Design document for synoptic project

By Pavel Vjalicin

Contents

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.

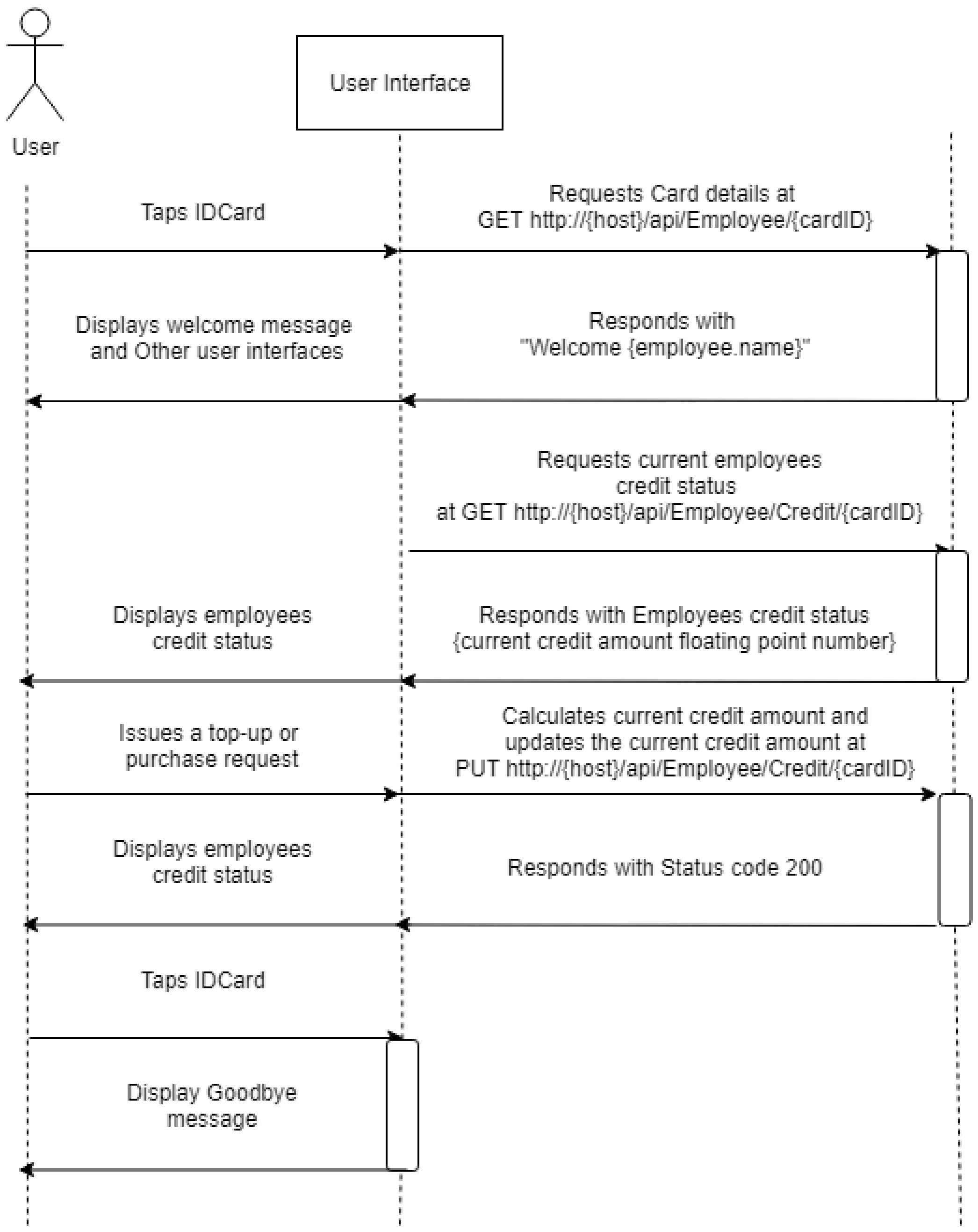
This document provides an abstract non-programming language specific abstractions for API implementation. For more detail about specifics of C# implementation refer to the source code files where comments are used to describe the functionality of the program in a more specific way.

