11.4.7 Create Front-End Functionality for Zookeepers

Now we're going to add the front-end functionality for the zookeepers. You've already downloaded all of the front-end code in the previous lesson—you just need to add some more functionality and set up the routes so that they properly serve the HTML.

REWIND

Every time a user navigates to a page, a request is sent to the server for that route. Adding a link to `/zookeepers` makes the browser request a resource from that route on the server. In this case, an HTML file is sent as a response.

Take a look at the page served from `http://localhost:3001/` again. The HTML already has the form that we need to allow users to create new zookeepers! But is that functionality hooked up in `public/assets/js/script.js`?

Let's start by figuring out the ID of the zookeeper form. To find that, look in `index.html`, which is shown in the following code:

```
<form id="zookeeper-form">
  <label for="zookeeper-name" class="form-label">Zookeeper Name</label>
  <input type="text" id="zookeeper-name" class="form-input input" placeholde
    name="zookeeper-name">

  <label for="age-input" class="form-label input">Age</label>
  <input type="number" id="age-input" class="form-input" placeholder="Enter

  <label for="favorite-animal" class="form-label">Favorite Animal</label>
  <input type="text" class="form-input input" id="favorite-animal" placehold
    name="favorite-animal">

  <button class="btn">Add Zookeeper</button>
</form>
```

Did you find it? That's right—it's `zookeeper-form`.

Let's add a query selector at the top of `public/assets/js/script.js`, like this:

```
const $zookeeperForm = document.querySelector('#zookeeper-form');
```

Now we just need to create a new function to handle the zookeeper creation to `script.js`, just as we've done for animals. Add the following code:

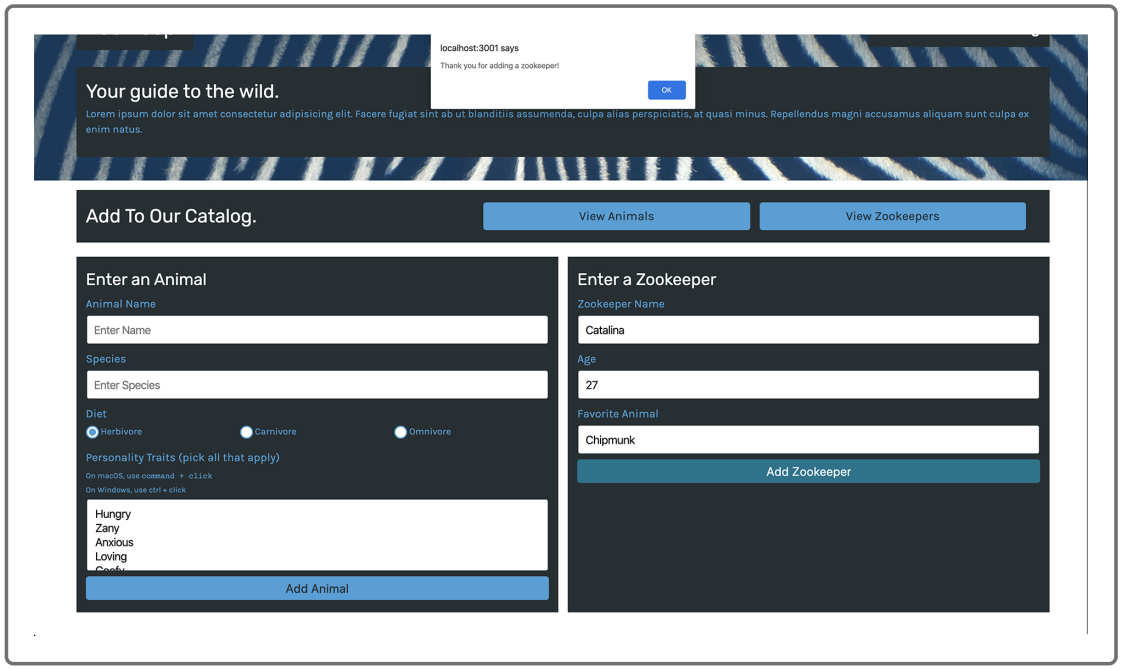
```
const handleZookeeperFormSubmit = event => {
  event.preventDefault();

  // get zookeeper data and organize it
  const name = $zookeeperForm.querySelector('[name="zookeeper-name"]').value
  const age = parseInt($zookeeperForm.querySelector('[name="age"]').value);
  const favoriteAnimal = $zookeeperForm.querySelector('[name="favorite-anima
```

```
  const zookeeperObj = { name, age, favoriteAnimal };
  console.log(zookeeperObj);
  fetch('api/zookeepers', {
    method: 'POST',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(zookeeperObj)
  })
    .then(response => {
      if (response.ok) {
        return response.json();
      }
      alert('Error: ' + response.statusText);
    })
    .then(postResponse => {
      console.log(postResponse);
      alert('Thank you for adding a zookeeper!');
    });
};
```

