Project Report

Cloud App Development

Author

Jeffrey Vavasour # x23175150

Repository link: https://github.com/JVNCI/articleApp

Deployed website link: http://16.16.25.134:3000

The website is not secured through https. Therefore, it may not open in your browser. The following steps allowed me to visit the site in chrome.

1. Go to `chrome://net-internals/#hsts`. Enter http://16.16.25.134:3000 under **Delete domain security policies** and press the Delete button.
2. Now go to `chrome://settings/clearBrowserData`, tick the box *Cached images and files* and press click the button *Clear data*.

Program using Ruby on Rails

Firstly, I created a new rails application

```
rails new articleApp
```

I then cd into the rails app

```
cd articleApp
```

I then generated an Article model

```
rails generate model Article title:string body:text published:boolean
```

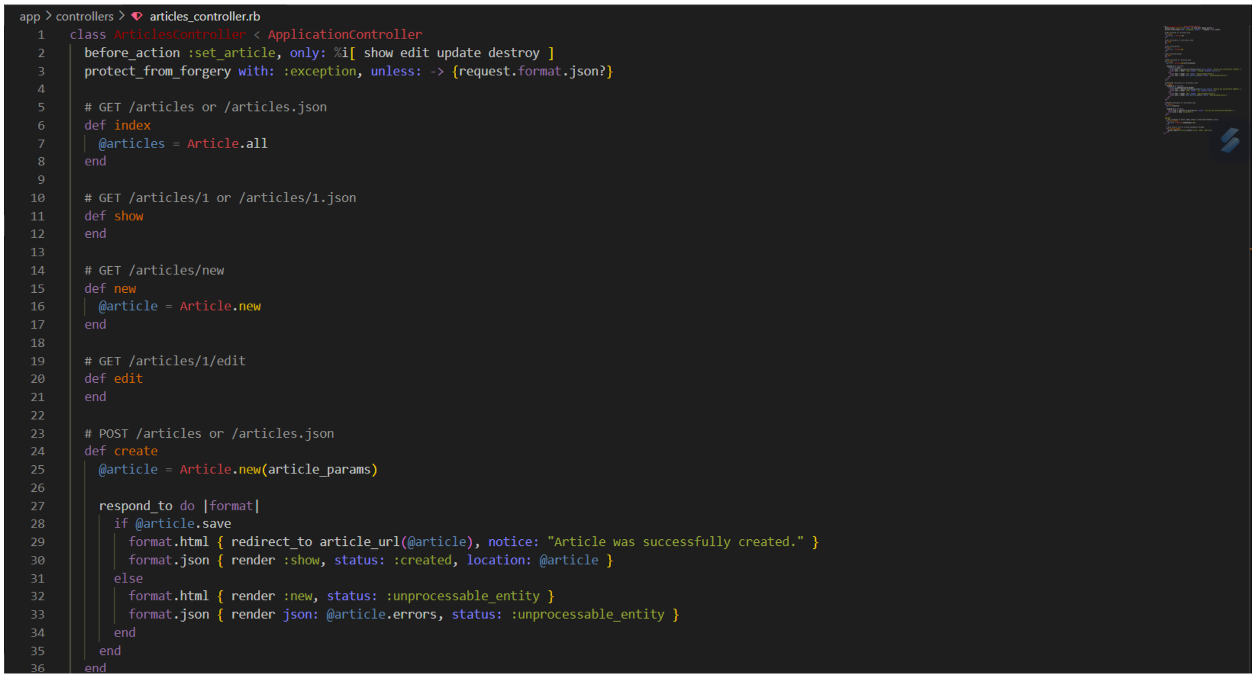
Next, I created and migrated the database

```
rails db:create
rails db:migrate
```

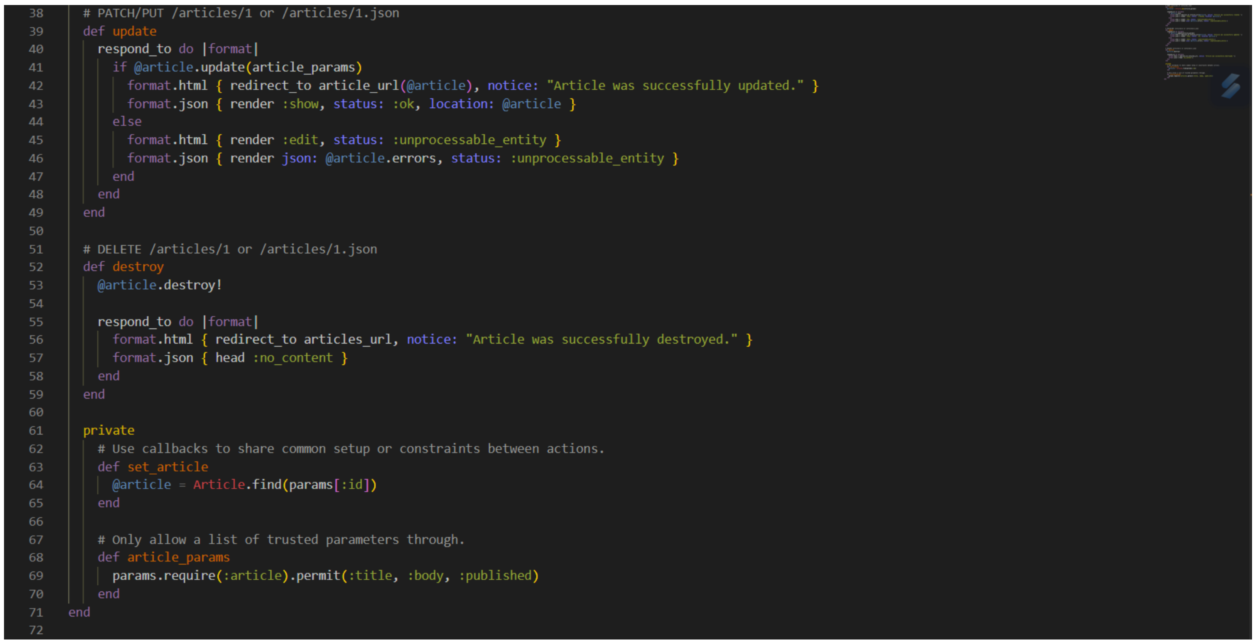
I then created the controller for the Articles

```
rails generate controller Articles
```

I then implemented the CRUD operations into the controller

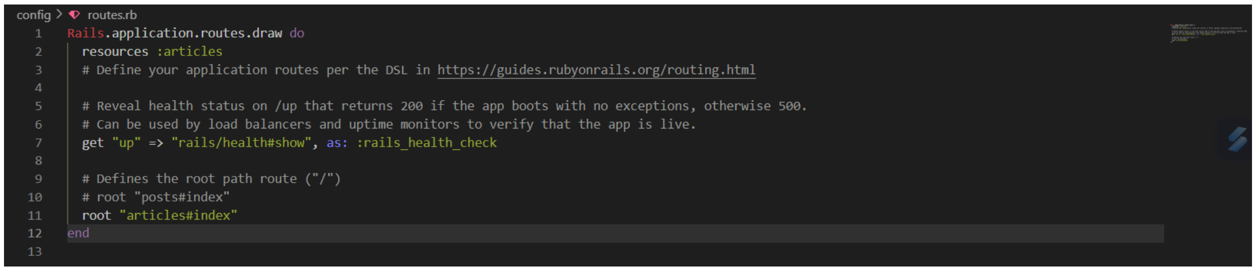


```ruby
class ArticlesController < ApplicationController
  before_action :set_article, only: %i[ show edit update destroy ]
  protect_from_forgery with: :exception, unless: -> {request.format.json?}

  # GET /articles or /articles.json
  def index
    @articles = Article.all
  end

  # GET /articles/1 or /articles/1.json
  def show
  end

  # GET /articles/new
  def new
    @article = Article.new
  end

  # GET /articles/1/edit
  def edit
  end

  # POST /articles or /articles.json
  def create
    @article = Article.new(article_params)

    respond_to do |format|
      if @article.save
        format.html { redirect_to article_url(@article), notice: "Article was successfully created." }
        format.json { render :show, status: :created, location: @article }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @article.errors, status: :unprocessable_entity }
      end
    end
  end
```



```ruby
  # PATCH/PUT /articles/1 or /articles/1.json
  def update
    respond_to do |format|
      if @article.update(article_params)
        format.html { redirect_to article_url(@article), notice: "Article was successfully updated." }
        format.json { render :show, status: :ok, location: @article }
      else
        format.html { render :edit, status: :unprocessable_entity }
        format.json { render json: @article.errors, status: :unprocessable_entity }
      end
    end
  end

  # DELETE /articles/1 or /articles/1.json
  def destroy
    @article.destroy!

    respond_to do |format|
      format.html { redirect_to articles_url, notice: "Article was successfully destroyed." }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_article
      @article = Article.find(params[:id])
    end

    # Only allow a list of trusted parameters through.
    def article_params
      params.require(:article).permit(:title, :body, :published)
    end
end
```

