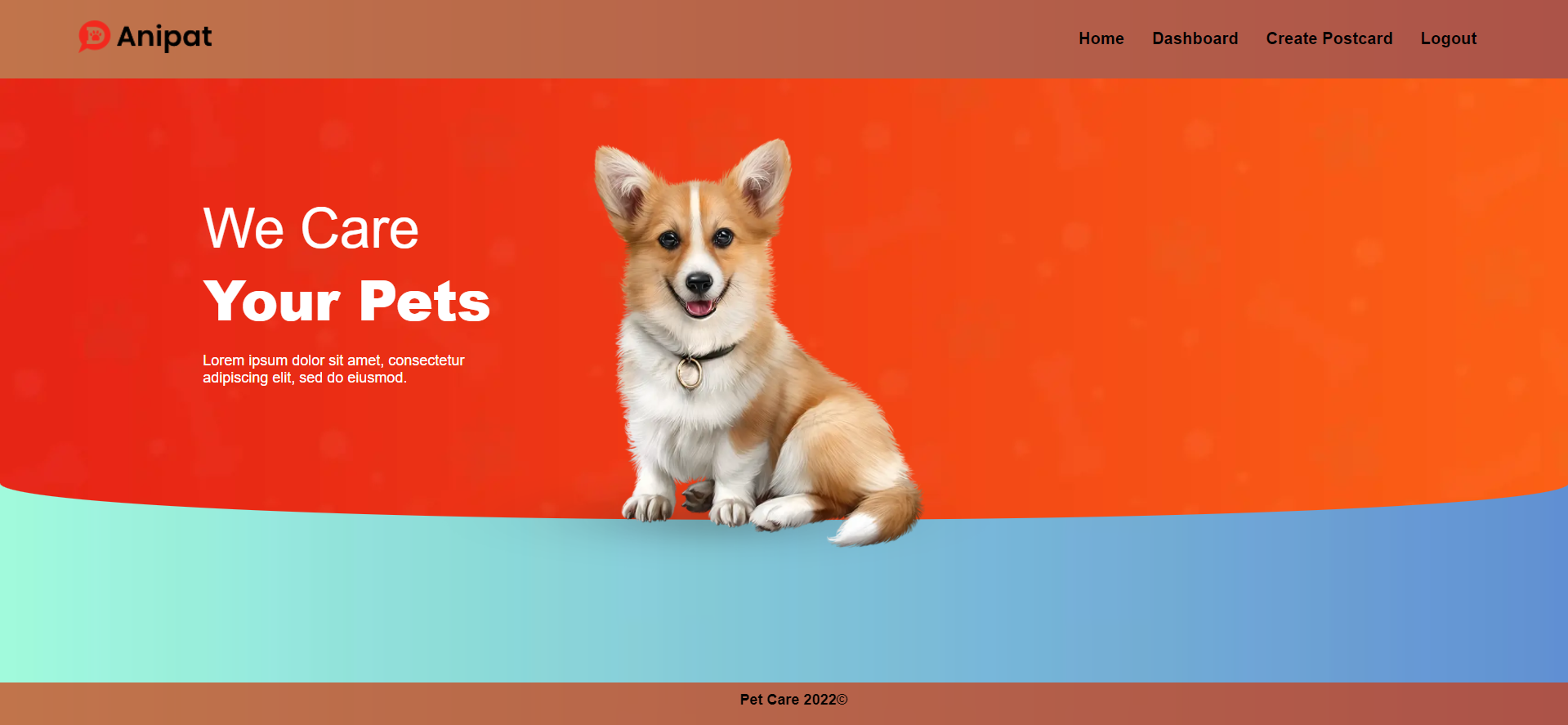
# JS Applications Exam – Pet Care

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

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

## Overview

**Implement** a front-end application (SPA) to see and manage **the pet**. The app allows visitors to **browse** different **pets**, including the **name, breed, and age**. Users can **register and login** with an **email** and **password**, which allows them to **create** their pet postcard. Pets **creators** can also **edit** or **delete** their 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 **negatively affect the tests**!

**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).

**Note:** When creating 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 will **adversely affect your assessment grade**. You may **add attributes** (such as **class** and **dataset**) 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.

## Application Requirements

### Navigation Bar (5 pts)

**Navigation** links should correctly change the current page (view). The **Home** link should redirect to the **Home** page. **Guests** (un-authenticated visitors) can see the links to the **Home,** **Dashboard,** **Login,** and **Register** page. The logged-in user navbar should contain the links to the **Home, Dashboard, Create Postcard** page, and a link for the **Logout** action.

**Guest** navigation example:



**User** navigation example:



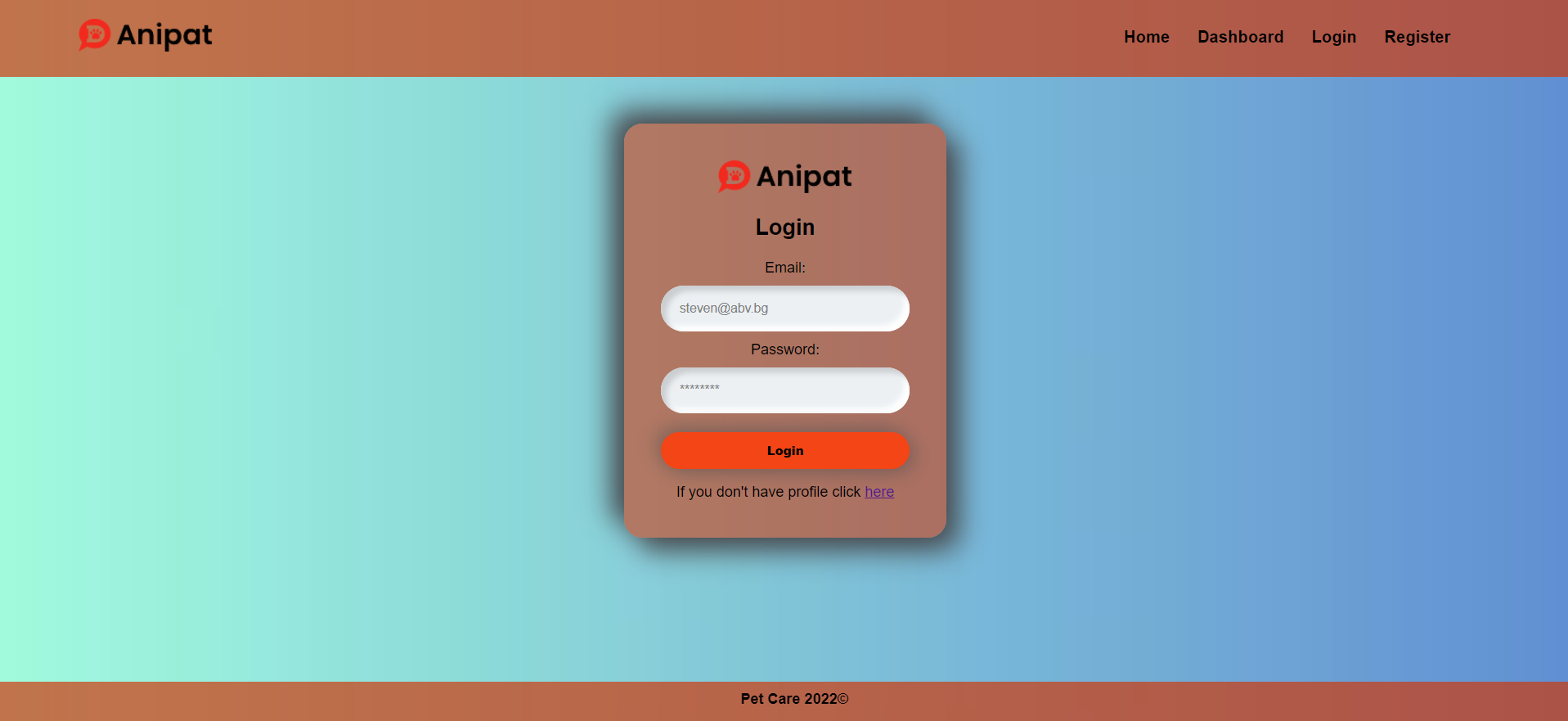
### Login User (5 pts)

