Exercise: HTTP and REST

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.

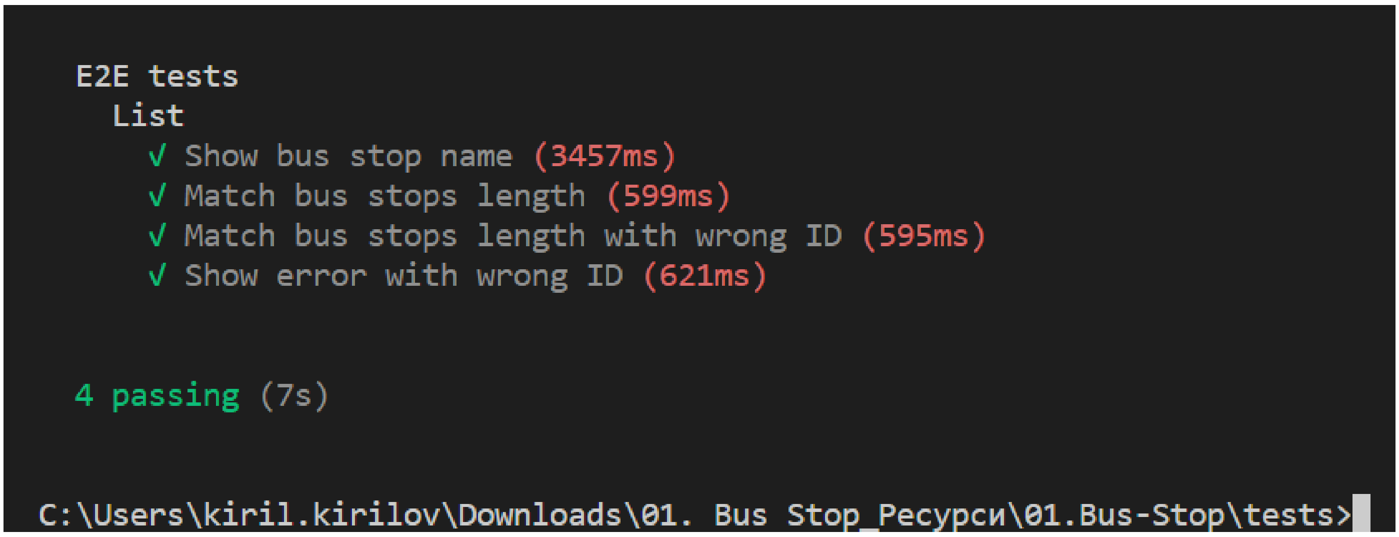
Requirements

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. Bus Stop

Write a JS program that displays arrival times for all buses by a given bus stop ID when a button is clicked. Use the skeleton from the provided resources.

When the button with ID **'submit'** is clicked, the name of the bus stop appears and the list bellow gets filled with all the buses that are expected and their time of arrival. Take the **value** of the input field with id **'stopId'**. Submit a **GET** request to **http://localhost:3030/jsonstore/bus/businfo/:busId** (replace the highlighted part with the correct value) and parse the response. You will receive a JSON object in the format:

```
stopId: {
  name: stopName,
  buses: { busId: time, … }
}
```

Place the name property as text inside the div with ID **'stopName'** and each bus as a list item with text:

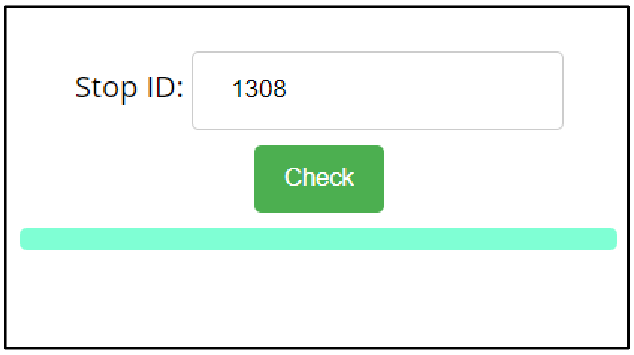
```
"Bus {busId} arrives in {time} minutes"
```

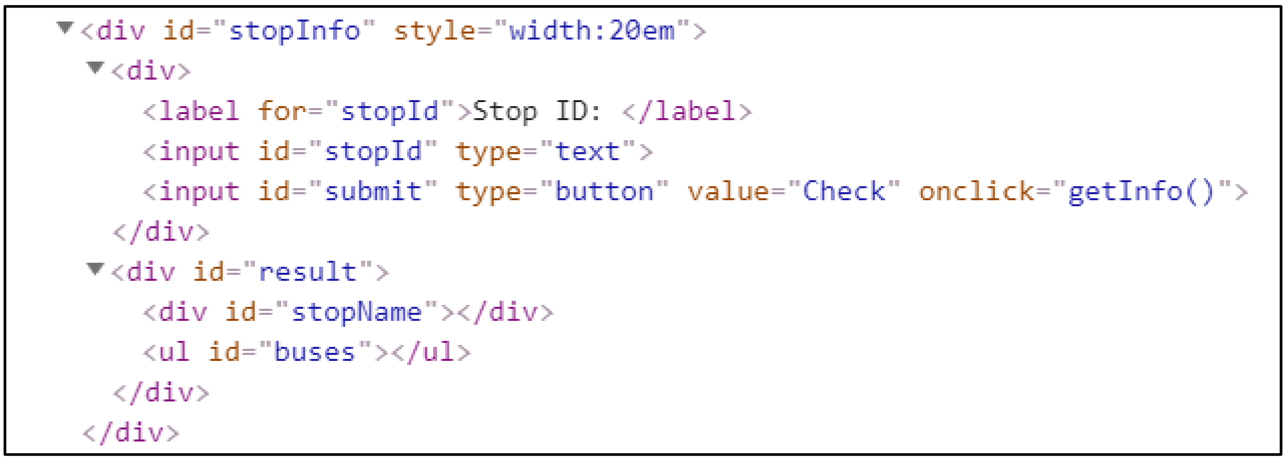
Replace all highlighted parts with the relevant value from the response. If the request is not successful, or the information is not in the expected format, display **"Error"** as **`stopName`** and nothing in the list. The list should be cleared before every request is sent.

Note: The service will respond with valid data to IDs 1287, 1308, 1327 and 2334.

See examples on the next page.

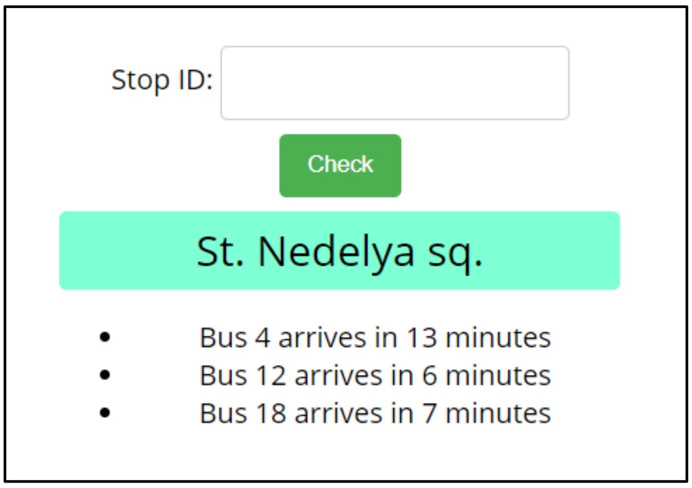
Examples

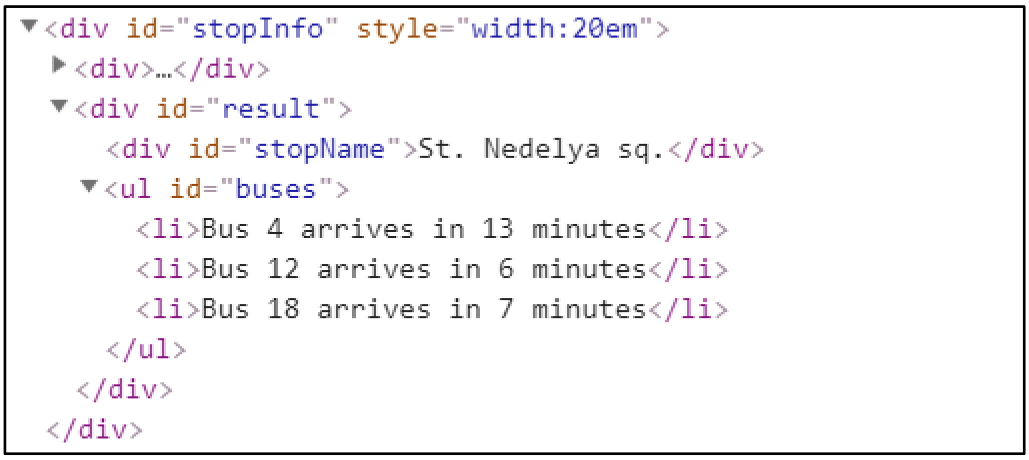




```
▼<div id="stopInfo" style="width:20em">
  ▼<div>
      <label for="stopId">Stop ID: </label>
      <input id="stopId" type="text">
      <input id="submit" type="button" value="Check" onclick="getInfo()">
    </div>
  ▼<div id="result">
      <div id="stopName"></div>
      <ul id="buses"></ul>
    </div>
  </div>
```

