Collaboration - GIRAF architecture

This chapter seeks to document the architectural design applied to this project, to ease the transition into the GIRAF project for future students. As previously described, the old frontend was discarded and a new cross-platform application has been developed. This application is built in Xamarin.Forms and uses the MVVM-pattern; Model-View-ViewModel. This chapter is written in collaboration between two groups, namely the Scrum/frontend group (SW610F18) and the other frontend group (SW608F18). This semester, we have been focusing strictly on the frontend client. Therefore, we will clarify some of the important architectural choices for future developers to help them get started on this project.

Furthermore, there will be explanations of the folder structure, dependencies, code examples and other important guidelines.

Project structure

Figure 1 represents the last applicable folder structure of the GIRAF-project.

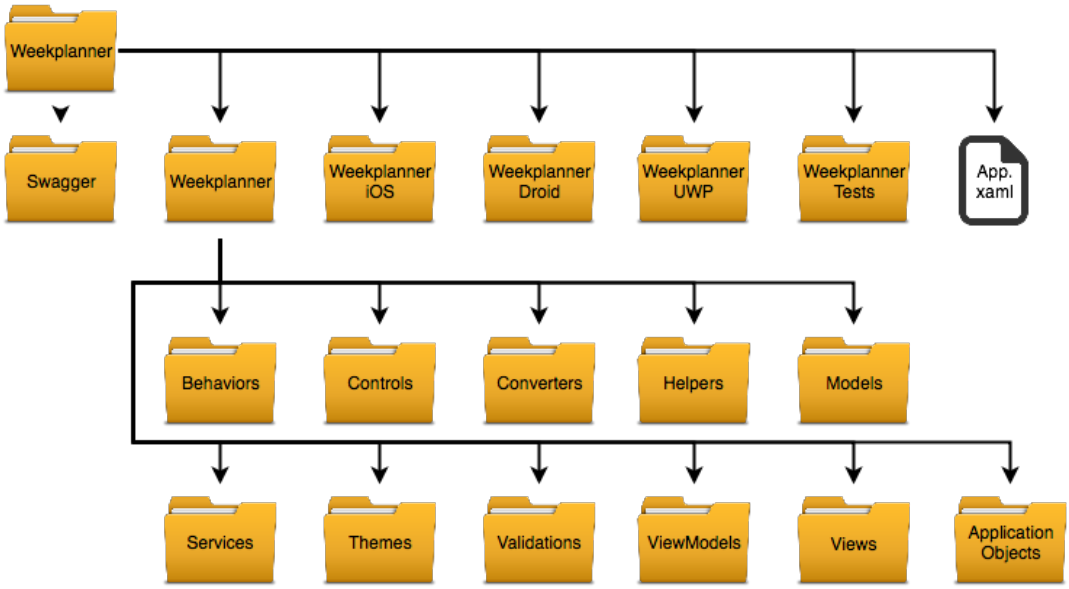

Figure 1: GIRAF-project Weekplanner folder structure

The following sections will document the use of the current structure, their purpose and responsibility of each folder and its content. The main purpose is to give

new developers an idea of how and where to place responsibilities when developing new components. This is especially important when considering the stability of GIRAF. Since stability is of great importance this semester —and also of great importance to our customers— it is crucial that we, and also future developers, maintain the current project structure and architecture.

Figure 1 depicts six folders inside the *Weekplanner* folder, namely; Swagger, Weekplanner, Weekplanner.iOS, Weekplanner.Droid, Weekplanner.UWP and Weekplanner.Tests. Xamarin.Forms is built with code-sharing in mind, and your common code is written in the Weekplanner-project. For ease of maintainability, one should strive to write all code in this project, rather than the platform-specific projects such as Weekplanner.iOS, Weekplanner.Droid, and Weekplanner.UWP. Writing in the platform-specific projects can, however, become necessary when dealing with platform related issues such as handling back-button presses.

The Weekplanner.Tests contains all the tests for the Week Planner project. Lastly, the Swagger project contains an autogenerated client-side API. There will be a thorough explanation of Swagger, its purpose, and how it is used in later sections.

For this chapter, we will be focusing primarily on the Swagger project and the Weekplanner project, because they are the key elements needed to get a fundamental understanding of the structure and architecture. Platform-specific projects and the test projects are self-explanatory and will not be covered further.

GIRAF-architecture

The GIRAF environment uses the client-server architecture. Figure 2 depicts the general structure of the GIRAF project.

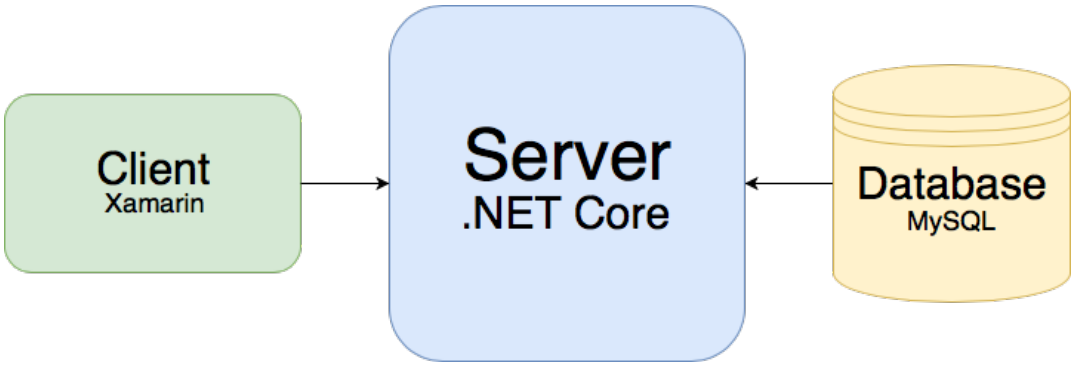

Figure 2: Architecture

There are several advantages in using this architecture. One of the main advantages of this architecture is the possibility of data sharing between clients. There can potentially be several clients at an institution which makes it possible to easily share data between all clients. This also means that changes to the backend or the frontend only needs to be done in one place. The same applies to the database. This separation of responsibility also contributes to stability and maintainability — which is a high priority this semester— since all components work independently of each other.

This architecture also allows a high degree of flexibility. At the moment we have several clients which all communicate with the backend through the API, namely the Weekplanner application(using Swagger) and the admin panel. They both use the same API to communicate and manipulate data in the backend. More clients can be added in the future without necessarily having to change anything in the backend.

Frontend-architecture

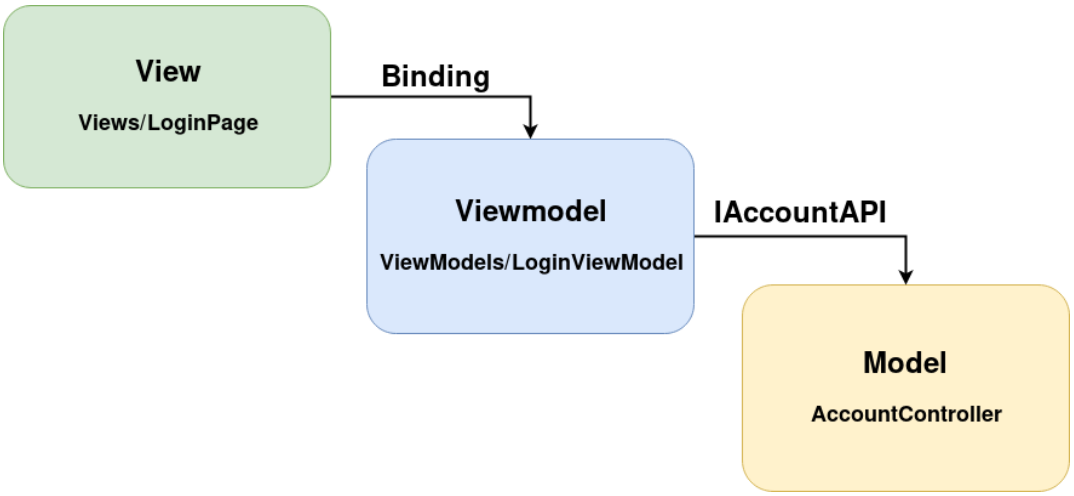

Figure 3: Architecture

Starting Point - App.xaml

Every thinkable program has a main component which initialises the program and Xamarin.Forms applications are no different, we will go through the global scoped resources in the .XAML file and the application class.