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

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

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

The **Login** page contains a form for existing user authentication. By providing an **email** and **password,** the app should login a user into the system if there are no empty fields.



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 shape:

{

email,

password

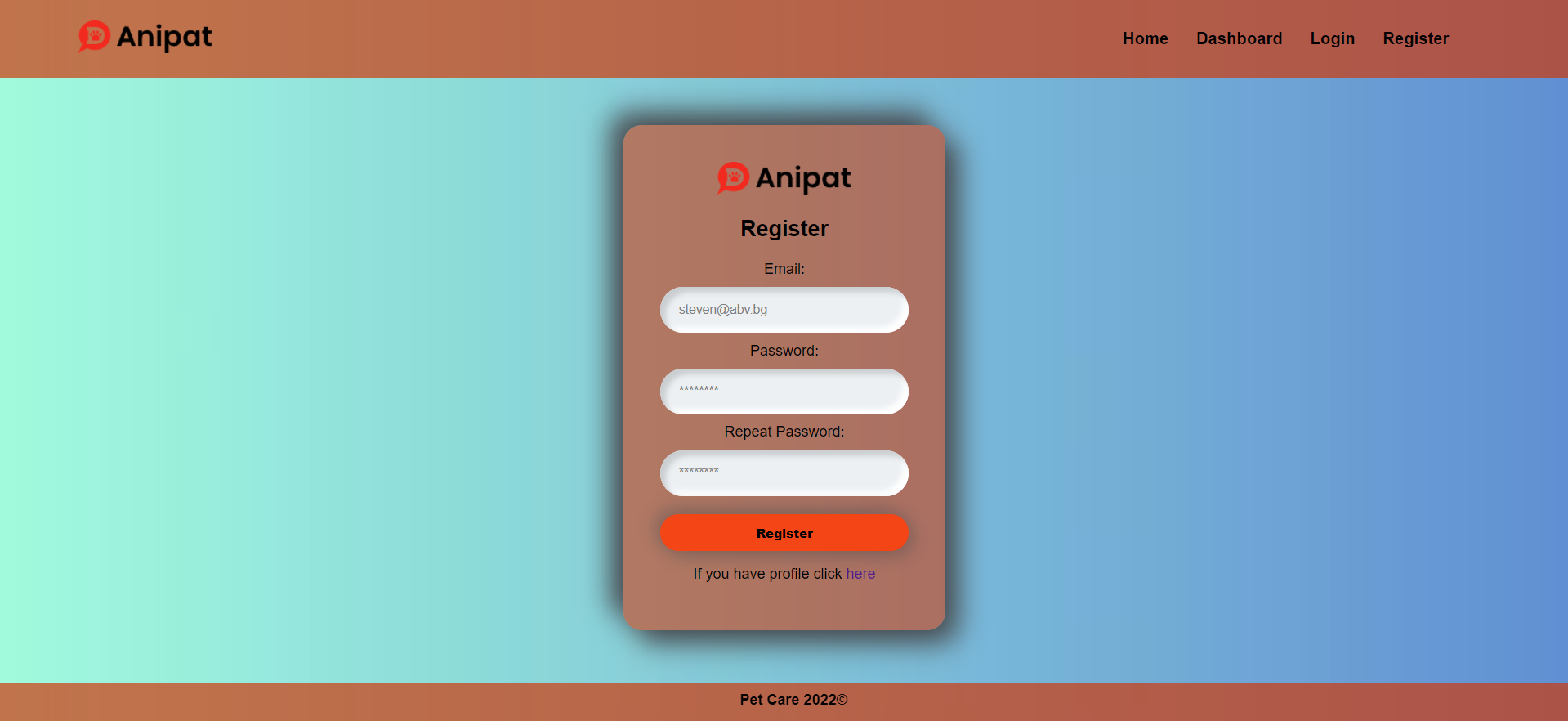
}

Upon success, the **REST service** will return information about the existing user along with 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 **Home** page. If there is an error, 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. By providing an **email** and **password,** the app should register a new user in the system if there are **no empty fields**.



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 **Home** 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

