# CarTube

Create a **Web application** (SPA) using JavaScript. The application should dynamically display content, based on user interactions. It should support **user profiles** and **CRUD** operations using a **REST** service.

## Overview

**Implement** a front-end application (SPA) to view and manage **car listings**. The application should allow visitors to browse through different car ads. Users should be able to **register** in the application providing a **username** and a **password**, this should allow them to **create** their own ads. Ad authors can also **edit** or **delete** their own publications at any time.

## Technical Details

The following resources are provided:

* **Project scaffold**: A **package.json** file, containing a list of common dependencies. You can 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 results.

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

* **HTML** **and CSS files**: All necessary views and pages for the app, including **sample** user-generated **content**, are included in the file **index.html**, which is linked to the 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 an id** attribute. Your application may use any preferred method (such as a **templating library** or manual visibility settings) to display the selected view and to **navigate** between views on user interaction.
* **Local** **REST service**: This is a server, which contains **sample data** and supports **user registration** and **CRUD operations** via REST requests. It is included in 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** in this document.

* **Automated tests**: A complete test suite is included, which can be used to test your solution.

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

**Note:** When creating the HTML Elements and displaying them on the page, **adhere as close as possible to the provided HTML** samples. Changing the structure of the document may **prevent the tests** from running correctly, which may **affect your assessment grade**. You may **add attributes** (such as **classes** and **datasets**) to any HTML Element, as well as **change "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.

## **Appendix A: Using the Local REST Service**

### **Starting the Service**

The **REST service** will be in the "**server**" folder in the provided resources. It has no dependencies and can be started by opening a terminal in its directory and executing:

**node server.js**

If it is 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 the 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 the 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** and returned by an earlier login or register request:

**X-Authorization:** *{session token}*

### **Server Response**

The data response:

**HTTP/1.1 200 OK**

**Access-Contrl-Allow-Origin: \***

**Content-Type: application/json**

*{JSON-encoded response data}*

Empty responses:

**HTTP/1.1 204 No Content**

**Access-Contrl-Allow-Origin: \***

**HTTP/1.1 400 Request Error**

**Access-Contrl-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).

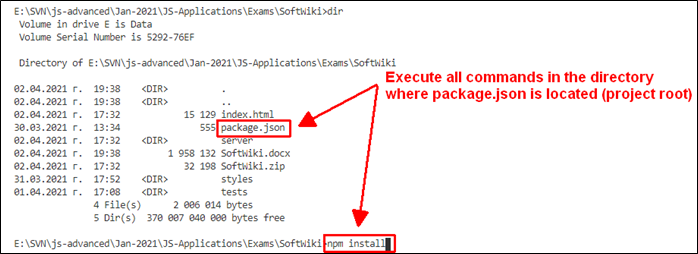
## **Appendix B: Running the Test Suite**

### **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 a server that 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**

If you changed the **devDependencies** section of the project the tests may not initialize properly.



### **Executing the Tests**

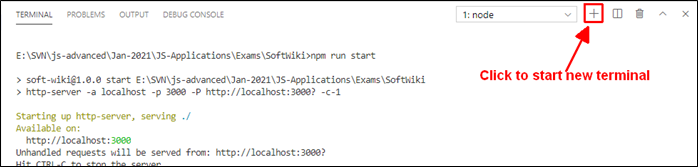
Before running the test suite, make sure a web server is operational and the application can be found at the root of its network address. To start the included dev-server, open a terminal in the folder containing **package.json** and execute:

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

**npm run test**

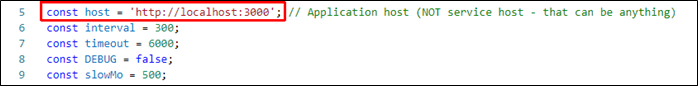


The test results will be displayed in the terminal, along with detailed information about encountered problems. You can perform this operation as many times necessary by re-running the command.

### **Debugging Your Solution**

If a test fails, you can view the information about the requirements that were not met by your application. Open the file **e2e.test.js** in the **tests** folder and navigate to the 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:

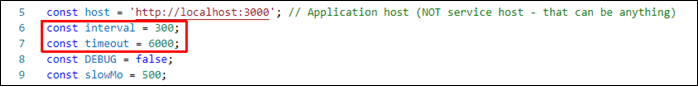


The value for **host** must be the address where your application is being served. Make sure that entering this address in a regular internet browser displays your application.

