Cloud Application Development (HDWD_SEP23OL)

Continuous Assessment Project

Dean Francis 22212761

Introduction

This project consists of three programs, a Ruby on Rails Application, a React Client, and a HTML Client.

The main functions if each of these programs is as follows:

- The Ruby on Rails application will mainly be used to support/provide the backend. This backend is used for the management of any articles that can be found within the project as a whole.
- The React Client and HTML Client both seek to provide an interactive, progressive web-based interface. Here a user can utilise the various methods and functions created to interact with the backend.

With regards to overall functionality, each of these three applications provides the user with CRUD Operations. Here, a user may choose to:

- View ALL articles stored within the backend.
- View the full details of a specific article found within the backend.
- Create a new article, which will be then stored within the backend and assessable to any of the three systems.
- Update an existing article store on the backend, which will then display all updated details on any of the systems.
- Delete an existing article store on the backend, which will then remove the article from all of the systems.

Attention has also been placed in this project on ensuring that each application utilises the appropriate HTTP requests and contains functional validation.

An example of validation implemented into all three of the application is form validation. More specifically, each application has been created to ensure that when a user attempts to create a new article via any system, they must enter both an article title and body in order to proceed.

Ruby on Rails Application

Start-up

An important point to note is that the Rudy on Rails Application has been set-up and configured to be ran on - http://localhost:4000/articles.

rails server --binding 0.0.0.0 -p 4000 (See Images for further details).

Unit Testing & Integration Testing

Model Unit Testing:

Unit tests have been used in this project to test single pieces of the applications code in isolation. More specifically, their goal is to examine the applications methods/functions and ensure that they are working as intended.

The project file (article_test.rb) seeks to test the specific unit methods/functions of the Article model. Direct focus was placed on examining both the applications CRUD operations (Saving, finding, deleting, and updating articles), as well as, looking at the display output when published is equal to true or false.

Each of the test written has been set in isolation of one another. As a result of this, each test is performed in an independent environment, not relying on either the previous or next test. All object which are necessary for the testing to run are declared within the setup method, located at the beginning of the testing code.

Additionally, no external dependencies are required for the testing. The tests instead utilise the created fixtures (located in test/fixtures.articles.yml - test_article_published_task and test_article_not_published_task).

Furthermore, due to the lack of any reliance on external dependencies, the written tests can be executed rapidly to reveal if they were successful or not.

The command line commands for the tests were as follows:

- rails test
- rails test:models

Controller Actions and Testing:

Controller testing has also been used in this project to analyse the behaviour of controller actions, ensuring that they worked as intended, handling requests in the correct manner and rendering the necessary responses.

To begin, a setup method was created to set up the testing environment. Here article is assigned to functional_test from fixtures. By default, a number of controller actions have been created. These focus on the CRUD operations for the applications articles.

An additional test action has been created called testpage, which will test if the test page of the application is accessible to the user. The command line commands for the tests were as follows:

- rails test
- Test page accessed via /testpage (For example: http://localhost:3000/testpage & http://localhost:4000/testpage).

Testing Framework – Minitest:

The testing framework chosen for use with this application is Minitest. Minitest is the default testing framework which is provided by Ruby on Rails. Minitest was chosen due to it being generally considered simple to use and it being directly integrated into Ruby on Rails.

Additionally, the tests undertaken comply with the Minitest framework, for example:

- Naming conventions – articles_controller_test.rb
- Inheritance - ArticlesControllerTest inherits from ActionDispatch::IntegrationTest.
- Methods – Defined using the “Test” keyword

Integration Testing:

Integration testing has been implemented into the Ruby on Rails Application. More specifically, four tests have been created:

- CreateArticleWorkflowTest: The objective of this test is to check the workflow for creating a new article. Here the new article page is visited, new article form is submitted, a check is made to ensure successful creation, and then verified that the new article is being displayed correctly.

- DeleteArticleWorkflowTest: The objective of this test is to check the workflow for deleting an existing article. Here a list of all articles is retrieved, a specific article is deleted, a check is made to ensure successful deletion, and then verified that the deleted article is no longer displayed.