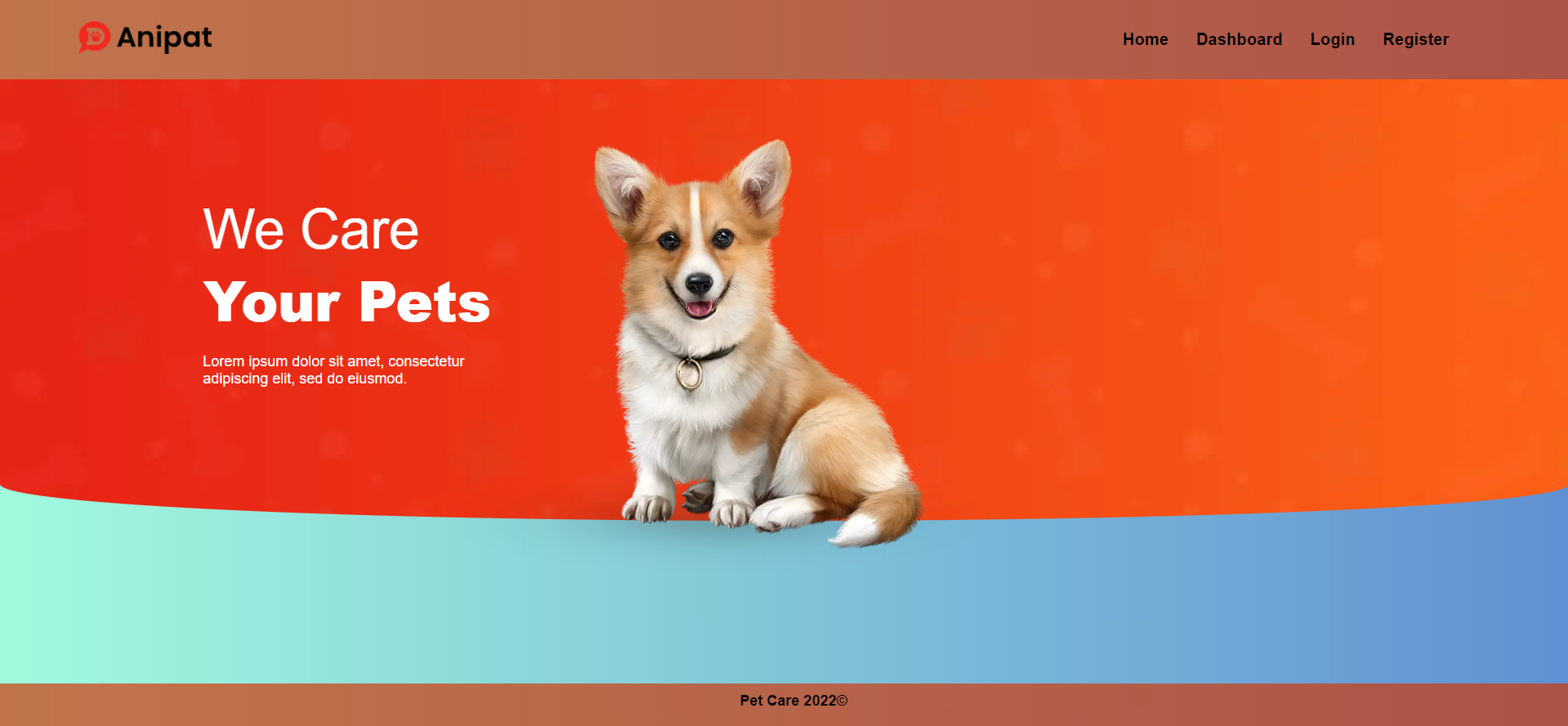
URL: /users/logout

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 browser storage.

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

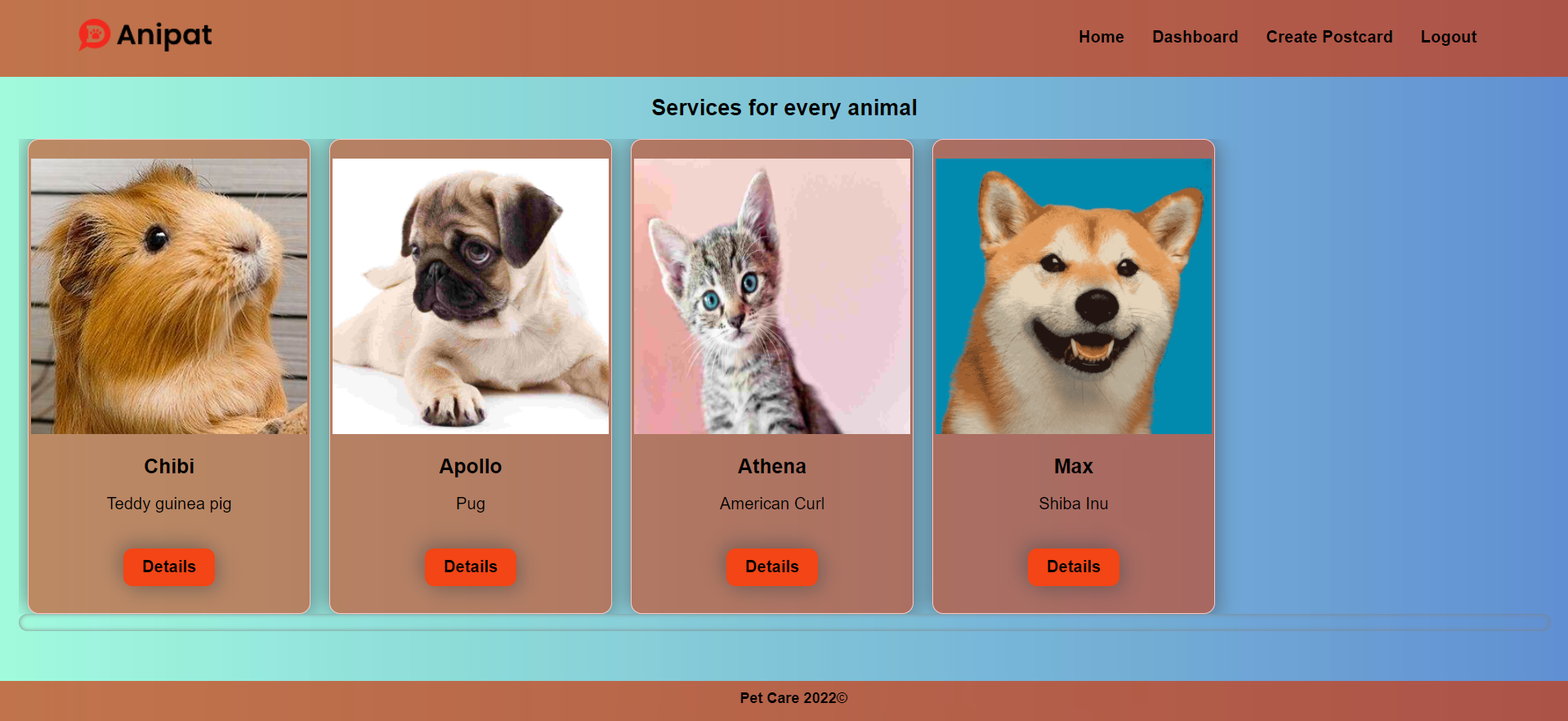
### Home Page (10pts)

All users should be greeted from the **homepage**. There is a static page:

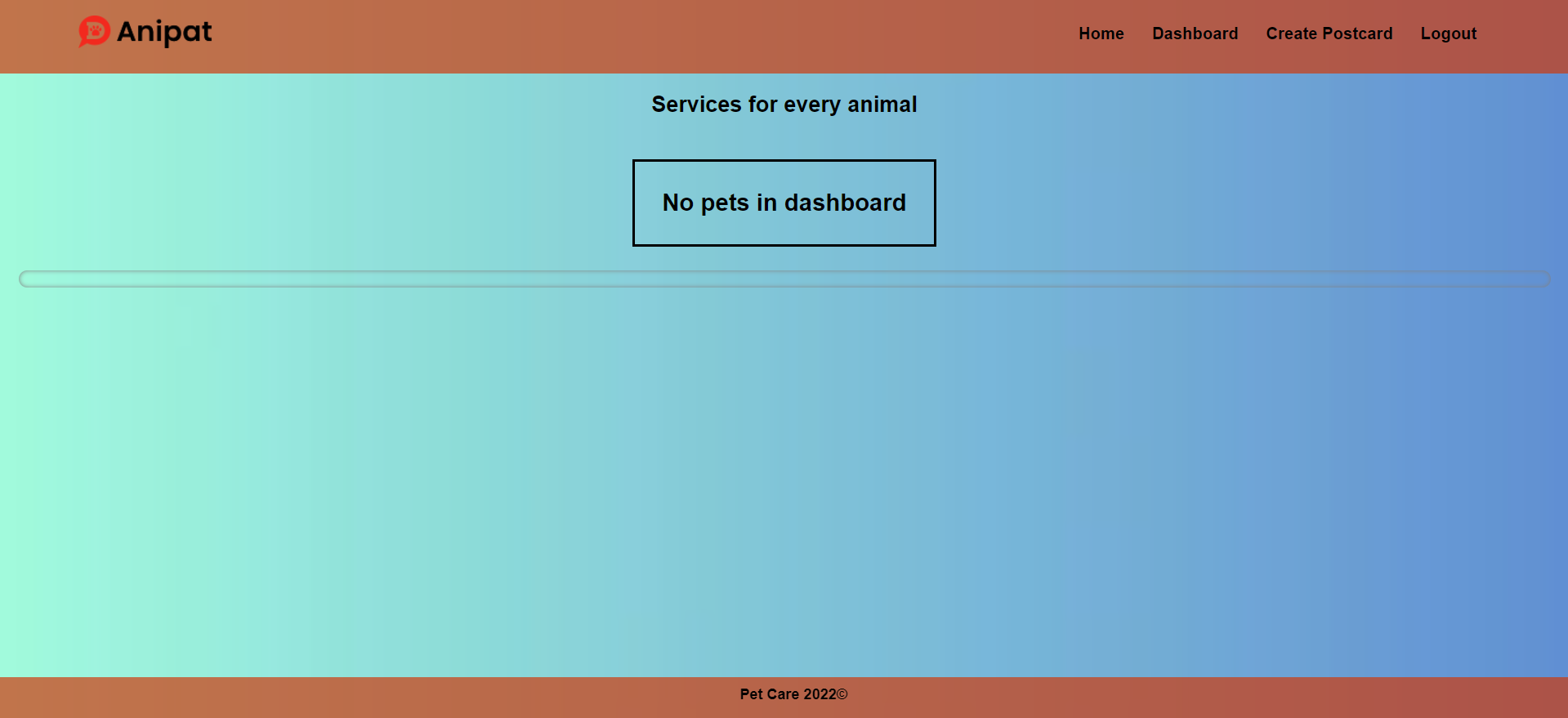


### Dashboard Page (15 pts)

The **Home** Page displays a list of **all pets** in the system. All **users** should be seeing the **list** and have access to **Details Page**.



If there are nopets yet, the following view should be displayed:



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

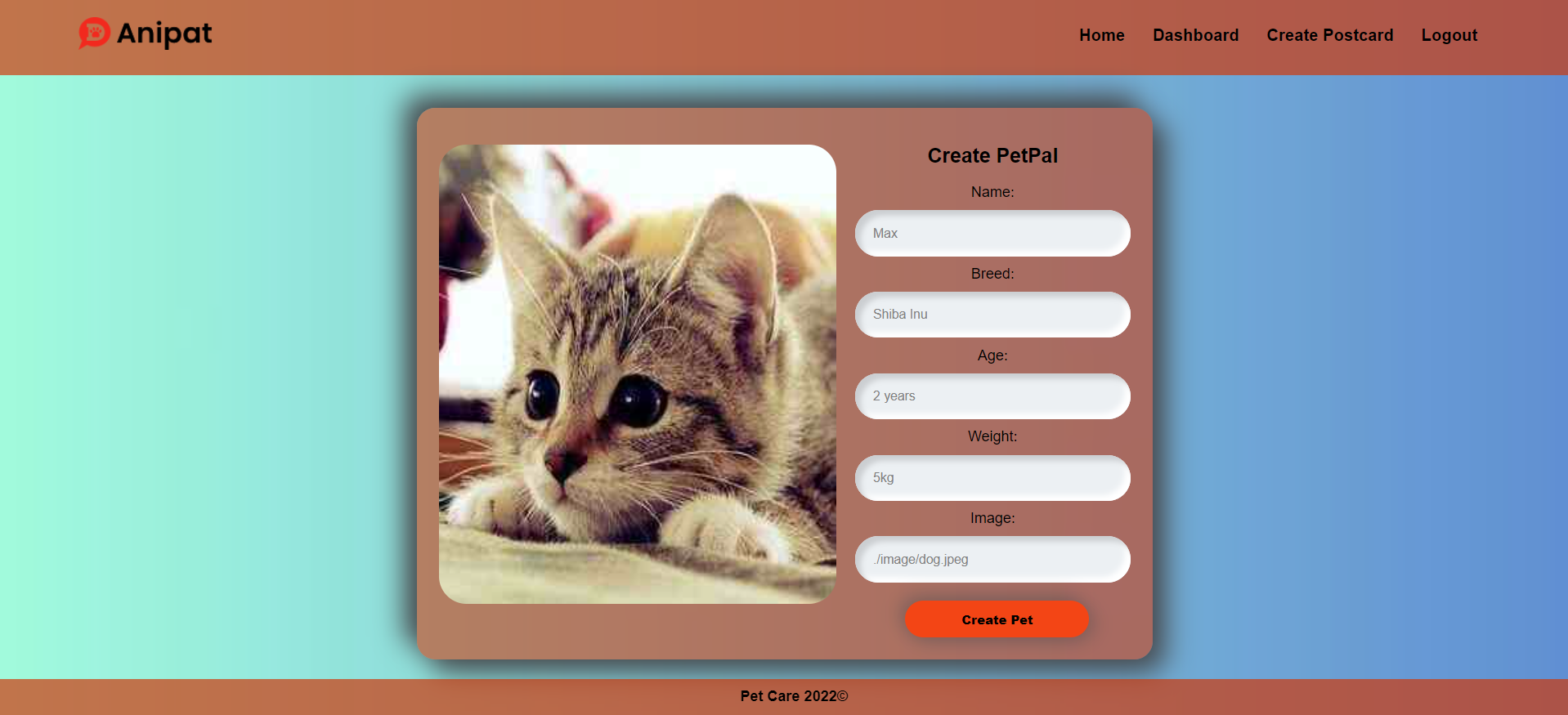
Method: GET

URL: /data/pets?sortBy=\_createdOn%20desc&distinct=name

Required **headers** are described in the documentation. The service will return an array of pets.

