System Requirements Specification Index

For

Tax Management Application

Version 1.0

TABLE OF CONTENTS

PROJECT ABSTRACT

In the world of financial management, there's a pressing need to modernize tax handling. The CFO of a leading financial institution, challenges a team of developers to create a Fullstack Tax Management Application.

Your task is to develop a digital solution that seamlessly manages tax calculations and related specifications, providing users with an intuitive platform for effective tax management.

BACKEND-JAVA

1. PROBLEM STATEMENT

The **Tax Management Application** is a Java-based RESTful Web API utilizing Spring Boot, with MySQL as the database. The application aims to provide a comprehensive platform for managing and organizing all tax related data for a company.

To build a robust backend system that effortlessly handles tax calculations. Here's what the developers need to accomplish:

FOLLOWING IS THE REQUIREMENT SPECIFICATION:

		Tax Management Application
	1	Tax
Tax Module Functionalities		
	1	Get all taxes
	2	Get tax by id
	3	Create a new tax
	4	Update a tax by id
	5	Delete a tax by id

2. ASSUMPTIONS, DEPENDENCIES, RISKS / CONSTRAINTS

2.1 Tax Constraints
- When fetching tax by id, if tax ID does not exist, the service method should throw a "Tax not found" message in the ResourceNotFoundException class.
- When updating a tax, if tax ID does not exist, the service method should throw a "Tax not found" message in the ResourceNotFoundException class.
- When removing a tax , if the tax ID does not exist, the service method should throw a "Tax not found" message in the ResourceNotFoundException class.

2.2 Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exceptions if data is invalid.
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only
- Do not change, add, remove any existing methods in service layer
- In Repository interfaces, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data wrapped in **ResponseEntity**

3. BUSINESS VALIDATIONS

- FormType should not be blank.
- FillingDate should not be null and must be current or past date.
- TotalTaxAmount should not be null and the value must be a positive number.
- UserID should not be null.

4. DATABASE OPERATIONS

- Tax class should be binded with a "taxes" table.
- taxFormId should be treated as primary key and should be generated with IDENTITY technique and should have a column name as "tax_form_id".
- formType should not be a nullable field and have a column name as "form_type".
- fillingDate should not be a nullable field, have a column name as "filling_date" and should have TemporalType.DATE.
- totalTaxAmount should not be a nullable field and have a column name as "total_tax_amount".
- userId should not be a nullable field and have a column name as "user_id".

4. REST ENDPOINTS

Rest End-points to be exposed in the controller along with method details for the same to be created

4.1 TaxController

URL Exposed		Purpose
1. /api/taxes		Fetches all the taxes
Http Method	GET	
Parameter	-	
Return	ResponseEntity<List<TaxDTO>>	
2. /api/taxes/{id}		Fetches a tax by id
Http Method	GET	
Parameter 1	Long (id)	
Return	ResponseEntity<TaxDTO>	
3. /api/taxes		Creates a new tax
Http Method	POST **The tax data to be created should be received in @RequestBody**	
Parameter	TaxDTO	
Return	ResponseEntity<TaxDTO>	
4. /api/taxes/{id}		Updates a tax by id
Http Method	PUT **The tax data to be updated should be received in @RequestBody**	
Parameter 1	Long (id) TaxDTO	
Return	ResponseEntity<TaxDT	

	O>	
5. /api/taxes/{id}		
Http Method	DELETE	
Parameter 1	Long (id)	Deletes a tax by id
Return	-	

5. TEMPLATE CODE STRUCTURE

5.1 PACKAGE: COM.TAXMANAGEMENT

Resources

Class/Interface	Description	Status
TaxManagementApplication (Class)	This is the Spring Boot starter class of the application.	Already implemented.

5.2 PACKAGE: COM.TAXMANAGEMENT.REPOSITORY

Resources

Class/Interface	Description	Status
TaxRepository (interface)	• Repository interface exposing CRUD functionality for tax Entity. • You can go ahead and add any custom methods as per requirements.	Already implemented.

5.3 PACKAGE: COM.TAXMANAGEMENT.SERVICE

Resources

Class/Interface	Description	Status
TaxService (interface)	• Interface to expose method signatures for tax related functionality. • Do not modify, add or delete any method.	Already implemented.

5.4 PACKAGE: COM.TAXMANAGEMENT.SERVICE.IMPL

Resources

Class/Interface	Description	Status
TaxServiceImpl (class)	• Implements TaxService. • Contains template method implementation. • Need to provide implementation for tax related functionalities. • Do not modify, add or delete any method signature	To be implemented.

5.5 PACKAGE: COM.TAXMANAGEMENT.CONTROLLER

Resources

Class/Interface	Description	Status
TaxController (Class)	• Controller class to expose all rest-endpoints for tax related activities. • Should also contain local exception handler methods	To be implemented

5.6 PACKAGE: COM.TAXMANAGEMENT.DTO

Resources

Class/Interface	Description	Status
TaxDTO (Class)	• Use appropriate annotations for validating attributes of this class.	Partially implemented.

Class/Interface	Description	Status
Response (Class)	● DTO created for response objects.	Already implemented.

5.7 PACKAGE: COM.TAXMANAGEMENT.ENTITY

Resources

Class/Interface	Description	Status
Tax (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **taxId** as primary key. ● Map this class with a **taxes** table. ● Generate the **taxId** using the IDENTITY strategy	Partially implemented.

5.8 PACKAGE: COM.TAXMANAGEMENT.EXCEPTION

Resources

Class/Interface	Description	Status
ResourceNotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the tax info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.

6. METHOD DESCRIPTIONS

6.1 TaxServiceImpl Class - Method Descriptions

Method	Task	Implementation Details
createTax	To create and save a new tax entry	- Accepts a `TaxDTO` object as input - Maps DTO to entity using `modelMapper` - Calls `taxRepository.save(tax)` to persist the data - Maps the saved entity back to DTO and returns it
deleteTaxById	To delete a tax entry by ID	- Checks existence using `taxRepository.existsById(id)` - If exists, deletes using `taxRepository.deleteById(id)` and returns true - If not found, throws `ResourceNotFoundException` with message 'Tax not found'
getAllTaxes	To retrieve all tax records	- Calls `taxRepository.findAll()` to fetch all records - Maps each entity to `TaxDTO` using `modelMapper` - Returns the list of mapped DTOs
getTaxById	To retrieve a single tax record by ID	- Uses `taxRepository.findById(id)` to find the tax - If found, maps it to `TaxDTO` and returns - If not found, throws `ResourceNotFoundException` with message 'Tax not found'
updateTax	To update an existing tax record	- Extracts ID from `taxDTO.getTaxFormId()` - Checks if tax exists using `taxRepository.existsById(taxId)` - If exists, maps `TaxDTO` to `Tax`, then saves using `taxRepository.save(tax)` - Maps updated entity back to DTO and returns it - If not found, throws `ResourceNotFoundException` with message 'Tax not found'

6.2 TaxController Class - Method Descriptions

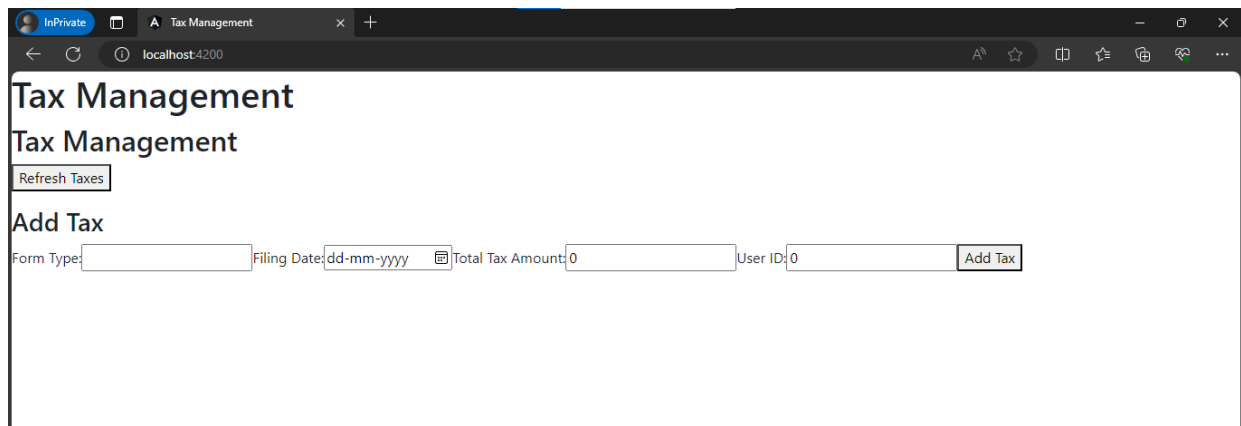| Method	Task	Implementation Details
createTax	To handle creation of a new tax record	- Request type: POST with URL `/api/taxes` - Method name: `createTax` returns `ResponseEntity<TaxDTO>` - Uses `@Valid @RequestBody` to accept `TaxDTO` - Calls `taxService.createTax(model)` - Returns created tax with `HttpStatus.CREATED`
updateTax	To update an existing tax record	- Request type: PUT with URL `/api/taxes/{id}` - Method name: `updateTax` returns `ResponseEntity<TaxDTO>` - Uses `@Valid @RequestBody` to accept updated `TaxDTO` - Calls `taxService.updateTax(model)` - If result is null, return `HttpStatus.NOT_FOUND` - Otherwise, return updated tax with `HttpStatus.OK`
deleteTax	To delete a tax record by ID	- Request type: DELETE with URL `/api/taxes/{id}` - Method name: `deleteTax` returns `ResponseEntity<Void>` - Uses `@PathVariable` to get ID - Calls `taxService.deleteTaxById(id)` - If result is false, return `HttpStatus.NOT_FOUND` - Otherwise, return `HttpStatus.NO_CONTENT`
getTaxById	To retrieve a tax record by ID	- Request type: GET with URL `/api/taxes/{id}` - Method name: `getTaxById` returns `ResponseEntity<TaxDTO>` - Uses `@PathVariable` to get ID - Calls `taxService.getTaxById(id)` - Returns the tax with `HttpStatus.OK`
getAllTaxes	To retrieve all tax records	- Request type: GET with URL `/api/taxes` - Method name: `getAllTaxes` returns `ResponseEntity<List<TaxDTO>>` - Calls `taxService.getAllTaxes()` - Returns list of taxes with `HttpStatus.OK`

FRONTEND-ANGULAR SPA

1 PROBLEM STATEMENT

The **Tax Management Application** frontend is a Single Page Application (SPA) built using Angular. Here's what the frontend developers need to achieve:

The frontend should provide a user-friendly interface for users to manage tax-related tasks like: add tax details, update tax details, delete tax and get all taxes.

2 PROPOSED TAX MANAGEMENT APPLICATION WIREFRAME

UI needs improvisation and modification as per given use case and to make test cases passed.

2.1 HOME PAGE

2.2 SCREENSHOTS

*** Add Tax***

Tax Management

Tax Management

Refresh Taxes

Add Tax

Form Type: ABC   Filing Date: 10-06-2024   Total Tax Amount: 1000   User ID:
001   Add Tax

Tax Management

Tax Management

Refresh Taxes

- ABC - 6/10/2024 - 1000 Select Update Delete

Add Tax

Form Type: Filing Date: dd-mm-yyyy   Total Tax Amount: 0   User ID:
0   Add Tax

Tax Management

Tax Management

Refresh Taxes

- ABC - 6/10/2024 - 1000 Select | Update | Delete

Update Tax

Form Type: ABC   Filing Date: 10-06-2024 📅 Total Tax Amount: 1010   User ID:
1   Update Tax

Tax Management

Tax Management

Refresh Taxes

- ABC - 6/10/2024 - 1010 Select | Update | Delete

Add Tax

Form Type: [   ]   Filing Date: dd-mm-yyyy 📅 Total Tax Amount: 0   User ID:
0   Add Tax

Tax Management

Tax Management

Refresh Taxes

- ABC - 6/10/2024 - 1000 Select Update Delete

Update Tax

Form Type: ABC   Filing Date: 10-06-2024   Total Tax Amount: 1010   User ID:
1   Update Tax

