

KARABUK UNIVERSITY

FACULTY OF ENGINEERING

**Vocational Training in Business**

**Practice Report**

|  |  |
| --- | --- |
| Student Number: | 2010205590 |
| Name Surname: | MOHAMAD ZUBI |
| Department: | Computer Engineering %30 I. |
| Year: | 2024 |

**KARABUK UNIVERSITY  
FACULTY OF ENGINEERING**

**Vocational Training in Business  
General Information**

**The Student Information**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Student’s Name Surname: | MOHAMAD ZUBI | | | T.R. Identity Number: | 99672148028 | | | Student Number: | 2010205590 | | | Department: | Computer Engineering %30 I. | | | Start Date: | 12 / 02 / 2024 | | | End Date: | 31 / 05 / 2024 | | |  | Date: | 31 / 05 / 2024 | |  | Student’s Signature: |  | |  |

**The Workplace Information**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Contact Info | |  |  | | --- | --- | | Workplace Name: | Oxygen Technology | | Address: | An Nakheel District, Riyadh, Saudi Arabia. 12282. | | Phone and Fax: | +966 55 888 8478 | | E-mail: | info@o2.com.sa | |
| Workplace Properties | |  |  | | --- | --- | | Number of employees: | 9 | | Number of graduate employees: | 7 | | Service, meal, etc. social services: | None | | Production type and capacity: | Technology Services | | Equipment pool: | None | | Other information: |  | |

**The Workplace Officer**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Authorized Signature | |  |  | | --- | --- | | Workplace Officer’s Name Surname: |  | | Position: |  | | Date: |  | | Signature, Cachet: |  | |

**Vocational Training in Business  
Weekly Work Plan**

|  |  |
| --- | --- |
| Weeks | Work Plan |
| 1st Week 12 / 02 / 2024 - 17 / 02 / 2024 | Getting to Know the Workplace and Colleagues, and Gemini API Integration |
| 2nd Week 19 / 02 / 2024 - 24 / 02 / 2024 | Creating an API Endpoint with Node.js and Express.js |
| 3rd Week 26 / 02 / 2024 - 02 / 03 / 2024 | Introduction to Project and mongodb Task |
| 4th Week 04 / 03 / 2024 - 09 / 03 / 2024 | Authentication APIs Development |
| 5th Week 11 / 03 / 2024 - 16 / 03 / 2024 | User Management Functions |
| 6th Week 18 / 03 / 2024 - 23 / 03 / 2024 | Plan Schema and Functionality |
| 7th Week 25 / 03 / 2024 - 30 / 03 / 2024 | Plan Schema - Subscription and Functionality |
| 8th Week 01 / 04 / 2024 - 06 / 04 / 2024 | Main components of Landing page |
| 9th Week 08 / 04 / 2024 - 13 / 04 / 2024 | Main components of Landing page |
| 10th Week 22 / 04 / 2024 - 27 / 04 / 2024 | oAuth google - github |
| 11th Week 29 / 04 / 2024 - 04 / 05 / 2024 | Front end Authentiaction integration |
| 12th Week 06 / 05 / 2024 - 11 / 05 / 2024 | Pdf Model & Controller |
| 13th Week 13 / 05 / 2024 - 18 / 05 / 2024 | Resume Generation Controller |
| 14th Week 20 / 05 / 2024 - 25 / 05 / 2024 | Payment Integration |
| 15th Week 27 / 05 / 2024 - 01 / 06 / 2024 | Payment Integration |
| Training Staff(Responsible Engineer)Title, Name, Signature, Cachet | |

O2 Technology, established in 2007, is a Saudi-based company that unites elite professionals across various tech fields. Their core services encompass the entire digital spectrum, from crafting user-friendly websites and e-commerce stores to mobile application development and trademark industry services.

O2 Technology isn't just about ticking boxes. Their approach is driven by a core set of values. They prioritize adaptability, tailoring strategies to offer each client the perfect technological fit. Innovation is a constant pursuit, with the team actively seeking cutting-edge technologies to integrate and propel their clients forward. Delivering exceptional work is paramount. They strive to exceed client expectations with every project, fostering a collaborative and flexible environment. Diversity is another core value, with O2 Technology embracing a team environment that celebrates a variety of skills and perspectives.

O2 Technology's journey began in 2007, fueled by a shared passion for technology and a desire to push its boundaries. Their official entry into the technology space came in 2008 with the opening of their first computer retail branch. Recognizing the growing demand for online solutions, O2 Technology expanded its expertise in 2009 to include programming and web development services. A year later, they launched O2 Cloud Web Hosting, offering clients a reliable and user-friendly platform. By 2016, the company had grown significantly, reaching nearly 40 skilled professionals. This expansion allowed them to venture into mobile application development, catering to the ever-increasing mobile landscape

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 1 | Page Number: | 1 |
| Getting to Know the Workplace and Colleagues, and Gemini API Integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 12 / 02 / 2024 |
| End Date |
| 17 / 02 / 2024 |

During my first week at Oxygen Technology, I had the opportunity to familiarize myself with the workplace environment and connect with my colleagues and fellow developers. It was an exciting start as I delved into the company culture and got to know the talented individuals I'll be collaborating with.

One of the key highlights of my week was being entrusted with a task that involved integrating the Gemini API into a Node.js environment. This task aimed to enhance my understanding of API integration processes while providing hands-on experience with Node.js development.

To kickstart the integration process, I began by researching the Gemini API documentation, understanding its functionalities, and exploring potential use cases within our projects. After gaining insights into the API's endpoints and authentication mechanisms, I proceeded to implement a basic integration framework in Node.js.