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

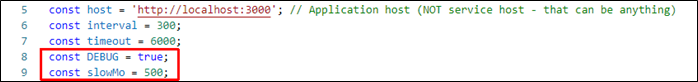


On slower machines, some of the tests may require more time to complete. You can instruct the tests to run slower by increasing the values for **interval** and **timeout**:



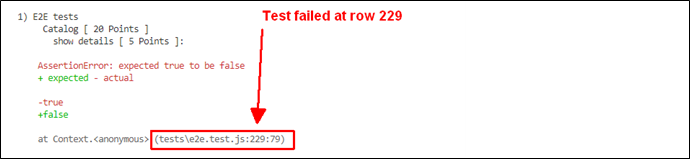
**interval** values greater than 500 and **timeout** values greater than 10000 are not recommended.

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 sees and where it fails (or timeouts):



If the actions are happening too fast, you can increase the value of **slowMo**. If the browser gets stuck, you can close it and abort any remaining tests by selecting the terminal window and pressing **[Ctrl+C]** followed by the letter "y" and **[Enter]**.

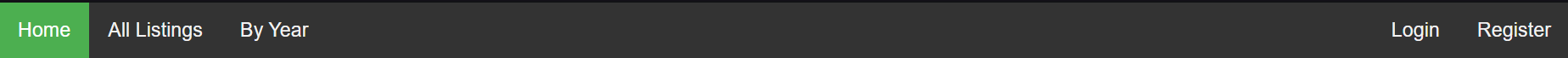
The detail thing to look for is the exact row where the test fails:



## Application Requirements

### Navigation Bar (5 pts)

The navigation links should change the current page (view). **Guests** (unauthenticated visitors) can see the links to the **All Listings** page, as well as the links to the **Login** and **Register** pages. Authenticated users should be able to access the links to the **All Listings** page, the **Create** page and a link to **Logout**.

**Guest** navigation example: 

**User** navigation example:



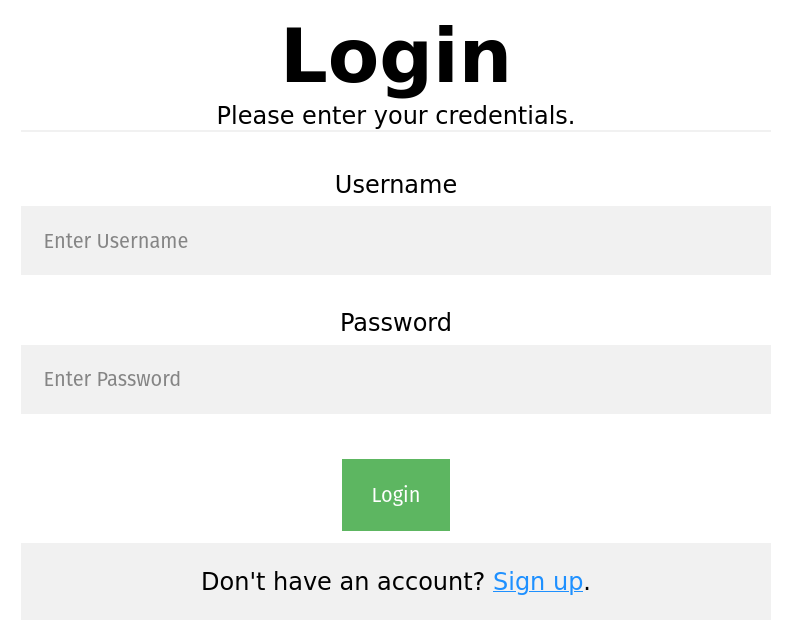
### Login User (5 pts)

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

{ "username": "Peter", "password": "123456" }

{ "username": "John", "password": "123456" }

The **Login** page contains a form to authenticate users. By providing a **username** and a **password,** the app should log a user in to the system if there are no empty **fields** and incorrect data.



