

Cat Shelter

The exercise itself will be distributed into several parts each containing more concrete information and guide steps on how to develop the functionality specified below.

"Cat Shelter" is a very simple **cat catalog** that shows the "database" (JSON file) of some **cat shelter** and **everyone** (registration is not required) user can be their **potential owner**. The application will consist of the basic CRUD operations (**Create** cat, **Read** cat, **Update** cat and **Delete** cat). **Each cat** has a **name**, **description**, **image** and **breed**.

Project Specification

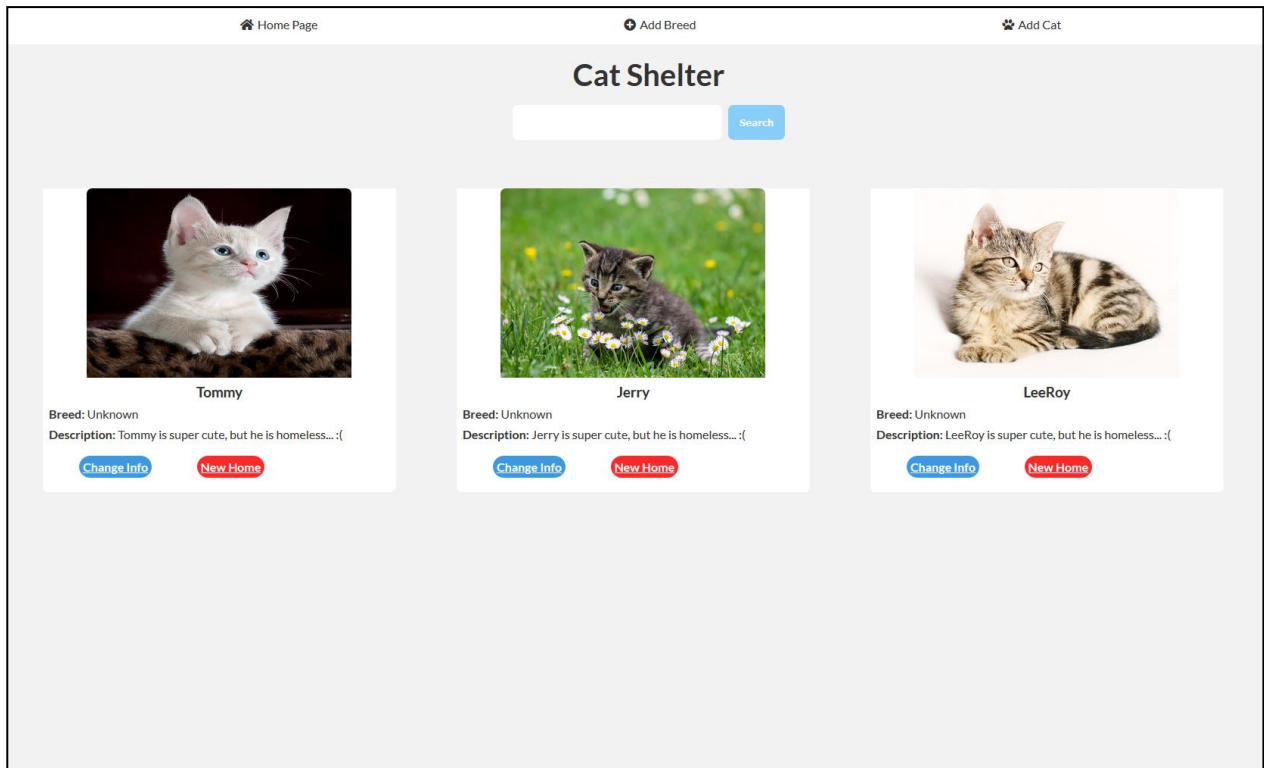
Design and implement a "**Cat Shelter**" **web application** (containing routing and multiple web pages) using HTML 5, CSS 3 and Node.js. It must contain the following functionality:

Functionality

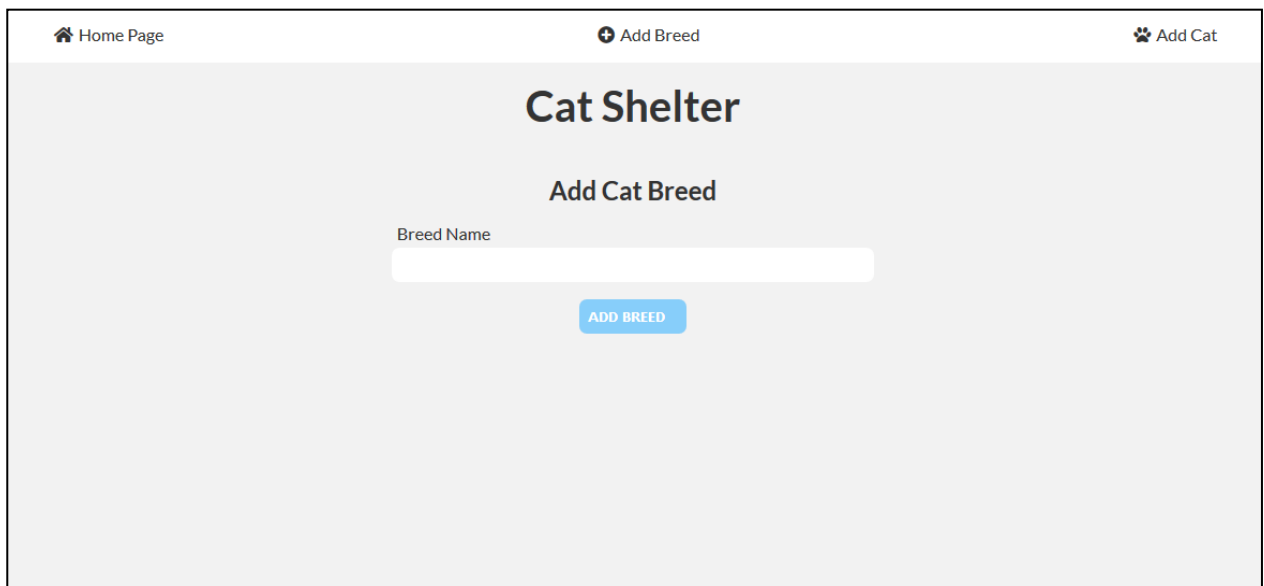
- **Add Cat Breed**
 - Create cat **breed** which later can be picked when a new cat is added to the shelter
- **Add Cat**
 - Create a new cat, which has a **name**, **description**, **image** and **breed**
 - **All cats** must be saved in a **JSON file** (that will be our database)
- **List All Cats**
 - List all cats from the "database" **no matter the breed**
- **Edit Cat**
 - Edits cat's information. Changes must be persisted in "database"
- **Delete Cat**
 - Deletes specific cat from database. Changes must be persisted in "database"
- **Search Cat**
 - Search in cat "database" by some **query string** and shows the results if any

Examples

- Home page ('/'), where **all** created **cats** in the database are **shown**.



- Add Breed (`'/addBreed'`), where a **new cat breed** can be created which later can be picked when a new cat is added to the shelter.



- Add Cat (`'/addCat'`), where the new cat can be created and stored in the shelter "database" and all created breeds before that are shown in the selected menu.

Home Page

Add Breed

Add Cat

Cat Shelter

Add Cat

Name

Description

Image

Разглеждане... Не е избран файл.

Breed

Bombay Cat

ADD CAT

Currently these 5 breeds are created (**Bombay Cat**, **American Bobtail Cat**, **Bengal Cat**, **British Shorthair Cat** and **Unknown**).

Home Page

Add Breed

Add Cat

Cat Shelter

Add Cat

Name

Description

Image

Разглеждане... Не е избран файл.

Breed

Bombay Cat

Bombay Cat

American Bobtail Cat

Bengal Cat

British Shorthair Cat

Unknown

Expected Behavior

Adding a new cat breed called "**Persian cat**".

[Home Page](#)[+ Add Breed](#)[Add Cat](#)

Cat Shelter

Add Cat Breed

Breed Name

Persian cat

ADD BREED

After that we create a new cat with the newly created breed: "Persian cat", name: "Niya", description: "Lonely and lazy cat seek for a hospitable owner" and imported image.

[Home Page](#)[+ Add Breed](#)[Add Cat](#)

Cat Shelter

Add Cat

Name

Description

Image

Разглеждане... Не е избран файл.

Breed

Bombay Cat

Bombay Cat

American Bobtail Cat

Bengal Cat

British Shorthair Cat

Unknown

Persian cat

Home Page
Add Breed
Add Cat

Cat Shelter

Add Cat

Name

Description

Image
 persianCat.jpg


Breed

ADD CAT

After clicking the **[ADD CAT]** button, a **redirect** should be followed and the Home page ('/') should be shown with all cats in the shelter, including the new one.


Home Page
Add Breed
Add Cat

Cat Shelter




Tommy

Breed: Unknown
Description: Tommy is super cute, but he is homeless... :(




Jerry

Breed: Unknown
Description: Jerry is super cute, but he is homeless... :(



LeeRoy

Breed: Unknown
Description: LeeRoy is super cute, but he is homeless... :(



Niya

Breed: Persian cat
Description: Lonely and lazy cat seek for a hospitable owner

For instance, if we click over **Jerry's [Change Info]** button, the following page should be shown.

[Home Page](#)[Add Breed](#)[Add Cat](#)

Cat Shelter

Edit Cat

Name
Jerry

Description
Jerry is super cute, but he is homeless... :(

Image
[Разглеждане...](#) Не е избран файл.

Breed
Unknown

[EDIT CAT](#)

We change Jerry's **name** and **breed**.