- RetrieveArticlesTest: The objective of this test is to retrieve all articles. It is essentially the foundation with which the other integrations tests are built on top of. Here it retrieves a list of all the articles created and then checks if those articles are being displayed correctly.

- EditArticleWorkflowTest: The objective of this test is to check the workflow for editing an already existing article. Here it visits the edit page for a specific article, submits a form to edit the article, a check is made to see if the article has been edited successfully, and then verifies that the edit article changes are being displayed correctly.

(See Images for screenshots of tests ran).

React Client

Start-up

The react client has been set up and configured to be ran on http://localhost:3000/ (See Images for further details). Additionally, the Ruby on Rails application must be running on http://localhost:4000/articles in order for the React Client to access the backend.

Unit Testing & Integration Testing

Component Unit Testing:

Unit testing has also been implemented in the React Client. These tests have been created to ensure that the applications CRUD operations work as they are expected to.

The tests are performed in the file (Article.Test.js), they include:

- Successful rendering: Checking that articles render correctly (Props and titles are rendered along with checking view details is displayed initially and hide details is not).

- Show details functionality: Checking the show details button works as intended when clicked (Simulates a click on the button and expects the edit and delete to then be displayed)
- Edit article functionality: Checking articles with specific props are rendered (Simulated a click on view details and edit, changes title/body and saves changes – calling the onEdit function).
- Delete article functionality: Checking articles with specific props are rendered (Simulated a click on view details and delete – calling the onDelete function).

Testing Framework – Jest and React Testing Library:

The testing framework which is being used here is a combination of Jest and React Testing Library. Jest was chosen due to its general simplicity when it comes to testing structure and the running of tests. While React Testing Library was used to perform the tests in a similar manner to that which the user would do, testing React components using utilities such as fireEvent and waitFor.

In the React Clients testing, Jest is used to organise the article tests into a test suite and to make expect assertions. React Testing Library is being used to render components (fireEvent – simulating user interactions i.e. clicks and waitFor – for asynchronous testing). Tests were conducted via running npm test on the command line.

Integration Testing:

Integration testing has been implemented into the React Client to ensure that the article components behaves in the correct manner when a user interacts with them. More specifically, five tests have been created:

- calls onEdit with updated data when article is edited: The objective of this test is to verify that when a user edits one of the articles and saves its changes, the onEdit function is called and the data then updated.

- calls onDelete with article id when delete button is clicked: The objective of this test is to verify that when a user deletes one of the articles, the onDelete function is called with that's articles id.

- does not call onEdit when cancelling edit: The objective of this test is to verify that when a user cancels the editing of one of the articles, the onEdit function is not called.

- does not call onDelete when cancelling delete: The objective of this test is to verify that when a user cancels the deleting of one of the articles, the onDelete function is not called.

- handles empty title or body correctly: The objective of this test is to verify that the component will handle the empty title or body in the correct way, rendering the article component with empty title and body props and then ensure the save button is not rendered.

(See Images for screenshots of tests ran).

HTML Client

Overview

Tests results can be viewed via the testresults.html page. Additionally, the Ruby on Rails application must be running on http://localhost:4000/articles in order for the HTML Client to access the backend.

Unit Testing & Integration Testing

JavaScript Unit Testing:

Unit tests have been created for the HTML Client. More specifically, for the JavaScript section. Mock functions were created to simulate the applications CRUD operations. These functions are placed within a try block, where they are executed, and their respective results are stored. Each of these test results is then displayed to the user in the testresults.html page. There are a total of 4 tests being conducted, one for each of the CRUD operations (Adding an article, getting an article, updating an article, and deleting an article). The goal of these tests is to ensure that the applications CRUD operations behave correctly and independent of each other.

Testing Framework – Custom JavaScript:

The testing framework used for the HTML Client was a custom framework where JavaScript code is used. This was chosen in order to reduce any dependency on external libraries for testing.

Essentially, the framework follows what has been generally covered in the module for rails testing:

1. Describe what is being tested: Testing of CRUD operations.
2. Set up a starting point for testing: Starting tests with an empty list of articles.
3. Test what should happen: Creating a test for each operation.
4. Check if successful: What is expected if the test passes/fails. (Add an Article should result in one additional article in the list).
5. Run tests: Run tests one after the other.
6. Show results: Display result of the tests. If any fails show an error message.

