Lab:Asynchronous Programming

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.

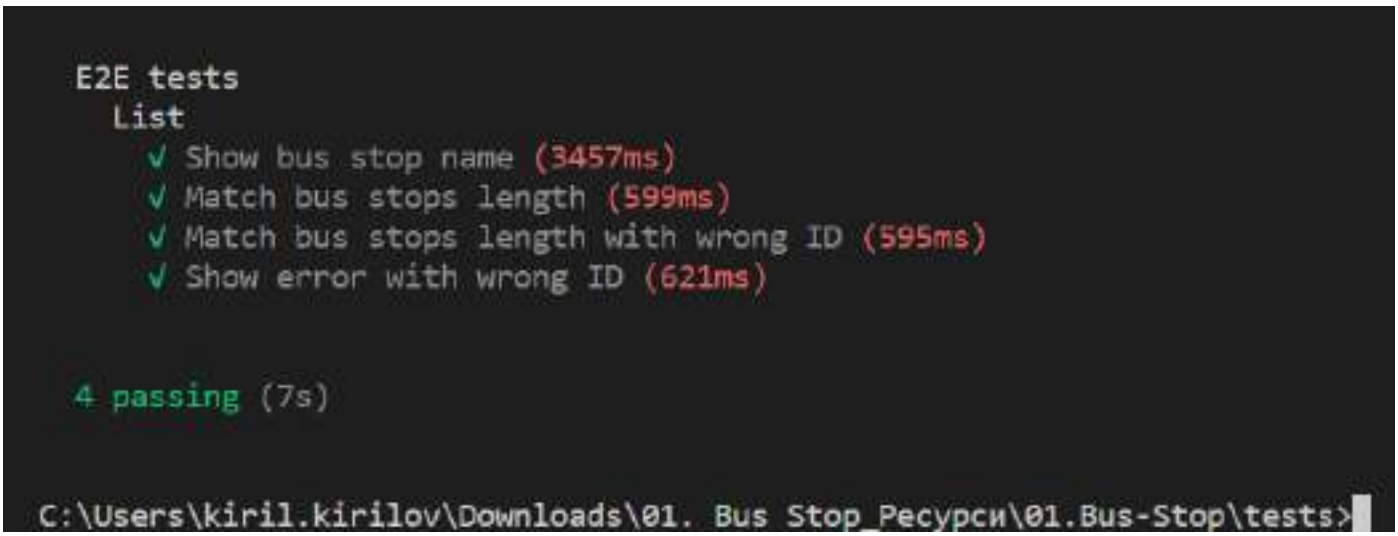
Requirements

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:



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:


```
▼<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 **☀** // ☀
- Partly sunny **⛅** // ⛅
- Overcast **☁** // ☁
- Rain **☔** // ☂
- Degrees **°** // °

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.