### Create Page (15 pts)

The **Create page** is available to logged-in users. It contains a form for creating a new pet postcard. Check if all the fields are filled out before you send the request.



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

Method: POST

URL: /data/pets

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

{

name,

breed,

age,

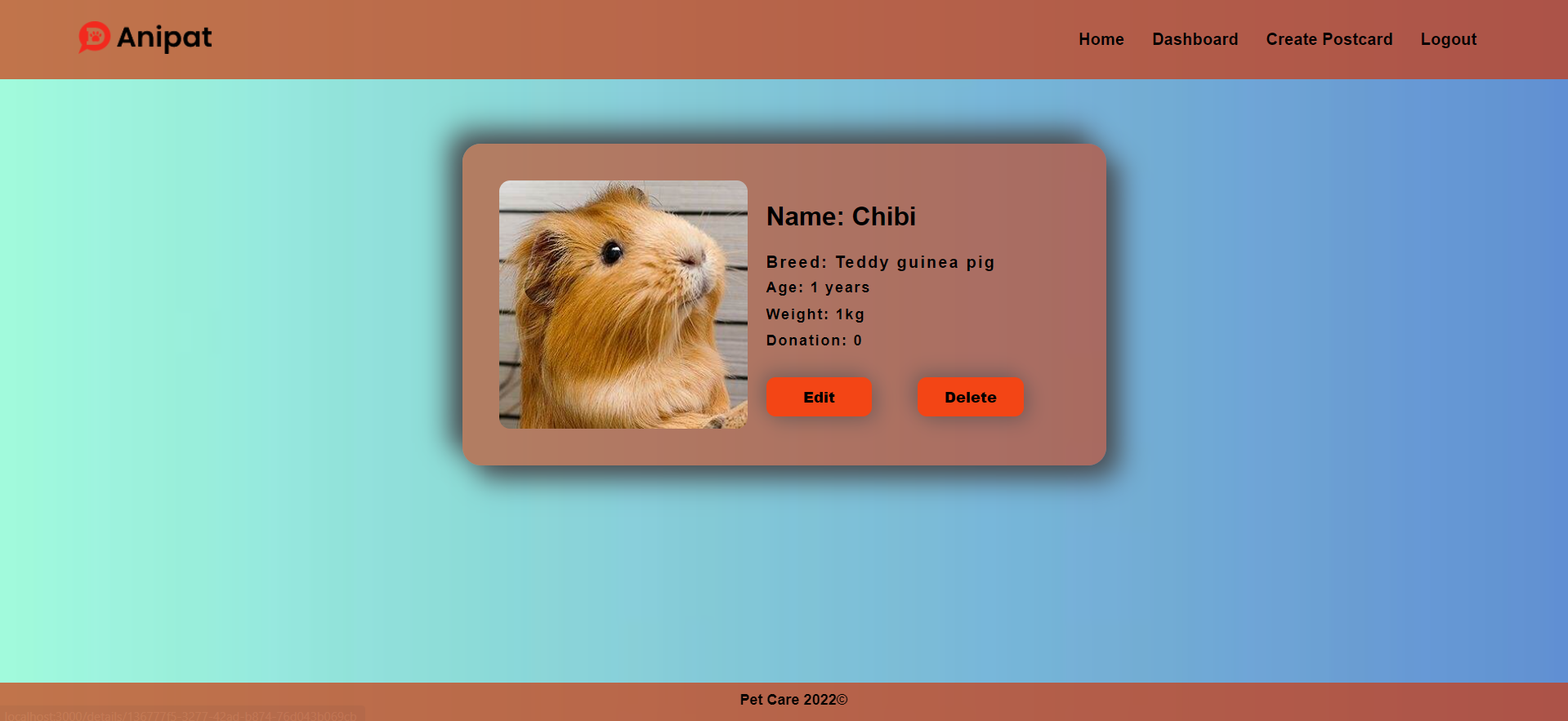
weight,

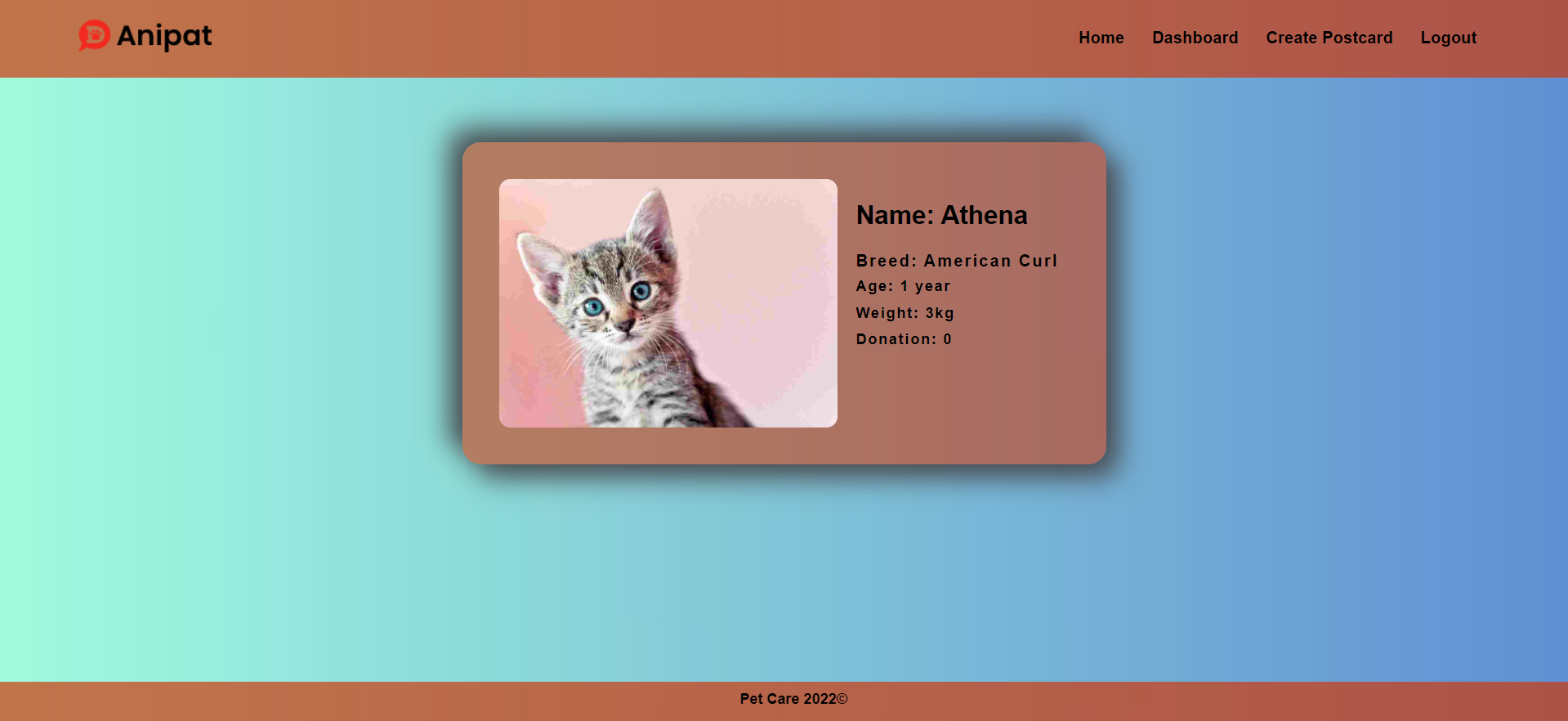
image

}

The service will return the newly created record. Upon success, **redirect** the user to the **Home** page.

### Details (10 pts)

Logged-in users should be able to **view details** aboutpets. Clicking the **Details** link in a **pet** should **display** the **Details** page. If the currently logged-in user is the **creator** of the pet postcard, the **Edit** and **Delete** buttons should be displayed. 



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



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

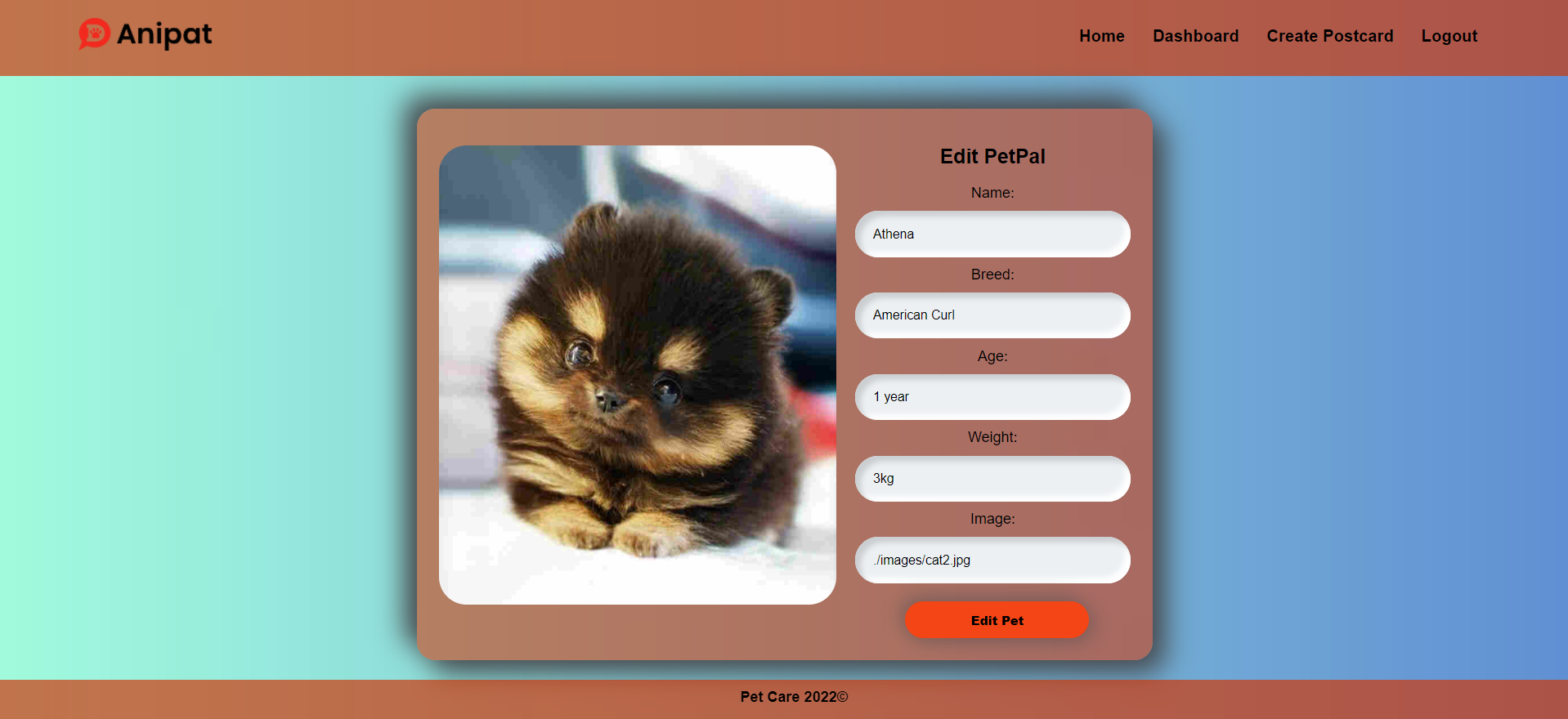
Method: GET

URL: /data/pets/*:id*

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

### Edit Page (15 pts)

The **Edit page** is accessible to logged-in users and allows the author to **edit** their pet postcard. Clicking the **Edit** button on the **details page** of a specific pet postcard should display the **Edit** page. It contains a form with input fields for all relevant properties. Make sure **all fields** are filled in before submitting the request. The fields must be filled in with the previous data when the page is first loaded.



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

Method: PUT

URL: /data/pets/*:id*

Where ***:id*** is the **id** of the desired pet postcard.

The service expects a body with the following shape:

{

name,

breed,

age,

weight,

image

}

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.

### Delete Article (10 pts)

The **Delete** action is available to logged-in users, for pets they have created. When the author clicks on the Delete action on any of their pets, a confirmation dialog should be displayed, and upon confirming this dialog, the pet should be deleted from the system.

