# JS Applications Retake Exam – Local Orphanages

*In this day and age, thousands of children have to live in a cold, lonely world, without the support of their parents. Left in an orphanage, they are forced to live without the presence of many things that bring them joy and happiness. Your task is to create an app and give an opportunity to volunteers to send resources which will support orphanages around the country.*

You are assigned to implement a **Single-Page Web application** (SPA) using JavaScript. The application should dynamically display content, based on the user’s interactions, and support user profiles and CRUD operations, using a REST service.

**Only the last submission will be evaluated!**

## Overview

**Implement** a front-end app (**SPA**) for viewing and managing the needs of **local orphanages**. The application allows visitors to browse through the posts catalog. Users may **register** with an **email** and **password,** which allows them to **make a post** about the resources needed or **donate** to different orphanages. Post authors can also **edit** and **delete** their own posts at any time.



## Technical Details

You are provided with the following resources:

* **Project scaffold**: A **package.json** file, containing a list of common dependencies. You may change the included libraries to your preference. The sections **devDependencies** and **scripts** of the file are used by the automated testing suite, altering them may result in incorrect test operation.

To **initialize** the project, execute the command npm install via the command-line terminal.

* **HTML** **and CSS files**: All views (pages) of the application, including **sample** user-generated **content**, are included in the file **index.html**, which links to CSS and other static files. **Each view is in a separate section** of the file, which can be identified by a **unique class name or id** attribute. Your application may use any preferred method (such as a **templating library** or manual visibility settings) to display only the selected view and to **navigate** between views upon user interaction.
* **Local** **REST service**: A special server, which contains **sample data** and supports **user registration** and **CRUD operations** via REST requests is included with the project. Each section of this document (where applicable) includes details about the necessary **REST endpoints**, to which **requests** must be sent, and the **shape** of the expected **request body**.

For **more information** on how to use the included server, see **Appendix A: Using the Local REST Service** at the end of this document.

* **Automated tests**: A complete test suite is included, which can be used to test the correctness of your solution. **Your work will be assessed, based on these tests.**

For **more information** on how to run the tests, see **Appendix B: Running the Test Suite** at the end of this document.

**Do not use CDN** for loading the dependencies because it can **affect the tests in a negative** **way**!

**Note:** When creating HTML Elements and displaying them on the page, **adhere as closely as possible to the provided HTML** samples. Changing the structure of the document may **prevent the tests** from running correctly, which will **adversely affect your assessment grade**. You may **add attributes** (such as **class** and **dataset**) to any HTML Element, as well as **change the "href"** attributes on links and add/change the **method** and **action** attributes of HTML Forms, to facilitate the correct operation of a routing library or another method of abstraction. You may also add hidden elements to help you implement certain parts of the application requirements.

## Application Requirements

### Navigation Bar (5 pts)

Implement a **NavBar** for the app: navigation links should correctly change the current screen (view).

Navigation links should correctly change the current page (view). **Guests** (non-authenticated visitors) can see the links to the **Dashboard,** as well as the links to the **Login** and **Register** pages. The logged-in user navbar should contain the links to the **Dashboard** page, **My Posts** page, the **Create Post** page, and a link for the **Logout** action.

**User** navigation example:



**Guest** navigation example:



### Login User (5 pts)

The **included REST service** comes with the following **premade** user accounts, which you may use for development:

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

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

The **Login** page contains a form for existing user authentication. After the user enters an **email and a password,** the app should log in the user in the system **if there are no empty fields.**

**Graphical user interface, website

Description automatically generated**

Send the following **request** to perform login:

Method: POST

URL: /users/login

Required **headers** are described in the documentation. The service expects a body with the following structure:

{

email,

password

}

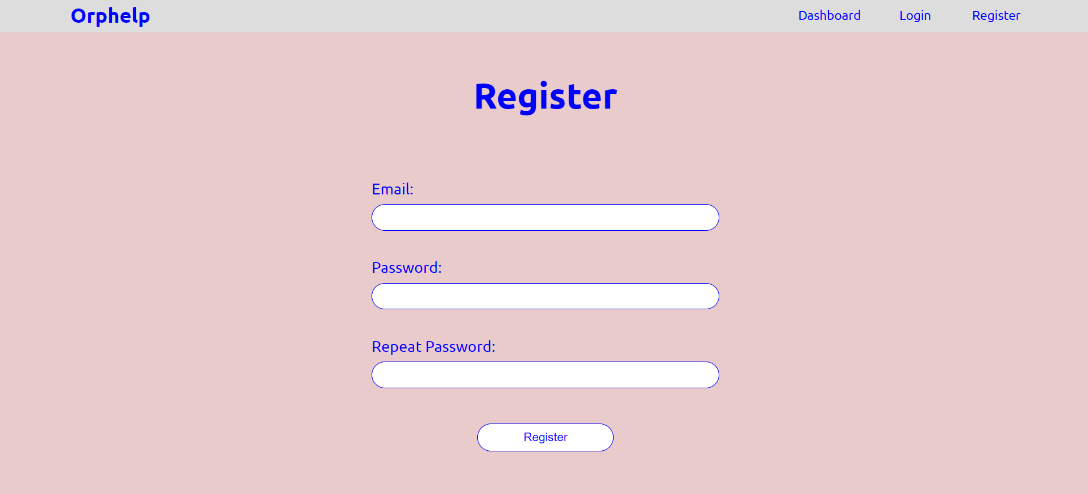
Upon success, the **REST service** will return the **newly created object** with an automatically generated **\_id** and a property **accessToken**, which contains the **session token** for the user **–** you need to store this information using **sessionStorage** or **localStorage,** to be able to perform authenticated requests.

If the login was successful, **redirect** the user to the **Dashboard** page. If there is an **error**, or the **validations** don’t pass, display an appropriate error message, using a system dialog (window.alert).

### Register User (10 pts)

The **Register** page contains a form for new user registration. After the user enters an **email,** and **password,** theapp should register a new user in the system.

**All fields** are **required** – if any of them is empty, display an **error**.



Send the following **request** to perform registration:

Method: POST

URL: /users/register

Required **headers** are described in the documentation. The service expects a body with the following shape:

{

email,

password

}

Upon success, the **REST service** will return the newly created object with an automatically generated **\_id** and a property **accessToken**, which contains the **session token** for the user **–** you need to store this information using **sessionStorage** or **localStorage,** to be able to perform authenticated requests.

If the registration was successful, **redirect** the user to the **Dashboard** page. If there is an error or the **validations** don’t pass, display an appropriate error message, using a system dialog (window.alert).

### Logout (5 pts)

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

Method: GET

URL: /users/logout

The 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 the browser storage.

If the logout was successful, **redirect** the user to the **Dashboard** page.

### Dashboard (15 pts)

The **Dashboard** **Page** displays a **list of all materials** needed by all the orphanages in the system. Clicking on the **[Details]** button in the cards leads to the **details page** for the selected post. This page should be visible to both **guests** and **logged-in** users.

Graphical user interface, application

Description automatically generated

If there are **no posts**, the following view should be displayed:

Graphical user interface, application

Description automatically generated

Send the following **request** to read the list of posts:

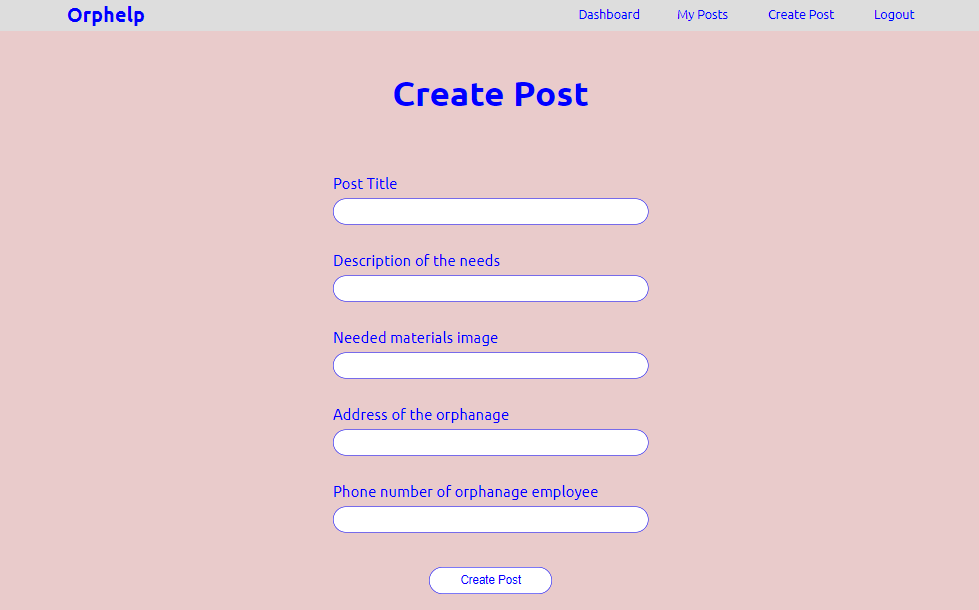
Method: GET

URL: /data/posts?sortBy=\_createdOn%20desc

The required **headers** are described in the documentation. The service will return an array of posts

