Framework:

- TFS Express: <https://visualstudio.microsoft.com/de/tfs/?rr=https%3A%2F%2Fwww.google.ch%2F>

- MS VS Community: <https://my.visualstudio.com/Downloads?PId=2226>

Lesson 0 (For Admin):

- Find a powerful server (min 8Gb RAM, 200 GB HD) and install TFS express on it.

- Install all then dependencies.

Lesson 1: Intro and Warmup

Preparation:

- Create a Microsoft account.

- Install Visual Studio Community and include all the features to develop Android, iOS and UWP applications. Include ASP Web Development.

- Create a github account and create a new repository.

- Install virtualBox and set up a build server on a big image

Overview:

- .NET landscape (.Net Core, .Net, .Net Standard, .Net Native)

- Multi-platform software development

- Integration with a server component (Web API on IIS)

Introduction to testing automation:

- Unit-testing

- Integration

- Acceptance

- Other forms (load, gui, penetration)

- Mocking

- Integration testing

- Continuous integration, delivery and deployment with TFS.

- Elements of clean coding

- Elements of refactoring

- Introduction to multi-threading and parallelism.

Start:

- From VS clone the repository. Create a new cross-platform project (.NET standard) and name it TripLog.

- Build it.

- Debug the Android part.

- Commit all the changes and push.

Description of the application:

- Purpose

Building:

- Remove

- Add a new ContentPage and name it MainPage.

- Change app.xaml.cs as follows:

MainPage = new NavigationPage(new MainPage());

- Create a folder Models and create a file TripLogEntry.cs as follows:

public class TripLogEntry

{

public string Title { get; set; }

public double Latitude { get; set; }

public double Longitude { get; set; }

public DateTime Date { get; set; }

public int Rating { get; set; }

public string Notes { get; set; }

}

- Edit MainPage.xaml.cs as follows:

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="TripLog.MainPage"

Title="TripLog">

<ContentPage.ToolbarItems>

<ToolbarItem Text="New" Clicked="New\_Clicked" />

</ContentPage.ToolbarItems>

<ContentPage.Content>

<ListView x:Name="trips" ItemTapped="Trips\_ItemTapped">

<ListView.ItemTemplate>

<DataTemplate>

<TextCell Text="{Binding Title}" Detail="{Binding Notes}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</ContentPage.Content>

</ContentPage>

This will change the main page so as to be able to display a list of trips for which two of the six attributes would be displayed.

- Add some hard-coded entries in order to be able to display something for the time being...

var items = new List<TripLogEntry>

{

new TripLogEntry

{

Title = "Washington Monument",

Notes = "Amazing!",

Rating = 3,

Date = new DateTime(2017, 2, 5),

Latitude = 38.8895,

Longitude = -77.0352

},

new TripLogEntry

{

Title = "Statue of Liberty",

Notes = "Inspiring!",

Rating = 4,

Date = new DateTime(2017, 4, 13),

Latitude = 40.6892,

Longitude = -74.0444

},

new TripLogEntry

{

Title = "Golden Gate Bridge",

Notes = "Foggy, but beautiful.",

Rating = 5,

Date = new DateTime(2017, 4, 26),

Latitude = 37.8268,

Longitude = -122.4798

}

};

trips.ItemsSource = items;

This represents a singe-page app containing three hard-coded entries. Data source will follow.

- It is now necessary to add a new page allowing the user to add new entries. Add a new NewEntryPage ContentPage to the solution as follows:

Title="New Entry">

<ContentPage.ToolbarItems>

<ToolbarItem Text="Save" />

</ContentPage.ToolbarItems>

<ContentPage.Content>

<TableView Intent="Form">

<TableView.Root>

<TableSection>

<EntryCell Label="Title" />

<EntryCell Label="Latitude" Keyboard="Numeric" />

<EntryCell Label="Longitude" Keyboard="Numeric" />

<EntryCell Label="Date" />

<EntryCell Label="Rating" Keyboard="Numeric" />

<EntryCell Label="Notes" />

</TableSection>

</TableView.Root>

</TableView>

</ContentPage.Content>

This represents a simple page showing a list of six entries named as above which, for now, need to manually populated and cannot be persisted.

- It is finally necessary to create a page which displays more info about the entry, including a map with pin pointing to the position. In order to do this it is necessary to create a new ContentPage DetailPage and modify its default constructor as follows:

public DatailPage (TripLogEntry entry)

{

InitializeComponent ();

}

The xaml file will need to show a map with a label on top of it displaying some details about the location. Here the code:

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"

x:Class="TripLog.DetailPage"

Title="Details">

<ContentPage.Content>

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="4\*" />

<RowDefinition Height="Auto" />

<RowDefinition Height="1\*" />

</Grid.RowDefinitions>

<maps:Map x:Name="map" Grid.RowSpan="3" />

<BoxView Grid.Row="1" BackgroundColor="White" Opacity=".8" />

<StackLayout Padding="10" Grid.Row="1">

<Label x:Name="title" HorizontalOptions="Center" />

<Label x:Name="date" HorizontalOptions="Center" />

<Label x:Name="rating" HorizontalOptions="Center" />

<Label x:Name="notes" HorizontalOptions="Center" />

</StackLayout>

</Grid>

</ContentPage.Content>

</ContentPage>

In order to use the map field and the related assembly it is necessary to add a reference to the assembly by making use of NuGet (Xamarin.Forms.Map). Update through NuGet any additionally required assembly. Solve any possible compilation error.

Now it is possible to show in the datail Page the map with a pin pointing to the actual position. Here the code:

public DetailPage (TripLogEntry entry)

{

InitializeComponent();

map.MoveToRegion(MapSpan.FromCenterAndRadius(new Position(entry.Latitude, entry.Longitude), Distance.FromMiles(.5)));

map.Pins.Add(new Pin

{

Type = PinType.Place,

Label = entry.Title,

Position = new Position(entry.Latitude, entry.Longitude)

});

title.Text = entry.Title;

date.Text = entry.Date.ToString("M");

rating.Text = $"{entry.Rating} star rating";

notes.Text = entry.Notes;

}

Finally it is necesary to update the target specific implementations by initializing the Maps binary as follows:

Android:

global::Xamarin.Forms.Forms.Init(this, savedInstanceState);

Xamarin.FormsMaps.Init(this, savedInstanceState);

iOS:

Xamarin.FormsMaps.Init();

LoadApplication(new App());

UWP:

Xamarin.FormsMaps.Init("Token to be added later!");

Xamarin.Forms.Forms.Init(e);

This completes the implementation. However, debugging the file application will not work due to a variety of settings still missing (and missing in the book). Here is the list of steps that need to be executed:

Replace the Android Manifest file with the following text:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:versionName="1.0" package="triplog.android" android:installLocation="auto">

<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="27" />

<!-- Missing in the book. -->

<!-- Google Maps for Android v2 requires OpenGL ES v2 -->

<uses-feature android:glEsVersion="0x00020000" android:required="true" />

<!-- Necessary for apps that target Android 9.0 or higher -->

<uses-library android:name="org.apache.http.legacy" android:required="false" />

<!-- These are optional, but recommended. They will allow Maps to use the My Location provider. -->

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION" />

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission android:name="triplog.android.permission.MAPS\_RECEIVE" />

<permission android:name="triplog.android.permission.MAPS\_RECEIVE" android:protectionLevel="signature" />

<application android:label="triplog.android">

<!-- Missing in the book, but available for download. -->

<!-- Get your personal API KEY by following the following guide: https://docs.microsoft.com/de-de/xamarin/android/platform/maps-and-location/maps/obtaining-a-google-maps-api-key?tabs=windows -->

<!-- https://nearplace.com/blog/how-to-generate-google-map-api-key-for-free/ -->

<meta-data android:name="com.google.android.maps.v2.API\_KEY" android:value="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" />

<meta-data android:name="com.google.android.gms.version" android:value="@integer/google\_play\_services\_version" />

</application>

</manifest>

Follow the following guide to obtain an API KEY. In the process you will need to past the thumbrint of a local certificate on a web form. Here the details:

"C:\Program Files (x86)\Java\jdk1.8.0\_131\bin\keytool.exe" -list -v -keystore "C:\Users\Umberto\AppData\Local\Xamarin\Mono for Android\debug.keystore"

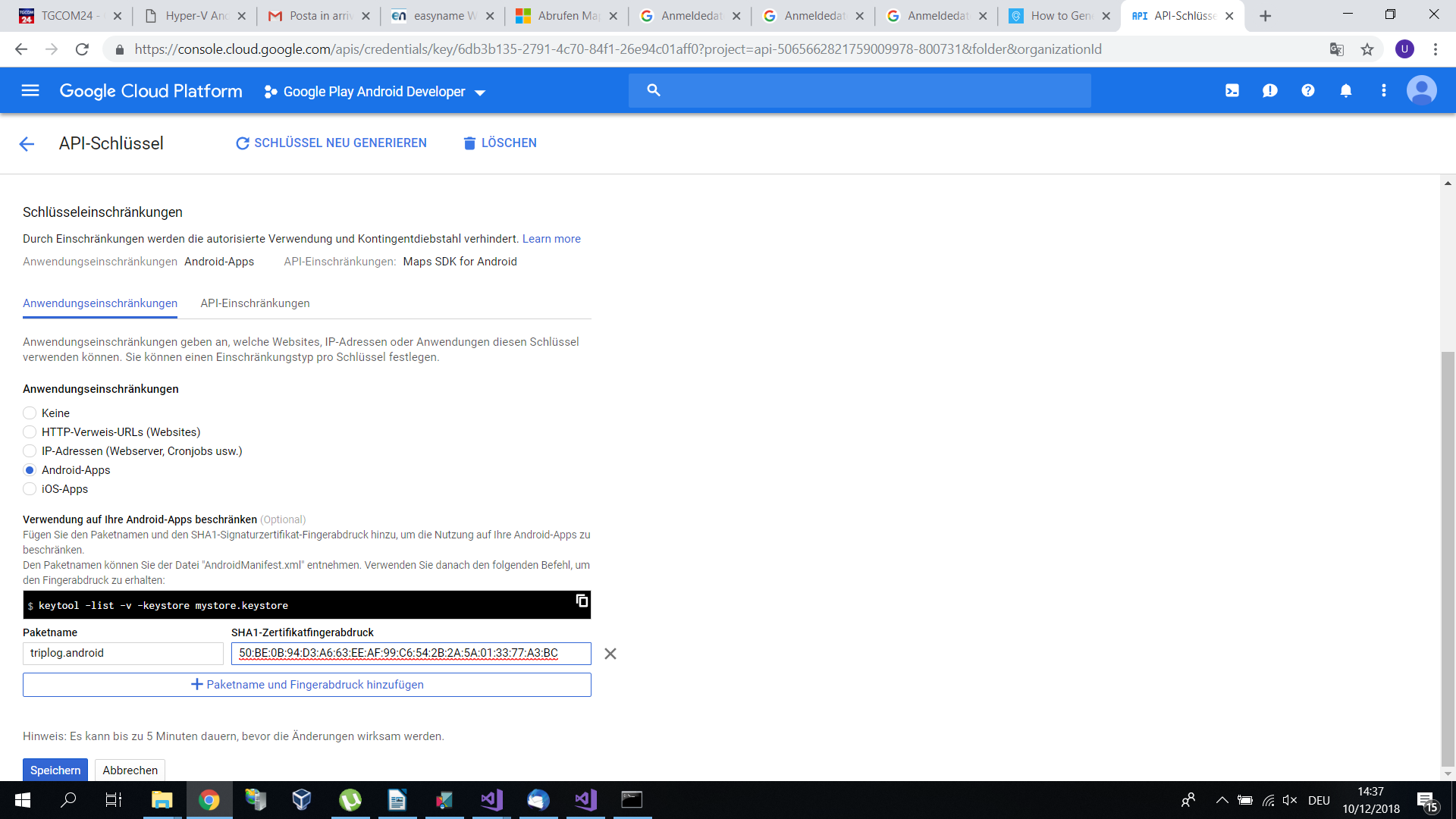
password=android

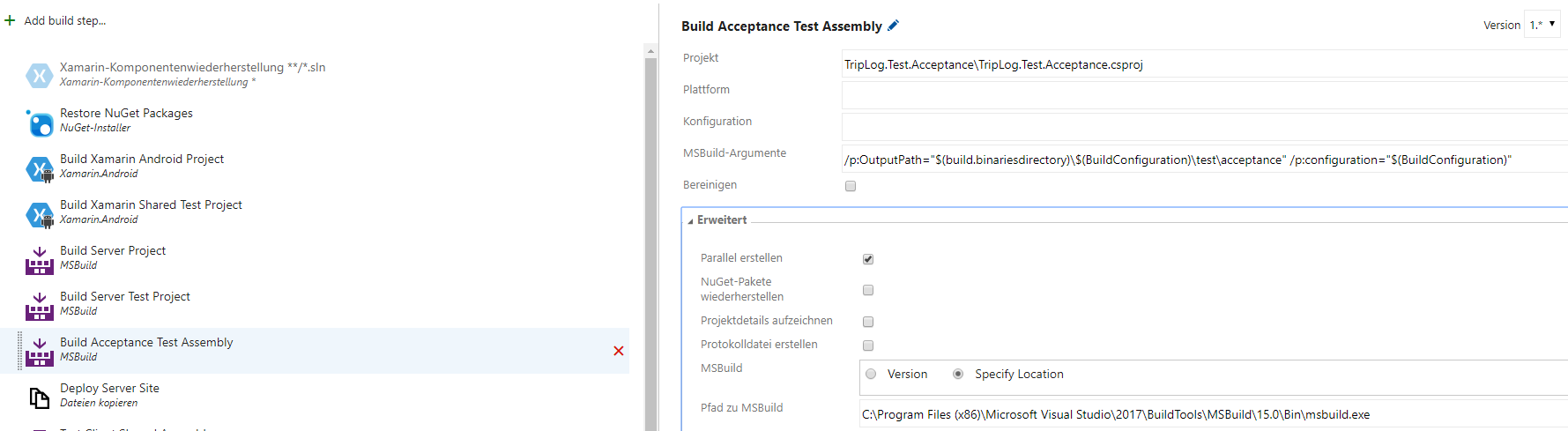
Keystore name: "debug.keystore"

Keystore password: "android"

Key alias: "androiddebugkey"

Key password: "android"





Finally paste the key on the file above at the appropriate position. Debug the software and check that Google maps is reachable.

Open point: Who is interested in installing, configuring and setting up TFS on his own machine/server?

Lesson 2:

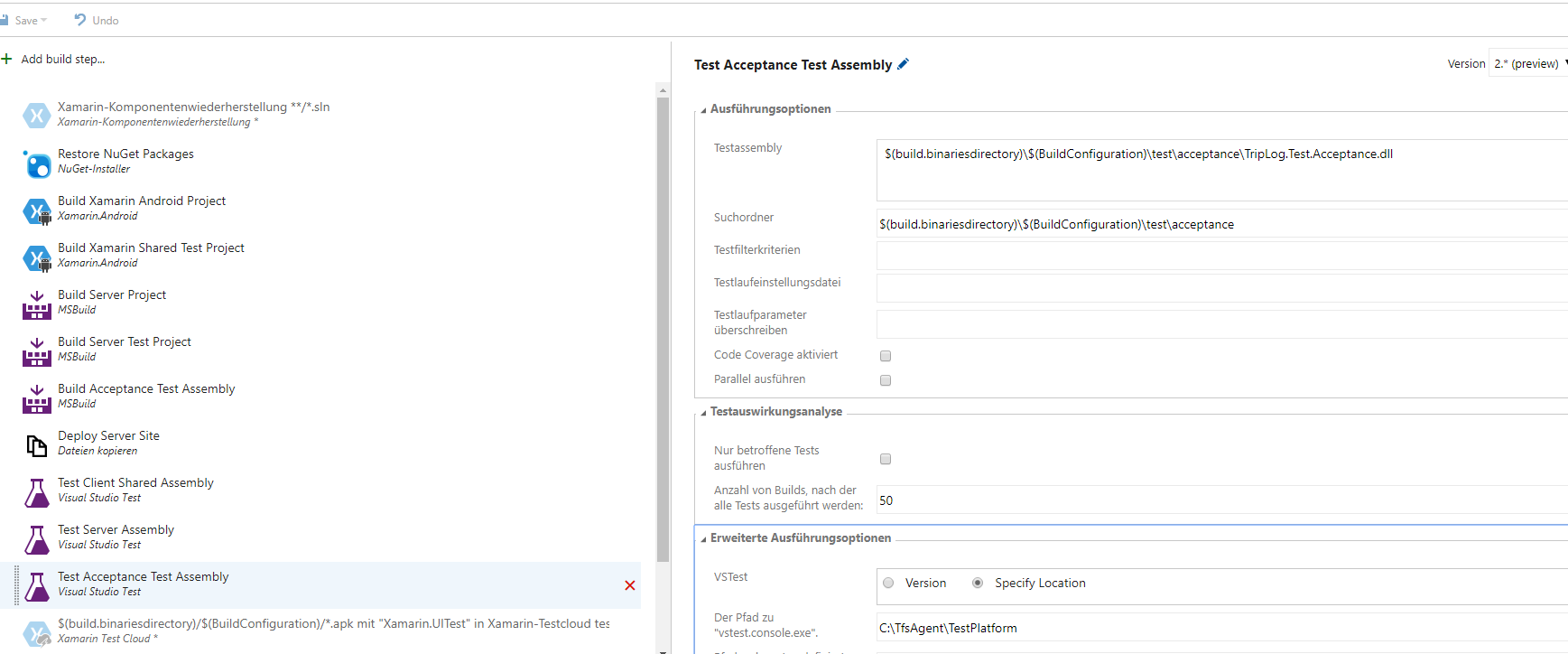
- Build Server introduction.

