Design document for synoptic project

By Pavel Vjalicin

Contents

[Design 3](#_Toc14272772)

[Sequence diagrams 3](#_Toc14272773)

[Use case 1: using REST API with already registered card. 4](#_Toc14272774)

[Use case 2: using REST API with not registered card. 5](#_Toc14272775)

[Project Setup 6](#_Toc14272776)

[Prerequisites 6](#_Toc14272777)

[Project Structure 6](#_Toc14272778)

[Compiling 7](#_Toc14272779)

[Data Structures 7](#_Toc14272780)

[Testing 8](#_Toc14272781)

[REST API Endpoints 8](#_Toc14272782)

[Testing 11](#_Toc14272783)

# Design

The following design was created based on the specifications provided in the SD Project E Membership System v1.2.pdf file.

My task was to design a REST API for managing membership cards and here is what I came up with.

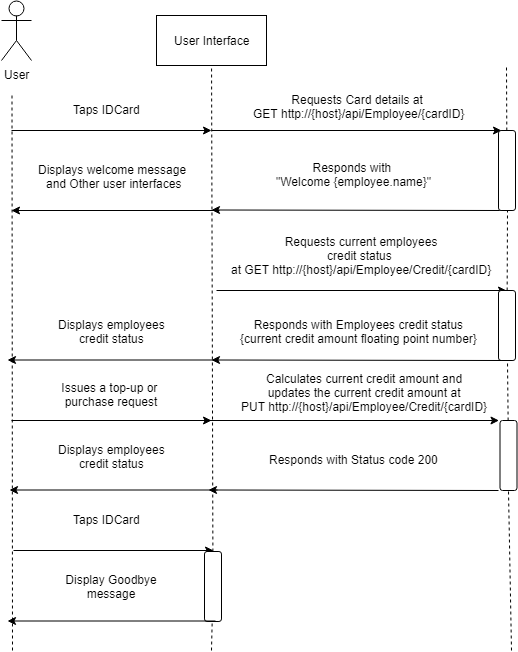
This document provides an abstract non-programming language specific abstractions for API implementation. For more detail about specifics of C# implementation refer to the source code files where comments are used to describe the functionality of the program in a more specific way.

## Sequence diagrams

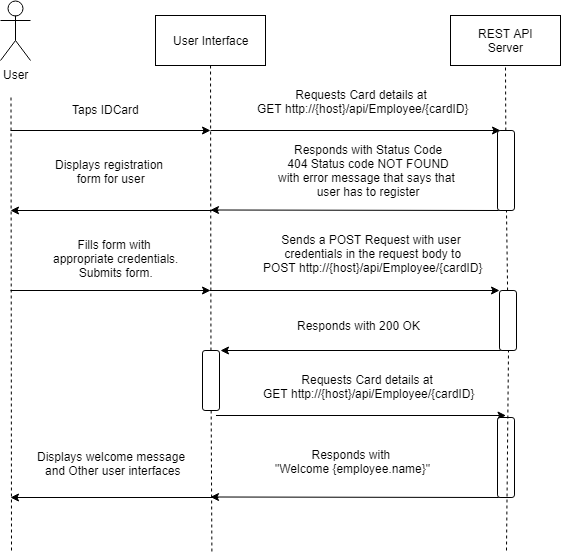
Based on the specifications provided I have created two UML sequence diagrams for different use cases.

Those diagrams display an abstract idea of how the REST API should be interacted with in the context of the whole application.

### Use case 1: using REST API with already registered card.



### Use case 2: using REST API with not registered card.



After registration of a card is finished the application should proceed as usual (as displayed in use case 1).

# Project Setup

The REST API server was made by using ASP.NET Core 2.2 with C#.

This particular REST API server was designed according to design principles of Microsoft.

The source code of the application is provided in the “qa\_synoptic\_project” folder or can be accessed from a GitHub repository: <https://github.com/PavelVjalicin/qa_synoptic_project>

## Prerequisites

Official Microsoft reference for OS support for ASP.NET CORE 2.2: <https://github.com/dotnet/core/blob/master/release-notes/2.2/2.2-supported-os.md>

Appropriate dotnet SDK for .NET Core 2.2 installed on the machine.

Link to SDK: <https://dotnet.microsoft.com/download>

## Project Structure

* ./qa\_synoptic\_project – Main project folder.
  + ./FirstCateringLtd.BackService – Contains REST API Implementation
    - ./FirstCateringLtd.BackService.csproj – REST API configuration file contains dependencies and framework libraries.
    - ./Program.cs – C# project initialisation, main class.
    - ./Startup.cs – ASP.NET Core configuration file. Used to initialise database, implement runtime libraries. Set up ASP.NET Core environments.
    - ./Data – Contains database contexts. Files that the database will use to generate an appropriate database based on the class interface.
    - ./Models – Contains database Model classes.
    - ./Controllers – Contains controllers that are used for REST API.
  + ./FirstCateringLtd.Tests – Contains Tests for REST API Implementation
    - ./FirstCateringLtd.Tests.csproj – Test project C# asp.net core configuration file contains dependencies and framework libraries.
    - ./FunctionalityTests.cs – Contains all of the REST API tests.

## Compiling

This project was not designed to run in production environment.

The specifications did not provide a clear idea of a development environment setup. To build an executable version of the project some project configuration changes will have to be made.

Notes from Microsoft about different environment settings:

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-2.2>

Notes from Microsoft explaining how to compile executable files for different operating systems:

<https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>

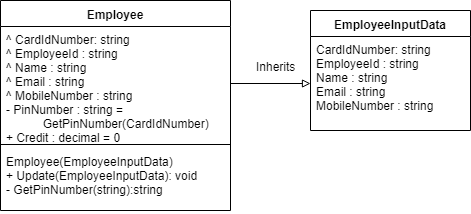
To run the project in development mode issue the following command in the command prompt from “qa\_synoptic\_project/FirstCateringLtd.BackService/” location: **dotnet run**

Here is a video by Microsoft employee on how to use and setup ASP.NET CORE REST API server and more: <https://www.youtube.com/watch?v=--lYHxrsLsc>

After issuing the following command the server will be run on localhost with a random port assigned to it. The address and port of the server will be display in the console.

## Data Structures

The following data structures were created for this project:



C# implementations of those models can be found in “qa\_synoptic\_project/FirstCateringLtd.BackService/Models.Employee.cs”

We use Employee class as our Model that we store in database and use EmployeeInputData to communicate with our database through REST API.

Configured database has to have an appropriate table called Employee with all of the fields provided above. The current implementation has a built-in database that is generated by ASP.NET Core framework. ASP.NET Core has multiple supported database types. In this instance we use SQLite implementation.

The generated database is currently called CateringDatabase.db and is stored in the root folder of the BackService project. The name of database can be changed from the Startup.cs in the FirstCateringLtd.BackService folder.

Currently database migrations are not supported. After changing any of the models to run project successfully the CateringDatabase.db file has to be deleted.

Here is a list of supported database types provided by Microsoft for ASP.NET Core:

<https://docs.microsoft.com/en-us/ef/core/providers/>

## Testing

Testing was done using xUnit tests for C# ASP.NET Core.

Testing project is located in the “/qa\_synoptic\_project/FirstCateringLtd.Tests” folder.

To run automated tests run a command prompt command “dotnet test” from “/qa\_synoptic\_project/FirstCateringLtd.Tests” location.

## REST API Endpoints

This project uses Swagger for REST API Endpoint documentation available at http://{host}/swagger while running the project server.

Beware: Swagger executes the API calls on the actual database changing the database in the process.

Note: Swagger API Documentation should be disabled while running the project in production mode.

The REST API endpoints of this implementation are declared in the /qa\_synoptic\_project/FirstCateringLtd.BackService/Controllers/EmployeeController.cs file.

Documentation for REST API endpoints:

Crucial endpoints:

**GET /api/Employee/{cardId}**

Used to authenticate an employee with a registered card.

Takes employee’s card Identification number.

Returns either a 200 welcome message with employees name if card is registered or a 404 with a message that the user needs to register first.

**POST /api/Employee**

Used to register employee’s card with employee credentials.