### Create Page (15 pts)

The **Create page** is available to **logged-in users**. It contains a form for adding a new post about necessary **materials or services**. Check if **all the fields are filled** out before you send the request.



To create a post, send the following **request**:

Method: POST

URL: /data/posts

The required **headers** are described in the documentation. The user must enter:

* **title** - the materials needed to be donated
* **description –** more detailed information about the needed materials – size, quantity, etc.
* **imageUrl** – an image of the expected material
* **address –** the address where the materials need to be sent
* **phone** – a phone number on which the orphanage employee can be contacted

The service expects a body with the following shape:

{

title,

description,

imageUrl,

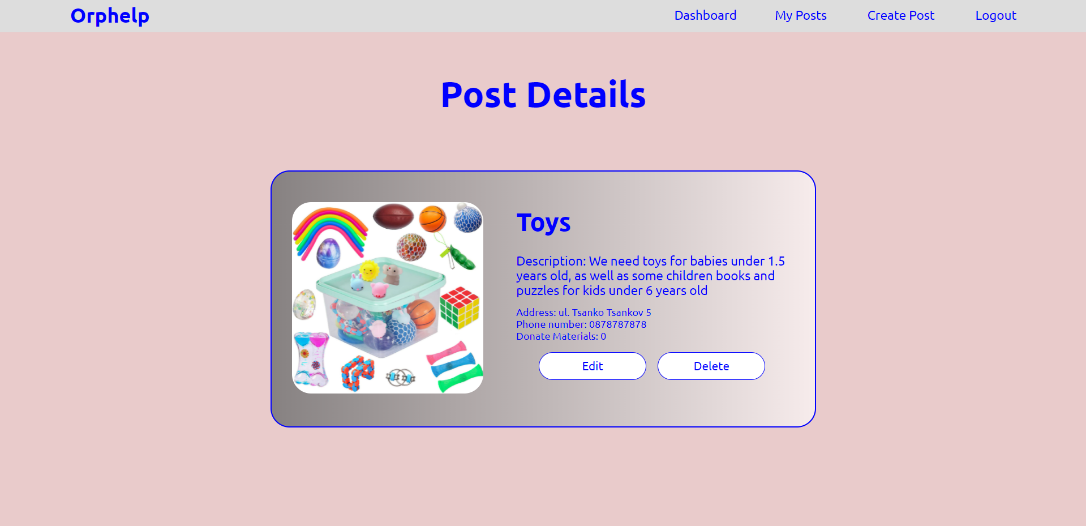
address,  
 phone

}

The required **headers** are described in the documentation. The service will return the newly created record. Upon success, **redirect** the user to the **Dashboard** page.

### Details (10 pts)

All users should be able to **view the details** aboutthe posts. Clicking the **[Details]** button on the **post** on the dashboard page should **display** the **Details** page. If the currently **logged-in user** is the **author**, the **Edit** and **Delete** buttonsshould be displayed, otherwise, they should **not be visible**. The view for the **authors** should look like this:



The view for the **no-author** should look like this:



If there is **no logged-in user** the details page should look like this:



Send the following **request** to read a single post:

Method: GET

URL: /data/posts/*:id*

Where ***:id*** is the **id** of the desired post. The required **headers** are described in the documentation. The service will return a single object.

### Edit Page (15 pts)

The **Edit page** is available to **logged-in users** and it allows **authors** to **edit** their **own** post. Clicking the **[Edit]** button of a particular post on the **Details** page should display the **Edit** page, with all fields filled with the data for the post. It contains a form with input fields for all relevant properties. Check if **all the fields** are **filled** before you send the request. The fields must be filled in with the previous data when the page is first loaded.

Graphical user interface, application

Description automatically generated

To edit a post, send the following **request**:

Method: PUT

URL: /data/posts/*:id*

Where ***:id*** is the **id** of the desired card.