Integration Testing:

Finally, integration testing has also been utilised in the HTML Client. The goal of these tests is to check that the created functions behave as they were intended to when a user interacts with them. For the HTML Client integration testing, four tests have been created:

- Data Loading Integration Test: The objective of this test is to ensure that the applications data is being loaded in successfully. This is achieved by getting the initial count of articles, simulating the loading of additional articles, checking if the count of articles has increase after the loading of new articles, displaying the test result to the html page.

- Article Addition Integration Test: The objective of this test is to check if a new article has been added successfully. This is checked similarly to the above, getting initial article count, adding a new article by simulating user interaction, checking if article count has increased by one and then displaying the result.

- Article Update Integration Test: The objective of this test is to check if an article is updates successfully. First the updated data is defined, then the last article is updated (through the saveEdit), the last article is then retrieved, it is then checked to see if its data now matches the initially defined data, result is then displayed.

- Article Deletion Integration Test: The objective of this test is to check if an article is deleted successfully. Here, the initial article count is retrieved, the last article is deleted, the now current count of articles is then retrieved, a check is made to see if the count has reduced by one after deletion, the results are then displayed.

(See Images for screenshots of tests ran).

Importance of Unit Testing and Integration Testing:

Unit Testing:

Unit testing places it focus on the testing of individual components, otherwise known as units, of the applications code. It conducts this testing of these units in isolation of one another. These units tested include models, controllers, and components.

Utilizing Unit testing is crucial in application development. It aid in identifying potential faulty code early in the development by ensuring units work as intend. Through unit testing, the purpose of each code unit is documented, allowing for a universal understand among developers on units intended use.

Through conducting Unit Testing, specific components of applications are tested in isolation. This results in faster debugging of flawed code. It also promotes the idea of DRY programming, don’t repeat yourself, due to it relying on developers to create much more modular code that is generally easier to read and maintain.

Integration Testing:

Integration Testing puts focus on testing the way in which various parts of the application would be used by a user. It is important in the context of developing applications, as it ensures that the application as a whole works as it was intended, by testing each individual components integration with one another and also how they interact with each other. Additionally, it will also be able to catch and highlight when these interactions face any errors or issues.

By incorporating Integration Testing developers ensure that the application functionality is executed flawlessly. Furthermore, it also can prove to be a great asset to use in conjunction with Unit Testing, potentially being capable of identifying previously missed system issues that unit testing did not catch.

Deployment & Cloud Integration

Steps involved in deploying applications:

When considering the steps which are involving in deploying applications to the cloud, it is important to first highlight the tools which are required to ensure its success. The following is an example of the tools which are required for deploying an application to the AWS cloud provider.

- Atom: Atom is a text editor, which in this projects case, has been used for writing the three applications.
- Git & GitHub: Is a code storage service, which utilises repositories in which applications and code can be sorted.
- CircleCI: Is a service that facilitations automatic application building and continuous delivery.
- Amazon EC2 instances and Security Groups: The AWS cloud provider allows the applications to be ran and be accessible over the cloud. Furthermore, it provides a variety of security measures, such as port restrictions, to ensure safe accessibility.
- Docker: The Docker Hub is installed onto the AWS EC2 Instance and allows for the publish of integrated images.

Now let's consider the steps involved in deploying the applications:

Step 1 – Application & Code Creation (Atom):

Firstly, in order to deploy any application, it must first actually be written. In this project for example, the Ruby on Rails Application, React Client, and HTML Client have all been written via Atom.

Step 2 – Local & Web Repository Creation (Git & GitHub):

Next, these created applications need to be stored within repositories. To begin, Git (A popular version control system) is used to store the code on a local repository. Doing this provides a number of benefits, such as allowing developers to manage and maintain code changes that are made during the applications development. Once stored on a local repository, to code can then be pushed to an online platform, such as GitHub. Here, the applications code can be accessed online by anyone who has been provided with access.