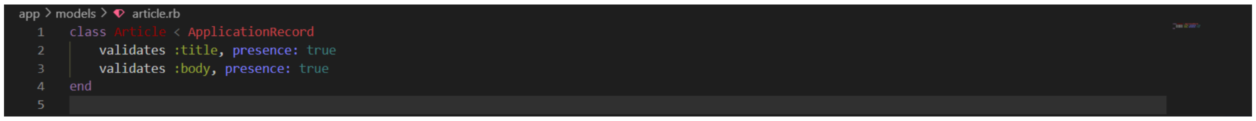
Then, I set up the routes in the config/routes.rb file



config > routes.rb

```ruby
Rails.application.routes.draw do
  resources :articles
  # Define your application routes per the DSL in https://guides.rubyonrails.org/routing.html

  # Reveal health status on /up that returns 200 if the app boots with no exceptions, otherwise 500.
  # Can be used by load balancers and uptime monitors to verify that the app is live.
  get "up" => "rails/health#show", as: :rails_health_check

  # Defines the root path route ("/")
  # root "posts#index"
  root "articles#index"
end
```

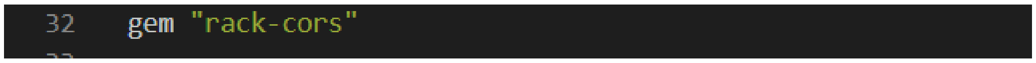
I added validation to the Article model, requiring the presence of a title and a body



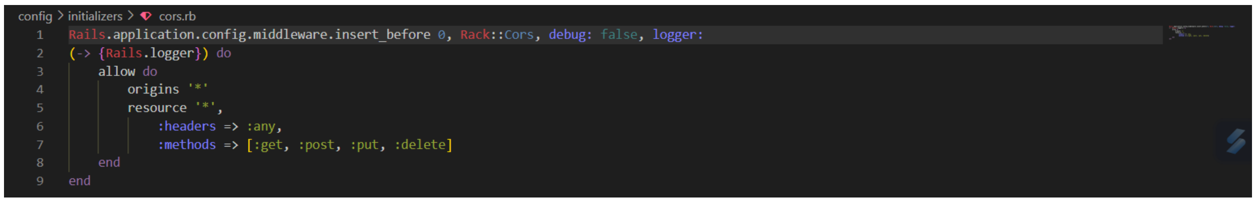
app > models > article.rb

```ruby
class Article < ApplicationRecord
    validates :title, presence: true
    validates :body, presence: true
end
```

Next, I implemented CORs policy by adding the "rack-cors" gem to the gemfile and also adding a cors.rb file in the config/initializers folder



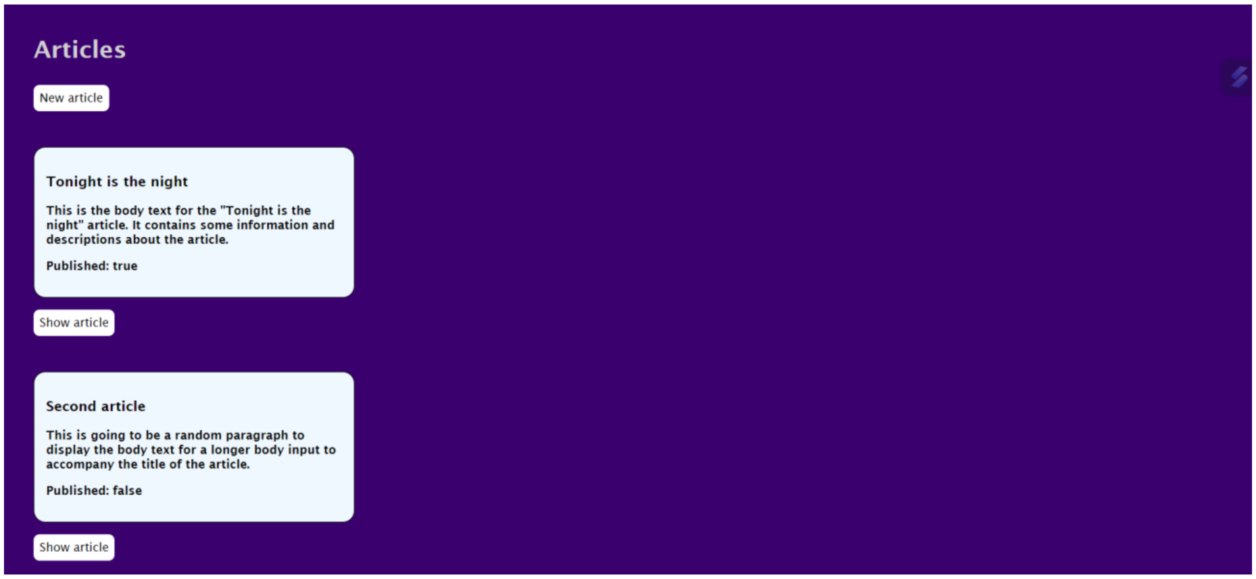
```ruby
gem "rack-cors"
```



config > initializers > cors.rb

```ruby
Rails.application.config.middleware.insert_before 0, Rack::Cors, debug: false, logger:
(-> {Rails.logger}) do
    allow do
        origins '*'
        resource '*',
            :headers => :any,
            :methods => [:get, :post, :put, :delete]
    end
end
```

HTML client


Articles

New article

Tonight is the night

This is the body text for the "Tonight is the night" article. It contains some information and descriptions about the article.

Published: true

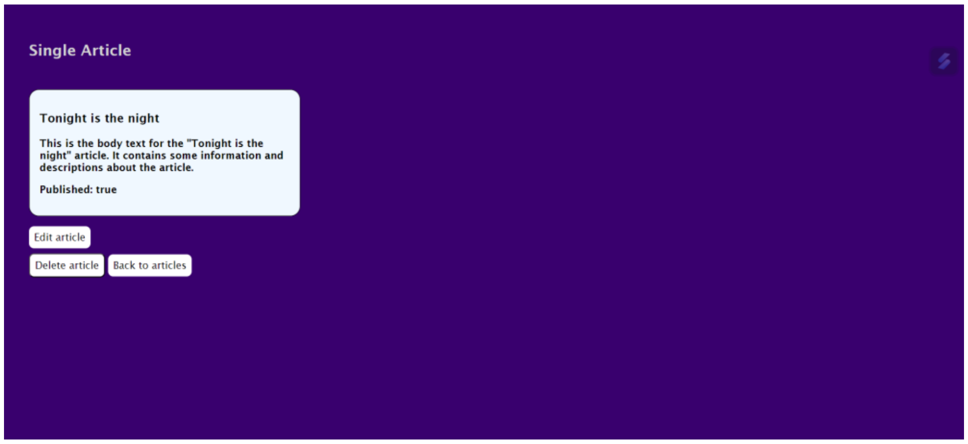
Show article

Second article

This is going to be a random paragraph to display the body text for a longer body input to accompany the title of the article.