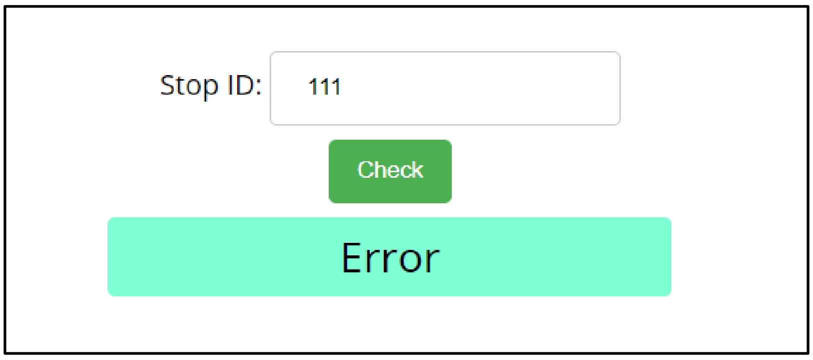
When the button is clicked, the results are displayed in the corresponding elements:

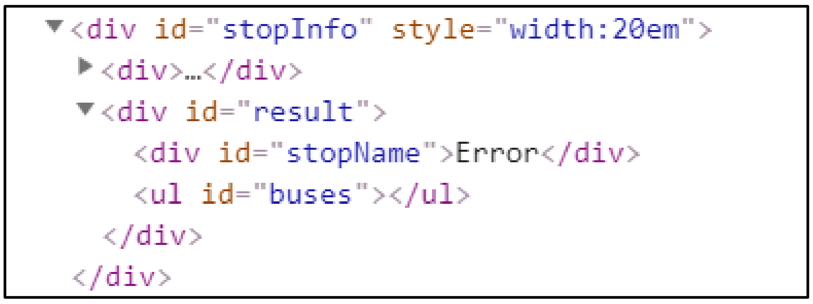

Stop ID:
Check
St. Nedelya sq.
Bus 4 arrives in 13 minutes
Bus 12 arrives in 6 minutes
Bus 18 arrives in 7 minutes



```
▼<div id="stopInfo" style="width:20em">
  ▶<div>…</div>
  ▼<div id="result">
      <div id="stopName">St. Nedelya sq.</div>
    ▼<ul id="buses">
        <li>Bus 4 arrives in 13 minutes</li>
        <li>Bus 12 arrives in 6 minutes</li>
        <li>Bus 18 arrives in 7 minutes</li>
      </ul>
    </div>
  </div>
```

If an error occurs, the stop name changes to Error:





```
▼<div id="stopInfo" style="width:20em">
  ▶<div>…</div>
  ▼<div id="result">
      <div id="stopName">Error</div>
      <ul id="buses"></ul>
    </div>
  </div>
```

2. Bus Schedule

Write a JS program that tracks the progress of a bus on it’s route and announces it inside an info box. The program should display which is the upcoming stop and once the bus arrives, to request from the server the name of the next one. Use the skeleton from the provided resources.

The bus has two states – **moving** and **stopped**. When it is **stopped**, only the button “**Depart**” is **enabled**, while the info box shows the name of the **current** stop. When it is **moving**, only the button “**Arrive**” is **enabled**, while the info box shows the name of the **upcoming** stop. Initially, the info box shows "**Not Connected**" and the "**Arrive**" button is **disabled**. The ID of the first stop is "**depot**".

When the "**Depart**" button is clicked, make a **GET** request to the server with the ID of the current stop to address **`http://localhost:3030/jsonstore/bus/schedule/:id`** (replace the highlighted part with the relevant value). As a response, you will receive a JSON object in the following format:

```
stopId {
  name: stopName,
  next: nextStopId
}
```

Update the info box with the information from the response, disable the “Depart” button and enable the “Arrive” button. The info box text should look like this (replace the highlighted part with the relevant value):

```
Next stop {stopName}
```

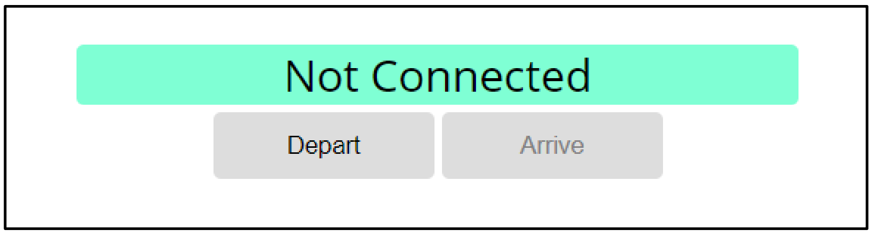
When the "**Arrive**" button is clicked, update the text, disable the “Arrive” button and enable the “Depart” button. The info box text should look like this (replace the highlighted part with the relevant value):

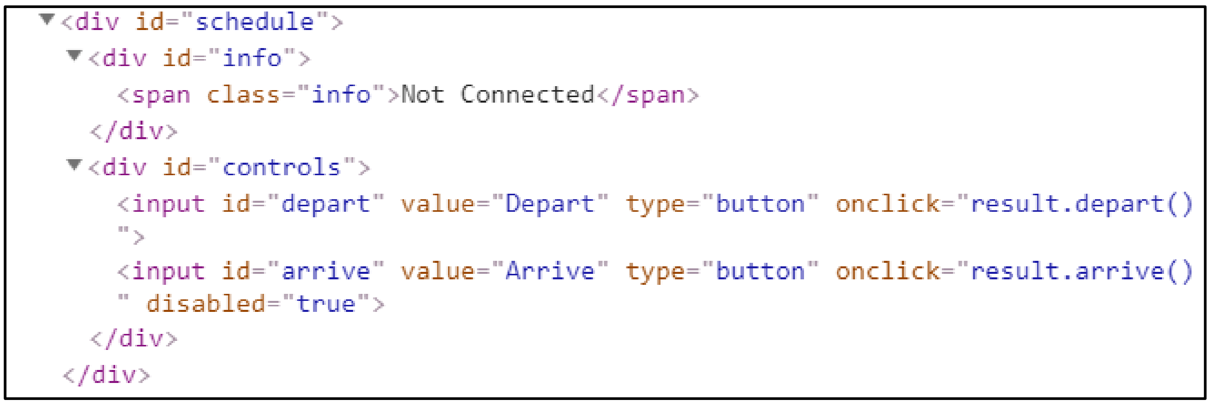
```
Arriving at {stopName}
```

Clicking the buttons successfully will cycle through the entire schedule. If invalid data is received, show "**Error**" inside the info box and **disable** both buttons.

Examples

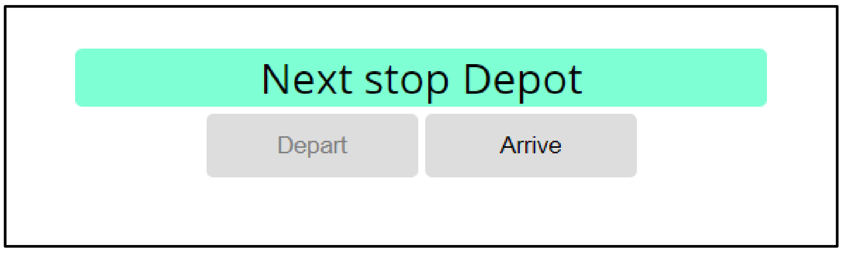
Initially, the info box shows “Not Connected” and the arrive button is disabled.