[Home Page](#)[Add Breed](#)[Add Cat](#)

Cat Shelter

Edit Cat

Name
Jerry Jr

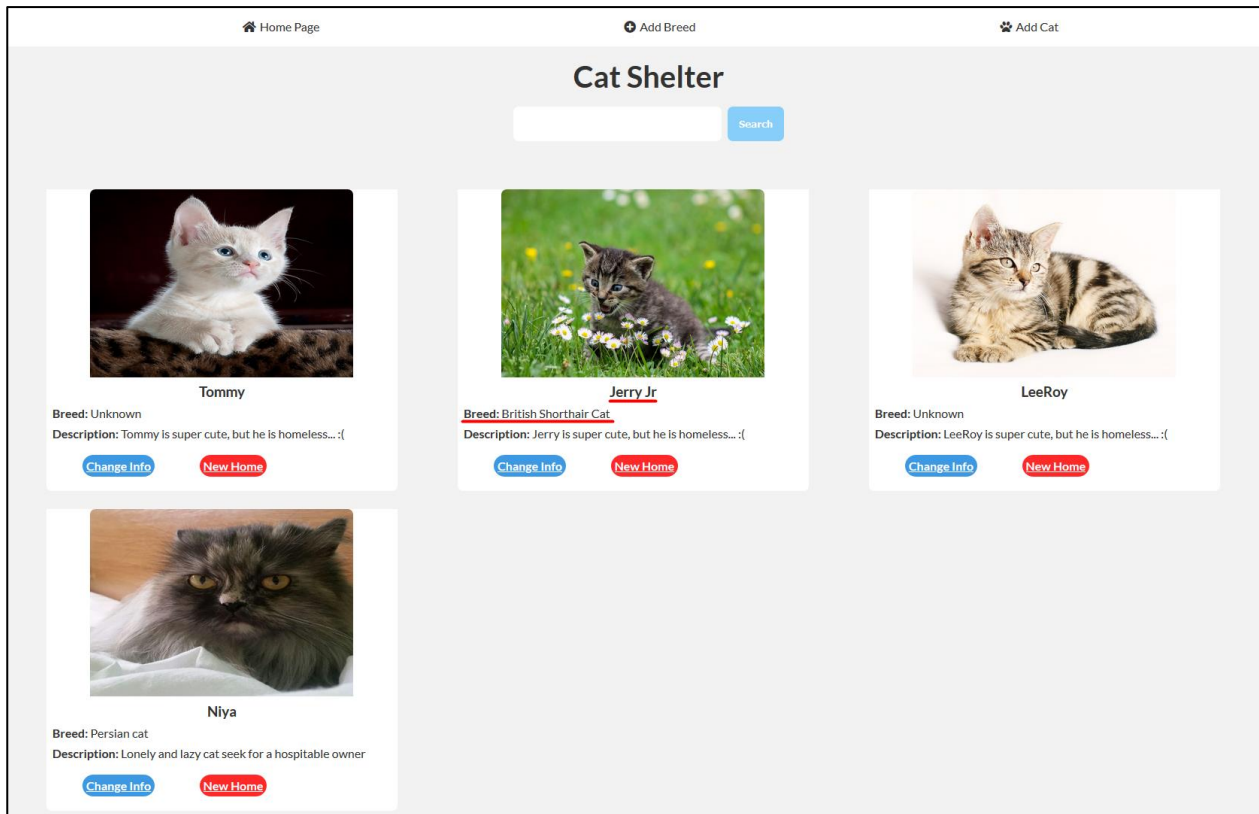
Description
Jerry is super cute, but he is homeless... :(

Image
[Разглеждане...](#) secondCat.jpg

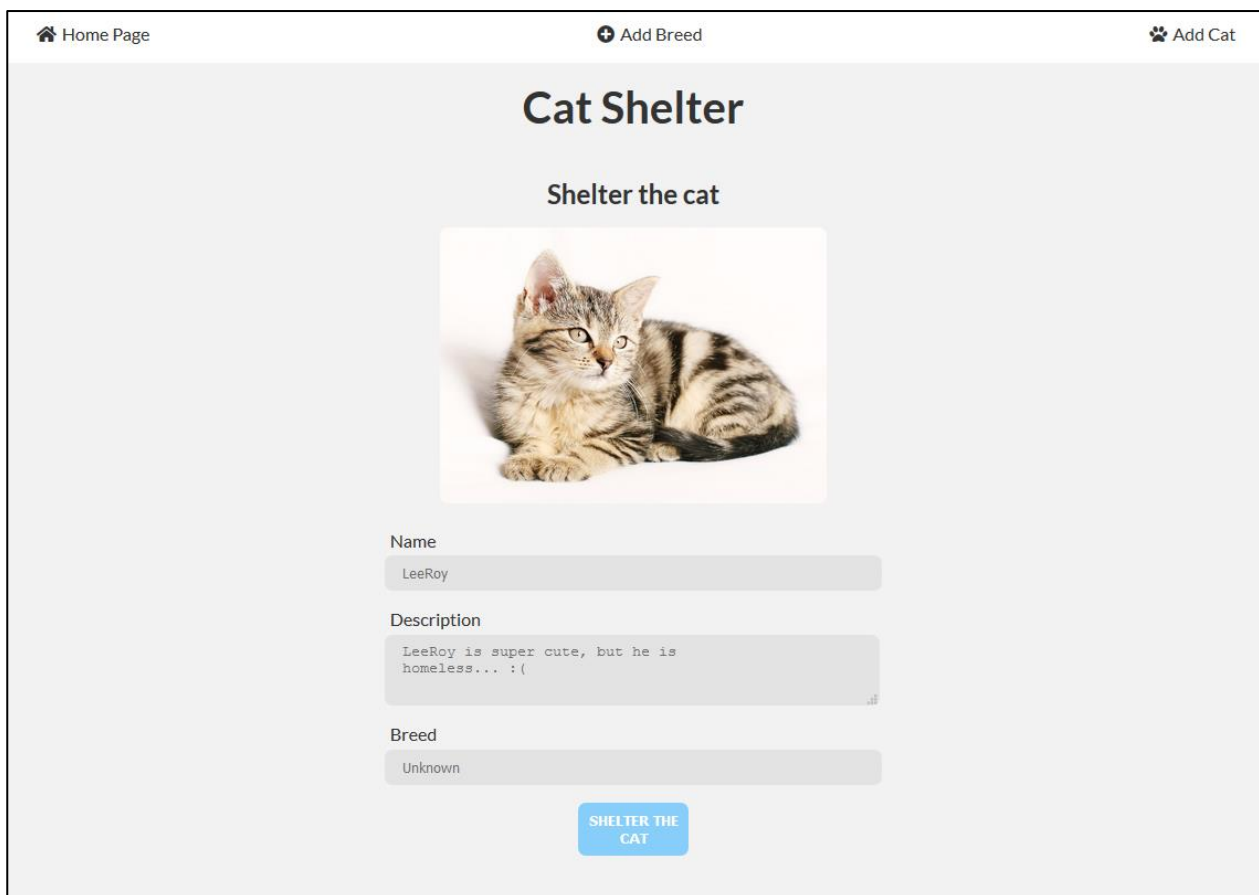
Breed
British Shorthair Cat

[EDIT CAT](#)

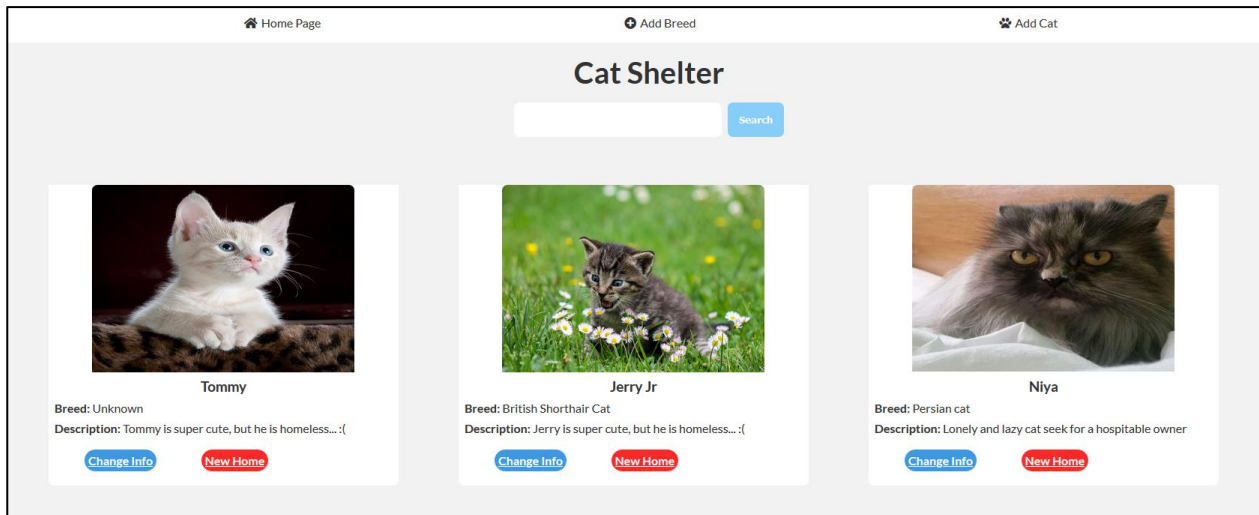
After clicking **[EDIT CAT] button**, the redirect to the Home page ('/') should follow and the changes should be applied.