The service expects a **body** with the following shape:

{

title,

description,

imageUrl,

address,  
 phone

}

The required **headers** are described in the documentation. The service will return the modified record. Note that **PUT** requests **do not** merge properties and will instead **replace** the entire record. Upon success, **redirect** the user to the **Details** page for the current post.

### Delete Post (10 pts)

The delete action is available to **logged-in users** for posts they have created. When the author clicks on the **[Delete]** **button** on any of their posts, a **confirmation dialog** should be displayed, and upon confirming this dialog, the post should be **deleted** from the system and the user should be **redirected** to the **Dashboard** page.

To **delete** a post, send the following **request**:

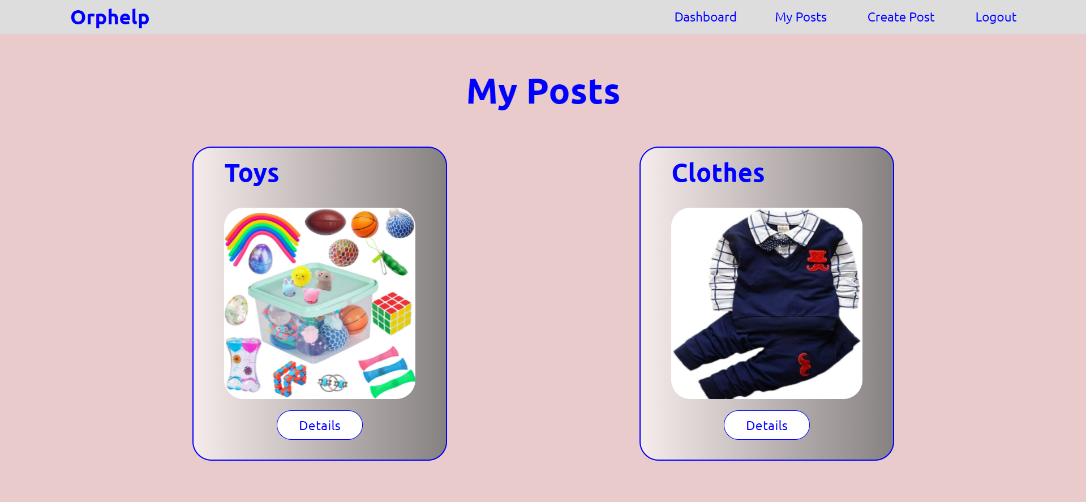
Method: DELETE

URL: /data/posts/*:id*

Where ***:id*** is the **id** of the desired post. The required **headers** are described in the documentation. The service will return an object containing the deletion time. Upon success, **redirect** the user to the **Dashboard** page.

### My Posts (10 pts)

Each **logged-in** **user** should be able to view his posts by clicking [**My Posts**]. All the posts added by the current user should be displayed.



If there are **no posts**, the following view should be displayed:

Graphical user interface, application

Description automatically generated

Send the following **request** to read the list of ads:

Method: GET

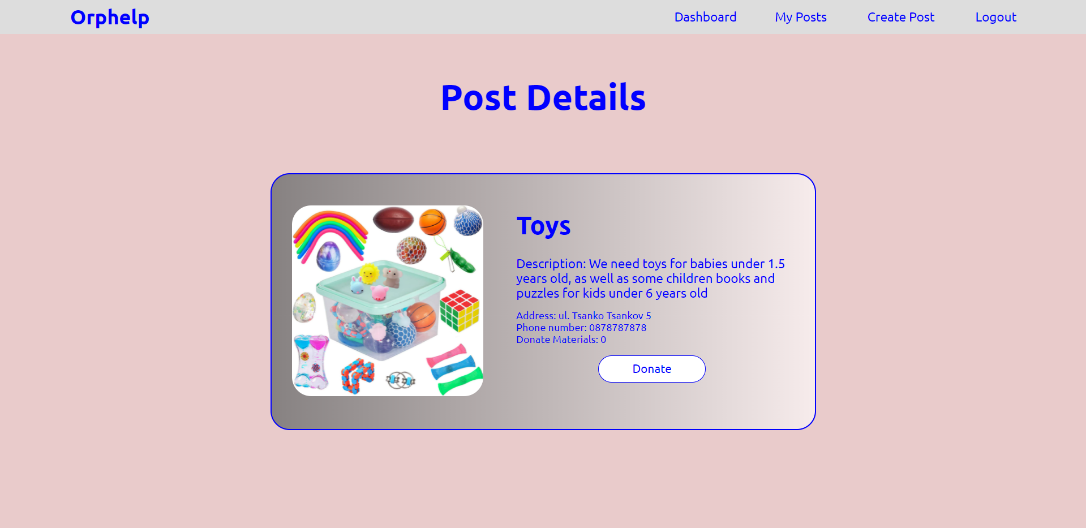
URL: /data/posts?where=\_ownerId%3D%22*{userId}*%22&sortBy=\_createdOn%20desc