- Linking Github with TFS by creating the necessary Oath entry point.

Go to your GitHub account, select developer settings and personal access token. Select Generate new token and tick the following options:

**- repo, user, admin:repo\_hook**

Type in a name such as TfsBuildTokenJuventus.



-

Click Generate token.

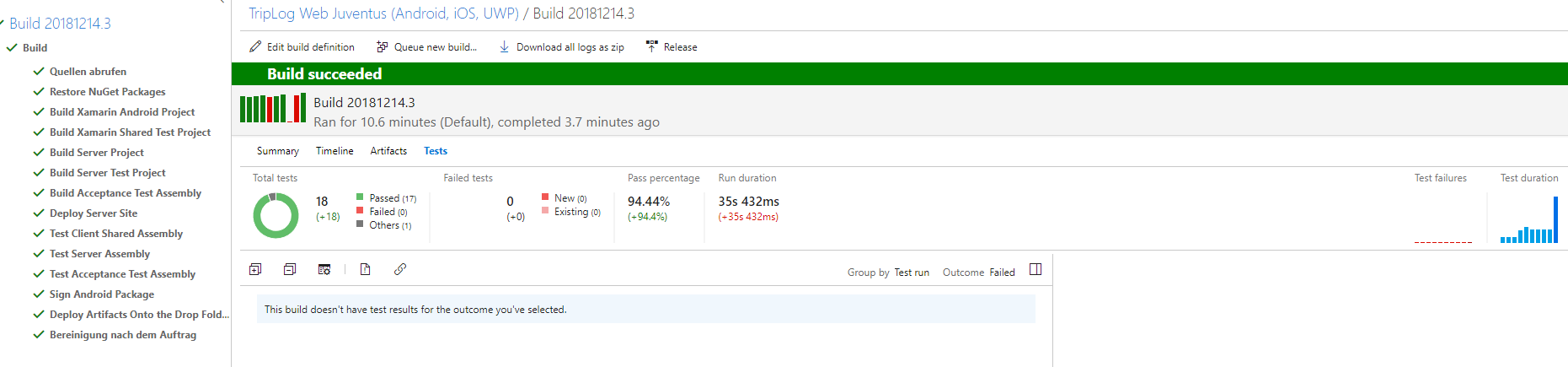
On the following screen ensure to copy the text on a safe place since it will only be displayed once.

This setup will allow TFS to poll the repository for changes and trigger a build every time a new check in takes place.

Creating a minimal build definition.

Open TFS by pasting the following URL: <http://desktop-kj5q2iu:8080/tfs/>

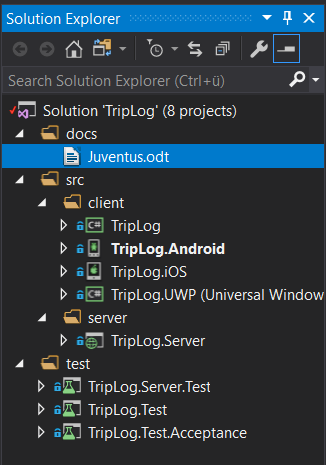
Choose create new project, fill out the necessary fields and click create:



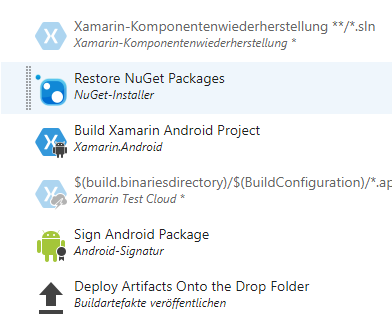
Since the main architecture of the build is to pull code from a gitHub repository choose "Build Code from an external Repository".

On the next screen click "New Definition"- Then choose "Xamarin.Android" and lick Next.

Choose "Remote Git Repository" and tick "Continuous Integration" and click Create as shown on the picture:

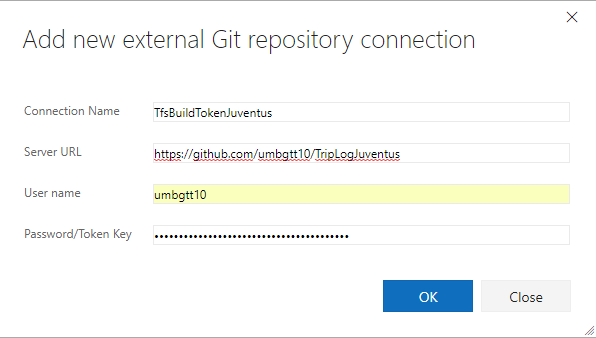


On the next screen remove the test project and update the build steps with more meaningful names:

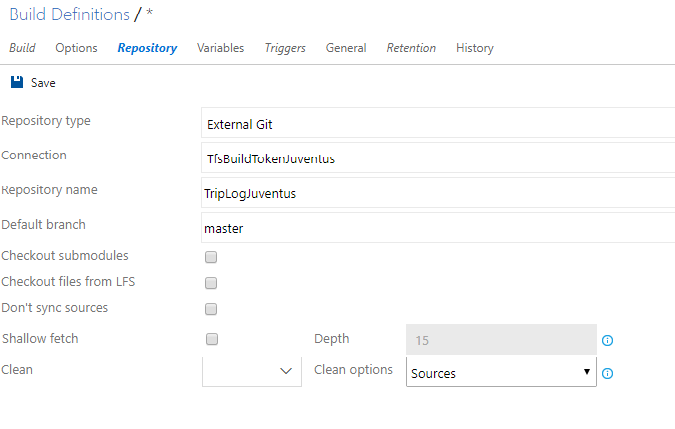


Click on Repository and on Manage on the right. With the following step we will establish a connection between the Build definition and the repository.

Start by clicking New Service Endpoint and selectin "External Git Repository". On the next screen paste the information obtained before and along with the URL of the rpository and click OK:



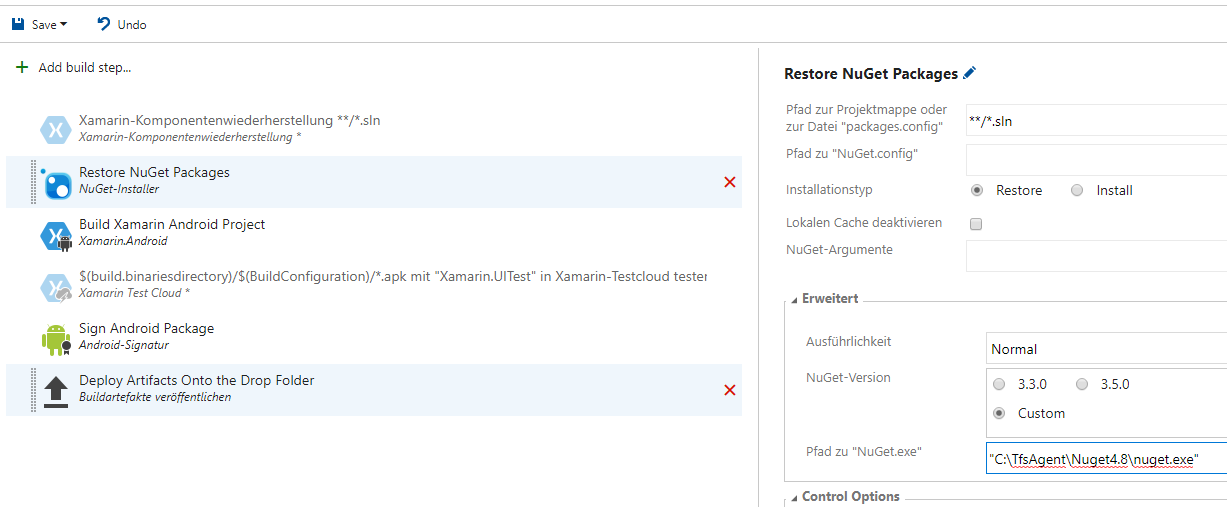
Go back to the build definition and click refresh on left: The newly created connection point should be selected automatically. Fill out the remaining fields. This should be the end result:



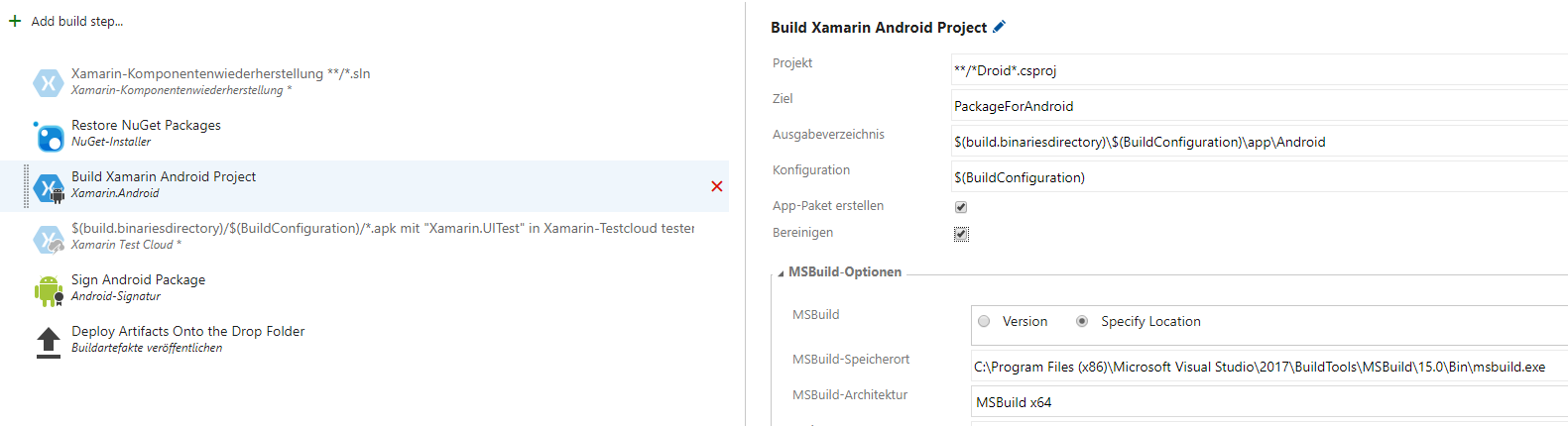
Go back to build and click save and provide a meaningful name for the build.

Chose the build steps one by one and update them as appropriate/follows:

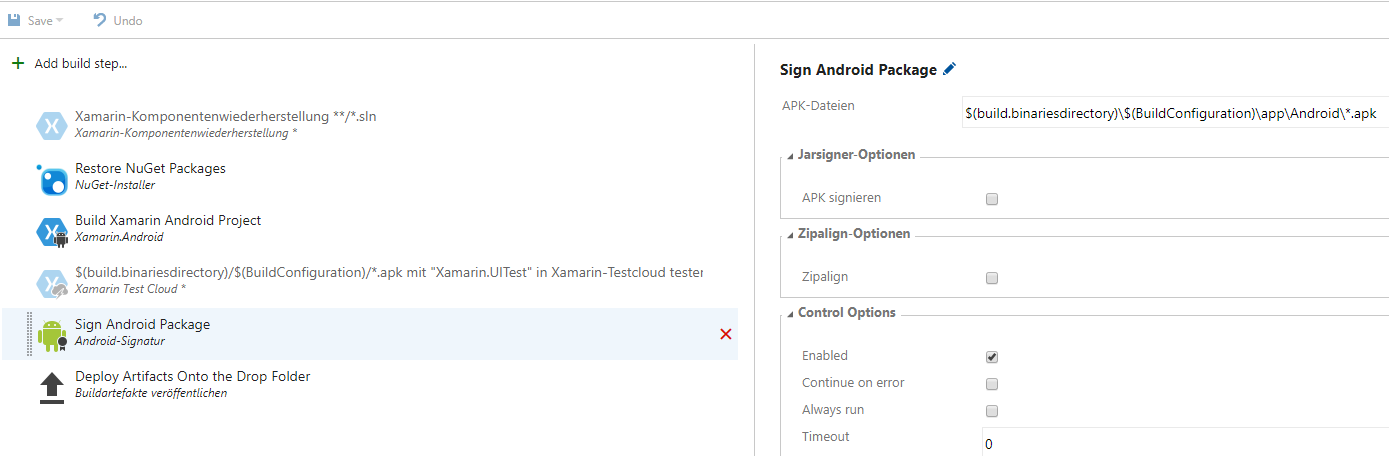
- Restore NuGet



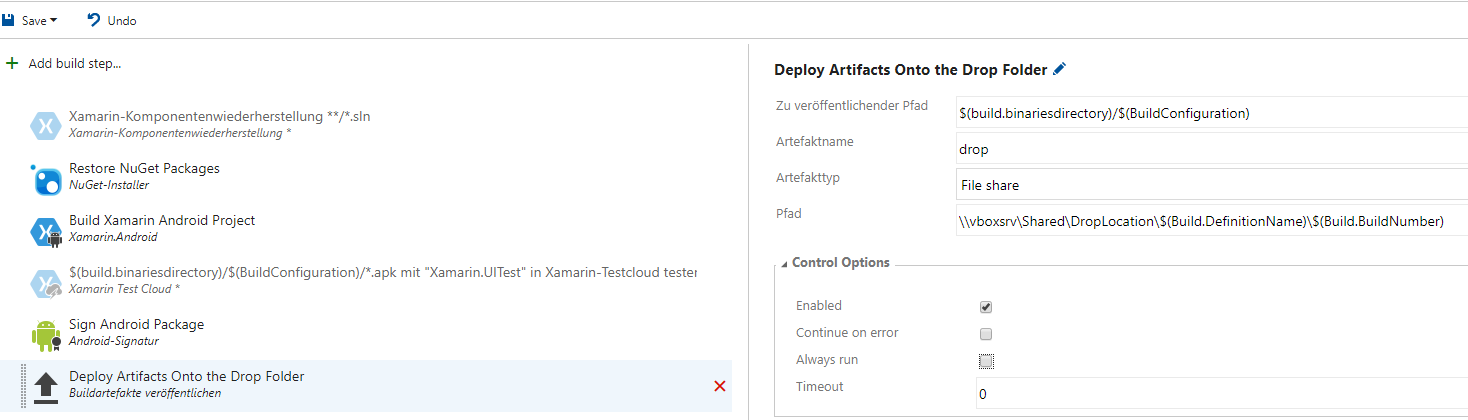
- Build Android Project



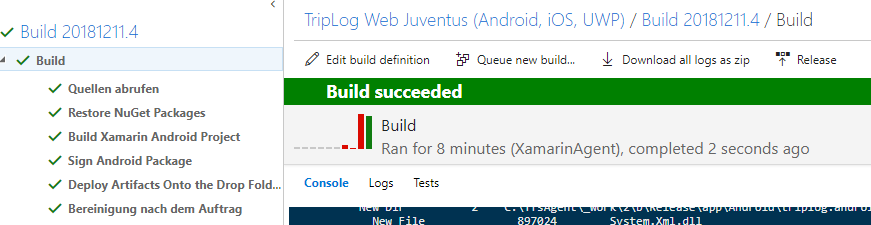
- Update the signing part appropritely:



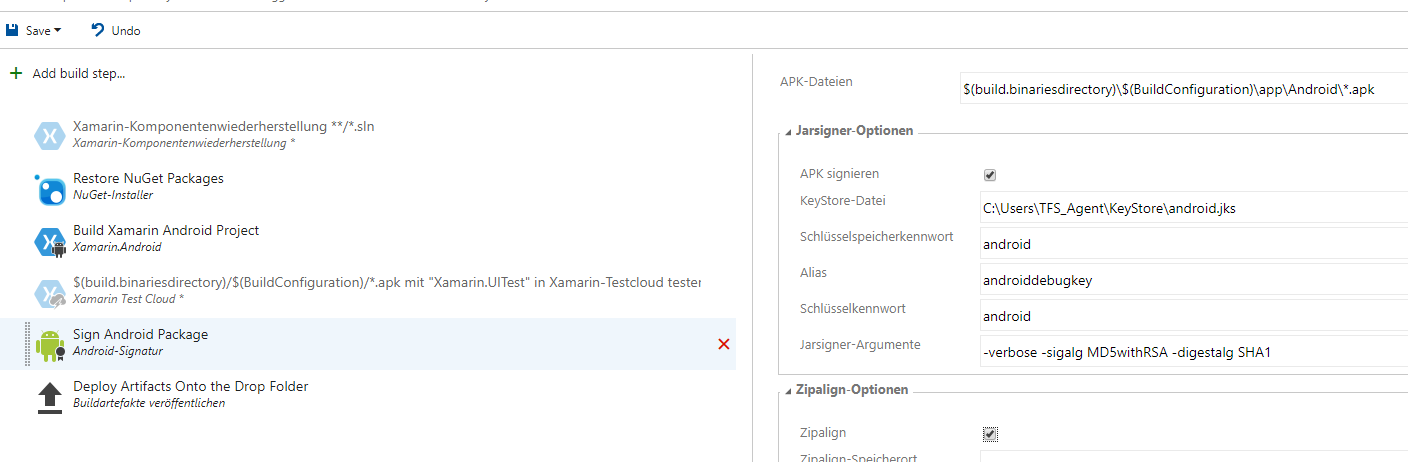
- Define a meaningful drop location



Click save and then enqueue a new build. This should succeed:

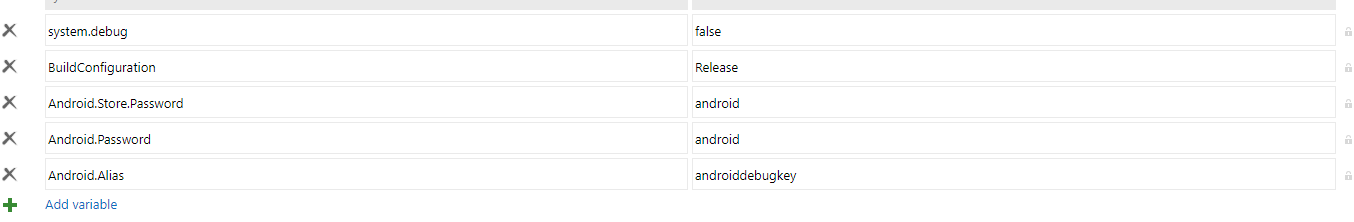


Now it is a good moment to add the signing part to the build. Open the sign step of the build and modify it as follows:

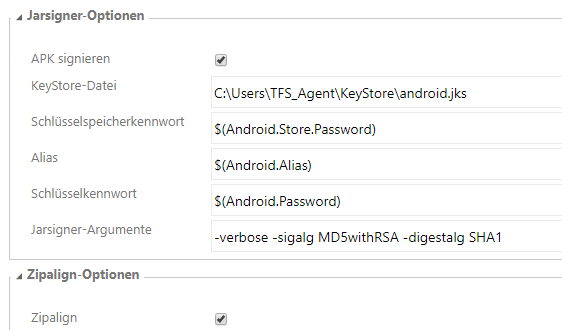


And enqueue a build.

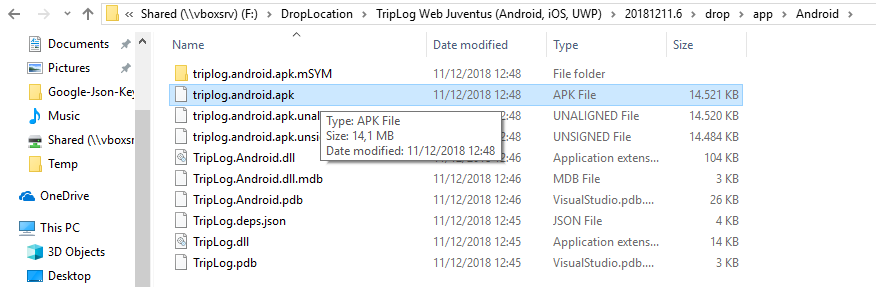
Now although this is a working setup, it is not advised to exposed credentials in plain text on a build server. It would be better if it was possible to hide these settings behind environment variables. Luckily this is possible by making use of the variables section. Click on Variables and create three variables as follows:



Then replace the definition of the previous step with the variables as follows:



Test the signing by enqueuing a new build and ensure that the signed package gets created accordingly:



To finalize the obfuscation of the variables go back to variables and click on the small padlocks beside the variables them selves

****

As a final check push a minor change to the source code repository and check that the build server spots it and build it successfully as documented above.

So the build server, after every check in onto the github repository, is now capable of:

- Building the Android version of the project.

- Signing the APK file.

- Deploying the singed APK onto a build-specific location.

Starting from now it is imperative to keep the source code and build server in sync and up to date so that every code change leads to the succefull creation of a new package.

The focus will now move to the code with the introduction and implementation of the infamous MVVM patern.

[Introduce the concept of MVVM]

Start by adding two folders to the TripLog project: Views and ViewModels.

Move the 3 main screens (.cs and .xaml) to the views folder. Update all the references and name spaces on every file and on the app.cs file in order for the project to build buildable.

In the ViewModel folder add a a base class for the future ViewModel classes as follows:

public abstract class BaseViewModel : INotifyPropertyChanged

{

protected BaseViewModel()

{

}

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

public event PropertyChangedEventHandler PropertyChanged;

}

As the code above shows the implementation of the PropertyChanged event is crucial: It allows pages to be notified when properties of the (corresponding/matching) ViewModel have changed.

Now start by creating a new file in the ViewModel folder called MainViewModel.cs. Add parameterless constructor which inherits from BaseViewModel as follows:

public class MainViewModel : BaseViewModel

Add an observable collection representing the list of TripLogEntry displayed on the main screen:

#region Observables

private ObservableCollection<TripLogEntry> \_logEntries;

public ObservableCollection<TripLogEntry> LogEntries

{

get

{ return \_logEntries; }

set

{

\_logEntries = value;

OnPropertyChanged();

}

}

#endregion

Now move the hard-coded trips from the view to the constructor of the MainPage to the constructor of MainViewModel:

public MainViewModel()

{

LogEntries = new ObservableCollection<TripLogEntry>();

// Add hard-coded entries here

var item1 = new TripLogEntry

{

Title = "Washington Monument",

Notes = "Amazing!",

Rating = 3,

Date = new DateTime(2017, 2, 5),

Latitude = 38.8895,

Longitude = -77.0352

};

var item2 = new TripLogEntry

{

Title = "Statue of Liberty",

Notes = "Inspiring!",

Rating = 4,

Date = new DateTime(2017, 4, 13),

Latitude = 40.6892,

Longitude = -74.0444

};

var item3 = new TripLogEntry

{

Title = "Golden Gate Bridge",

Notes = "Foggy, but beautiful.",

Rating = 5,

Date = new DateTime(2017, 4, 26),

Latitude = 37.8268,

Longitude = -122.4798

};

LogEntries.Add(item1);

LogEntries.Add(item2);

LogEntries.Add(item3);

}

Update the constructor of the MainPage by setting the binding between View and ViewModel:

public MainPage()

{

InitializeComponent();

BindingContext = new MainViewModel();

}

Finally update MainPage.xaml by explicitly setting the binding as follows:

<ListView x:Name="trips" ItemTapped="Trips\_ItemTapped" ItemsSource="{Binding LogEntries}">

Debug the solution and ensure that the behaviour of the application is unchanged. Check in the changes.

Now proceed by creating the NewEntryViewModel.cs in the viewModels folder as before:

public class NewEntryViewModel : BaseViewModel

{

public NewEntryViewModel()

{

}

}

Proceed by adding observable properties for each of the fields of interest and by setting some defaults for the constructor:

#region Observables

private string \_title;

public string Title

{

get { return \_title; }

set { \_title = value; OnPropertyChanged(); }

}

private double \_latitude;

public double Latitude

{

get { return \_latitude; }

set { \_latitude = value; OnPropertyChanged(); }

}

private double \_longitude;

public double Longitude

{

get { return \_longitude; }

set { \_longitude = value; OnPropertyChanged(); }

}

private DateTime \_date;

public DateTime Date

{

get { return \_date; }

set { \_date = value; OnPropertyChanged(); }

}

private int \_rating;

public int Rating

{

get { return \_rating; }

set { \_rating = value; OnPropertyChanged(); }

}

private string \_notes;

public string Notes

{

get { return \_notes; }

set { \_notes = value; OnPropertyChanged(); }

}

#endregion

public NewEntryViewModel()

{

Date = DateTime.Today;

Rating = 1;

}

Now we want to add the necessary logic to be able to handle saving event triggered by clicking the save button on the NewEntryPage. This can be accomplished by implementing a Command pattern and by registering the execution of the command to the event handler of the save button.

Here a possible implementation:

#region Commands

private Command \_saveCommand;

public Command SaveCommand

{

get

{

if (\_saveCommand == null)

{

\_saveCommand = new Command(ExecuteSaveCommand, CanSave);

}

return \_saveCommand;

}

}

private bool CanSave(object arg)

{

return !string.IsNullOrEmpty(Title);

}

private void ExecuteSaveCommand(object obj)

{

var newEntry = new TripLogEntry

{

Title = Title,

Notes = Notes,

Rating = Rating,

Date = Date,

Latitude = Latitude,

Longitude = Longitude

};

// Save this new entry later: Introduce a data layer.

}

#endregion

Update the Title properties so as to take into account whether a command can be fired or not by adding the following code:

public string Title

{

get { return \_title; }

set

{

\_title = value;

OnPropertyChanged();

SaveCommand.ChangeCanExecute();

}

}

Next update the constructor of the NewEntryPage in order to set the binding context:

public NewEntryPage ()

{

InitializeComponent ();

BindingContext = new NewEntryViewModel();

}

Then update the NewEntryPage.xaml.cs so as to bind the different texts to the observable properties:

<TableSection>

<EntryCell Label="Title" Text="{Binding Title}"/>

<EntryCell Label="Latitude" Text="{Binding Latitude}" Keyboard="Numeric" />

<EntryCell Label="Longitude" Text="{Binding Longitude}" Keyboard="Numeric" />

<EntryCell Label="Date" Text="{Binding Date, StringFormat='{0:d}'}" />

<EntryCell Label="Rating" Text="{Binding Rating}" Keyboard="Numeric" />

<EntryCell Label="Notes" Text="{Binding Notes}"/>

</TableSection>

Finally hook the save Command to the Save button:

<ToolbarItem Text="Save" Command="{Binding SaveCommand}"/>

Build and debug the software and check that the Save button is grayed out as long as the Title is empty.

Now create a new file named DetailViewModel.cs as before and add an observable property as follows:

public class DetailViewModel : BaseViewModel

{

#region Observables

private TripLogEntry \_entry;

public TripLogEntry Entry

{

get { return \_entry; }

set { \_entry = value; OnPropertyChanged(); }

}

#endregion

}

Add a constructor taking a TripLogEntry as argument as follows:

public DetailViewModel(TripLogEntry entry)

{

Entry = entry;

}

Update the constructor of the DetailPage class so as to set up the binding:

public DetailPage (TripLogEntry entry)

{

InitializeComponent();

BindingContext = new DetailViewModel(entry);

//

(This will need to be refactored later...)

Comment out the following lines in the same constructor:

//title.Text = entry.Title;

//date.Text = entry.Date.ToString("M");

//rating.Text = $"{entry.Rating} star rating";

//notes.Text = entry.Notes;

Then add the binding in the Detail.xaml page as follows so as to set up the binding:

<StackLayout Padding="10" Grid.Row="1">

<Label x:Name="title" Text="{Binding Entry.Title}" HorizontalOptions="Center" />

<Label x:Name="date" Text="{Binding Entry.Date, StringFormat='{0:M}'}" HorizontalOptions="Center" />

<Label x:Name="rating" Text="{Binding Entry.Rating, StringFormat='{0} star rating!'}" HorizontalOptions="Center" />

<Label x:Name="notes" Text="{Binding Entry.Notes}" HorizontalOptions="Center" />

</StackLayout>

Finally update the DetailPage so as to use the ViewModel to set GUI properties as follows:

private DetailViewModel \_vm

{

get

{

return BindingContext as DetailViewModel;

}

}

public DetailPage (TripLogEntry entry)

{

InitializeComponent();

BindingContext = new DetailViewModel(entry);

var position = new Position(\_vm.Entry.Latitude, \_vm.Entry.Longitude);

map.MoveToRegion(MapSpan.FromCenterAndRadius(position, Distance.FromMiles(.5)));

map.Pins.Add(new Pin

{

Type = PinType.Place,

Label = entry.Title,

Position = position

});

Build and debug the software. Check in and ensure that the build is successful.

Lesson 3

- Introduction to the Factory design pattern

The quality of a software project cannot be marked as sufficient until at least the creation of the object is completely under control. A relatively simple way to organize the creation of similar objects is by means of the Factory pattern.

[Add references, links and class diagram]

Limited to the contest of the TripLog application, it is easy to recognize that ViewModels and Views represents two possible groups of similar objects.

Let´s try to refactor the software, thus introducing the factory pattern, without altering the behaviour of the application.

In order to do that some initial refactoring is required. Start by adding the following two methods to the BaseViewModel class:

public abstract void Init();

public abstract void Init(TripLogEntry entry);

Implement them in all the derivatives as follows:

- DetailViewModel:

public DetailViewModel()

{

}

public override void Init()

{

throw new NotImplementedException();

}

public override void Init(TripLogEntry entry)

{

Entry = entry;

}

- MainViewModel:

public MainViewModel()

{

}

public override void Init()

{

LogEntries = new ObservableCollection<TripLogEntry>();

// Add hard-coded entries here

var item1 = new TripLogEntry

{

Title = "Washington Monument",

Notes = "Amazing!",

Rating = 3,

Date = new DateTime(2017, 2, 5),

Latitude = 38.8895,

Longitude = -77.0352

};

var item2 = new TripLogEntry

{

Title = "Statue of Liberty",

Notes = "Inspiring!",

Rating = 4,

Date = new DateTime(2017, 4, 13),

Latitude = 40.6892,

Longitude = -74.0444

};

var item3 = new TripLogEntry

{

Title = "Golden Gate Bridge",

Notes = "Foggy, but beautiful.",

Rating = 5,

Date = new DateTime(2017, 4, 26),

Latitude = 37.8268,

Longitude = -122.4798

};

LogEntries.Add(item1);

LogEntries.Add(item2);

LogEntries.Add(item3);

}

public override void Init(TripLogEntry entry)

{

throw new NotImplementedException();

}

- NewEntryViewModel:

public NewEntryViewModel()

{

}

public override void Init()

{

Date = DateTime.Today;

Rating = 1;

}

public override void Init(TripLogEntry entry)

{

throw new NotImplementedException();

}

Modify the pages so as to take the former changes into account:

Main:

public MainPage()

{

InitializeComponent();

var viewModel = new MainViewModel();

viewModel.Init();

BindingContext = viewModel;

}

New:

public NewEntryPage ()

{

InitializeComponent ();

var viewModel = new NewEntryViewModel();

viewModel.Init();

BindingContext = viewModel;

}

Details:

public DetailPage (TripLogEntry entry)

{

InitializeComponent();

var viewModel = new DetailViewModel();

viewModel.Init(entry);

BindingContext = viewModel;

Rebuild and test the software. Check in.

COntinue by introducing an enumeration defining the ViewModel type as follows:

public enum TripLogViewModelType

{

Main = 0,

Detail = 1,

New = 2

}

The meaning is obvious. Now by making use of this enumeration we are in a position to map ViewModel types to thier specific implementation. Here a possible implementation of the ViewModelFactory:

public class ViewModelFactory

{

public ViewModelFactory()

{

}

public BaseViewModel Build(TripLogViewModelType viewModelType)

{

switch(viewModelType)

{

case TripLogViewModelType.Detail:

return new DetailViewModel();

case TripLogViewModelType.Main:

return new MainViewModel();

case TripLogViewModelType.New:

return new NewEntryViewModel();

default:

throw new Exception($"Unknown {viewModelType}");

}

}

}

In order to integrate the factory mentioned above, it is necessary to inject the ViewModels into their respective models. As a result, all the views´ constructors need to be refactored so as to expose a BaseViewModel as a parameter. Here the implementations:

Main:

public MainPage(BaseViewModel viewModel)

{

InitializeComponent();

BindingContext = viewModel;

}

New:

public NewEntryPage (BaseViewModel viewModel)

{

InitializeComponent ();

BindingContext = viewModel;

}

Detail:

public DetailPage (BaseViewModel viewModel)

{

InitializeComponent();

BindingContext = viewModel;

Consistently the calling methods need to instantiate and initialize the ViewModels before they injected. Here the implementation:

- Main (In the app.xaml.cs):

public App()

{

InitializeComponent();

var viewModel = new MainViewModel();

viewModel.Init();

var mainPage = new MainPage(viewModel);

MainPage = new NavigationPage(mainPage);

}

- New (in MainPage.xaml.cs):

public void New\_Clicked(object sender, EventArgs e)

{

var viewModel = new NewEntryViewModel();

viewModel.Init();

var newEntryPage = new NewEntryPage(viewModel);

Navigation.PushAsync(newEntryPage);

}

- Detail (in MainPage.xaml.cs):

async void Trips\_ItemTapped(object sender, ItemTappedEventArgs e)

{

var trip = (TripLogEntry)e.Item;

var viewModel = new DetailViewModel();

viewModel.Init(trip);

var detailPage = new DetailPage(viewModel);

await Navigation.PushAsync(detailPage);

// Clear selection

trips.SelectedItem = null;

}

Test the application and check in.