**Configuration Explanation:**

The configuration section sets up parameters for content generation, influencing the behavior and quality of the generated output. These parameters include temperature, which controls the diversity of generated text by adjusting randomness; topP and topK, which determine the probability cutoff and the number of tokens considered during generation; and maxOutputTokens, defining the maximum length of the generated content. By configuring these settings, we can fine-tune the generative process to suit specific requirements and ensure the desired quality and relevance of the generated content.

const { GoogleGenerativeAI } = require("@google/generative-ai");

// Configuration

const api = process.env.apiKey;

const genAI = new GoogleGenerativeAI(api);

const generationConfig = {

  temperature: 0.9,

  topP: 1,

  topK: 1,

  maxOutputTokens: 4096,

};

const model = genAI.getGenerativeModel({

  model: "gemini-pro",

  generationConfig,

});

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 1 | Page Number: | 2 |
| Getting to Know the Workplace and Colleagues, and Gemini API Integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 12 / 02 / 2024 |
| End Date |
| 17 / 02 / 2024 |

**Function Explanation:**

The generateContent function is responsible for asynchronously generating content based on a provided prompt using the configured generative model. Upon invocation with a prompt as input, the function interacts with the generative model to produce text that continues or responds to the given prompt. It handles the asynchronous nature of the content generation process using async/await, ensuring smooth execution and error handling. Once the generation process is complete, the function returns the generated text for further processing or display. This function encapsulates the core functionality of interacting with the generative model and abstracts away the complexities of the underlying implementation, providing a straightforward interface for content generation tasks.

async function generateContent(prompt) {

    try {

      const result = await model.generateContent(prompt);

      const response = await result.response;

      return response.text();

    } catch (error) {

      console.error("Error generating content:", error);

      throw error;

    }

  }

console.log(generateContent("what does hello world mean?"));

Throughout this process, I encountered challenges and learning opportunities, such as handling authentication tokens securely and optimizing API requests for efficiency. By leveraging the support of my colleagues and engaging in collaborative problem-solving, I was able to overcome these hurdles and make significant progress in the integration task.

Overall, my first week at Oxygen Technology was enriching and rewarding. I look forward to building upon this foundation in the coming weeks, contributing meaningfully to the team's projects, and continuing my growth as a Full-stack Developer intern.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 2 | Page Number: | 3 |
| Creating an API Endpoint with Node.js and Express.js | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 19 / 02 / 2024 |
| End Date |
| 24 / 02 / 2024 |

During my second week at Oxygen Technology, I was assigned an engaging task that involved building an API endpoint using Node.js and Express.js. The objective was to develop an endpoint where a prompt could be provided, and in return, the endpoint would generate content using the Gemini API.

To commence this task, I embarked on a journey of understanding Express.js and its functionalities. I delved into the documentation, grasping concepts such as routing, middleware, and request handling. This foundational knowledge laid the groundwork for implementing the API endpoint effectively.

Using the acquired understanding, I proceeded to craft the endpoint in Node.js, leveraging the robust capabilities of Express.js to streamline the development process. I structured the endpoint to accept incoming prompts, process them, and interact with the Gemini API to generate relevant content dynamically.

Throughout the development phase, I continuously referenced the Gemini API documentation to ensure seamless integration and adherence to best practices. Additionally, I utilized Postman, an invaluable tool, to test the functionality of the API endpoint rigorously. Postman facilitated the sending of various prompts and verifying the corresponding generated content, thereby ensuring the reliability and efficiency of the endpoint.

Here's a simplified example illustrating the implementation of the API endpoint using Express.js:

const { GoogleGenerativeAI } = require("@google/generative-ai");

const express = require("express");

const app = express();

const port = 3000;

app.get("/generate", async (req, res) => {

  const prompt = req.query.prompt;

  if (!prompt) {

    return res.status(400).send("Prompt is required");

  }

  try {

    const response = await generateContent(prompt);

    res.send(response);

  } catch (error) {

    res.status(500).send("Error generating content");

  }

});

app.listen(port, () => {

  console.log(`Server is running on port ${port}`);

});

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 2 | Page Number: | 4 |
| Creating an API Endpoint with Node.js and Express.js | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 19 / 02 / 2024 |
| End Date |
| 24 / 02 / 2024 |

This endpoint allows clients to send prompts via a POST request to /generate-content, and in response, it provides the generated content retrieved from the Gemini API.

A screenshot of a computer

Description automatically generated

In conclusion, the second week of my internship was filled with valuable learning experiences, from learning Express.js to developing a functional API endpoint. Moving forward, I am excited to continue exploring new technologies and contributing meaningfully to the projects at Oxygen Technology.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 3 | Page Number: | 5 |
| Introduction to Project and mongodb Task | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 26 / 02 / 2024 |
| End Date |
| 02 / 03 / 2024 |

This week marked my introduction to the project at the software company where I am interning. The project focuses on developing a resume generator/enhancer utilizing AI, specifically the Gemini API. My primary task for the week was to set up MongoDB and create a basic schema for the project.

The overarching goal of the project is to provide users with a platform where they can generate or enhance resumes efficiently. Additionally, the project includes a payment integration feature.

Users will have the option to sign up using either their GitHub or Google accounts. Upon registration, users will be automatically enrolled in a free plan, termed as the "Starter Plan." This plan grants users 5 free credits, which they can utilize for various actions such as generating a resume, enhancing an existing resume, or creating one from scratch.

One of the key functionalities of the project is the "Generate Resume" feature. This feature acts as a converter, enabling users to transform resumes of any style or format into an ATS-friendly Harvard resume template.

Throughout the week, I focused on setting up MongoDB to facilitate the storage and retrieval of user data efficiently. Additionally, I worked on designing a basic schema to organize the data effectively within the database.

const mongoose = require("mongoose");

In this line, I downloaded the mongoose package which is a mongodb related package to handle schema creation and model functions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 3 | Page Number: | 6 |
| Introduction to Project and mongodb Task | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 26 / 02 / 2024 |
| End Date |
| 02 / 03 / 2024 |

mongoose

  .connect(

    "mongodb+srv://mozubi128:mWKVbkOQTRvVW6WT@crave.qytt8kn.mongodb.net/ResumeEnhancer"

  )

  .then(() => {

    app.listen(port, () => {

      console.log("server is on " + port);

    });

  })

  .catch((err) => {

    console.log(err.message);

  });

The provided code utilizes Mongoose, a MongoDB object modeling tool. It establishes a connection to the MongoDB database by invoking the mongoose.connect() method with a MongoDB connection string as an argument. This string includes the necessary credentials and specifies the MongoDB Atlas URI along with the database name.

const usersSchema = new Schema(

  {

    userName: {

      type: String,

      required: true,

    },

    password: {

      type: String,

      required: false,

      min: 6,

    },

});