Published: false

Show article


Single Article

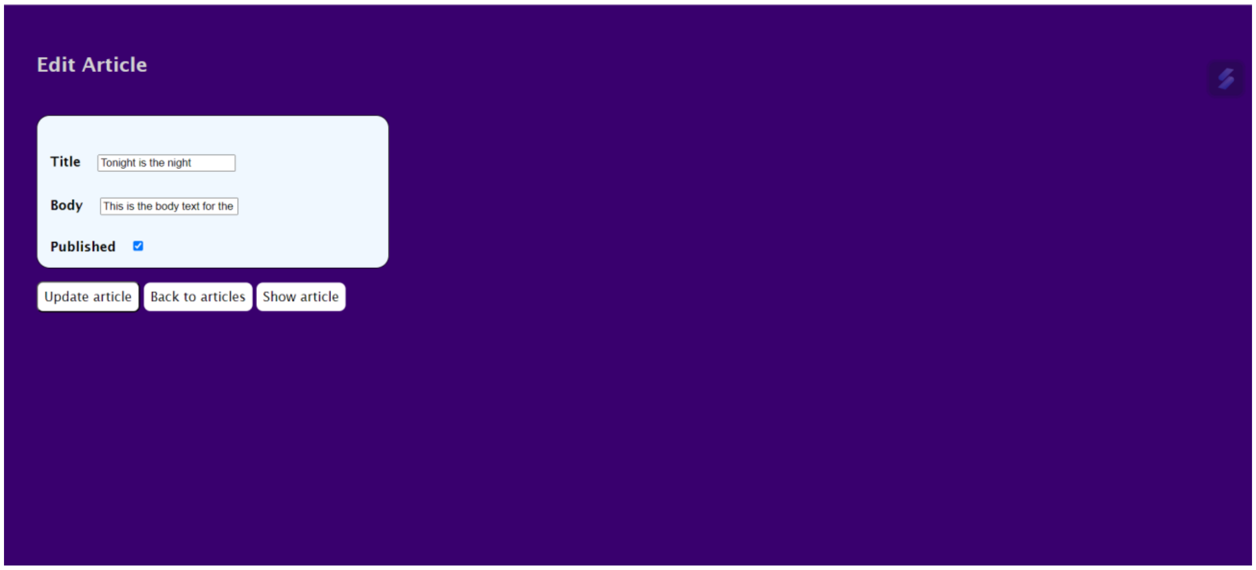
Tonight is the night

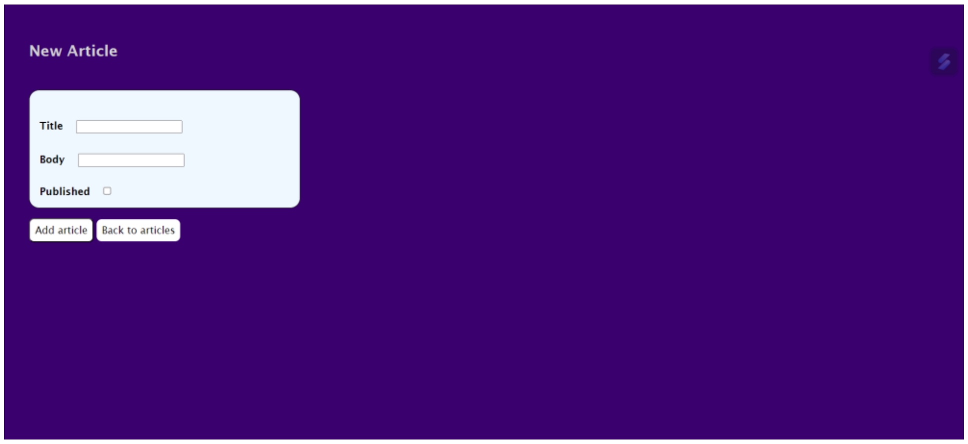
This is the body text for the "Tonight is the night" article. It contains some information and descriptions about the article.

Published: true

Edit article

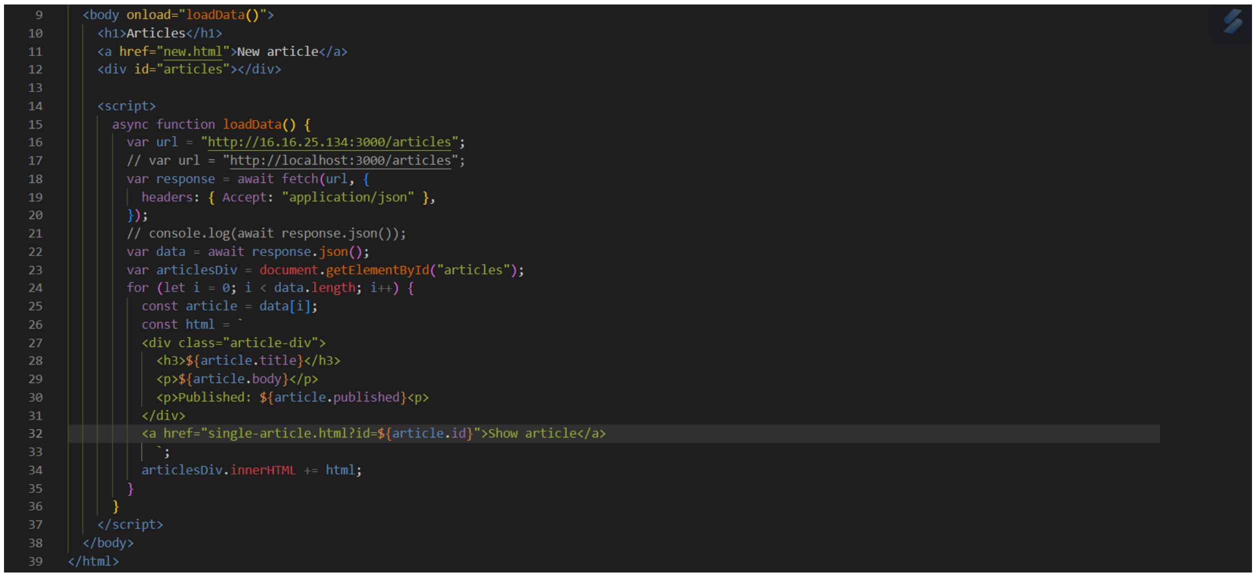
Delete article Back to articles





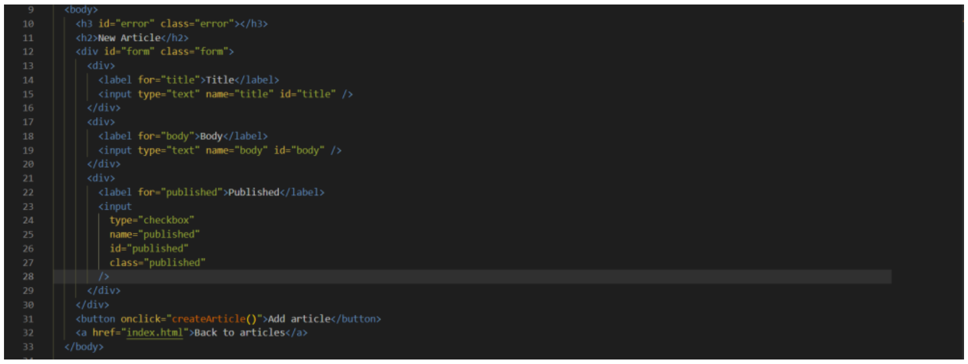
I created the HTML client files inside the public folder of the rails app. I used one stylesheet using vanilla CSS to style all the pages.

Inside the index.html file, I used a function inside the body tag that is called when the page is loaded. This is an asynchronous function that fetches the data from the server. I have the deployed application URL in use and have the local URL commented out in case one would like to use it for running the application locally. When the data has been fetched, it is put into JSON format so it can be iterated over. I use a for loop to create the HTML for each article that is iterated over. This HTML div is then appended to an existing div in the HTML body, using the document.getElementById function. I added a link at the end of each article to display the specific article by itself. This link specifies the id of an article in its parameters.



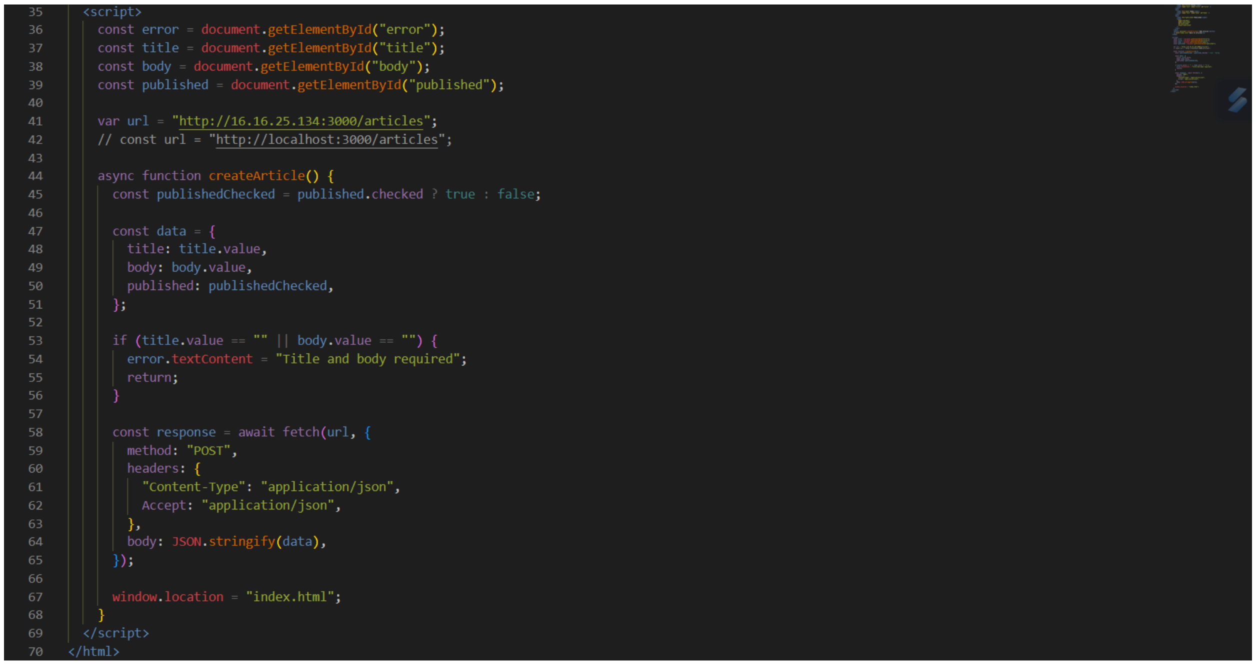
```html
  <body onload="loadData()">
    <h1>Articles</h1>
    <a href="new.html">New article</a>
    <div id="articles"></div>

    <script>
      async function loadData() {
        var url = "http://16.16.25.134:3000/articles";
        // var url = "http://localhost:3000/articles";
        var response = await fetch(url, {
          headers: { Accept: "application/json" },
        });
        // console.log(await response.json());
        var data = await response.json();
        var articlesDiv = document.getElementById("articles");
        for (let i = 0; i < data.length; i++) {
          const article = data[i];
          const html = `
          <div class="article-div">
            <h3>${article.title}</h3>
            <p>${article.body}</p>
            <p>Published: ${article.published}<p>
          </div>
          <a href="single-article.html?id=${article.id}">Show article</a>
            `;
          articlesDiv.innerHTML += html;
        }
      }
    </script>
  </body>
</html>
```

