System Requirements Specification Index

For

Debt Management App

Version 1.0

IIHT Pvt. Ltd.
IIHT Ltd, No: 15, 2nd Floor, Sri Lakshmi Complex, Off MG Road, Near SBI LHO,
Bangalore, Karnataka – 560001, India
fullstack@iiht.com

Debt Management
System Requirements Specification

1. Business-Requirement:

1.1 Problem Statement:

Debt Management Application is .Net Core web API 3.1 application integrated with MS SQL Server, where it refers to foundation for developing a comprehensive Debt Management System with CRUD operations. It outlines the key Debts, requirements, and expectations from the system. Developers can use this problem statement as a guide to creating a solution that meets the specific needs of Finance Corporation.
H

1.2 Following is the requirement specification:

	Debt Management
Modules	
1	Debt
Debt Module Functionalities	
1	Create a Debt
2	Update the existing Debt
3	Get a Debt by Id
6	Fetch all Debts
7	Delete an existing Debts

2. Assumptions, Dependencies, Risks / Constraints

2.1 Debt Constraints:

- While deleting the Debt, if Debt Id does not exist then the operation should throw a custom exception.
- While fetching the Debt details by id, if Debt id does not exist then the operation should throw a custom exception.

2.2 Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception if data is invalid
- All the business validations must be implemented in model 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

3.1 Debt Class Entities

- Debt Id (Int) Not null, Key attribute.
- Debt Number (string) Not null.
- Debt Name (string) is not null, min 3 and max 100 characters.
- Premium Amount (decimal) is not null.
- Start Date (date)
- End Date(date)
- Customer Id(Int)

4. Considerations

- There is no roles in this application
- You can perform the following possible actions

Debt

5. Rest Endpoints

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

5.1 DebtController

<table>
<tr><th>URL Exposed</th><th>Purpose</th></tr>
<tr><td>/debt
<table><tr><td>Http Method</td><td>POST</td></tr><tr><td>Parameter 1</td><td>Debt model</td></tr><tr><td>Return</td><td>HTTP Response StatusCode</td></tr></table></td><td>Create Debt</td></tr>
<tr><td>/debt
<table><tr><td>Http Method</td><td>PUT</td></tr><tr><td>Parameter 1</td><td>Long Id</td></tr><tr><td>Parameter 2</td><td>DebtViewModel model</td></tr><tr><td>Return</td><td>HTTP Response StatusCode</td></tr></table></td><td>Update a Debt</td></tr>
<tr><td>/debts
<table><tr><td>Http Method</td><td>GET</td></tr><tr><td>Parameter 1</td><td>-</td></tr><tr><td>Return</td><td><IEnumerable<Debt >></td></tr></table></td><td>Fetches the list of all Debts</td></tr>
<tr><td>/debt?id={id}
<table><tr><td>Http Method</td><td>GET</td></tr><tr><td>Parameter 1</td><td>Long (id)</td></tr><tr><td>Return</td><td><Debt></td></tr></table></td><td>Fetches the details of a Debt</td></tr>
<tr><td>/debt?id={id}
<table><tr><td>Http Method</td><td>DELETE</td></tr><tr><td>Parameter 1</td><td>Long (id)</td></tr><tr><td>Return</td><td>HTTP Response StatusCode</td></tr></table></td><td>Delete a Debt</td></tr>
</table>

6. Template Code Structure

6.1 Package: DebtManagement

Resources

Names	Resource	Remarks	Status
Package Structure			
controller	DebtController	Controller class to expose all rest-endpoints for debt related activities.	Partially implemented
Startup.cs	Startup CS file	Contain all Services settings and SQL server Configuration.	Already Implemented
Properties	launchSettings.json file	All URL Setting for API	Already Implemented
	appsettings.json	Contain connection string for database	Already Implemented

6.2 Package: DebtManagement.BusinessLayer

Resources

Names	Resource	Remarks	Status
Package Structure			
Interface	IDebtServices interface	Inside all these interface files contains all business validation logic functions.	Already implemented

Service	DebtServices CS file	Using this all class we are calling the Repository method and use it in the program and on the controller.	Partially implemented
Repository	IDebt Repository Debt Repository (CS files and interfaces)	All these interfaces and class files contain all CRUD operation code for the database. Need to provide implementation for service related functionalities	Partially implemented
ViewModels	Debt ViewModel	Contain all view Domain entities for show and bind data. All the business validations must be implemented.	Partially implemented

6.3 Package: DebtManagement.DataLayer

Resources

Names	Resource	Remarks	Status
Package Structure			
DataLayer	DebtDBContext cs file	All database Connection,collection setting class	Already Implemented

6.4 Package: DebtManagement.Entities

Resources

Names	Resource	Remarks	Status
Package Structure			
Entities	Debt ,Response (CS files)	All Entities/Domain attribute are used for pass the data in controller and status entity to return response Annotate this class with proper annotation to declare it as an entity class with **Id** as primary key. Generate the **Id** using the **IDENTITY** strategy	Partially implemented