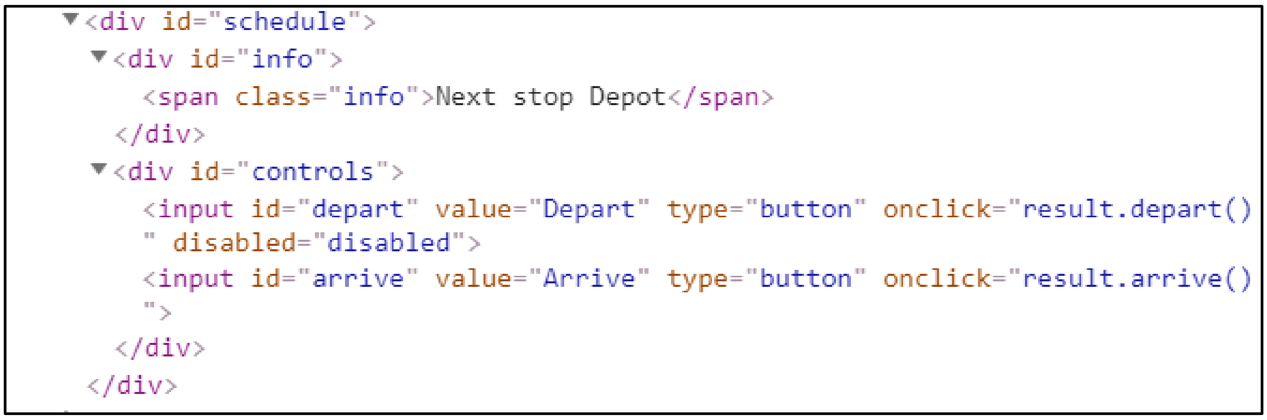




```
▼<div id="schedule">
  ▼<div id="info">
      <span class="info">Not Connected</span>
    </div>
  ▼<div id="controls">
      <input id="depart" value="Depart" type="button" onclick="result.depart()
      ">
      <input id="arrive" value="Arrive" type="button" onclick="result.arrive()
      " disabled="true">
    </div>
  </div>
```

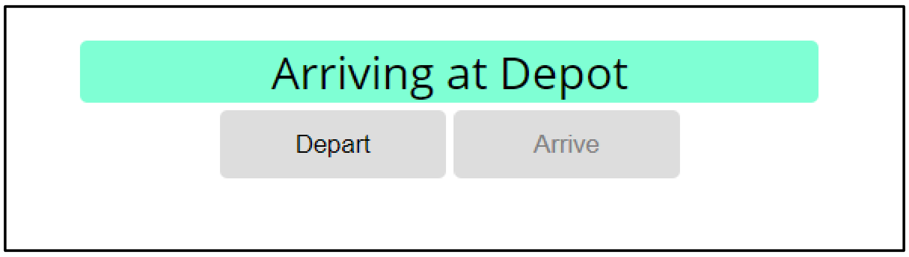
When Depart is clicked, a request is made with the first ID. The info box is updated with the new information and the buttons are changed:

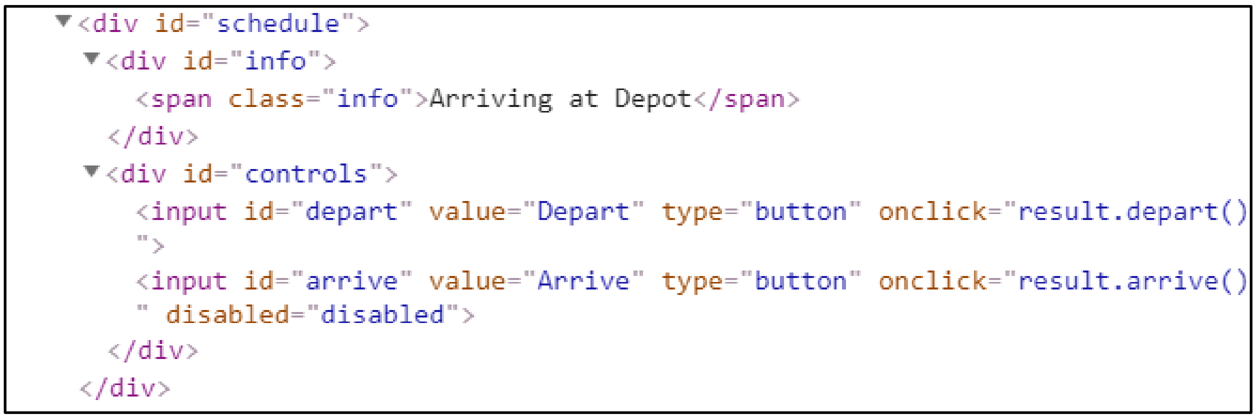




```
▼<div id="schedule">
  ▼<div id="info">
      <span class="info">Next stop Depot</span>
    </div>
  ▼<div id="controls">
      <input id="depart" value="Depart" type="button" onclick="result.depart()
      " disabled="disabled">
      <input id="arrive" value="Arrive" type="button" onclick="result.arrive()
      ">
    </div>
  </div>
```

Clicking Arrive, changes the info box and swaps the buttons. This allows Depart to be clicked again, which makes a new request and updates the information:





```
▼<div id="schedule">
  ▼<div id="info">
     <span class="info">Arriving at Depot</span>
   </div>
  ▼<div id="controls">
     <input id="depart" value="Depart" type="button" onclick="result.depart()
     ">
     <input id="arrive" value="Arrive" type="button" onclick="result.arrive()
     " disabled="disabled">
   </div>
 </div>
```

3. Forecaster

Write a program that **requests** a weather report **from a server** and **displays** it to the user.

Use the skeleton from the provided resources.

When the user writes the name of a location and clicks "**`Get Weather`**", make a **GET** request to the server at address **`http://localhost:3030/jsonstore/forecaster/locations`**. The response will be an array of objects, with the following structure:

```
{
  name: locationName,
  code: locationCode
}
```

Find the object, corresponding to the name that the user submitted in the input field with ID "**`location`**" and use its **code** value to make **two more GET requests**:

- For current conditions, make a request to:

`http://localhost:3030/jsonstore/forecaster/today/:code`

The response from the server will be an object with the following structure:

```
{
  name: locationName,
  forecast: { low: temp,
              high: temp,
              condition: condition }
}
```

- For a 3-day forecast, make a request to:

`http://localhost:3030/jsonstore/forecaster/upcoming/:code`

The response from the server will be an object with the following structure:

```
{
  name: locationName,
  forecast: [{ low: temp,
               high: temp,
               condition: condition }, … ]
}
```

Use the information from these two objects to compose a forecast in HTML and insert it inside the page. Note that the **`<div>`** with ID "**`forecast`**" must be set to **`visible`**. See the examples for details.

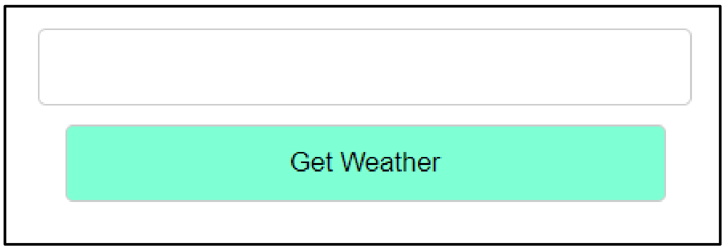
If an **error** occurs (the server doesn't respond or the location name cannot be found) or the data is not in the correct format, display "**`Error`**" in the **forecast section**.