I created a new.html file for the creation of new articles. Inside the new.html file I added a form for the user to input the information. I added a button which has an onClick event listener to call a function which will create an article based off the user inputs. I also added a link to return to the main articles page and a div to allow an error message to be placed into if an error is returned.



```html
  <body>
    <h3 id="error" class="error"></h3>
    <h2>New Article</h2>
    <div id="form" class="form">
      <div>
        <label for="title">Title</label>
        <input type="text" name="title" id="title" />
      </div>
      <div>
        <label for="body">Body</label>
        <input type="text" name="body" id="body" />
      </div>
      <div>
        <label for="published">Published</label>
        <input
          type="checkbox"
          name="published"
          id="published"
          class="published"
        />
      </div>
    </div>
    <button onclick="createArticle()">Add article</button>
    <a href="index.html">Back to articles</a>
  </body>
```

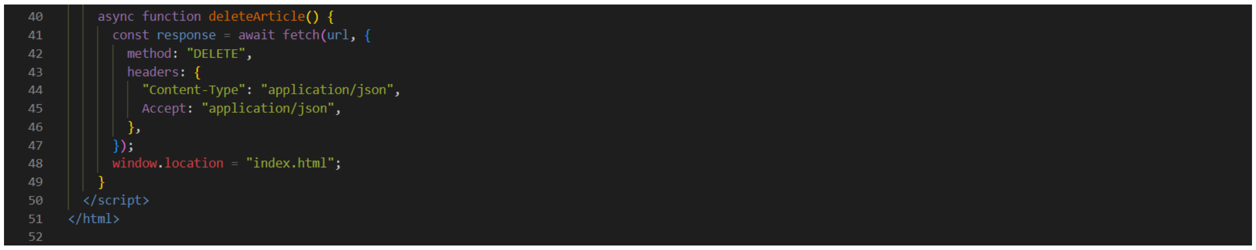
In the script section, firstly, I selected multiple elements from the DOM with the getElementById function. Some of these were used to get the user input values from the form. I then tested to see if these values were empty using an if statement, which would populate the error message div with an error message if they were empty. If the function passed this if statement, there is then a POST request sent to the server. This POST request uses the data inputted by the user in the form. It will create an article in the database with this information. The function then navigates the user back to the index.html page where they can see all the articles along with the newly created one.



```
  <script>
    const error = document.getElementById("error");
    const title = document.getElementById("title");
    const body = document.getElementById("body");
    const published = document.getElementById("published");

    var url = "http://16.16.25.134:3000/articles";
    // const url = "http://localhost:3000/articles";

    async function createArticle() {
      const publishedChecked = published.checked ? true : false;

      const data = {
        title: title.value,
        body: body.value,
        published: publishedChecked,
      };

      if (title.value == "" || body.value == "") {
        error.textContent = "Title and body required";
        return;
      }

      const response = await fetch(url, {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          Accept: "application/json",
        },
        body: JSON.stringify(data),
      });

      window.location = "index.html";
    }
  </script>
</html>
```

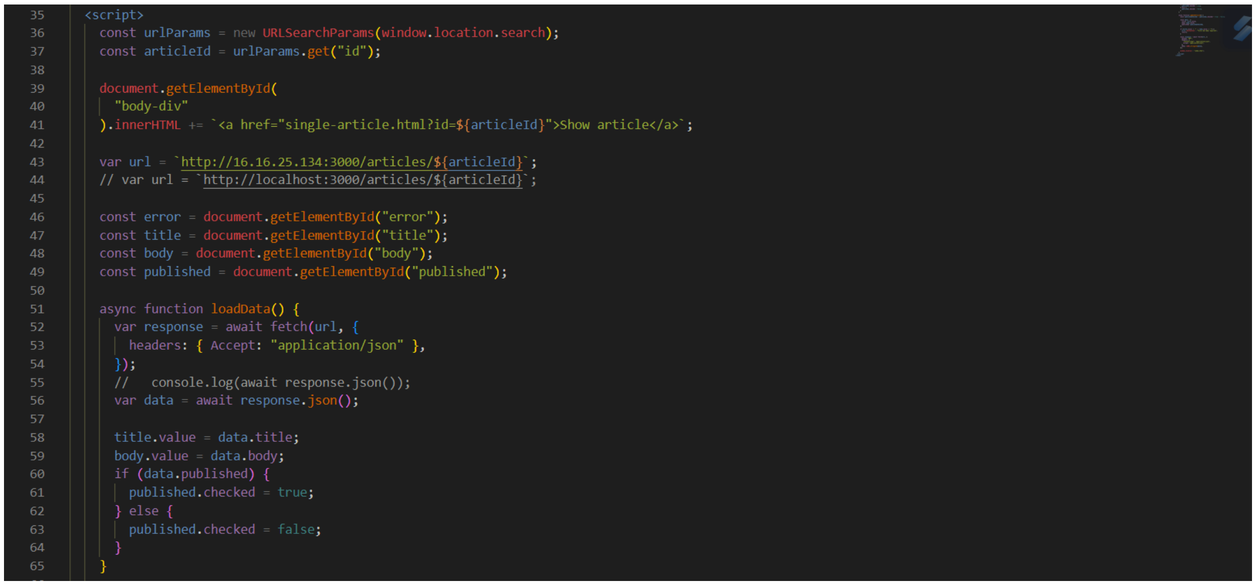
I added a show.html file for displaying singular articles. This page is reached through a link which includes a specific article id in its parameters. This id is then accessed and stored inside a variable. There is an onLoad event listener on the body tag. This will call a function which will fetch the data from the server for the specific article id stored in the variable. A HTML element is then created and appended to the body with the received data.

This page also includes a button element with a onClick event listener attached. This event listener calls a function to delete the said article. It uses a DELETE request with a specific URL containing the article id to delete it.



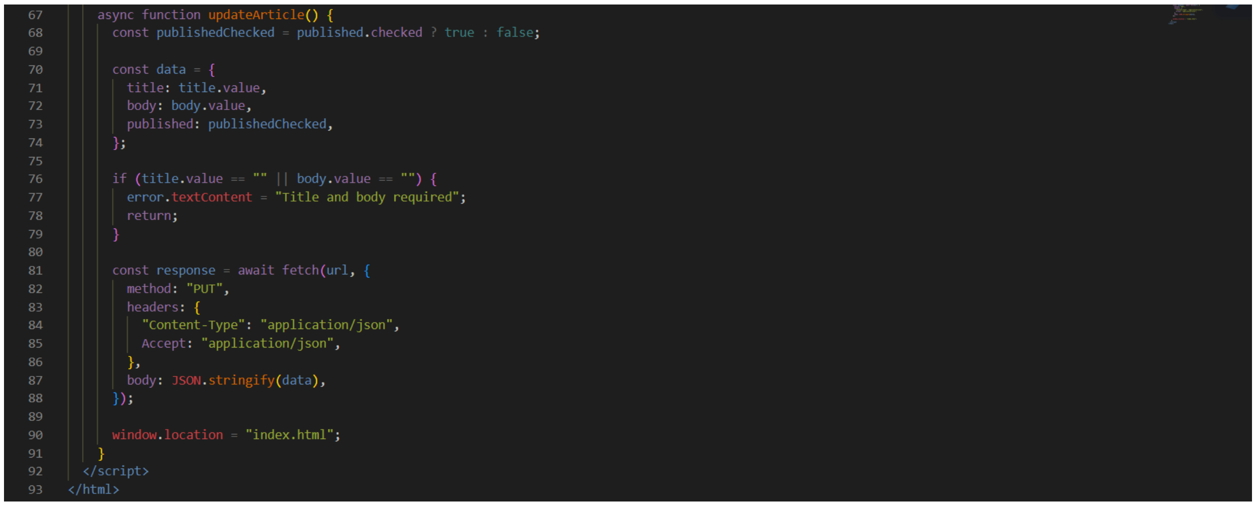
```
    async function deleteArticle() {
      const response = await fetch(url, {
        method: "DELETE",
        headers: {
          "Content-Type": "application/json",
          Accept: "application/json",
        },
      });
      window.location = "index.html";
    }
  </script>
</html>

```

I also implemented an edit.html file to edit existing articles. This page is reached through a link which includes a specific article id in its parameters. This id is then accessed and stored inside a variable. There is an onLoad event listener on the body tag. This will call a function which will fetch the data from the server for the specific article id stored in the variable. The fetched data is then used to populate the form fields with the current article information.



