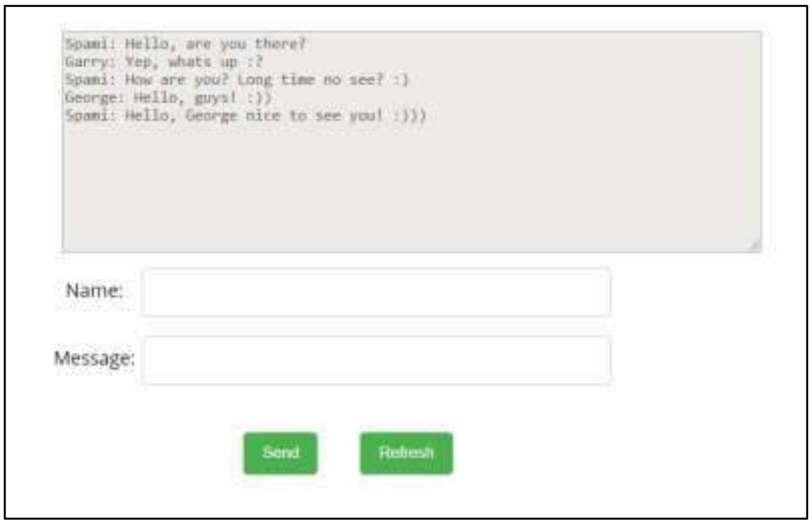
Examples



```
Spami: Hello, are you there?
Garry: Yep, whats up :?
Spami: How are you? Long time no see? :)
George: Hello, guys! :))
```

Name: Spami

Message: Hello, George nice to see you! :)))



```
Spami: Hello, are you there?
Garry: Yep, whats up :?
Spami: How are you? Long time no see? :)
George: Hello, guys! :))
Spami: Hello, George nice to see you! :)))
```

Name:

Message:

3. Phonebook

Write a JS program that can load, create and delete entries from a Phonebook. You will be given an HTML template to which you must bind the needed functionality.

When the **[Load]** button is clicked, a **GET** request should be made to the server to get all phonebook entries. Each received entry should be in a **li** inside the **ul** with **id="phonebook"** in the following format with text **"<person>: <phone> "** and a **[Delete]** button attached. Pressing the **[Delete]** button should send a **DELETE** request to the server and delete the entry. The received response will be an object in the following format:
{<key>:{person:<person>, phone:<phone>}, <key2>:{person:<person2>, phone:<phone2>,…} where **<key>** is an unique key given by the server and **<person>** and **<phone>** are the actual values.

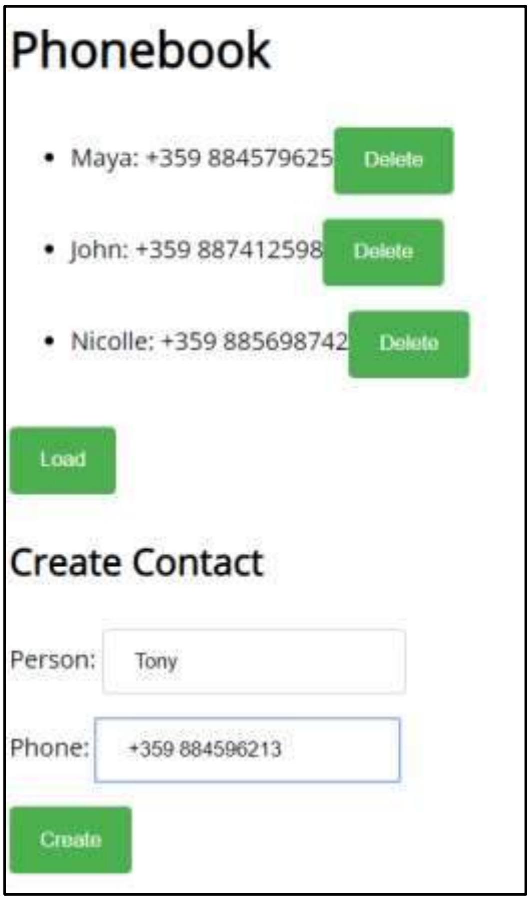
When the **[Create]** button is clicked, a new **POST** request should be made to the server with the information from the Person and Phone textboxes, the Person and Phone textboxes should be cleared and the Phonebook should be automatically reloaded (like if the **[Load]** button was pressed).

The data sent on a POST **request should be a valid JSON object, containing properties** person **and** phone. **Example format:**

```
{
  "person": "<person>",
  "phone": "<phone>"
}
```
The **url's** to which your program should make requests are:

- **GET** and **POST** requests should go to **http://localhost:3030/jsonstore/phonebook**
- **DELETE** requests should go to **http://localhost:3030/jsonstore/phonebook/:key>** , where **:key** is the unique key of the entry (you can find out the **key** from the key property in the **GET** request)

Screenshots


Phonebook

- Maya: +359 884579625 Delete
- John: +359 887412598 Delete
- Nicolle: +359 885698742 Delete

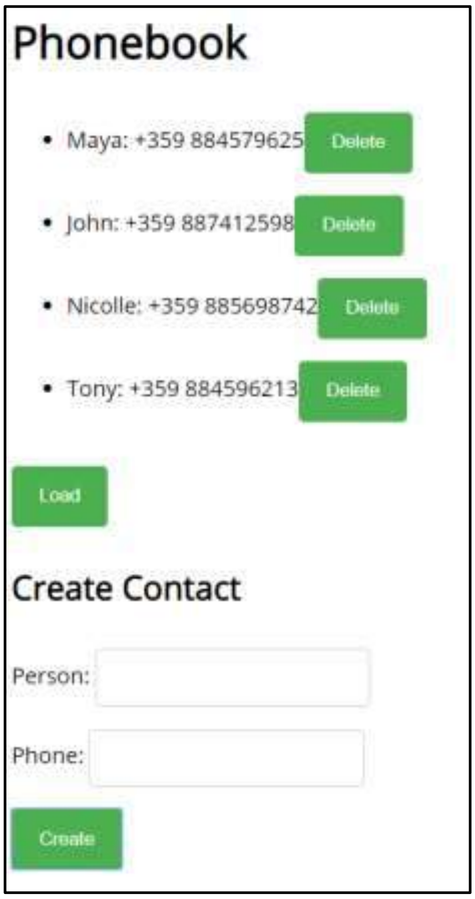
Load

Create Contact

Person: Tony

Phone: +359 884596213

Create


Phonebook

- Maya: +359 884579625 Delete
- John: +359 887412598 Delete
- Nicolle: +359 885698742 Delete
- Tony: +359 884596213 Delete

Load

Create Contact

Person:

Phone:

Create

4. Students

Your task is to implement functionality for creating and listing students from a database. Create a new collection called "**students**",

Each student has:

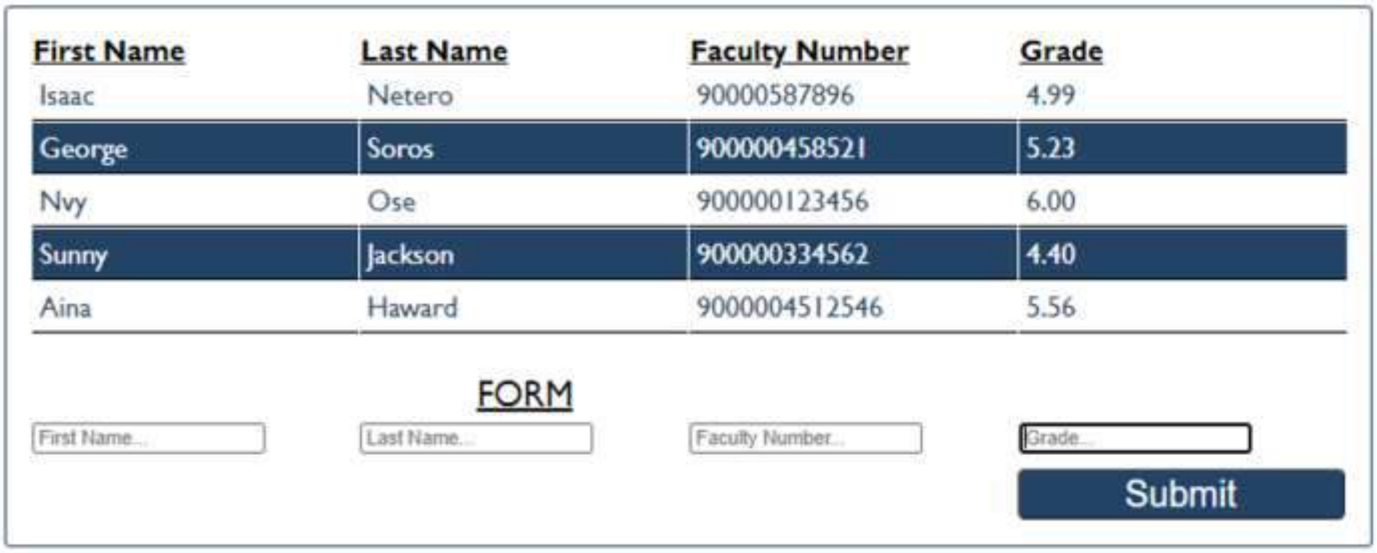
- `firstName` - `string`, non-empty
- `lastName` - `string`, non-empty
- `facultyNumber` - `string of numbers`, non-empty
- `grade` - `number`, non-empty

You need to write functionality for creating students. When creating a new student, make sure you name the properties accordingly.

You will also need to extract students. You will be given an **`HTML template`** with a table in it. Create an **`AJAX request`** that extracts all the students.

URL for this task: **http://localhost:3030/jsonstore/collections/students**

Screenshots



First Name	Last Name	Faculty Number	Grade
Isaac	Netero	90000587896	4.99
George	Soros	900000458521	5.23
Nvy	Ose	900000123456	6.00
Sunny	Jackson	900000334562	4.40
Aina	Haward	9000004512546	5.56

FORM

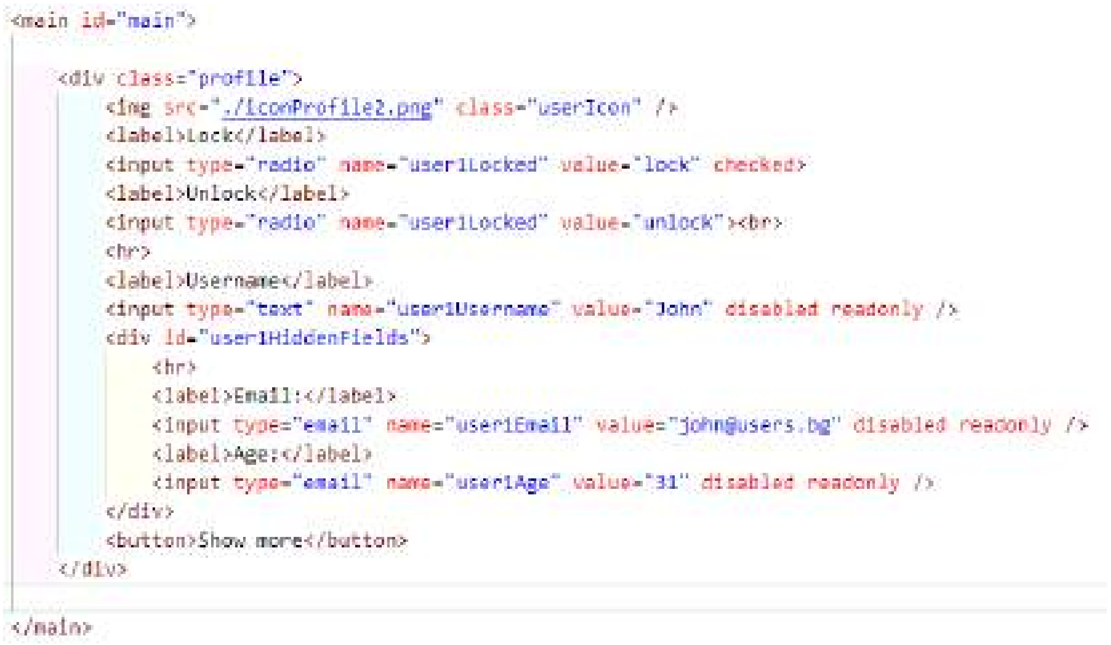
First Name...	Last Name...	Faculty Number...	Grade...
			Submit

5. Locked Profile

In this problem, you must **create a JS program** which **shows** and **hides** the additional information about users, which you can find by making a **GET** request to the server at address:

`http://localhost:3030/jsonstore/advanced/profiles`

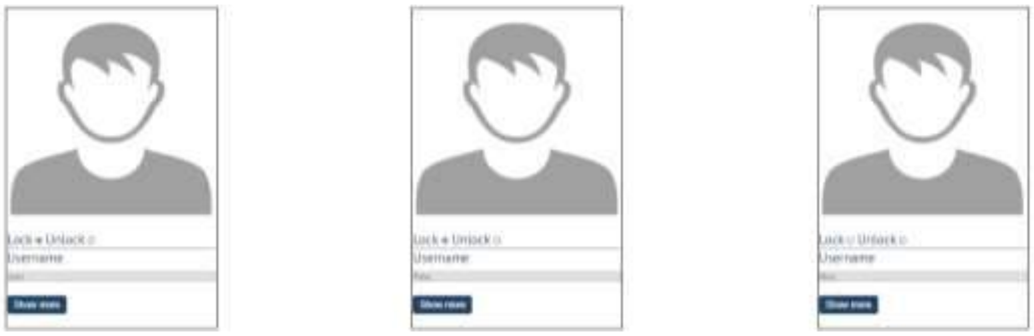
The response will be an object with the information for all users. Create a profile card for every user and display it on the web page. Every item should have the following structure:



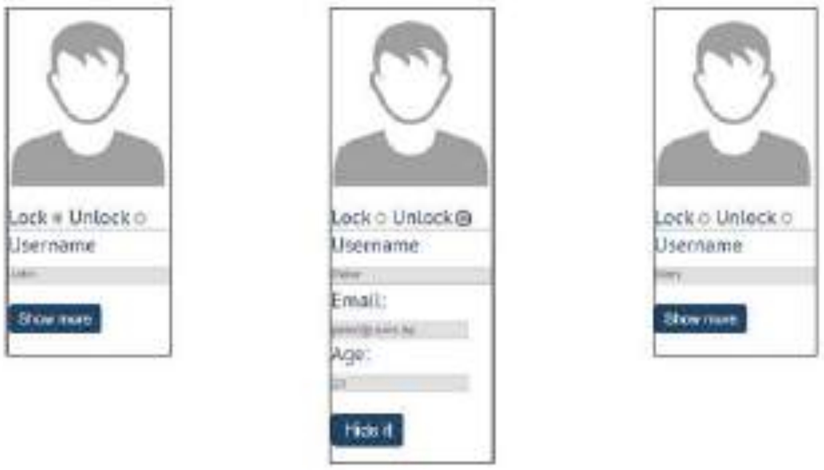
```html
<main id="main">

    <div class="profile">
        <img src="./iconProfile2.png" class="userIcon" />
        <label>Lock</label>
        <input type="radio" name="user1Locked" value="lock" checked>
        <label>Unlock</label>
        <input type="radio" name="user1Locked" value="unlock"><br>
        <hr>
        <label>Username</label>
        <input type="text" name="user1Username" value="John" disabled readonly />
        <div id="user1HiddenFields">
            <hr>
            <label>Email:</label>
            <input type="email" name="user1Email" value="john@users.bg" disabled readonly />
            <label>Age:</label>
            <input type="email" name="user1Age" value="31" disabled readonly />
        </div>
        <button>Show more</button>
    </div>

</main>
```



When one of the [**Show more**] **buttons** is clicked, the **hiden information** inside the div should be shown, only if **the profile is not locked**! If the current profile is **locked,** nothing should happen.



