

Facharbeit

Erstellung einer Webapplikation

Programmierung einer Applikation zur Speicherung von Schulnoten

Verfasser: Robert Brock
Kurs: Grundkurs Informatik 12
Tutor: Herr Heilmann

1. Vorüberlegungen zum Projekt	4
1.1 Theoretische Abstraktionen des Projekts	4
1.1.1 Klärung der Fachtermini	4
1.1.2 Aufbau des Projekts	5
1.1.3 Umsetzung der REST-Architektur	6
1.2 Praktische Überlegungen an das Projekt	6
1.2.1 Komponenten und Systeme	6
1.2.2 Klassen und Verhalten in der Backend-Api	8
1.2.3 Benötigte Komponenten und Klassen	9
1.2.4 Funktionen planen - Abläufe strukturieren	14
2. Projekt-Verwirklichung	17
2.1 Backend - WebApi - Programmierung mit C#	17
2.1.1 Anlegen des Projekts + Einrichtung	17
2.1.2 Installieren der benötigten Pakete	19
2.1.3 Die Model-Klassen anlegen	20
2.1.4 Die Datenbank vorbereiten	22
2.1.5 Das ResponseModel - Schnittstelle zum Nutzer	23
2.1.6 Der Service (Interface und Klasse)	24
2.1.6 Abhängigkeiten zwischen Interface und Klasse herstellen	25
2.1.7 Die Service-Klasse (Implementierung des Interface)	25
2.1.8 Datenbank-Aktionen ausführen	28
2.1.9 Der Controller der Serviceklasse	29
2.2 Backend - WebApi - Funktionstest mit Swagger	31
2.3 Frontend - GUI - Programmierung mit Angular	34
2.3.1 Vorbereitungen treffen - GUI Projekt einrichten	34
2.3.2 Vorbereitungen treffen - Abhängigkeiten klären	37
2.3.3 Vorbereitungen treffen - Skripte anlegen	38
2.3.4 Vorbereitungen treffen - Api verknüpfen	39
3. GUI Umsetzung - TypeScript und Angular	41
3.1 Anlegen eines Components	41

3.2 Die component.ts Datei - Logikimplementierung	42
3.3 Die HTML des Components	44
4. Starten der Anwendung	46
5. Software-Testing - Entwicklung	48
5.1 Übersicht Thema Software-Tests	48
5.1.1 Arten der Software-Testung	48
5.1.2 Der Unit-Test	49
5.2 Software-Tests anlegen	50
5.2.1 Projektstart - XUnitTests für die API	50
5.2.2 Die Infrastructure anlegen	51
5.2.3 Die Assertions anlegen	53
5.2.3 Den Test schreiben - Unit-Test	54
5.2.4 Der Test - Explorer	55
6. Abschließende Worte zum Projekt	56
7. Quellenverzeichnis	57

1. Vorüberlegungen zum Projekt

1.1 Theoretische Abstraktionen des Projekts

1.1.1 Klärung der Fachtermini

Da sich im Laufe des Projektes und dieser Projektausarbeitung verschiedene Fachbegriffe ansammeln werden, welche sich nicht explizit an der jeweiligen Stelle erklären lassen, besteht dieses Unterkapitel als eine Art vorgeschoenes Lexikon verschiedener, für den Kontext der Arbeit wichtiger, Fachbegriffe aus der Software- und Webentwicklung. Ziel dieses Lexikons ist weder ein breitgefächterter Einstieg in die Welt der Fachbegrifflichkeiten der Informatik, noch einer vollkommenen Definition mit Vollständigkeitsanspruch der jeweiligen Begriffe. Vielmehr soll dies Fachwortsammlung als moderater Einstieg in das Thema, sowie zum besseren Verständnis des Vokabulars der vorliegenden Facharbeit dienen.

Begriff	Erklärung
Frontend	Umfassender Begriff für alle Teile einer Anwendung, welche der Nutzer sieht und mit denen er vornehmlich interagieren kann. Beispielsweise Buttons, Textfelder, Überschriften und Absätze auf Webseiten, Seiten einer Webseite, Medien wie Bild, Video und andere, sowie Posts und Co. Auch zählen dazu die Logik und der Programmcode, welchen der Nutzer zwar nicht sieht, jedoch das Verhalten der Anwendung im Bezug auf die Nutzererfahrung beeinflussen.
Backend	Umfassender Begriff für alle Teile einer Anwendung, welche der Nutzer nicht sieht und mit denen er nicht im direkten Kontakt steht. Server, Datenbanken, Datenstrukturen, Speichermedien und all deren Logik dahinter um die Verarbeitung der Daten die diese speichern, lesen, oder ändern zu gewährleisten.
App	Applikation, Computerprogramm
GUI/ User-Interface	[Graphical User Interface] ==> Beschreibt die grafische Benutzeroberfläche von Computerprogrammen. Begriff für das „Bild“, welches der Nutzer von der Applikation wahrnimmt. Die Elemente aus welchen das Frontend aufgebaut ist.
Interface	[Schnittstelle]: Übergangsstelle zwischen verschiedenen Komponenten einer IT-Anwendung. Zum Realisieren des Datenaustausches und deren Verarbeitung
Class	Klasse ==> Struktur um Variablen und Funktionen eines bestimmten Logik-Bereiches zusammenzufassen.
Method	Funktionen in einer Klasse
Function	Funktionen => Programmstruktur zum bündeln wiederverwendbaren Programmcodes und zur Umsetzung von Algorithmen in Computerprogrammen.
Variablen	Kleinster Baustein einer programmierten Anwendung zur Speicherung
API	[Application Programming Interface]: Satz von Befehlen, Funktionen, Protokollen und Objekten, die Programmierer verwenden können, um eine Software zu erstellen oder mit einem externen System zu interagieren.
JSON	[JavaScript Object Notation]: kompaktes Datenformat in einer einfach lesbaren Textform für den Datenaustausch zwischen Anwendungen

1.1.2 Aufbau des Projekts

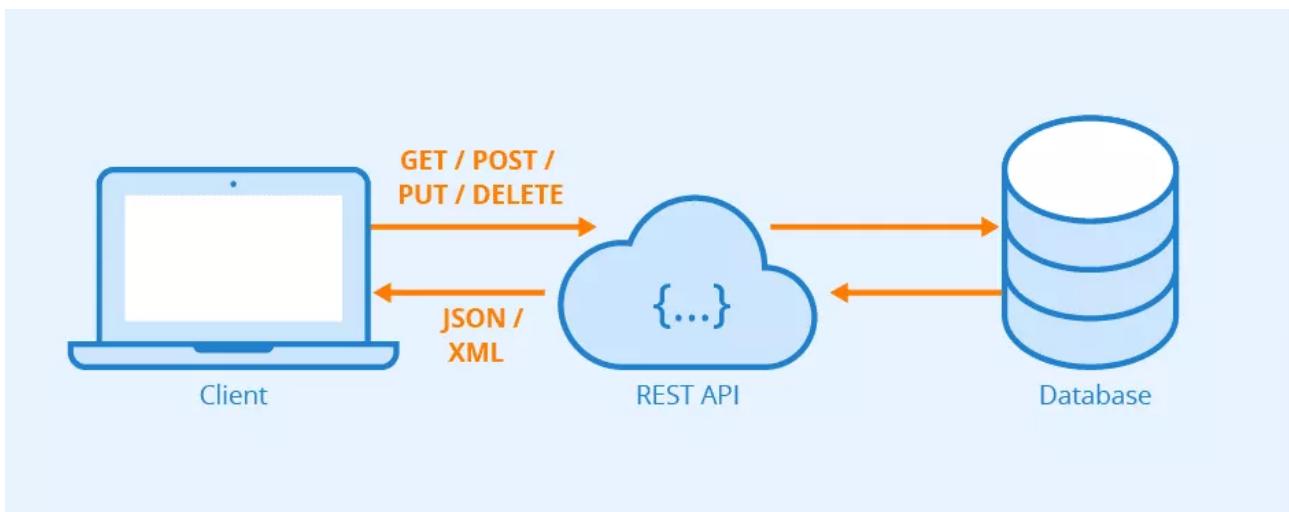
Bevor das Projekt begonnen wird, muss zu allererst überlegt werden, welche Architektur sich für das Projekt am besten eignet. Ziel der Applikation-Entwicklung sollte eine störungsfreie Kommunikation zwischen einer benutzerfreundlichen GUI (Graphical User Interface), also der tatsächlichen Anzeige auf dem Client-PC und einer organisierten Datenbank zur Speicherung der verarbeiteten Daten der Applikation sein.

Grundlegende Ziele der Anwendung:

- Eingabe und Ausgabe von nutzerspezifischen Daten (Noten, Termine und Co.)
- Speicherung der Daten im persistenten Speicher um eine dauerhafte Verfügbarkeit der Daten zu gewährleisten
- Lesen und Überschreiben/ Bearbeiten von vormals eingegebenen Daten

Was ist eine Rest-API ((Representational State Transfer)

- Auch bekannt als RESTful API
- Eine API (eine Programmierschnittstelle), welche der REST-Architektur unterliegt
- Eine API besteht aus mehreren Tools, Definitionen und Protokollen zur Entwicklung und Integration von Anwendungssoftware
- Die API stellt sicher, dass das System mit der Datenstruktur so kommuniziert, dass sie die Anforderung verstehen und erfüllen kann
- Vermittler zwischen Nutzern oder Clients und den von ihnen benötigten Ressourcen oder Webservices
- REST ==> Sammlung von Architekturbeschränkungen
- REST-API überträgt nach einer Anfrage eine Repräsentation des Ressourcenstatus an den Anforderer oder Endpunkt
- Diese Repräsentation werden in einem HTTP-Format übermittelt: JSON (JavaScript Object Notation), HTML, XML, Python, PHP oder Plain-Text



Was zeichnet eine RESTful-API aus?

- Eine aus Clients, Servern und Ressourcen bestehende Client/Server-Architektur, die Anforderungen per HTTP verwaltet
- Zwischen GET-Anforderungen werden keine Client-Informationen gespeichert
- Die einzelnen Anforderungen sind separat und nicht verbunden
- Eine einheitliche Schnittstelle zwischen Komponenten, um Informationen in standardisierter Form zu übertragen

1.1.3 Umsetzung der REST-Architektur

Um die Besonderheiten der RESTful-API Architektur im vorliegenden Projekt umzusetzen, müssen einige Vorübergegangen getroffen werden, welche sich vor allem (noch nicht tiefgreifend) an die benötigten Komponenten (vorerst nur oberflächlich betrachtet) richtet.

- Datenbank: SQL, MySQL, Localhost Server um die Daten der Anwendungen lokal zu speichern und eine persistente Speicherung auf einem Server zu simulieren
- Web-Api: Schnittstelle zwischen der Datenbank-Logik und der grafischen Benutzeroberfläche des Anwenders um Daten zu speichern, zu laden und/ oder zu bearbeiten, wenn der Nutzer dies wünscht
- Frontend: Grafische Benutzeroberfläche um Benutzerfreundlichkeit der Anwendung zu erhöhen

1.2 Praktische Überlegungen an das Projekt

1.2.1 Komponenten und Systeme

Als nächstes folgt die Auswahl und die Überlegungen an die Umsetzbarkeit des Projektes. Hierzu zählen die geeignete Auswahl an Komponenten um ein möglichst gutes Ergebnis gewährleisten zu können. Dabei gliedert sich das ganze in folgende Unterteilungen: die Protokoll-Komponenten und die Entwicklungs-Komponenten. Erster kommen dahingehend zum Einsatz, dass das Projekt dokumentiert, gespeichert, und versioniert werden kann. Letztere dienen ausschließlich der Entwicklung der Applikation.

Kategorie	Komponenten	Vorteile für das Projekt
Protokoll	GIT	<ul style="list-style-type: none">• Software zur verteilten Versionsverwaltung von Dateien
	GITHUB	<ul style="list-style-type: none">• Serverlösung um die Repositorys des Git-Projektes zu speichern und so plattformunabhängig zugänglich zu machen
	Snipping-Tool	<ul style="list-style-type: none">• Zum Anfertigen von Screenshots um verschiedene Abschnitte des Entwicklungsprozesses für die Projektarbeit zu protokollieren
	Pages	<ul style="list-style-type: none">• Apple-Office Anwendung um die Projektarbeit zu schreiben
Entwicklung	MySQLWorkbench	<ul style="list-style-type: none">• [MacOs]• Datenbank-Modellierungswerkzeug, das Datenbankdesign, Modellierung, Erstellung und Bearbeitung von MySQL-Datenbanken in eine Umgebung integriert.
	SQL Server Management Studio	<ul style="list-style-type: none">• [Windows]• Softwareanwendung, die zum Konfigurieren, Verwalten und Verwalten aller Komponenten in Microsoft SQL Server verwendet wird

Kategorie	Komponenten	Vorteile für das Projekt
Allgemeine Komponenten	Visual Studio	<ul style="list-style-type: none"> Von Microsoft bereitgestellte Entwicklungsumgebung zum Programmieren mit höheren Programmiersprachen
	Visual Studio Code	<ul style="list-style-type: none"> Plattformübergreifender Quelltext-Editor zum Schreiben von HTML und CSS Code, sowie von Script-Programmiersprachen wie JavaScript
	Swagger	<ul style="list-style-type: none"> Sammlung von Open-Source-Werkzeugen, um HTTP-Webservices zu entwerfen, zu erstellen, zu dokumentieren und zu nutzen Dient zur Virtualisierung der programmierten Schnittstelle (API) zwischen Server und Code
	Star UML / GitMind	<ul style="list-style-type: none"> [Teils Dokumentation/ Development] Erstellen von Klassendiagrammen
	Structorizer	<ul style="list-style-type: none"> [Teils Dokumentation/ Development] Erstellen von Strukturdiagrammen zum visualisieren und planen von Funktions- / Methodenabläufen
	Node	<ul style="list-style-type: none"> Node.js als plattformübergreifende Open-Source-JavaScript-Laufzeitumgebung Zum ausführen von Javascript Code außerhalb eines Browsers
	npm	<ul style="list-style-type: none"> Paketmanager für die JavaScript-Laufzeitumgebung Node.js
Spezifische Komponenten	C#	<ul style="list-style-type: none"> Höhere Programmiersprache Objektorientierte Programmiersprache zum schreiben der Backend-Logik
	ASP.NET	<ul style="list-style-type: none"> Web Application Framework zum Erstellen von dynamischen Webseiten, Webanwendungen und Webservices Zur Entwicklung der REST-API
	Angular	<ul style="list-style-type: none"> TypeScript-basiertes Front-End-Webapplikationsframework JavaScript-Framework zur Erstellung dynamischer Webanwendungen Zur Entwicklung des Frontend
	TypeScript	<ul style="list-style-type: none"> Stark typisierte Script-Sprache und Framework für JavaScript-Code Enthält Logik für Datenstrukturen, Objekte und Klassen

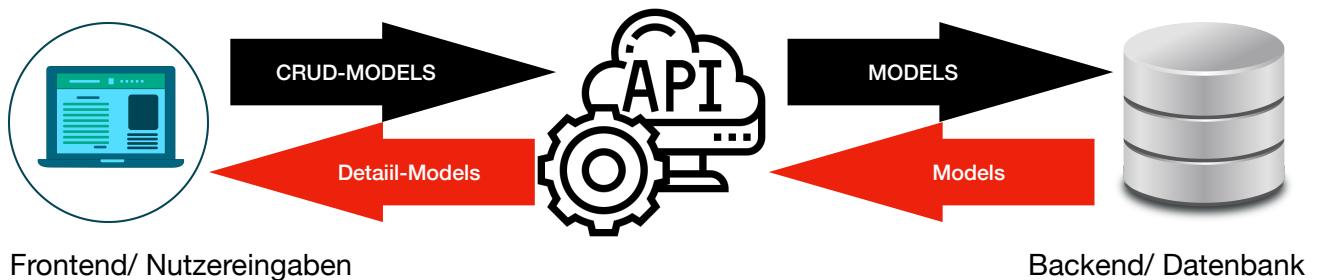
1.2.2 Klassen und Verhalten in der Backend-Api

Bevor es mit der Entwicklung beginnen kann, ist es wichtig zu verstehen wie eine API- im Backend (unter zu Hilfenahme des Frameworks ASP.NET) und ein Frontend mit Angular aufgebaut ist.

Backend-Api

Die Backend-WebApi ist im Groben gesagt eine Schnittstelle zwischen einem Datenbanksystem auf der einen Seite und der zu verwaltenden Programmlogik innerhalb der API, also der WebApplikation. Diese Web-Applikation unterteilt sich in 4 verschiedene, miteinander kommunizierende und (teilweise) voneinander abhängige Komponenten.

- Den **Model-Classes**: Diese repräsentieren die Entitäten der Datenbanken. Im Beispiel des Noten-Applikation Projektes also die Grads (Zensuren/ Notenpunkte), Subjects (Unterrichtsfächer), SchoolYears (Schuljahre), Schools (Schulen) und Holidays (Ferien) ==> die wichtigsten dieser Entitäten für die Verwaltung der Noten in der App sind hier Noten und Fächer, jedoch werden alle anderen Models zum reibungslosen Ablauf der App benötigt
- Den **Services**: Diese bestehen jeweils aus einem Interface, welches die Grund-Logik zur Verfügung stellt und einer davon erbenden Class ==> Diese sind keine Entitäten der Datenbank-Logik, sondern stellen die (für die jeweiligen Klassen) benötigte Verwaltungslogik bereit. Jede Klasse hat also einen Service.
- Die **Details-Classes**: Repräsentieren die reinen Informationen der Entitäten. Da der Nutzer später nicht direkt mit der Datenbank-Logik und seiner Entitäten konfrontiert wird und mit diesen arbeitet, benötigt die API eigene (den Model-Classes ähnliche) Klassen um die Informationen der Datenbanken zu transportieren und dem Nutzer aufzubereiten.
- Den **CRUD-Classes**: Das Akronym CRUD umfasst die vier grundlegenden Operationen persistenter Speicher Create (Datensatz anlegen), Read (Datensatz lesen), Update (Datensatz aktualisieren) und Delete (Datensatz löschen). ==> Diese Klassen der API repräsentieren also Dummys um Einträge in der Datenbank zu verwalten, ohne dass der spätere Endnutzer mit der richtigen Datenbank-Logik und seiner Entitäten konfrontiert wird.



Um nachfolgend einen groben Überblick über das genaue Verhalten der einzelnen Klassen zu erlangen werden diese in sogenannten UML-Klassendiagrammen (Unified Modeling Language) abgebildet. Bei diesem Schritt geht es noch nicht primär um einen genauen Planung der einzelnen Services und Entitäten der Datenbank, sondern vielmehr um einen grobe Planung wichtiger Knotenpunkte.

Doch zunächst bedarf es einer kurzen Einführung in die Thematik der UML-Klassendiagramme um die spezifische Anwendung im vorliegenden Projekt nachvollziehen zu können. Auch hier wird kein genauer Einstieg in das Thema der grafischen Modellierungssprache UML vorgenommen, sondern nur die wichtigsten Eckpunkte der Klassendiagramme erläutert.

Ein Klassendiagramm, wie es im Rohzustand auszusehen hat, besteht aus 3 Bereichen. Dem Kopfbereich, in dem der Name der Klasse (+ optional der Typ) vermerkt ist, dem Bereich für die Attribute, welche die Klasse beschreiben, sowie dem Methoden-Bereich, also der Bereich in dem die klasseneigenen Funktionen beschrieben und aufgeführt werden. Das Klassendiagramm enthält dabei nur einen Übersicht über eben genannte Komponenten und keine tiefgreifende, ausführliche, oder detaillierte Beschreibung, weder zur Beschaffenheit, noch zur Funktion oder Wirkungsweise der Komponenten.

Classname
+ field: type
- field: type
+ method(type): type

- Die Zugriffsmodifikatoren der Attribute der Klasse werden mit + / - /# beschreiben
- + = public / - = private / # = protected
- Gleches zählt für die Methoden
- Variablen werden immer mit Namen + „ : “ und dem Typen

Nachdem nun also ein kurzer Überblick über die einfache Struktur eines solchen Klassendiagrams gegeben wurde, komme ich zu einem Anwendungsbeispiel um die einzelnen Klassen des vorliegenden Projektes zu visualisieren und später konstruieren zu können. Dies werde ich anhand der Subject-Klasse vornehmen, welche als Repräsentation eines Unterrichtsfaches eine zentrale Logik-Einheit der Anwendung darstellt.

1.2.3 Benötigte Komponenten und Klassen

Bevor wir aber nun zur Planung einer speziellen Klasse kommen können, müssen wir noch einmal einen Schritt zurückgehen und vorerst die gesamte Applikation im Vorhinein planen und strukturieren. Das heißt, bevor es mit der Entwicklung einzelner Komponenten beginnen kann, bedarf es einiger grundlegender Überlegungen nach dessen Ergebnissen die Models in der Datenbank und die Klassen in der Backend-API entstehen werden.