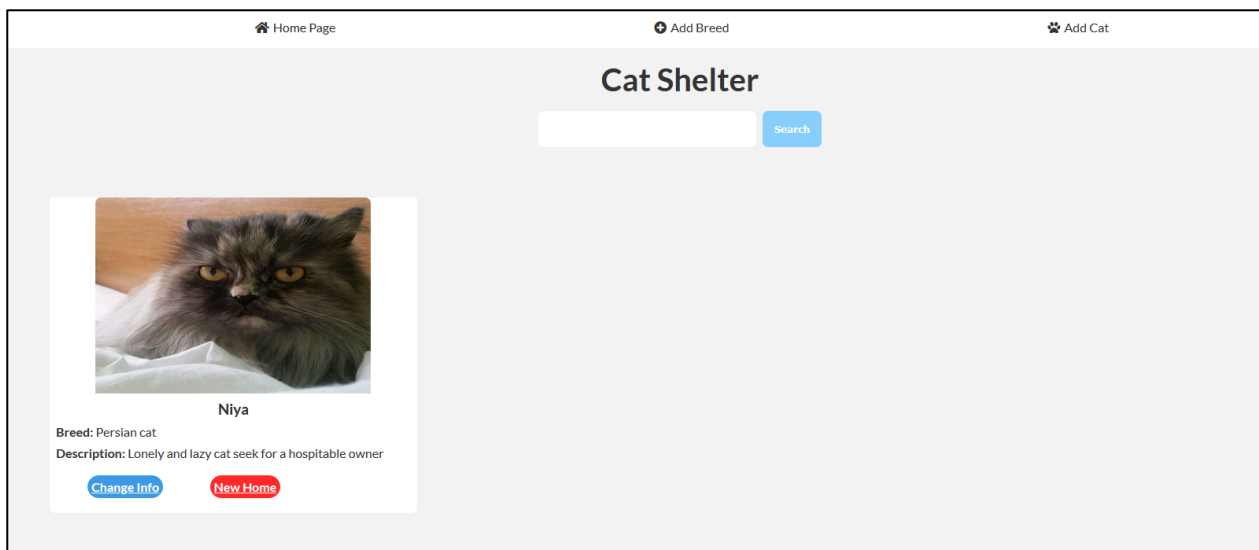
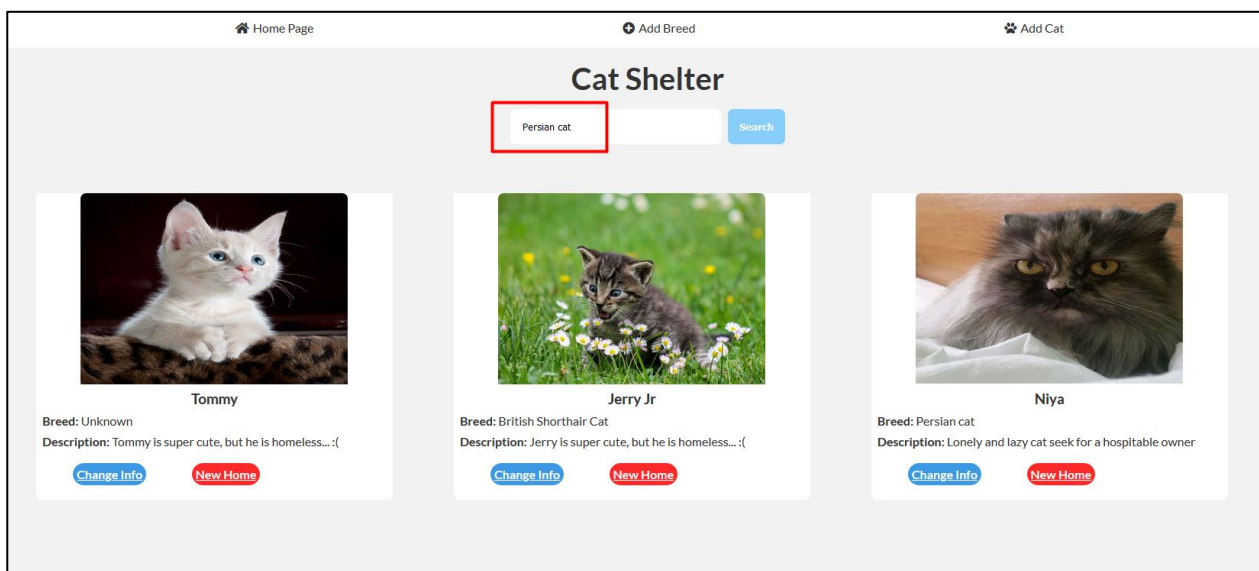
Now if we click over LeeRoy's **[New Home]** button the following page should be shown.



If we click over **[SHELTER THE CAT]** button, a Home page ('/') should be shown by redirection and the LeeRoy should be **gone** because someone offered him shelter and he became a pet.



If we use the search bar to check all Persian cats, the result should be:



Routing

There will be different views that are displayed based on the routing (the URL). There are no strict rules of how the current application routing should look like but there are some common set of rules to follow:

1. Use **short** and **clear** URL's
 - a. Good examples
 - `"/cats/add"`
 - `"/cats/edit/3 "`
 - b. Bad examples
 - `"/catAddInfo.html"`
 - `"showAllProductsByCategoryName/{categoryName}"`
2. Follow the **HTTP standards**
 - a. Proper usage of HTTP methods (GET, POST, PUT etc.)
 - b. Follow the GET – POST – Redirect pattern

Models

Cat

- **id** - **string or number** which is **required** and **unique**
- **name** - **string** which is required
- **description** - **string** containing some additional information about the cat
- **image** - **string** containing **reference** to an **image** that displays the given cat
- **breed** - **string** referring to the real breed on that cat

Laying the Project Fundamentals

1. Setup IDE

For starters configure your IDE or text editor. As mentioned above the following steps will include screenshots from [Visual Studio Code](#).

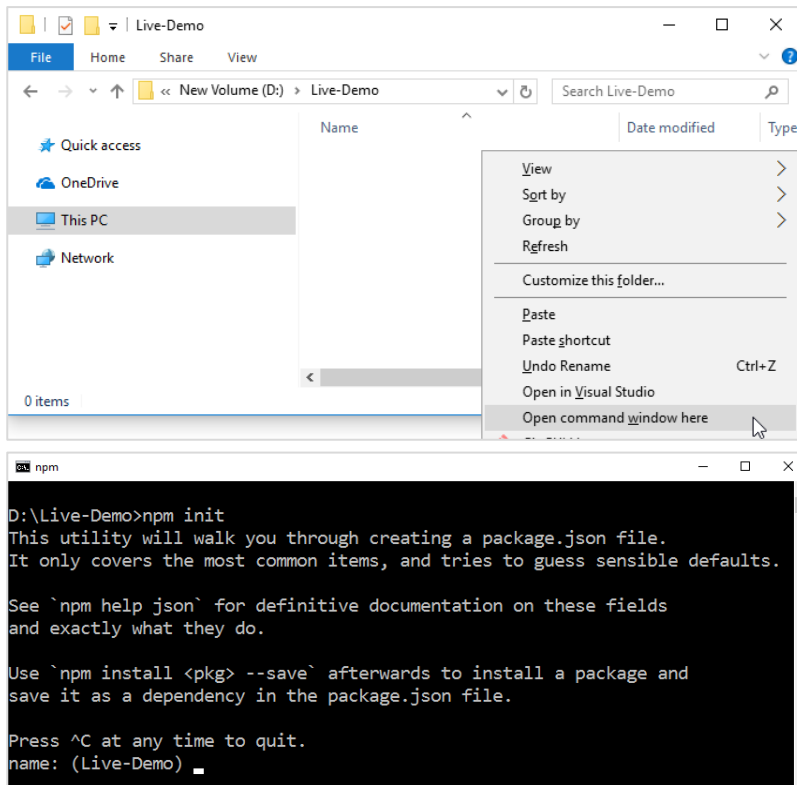
Other popular choices are [Atom](#) and [WebStorm](#).

Before you continue with the next step make sure everything about your IDE is configured and you are up and ready to go.

2. Initial Setup

First, let's create our project.

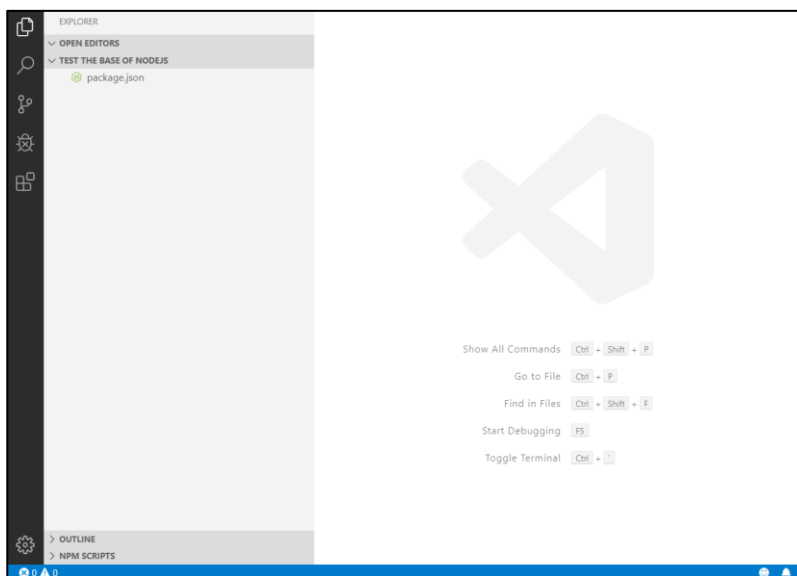
Go to the folder where you want the project to be, then press *"Shift" + "Right mouse click"* and use *"Open command window here"*, then type *"npm init"* and fill the project's data as you see fit.



