

CT071-3-3-DDAC - DESIGNING & DEVELOPING CLOUD APPLICATIONS

Mark Jeremy Essa

TP029006

UC3F1603SE

# Table of Contents

Contents

[Table of Contents 2](#_Toc143787098)

[Introduction 5](#_Toc143787099)

[Implementation 6](#_Toc143787100)

[Cloud Services Deployment Screenshot and discussions 6](#_Toc143787101)

[AWS RDS Deployment and usage 6](#_Toc143787102)

[AWS EC2 Implementation and discussions 8](#_Toc143787103)

[Code Snippets for each function related to cloud services 9](#_Toc143787104)

[Database implementation from the backend (DB Related): 9](#_Toc143787105)

[Backend code snippet for CRUD regarding Users (DB Related) 9](#_Toc143787106)

[Backend Code Snippet for Products (DB Related) 12](#_Toc143787107)

[Backend Code Snippet for Orders (DB Related) 13](#_Toc143787108)

[Backend Code Snippet for Order Items (DB Related) 14](#_Toc143787109)

[UI Code Snippet for Login from data from EC2 16](#_Toc143787110)

[UI Code Snippet for getting products from EC2 Instance 17](#_Toc143787111)

[UI Code Snippet for fetching user orders from EC2: 18](#_Toc143787112)

[User Manual 19](#_Toc143787113)

[Landing Page 19](#_Toc143787114)

[Log In Page 19](#_Toc143787115)

[Sign In Page 20](#_Toc143787116)

[Products Page 20](#_Toc143787117)

[Products Page (Cont.) 21](#_Toc143787118)

[Products Page (Cont.) 22](#_Toc143787119)

[Cart 22](#_Toc143787120)

[Cart (Cont.) 23](#_Toc143787121)

[Checkout Page 23](#_Toc143787122)

[Orders Page 24](#_Toc143787123)

[Orders Detail Page 24](#_Toc143787124)

[Profile Page 25](#_Toc143787125)

[Test Plan 26](#_Toc143787126)

[Unit & Integration Tests 26](#_Toc143787127)

[Objective: 26](#_Toc143787128)

[Scope: 26](#_Toc143787129)

[Testing Approach: 26](#_Toc143787130)

[Test Environment: 27](#_Toc143787131)

[Test Scenarios: 27](#_Toc143787132)

[Test Execution: 28](#_Toc143787133)

[Test Reporting: 28](#_Toc143787134)

[Conclusion: 28](#_Toc143787135)

[Performance Testing 28](#_Toc143787136)

[Objective: 28](#_Toc143787137)

[Scope: 29](#_Toc143787138)

[Performance Testing Types: 29](#_Toc143787139)

[Test Environment: 29](#_Toc143787140)

[Performance Test Scenarios: 29](#_Toc143787141)

[Performance Metrics: 30](#_Toc143787142)

[Test Execution: 30](#_Toc143787143)

[Test Reporting: 30](#_Toc143787144)

[Conclusion: 30](#_Toc143787145)

[System Limitation 31](#_Toc143787146)

[Conclusions & Reflections 33](#_Toc143787147)

[Bibliography 34](#_Toc143787148)

# Introduction

In today's rapidly evolving business landscape, the ability to adapt to change and to be able to leverage whatever technology that is available is essential for sustained growth and success. This project is done in such a way that it puts a cake and pastries shop on a journey to address the business growth challenges faced by Lil-Lucia Cake Shop which is what the bakery shop is named, a local bakery renowned for its delectable confections and handmade ice-cream. By harnessing the power of modern technologies and the AWS cloud, this initiative aims to provide a comprehensive solution to propel Lil-Lucia Cake Shop into a new phase of expansion and prosperity.

Lil-Lucia Cake Shop has established itself as a beloved local bakery, celebrated for its exceptional cakes, pastries and ice-cream. However, in the face of increasing competition, changing customer preferences, and the recent challenges posed by the Covid-19 pandemic, the bakery is confronting new challenges to its growth trajectory. The number of customers coming in to physically purchase their goods, event orders have declined, and there's an urgent need to reinvigorate the business model to thrive in the digital age.

This project centers on developing a cloud solution by using what AWS services are available for the simple implementation of the website, where Node.js for the backend, React for the frontend, and MySQL for the database. By leveraging cloud architecture, the Lil-Lucia Cake Shop will gain access to a flexible and scalable infrastructure that can accommodate varying levels of demand, ensuring seamless user experiences even during peak times.

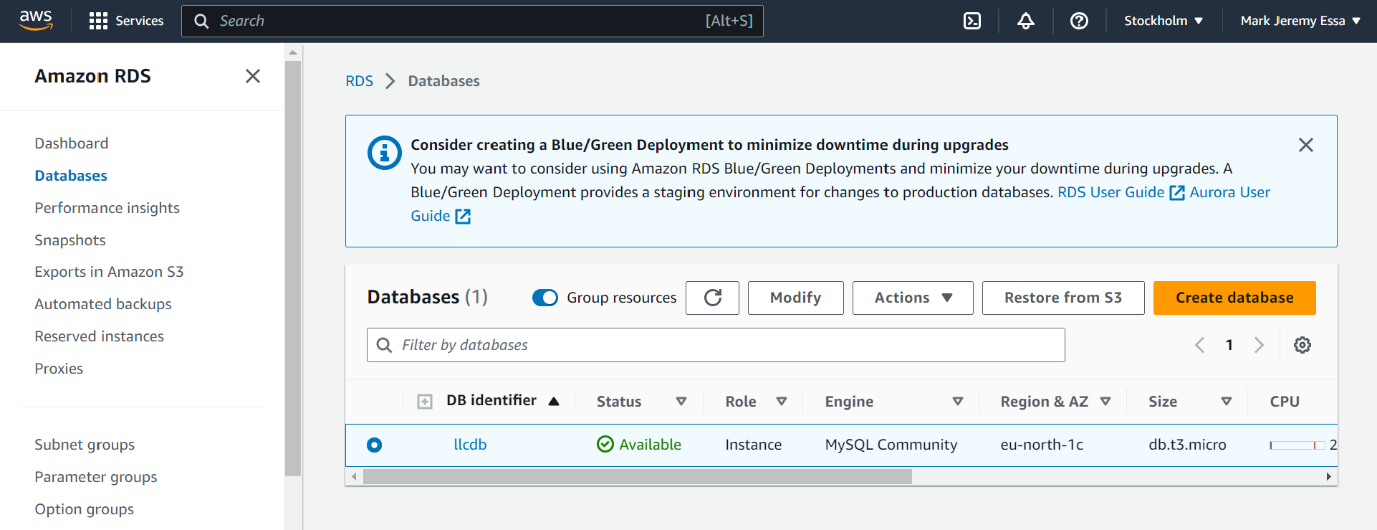
The development of an AWS cloud solution utilizing Node.js, React, and MySQL presents an exciting opportunity to transform Lil-Lucia Cake Shop's trajectory in the current market. This project embodies the spirit of innovation and adaptability, showcasing how technology can be harnessed to not only solve business challenges but also pave the way for growth, resilience, and continued delight for customers and enthusiasts alike.

