# FEEDBACK MANAGER V1

REST API FOR MANAGING USERS FEEDBACKS

VLADO KRAGUJEVSKI, JANUARY 2018

# CONTENTS

# SCOPE

The scope of the document is to provide a short overview of the Feedback Manager in terms of architecture, design and technologies. It is intended to be used by Developers to get a basic knowledge point about the project.

# TECHNOLODGIES

Table 1 provides the technologies used for developing the Feedback Manager.

| | |
|---|---|
| Operating System | Windows 10 64bit |
| Developing IDE | Visual Studio 2017 Community Edition |
| Programming Framework | .NET 4.6.1 |
| Programming Language | C# 6.0 |
| REST Framework | Microsoft Web Api 2 |
| Database | Microsoft SQL Server 2012 – Express Edition |
| Data Access Technology | Microsoft LINQ to SQL |
| Source Control | Git |
| Source Repository | https://github.com/vladokr/FeedbackManager |
| Unit Test Framework | NUnit 3.9 |
| Unit Test Runner | Nunit 3 Test Adapter – VS 2017 extension |
| Isolation (mocking) Framework | NSubstitue 3.1 |
| IoC container | Unity 5.5.5 |
| Mapper DAL to Model objects | AutoMapper 6.2.2 |
| Coding  Spell Checker | VS 2017 Spell Checker - extension |
| | |

TABLE 1 TECHNOLGIES

# ARCHITECTURE

The application currently has only two tires. One is the web application, the REST service running in IIS, and the other one is the database running in SQL Server. From logical point of view the web application is divided in three layers: Web Service Layer, Business Services Layer, and Data Access Layer.

The Web Service Layer provides the REST services for managing the Feedbacks. It represents the frontend of the application. It is the layer that receives the requests and provides response to the users.

The Business Layer seats between the Web Service Layer and the Data Access Layer. It is the layer that should contain the business logic. Since the current version of Feedback Manager doesn't contain any sort of business logic, the Business Layer can be easily removed; thus simplifying the logical design of the application. Its only current function is to receive calls from the Web Service Layer and to transfer them to the Data Access Layer.

The Data Access Layer is in charge of reading and writing the data from and to the Data Storage.

# DESIGN

This section provides overview of the database design and the application design.

.

# DATABASE DESIGN

The Picture 1 displays the Database design diagram.

The database contains 5 tables:

- FM_User – stores the registered users
- FM_Game – represents an enumeration of all games in the system
- FM_Game_Session – represents a played session of some game in the time
- FM_Feedback – stores user feedbacks for a given game session
- FM_Role – represents an enumeration of all user roles (PLAYERS and OPERATORS)

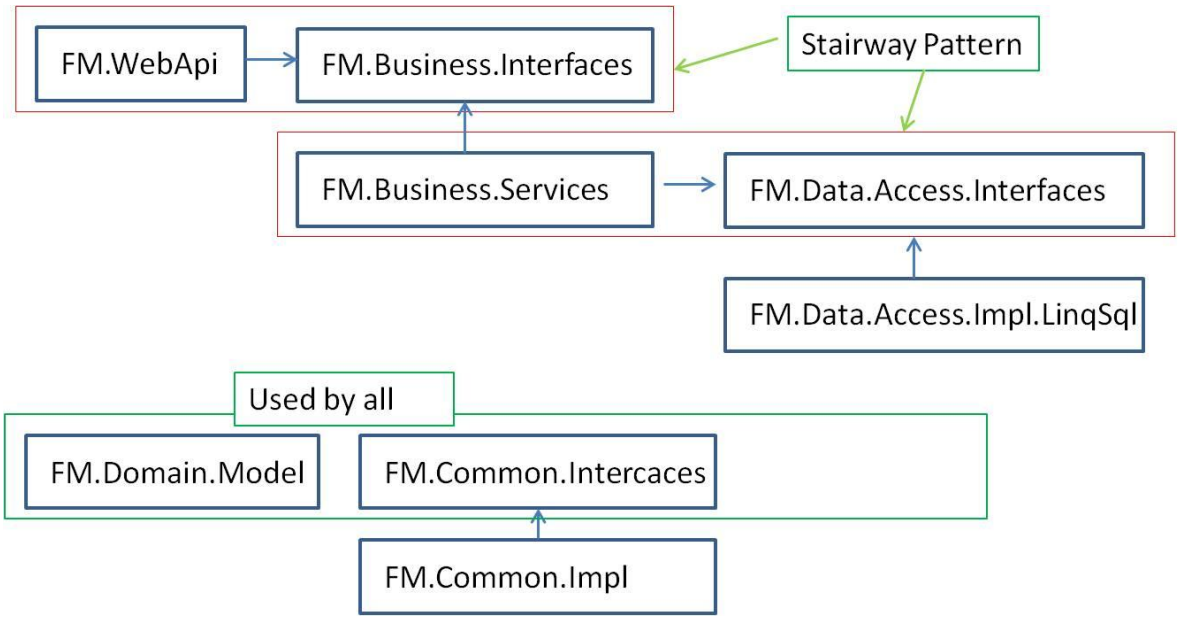
The following is common for every table:

- id – as a primary key column
- create_date – the creation time of a tuple; set by the database
- update_date – the last modification of a tuple; set by the application
- PK_{something} – the format of all primary keys names
- FK_{something} – the format of all foreign keys names
- UQ_{something}- the format of all unique constraint names


PICTURE 1 DATABASE DIAGRAM

## APPLICATION DESIGN

The three layered architecture is designed following the SOLID principles and the Stairway pattern for componentization. Picture 2 shows the Module diagram of the application. The idea behind the Stairway pattern is to keep the communication between the layers hidden behind abstractions. As we can see on Picture 2 the WebApi communicates with the Business Layer using the Business Interfaces. The Business Services then implements the Business Interfaces, and it stays one "stair" bellow. The same is valid for the other two Data Access modules. The Domain Model and Common Interfaces are shared modules between all the layers.




PICTURE 2 MODULES OVERVIEW

For every logical layer there are at least two modules (DLLs). One is the abstract module, or the interface, and the other one is the implementation module, or the concrete module. For example, on picture 2, for the DataAccess layer, there is one abstract layer DataAccessInterfaces, and one concrete layer DataAccessLinqSql. In the future we may need to add more concrete layers to implement interactions with other database storage systems; LINQ to SQL supports only Microsoft SQL Server.

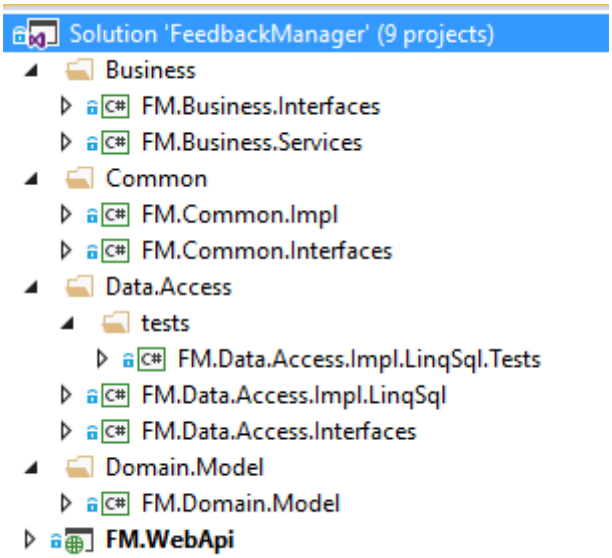
In every class the dependencies are plugged as constructor parameters, and they are resolved in runtime by the Unity container. The container is configured in the highest application layer i.e. WebApi. In this way by just changing the configuration we can switch the Feedback Manager to use another DataAccess implementation.

The DataAccess layer is implemented using the LINQ to SQL technology. Visual Studio has an object relational mapper (ORM) which generates the Data Access Layer (DAL). This layer contains objects (entities) for every database object. Since these classes are strictly depended on the LINQ to SQL we cannot use them throughout the application. That's why we have a shared project, i.e DomainModel, that contains the Feedback Manager entities like User, Game,

GameSession etc. The DataAccessLinqSql module must convert the DomainModel objects to DAL objects and vice versa. To ease this conversion we use a library called AutoMapper.

The Common Interfaces module is intended to be shared among all application layers. In our case it is used only for sharing an abstraction for a logger class. This way the rest of the application layers doesn't need to have a dependency to the concrete logger implementation; in our case we use an external library, i.e. log4net.

## PROJECTS




PICTURE 3 PROJECTS OVERVIEW

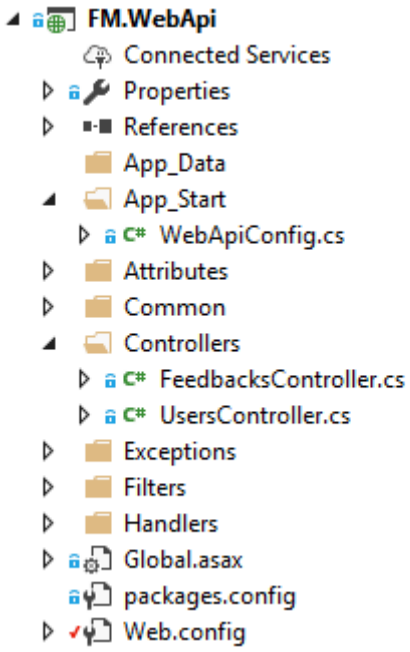
Picture 3 shows the projects inside Visual Studio Solution Explorer. They are grouped inside solution folders to better split the logical layers.

## WEB API

This module contains the REST service, and represents the highest application layer of Feedback Manager. Picture 4 displays WebApi in Solution Explorer.

Next we will provide overview of the most important design features.

**Authentication**
Every REST request first passes through an authentication handler searching for authentication data. The authentication data is composed of application key, username and password. The application key is stored in web config, and the username and password are checked against the database. The handler first checks the application key, because this operation is much faster, and then checks the username and password. If both checks are successful the user is authenticated and its principal is set to the current thread, otherwise a 401 Unauthorized is returned to the caller.




PICTURE 4 WEB API PROJECT

**Authorization**

To allow only to the PLAYERS to create feedbacks and only to the OPERATORS to retrieve feedbacks we use an authorization attribute. The authorization attribute is implemented as a filter. The request passes throughout the filter much after the authentication handler. Thus, if the filter is reached we know the user is authenticated and we only need to check for authorization. If the user is not authorized we return 403 Forbidden.

**Logging**

For every application layer we have one group of exceptions, i.e. IDataAccessException, IBusinessException and RestApiException. This allows performing different logging actions. The lower layers bubbles the exceptions to the WebApi layer where they get logged by the Exception Filter. Every handled exception (exceptions coming from lower layer like IDataAccessException) and every not handled exception (unpredicted exceptions thrown by the framework) are caught by the Exception Filter. The details of the exception are logged like StackTrace and InnerException, and a shorter message is returned to the caller.

# TODO TASKS

## TESTS

To further improve the design and the quality of the application it is necessary to include, where needed, unit tests and integration tests. Currently a full coverage of unit tests is provided only to the Mppers between the Model object and the DAL objects.

## ERROR MESSAGES

The error messages and error codes must be standardized, and taken apart for easier localization. There should be a separate module that keeps all the messages localized.

## REST ENDPOINT COMPLETION

The REST endpoints must be completed to cover all possible operations.

## CODE DOCUMENTATION

The code is missing documentation based on XML Documentation Comments. A SandCastle[1] based documentation then needs to be generated from the XML documentation.

## SWAGGER UI

A Swagger UI should be set to better expose the REST web service.

[1] A tool for generating documentation from the XML comments