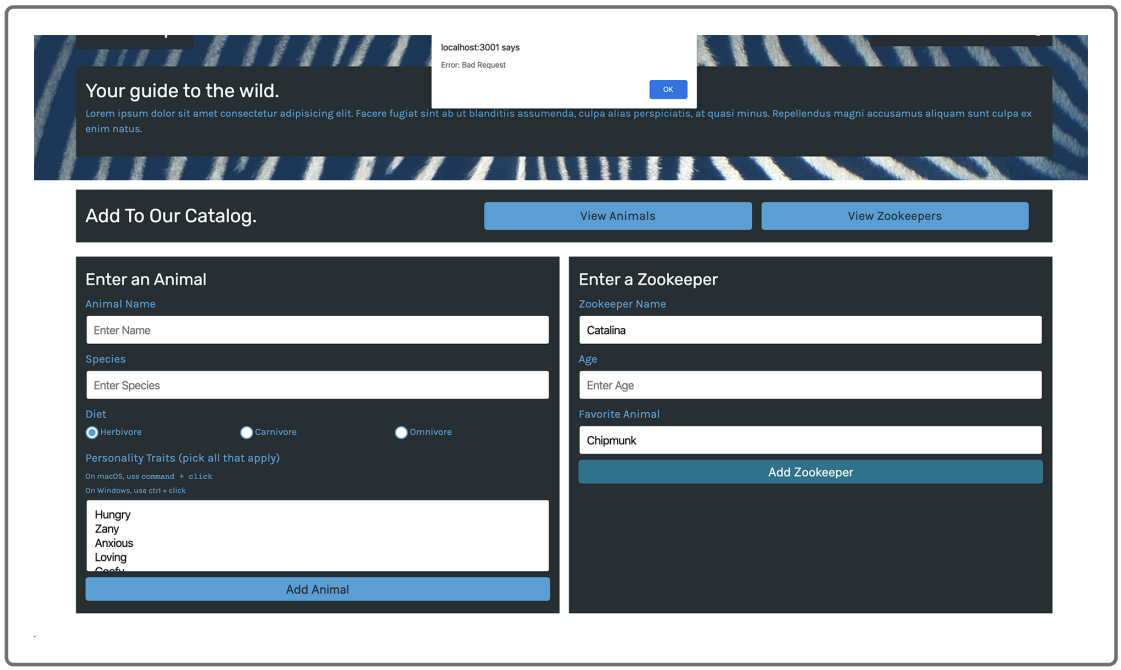
Following that code block, add this above the `animalForm` event listener that's at the bottom of the file:

```
$zookeeperForm.addEventListener('submit', handleZookeeperFormSubmit);
```

Now make sure to test the new form and ensure that everything works when we add a zookeeper. You should get an alert, as shown in following image:



Make sure to test the form validation as well by submitting a form that has missing data. Try leaving out out the zookeeper's age. You should get an error alert like the one shown in the following image:



Good work! The homepage works with our new zookeeper routes. Now we're going to finish up by adding a zookeeper display page to the application.

Review your `routes/htmlRoutes/index.js` file. We previously created the zookeepers route in `server.js`, then moved it to this `index.js` file, and revised it to use the router. Double-check that the `routes/htmlRoutes/index.js` has this code:

```
const path = require("path");
const router = require("express").Router();

router.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../../public/index.html"));
});

router.get("/animals", (req, res) => {
  res.sendFile(path.join(__dirname, "../../public/animals.html"));
});

router.get("/zookeepers", (req, res) => {
  res.sendFile(path.join(__dirname, "../../public/zookeepers.html"));
});

module.exports = router;
```

Think back to our JSON response from the zookeepers route. Unlike the animals, there's no property on the zookeepers that could be a multiple choice or multi-select, like `personalityTraits` or `diet`. We could potentially create a multi-select from the zookeeper's favorite animals, but that could make for a lot of choices.

Let's update the `zookeepers.html` so that it includes a search section, as shown in the following image:

Add the new HTML as a `<section>` that follows the other two `<section>` elements that already exist in the code. The HTML for the display area section should now look something like the following code:

```
<section class="my-2 flex-row justify-space-between">
  <div class="col-12 col-lg-3 bg-dark p-3">
    <h3 class="text-light">Search For Zookeepers</h3>
    <form id="zookeeper-form">
      <label for="name-input" class="form-label">Search by name</label>
      <input name="name" id="name-input" class="form-input" />
      <label for="age-input" class="form-label">Search by age</label>
      <input name="age" id="age-input" class="form-input" type="number" />
      <button class="btn">Get Zookeepers</button>
    </form>
  </div>
  <div class="col-12 col-lg-9 bg-dark p-3">

    <div class="flex-row justify-space-between" id="display-area"></div>
  </div>
</section>
```

Open `public/assets/js/zookeeper.js` in your IDE. Note that it has two functions similar to what is found in the `public/assets/js/animals.js` front-end file: `printResults()` and `getZookeepers()`.

