**Michtava – A Web based Learning Management System**

**Maintenance Manual**

**Magal Weiss**

**Shay Dayan**

**Roi Pillar**

Index

Introduction…………………………………………………………………..…...**3-4**

Installation ………………………………………………………………………....**4-5**

Maintenance Information……………………………………………….….**6-11**

Server-side…………………………………………………………….…….**7-9**

Client-side ……………………………………………………….………..**9-11**

Adding features……………………………………………………………..…**11-14**

Useful commands and troubleshooting ……………………………**14-16**

1..Introduction

The main technologies on which Michtava relies upon all live in the .Net Framework 4 stack; C#7, Asp.Net MVC 5 for front-end, and Entity Framework 6 for persistent data access.

Because of its hefty reliance upon the .Net stack, Michtava will not run on a Linux server in its current form.

The following is a list of frameworks/libraries which are all essential knowledge in maintaining Michtava:

*Client Side -*

Asp.Net – MVC5, Identity, Owin, Razor syntax,

Ninject Dependency Injector,

AutoMapper – a Convention based Model to Model converter.

jQuery – for Forms validation mostly.

*Server Side –*

Entity Framework 6,

IIS 8-10 (either is fine)

AspNet.Identity.EntityFramework – the relationship between an Owin User and the persistent data attributed to him through EF.

2..Installation

1. Download the following:

* Visual Studio 2012 or newer
* IIS 8 or newer
* .Net Core 4.xx
* Sql Server 2012 or newer, with LocalDB
* A Git client of your choice

1. Clone repository at <https://quwala@bitbucket.org/michtava/michtava.git>
2. To create the SQL Server instance:

* Open CMD
* Type in:

SqlLocalDb create <your\_instance\_name>

And then

* SqlLocalDb start <your\_instance\_name>
* In MichtavaSol(solution)/Frontend/Web.Config,

Change the connection string as follows:

<connectionStrings>

<add name="*<your\_connection\_name>*" connectionString="Data Source=(LocalDb)\*<your\_instance\_name*>;AttachDbFilename=|DataDirectory*|*

*\<your\_db\_name>*;Integrated Security=True" providerName="System.Data.SqlClient"/>

</connectionStrings>

1. To run the web server, open IIS and follow these instructions: <https://www.pluralsight.com/blog/tutorials/windows-server-2008-iis7>

3..Maintenance Information

The Projects' roles within the MichtavaSol.sln solution are as follows:

* Frontend – the entirety of the Michtava client.
* Entities – the logical entities of Michtava – User data models, and models for all other entities such as texts, homeworks, and so forth.
* Dal – (Data Access Layer) – table management through EF, includes repositories for all logical entities, utilizes a lot of generics and inheritance.
* Dal\_Tests – Unit and Integration tests for the Dal project
* Services – the main API through which the client and the server communicate; this project contains services for all "Big" entities in Michtava, through which the client can send asynchronous commands to the server.
* Services\_Tests – Unit, Integration, and System tests for the Services project.
* All other projects' purpose is self-explanatory.

3.1..Server-side

Database creation flow

Upon startup, in the Frontend/Startup.auth.cs file, a new ApplicationDbContext object is instantiated, this object can be seen as the wrapper object for our database (for more info, visit <http://www.entityframeworktutorial.net/EntityFramework4.3/dbcontext-vs-objectcontext.aspx>).

Once the DbContext is created, initial data is seeded to it through the Dal/migrations/configuration.cs file (using DatabaseSeeder, more on that in the following pages. Any and other configuration settings concerning the DbContext can be set in the Configuration class constructor (automatic migrations, automatic data loss upon update, etc).

ApplicationDbContext is not accessed directly by any component but the Dal/Repositories/GenericRepository.cs file, (this is achieved through inheritance), which is the "closest" our code ever gets to the actual data (excluding manual SQL queries in special cases).

Database querying flow

Querying the database is modeled through the Repository design pattern (https://code.tutsplus.com/tutorials/the-repository-design-pattern--net-35804).

All repositories in Dal are conceptually a heap of queries for their respective entity.

In the same manner, all services in Services (project) are a heap of queries for their respective entities, using any number of repositories. The services are the interface through which the client speaks with the server.

Data Seeding and migrations

Initial Data is seeded in to the Database through the call to Dal/DatabaseSeeder.cs Seed() method.

After changing the seed data, it is recommended to create a new DB file (more in Chapter 5).

As an intro, it is recommended to read up on Automatic Migrations in Code First EF.

In its default form, Michtava will automatically migrate upon detecting the need for DB schema changes, and will delete existing data in order to do so (more in Chapter 4 – adding features).

Dependency Injection

All throughout the solution, Ninject dependency injections are used to slave Interfaces to Actual implementations – for example, whenever a developer is writing a service, all they need in order to get their required repositories, is simply add the appropriate Interface to the ctor of the service, and Ninject will check the implementation bound to that interface, and inject it. (examples in Services/TextService). The Configuration file location for Ninject is Frontend/Infra/NinjectDependencyResolver.cs

Tests

All Tests can be found in Services\_Tests and Dal\_Tests projects.

The tests are all written using Effort.EF7, a testing framework that utilized NMemory for in-memory database creation and manipulation, with NUnit as the testing framework. The test files all provide good templates for new test creation.

3.2..Client-side

Dependency Injection

All throughout the solution, Ninject dependency injections are used to slave Interfaces to Actual implementations – for example, whenever a developer is writing a controller (MVC) in the client, all they need in order to get their required services, is simply add the appropriate Interface to the ctor of the controller, and Ninject will check the implementation bound to that interface, and inject it. (examples in Frontend/Areas/Administration/Controllers).

The Configuration file location for Ninject is Frontend/Infra/NinjectDependencyResolver.cs.

MVC

The client is implemented mostly using MVC, with ASP.NET's nice scaffolding components.

The Student and Teacher MVC can be found in Frontend/Controllers, Frontend/Models, and Frontend/Views/Students or Frontend/Views/Teachers respectively.

The Administrator MVC can be found in Frontend/Areas/Administration.

Routing Info

The routing info for students and teachers can be found in Frontend/App\_Start/RouteConfig.cs.

The routing info for admins can be found in Frontend/Areas/Administration/AdministrationAreaRegistration.cs.

Form validation

In the Administration area, form validation is done via data annotations in the ViewModels (required, regular expressions, etc).

In the Teachers/ Students side, validation is done with jQuery.

AutoMapper

In Certain Controller actions, the mapping between ViewModels (client) and Entity Models (back-end) is done explicitely; in other actions it is done using the AutoMapper library, an object-object converter that is suited to Michtava's needs. The mapping might look like this:

AdministratorDetailsEditModel adminModel =

Mapper.Map<Administrator, AdministratorDetailsEditModel>(administrator);

For single object-to-object conversions,

or this:

IQueryable<AdministratorListViewModel> administrators =

this.administratorService.All().Project().To<AdministratorListViewModel>();

For conversions of entire collections with 1 instruction.

4.. Adding Features

Adding Entities

When adding an entity to the system, one has to do the following:

1. Create the model for the entity in Entities/Models, where all of its relationships with the other entities will be modeled.
2. If Necessary, add an appropriate DbSet in Dal/ApplicationDbContext. It is only necessary if the new entity has no connections to old entities through its members, for example, there is no Questions DbSet because the Question entity only has meaning when being part of a Homework entity. Another example is the WordsDefinition DbSet, which has no connection to other entities in Michtava and so requires a DbSet in the DbContext.
3. If a DbSet has been added following step 2), there is also need to:
   1. Create an appropriate service and repository( see Adding Services and Adding Repositories) + interfaces in the appropriate projects (see server-side maintenance)
   2. Define the Dependency between the interfaces and implementation via Ninject (Frontend/Infra/NinjectDependencyResolver.cs).

If a DbSet has not been added following step 2), new control methods must be written in all services of entities that share a relationship with our new entity.

Adding Services

Every Service Must implement IRepositoryService – it is recommended to create a new interface (that implements the above interface) for each newly added service so that the DependencyResolver will come into play and help out with the bindings.