If you are using **Visual Studio Code**, then in the same console (after you have inserted all the needed information) type: **"code ."** -> This will open the editor for you.

```
C:\WINDOWS\system32\cmd.exe
"\"description": "",
"main": "index.js",
"scripts": {
  "test": "echo \\\"Error: no test specified\\\" && exit 1"
},
"author": "GS",
"license": "MIT"
}

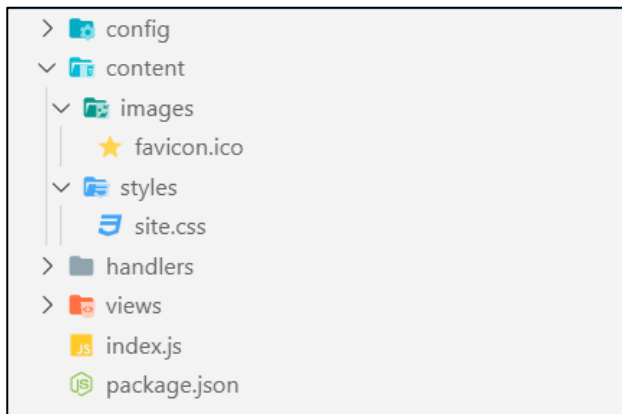
Is this ok? (yes) y
D:\Live-Demo>code .
```



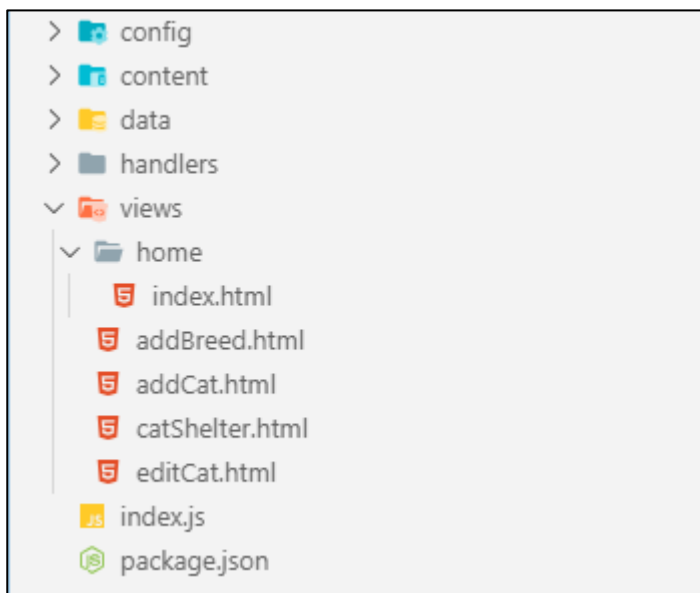
Now let's create the initial project folder structure. It may look like this:



Download the [resources](#) and put the **site.css** and **favicon.ico** in the "**content**" folder like this:



And move all **views** into the **views** folder like this:



Start the Server

Go to the `index.js` file and start implementing the server.

First, you should create **two constants**, one for the **http module** and another one for the **port** we will use for our server.

```

JS index.js > ...
1  const http = require('http');
2  const port = 3000;
3

```

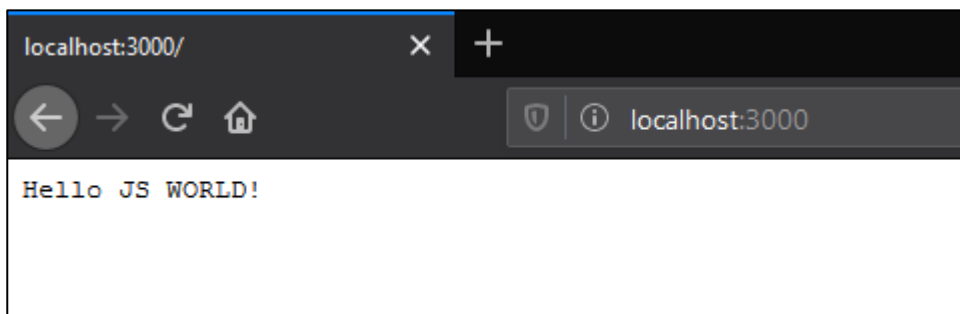
Once we require our http module, we can create our server via the **createServer** function.

```

7  http.createServer((req, res) => {
8      res.writeHead(200, {
9          'Content-Type': 'text/plain'
10     });
11
12     res.write('Hello JS WORLD!');
13     res.end();
14
15 }).listen(port);

```

When you do that, just open the Terminal with (**Ctrl + `**) and run the server with **node index** command.



Display the Home Page

We will start implementing the logic for application in the following steps. Note that the application doesn't have users.

First, whenever we access our site, we want to display the "default" page or so-called "**home**" page. In order to do that we have to:

1. Implement back-end logic of what to be displayed
2. Write our view (the HTML and CSS)
3. Make the server execute the logic we implement in the first step

Now go to "**handlers**" folder and add new "**home.js**". In the beginning add the modules we are going to use:

```

JS home.js  X
handlers > JS home.js > <unknown> > module.exports > index.on('data') callback
1  const url = require('url');
2  const fs = require('fs');
3  const path = require('path');
4

```

As we mention above, we will use **json files** to simulate a database, so create one folder called "**data**" and create 2 json files (**cats** and **breeds**).



And don't forget to include in into the home.js file. We will need them later, but lets include them anyway.

```
handlers > JS home.js > ...
1  const url = require('url');
2  const fs = require('fs');
3  const path = require('path');
4  const cats = require('../data/cats');
5
```

Then let's start with exporting the logic as a function which is accepting both request and response. Then we can parse the requested URL and attach it to the request object:

```
7
8  module.exports = (req, res) => {
9    const pathname = url.parse(req.url).pathname;
10
```

Now we should tell the server when the home handler will handle request (when the requested URL is "/" and the request method is "GET"):

```
11    if (pathname === '/' && req.method === 'GET') {
12      // Implement the logic for showing the home html view
13    } else {
14      return true;
15    }
16  }
17
18
19 }
```

If we could not handle the current request, we will notify the server of that by returning true (is request not handled - true)

What is left is to find the HTML5 page read it and send it as a response – here is how it could be done:

Hints

Inside if statement above we should locate the local index.html file (our home page) and store the path to it into a variable called **filePath** for instance. Use **path.normalize** and **path.join** functions and **__dirname** property to achieve that.

```
11     let filePath = path.normalize(  
12         path.join(__dirname, '../views/home/index.html')  
13     );
```

Use the fs module to read the HTML file using the **readFile** function with the given **filePath**. If an error occurs send a **404 response** code and some **plain text message**. If the specified file path name is correct send the **HTML** as a response with code **200** and content type **"text/html"**.

```
14     fs.readFile(filePath, err, data => {  
15         if (err) {  
16             console.log(err);  
17             res.writeHead(404, {  
18                 'Content-Type': 'text/plain'  
19             });  
20             res.writeHead(404, 'Not Found');  
21             res.end();  
22             return;  
23         }  
24         res.writeHead(200, {  
25             'Content-Type': 'text/html'  
26         });  
27         res.writeHead(data);  
28         res.end();  
29     });
```

Our html page (home page) is almost ready (later the cats will be added). We are almost ready to test if everything mentioned above is about to work.

Go to the **"handlers"** folder and add **"index.js"** file.

OPEN EDITORS

TEST THE BASE NODEJS

config

content

data

handlers

home.js

index.js

views

index.js

package.json

handlers > index.js > ...
1 const homeHandler = require('./home');
2
3 module.exports = [homeHandler];

This **index.js** file inside the "**handlers**" folder will be our file which exports all future handlers which we will create (**static file handler** and **cat handler**).

After that go back to the **root folder** and open the **index.js** (that file where we create our http server).

As we mentioned above, the **index.js** file inside the "**handlers**" folder will export **all handlers** to the "open world". That's why we should require them in this **index.js** file. Like:

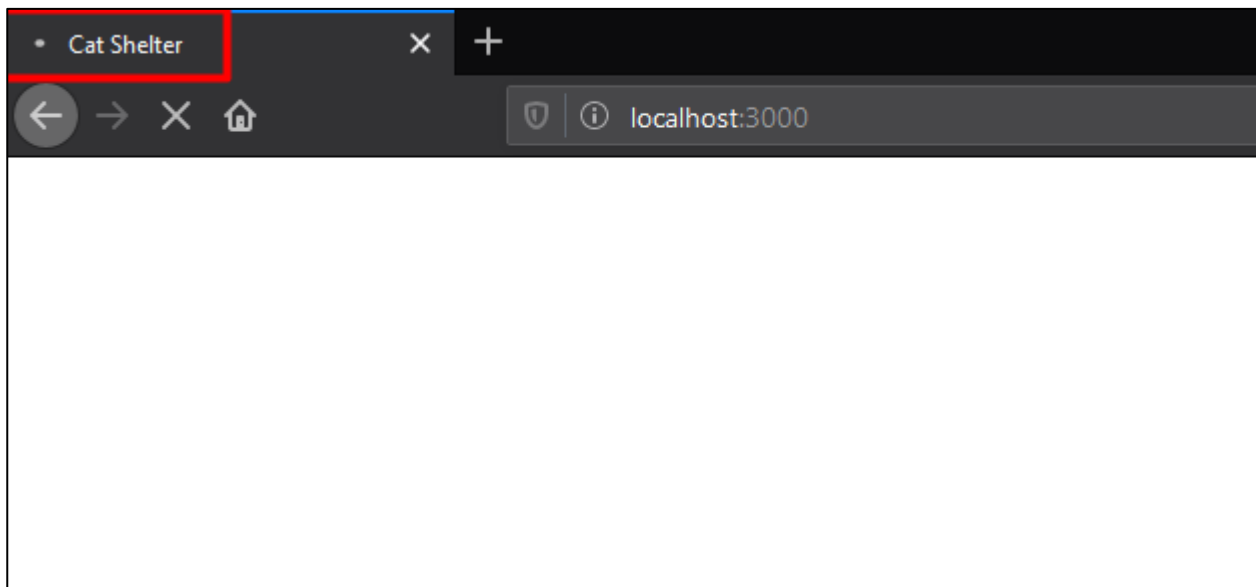
```
index.js > http.createServer() callback
1  const http = require('http');
2  const port = 3000;
3  const handlers = require('./handlers');
4
```

And loop through all handlers and if the right handler is found break the loop.