The code snippet defines a Mongoose schema for users, specifying two fields: userName and password. The userName field is required and expects a string value, while the password field is optional, requiring a string with a minimum length of six characters. This schema provides a structured blueprint for storing user data in the MongoDB database, ensuring data consistency and validation.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 4 | Page Number: | 7 |
| Authentication APIs Development | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 04 / 03 / 2024 |
| End Date |
| 09 / 03 / 2024 |

During the fourth week of my internship, I was tasked with developing authentication APIs for user registration and login. Leveraging Express.js, MongoDB, and Node.js, I embarked on creating robust endpoints to facilitate user authentication. This involved implementing functionalities for user registration, which includes securely storing user credentials in the MongoDB database, and user login, allowing registered users to authenticate themselves securely. By utilizing these technologies, I aimed to ensure the seamless and secure management of user authentication within the project's framework.

const createToken = (\_id, email, roles) => {

  return jwt.sign({ \_id, email, roles }, process.env.JWT\_SECRET, {

    expiresIn: "30d",

  });

};

This function, createToken, utilizes JWT to generate tokens for users. It takes user-specific information such as \_id, email, and roles, and signs them with the JWT secret stored in the environment variable JWT\_SECRET. The resulting token expires after 30 days, enhancing security by limiting its validity period.

const register = async (req, res) => {

  const { email, password, userName, provider } = req.body;

  if (provider) {

  }

  console.log(email);

  try {

    const user = await User.signup(email, password, userName);

    const sub = await Subscription.findOne({ userId: user.\_id });

    if (!sub) await subscribeUserFree(user.\_id);

    const token = createToken(user.\_id, user.email, user.permissions);

    res.status(200).json({ email, token });

  } catch (error) {

    res.status(400).json({ Error: error.message });

  }

};

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 4 | Page Number: | 8 |
| Authentication APIs Development | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 04 / 03 / 2024 |
| End Date |
| 09 / 03 / 2024 |

The register function handles the registration process for users. It extracts relevant data such as email, password, userName, and provider from the request body. Within a try-catch block, it attempts to sign up the user using the User.signup method, which presumably handles user registration in the database. If successful, it checks if the user has an existing subscription; if not, it subscribes the user to the free plan using subscribeUserFree. Subsequently, it generates a JWT token using the createToken function, containing the user's \_id, email, and permissions. Finally, it responds with a JSON object containing the user's email and the generated token on success, or an error message on failure.

const login = async (req, res) => {

  const { email, password } = req.body;

  console.log(email, password);

  try {

    const user = await User.login(email, password);

    //create token

    const token = createToken(user.\_id, user.email, user.permissions);

    const sub = await Subscription.findOne({ userId: user.\_id });

    if (!sub) await subscribeUserFree(user.\_id);

    res.status(200).json({ email, token });

  } catch (error) {

    res.status(400).json({ msg: error.message });

  }

};

userRouter.post("/signIn", login);

userRouter.post("/signUp", register)

The login function facilitates user login by extracting the email and password from the request body. It then attempts to log in the user using the User.login method, which presumably verifies the user's credentials. Upon successful login, it generates a JWT token for the user using the createToken function, containing the user's \_id, email, and permissions. Additionally, it checks if the user has an existing subscription; if not, it subscribes the user to the free plan using subscribeUserFree. Finally, it responds with a JSON object containing the user's email and the generated token on success, or an error message on failure. The endpoints for both signing up (register) and signing in (login) handle the user authentication process within the application.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 5 | Page Number: | 9 |
| User Management Functions | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 11 / 03 / 2024 |
| End Date |
| 16 / 03 / 2024 |

During the fifth week of my internship, I was assigned the task of expanding the functionality related to user management within the application. This involved the creation of several new functions to perform various user-related operations. These functions include getUsers, which retrieves a list of users, deleteUser, which removes a user from the system, addRole, which assigns a new role to a user, and deleteRole, which removes a role from a user. By implementing these functions, the application's capability to manage users is enhanced, providing administrators with the necessary tools to effectively manage user accounts and permissions.

const getUsers = async (req, res) => {

  const users = await User.find({}).sort({ createdAt: -1 });

  res.status(200).json(users);

};

The provided code is the function for retrieving users from the database, using the await User.find({}) this part gets all the users from the User model, and then using the .sort() I sorted them in ascending order.

const deleteUser = async (req, res) => {

  const { id } = req.params;

  try {

    if (!mongoose.Types.ObjectId.isValid(id)) {

      return res.status(404).json({ error: " no such user " });

    } else {

      const user = await User.findOneAndDelete({ \_id: id });

      if (!user) {

        return res.status(400).json({ error: " no such user " });

      }

      res.status(200).json("success");

    }

  } catch (error) {

    res.status(400).json({ msg: error.message });

  }

};

The provided code is the function that handles the deletion of a user from the database, basically it takes the id of the user from the request params. And then validates the id to check if it’s a valid MongoDB id. After that using the User.findOneAndDelete() it permanently deletes the user from the database and return success as a response message.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 5 | Page Number: | 10 |
| User Management Functions | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 11 / 03 / 2024 |
| End Date |
| 16 / 03 / 2024 |

const addRole = async (req, res) => {

  const { id } = req.params;

  let data = "";

  try {

    const { permission } = req.body; // admin

    if (!permission) {

      throw new Error("permission field is required");

    }

    if (!(permission in Roles)) {

      throw new Error("Permission field is invalid");

    }

    const checkUser = await User.findById(id);

    if (checkUser.permissions.includes(permission)) {

      throw new Error("Permission already exists");

    }

    const user = await User.findOneAndUpdate(

      { \_id: mongoose.Types.ObjectId.isValid(id) ? id : null },

      { $push: { permissions: permission } },

      { new: true }

    );

    if (!user) {

      return res.status(404).json({ error: "No such user" });

    }

    return res.status(200).json({ Data: data, user });

  } catch (error) {

    return res.status(400).json({ msg: error.message });

  }

};

The addRole function assigns a new role to a user identified by their ID. It validates the provided role, checks if the user already has the role, and updates the user's permissions accordingly. If successful, it returns the updated user data; otherwise, it returns an error message.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 6 | Page Number: | 11 |
| Plan Schema and Functionality | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 18 / 03 / 2024 |
| End Date |
| 23 / 03 / 2024 |