Welche Komponenten werden benötigt?

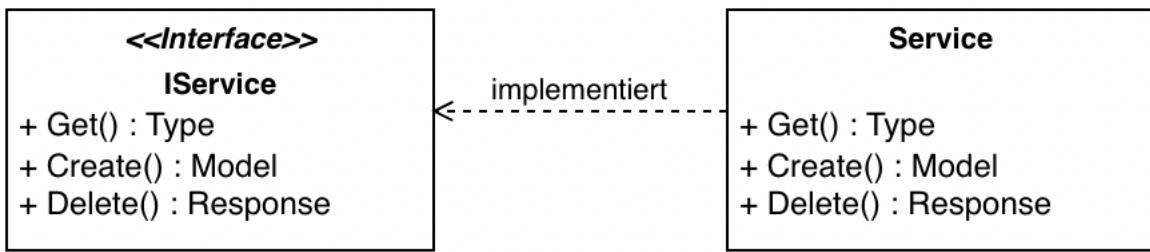
Hauptkomponenten:

- **Zensuren (Grads)** als zentrale Einheit der Noten-App. Diese bilden das Rückgrat, denn der Sinn der fertigen Applikation wird es sein, dass Noten gespeichert, angezeigt, gegengerechnet und geplant werden können.
- **Unterrichtsfächer (Subjects)** als Verwaltungsebene für die Noten. Jede Schulnot wird in einem speziellen Fach erbracht. Jedes Fach bildet später eine Gesamtnote, welche über den allgemeinen Notenschnitt entscheiden wird.
- **Schuljahre (SchoolYears)** als oberste Verwaltungseinheit. Diese werden die spezifischen Unterrichtsfächer, mit all ihren individuellen Noten, bündeln und in einen Kontext stellen. Zudem wird es dem späteren Nutzer so ermöglicht mehrere solcher Verwaltungseinheiten für verschiedene Schuljahre anzulegen und logisch voneinander zu trennen

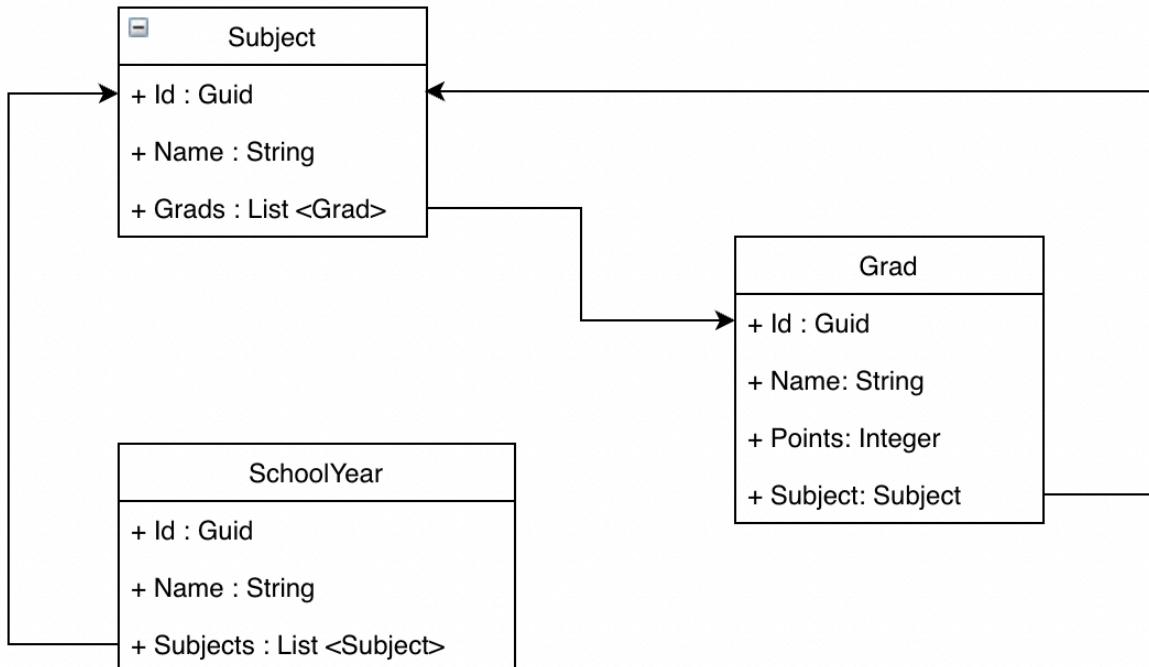
Nebenkomponenten:

- **Schulen (Schools)** erfüllen für die Hauptfunktion der Applikation keinen relevanten nutzen, sondern dienen neben den Schuljahren als zuzügliche Verwaltungsebene für die erbrachten Noten und Leistungen. Zudem werden sie ein Bundesland-String zur Verfügung stellen, anhand dessen in der Datenbank nach den Ferien und freien Tagen im Schuljahr gesucht werden kann.
- **Ferien (Holidays)** dienen lediglich dem organisatorischen Design der Applikation, da diese im Endzustand einen Kalender enthalten wird in dem zukünftige und anstehende Leistungstests hinterlegt werden können, oder sich Deadlines für die Abgabe von Ausarbeitungen gesetzt werden können.

Die beiden Nebenkomponenten dienen in den ersten Stunden der Projektarbeit lediglich einem untergeordneten Zweck und ihre Existenz geht allein auf die Tatsache zurück die Applikation zum einem späteren Zeitpunkt ausbauen zu können und zu erweitern. So könnte die Applikation zu einem späteren Zeitpunkt problemlos um einen Kalender und dazugehörigen Funktionalitäten erweitert werden.



Im obigen Bild ist vereinfacht noch einmal die Beziehung zwischen dem IService-Interface (Präfix nicht zwingend) und der implementierenden Service-Klasse dargestellt.



Im oberen, sehr schlicht gehaltenen, Diagramm wird der Zusammenhang zwischen den drei Hauptklassen ersichtlich. So besitzt jede der Entitäten in der späteren Datenbank mindestens einen Fremdverweis auf eine andere Entität. Diese zusammenhängende Logik, welche in der Datenbanksprache auch als „Entity-Relationship“ bekannt ist, wird es der Applikation erlauben die unterschiedlichen Noten die später einmal getrackt werden sollen in Logikbereiche zu unterteilen und zu gliedern zu speichern.

Nicht nur werden die Entitäten der Datenbank, die in der API später durch die Models repräsentiert werden, in eben jenem Schema zueinander in Verbindung stehen, sondern auch deren Hilfsklassen wie DetailModels und CRUDModels und andere, zum jetzigen Zeitpunkt noch nicht ersichtlichen weiteren Ableitungen der Hauptklassen.

Nachdem die grundlegenden Klassen und Strukturen also geplant wurden, kann es nun daran gehen für jede dieser Klassen ein Cluster aus (für den späteren Betrieb der API) existenziellen Klassen zu erstellen. Jedes dieser Cluster enthält sowohl eine Model-Klasse, eine Details-Klasse, ein- oder mehrere CURD-Klassen, sowie ein Service-Interface, welches die benötigten Funktionen für jedes Model bereitstellt und eine dieses Interface implementierende Service-Klasse, welche die Programmlogik dieses Interfaces verschrieben wird.

In der folgenden tabellarischen Auflistung finden sich die UML-Klassendiagramme für jede Komponente eines solchen Clusters, mit den geplanten Eigenschaften dieser Komponente und einer Erklärung. Diese Auflistung eines solchen Verbunds wird hier (wie oben bereits beschrieben) an der Subject-Class, welche ein Schulfach in der späteren Applikation repräsentieren wird, vorgenommen.

Bereich	Klassendiagramm	Erklärung
Model	<pre> classDiagram class Subject { +Id : Guid +subjectId : String +Name : String +ShortName : String +SchoolYear : SchoolYear +Grads : List<Grads> } </pre>	<p>Die Model-Klasse, welche die Entität des Unterrichtsfaches in der Datenbank vertritt wird durch einen Namen, einen Kurznamen, einer internen Id und einer Datenbank-Id beschrieben.</p> <ul style="list-style-type: none"> Die Datenbank-Id wird nicht von der API vergeben und wird durch einen gesonderten Datentyp (Guid) repräsentiert, welcher eine eindeutige UUID generiert, im Allgemeinen wird dieser jedoch als einfacher String gehandelt Zudem befinden sich in der Model-Class 2 Fremdverweise auf andere Model-Classes (SchoolYear und Grad) Grads speichert eine Liste (ähnlich eines Arrays) der Noten des Faches SchoolYear speichert eine zugehörige Schule zum Schuljahr
Details	<pre> classDiagram class SubjectDetails { +Id : Guid +Name : String +ShortName : String +Grads : List<GradDetails> } </pre>	<p>Die Details-Klasse repräsentiert nur die Daten, welche ein späterer Nutzer per GET-Methode mittels der API von der Datenbank bezieht. Dies sind die wichtigsten Verweise auf die Attribute des Models. (Können noch erweitert werden bei Bedarf)</p> <ul style="list-style-type: none"> Wichtig : Auch hier wird eine Liste an Noten zurückgegeben, da es sich aber um den Nutzer zugängliche Daten handelt werden auch hier nur Details der Noten-Klasse übergeben
CRUD - Create CRUD - Update	<pre> classDiagram class CreateSubjectModel { +SubjectId : String +Name : String +ShortName : String +SchoolYear : Guid +Grads : List<Guid> } </pre> <pre> classDiagram class UpdateSubjectModel { +SubjectId : String +Name : String +ShortName : String +SchoolYear : Guid +Grads : List<Guid> } </pre>	<p>Die CRUD-Model-Klassen sind die „Bauplanklassen“, wenn man so möchte, welche der Nutzer in einer späteren Eingabe mit den gewünschten Werten bestücken wird. Diese werden dann im Service der Klasse gegen Model-Klassen ausgetauscht und gegen die Datenbank geschickt um eine richtige Entität innerhalb dieser zu kreieren.</p> <ul style="list-style-type: none"> In diesem Beispiel (Subject) sind beide CRUD-Klassen baugleich, jedoch ist dies nicht unbedingt vorgesehen und hier rein zufällig, wegen des Nutzens Nicht ersichtlich im Diagramm ist, dass jeweils der String „SubjectId“ [required] erforderlich ist ==> da die eigentliche Id zur Identifizierung erst von der Datenbank erstellt und bereitgestellt wird muss gewährleistet werden, dass es zu keiner Doppeleintragung kommt

Bereich	Klassendiagramm	Erklärung
Service	<pre> classDiagram class SubjectService { + GetAllSubjects() : List<SubjectDetails> + GetSubjectById(subjectId : Guid) : SubjectDetails + GetGradsOfSubject(subjectId : Guid) : List<GradDetails> + CreateSubject(createModel : CreateSubjectModel) : ResponseModel + UpdateSubject(subjectId : Guid, updateModel : UpdateSubjectModel) : ResponseModel + AddGradToSubject(subjectId : Guid, gradId : Guid) : ResponseModel + DeleteSubject(subjectId : Guid) : ResponseModel } </pre>	<p>Der Service, hier als Klasse, erbt seine Methoden aus einem Service-Interface, welches im UML-Diagramm mit den Präfix {{Interface}} über dem eigentlichen Klassennamen erkennbar wäre. Da jedoch inhaltlich keine Unterschiede in unserem Projekt stehen (zumindest auf UML-Ebene) ist hier nur die erbende Klasse dargestellt</p> <ul style="list-style-type: none"> Wichtig : Gewisse POST-, DELETE- und PATCH-Methoden geben per „return“ ein sogenanntes ResponseModel als Rückgabedatentyp wieder Ein ResponseModel ist eine eigene, selbst programmierte Klasse, welche nur die Aufgabe hat zwischen Nutzer und API zu kommunizieren ====> in der nächsten Spalte aufgeschlüsselt.
ResponseModel	<pre> classDiagram class ResponseModel { + IsSuccess : Boolean + Message : String } </pre>	<p>Das ResponseModel als Nachrichtenübermittler zwischen der API und dem Nutzer (hier repräsentiert als Programmierer → Endnutzer erfährt nur durch die Reaktion des Frontend-Codes von der Nachricht des ResponseModels).</p> <ul style="list-style-type: none"> Diese Klasse enthält einen Wahrheitswert (Boolean / Booleschen-Ausdruck), welcher von der API auf den Status der Anfrage gesetzt wird True = Anfrage geglückt / False = Anfrage mit Fehler abgebrochen In dem String „Message“ wird anschließend der FehlerText, oder die Erfolgsmeldung gespeichert Beides wird an den Nutzer gesegnet <hr/> <p>Wird ein neues Subject also beispielsweise erfolgreich in die Datenbank gespeichert, füllt die CreateSubject-Methode also das ResponseModel und gibt dieses an den Nutzer zurück. Der Wahrheitswert würde auf „True“ gesetzt werden und die Nachricht gefüllt</p>

Frontend - GUI

Da sich eine Webapplikation in Angular und Typescript erheblich anders aufbaut als eine Web-API geschrieben in einer höheren Programmiersprache und auch Konzepte wie Klassen, Objekte und Vererbung, in JavaScript nur bedingt und in TypeScript gänzlich anders verhalten als in einer klassischen Objektorientierten Programmiersprache, werde ich an dieser Stelle darauf verzichten die Komponenten der Frontend-Schnittstelle per UML-Klasendiagramm zu erklären. Stattdessen werde ich auf die theoretischen Strukturen der sogenannten Components in Angular eingehen.

Da Angular ein Framework für die Internet-Scriptsprache JavaScript ist, könnte man auf den Gedanken kommen, dass sich das UserInterface aus einer HTML-Datei (für die Struktur der Webseite) und CSS-Dateien (für das Aussehen der HTML-Elemente) zusammensetzt und durch ein Script ergänzt wird, welches die Logik in vorhandene Elemente implementiert. So würde man beispielsweise eine normale Webseite auch aufbauen. In der (meist index.html genannten) Startdatei der Webseite würde man mittels der Auszeichnungssprache HTML verschiedene Komponenten beschreiben (wichtig : nicht programmieren), welche man mit den CSS-Befehlen richtig „stylen“ würde. Im Fußbereich der HTML-Seite würde ein Link zur Script-Datei liegen in der per JavaScript-Code den verschiedenen <button> und <input> Feldern Leben eingehaucht würde. Jedoch wäre diese Art eine Anwendung zu beschreiben zu statisch und unflexibel um einen Webapplikation zu entwickeln, welche direkt auf verschiedene Ereignisse innerhalb dieser reagieren müsste. So eignet sich gerade erwähntes Konzept viel mehr zum erstellen (größtenteils) statischer Webseiten, als zum programmieren einer Webanwendung.

Angular bietet einen leichteren, flexibleren und agileren Art und Weise solch ein Projekt anzugehen.

Das Framework Angular lebt nicht von einer einzelnen HTML-Datei, in welcher die Gesamtheit aller in der App verwendeten Elemente untergebracht ist. Angular spaltet jegliche Logikbereiche der App in einzelne Komponenten (genannt Components) auf, welche aus jeweils einer HTML-Datei, einer CSS-Datei, sowie 2 Logik-Dateien für TypeScript-Code bestehen. Ein Component könnte so beispielsweise das Component mit dem Namen „Subject-Overview“ sein, welches, wie der Name schon suggeriert, nur für einen einzigen Zweck besteht, nämlich lediglich alle vorhandenen Subjects (also Fächer) anzuzeigen. Beispielsweise könnte dies in einer Tabelle geschehen.

In der HTML-Datei des Components würde sein Gerüst beschrieben, welches in seiner CSS-Datei optisch aufgewertet würde. In einer der beiden TypeScript-Dateien würde die Programmlogik für das Component beschrieben werden, also beispielsweise die Kommunikation mit der API um sich erforderliche Daten aus der Datenbank zu holen, sowie die Logik um diese auf der Seite des Components anzuzeigen, zu verändern, zu ordnen, oder ähnliches mit diesen zutun.

Jede Angular-Applikation besitzt darüber hinaus noch eine „Grund-Komponente“ (ebenfalls als Component aufgebaut ==> app.component), in welcher das jeweils aktuelle Component vorladen und angezeigt werden kann, wenn es der Programmfluss erfordert.

Hinweise:

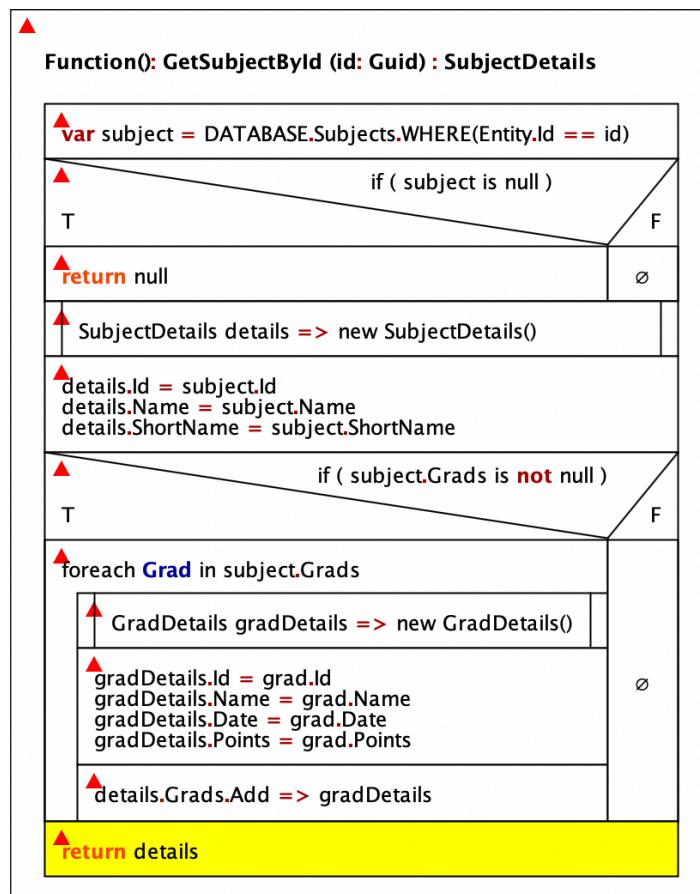
Da die Grundaufgabe dieser Projektarbeit vor allem darin besteht eine Applikation und deren Logik zu programmieren und weniger darin eine möglichst ausgereifte, ansehnliche GUI zu entwerfen, werde ich ein CSS-Framework zu Hilfe nehmen, welches sich bestens zur Verwendung mit Angular eignet, sowie (ohne größeres Geschick in HTML und CSS) eine ansprechende Benutzeroberfläche zu generieren. Gemeint dies das Framework Bootstrap, auch manchmal als Twitter-Bootstrap bezeichnet. Auf Besonderheiten dieses Frameworks gehe ich an geeigneter Stelle tiefer ein.

1.2.4 Funktionen planen - Abläufe strukturieren

Nachdem nun die Klassen, sowie deren Beziehungen untereinander und deren Design, geplant wurden muss sich des Weiteren Gedanken über die Umsetzung der verschiedenen Interface- und Service-Methoden für die Kommunikation mit der API/ Datenbank gemacht werden. Die wichtigsten, sich in jeder Service-Klasse (in unterschiedlicher Ausführung) wiederfindenden, Methoden (CRUD-Methoden) sind jene Funktionen/ Methoden, welche die CRUD-Befehle (also Lesen, Schreiben, Ändern, usw.) in der Datenbank-Logik ausführen. Diese wären:

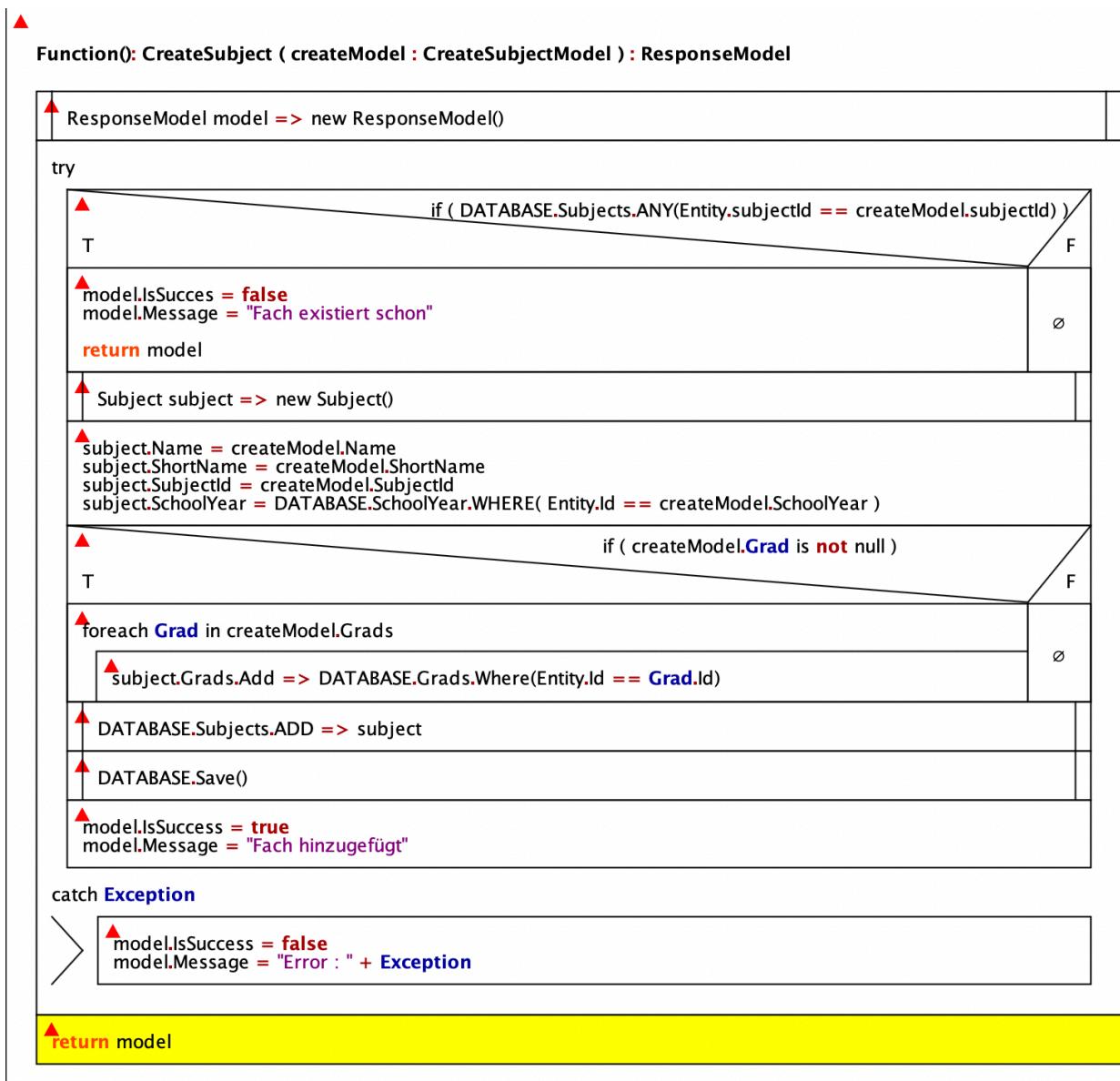
- **GET** -Methoden: Zum Beispiel GetAllOfSomething(), oder auch GetSomethingById(), aber auch GetThisFromThis() ==> diese Methoden rufen Daten über die Schnittstelle (API) vom Server aus der Datenbank ab. Wenn diese Methoden keine Parameter entgegennehmen, dann handelt es sich meist um einen GETALL-Aufruf, welcher lediglich alle zur Verfügung stehenden Daten eines Types zurückgibt. Parameter können diese Rückgaben bei Bedarf filtern und einschränken.
- **CREATE** -Methoden: Zum Beispiel CreateSomething() ==> Erstellen einen Datenbankeintrag am Server und nehmen als Parameter ein so genantes CREATEMODEL entgegen (siehe Absatz 1.2.2) nach dessen Abbild ein Model generiert und eine Entität in der Datenbank angelegt wird.
- **UPDATE** -Methoden: Zum Beispiel UpdateSomething(), oder SetSomethingOnThis(), oder AddSomethingToThis() ==> Suchen einen Datensatz aus der Datenbank am Server (nach Muster welches als Parameter übergeben wird —> beispielsweise der Id), ändern diesen nach dem Vorbild des übergebenen UPDATEMODELS (oder erstellen ihn neu und speichern den Datensatz auf jenem Speicherslot) und updaten die Datenbank an der Stelle des Eintrags
- **DELETE** -Methoden: Zum Beispiel DeleteSomething() ==> Nehmen als Parameter einen Vergleichswert (zum Beispiel eine Id) um die Datenbank nach dem gesuchten Eintrag zu filtern. Löschen den Eintrag an der gefundenen Stelle und speichern anschließend die Datenstruktur wieder ab.

Im folgenden Beispiel veranschauliche ich die Wirkungsweise einer GET-Methode (GetSubjectById()) anhand eines sogenannten Struktogramms, also einen Funktionsablaufplan:



- Der Funktion wird eine Id als Parameter übergeben
 - In einer Variablen wird die Entität aus der Datenbank gespeichert (Datenbankabfrage in der Tabelle Subjects wo die Id übereinstimmt)
 - Eine Abfrage ob ein Wert gefunden wurde, oder die Variable „null“, also leer ist
 - Wenn dies der Fall sein sollte, so wird direkt der Wert null zurückgegeben und die Funktion/Methode verändert
 - Eine Instanz der Klasse Subject-Details wird erstellt —> der geplante Rückgabe-Datentyp
 - Informationen der Entität (des Models) werden in das Detail-Model übernommen
 - Wenn die Entität eine Liste aus Zensuren enthält, dann wird diese übernommen
 - Dafür muss jede Note (Grad) durchlaufen werden —> foreach-Schleife (für jedes Element in ...)
 - Für jedes Element in der Liste wird ein Details-Model erstellt + deren Daten übernommen und schließlich in die Liste des SubjectDetails-Model übernommen
 - Rückgabe des DetailsModels
-

Zunächst noch eine Vorüberlegung um eine CREATE-Methode zu planen und zu programmieren, da sich dies ein wenig komplizierter gestalten dürfte, als eine reine Datenbank-Abfrage. Denn in dieser muss nicht nur ein Model mit einem DetailsModel ausgetauscht und zurückgegeben werden. Vielmehr muss mit einem übergebenen artfremden Model (CREATEMODEL), welches keine Datenbank-Entität widerspiegelt, ein Datenbankeintrag mit einem echten Model angelegt werden. Nachfolgend ein Struktogramm zur Planung der CreateSubject()-Methode:

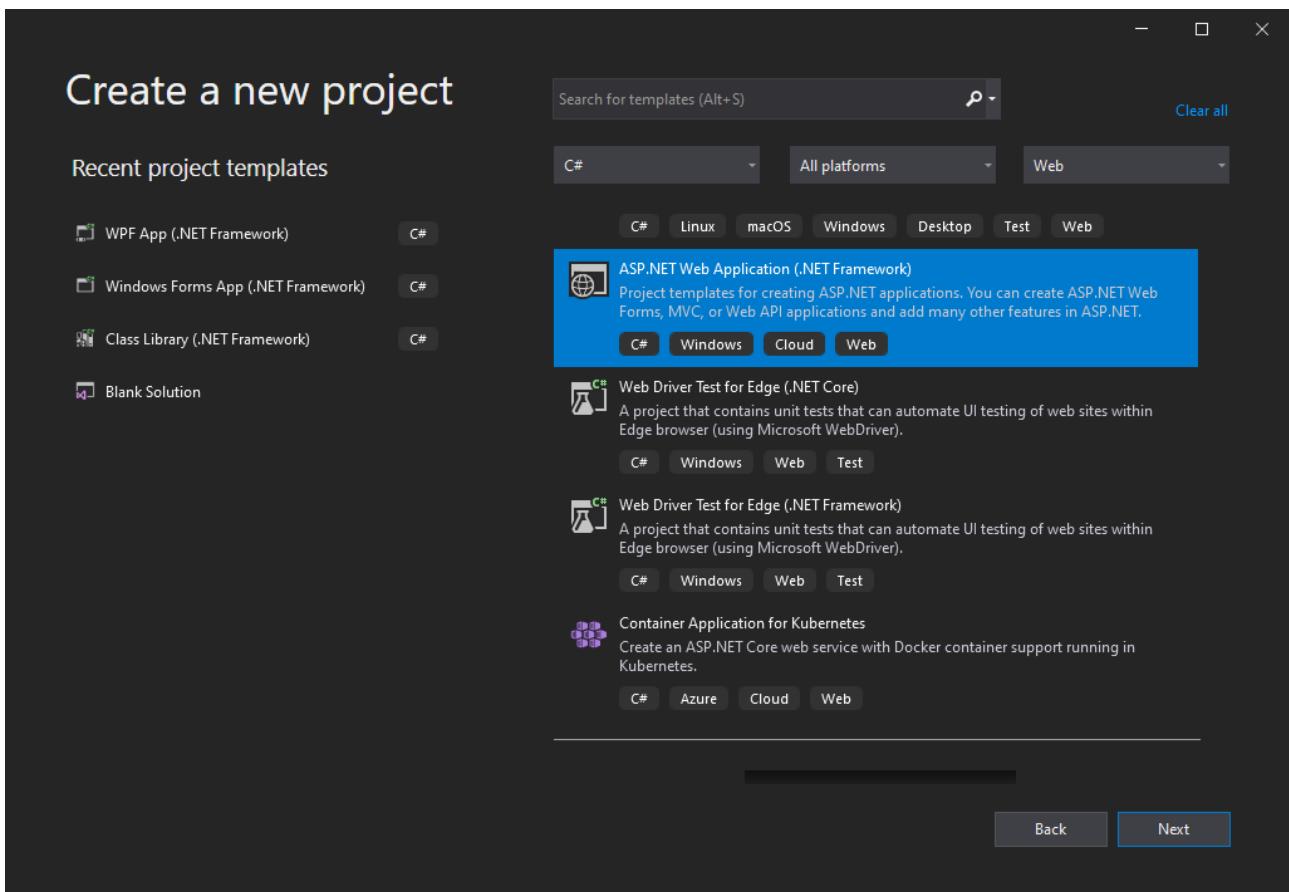


- Objekt aus der Klasse ResponseModel instanziieren
- Prüfung ob es bereits ein Subject mit der (im CREATEMODEL) übermittelten SubjectId gibt —> wenn ja, dann return ResponseModel
- Instanz von Subject-Klasse anlegen um eine Entität in der Datenbank anlegen zu können
- Werte der Entität mit den Werten aus dem CreateModel füllen
- Dabei für jede UUID in der Grads-List des CreateModels die Note anhand der Id aus der Datenbank suchen und in Liste speichern (ggf. mit gesonderten Abfrage ob es Id tatsächlich gibt)
- Datenbank.Add ==> Eintrag anlegen + Datenbank-Änderungen abspeichern
- Umgeben von einem Try-Catch Block (Versucht den Code im Try-Block auszuführen, wenn dies nicht klappt wird die Exception im Catch-Block geworfen und model gefüllt —> Programmabbruch)
- Am Ende wird das ResponseModel gefüllt zurückgegeben

2. Projekt-Verwirklichung

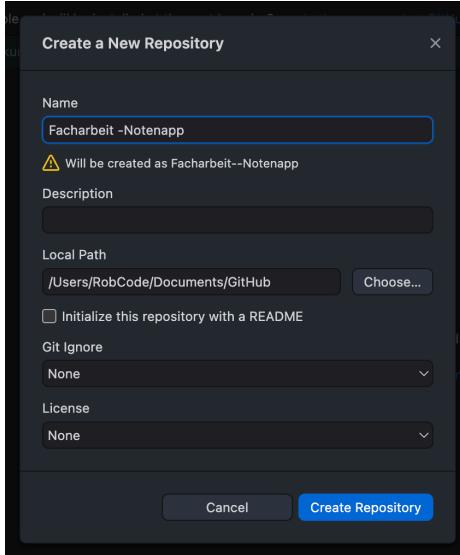
2.1 Backend - WebApi - Programmierung mit C#

2.1.1 Anlegen des Projekts + Einrichtung

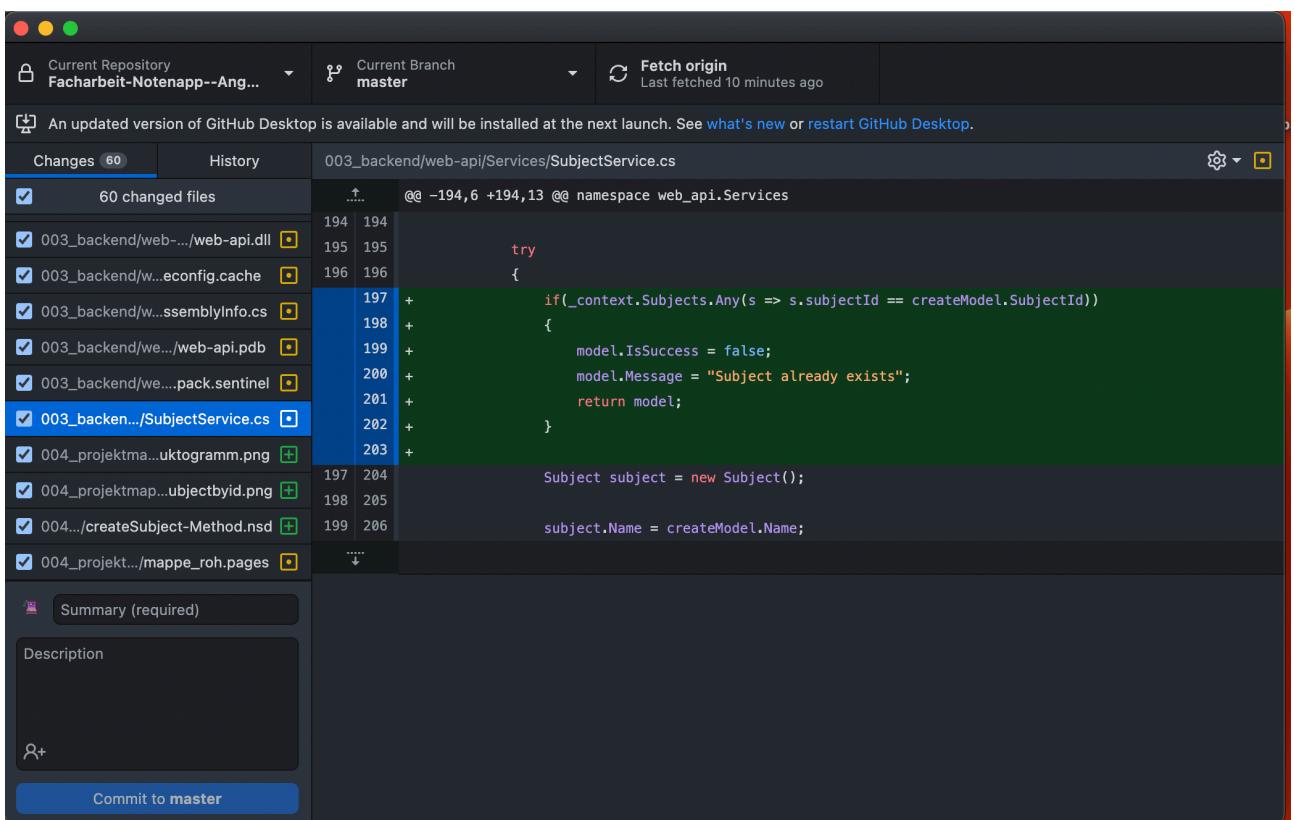


Nun geht es an die tatsächliche Codierung der Web-API. Als IDE (Integrated Development Environment), also als Entwicklungsumgebung für das Projekt Backend-Api wird die von Microsoft angebotene IDE Visual Studio in der Version von 2022 verwendet. Als Framework wird das .NET-Framework in der Version 6.0 verwendet. Es wird ein neues Projekt in Visual Studio 2022 mit dem Framework ASP.NET angelegt. Dazu wird das Template der Web-API verwendet. Dem Projekt wird ein Name vergeben, sowie die grundlegende Ordnerstruktur angelegt. Bevor es nun mit der Programmierung innerhalb der IDE weitergeht werden noch die Vorbereitungen für die Versionierung getroffen. Damit das Projekt jederzeit und auf jedem Endgerät, welches an der Entwicklung beteiligt ist, verfügbar ist, wird ein sogenanntes Git-Repository angelegt und dieses über Github gehostet, damit es in unterschiedlichen Instanzen und von unterschiedlichen Geräten aus bearbeitet, geändert und geupdatet werden kann. Ich werde an dieser Stelle nicht tiefer in das Themengebiet der Versionierung von Software im Allgemeinen, oder Git und Github im Speziellen einsteigen. Dies würde Stoff für eine eigenständige Facharbeiter schaffen.

Angemerkt sei, dass ich im Sinne dieses spezifischen Projektes nicht die vollen Fähigkeiten von Git ausreizen werde und die Versionierung der Software auf ein Minimum beschränken werde. Vielmehr werde ich die Möglichkeiten, welche mir Git und Github bieten lediglich zum „stagen“ und „commiten“ meiner Änderungen am Projekt verwenden, sowie um es über den Github-Server auf möglichst vielen Geräten zugänglich zu machen.



- Im Bild nebenan sieht man einen Ausschnitt der Github-Desktop Applikation, einer GUI - Schnittstelle für die Versionierungssoftware Git
- Dennoch können Git-Projekte, so genannte Repositorys auch über das Kommandozeilen-Tool und Terminal angelegt werden
- Der geeignete Befehl um ein Repository über die Kommandozeile anzulegen ist: git init (auszuführen im Projektordner der getrackt werden soll)
- Innerhalb dieser GUI lässt sich ebenfalls ein Repository anlegen und sämtliche Änderungen an diesem verfolgen, commiten und schließlich pushen
- Ein Push wird ausgeführt, wenn das Repository in seiner aktuellsten Form auf den Server geladen wird
- zieht man hingegen eine aktueller Version des Projektes vom Server in seine lokale Entwicklungsumgebung spricht man vom pull



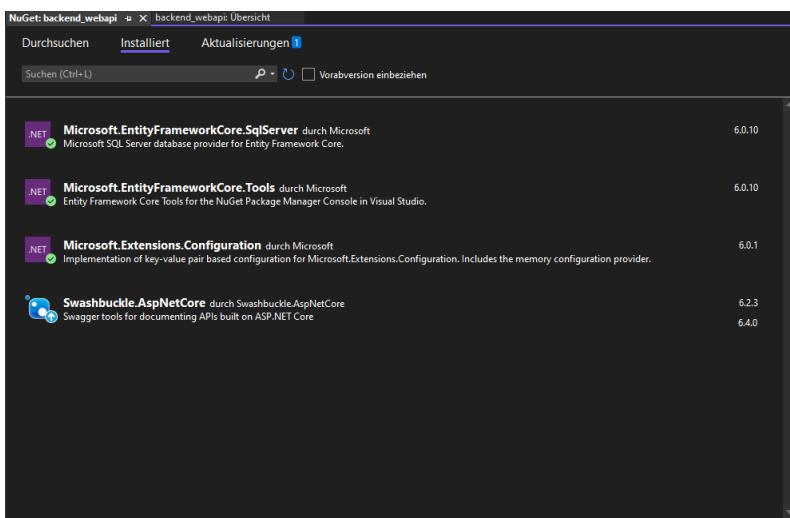
Oben zu sehen ist das Dashboard des Git-GUI-Clients. Im linken Fenster findet sich eine Übersicht aller Änderungen am Projekt seit dem letzten Commit, welche ausgewählt und commited werden können. Rechts im Schaufenster sieht man die Details der ausgewählten Änderung. Grün unterlegt die Parts die seit dem letzten Commit dazukamen, rot unterlegt jene die gelöscht, oder überschrieben/ aussortiert wurden.

Da es jedoch nicht im speziellen darum geht wie Git und die Verwaltung des Projekts NotenApp im Allgemeinen funktioniert, werde ich an dieser Stelle nicht weiter auf die Vorbereitungen des Projekts eingehen. Für eine vollständige Übersicht des Projektes kann die URL (nach meiner Freigabe) verwendet werden: [https://github.com/robCode93/Facharbeit-Notenapp--Angular-und-C-Sharp-]

2.1.2 Installieren der benötigten Pakete

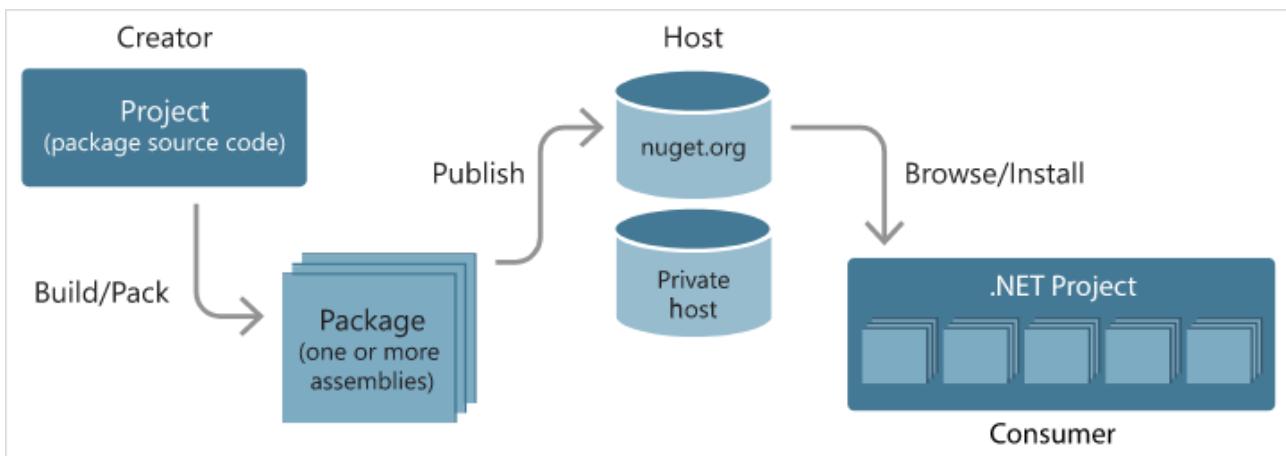
Um besser und übersichtlicher das Projekt strukturieren zu können, sowie um sich eine Menge unnötigen Code sparen zu können, stellt Microsoft Visual Studio mit der Software NuGet, sowie dessen Paketmanager, ein Tool zur Verfügung um das eigene Projekt zu ergänzen und auszubauen.

Unverzichtbar für jede moderne Entwicklungsplattform ist ein Mechanismus, durch den Entwickler nützlichen Code erstellen, freigeben und verwenden können. Solch ein Code wird oft in „Paketen“ gebündelt, in denen kompakter Code zusammen mit anderen Inhalten enthalten ist, die in Projekten benötigt werden, die diese Pakete nutzen. Doch was ist so ein Paket überhaupt? Einfach gesagt ist ein NuGet-Paket eine einzelne ZIP-Datei die fertig kompilierten Code und andere Dateien in dessen Zusammenhang, so wie ein beschreibendes Manifest enthält, in dem Informationen wie die Versionsnummer des Pakets enthalten sind.



- Der NuGet-Paketmanager und seine virtuelle Oberfläche in Visual Studio 2022

Doch wie kommen die Pakete zum Entwickler?



An dieser Stelle kommt der NuGet-Paketmanager (Abbildung oben) zum Einsatz, welcher als Vermittler zwischen Entwicklern auf der einen Seite und den Paket-Entwicklern auf der anderen Seite fungiert. Als öffentlicher Host verwaltet NuGet selbst das zentrale Repository von über 100.000 eindeutigen Paketen.

Dies reicht als Überblick über Pakete, dem Paketmanager und deren Verhalten vollkommen aus um im Zusammenhang mit diesem Projekt den Einsatz eben solcher verstehen zu können.

Natürlich ist NuGet nicht der einzige Paketmanager, so gibt es auch den npm (Node Package Manager) von Node, oder Snap für Linux und dutzende weitere.

Doch welche Pakete werden für die Umsetzung des Projektes „Noten-App“ gebraucht? Wie heißen diese? Und welches Verhalten, welche Neuerungen und/ oder welche Erleichterungen stellen sie dem Code im Projekt zur Verfügung?

- **Microsoft.EntityFrameworkCore.SqlServer** ==> stellt Funktionen und Möglichkeiten für das EntityFrameworkCore zur Verfügung um mit SQL-Datenbanken zu kommunizieren, dies zu erleichtern und zu verbessern
- **Microsoft.EntityFrameworkCore.Tools** ==> weitere Tools um besser mit Datenbanken kommunizieren zu können. Stellt Befehle wie *Add-Migration* (um eine Migration für eine Datenbank zu erstellen) oder *Update-Database* (um die Datenbank nach der aktuellen Migration anzupassen und zu aktualisieren) und viele andere nützliche Befehle und Commands zur Verfügung
- **Microsoft.Extensions.Configuration** ==> stellt weitere Konfigurationen zur Verfügung (in Key-Value Paaren), sowie einen Memory Provider zur Verfügung.

Um einen genaueren Überblick über die Pakete zu verschaffen lohnt es sich die ReadMe-Datei und die Dokumentation auf GitHub und/oder der Webseite des Pakets durchzulesen. Nachdem nun also die Pakete installiert wurden kann es endlich mit der tatsächlichen Programmierung beginnen.

2.1.3 Die Model-Klassen anlegen

In diesem Schritt werden nun also die Model-Klassen für das Projekt angelegt. Zur Erinnerung: Die Model-Klassen dienen zur Verkörperung der Entitäten in der späteren Datenbank im Code der API. Sie stellen die Grund-Attribute der Datenbankeinträge zur Verfügung.

Im Bild oben sehen wir die typisch für Visual Studio (im blau-grün-lastigen Farbschema gehaltene) Entwicklungsumgebung. Die Programmiersprache die verwendet wurde ist C#.

```
1  using System.ComponentModel.DataAnnotations;
2  using web_api.Models.DetailModels;
3
4  namespace web_api.Models
5  {
6      public class Subject
7      {
8          [Key]
9          public Guid Id { get; set; }
10         public string subjectId { get; set; }
11         public string? Name { get; set; }
12         public string? ShortName { get; set; }
13         public SchoolYear? SchoolYear { get; set; }
14
15         // Listen der Klasse
16         public List<Grad>? Grads { get; set; } = new List<Grad>();
17     }
18 }
19
```

Jede Klasse, welche in C# angelegt wird bekommt einen eindeutigen **Namespace**. Namespaces werden dazu verwendet, um den Code in logischen Gruppen zu organisieren und eventuell auftretende Namenskonflikte zu vermeiden. So kann es durchaus in der Applikation vorkommen, dass zwei gleichgenante Klassen auftreten können, sind diese jedoch in verschiedenen Namespaces verwaltet sieht der Compiler später kein Problem. Jedoch ist es dringend davon abzuraten solch eine Gleichnennung zu praktizieren, da es hin und wider zu verwirrenden und

fehlerhaften Ergebnissen kommen kann, wenn man die Flasche Klasse einbindet, weil man sich nur deren Namen, nicht jedoch ihren Namespace gemerkt hat. Dies kann schneller der Fall sein als einem lieb sein durfte, denn Visual Studio nimmt dem Programmierer bei der Einbindung der sogenannten „using“- Direktiven eine Menge Arbeit ab. Einmal automatisch den falschen Namespace eingebunden wäre das Problem da und die falsche Klasse mit gleichem Namen angesprochen.

Da ich schon gerade beim Thema „**using**“-**Direktive** war: Diese werden oberhalb (quasi als Kopfzeile) in den Code der jeweiligen Klasse eingebunden und repräsentieren die Einbindung von anderen Namespaces, Klassen, Paketen aus dem NuGet- und Microsoft Paketdienst, oder Funktionen. Sie werden mit dem Schlüsselwort „using“ eingeleitet und per Punktnotation von ihren einzelnen Segmenten logisch getrennt. Microsoft legt beim Erstellen einer neuen Klasse diese automatisch an, sodass der Entwickler nur die zusätzlich nötigen Direktiven von Hand einbinden muss.

Innerhalb des Namespace wird die **Class** selbst beschrieben. Diese besteht aus Attributen (welche die späteren Instanzen/ Objekte dieser beschreiben) und Funktionen (in Klassen Methoden genannt), sowie einem Konstruktor (und ab und an einen Destruktor), einer Art Funktion die beschreibt wie das Objekt, also die spätere Instanz der Klasse erstellt wird (Destruktor ==> beim Vernichten der Instanz). Ist kein Konstruktor händisch angegeben benutzt der Compiler den typischen, universalen Konstruktor für Klassen.

In unserem Falle ist dies alles nicht nötig, da keine direkten Instanzen der Klassen innerhalb erstellt werden müssen die eigenen Funktionen bereitstellen. Lediglich die Attribute, welche die Instanzen der Klasse beschreiben werden so von Nöten sein. Diese wären im vorliegenden Model:

- Id, welche die [Key]-Eigenschaft übergeordnet hat. Dies beschreibt dem Compiler später, dass es sich bei dieser Eigenschaft später um den Primärschlüssel der Entität in der SQL-Datenbank handeln wird. Diese ist durch einen „String“, im besonderen Falle durch eine UUID gekennzeichnet und besitzt den Datentyp Guid in C#
- Wir sehen die Eigenschaften Name, Kurznamen und auch einen Fremdverweis für eine Schule
- Zudem besitzt die Klasse eine Liste aus Grads (also Zensuren)

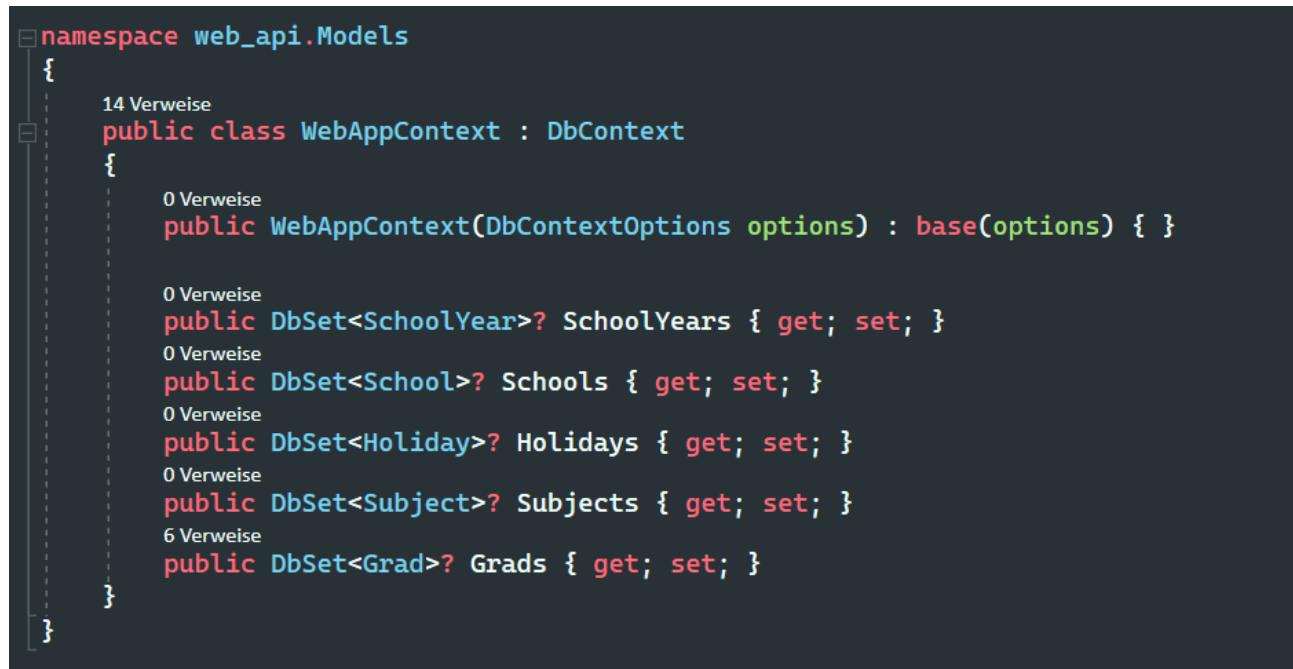
Kurzer Exkurs zum Thema UUID und Guid

dc285007-7805-4d53-8bbe-09fc6fce1d25

Hier sehen wir eine typische UUID, eine (in einer Struktur - z.B. einer Datenbank) eindeutige Kennung für einen Wert. Eine „Universally Unique Identifier“ (UUID) ist eine 128-Bit-Zahl, welche zur Identifikation von Informationen in Computersystemen verwendet wird. Diese besteht aus einem 8 Zeichen langen Bereich, darauffolgenden drei 4 Zeichen langen Bereichen und einem 12 Zeichen langen Bereich. Bei der Generierung nach den Standardmethoden können UUIDs für praktische Zwecke als global eindeutig angenommen werden. Die Wahrscheinlichkeit, dass eine UUID kopiert wird ist zwar nicht null, jedoch so verschwindend gering, dass es zu vernachlässigen ist. Besonders in kleinen Projekten wie diesem hier ist es kaum nötig, dass auf eine Duplizierung geprüft werden muss. Zudem wird die spätere Datenbank diese automatisiert anlegen und der Nutzer und auch der Programmierer werden nicht selbst diese anlegen müssen. Dadurch, dass die Datenbank diesen String also selbst (und zwar beim Entitätseintrag in die Datenbank) die UUID anlegt können Fehler ausgeschlossen werden.

2.1.4 Die Datenbank vorbereiten

Im Nächsten Schritt, bevor es mit der Programmierung des Clusters rund um die Entitäten weitergehen kann, muss die Datenbank und deren Anbindung an die WebApi sichergestellt werden. Dazu sind einige Schritte erforderlich.



```
namespace web_api.Models
{
    14 Verweise
    public class WebAppContext : DbContext
    {
        0 Verweise
        public DbSet<SchoolYear>? SchoolYears { get; set; }

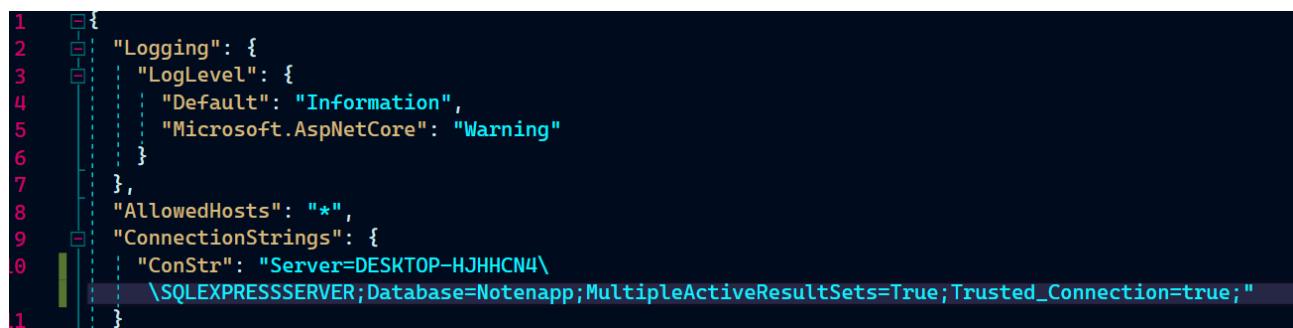
        0 Verweise
        public DbSet<School>? Schools { get; set; }

        0 Verweise
        public DbSet<Holiday>? Holidays { get; set; }

        0 Verweise
        public DbSet<Subject>? Subjects { get; set; }

        6 Verweise
        public DbSet<Grad>? Grads { get; set; }
    }
}
```

In dem Ordner für die Model-Klassen muss zu erst eine Context-Klasse angelegt werden. Diese dient als zentraler Knotenpunkt der WebApi und seiner Verbindung zur späteren SQL-Datenbank. Die Context-Klasse erbt von der DbContext-Klasse und daraus ihre erforderlichen Eigenschaften zum Verwalten der Datenbank. Der Konstruktor (immer mit Zugriffsmodifikator `public` und dem Namen der Klasse beschrieben) erbt den Konstruktor seiner Eltern-Klasse DbContext. Als Parameter werden die Optionen für den Datenbank-Kontext übergeben. Anschließend wird für jedes Model, welches später eine Entität in der Datenbank begründen soll, ein DbSet, also ein Datenbanksatz angelegt vom Typ des Models mit einem freiwählbaren Namen. Hierbei eignen sich vor allem die Namen der Models um Verwirrung und Verwechslung vorzubeugen und die Lesbarkeit des Codes zu gewährleisten. Das ? Hinter dem Datentyp gibt an, dass dieser „null“ sein darf, also leer (ein ! würde suggerieren, dass alles vor dem Zeichen gesetzt sein wird, also nicht null sein wird). Die Getter (get) und Setter (set) beschreiben, dass der Datensatz sowohl „gesetzt“ also geschrieben werden kann, als auch abgerufen werden kann (get = bekommen).



```
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      },
8      "AllowedHosts": "*",
9      "ConnectionStrings": {
10         "ConStr": "Server=DESKTOP-HJHHCN4\\
11             \SQLEXPRESSSERVER;Database=Notenapp;MultipleActiveResultSets=True;Trusted_Connection=true;"
12     }
13 }
```

Nun wird der sogenannte Connection-String gesetzt werden. Dieser wird in der Datei `appSettings.json` gespeicherter und ist eine Datei im JSON (JavaScript Object Notation) Format. Der Connection-String enthält einen Key (Schlüssel = „ConStr“) und einen dazugehörigen Value (Wert) versehen. Im Wert werden die Parameter für die Datenbankkonektivität beschrieben. Dazu gehören der Servername, den MARS (Multiple Active Resultsets), welcher das Ausführen mehrerer

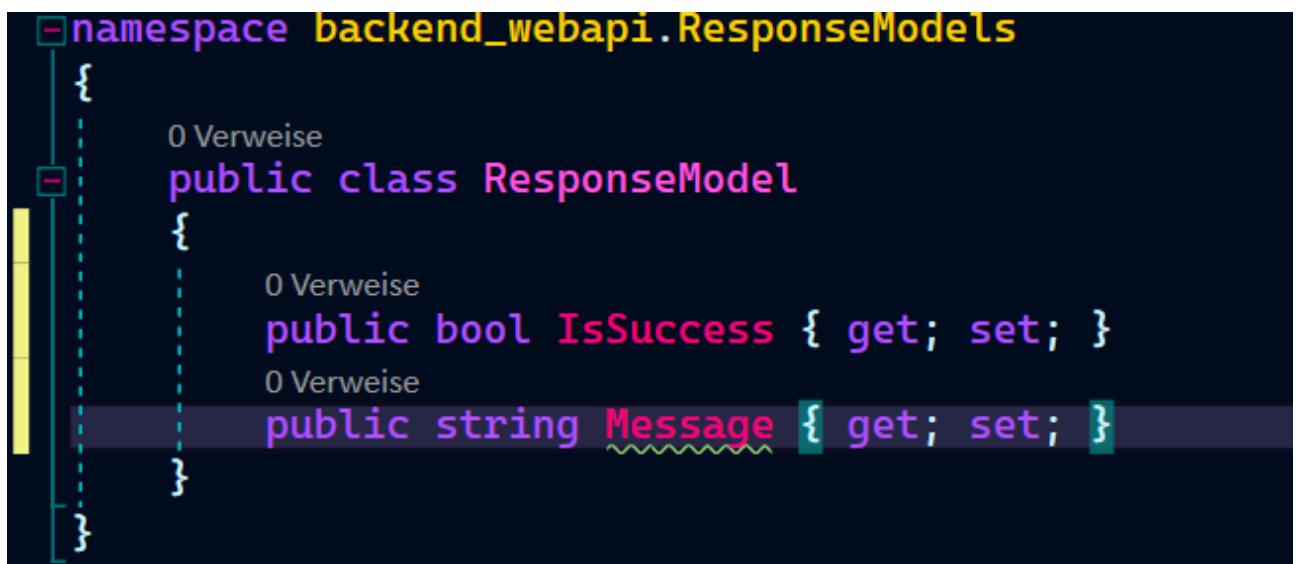
Batches über eine einzelne Verbindung ermöglicht. Und dass die Verbindung als vertrauenswürdig eingestuft wird.

Anschließend wird in der Startdatei der WebApi (Program.cs) der Datenbank-Kontext mit dem Connection-String verknüpft um eine Verbindung zur Datenbank herstellen zu können.

```
// Add DbConnection for connection with the sql-database and the server
builder.Services.AddDbContext<ApplicationContext>(x => x.UseSqlServer
    (builder.Configuration.GetConnectionString("ConStr")));
```

2.1.5 Das ResponseModel - Schnittstelle zum Nutzer

Bevor es mit der Programmierung der eigentlichen Klassen weitergehen kann, muss noch die Informationsschnittstelle zum Anwender implementiert werden. Diese wird in Form der „ResponseModel“ Klasse angelegt (wie schon weiter vorn erwähnt). Diese wird immer dann gefüllt und an den Nutzer übermittelt, wenn ein HTTP-Request erfolgt, also beispielsweise Daten aus der Datenbank per Get-Methode abgerufen werden.



The screenshot shows a code editor with a dark theme. It displays a single file named 'ResponseModel.cs' under the namespace 'backend_webapi.ResponseModels'. The code defines a public class 'ResponseModel' with two properties: 'IsSuccess' (a boolean) and 'Message' (a string). The 'Message' property is highlighted with a red squiggle underline, indicating a potential error or warning in the code editor.

```
namespace backend_webapi.ResponseModels
{
    public class ResponseModel
    {
        public bool IsSuccess { get; set; }
        public string Message { get; set; }
    }
}
```

Eine einfacher Architektur der Klasse, jedoch sind die beiden Attribute der Klasse auch absolut ausreichend. Im „IsSuccess“ Attribut wird lediglich der Status des Request zur Api gespeichert. Dieser kann entweder erfolgreich (true) oder fehlgeschlagen (false) sein. Deshalb eignet sich hierfür ein Boolescher-Ausdruck (ein Wahrheitswert) am besten.

Im „Message“ String wird die Nachricht an den Nutzer gespeichert. Also entweder ein Text (auf Englisch um international verständlich zu sein) mit der Statusmeldung, oder auch (z.B. beim Fehlschlagen des Request) eine Exception (ein Fehler/ nicht geplantes Ereignis) die beim Fehlschlag einer HTTP-Anfrage geworfen wird.

2.1.6 Der Service (Interface und Klasse)

Als nächstes kümmern wir uns um die Methoden der einzelnen Entitäten/ Models. Die Methoden, oder auch Funktionen, die benötigt werden um Einträge in der Datenbank anzulegen, abzurufen, zu verändern, oder auch bei Bedarf aus dieser zu entfernen. Diese Services werden in einem Interface angelegt, welches als „Bauplan“ für die späteren Klassen dienen wird, welche dieses Interface implementieren werden.

```
1  using web_api.Models;
2  using web_api.Models.DetailModels;
3  using web_api.CRUDModels;
4
5  namespace web_api.Services.ServiceInterfaces
6  {
7      public interface ISubjectService
8      {
9          SubjectDetails GetSubjectById(Guid subjectId);
10         List<SubjectDetails> GetAllSubjects();
11         List<GradDetails> GetGradsOfSubject(Guid subjectId);
12
13         ResponseModel CreateSubject(CreateSubjectModel createModel);
14         ResponseModel UpdateSubject(Guid subjectId, UpdateSubjectModel updateModel);
15
16         ResponseModel AddGradToSubject(Guid subjectId, Guid gradId);
17
18         ResponseModel DeleteSubject(Guid subjectId);
19     }
20 }
21
```

In dem gezeigten Beispiel ist das Service-Interface für den SubjectService zu sehen. Dieses gibt vor, dass eine Klasse, welche dieses Interface implementieren wird (also der eigentliche Service), bestimmt Methoden zur Verfügung stellen muss. In diesem Falle verschiedene GET-Methoden (Präfix Get), welche Daten aus der Datenbank abfragen werden, POST-Methoden (Präfix Create) um einen Eintrag in der Datenbank anzulegen, PATCH-Methoden (Präfix Update, oder Add) um Einträge zu verändern und zu bearbeiten, sowie diese zu aktualisieren und DELETE-Methoden (Präfix Delete) um Einträge bei Bedarf aus der Datenbank zu löschen.

Wichtig: Notiert werden im Interface lediglich die Methodennamen, der vorangestellte Rückgabedatentyp, welchen die Rückgabewerte haben sollten (void bei keinem) und die Parameterliste in den runden Klammern. Diese Parameter, die der Methode übergeben werden können werden mit einem Datentyp und einem Namen beschrieben.

2.1.6 Abhängigkeiten zwischen Interface und Klasse herstellen

Nun müssen, in der Startdatei der Applikation, noch die Dependencies, also die Abhängigkeiten zwischen den Service-Interfaces und den Service-Klassen bereitgestellt werden. Das bedeutet, dass wann auch immer eine Funktion aus eben so einem Interface aufgerufen wird, die Logik der Klasse aufgerufen wird um die Service-Methode auszuführen. Dies ist nötig, da sich die Services nicht selbst über die Klasse und so einer Instanz aufrufen werden, sondern über Swagger und den Service-Controllern. Dies bedeutet im Umkehrschluss, dass es keine Objektinstanzen von den Service-Klassen im Programm geben wird, welche nötig wären um einen Funktion aus der Klasse aufzurufen, sondern direkt über das Interface auf die Methoden zugegriffen wird.

```
// Dependencies for IService-Interfaces and Service-Classes
builder.Services.AddScoped<ISchoolYearService, SchoolYearService>();
builder.Services.AddScoped<ISubjectService, SubjectService>();
builder.Services.AddScoped<IGradService, GradService>();
```

2.1.7 Die Service-Klasse (Implementierung des Interface)

Nun wird die Klasse geschrieben, welche das Service-Interface implementieren wird. Dazu schauen wir uns an wie diese Klasse aufgebaut werden muss um einen reibungslosen Ablauf für die darin befindlichen Methoden zu gewährleisten.

Die Service-Klasse (dabei ist es egal welche Service-Klasse => hier am Beispiel der SubjectService-Klasse) enthält ein Attribut vom Typen WebAppContext. Dies ist der Datenbankkontext, welchen wir selbst angelegt haben und in dem die Datenbanksätze der verschiedenen Models, sowie die Datenbankkontext-Optionen gespeichert sind. Dieses Attribut macht dem Service die Datenbankstruktur zugänglich um mit den Werten dieser zu arbeiten. Nur so können später Einträge angelegt werden, gelöscht werden, oder auch bearbeitet und verändert werden. In jeder einzelnen Serviceklasse wird der gesamte Kontext der Datenbank vermerkt, damit auch jede Klasse einen Zugriff auf eben jede Strukturen in dieser Datenbank erhalten.

```
using web_api.Models.DetailModels;
using web_api.Services.ServiceInterfaces;

namespace web_api.Services
{
    public class SubjectService : ISubjectService
    {

        WebAppContext _context;

        public SubjectService(WebAppContext context)
        {
            _context = context;
        }
    }
}
```

Im Konstruktor der Klasse wird (bei der Initialisierung und Erstellung des Serviceklassen-Objekts) wird ein übergebener Datenbankkontext auf das Klassen-Attribut gesetzt. Diese Instanzen werden später nicht vom Programm der API selbst, sondern über das Frontend erzeugt werden.

Im nächsten Schritt schauen wir uns die Implementierung einer der Servicefunktionen einmal genauer an.

```

public List<SubjectDetails>? GetAllSubjects()
{
    List<SubjectDetails> detailsList = new List<SubjectDetails>();

    var allSubjects = _context.Subjects.Include(s => s.Grads).ToList();

    foreach(Subject subject in allSubjects)
    {
        SubjectDetails details = new SubjectDetails();
        details.Id = subject.Id;
        details.Name = subject.Name;
        details.ShortName = subject.ShortName;

        if(subject.Grads != null)
        {
            foreach (Grad grad in subject.Grads)
            {
                GradDetails gradDetails = new GradDetails();
                gradDetails.Id = grad.Id;
                gradDetails.Name = grad.Name;
                gradDetails.Date = grad.Date;
                gradDetails.Points = grad.Points;

                details.Grads.Add(gradDetails);
            }
        }

        detailsList.Add(details);
    }
}

return detailsList;
}

```

Dies ist das Beispiel einer Implementierung einer der Service-Methoden. Hier ist es eine GET-Methode mit dem Namen GetAllSubjects(). Diese Methode soll später im Programm eine Liste aus Subjects (Detailmodels) zurückgeben, nämlich aller vorhanden im Kontext ihres Aufrufs.

- Im ersten Schritt wird eine neue leere Liste vom Typ SubjectDetails angelegt. In dieser werden alle Details der einzelnen Fächer, die abgefragt werden gespeichert werden und zurückgegeben werden.
- Als nächstes speichert die Methode alle, in der Datenbank, gefundenen Subjects (Models) in einer Variable. Hierfür könnte als Datentyp auch eine Liste aus SubjectModels verwendet werden. Jedoch verwenden wir das Schlüsselwort „var“, sodass sich der Compiler selbst den benötigten Datentyp generieren kann. Dies ist unter anderem dabei hilfreich, wenn man nicht genau weiß welche Daten tatsächlich von der Anfrage zurückkommen.
- Ausgeführt wird die Datenbankabfrage in LINQ-Code. LINQ ist ein programmtechnisches Verfahren von Microsoft zum Zugriff auf Daten aus einer Datenbank und ähnelt (stark vereinfacht) normalen SQL-Anweisungen ==> Im Context in der Tabelle Subjects alle Daten in einer Liste abholen (ToList()) inklusive der Grads im jeweiligen Subject (Include())
- Danach durchläuft die Methode eine foreach-Schleife, bei der jedes Objekt in der Liste (hier könnte auch wieder var verwendet werden) mit einem Details-Model ausgetauscht wird.

- Zuzüglich befindet sich noch eine weitere if-Abfrage in der Schleife. Diese fragt ab, ob Noten im gerade durchlaufenen Objekt vorhanden sind. Wenn ja, dann wird für jede Note (wieder foreach-Schleife) ein GradDetails-Objekt angelegt und die benötigten Daten darin übertragen.
- Anschließend wird dieses GradDetails-Model zur GradDetails-Liste im SubjectDetails-Model hinzugefügt. Das fertige Objekt enthält so also auch alle Eigenschaften der tatsächlichen Entität, lassen aber keinen Rückschluss auf die Datenbank zu, welchen der Nutzer später nicht benötigen wird. Ebenfalls wird so Rekursionen vorgebeugt, also Zirkelschlüssen, welche entstehen würden, wenn versucht würde eine tatsächliche Entität der Datenbank im JSON-File eines HTTP-Request unterzubringen. Da jedes Subject eine Liste aus Grads besitzt in der Grads gespeichert sind, die wiederum ein Subject besitzen, würde es zu einer unendlichen langen Verkettung von Abhängigkeiten kommen, welche nicht in einem JSON-File abgebildet werden könnten.
- Anschließend wird der zu anfangs erzeugten detailsList das fertig zusammenbaute SubjectDetails-Model hinzugefügt
- Ist das letzte DetailsModel fertig und in der Liste wird diese per return an den Nutzer zurückgegeben, gibt es keine Subjects kommt eine leere Liste an den Nutzer zurück. So erhält der Auftraggeber immer, wenn dieser Request ausgeführt wird einen genauen Überblick darüber wie viele Entitäten sich in der Liste befinden.

```

public ResponseModel CreateSubject(CreateSubjectModel createModel)
{
    ResponseModel model = new ResponseModel();

    try
    {
        if(_context.Subjects.Any(s => s.subjectId == createModel.SubjectId))
        {
            model.IsSuccess = false;
            model.Message = "Subject already exists";
            return model;
        }

        Subject subject = new Subject();

        subject.Name = createModel.Name;
        subject.ShortName = createModel.ShortName;
        subject.subjectId = createModel.SubjectId;
        subject.SchoolYear = _context.SchoolYears.FirstOrDefault(sy => sy.Id == createModel.SchoolYear);

        if(createModel.Grads != null)
        {
            foreach(Guid gradId in createModel.Grads)
            {
                subject.Grads.Add(_context.Grads.FirstOrDefault(g => g.Id == gradId));
            }
        }

        _context.Subjects.Add(subject);
        _context.SaveChanges();

        model.IsSuccess = true;
        model.Message = "Subject added successfully";
    }
    catch(Exception ex)
    {
        model.IsSuccess = false;
        model.Message = "Error : " + ex.Message;
    }
}

return model;
}

```

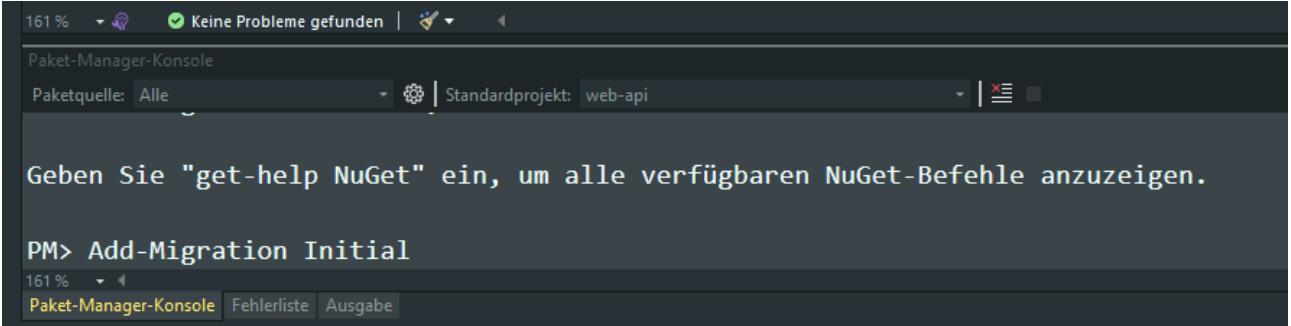
In diesem Bild zu sehen ist eine Service-Methode um ein neues Subject anzulegen. Hierbei wird der Methode ein CREATEModel übergeben, welches im Inneren mit einem Model ausgetauscht wird. All das findet innerhalb eines Try-Catch-Konstruktes statt, welches eine Exception wirft, wenn innerhalb des Try-Blocks etwas schiefgehen sollte. Auch zu sehen ist, dass hierbei kompliziertere LINQ-Anweisungen zum Einsatz kommen, beispielsweise ANY() eine Methode die

jedes Elemente in einer Struktur auf eine Bedingung prüft. Diese Bedingung wird per Lambda-Ausdruck an die ANY() Funktion übergeben. Im oberen Abschnitt wird so beispielsweise jedes Elementen, welches in der Variable „s“ zwischengespeichert wird auf seine Id geprüft und ob diese mit einer übergebenen Id übereinstimmt. Mit anderen Worten wird in diesem Ausdruck also lediglich geprüft ob in der Struktur ein Element mit eben dieser Id vorhanden ist. Diese Funktion liefert innerhalb des If-Statements einen Wahrheitswert zurück (ja oder nein ==> true oder false). Der Befehl FirstOrDefault() gibt an, dass das erste Element oder, wenn nicht vorhanden default (meist null) zurückgegeben werden soll. Das Pendant dazu wäre SingleOrDefault() für ein Element oder keines.

Am Ende wird per Add() ein Datensatz in die Datenbank in der ausgewählten Tabelle hinzugefügt und mit SaveChanges() werden alle Änderungen an dieser gespeichert.

2.1.8 Datenbank-Aktionen ausführen

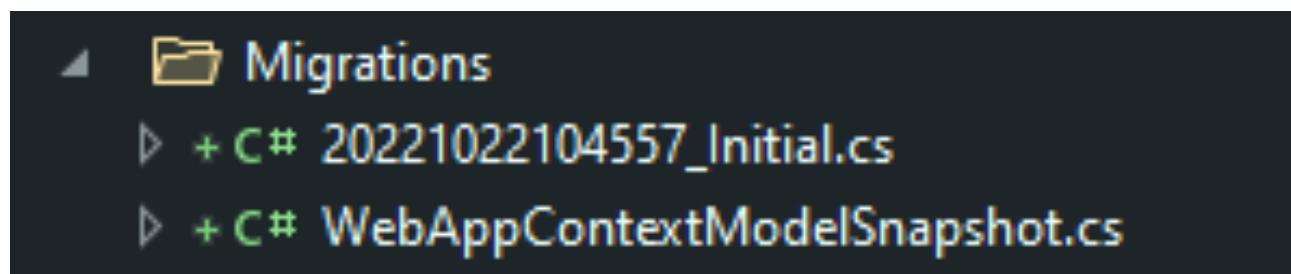
Jetzt muss die Datenbank vorbereitet werden. Hierfür war der Connection-String ausschlaggebend. In diesem wurde der Server und die Datenbank bereits beschrieben. Wenn wir nun nachfolgende Befehle ausführen und auf dem eingetragenen Server keine Datenbank mit eben jenem Namen vorhanden sein sollte, dann wird die Datenbank mit eben jenen Namen neu angelegt. Um jedoch eine Datenbank aus den Models der API anlegen zu klönen, bedarf es einer sogennannten Migration, welche die Daten in das Zieldatenformat (SQL) emigriert.



```
161% Keine Probleme gefunden | ↗
Paket-Manager-Konsole
Paketquelle: Alle | Standardprojekt: web-api
Geben Sie "get-help NuGet" ein, um alle verfügbaren NuGet-Befehle anzuzeigen.

PM> Add-Migration Initial
161% ↗
Paket-Manager-Konsole Fehlerliste Ausgabe
```

Um diese zu veranlassen wird in der Paketmanagerkonsole von NuGet, oder auch (mit anderem Befehl) in einem anderen Kommandozeilen-Tool der Befehl „Add-Migration“ + der Name der Migration ausgeführt. Hier als erste Migration „Initial“. Später wird die Datenbank ihre Einträge immer nach der aktuellsten Migration ausrichten. Dies bedeutet im Umkehrschluss, immer wenn etwas grundlegendes an den Models der API verändert wird, so muss eine neue Migration ausgeführt werden.



Angelegt werden die Migrationen im Projektmappen-Explorer in Visual Studio. Ein Snapshot der Datenbank und die einzelnen Migrationen liegen dort drinnen. Es ist darauf zu achten, dass leere Migrationen, wenn möglich nicht ausgeführt werden, oder gelöscht werden, wenn diese nicht gebraucht werden können (in den meisten Fällen). Damit wird verhindert das die Struktur der Datenbank auf dem Server durcheinander kommt.

```

namespace web_api.Migrations
{
    public partial class Initial : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Schools",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    Name = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    FedState = table.Column<string>(type: "nvarchar(max)", nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Schools", x => x.Id);
                });
            migrationBuilder.CreateTable(
                name: "Holidays",

```

In diesem Bild ist ein Ausschnitt der Migration zu sehen, welche wir angelegt haben. Das Format dieser Migration ist ein, für die SQL-Datenbank, lesbares Format in dem beschrieben wird wie die Datenbank und ihre Tabellen und Abhängigkeiten auszusehen haben.

Gleichzeitig werden wir in der Startdatei auch noch den Zugriff auf die API von außen freigeben.

```

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
    app.UseCors(c => c.AllowAnyHeader().AllowAnyMethod().AllowAnyOrigin().Build());
}

```

Dies ist fundamental wichtig, damit wir später über das Fronten auf diese zugreifen können. Dies geschieht über die Befehle von `app()`

2.1.9 Der Controller der Serviceklasse

Um die Methoden der Service-Klassen letzten Endes auch in der API, wenn diese einmal im Webserver läuft (oder im Localhost während der Entwicklung), zu benutzen bedarf es eines API_Controllers, welcher für jede Serviceklasse der API angelegt werden muss. Dazu nehmen wir einen neuen leeren API-Controller und schreiben dort die CRUD-Methods des Services hinein. Der API-Controller enthält eine eigene Route, welche im [Route] angegeben ist. Auch jede CRUD-Methode wird später eine eigene Route besitzen unter der sie zu erreichen und anzusprechen sein wird. des Weiteren besitzt der Controller ein Attribut vom Typen des Services, welchen er repräsentieren wird. In diesem Falle der GradService zur Verwaltung der Noten(-Punkte) innerhalb der Webanwendung. Dieses Attribut wird im Konstruktor gesetzt.

```
namespace web_api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 Verweis
    public class GradServiceController : ControllerBase
    {
        IGradService gradService;

        0 Verweise
        public GradServiceController(IGradService gradService)
        {
            this.gradService = gradService;
        }
    }
}
```

Hier zu sehen der Aufbau einer Controller-Funktion für eine [HttpGet] Methode. Des Weiteren ist zu vermerken, welche Art Response (nur bei Get-Methoden) die Funktion produzieren wird. Meist ein Status-Code (z.B 404, welchen wir alle aus der Error 404 Meldung kennen). In diesem Falle den Statuscode 200OK, welcher dafür steht dass die Anfrage geklappt hat. Zudem kann in der Route die Id für den Get-Befehl übermittelt werden. Der erwartet Typ der Rückgabe (GradDetails) wird ebenfalls angegeben. Innerhalb der Funktion passiert nicht viel, außer dass lediglich die Funktion des Services aufgerufen wird und geprüft wird ob der Aufruf erfolgreich verlaufen ist, also ob der Service erreichbar war.

```
[HttpGet]
[Route("{id}/[action]")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(GradDetails))]
public IActionResult GetGradById([FromRoute] Guid id)
{
    try
    {
        var model = gradService.GetGradById(id);
        return Ok(model);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

Nachfolgend noch eine CRUD-Operation für den Aufruf der Create-Methode aus dem Service. Die jeweiligen Controller-Action unterscheiden sich nur marginal voneinander, sind es schließlich nur Aufrufe, welche getätigkt werden. Da nun die vorläufige Programmierung des Backend abgeschlossen ist (fürs erste), können wir uns nun darauf konzentrieren Datensätze anzulegen, indem wie die API per Swagger austesten.

```

// ##### POST-Methoden #####
[HttpPost]
[Route("[action]")]
public IActionResult CreateGrad(CreateGradModel createModel)
{
    try
    {
        var model = gradService.CreateGrad(createModel);
        return Ok(model);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}

```

2.2 Backend - WebApi - Funktionstest mit Swagger

The screenshot shows the Swagger UI interface running at localhost:7096/swagger/index.html. The top navigation bar includes links for Gmail, YouTube, Maps, Übersetzen, Nachrichten, and WhatsApp. The main area displays the 'Select a definition' dropdown set to 'web-api v1'. Below it, the 'web-api 1.0 OAS3' section shows two main groups: 'Grad' and 'Holiday'. The 'Grad' group contains five operations: GET /api/Grad/GetAllGrads, GET /api/Grad/{id}/GetGradById, POST /api/Grad/CreateGrad, PATCH /api/Grad/UpdateGrad/{id}, and DELETE /api/Grad/{id}/DeleteGrad. The 'Holiday' group contains three operations: GET /api/Holiday/{id}/GetHolidayById, GET /api/Holiday/GetAllHolidays, and GET /api/Holiday/GetHolidaysByFedState.

Dieses Unterkapitel wird vergleichsweise kurz gefasst werden, da sich so ein Test, der hier auch noch vorerst händisch durchgeführt wird, nur schwer dokumentieren lässt. Jedoch bietet es die Möglichkeit Swagger und seine Funktion ein wenig besser zu beleuchten und zu erklären.
 Hat man die API in Visual Studio über den „Run“ Button gestartet, so öffnet sich eine lokale Testumgebung für eben jene API, genannt Swagger, in der die Befehle der API für die kommenden Requests ausgiebig getestet werden können und echte Einträge in der Datenbank angelegt werden können.

Doch was genau ist Swagger?

Swagger ist eine Sammlung von Open-Source-Werkzeugen, um HTTP-Webservices zu entwerfen, zu erstellen, zu dokumentieren und zu nutzen. Swagger benutzt dazu den Beschreibungsstandard OpenAPI. Swagger ist sozusagen eine Sammlung aus Werkzeugen die das Testen der API in einer lokalen Umgebung erlauben.

DESKTOP-HJHHCN4\...p - dbo.Holidays ➔ X SQLQuery1.sql - D...JHHCN4\rober (59)						
	Id	Name	StartDate	EndDate	Schoold	FederalState
	0ea3994e-3eae-...	Weihnachtsferi...	2022-12-22 00:0...	2023-01-02 00:0...	NULL	Sachsen
	b3b4df1b-78a5...	Winterferien 2023	2023-02-13 00:0...	2023-02-24 00:0...	NULL	Sachsen
	531a47e8-6979-...	Osterferien 2023	2023-04-07 00:0...	2023-04-15 00:0...	NULL	Sachsen
	1ff712d1-798e-...	Pfingstferien 20...	2023-05-19 00:0...	2023-05-19 00:0...	NULL	Sachsen
	8dabacee-5c42...	Sommerferien ...	2023-07-10 00:0...	2023-08-18 00:0...	NULL	Sachsen
	29d03278-167d...	Herbstferien 2023	2023-10-02 00:0...	2023-10-14 00:0...	NULL	Sachsen
	cf63d27b-73f7-...	Herbstferien 2023	2023-10-30 00:0...	2023-10-30 00:0...	NULL	Sachsen
▶*	NULL	NULL	NULL	NULL	NULL	NULL

Zum Testzweck der API-Get-Methoden habe ich in die Datenbank im SQL Management Studio einige Einträge angelegt. Hier zu sehen sind einige Daten für die Ferien in Sachsen im Schuljahr 2022/ 2023.

The screenshot shows the Swagger UI for a .NET Core API. The endpoint is `/api/Holiday/GetHolidaysByFedState`. In the 'Parameters' section, there is a single parameter `fedState` of type `string` with a value of `Sachsen`. The 'Responses' section shows a 'Curl' command and a 'Request URL'. Under 'Server response', the status code is 200 and the 'Response body' is a JSON array of holiday records:

```
[{"id": "531a47e8-6979-440f-8977-41f477526c16", "name": "Osterferien 2023", "startDate": "2023-04-07T00:00:00", "endDate": "2023-04-15T00:00:00"}, {"id": "0ea3994e-3eae-4584-8599-61ae5d66a77f", "name": "Weihnachtsferien 2022", "startDate": "2022-12-22T00:00:00", "endDate": "2023-01-02T00:00:00"}, {"id": "cf63d27b-73f7-462e-9d42-67093b2d90b7", "name": "Herbstferien 2023", "startDate": "2023-10-30T00:00:00", "endDate": "2023-10-30T00:00:00"}]
```

Die einzelnen Service-Methoden (geordnete nach Service) können in Swagger per Klick auf die Methode → dann mit Klick auf den „Try it out“ Button getestet werden. Per Klick auf den „Execute“ Button kann der Http-Request abgesendet werden. Dieser wird gegen die Api, die zu diesem Zeitpunkt lokal läuft gesendet. Diese wiederum sendet die Daten, beziehungsweise die Anfrage an die Datenbank und liefert die Ergebnisse als JSON-File zurück. Hier zu sehen die „GetHolidayByFedState()“ Methode, welche einen String mit dem Namen des Bundeslandes

entgegennimmt und anschließend alle Ferien für übergebenes Bundesland in einem JSON-File zurückgibt.

Das sollte vorerst reichen die Backend-API und deren Erstellung zu thematisieren. Da nun die API lokal läuft und jeder Service das tut was er soll, kann nun mit der Entwicklung der Frontend-GUI unter Angular und TypeScript begonnen werden.

2.3 Frontend - GUI - Programmierung mit Angular

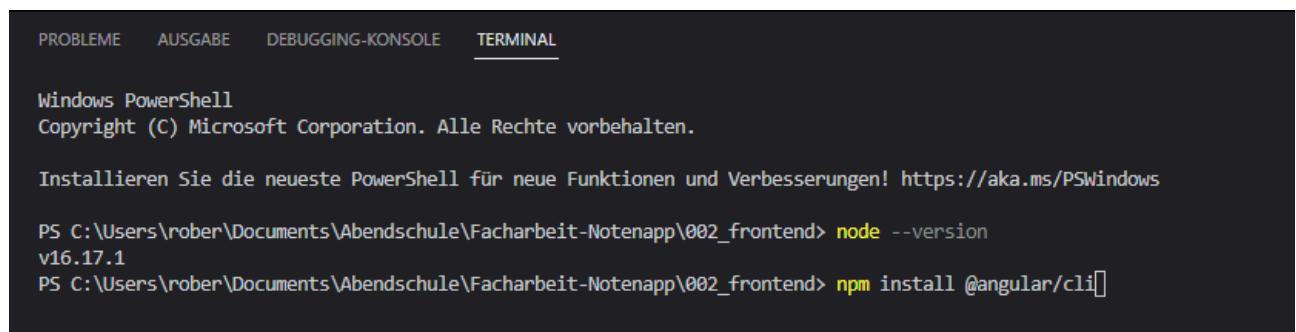
2.3.1 Vorbereitungen treffen - GUI Projekt einrichten

Um die Funktionen der Web-API nutzen zu können, ohne kryptische Befehle an Swagger zu übergeben und JSON-Files an den richtigen Stellen mit Parametern füllen zu müssen, dass der Request gegen die API auch angenommen wird, wird eine nutzerfreundliche Oberfläche benötigt, welche man auch allgemein als Userinterface, oder GUI (Graphical User Interface => Grafische Nutzeroberfläche) bezeichnet. Diese kann, solange sie mit der API kommunizieren kann, in sämtlichen Varianten angelegt werden.

Wir entscheiden uns an diesem Punkt für eine Umsetzung der Webapplikation mit Angular. Denn Angular bringt verschiedene Vorteile für dieses Projekt mit. Zum einen benutzt es die streng typisierte Programmiersprache TypeScript. Ein Ableger von JavaScript, welcher streng typisiert ist und um objektorientierte Programmieraspekte erweitert wurde. Zum anderen ist es weit weniger statisch als eine gewöhnliche Webanwendung, in welcher mit mehreren HTML-Files gehandelt werden müsste, welche statische Inhalte präsentieren würde. Was natürlich für eine dynamische Webanwendung nicht „State-of-the-art“.

Als Entwicklungsumgebung für eine Angular-WebApp kann jeder handelsübliche Webeditor verwendet werden. Ich werde für die Umsetzung dieses Projektes Visual Studio Code verwenden, da dieser durch seine Extensions die Entwicklung von WebCode erheblich erleichtert. So können beispielsweise Pakete direkt im Editor nachinstalliert werden, welche die Entwicklung erleichtern und verbessern. Zuzüglich könnte man mit Code (Kurzform für Visual Studio Code) auch noch (durch die Erweiterung durch gewisse Pakete) auch noch in höheren Programmiersprachen programmieren, was für dieses Projekt jedoch kein ausschlaggebendes Argument sein wird.

Zur besseren Arbeit in unserem Frontend werden wir noch eine Vor-Vorbereitung treffen müssen. Zu dieser gehört die Installation von node.js und dem npm, also dem Node Package Manager, um Webprojekte anlegen zu können und bei Bedarf verschiedene Extensions und Pakete für die Arbeit mit Angular installieren und verwenden zu können. Ist dies geschehen, so kann node.js mit dem Befehl `node --v` in einem Terminal auf seine vollständige Installation geprüft werden (gibt die Version des aktuell installierten node.js zurück). In der Kommandozeile (entweder in Code selbst, oder einem anderen Kommandozeilen-Tool) werden anschließend auch die Installationen der Pakete ausgeführt. Die Syntax für eine solche Installation (mit npm) lautet `npm -i -g <name_package>`.



The screenshot shows a Windows PowerShell window with the title bar "Windows PowerShell". Below the title bar, there are tabs: PROBLEME, AUSGABE, DEBUGGING-KONSOLE, and TERMINAL, with TERMINAL being underlined. The main area of the window displays the following command-line session:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

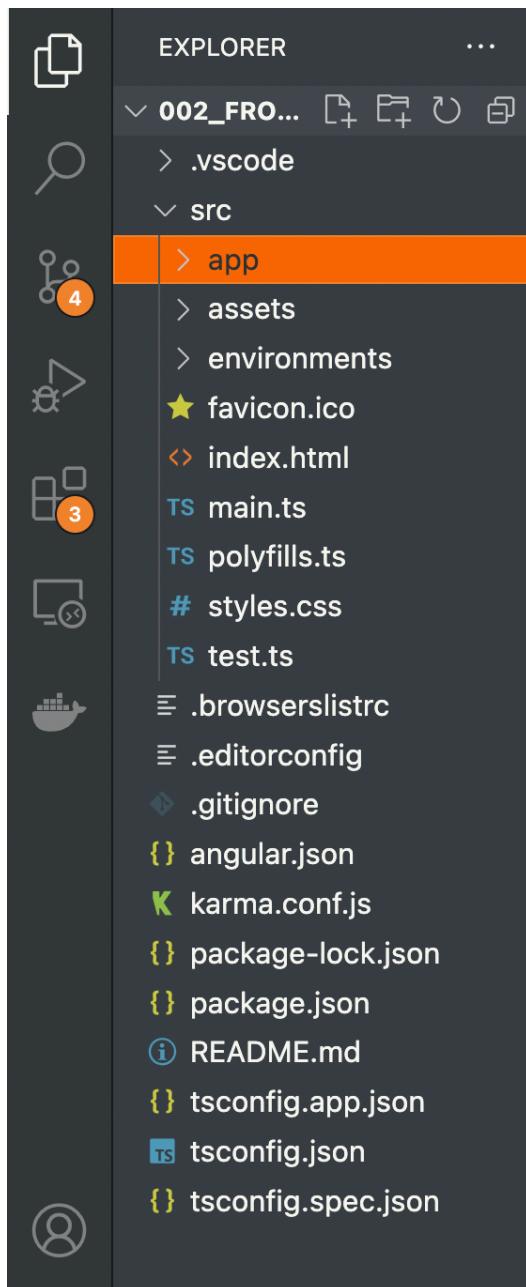
Installieren Sie die neueste PowerShell für neue Funktionen und Verbesserungen! https://aka.ms/PSWindows

PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp\002_frontend> node --version
v16.17.1
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp\002_frontend> npm install @angular/cli[]
```

Zuerst wird also die aktuelle Version der Angular-CLI per npm Befehl installiert. Eine CLI ist eine Kommandozeile. Dies bedeutet, dass das installierte Paket Tools für die Kommandozeile mitbringt, mit welcher man Angular-Projekte über das Terminal anlegen und verwalten können wird. Zudem weitere Befehle, welche Angular-spezifische Dienste und Funktionen ausführen kann, wie z.B. die Erstellung von Components, oder das Starten der Anwendung.

```
Directory is already under version control. Skipping initialization of git.  
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp> ng new 002_frontend
```

Nun kann mit dem Befehl `ng new` ein neues Angular-Projekt angelegt werden. Angular wird nach Betätigung der Eingabetaste anschließend die übergeordnete Struktur für die Anwendung anlegen.



Auf der linken Seite sehen wir die Struktur, welche nach Anlegung des Angular-Projekts angelegt wurde. Im Großen besteht die Angular App aus einem Source-Ordner, welcher später alle Components, das App-Component, sämtliche CSS und HTML Dateien und andere Dateien (die nun noch nicht wichtig sind) enthält.

Der Source-Ordner wiederum unterteilt sich in einen Ordner „app“, welcher alle Informationen zur API und zu den Angular-Components speichert und verwaltet. Zudem sind die Services und die Models der API hier gespeichert. Des weiteren befindet sich beispielsweise eine „styles.css“ in diesem Ordner, in welcher grundlegende und das gesamte Projekt betreffende CSS-Anweisungen vermerkt werden können.

Außerdem befinden sich hier auch noch verschiedene JSON-Files, welche Informationen zur App, zu den Components, zu den Einbindungen und den installierten Paketen enthalten. In diesen Dateien werden wir später unsere Node-Skripte schreiben werden, welche zur erleichterten Bedienung einiger zuzüglich benutzter Pakete gebraucht werden. Beispielsweise dem Paket „OpenApi“, welches Funktionen zur Verfügung stellen wird, mit denen wir direkt mit der API und der daran befindlichen Datenbank kommunizieren können, so wie deren Services und Methoden nutzbar gemacht werden.

Außerdem werden wir in diesen Dateien unsere Abhängigkeiten für das Projekt anlegen, welche, wenn das Projekt auf einem anderen Rechner läuft, automatisch und ohne Probleme nachinstalliert werden können um keine Fehler im Programm auszulösen und damit ein anderer Entwickler nicht händisch sämtliche Pakete die benötigt werden nachinstallieren muss.

Als nächstes werden wir die benötigten Extensions und Pakete über den Node Package Manager nachinstallieren. Da wir uns in diesem Projekt vornehmlich auf den programmatischen Anteil der Anwendung konzentrieren wollen, wird unsere erste Installation also Bootstrap sein.

```
Directory is already under version control. Skipping initialization of git.  
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp> npm i bootstrap
```

Bootstrap ist ein CSS-Framework, welches vorgefertigte HTML-Elemente und deren CSS-Styling bereitstellt. Das bedeutet, dass so eine GUI nun wie ein Baukastensystem zusammengesetzt werden kann und sich der Programmierer nicht um das stilvolle Design selbst kümmern muss, was viel Arbeit in Anspruch nehmen würde, wenn man eine ansprechende Nutzeroberfläche designen wollen würde. Auch werden mit einigen JavaScript-Files die Komponenten von Bootstrap mit einigen Standardverhalten befähigt, welche unnötige Programmierarbeit abnehmen. Beispielsweise muss sich der Programmierer nicht mehr darum kümmern, dass per Klick auf einen DropDown-Button ein Menü unterhalb dieses aufploppt, sondern kann sich darauf konzentrieren wie Daten verarbeitet werden die der Nutzer eingibt und wie diese per Service mit der API kommunizieren. Kurzum Bootstrap nimmt lästige designtechnische und logische Kleinarbeiten ab und präsentiert eine annehmbare Oberfläche in wenigen Kicks.

```
found 0 vulnerabilities
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp> npm i jquery
```

Als nächstes installieren wir die JavaScript Erweiterung jQuery, welche von Bootstrap gebraucht wird, da die Skripte in Bootstrap mit jQuery geschrieben wurden. Um genau zu sein ist jQuery eine freie JavaScript-Bibliothek, die Funktionen zur DOM-Navigation und -Manipulation zur Verfügung stellt. Das DOM ist das „Document Object Model“, mit einfachen Worten die Baumstruktur eines jeden HTML-Dokuments in welchem alle HTML-Elemente, Header und andere Daten rund um das Dokument gespeichert sind. Manipulation bedeutet in diesem Zusammenhang, dass per JavaScript-Befehl also Elemente dem DOM hinzugefügt werden können, verändert werden können, oder aus diesem gelöscht werden können. Zudem erleichtert jQuery erheblich die Nutzung dieser DOM-Elemente im JavaScript-Code. Dazu ein Beispiel:

Element in eine JS-Variable speichern

• JavaScript-Code:

- document.getElementById(“) um ein Element anhand seiner Id aus dem DOM zu filtern
- document.querySelector(“) um ein Element anhand seiner Klasse zu filtern —> erstes was gefunden wird
- document.querySelectorAll(“) um eine Liste an Elementen anhand ihrer Klasse zu filtern

• jQuery-Code

- \$(“#“) => Element nach seiner Id filtern
- Per Punktnotation erweiterbar um auf verschiedene Eigenschaften des gefilterten Elements zuzugreifen
- \$("#myDiv").css("border", "3px solid red") ==> hier wird die border (der Rahmen) eines Elements (vermutlich ein <div>) mit der Id: „myDiv“ auf rot gestellt, beziehungsweise rot eingefärbt.

```
Installieren Sie die neueste PowerShell für neue Funktionen und Verbesserungen! https://aka.ms/PSWindows
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp\002_frontend> npm install ng-openapi-gen
```

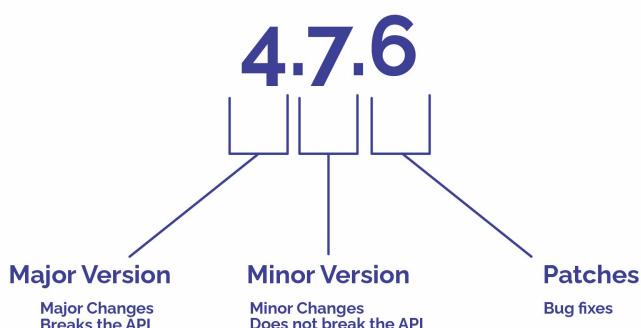
Als nächstes wird das Paket OpenApi installiert. Dieses hilft uns bei der Kommunikation mit unserer Backend-API. Es ist eine OpenSource-Software, welche Funktionen bereitstellt mit einer REST-API zu kommunizieren. Beispielsweise ist eine dieser Funktionen ein Call() an die API, um die Models und Services zurückzubekommen. Aus diesen kann OpenApi anschließend die Models und Services automatisch im Angular-Projekt verknüpfen und anlegen. Dies erspart eine Menge Schreibarbeit, da die Verknüpfung zum Backend so automatisch bereitgestellt wird. Wichtig ist dabei zu beachten, dass wann auch immer so ein Call() aus der OpenApi Software erfolgt die Backend-API im Hintergrund laufen muss, damit sich OpenApi auch die Daten ziehen kann.

2.3.2 Vorbereitungen treffen - Abhängigkeiten klären

Wie vorher schon beschrieben müssen natürlich die Dependencies, also die Projektabhängigkeiten, in einer JSON-Datei hinterlegt werden. Dies dient dazu, dass sich diese Pakete bei Bedarf automatisch installieren. Um die Installation anzustoßen muss der Befehl `npm install --dev` im Terminal im Projektordner wo sich die JSON-Files (package.json und Package-lock.json) befinden ausgeführt werden. Node wird anschließend die Dependencies in den JSON-Files auslesen und anhand dieser die Pakete nachinstallieren, welche noch nicht installiert wurden. So können beispielsweise andere Entwickler auf anderen Rechnern ebenfalls am Projekt mitentwickeln, ohne händisch stundenlang aus dem Code herausfinden zu müssen, welche Pakete installiert werden müssen und diese dann nachinstallieren. Also wieder eine Eigenschaft von Node, welche sehr viel unnötige Arbeit abnimmt. Eine solche Dependency wird aus einem Key (Schlüssel ==> Name) und einem Value (Wert ==> Version) zusammengesetzt. Ein `^` vor der Versionsnummer gibt an, dass Node Versionen installieren soll, welche nur Patches und Bugfixes enthalten, sowie kleinere Features. Ein `~` vor der Versionsnummer erlaubt Node auch neuere Versionen des Packages zu installieren.

```
12  "dependencies": {  
13    "@angular/animations": "^14.2.0",  
14    "@angular/common": "^14.2.0",  
15    "@angular/compiler": "^14.2.0",  
16    "@angular/core": "^14.2.0",  
17    "@angular/forms": "^14.2.0",  
18    "@angular/platform-browser": "^14.2.0",  
19    "@angular/platform-browser-dynamic": "^14.2.0",  
20    "@angular/router": "^14.2.0",  
21    "rxjs": "~7.5.0",  
22    "tslib": "^2.3.0",  
23    "zone.js": "~0.11.4",  
24    "bootstrap": "^5.2.2",  
25    "jquery": "^3.6.1",  
26    "openapi": "^1.0.1"  
27  },
```

Dazu ein kleiner Exkurs zum Thema Versionierung und der Versionsnummer von Software und Anwendungen.