Sequence diagrams

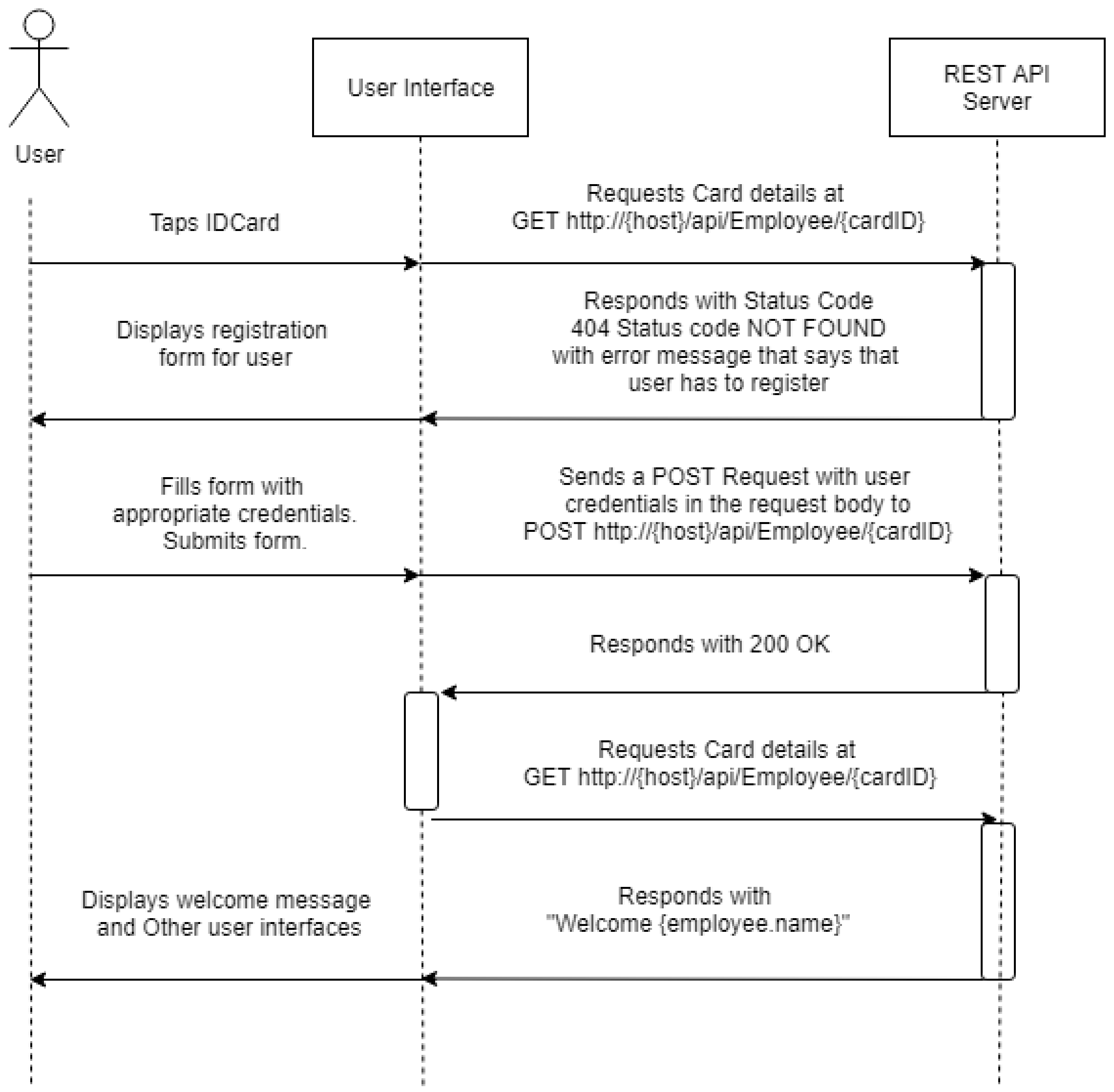
Based on the specifications provided I have created two UML sequence diagrams for different use cases.

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.


User
User Interface
Taps IDCard
Requests Card details at GET http://{host}/api/Employee/{cardID}
Displays welcome message and Other user interfaces
Responds with "Welcome {employee.name}"
Requests current employees credit status at GET http://{host}/api/Employee/Credit/{cardID}
Displays employees credit status
Responds with Employees credit status {current credit amount floating point number}
Issues a top-up or purchase request
Calculates current credit amount and updates the current credit amount at PUT http://{host}/api/Employee/Credit/{cardID}
Displays employees credit status
Responds with Status code 200
Taps IDCard
Display Goodbye message

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


User
User Interface
REST API Server
Taps IDCard
Requests Card details at GET http://{host}/api/Employee/{cardID}
Responds with Status Code 404 Status code NOT FOUND with error message that says that user has to register
Displays registration form for user
Fills form with appropriate credentials. Submits form.
Sends a POST Request with user credentials in the request body to POST http://{host}/api/Employee/{cardID}
Responds with 200 OK
Requests Card details at GET http://{host}/api/Employee/{cardID}
Responds with "Welcome {employee.name}"
Displays welcome message and Other user interfaces

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 with ASP.NET Core 2.1 framework.

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 2.1: https://github.com/dotnet/core/blob/master/release-notes/2.1/2.1-supported-os.md

Appropriate dotnet SDK for .NET Core 2.1 or later version installed on the machine.

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

Project Structure

- ./qa_synoptic_project – Main project folder.
    - ./FirstCateringLtd.BackService –REST API project
        - ./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. ORM context used to generate and work with a built-in database. Does not depend on the database implementation.
        - ./Models – Contains database Model classes.
        - ./Controllers – Contains controllers that are used for REST API.
        - ./Properties – Contains project properties.
    - ./FirstCateringLtd.Tests –Project that tests REST API server
        - ./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-2.1

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.

Testing

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

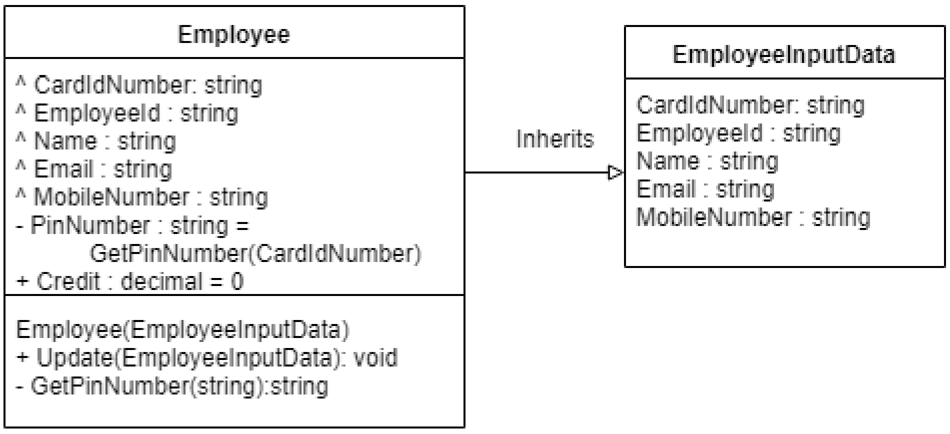
Testing project is located in the “/qa_synoptic_project/FirstCateringLtd.Tests” folder.

To run automated tests run a command prompt command “dotnet test” from “/qa_synoptic_project/FirstCateringLtd.Tests” location.

Test documentation is provided below in the Test Documentation section of this document.

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 uses built-in ORM system to manage database queries. 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/

REST API Endpoints

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

Swagger documentation from Microsoft: https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-2.1

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.

Current REST API is implemented to only work with JSON request body format.

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.

Example request:

```
curl -X GET --header 'Accept: text/plain'
'http://localhost:61174/api/Employee/SomeCardID1234'
```

Example response:

Code: 200

Headers:

```
{
  "date": "Thu, 18 Jul 2019 10:33:33 GMT",
  "server": "Kestrel",
  "transfer-encoding": "chunked",
  "content-type": "text/plain; charset=utf-8"
}
```

Body: Welcome Pavel Vjalicin

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.

Example request:

```
curl -X POST --header 'Content-Type: application/json-patch+json' -d '{ \
   "cardIdNumber": "SomeCardID1234", \
   "email": "email%40email", \
   "employeeId": "employeeIDExample", \
   "mobileNumber": "+1234-1(234) 1234", \
   "name": "Pavel Vjalicin" \
 }' 'http://localhost:61174/api/Employee'
```

Example response:

Code: 200

Headers:

```
{
  "date": "Thu, 18 Jul 2019 10:31:02 GMT",
  "content-length": "0",
  "server": "Kestrel",
  "content-type": null
}
```

Body: no content

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

Example request:

```
curl -X GET --header 'Accept: application/json'
'http://localhost:61174/api/Employee/credit/SomeCardID1234'
```

Example response:

Code: 200

Headers:

```
{
  "date": "Thu, 18 Jul 2019 10:35:27 GMT",
  "server": "Kestrel",
  "transfer-encoding": "chunked",
  "content-type": "application/json; charset=utf-8"
}
```

Body: 0

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.