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

Method: POST

URL: /users/login

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

{

username,

password

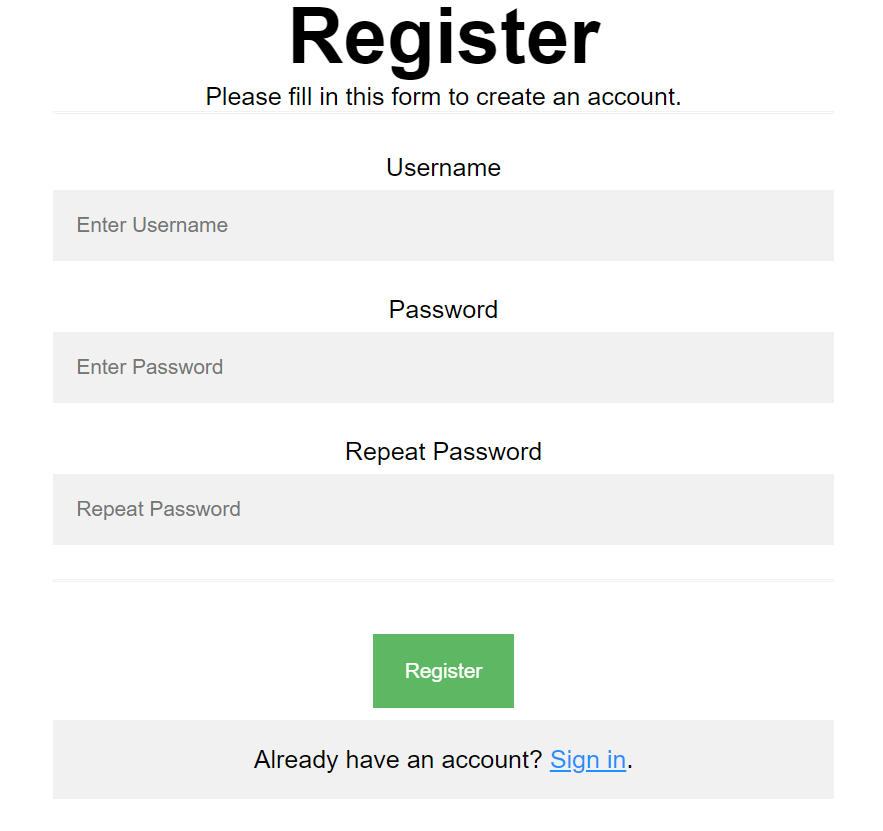
}

If successful, the **REST service** will return the information of the user along with an **accessToken** property, which contains the **session token** for the user**.** This information needs to be stored using **sessionStorage** or **localStorage** in order to be able to perform authenticated requests.

If the login was successful, **redirect** the user to the **All Listings** page. If there is an error, display an error message using a system dialog (window.alert).

### Register User (10 pts)

The **Register** page contains a form to register a new user. By providing a **username** and a **password,** the app should register a new user in the system if there are no empty **fields** or incorrect data.



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

Method: POST

URL: /users/register

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

{

username,

password

}

After a successful registration the **REST service** will return a newly created object with an automatically generated **\_id** property and an **accessToken** property, which contains the **session token** for the user. This information needs to be stored using **sessionStorage** or **localStorage** in order to be able to perform authenticated requests.

