**Exercise: HTTP and REST**

Problems for exercises and homework for the ["JS Front-End" course @ SoftUni](https://softuni.bg/trainings/3976/js-front-end-february-2023)

**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](https://github.com/softuni-practice-server/softuni-practice-server).

## 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:

Text

Description automatically generated

## 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





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





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





## 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.





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:





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:





## 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.









## 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.**

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**".





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**.



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.





## 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

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

## 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

Graphical user interface, text, application, chat or text message

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

## 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**

Graphical user interface, application

Description automatically generated

## 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"



### 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:



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**](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.



Картина, която съдържа текст

Описанието е генерирано автоматично

Картина, която съдържа текст

Описанието е генерирано автоматично