Step 3 – Building Applications Automatically (CircleCI):

Moving on, the GitHub repository is linked to an automation software service, such as CircleCI. CircleCI services facilitates the building, testing, and deploying of project applications automatically. Furthermore, once a repository is linked, CircleCI observes and detects any code changes being pushed. Following this its runs any jobs or tests it has been set out to do in files such as config.yml.

Step 4 – Deploying Applications Automatically (CircleCI):

Adding on top of the previous step, in addition to building & integration, CircleCI also provides automatic delivery and deployment capabilities. To achieve these capabilities, files such as config.yml and deploy.sh are used to tell CircleCI the steps required for deployment.

Step 5 – AWS EC2 Instance Start-Up and Running (AWS):

Next, Amazons web-based cloud computing platform, otherwise known as AWS, is utilised. One of the many services AWS offers is what's called an EC2 Instance, essentially it is an instance of a virtual could server. By using a EC2 Instance any of the applications can be ran and also be accessible over the cloud. Furthermore, in a step below, you'll see more important features it provides.

Step 6 – Docker & Docker Hub integration (Docker Hub):

Another service which is used is Docker and Docker Hub. At their essence, these platforms place the applications into packages, so that they can be managed easier, deal with their scaling and also assist in deployment. These packages are often referred to as containers. Breaking it down even further, these containers are individual pieces of software that contain all of the code required for the application to run.

Step 7 – AWS Security Layers (AWS):

Finally, at this point the applications can be integrated, deployed, and delivered. Furthermore, they would be running over the cloud on an EC2 instances that anyone can access. However, now comes the issue of access itself. With no security or restrictions in place the application is open to exploitation. Now the other features provide by AWS are considered. Some of these features include:

- Port Restrictions: Which involves managing the rules of the AWS Security Groups. Essentially, port restrictions can control both the inbound and outbound traffic to the application by setting the port access to specific ports i.e. port :8443 or 8080.
- Secure Hypertext Transfer Protocol (HTTPS): Which seeks to add an additional layer of security to the standardly used HTTP via SSL encryption (Installing OpenSSL onto the EC2 Instance).

It is through the steps above that any application, be that a Ruby on Rails Application, React Client, or HTML Client, can be deployed onto a cloud provider (In this case AWS), using automation software, containerisation, and security measures.

Configurations & considerations needed during devolvement for proper functioning:

In order to ensure that the application functions as it was intended to once it has been deployed, there are several considerations to be looked at and configurations settings to be managed.

GitHub Repository: In order to have the application be accessible over the cloud, it first needs to be pushed to GitHub. As a result, this means a GitHub account is required and a related repository created.

Configuration files: If the application is utilising a service such as CircleCI, it will require a config.yml file, for setting out certain integration and deployment instructions.

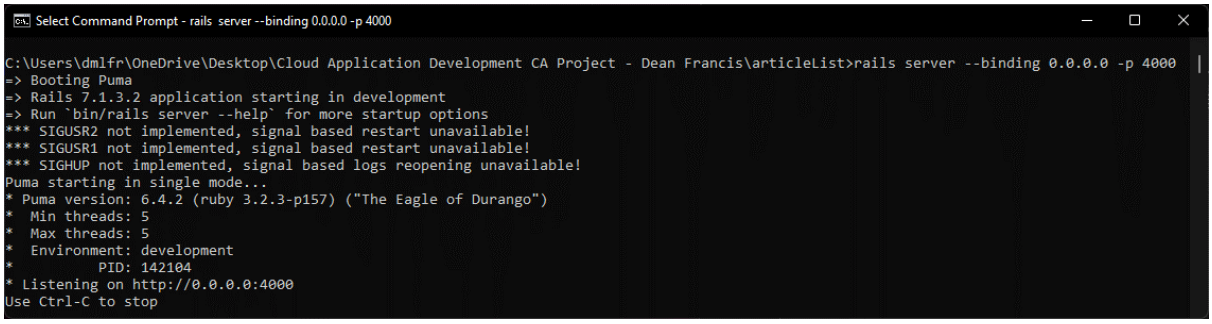
Deployment files: Additionally, in the case of automatic deployment via CircleCI, the project will also require a deploy.sh file, for setting out the steps necessary for deployment over the cloud.

