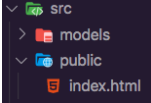# JavaScript Frameworks
Section 2: Node and Express Deployment

4.1 Deploying to Heroku

To deploy an app to Heroku you would need a Heroku account as well as a Guthub account. In the index.js file we would use the following line of code to make reference to a public folder which will have the entry point i.e. index.html file.

```
app.use(express.static('src/public'));
```


To test that the above app.use static method is working and the index.html page will work with our web server we can run the terminal command npm start (nodemon) script to run our server, when we visit the http://localhost:3000 root route, the index.html page should be served.

The Heroku page provides a documented guide on deploying a Node.js application to their platform (https://devcenter.heroku.com/articles/getting-started-with-nodejs).

Once the Heroku CLI has been installed on your machine (using the command line heroku --version will confirm which version is installed on your local machine) we would need to prepare the application for Heroku deployment.

In the terminal cd into the root folder directory of the app we want to deploy we would want to run the following command:

```
$ heroku create    $ git remote -v
```

This will run and come up with a unique name for our app and create a git remote attached to our GitHub. We can run the git remote - v command which will display the heroku and GitHub remotes we have which were created by the first command above.

We should ensure that we are on our master branch before running the next command. To do this we can run the following commands:

```
$ git brach    $ git checkout master
```

The first command will tell us which branch we are on while the second command will change us to the desired branch. Once we are on the master branch we can run the following command:

```
$ git push heroku master
```
This will push our remote git app to heroku.

If this command should not work we can do this manually by running the following command:

```
$ heroku git:remote -a [appName]
```

The [appName] should be replaced with the unique name of the app that was created with the heroku create command. This command is for Heroku setting up a connect between the git remote. The -a flag refers to the app which is why we pass in the name of the app after this flag.

This command should set our git remote heroku to the Heroku app URL which ends with a ".get" which is important. Now that our git remote is connected with Heroku we can run the previous command git push heroku master. This should now run the process of compressing our application and configuring it in a way that Heroku would be able to then deploy the app. We should see the following status printed in the terminal:
Build Succeeded!
Compressing…
Launching…
Verifying deploy…
done.

If we now run the below command this should open our Heroku application in a new browser window/tab:

```
$ heroku open
```

This will serve the file in our static method i.e. the index.html file in the public folder. If we see this in the browser we have successfully deployed our Node/Express application to Heroku and is available to the world on the web.

4.2 Build Input Form

In the index.html file we can build out a form where the user can input values which will allow users to use the RESTful API that we build with Node and Express and Mongoose via the browser.

So far we have been using Postman which is fine for testing routes during production and making sure the API routes are doing what they should be doing. We use input forms so that the user has somewhere to interact and enter values from the browser which will enable them to interact with the database (i.e. API). Below is an example HTML form:

```html
<form>
    <field>
        <label>RESTful API</label>
        <input type="text">
    </field>
</form>
```

Nothing will happen with the form because it is not connected to an event handler. This is the beginning of the RESTful API i.e. an input field/form for GET request, one for POST, PUT and a DELETE requests. We can then use JavaScript to wire the form together with the eventHandler request logic.

4.3 Fetch GET and POST Methods

To create an API for our frontend to communicate with our backend, we could setup a second server on our frontend so that the two can communicate directly but another option is to use jQuery AJAX methods or we could use Fetch.

Fetch is a promise based HTTP request API. Since Fetch is promise based it makes chaining functionality nice and easy. Fetch also provides a generic definition of the request and response objects along with other things involved with network requests.