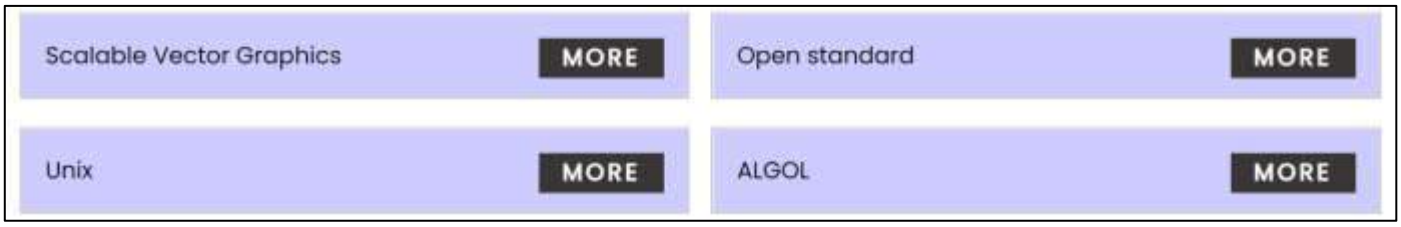
If the **hidden information is displayed** and we **lock the profile again**, the [**Hide it**] button should **not be working**! Otherwise, when the profile is **unlocked** and we click on the **[Hide it]** button, the new fields must hide again.

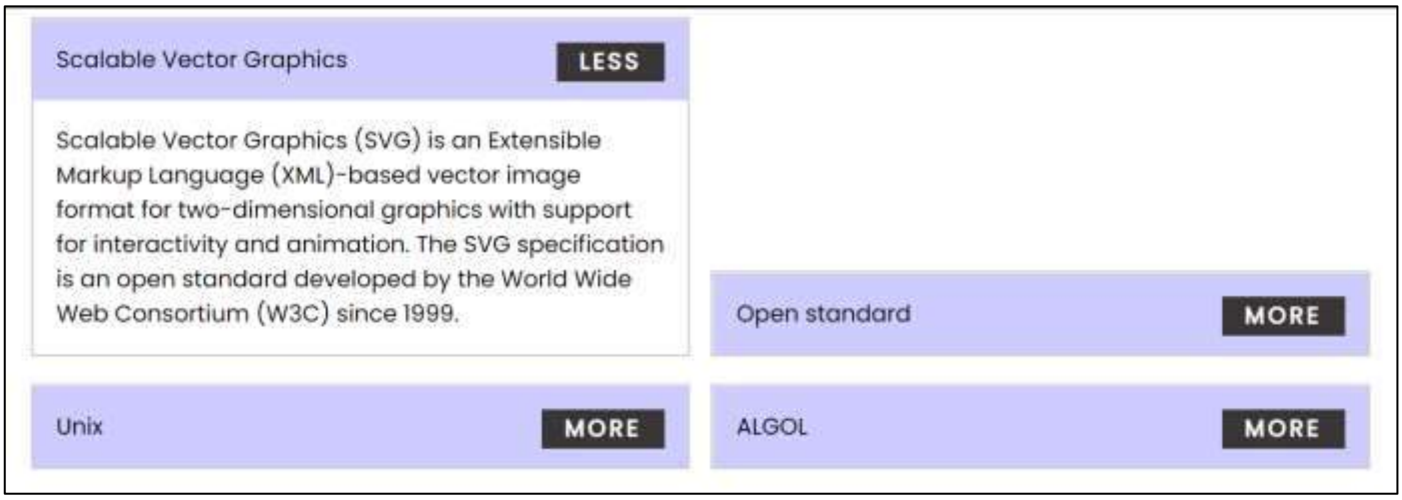
6. Accordion

An **html** file is given and your task is to show **more**/**less** information for the selected article. At the start you should do a **GET** request to the server at adress: **`http://localhost:3030/jsonstore/advanced/articles/list`** where the response will be an object with the titles of the articles.

By clicking the **[More] button** for the selected **article**, it should **reveal** the content of a **hidden** div and **changes** the text of the button to **[Less]**. Obtain the content by making a **GET** request to the server at adress: **`http://localhost:3030/jsonstore/advanced/articles/details/:id`** where the response will be an object with property **id**, **title**, **content.** When the same button is clicked **again** (now reading **Less**), **hide** the div and **change** the text of the button to **More**. Link action should be **toggleable** (you should be able to click the button infinite amount of times).

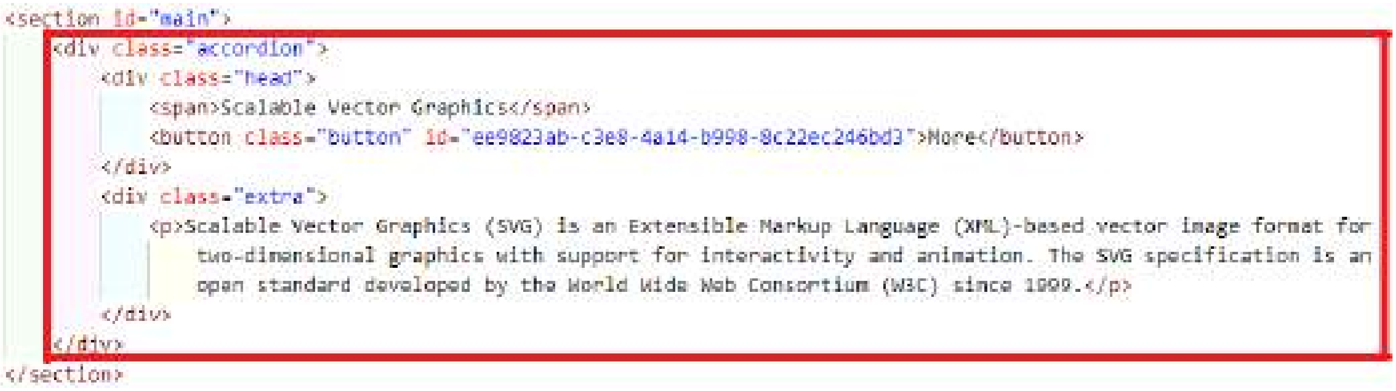
Example





Scalable Vector Graphics (SVG) is an Extensible Markup Language (XML)-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

Every item should have the **`following structure`**:

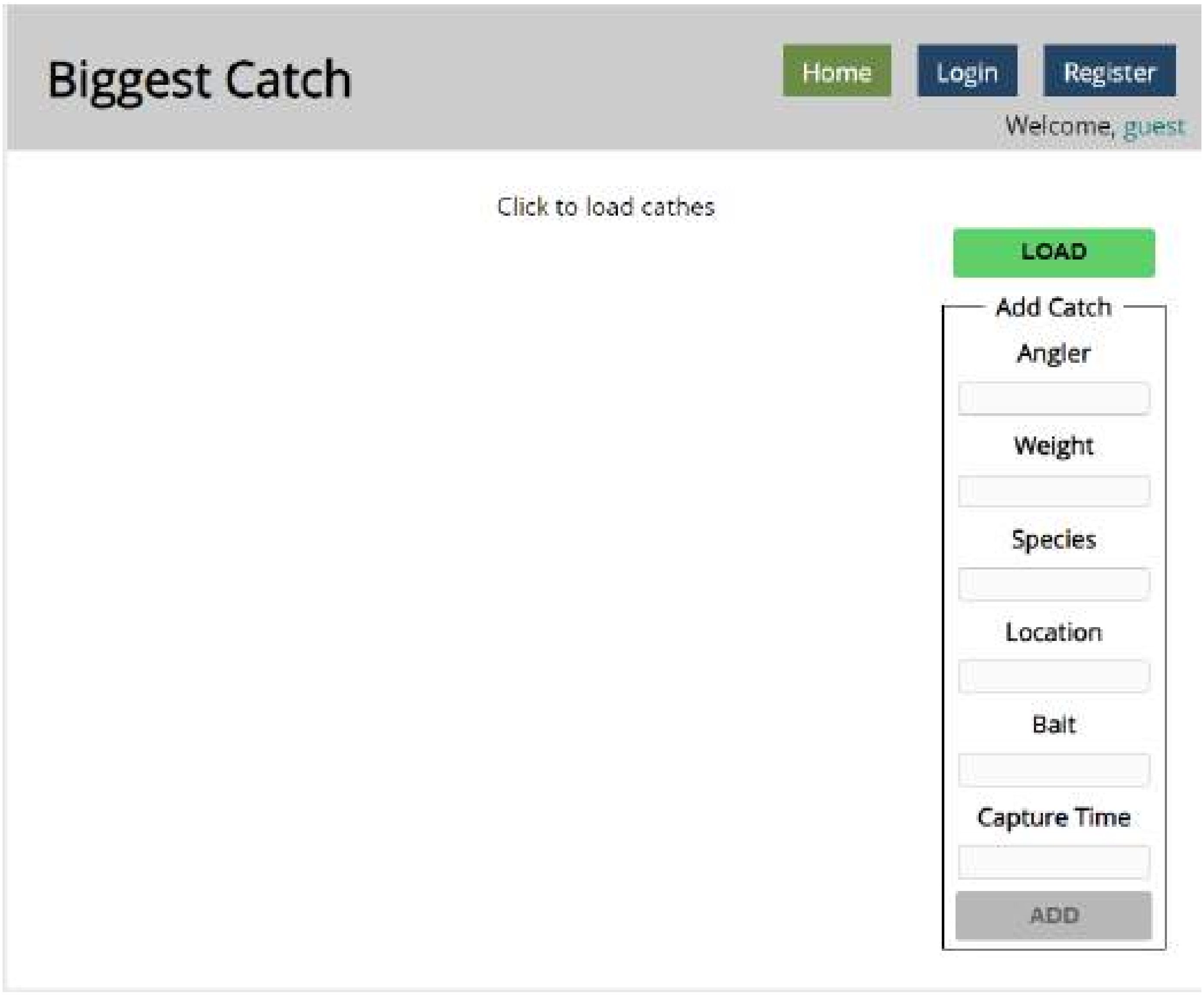


```
<section id="main">
    <div class="accordion">
        <div class="head">
            <span>Scalable Vector Graphics</span>
            <button class="button" id="ee9823ab-c3e8-4a14-b998-8c22ec246bd3">More</button>
        </div>
        <div class="extra">
            <p>Scalable Vector Graphics (SVG) is an Extensible Markup Language (XML)-based vector image format for
                two-dimensional graphics with support for interactivity and animation. The SVG specification is an
                open standard developed by the World Wide Web Consortium (W3C) since 1999.</p>
        </div>
    </div>
</section>
```

You are allowed to add new attributes, but do not change the existing ones.

7. Fisher Game

Use the provided skeleton and the server.

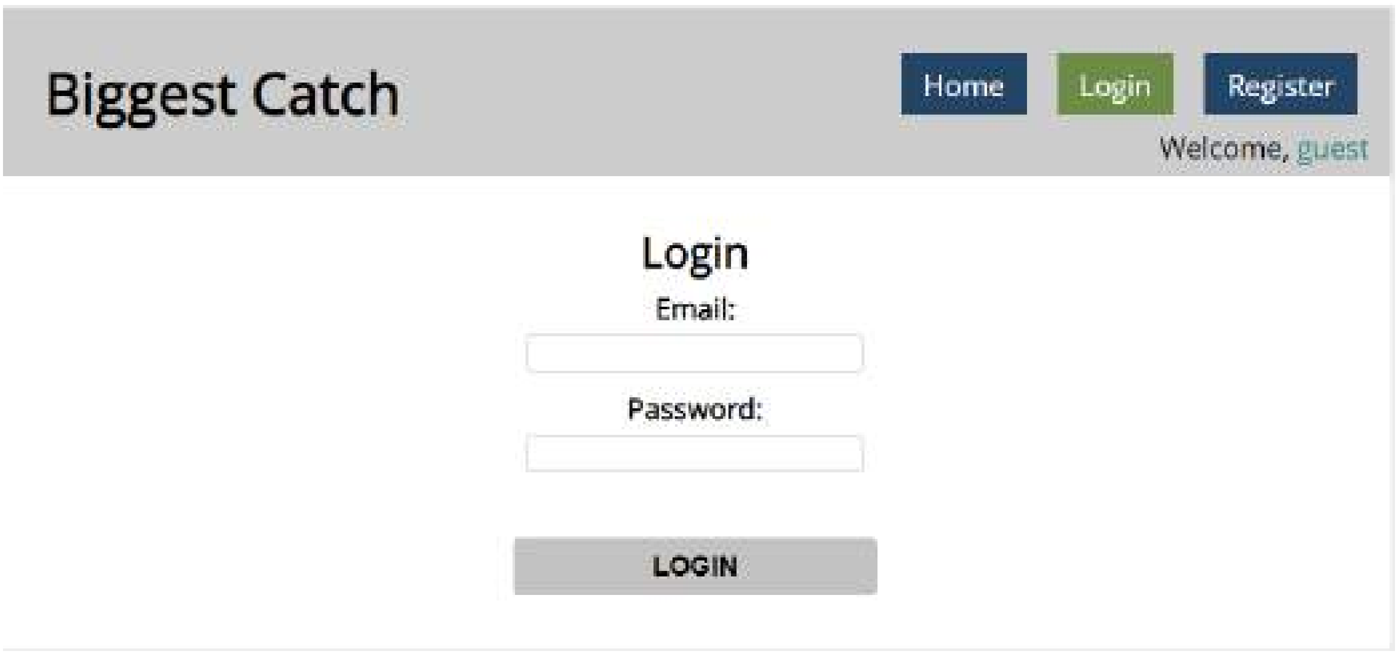


Login User

The **Login** page contains a form for existing user authentication. By given **email** and **password,** the app should login an existing user.