Now the viewModelFactory can be integrated and injected as follows on the app.xaml.cs:

private ViewModelFactory \_viewModelFactory;

public App()

{

InitializeComponent();

\_viewModelFactory = new ViewModelFactory();

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.Main);

viewModel.Init();

var mainPage = new MainPage(\_viewModelFactory, viewModel);

MainPage = new NavigationPage(mainPage);

}

and used inside main.xaml.cs as follows:

public partial class MainPage : ContentPage

{

private ViewModelFactory \_viewModelFactory;

public MainPage(ViewModelFactory viewModelFactory, BaseViewModel viewModel)

{

InitializeComponent();

\_viewModelFactory = viewModelFactory;

BindingContext = viewModel;

}

public void New\_Clicked(object sender, EventArgs e)

{

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.New);

viewModel.Init();

var newEntryPage = new NewEntryPage(viewModel);

Navigation.PushAsync(newEntryPage);

}

async void Trips\_ItemTapped(object sender, ItemTappedEventArgs e)

{

var trip = (TripLogEntry)e.Item;

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.Detail);

viewModel.Init(trip);

var detailPage = new DetailPage(viewModel);

await Navigation.PushAsync(detailPage);

// Clear selection

trips.SelectedItem = null;

}

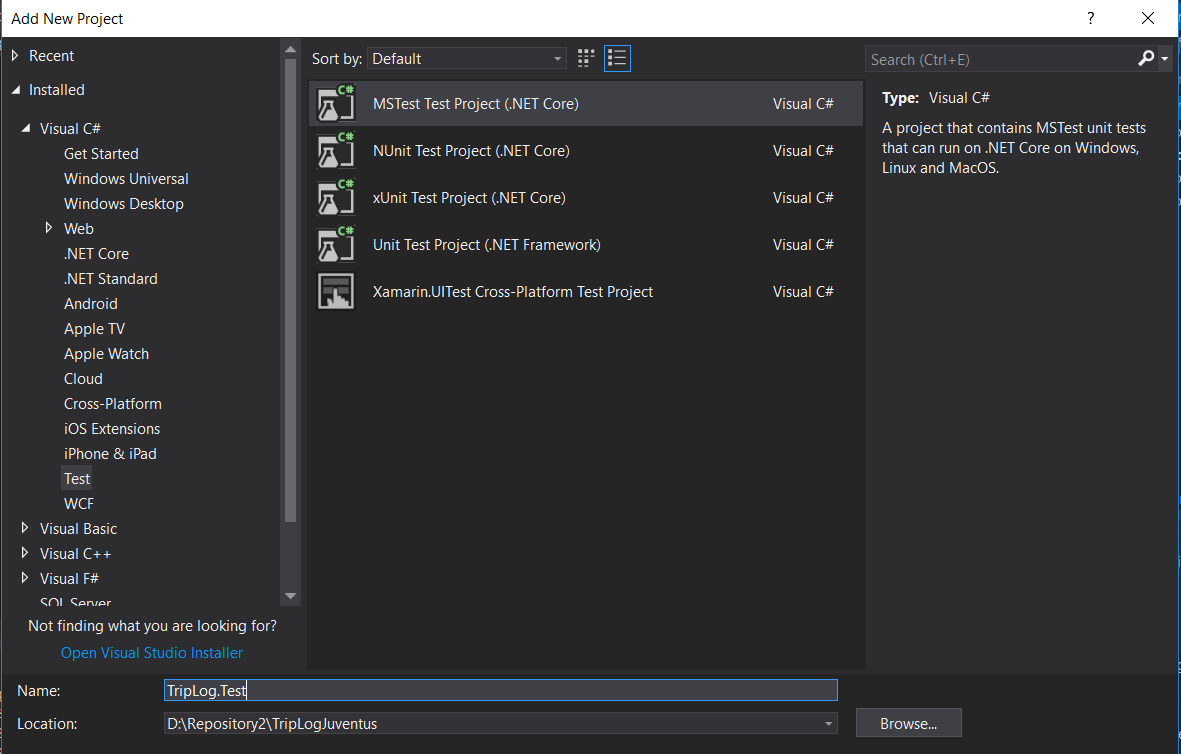
}

The end result so far is that the views no longer know how to build the corresponding model. The factory is now the owner of this knowledge (SRP!).

Having the ViewModel class in place is a good occasion to implement some unit tests (at this point for didactic purposes).

[Define unit tests, elicit their importance and the difference between them and any other kind of tests]

Add a new project the the solution and name it TripLog.Test as follows:



We intend to add a test for each possible viewModel that can be created with factory defined before. Start by creating three empty methods named intuitively:

[TestClass]

public class ViewModelFactoryTests

{

[TestMethod]

public void MainViewModelCreationTest()

{

}

[TestMethod]

public void NewEntryViewModelCreationTest()

{

}

[TestMethod]

public void DetailViewModelCreationTest()

{

}

}

Populate the first test by following the standard uni test pattern: Build, Setup, Check.

Here the implementation of the first test:

[TestMethod]

public void MainViewModelCreationTest()

{

var viewModelFactory = new ViewModelFactory();

var viewModel = viewModelFactory.Build(TripLogViewModelType.Main);

Assert.IsInstanceOfType(viewModel, typeof(MainViewModel));

}

Run the test and ensure it is green. Populate the remaining two tests. Remove duplication.

Exercise: Add unit tests for each ViewModel as appropriate.

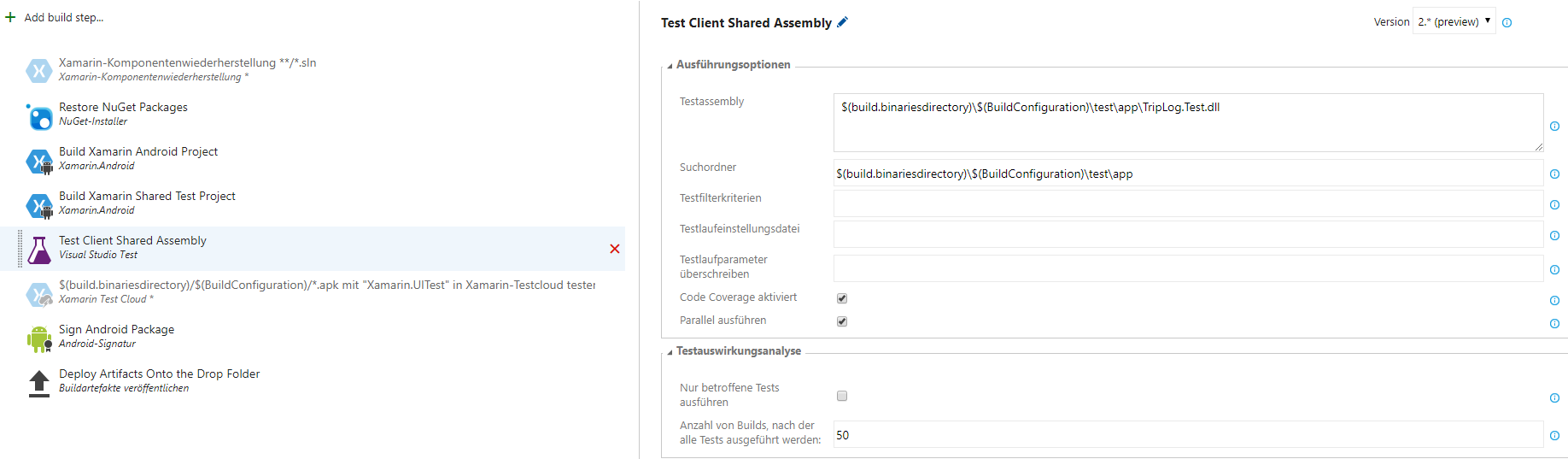
Check in.

At this point it necessary to include the execution of the actually available tests in the Build. In order to do that it is ncessary to add two build steps: one for thecreation and deplyoment of the test binaries and one to automatically test them.

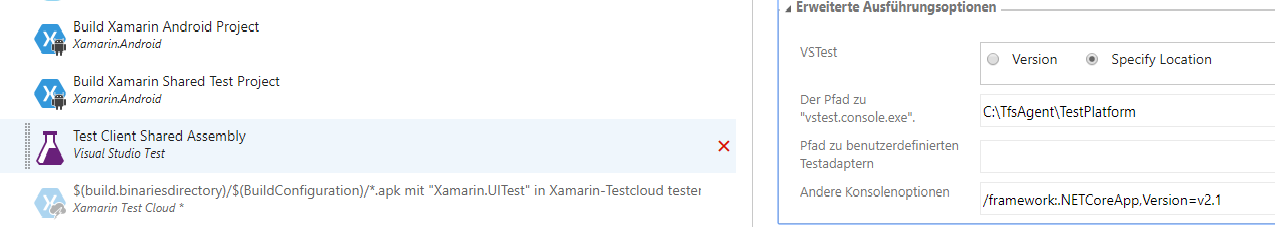
Let´s start by creating a build step as follows:



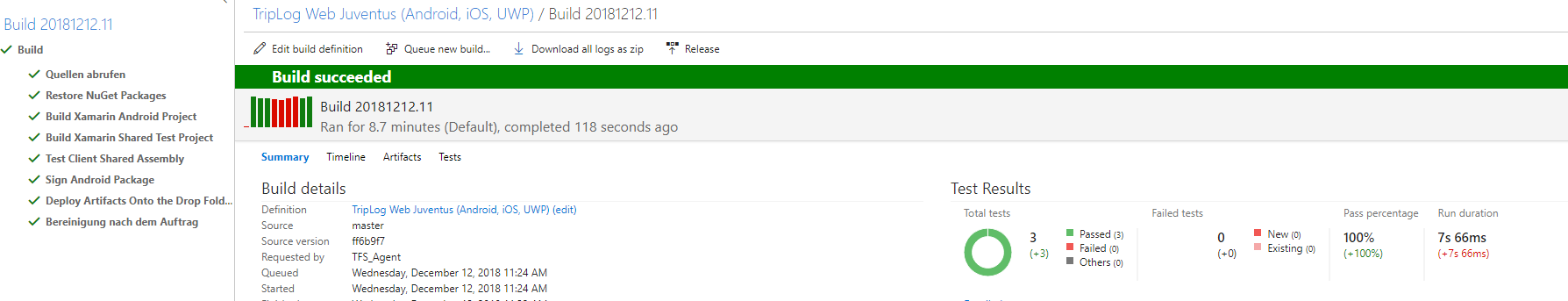
This will build the test binary and deploy it as configured. Then it is required to add another build step to have the tests executed. This can be accomplished in the following way:



Do not forget to add the special argument for MSBuild:



Lunch a build and verify that all the steps are executed succesfully and the tests pass:



By the same token it is now possible to introduce a view factory in order to concentrate the creation logic of the views. This can be achieved by defining the factory as follows:

public class ViewFactory

{

private readonly ViewModelFactory \_viewModelFactory;

public ViewFactory(ViewModelFactory viewModelFactory)

{

\_viewModelFactory = viewModelFactory;

}

public ContentPage Build(TripLogViewModelType modelType, BaseViewModel viewModel)

{

switch(modelType)

{

case TripLogViewModelType.Main:

var mainPage = new MainPage(this, \_viewModelFactory, viewModel);

return mainPage;

case TripLogViewModelType.New:

var newEntryPage = new NewEntryPage(viewModel);

return newEntryPage;

case TripLogViewModelType.Detail:

var detailPage = new DetailPage(viewModel);

return detailPage;

default:

throw new Exception($"Unknown {modelType}");

}

}

}

Note how the MainPage is created differently from the other views.

In order to us it is necessary to update the app.xaml.cs file as follows:

private ViewModelFactory \_viewModelFactory;

private ViewFactory \_viewFactory;

public App()

{

InitializeComponent();

\_viewModelFactory = new ViewModelFactory();

\_viewFactory = new ViewFactory(\_viewModelFactory);

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.Main);

viewModel.Init();

var mainPage = \_viewFactory.Build(TripLogViewModelType.Main, viewModel);

MainPage = new NavigationPage(mainPage);

}

The MainPage.xaml.cs also needs to be updated accordingly:

public partial class MainPage : ContentPage

{

private ViewModelFactory \_viewModelFactory;

private ViewFactory \_viewFactory;

public MainPage(ViewFactory viewFactory, ViewModelFactory viewModelFactory, BaseViewModel viewModel)

{

InitializeComponent();

\_viewModelFactory = viewModelFactory;

\_viewFactory = viewFactory;

BindingContext = viewModel;

}

public void New\_Clicked(object sender, EventArgs e)

{

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.New);

viewModel.Init();

var newEntryPage = \_viewFactory.Build(TripLogViewModelType.New, viewModel);

Navigation.PushAsync(newEntryPage);

}

async void Trips\_ItemTapped(object sender, ItemTappedEventArgs e)

{

var trip = (TripLogEntry)e.Item;

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.Detail);

viewModel.Init(trip);

var detailPage = \_viewFactory.Build(TripLogViewModelType.Detail, viewModel);

await Navigation.PushAsync(detailPage);

// Clear selection

trips.SelectedItem = null;

}

}

At this point the main page completely delegates both the creation of the views and of the models to factories. The next step will be to move both factories to single combined factory.

[Exercise: Add unit tests for the new factory methods. This will lead to a bad issue that could only be fixed by injecting another dependency.....Set the test to Ingore and continue]

Finally a combined factory can be implemented and integrated so as to pack the creation of views and viewModels in a single class. Here a possible implementation:

public class CombinedFactory

{

private readonly ViewFactory \_viewFactory;

private readonly ViewModelFactory \_viewModelFactory;

public CombinedFactory(ViewFactory viewFactory, ViewModelFactory viewModelFactory)

{

\_viewFactory = viewFactory;

\_viewModelFactory = viewModelFactory;

}

public ContentPage Build(TripLogViewModelType modelType)

{

var viewModel = \_viewModelFactory.Build(modelType);

viewModel.Init();

var view = \_viewFactory.Build(modelType, viewModel);

return view;

}

public ContentPage Build(TripLogViewModelType modelType, TripLogEntry entry)

{

var viewModel = \_viewModelFactory.Build(modelType);

viewModel.Init(entry);

var view = \_viewFactory.Build(modelType, viewModel);

return view;

}

}

In order to use it the app.xaml.cs needs to be modified as follows:

public partial class App : Application

{

private ViewModelFactory \_viewModelFactory;

private ViewFactory \_viewFactory;

private CombinedFactory \_combinedFactory;

public App()

{

InitializeComponent();

\_viewModelFactory = new ViewModelFactory();

\_viewFactory = new ViewFactory(\_viewModelFactory);

\_combinedFactory = new CombinedFactory(\_viewFactory, \_viewModelFactory);

var viewModel = \_viewModelFactory.Build(TripLogViewModelType.Main);

viewModel.Init();

var mainPage = new MainPage(viewModel);

mainPage.Init(\_combinedFactory);

MainPage = new NavigationPage(mainPage);

}

Finally the mainPage.xaml.cs can be simplified as follows:

private CombinedFactory \_combinedFactory;

public MainPage(BaseViewModel viewModel)

{

InitializeComponent();

BindingContext = viewModel;

}

public void Init(CombinedFactory combinedFactory)

{

\_combinedFactory = combinedFactory;

}

public void New\_Clicked(object sender, EventArgs e)

{

var newEntryPage = \_combinedFactory.Build(TripLogViewModelType.New);

Navigation.PushAsync(newEntryPage);

}

async void Trips\_ItemTapped(object sender, ItemTappedEventArgs e)

{

var trip = (TripLogEntry)e.Item;

var detailPage = \_combinedFactory.Build(TripLogViewModelType.Detail, trip);

await Navigation.PushAsync(detailPage);

// Clear selection

trips.SelectedItem = null;

}

Rename TripLogViewModelType to ViewType.

Test the application, run the test and check in.

Lesson 4: Introduce a platform-dependent Geografic location service

After clicking the New button the application is supposed to provide the longitude and the latitude of the actual location as default coordinates. Such an implementation is platform-dependent. Therefore the objectives of this lessons are:

- Provide a basic hooking point for the geografic location service in the TripLog project.

- Provide three different implementations for the geografic location service (Android, iOS, UWP).

- Show how these three different implementations can be injected by using a so-called IOC container.

- Augment/update the existing tests by making use of a standard mocking library: Nmock.

Let´s start by defining an interface (to be put inside a new folder Services) as follows:

public interface GeoLocationService

{

Task<GeoCoords> PullCoordinatesAsync();

}

Please note that this reprsents the signature of an asynchronous function.

The GeoCoords class is to be added to the model folder and is intuitively defined as follows:

public class GeoCoords

{

public double Latitude { get; set; }

public double Longitude { get; set; }

}

The service defined above is to be used inside the new entry viewModel class inside the the Init method. In order to be able to do that we are going again to make use of the dependency injection pattern, thus modifying the constructor as follows:

private GeoLocationService \_locationService;

public NewEntryViewModel(GeoLocationService locationService)

{

\_locationService = locationService;

}

Now the service can be used as follows:

public async override void Init()

{

Date = DateTime.Today;

Rating = 1;

var coords = await \_locationService.PullCoordinatesAsync();

Longitude = coords.Longitude;

Latitude = coords.Latitude;

}

In order for the code to build the ViewModelFactory needs to be updated as well so as to take this dependency into account:

public class ViewModelFactory

{

private GeoLocationService \_locationService;

public ViewModelFactory(GeoLocationService locationService)

{

\_locationService = locationService;

}

public BaseViewModel Build(ViewType viewModelType)

{

switch(viewModelType)

{

case ViewType.Detail:

return new DetailViewModel();

case ViewType.Main:

return new MainViewModel();

case ViewType.New:

return new NewEntryViewModel(\_locationService);

default:

throw new Exception($"Unknown {viewModelType}");

}

}

}

The code will still not build because inside the app.xaml.cs an implementation of GeoLocationService is not available, since this is platform specific and has to be injected from the different platforms. The test code will also not build either for the same reason. Before fixing the productive code, it is advantageous to fix the test code by showing how to mock the missing dependency. But first we will make the productive code compilable by setting a placeholder as follows:

public App()

{

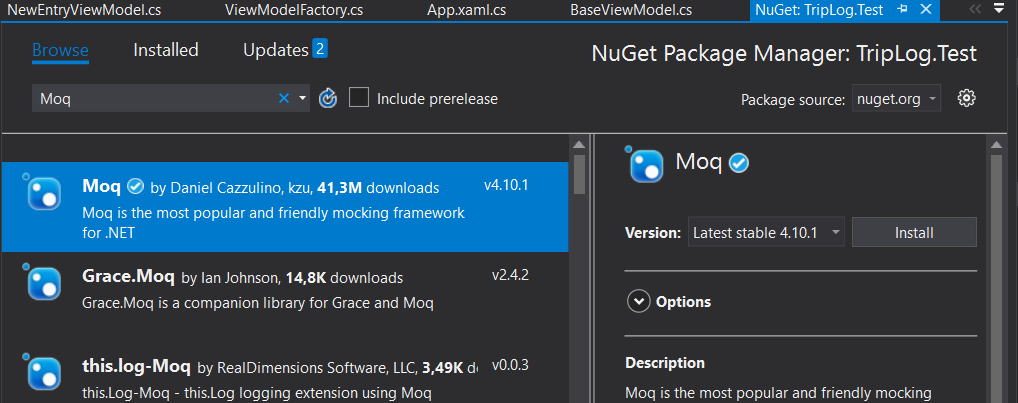
InitializeComponent();

// TODO: FIX THIS ASAP!

\_viewModelFactory = new ViewModelFactory(null);

Now the productive code builds. Let´s fix the test code.

Start by adding a new NuGet package, Moq, to the TripLog.Test project as follows:



[Provide a brief explanation of how mocking works and why it is useful]

Update the broken code with the following statements:

public static Mock<GeoLocationService> \_mockedGeoLocationService = new Mock<GeoLocationService>();

public static ViewModelFactory ViewModelFactory = new ViewModelFactory(\_mockedGeoLocationService.Object);

The meaning of this is simple: The mocking library will create a fake implementation of the service whose only method will return a null when called from inside the factory. This would break our tests. In order to avoid this, the mock object needs to be configured so as to return the value WE want it to return. Start by moving the mock to another file in the test project as follows:

public static class TestInit

{

public static Mock<GeoLocationService> MockedGeoLocationService = new Mock<GeoLocationService>();

public static GeoCoords FakeCoordinates;

static TestInit()

{

Now populate the static constructor with the configuration of the mocked interface as follows:

static TestInit()

{

FakeCoordinates = new GeoCoords();

FakeCoordinates.Latitude = 123;

FakeCoordinates.Longitude = 321;

MockedGeoLocationService.Setup(query => query.PullCoordinatesAsync()).

ReturnsAsync(FakeCoordinates);

}

Essentially, \_mockedGeoLocationService.Object is now equivalent to this:

public class MockedGeoLocationService : GeoLocationService

{

public Task<GeoCoords> PullCoordinatesAsync()

{

return new Task<GeoCoords>(Get);

}

public GeoCoords Get()

{

return FakeCoordinates;

}

}

Update the tests accordingly and ensure they are all green.

Now that the tests are green it is time to proceed with the platform-specific implementations. Start by creating a folder Services on all the platform-specific projects, creating a <Platform>LocationService class inheriting from GeoLocationService as follows:

Android:

public class AndroidLocationService : GeoLocationService

{

public Task<GeoCoords> PullCoordinatesAsync()

{

throw new NotImplementedException();

}

}

iOS:

public class IosLocationService : GeoLocationService

{

public Task<GeoCoords> PullCoordinatesAsync()

{

throw new NotImplementedException();

}

}

UWP:

public class UwpLocationService : GeoLocationService

{

public Task<GeoCoords> PullCoordinatesAsync()

{

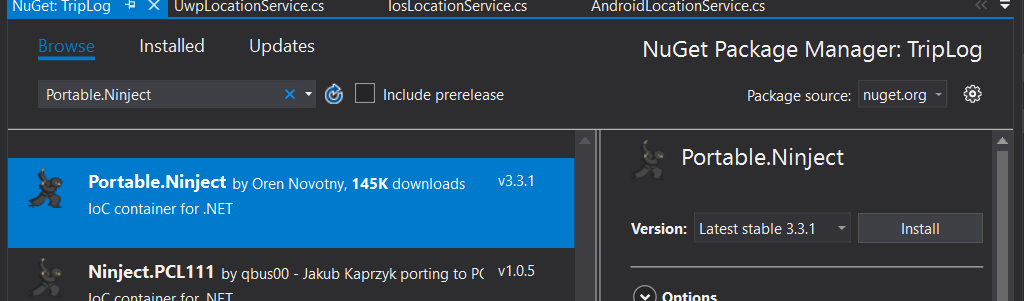
throw new NotImplementedException();

}

}

Now create a folder Modules on each of the platform-specific project.

Now add the following NuGet package to all the 4 projects: Portable.Ninject



Now create a class TripLogPlatformModule : NinjectModule for each of the platform-specific project and populate it as follows:

- Android:

public class TripLogPlatformModule : NinjectModule

{

public override void Load()

{

Bind<GeoLocationService>().

To<AndroidLocationService>().

InSingletonScope();

}

}

-iOS:

public class TripLogPlatformModule : NinjectModule

{

public override void Load()

{

Bind<GeoLocationService>().

To<IosLocationService>().

InSingletonScope();

}

}

-UWP

public class TripLogPlatformModule : NinjectModule

{

public override void Load()

{

Bind<GeoLocationService>().

To<UwpLocationService>().

InSingletonScope();

}

}

In order to implement the IOC pattern with Ninject it is now necessary to implement and configure a so-called Kernel. This represent the IOC container. So update the TripLog application´s main app by introducing dependency injection in the constructor as follows:

private ViewModelFactory \_viewModelFactory;

private ViewFactory \_viewFactory;

private CombinedFactory \_combinedFactory;

private IKernel \_kernel;

public App(params NinjectModule[] platformModules)

{

InitializeComponent();

\_kernel = new StandardKernel();

\_kernel.Load(platformModules);

var locationService = \_kernel.Get<GeoLocationService>();

Now the previously set placeholder can be removed and correct dependency injected:

\_viewModelFactory = new ViewModelFactory(locationService);

In order to finalize the injection it is now necessary to update the platform-specific implementations of the App class by passing the specific GeoLocationService implementation as dependency as follows:

- Android:

LoadApplication(new App(new TripLogPlatformModule()));

-iOS:

LoadApplication(new App(new TripLogPlatformModule()));

-UWP:

LoadApplication(new TripLog.App(new TripLogPlatformModule()));

In order to complete this section it is now necessary to provide the three platform-specific implementation of the GeoLocationService. The minimal implementations that we need are readily available on the Internet. Here are some proposals:

-Android:

public class AndroidLocationService : Java.Lang.Object, GeoLocationService, ILocationListener

{

private TaskCompletionSource<Location> \_tcs;

public async Task<GeoCoords> PullCoordinatesAsync()

{

try

{

var locationManager = (LocationManager)Application.Context.GetSystemService(Context.LocationService);

\_tcs = new TaskCompletionSource<Location>();

locationManager.RequestSingleUpdate(LocationManager.GpsProvider, this, null);

var location = await \_tcs.Task;

var result = new GeoCoords();

result.Latitude = location.Latitude;

result.Longitude = location.Longitude;

return result;

}

catch (Exception e)

{

return new GeoCoords();

}

}

public void OnLocationChanged(Location location)

{

\_tcs.TrySetResult(location);

}

public void OnProviderDisabled(string provider)

{

}

public void OnProviderEnabled(string provider)

{

}

public void OnStatusChanged(string provider, [GeneratedEnum] Availability status, Bundle extras)

{

}

}

-iOS:

public class IosLocationService : GeoLocationService

{

private TaskCompletionSource<CLLocation> \_locationTaskCompletion;

public async Task<GeoCoords> PullCoordinatesAsync()

{

var locationManager = new CLLocationManager();

\_locationTaskCompletion = new TaskCompletionSource<CLLocation>();

if (UIDevice.CurrentDevice.CheckSystemVersion(8, 0))

{

locationManager.RequestWhenInUseAuthorization();

}

locationManager.LocationsUpdated += OnLocationUpdated;

locationManager.StartUpdatingLocation();

var location = await \_locationTaskCompletion.Task;

var result = new GeoCoords();

result.Latitude = location.Coordinate.Latitude;

result.Longitude = location.Coordinate.Longitude;

return result;

}

private void OnLocationUpdated(object sender, CLLocationsUpdatedEventArgs e)

{

\_locationTaskCompletion.TrySetResult(e.Locations[0]);

}

}

-UWP:

public class UwpLocationService : GeoLocationService

{

public async Task<GeoCoords> PullCoordinatesAsync()

{

var locator = new Geolocator();

var coordinates = await locator.GetGeopositionAsync();

GeoCoords result = new GeoCoords();

result.Latitude = coordinates.Coordinate.Point.Position.Latitude;

result.Longitude = coordinates.Coordinate.Point.Position.Longitude;

return result;

}

}

Lesson 5: Introducing a data layer in the form of a deployable web API.

[Introduce the topic and the motivations...]

[Briefly describe what a web API is and why they are so fashonable nowadays...]

Now that the structure of the application is in place and real data is acquired every time the new button is pushed, the time has come to introduce a persistency layer. There are several options and possibilities both in terms of custom implementation and by making use of cloud providers such as MS and Google.

The preferred approach is to build a web API hosted on IIS leveraging on an open souce data layer.

Objectives of this lesson are:

- Definition, implementation and deployment on the development machine´s IIS of a simple ASP-based web API making use of a standard persistency layer.

- Automated testing of this newly introduced data layer and its integration in the build.

- Integration of the data layer access in the TripLog application by means of a standard HTTP client.

[Provide an overview of the architecture with a deployment diagram]

Let´s start from the end by defining an interface expressing the needs of the TripLog application. The minimal needs from the application standpoint are:

- The possibility to persist a newly created entry.

- The posibility to retrieve all the entries created.

These requirements can be captured by the following interface (to be created in the Services folder):

public interface TripLogDataService

{

Task<IList<TripLogEntry>> ReadAllEntriesAsync();

Task AddEntryAsync(TripLogEntry entry);

}

The pages requiring access to such interface are:

- The main page right after the application is started in order to retrieve the already existing entries.

- The new page in order to persist a freshly created entry.

Let´s start by updating the constructor of MainViewModel so as accept such interface:

private TripLogDataService \_tripLogDataService;

public MainViewModel(TripLogDataService tripLogDataService)

{

\_tripLogDataService = tripLogDataService;

}

And then by updating the Init method so as to call the newly defined service:

LogEntries = new ObservableCollection<TripLogEntry>(

await \_tripLogDataService.ReadAllEntriesAsync());

(For the time being we will leave the hard-coded entries...)

In a similar way let´s augment the constructor of the NewEntryViewModel:

private GeoLocationService \_locationService;

private TripLogDataService \_tripLogDataService;

public NewEntryViewModel(GeoLocationService locationService, TripLogDataService tripLogDataService)

{

\_locationService = locationService;

\_tripLogDataService = tripLogDataService;

}

and update the ExecuteSaveCommand method so as to take the new service into account:

private async void ExecuteSaveCommand(object obj)

{

var newEntry = new TripLogEntry

{

Title = Title,

Notes = Notes,

Rating = Rating,

Date = Date,

Latitude = Latitude,

Longitude = Longitude

};

// Save this new entry later: Introduce a data layer.

await \_tripLogDataService.AddEntryAsync(newEntry);

}

The factories need at this point to be updated accordingly. Start by updating the constructor of the ViewModelFactory:

private GeoLocationService \_locationService;

private TripLogDataService \_tripLogDataSevice;

public ViewModelFactory(GeoLocationService locationService, TripLogDataService tripLogDataSevice)

{

\_locationService = locationService;

\_tripLogDataSevice = tripLogDataSevice;

}

and then the Build method:

public BaseViewModel Build(ViewType viewModelType)

{

switch(viewModelType)

{

case ViewType.Detail:

return new DetailViewModel();

case ViewType.Main:

return new MainViewModel(\_tripLogDataSevice);

case ViewType.New:

return new NewEntryViewModel(\_locationService, \_tripLogDataSevice);

default:

throw new Exception($"Unknown {viewModelType}");

}

}

The productive code ist still broken. Let´s put a placeholder for the time being as follows and jump on to the test code. Here the placeholder:

// TODO: Provide an implementation

\_viewModelFactory = new ViewModelFactory(locationService, null);

Due to the introduction of this new dependency the tests will need to be updated. We can follow the same pattern we introduced before by making use of the Moq library. Here a possible implementation:

public static Mock<GeoLocationService> MockedGeoLocationService = new Mock<GeoLocationService>();

public static Mock<TripLogDataService> MockedTripLogDataService = new Mock<TripLogDataService>();

public static ViewModelFactory ViewModelFactory = new ViewModelFactory(MockedGeoLocationService.Object, MockedTripLogDataService.Object);

And here the configuration of the MockedTripLogDataService:

public static Mock<GeoLocationService> MockedGeoLocationService = new Mock<GeoLocationService>();

public static Mock<TripLogDataService> MockedTripLogDataService = new Mock<TripLogDataService>();

public static ViewModelFactory ViewModelFactory = new ViewModelFactory(MockedGeoLocationService.Object, MockedTripLogDataService.Object);

public static ViewFactory ViewFactory = new ViewFactory(ViewModelFactory);

public static GeoCoords FakeCoordinates;

public static IList<TripLogEntry> FakeTripLogEntryCollection;

static TestInit()

{

FakeCoordinates = new GeoCoords();

FakeCoordinates.Latitude = 123;

FakeCoordinates.Longitude = 321;

FakeTripLogEntryCollection = new List<TripLogEntry>() { new TripLogEntry() };

MockedTripLogDataService.Setup(query => query.ReadAllEntriesAsync()).

ReturnsAsync(FakeTripLogEntryCollection);

MockedGeoLocationService.Setup(query => query.PullCoordinatesAsync()).

ReturnsAsync(FakeCoordinates);

}

Update all the tests so that they are green (one change required...).

Now that all the tests are green it is possible to provide a first stubbed implementation of the TripLogService so as to make the application buildable and runnable:

public class RestTripLogDataService : TripLogDataService

{

public Task AddEntryAsync(TripLogEntry entry)

{

var task = new Task(GetAdd);

task.Start();

return task;

}

private void GetAdd()

{

}

public Task<IList<TripLogEntry>> ReadAllEntriesAsync()

{

var task = new Task<IList<TripLogEntry>>(GetRead);

task.Start();

return task;

}

private IList<TripLogEntry> GetRead()

{

return new List<TripLogEntry>();

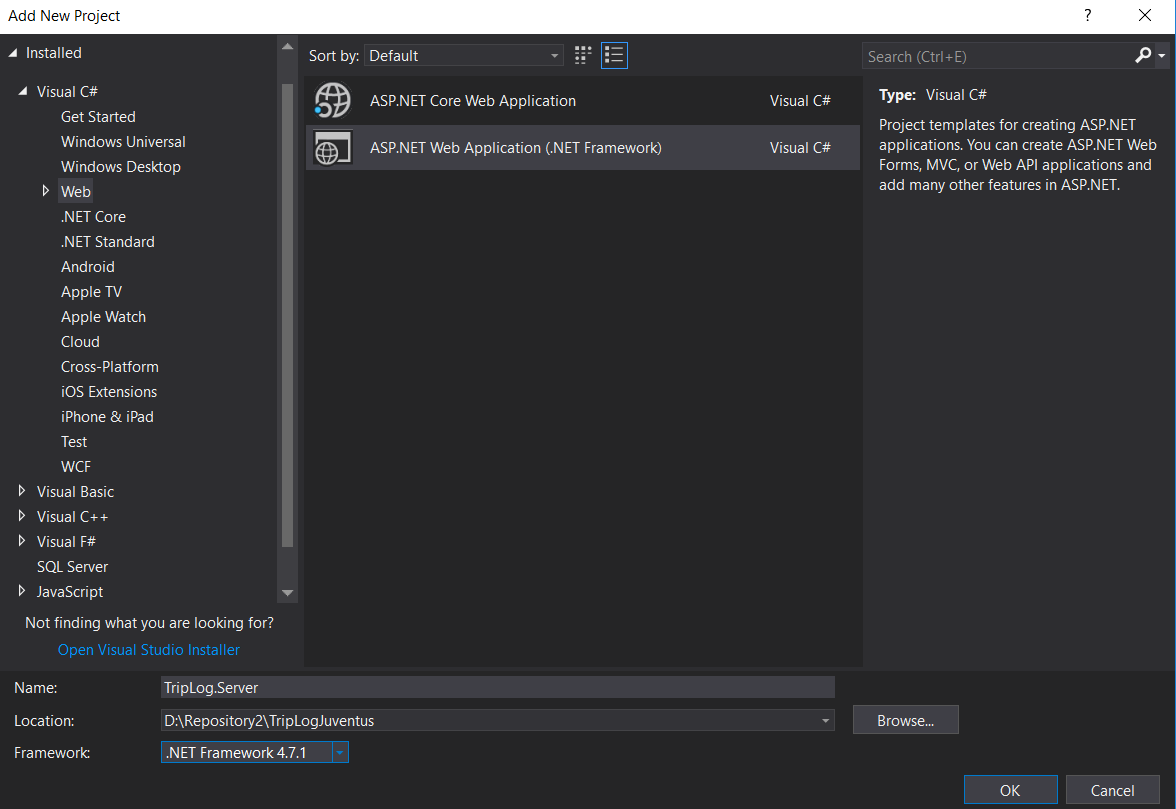
}

}

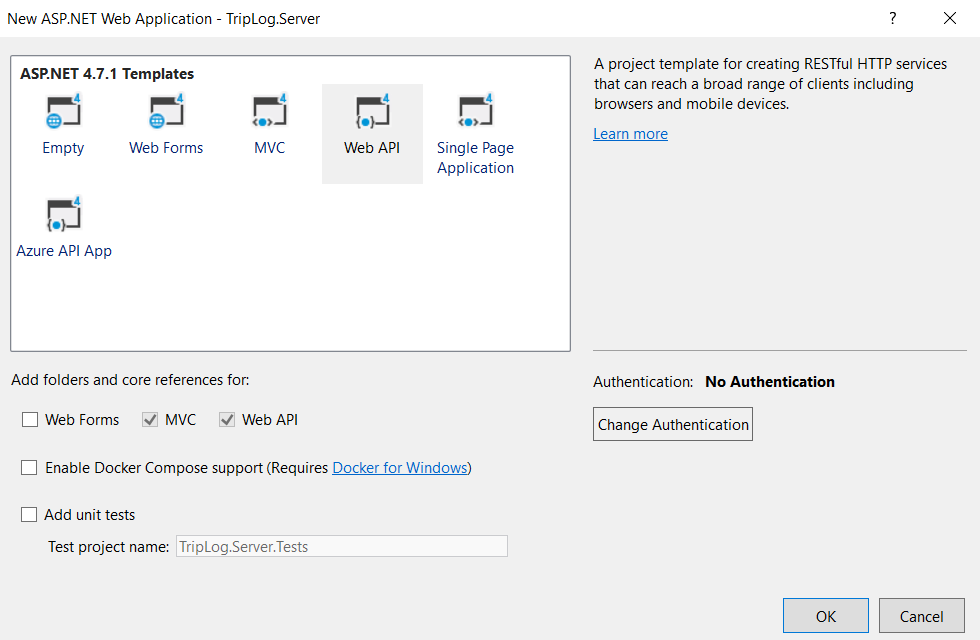
Please pay attention to the name of the class: It is a technological specialization of the name of the interface it derives from (Clean code!).

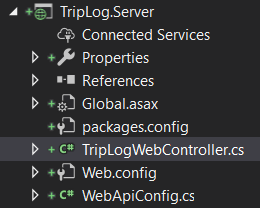
Debug the application and ensure that the newly added code does not alter the behaviour. Check in and ensure that the build succeeds.

Now it is time to move to the backend and implement simple Web API. Start by adding an ASP .NET Core project to be named TripLog.Server:



On the next screen choose Web API and click OK.



This will create a large number of files and folders. Remove and rename as shown on the following

picture:

Ensure that the TripLogWebController has the following format:

public class TripLogWebController : ApiController

{

// GET api/TripLogWeb

public IEnumerable<string> Get()

{

return new string[] { "value1", "value2" };

}

// GET api/TripLogWeb/5

public string Get(int id)

{

return "value";

}

// POST api/TripLogWeb

public void Post([FromBody]string value)

{

}

// PUT api/TripLogWeb/5

public void Put(int id, [FromBody]string value)

{

}

// DELETE api/TripLogWeb/5

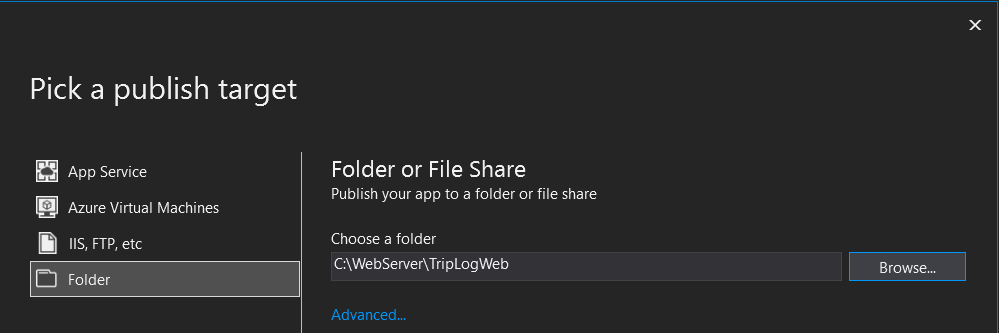
public void Delete(int id)

{

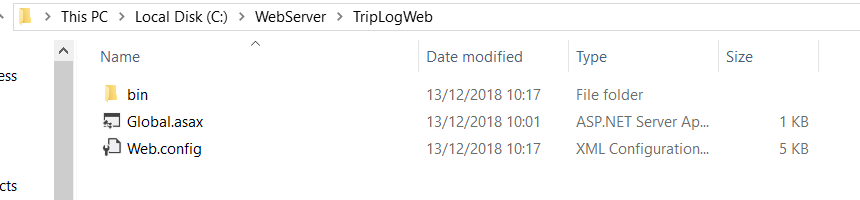
}

}

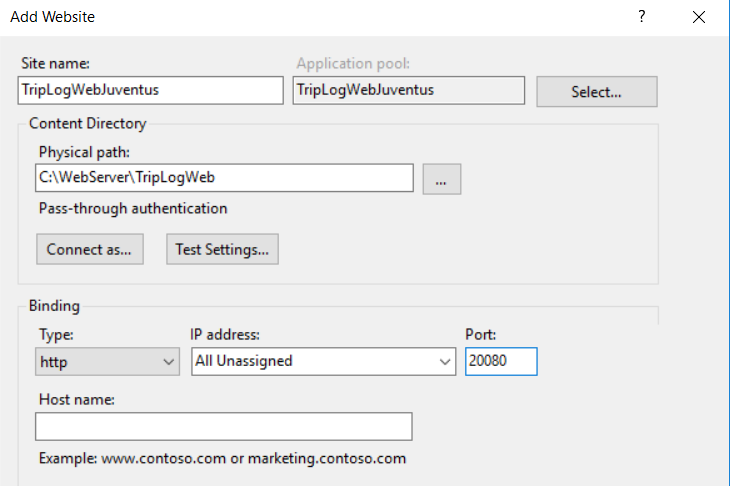
Build the project. Now select Publish from the drop-down menu and select a folder where the Web API should be published:



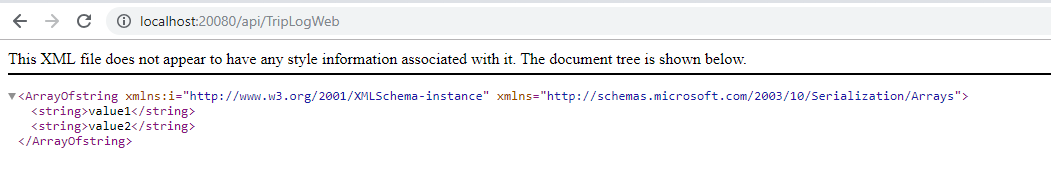
Select create Profile and configure the profile so as to delete all the files every time the software is published. Click Publish. This should be the end result:



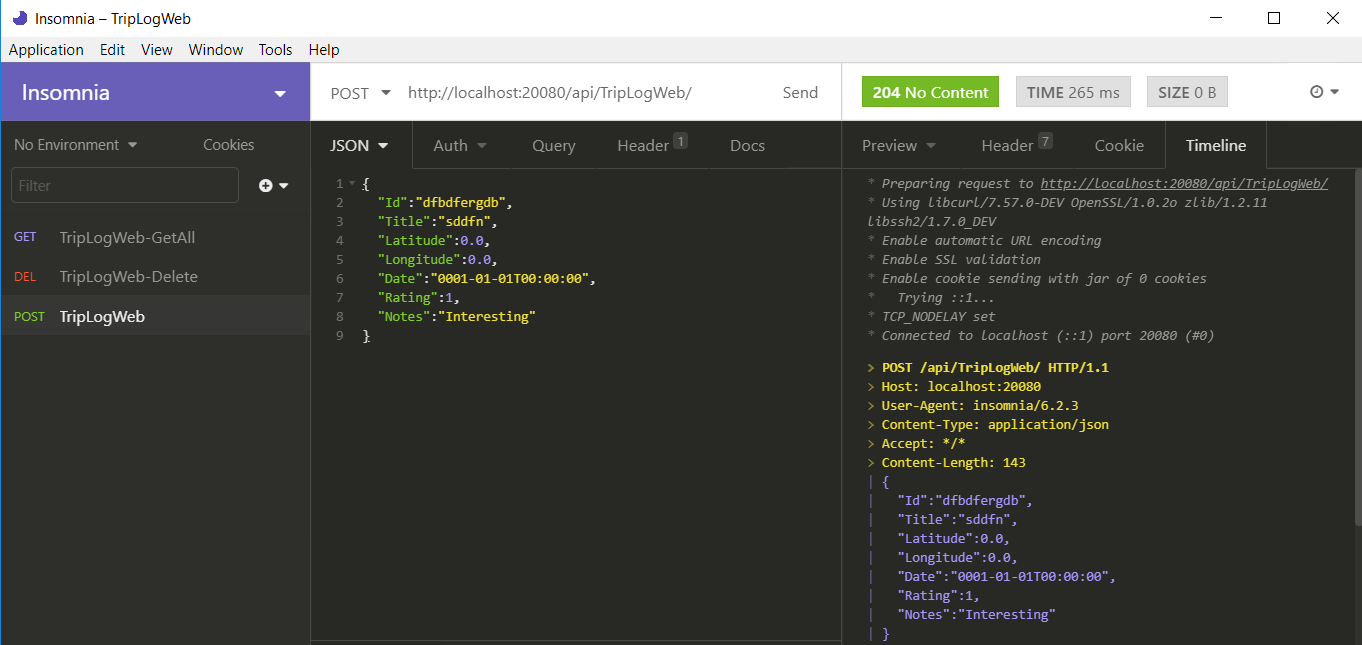
Now open IIS, create a new site and choose the options according to the folder created above:



Open a browser, browse to localhost:20080/api/TripLogWeb. This should be the result:

This means that the site is running and responding to a Get. Debugging the the project leads to the same result. Check in the code.

In order to check the other web methods it is already possible to use a Rest client capable of sending Posts. One such possible option would be Insomnia. Here a screenshot:

Now it is time to specialize the Web API for our client application. The Get and Post API´s will be the hooking points for our TripLogDataService and need to be updated accordingly. Start by providing an empty implementation:

// GET api/TripLogWeb

public IEnumerable<TripLogEntry> Get()

{

return new TripLogEntry[] { new TripLogEntry(), new TripLogEntry() };

}

// POST api/TripLogWeb

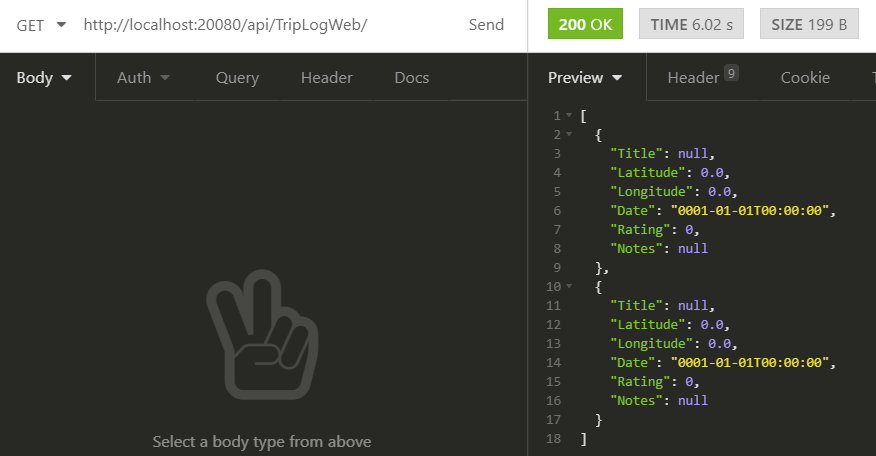
public void Post([FromBody]TripLogEntry value)

{

// Add entry to the persistency layer...

}

Publish the site and test them. Here the expected result:



At this point it is necessary to implement the persistency layer. The options are unlimited, ranging from standard relational DB, to NoSQL DB, through flat file.

Dbreeze is an open source file-based key-value database implementation that can be used to implement the backend quickly.

Therefore add the corresponding NuGet library to the server project. Then define an interface:

public interface TripLogPersistency : IDisposable

{

IEnumerable<TripLogEntry> GetAll();

void Add(TripLogEntry value);

}

The reason why this needs to inherit IDisposable will be given later.

And the corresponding empty implementation:

public class DbreezeTripLogPersistency : TripLogPersistency

{

public void Add(TripLogEntry value)

{

}

public IEnumerable<TripLogEntry> GetAll()

{

return new TripLogEntry[] { new TripLogEntry(), new TripLogEntry() };

}

public void Dispose()

{

}

}

Add the following code in the controller to establish the integration:

private TripLogPersistency \_persistency;

public TripLogWebController()

{

\_persistency = new DbreezeTripLogPersistency();

}

// GET api/TripLogWeb

public IEnumerable<TripLogEntry> Get()

{

return \_persistency.GetAll();

}

// POST api/TripLogWeb

public void Post([FromBody]TripLogEntry value)

{

\_persistency.Add(value);

}

Test the API and ensure that the behaviour is unchanged.

Now create a new test project called TripLog.Server.Test and name the generated file PersistencyTests.cs.

Add the following two dummy tests:

[TestMethod]

public void AddElementToPersistencyTest()

{

var persistency = new DbreezeTripLogPersistency();

persistency.Add(new TripLogEntry());

}

[TestMethod]

public void GetAllElementsFromPersistencyTest()

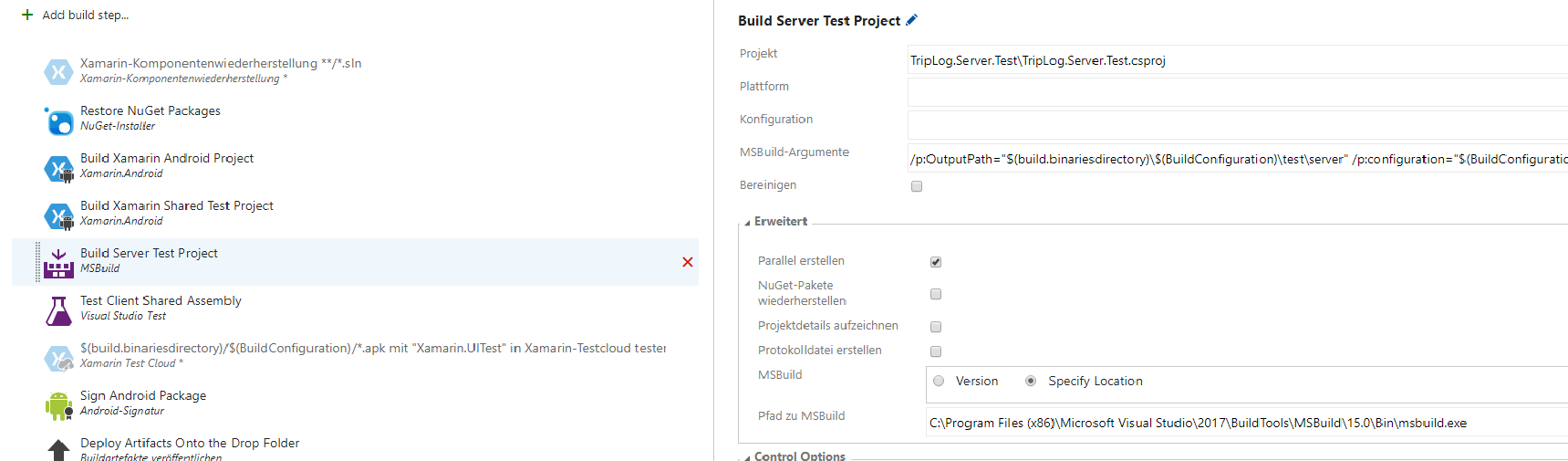
{

var persistency = new DbreezeTripLogPersistency();

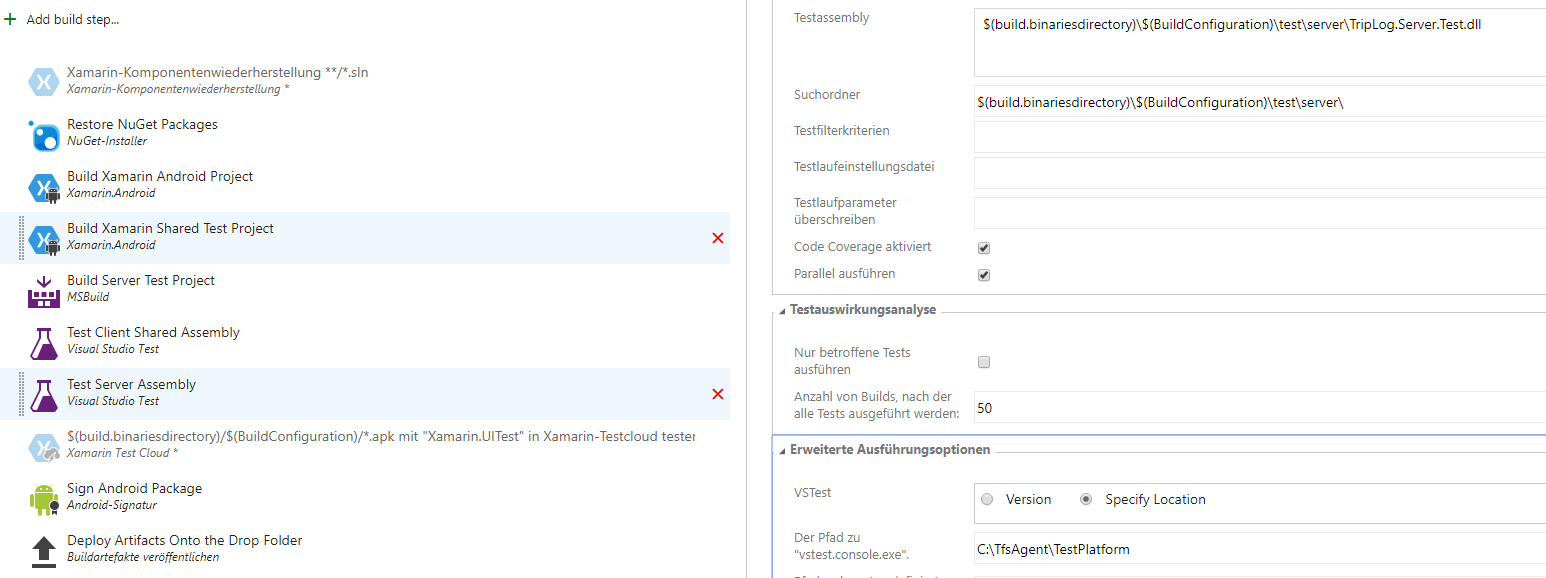
var allEntries = persistency.GetAll();

}

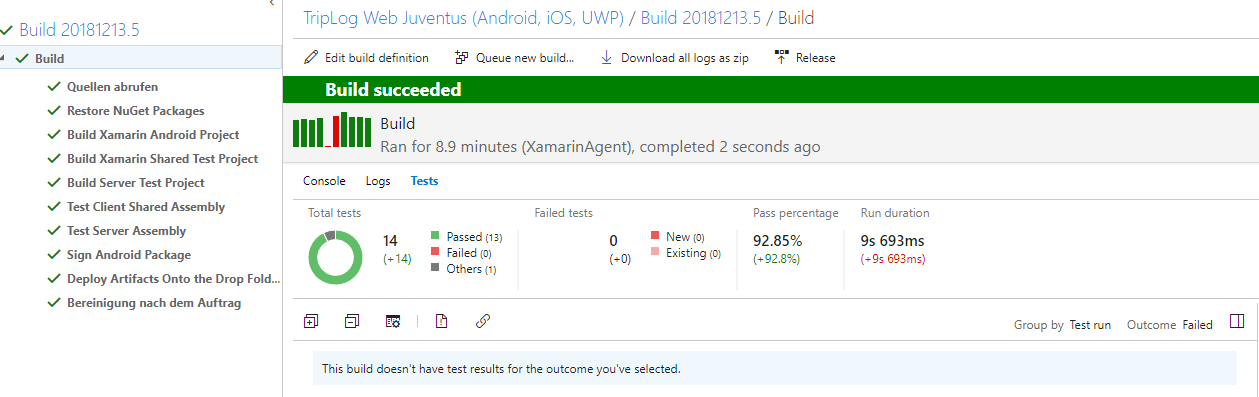
These are only placeholders and are only required to update the build with additional steps. So let´s move back to the build server and add two new build steps. First add a build task to build the newly created server project as follows:



The add a step to test it:



Enqueue a build and ensure that it completes and that all tests are executed succesfully:



Two tasks are now pending:

- A real implementation of the TripLogService on the client side.