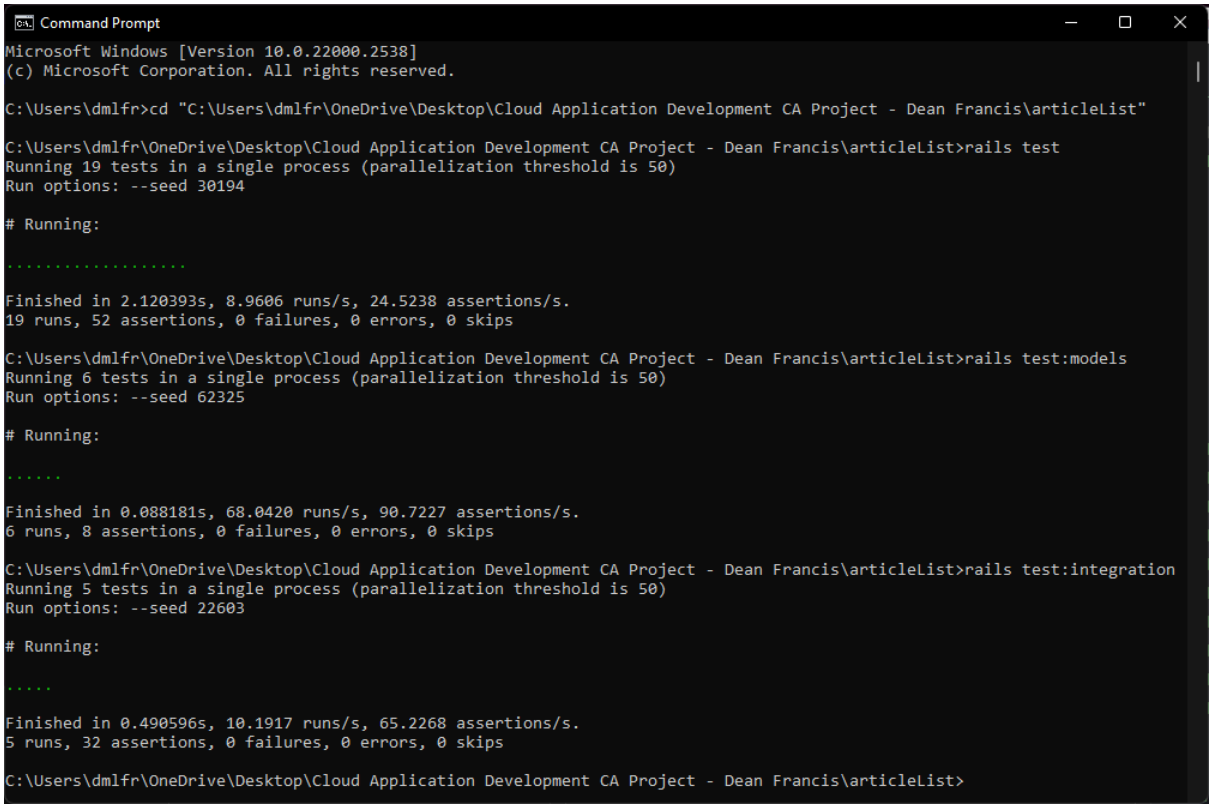
Docker files: If the project is utilising methods such as containerisation via Docker and Docker Hub it will require files to ensure success, such as .dockerignore and Dockerfile.

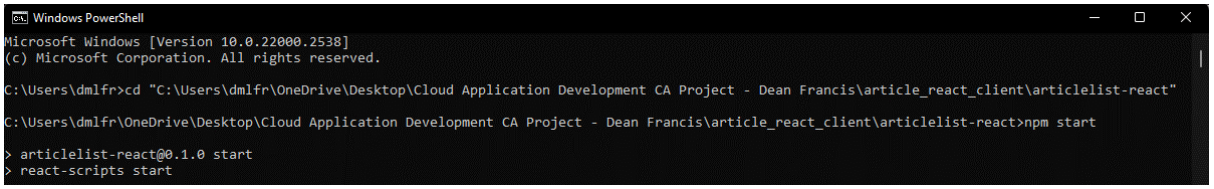
Environment Variables and SSH Keys: Integration and deployment also requires the use of sensitive data such as public and private keys. These keys include those created by the EC2 Instances and Docker: SSH PUBLIC & PRIVATE KEYS/AWS_ACCESS_KEY/AWS_SECRET_ACCESS_KEY/SECRET_KEY_BASE/EC2_USERNAME/EC2_PUBLIC_DNS/DOCKER_USERNAME/DOCKER_PASSWORD/CONTAINER_NAME/IMAGE_NAME