In the sixth week of my internship, I was tasked with defining the structure of plans that will be available on the platform and implementing basic functionality to create these plans using Mongoose. I created a Mongoose schema to represent the attributes of each plan, such as name, description, price, and features. Additionally, I developed a createPlan function to facilitate the creation of new plans within the system. This function allows for the specification of plan details and inserts them into the MongoDB database, enabling the platform to offer a variety of subscription plans to users.

const planSchema = new Schema({

  name: {

    type: String,

    required: true,

  },

  description: {

    type: String,

    required: true,

  },

  credits: {

    type: Number,

    required: true,

    min: 0,

  },

  price: {

    type: Number,

    required: true,

    min: 0,

  },

  isActive: {

    type: Boolean,

    default: true,

  },

  features: {

    type: [String],

    required: true,

  },

});

The provided schema, planSchema, defines the structure of plans available on the platform. It includes fields such as name, description, credits, price, isActive, and features. Each plan requires a name and description, while credits and price specify the plan's monetary and credit-based attributes, respectively, with minimum values enforced. The isActive field indicates whether the plan is currently active, defaulting to true. Additionally, the features field is an array of strings representing the features included with the plan, ensuring flexibility in defining plan offerings.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 6 | Page Number: | 12 |
| Plan Schema and Functionality | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 18 / 03 / 2024 |
| End Date |
| 23 / 03 / 2024 |

In the newly created plansController.js file, I implemented a createPlan function to manage the creation of plans. This function accepts the necessary plan details such as name, description, credits, price, isActive, and features. Utilizing the planSchema, it constructs a new plan object with the provided details and saves it to the MongoDB database using Mongoose's save method. Upon successful creation, it returns the newly created plan data, enabling the platform to offer a diverse range of subscription plans to users.

const createPlan = async (req, res) => {

  try {

    const { name, description, credits, price, isActive } = req.body;

    const newPlan = await Plan.create({

      name,

      description,

      credits,

      price,

      isActive,

    });

    return res.status(200).json(newPlan);

  } catch (error) {

    return res.status(401).json({ error: error.message });

  }

};

Following the creation of the createPlan function in the plansController.js file, I established a new router in the plansRouter.js file to handle requests related to plan creation. Utilizing Express.js, I defined a POST endpoint /create within the plansRouter to handle requests for creating new plans. This endpoint is associated with the createPlan function, ensuring that incoming POST requests to /create are directed to the respective function in the plansController. This modular approach enables efficient routing and separation of concerns, facilitating the development and maintenance of the platform's plan management functionality.

plansRouter.post("/create", createPlan);

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 7 | Page Number: | 13 |
| Plan Schema - Subscription and Functionality | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 25 / 03 / 2024 |
| End Date |
| 30 / 03 / 2024 |

During the seventh week of my internship, I continued to enhance the plan management functionalities within the platform. I was tasked with implementing additional APIs and functions to retrieve plans and individual plan details. To achieve this, I created a getPlans API and function to retrieve a list of all available plans, as well as a getSingularPlan API and function to fetch details of a specific plan based on its ID. Additionally, I designed a subscription schema to represent the subscription plans available to users. This schema serves as the blueprint for organizing subscription-related data, enabling efficient management and integration of subscription functionalities within the platform. Through these developments, the platform gains enhanced capabilities for managing and offering subscription plans to users.

const getAllPlans = async (req, res) => {

  try {

    const plans = await Plan.find({});

    return res.status(200).json(plans);

  } catch (error) {

    return res.status(401).json({ error: error.message });

  }

};

const getSingularPlan = async (req, res) => {

  try {

    const { id } = req.params;

    const plan = await Plan.findById(id);

    return res.status(200).json(plan);

  } catch (error) {

    return res.status(401).json({ error: error.message });

  }

};

In the platform's development, I've added two essential functions: getAllPlans and getSingularPlan. These functions are crucial for retrieving plan data, either in bulk or individually.

The getAllPlans function retrieves all available plans from the database using Mongoose's find method without any specific filter criteria. Upon successful retrieval, it returns a JSON response with a status code of 200, containing the array of plans.

On the other hand, the getSingularPlan function retrieves a specific plan based on its unique ID. It extracts the plan ID from the request parameters and queries the database using Mongoose's findById method. If the plan is found, it returns the plan object in a JSON response with a status code of 200. Otherwise, it returns an error message with a status code of 401 if the plan is not found or if there's an error during the retrieval process.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 7 | Page Number: | 14 |
| Plan Schema - Subscription and Functionality | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 25 / 03 / 2024 |
| End Date |
| 30 / 03 / 2024 |

The userSubscriptionSchema is pivotal for structuring user subscription data within the platform's architecture. It comprises several key fields to comprehensively capture subscription details. Firstly, the userId field serves as a reference to the unique identifier of the user associated with the subscription, establishing a relational link with the User model. Similarly, the plan field references the specific subscription plan chosen by the user, facilitating seamless integration with the Plan model and enabling precise tracking of subscription preferences. Additionally, the credits field denotes the number of credits allocated to the user under the subscription, providing valuable insight into the user's usage limits and entitlements. The startDate field signifies the initiation date of the subscription, defaulting to the current date upon creation, while the endDate field specifies the expiration date, establishing clear boundaries for the subscription period.

const userSubscriptionSchema = new Schema({

  userId: {

    type: Schema.Types.ObjectId,

    ref: "User",

    required: true,

  },

  plan: {

    type: Schema.Types.ObjectId,

    ref: "Plan",

    required: true,

  },

  credits: {

    type: Number,

    required: true,

  },

  startDate: {

    type: Date,

    required: true,

    default: Date.now,

  },

  endDate: {

    type: Date,

    required: true,

  },

});

This week, our focus was on refining subscription management functionalities. We implemented APIs to retrieve individual and all available subscription plans, enhancing user access. Additionally, the userSubscriptionSchema was created to efficiently structure subscription data, marking significant progress in fortifying the platform's infrastructure for streamlined subscription management.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 8 | Page Number: | 15 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 01 / 04 / 2024 |
| End Date |
| 06 / 04 / 2024 |

