Cloud Application Development CA

Pawel Wierzgon - 23222166

Higher Diploma in Science in Computing Information (HDWD_SEP23OL)

Applications Overview

Repository link: https://github.com/pawelwierzgon/CAD-CA

Backend Rails App

Steps taken to built the backend app:

1. I run the `rails g model Articles title:string body:text published:boolean` command to generate the Article model.
2. Then, in the db/migrate folder, I updated the migration file to set the default value of published field to false.

```ruby
class CreateArticles < ActiveRecord::Migration[7.1]
  def change
    create_table :articles do |t|
      t.string :title
      t.text :body
      t.boolean :published, default: false

      t.timestamps
    end
  end
end
```

3. After that, I run the migration to create the table using `rails db:migrate` command.
4. I used the `rails g controller Articles` to generate a controller file (and routes).
5. I modified the controller file to implement all the CRUD actions:

```ruby
class ArticlesController < ApplicationController
  before_action :set_article, only: [:show, :update, :destroy]

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

  # GET /articles/:id
  def show
    render json: @article
  end

  # POST /articles
  def create
    @article = Article.new(article_params)
    if @article.save
      render json: @article, status: :created
    else
      render json: @article.errors, status: :unprocessable_entity
    end
  end
```

```ruby
  # PATCH/PUT /articles/:id
  def update
    if @article.update(article_params)
      render json: @article
    else
      render json: @article.errors, status: :unprocessable_entity
    end
  end

  # DELETE /articles/:id
  def destroy
    @article.destroy
  end

  private

  # Allow finding articles by id
  def set_article
    @article = Article.find(params[:id])
  end

  # Only allow title, body, and published in request body
  def article_params
    params.require(:article).permit(:title, :body, :published)
  end
end
```

The Articles controller responds with an error if the data is not valid:

```ruby
if @article.save
  render json: @article, status: :created
else
  render json: @article.errors, status: :unprocessable_entity
end
```

6. To validate user requests, I added validation for the Article model that requires the request body to have both title and body:

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

7. For the CORS (Cross-Origin Resource Sharing) policy, I used rack-cors gem. To install it I used the gem install rack-cors command.
8. The rack-cors is configured in config/initializers/cors.rb file to allow get, post, patch, put, and delete requests for any resource (article) from any origin:

```
Rails.application.config.middleware.insert_before 0, Rack::Cors do
    allow do
        origins '*'
        resource '*', headers: :any, methods: [:get, :post, :patch, :put, :delete]
    end
end
```

9. To deactivate Cross site request forgery (CSRF/XSRF), as the backend will only work as an API, I added the following line to the application_controller.rb file:

```
class ApplicationController < ActionController::Base
    skip_before_action :verify_authenticity_token
end
```

10. To test the backend app, I created model tests in the test/models/article_test.rb file. It checks whether the app prevents saving the article without title:

```
test "should not save article without title" do
  article = Article.new(body: "Body of the article", published: false)
  assert_not article.save, "Saved the article without a title"
end
```

And without body:

```
test "should not save article without body" do
  article = Article.new(title: "Title of the article", published: false)
  assert_not article.save, "Saved the article without a body"
end
```

And if the article is successfully saved with the default published value set to false:

```
test "should save article with default published value" do
  article = Article.new(title: "Title of the article", body: "Body of the article")
  assert article.save
  assert_not article.published, "Published attribute was not set to false by default"
end
```

11. Before creating the controller test, I created the articles test fixture:

```
one:
  title: First Article
  body: Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  published: false
```

12. Once the fixture was in place, I created the articles_controller_test.rb file to test the CRUD operations of the backend app. It tests whether getting all/single article works, as well as the article creation, update, and removal.

Frontend HTML App

I started writing the vanilla html/css/js app by creating the **index.html** file with the boilerplate code. I included the main heading and some basic containers for the content. Once it was done, I proceeded with the **JS script** file that I linked in the html file. In the script file, I started by writing basic CRUD operations using fetch functions. Once I was happy with the results, I created a separate API module for these calls and placed it in the utils folder. I linked this extra js file in the html and imported it in the main script.js file.

Then, I used the querySelector method to find the articles container inside the html document. I created the **articles** array to store all the articles, **editMode** boolean variable to store the information whether the user is currently editing an article, and **editedArticleId** variable to store the id of the currently edited article.