In the public directory of our application we can create a new file called api.js which will have an event listener so that when a submit button is clicked it will be heard by the listener and a callback function will be triggered to handle the request. This is setup similar to how routes work i.e. there is a request that will come in and this will trigger a callback function to perform a set of function. Below is an example code using familiar patterns we should now be accustom to:

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Node & Express - Heroku Deployment</title>
    </head>
    <body>
        <p>Welcome to the Movies RESTful API</p>
        <form id="user_form">
            <field>
                <label>RESTful API</label>
                <br><br>
                <label>GET</label>
                <button id="getButton" name="getButton" value="submit" type="submit">Get Movies</button> <br><br>
            </field>
        </form>

        <p>Movies Database:</p>
        <div id="results"></div>

        <script src="./api.js"></script>
    </body>
</html>
```

```javascript
const getButton = document.getElementById('user_form');
getButton.addEventListener('submit', getRequest);

function getRequest(event) {
    event.preventDefault();

    fetch('/movies')
        .then(function(res) {
            return res.json();
        })
        .then(function(data) {
            console.log(data);
            console.log(JSON.stringify(data));      //Convert JSON to String representation

            for(var i in data) {
                document.getElementById("results").innerHTML += data[i].movieTitle + '<br>';
            };
        });
};
```

First we set a variable for the button we wish to listen for. We can get this from the HTML by using .getElementById and selecting the user_form id. We can now use this variable to add a new event listener on it to trigger off an event when the event type is triggered.

The addEventListener takes in two arguments. The first is the event type e.g. submit and the second is a callback function to run on the triggered event. We would then have to build out that event function.

This function we create will get passed in the event object which we can then use for example we can use the event object to prevent the default refresh browser behaviour when a form is submitted using the .preventDefault method on this event object. We can then use the Fetch method to fetch from our route and then chain on .then( ) methods on this promise to run functions when the promise resolves for example we can parse the result as a JSON object and then run another function to display the results in the browser or JavaScript console.

The .stringify( ) JavaScript methods allows us to convert a JSON object into a string representation. We can use the .innerHTML method to print the results to our HTML DOM so that they display on the browser window instead of the console. We can use the for loop to achieve this by looping over the data array and adding it to the innerHTML of the selected HTML element.

We now have a way for users interact with our API using web forms to send off a HTTP GET request to our database using Fetch and using the promise to display the results either in the JavaScript console or the browser window.

Below is an example for a POST request form which allows a user to post to the database using the API which is connected via a web form and an Event Handler.

```html
<form id="user_form_post">
    <field>
        <label>POST</label>
        <label>Movie Title</label>
        <input id="movieInput" type="text" name="movieTitle" placeholder="Post Movie" />
        <label>Movie Director</label>
        <input id="directorInput" type="text" name="movieDirector" placeholder="Post Director" />

        <button id="postButton" name="postButton" method="POST">Post Movie</button>
    </field>
</form>
```

The input name attribute is important because we can take the value of the input by targeting that name attribute on the event object that gets passed into our callback function. This allows us to grab the input from the front end and plug it into our route API.

The postButton variable will operate on the user_form_post element where we have a event listener listening for the submit event and when it hears the event it will fire off the newPost callback function.
The newPost function will prevent the default form behaviour on submit event and will capture the input field values from the event.target using the input name attribute and then store these into a post object which is modelled from the schema's expected values. The options object stores the fetch options required for the POST request i.e. the method type, body and headers to be past along with the POST fetch request.

```javascript
const postButton = document.getElementById('user_form_post');
postButton.addEventListener('submit', newPost);

function newPost(event, post) {
    event.preventDefault();
    const movieTitle = event.target.movieTitle.value;
    const movieDirector = event.target.movieDirector.value;
    // console.log(movieTitle, movieDirector);

    post = {
        movieTitle,
        movieDirector,
    };
    // console.log(post);

    const options = {
        method: 'POST',
        body: JSON.stringify(post),
        headers: new Headers({
            'Content-Type': 'application/json'
        })
    };

    return fetch('/movies', options)
        .then(res => res.json())
        .then(res => console.log(res))
        .then(error => console.error('error: ', error));
};
```

Finally, the fetch request takes in the route along with the options object and then using the resolved promise we can get the result object and do whatever we want with it or print the error message should an error occur on the promise.

Important Note: The post object could have been written as the below example but ES6 shorthand syntax can also be used as seen on the example to the left, if the property name and the value have exactly the same name:

```javascript
post = {
    movieTitle: movieTitle,
    movieDirector: movieDirector,
};
```

The option object specifies the method we are using, the body for the request and the headers for the request i.e. for a POST request we are setting the headers content to a JSON. This option object is then used with our fetch request i.e. we use the /movies route and pass in the options into it. This is how the /movies route knows that the request is a POST request (.post( ) method) and not a GET request (.get( ) method) when the callback function gets called.

The options object takes in the post object which is passed into the body and stringified into a string representation of the JSON object and send to our database to create a new movies record.

Fetch makes it very nice and clean to write code to send a request to our database and do something with the response because we can work with promises which is represented with the .then( ) method calls. For example, we can run the .json( ) method on the result in one line and then take that response and send it to the console or the browser window and also print out the error.

This is how we can use Fetch API to easily perform GET and POST requests to our database using a frontend web form for the user to interact with our Node & Express server API. We can also use the Fetch API to perform the other HTTP methods such as PUT and DELETE requests.

We now have a Node and Express app on heroku which has a frontend web form that allows the user to interact with our RESTful API to interact with our database.

4.4 Fetch PUT and DELETE Methods

To complete the NodeJS and Express RESTful API, below is an example code for how we would approach the PUT and DELETE methods using the Fetch API.

```html
<form id="user_form_put">
    <field>
        <label id="request_type">UPDATE:</label><br>
        <label>Movie ID</label>
        <input id="movieId" type="text" name="movieId" placeholder="Movie ID" />
        <label>Movie Title</label>
        <input id="movieInput" type="text" name="movieTitle" placeholder="Update Movie" />
        <label>Movie Director</label>
        <input id="directorInput" type="text" name="movieDirector" placeholder="Update Director" />

        <button id="putButton" name="putButton" method="PUT">Update Movie</button>
    </field>
</form>
```

```javascript
function putPost(event) {
    event.preventDefault();
    const movieId = event.target.movieId.value;
    const movieTitle = event.target.movieTitle.value;
    const movieDirector = event.target.movieDirector.value;
    console.log('movieId: ', movieId, 'movieTitle: ', movieTitle, 'movieDirector: ', movieDirector);

    post = {
        movieTitle,
        movieDirector
    };
    // console.log(post);

    const options = {
        method: 'PATCH',
        body: JSON.stringify(post),
        headers: new Headers({
            'Content-Type': 'application/json'
        })
    };

    const URL = `/movies/${movieId}`;

    return fetch(URL, options)
        .then(res => res.json())
        .then(data => console.log('Movie updated to: ', data))
        .then(error => console.error('error: ', error));
};
```

Important Note: The method property value in the options object must match the nomenclature used on the backend i.e. we used PATCH not PUT on the backend for our update route.

```html
<form id="user_form_delete">
    <field>
        <label id="request_type">DELETE:</label><br>
        <label>Movie ID</label>
        <input id="movieId" type="text" name="movieId" placeholder="Movie ID" />

        <button id="deleteButton" name="deleteButton" method="DELETE">Delete Movie</button>
    </field>
</form>
```

```javascript
function deletePost(event) {
  event.preventDefault();
  const movieId = event.target.movieId.value;
  console.log('movieId: ', movieId);

  const options = {
    method: 'DELETE',
    body: JSON.stringify({
      movieId
    }),
    headers: new Headers({
      'Content-Type': 'application/json'
    })
  };

  const URL = `/movies/${movieId}`;

  return fetch(URL, options)
    .then(res => res.json())
    .then(data => console.log('Movie to delete: ', data))
    .then(error => console.error('error: ', error));
};
```

This wraps up how to create a web server on the backend using NodeJs and Express to create a RESTful API that utilises the Fetch API to interact with the backend database using the frontend webforms which is hosted as an application on the web via Heroku.

END