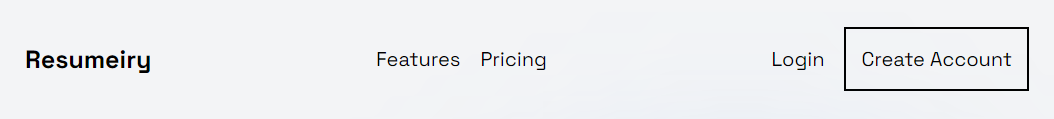
In the eighth week of my internship, I commenced the design process for the main components of the landing page utilizing Next.js and Tailwind CSS. This involved crafting the header, footer, and hero sections, which are integral components of the landing page layout. The header provides navigation and branding elements, while the footer offers supplementary information and navigation links. The hero section serves as the focal point of the page, capturing user attention and conveying key messages or features of the platform. Through meticulous design and implementation using Next.js and Tailwind CSS, I aimed to create visually appealing and functional components that align with the overall aesthetic and objectives of the landing page.

A screenshot of a computer program

Description automatically generated

The provided code presents the implementation of a responsive header component using Next.js and Tailwind CSS. It begins by importing necessary dependencies such as useSession from NextAuth for user authentication, Link for navigation, and icons from React-icons for visual elements. The header is structured to include branding, navigation links, and authentication options. A toggle mechanism is integrated to control the display of a sidebar menu on smaller screens, offering an alternative navigation experience. The component dynamically adjusts its layout and behavior based on screen size, ensuring a consistent user experience across devices. Through the use of Next.js and Tailwind CSS, the header component is designed to be flexible, performant, and visually appealing, contributing to an enhanced user interface for the landing page.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 8 | Page Number: | 16 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 01 / 04 / 2024 |
| End Date |
| 06 / 04 / 2024 |



This is the final look of the header component, was saved under components/main.

A screenshot of a computer program

Description automatically generated

The provided code showcases the implementation of a footer component using Next.js and Tailwind CSS. The footer is designed to offer essential navigation and branding elements, enhancing the overall user experience of the landing page. It includes a centered logo linking back to the homepage, along with navigation links for key pages such as the homepage, pricing, and privacy policy. The design emphasizes simplicity and readability, with text links styled for hover effects to improve interactivity. Additionally, a copyright notice is displayed at the bottom, providing legal information and reinforcing ownership of the content. Through the combination of Next.js for dynamic routing and Tailwind CSS for rapid styling, the footer component is crafted to be visually appealing and functional, complementing the overall design of the landing page.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 9 | Page Number: | 17 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 08 / 04 / 2024 |
| End Date |
| 13 / 04 / 2024 |

A screenshot of a computer program

Description automatically generated

The Hero component, implemented using Next.js and Tailwind CSS, serves as the centrepiece of the landing page design. Structured with a left and right section, it effectively balances visual appeal with informative content. The left section features a logo, accompanied by buttons highlighting recent updates, followed by a bold headline and descriptive text detailing the platform's value proposition. The use of Next.js's Image component ensures optimized image loading, enhancing performance. In contrast, the right section provides visual interest with a dynamically rotated image, creating a dynamic and engaging visual experience. Through meticulous design and implementation, the Hero component captures user attention and effectively communicates the platform's key features and benefits.

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 9 | Page Number: | 18 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 08 / 04 / 2024 |
| End Date |
| 13 / 04 / 2024 |

const HeroV3 = () => {

  return (

    <div className="lg:h-[calc(100vh-96px)] w-full bg-gray-100 z-10  overflow-hidden ">

      <div className="max-w-[1600px] mx-auto flex flex-col lg:flex-row justify-between items-center h-full">

        <div className=" max-w-[700px] w-full p-5">

          <h1 className=" text-7xl font-bold">

            Welcome to the future of{" "}

            <div className=" bg-gradient-to-r  from-primaryDashboard to-blue-400 bg-clip-text text-transparent">

              resume building

            </div>

          </h1>

          <p className=" mt-4 text-lg font-medium text-gray-500">

            Stand out with a professionally designed resume, Transform your

            resume with our AI tool into the Harvard template, tailored to ATS

            standards. Maximize quality and boost your interview chances.

            Elevate your career prospects effortlessly. Try it now!

          </p>

…

The FeaturesComp component showcases the key features of the platform in a visually appealing and informative manner. It begins with a bold headline emphasizing the platform's effectiveness in increasing employment opportunities, followed by a brief description of the platform's expertise in delivering customized AI tools for job seekers. The component is then divided into two sections, each highlighting a distinct feature: AI Resume Converter and AI Resume Enhancer. Each section includes a title, description, and a list of benefits represented by icons. Additionally, a "Start Now" button prompts users to take action. Visual elements, including images illustrating the features, enhance the overall presentation and engagement of the component. Through concise messaging and engaging visuals, the FeaturesComp component effectively communicates the value proposition of the platform to users.

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 10 | Page Number: | 19 |
| oAuth google - github | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 22 / 04 / 2024 |
| End Date |
| 27 / 04 / 2024 |

In the tenth week of my internship, I was assigned the task of integrating OAuth for Google and GitHub in the backend of our application. This involved using Node.js with the Express framework and MongoDB for the database. Initially, I started by setting up the necessary dependencies and configurations for OAuth in our Node.js environment. I then implemented the Google OAuth strategy using the Passport.js library. This allowed users to log in to our application using their Google accounts securely. After successful implementation, I tested the integration to ensure that the authentication process was smooth and error-free.