Example request:

```
curl -X PUT --header 'Content-Type: application/json-patch+json' -d '200.55'
'http://localhost:61174/api/Employee/credit/SomeCardID1234'
```

Example Response:

Code: 200

Headers:

```
{
  "date": "Thu, 18 Jul 2019 10:37:46 GMT",
  "content-length": "0",
  "server": "Kestrel",
  "content-type": null
}
```

Body: no content

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

Example request:

Example Response:

Code: 200

Headers:

Body:

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

Test Documentation

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 tests 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

Database

Current implementation of the REST API uses a built-in database provided by ASP.NET Core. This limits the development flexibility. I advise implementing a standalone remote database.

I also recommend using ORM system for database management. Using ORM systems provides several benefits:

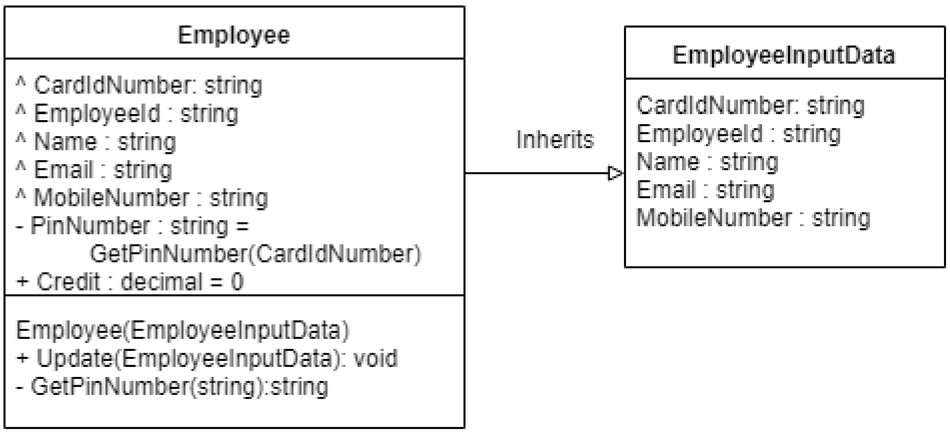
1) Built-in SQL Injection prevention.
2) Database creation, migration and queries are based on Classes rather than direct SQL queries. This improves database system maintenance and provides with easier way to migrate to a different database system if needed.

Current implementation uses ASP.NET Core ORM.

ASP.NET Core ORM documentation from Microsoft: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/work-with-data-in-asp-net-core-apps

ORM option for java: https://ebean.io/

Here are the steps for setting up a database:



EmployeeInputData is only used for data transit purposes. Should not be stored in the database or have a table assigned to it.

Implement Employee table as described above with proper fields.

Model Field Descriptions

CardIdNumber:

1) Should be unique.

2) Should be at least 4 characters long. (Specification does not provide with the format required for the employee card ID so I assumed the minimal length of id to create a pin code)
3) Should only allow alphanumeric characters. I validate this with the following regex expression: “`^[a-zA-Z0-9]*$`”.
4) Should be used as a table ID
5) Should be stored as an encrypted hash number for security reasons. (I did not implement data encryption to simplify future implementations. The way you encrypt data would depend on the development environment and requirements)

EmployeeId: Should be required.

Name:

1) Should be required.
2) (Optional) Can be split up into first name, last name, middle name columns based on the requirements.

Email:

1) Should be required.
2) (Optional) Should be validated for proper email format. (I opted out of email validation. Email validation can be tricky to implement and depend on the specifications. Bad email validation can lead to false positives that drastically worsen user experience)

MobileNumber:

1) Should be required.
2) (Optional) Should be validated. (Mobile phone validation is quite tricky to implement. Mobile number can have multiple formats so I generally recommend to stick to just storing a string value without any validation)

PinNumber:

1) Should be required.
2) Should have a length of exactly 4 characters
3) Should be generated from card Id by taking 4 last characters of card Id.
4) Should be stored as an encrypted hash number for security reasons. (I did not implement data encryption to simplify future implementations. The way you encrypt data would depend on the development environment and requirements)
5) (Note) The specification does not provide a use case for PinNumber. Included only for purposes of further development.