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

### Logout (5 pts)

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

Method: GET

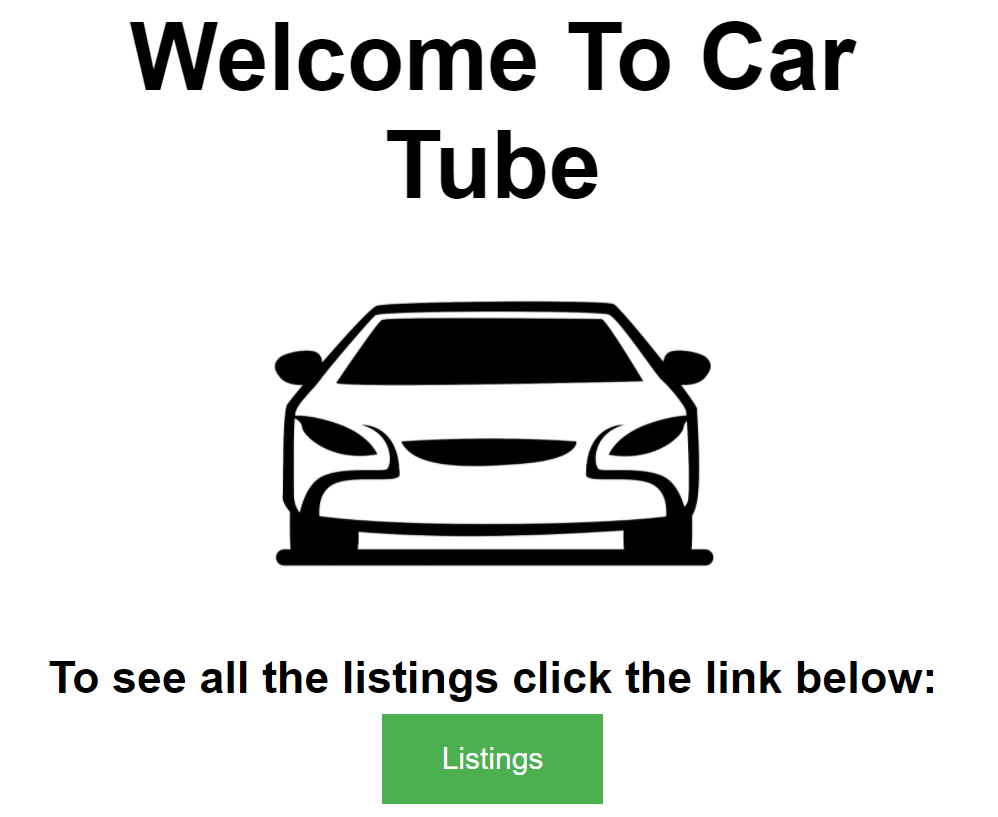
URL: /users/logout

The required **headers** are described in the documentation. Upon a successful logout, the **REST service** will return an **empty response**. Clear any session information that is stored in browser storage.

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

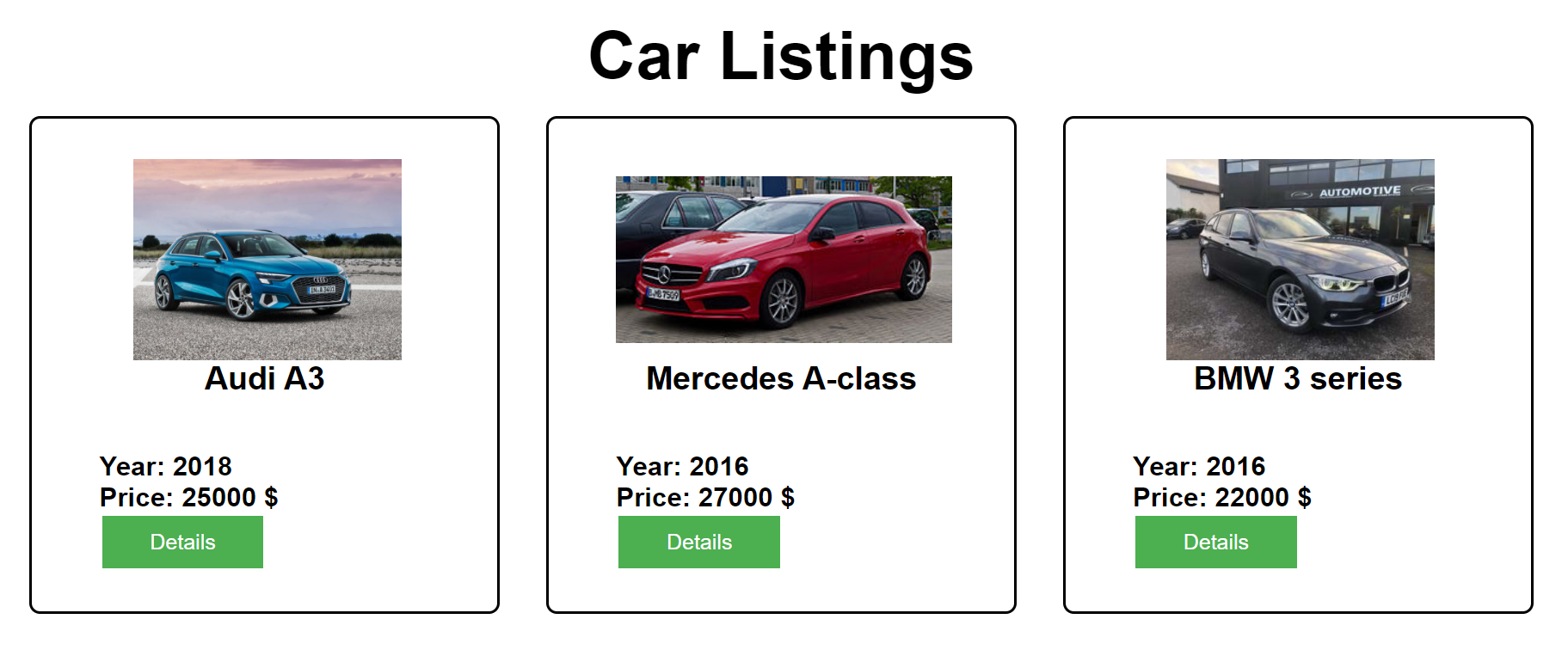
### Home Page (5 pts)