App.xaml

App.xaml defines a static resource dictionary, which can be queried from anywhere inside the project. It is especially useful for theming and colors. In Listing 1 on line 6, a new value for the dictionary is created using XAML-syntax. When the color MondayColor is needed in a view, one can simply look it up in the StaticResource dictionary. Using static styling eases the maintainability of the design because updates only need to happen in one place. One is not limited to only defining global colornames. It is also possible to style whole elements, as seen in Listing 1 on lines 8-15, where the background- and text-color is being set for all buttons in the project. We define the scope of a style by using the TargetType-attribute, but it can also be done based on id's using the `x:Key`-attribute as is done with the colors.

> **Do:** If you find yourself reusing styles in several places, then extend the resource dictionary with it, instead of relying on copying.

```
1  <Application XML namespaces
2              x:Class="WeekPlanner.App">
3      <Application.Resources>
4          <ResourceDictionary>
5              <!--COLORS-->
6              <Color x:Key="MondayColor">#067700</Color>
7              ...
8              <Style TargetType="Button">
9                  <Setter
10                     Property="BackgroundColor"
11                     Value="{StaticResource Primary}" />
12                 <Setter
13                     Property="TextColor"
14                     Value="{StaticResource WhiteColor}"
15                 />
16             </Style>
17             <!-- CONVERTERS -->
18             <converters:ItemTappedEventArgsConverter
19                 x:Key="ItemTappedEventArgsConverter" />
20             ...
21             <ControlTemplate x:Key="NavBarTheme">
22             ...
23             </ControlTemplate>
24         </ResourceDictionary>
25     </Application.Resources>
26 </Application>
```

Listing 1: App.xaml

App.xaml.cs - code-behind

App.xaml.cs, Listing 2, is the first class that is run when starting the application. It is responsible for starting the application, making sure dependency injection is set up, and initialise the external libraries FlowListView and FFImageLoading.

```
1  public App()
2  {
3      InitializeComponent();
4      FlowListView.Init();
5      InitApplication();
6      MainPage = new MasterPage();
7  }
8  private static void InitApplication()
9  {
10     var appSettings = GetApplicationSettings();
11     AppSetup setup = new AppSetup();
12     AppContainer.Container = setup.CreateContainer(appSettings);
13     InitFFImage();
14 }
```

Listing 2: App.xaml.cs

ApplicationObjects

The ApplicationObjects-folder is used for dependency injection. AppContainer.cs, Listing 3, is has the IoC container and AppSetup.cs registers all the dependencies.

```
1  public sealed class AppSetup
2  {
3      private void RegisterDependencies(ContainerBuilder cb)
4      {
5  olivegreen       olivegreen//olivegreen olivegreen***olivegreen
               olivegreenConstantolivegreen olivegreenRegistrationsolivegreen
               olivegreen***
6  olivegreen       olivegreen//olivegreen olivegreenViewModels
7          cb.RegisterType<LoginViewModel>();
8  olivegreen       olivegreen//olivegreen olivegreenServices
9          cb.RegisterType<NavigationService>()
10             .As<INavigationService>();
11         cb.RegisterType<RequestService>()
12             .As<IRequestService>();
13         cb.RegisterType<SettingsService>()
14             .As<ISettingsService>()
15             .SingleInstance()
16             .WithParameter("appSettings", _appSettings);
17         var accountApi = new AccountApi {
```

```
            Configuration = {
                BasePath = _appSettings["BaseEndpoint"]
                    .ToString()
            }
        };
        cb.RegisterInstance<IAccountApi>(accountApi);
        cb.RegisterType<LoginService>().As<ILoginService>();
        );
    }
}
```

Listing 3: ApplicationObjects/AppSetup.cs

Pages

In Xamarin, the best practice for creating views is using XAML (a superset of XML) for defining all the markup. You could also code the view strictly in C#, however, this has a few disadvantages in contrast to XAML. XAML makes it easy to grasp the tree-like structure and results in more concise, easy-to-read markup. However, XAML is limited. If you need complex view-based logic then you may have to fall back to coding your view. In addition, using XAML for your views makes it hard to mix your presentation- and business logic.

```
<?xml version="1.0" encoding="UTF-8"?>
<base:PageBase xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:behaviors="clr-namespace:WeekPlanner.Behaviors;assembly=WeekPlanner"
            xmlns:converters="clr-namespace:WeekPlanner.Converters;assembly=WeekPlanner"
            xmlns:base="clr-namespace:WeekPlanner.Views.Base;assembly=WeekPlanner"
            x:Class="WeekPlanner.Views.LoginPage"
            Title="Log ind">
    <ContentPage.Resources>
        <ResourceDictionary>
            <converters:FirstValidationErrorConverter
                x:Key="FirstValidationErrorConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <ContentView ControlTemplate="{StaticResource NavBarTheme}">
            <StackLayout>
                <Entry x:Name="UsernameEntry"
                    Text="{Binding Username.Value}"
                    Placeholder="Brugernavn"
                    WidthRequest="250"
                    Completed="Username_Completed">
                    <Entry.Behaviors>
```

```
                    <behaviors:EventToCommandBehavior
                        EventName="TextChanged"
                        Command="{Binding ValidateUsernameCommand}" />
                </Entry.Behaviors>
            </Entry>
            <Button>
                ...
                Command="{Binding LoginCommand}"
                Text="Log ind"
                ...
            </Button>
        </StackLayout>
    </ContentView>
</ContentPage.Content>
</base:PageBase>
```

Listing 4: Views/LoginPage.xaml

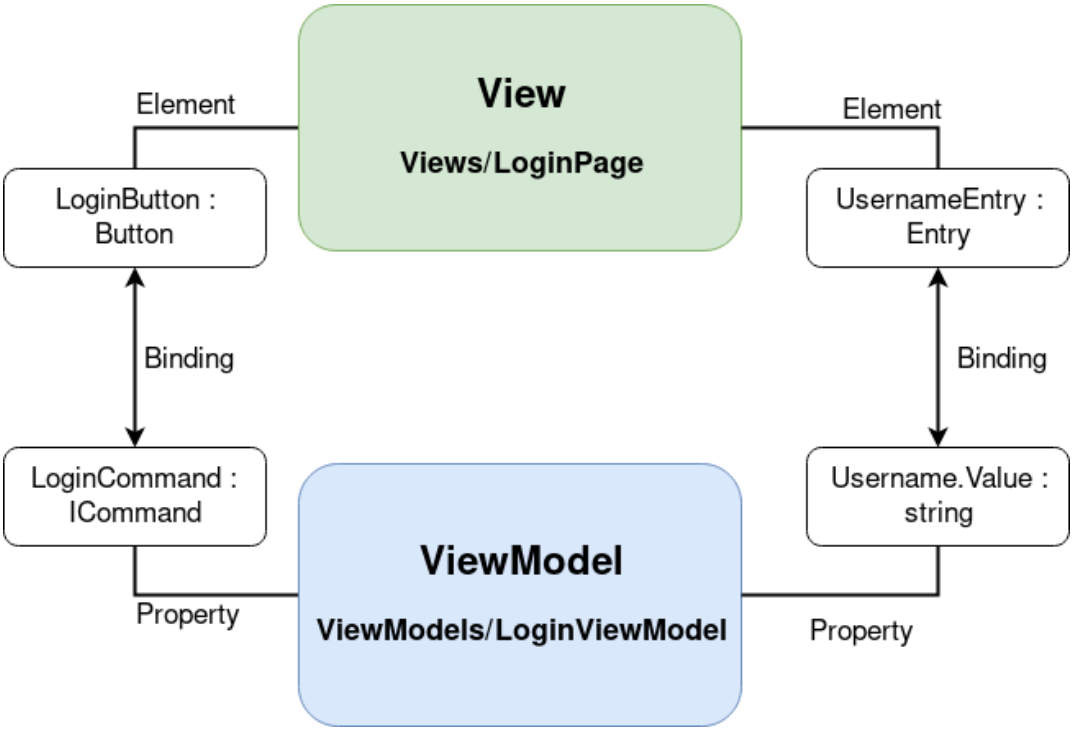

Figure 4: LoginPage

Do: Use XAML to structure your views.

Pages are responsible for showing the title, navigation and holding all the content. The content is displayed using various types of views such as ListView, Label,

and Picker, and layouts such as Grid, StackLayout, and ScrollView. Layouts derive from the View class which is why some of the names overlap.

Listing 4 shows the typical structure of a page. Notice that it derives from PageBase rather than ContentPage. We created the abstract class PageBase in order to customise all of our pages, so they disable the builtin NavigationBar and has a configurable behaviour for when the back-button is being pressed. At the top, resources are defined. Resources defined in App.xaml are inherited throughout the app. Inside the Content we typically have a ContentView referencing a NavigationBar ControlTemplate.

Converters

The best practice is to use the MVVM pattern in your Xamarin.Forms application. When you are using MVVM the ViewModel has a model which holds your data. If you want to display this data you use data binding. Databinding only works on properties inside of the BindingContext. However, you may want to apply some sort of logic to your data before displaying it. There at two common approaches to this; change your properties in the ViewModel, or use a converter. The first approach is seldom chosen because the properties often directly refer to the generated models from Swagger. The better alternative is therefore converters. Converters are classes implementing the IValueConverter interface, which has a Convert-method that takes an input (typically a data-bound property from your ViewModel) and returns the converted output. An example of using a converter can be seen in Listing 5 where the ID of a pictogram is converted to a URL.

```
1 <Image Source="{Binding Pictogram.Id,
2 Converter={StaticResource IdToRawPictoUrlConverter}}" />
```

Listing 5: Converter example

Controls

Controls are reusable views/components. If you find yourself repeating parts of your pages you may want to extract it into a reusable control. One example is the WeekDayListView, which displays the activities for a single day in a listview, and is being used by the WeekPlanner-view.

> **Do:** Extract repeated parts of a view to a separate control.

Behaviours

Behaviours let you add functionality to your controls. Our main use of behaviours is EventToCommandBehavior which makes it possible to redirect events, which

typically need to be handed in code-behind (the C# file), to commands which can be created in your ViewModels.

ViewModels

The responsibility of a viewmodel is twofold; it provides data and functionality to the view via bindings, as well as handling all the logic.

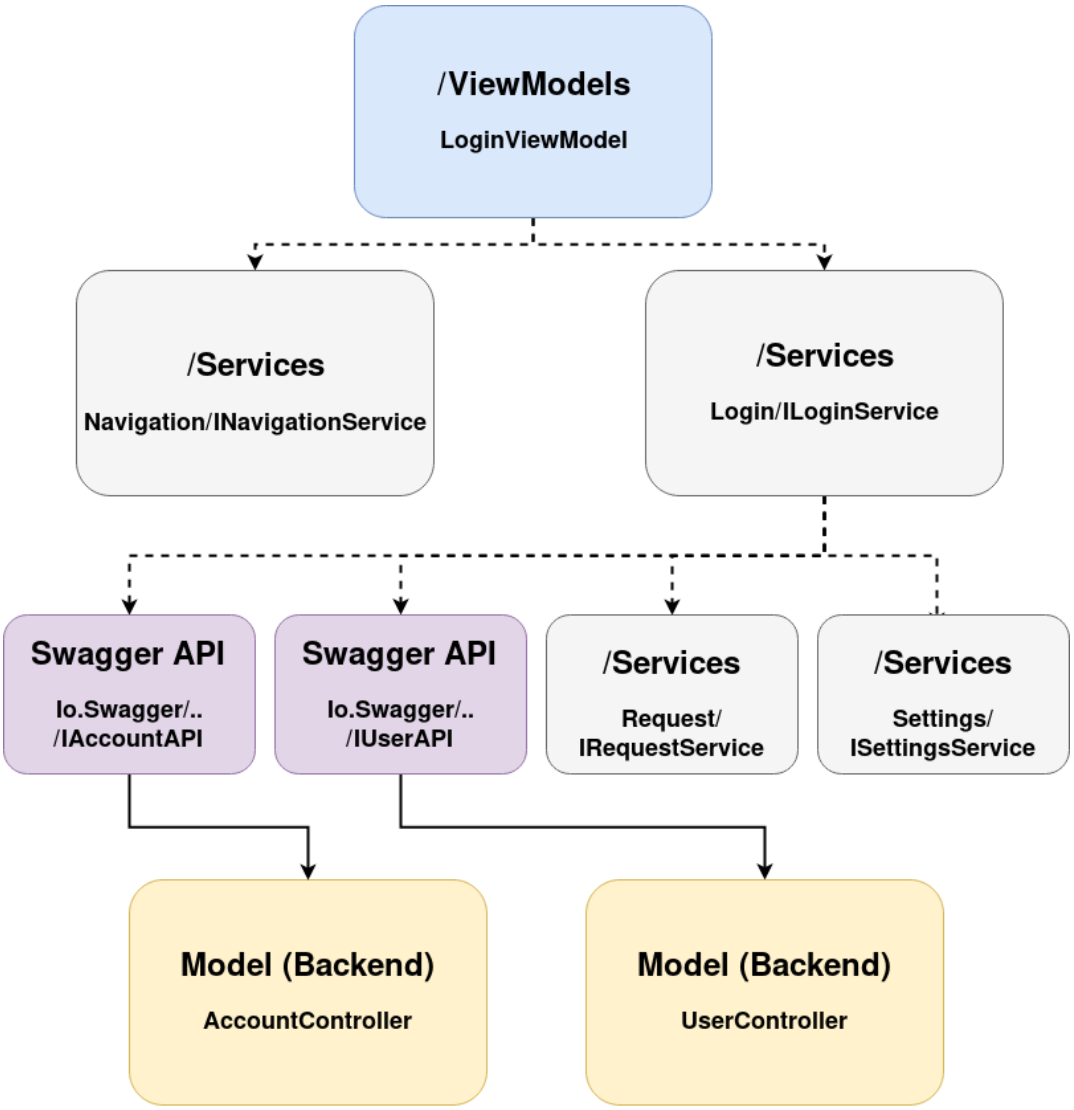

Figure 5: LoginViewModel

In our example of `LoginPage`, whenever the user writes an input in the `Username`-entry, the setter-property gets called, with the value, see Listing 6, 17-21. Lets

assume there are elements, for example labels, which display the value of `Username`. Whenever the setter property is invoked, the elements get notified of this change of state, and they update their view to reflect the new underlying state (see line 20).

We consider it good practice to keep getters free of side-effects, to better reason about when the state changes. Pure code is generally more testable and readable, but especially in the case of these kinds of getters, since there is no complete control over when the getters are called by the view.

> **Do:** Write your getters without side-effects.

We bind the `LoginButton` to `LoginCommand`, this will have the effect of invoking the function `LoginIfUsernameAndPasswordAreValid`, when the button is pressed.

We consider it good practice to make commands run asynchronously, to ensure that other running threads, such as the UI-thread can run concurrently with the command.

> **Do:** Make long-running tasks asynchronous, to maintain a responsive application.

In the case of the `LoginCommand`, the command makes a request to the server in order to log in and authenticate the user. Instead of cluttering the ViewModel with internal functionality, one should delegate the responsibility of making requests to a different component, in this case, `LoginService`.

> **Do:** Write a new component if you exceed the responsibility of the ViewModel. Especially if functionality can be reused in other ViewModels.

As can be seen in line 3, the ViewModel is coupled to the interface, `ILoginService`. In the constructor, the reference gets initialised with a value implementing the ILoginService interface. A unique dependency to ViewModels, is `INavigationService`, this interface provides functionality to navigate between pages. Navigation is common to all ViewModels, so `ViewModelBase` has a reference to the INavigationService, and a constructor which initialises `NavigationService`. The dependencies of `LoginViewModel` are shown in Figure 5, and are injected on runtime based on the types registered in the setup, described in . The internals and dependencies of `LoginService` will be explained in .

```
1 public class LoginViewModel : ViewModelBase
2 {
3     private readonly ILoginService _loginService;
4     public LoginViewModel(INavigationService navigationService,
5         ILoginService loginService) : base(navigationService)
6     {
7         _loginService = loginService;
8         ...
```

```
9       }
10      public ICommand LoginCommand => new Command(
11          async () =>
12          await LoginIfUsernameAndPasswordAreValid()
13      );
14      public ValidatableObject<string> Username
15      {
16          get => _username;
17          set
18          {
19              _username = value;
20              RaisePropertyChanged(() => Username);
21          }
22      }
23  }
```

Listing 6: ViewModels/LoginViewModel.cs

Services

Services are created with the purpose of offering functionality to the ViewModels. The LoginService, Listing 7, for instance, provides authentication functionality to the ViewModel. Each service can also depend on other services in order to maximise code reuse. The LoginService has several dependencies:

- `IAccountApi` Acts as an intermediary component between the frontend and the account controller in the backend. This API is generated by swagger and contains methods for each endpoint in the AccountController. This API is used for login-requests.
- `IRequestService` is an omnipresent service, since most view-models has a need for making a web-request. RequestService is able to make the wished request, invoke a method with the response as input, and handle possible errors which might occur.
- `ISettingsService` is responsible for managing the settings a citizen can have, week plan colours, Weekplanner theme, and etc.
- `IUserApi` is reminiscent of IAccountApi, but it is used for requesting data about an authenticated guardian or a specific citizen, rather than dealing with authentication.

Do: Thoroughly test your services, because they are used in several locations.

Do: Use IRequestService to create requests to ensure that ApiExceptions and errors in the response are handled.

```
public class LoginService : ILoginService
{
    public LoginService(IAccountApi accountApi,
    IRequestService requestService,
    ISettingsService settingsService,
    IUserApi userApi)
    {
        _accountApi = accountApi;
        _requestService = requestService;
        _settingsService = settingsService;
        _userApi = userApi;
    }
}
```

Listing 7: Services/Login/LoginService.cs

IO.Swagger

Swagger generates classes based on the backend implementation. The important generated classes are the response classes, Data-Transfer-Objects(DTO's), and the APIs.

Avoid: Hacking swagger to fit your purpose. Your changes will be overwritten on next swagger-generation. Re-evaluate the backend design instead.

Validations

In order to use validations you need to create a ValidationRule if one doesn't exist for your purpose. Then you simply wrap this rule in a ValidatatableObject<T> where T is the type you want to validate as seen in Listing 8.

```
Password = new ValidatableObject<string>(
    new IsNotNullOrEmptyRule<string>
    { ValidationMessage = "A password is required." }
);
```

Listing 8: Validation example

Models

Models are mostly defined in the Swagger-project, since we use the DTO's directly in our ViewModels. We have considered implementing all the models and have

them implement INotifyPropertyChanged. This would save a bit of code since we have quite a bit of "wrapper-properties", properties of which the only purpose is to reference another property and RaisePropertyChanged. This was done for the ActivityDTO, with the class ActivitiyWithNotifyDTO, because it was needed for changes in an activity to be updated in the WeekPlannerPage.

In the case of the WeekdayColors model, the SettingsViewModel was growing quite big, so we extracted WeekdayColors to have better separation of concerns. WeekPlannerViewModel could use quite a bit of refactoring as well.

> **Do:** Refactor your ViewModels for separation of concerns to avoid them growing too big.

Helpers

The main purpose of helpers is to help with small important tasks. Ideally, this folder only contains small static methods, which are being used multiple places in the application. An example of this is the ErrorCodeHelper, which provides friendly text messages whenever an error occurs, based on a specific error code. The method takes an error code as input and returns a string with an appropriate string describing the error. This ensures that we only have to define error messages once and that we can provide consistent error messages.

Themes

This folder contains files defining all GIRAF themes available. Every theme has its own file. Essentially, this file is a resource dictionary defining theme colours and other theme specific styles. To create a new theme, simply add an additional file containing style definitions etc. At the moment, it is not possible for the guardian to create new themes – all themes are predefined by the developers.