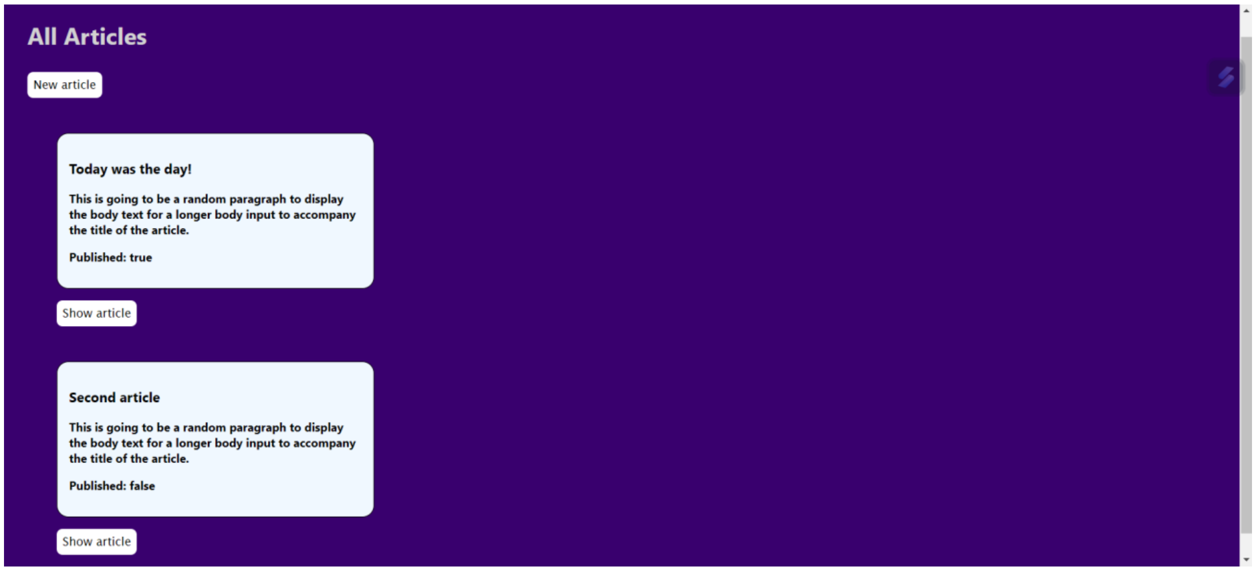
```
  <script>
    const urlParams = new URLSearchParams(window.location.search);
    const articleId = urlParams.get("id");

    document.getElementById(
      "body-div"
    ).innerHTML += `<a href="single-article.html?id=${articleId}">Show article</a>`;

    var url = `http://16.16.25.134:3000/articles/${articleId}`;
    // var url = `http://localhost:3000/articles/${articleId}`;

    const error = document.getElementById("error");
    const title = document.getElementById("title");
    const body = document.getElementById("body");
    const published = document.getElementById("published");

    async function loadData() {
      var response = await fetch(url, {
        headers: { Accept: "application/json" },
      });
      //   console.log(await response.json());
      var data = await response.json();

      title.value = data.title;
      body.value = data.body;
      if (data.published) {
        published.checked = true;
      } else {
        published.checked = false;
      }
    }
```

A button is also included which has an onClick event listener to call a function which will update an article based off the new user inputs. This function checks for any empty inputs and will populate an HMTL div with an error message if so. If the credentials are valid, a PUT request is sent to the server and the user is navigated back to the index.html page.



```
    async function updateArticle() {
      const publishedChecked = published.checked ? true : false;

      const data = {
        title: title.value,
        body: body.value,
        published: publishedChecked,
      };

      if (title.value == "" || body.value == "") {
        error.textContent = "Title and body required";
        return;
      }

      const response = await fetch(url, {
        method: "PUT",
        headers: {
          "Content-Type": "application/json",
          Accept: "application/json",
        },
        body: JSON.stringify(data),
      });

      window.location = "index.html";
    }
  </script>
</html>
```

React client



All Articles

New article

Today was the day!

This is going to be a random paragraph to display the body text for a longer body input to accompany the title of the article.

Published: true

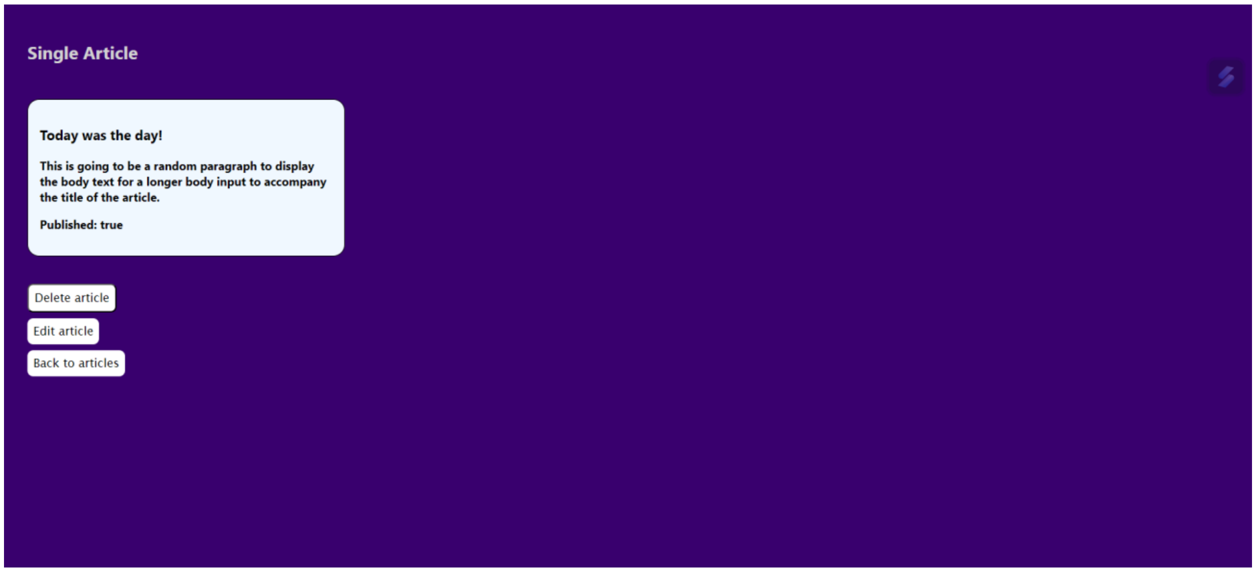
Show article

Second article

This is going to be a random paragraph to display the body text for a longer body input to accompany the title of the article.

Published: false

Show article



Single Article

Today was the day!

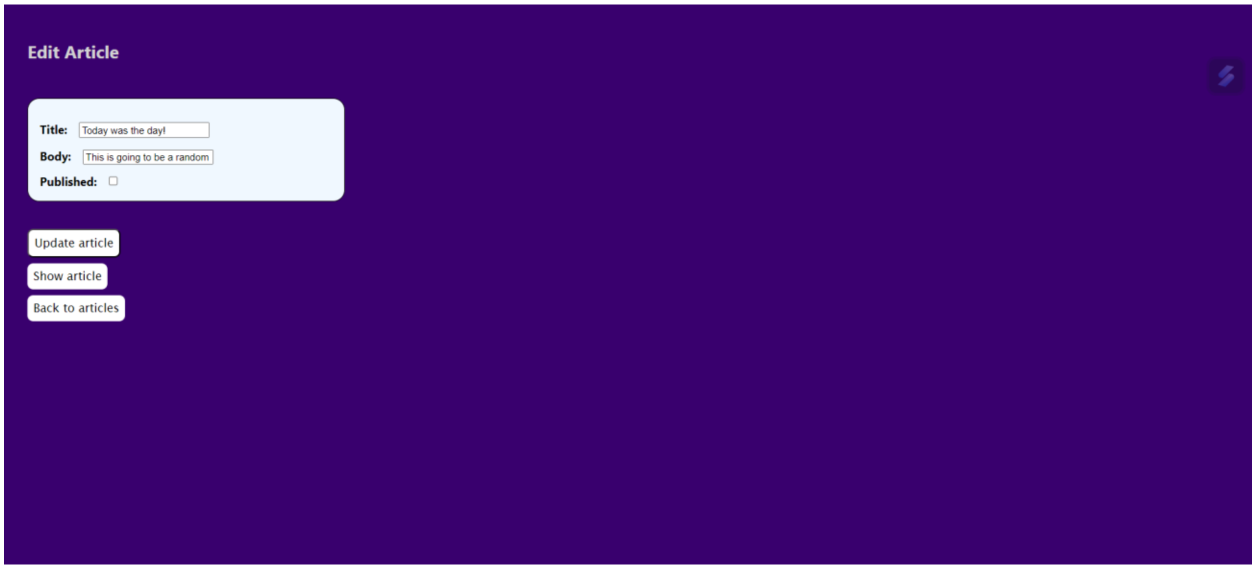
This is going to be a random paragraph to display the body text for a longer body input to accompany the title of the article.