All users should be welcomed by the Homepage, from where they can go to the **Listings** view.

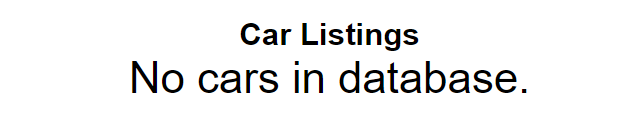


### All Listings Page (10 pts)

This page displays all listings in the system. Clicking on the **details** button in the cards leads to the details page of the selected listing.



If there are nolistings, the following view should be displayed:



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

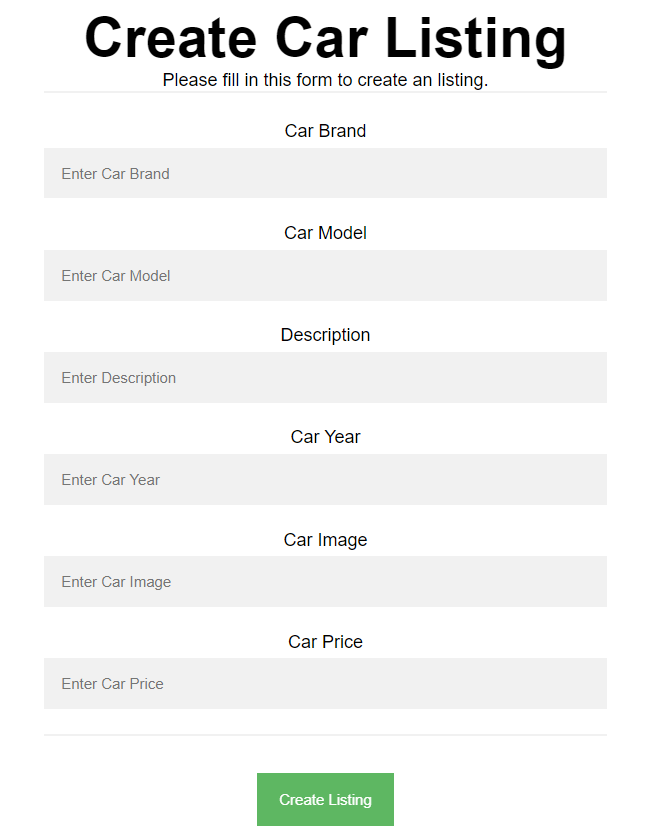
Method: GET

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

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

### Create Car Listing (15 pts)

The **Create** page is available to logged-in users. It contains a form to create new listings. Check if all the fields are filled before sending the request.