Just as with `public/assets/js/animals.js`, we want to setup the `public/assets/js/zookeeper.js` file so that it can handle queries. Let's start by updating `getZookeepers()`, as shown in the following code:

```
const getZookeepers = (formData = {}) => {
  let queryUrl = '/api/zookeepers?';

  Object.entries(formData).forEach(([key, value]) => {
    queryUrl += `${key}=${value}&`;
  });

  fetch(queryUrl)
    .then(response => {
      if (!response.ok) {
        return alert(`Error: ${response.statusText}`);
      }
      return response.json();
    })
    .then(zookeeperArr => {
      console.log(zookeeperArr);
      printResults(zookeeperArr);
    });
};

getZookeepers();
```

Great. Take a moment to check that the zookeeper page automatically populates with zookeepers upon page load, as shown in the following image:

This is good, but we still need to add a function to handle the form data, then pass it as an argument to `getZookeepers()`. Create a function called `handleGetZookeepersSubmit()`. This function should take the values from the form in `zookeepers.html` and pass them as an object to `getZookeepers()`.

Once you're finished, the code should look like this:

```
const handleGetZookeepersSubmit = event => {
  event.preventDefault();
  const nameHTML = $zookeeperForm.querySelector('[name="name"]');
  const name = nameHTML.value;

  const ageHTML = $zookeeperForm.querySelector('[name="age"]');
  const age = ageHTML.value;

  const zookeeperObject = { name, age };

  getZookeepers(zookeeperObject);
};
```

Now we just need to create a reference to the zookeeper form and put a submit event listener on it.

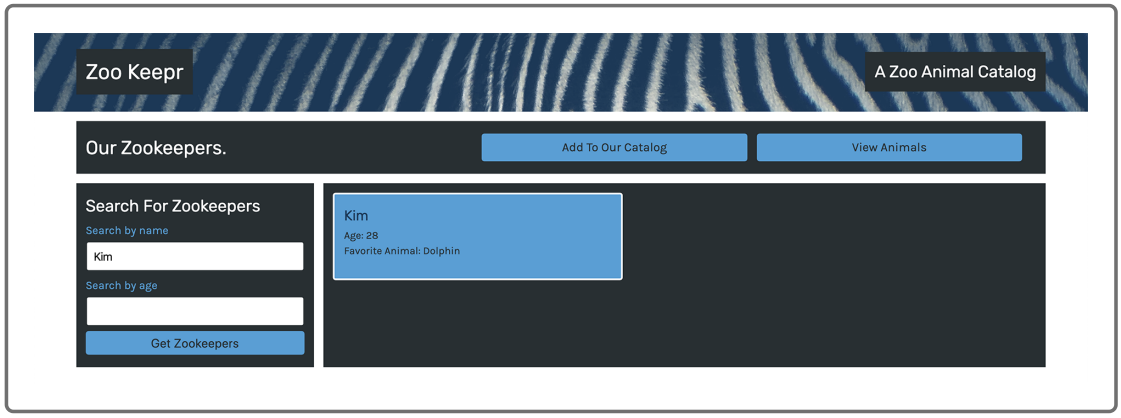
At the top of the `public/assets/js/zookeeper.js` file, add this, which will establish a reference to the form:

```
const $zookeeperForm = document.querySelector("#zookeeper-form");
```

Then, at the bottom of the file, but—importantly—above the `getZookeepers()` function call, add a submit event listener on the zookeeper form, like this:

```
$zookeeperForm.addEventListener('submit', handleGetZookeepersSubmit);
```

Now try testing the new form by searching for one of the zookeepers by name or age. You should see that the results are limited to your new query, as shown in the following image:



Fantastic—we've now added the ability to search for zookeepers by name or age, which will surely come in handy as the zoo adds new employees to their team.

Now that you're all finished, go ahead and redeploy your application to Heroku, and merge the feature branch into `main` in GitHub.

Take a minute to quiz yourself on what you learned during this lesson:

Consider the following scenario:

In the `server.js` file, we have the following code:

```
(app.use('/demo-prefix', routes)
```

`routes` refers to a file that contains this route:

```
router.get('/users', () => ...)
```

What path will you need to get `users`?

- `localhost/users/demo-prefix`
- `localhost/users/:demo-prefix`
- `localhost/demo-prefix/users` ✔
- `localhost/demo-prefix/:users`

Response-specific feedback
Correct! `app.use()` tells all of `routes` to use the `/demo-prefix` path before their individual paths.

Which of the following is **NOT** a reason that we use `jest.mock()`?

- It helps us to test pieces of code in isolation.
- It prevents side effects from happening like writing to files or databases.
- It eliminates the need to create example objects for testing. ✔

Response-specific feedback
Correct. While mocking can eliminate side effects like `writeFile`, mocking does not eliminate the need for example objects.

 Retake