Takes an EmployeeInputData Data structure in the request body as JSON.

EmployeeInputData Data Structure is described in the Implementation section of this document.

Returns 200 if successful.

Returns 400 with an error message of what went wrong if the input data was not filled in properly.

**GET /api/Employee/credit/{cardId}**

Used to retrieve employees current balance.

Takes employee’s card ID

Returns 200 with a current balance number.

Returns 404 if card id was not found

**PUT /api/Employee/credit/{cardId}**

Used to change the current balance of an employee.

Takes a double value in request body that is going to replace the current balance of an employee.

Returns 200 if API call went through.

Returns 400 if the request body was not filled in properly

Returns 404 with error message if card id was not found in the database.

Additional optional endpoints:

The specification does not specify if I need to add standard REST API nodes for data deletion, retrieval and updating. I added those to comply with REST API standards and for testing purposes. If those are not needed I advise to remove the following endpoints while deploying to production environment.

**GET /api/Employee**

Returns 200 array of all employees

**PUT /api/Employee**

Used to replace employee with different values.

Takes EmployeeInputData in request body.

Returns 200 if API call went through.

Returns 400 if the request body was not filled in properly

Returns 404 with error message if card id was not found in the database.

**DELETE /api/Employee/{cardId}**

Deletes employee with appropriate card id.

Returns 204 if successful.

Returns 404 if card id was not found

# Testing

To make sure the project works properly I have written multiple automated tests to test that the REST API works properly.

Project tests are located in qa\_synoptic\_project/FirstCateringLtd.Tests/FunctionalTests.cs.

The test are written with the xUnit testing framework provided by Microsoft.

All of those tests are unit tests meaning that they test only one specific thing.

These tests are focused on two things: 1) Making sure that Employee Data Structure is initialised properly based on the EmloyeeInputData Data Structure. 2) REST API endpoints respond appropriately to queries.

One of the requirements for this project was to setup a database to store employee data. Because of that, while testing REST API endpoints I Mock a database with fake data (Temporary in-memory database is created for this purpose), issue appropriate queries and assert the results.

The following tests were created for this project:

Function Name: **EmployeeIsInitialisedProperlyFromEmployeeInputData**

Purpose: Employee should be initialised properly from Employee Input Data:

Description: Employee class has some fields that are not described from input data. PinNumber (is calculated based on the last 4 characters of CardId), Credit (is assigned with a default value of 0). We assert the created employee based on employee input data and check if additional fields were assigned.

Result: **PASS**

Function Name: **CardIdMustOnlyCantainAlphaNumericCharacters**

Purpose: Card Id must only consist of alphanumeric characters

Description: We try to create an employee with incorrect card id (that contains not allowed characters) and expect to see a failed result.

Result: **PASS**

Function Name: **GetIdCardMustReturnNotFoundIfCardIdIsNotRegistered**

Purpose: Test GET Employee by card id NotFound response.

Description: We pass not existing card id to employee get by id endpoint and assert looking for a response that informs us that we have to register the card.

Result: **PASS**

Function Name: **GetIdCardMustReturnWelcomeMessageIfCardIsRegistered**

Purpose: Test GET Employee by card id Ok response.

Description: We assign a new employee to our mock in memory database and test if we get a response of “Welcome {employee.name}”

Result: **PASS**

Function Name: **EmployeeMustBeAbleToCheckCredit**

Purpose: Test if a registered employee can request their current credit details.

Description: We create an employee and issue a get credit call from our REST API controller. We expect to see an OK result with a default value of 0 decimal.

Result: **PASS**

Function Name: **InValidCardIdGetCreditCallShouldReturnNotFound**

Purpose: To check if get credit call returns a not found status when issued with unregistered card id.

Description: We issue get call with invalid card id and expect a not found status with "User with this card id doesn't exist." message.

Result: **PASS**

Function Name: **EmployeeMustBeAbleToUpdateCredit**

Purpose: To check if employee can update his credit amount.

Description: We create employee, issue a put credit call to API expecting Ok results. After we issue a get credit API request and verify the result we get based on the data we used previously.

Result: **PASS**

# Implementation