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

Method: DELETE

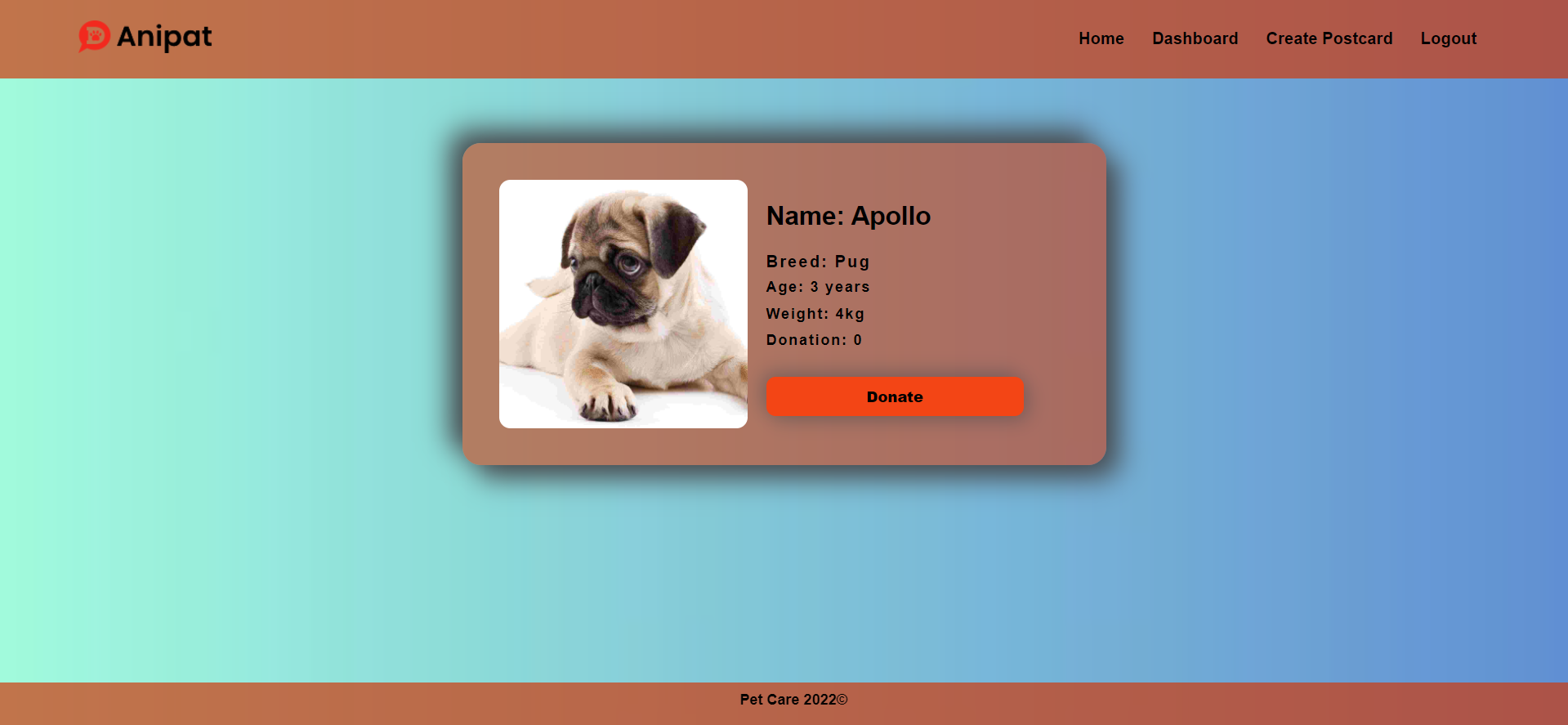
URL: /data/pets/*:id*

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

### (BONUS) Donate Action (15 pts)

Every logged-in user should be able to **donate to other pets**, but **not his own**. By clicking on the **[Donate]** button, **the donation of each pet increases by 100**. ( You need to multiply the counter of each pet donation by 100 )

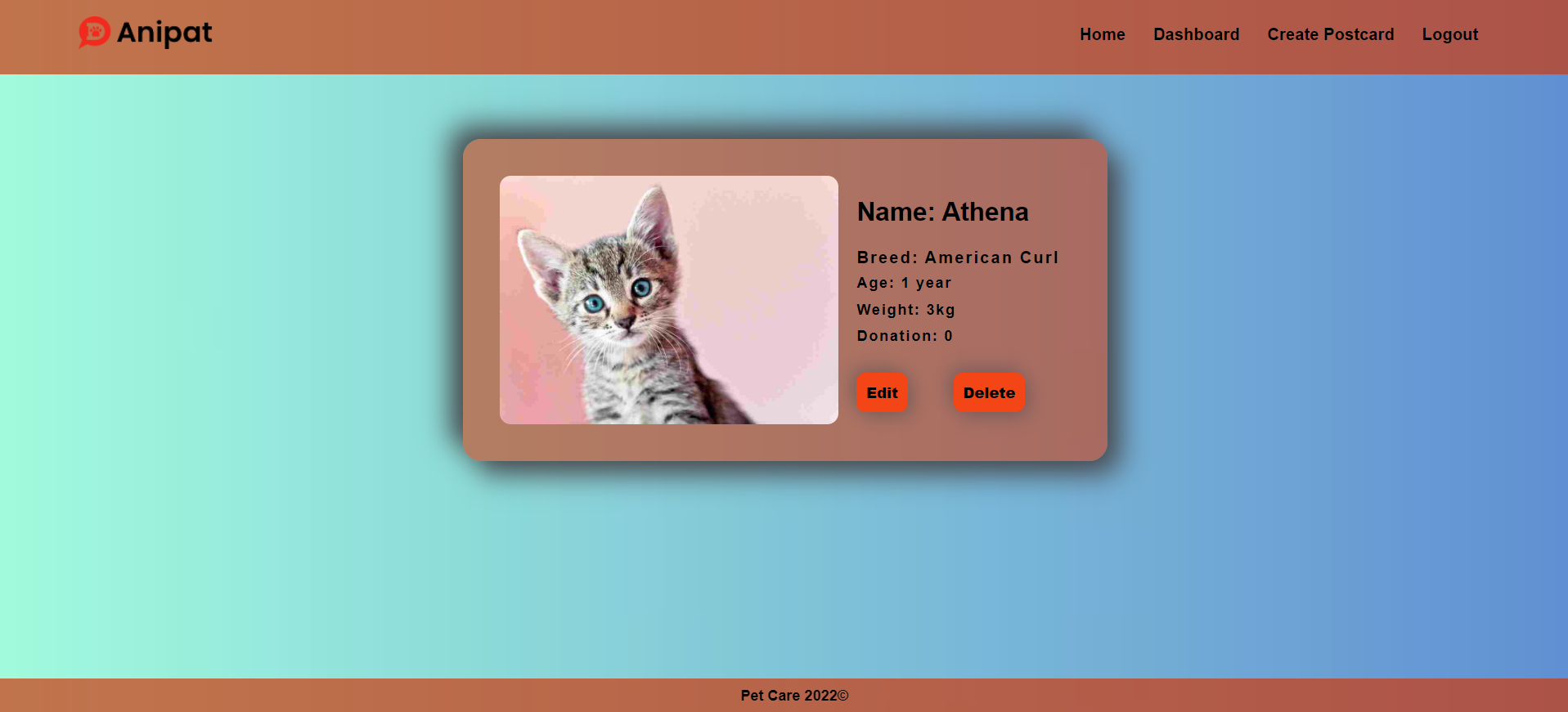
The view when the user did not press the **[Donation]** button should look like this:



When the user **Donation** for **pet** the **[Donation]** button should not be available and the counter should be increased by 100.