Where ***{userId}*** is the **id** of the currently logged-in user. The required **headers** are described in the documentation. The service will return an array of books.

### BONUS: Donate materials (15 Pts)

Every logged-in user should be able to **donate materials requested** in a post. By clicking on the **[Donate]** button,the **donation of each post increases by 1.**

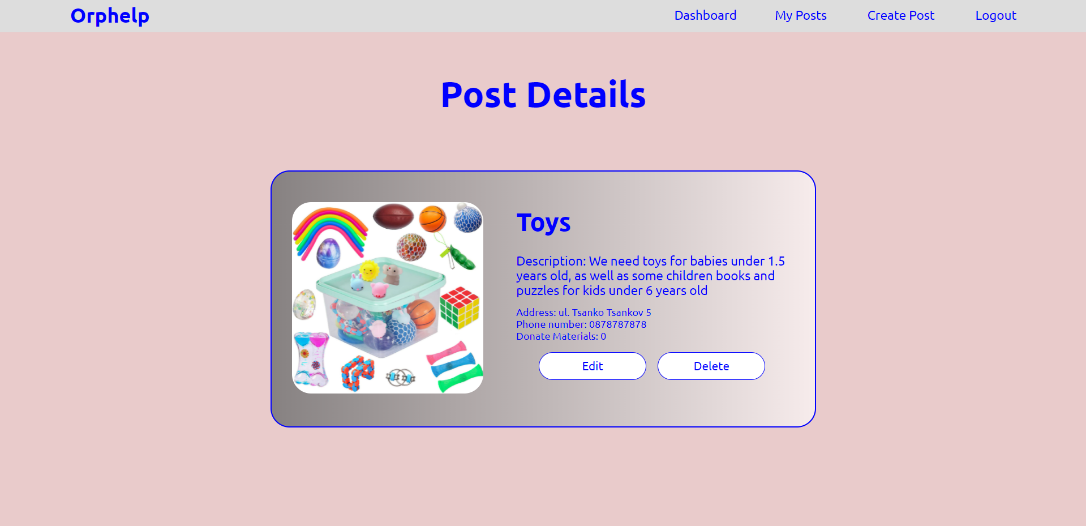
The view when the user has not pressed the **[Donate]** button should look like this:



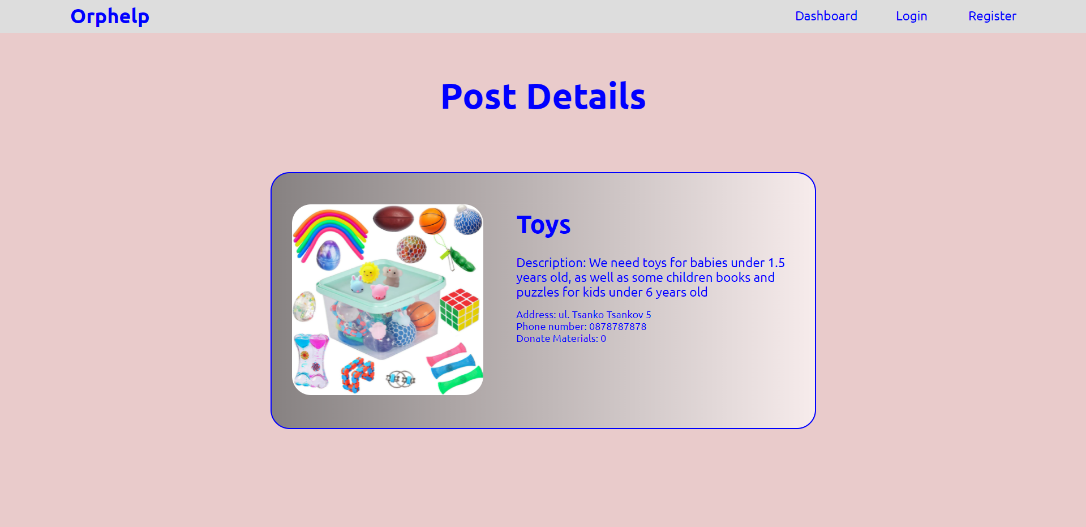
When the user **has made a donation** using the **[Donate]** button, the view should look like this:



**The creator** should not be able to see the **[Donate]** button. The view should look like this:



**Guests** should not be able to see the **[Donate]** button. The view for **guests** should look like this:



Send the following **request to make a donation:**

Method: POST

URL: /data/donations

The service expects a **body** with the following shape:

{

postId

}

The required **headers** are described in the documentation.

Send the following **request to get the total donation count for a post**:

Method: GET

URL: /data/donations?where=postId%3D%22${postId}%22&distinct=\_ownerId&count

Where ***{postId}*** is the **id** of the desired post. Required **headers** are described in the documentation. The service will return the **total donations** count.

Send the following **request to get a donation for a post from a specific user**:

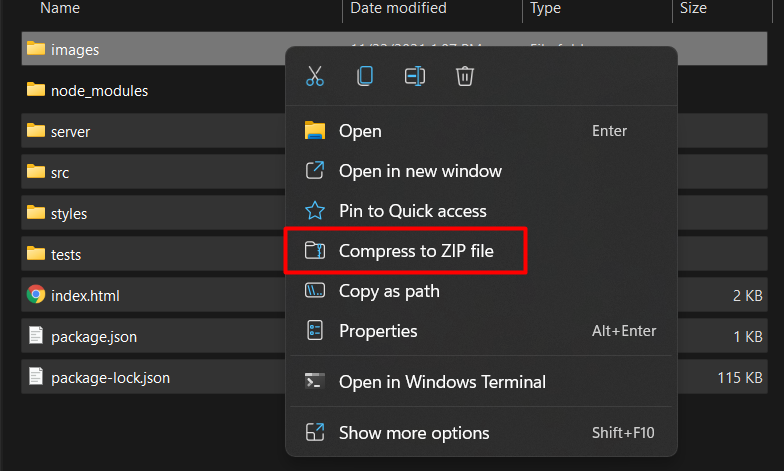
Method: GET

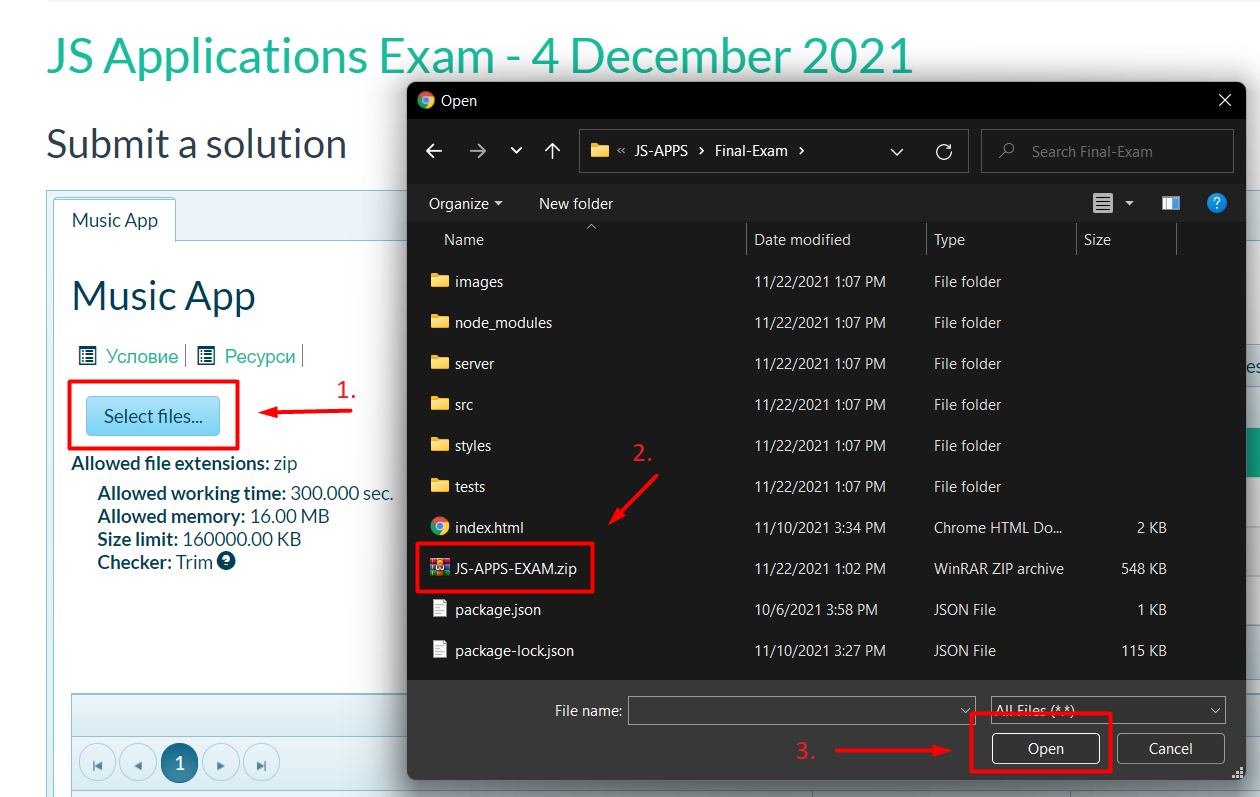
URL:/data/donations?where=postId%3D%22${postId}%22%20and%20\_ownerId%3D%22${userId}%22&count