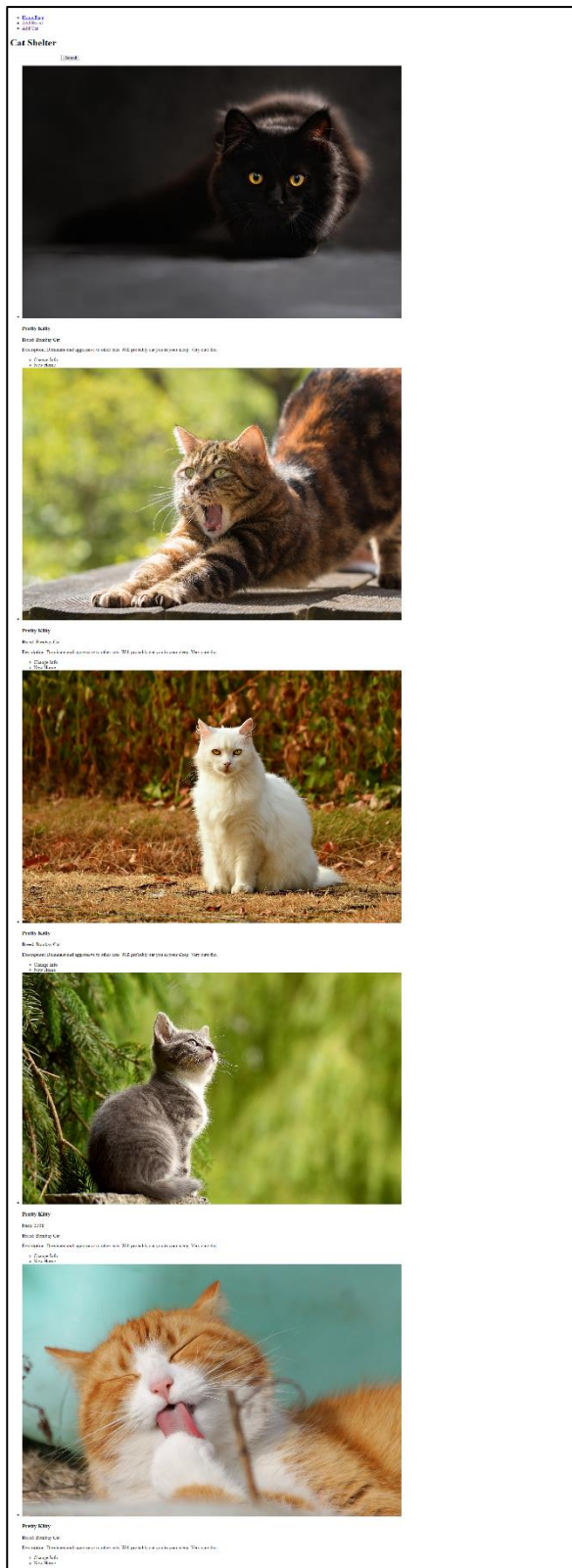
```
5  http.createServer((req, res) => {
6
7      for (let handler of handlers) {
8          if (!handler(req, res)) {
9              break;
10         }
11     }
12
13 }).listen(port);
```

Start the server again and type **localhost:3000** in the browser to see the result.

Sadly, the result will be **waiting on the localhost...**



Stop the page loading by clicking on the **[X]** button.



Our **index.html** file but **without any styles**, if you open the **network tab** inside **dev tools** and refresh the page the result will be something like this:

Состояние	Метод	Домен	Файл	Основание	Вид	Пренесени	Големина	Друг
200	GET	localhost:3000	/	document	html	4,24 KB	4,11 KB	1 KB
200	GET	localhost:3000	site.css	stylesheet		0B	0B	
200	GET	use.fontawesome.com	all.css	stylesheet	css	9,11 KB (raced)	34,53 KB	
200	GET	cdn.pixabay.com	cat-694730_1280.jpg	img	jpeg	87,51 KB (raced)	86,91 KB	
200	GET	cdn.pixabay.com	cat-614952_1280.jpg	img	jpeg	233,13 KB (raced)	232,53 KB	
200	GET	cdn.pixabay.com	cat-2083492_1280.jpg	img	jpeg	237,06 KB (raced)	236,46 KB	
200	GET	cdn.pixabay.com	cat-323262_1280.jpg	img	jpeg	200,66 KB (raced)	200,06 KB	
200	GET	cdn.pixabay.com	cat-3591346_1280.jpg	img	jpeg	буфериран	337,22 KB	
200	GET	localhost:3000	pawprint.ico	img	x-icon	4,02 KB (raced)	4,02 KB	

In other words, the server doesn't have the functionality to serve static files. We will take care of that in the next section.

3. Serve Static Files

In this step we will serve static files. In other words, load **css**, **js** and **image** files.

Let's begin by adding the back-end logic. In "**handler**" folder add a new file "**static-files.js**". It will behave like a normal handler but instead of returning html it will return file(s). Our public folder will be the "**content**":

First, let's create one function called **getContentType** which will receive our **pathname** (url), checks the **file extension** and **returns** the **correct content-type**. Like:

```

5  function getContentType(url){
6
7      if(url.endsWith('css')){
8          return 'text/css';
9      } else if('TODO...'){
10         //TODO...
11     }
12 }
```

As it's shown, just continue the logic to wrap the other extensions like (**html**, **png**, **js** etc...).

After that you should **export** a **function** which will receive our so familiar **request** and **response** parameters and checks the **pathname** and the **request** method:

```

20  module.exports = (req, res) => {
21      const pathname = url.parse(req.url).pathname;
22
23      if (pathname.startsWith('/content') && req.method === 'GET') {
24
25          //TODO...
26
27      } else {
28          return true;
29      }
30  };
```

Implement the missing logic.

1. **Read a file** via **readFile** function from the **file system (fs)**
2. Check for errors
3. Deliver the correct content type
4. **Send** the correct **response** with the **received data** from the fs module

```

25     fs.readFile(`${pathName}`, 'utf-8', (err, data) => {
26         if (err) {
27             console.log(err);
28
29             res.writeHead(404, {
30                 'Content-Type': 'text/plain'
31             });
32
33             res.write('Error was found');
34             res.end();
35             return;
36         }
37
38         console.log(pathName);
39         res.writeHead(
40             200,
41             { 'Content-Type': getContentType(pathName) }
42         );
43
44         res.write(data);
45         res.end();
46     });
47

```

This will work for now, but when we load the cat images locally, we should make some changes here...

Go back to "**handlers/index.js**" and add the static file handler:



```

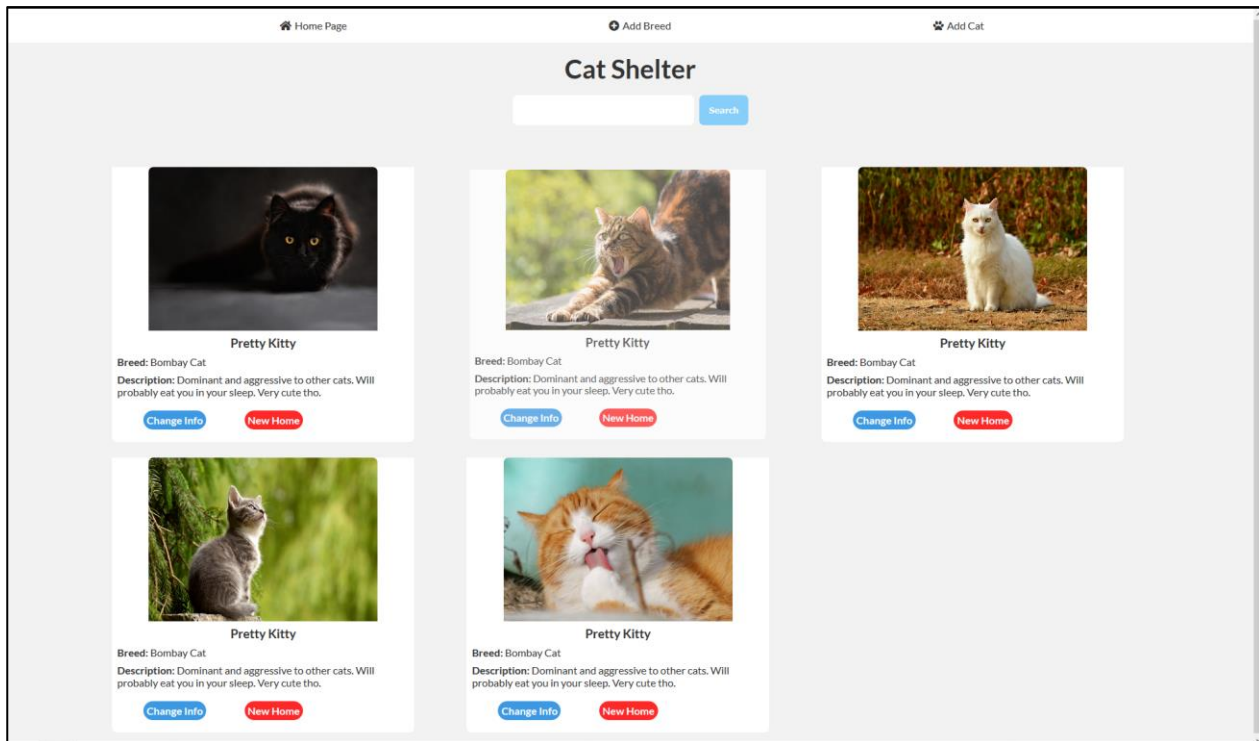
handlers > index.js > ...
1  const homeHandler = require('./home');
2  const staticFiles = require('./static-handler');
3
4  module.exports = [homeHandler, staticFiles];

```

If you have not included the "**site.css**" file and the "**favicon.ico**" in "**home/index.html**" go back and do it.

Note: Make sure that all **href** attributes inside the HTML start with **"/content/..."**

Start the web application again:



4. Implement "Database"

As we mentioned early, we will use **json files**. So, create a new folder called **"data"** (if you haven't done that already) and add **two** json files in there (**cats** and **breeds**). In the beginning, they will be an **empty array** (both files) but when we start creating **cats** and **breeds**, we will fill them.



Accessing All Views

A new handler should be created for the cat logic.

Create "**cat.js**" inside the "**handlers**" folder. The handler will be responsible for displaying the (html) form **or** parsing the data from it and add a new cat inside the **json** file:

First, include all needed libraries:

```
handlers > cat.js > ...
1  const url = require('url');
2  const fs = require('fs');
3  const path = require('path');
4  const qs = require('querystring');
5  const formidable = require('formidable');
6  const breeds = require('../data/breeds');
7  const cats = require('../data/cats');
8
```

This time they are quite a lot, but don't worry, we will use every one of them at a specific moment.

Feel free to check the official [Node documentation](https://nodejs.org/en/docs/guides/best-practices/#require-all-needed-dependencies) to understand what each module does.

Now, the logic is similar to before. Using the if statement, you should check every single **pathname** and **request method** and load an HTML page or parse the incoming data.

Create the variable, which will be the parsed **request** and get the **url** current **pathname** and check the different cases.

```
11  module.exports = (req, res) => {
12
13      const pathname = url.parse(req.url).pathname;
14
15      if (pathname === '/cats/add-cat' && req.method === 'GET') {
16
17          // TODO ...
18
19      } else if (pathname === '/cats/add-breed' && req.method === 'GET') {
20
21          // TODO...
22
23      } else {
24          return true;
25      }
26
27  };
```

You already have all needed views. Use **readFile** or **createReadStream** function to read each of the HTML files (Check the difference between these two functions).

```

let filePath = path.normalize(path.join(__dirname, '../views/addCat.html'));

const index = fs.createReadStream(filePath);

index.on('data', (data) => {
  res.write(data);
});

index.on('end', () => {
  res.end();
});

index.on('error', (err) => {
  console.log(err);
});

```

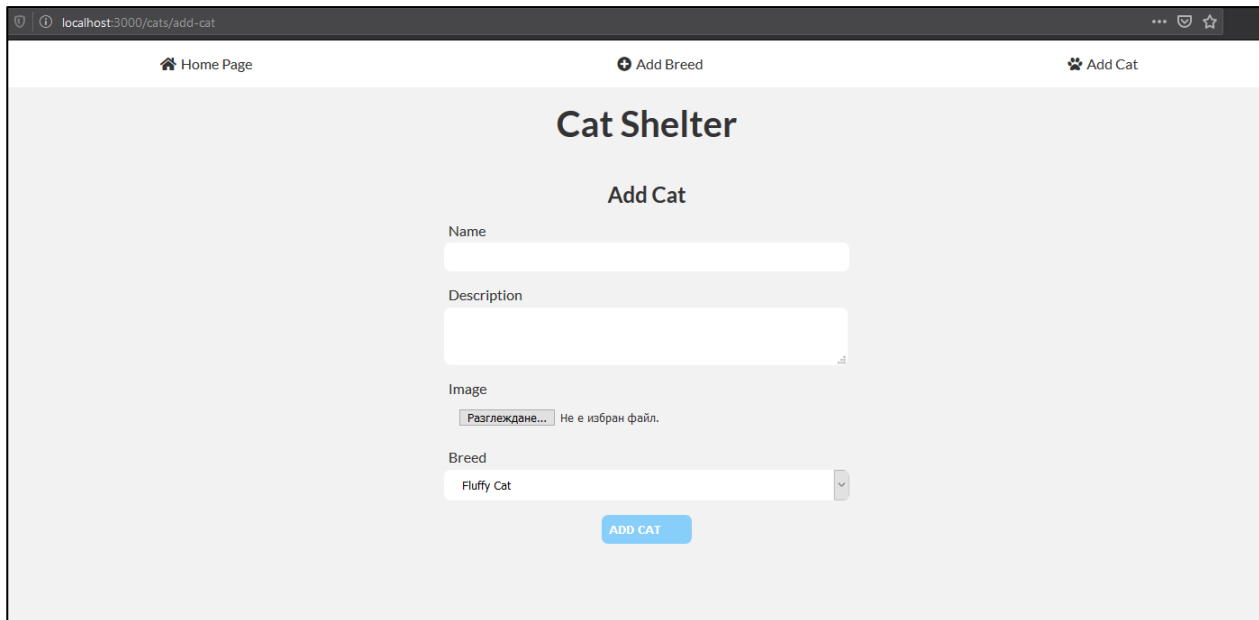
One more thing before we test our application – go to "**handlers/index.js**" and add our cat handler.

```

handlers > js index.js > ...
1  const homeHandler = require('./home');
2  const staticFiles = require('./static-handler');
3  const catHandler = require('./cat');
4
5  module.exports = [homeHandler, staticFiles, catHandler];

```

Restart the server and go to the **localhost:3000/cats/add-cat**



Handling Post

In order to create a new cat, first, we should create the cat breeds functionality, because they will be listed on the **addCat** page as **options elements** inside the **select menu**.

Handle the post request in **/cats/add-breed**, save the given breed inside the **breeds.json** file and when the **addCat.html** file is loaded we should use the breeds inside that **json** file, so they could be picked when a new cat is added to our application.

To handle **POST request**, the logic is similar:

```
15 > if (pathname === '/cats/add-cat' && req.method === 'GET') { ...
33 > } else if (pathname === '/cats/add-breed' && req.method === 'GET') { ...
49 > } else if (pathname === '/cats/add-breed' && req.method === 'POST') {
50 |     // TODO...
51 > } else if (pathname === '/cats/add-cat' && req.method === 'POST') {
52 |     // TODO...
53 > }
54 | else {
55 |     return true;
56 > }
57 |
58 > };
```

You have to:

1. **Parse** the **incoming data** from the **form**
2. **Read** the **breeds.json** file
3. **Modify** the **breeds.json** file
4. **Update** the **breeds.json** file
5. **Redirect** to the home page ('/') and **end** the **response**

```

51     let formData = '';
52
53     req.on('data', (data) => {
54         formData += data;
55     });
56
57     req.on('end', () => {
58
59         let body = qs.parse(formData);
60
61         fs.readFile('./data/breeds.json', (err, data) => {
62             if(err) {
63                 throw err;
64             }
65
66             let breeds = JSON.parse(data);
67             breeds.push(body.breed);
68             let json = JSON.stringify(breeds);
69
70             fs.writeFile('./data/breeds.json', json, 'utf-8', () => console.log('The breed was uploaded successfully!'));
71         });
72
73         res.writeHead(301, { location: '/' });
74         res.end();
75     });
76

```

Here's an example of how it works:

breeds.json before

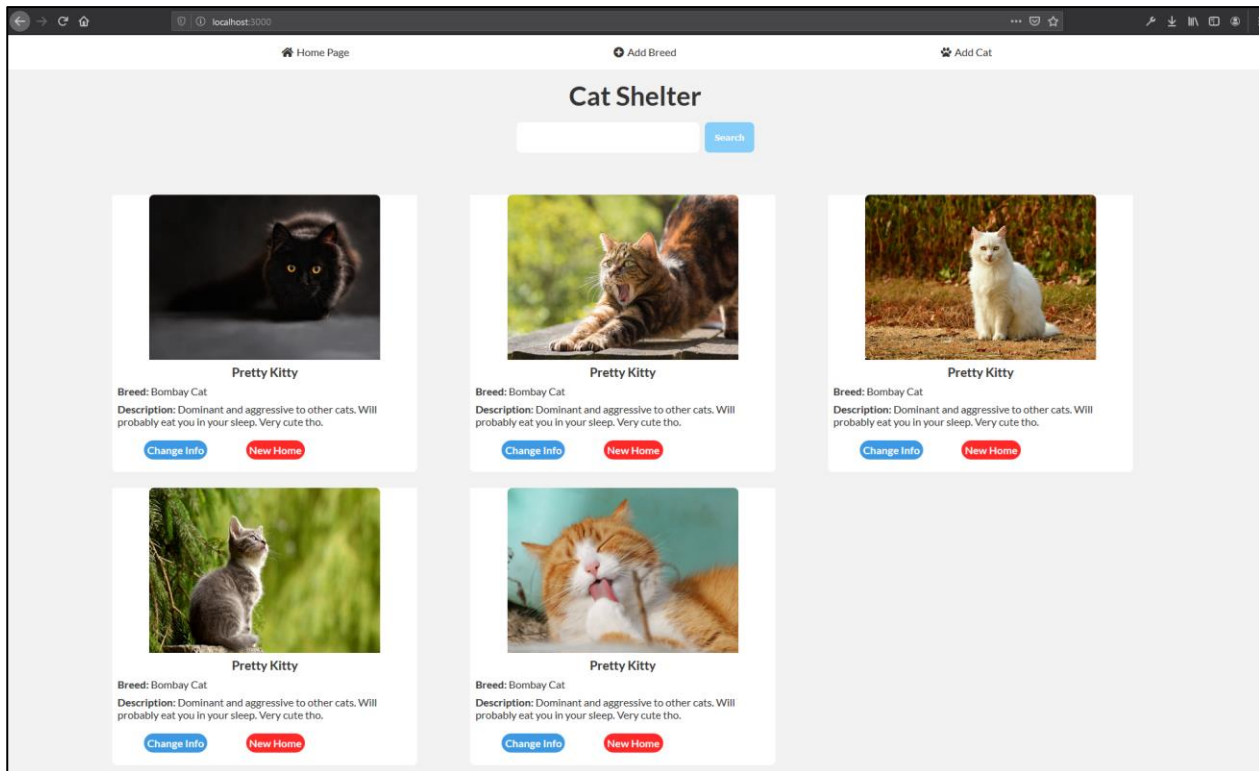
```

data > {} breeds.json
1  []

```

Adding a "Unknown breed"

If you got redirected to the home page, you have successfully implemented adding a breed.



Now if we check the **breeds.json** again, the result is:

```
data > { } breeds.json > ...
1  ["Unknown breed"]
```

Now every single breed which is created by the app is saved in this json file. The next step is to visualize all breeds from this json file, into the **select menu** as **options** in the **addCat.html** because now the Fluffy Cat Breed is just a placeholder.

Making Custom Templates

To continue, we must create **2 templates**. One for all created **breeds** inside the json file and one for all listed **cats** into the home page. Let's start with the simple one (breeds) because we can't create a new cat yet.

Here's the structure


```

<label for="group">Breed</label>
<select id="group">
  <option value="Fluffy Cat">Fluffy Cat</option>
</select>

```

Every breed should be an **option** which has a **value** with the current breed and **text content** also with the current breed. And all of them should be inside the **select element** with id **"group"**.

Now, when we render the **addCat.html** file we should **replace** some of the content inside the HTML. To do that we simply modify the **addCat.html** file to be like this:

```

views > addCat.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <link rel="stylesheet" href="../content/styles/site.css">
8    <link href="https://use.fontawesome.com/releases/v5.0.7/css/all.css" rel="stylesheet">
9    <title>Cat Shop</title>
10 </head>
11 <body>
12   <header>
13     <nav>
14       <ul class="navigation">
15         <li><a href="/">Home Page</a></li>
16         <li><a href="/cats/add-breed">Add Breed</a></li>
17         <li><a href="/cats/add-cat">Add Cat</a></li>
18       </ul>
19     </nav>
20     <h1>Cat Shelter</h1>
21   </header>
22   <form action="#" class="cat-form">
23     <h2>Add Cat</h2>
24     <label for="name">Name</label>
25     <input type="text" id="name">
26     <label for="description">Description</label>
27     <textarea id="description"></textarea>
28     <label for="image">Image</label>
29     <input type="file" id="image">
30     <label for="group">Breed</label>
31     <select id="group">
32       {{catBreeds}}
33     </select>
34     <button>Add Cat</button>
35   </form>
36 </body>
37 </html>

```

After that open the **cat.js handler** and open the statement where you render the **addCat.html** file.

Modify the data. Replace the **"{{catBreeds}}"** with the current **breed placeholder** like this:

```

23   let catBreedPlaceholder = breeds.map((breed) => `<option value="${breed}">${breed}</option>`);
24   let modifiedData = data.toString().replace('{{catBreeds}}', catBreedPlaceholder);
25

```

In this case, the **breed** is a variable that is **required** at the top of the file and refers to **breeds.json**.

We can replace/modify the data and pass it to the **res.write()** method as simple as sounds.

Now let's add another breed just to check if this logic is working fine:

And after we open the **Add Cat** Tab, you will see this time the options are the actual breeds inside the json file.

Voilà 😊

As you can guess we can use the same logic to render all created cats on the home page, but first you should make the logic where a new cat is added to our **cat.json** file via form.

But here comes the funny part, because add a new cat form contains the input where the type is **'file'**. That means this time we must process the incoming data differently.

We will use **formidable**. You can check how this library works, but simply you can use the following code structure:

```

82     } else if (pathname === '/cats/add-cat' && req.method === 'POST') {
83
84         let form = new formidable.IncomingForm();
85
86         form.parse(req, (err, fields, files) => {
87             // TODO ...
88         });
89     }

```

In this case, **fields** and **files** will be **objects**, where **fields** are the incoming data from the form and **files** is the information about the uploaded file via form.

```

36 form.parse(req, (err, fields, files) => {
37   if (err) throw err;
38
39   let oldPath = files.upload.path;
40   let newPath = path.resolve(path.join(globalPath, '/content/images/') + files.upload.name);
41
42   fs.rename(oldPath, newPath, (err) => {
43     if (err) throw err;
44     console.log('Files was uploaded successfully!');
45   });
46
47   fs.readFile('./data/cats.json', 'utf8', (err, data) => {
48     if (err) throw err;
49
50     let allCats = JSON.parse(data);
51     allCats.push({ id: cats.length + 1, ...fields, image: files.upload.name });
52     let json = JSON.stringify(allCats);
53     fs.writeFile('./data/cats.json', json, () => {
54       res.writeHead(200, { location: '/' });
55       res.end();
56     });
57   });
58 });

```

Use **rename()** function to change the location on the uploaded file.

In other words, you can **save it** somewhere in the local files. Just get the **old** and the **new path** on that file and pass them as arguments in that function.

And via **readFile()** and **writeFile()** functions in **fs module** you can get all cats inside **json.file**, modify them and write them back to the json file where the new cat will be included.

Here's the example:

This time if we open the **cats.json** file it won't be an empty array anymore.

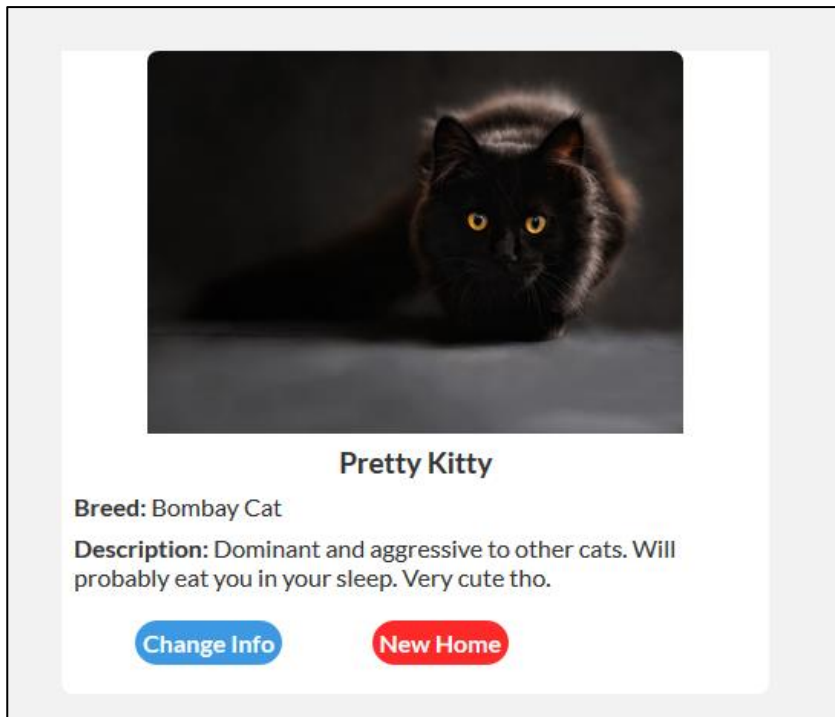
```

data > {} cats.json > ...
1  [{"id":1,"name":"New Cat","description":"New Description","breed":"Unknown breed","image":"luboCutted.png"}]

```

Now we already can create new cats. To fully check if the entire logic is working, you can check the **/content/images/**. If the uploaded picture from you is there, everything is fine. So, we can use the given template and render all cats inside the **cats.json file**.

Here is the cat structure:



```
</li>
  
  <h3>Pretty Kitty</h3>
  <p><span>Breed: </span>Bombay Cat</p>
  <p><span>Description: </span>Dominant and aggressive to other cats. Will probably eat you in your sleep. Very cute tho.</p>
  <ul class="buttons">
    <li class="btn edit"><a>Change Info</a></li>
    <li class="btn delete"><a>New Home</a></li>
  </ul>
</li>
```

When rendering the **index.html** page we should loop through all cats inside **cats.json** and use the placeholder from above to change and include them into the HTML file before it is rendered.

Here's how that work can be done:

Go to **home.js**, because there is the logic we should change. There we load **index.html** file but with static data. Change it to this:

```
27 let modifiedCats = cats.map((cat) => `<li>
28   
29   <h3>${cat.name}</h3>
30   <p><span>Breed: </span>${cat.breed}</p>
31   <p><span>Description: </span>${cat.description}</p>
32   <ul class="buttons">
33     <li class="btn edit"><a href="/cats-edit/${cat.id}">Change Info</a></li>
34     <li class="btn delete"><a href="/cats-find-new-home/${cat.id}">New Home</a></li>
35   </ul>
36   </li>`);
37 let modifiedData = data.toString().replace('{{cats}}', modifiedCats);
38
```

Loop through all cats inside the **cats.json** file and use the given from the skeleton placeholder to fill every single cat information into the right place. Don't forget to **write()** and **end()** the response.