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



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



Send the following **request to add donation:**

Method: POST

URL: /data/donation

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

{

petId

}

Required **headers** are described in the documentation. The service will return the newly created record.

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

Method: GET

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

Where ***{petId}*** is the **id** of the desired pet. 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 pet from a specific user**:

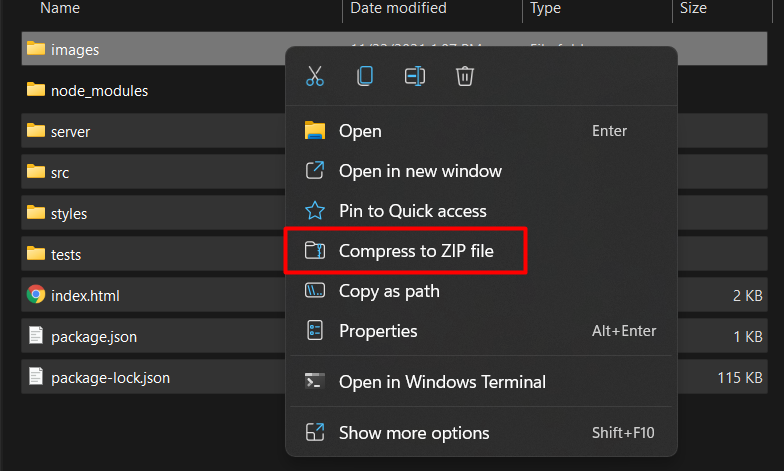
Method: GET

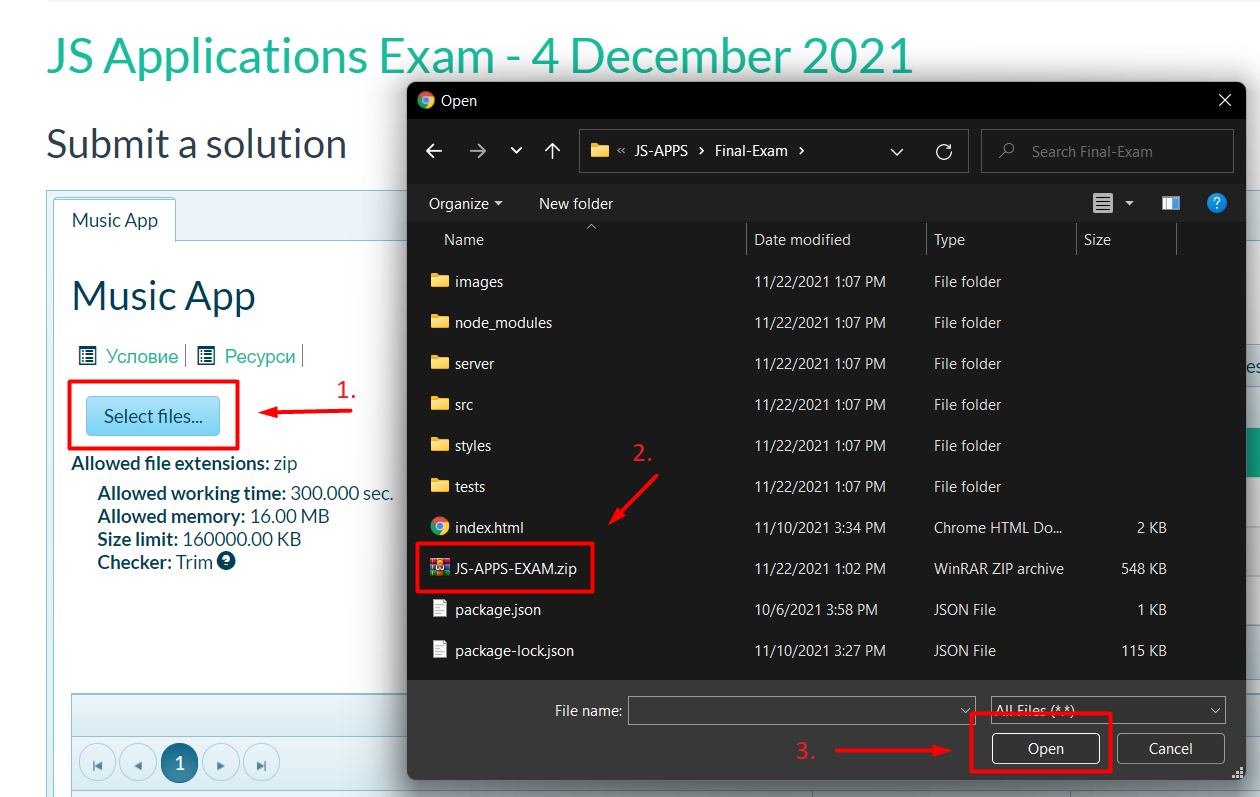
URL:/data/donation?where=petId%3D%22${petId}%22%20and%20\_ownerId%3D%22${userId}%22&count

Where ***{petId}*** is the **id** of the desired pet 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.







**It will take several minutes for Judge to process your solution!**



## 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 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 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-Contrl-Allow-Origin: \*

Content-Type: application/json

*{JSON-encoded response data}*

Empty response:

HTTP/1.1 204 No Content

Access-Contrl-Allow-Origin: \*

Error response:

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 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

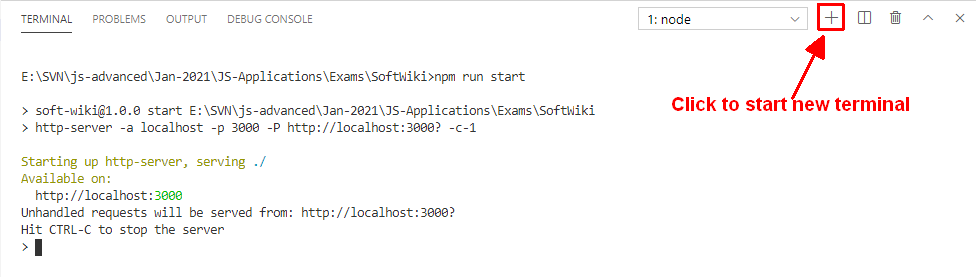
Before running the test suite, make sure a web server is operational, and the application can be found at the root of its network address. To start the included dev-server, open a terminal in the folder containing package.json and execute:

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 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 sees, and where it fails (or hangs):



If the actions are happening too fast, you can increase the value of slowMo. 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:

