\*08-04-2025

4th Internship

Every week evaluation will be there and also trained with project.

Milestones will be given by Vijay . I should finsh and submit the report.

To do List:

**1.Solid and Acid** principle for development.

**2.Response Structure (API).**

**3.ResponseEntity.**

Documents-I need to go through and upskill the concepts.

**4.GithubGUI**-how to push and pull the code.

**5.Shared Document**-developer need to create document of API,URL,Role,Post,Put for frontend team or other teams.

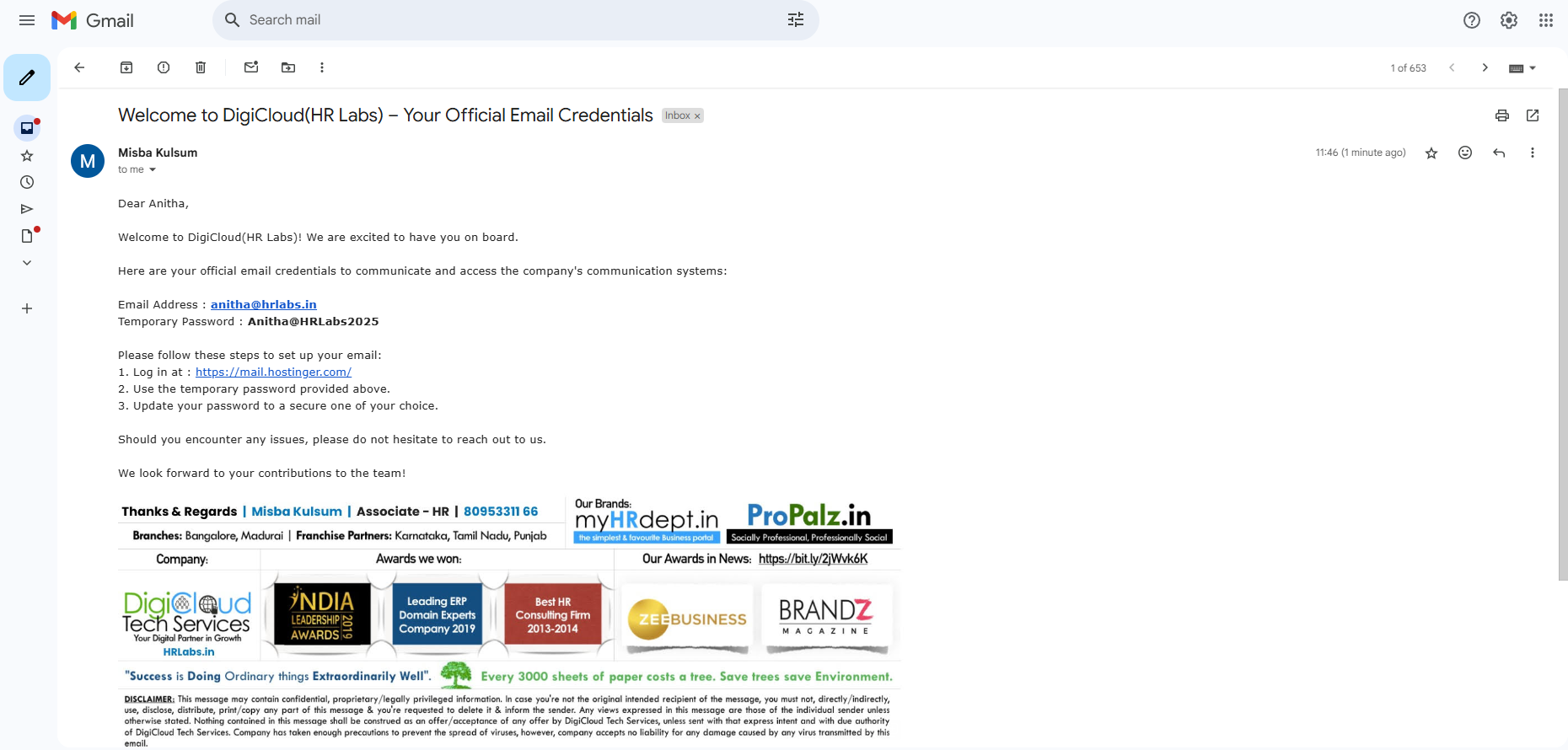
6.**Falcal Dependency** for automatically created the document based on our code.

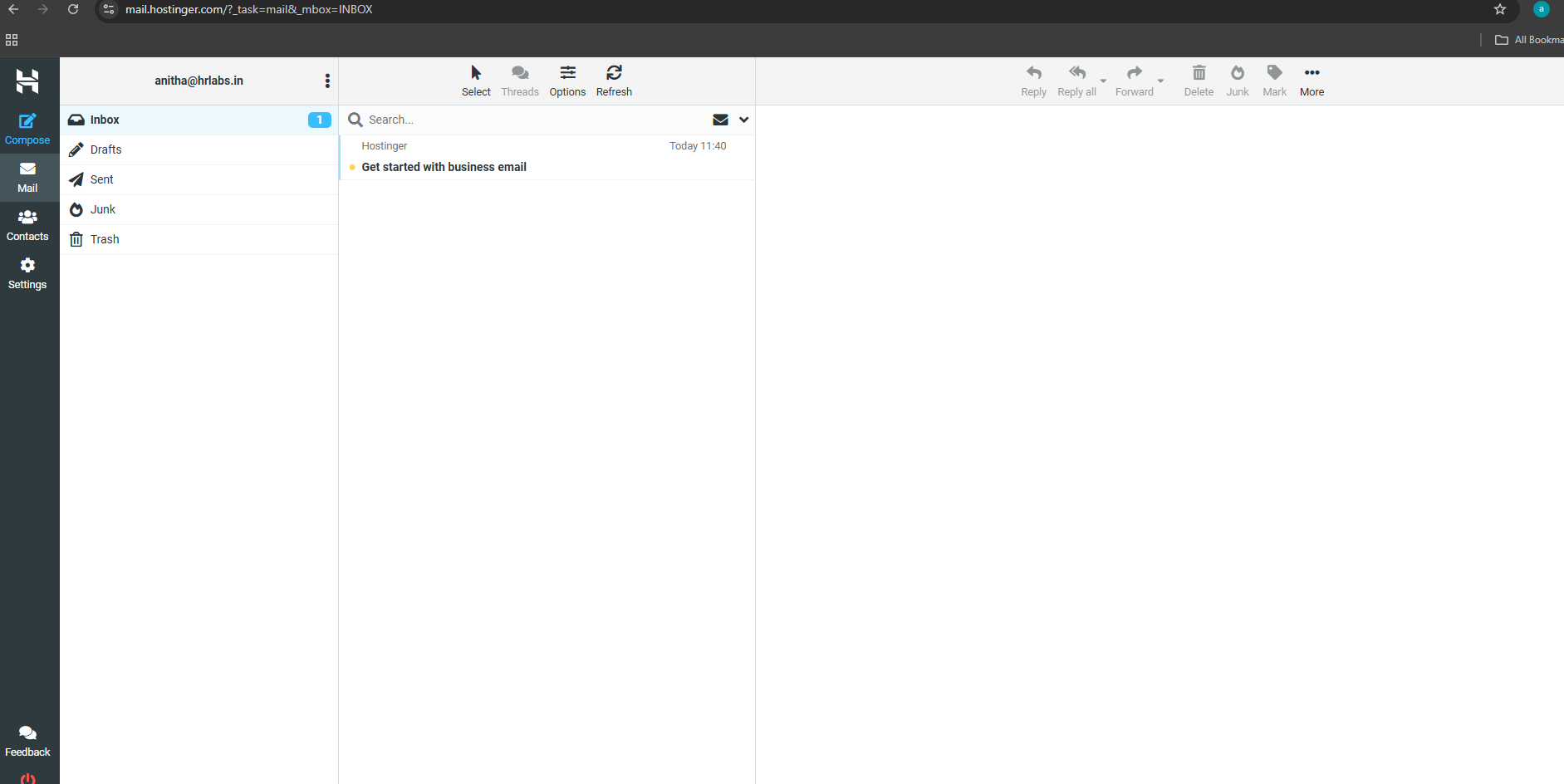
7.**Swager @description**

**8.DTO**

**9.Security**

**10.Postman**

**Official email **

****

**1.Solid and Acid**

**The SOLID and ACID principles are two foundational concepts in software development, but they apply to different areas:**

**✅ SOLID Principles – *Object-Oriented Programming (OOP) Design***

**These are five design principles for writing maintainable, scalable, and understandable OOP code, especially in backend and frontend applications (e.g., in Java, C#, JavaScript, TypeScript):**

1. **S – Single Responsibility Principle (SRP)**

**A class should have only one reason to change.**

* + **Each class or module should do one thing only.**
  + **Keeps code modular and easier to test.**

1. **O – Open/Closed Principle (OCP)**

**Software entities should be open for extension but closed for modification.**

* + **You can extend behavior without changing existing code.**
  + **Encourages the use of interfaces and abstraction.**

1. **L – Liskov Substitution Principle (LSP)**

**Objects of a superclass should be replaceable with objects of its subclasses.**

* + **A subclass should behave like its parent class without breaking the application.**

1. **I – Interface Segregation Principle (ISP)**

**No client should be forced to depend on methods it does not use.**

* + **Split large interfaces into smaller, more specific ones.**

1. **D – Dependency Inversion Principle (DIP)**

**High-level modules should not depend on low-level modules. Both should depend on abstractions.**

* + **Use dependency injection and interfaces to reduce tight coupling.**

**🧪 ACID Principles – *Database Transactions***

**ACID is a set of properties that guarantee reliable processing of database transactions, mainly used in relational databases (like MySQL, PostgreSQL):**

1. **A – Atomicity**

**A transaction is all or nothing.**

* + **If any part of the transaction fails, the entire transaction fails and the database is left unchanged.**

1. **C – Consistency**

**A transaction must bring the database from one valid state to another.**

* + **All rules and constraints are respected before and after the transaction.**

1. **I – Isolation**

**Transactions should not interfere with each other.**

* + **Concurrent transactions are executed as if they were run sequentially.**

1. **D – Durability**

**Once a transaction is committed, it remains so—even in case of a system crash.**

* + **Ensures data is saved to disk and not just in memory.**

**🔁 Summary of When to Use**

| **Principle** | **Area** | **Purpose** |
| --- | --- | --- |
| **SOLID** | **Software design (OOP)** | **Write clean, maintainable, extendable code** |
| **ACID** | **Database transactions** | **Ensure reliable, safe data operations** |

**4.GitHubGUI**

**🖥️ What is GitHub GUI?**

**GitHub GUI usually refers to a Graphical User Interface tool that helps you interact with your GitHub repositories without using command-line.**

**Instead of typing git clone, git push, etc., you can click buttons to do it.**

**✅ Popular GitHub GUI Tools**

**1. GitHub Desktop (Official Tool) 🎯**

* **✅ Made by GitHub**
* **✅ Very beginner-friendly**
* **✅ Available for Windows and Mac**

**📥 Download from:** [**https://desktop.github.com/**](https://desktop.github.com/)

**What you can do:**

* **Clone repositories**
* **Commit and push code**
* **Switch branches**
* **Create new branches**
* **View changes & history**

**2. Other GUI Git Clients (Optional)**

| **Tool** | **Description** |
| --- | --- |
| **Sourcetree** | **Powerful UI for Git (by Atlassian)** |
| **GitKraken** | **Visual & modern Git client** |
| **VS Code Git UI** | **Git features built into VS Code editor** |

**But for starters, I highly recommend GitHub Desktop. It's the easiest way to integrate with your existing GitHub account.**

**🔌 Can You Integrate GitHub GUI with Your Account?**

**Yes! Since you already have a GitHub account:**

1. **Download and install GitHub Desktop**
2. **Open it → Sign in with GitHub**
3. **It connects directly to your repositories**
4. **Now you can:**
   * **Clone your repo to local**
   * **Make changes**
   * **Push back with GUI**

**📸 Example Flow**

1. **Open GitHub Desktop**
2. **Click "Clone a Repository"**
3. **Select from your GitHub account**
4. **Code locally**
5. **Use buttons: Commit → Push**

**No terminal needed 😄**

**Would you like me to walk you through installing and using GitHub Desktop step-by-step?**

**8.DTO-Data Transfer Object**

**Absolutely! Let’s simplify it a lot — like explaining to someone new to coding.**

**🧱 What is a DTO?**

**👉 DTO means Data Transfer Object.  
It’s just a simple class used to send or receive data in your app — usually between backend and frontend.**

**🎯 Why Do We Use DTO?**

**🔒 1. To hide sensitive info**

**You don’t want to send things like passwords or internal details to the frontend.**

**🍔 2. To give only what’s needed**

**Instead of giving a full object (with too much info), you send only the necessary data.**

**🧠 Very Simple Example:**

**Imagine you have a User saved in the database like this:**

**public class User {**

**private Long id;**

**private String name;**

**private String email;**

**private String password; // 🔐 sensitive**

**}**

**Now, when someone logs in, you don’t want to send the password to the frontend, right?**

**So you create a DTO like this:**

**public class UserDTO {**

**private Long id;**

**private String name;**

**private String email;**

**}**

**Then, your backend sends only this DTO to the frontend — safe and clean! 🎯**

**🍕 Easy Analogy**

**You go to a restaurant and order pizza.  
You don’t get the whole kitchen or the recipe, you only get the pizza — that’s your DTO.**

**The kitchen (database/entity) is big and complex.  
You just need your food (clean, simple data = DTO).**

**✅ Summary (Super Simple):**

| **Thing** | **What it is** |
| --- | --- |
| **Entity** | **Full database object** |
| **DTO** | **Simple version you send/receive** |
| **Why?** | **To protect, simplify, and separate data** |
| **5.Shared Document**  **Perfect! Let’s go step-by-step on how a backend developer creates a shared document for APIs (like the ones your manager was talking about).**  **👨‍💻 How Developers Create a Shared Document in Real Projects:**  **✅ Option 1: Manually (Old way)**  **They might write the APIs in a document or Excel sheet:**   | **API Name** | **Method** | **URL** | **Description** | **Role** | | --- | --- | --- | --- | --- | | **Get Employees** | **GET** | **/api/employees** | **List all employees** | **Admin** | | **Apply for Leave** | **POST** | **/api/leave/apply** | **Employee applies** | **Employee** | | **Get Leave Status** | **GET** | **/api/leave/status** | **See leave history** | **Both** |   **⚠️ Disadvantages:**   * **Hard to maintain** * **Easy to forget updating when code changes**   **✅ Option 2: Automatically (Modern way)**  **Using tools like springdoc-openapi (Swagger UI)**  **This is the preferred way! 🎯**  **🔧 Steps to Create Shared API Docs Automatically (Swagger UI)**  **📁 1. Add Dependency in pom.xml (for Maven project)**  **<dependency>**  **<groupId>org.springdoc</groupId>**  **<artifactId>springdoc-openapi-ui</artifactId>**  **<version>1.7.0</version>**  **</dependency>**  **🧑‍💻 2. Write Your API as Usual in Spring Boot**  **@RestController**  **@RequestMapping("/api/employees")**  **public class EmployeeController {**  **@GetMapping**  **public List<Employee> getAllEmployees() {**  **// return all employees**  **}**  **@PostMapping("/leave")**  **public String applyLeave(@RequestBody LeaveRequest request) {**  **// apply for leave**  **}**  **}**  **🌐 3. Run Your Project**  **Start your Spring Boot app:**  **mvn spring-boot:run**  **🔍 4. Open Swagger UI in Browser**  **Go to:**  **http://localhost:8080/swagger-ui/index.html**  **You’ll see a full shared documentation UI like:**   * **All API endpoints** * **Parameters** * **Example requests/responses** * **Roles (if you use security annotations like @PreAuthorize)**   **✅ This is your shared API document — auto-created, up-to-date, and shareable with other developers or frontend teams.**  **🔐 Bonus (Add Roles if needed)**  **If you use Spring Security, add something like:**  **@PreAuthorize("hasRole('ADMIN')")**  **@GetMapping("/admin")**  **public String onlyAdminAccess() {**  **return "Secret data";**  **}**  **Swagger UI will show this endpoint is protected and needs a role.