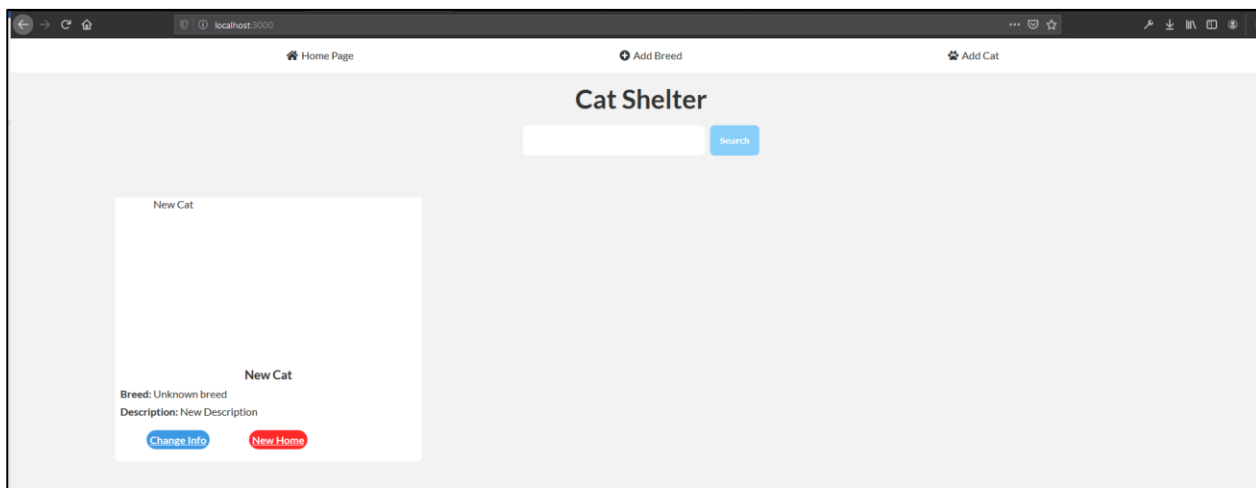
Don't forget to change the **index.html** as well.

```

views > home > index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <link href="https://use.fontawesome.com/releases/v5.0.7/css/all.css" rel="stylesheet">
8      <link rel="stylesheet" href="../../content/styles/site.css">
9      <link rel="shortcut icon" type="image/png" href="../../content/images/pawprint.ico"/>
10     <title>Cat Shelter</title>
11 </head>
12 <body>
13     <header>
14         <nav>
15             <ul class="navigation">
16                 <li><a href="/">Home Page</a></li>
17                 <li><a href="/cats/add-breed">Add Breed</a></li>
18                 <li><a href="/cats/add-cat">Add Cat</a></li>
19             </ul>
20         </nav>
21         <h1>Cat Shelter</h1>
22         <form action="/search">
23             <input type="text">
24             <button type="button">Search</button>
25         </form>
26     </header>
27     <main>
28         <section class="cats">
29             <ul>
30                 <li>{{cats}}</li>
31             </ul>
32         </section>
33     </main>
34 </body>
35 </html>

```

Now, if you restart the server and check the home page (**localhost:3000**), the result might be something like this:



The picture is not visualized on the page. The reason behind that is because when we serve static files, we don't think about all edge cases. To solve this problem, we should extend our **static-handler.js** logic.

```

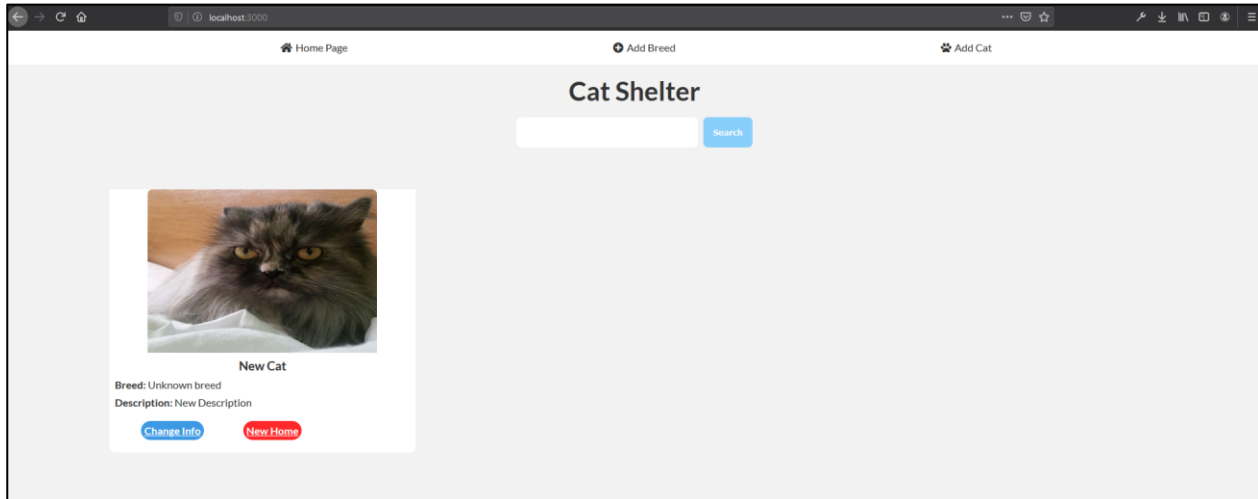
20 module.exports = (req, res) => {
21     const pathname = url.parse(req.url).pathname;
22
23     if (pathname.startsWith('/content') && req.method === 'GET') {
24
25         if (pathname.endsWith('.png') || pathname.endsWith('.jpg') || pathname.endsWith('.jpeg') || pathname.endsWith('.ico') && req.method === 'GET') {
26
27             fs.readFile(`./${pathname}`, (err, data) => { ...
28             });
29
30         } else {
31             fs.readFile(`./${pathname}`, 'utf8', (err, data) => { ...
32             });
33         }
34     } else {
35         return true;
36     }
37 }

```

This is the structure you should use to resolve this problem. The logic inside these two `readFile()` function is 99% the same. The only difference between them is the **encoding** which is the second argument and it's optional. When some of the following files are served, there should be no encoding or at least no **utf8** (**png**, **jpg**, **jpeg** etc...).

You should just **check** when the **pathname** starts with **'/content'** and the **request method** is **'GET'**, and also the **pathname ends with** some of the image extension, read the file, pass the **pathname** and the **callback** as arguments that's it. In any other cases, no matter the **pathname** ending you should put the encoding (**'utf8'**) for now. As it's done above in the picture.

If you do that the result will be:



5. Edit and Delete

[**Change Info**] and [**New Home**] are buttons that every cat should have. You have been given all the necessary views, including these two (**editCat.html** and **catShelter.html**) clicking over some of them, the app should show the current view with current cat info inside it.

So, the missing logic we should implement is:

```
111     } else if (pathname.includes('/cats-edit') && req.method === 'GET') {
112         // TODO...
113     } else if (pathname.includes('/cats-find-new-home') && req.method === 'GET') {
114         // TODO...
115     } else if (pathname.includes('/cats-edit') && req.method === 'POST') {
116         // TODO...
117     } else if (pathname.includes('/cats-find-new-home') && req.method === 'POST'){
118         // TODO...
119     }
120     }
121     else {
122         return true;
123     }
124 }
```

When some of the [**Change info**] buttons are clicked, you should check the **cat id** which is provided into the **url**, search that cat in the **cat.json** file and use templates to replace the static data with the current cat information.

If you check **home.js handler**, you will see this:

```
27 let modifiedCats = cats.map((cat) => `- 28   
29   <h3>${cat.name}</h3>
30   <p><span>Breed: </span>${cat.breed}</p>
31   <p><span>Description: </span>${cat.description}</p>
32   <ul class="buttons">
33     <li class="btn edit"><a href="/cats-edit/${cat.id}">Change Info</a></li>
34     <li class="btn delete"><a href="/cats-find-new-home/${cat.id}">New Home</a></li>
35   </ul>
36 </li>`);
37 let modifiedData = data.toString().replace('{{cats}}', modifiedCats);
38

```

This is our **cat template**. When you get the current cat from the **json.file**, you can use the logic from above to get the file and render it and just apply the template:

```
151 let modifiedData = data.toString().replace('{{id}}', catId);
152 modifiedData = modifiedData.replace('{{name}}', currentCat.name);
153 modifiedData = modifiedData.replace('{{description}}', currentCat.description);
154
155 const breedsAsOptions = breeds.map((b) => `
```

Also **don't forget** to change the **editCat.html** file as well:

```
<form action="/cats-edit/{{id}}" class="cat-form" enctype="multipart/form-data">
  <h2>Edit Cat</h2>
  <label for="name">Name</label>
  <input type="text" id="name" value="{{name}}">
  <label for="description">Description</label>
  <textarea id="description">{{description}}</textarea>
  <label for="image">Image</label>
  <input name="upload" type="file" id="image">
  <label for="group">Breed</label>
  <select name="breed" id="group">
    {{catBreeds}}
  </select>
  <button type="submit">Edit Cat</button>
</form>
```

The logic after clicking over **[New Home]** button is the same:

1. Check the **url** to get the current **cat id**
2. **Search** through **cat.json** file that id
3. **Replace** editCat.html with the current cat information like above

POST request for these two actions is almost the same as the logic for **adding a new cat**. With the simple difference when you **edit** some of the cat information you **change** that cat inside the json file instead of creating a new one (change it with the incoming information from the form). And **delete** the current cat from the shelter (**cats.json file**).

The steps are the same:

Parse the incoming data from the form (if you are editing the cat's information)

Read the **cats.json** file via **fs** module update the cats (edit the cat or deleting it)

Rewriting the **cats.json** file with the applied changes.

```

fs.readFile('data/cats.json', 'utf8', function readfileCallback(err, data) {
  if (err) {
    console.log(err);
  } else {
    let currentCats = JSON.parse(data); //now it an object
    let catId = req.url.split('/')[1];

    currentCats = currentCats.filter((cat) => cat.id !== catId); //add some data
    let json = JSON.stringify(currentCats); //convert it back to json
    fs.writeFile('data/cats.json', json, 'utf8', () => {
      res.writeHead(200, { 'location': '/' });
      res.end();
    }); // write it back
  }
});

```

6. Search*

Having a web page with cats is fun but sometimes you want to filter them by some criteria. Therefore, our home page includes a little form that has only one text field (cat's name or part of it). After submitting the form, all the cats containing such text or having the same name should be displayed.

GOOD LUCK 😊