- After a **successful login** the **home page should be displayed**.
- In case of **error**, an appropriate error **message** should be displayed and the user should be able to fill in the login form again.
- Keep the user data in the browser's **session or locale storage**.
- POST request: **http://localhost:3030/users/login**
- Payload to test in postman:

  {
    "email": george@abv.bg,
    "password": "123456",
  }

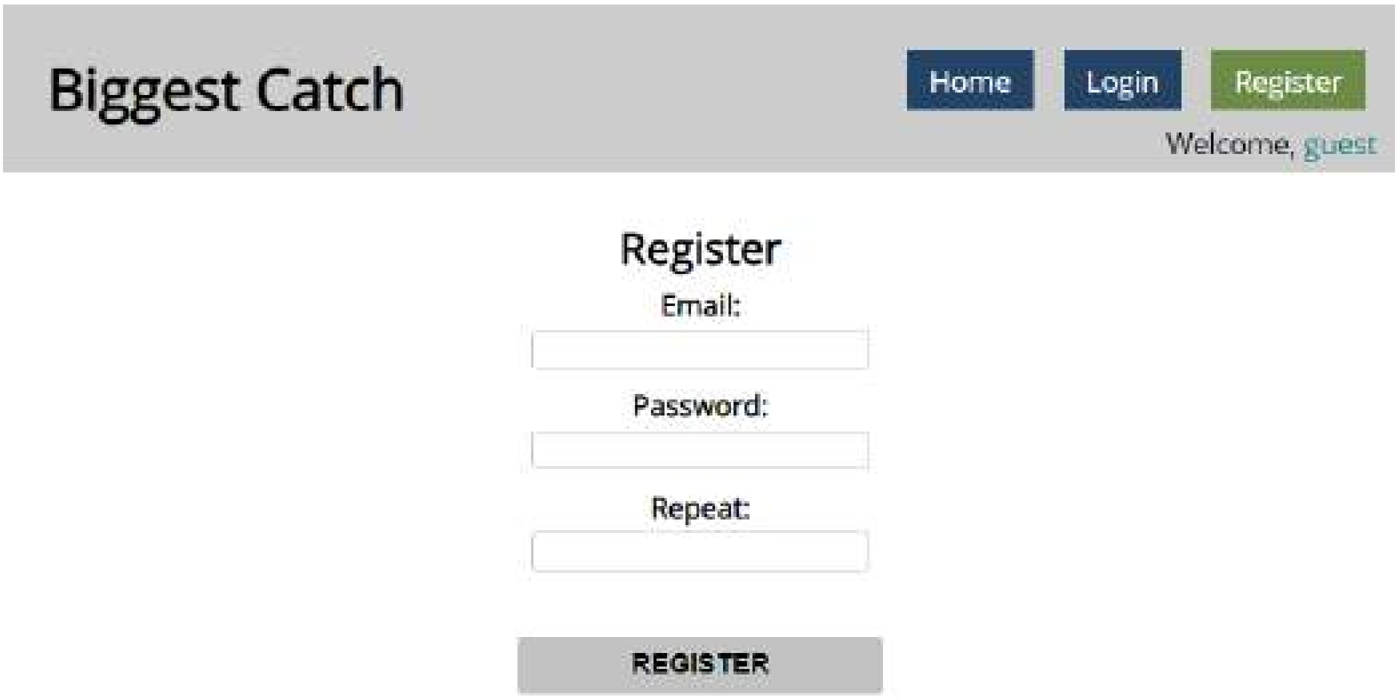


If the user is not logged in, all the buttons should be disabled except the "LOAD" button.

Register User

By given **email** and **password,** the app should register a new user in the system.

- In case of **error** (eg. invalid username/password), an appropriate error **message** should be displayed, and the user should be able to **try** to register again.
- Keep the user data in the browser's **session or local storage**.
- After a **successful registration** the **home page should be displayed**.
- POST request: **http://localhost:3030/users/register**



Logout

The logout action is available to **logged-in users**. Send the following **request** to perform logout:

- Get: **http://localhost:3030/users/logout**

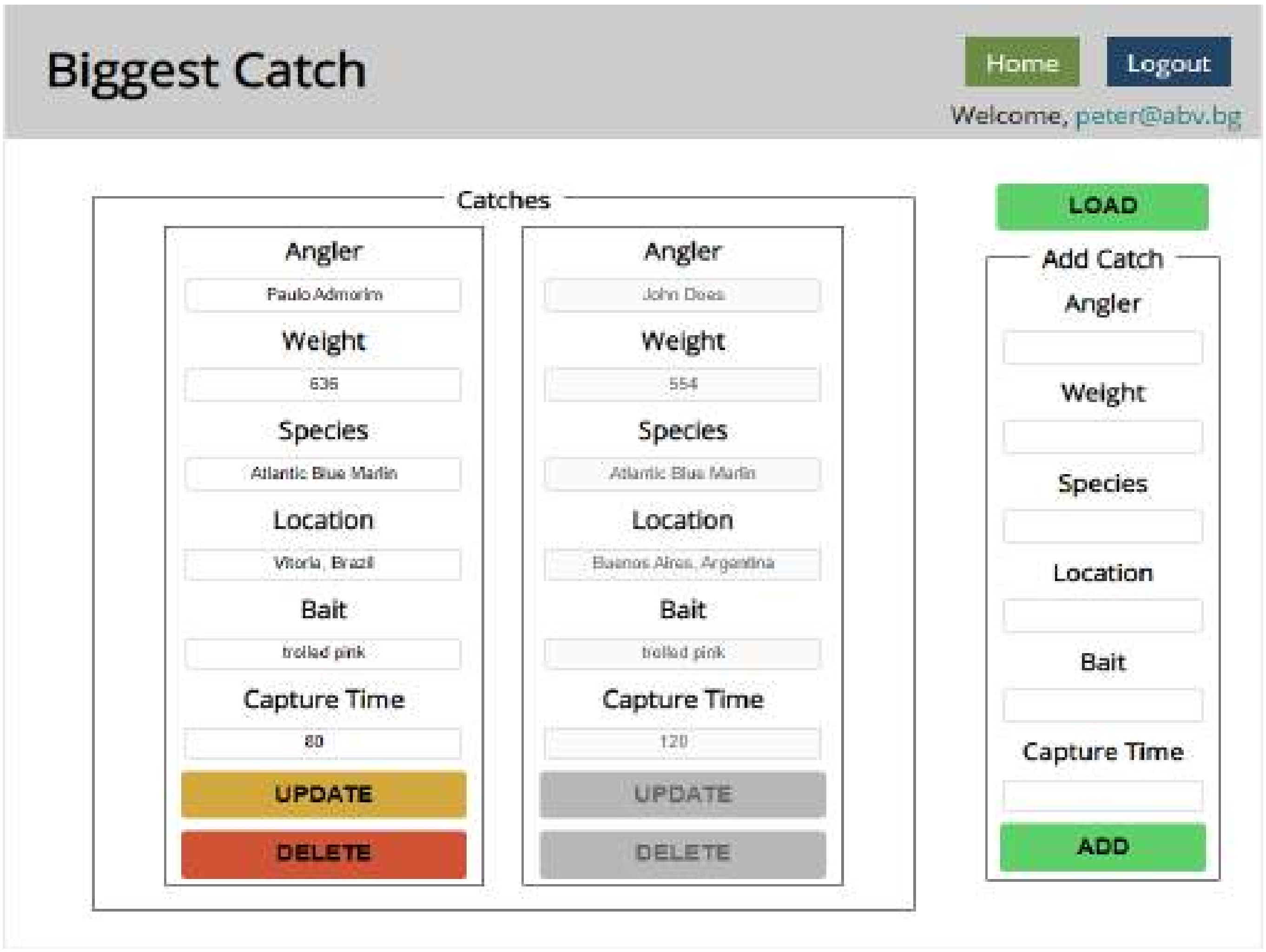
Required **headers** are described in the documentation. Upon success, the **REST service** will return an **empty response**. Clear any session information you've stored in browser storage.

If the logout was successful, **redirect** the user to the **Home** page and change the button in navigation.

Load catches

By clicking it you have to load all the catches from the server and render them like on the picture:

- Pressing the **[Load]** button should **list all** catches. (For all users)
- Pressing the **[Update]** button should send a **PUT** request, updating the catch in **http://localhost:3030/data/catches/:id**. (**Only for the creator of the catch**)
- Pressing the **[Delete]** button should delete the catch from **http://localhost:3030/data/catches/:id**. (**Only for the creator of the catch**)
- Pressing the **`[Add]`** button should submit a new catch with the values of the inputs in the fieldset with **`id="addFrom"`**. (**Only for logged in users**)
- Button **[Add]** should be **disabled** in there are no **logged in user**.
- Buttons **[Update]** and **[Delete]** should be **disabled** if the currently logged-in user is not the **author** of the catch.