# Implementation

## Cloud Services Deployment Screenshot and discussions

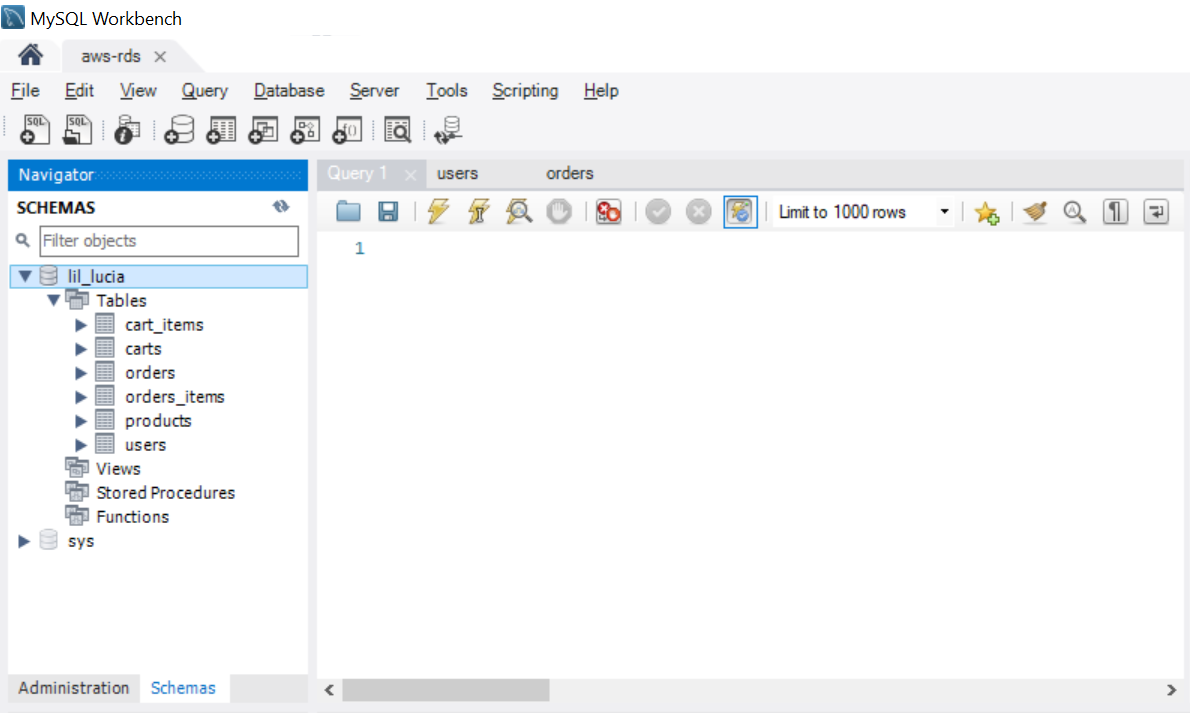
### AWS RDS Deployment and usage

For the implementation of the database portion of this project, AWS RDS MySQL was utilized. The selection for this is as AWS quotes “As a managed service, Amazon RDS provides a high level of security for your MySQL databases. These include network isolation using Amazon Virtual Private Cloud (VPC), encryption at rest using keys you create and control through AWS Key Management Service (KMS) and encryption of data in transit using SSL.” Though in this case, we will not be using the KMS as currently the implementation is meant as an initial release and to see how the customers would react to the shop going online. The screenshots are shown below of the implementation of RDS.



For us to access this instance we are using MySQL Workbench,

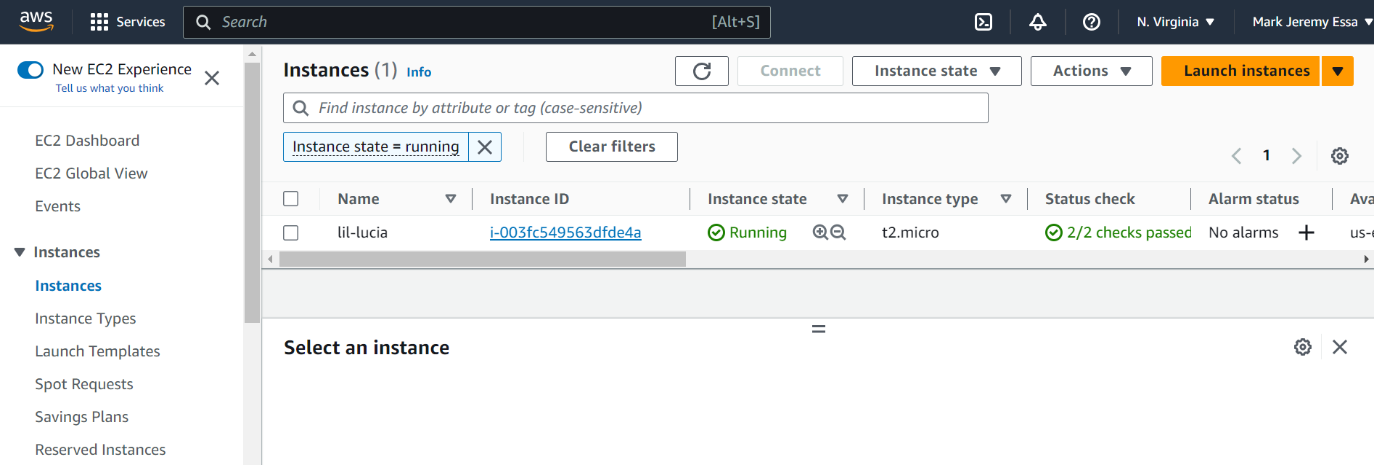




For the code to be able to access the database on the other hand is through the implementation of sequelize from the Node.js portion of code where we use a specific user to access, create, modify and delete entries from the database as well.

### AWS EC2 Implementation and discussions

For the implementation of the EC2 instance on the other hand is the deployment of the backend portion of the code where only the backend is implemented on the cloud as for now the UI is still under consideration of implementing a custom server or if the shop would like to implement further implementation on the cloud using S3 and CloudFront from AWS as of the discussion of the owners of Lil-Lucia is still ongoing. The screen captures of the implementation of the EC2 instance is as below:



How I have deployed the code onto this EC2 instance on the other hand is accessing the instance and then installing what is needed to pull the code repository from GitHub which is the chosen version control for this project in which is then deployed onto the EC2 instance. Making sure the code is updated is via the usage of git available on the server as well.

## Code Snippets for each function related to cloud services

### Database implementation from the backend (DB Related):

|  |
| --- |
| const Sequelize = require("sequelize");  const sequelize = new Sequelize("lil\_lucia", "admin", "adminPass", {  host: "llcdb.cxmvq8oh8y1x.eu-north-1.rds.amazonaws.com",  port: 3306,  dialect: "mysql"  });  module.exports = sequelize; |

### Backend code snippet for CRUD regarding Users (DB Related)

|  |
| --- |
| const signup = async (req, res, next) => {  const { FirstName, LastName, Address, Email, Password } = req.body;  const identifiedUser = await Users.findOne({ where: { Email: Email } }).catch(  (errors) => {  return next(new HttpError(errors.message, 500));  }  );  if (identifiedUser) {  return next(  new HttpError("User already exists, please log in instead", 422)  );  }  const createdUser = await Users.create({  FirstName,  LastName,  Address,  Email,  Password,  IsAdmin: false,  }).catch((err) => {  return next(new HttpError(err.message, 422));  });  res.status(201).json({ user: createdUser });  };  const login = async (req, res, next) => {  const errors = validationResult(req);  if (!errors.isEmpty()) {  return next(  new HttpError("Invalid inputs passed, please check your data", 422)  );  }  const { Email, Password } = req.body;  const identifiedUser = await Users.findOne({ where: { Email: Email } }).catch(  (errors) => {  return next(  new HttpError(  "There is an issue with the log in, please try again",  500  )  );  }  );  if (!identifiedUser || identifiedUser.Password !== Password) {  return next(  new HttpError("There is an issue with the log in, please try again", 401)  );  }  res.json({ message: "Logged in successfully", user: identifiedUser });  };  exports.signup = signup;  exports.login = login; |

### Backend Code Snippet for Products (DB Related)

|  |
| --- |
| const getProductById = async (req, res, next) => {  const productId = req.params.pid;  const foundProduct = await Products.findByPk(productId).catch((err) => {  return next(new HttpError("Product is not found", 422));  });  if (!foundProduct) {  return next(new HttpError("Product is not found", 422));  }  res.status(200).json({ product: foundProduct });  };  const getProductByCategory = async (req, res, next) => {  const category = req.params.category;  const foundProduct = await Products.findAll({  where: { Category: category },  }).catch((err) => {  return next(new HttpError("Products are not found", 422));  });  if (foundProduct.length <= 0) {  return next(new HttpError("Products are not found", 422));  }  res.status(200).json({ products: foundProduct });  };  exports.getProductById = getProductById;  exports.getProductByCategory = getProductByCategory; |

### Backend Code Snippet for Orders (DB Related)

|  |
| --- |
| const getAllOrders = async (req, res, next) => {  const allOrders = await Orders.findAll();  if (allOrders.length <= 0) {  return next(new HttpError("Could not find any orders", 404));  }  res.status(200).json({ orders: allOrders });  };  const getOrdersByUser = async (req, res, next) => {  const userId = req.params.uid;  const foundOrders = await Orders.findAll({  where: { UserID: userId },  }).catch((err) => {  return next(new HttpError("No orders were found", 422));  });  if (foundOrders.length <= 0) {  return next(new HttpError("No orders were found", 422));  }  res.status(200).json({ orders: foundOrders });  };  exports.getAllOrders = getAllOrders;  exports.getOrdersByUser = getOrdersByUser; |

### Backend Code Snippet for Order Items (DB Related)

|  |
| --- |
| const getOrderItemsByOrderID = async (req, res, next) => {  const orderId = req.params.oid;  const foundOrderItems = await orderItems.findAll({  where: { OrderID: orderId },  }).catch((err) => {  return next(new HttpError("No order items were found", 422));  });  if (foundOrderItems.length <= 0) {  return next(new HttpError("No order items were found", 422));  }  res.status(200).json({ orderItems: foundOrderItems });  };  const createOrderItem = async (req, res, next) => {  const errors = validationResult(req);  if (!errors.isEmpty()) {  throw new HttpError("Invalid inputs passed, please check your data.", 422);  }  const { OrderID, ProductID, Quantity, Subtotal } = req.body;  const createdOrderItem = await orderItems.create({  OrderID,  ProductID,  Quantity,  Subtotal  }).catch((err) => {  next(new HttpError(err.message, 422));  });  res.status(201).json({ createdOrderItem: createdOrderItem.dataValues });  };  exports.getOrderItemsByOrderID = getOrderItemsByOrderID;  exports.createOrderItem = createOrderItem; |