Published: true

Delete article

Edit article

Back to articles

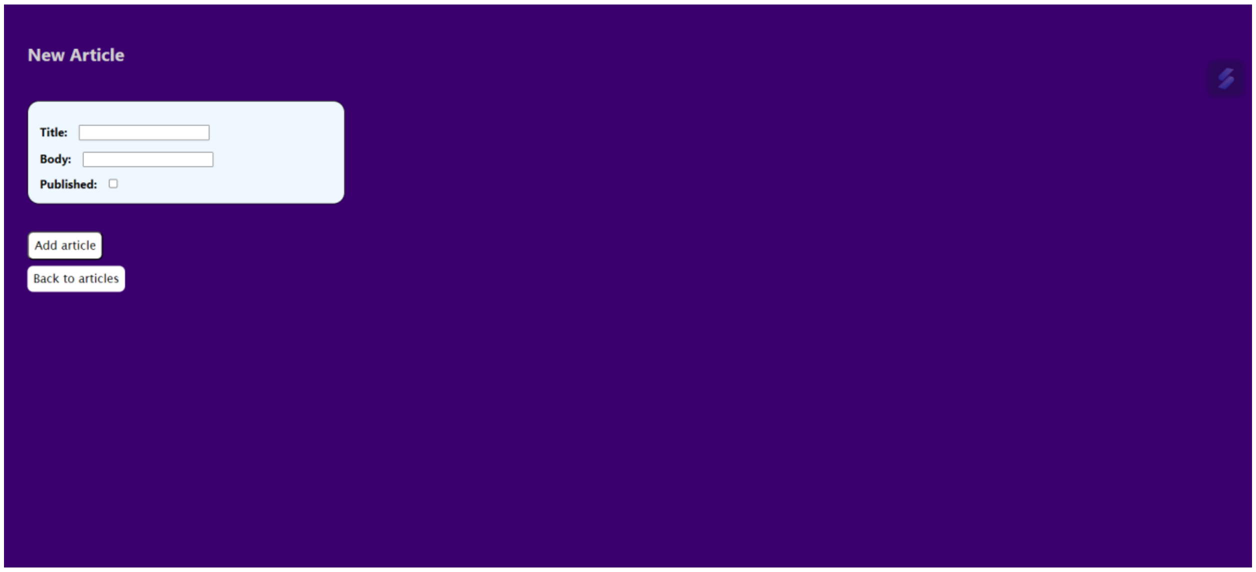


Edit Article

Title: Today was the day!

Body: This is going to be a random

Published: ☐



New Article

Title:

Body:

Published: ☐

I started off by creating a react application

```
npx create-react-app article-app-react-client
```

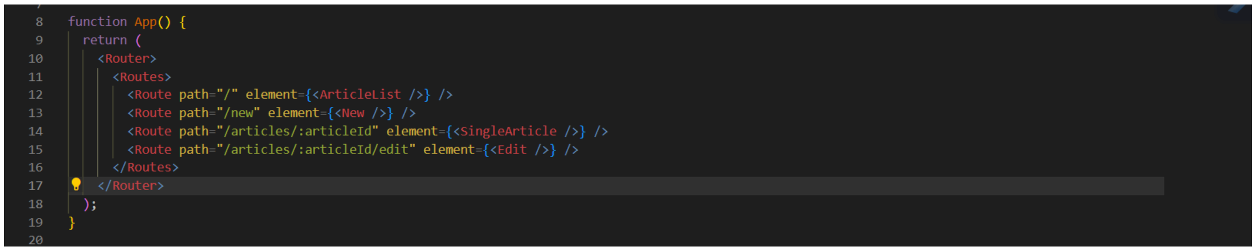
I installed axios to use for the API calls

```
npm install axios
```

I installed react-router-dom for the routing

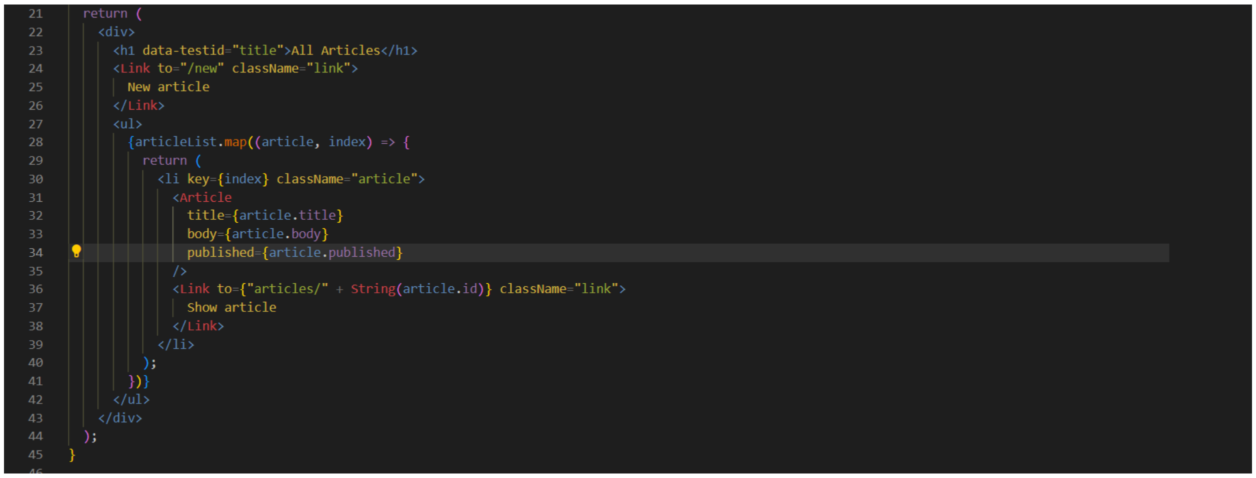
```
npm install react-router-dom
```

In the App.js file, I used a Browser Router from react-router-dom to set the routes for the application. The root was set to navigate to the ArticleList component. The App.js file also imports the App.css file which contains the styling for all of the application.



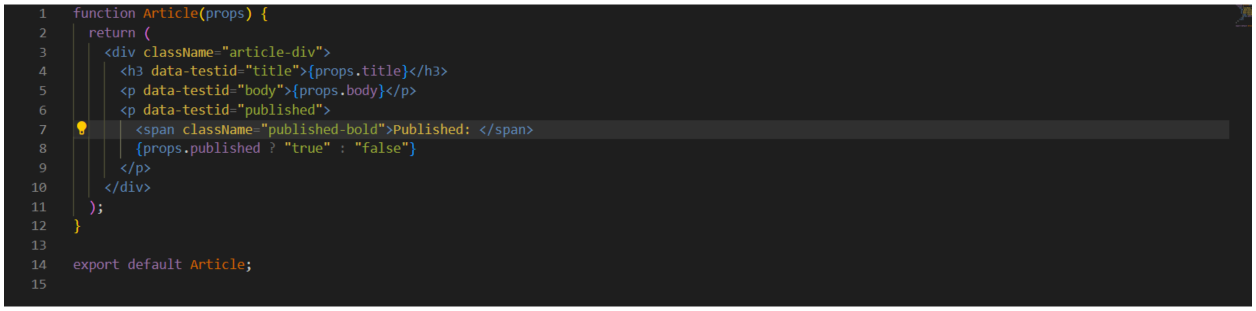
```
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<ArticleList />} />
        <Route path="/new" element={<New />} />
        <Route path="/articles/:articleId" element={<SingleArticle />} />
        <Route path="/articles/:articleId/edit" element={<Edit />} />
      </Routes>
    </Router>
  );
}
```

The ArticleList component uses a useEffect function that calls a function to get the data from the server. This is a GET request to get all the existing articles in the database. The data is then stored in a useState variable. The component then maps over this state variable, creating an article for each iteration, passing specific values to the Article.js component.



```
  return (
    <div>
      <h1 data-testid="title">All Articles</h1>
      <Link to="/new" className="link">
        New article
      </Link>
      <ul>
        {articleList.map((article, index) => {
          return (
            <li key={index} className="article">
              <Article
                title={article.title}
                body={article.body}
                published={article.published}
              />
              <Link to={"articles/" + String(article.id)} className="link">
                Show article
              </Link>
            </li>
          );
        })}
      </ul>
    </div>
  );
}
```