- A real implementation of the TripLogPersistency on the server side.

Lesson 6: Implementation of the two interfaces and relative integration testing.

6.1: Implementation of TripLogPersistency

Being the DbreezeEngine a file-based data layer, its constructor requires a directory which will then be containing the db´s files. The constructor will need to be implemented accordingly:

private DBreezeEngine \_db;

protected DirectoryInfo \_dbDirectory;

public DbreezeTripLogPersistency(DirectoryInfo directory)

{

\_dbDirectory = directory;

\_db = new DbreezeEngine(directory.FullName);

\_db.Setup();

}

The instantiation of the class needs to updated accordingly.

\_persistency = new DbreezeTripLogPersistency(new DirectoryInfo(@"C:\WebServer\Persistency"));

(The hard-coded path will be removed later)

The tests also need to updated accordingly. Now since these tests are executed every time inside a different folder, some questions arise:

1) Where should the db be created?

2) Should the db be recreated for every test?

[Definition of integration test]

These are some the standard questions that need to be answered every time integration tests are considered because in principle every integration test alters the status of the underlying infrastructure, which, in turns, defines the initial conditions for the execution of the other tests.

There are no standard answers for the questions above without knowing the particular conditions of the system. For the purpose of our applications each test will create a fresh db and keep it consistent programmatically.

This could be accomplished with the following code:

private string \_dbSubFolder = "DbTemp";

[TestMethod]

public void AddElementToPersistencyTest()

{

var dbFolder = new DirectoryInfo(Path.Combine(Directory.GetCurrentDirectory() + \_dbSubFolder));

Directory.CreateDirectory(dbFolder.FullName);

var persistency = new DbreezeTripLogPersistency(dbFolder);

persistency.Add(new TripLogEntry());

}

[TestMethod]

public void GetAllElementsFromPersistencyTest()

{

var dbFolder = new DirectoryInfo(Path.Combine(Directory.GetCurrentDirectory() + \_dbSubFolder));

Directory.CreateDirectory(dbFolder.FullName);

var persistency = new DbreezeTripLogPersistency(dbFolder);

var allEntries = persistency.GetAll();

}

Running these two tests separately works, whereas, unsurprisingly, running them in a single batch fails partially. This happens because the execution of the second test interferes with some undisposed resources left by the first test. By checking the documentation it is possible to find a Dispose method which can be integrated:

public void Dispose()

{

if (\_db != null)

{

\_db.Dispose();

}

}

Adding a call to Dispose() at the end of each test solves the problem.

The tests clerly need to be refactored in order to reduce as much repetition as possible. Start by creating a setup method in the class DbreezeTripLogPersistency:

public void Setup()

{

if (!\_dbDirectory.Exists)

{

\_dbDirectory.Create();

}

}

and refactor the test accordingly by introducing two new constructs:

- TestInitialize

- TestCleanup

These two methods are called respectively before and after EACH TestMethod:

[TestInitialize]

public void Setup()

{

\_dbFolder = new DirectoryInfo(Path.Combine(Directory.GetCurrentDirectory(), \_dbSubFolder));

\_db = new DbreezeTripLogPersistency(\_dbFolder);

\_db.Setup();

}

[TestCleanup]

public void ShutDown()

{

if (\_db != null)

{

\_db.Dispose();

}

}

This is a typical approach for integration tests: before each test is execute the contest has to be reset and after each test a clean up needs to be performed. Now the tests can be refactored as follows:

[TestMethod]

public void AddElementToPersistencyTest()

{

\_db.Add(new TripLogEntry());

}

[TestMethod]

public void GetAllElementsFromPersistencyTest()

{

var allEntries = \_db.GetAll();

}

These are still dummy tests, but the acquired knowledge allow us now to update the productive code as follows for the interface:

public interface TripLogPersistency : IDisposable

{

void Setup();

IEnumerable<TripLogEntry> GetAll();

void Add(TripLogEntry value);

}

(This is not the best possible solution and is slight, but perceptible, violation of the ISP...)

and the controller:

public TripLogWebController()

{

\_persistency = new DbreezeTripLogPersistency(new DirectoryInfo(@"C:\WebServer\Persistency"));

\_persistency.Setup();

}

// GET api/TripLogWeb

public IEnumerable<TripLogEntry> Get()

{

var results = \_persistency.GetAll();

\_persistency.Dispose();

return results;

}

// POST api/TripLogWeb

public void Post([FromBody]TripLogEntry value)

{

\_persistency.Add(value);

\_persistency.Dispose();

}

Debugging the API should show not behavioural change. Check in the code and ensure that the build is successful.

Now it is time to implement the real methods. Add can be implemented as follows:

using (var transaction = \_db.GetTransaction())

{

transaction.Insert(\_tableName, value.Id, TripLogEntry.Serialize(value));

transaction.Commit();

}

Being the db a key-value map which only supports basic types(!!!!!), the key needs to be defined in such a way so as to be unique. This can be done conveniently in the TripLogEntry´s constructor with Guid as follows:

public TripLogEntry()

{

Title = string.Empty;

Id = Guid.NewGuid().ToString();

Notes = string.Empty;

}

public TripLogEntry(string title) : this()

{

Title = title;

}

In order to convert an instance of TripLogEntry to a string a possible solution is to use serialization such as the one provided by Json:

public static string Serialize(TripLogEntry entry)

{

return JsonConvert.SerializeObject(entry);

}

Objects read from the persistency will then need to be deserialized:

public static TripLogEntry Deserialize(string serializedTripLogEntry)

{

var entry = JsonConvert.DeserializeObject<TripLogEntry>(serializedTripLogEntry);

TripLogEntry result = new TripLogEntry();

result.Id = entry.Id;

result.Title = entry.Title;

result.Latitude = entry.Latitude;

result.Longitude = entry.Longitude;

result.Date = entry.Date;

result.Rating = entry.Rating;

result.Notes = entry.Notes;

return result;

}

Now the following tests can be executed. However, their outcome is inconsistent because entries created in a test would affect other tests:

[TestMethod]

public void AddElementsToPersistencyTest()

{

var title1 = "Number1";

var title2 = "Number2";

\_db.Add(new TripLogEntry(title1));

\_db.Add(new TripLogEntry(title2));

var allEntries = \_db.GetAll();

Assert.AreEqual(2, allEntries.Count());

Assert.AreEqual(1, allEntries.Count(elem => elem.Title.Equals(title1)));

Assert.AreEqual(1, allEntries.Count(elem => elem.Title.Equals(title2)));

}

[TestMethod]

public void AddSameElementTwiceToPersistencyTest()

{

var title1 = "Number1";

\_db.Add(new TripLogEntry(title1));

\_db.Add(new TripLogEntry(title1));

var allEntries = \_db.GetAll();

Assert.AreEqual(1, allEntries.Count());

Assert.AreEqual(1, allEntries.Count(elem => elem.Title.Equals(title1)));

}

[TestMethod]

public void GetAllElementsFromPersistencyTest()

{

var allEntries = \_db.GetAll();

Assert.AreEqual(0, allEntries.Count());

}

So in order to make the tests repeatable and consistent there are two options:

1) Remove the Db altogether after every test.

2) Add a RemoveAll method to the class.

Option number 1 is the nuclear solution and and normally should be avoided. Option number 2 would mean adding a method in the productive code that is not strictly required. This would be also wrong. The clean way is to subclass main db class and put the derivative in the test code, where an additional method can be confortably be added. Here a possible implementation:

public class ExtendedDbreezeTripLogPersistency : DbreezeTripLogPersistency

{

public ExtendedDbreezeTripLogPersistency(DirectoryInfo directory) : base(directory)

{

}

public void RemoveAll()

{

using (var transaction = \_db.GetTransaction())

{

transaction.RemoveAllKeys(\_tableName, true);

transaction.Commit();

}

}

}

Now updating the the tests as follows makes the tests runnable in every scenario:

[TestInitialize]

public void Setup()

{

\_dbFolder = new DirectoryInfo(Path.Combine(Directory.GetCurrentDirectory(), \_dbSubFolder));

\_db = new ExtendedDbreezeTripLogPersistency(\_dbFolder);

\_db.Setup();

\_db.RemoveAll();

}

[TestCleanup]

public void ShutDown()

{

if (\_db != null)

{

\_db.RemoveAll();

\_db.Dispose();

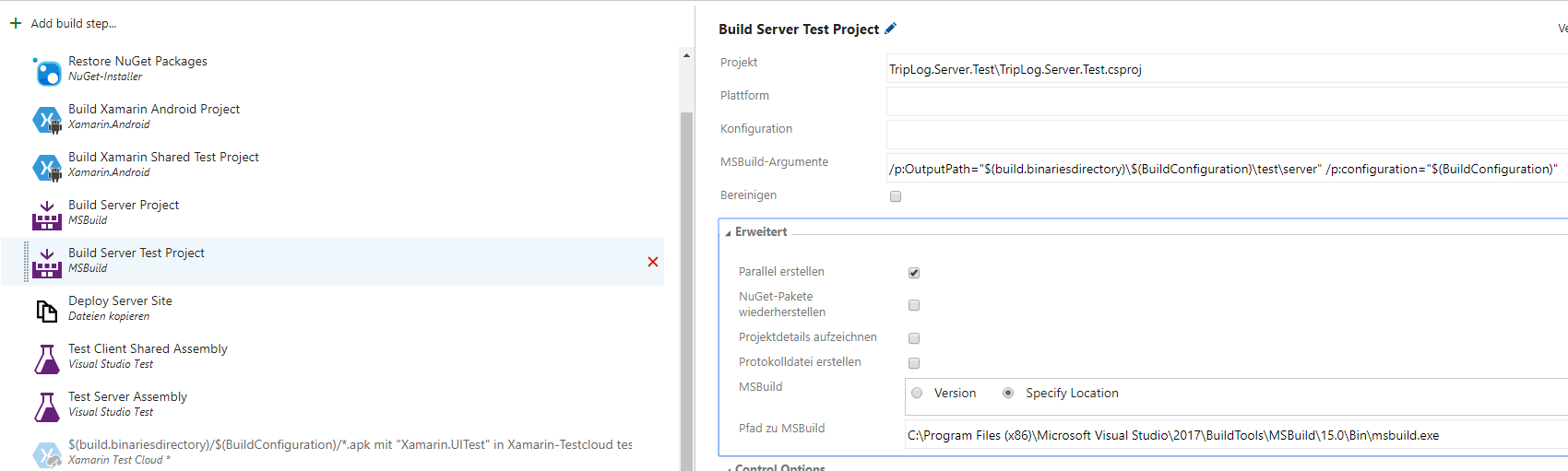
}

}

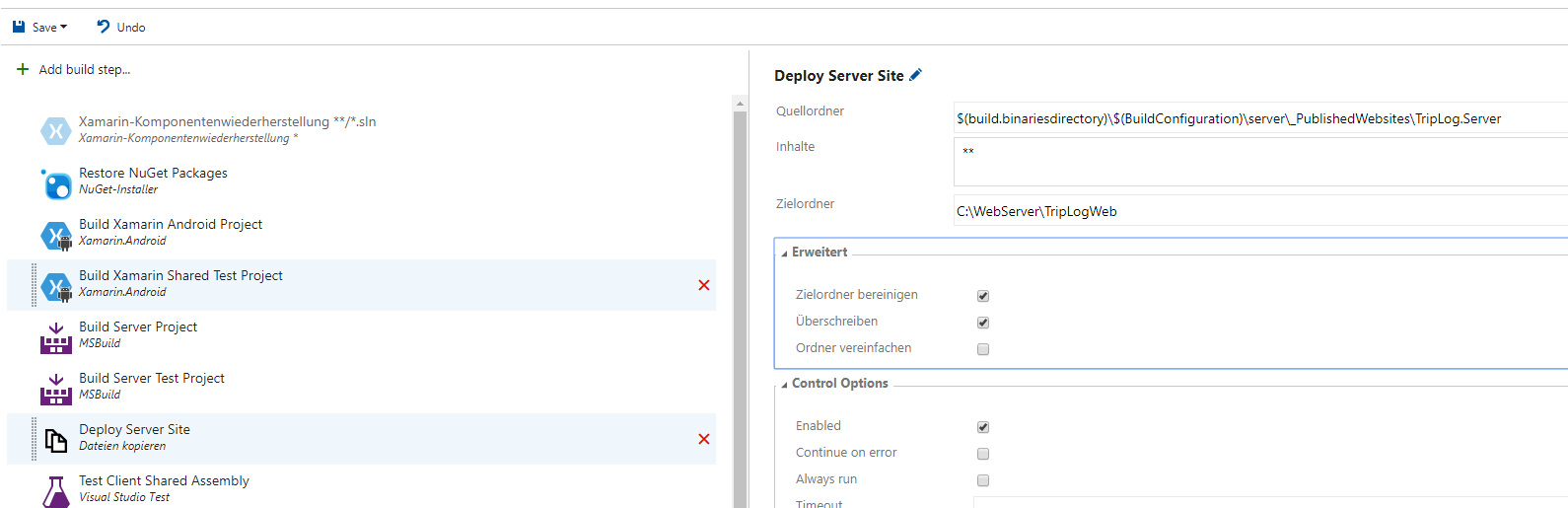
The backend part is implemented and finalized. Debugging the site and using Insomnia will display the right results.

However, in order for the build to work correctly, it is still necessary to a add build steps building and deploying the web site onto the right folder (hard-coded at the moment).

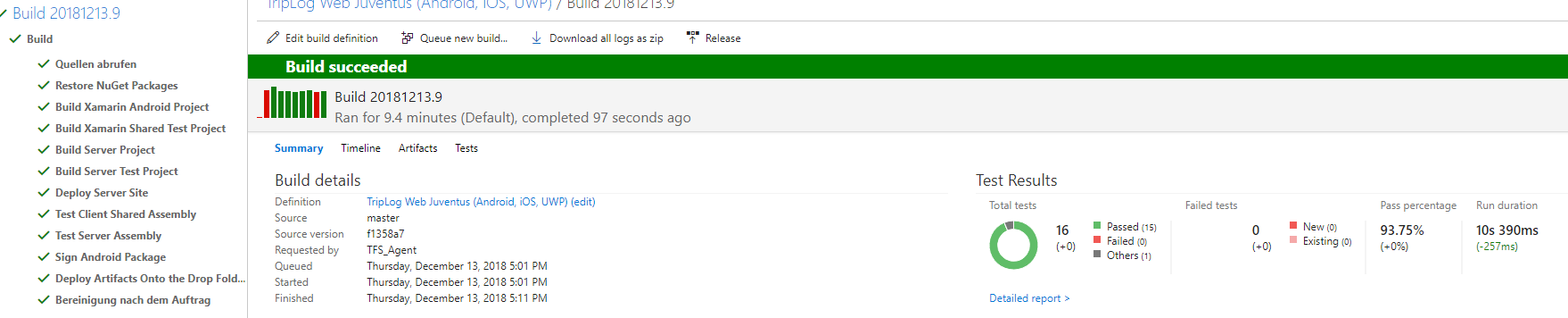
First we need to add a build step to build the server software:



Then we need to deploy the software onto the target folder:



Create a similar site on the build server, check in and ensure that the build runs succesfully.



