HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

BrainingOutOfBox

Review & Findings

Hochschule für Technik Rapperswil

IFS INSTITUTE FOR SOFTWARE

1 Inhalt des Reviews

Dieses Review enthält Findings zum Programmcode:

- Server / Master stand `af00957` https://github.com/BrainingOutOfBox/API.git
- Client / Master stand `9272cfc` https://github.com/BrainingOutOfBox/App.git

2 Server

2.1 MVC nicht eingehalten

Controller enthalten direkte Zugriffe auf die DB und somit auch Business Logik (konkret: kein MVC). Um die Controller nicht an eine DB-Implementation zu binden, sollten Services (und falls viel Business Logik folgt, ebenfalls Data Access Objects) eingeführt werden. Dies ermöglicht Microtesting, wartbareren und testbareren Code sowie Separation of Concerns. Die Basis dazu stellt Google Play mittels des DI Containers zu Verfügung. Ein eigentliches Layering konnte im Server-System nicht gefunden werden.

2.2 Filters zur Code-Reduktion verwenden

Server-Side könnten Filters für die Security Headers (JWT) eingesetzt werden (siehe https://www.playframework.com/documentation/2.7.x/SecurityHeaders). Dies ermöglicht es, den Code in den Controllers zu reduzieren.
Ebenfalls ist es ratsam die JSON-Serialisierung sowie das Fehlerhandling in einen Filter auslagern, da dies sehr repetitiv ist und zentral effizienter gelöst werden kann (z.B. einheitliche Statuscode- und Error-Mappings / Success Messages / …). Andere Frameworks wie Spring Boot bieten solche Features "out-of-the-box".

2.3 DTOs nicht vorhanden

Teilweise arbeiten die Controllers direkt mit den JSON-Nodes anstatt mit Data Transfer Objects (Methode putBrainsheet). Dies resultiert in sehr instabilen Schnittstellen, welche bei kleinen Feld-Namensänderungen eine Shotgun Surgery https://refactoring.guru/smells/shotgun-surgery auslösen.

2.4 Technologie zielgerichtet eingesetzt

Controller-Code teils ungünstig lange; beispielsweise könnte anstatt einer eigenen anonymen Implementation von `SingleResultCallback` mit Lambdas gearbeitet werden (sofern Java 8+ eingesetzt wird).

Anmerkung: Der Code setzt stark auf ASYNC-APIs. Dies macht den Code einiges komplexer. Wäre die synchrone Standard-API auf https://www.playframework.com/modules/mongo-1.1/home nicht bereits ausreichend?

Unit Tests fehlen.

3 Client

3.1 UI / ViewModel

UI-Strings (z.B. Status Messages) wurden vielfach im View Model definiert (`JoinTeamPageViewModel.JoinTeam`). Dafür bietet .NET Resource Manifests (Project Settings → Resources) oder alternativ Static Resources als XAML Files.

Responsibilities wie Navigation sollten in spezialisierte UI-Services ausgelagert werden (befindet sich im Moment im `MainPageViewModel`).

3.2 Model / Services

Business Services sind nicht enthalten. Es wird aus dem UI (View Model) direkt auf den Data Access Layer (http Backend Abstraktion) zugegriffen.
Beispiel: Die Login-Prozedur mit entsprechenden States, welche fürs Login gespeichert werden müssen, befinden sich somit im ViewModel. Hier könnte ein "Login"-Service diese Responsiblities übernehmen und die Funktionalität kapseln. Separation of Concerns wäre somit sichergestellt, "globale" States wie im `BrainstormingContext` gehalten würden so in den Information Experts platziert. Entsprechend ist im Moment das Zusammenspiel der Events/Properties der `BrainstormingContext` Klasse sehr intransparent implementiert.

Die Benennung der DTOs ist zu http/REST-spezifisch; der Code liest sich so, als ob das ViewModel direkt mit dem http-Backend kommunizieren würde (z.B. `LoginPageViewModel.Login()`).

3.3 Data Access Layer

Die Server-URL befindet direkt im Code (d.h. fix definiert). Dies ist unglücklich, falls die Applikation im Testbetrieb auf verschiedene Umgebungen zugreifen muss. Eine Konfiguration pro Deployment-Target würde Abhilfe schaffen.

`RestResolverBase` bietet als Basisklasse ausschliesslich statische Methoden an. Dies wäre ein typischer Anwendungsfall eines Low-Level Resource Services, welcher eine erste Abstraktion des http-Backends anbietet; die Higher-Level Resource Services würden sich diesen via DI injecten lassen. Dieselben Mime-Types wurden in der `RestResolverBase` mehrfach definiert.

Data Access Objects (z.B. `TeamRestResolver`) sollten nie direkt vom Code (View Model) instanziiert werden. Dafür bietet der DI-Container von PRISM eine entsprechende Build-Up Funktionalität, um das automatisierte Testen (Unit Tests) so einfach und flexibel wie möglich zu gestalten.

3.4 Technologie zielgerichtet einsetzen

Generelle Code-Hygiene sollte erhöht werden (siehe https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions):

- Statements wie `findingItems.Count > 0;` könnten vereinfacht werden zu `.Any();`
- Wo immer möglich String-Interpolation anstatt "x" + "y" verwenden.
- `this.` .. in C# weglassen, wenn es nicht zwingend gebraucht wird.
- Fields an den Beginn der Klasse platzieren.
- Tools wie Resharper können hilfreich sein, um solche Verbesserungen zu erkennen.

Der Code sollte keine Console.Logs() enthalten. Hier könnten beispielsweise Logging-Frameworks (z.B. NLog) helfen.

Jeder Layer (ViewModel / Model / Data Access) sollte in ein eigenes Projekt (oder Library) platziert werden. Somit werden zyklische Abhängigkeiten von Beginn an vermieden.

Unit Tests fehlen.