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

Method: POST

URL: /data/cars

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

{

brand,

model,

description,

year,

imageUrl,

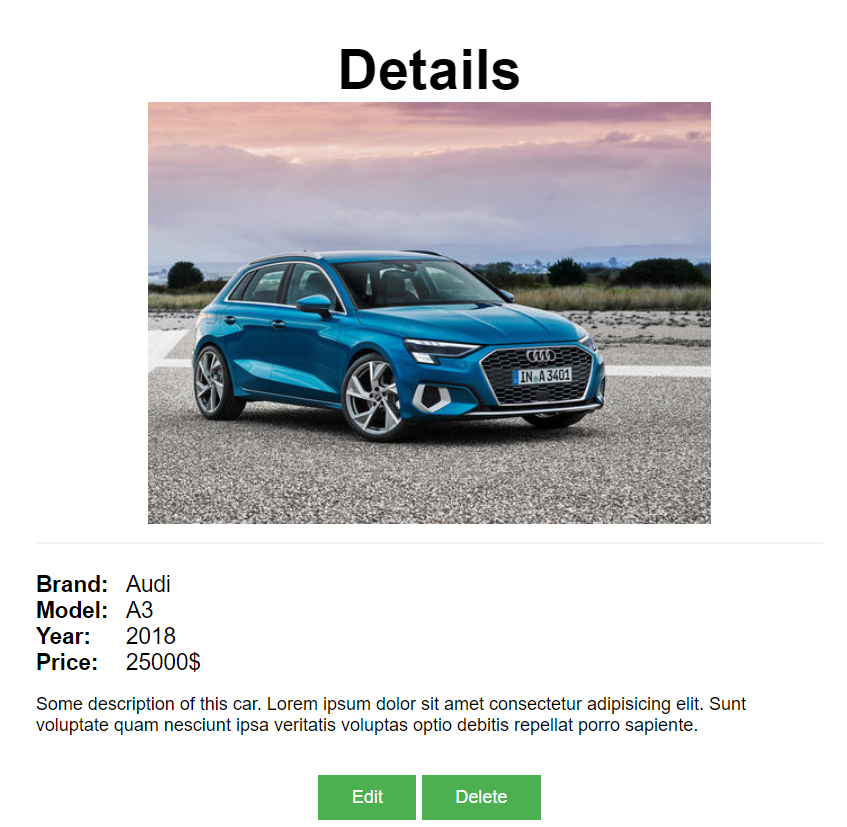
price

}

The required **headers** are described in the documentation. The values for year and the price must be positive numbers. The service will return a newly created record. If successful, **redirect** the user to the **All Listings** page.

### Details (10 pts)

All users should be able to **view details** aboutlistings. Clicking on the **Details** link in the **car ad** should **display** the **Details** page. If the currently logged-in user is the creator of the listing, the **Edit** and **Delete** buttonsshould be displayed.



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

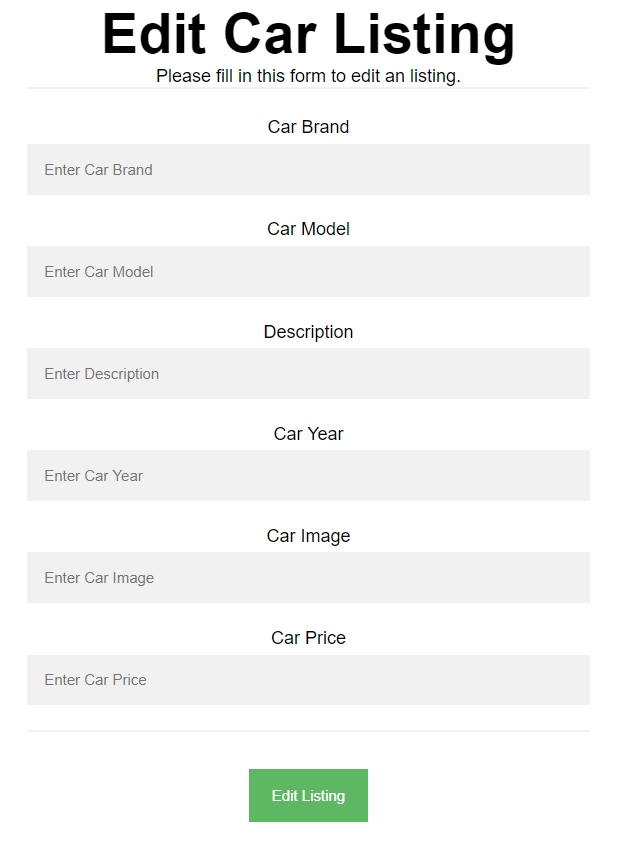
Method: GET

URL: /data/cars/*:id*

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

### Edit Listing (15 pts)

The **Edit** page is available to logged-in users and allows the author to **edit** their **own** listings. Clicking on the **Edit** link of a specific listing on the **Details** page should display the **Edit** page. It contains a form with input fields for all relevant properties. Check if all the fields are filled before you send the request.