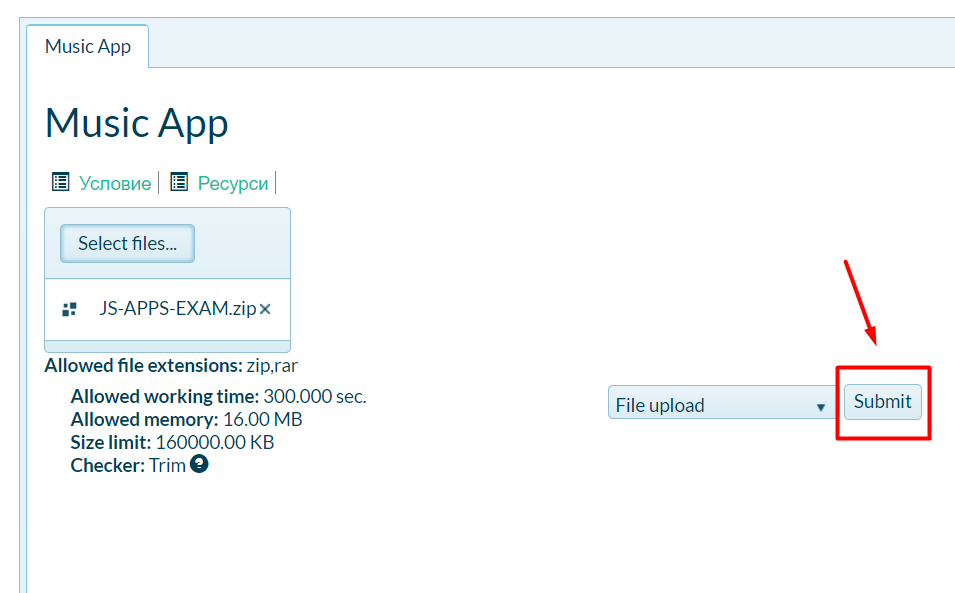
Where ***{postId}*** is the **id** of the desired post and ***{userId}*** is the **id** of the currently logged-in user.  
Required **headers** are described in the documentation. The service will return either **0** or **1**. Depending on that result the **[Donation]** button should be displayed or not.

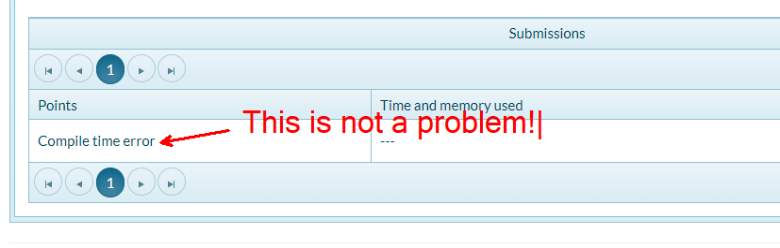
## Submitting Your Solution

Place in a **ZIP** file your project folder. Exclude the node\_modules. Upload the archive to Judge.