- Links sehen wir eine solche Versionsnummer
- Die erste Zahl der drei Zahlen in einer solchen Nummer gibt die Hauptversion an, diese wird geändert, wenn sich das Programm grundlegend verändert, das heißt vorangegangen Funktionen und Methoden überschreibt, welche vermutlich in der neueren Version dann nicht mehr funktionieren

werden

- Die zweite Zahl gibt die Features an. Diese Zahl erhöht sich bei größeren Änderungen innerhalb einer Version. Zum Beispiel neuen Funktionen oder Features, welche laufen ohne andere in Mitleidenschaft zu ziehen. Alles bisherigen Funktionen und Features laufen dabei noch ohne Probleme weiter.
- Die letzte Nummer gibt die Upgrade-Nummer für Bug Fixes und Co. an. Das heißt diese wird bei kleineren Patches erhöht. Dazu zählen beispielsweise die Beseitigungen von Bugs, oder kleinere Design-Eingriffe um beispielsweise eine GUI der Software ein wenig auszubessern.

```
25      "styles": [
26        "src/styles.css",
27        "./node_modules/bootstrap/dist/css/bootstrap.min.css",
28        "./node_modules/bootstrap/dist/css/bootstrap.min.css"
29      ],
30      "scripts": [
31        "./node_modules/jquery/dist/jquery.min.js",
32        "./node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"
33    ]
```

Zu dem Setzen der dependencies gehört ebenso noch die Verlinkung der BootStrap-StyleSheets in den Stylesheets des Angular-Projekts. Diese Verknüpfung der Styles und Skripte ermöglicht es uns später die Elemente aus der Bootstrap-Bibliothek zu verwenden.

```
# styles.css ×
src > # styles.css
1  /* You can add global styles to this file, and also import other style files */
2
3  @import "~bootstrap/dist/css/bootstrap.min.css";
4
```

Zuletzt muss noch in der Haupt-Style-Datei (styles.css) Die Verbindung zum Bootstrap-Package gesetzt werden. Dies geschieht über einen `@import` Befehl, welcher auch gelegentlich in Webanwendungen lesbar ist um Schriften und andere Abhängigkeiten in ein Stylesheet einzubinden.

2.3.3 Vorbereitungen treffen - Skripte anlegen

Als letzten vorbereitenden Schritt auf dem Weg zur Frontend Angular WebApp müssen noch einige Skripte angelegt werden, welche die Daten aus der Datenbank und von der API snappen werden. Dies wird gemacht, damit man sich bei einer Änderung an der API eine Menge Schreibarbeit ersparen kann. Mit einem kurzen Script-Befehl lässt sich dies quasi mehr nebenher und vor jedem Start der Anwendung ausführen um Fehler in der API zu vermeiden und gleichzeitig nicht unnötig lange Textbausteine aneinanderreihen zu müssen. Also wieder ein Punkt für die Zeitsparnis durch OpenApi.

```
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test",
  "generate-openapi": "ng-openapi-gen --output src/app/api --input http://localhost:7096/swagger/v1/swagger.json"
},
```

Im vorher zusehenden Screenshot ist der Befehl für die grappen der Services und Models aus API und Datenbank vermerkt. Dieser beinhaltet eine Menge Parameter wie die Adresse der API und den Speicherort für die Models und Services. Das Skript enthält einen Key mit dem Namen „generate-openapi“. Über diesen kann das Script vor jedem Start der Anwendung in seiner Entwicklung aufgerufen werden um immer die neueste Version der Datenbank abgebildet zu bekommen und Fehler in den Anfragen an die Api zu vermeiden. Ausgeführt werden können die in der angular.json Date befindlichen Skripte mit dem Präfix „npm run“.

```
found 0 vulnerabilities
PS C:\Users\rober\Documents\Abendschule\Facharbeit-Notenapp\002_frontend> npm run generate-openapi
```

Um das OpenApi-Script also auszuführen wird der Befehl *npm run generate-openapi* in der Befehlszeile des Terminals im Projektordner ausgeführt. Anschließend grapt sich OpenApi alle benötigten informationen aus der Api und der Datenbank und speichert diese im angegebene Pfad, also in einem Unterordner des Source und App Ordners mit dem Namen „api“. Hier wird der Befehl ausgeführt.

2.3.4 Vorbereitungen treffen - Api verknüpfen

Bevor jedoch die Api angesprochen werden kann, oder das Script der OpenApi ausgeführt werden kann, bedarf es noch einiger Einstellungen um die API überhaupt erreichen zu können. Dazu gehören die URL für die API zu setzen unter welcher diese erreicht werden kann.

```
"profiles": {
  "web_api": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "http://localhost:7096",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
}
```

Um diese setzen zu können müssen zuerst im Projekt der WebApi, also im Backend Projekt, die Einstellungen für die URL getroffen werden. Dies geschieht in der Datei „launchSettings.json“ in den Properties des Web-App Projekts. Oben auf dem Bild zu sehen ist, dass die applicationUrl und der dazugehörige Port ausfindig gemacht werden muss, oder auf einen gewissen Wert gesetzt werden muss. Hier in der Entwicklungsumgebung handelt es sich dabei um einen Localhost und einen lokalen Port um diese anzusprechen. Würde die Api später einmal an den Kunden ausgeliefert, so würde hier die Adresse des WebServers verknüpft werden müssen auf dem die API später zugänglich gemacht werden wird.

Nun springen wir zurück in Visual Studio Code um die Verbindung zur API auch noch im Frontend-Projekt zu verknüpfen. Dies geschieht wieder über die Verknüpfung von Properties, welche einen lokale Adresse ansprechen, solange wir uns in der Entwicklungsumgebung befinden und noch nicht die App veröffentlichen um sie einer Nutzerschaft zur Verfügung zu stellen.

```

1 import { NgModule } from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3 import { AppComponent } from './app.component';
4 import { SchoolyearViewComponent } from './schoolyear-view/schoolyear-view.component';
5 import { HttpClientModule} from '@angular/common/http'
6 import { ApiModule } from './api/api.module';
7
8 @NgModule({
9   declarations: [
10     AppComponent,
11     SchoolyearViewComponent
12   ],
13   imports: [
14     BrowserModule,
15     HttpClientModule,
16     ApiModule.forRoot({
17       rootUrl: "http://localhost:7096"
18     })
19   ],
20   providers: [],
21   bootstrap: [AppComponent]
22 })
23 export class AppModule { }
24

```

Dies geschieht in der Datei „appModule.ts“. Hier werden die Url für die API, welche angesprochen werden soll per `forRoot` Befehl gesetzt. Später werden in dieser Anwendung auch die Routen für die Programmnavagation abgelegt, sowie Einstellungen für Daten getroffen, welche über die URL zwischen den Components ausgetauscht werden müssen. Doch zu dieser Praxis an anderer Stelle mehr.

```

5 export const environment = {
6   production: false,
7   apiUrl: "http://localhost:7096"
8 };
9

```

Zuletzt wird die URL für die WebApi noch in der Environment-Datei des Frontend gesetzt. Dies geschieht in der Datei „environment.ts“. Der Parameter `production` bleibt hingegen so lange auf `false`, wie die App sich noch in der Entwicklung befindet und noch nicht im Produktionsumfeld eingesetzt werden muss.

Nun da die wichtigsten Vorbereitungen getroffen wurden, kann es mit der Programmierung des Frontend losgehen. Diese werde ich im nächsten Kapitel kurz beschrieben.

3. GUI Umsetzung - TypeScript und Angular

3.1 Anlegen eines Components

Da alle Teile einer Angular-Webanwendung als Components behandelt werden, so müssen wir uns im ersten Schritt mit der Erstellung eines solchen auseinandersetzen. Da wir die Angular-CLI, also die Kommandozeilen-Tools für Angular installiert haben, können wir alle nun folgenden Dinge in der Kommandozeile ausführen um uns automatisch (mehr oder weniger) ein solches Component erschaffen zu lassen.

```
o (base) → 002_frontend git:(master) ✘ ng generate component schoolyear-view
```

Im Oberen Befehl sehen wir die Syntax des Angular-Befehls um ein Component anzulegen. Da es, zu Beginn der Entwicklung der GUI unser Wunsch sein wird, zuerst eine Art Home-Screen zu schaffen auf der (beim Start der Webanwendung) erst einmal alle vorhandenen Schuljahre, also Verwaltungseinheiten für unsere Daten, angezeigt werden, entscheiden wir uns für das anlegen eines „schoolyear-view“ Components.

Hinweis:

In einer Angular-Anwendung sollte jedes Component nur für eine einzige, explizite Aufgabe verwendet werden um die Übersichtlichkeit zu gewährleisten. In diesem Beispiel besitzt das „schoolyear-view“ Component lediglich die Funktion alle, in der Datenbank auf dem Server befindlichen, Schuljahre als Verwaltungseinheiten anzuzeigen. Mit Klick auf eines dieser sollte ein neues Component geladen werden, welches sich um die Darstellung der darin befindlichen Einheiten (wie Noten und Fächer) kümmert.

Dazu ein weiterer kleiner Exkurs, welcher zum Themenbereich der Planung zugeordnet werden kann. Zunächst müssen wir uns grob Planen, welche Components benötigt werden und in welchen Zusammenhängen diese auftreten. Dazu habe ich nachfolgend ein UML-Diagramm angefertigt.

In diesem oberflächlichen UML-Components Diagramm werden die Hauptkomponenten der zukünftigen Webapplikation angezeigt. Die Pfeile symbolisieren die Wege der Routen innerhalb der Webapplikation. Zudem besitzt jedes Component mindestens ein Attribut, sowie ein- oder mehrere Funktionen, welche innerhalb dieses Components ausgeführt werden können.

```
schoolyear-view
  # schoolyear-view.component.ts
  <> schoolyear-view.component.html
  TS schoolyear-view.component.css
  TS schoolyear-view.component.spec.ts
```

- Im Bild nebenan sehen wir das Component, welches von Angular nach unserem Befehl generiert wurde
- Ein jedes Component besteht aus einer HTML-, einer CSS-, und 2 TypeScript-Dateien
- In der HTML-Datei wird das Gerüst für die Anzeige beschrieben
- Die CSS-Datei stylet diese Anzeige (bei diesem Projekt nicht, weil Bootstrap dies übernimmt)
- Und in der component.ts wird die Programmlogik des Components beschrieben

3.2 Die component.ts Datei - Logikimplementierung

Kommen wir nun zur component.ts, in welcher wir mit den Servicen der API Sprechen werden um unsere Daten aus der Datenbank vom Server zu erhalten.

```
11  export class SchoolyearViewComponent implements OnInit {  
12  
13    title: string = "Schuljahre anzeigen lassen";  
14    subTitle: string = "Deine Übersicht";  
15  
16    availableSchoolYears: SchoolYearDetails[] = [];  
17  
18  
19    constructor(private schoolYearService: SchoolYearService, private router: Router) {  
20  
21      this.schoolYearService.apiSchoolYearGetAllSchoolYearsGet$Json().subscribe({  
22        next: sy => this.availableSchoolYears = sy  
23      });  
24    }  
25  }  
26
```

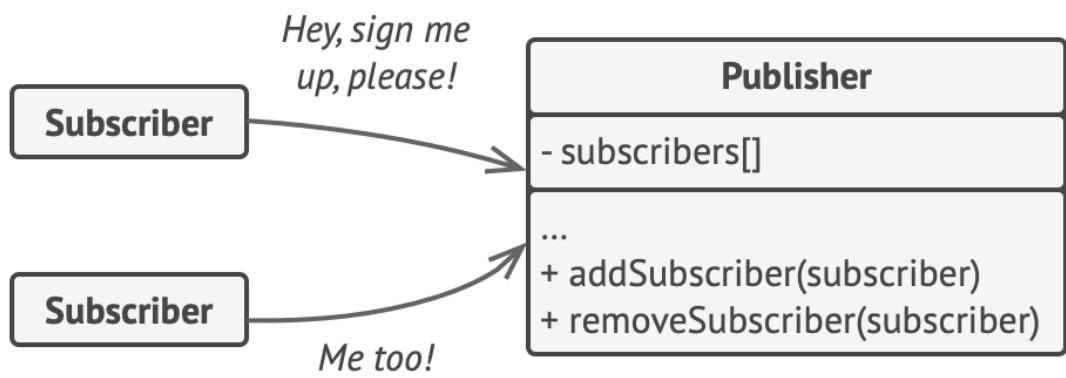
In der component.ts der schoolyear-view Component habe ich lediglich einen Titel und einen Untertitel implementiert, welchen ich auf der Seite des Components anzeigen lassen möchte. Zu sehen ist auch hier ein Konstruktor, welcher bei der Erstellung eines Components im laufenden Betrieb gewissen Funktionen ausführt. In diesem speziellen Falle lade ich einen SchoolYearService vor. Also der Service der API, dessen Funktionen wir benutzen wollen. In dem speziellen Fall vor allem die GetAll()-Funktion um uns die Schuljahre auf der Seite des Components anzeigen zu lassen. Zudem lade ich im Konstruktor noch einen Router vor, ein Element, welches uns in die Lage versetzt zwischen den Components und Services zu routen, also zu navigieren.

Im Methodenkörper des Konstruktors befülle ich ein weiteres Attribut der Component-Klasse, nämlich das Attribut „availableSchoolYears“, welches vom Typ Array aus SchoolYearDetails besteht. Dabei sehen wir eine Funktion des Frameworks RX.js, welches mit Angular und OpenApi ausgeliefert wird. Es besteht aus einer Callback-Funktion über die ich die Funktion der API ansprechen kann. Diese Funktionen des RX.js nutzen dabei das Programmier Design-Pattern des Observers. Dabei ist die Methode der API ein so genanntes Observable-Object, welches der Observer beobachtet und sich auf diesem „subscribet“, also leichenhaft gesprochen diesen Service abonniert. Bei einer Veränderung an diesem Service kann eine Funktion innerhalb des subscribe-Mechanismus ausgeführt werden. In diesem Falle werden die Daten die von der API zurückkommen in das Attribut „availableSchoolYears“ geladen.

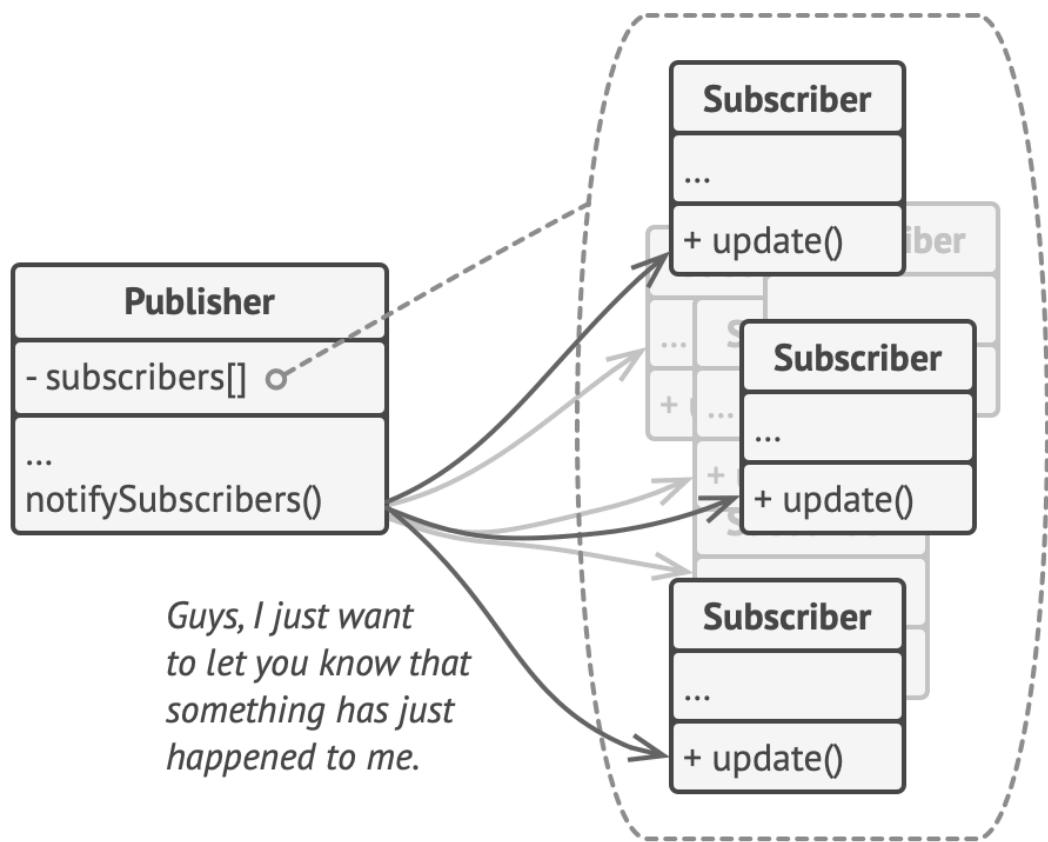
Observer-Pattern

Als schnellen Einstieg in die Wirkungsweise eines Observer-Pattern an dieser Stelle eine Erklärung zum Entwurfsmuster. Es ist ein Design-Muster aus der Softwareentwicklung, welches eine Eins-zu-eins-Abhängigkeit zwischen zwei oder mehreren Objekten definiert, um sämtliche Änderungen an einem bestimmten Objekt auf möglichst unkomplizierte und schnelle Weise zu übermitteln. Zu diesem Zweck können sich beliebige Objekte (Observer oder Beobachter) bei einem anderen Objekt registrieren. Dieses Objekt (Observable oder Subjekt) informiert die registrierten Beobachter, sobald es sich verändert bzw. angepasst wird. Dabei können Die Observer auch nur spezielle Funktionen an einem Subjekt registrieren.

Ohne dieses Entwurfsmuster müssten die Beobachter in regelmäßigen Intervallen um Status-Updates bitten. Die meisten dieser Anfragen würden ins leere laufen und unnötig Datenmüll produzieren. In unserem Falle dient der Observer also als Zuhörer/ Beobachter für die API um informiert zu werden, wenn von dieser die angeforderten Daten bereitstehen und übermittelt werden. Dann kann dieser auf die Daten reagieren und mit diesen interagieren.



A subscription mechanism lets individual objects subscribe to event notifications.



3.3 Die HTML des Components

Im letzten Schritt dieser Dokumentation sehen wir uns noch die HTML-Datei eines solchen Components an. In dieser wird das Grundgerüst, also der Aufbau aus HTML-Elementen dessen beschrieben. Da wir keine CSS-Funktionen verwenden müssen, weil Bootstrap als Framework sämtliche Styles bereitstellt, müssen wir uns die CSS Datei nicht ansehen.

Im Bild hier sehen wir einen Ausschnitt aus eben jenem HTML-Gerüst des Components „SchoolYear-View Component“. Der Abschnitt zeigt die Implementierung der Anzeige der einzelnen Schuljahre, welche in dieser Datenbank gespeichert wurden. Aber alles Schritt für Schritt.

```
<div>
  <div *ngIf="availableSchoolYears.length > 0">
    <div *ngFor="let schoolYear of availableSchoolYears">

      <div class="card mb-3">
        <div class="card-body">
          <h5 class="card-title">{{schoolYear.name}}</h5>
          <h6 class="card-subtitle mb-2 text-muted" *ngIf="schoolYear.school != null">{{schoolYear.school}}</h6>
          <small>{{schoolYear.id}}</small>
        </div>
      </div>
    </div>
  </div>
```

In Bootstrap implementieren sich die Styles der einzelnen HTML-Elemente über CSS-Klassen. In den weiten Tiefen des Bootstrap-Moduls liegen für diesen Zweck Dutzende verschiedene Stylesheets bereit, welche hunderte, wenn nicht sogar tausende Zeilen CSS-Regeln umfassen bereit. Ein einzelnes Element wird dabei nun über eben jene Klassen gestylet. Beispielsweise in dem vierten `<div>` Element von oben auf diesem Bild sehen wir einige der Klassen. Die exakte Verwendung und Bedeutung dieser ist zu jeder Zeit in der Dokumentation von Bootstrap (www.getbootstrap.com) nachzulesen. Für die einfache Verständlichkeit dieser wenigen Klassen sei gesagt, dass es dem Element das Styling einer Bootstrap-Card verleiht (`class="card"`) und einen Abstand zum darunter gelegenen Element von 3 Einheiten haben soll (`class="mb-3" ==> margin-bottom`). Welche Einheiten dies im genauen sind steht ebenfalls in der Dokumentation beschrieben.