6.2: Implementation of TripLogService

We decided to implement the interface TripLogService with a REST-based client. This in turn should be a specialized implementation of an http client. Now it is not required to write such a client from scratch since standard implementations are available on the network and can be downloaded quickly. However, this implementation has to be asynchronous. Here the file:



Please add it to your core project under the Services folder and ensure you understand its meaning.

It is now possible to implement the RestTripLogService, by providing standard client above and the Url of the Web API as arguments to the constructor as follows:

private StandardAsyncHttpClient \_httpClient;

protected readonly Uri \_baseUri;

protected readonly IDictionary<string, string> \_headers;

public RestTripLogDataService(StandardAsyncHttpClient httpClient, Uri baseUri)

{

\_httpClient = httpClient;

\_baseUri = baseUri;

\_headers = new Dictionary<string, string>();

}

The implementation of the methods is straightforward:

public async Task AddEntryAsync(TripLogEntry entry)

{

var response = await \_httpClient.SendRequestAsync<TripLogEntry>(\_baseUri, HttpMethod.Post, \_headers, entry);

}

public async Task<IList<TripLogEntry>> ReadAllEntriesAsync()

{

var response = await \_httpClient.SendRequestAsync<TripLogEntry[]>(\_baseUri, HttpMethod.Get, \_headers);

return response;

}

To finalize the integration of the client some additional arrangements need to be taken care of in the aap.xaml.cs so in order to take these dependencies into account:

var httpClient = new StandardAsyncHttpClient();

var backendUri = new Uri("http://192.168.56.10:20080/api/TripLogWeb/");

var restTripLogDataService = new RestTripLogDataService(httpClient, backendUri);

The Uri is at the moment hard-coded to the IP of a test server on which we deployed the web API. This additional limitation will be removed later.

Moreover it must be noted that no security measures are being taken at the moment: The traffic flows unencrypted.

Now that the implementation is completed to some degree the question is whether and to what extent the whole integration can be tested. Well, what can be done immediately is to test the web API through the client. Start by creating a new Test project to be named TripLog.Test.Acceptance and update the setup method intuitively as follows:

private RestTripLogDataService \_client;

private Uri \_url;

[TestInitialize]

public void Setup()

{

\_url = new Uri("http://localhost:20080/api/TripLogWeb/");

var httpClient = new StandardAsyncHttpClient();

\_client = new RestTripLogDataService(httpClient, \_url);

}

The localhost will ensure that the tests can be run any server, provided that the web API is deployed there.

As we did for testing the persistency, add a test method sending a single POST:

[TestMethod]

public async Task SyncSimpleInsertRetrieve()

{

await \_client.AddEntryAsync(First);

var retrieved = await \_client.ReadAllEntriesAsync();

Assert.AreEqual(1, retrieved.Count);

Assert.AreEqual(First, retrieved.First());

}

Running this test might work or not depending on the status of the backend. This means again that the outcome of the test depends on environmental conditions. As before we can overcome this issue by deriving from the client class and adding the missing/required methods to make the tests run. However, in this case a new consistent version of the web API would also need to be required so as to support deletion, but this would mean deploying software capable of removing all the entries from the database!

The issue can be overcome by introducing configuration parameters and a backend factory so as to support the deployment of the software configured differently for different environments.

Let´s start by finalizing the tests as follows:

[TestInitialize]

public async Task Setup()

{

\_url = new Uri("http://localhost:20080/api/TripLogWeb/");

var httpClient = new StandardAsyncHttpClient();

\_client = new ExtendedRestTripLogDataService(httpClient, \_url);

await SetBaseLine();

}

[TestCleanup]

public async Task ShutDown()

{

await SetBaseLine();

}

private async Task SetBaseLine()

{

if (\_client != null)

{

await \_client.RemoveAll();

}

}

and defining the extended class intuitively as follows:

public class ExtendedRestTripLogDataService : RestTripLogDataService

{

public ExtendedRestTripLogDataService(StandardAsyncHttpClient httpClient, Uri baseUri)

: base(httpClient, baseUri)

{ }

public async Task RemoveAll()

{

var response = await \_httpClient.SendRequestAsync<TripLogEntry>(\_baseUri, HttpMethod.Delete, \_headers);

}

}

The backend web API needs to be modified as follows:

private TripLogPersistency \_persistency;

private Environment \_environment;

public TripLogWebController()

{

\_environment = Environment.Test;

\_persistency = new TripLogPersistencyBuilder(new DirectoryInfo(@"C:\WebServer\Persistency")).

Build(\_environment);

\_persistency.Setup();

}

public void Delete()

{

if (\_environment == Environment.Test)

{

((ExtendedDbreezeTripLogPersistency)\_persistency).RemoveAll();

\_persistency.Dispose();

}

else

{

// Nothing or exception?

}

}

The variable test is hard-coded and its value needs to be set based on a configuration file. We will get to that later.

Now running the following test will lead to 1 fail, 1 success:

[TestMethod]

public async Task SimpleInsertRetrieveTest()

{

await \_client.AddEntryAsync(First);

var retrieved = await \_client.ReadAllEntriesAsync();

Assert.AreEqual(1, retrieved.Count);

Assert.AreEqual(First, retrieved.First()); // <==== This seems to fail...why?

}

[TestMethod]

public async Task RemoveAllTest()

{

await \_client.RemoveAll();

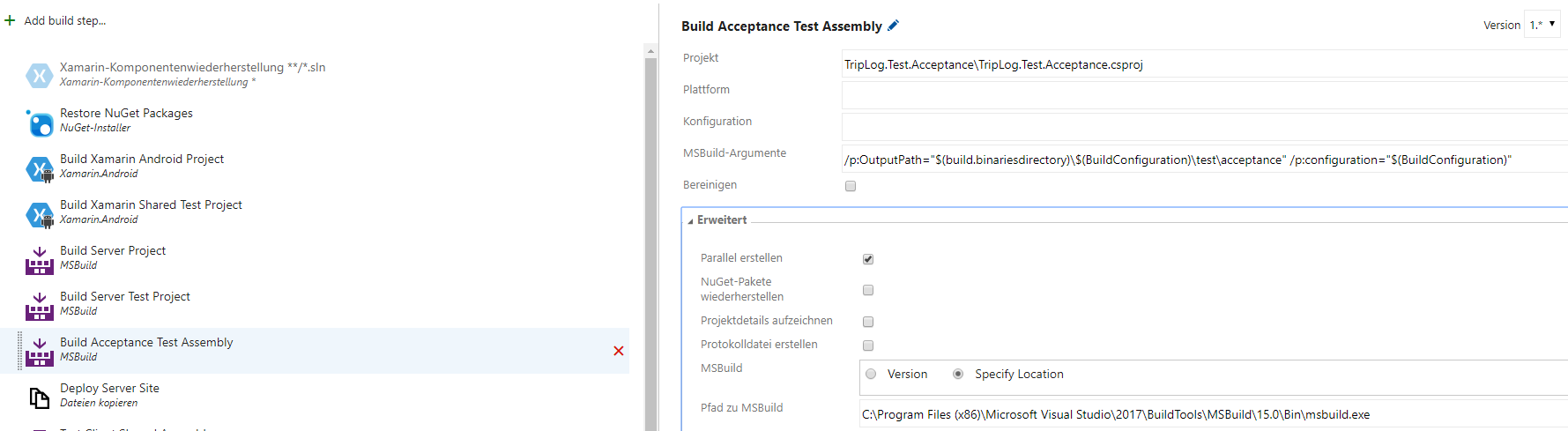
var retrieved = await \_client.ReadAllEntriesAsync();

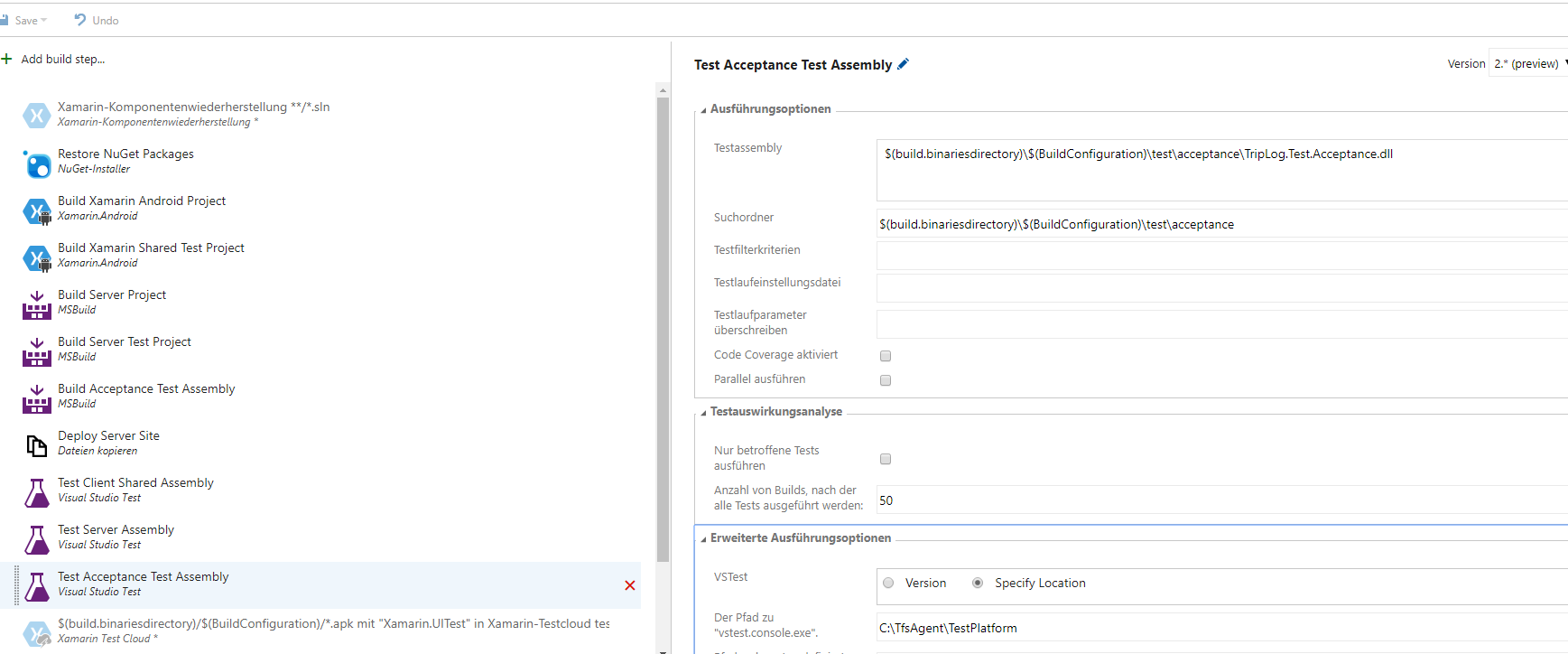
Assert.AreEqual(0, retrieved.Count);

}

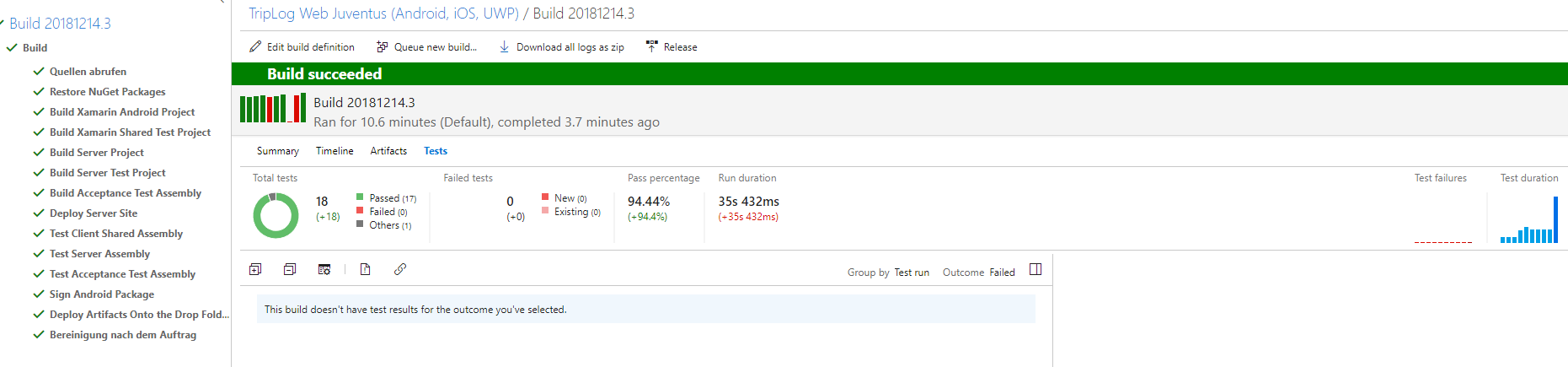
why? Please fix the issue.

We now need to add two new test steps to the build in order to build and execute these additional tests. Here the possible implementations:





Now check in and ensure that the resulting build completes succesfully:



Please note the significant increment in execution time due to the introduction of the acceptance tests.

By running the application, clicking new and saving a new location and moving back to the main screen, you will notice that the new entry is not being displayed. Fix this issue...

Lesson 7: Closing the deployment loop

So far we have built a simple multi-platform application communicating with a minimal backend, we have written a minimal set of unit, integration and acceptance testing and we have set up a CI system. We would like raise the bar by making the necessary arrangements to have both the server and the client software automatically deployed.

Objectives of this lesson are:

- Maturity levels of a build pipeline: Integration, Delivery, Deployment.

- Fine-tuning the code: clean code and refactoring

Let´s start by refactoring some code. The different factories and the kernel in app.xaml.cs can be grouped together in project-wide factory and moved to another file. Name this TripLogFactory.cs and move the code as follows:

public class TripLogFactory

{

private ViewModelFactory \_viewModelFactory;

private ViewFactory \_viewFactory;

private CombinedFactory \_combinedFactory;

private IKernel \_kernel;

public TripLogFactory(params NinjectModule[] platformModules)

{

\_kernel = new StandardKernel();

\_kernel.Load(platformModules);

}

public ContentPage Build()

{

var locationService = \_kernel.Get<GeoLocationService>();

var httpClient = new StandardAsyncHttpClient();

var backendUri = new Uri("http://192.168.56.10:20080/api/TripLogWeb/");

var restTripLogDataService = new RestTripLogDataService(httpClient, backendUri);

\_viewModelFactory = new ViewModelFactory(locationService, restTripLogDataService);

\_viewFactory = new ViewFactory(\_viewModelFactory);

\_combinedFactory = new CombinedFactory(\_viewFactory, \_viewModelFactory);

var viewModel = \_viewModelFactory.Build(ViewType.Main);

viewModel.Init();

var mainPage = new MainPage(viewModel);

mainPage.Init(\_combinedFactory);

return mainPage;

}

}

The app.xaml.cs will now look as follows:

public App(params NinjectModule[] platformModules)

{

var factory = new TripLogFactory(platformModules);

var mainPage = factory.Build();

MainPage = new NavigationPage(mainPage);

}

Rearrange all the dependencies on every file so that:

- No unnencessary dependency is present.

- Dependencies are listed in alphabetical order.

- The dependencies are grouped in the following order: framework, libs, project.

Here a sample:

namespace TripLog.Test.Acceptance

{

using System;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using TripLog.Models;

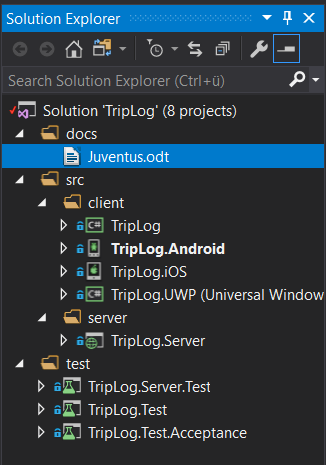
using TripLog.Services;

[TestClass]

public class RestTripLogDataServiceTests

{

Then rearrange the projects with folders a follows:



Lesson 8: Open points, missing features and possible extensions:

1) Remove all the hard-coded settings from the software by making the settings configurable. Define and check in the necessary client and server configuration files. Update the deployment pipeline accordingly so that it can support at least two environments: DEV/ENG/TEST and PROD. Ensure that a developer can test everything on his machine.

2) Make all the necessary arrangements so that the iOS version of the software can be built and deployed on a target Iphone. Ensure that the deployment pipeline is capable of publishing the package in an automated fashion onto an app store of choice. Provide the necessary documentation.

3) Make all the necessary arrangements so that the UWP version of the software (to be choosen among x86, x64, ARM, ARM64) can be built and deployed. Ensure that the deployment pipeline is capable of publishing the package in an automated fashion onto an app store of choice. Provide the necessary documentation.

4) Update the the whole application (client and backend) so as to support a user-specific or device-specific persistency of the trips so that each user can only retrieve his trips and not all of them! The db data migration is not required.

5) Introduce user authentication on both the client and the server.

6) Introduce caching so that the application can store data locally if is offline and improve the overall user experience of the application.

7) Introduce a logging service sending events to a to-be defined backend API. These events fall into the following categories:

- Login

- Logout

- Move to screen X from screen Z

- Error

8) Migrate the server project and the server test project to ASP Core.

9) Add the following features to the client:

- A quit button to close the application in a clean way.

- A functionality to move from the new entry form the main form after having clicked save.

- A configuration page allowing for example to configure the URL of the backend.

- A remove button on the main page.

10) Find a way to make the ignored test green.