7. Method Descriptions

1. DebtService: Method Descriptions

Method	Task	Implementation Details
`CreateDebt`	To implement logic for creating a new debt record.	- Accept a Debt object as input - Call _DebtRepository.CreateDebt(Debt) - Return the result from repository
`DeleteDebtById`	To implement logic to delete a debt record by ID.	- Accept an integer ID as input - Call _DebtRepository.DeleteDebtById(id) - Return true if deletion succeeds
`GetAllDebts`	To fetch all debt records from the database.	- Call _DebtRepository.GetAllDebts() - Return the list of debt records

GetDebtById	To retrieve a specific debt record using its ID.	- Accept an integer ID as input - Call _DebtRepository.GetDebtById(id) - Return the corresponding Debt object
UpdateDebt	To update an existing debt record using a view model.	- Accept DebtViewModel object as input - Call _DebtRepository.UpdateDebt(model) - Return the updated Debt object

2. DebtRepository: Method Descriptions

Method	Task	Implementation Details
CreateDebt	To implement logic for creating a new debt record.	- Use try-catch block - In try: Use _dbContext.Debts.AddAsync(debt) to add the new record - Call SaveChangesAsync() to save it - Return the created Debt object - In catch: throw the caught exception
DeleteDebtById	To implement logic to delete a debt record by ID.	- Use try-catch block - In try: Use LINQ to find the record using debtId - Call _dbContext.Remove() to delete it - Call SaveChanges() to persist deletion - Return true if successful - In catch: throw the caught exception
GetAllDebts	To implement logic to fetch the latest 10 debt records.	- Use try-catch block - In try: Use _dbContext.Debts.OrderByDescending(x => x.debtId).Take(10).ToList() - Return the list of debts - In catch: throw the caught exception
GetDebtById	To implement logic to fetch a debt record by ID.	- Use try-catch block - In try: Use _dbContext.Debts.FindAsync(id) to get the debt - Return the found Debt object - In catch: throw the caught exception

UpdateDebt	To implement logic to update an existing debt record.	- Use try-catch block - In try: • Use _dbContext.Debts.FindAsync(model.debtId) to fetch the record • Update necessary fields in the fetched entity • Use _dbContext.Debts.Update(entity) • Call SaveChangesAsync() to persist changes - Return the updated Debt object - In catch: throw the caught exception

3. DebtController: Method Descriptions

- **Make sure that you add correct dependencies in the controller before working on below method logics.**

Method	Task	Implementation Details
CreateDebt	To implement logic to create a new debt record with validation.	- Request type: POST, URL: /debt - Accept [FromBody] Debt model - Call _DebtService.GetDebtById(model.debtId) to check if debt already exists - If exists, return StatusCode 500 with message: 'Debt already exists!' - Else, call _DebtService.CreateDebt(model) - If result is null, return StatusCode 500 with message: 'Debt creation failed! Please check details and try again.' - Return Ok with message: 'Debt created successfully!'
UpdateDebt	To implement logic to update an existing debt record.	- Request type: PUT, URL: /debt - Accept [FromBody] DebtViewModel model - Call _DebtService.UpdateDebt(model) - If result is null, return StatusCode 500 with message: 'Debt With Id = {model.debtId} cannot be found' - Else, return Ok with message: 'Debt updated successfully!'
DeleteDebt	To implement logic to delete a debt record by ID.	- Request type: DELETE, URL: /debt?id={id} - Accept id as query parameter - Call _DebtService.GetDebtById(id) to verify existence - If not found, return StatusCode 500 with message: 'Debt With Id = {id} cannot be found' - Else, call _DebtService.DeleteDebtById(id) - Return Ok with message: 'Debt deleted successfully!'

`GetDebtById`	To fetch a specific debt record using its ID.	- Request type: GET, URL: /debt?id={id} - Accept id as query parameter - Call _DebtService.GetDebtById(id) - If not found, return StatusCode 500 with message: 'Debt With Id = {id} cannot be found' - Else, return Ok with the debt object
`GetAllDebts`	To implement logic to fetch all debt records.	- Request type: GET, URL: /debts - Call _DebtService.GetAllDebts() - Return the list of all debts

8. Execution Steps to Follow

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. **On command prompt, cd into your project folder (cd <Your-Project-folder>).**

4. **To connect SQL server from terminal:**
   (DebtManagement /sqlcmd -S localhost -U sa -P pass@word1)
   - **To create database from terminal -**

     1> Create Database DebtDb
     2> Go

5. **Steps to Apply Migration(Code first approach):**
   - **Press Ctrl+C to get back to command prompt**
   - **Run following command to apply migration-**
     (DebtManagement /dotnet-ef database update)

6. **To check whether migrations are applied from terminal:**
   (DebtManagement /sqlcmd -S localhost -U sa -P pass@word1)

```
1> Use DebtDb
2> Go
1> Select * From __EFMigrationsHistory
2> Go
```

7. **To build your project use command:**
   (DebtManagement /dotnet build)

8. **To launch your application, Run the following command to run the application:**
   (DebtManagement /dotnet run)

9. **This editor Auto Saves the code.**

10. **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.**

11. **To test web-based applications on a browser, use the internal browser in the workspace. Click on the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.**

    Note: The application will not run in the local browser

12. **To run the test cases in CMD, Run the following command to test the application:**
    (DebtManagement .Tests/dotnet test --logger "console;verbosity=detailed")
    (You can run this command multiple times to identify the test case status,and refactor code to make maximum test cases passed before final submission)

13. **If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B - command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.**

14. **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.**

15. **You need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.**