## Appendix A: Using the Local REST Service

### Starting the Service

The REST service will be in a folder named “server” inside the provided resources archive. It has no dependencies and can be started by opening a terminal in its directory and executing:

node server.js

If everything initialized correctly, you should see a message about the **host address and port** on which the service will respond to requests.

### Sending Requests

To send a request, use the **hostname** and **port**, shown in the initialization log and **resource address** and **method** as described in the **application requirements**. If data needs to be included in the request, it must be **JSON-encoded**, and the appropriate **Content-Type** **header** must be added. Similarly, if the service is to return data, it will be JSON-encoded. Note that **some requests do not return a body** and attempting to parse them will throw an exception.

Read requests, as well as login and register requests do not require authentication. All other requests must be authenticated.

### Required Headers

To send data to the server, include a **Content-Type** header and encode the body as a JSON-string:

Content-Type: application/json

*{JSON-encoded request body as described in the application requirements}*

To perform an authenticated request, include an **X-Authorization** header, set to the value of the **session token**, returned by an earlier login or register request:

X-Authorization: *{session token}*

### Server Response

Data response:

HTTP/1.1 200 OK

Access-Control-Allow-Origin: \*

Content-Type: application/json

*{JSON-encoded response data}*

Empty response:

HTTP/1.1 204 No Content

Access-Control-Allow-Origin: \*

Error response:

HTTP/1.1 400 Request Error

Access-Control-Allow-Origin: \*

Content-Type: application/json

*{JSON-encoded error message}*

### More Information

You can find more details on the [GitHub repository of the service](https://github.com/softuni-practice-server/softuni-practice-server/blob/master/README.md).

## Appendix B: Running the Test Suite

### Project Setup

The tests require a web server to deliver the content of the application. There is a development web server included in the project scaffold, but you may use whatever server you are familiar with. Note that specialized tools like **BrowserSync** may interfere with the tests. To initialize the project with its dependencies, open a terminal in the folder, containing the file package.json and execute the following:

npm install

Note that if you changed the section devDependencies of the project, the tests may not initialize properly.



### Executing the Tests

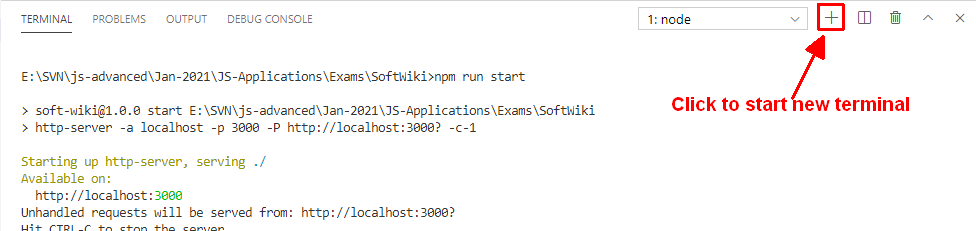
Before running the test suite, make sure a web server is operational, and the application can be found at the root of its network address. To start the included dev-server, open a terminal in the folder containing package.json and execute:

npm run start

This is a one-time operation unless you terminate the server at any point. It can be restarted with the same command as above.

To execute the tests, open a new terminal (do not close the terminal, running the web server instance) in the folder containing package.json and execute:

npm run test



Test results will be displayed in the terminal, along with detailed information about encountered problems. You can perform this operation as many times as it is necessary by re-running the above command.

### Debugging Your Solution

If a test fails, you can view detailed information about the requirements that were not met by your application. Open the file e2e.test.js in the folder tests and navigate to the desired section as described below.

**This first step will not be necessary if you are using the included web server.** Make sure the application host is set correctly:



The value for the host must be the address where your application is being served. Make sure that entering this address in a regular internet browser shows your application.

To make just a single test run, instead of the full suite (useful when debugging a single failing test), find the test and append .only after the it reference:



On slower machines, some of the tests may require more time to complete. You can instruct the tests to run more slowly by slightly increasing the values for interval and timeout:



Note that interval values greater than 500 and timeout values greater than 10000 are not recommended.

If this doesn’t make the test pass, set the value of DEBUG to true and run the tests again – this will launch a browser instance and allow you to see what is being tested, what the test checks, and where it fails (or hangs):



If the actions are happening too fast, you can increase the value of slow-mo. If the browser hangs, you can just close it and abort any remaining tests by focusing on the terminal window and pressing [Ctrl+C] followed by the letter "y" and [Enter].

The final thing to look for is the exact row where the test fails:

