Protokoll

Design und Architektur
Ich habe mich beim Design der Applikation an das MVVM-Pattern orientiert. Die Projektstruktur besteht aus 4 verschiedenen Projekten, dem WPF Projekt mit allen Views und Viewmodels, einer Class Library mit allen notwendigen Klassen und Interfaces, einem ASP.NET API auf dem die Datenbank aufgesetzt ist und das letzte Projekt welches alle Unit-Tests enthält.

Das verwendete MVVM-Pattern meines WPF Projekts besteht aus zwei Foldern, einer enthält die Views. nämlich:

- GeneralView: Zeigt die allgemeine Information einer Tour an (z.B. Name, Distanz, Beschreibung), diese können nach Belieben verändert werden. Bei veränderten Start- oder Ziel-Daten wird eine Anfrage an die MapQuest API geschickt, welche das Bild und die Distanz bzw. Dauer aktualisiert.
- RouteView: Zeigt ein Bild der Tour an und die dazugehörigen gespeicherten Logs
- TourLogWindowView: Ein Fenster das beim erstellen eines Tour Logs angezeigt wird, um die Eingaben des Benutzers zu erfassen
- OtherView: Platzhalter für spätere Features

Der zweite Folder enthält die Viewmodels, die die Logik der Views steuern.

Misserfolge
Im Laufe meiner Implementierung hatte ich mehrere Probleme:

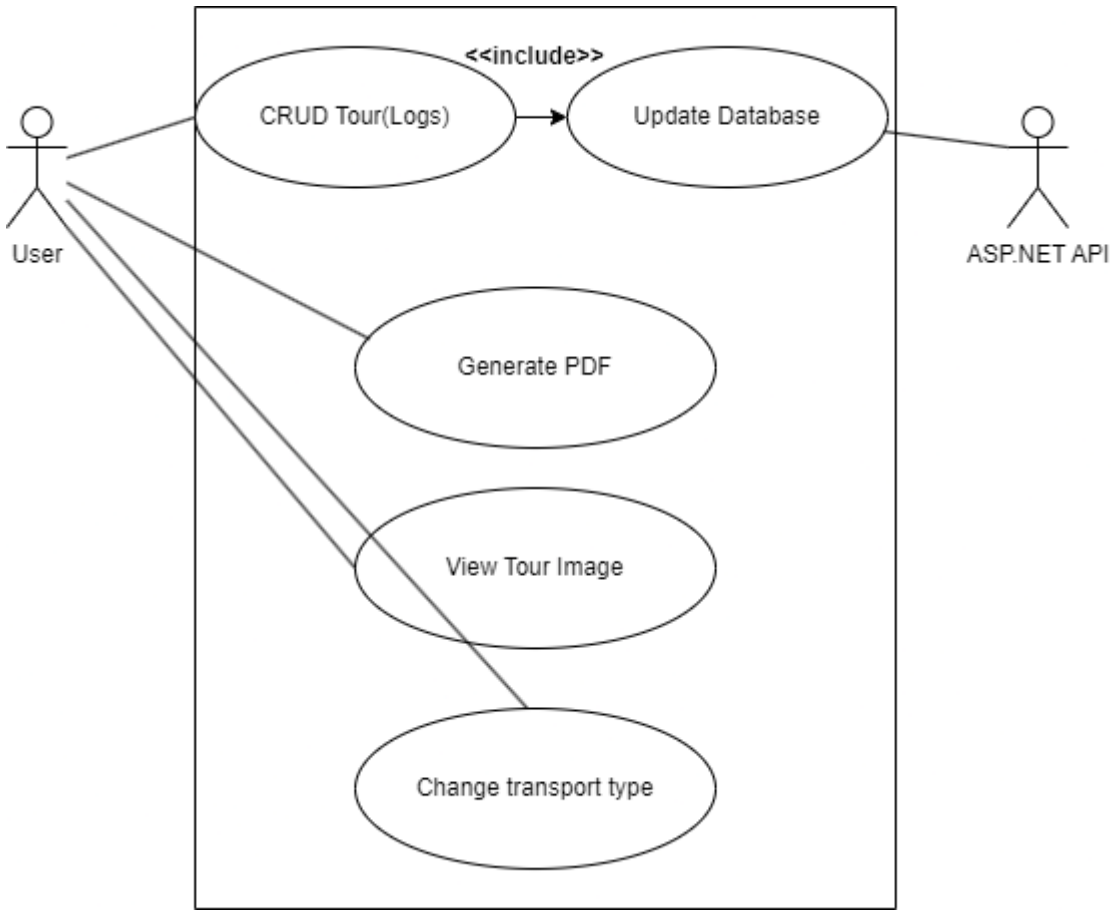
- Nicht funktionierendes Databinding bei Verwendung von Themes bzw. Templates für meine Controls
- Probleme mit json serializer und deserializer bei meinem ASP.NET API, gewisse Datentypen wie z.B. DateTime wurden nicht richtig formatiert.
- Probleme mit der Aktualisierung der Views, nachdem eine Änderung der Tour Collection vorgenommen wurde.
- Probleme mit asynchronen Funktionsaufrufen

Lessons-Learned
Nach intensiver Auseinandersetzung mit den Bestandteilen von WPF kann ich mit Zuversicht sagen, dass ich vieles gelernt habe, insbesondere das Verständnis von Views und Viewmodels und wie diese miteinander interagieren. Darüber hinaus hat das Bauen der App mir die Stärke von Asynchronität beigebracht, und wie diese zu einem benutzerfreundlichen und dynamischen Programm beiträgt. Ich war anfangs nicht überzeugt, wie das MVVM Pattern tatsächlich nützlich ist, allerdings wurde mir zu Ende des Projekts klar, wie jenes Pattern zu übersichtlichen und wartbaren Code führt.
Die Verwendung des Entity Frameworks war für mich eine Erleuchtung, da es die aufwendige Erstellung von Datenbank Schemas für mich übernommen hat.

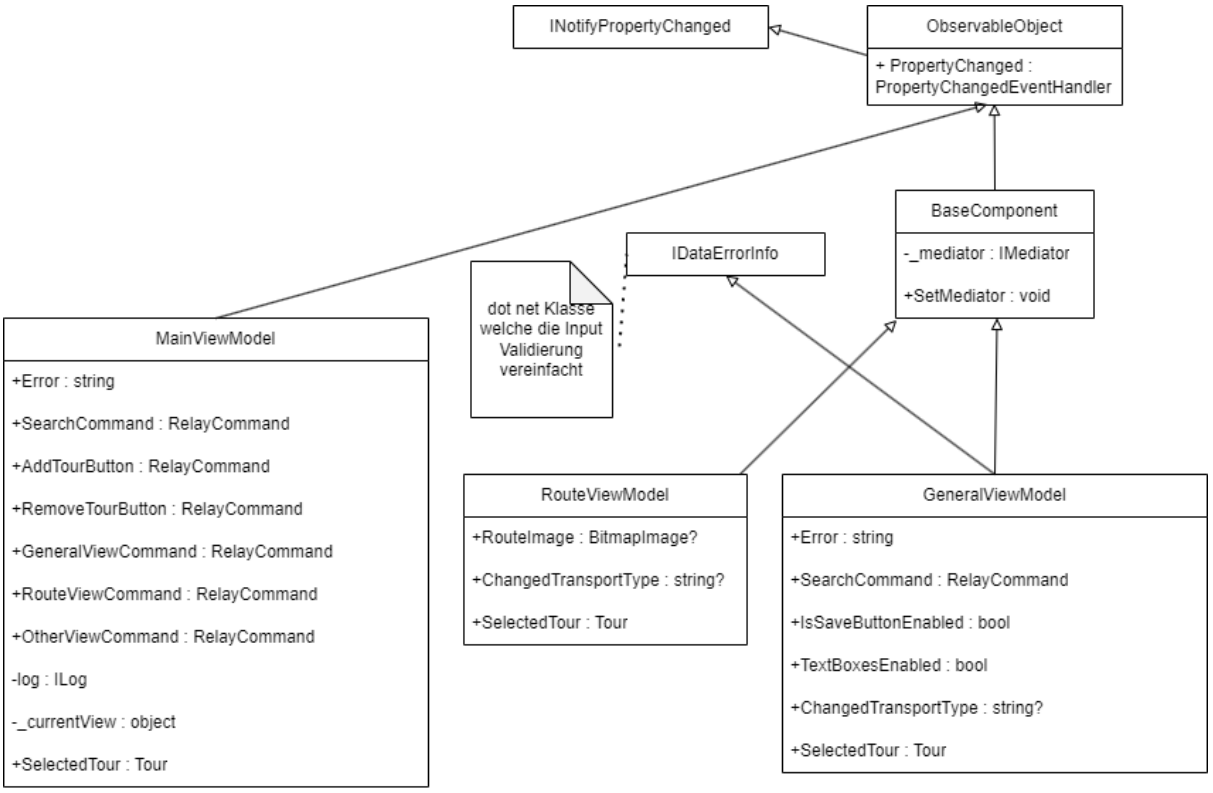
Selected Solutions

Wie bereits vorhin angesprochen, habe ich das MVVM Pattern verwendet, um eine wartbare und erweiterbare App zu bauen. Darüber hinaus habe ich ein API erstellt die den ganzen Datenbankzugriff, also die DAL (Data Access Layer) abkapselt. Ich fand die Lösung eleganter als eine lokale Datenbank, da sie auf die Weise zukunftssicherer und nutzvoller ist. Es besteht dadurch die Möglichkeit, mit anderen Benutzern der App zu kommunizieren und gewisse Touren zu teilen, ohne eine Art Exportfunktion einbauen zu müssen. Die Vorlage für mein API ist das bekannte Framework ASP.NET. Ich habe es deshalb ausgewählt, weil es heutzutage eine relevante Lösung zur Implementierung von erweiterbaren APIs ist, und ich damit Erfahrung sammeln wollte.

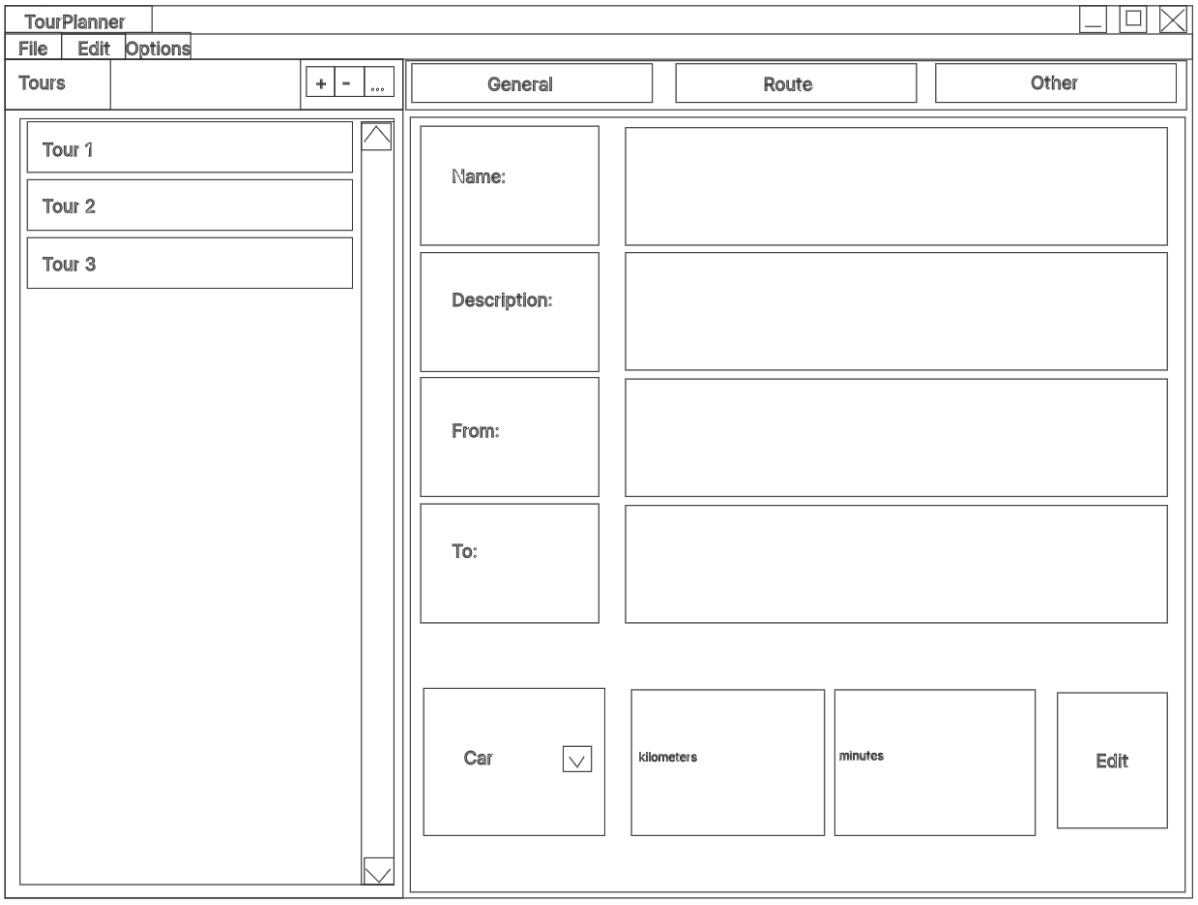
Use-Case Diagram



Class Diagram



Wireframe



https://www.figma.com/file/1JzVf1v0L2ADC0UzU6BVot/TourPlanner-UI-Wireframe?type=design&node-id=0-1&t=1FdwJeoah0Eb132Q-0

Unit Tests
Die Unit Tests testen die Code Bereiche meines Projekts, bei denen es besonders wichtig war, dass sie fehlerfrei sind. Insbesondere habe ich mich dabei auf sogenannte “Tour Attributes” konzentriert, also die verschiedenen Attribute, die eine Tour hat.
Unter den Attributen befinden sich die Popularität und die Kinderfreundlichkeit der Tour.
Die beiden Attribute werden von gewissen Metriken der Tour abgeleitet.
Die Popularität wird von der Anzahl der Logs beeinflusst, umso mehr Logs eine Tour besitzt, umso populärer ist sie.
Die Kinderfreundlichkeit wiederum wird durch den durchschnittlichen Schwierigkeitsgrad einer Tour, dessen Dauer und Distanz beeinflusst. Übersteigt mindestens eine dieser Metriken einen vordefinierten Schwellenwert, z.B. darf die durchschnittliche Schwierigkeit nicht größer gleich 3 sein, dann wird die Tour als nicht kinderfreundlich eingestuft.
Um sicherzustellen, dass diese Logik richtig funktioniert, schrieb ich einige Unit Tests, die genau die Funktionen, welche für die Bewertung verantwortlich sind, testen.
Abgesehen von den Attributen einer Tour habe ich auch eine Methode getestet, die sicherstellt, dass das von der MapQuest API geschickte byte array tatsächlich ein JPEG ist.
Die letzte Reihe an Unit Tests überprüft, ob die generierten API request strings richtig sind.

Time Spent
Die Zeit, die ich zur Umsetzung des Projekts benötigt habe, beträgt schätzungsweise 80 Stunden, ich habe das Projekt im Laufe des ganzen Semesters umgesetzt, deshalb tue ich mir mit der Schätzung ein wenig schwer. Ich habe auch versucht, möglichst oft auf Git zu pushen und die Kommentare möglichst informativ zu schreiben, um in Zukunft einen besseren Überblick über den Verlauf der Entwicklung zu haben. Nun setzt sich mein git repo aus 35 commits zusammen, und man sieht auch den veränderten Source Code nach jeder Änderung. Diese Tatsache hat mir öfter bei der Entwicklung geholfen, um zu sehen, welcher Code Teil verändert wurde, um bestimmte Features zu implementieren, was wiederum zum Debugging bei später auftretenden Problemen notwendig war.

Design Pattern
Bei der Implementierung der Logik, die für das Aktualisieren des Bildes verantwortlich war, bemerkte ich ein Problem. Angenommen wir erstellen eine neue Tour, diese Tour hat noch kein Bild, da wir weder den Start noch das Ziel kennen. Nun geben wir im General View, der ja für die Eingabe der Tourinformation dient, diese Daten ein. Bei der Veränderung dieser Daten muss das Bild im RouteView aktualisiert werden, da wir entweder noch kein Bild oder ein veraltetes Bild haben. Das Problem ist nun, dass das GeneralViewModel und das RouteViewModel voneinander abgekapselt sind, und ich keinerlei Möglichkeit habe, den RouteView darüber zu informieren, dass sich die Daten verändert haben. Hier kommt das von mir ausgewählte Design Pattern ins Spiel, nämlich das Mediator Pattern. Das Mediator Pattern ist dafür verantwortlich, eine Kommunikation von Komponenten zu ermöglichen, ohne dass diese Komponente Abhängigkeiten implementieren müssen und dadurch zur Wiederverwendung ungeeignet werden. Um dies zu erreichen, schrieb ich eine Klasse namens TourMediator die ein Interface IMediator implementiert welche eine einzige Methode Namens Notify hat, diese feuert ein Event los das beim RouteView das Bild aktualisiert.
Danach habe ich im Konstruktor vom MainViewModel den Mediator initialisiert und die ViewModel des General- und RouteViews dem Mediator weitergegeben.
Nun kann durch Aufrufen der Notify Funktion im GeneralViewModel ein Event gestartet werden, welches mit hilfe des Mediators an das RouteViewModel weitergegeben wird.