### UI Code Snippet for Login from data from EC2

|  |
| --- |
| const authenticationHandler = async (e: FormEvent<HTMLFormElement>) => {  e.preventDefault();  if (isLogin) {  try {  const responseData = await sendRequest(  "http://ec2-54-80-72-110.compute-1.amazonaws.com:5000/api/user/login",  "POST",  JSON.stringify({  Email: formState.inputs.email.value,  Password: formState.inputs.password.value,  }),  { "Content-Type": "application/json" }  );  auth.login(responseData.user as User);  } catch (error) {}  } else {  try {  const responseData = await sendRequest(  "http://ec2-54-80-72-110.compute-1.amazonaws.com:5000/api/user/signup",  "POST",  JSON.stringify({  FirstName: formState.inputs.firstName.value,  LastName: formState.inputs.lastName.value,  Address: formState.inputs.address.value,  Email: formState.inputs.email.value,  Password: formState.inputs.password.value,  }),  { "Content-Type": "application/json" }  );  auth.login(responseData.user as User);  } catch (error) {}  }  }; |

### UI Code Snippet for getting products from EC2 Instance

|  |
| --- |
| useEffect(() => {  const fetchProducts = async () => {  try {  const responseData = await sendRequest(  "http://ec2-54-80-72-110.compute-1.amazonaws.com:5000/api/products"  );  setLoadingProducts(responseData.products);  } catch (e) {}  };  fetchProducts();  }, []); |

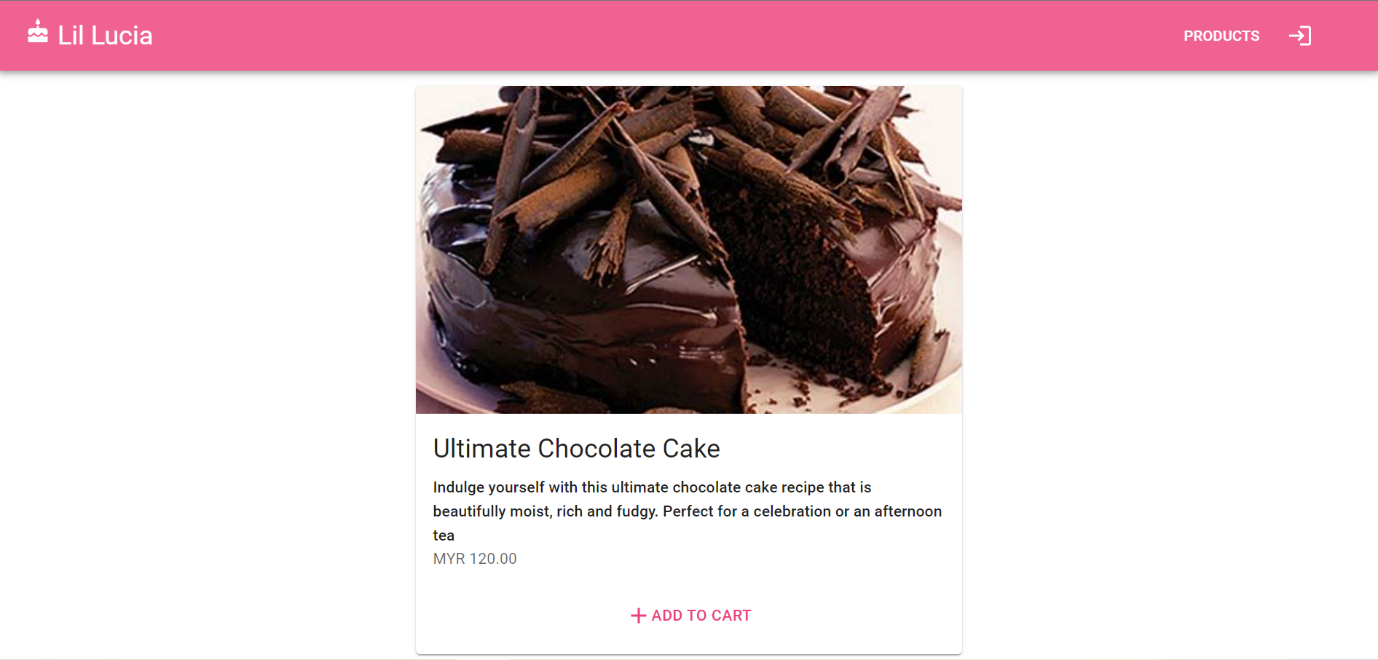
### UI Code Snippet for fetching user orders from EC2:

|  |
| --- |
| useEffect(() => {  const fetchOrders = async () => {  try {  const responseData = await sendRequest(  `http://ec2-54-80-72-110.compute-1.amazonaws.com:5000/api/orders/user/${auth.user.UserID}`  );  setLoadedOrders(responseData.orders);  } catch (e) {}  };  fetchOrders();  }, []); |

## User Manual

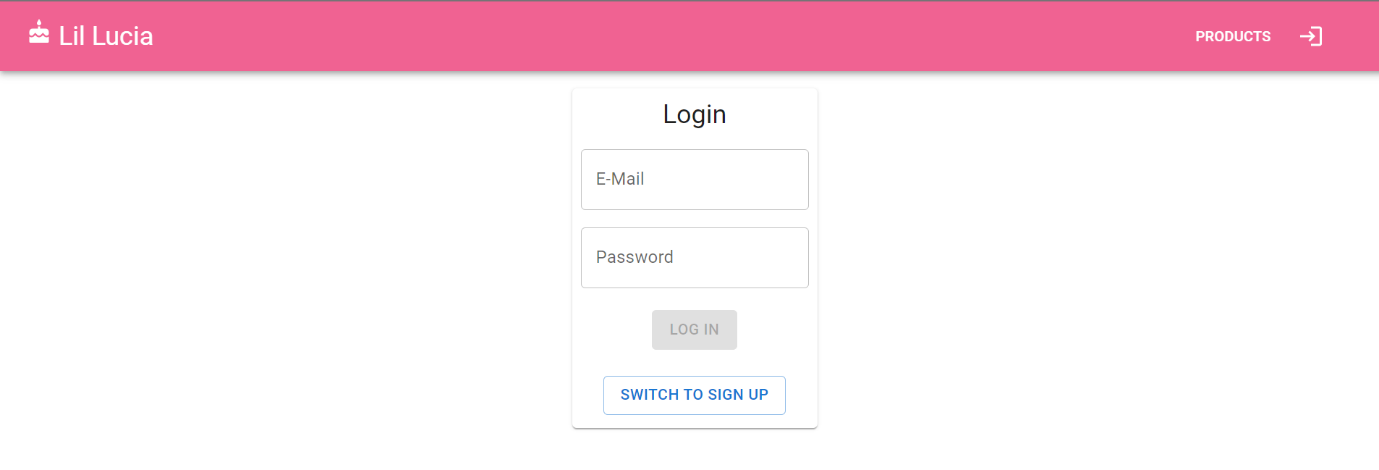
### Landing Page

For the Landing page of the application, it directly enters into the viewing product where users could then view all possible products for them to purchase. Though for them to continue is to log into the application. Attached is the screenshot of the landing page:



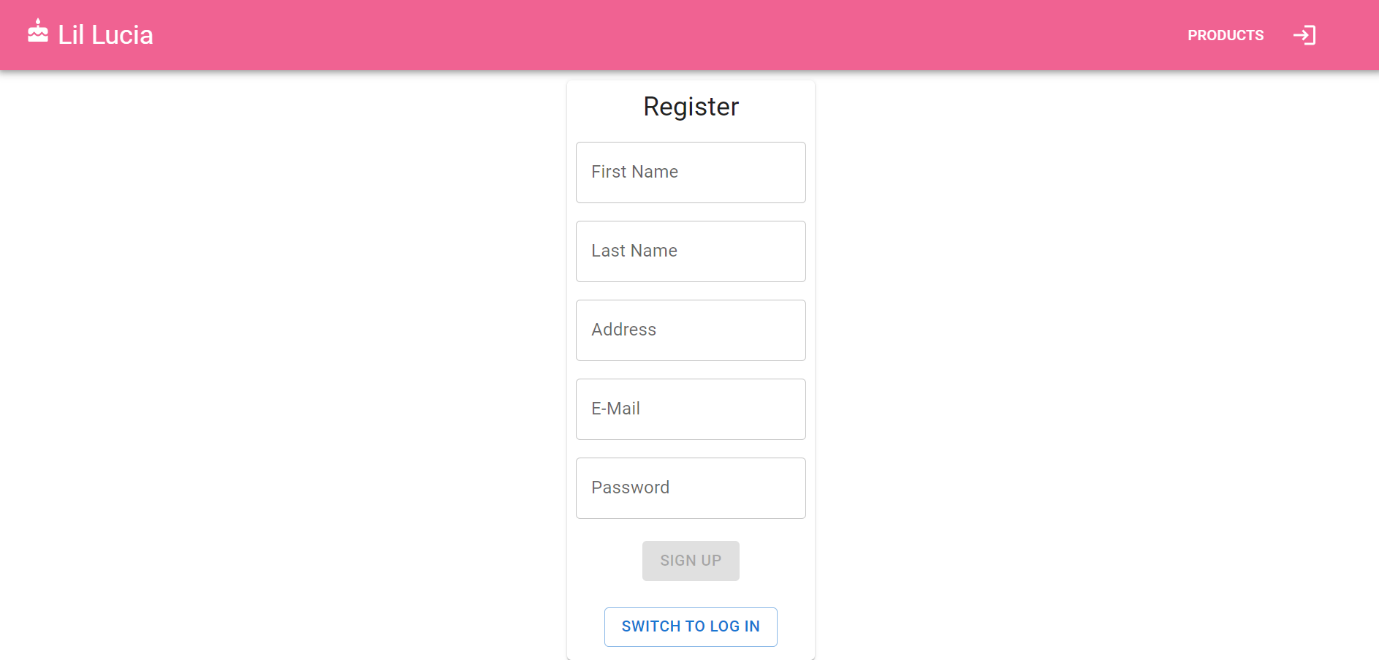
### Log In Page

Upon trying to further access the page you will then be brought into the log in portion of the page. If you have not registered with the application before you can sign up but the following is the screenshot for the log in of the application:



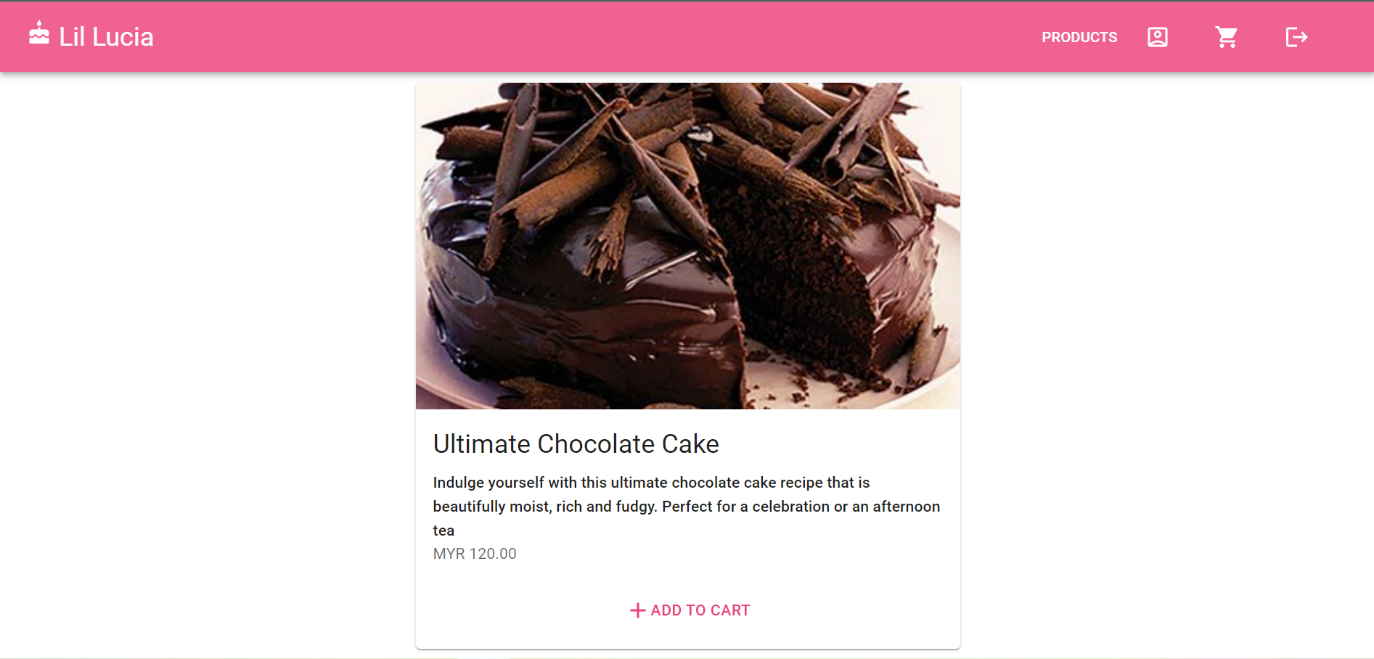
### Sign In Page

If you switch to the sign in portion of the page you can then register with the website and this will then keep the information stored securely on the AWS RDS instance that Lil-Lucia is deployed on. The following is the screenshot for the sign in portion of the page:

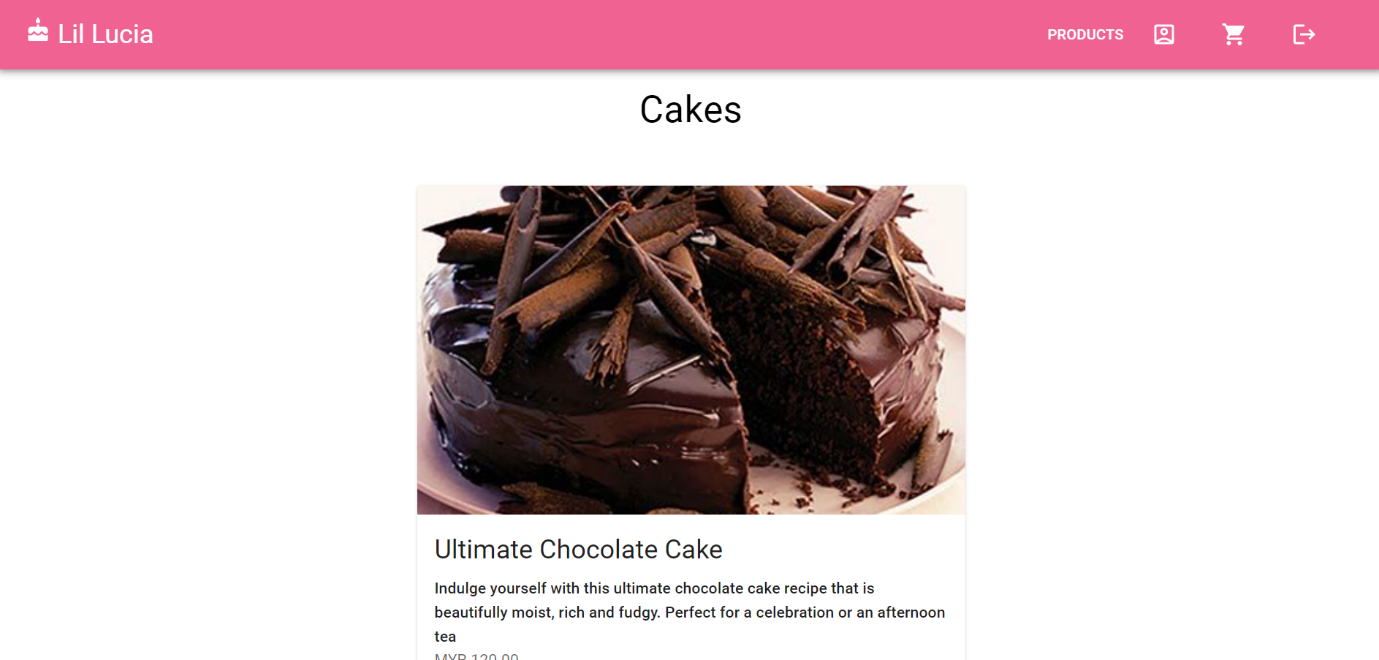


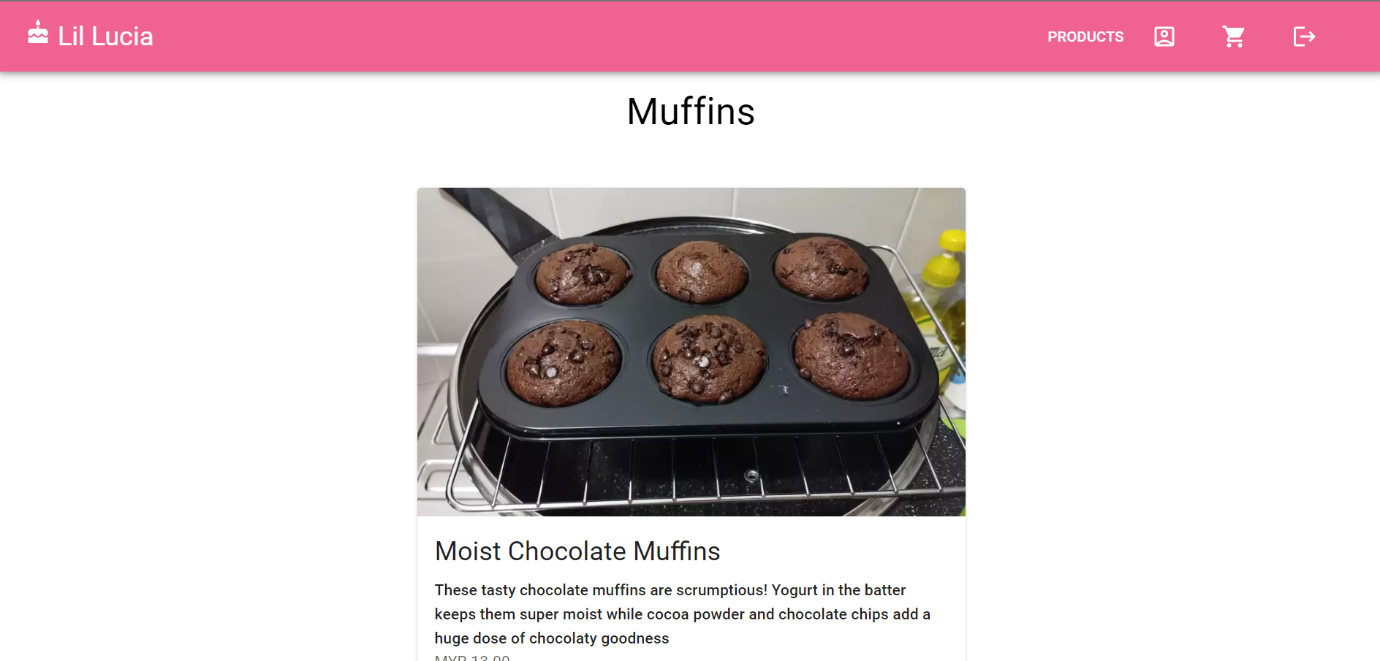
### Products Page

This is the products portion where you could then have it added to the cart for you to proceed to the checkout page and have your orders being listed as well. The following screenshots are related to the products page:

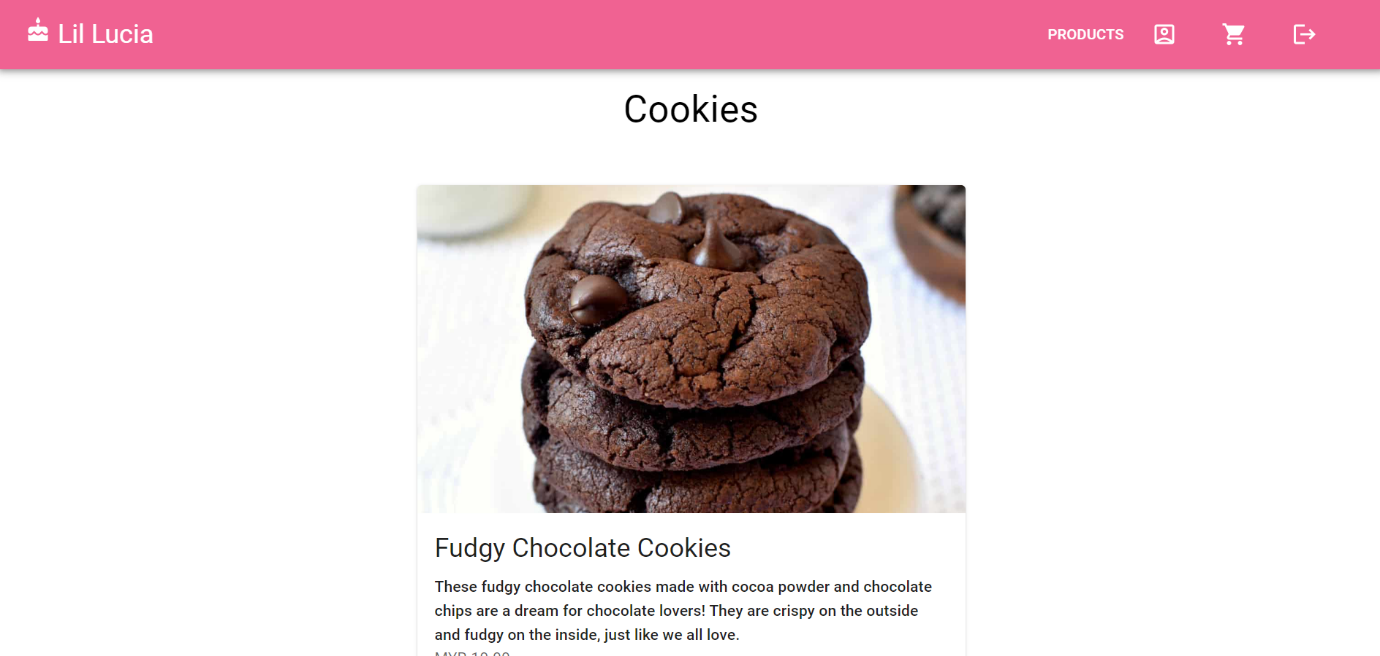


### Products Page (Cont.)



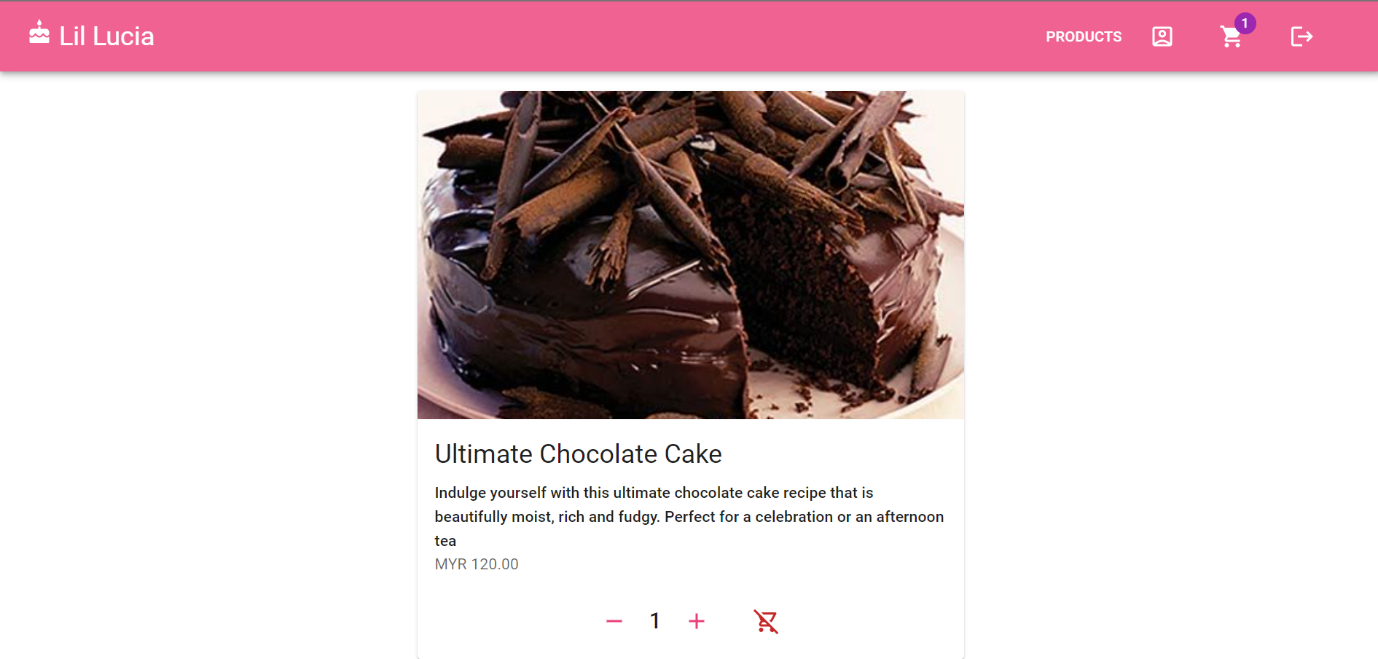


### Products Page (Cont.)

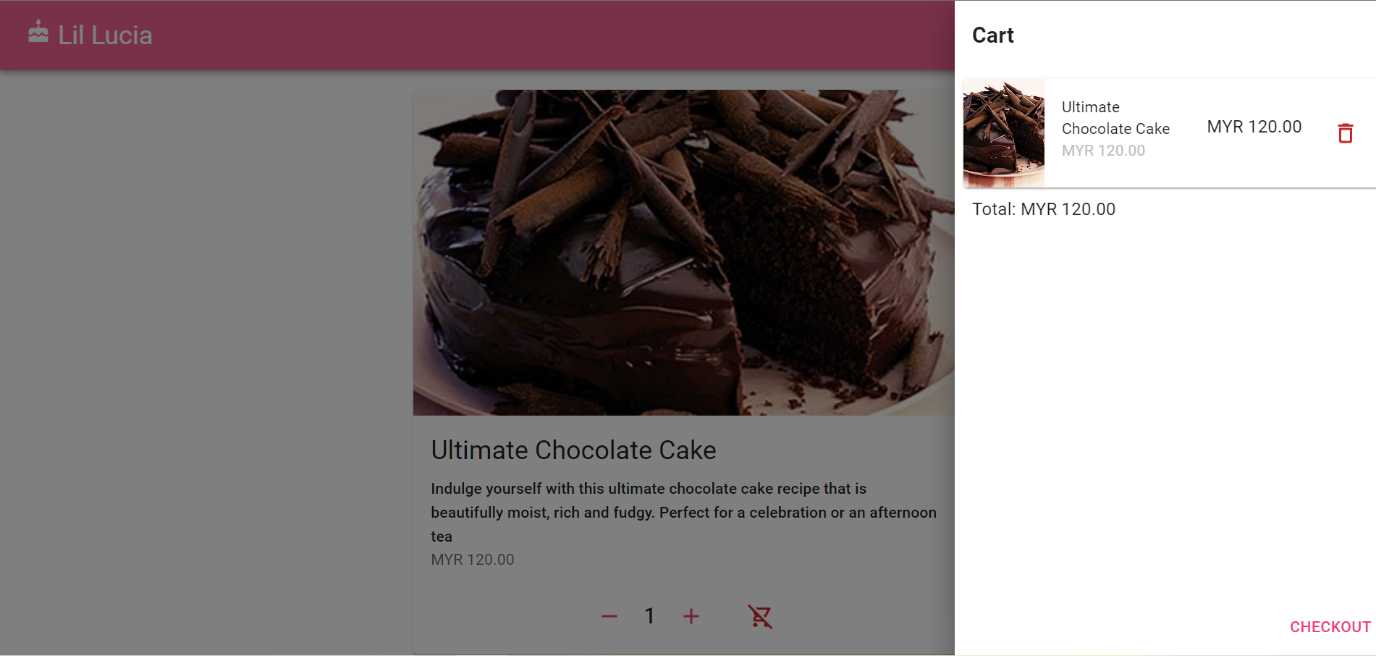


### Cart

Once you have decided on which of the items you would like to order you can add the items to cart and they will then show up in the shopping cart icon after. The screenshots of the adding and checking of the cart is attached as the following:

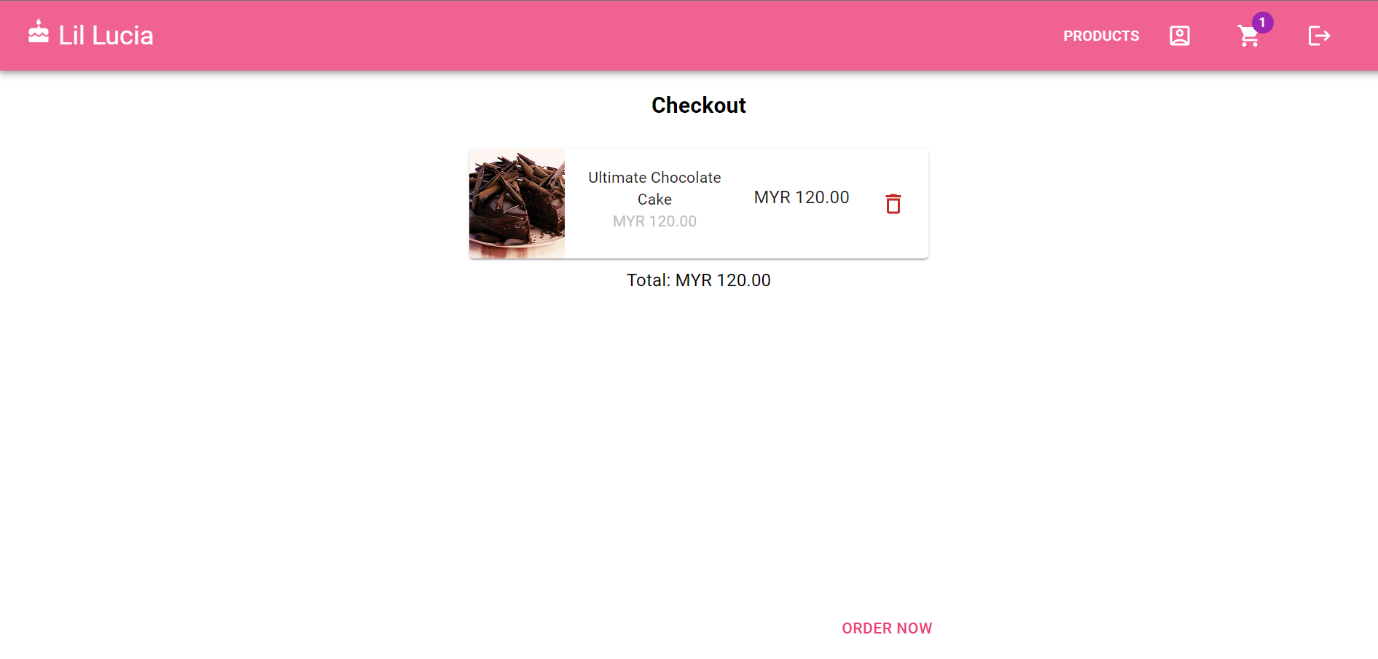


### Cart (Cont.)



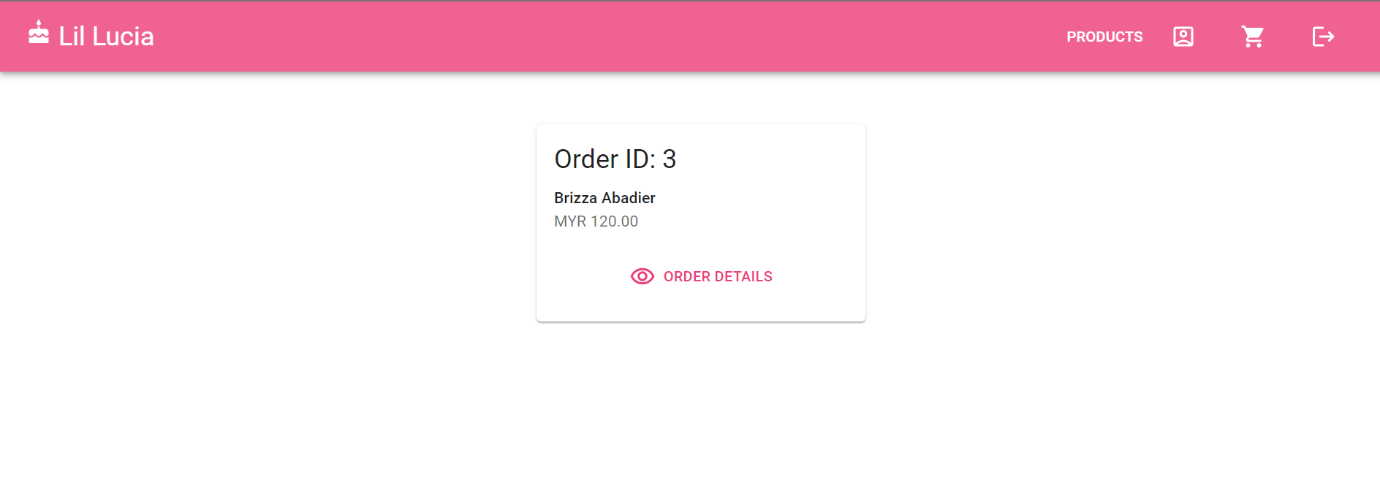
### Checkout Page

When the customer would like to checkout the items they will be redirected to the following page. In this instance the payment gateway is not implemented as this is just an initial release of the website as well. The checkout page is as follows:



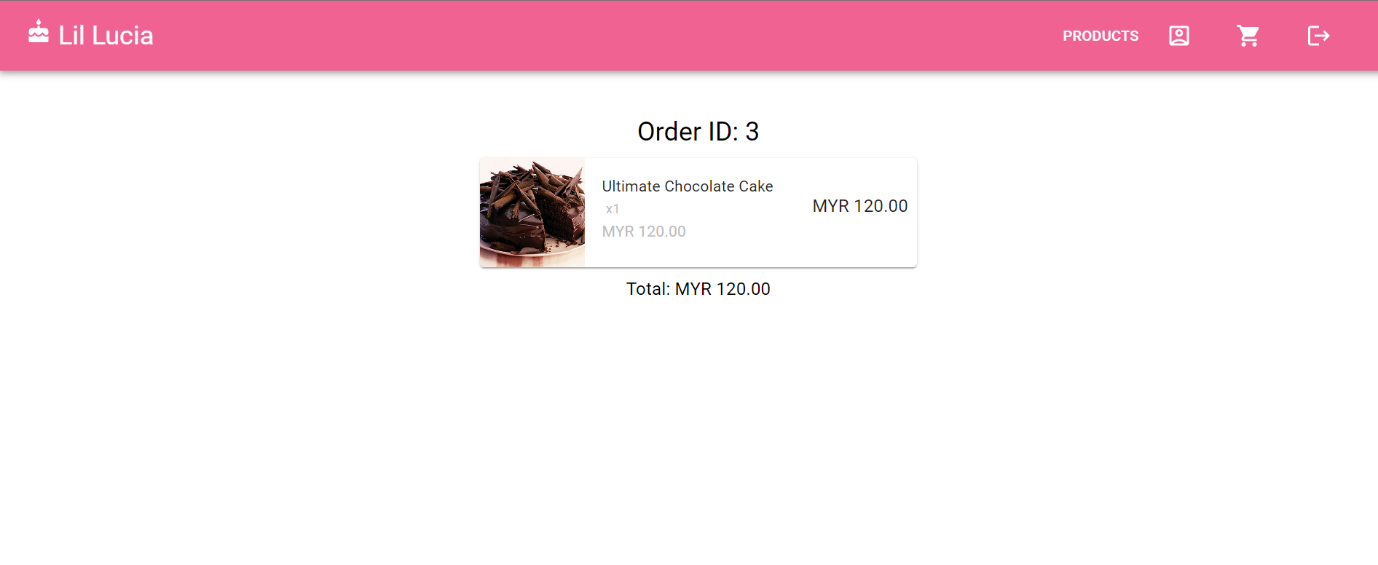
### Orders Page

After the order is made, this orders page acts as the history of the orders done by you as you can keep track of what you previously ordered as well. You could also dive deeper into what you ordered and this would bring you to the order detail page. The Orders page is as follows:



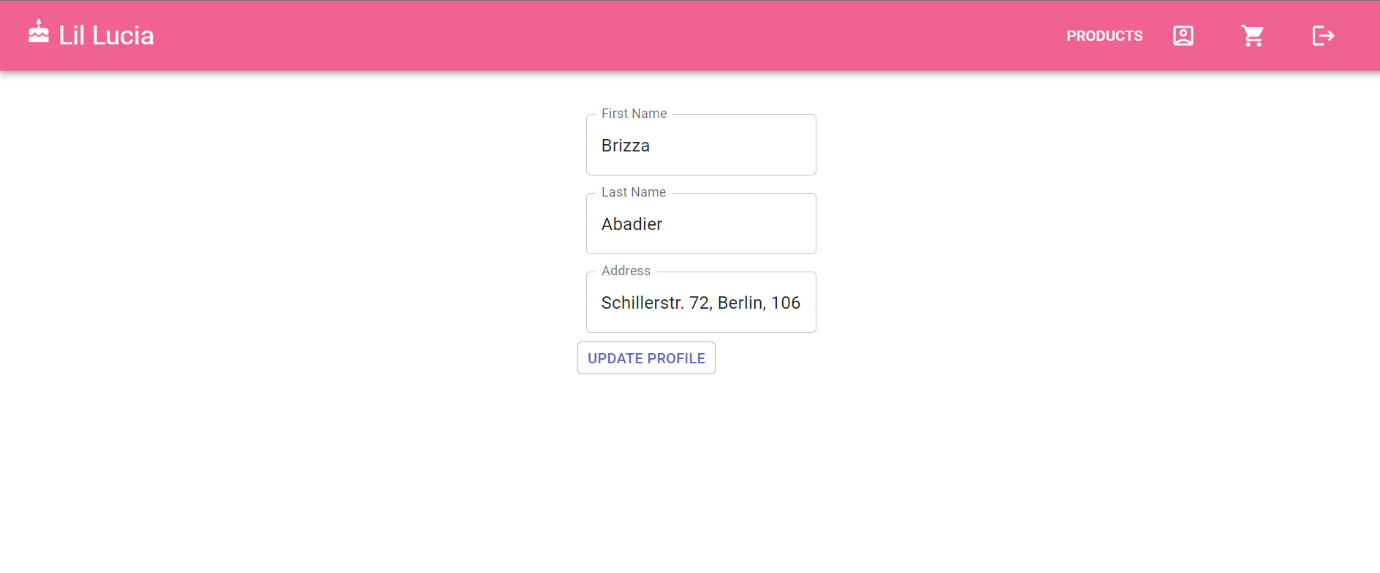
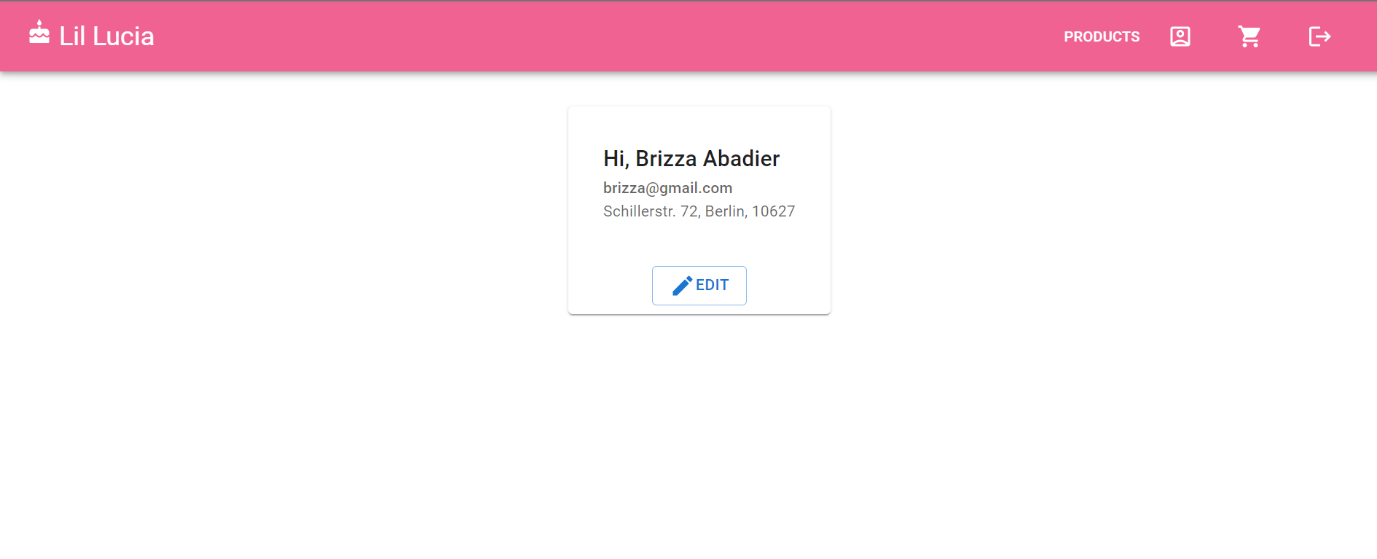
### Orders Detail Page

Continuing from the statement above, this page acts as more of a details page to what you have ordered in that single order sheet from your history. It is shown as the following:



### Profile Page

This on the other hand is the page where you could see all your details in which you have input before from the signup and you could also update the information. The following screenshots are from the profile page:



# Test Plan

## Unit & Integration Tests

### Objective:

The purpose of this test plan is to outline the approach for conducting unit and integration testing for the Products, Orders, and Users modules in the Lil-Lucia Cake Shop website development. The testing aims to ensure the reliability, functionality, and compatibility of these modules within the Node.js, React, sequelize, and MySQL-based application.

### Scope:

This test plan covers the following modules:

* Products
* Orders
* Users

### Testing Approach:

**Unit Testing:**

* Unit testing will focus on testing individual components and functions in isolation.
* Mocks and stubs will be used to isolate dependencies.
* Test cases will cover edge cases, and common scenarios.

**Integration Testing:**

* Integration testing will focus on testing interactions between different modules and components.
* Real databases and services will be used to simulate real-world scenarios.
* Test cases will validate data flow, API interactions, and module integration.

### Test Environment:

* Node.js environment for backend testing.
* React testing libraries for frontend testing.
* Sequelize ORM for database interactions.
* MySQL database (test instance) for database testing.

### Test Scenarios:

1. **Products Module:**

* Verify the creation of new products.
* Test product retrieval by ID and name.
* Validate updating product details.
* Test deletion of products.
* Check handling of invalid inputs and edge cases.

2. **Orders Module:**

* Verify the creation of new orders.
* Test retrieval of orders by user ID and order status.
* Validate updating order status.
* Test calculation of order totals and tax.
* Check handling of order cancellation and invalid inputs.

3. **Users Module:**

* Verify user registration and creation.
* Test user login and authentication.
* Validate user profile retrieval and updates.
* Check handling of invalid credentials and edge cases.