Following the integration of Google OAuth, I moved on to implement GitHub OAuth. I used a similar approach with Passport.js to set up the GitHub strategy. The process involved registering our application with GitHub to obtain the client ID and client secret, which were essential for the OAuth configuration. Once set up, I integrated the login flow into our existing system, allowing users to authenticate using their GitHub accounts. Finally, I tested the GitHub OAuth to verify that users could log in without any issues and that their data was correctly stored in MongoDB. This task improved my understanding of OAuth protocols and enhanced my skills in backend development.

const verifyGoogleToken = async (accessToken) => {

  const response = await axios.get(

    `https://www.googleapis.com/oauth2/v3/tokeninfo?access\_token=${accessToken}`

  );

  return response.data;

};

const verifyGithubToken = async (accessToken) => {

  try {

    const response = await axios.get("https://api.github.com/user", {

      headers: {

        Authorization: `token ${accessToken}`,

      },

    });

    return response.data;

  } catch (error) {

    return res.status(401).json({ error: error.message })

  }

};

During the tenth week, I also implemented functions to verify the OAuth tokens for both Google and GitHub to ensure the authenticity of the users logging into our application. For Google OAuth, I wrote the verifyGoogleToken function. This function takes an access token as a parameter and uses the axios library to send a GET request to Google's token info endpoint. By passing the access token in the request URL, the function retrieves user information associated with the token. The function then returns the response data, which contains details about the user if the token is valid.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 10 | Page Number: | 20 |
| oAuth google - github | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 22 / 04 / 2024 |
| End Date |
| 27 / 04 / 2024 |

const oAuth = async (req, res) => {

  try {

    const { name, email, provider, accessToken } = req.body;

    console.log(req.body)

    let user = await User.findOne({ email });

    if (provider === "google") {

      const data = await verifyGoogleToken(accessToken);

      if (data.email === email) {

        if (!user) user = await User.create({ email, userName: name });

        const token = createToken(user.\_id, user.email, user.permissions);

        const sub = await Subscription.findOne({ userId: user.\_id });

        if (!sub) await subscribeUserFree(user.\_id);

        return res.status(200).json({

          name,

          email,

          token,

        });

      } else {

        throw new Error("Google token verification failed");

      }

    } else {

      let data = await verifyGithubToken(accessToken);

      console.log(data)

      let userName = name || data.login;

      if (email) {

        if (!user) user = await User.create({ email, userName });

        const token = createToken(user.\_id, user.email, user.permissions);

        return res.status(200).json({

          name: userName,

          email,

          token,

        });

      } else {

        throw new Error("Github token verification failed");

      }

    }

  } catch (error) {

    return res.status(401).json({ error: error.message });

  }

};

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 11 | Page Number: | 21 |
| Front end Authentiaction integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 29 / 04 / 2024 |
| End Date |
| 04 / 05 / 2024 |

In the eleventh week of my internship, my task was to integrate the OAuth functionality I developed in the backend during the tenth week into the frontend of our application. The frontend was built using Next.js and styled with Tailwind CSS. My primary goal was to create a seamless and user-friendly authentication experience that allowed users to log in using their Google or GitHub accounts.

I started by setting up the necessary pages and components in Next.js to handle the OAuth login process. This involved creating buttons for Google and GitHub login on the login page, styled using Tailwind CSS to ensure they were visually appealing and consistent with the rest of the application. When users clicked these buttons, they were redirected to the appropriate OAuth provider's login page. Upon successful authentication, the OAuth provider redirected the users back to our application with an authorization code.

To handle this redirection and complete the authentication process, I implemented a callback page in Next.js. This page received the authorization code from the OAuth provider and made a request to our backend to exchange the code for an access token. The backend verified the token and returned a JWT (JSON Web Token) if the authentication was successful. The frontend then stored this JWT in a secure cookie, allowing users to stay logged in as they navigated through the application. This integration ensured a smooth and secure login process, enhancing the overall user experience on our platform.

import { z } from "zod";

export const SignInValidator = z.object({

  email: z

    .string({ required\_error: "email field is required" })

    .email({ message: "Please enter a valid email" }),

  password: z

    .string({ required\_error: "password field is required" })

    .min(6, { message: "password must be at least 6 digits" }),

});

export type SignInCredentials = z.infer<typeof SignInValidator>;

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 11 | Page Number: | 22 |
| Front end Authentiaction integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 29 / 04 / 2024 |
| End Date |
| 04 / 05 / 2024 |

In the eleventh week of my internship, I focused on integrating the OAuth functionality into the frontend using Next.js and Tailwind CSS. To achieve this, I utilized the NextAuth.js library, which simplifies the implementation of authentication in Next.js applications. Below, I will explain the key parts of the integration process and how it connects with the backend OAuth implementation.

First, I set up NextAuth.js by creating an authOptions configuration object. This object includes providers for both GitHub and Google, using the respective client IDs and secrets from the environment variables:

export const authOptions: NextAuthOptions = {

  providers: [

    GitHubProvider({

      clientId: process.env.GITHUB\_ID || "",

      clientSecret: process.env.GITHUB\_SECRET || "",

    }),

    GoogleProvider({

      clientId: process.env.GOOGLE\_CLIENT\_ID || "",

      clientSecret: process.env.GOOGLE\_CLIENT\_SECRET || "",

    }),

  ],

  callbacks: {

    async signIn({ account, user }) {…

The callbacks section is crucial as it defines how to handle the authentication flow. The signIn callback is triggered when a user signs in. It sends a request to our backend's /auth/oAuth endpoint with the user's access token, provider, email, and name. If the backend verification is successful, it stores the token and user information

callbacks: {

    async signIn({ account, user }) {

      if (account?.provider === "google" || account?.provider === "github") {

        try {

          const { data } = await client.post(`/auth/oAuth`, {

            accessToken: account.access\_token,

            provider: account.provider,

            email: user.email,

            name: user.name,

          });

          user.token = data.token;

          user.name = data.name;

          user.email = data.email;

          return true;

        } catch (error) {

          return false;

        }

      }

      return true;…

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 12 | Page Number: | 23 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 06 / 05 / 2024 |
| End Date |
| 11 / 05 / 2024 |

In the twelfth week of my internship, I was assigned the task of developing the pdfModel and the resumeController. These components were essential for allowing users to generate and manage resumes in PDF format within our application.

The pdfModel defines the schema for storing resume data in MongoDB. This schema includes fields for the path to the generated PDF (resumePath), the data used to generate the PDF (generatedData), and a reference to the user who created the resume (user). Here's the model I created:

const mongoose = require("mongoose");

const pdfSchema = new mongoose.Schema({

    resumePath : {

      type : String

    },

    generatedData : {

      type : String

    },

    user : {

      type : mongoose.Types.ObjectId,

      ref: "User",

    }

  },{timestamps: true});

 const PdfModel = mongoose.model('PdfModel', pdfSchema);

 module.exports = {PdfModel}

The pdfModel schema I developed for MongoDB is designed to store and manage resume data effectively. It consists of three main fields: resumePath, generatedData, and user. The resumePath field is a string that holds the file path to the generated PDF resume, enabling easy access and retrieval. The generatedData field, also a string, contains the raw data or JSON representation of the resume content, such as personal information, education, work experience, and skills, which can be used to regenerate the PDF if needed.