**  **✅ Final Summary**   | **Step** | **Description** | | --- | --- | | **Add Dependency** | **springdoc-openapi-ui in pom.xml** | | **Write APIs** | **In your Spring Boot controller as usual** | | **Run App** | **Start Spring Boot project** | | **Open Swagger** | **http://localhost:8080/swagger-ui/index.html** | | **Share It** | **Copy that link and give to frontend/testers/team** | | **7.Swagger**  Great! Let’s talk about how to use **@Description-like annotations** in Swagger (OpenAPI) with Spring Boot to make your API documentation clear and helpful 📝  **🌟 Goal:**  You want to show **descriptions** for:   * Controllers (what the API is for) * Endpoints (what each method does) * Parameters * Models (DTOs)   **✅ Tools You Need**  You should be using the **springdoc-openapi-ui** dependency (Swagger 3):  <dependency>  <groupId>org.springdoc</groupId>  <artifactId>springdoc-openapi-ui</artifactId>  <version>1.7.0</version>  </dependency>  **✍️ Add Descriptions in Your Code**  **1. Controller-level Description**  @Tag(name = "Employee API", description = "API for managing employee operations")  @RestController  @RequestMapping("/api/employees")  public class EmployeeController {  **2. Endpoint (Method) Description**  @Operation(  summary = "Get all employees",  description = "This endpoint returns a list of all employees from the database"  )  @GetMapping  public List<Employee> getAllEmployees() {  // code here  }  **3. Parameter Descriptions**  @Operation(summary = "Get employee by ID")  @ApiResponses(value = {  @ApiResponse(responseCode = "200", description = "Employee found"),  @ApiResponse(responseCode = "404", description = "Employee not found")  })  @GetMapping("/{id}")  public Employee getEmployeeById(  @Parameter(description = "ID of the employee to retrieve")  @PathVariable Long id) {  // code here  }  **4. Model (DTO) Description**  @Schema(description = "Data Transfer Object for Employee")  public class EmployeeDTO {  @Schema(description = "Employee's full name", example = "John Doe")  private String name;  @Schema(description = "Email address", example = "john@example.com")  private String email;  }  **🎯 Result**  When you go to: http://localhost:8080/swagger-ui/index.html  You’ll see:   * Nicely formatted docs * Endpoint summaries & details * Parameter tooltips * Model property descriptions   **📌 Summary of Swagger Annotations**   | **Annotation** | **Purpose** | | --- | --- | | @Tag | Description for controller | | @Operation | Description for endpoint | | @Parameter | Description for method parameters | | @Schema | Description for model/DTO properties | | @ApiResponse | Describe success/failure responses |   Would you like me to generate a complete working example you can copy-paste and run?  **10.Postman**  **Awesome! Let’s now talk about Postman — it's a super handy tool every developer should know 💪**  **🚀 What is Postman?**  **Postman is a GUI tool used to test APIs — just like a browser is used to open websites.**  **You can use Postman to:**   * **Send GET, POST, PUT, DELETE requests** * **Test your backend APIs** * **See the response (data, status, errors)** * **Send body data (like JSON), headers, auth tokens, etc.**   **🔧 Example: Testing a POST API in Postman**  **Let’s say your Spring Boot app has this API:**  **POST http://localhost:8080/api/employee**  **And it expects this JSON:**  **{**  **"name": "John",**  **"email": "john@example.com"**  **}**  **Here’s how you test this in Postman:**   1. **Open Postman** 2. **Set method to POST** 3. **Enter URL: http://localhost:8080/api/employee** 4. **Click on Body tab → select raw → choose JSON** 5. **Paste this:** 6. **{** 7. **"name": "John",** 8. **"email": "john@example.com"** 9. **}** 10. **Click Send 🚀**   **You’ll get the response from your backend like:**  **{**  **"id": 1,**  **"name": "John",**  **"email": "john@example.com"**  **}**  **💡 Why Use Postman?**   | **Feature** | **Use Case** | | --- | --- | | **Test APIs** | **Before connecting frontend** | | **Debug backend** | **See exact errors or success** | | **Add Headers/Auth** | **Easily test with tokens, headers, roles, etc.** | | **Save Requests** | **Save and organize requests in collections** | | **Share with team** | **Share collections with team members** |   **📸 Bonus: Looks similar to frontend testing**  **Yes! Postman simulates the request just like a frontend would, but:**   * **Without writing any UI code** * **Without a browser** * **Just hit API directly**   **✅ Summary**   | **Tool** | **Purpose** | | --- | --- | | **Postman** | **Test and interact with APIs** | | **Swagger** | **View and understand APIs (Docs)** | | **Browser** | **Use your web pages** |   **Would you like a Postman testing guide PDF or a sample collection to import and test your Spring Boot project?** |  | |  |