*** Delete Tax***

Tax Management

Tax Management

Refresh Taxes

Add Tax

Form Type: Filing Date: mm/dd/yyyy   Total Tax Amount: 0   User ID: 0
Add Tax

InPrivate   Tax Management

localhost:4200
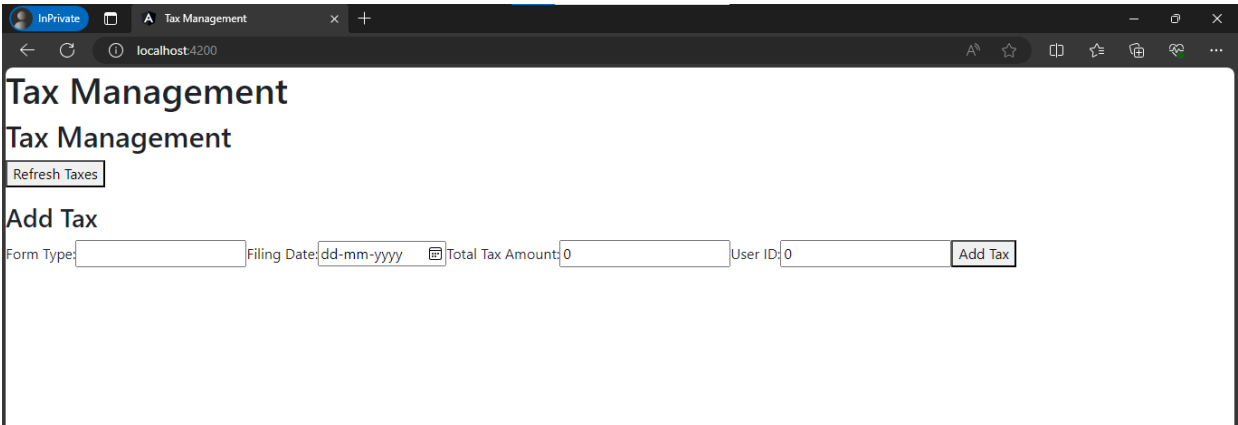
Tax Management

Tax Management

Refresh Taxes

Add Tax

Form Type: Filing Date: dd-mm-yyyy   Total Tax Amount: 0   User ID: 0   Add Tax

3 BUSINESS-REQUIREMENT:

As an application developer, develop the Tax Management Application (Single Page App) with below guidelines:

User Story #	User Story Name	User Story
US_01	Home Page	As a user I should be able to visit the Home page as the default page. ## Acceptance Criteria ### AppComponent **Purpose** • Acts as the shell of the application. • Hosts the `TaxManagementComponent`. **HTML Structure** · Use a top-level `<div>` to wrap everything. · Add a `<h1>` with the text: "Tax Management". · Render the `<app-tax-management>` component below the heading. ### TaxManagementComponent **Purpose** • Handles all UI logic for listing, creating, updating, and deleting tax records. • Handling form input. • Uses a shared service (`TaxService`) to communicate with the backend. **State Variables** **taxes (array of Tax)**

- Stores the full list of tax records.

selectedTax (Tax object)

- Represents the current form's tax data, either for creating or updating.

Functions & Responsibilities

ngOnInit()
- Calls `loadTaxes()` when the component initializes.

loadTaxes()
- Sends a `GET` request to retrieve all tax entries.
- On success: updates the `taxes` list.
- On error:
    - Log to console:
      ```
      console.error('Error loading
      taxes:', error);
      ```

addTax()
- Sends a `POST` request with the `selectedTax` object.
- On success:
    - Refreshes tax list via `loadTaxes()`
    - Resets form with `createEmptyTax()`
- On error:
    - Log to console:
      ```
      console.error('Error adding tax:',
      error);
      ```

showUpdateForm(id)
- Finds the tax by ID from the `taxes` array.
- Prefills the form for editing by updating `selectedTax`.

updateTaxApi()
- Sends a `PUT` request to update the selected tax.
- On success:
    - Refreshes tax list
    - Resets the form

- On error:
  - Log to console:

    ```
    console.error('Error updating
    tax:', error);
    ```

deleteTax(id)
- Sends a DELETE request by `taxFormId`.
- On success:
  - Refreshes the list
  - Clears form via `createEmptyTax()`
- On error:
  - Log to console:

    ```
    console.error('Error deleting
    tax:', error);
    ```

selectTax(tax)
- Assigns a selected tax to the form by cloning its values into `selectedTax`.

createEmptyTax()
- Returns a fresh object with default values to reset the form.

HTML Structure

· Use a top-level `<div>` to wrap the full component UI.

· Add a heading `<h2>`: "Tax Management".

· Add a **Refresh Taxes** button:

- Click triggers `loadTaxes()` to re-fetch data.

· Display all taxes using an `<ul>` list:

- Loop using `*ngFor` over `taxes`
- For each item, show:
  - · `formType`
  - · `filingDate` (formatted using Angular `date` pipe)
  - · `totalTaxAmount`

- Include 3 action buttons:
  - · "Select" → sets the selected tax in the form
  - · "Update" → fills form with selected tax for editing
  - · "Delete" → removes the tax record

· Below the list, add a form with:
- Heading `<h3>`:
  - · "Add Tax" if `taxFormId` is 0
  - · "Update Tax" if editing an existing one

- Form Fields:
  - · **Form Type** – text input
  - · **Filing Date** – date input
  - · **Total Tax Amount** – number input
  - · **User ID** – number input

- Submit Button:
  - · Label: "Add Tax" or "Update Tax" based on context
  - · Click calls `addTax()` if `taxFormId` is 0, else `updateTaxApi()`

TaxService

Purpose
- Provides reusable HTTP methods to manage tax data from the backend.
- Communicates with API at:
  `http://127.0.0.1:8081/taxmanagement/api/taxes`

Functions & Responsibilities

getAllTaxes()
- Sends a `GET` request to fetch all tax entries.
- Returns an observable of `Tax[]`.

getTaxById(id)

- Sends a GET request to fetch one tax by ID.
- Returns an observable of a single Tax.

createTax(tax)

- Sends a POST request with a Tax object to create a new record.
- Returns the created object.

updateTax(id, tax)

- Sends a PUT request to update an existing tax record.
- Takes the id and updated Tax object.
- Returns: the updated tax

deleteTax(id)

- Sends a DELETE request to remove a tax record by ID.
- Returns: void

Model: Tax

Structure

Field	Type	Description
taxFormId	number	Unique identifier
formType	string	Type of tax form
filingDate	Date	Filing date of the form
totalTaxAmount	number	Amount paid in taxes
userId	number	Associated user's ID

		Dynamic Behavior
		- On load: loadTaxes() fetches and displays all tax data. - Form resets after every successful **Add**, **Update**, or **Delete** - On **Add**: - New entry is sent to the backend. - UI refreshes with updated data. - On **Update**: - Form is prefilled with selected tax. - PUT request sent on submit. - On **Delete**: - Deletes record and refreshes UI. - The form dynamically switches between "Add" and "Update" mode based on whether taxFormId is 0 or not. **** Kindly refer to the screenshots for any clarifications. ****

EXECUTION STEPS TO FOLLOW FOR BACKEND

1. All actions like build, compile, running application, running test cases will be through Command Terminal.
2. To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.
3. cd into your backend project folder
4. To build your project use command:
    mvn clean package -Dmaven.test.skip
5. To launch your application, move into the target folder (**cd target**). Run the following command to run the application:
    java -jar <your application jar file name>
6. This editor Auto Saves the code.

7. These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

8. To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN.

9. To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.

10. Default credentials for MySQL:
    a. Username: root
    b. Password: pass@word1

11. To login to mysql instance: Open new terminal and use following command:
    a. sudo systemctl enable mysql
    b. sudo systemctl start mysql

       NOTE: After typing any of the above commands you might encounter any warnings.

       >> Please note that this warning is expected and can be disregarded. Proceed to the next step.

    c. mysql -u root -p
       The last command will ask for password which is 'pass@word1'

12. Mandatory: Before final submission run the following command:
       mvn test

EXECUTION STEPS TO FOLLOW FOR FRONTEND

1. All actions like build, compile, running application, running test cases will be through Command Terminal.

2. To open the command terminal the test takers, need to go to

   Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.

3. This is a web-based application, to run the application on a browser, use the internal browser in the environment.

4. You can follow series of command to setup Angular environment once you are in your project-name folder:

   a. npm install -> Will install all dependencies -> takes 10 to 15 min

   b. npm run start -> To compile and deploy the project in browser. You can press <Ctrl> key while clicking on localhost:4200 to open project in browser -> takes 2 to 3 min

   c. npm run test -> to run all test cases. **It is mandatory to run this command before submission of workspace ->** takes 5 to 6 min