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

Method: PUT

URL: /data/cars/*:id*

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

The service expects a body with the following shape:

{

brand,

model,

description,

year,

imageUrl,

price

}

The required **headers** are described in the documentation. The values for year and the price must be positive numbers. The service will return the modified record. If successful, **redirect** the user to the **Details** page.

**Note:** **PUT** requests **do not** merge properties and will instead **replace** the entire record.

### Delete Listing (10 pts)

The delete action is available to **logged-in** users and for listings that they have created. When the author clicks **Delete** a confirmation dialog should be displayed. Upon completing this dialog, the listing should be **deleted** from the system.

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

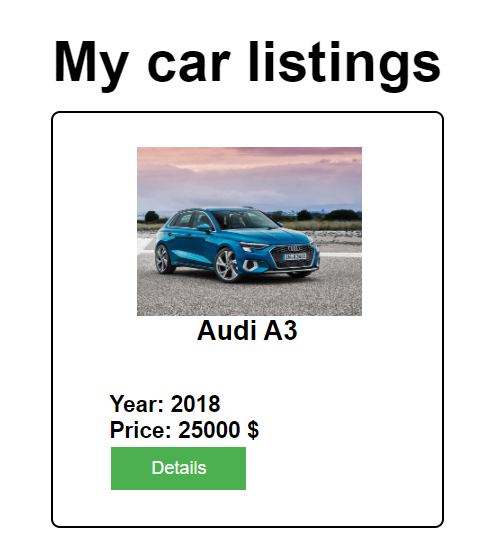
Method: DELETE

URL: /data/cars/*:id*

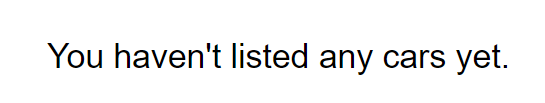
The ***:id*** is the **id** of the desired listing. The required **headers** are described in the documentation. The service will return an object, containing the deletion time. Upon success, **redirect** the user to the **All Listings** page.

### My Listings (10 pts)

This page displays all listings made by the current user.



If there are nolistings, the following view should be displayed:



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

Method: GET

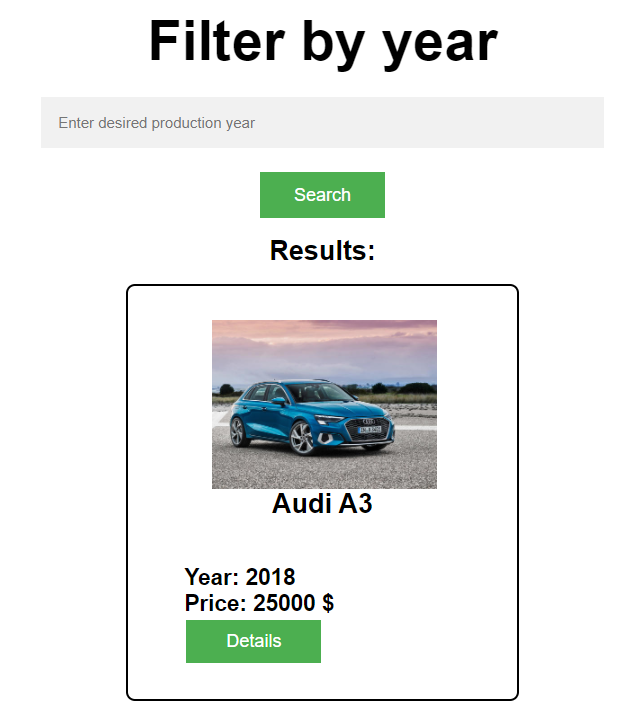
URL: /data/cars?where=\_ownerId%3D%22*{userId}*%22&sortBy=\_createdOn%20desc

The ***{userId}*** is the id of the currently logged-in user.

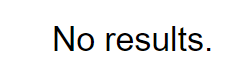
The required **headers** are described in the documentation. The service will return an array of listings.

### BONUS: Search (5 pts)

The **Search** page allows users to filter listings by their production year. It contains an input field and upon submitting a query it displays all matching listings.



If there are nomatching results, the following view should be displayed:



Send the following **request** to read a filtered list of ads by their production year:

Method: GET

URL: /data/cars?where=year%3D*{query}*

***{query}*** is the search information that the user has entered in the input field. The required **headers** are described in the documentation. The service will return an array of listings. If there are no matches, display the text **"No matching listings"**.