After that, I focused on the main function (**generateArticleDiv**) to generate a single article div element. It takes an article as a parameter and creates all DOM elements, assigns the dataset values of the article id, class names, ids, innerText values, etc. This function checks also if a given article is being edited, and if so, it populates the article div with a different set of elements (input field, textarea, enabled checkbox for the published value, Save, and Cancel buttons). The function creates also all the buttons and adds the click event listeners to them. There is an extra client-side data validation inside these events. Finally, it appends all these elements together and returns an article div element.

Another important function added later in the code was the **refreshArticles** one. As the name suggests, it clears out the content of the **articles container** and then iterates through the **articles** array to append the article div (by calling the **generateArticleDiv** function). Finally, it creates a new button to create a new article and shows it on the page only if the user is not currently editing an article (**editMode** === false).

I realized that I repeat parts of my code when creating the **generateArticleDiv** function, so I wrote a helper function **addParagraph** that takes two arguments – **content** and **className** to return a new paragraph element with the provided content and class.

After some testing, I realized that the user inputs should be disabled when waiting for the API calls to respond. Therefore, I created the **disableInputs** function that can either disable all inputs (waiting for the API), enable inputs for a given article (API returned an error), or enable all buttons (API successfully created/updated/removed an article).

Finally, there is an **anonymous and asynchronous function** that is automatically called when the page is loaded to retrieve all articles and store them in the articles array. Once it is done, it refreshes the page to populate it with the article divs.

Simultaneously to writing the JS code, I was adding new styling elements to the **styles.css** file. I decided to keep the things simple and clear. I added a background gradient, specified sizes, margins, and paddings of the containers and inputs. Finally, I added basic div colors and div shadow.

Once the page was functional, I decided to write the tests. It was a bit challenging to find the right tools for the vanilla website testing but I opted for using **Jest**. I started with an **API utility module tests**. It checks the following functions:

- **getArticles()**
    - should return an array
- **createArticles()**
    - should create a new article
    - should return a new article
    - shouldn't create a new article when there is no title
    - shouldn't create a new article when there is no body
    - shouldn't create a new article when there is no title and no body
- **updateArticle()**
    - should update an article
    - shouldn't update an article when there is no title
    - shouldn't update an article when there is no body
    - shouldn't update an article when there is no title and no body
- **removeArticle()**
    - should return status 204 when removing an article
    - should return status 404 when the article does not exist

Then, I focused on the **end-to-end tests** in my **html.test.js** file. It uses the **Puppeteer** library to run the tests in the browser headless mode. It checks the following scenarios:

- Check if the page has the correct title
- Check if the page has the correct heading
- Check if the page has the new article button
- Check if when the new article button is clicked, the input fields appear
- Check if when the cancel button is clicked, the article div disappears