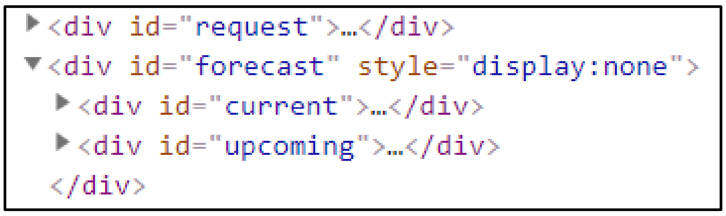
Use the following codes for weather symbols:

- Sunny **&#x2600;** // ☀
- Partly sunny **&#x26C5;** // ⛅
- Overcast **&#x2601;** // ☁
- Rain **&#x2614;** // ☔
- Degrees **&#176;** // °

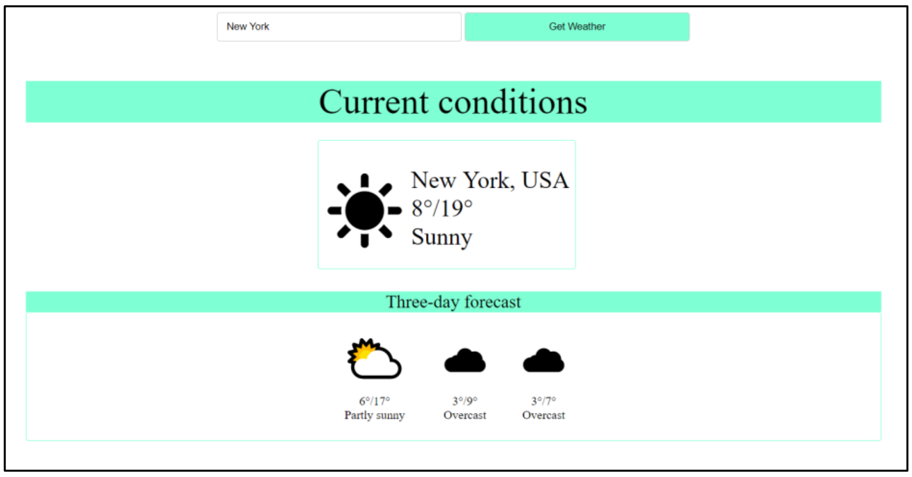
Examples

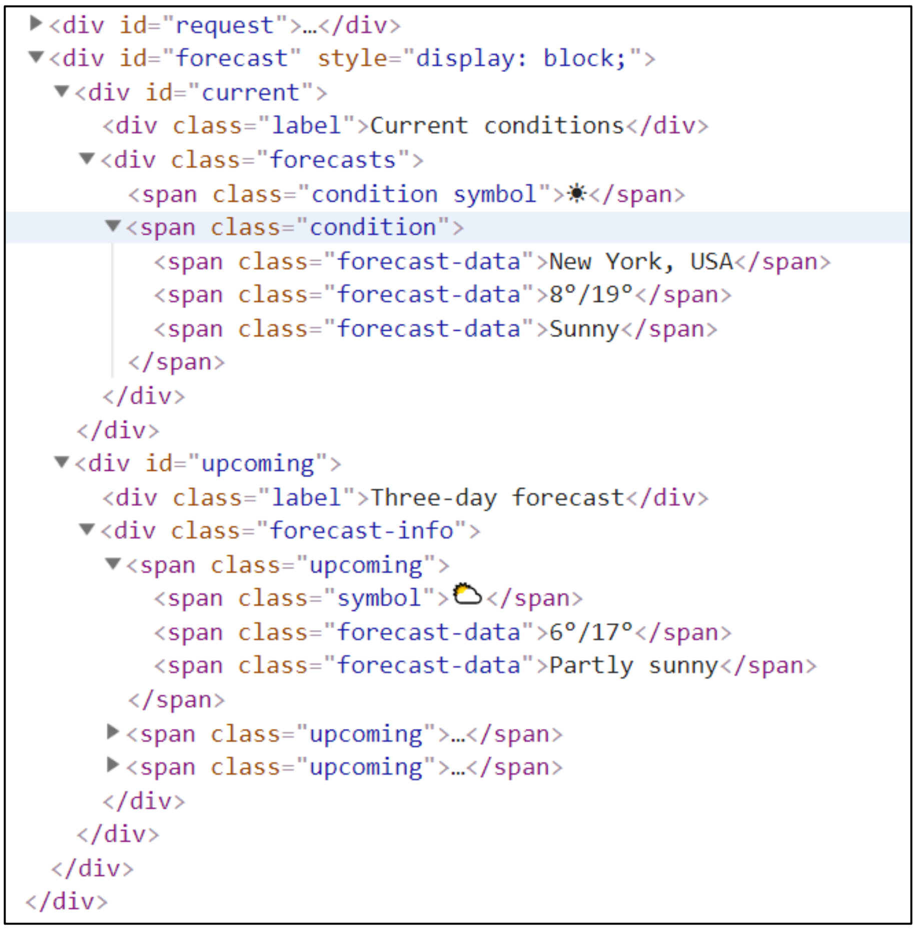
When the app starts, the **forecast `div`** is **`hidden`**. When the user **enters a name** and **clicks** on the button **`Get Weather`**, the requests being.





```
▸<div id="request">…</div>
▾<div id="forecast" style="display:none">
  ▸<div id="current">…</div>
  ▸<div id="upcoming">…</div>
 </div>
```





```
▶<div id="request">…</div>
▼<div id="forecast" style="display: block;">
  ▼<div id="current">
      <div class="label">Current conditions</div>
    ▼<div class="forecasts">
        <span class="condition symbol">☀</span>
      ▼<span class="condition">
          <span class="forecast-data">New York, USA</span>
          <span class="forecast-data">8°/19°</span>
          <span class="forecast-data">Sunny</span>
        </span>
      </div>
    </div>
  ▼<div id="upcoming">
      <div class="label">Three-day forecast</div>
    ▼<div class="forecast-info">
      ▼<span class="upcoming">
          <span class="symbol">⛅</span>
          <span class="forecast-data">6°/17°</span>
          <span class="forecast-data">Partly sunny</span>
        </span>
      ▶<span class="upcoming">…</span>
      ▶<span class="upcoming">…</span>
      </div>
    </div>
  </div>
</div>
```

4. Blog

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

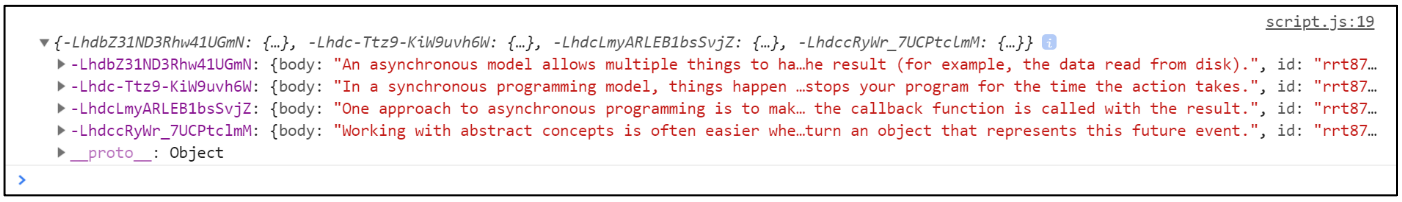
Request URL's:

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.**

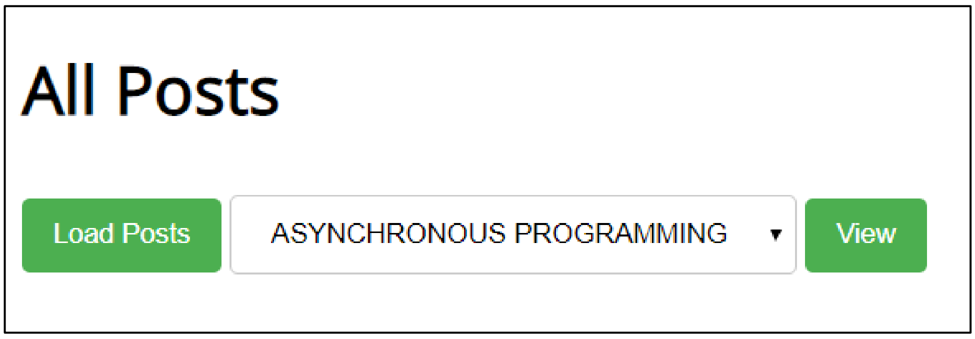


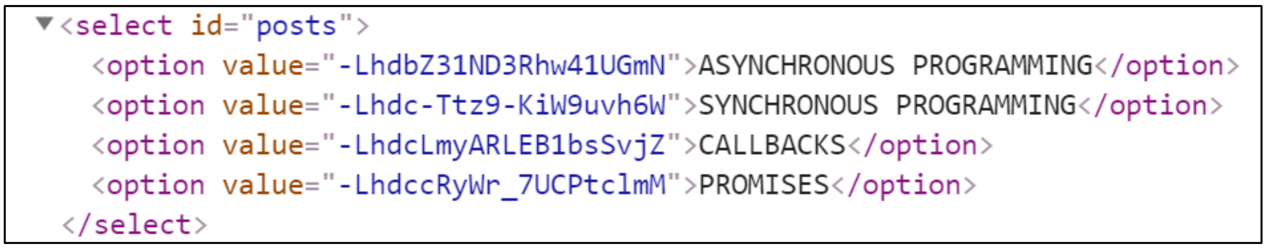
```
                                                                                                        script.js:19
▼{-LhdbZ31ND3Rhw41UGmN: {…}, -Lhdc-Ttz9-KiW9uvh6W: {…}, -LhdcLmyARLEB1bsSvjZ: {…}, -LhdccRyWr_7UCPtclmM: {…}}
  ▶-LhdbZ31ND3Rhw41UGmN: {body: "An asynchronous model allows multiple 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`**".

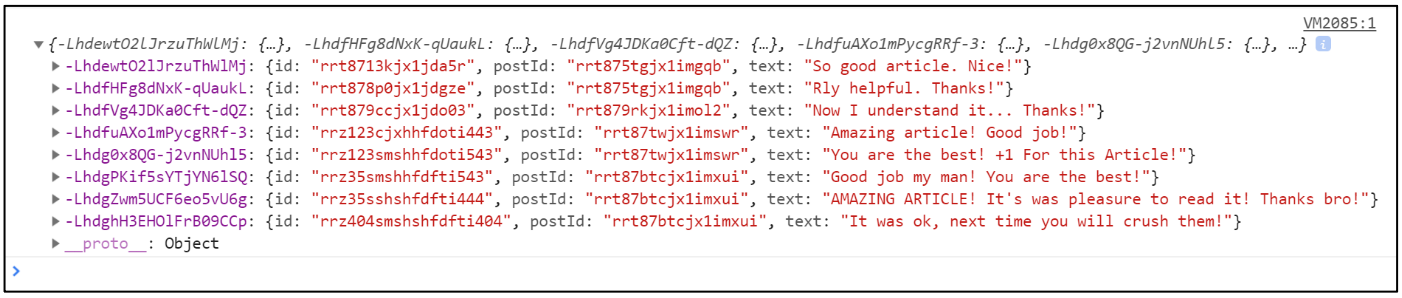




```
▼<select id="posts">
    <option value="-LhdbZ31ND3Rhw41UGmN">ASYNCHRONOUS PROGRAMMING</option>
    <option value="-Lhdc-Ttz9-KiW9uvh6W">SYNCHRONOUS PROGRAMMING</option>
    <option value="-LhdcLmyARLEB1bsSvjZ">CALLBACKS</option>
    <option value="-LhdccRyWr_7UCPtclmM">PROMISES</option>
  </select>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

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**.



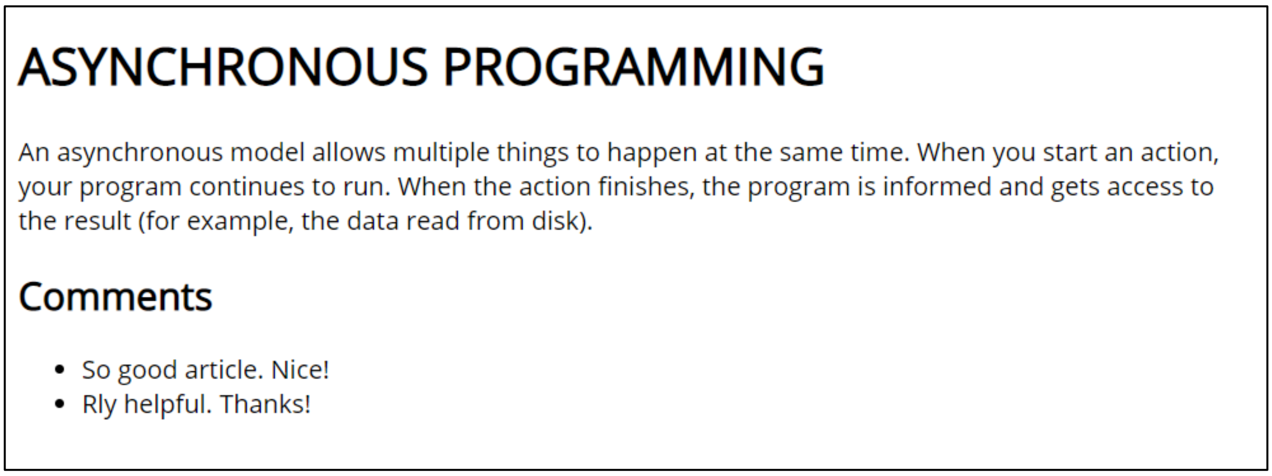
```
                                                                                                                   VM2085:1
▼{-LhdewtO2lJrzuThWlMj: {…}, -LhdfHFg8dNxK-qUaukL: {…}, -LhdfVg4JDKa0Cft-dQZ: {…}, -LhdfuAXo1mPycgRRf-3: {…}, -Lhdg0x8QG-j2vnNUhl5: {…}, …}
  ▶-LhdewtO2lJrzuThWlMj: {id: "rrt8713kjx1jda5r", 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!"}
  ▶-LhdfuAXo1mPycgRRf-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!"}
  ▶-LhdgPKif5sYTjYN6lSQ: {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!"}
  ▶-LhdghH3EHOlFrB09CCp: {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.

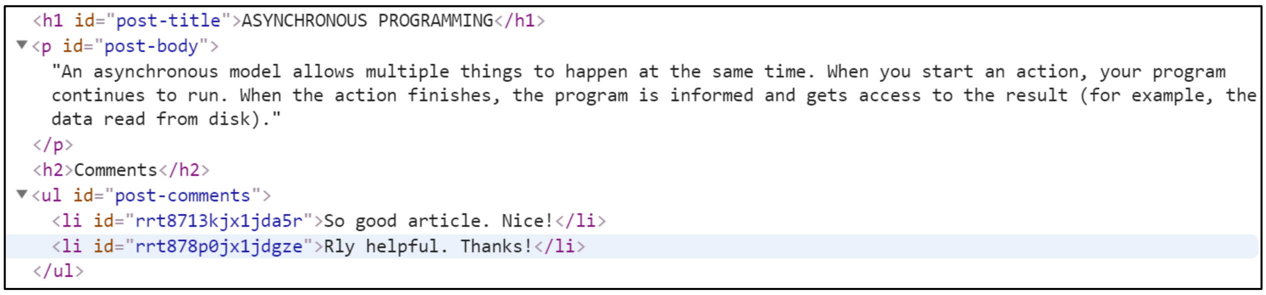


ASYNCHRONOUS PROGRAMMING

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>
```

5. 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.

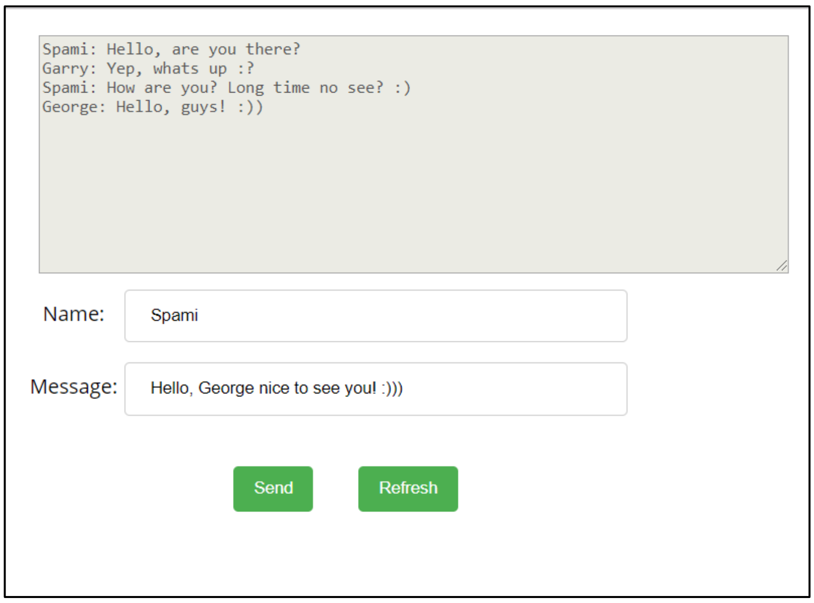
The url for the requests - **http://localhost:3030/jsonstore/messenger**

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:

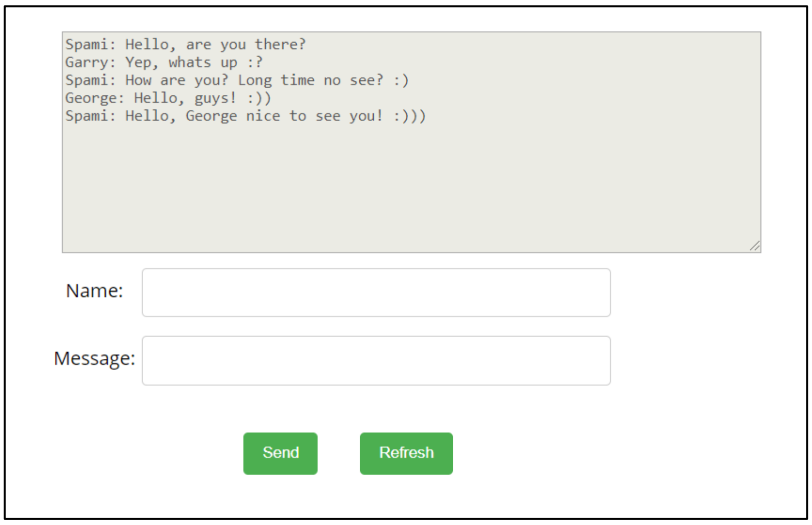
```
{
  author: authorName,
  content: msgText,
}
```

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! :))
```



```
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! :)))
```

6. 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).

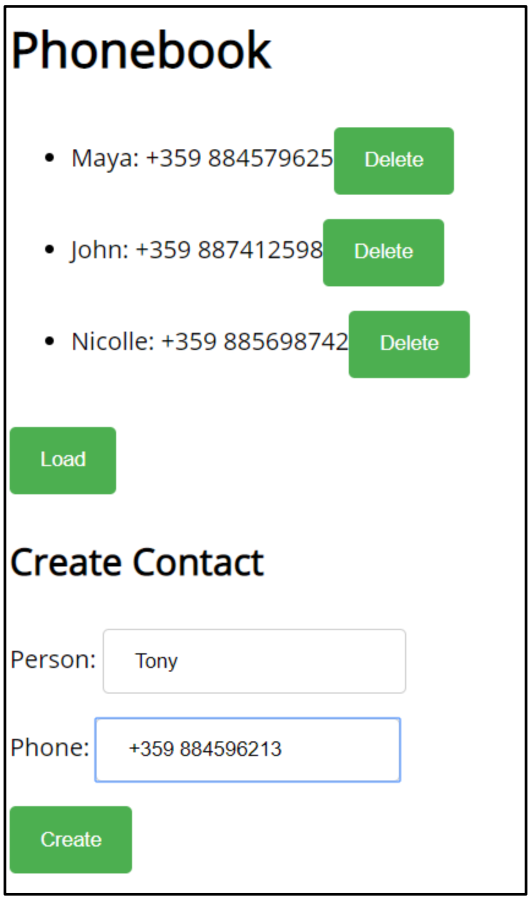
The data sent on a POST **request should be a valid JSON object, containing properties** person **and** phone. **Example format:**

```
{
  "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



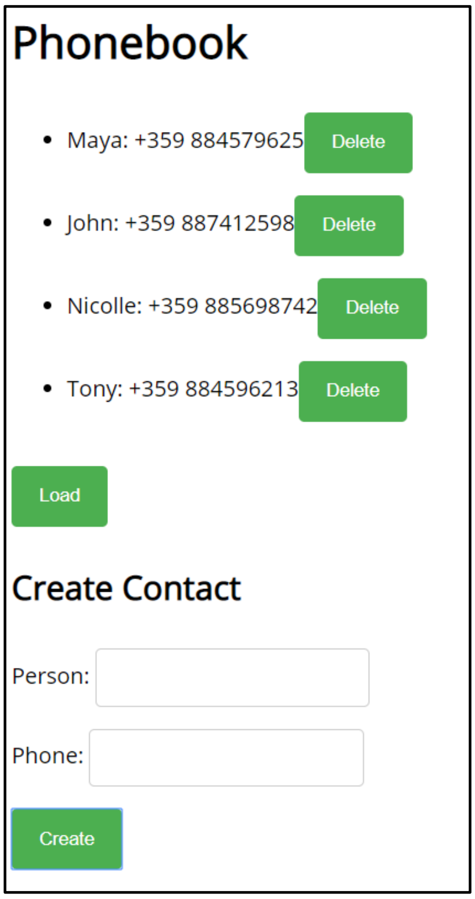
Phonebook

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

Create Contact

Person: Tony

Phone: +359 884596213



Phonebook

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

Create Contact

Person:

Phone:

7. Students

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

Each student has:

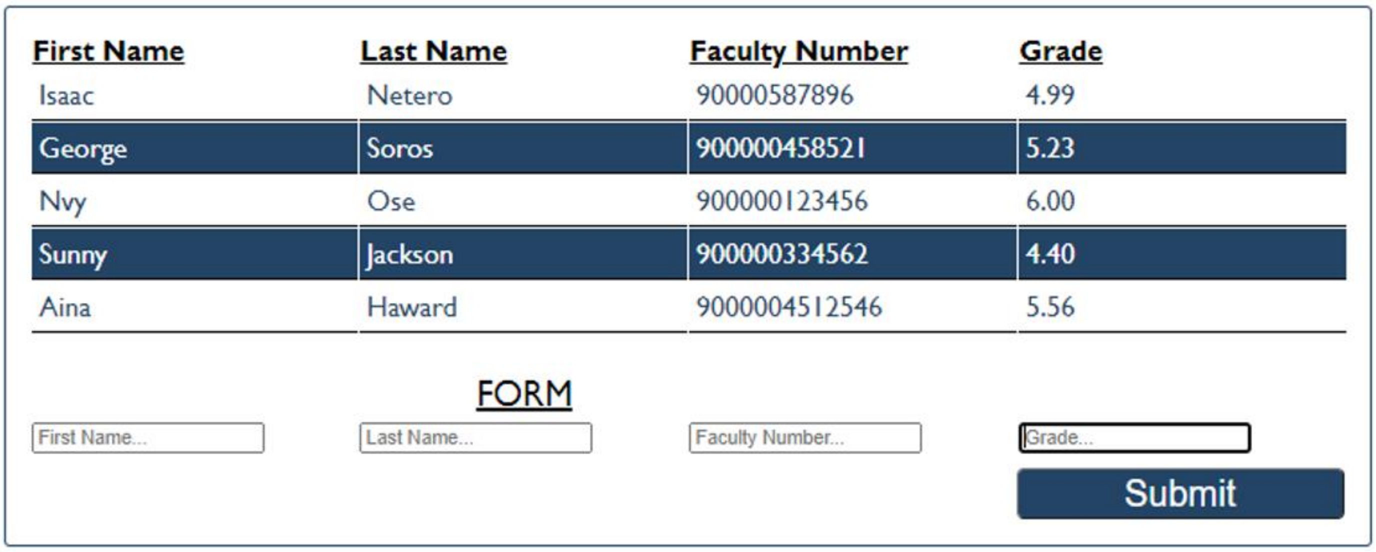
- **firstName** - **string**, non-empty
- **lastName** - **string**, non-empty
- **facultyNumber** - **string of numbers**, non-empty
- **grade** - **number**, non-empty