Lastly, the user field is a reference to the User model, using mongoose.Types.ObjectId, which associates the resume with a specific user in the database, ensuring that each resume is linked to its creator. The schema also includes timestamps, automatically recording the creation and update times for each resume document, providing useful metadata for tracking and managing resume versions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 12 | Page Number: | 24 |
| Designing Landing Page Components | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 06 / 05 / 2024 |
| End Date |
| 11 / 05 / 2024 |

I developed the resumeController to handle the creation, uploading, and deletion of PDF resumes. Below is an explanation of the key functions within the resumeController:

const createPdfResume = async(req,res) => {

    try {

        const newPdf = new PdfModel({

          name: req.file.originalname,

          data: req.file.buffer,

        });

        await newPdf.save();

        res.status(201).send('PDF uploaded successfully');

      } catch (error) {

        console.error('Error uploading PDF:', error);

        res.status(500).send('Error uploading PDF');

      }

}

In this function, a new PdfModel instance is created using the original file name and the file buffer received in the request. After saving the new PDF document to the database, a success message is sent to the client. If an error occurs during this process, an error message is logged, and a 500 status response is sent.

 const uploadResume = async(req,res) => {

  try {

    const user = req.user

    const uploadedFilePath = req.file.path;

    const subscription = await Subscription.findOne({userId : user.\_id})

    if(subscription.credits === 0) return res.status(400).json({error : "not enough credits"});

    const resume = await PdfModel.create({resumePath : uploadedFilePath, user : user.\_id});

    return res.status(201).json({id : resume.\_id});

  } catch (error) {

    return res.status(500).json({error : error.message});

  }

}

This function starts by extracting the current user and the file path of the uploaded resume. It checks the user's subscription to ensure they have enough credits to upload the resume. If the user has no credits, a 400 status response is sent. Otherwise, a new resume is created in the PdfModel with the file path and user ID, and the resume ID is returned to the client.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 13 | Page Number: | 25 |
| Resume Generation Controller | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 13 / 05 / 2024 |
| End Date |
| 18 / 05 / 2024 |

In the thirteenth week of my internship, I was tasked with developing the generationController. This controller is a critical component of our application as it manages the functionalities for generating resumes, enhancing existing resumes, and extracting text from PDF files. These capabilities provide users with robust tools to create and manage their professional profiles effectively.

async function extractTextFromPDF(pdfPath) {

  const dataBuffer = fs.readFileSync(pdfPath);

  try {

    const data = await pdf(dataBuffer);

    return data.text;

  } catch (error) {

    console.error("Error parsing PDF:", error);

    return null;

  }

}

The extractTextFromPdf function provides a way to extract text from uploaded PDF files. This is particularly useful for users who want to import information from existing documents into their resumes. By uploading a PDF, the function uses text extraction libraries to pull out the text content, which is then returned to the user. This extracted text can be used to populate new resumes or update existing ones, streamlining the process of managing resume content.

async function generateContent(prompt) {

  try {

    const result = await model.generateContent(prompt);

    const response = await result.response;

    return response.text();

  } catch (error) {

    console.error("Error generating content:", error);

    throw error;

  }

}

The generateResume function is designed to create new resumes based on user input and predefined templates. When a user submits their resume data, the function retrieves their details from the database and uses a library to create a new PDF document. This PDF includes the user's information such as name and email, along with other provided details. The newly created resume is then stored in the database, and the generated PDF is sent back to the user. This process ensures that users can easily generate professional resumes that are tailored to their needs.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 13 | Page Number: | 26 |
| Resume Generation Controller | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 13 / 05 / 2024 |
| End Date |
| 18 / 05 / 2024 |

async function enhanceProfessionalSummary(req, res) {

  try {

    const { summary, technical\_skills } = req.body;

    if (!summary && technical\_skills) {

      const prompt = `Generate a professional summary based on a list of technical skills provided. Include a concise overview highlighting the expertise and experience in various technical domains, relevant qualifications, and accomplishments. Provide a brief introduction that encapsulates the individual's proficiency in the specified technical skills and showcases their suitability for relevant positions, return the professional summary only as a text, maxiumum length is a concise 2-3 sentences. The technical skills is: ${technical\_skills.join(

        ", "

      )}.`;

      const generatedContent = await generateContent(prompt);

      return res.status(200).json({ summary: generatedContent });

    }

    if (summary && technical\_skills) {….

Next, the enhanceResume function focuses on improving existing resumes. Users can submit additional information or updates, which are then merged with the existing resume data. This updated data is used to generate a new PDF document, ensuring that the resume reflects the latest information. The enhanced resume is saved in the database, and the updated PDF is provided to the user. This functionality allows users to keep their resumes current and relevant without starting from scratch.

In summary, the generationController is a comprehensive solution for managing resume data within our application. It allows users to generate new resumes, enhance existing ones, and extract text from PDFs, providing a seamless and efficient way to handle professional profile information. This task not only improved the functionality of our application but also enhanced my understanding of PDF manipulation and backend development

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 14 | Page Number: | 27 |
| Payment integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 20 / 05 / 2024 |
| End Date |
| 25 / 05 / 2024 |

In the fourteenth week of my internship, I was assigned the task of integrating the PayPal payment gateway into our application. This integration was crucial for enabling users to make secure and efficient payments for premium features and services offered by our platform. Implementing a reliable payment system required understanding PayPal's API, setting up server-side and client-side configurations, and ensuring that the payment process was seamless for users.