Angler	Angler
Paulo Admorim	John Does
Weight	Weight
636	554
Species	Species
Atlantic Blue Marlin	Atlantic Blue Marlin
Location	Location
Vitoria, Brazil	Buenos Aires, Argentina
Bait	Bait
trolled pink	trolled pink
Capture Time	Capture Time
80	120

Each catch should have:

- `angler` - `string` representing the name of the person who caught the fish
- `weight` - `floating-point number` representing the weight of the fish in kilograms
- `species` - `string` representing the name of the fish species
- `location` - `string` representing the location where the fish was caught
- `bait` - `string` representing the bait used to catch the fish
- `captureTime` - `integer number` representing the time needed to catch the fish in minutes

Use the following requests to access your data:

- **List All Catches**
  - Endpoint - **http://localhost:3030/data/catches**
  - Method: **GET**
- **Create a New Catch**
  - Endpoint: **http://localhost:3030/data/catches**
  - Headers: X-Authorization: "…." (accessToken after login)
  - Method: **POST**
  - Request body (JSON): **`{"angler":"…", "weight":…, "species":"…", "location":"…", "bait":"…", "captureTime":…}`**
- **Update a Catch**
  - Endpoint: **http://localhost:3030/data/catches/:catchId**
  - Headers: X-Authorization: "…." (accessToken after login)
  - Method: **PUT**
  - Request body (JSON): **`{"angler":"…", "weight":…, "species":"…", "location":"…", "bait":"…", "captureTime":…}`**
- **Delete a Catch**
  - Endpoint: **http://localhost:3030/data/catches /:catchId**
  - Headers: X-Authorization: "…." (accessToken after login)
  - Method: **DELETE**

8. Furniture

Your task is to write the functionality of app, which shows list of furniture. By logged in user there is a possibility to buy furniture and list the bought products of the logged user. Also logged user can create new products (offers).

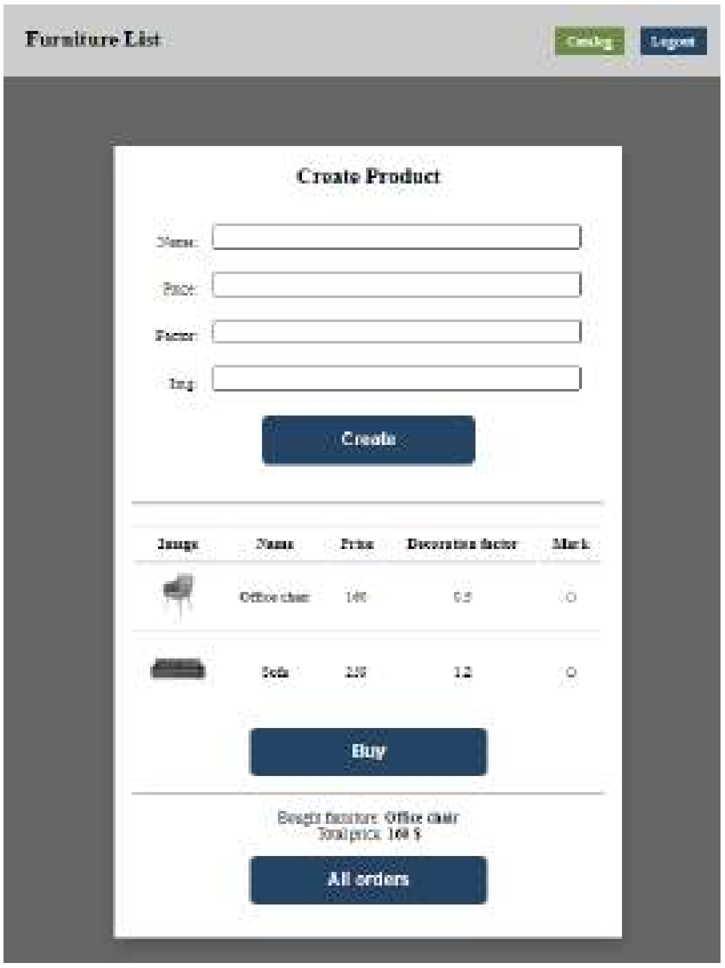


Image	Name	Price	Decoration factor	Mark
	Office chair	160	0.5	○
	Sofa	259	1.2	○

Bought furniture: Office chair
Total price: 160 $

Home page (not logged)

When the page is loaded the app should list all the furnitures in a table:

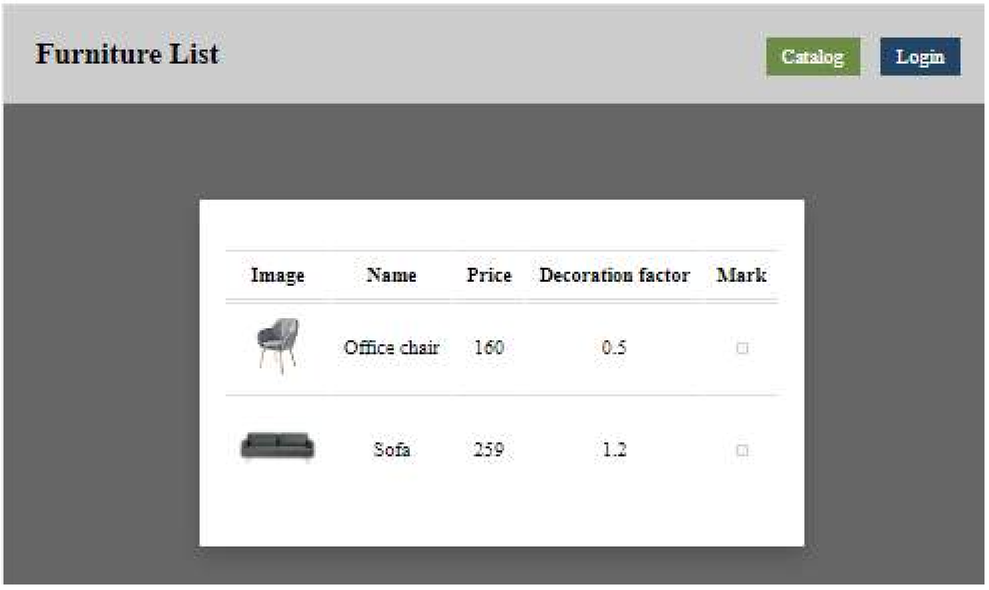


Image	Name	Price	Decoration factor	Mark
	Office chair	160	0.5	☐
	Sofa	259	1.2	☐

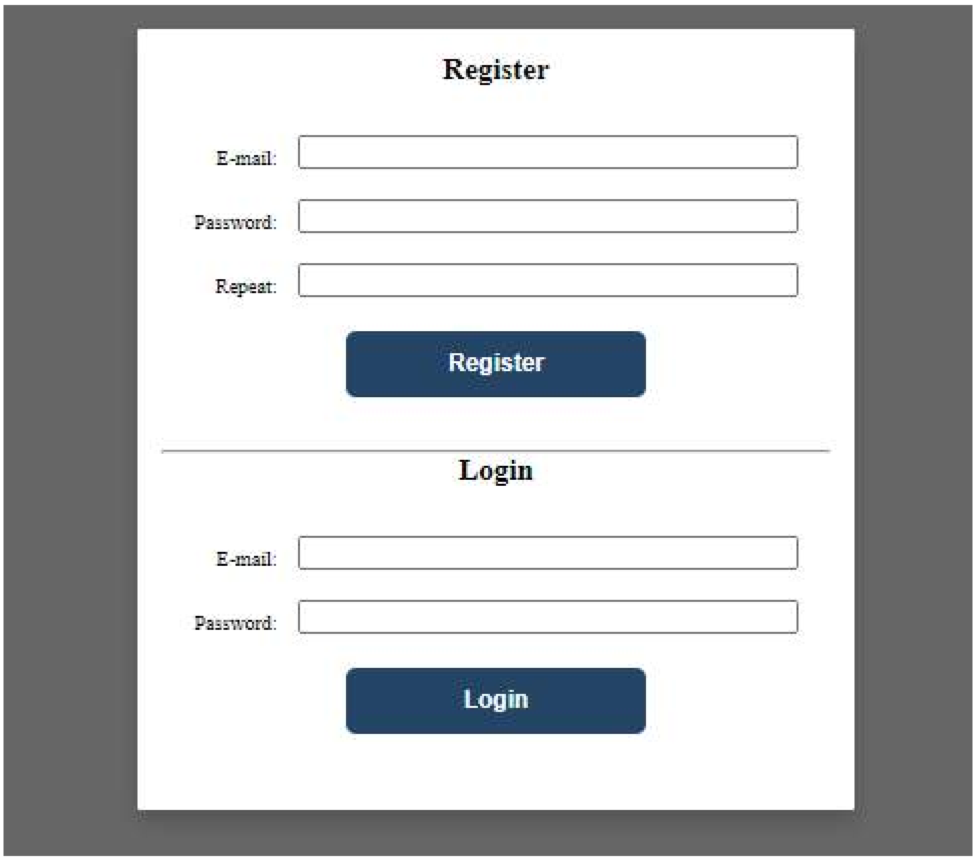
The checkbox should be disabled. You can send GET request on the URL: **http://localhost:3030/data/furniture**

Auth page

When "Login" is clicked, the app should redirect to "Login page". There are two possibilities:

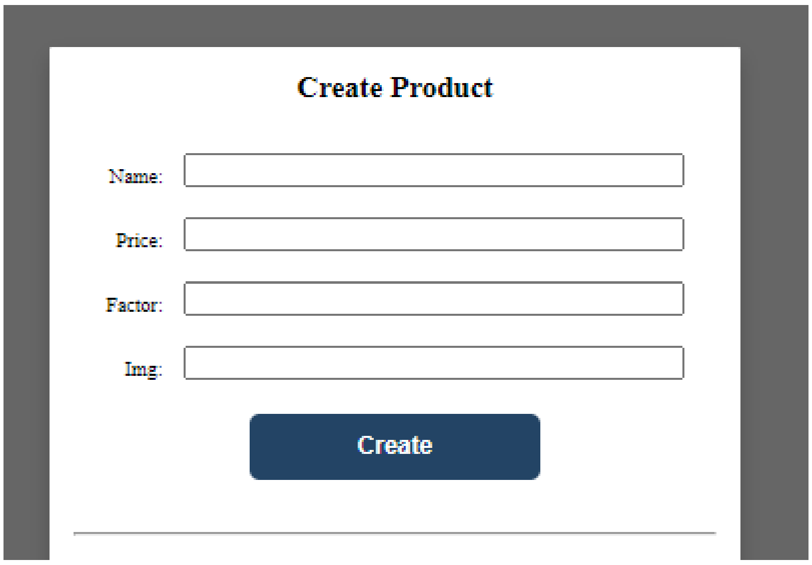
- to register a new user, send a POST request to the URL: **http://localhost:3030/users/register**

- to login, send a POST request to the URL: **http://localhost:3030/users/login**



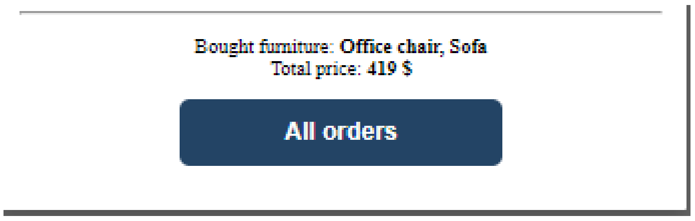
Home page (logged in)

When the **"`Create`" button is clicked**, add a **new row to the table** for each piece of furniture with **name, `price`, `factor` and `img`**. Send POST request to: **http://localhost:3030/data/furniture**



When the **"Buy"** button is clicked, get all **checkboxes that are marked** and save the information for these orders on the server. Make POST request to: **http://localhost:3030/data/orders**

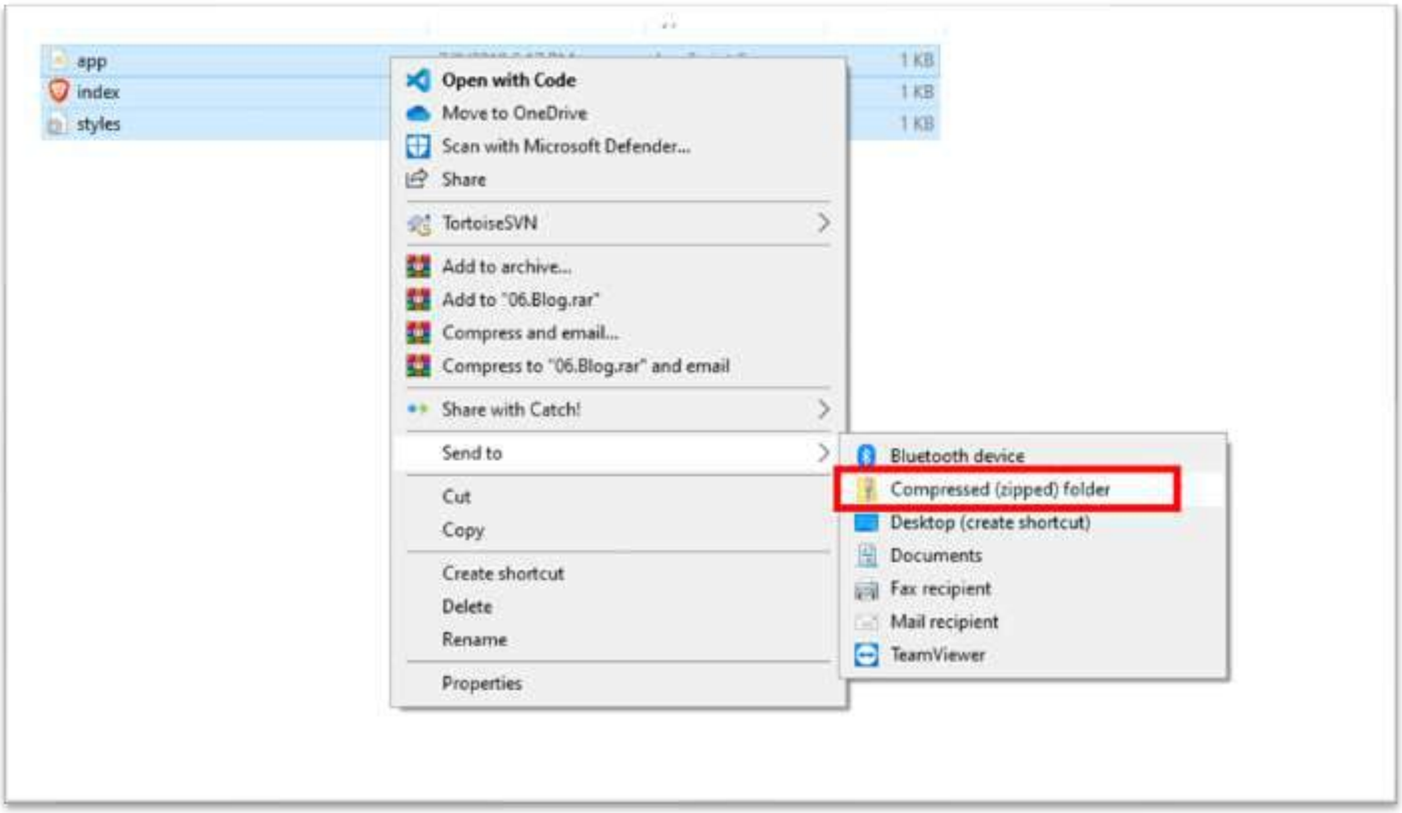
When the "**All orders**" button is clicked, get all bought furniture of the current user, and show their names and the total price, as shown on the picture:

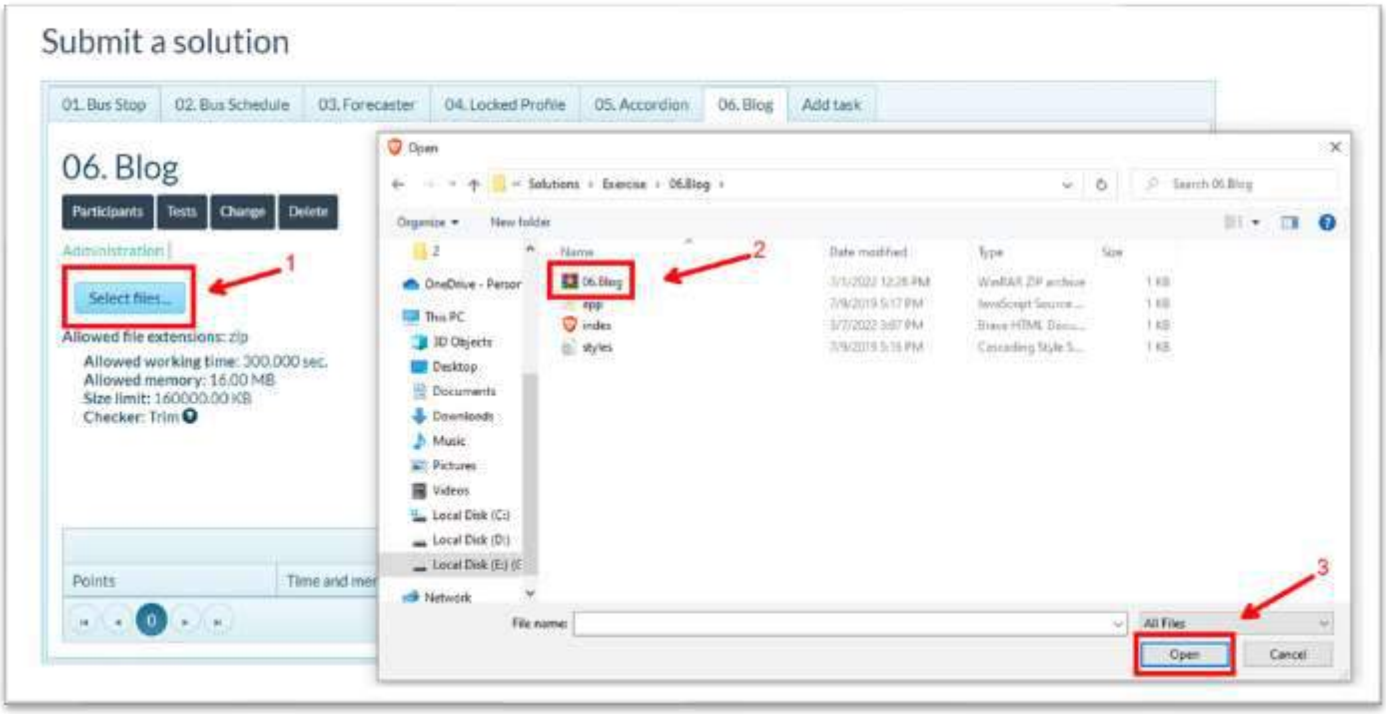

Bought furniture: Office chair, Sofa
Total price: 419 $

This could happen with GET request on this URL: http://localhost:3030/data/orders?where=_ownerId%3D{userId}

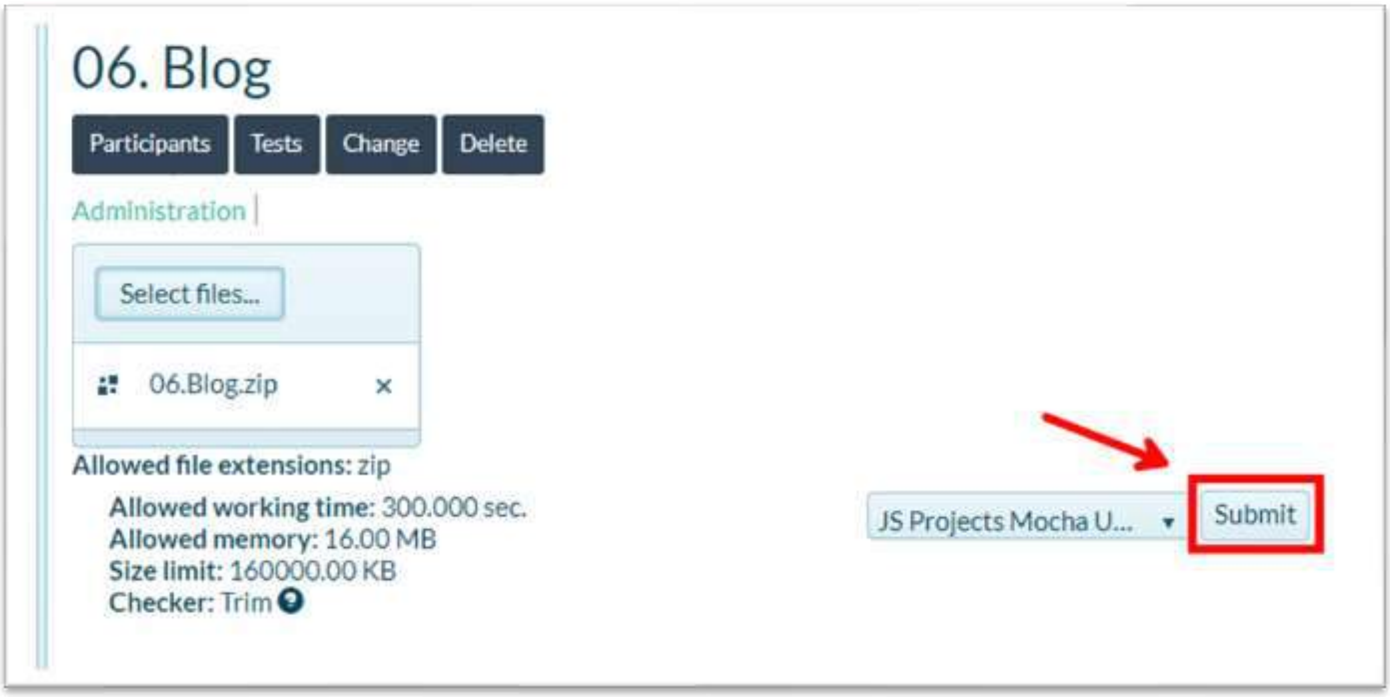
Submitting Your Solution

Place in a **ZIP** file the content of the given resources including your solution. Exclude the **`node_modules`** & **`tests`** folders. Upload the archive to Judge.





Allowed file extensions: zip
Allowed working time: 300.000 sec.
Allowed memory: 16.00 MB
Size limit: 160000.00 KB
Checker: Trim



Allowed file extensions: zip
Allowed working time: 300.000 sec.
Allowed memory: 16.00 MB
Size limit: 160000.00 KB
Checker: Trim