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



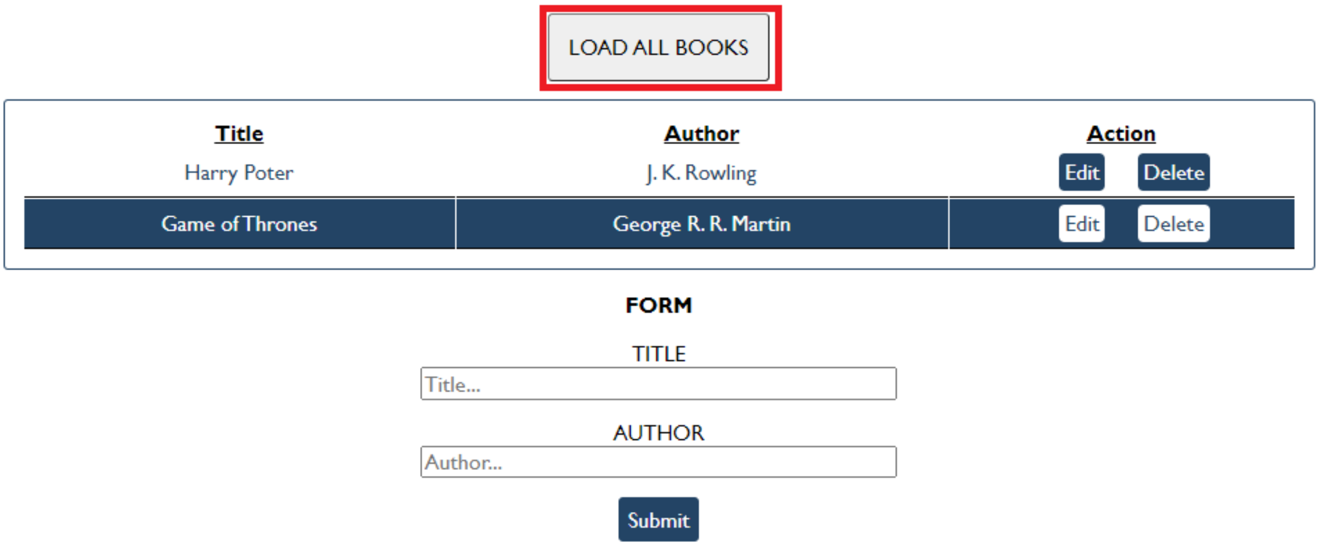
First Name	Last Name	Faculty Number	Grade
Isaac	Netero	90000587896	4.99
George	Soros	900000458521	5.23
Nvy	Ose	900000123456	6.00
Sunny	Jackson	900000334562	4.40
Aina	Haward	9000004512546	5.56

8. Book Library

First task is to "**GET**" all books. To consume the request with **POSTMAN** your **url** should be the **following**: **http://localhost:3030/jsonstore/collections/books**

Using the provided skeleton, write the missing functionalities.

Load all books by clicking the button "LOAD ALL BOOKS"



Title	Author	Action
Harry Poter	J. K. Rowling	Edit Delete
Game of Thrones	George R. R. Martin	Edit Delete

Get Book

This functionality is not needed in this task, but you can try it with postman by sending request to "GET" the Book with id:" d953e5fb-a585-4d6b-92d3-ee90697398a0". Send a GET request to this URL:

http://localhost:3030/jsonstore/collections/books/:id

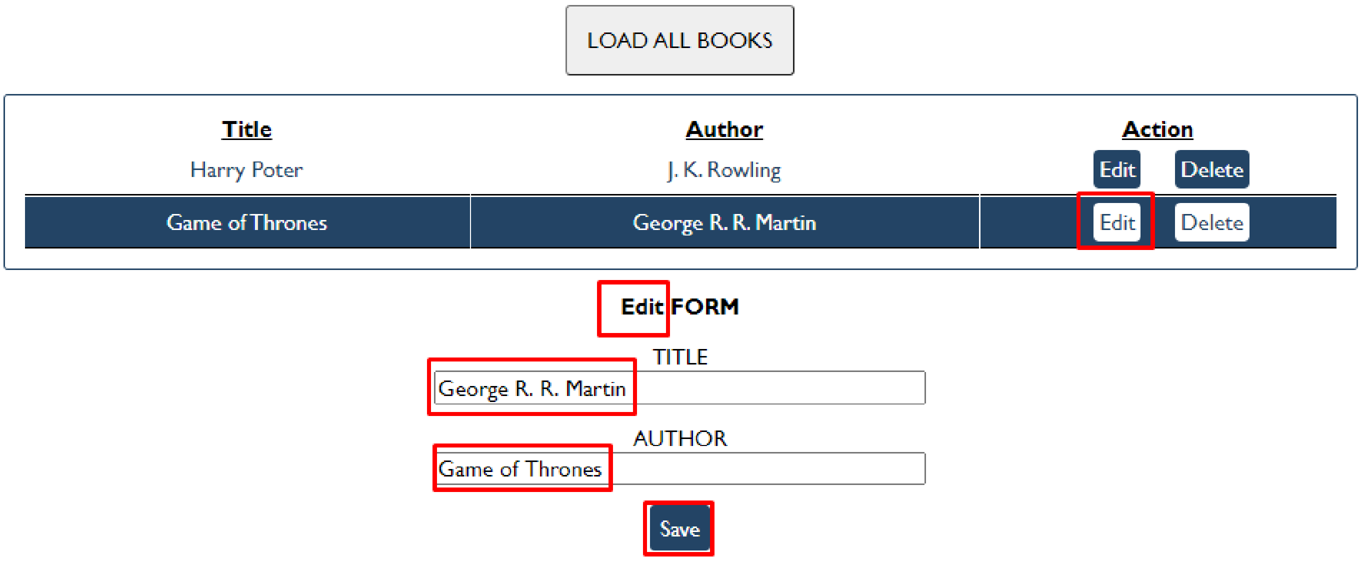
Create Book

Write functionality to create a new book, when the submit button is clicked. Before sending the request be sure the fields are not empty (make validation of the input). To **create** a book, you have to send a "**POST**" request and the JSON body should be in the **following** format:

```
{
  "author": "New Author",
  "title": "New Title"
}
```

Update Book

By clicking the edit button of a book, change the form like this:



Title	Author	Action
Harry Poter	J. K. Rowling	Edit Delete
Game of Thrones	George R. R. Martin	Edit Delete

Edit FORM

TITLE: George R. R. Martin

AUTHOR: Game of Thrones

The HTTP command "**PUT**" **modifies** an existing HTTP **resource**. The URL is:

http://localhost:3030/jsonstore/collections/books/:id

The JSON body should be in the **following** format:

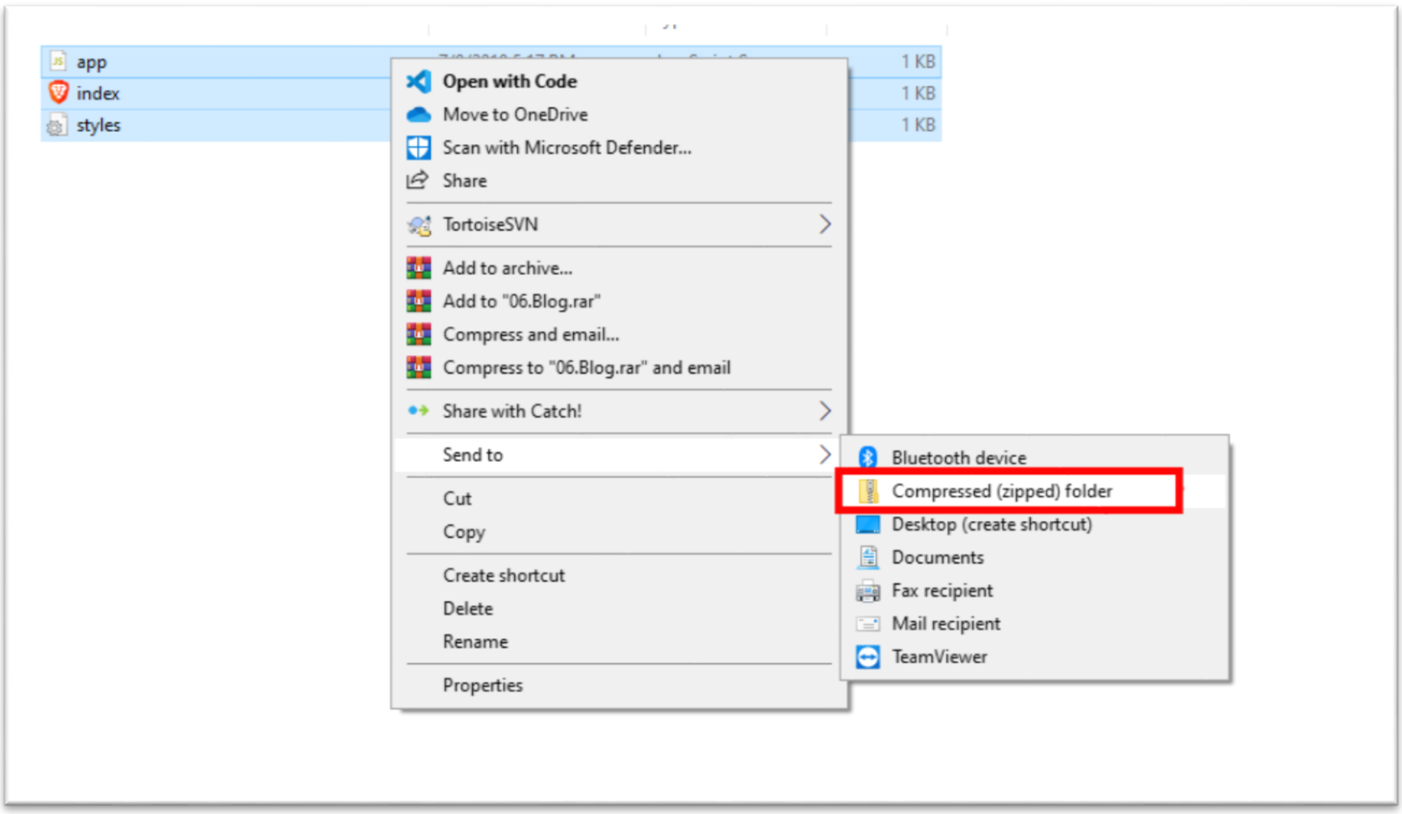
```
{
  "author": "Changed Author",
  "title": "Changed Title"
}
```