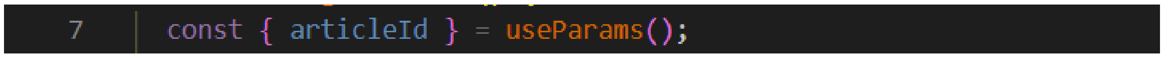
The Article.js component takes in these properties and uses them to return information out to the user.



```
function Article(props) {
  return (
    <div className="article-div">
      <h3 data-testid="title">{props.title}</h3>
      <p data-testid="body">{props.body}</p>
      <p data-testid="published">
        <span className="published-bold">Published: </span>
        {props.published ? "true" : "false"}
      </p>
    </div>
  );
}

export default Article;
```

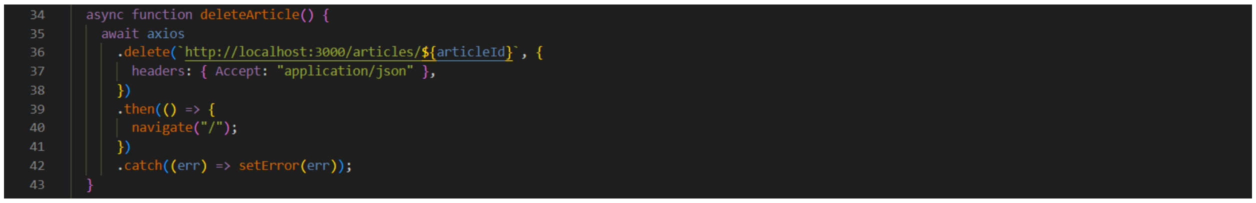
A New.js view was created to let the user add articles. This view contained a form to accept user inputs for title, body and published values. Each input has a onChange listener attached to update state variables once the user inputs information. When the user clicks on a button to add the article, a POST request is sent using the current state variables. Upon successful completion of the request, the user will be navigated back to the home page using the useNavigate function imported from react-router-dom. If the request is unsuccessful, an error message will be sent to the user's page.

A SingleArticle.js view was created to let the user display a specific article. This view made use of the useParams function from react-router-dom.



```
const { articleId } = useParams();
```

This allowed the specific article id to be stored in a variable. This variable was then used with a GET request to the server. The returned data was stored in useState variables. The view then returned the article based on the current state variables. A delete button was also implemented to give the user the ability to remove said article from the database.



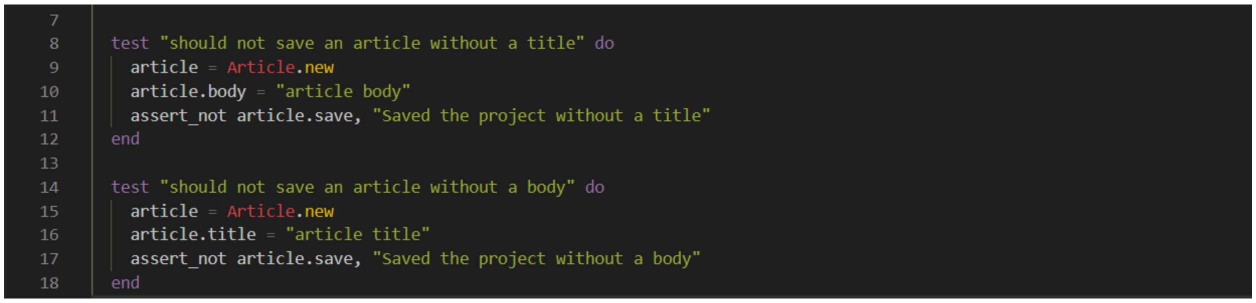
```
async function deleteArticle() {
  await axios
    .delete(`http://localhost:3000/articles/${articleId}`, {
      headers: { Accept: "application/json" },
    })
    .then(() => {
      navigate("/");
    })
    .catch((err) => setError(err));
}
```

An edit link is available on this view for the user to edit the specific article in question. This routes to the Edit.js view.

In the Edit.js view, a useEffect is used to call a funtion, which gets the data for the specific article which needs to be edited. This again is happening through the use of the useParams function. The received data then sets state variables, which are used to populate the form the user sees. These form elements also have onChange listeners to update the state variables when the user inputs data. When the user clicks on the button to update the article, a PUT request is sent to the server. On successful completion of the request, the user is navigated back to the home page. If the request is unsuccessful, an error message will be sent to the user's screen.

Testing

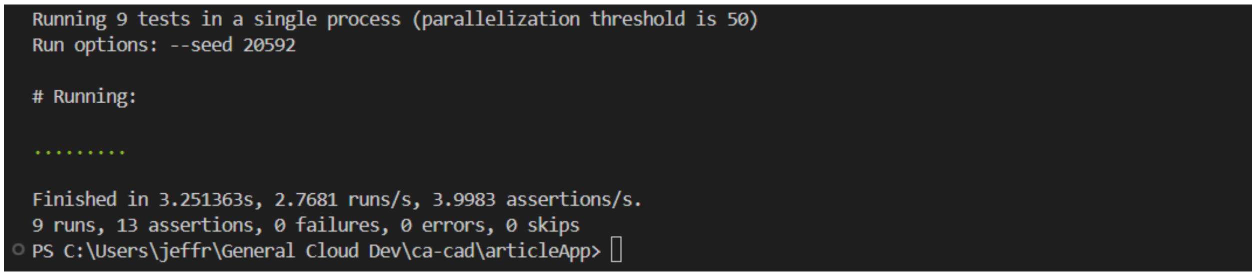
For the backend application, firstly, I implemented model tests in the test/models/article_test.rb file to test if the article will be saved without a title or a body.



```
test "should not save an article without a title" do
  article = Article.new
  article.body = "article body"
  assert_not article.save, "Saved the project without a title"
end

test "should not save an article without a body" do
  article = Article.new
  article.title = "article title"
  assert_not article.save, "Saved the project without a body"
end
```

Then, I implemented tests inside the test/controllers/article_controllers.rb. These tested the CRUD operation of the backend application, including getting articles/article, updating articles, deleting articles and checked if the routing worked for each URL.

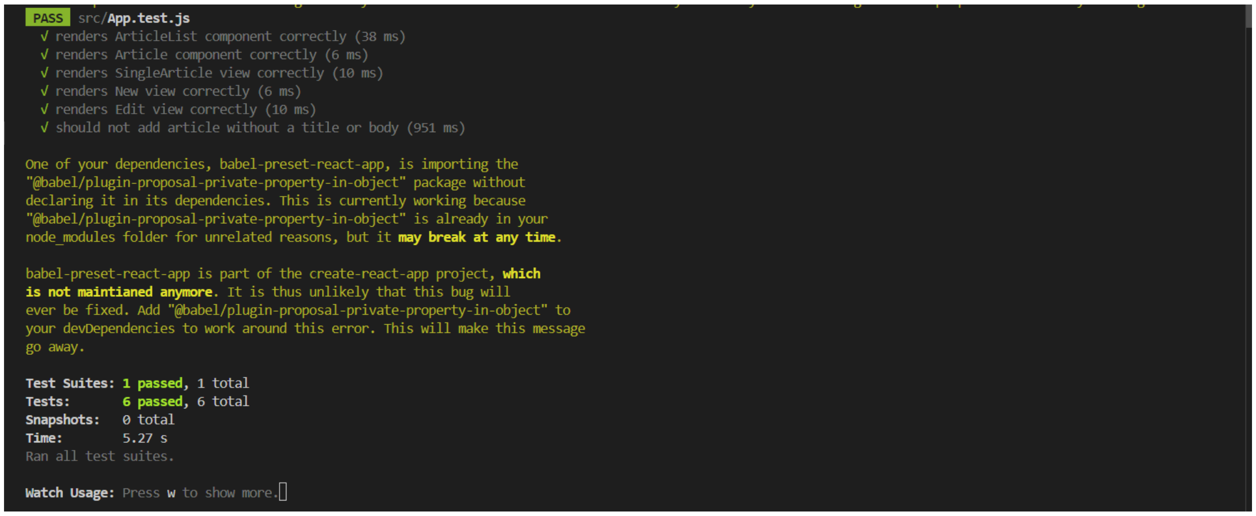
These tests can be ran using the “rails test” command.



```
Running 9 tests in a single process (parallelization threshold is 50)
Run options: --seed 20592

# Running:

.........

Finished in 3.251363s, 2.7681 runs/s, 3.9983 assertions/s.
9 runs, 13 assertions, 0 failures, 0 errors, 0 skips
PS C:\Users\jeffr\General Cloud Dev\ca-cad\articleApp>
```

For the react client I implemented test using the Jest library. I added the tests into the App.test.js file. These tests made sure each component and view would be rendered correctly. They also tested for the presence of a title and a body upon an article creation.