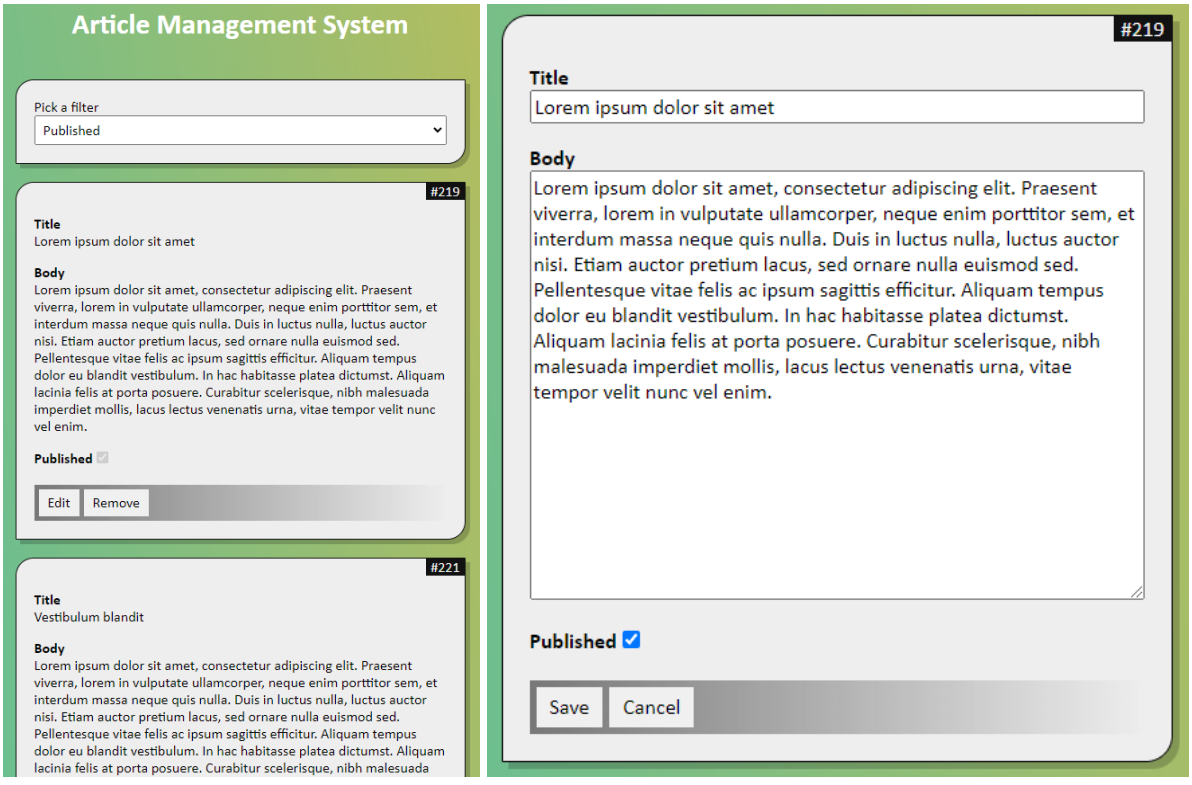
- Check if the new article is saved when the Save button is clicked
- Check if a dialog window is showed when the article title or body is missing
- Check if the edited article is saved
- Check if the article removal can be canceled
- Check if the article can be removed

Final test results:

```
Test Suites: 2 passed, 2 total
Tests:       22 passed, 22 total
Snapshots:   0 total
Time:        8.589 s, estimated 20 s
Ran all test suites.
```

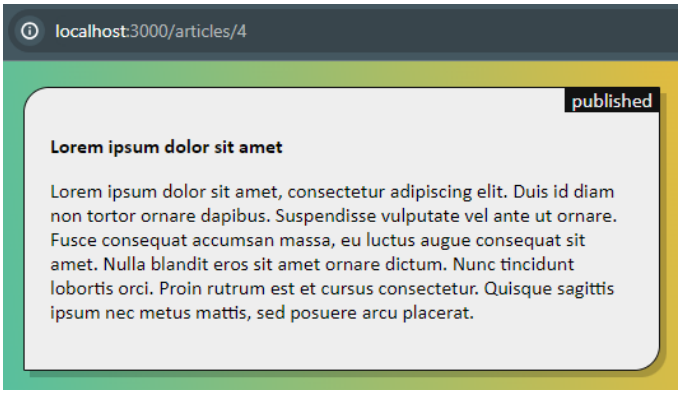
All these tests can be run from the frontend-html folder using **npm test** command. I used 3 more dependencies to be able to run these tests. One of them is **live-server** to start a local server hosting the index.html file. Another one is **concurrently** to run the backend app, frontend app, and tests at the same time. The last one is **wait-on** to wait for the backend to start before serving the frontend app and starting the tests.