### Test Execution:

* Develop unit test cases for individual components and functions.
* Implement integration test cases to validate module interactions.
* Execute unit tests using testing frameworks.
* Execute integration tests with real databases and services.
* Monitor test results and identify failures.
* Debug and fix issues based on failed test cases.
* Retest fixed issues to ensure successful resolution.
* Maintain a record of test results, including passed and failed cases.

### Test Reporting:

* Maintain a detailed test log documenting each test case, its inputs, expected outcomes, and actual outcomes.
* Clearly categorize and report passed and failed test cases.
* Provide additional notes for failed test cases, explaining the cause of failure.

### Conclusion:

By meticulously planning and executing unit and integration tests for the Products, Orders, and Users modules, we aim to ensure that the Lil-Lucia Cake Shop website functions reliably and consistently. This testing approach will help identify and rectify issues early in the development process, contributing to the delivery of a high-quality and robust web application.

## Performance Testing

### Objective:

The purpose of this performance testing plan is to outline the strategy for evaluating the performance, scalability, and responsiveness of the Lil-Lucia Cake Shop website developed using Node.js, React, sequelize, and MySQL. Performance testing will identify potential bottlenecks, optimize resource utilization, and ensure the application's ability to handle varying user loads.

### Scope:

Performance testing will cover the following aspects:

* Response time: Measuring the time taken to respond to user actions.
* Scalability: Evaluating the application's ability to handle increased user loads.
* Resource Utilization: Monitoring CPU, memory, and database usage.

### Performance Testing Types:

**Load Testing:**

* Assess application performance under expected and peak load conditions.
* Identify performance degradation, bottlenecks, and breaking points.
* Determine if the application meets response time expectations.

**Stress Testing:**

* Evaluate the application's behaviour and stability under extreme conditions.
* Determine the maximum capacity the application can handle before crashing.
* Identify weak points and the application's ability to recover gracefully.

**Scalability Testing:**

* Assess how well the application scales as user load increases.
* Measure response times and resource utilization at various user load levels.

### Test Environment:

* Virtualized environment simulating real-world conditions.
* Monitoring tools to capture resource utilization and response times.
* Performance testing will be conducted on servers.

### Performance Test Scenarios:

1. **Load Testing Scenarios:**

* Simulate gradual user load increase to identify performance thresholds.
* Evaluate application responsiveness as concurrent users increase.
* Test common user actions: browsing products, adding to cart, placing orders.

2. **Stress Testing Scenarios:**

* Subject the application to user loads beyond normal capacity.
* Evaluate how the system handles unexpected spikes in traffic.
* Measure response times and system behaviour under stress.

3. **Scalability Testing Scenarios:**

* Gradually increase the user load to assess scalability.
* Evaluate how additional resources affect system performance.
* Measure response times, CPU utilization, memory usage, and database performance.

### Performance Metrics:

* Response Time: Measure the time taken for key user actions to complete.
* Latency: Measure the delay between sending a request and receiving a response.
* Error Rate: Track the percentage of failed requests or errors encountered.

### Test Execution:

* Prepare the environment and setup monitoring tools.
* Gradually ramp up user load and monitor system behaviour.
* Analyse performance metrics and identify bottlenecks or issues.
* Make necessary optimizations and adjustments based on test results.
* Re-test after adjustments to ensure improvements.

### Test Reporting:

* Maintain a detailed performance test report outlining the test scenarios, executed tests, results, and observations.
* Highlight any issues encountered, their impact, and suggested resolutions.

### Conclusion:

Performance testing is crucial to ensure that the Lil-Lucia Cake Shop website performs optimally under various conditions. By executing thorough load, stress, and scalability tests, we can pinpoint potential weaknesses and enhance the application's overall performance, ensuring a seamless and satisfying user experience, even during peak usage.

# System Limitation

While AWS EC2 (Elastic Compute Cloud) and AWS RDS (Relational Database Service) are powerful resources that offer hosting as well as managing applications like Lil-Lucia Cake Shop, there are certain limitations to be aware of. These limitations may impact the development, deployment, and performance of the application. Understanding these limitations is essential for effectively planning and mitigating potential challenges:

1. **Instance Type Limitations:**

* **EC2**: The choice of EC2 instance types can impact CPU, memory, and network performance. Selecting an inappropriate instance type might lead to resource bottlenecks or underutilization.
* **RDS**: While RDS provides various instance types optimized for different workloads, the choice might affect database performance and scalability.

2. **Resource Constraints:**

* **EC2**: Each EC2 instance type has specific limitations on CPU, memory, and storage capacity. Running resource-intensive applications may require larger instance types.
* **RDS**: Database size and performance depend on the allocated resources, and limitations can arise if the instance type or storage capacity is insufficient for the application's needs.

3. **Scalability Limits:**

* **EC2**: Scaling EC2 instances manually or automatically might be constrained by the number of available instance types, impacting the application's ability to handle varying traffic loads. This could also inflate the costs and this is an unwanted situation as well.
* **RDS**: While RDS supports read replicas and multi-AZ deployments for enhanced availability, scalability might be limited based on the chosen instance types and configurations.

4. **Security Considerations:**

* **EC2**: Security group configurations might inadvertently expose services or components, leading to potential vulnerabilities.
* **RDS**: Careful configuration of security groups and database access controls is crucial to prevent unauthorized access.

*Mitigation Strategies:*

* Choose appropriate instance types considering CPU, memory, and network requirements.
* Monitor resource utilization regularly and scale instances as needed.
* Implement a well-architected network topology to minimize latency.
* Implement auto-scaling mechanisms and use RDS read replicas for improved scalability.
* Back up application data using reliable storage solutions or services like Amazon S3.

In summary, understanding the limitations of AWS EC2 and RDS is essential for designing a resilient, scalable, and high-performance application like Lil-Lucia Cake Shop. By carefully considering these limitations and implementing appropriate mitigation strategies, you can ensure the successful development, deployment, and operation of your application on the AWS cloud.

# Conclusions & Reflections

In the journey to enhance the Lil-Lucia Cake Shop's growth and resilience, the project has not only addressed the challenges faced by a somewhat traditional bakery but has also showcased the power of how modern technology could drive positive change to the current market of things. Through the integration of Node.js, React, Sequelize, MySQL, and AWS services, we've successfully crafted quite a comprehensive solution in which that with Lil-Lucia's objectives and paves the way for an exciting future.

The development of a cloud-based web application has redefined Lil-Lucia's online presence, offering customers an intuitive and engaging platform to explore, discover, and celebrate their love for delectable pastries and cakes. The carefully designed frontend, coupled with an efficient backend, empowers users to navigate through the extensive array of products, place orders, and foster an active baking expertise that is found with Lil-Lucia in which this also paves the way forward for any collaboration with other shops products as well.

The adoption of cloud architectural styles, the streamlined two-tier architecture for simplicity, demonstrates the versatility of technology in addressing diverse business needs. By integrating AWS EC2, and RDS, we've ensured the application's stability, scalability, and data security, all essential components for a successful online venture.

Throughout the development process, testing methodologies, to which include unit, integration, as well as performance testing, have been employed to validate the application's functionality, reliability, and responsiveness. These efforts not only guarantee the delivery of a great product but also underscore our commitment to delivering a seamless user experience, even during peak usage scenarios.

In conclusion, the Lil-Lucia Cake Shop project has moved beyond the traditional bakery model by embracing innovation and technology. The journey undertaken here demonstrates how the fusion of pastries and cutting-edge technology can yield exceptional results. As Lil-Lucia Cake Shop brings forward change in this new era, I am confident that the solutions put forth in this project will serve as a foundation for sustainable growth, and continued excellence in the world of sweet indulgence.

# Bibliography

*Amazon AWS EC2* (1987) *Amazon*. Available at: https://aws.amazon.com/ec2/ec2-get-started/ (Accessed: 24 August 2023).

*Amazon RDS* (2007) *Amazon*. Available at: https://aws.amazon.com/rds/mysql/ (Accessed: 24 August 2023).

Ku, W. (no date) *Using mysql workbench to connect to your mysql server on AWS EC2 (ubuntu) instance*, *LinkedIn*. Available at: https://www.linkedin.com/pulse/using-mysql-workbench-connect-your-server-aws-ec2-ubuntu-william-ku (Accessed: 24 August 2023).

Mathew, N. (2021) *Connect MySQL Workbench to EC2 instance*, *Bobcares*. Available at: https://bobcares.com/blog/connect-mysql-workbench-to-ec2-instance/ (Accessed: 24 August 2023).

Mishra, S. (2020) *How to deploy node.js app on AWS with github*, *Medium*. Available at: https://sumantmishra.medium.com/how-to-deploy-node-js-app-on-aws-with-github-db99758294f1 (Accessed: 24 August 2023).

Shivalkar, R. (2023) *Deploy nodejs app to AWS in EC2 server*, *ClickIT*. Available at: https://www.clickittech.com/devops/deploy-nodejs-app-to-aws/ (Accessed: 24 August 2023).