These tests were run using the “npm test” command.



```
PASS  src/App.test.js
  √ renders ArticleList component correctly (38 ms)
  √ renders Article component correctly (6 ms)
  √ renders SingleArticle view correctly (10 ms)
  √ renders New view correctly (6 ms)
  √ renders Edit view correctly (10 ms)
  √ should not add article without a title or body (951 ms)

One of your dependencies, babel-preset-react-app, is importing the
"@babel/plugin-proposal-private-property-in-object" package without
declaring it in its dependencies. This is currently working because
"@babel/plugin-proposal-private-property-in-object" is already in your
node_modules folder for unrelated reasons, but it may break at any time.

babel-preset-react-app is part of the create-react-app project, which
is not maintianed anymore. It is thus unlikely that this bug will
ever be fixed. Add "@babel/plugin-proposal-private-property-in-object" to
your devDependencies to work around this error. This will make this message
go away.

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        5.27 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Importance of testing

Both unit testing and integration testing are invaluable in the software development process. Each provides abilities such as the reliability, stability, and quality of a software product.

unit testing

Unit testing looks at small specific sections of code, to make sure they behave correctly. This type of testing helps to specify exactly where something may have gone wrong as the tests are aimed at very specific functionalities. This can help to prevent code breaking when new changes are made. It can help to catch any bugs or errors in the code early. It can also increase code reliability as the developer may be more likely to use modular code.

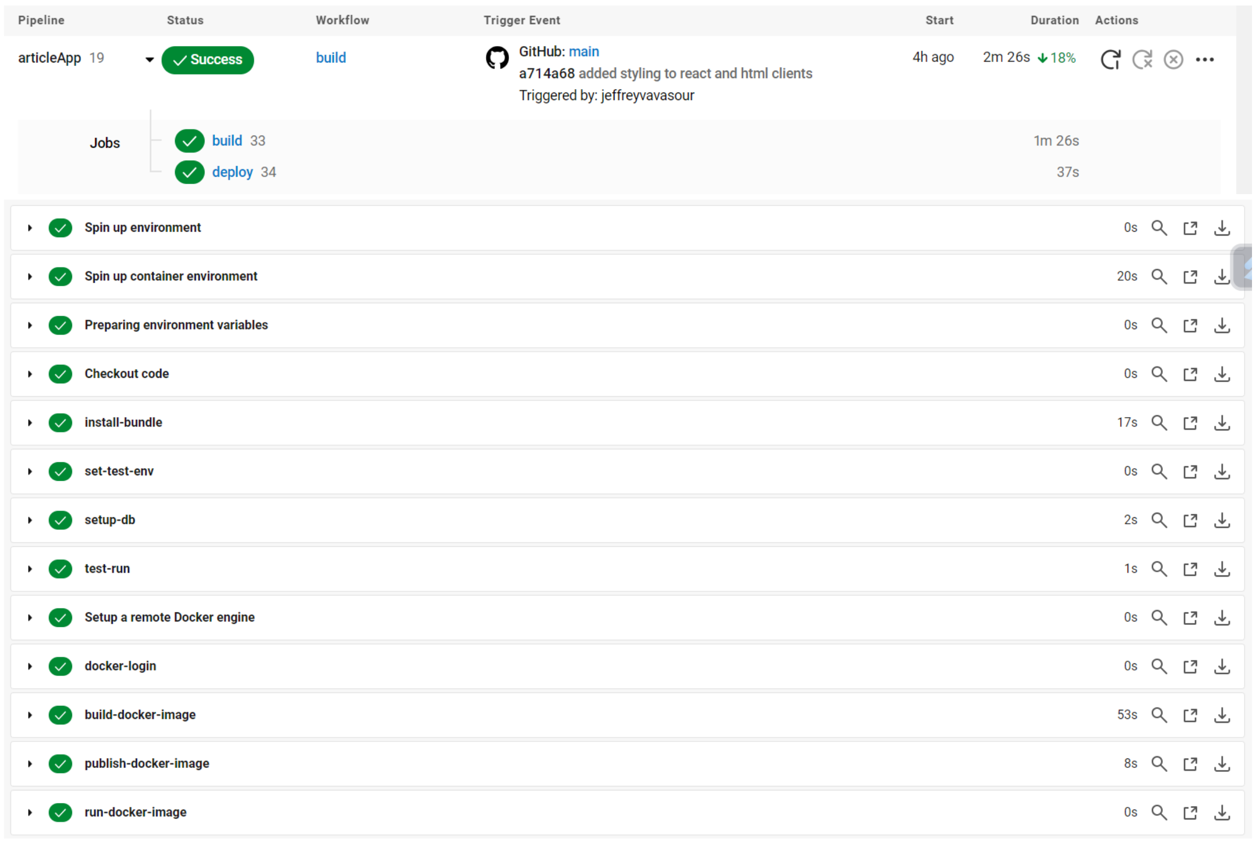
Integration testing

Integration testing focuses on how these parts work together. It checks that different pieces of the software interact correctly and that data flows smoothly between them. Integration testing ensures the whole system functions well together, helps maintain system stability, ensures different modules can communicate properly, and checks the performance of the system when multiple parts are working together.

Deployment

1. Created a GitHub repository
2. Connected local repo with GitHub repository
3. Added a .circleci folder with a config.yml file inside
4. Created a new project in circleci account connected to the github repository
5. Push current code to github to see if circleci is building and deploying
6. Create a Dockerfile to be executed with the docker build command
7. Create a new repository in Docker Hub account
8. Create environment variable in circleci for docker username and password and a secret key base
9. Navigate to Amazon Web Services (aws) to create an EC2 instance with ubuntu
10. Edit the inbound security rules to allow for IpV4 and IpV6 addresses on port 3000
11. Add environment variables in circleci for EC2 username and public DNS
12. Get access key and secret access key from aws and set these as environment variables in circleci
13. Add SSH which is used to access the EC2 instance into circleci
14. Create a deploy.sh file in application

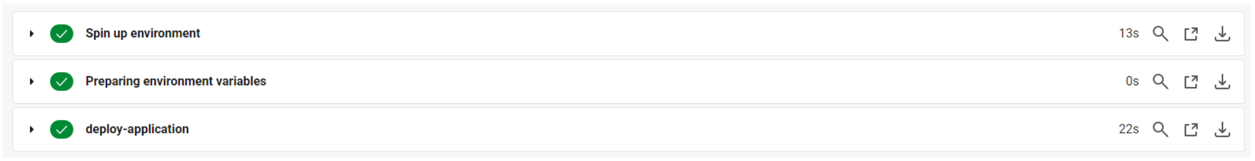
15. Add environment variables in circleci for container name and image name
16. Update the config.yml file which will be ran when circleci accepts it



Pipeline	Status	Workflow	Trigger Event	Start	Duration	Actions
articleApp 19	Success	build	GitHub: main a714a68 added styling to react and html clients Triggered by: jeffreyvavasour	4h ago	2m 26s ↓18%	

Jobs	
build 33	1m 26s
deploy 34	37s

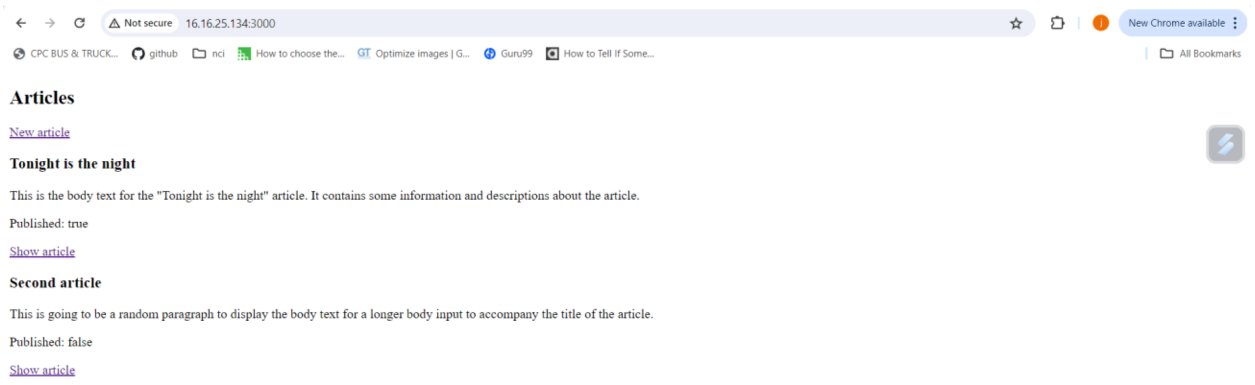
Spin up environment	0s
Spin up container environment	20s
Preparing environment variables	0s
Checkout code	0s
install-bundle	17s
set-test-env	0s
setup-db	2s
test-run	1s
Setup a remote Docker engine	0s
docker-login	0s
build-docker-image	53s
publish-docker-image	8s
run-docker-image	0s



Spin up environment 13s

Preparing environment variables 0s

deploy-application 22s



Not secure 16.16.25.134:3000
CPC BUS & TRUCK... github nci How to choose the... Optimize images | G... Guru99 How to Tell If Some...
All Bookmarks
New Chrome available

Articles

New article

Tonight is the night

This is the body text for the "Tonight is the night" article. It contains some information and descriptions about the article.

Published: true

Show article

Second article

This is going to be a random paragraph to display the body text for a longer body input to accompany the title of the article.

Published: false

Show article