Frontend React App



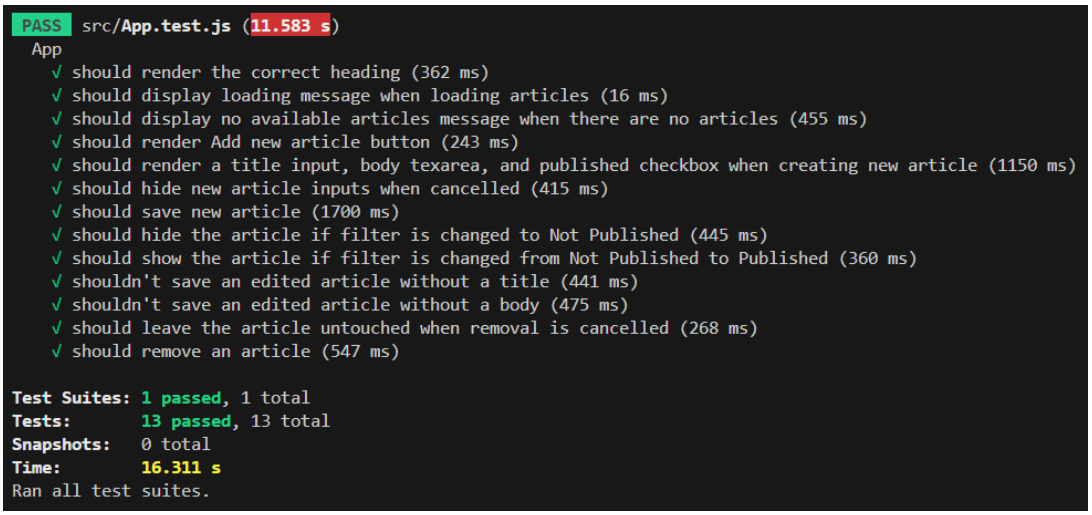
1. I started creating my React app with the `npx create-react-app frontend-react` command.
2. I decided to use Axios for the API calls. I used `npm i axios` to install it.
3. For the client-side routing I used React-Router-DOM library. I installed it with `npm i react-router-dom command`.

4. I began writing the app by creating the CRUD operations using **Axios** in the following 4 functions: **getArticles**, **getArticle**, **createArticle**, **updateArticle**, and **removeArticle**. Then, I created the **ARTICLE_ACTIONS** object to store the create, update, and remove functions inside in order to pass it as a prop.
5. Once it was done, I created the **Articles** view that generates an **Article** component for every article passed as an array in the props. Every **Article** component is provided with the following props: **article** (containing title, body, and published field values), **handleArticleChange** (function to update the input fields), **handleEditCancel** (function to restore the original value of the edited article), **toggleEditMode** (function to turn the edit mode on/off), **editMode** (current state to signify if the user is editing an article), **edited** (a boolean value based on the **editModeArticle** state to show the relevant DOM elements, depending if a given article is being edited), **actions** (object containing the aforementioned **ARTICLE_ACTIONS**), and **isUpdating** (a state used to disable the inputs when waiting for the API response).
6. The **Article** component renders the article div element with the relevant DOM elements depending on the current **editMode** and if a given article is being edited. If the article is being edited, it enables the published checkbox and replaces the span elements with the input/textarea fields. It also displays the Save and Cancel buttons.
7. Then, I went back to the **App.js** file to implement additional states, including **isLoading** (used to render loading information when waiting for the API), **isUpdating** (used to disable input fields when waiting for the API), and **error** (to display the **ErrorPopup** component with the error message).
8. Except the functions mentioned in step 5 that were created earlier, I added a **validateArticle** function to do the client-side validation before calling an API. It checks if the article object contains title and body. It also checks their type.
9. Once I had all these elements in place, I used the **useEffect** hook to fetch all the articles when the **App** gets rendered.
10. I created a new **SingleArticle** view that is using similar logic to pull the data from the API as the App.js. It uses the **useParams** hook to retrieve the article id from the URL. Then, it shows a simple, read-only version of the article.



11. I moved the **Axios** instance to the **index.js** file to keep it in one place for rendering both all articles and a single article view. Then, I imported the **React-Router-DOM** elements and created the app routes to render the relevant view or a div letting user know that a path does not exist.

12. Finally, I moved on to the filtering part of the app. I created the **Filter** component and conditionally rendered it above the articles. I modified the logic in the App.js file to render only the filtered articles instead.
13. I implemented similar CSS styling to the one created for the html version of the app with a few minor tweaks.
14. At that point, the app was ready for testing. I decided to use **Jest** library again. This time, it was already installed with the `npx create-react-app` command. I only had to update **@testing-library/user-event** library to test certain user events.
    I prepared the following tests:



    To run these tests, there has to be a running backend server and there has to be no articles saved in the DB. The tests can be started with the `npm test` command.