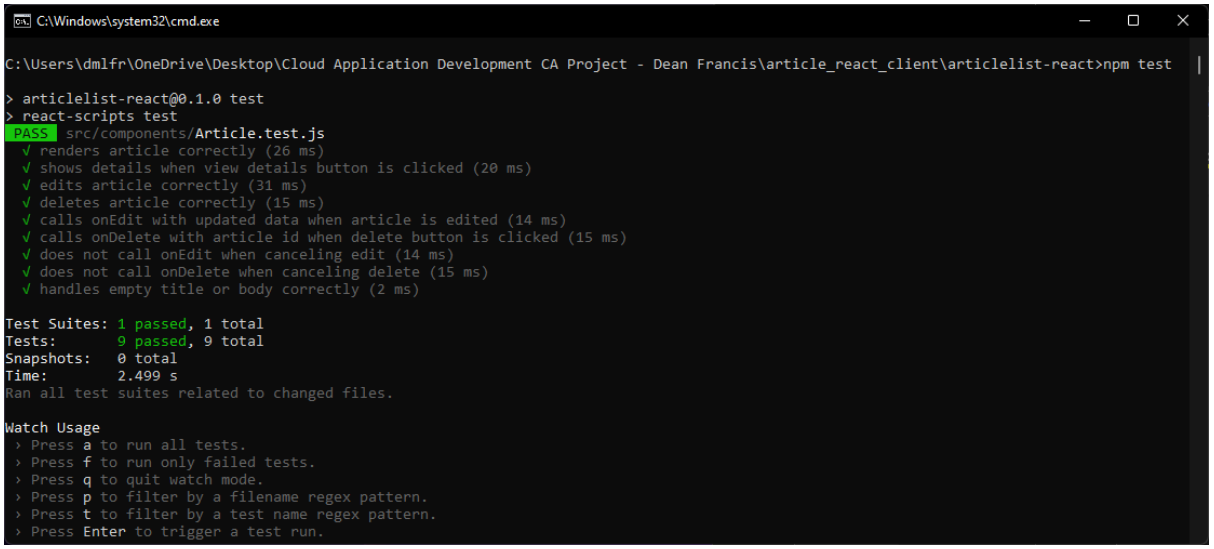
Accessing the application: If port restrictions have been put in place, developers should ensure that the correct port usage is communicated, for example, if application access is limited to port 8080 then an address like http://12.345.67.891:8080/ would be required.

Images:









Unit & Integration Test Information

Framework: Custom JavaScript

Tests Performed: CRUD Operations

How Tests Were Done: Tests were written using JavaScript functions to check the functionality of each CRUD operation.

Files Used: index.html, script.js, test.js

Unit Test Results:

Add Item Test:

This test checks whether the `addItem` function successfully adds an item to the articles array.

Result: Passed

Get Item Test:

This test checks whether the `getItem` function successfully retrieves an item from the articles array.

Result: Passed

Update Item Test:

This test checks whether the `updateItem` function successfully updates an item in the articles array.

Result: Passed

Delete Item Test:

This test checks whether the `deleteItem` function successfully deletes an item from the articles array.

Result: Passed

Integration Test Results:

Data Loading Integration Test:

This test checks whether data is loaded successfully and the article count is updated accordingly.

Result: Passed

Article Addition Integration Test:

This test checks whether a new article is successfully added to the articles array.

Result: Passed

Article Update Integration Test:

This test checks whether an article is successfully updated in the articles array.

Result: Passed

Article Deletion Integration Test:

This test checks whether an article is successfully deleted from the articles array.

Result: Passed

Back to Application