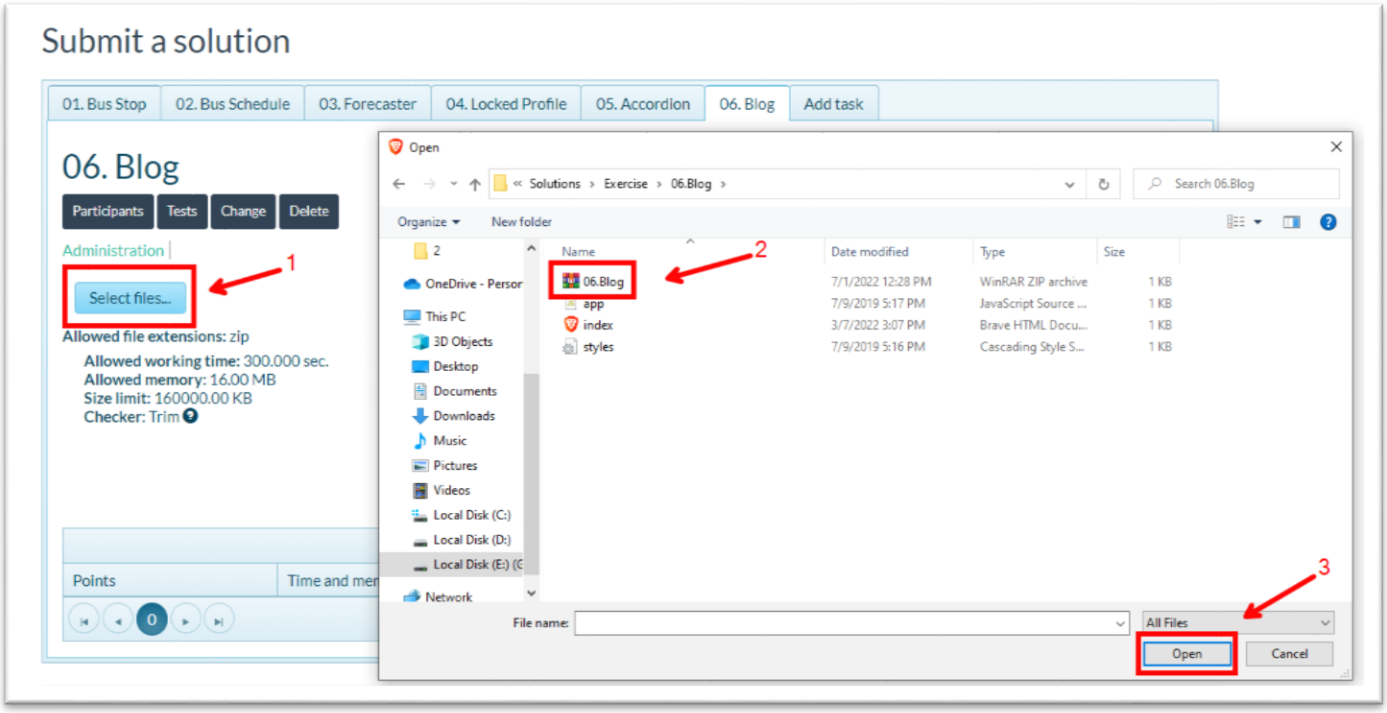
Delete Book

By clicking the delete button you have to delete the book, without any confirmation. To delete a book use "**DELETE**" command and send **REQUEST**: http://localhost:3030/jsonstore/collections/books/:id

Submitting Your Solution

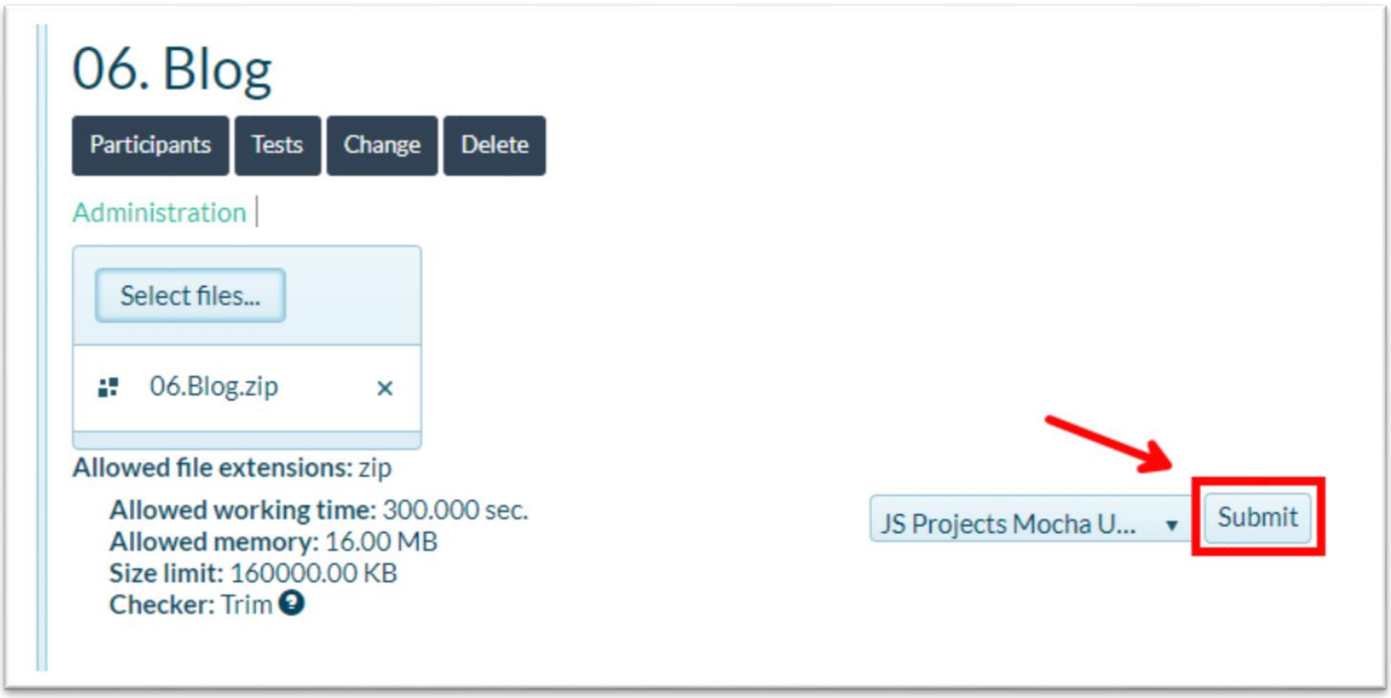
Place in a **ZIP** file the content of the given resources including your solution. Exclude the **`node_modules`** & **`tests`** folders. Upload the archive to Judge.





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

Name	Date modified	Type	Size
06.Blog	7/1/2022 12:28 PM	WinRAR ZIP archive	1 KB
app	7/9/2019 5:17 PM	JavaScript Source ...	1 KB
index	3/7/2022 3:07 PM	Brave HTML Docu...	1 KB
styles	7/9/2019 5:16 PM	Cascading Style S...	1 KB



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