Deployment and Cloud Integration

There are several ways of deploying the app to a cloud provider such as AWS. Here are some example steps on how to build a simple DevOps pipeline using **GitHub**, **CircleCI**, and **AWS** virtual machine:

1. First, we need to create a **Git** repository and push our code to **GitHub**.
2. Then, we can focus on the **AWS EC2** instance
   - we can create a free instance using **t3.micro** type
   - we need to generate a **private** and **public key** to use **SSH** to log in into the machine
   - then, we need to modify the security inbound rules to **allow inbound requests** for the ports used by our apps (e.g. Custom TCP connection, port 3000 for IPv4 and IPv6).
3. Then, we can create a new project in **CircleCI**
   - first step is to connect CircleCI with our **repository**
   - then, we can specify the desired actions taken by CircleCI when the code is pushed into GitHub by editing the **config.yml** file
   - in **config.yml** file we can specify different jobs for our apps, including the docker image to user, the build of the React app (`npm run build`), testing (e.g. `npm test` or `rails test`), and deployment one that can use a bash script to run the desired command on the **AWS server**
   - it is a good idea to store certain values (like the SSH key or port numbers) as

variables for the security and flexibility of our pipeline
- finally, our **bash script** can finish the job on the AWS server by installing the required packages, stopping the existing processes, removing the existing directories (you may want to leave your current db intact!), cloning the repository (`git clone`), installing the required dependencies (`npm i` or `bundle install`), migrating any changes in the db (`rails db:migrate`), and starting the processes (e.g. `rails server`, `npm start`, `live-server`). We can run the processes simultaneously, by using processes managers such as **pm2**.

This is just one of the ways of deploying the app. It is also a good idea to use **Docker** to make sure the app works exactly the same in other environments. It is also easier to migrate and scale our app this way. You may also want to serve the page using the **HTTPS** protocol, which requires generating an **SSL** certificate, a domain, and an authority approval to validate it. To make the app even safer we should also amend the backend **CORS** policy.

Testing

Both unit testing and integration testing are crucial parts of the software development process. They serve different purposes but work together to ensure the quality and reliability of software systems.

Unit Testing

Unit testing involves testing individual parts of a software application separately. These parts are usually small and include things like functions, methods, or classes. Here are some important points about unit testing:

- **Isolation:** Unit tests are designed to isolate specific pieces of code, allowing developers to test each unit independently of other parts of the system. This isolation makes it easier to identify and fix bugs.
- **Early Detection of Defects:** Unit tests are usually written by developers during the coding phase, allowing them to catch defects early in the development process. This helps prevent issues from propagating to other parts of the system, reducing the overall cost of fixing bugs.
- **Documentation and Specification:** Unit tests serve as executable documentation for the behavior of individual units of code. They define the expected behavior of each unit, making it easier for developers to understand how the code should work.

Integration Testing

Integration testing focuses on testing the interactions between different components or modules of a software application. Unlike unit testing, which tests individual units in isolation, integration testing verifies that these units work together as expected when integrated into the larger system. Here are some key aspects of integration testing:

- **Identifying Interface Issues:** Integration tests help uncover issues related to the interactions between different components, such as incorrect data exchange, interface mismatches, or communication failures.

- **End-to-End Validation:** Integration tests validate the end-to-end functionality of the system by testing the integration points between various modules. This ensures that the system behaves as expected from the user's perspective.
- **Detecting Integration Bugs:** Integration testing helps detect bugs that may arise due to the integration of different modules or external dependencies. These bugs may not be apparent during unit testing, making integration testing essential for ensuring overall system reliability.

Benefits of Unit Testing and Integration Testing

- **Improved Software Quality:** By identifying and fixing defects early in the development process, both unit testing and integration testing contribute to higher software quality and reliability.
- **Reduced Debugging Effort:** Unit testing helps localize defects to specific units of code, making it easier to identify the root cause of issues. Integration testing ensures that these units work together seamlessly, reducing the effort required for debugging and troubleshooting.
- **Faster Time to Market:** By catching bugs early and facilitating rapid iteration, unit testing and integration testing help accelerate the development process, allowing software to be released to market more quickly.
- **Enhanced Maintainability:** Unit tests serve as living documentation for the codebase, making it easier for developers to understand and maintain the software over time. Integration tests provide assurance that changes to one part of the system do not break functionality in other areas.

In summary, unit testing and integration testing are essential components of the software development lifecycle, playing complementary roles in ensuring the quality, reliability, and maintainability of software systems. By incorporating both types of testing into the development process, teams can minimize defects, improve productivity, and deliver higher-quality software to end users.