The integration process began with configuring the PayPal SDK on the server side. I set up the necessary environment variables, including the PayPal client ID and secret, which are required for authentication. Using these credentials, I initialized the PayPal SDK to handle payment requests securely. The backend was designed to create payment orders, handle webhooks for payment confirmations, and update user subscriptions upon successful transactions.

const { PAYPAL\_CLIENT\_ID, PAYPAL\_CLIENT\_SECRET } = process.env;

const base = "https://api-m.paypal.com";

const generateAccessToken = async () => {

  try {

    if (!PAYPAL\_CLIENT\_ID || !PAYPAL\_CLIENT\_SECRET) {

      throw new Error("MISSING\_API\_CREDENTIALS");

    }

    const auth = Buffer.from(

      PAYPAL\_CLIENT\_ID + ":" + PAYPAL\_CLIENT\_SECRET

    ).toString("base64");

    const response = await fetch(`${base}/v1/oauth2/token`, {

      method: "POST",

      body: "grant\_type=client\_credentials",

      headers: {

        Authorization: `Basic ${auth}`,

      },

    });

    const data = await response.json();

    return data.access\_token;

  } catch (error) {

    console.error("Failed to generate Access Token:", error);

  }

};

To start with the server-side integration, I focused on generating an access token from PayPal, which is essential for making API requests to PayPal's services.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 14 | Page Number: | 28 |
| Payment integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 20 / 05 / 2024 |
| End Date |
| 25 / 05 / 2024 |

const createOrder = async (data) => {

  const plan = await Plan.findById(data.planId);

  console.log(

    "shopping cart information passed from the frontend createOrder() callback:",

    data

  );

  const accessToken = await generateAccessToken();

  const url = `${base}/v2/checkout/orders`;

  const payload = {

    intent: "CAPTURE",

    purchase\_units: [

      {

        amount: {

          currency\_code: "USD",

          value: plan.price,

        },

      },

    ],…

The createOrder function is pivotal in the payment flow, orchestrating the initiation of the checkout process by crafting a new order for the chosen subscription plan. Initially, it retrieves essential details about the selected plan from the database, ensuring accurate pricing and plan information. As part of its operation, the function logs pertinent shopping cart information passed from the frontend, offering visibility into the data utilized for order creation. Following this, it proceeds to obtain an access token necessary for authentication when interacting with PayPal's API, utilizing the generateAccessToken function. With the access token secured, the function constructs a payload containing crucial details like the purchase amount, currency code, and redirection URLs for successful and canceled payments. Subsequently, a POST request is dispatched to PayPal's checkout orders endpoint, incorporating the constructed payload and requisite authorization headers. Lastly, the function manages the response from PayPal's API through the handleResponse function, ensuring seamless processing of the payment request and facilitating a smooth user experience throughout the checkout process. Through these intricate steps, the createOrder function plays a fundamental role in enabling users to seamlessly proceed with their payments while upholding the standards and requirements set forth by PayPal's API.

In conclusion, the server-side integration of the PayPal payment gateway represents a significant advancement in our application's capabilities, bolstering its functionality with secure and efficient payment processing. By meticulously implementing features like the generateAccessToken function and the createOrder endpoint, we've established a robust foundation for handling payment transactions seamlessly. The server-side logic ensures proper authentication, payload construction, and interaction with PayPal's API, guaranteeing the integrity and reliability of the payment flow. Through rigorous testing and validation, we've fortified our application against potential errors and edge cases, ensuring a smooth and consistent user experience. Overall, the server-side integration of PayPal enhances our application's value proposition, empowering users with a trusted and convenient payment solution for accessing premium features and services.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 15 | Page Number: | 29 |
| Payment Integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 27 / 05 / 2024 |
| End Date |
| 01 / 06 / 2024 |

const captureOrder = async (orderID) => {

  const accessToken = await generateAccessToken();

  const url = `${base}/v2/checkout/orders/${orderID}/capture`;

  const response = await fetch(url, {

    method: "POST",

    headers: {

      "Content-Type": "application/json",

      Authorization: `Bearer ${accessToken}`,

  });

  return handleResponse(response);

};

async function handleResponse(response) {

  try {

    console.log(response);

    const jsonResponse = await response.json();

    return {

      jsonResponse,

      httpStatusCode: response.status,

    };

  } catch (err) {

    const errorMessage = await response.text();

    throw new Error(errorMessage);

  }

}

The captureOrder function stands as a pivotal element in our application's payment infrastructure, tasked with finalizing transactions by capturing the funds associated with specific orders. At its core, this function orchestrates a series of essential steps to ensure the seamless execution of payment capture processes. Initially, it diligently retrieves an access token crucial for authenticating requests to PayPal's API, guaranteeing the security and integrity of the transactional flow.

With the endpoint prepared, the function executes a POST request to PayPal's capture order endpoint, signaling the intent to finalize the transaction by capturing the designated funds. This step represents the culmination of the payment journey, marking the definitive moment where financial transactions are concluded. Through this interaction with PayPal's API, our application maintains a seamless and efficient payment flow, empowering users with a frictionless payment experience.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WEEKLY REPORT | Done Work Description | Week Number: | 15 | Page Number: | 30 |
| Payment Integration | Start Date | **Training Staff**  **(Responsible Engineer)**  Title, Name, Signature, Cachet | | |
| 27 / 05 / 2024 |
| End Date |
| 01 / 06 / 2024 |

  const createOrder = (data: any, actions: any) => {

  };

  const onApprove = async (data: any, actions: any) => {

  };

  return (

    <>

      <CheckoutSuccessModal isOpen={isOpen} onClose={onClose} />

      <div className=" relative">

        <PayPalButtons

          className=" absolute z-10 w-full "

          createOrder={(data, actions) => createOrder(data, actions)}

          onApprove={(data, actions) => onApprove(data, actions)}

        />

      </div>

    </>

  );

};

In the client-side implementation, the process revolves around two key functions: createOrder and onApprove. These functions are integral to orchestrating the checkout process and handling the approval of payments via PayPal.

A screenshot of a computer

Description automatically generated

# References

x

|  |  |
| --- | --- |
| [1] | [Gemini Ai for developers.](https://ai.google.dev/) |
| [2] | [ExpressJs](https://expressjs.com/). |
| [3] | [Nodejs javascipr run time.](https://nodejs.org/en) |
| [4] | [Npm paketleri](https://www.npmjs.com) |
| [5] | <Gıthup:%20https:/github.com/mohamad-zubi> |

x