Adding Repositories

Every Repository Must implement IDeletableEntityRepository – it is recommended to create a new interface (that implements the above interface) for each newly added repo.

Configuring entity relationships

In case of trouble when trying to update the database after a schema change, the solution will often be explicitly defining the properties of an entity relation to EF, using the *Fluent API*. An example of this can be seen in Dal/ApplicationDbContext/OnModelCreating method, and that's where all new Fluent statements should be.

Adding MVC

Adding new web pages (MVC) is done as follows:

1. Determine which area the new web page should belong to (Teacher side, Student side, Admin side, Guest side?)
2. Find the location of the MVC of your chosen Area (see Server-side maintenance for more info).
3. You may use Asp.Net's scaffolding tool as we have, to easily create controllers and views.
4. Controller – choose all of your required services for that MVC, and put the interface that they implement in the constructor of your controller, the actual instance will magically populate your service member (Client-side maintenance)
5. View - If you need to create custom styles, it is recommended you use Frontend/Content/Site.css to expand Michtava's style profile.
6. You will most likely need to add new RoutingInfo to support your MVC, see client-side maintenance to see how it's done.

Adding Seed Data

You might want to add new Seeded data to Michtava, that is data that will populate the DB by default – it is used for testing, creating default data conditions and comfortable publication of the code to a remote server.

The seeding data is in Dal/DatabaseSeeder.cs, it is fairly simple to understand, expand, or edit.

Updating DB

It is important to run Update-Database twice (!!) after every change to the Seeded data(see the following chapter for a more detailed explanation).

5..Useful commands and troubleshooting

Database Commands

All Database-related commands are executed via the Package Manager Console (NuGet). (In VS, Tools => NuGet Package Manager => Package Manager Console).

In order to manually create a database Schema migration:

add-migration -ProjectName Dal -StartUpProject Frontend

In order to enable automatic migrations (shouldn't be used generally, they are already enabled):

enable-migrations -ContextProjectName Dal -StartUpProjectName Frontend

-ContextTypeName Dal.ApplicationDbContext -ProjectName Dal -force

In order to update the database schema (and data!) after any changes that would cause a DB Scheme change (this is the most used DB command):

Update-Database -ConfigurationTypeName Dal.Migrations.Configuration

-ProjectName Dal -verbose

In order to update the database according to a specific migration:

Update-Database -ConfigurationTypeName Dal.Migrations.Configuration

-ProjectName Dal -TargetMigration <your\_migration\_name> -verbose

Creating a new DB file

A very useful troubleshooting action is to create a new DB file (.mdf) to get rid of any Schema or data inconsistency. This is done by changing Frontend/Web.Config as follows:

<connectionStrings>

<add name="*<your\_connection\_name>*" connectionString="Data Source=(LocalDb)\*<your\_instance\_name*>;AttachDbFilename=|DataDirectory*|*

*\<CHANGE\_ME\_FOR\_A\_NEW\_DB\_FILE>*;Integrated Security=True" providerName="System.Data.SqlClient"/>

</connectionStrings>

And running Update-Database twice. (one for schema, one for seed data).

MichtavaResult

Accessing a Service method will sometimes (on Add, Delete, and Update methods) encounter an error – sometimes a runtime exception, but sometimes a logical error that is known to happen – for this we return a MichtavaResult object, a type of wrapper for the result of the query – upon executing a service method you could ask, for example, if the result that was returned is of type MichtavaSuccess indicating a successful query, or MichtavaFailure indicating a failed one, and access the Message member of the MichtavaResult to get more details about what happened.

Exception Policy

Generally speaking, we do not use exceptions as part of the regular control flow, but rather return a MichtavaFailure, however Exceptions are thrown during run-time, when the various libraries and frameworks we use throw them.

EF known issues

Often when writing code in the services, we have encountered problems with EF's lazy-loading of entities – you might not get all of the Entities members when retrieving it from the DbContext. A solid hotfix for this is, given an IQueryable<SomeEntity> that you got your hands on through All() or Get(…), you might request to Include your desired member when retrieving the entity, as follows:

this.someRepository.All().Include(x => x.someMember). FirstOrDefault( x =>

<some\_expression>);