Credit:

1) Should be required.
2) Default value should be 0.
3) Should be limited in the amount of decimal point’s stored based on requirements. (I opted out of this because of lack requirements)
4) (Optional) Should have a currency assigned to the value if specification requires. (My implementation only support one arbitrary currency type that should be managed by the client side)

5) `Should be stored as an encrypted hash number for security reasons. (I did not implement data encryption to simplify future implementations. The way you encrypt data would depend on the development environment and requirements)`

Implement following functions to simplify the work with Employee Model:

Employee(EmployeeInputData)

Employee constructor that construct Employee object from EmployeeInputData.

Update(EmployeeInputData):void

Employee function that changes the state of an employee based on EmployeeInputData.

GetPinNumber(string):string

Employee function that takes a cardId field as a parameter and returns a valid pincode.

Pin code is generated based on 4 last characters of the card ID.

C# implementation for the above is available at:
qa_synoptic_project/FirstCateringLtd.BackService/Models/Employee.cs

Create Table SQL query example

Example SQL query to set up an Employee table:

Should be executed on the appropriate database.

```sql
CREATE TABLE Employee (
   CardIdNumber VARCHAR (255) NOT NULL UNIQUE,
   EmployeeId VARCHAR (255) NOT NULL,
   Name VARCHAR (255) NOT NULL,
   Email VARCHAR (255) NOT NULL,
   MobileNumber VARCHAR (255) NOT NULL,
   PinNumber VARCHAR(4) NOT NULL
);
```

REST API Endpoints

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

Swagger documentation from Microsoft: https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-2.1

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.

Additional documentation for REST API endpoints is available above at Project Setup – REST API Endpoints section.

For business logic and Endpoint implementations refer to /qa_synoptic_project/FirstCateringLtd.BackService/Controllers/EmployeeController.cs /qa_synoptic_project/FirstCateringLtd.BackService/Models /Employee.cs

Security

Specifications did not provide enough information about the production environment, because of that some of the security features were left out in this implementation.

Client Authentication

Currently client side authentications is not implemented. This might cause security vulnerabilities if the REST API can be accessed from the global internet.

To prevent unauthorised access client authentication should be added. My recommendation would be adding a security header to every http request that is then validated by the server.

Here is OWASP documentation for good REST API security practices: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Those MUST be implemented if the REST API server needs to be accessed outside of local area network.

XSS Database Storage Vulnerability Prevention

There is a potential risk of attacker storing client side executable code on the database. This measure should be prevented through client-side input sanitization. All user inputs and outputs should be interpreted as strings and not executable code.

Example:

Post request: curl -X POST --header 'Content-Type: application/json-patch+json' -d '{ \

```
   "cardIdNumber": "SomeCardID1234", \
   "email": "email%40email", \
   "employeeId": "employeeIDExample", \
   "mobileNumber": "+1234-1(234) 1234", \
   "name": "<script>alert(“Injection”)</script>" \
 }' 'http://localhost:61174/api/Employee'
```

In this case GET api/Employee/SomeCardID1234 request would return “Welcome <script>alert(“Injection”)</script>” which would execute a javascript tag in the browser. This can be used to steal user information without user’s knowledge.

OWASP Documentation on preventing XSS:
https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

SQL Injection

Current implementation prevents SQL Injections by using ORM system that automatically sanitises database queries.

If the database is going to be remade with direct SQL queries the queries need to be sanitised to prevent attacker from having full access of the database.

OWASP Documentation to prevent SQL Injections:
https://www.owasp.org/index.php/SQL_Injection

Logging

Because this project is dealing with sensitive financial transactions request logging is necessary. Specifications mention that all transactions are going to be verified by IT department.

Current implementation doesn’t include logging features because of lack of logging formatting specifications.

Documentation on logging for ASP.NET Core:

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-2.1

Encryption

I advise some if not all of the data that is stored on the database to be encrypted. You do not want to store raw data on the database in case an attacker gets access to your database. Instead you want to store data as encrypted hashes that you decrypt during runtime.

The fields I highly recommend to encrypt: card id number, credit amount, pin number.

Current implementation doesn't include data encryption, because the way you encrypt and store data will depend on the production environment and business requirements.

Here is OWASP documentation about encryption practices: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

Financial Transactions

Current implementation credit amount relies on client-side credit calculations. Financial data should be calculated on a server for security reasons. More detailed specifications about financial transactions are needed to develop an appropriate transaction system.

I would not recommend using current credit transaction system and instead would rewrite the system completely for production environment. Proper transaction system is way more advanced and would require a lot more time to implement.