Kommen wir nun aber zu den Angular-Direktiven, welche in dem Bild eben noch zu sehen sind. Die Rede ist von sogennaten Template-Befehlen in Angular, oder auch Direktiven. Dazu gehört eine ganze Palette von Anweisungen und Direktiven, welche in Angular gewisse Funktionen zur Verfügung stellen. Deshalb konzentriere ich mich in diesem Beispiel nur auf die `ngFor` und `ngIf` Direktiven die in diesem Bild zu sehen sind. Der `*` vor der Direktive gibt an, dass es sich um eine der "Structural Directives" und nicht etwas um eine Attribut-Direktive, welche auf die Attribute innerhalb des DOMs zugreift und regiert, handelt.

- **ngIf ==>** Diese Direktive funktioniert wie eine normale if-Abfrage in der Programmierung. Dieser wird eine Bedingung übergeben welche erfüllt sein muss, damit dieses Element, welches damit beschrieben ist, auf der Seite angezeigt werden soll.
 - In unserem Falle wird dieses Element (`<div>`), welches als Platzhalter für die Liste der in der Datenbank gespeicherten Schuljahre fungiert, nur dann angezeigt, wenn die Liste an verfügbaren Schuljahren (in der component.ts) mindestens 1 Element besitzt und nicht leer ist.
- **ngFor ==>** Diese Direktive funktioniert wie eine normale for-Schleife in JavaScript. Dabei funktioniert diese wiederum wie entweder wie eine foreach-Schleife, oder eine Zählerschleife.
 - In dem Falle wird mit der ngFor-Schleife nun jedes Element in „availableSchoolYears“ durchlaufen und für jedes das Innere des `<div>`-Elements produziert. Mit anderen Worten: Jedes Schuljahr bekommt per Schleife also seine eigene Bootstrap-Card.

Nun noch ein Wort zu der oben zu sehenden `{()}`-Schreibweise. Dies ist eine Schreibweise des sogenannten Property-Binding in Angular und bindet, verbindet also einen Wert aus einem Element mit dem HTML-Element. Es gibt viele Arten in Angular Bindings herzustellen und so Werte auf der Webseite eintragen lassen zu können, von denen man während der Entwicklung kein genaues Bild haben muss. Wichtig ist hierbei nur zu wissen, dass wir zur Entwicklungszeit nicht wissen, welches Element uns im Schleifendurchlauf in der Variable „schoolYear“ zurückgegeben wird. Was wir wissen ist jedoch, dass es ein SchoolYearDetailsModel ist, denn dieses haben wir in der component.ts von der API abgefragt (in einer Liste aus Elementen ==> getAll()). Also wird jedes Element in der Liste ein SchoolYearDetails-Element sein und so hat jedes davon eine Id, einen Namen, eine Schule und andere uns bekannte Attribute, welche wir hier ansprechen und zu Platzhaltern für die späteren Daten zur Laufzeit machen.

4. Starten der Anwendung

Wenn wir alles richtig gemacht haben, und schon einige Schuljahre angelegt haben sollten (Hierbei zu Testzwecken und noch nicht dynamisch), können wir die Angular-Anwendung nun starten und einen ersten Blick auf die Oberfläche werfen.

```
(base) → 002_frontend git:(master) ✘ ng serve --open
✓ Browser application bundle generation complete.

Initial Chunk Files      | Names          | Raw Size
vendor.js                | vendor         | 2.47 MB |
styles.css, styles.js    | styles         | 679.19 kB |
polyfills.js              | polyfills     | 318.02 kB |
scripts.js                | scripts        | 166.21 kB |
main.js                   | main           | 116.00 kB |
runtime.js                | runtime        | 6.52 kB |

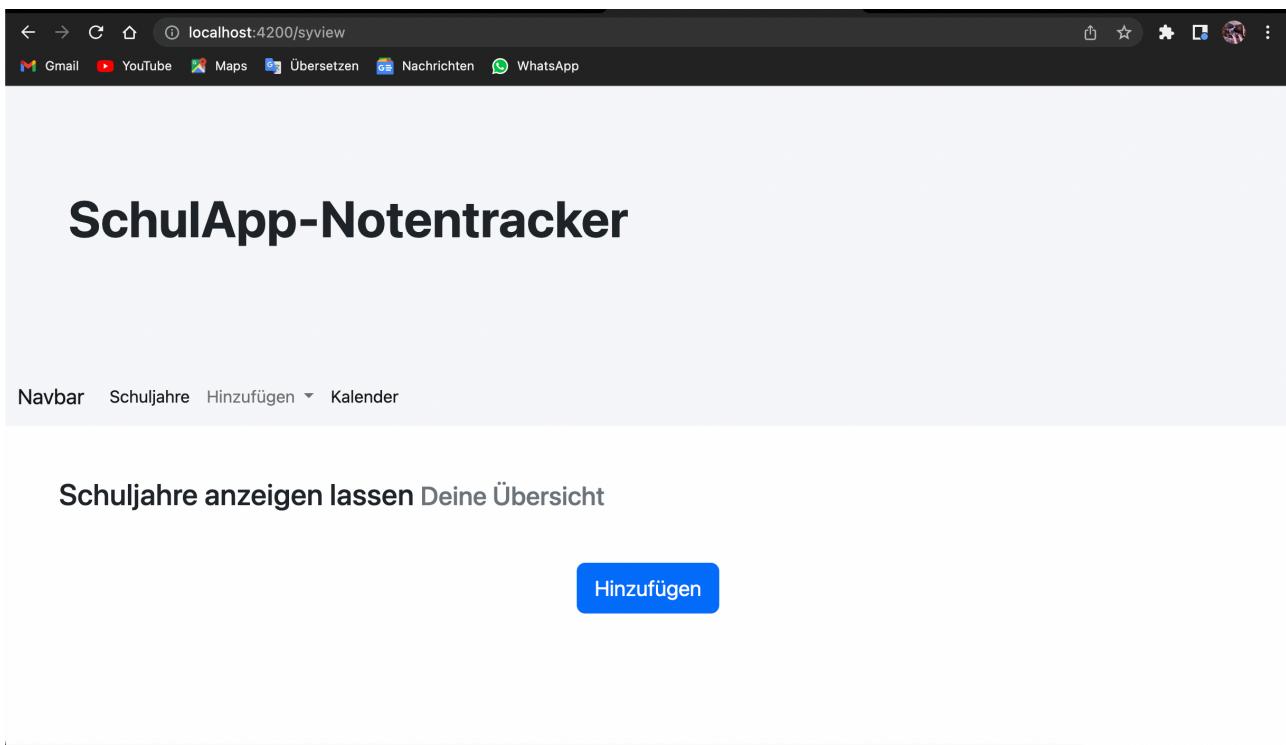
| Initial Total | 3.73 MB

Build at: 2022-11-05T12:51:53.964Z - Hash: 520b4818ab942d14 - Time: 6986ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✓ Compiled successfully.
```

Mit dem Kommandozeilen-Befehl `ng serve --open` wird die Anwendung über das Terminal gestartet. Jetzt prüft Angular ob es Konflikte, beziehungsweise Probleme bei der Kompilierung des Programmes gibt. Sollte dies nicht der Fall sein, so wird im Standard-Browser des Computers eine Instanz des Projektes geöffnet werden.



Da ich zu Testzwecken noch keine Schuljahre angelegt habe, erscheinen bei mir in der Ansicht im Localhost keine Verfügbaren Schuljahre, wenn ich auf den Reiter Schuljahre klicke. Lediglich wird mir ein Button angezeigt, welchen ich so programmiert habe (*ngIf), dass er nur dann zu sehen ist, wenn keine Schuljahre verfügbar sind, also die Liste an verfügbaren Schuljahren in der component.ts eine leere Liste ist.

Nun da die ersten Test abgeschlossen sind und die Anwendung in Betrieb ist und läuft, kann damit begonnen werden nach und nach jedes einzelne Component im Angular-Code anzulegen und zu befüllen. Da dies jedoch sich immer und immer wieder stark gleichen wird, wird dies nicht mehr Gegenstand dieser Dokumentation sein. Vielmehr wird diese sich noch dem letzten Aspekt einer solchen Entwicklung einer WebApp widmen.

Nachdem wir also schon die Planung des Projektes (inklusive der Vorauswahl an Techniken, Software und Co) abgeschlossen haben, das Backend geplant und angelegt haben, sowie ein Frontend begonnen haben zu bauen, ist der letzte Schritt auf dem Weg eine professionelle Software zu entwickeln natürlich noch der Schritt der Software-Testung. Diese wird Gegenstand des letzten kleineren Kapitels sein.

5. Software-Testing - Entwicklung

5.1 Übersicht Thema Software-Tests

5.1.1 Arten der Software-Testung

Nun da wir am wohl wichtigsten Punkt der Softwareentwicklung angelangt sind, wird es Zeit diesem, oftmals unbekannten Themengebiet einen kleinen Einstieg zu geben. Die Rede ist von Softwaretests. Softwaretests sind ein wichtiger Bestandteil der Entwicklung von Anwendungen und Programmen. Dabei werden diese nicht erst ganz zum Schluss des Projektes angelegt und bearbeitet, sondern schon während der eigentlichen Entwicklung.

Doch zunächst müssen wir uns einen Überblick über die beiden großen Arten von Software-Tests verschaffen. Die übergeordnete Unterscheidung liegt in der Art und Weise wie die Software-Tests ausgeführt werden. Zum einen gibt es automatische Software-Tests, zum anderen gibt es manuelle Tests.

- **Manuelle Tests** werden von einem Menschen durchgeführt, der sich durch die Anwendung klickt oder mithilfe der richtigen Tools mit der Software und den APIs interagiert.
- **Automatische Tests** werden von einem Computer durchgeführt, der ein zuvor geschriebenes Testskript ausführt.

Für unser Projekt ist es vor allem nötig die Web-Api im Backend zu testen. Würde man diese Tests manuell durchführen wollen, so müsste man sich ein Protokoll anlegen und sehr genau überlegen, welche Bereiche man wie testen würde. Gewisse Fehlerquellen würden hierbei aber einfach durch das Raster der Möglichkeiten rutschen. So könnte man nur sehr schwer einen abrupten Abbruch der Datenbank-Verbindung händisch simulieren und wie genau wäre dann noch das Ergebnis zu bewerten? Aus diesem Grund werden wir uns in diesem Kapitel anschauen wie wir für unsere Backend-WebApi einige Software-TestScripte schreiben.

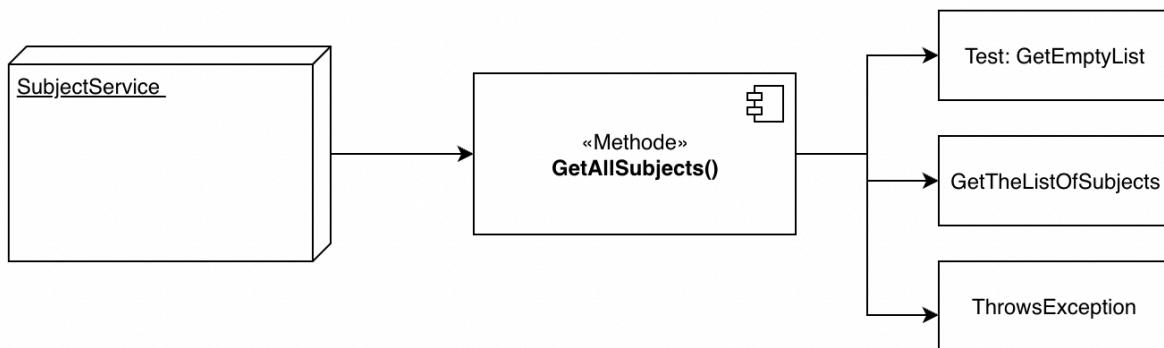
Doch zunächst müssen wir uns anschauen, welche Arten von automatisierten Software-Tests es überhaupt gibt.

- **Unit-Tests** zum Testen einzelner Methoden und Funktionen der von der Software verwendeten Klassen, Komponenten oder Module. Diese sind unabhängig voneinander, simulieren also alles außerhalb der getesteten Unit. So wird dabei lediglich das Verhalten jeder Einheit im Programm an sich getestet => in unserem Falle ist eine Unit jeweils eine Methode aus einem Service. Die Unit könnte also die „GetAllSubjects()“ Methode aus dem SubjectService sein. Dabei ist uns innerhalb dieses Tests egal ob beispielsweise CreateMethode davor, um Einträge in die Datenbank einzupflegen geklappt hat, diese würde in einem separaten UnitTest erfolgen.
- **Integrations-Tests** stellen sicher, dass verschiedene von der Anwendung genutzte Module oder Services problemlos ineinander greifen. Diese großen Tests werden wir vorher außen vor lassen, da es kaum voneinander abhängige Module und Services gibt. Jeder Service steht bei uns in der Applikation für sich allein und nutzt keinen anderen Service zuzüglich.
- **Sonstige Tests** wie Funktions- oder Leistungstests werden wir ebenfalls nicht benötigen. Auch hier gilt: In der Welt der Software-Tests gibt es Unmengen verschiedener, für den jeweiligen Zweck und Nutzen nötiger Tests.

5.1.2 Der Unit-Test

Auch Modultest genannt ist ein Softwaretest, mit dem einzelne, abgrenzbare Teile von Computerprogrammen auf ihre Funktionalität und Stabilität hin überprüft werden. Das Ziel dieses Tests ist es die technische Lauffähigkeit und die Korrektheit ihrer Ergebnisse zu überprüfen.

Um einen UnitTest zu schreiben werden wir einige Vorüberlegungen treffen müssen, welche natürlich nicht so tiefgründig ausfallen werden wie bei der Planung der Applikation an sich. Denn auch hier kommt uns die IDE Visual Studio 2022 sehr stark entgegen und wird uns als Entwickler dabei unterstützen solche Tests anzulegen und nicht geprüfte Code-Pfade zu ermitteln um die Tests für diese noch nachzuholen. So werden wir am Ende dieses Kapitels zumindest in der Lage sein sämtliche Codepfade innerhalb der Services unserer Api zu prüfen und die Ergebnisse davon abzurufen.

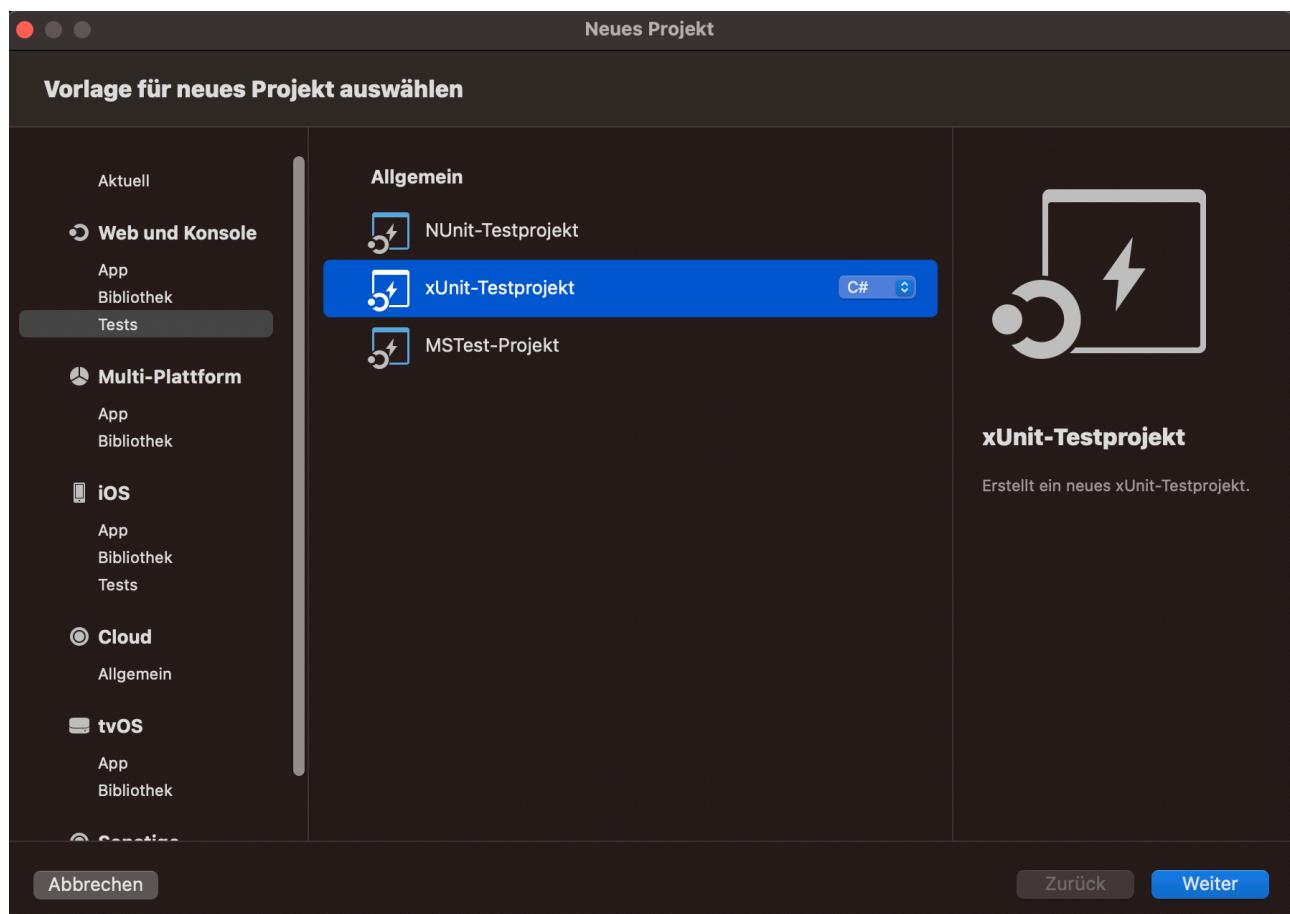


In dem Bild oben sehen wir ein kleines Diagramm, welches ich zu diesem Zweck erstellt habe. Die Vorüberlegung beschränkt sich also darauf, dass wir jeden Service in unserem Programm (also der WebApi) in seine einzelnen Methoden zerlegen werden. Für jede dieser Methoden werden wir verschiedene Tests entwickeln. Oben am Beispiel der GetAllSubjects()-Methode könnten wir beispielsweise drei verschiedene Tests schreiben (welche einfache Methoden/Funktionen sind), die zum Beispiel prüfen ob eine unter einem speziellen Fall wirklich eine leere Liste zurückgegeben wird (Test 1), oder die tatsächliche Liste zurückgeben wird (Test 2) oder eine Exception geworfen wird, zum Beispiel weil die Verbindung zum Datenbank-Server verloren wurde.

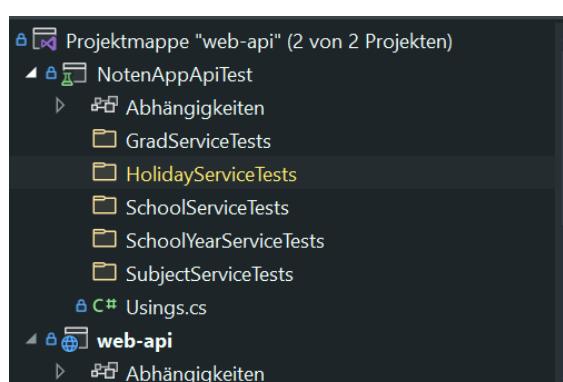
5.2 Software-Tests anlegen

Starten wir nun also in der IDE (Visual Studio) mit den Grundeinstellungen um Tests an der Backend-API vorzunehmen. Dazu wird ein neues Projekt benötigt, welches wir innerhalb des Projektordners auf höchster Ebene hinzufügen. Wichtig ist es, dass wir das Test-Projekt dem Web-Api Projekt hinzufügen und kein neues eigenständiges Projekt anlegen. Dadurch werden gewisse Verknüpfungen der Namespaces schon von der IDE übernommen, welche sonst händisch hinzugefügt werden müssten.

5.2.1 Projektstart - XUnitTests für die API



Wir legen nun also ein neues Projekt vom Typ XUnit-Tests für die Sprache C# an. Das .Net Framework wird für den Langzeit-Support auf der höchsten aktuell verfügbaren Version angelegt. Ein Name wird vergeben, in dem Falle habe ich mich für „NotenAppTests“ entschieden, damit aus dem Projektnamen bereits ersichtlich wird, dass es sich um ein Projekt mit Software-Tests handelt.



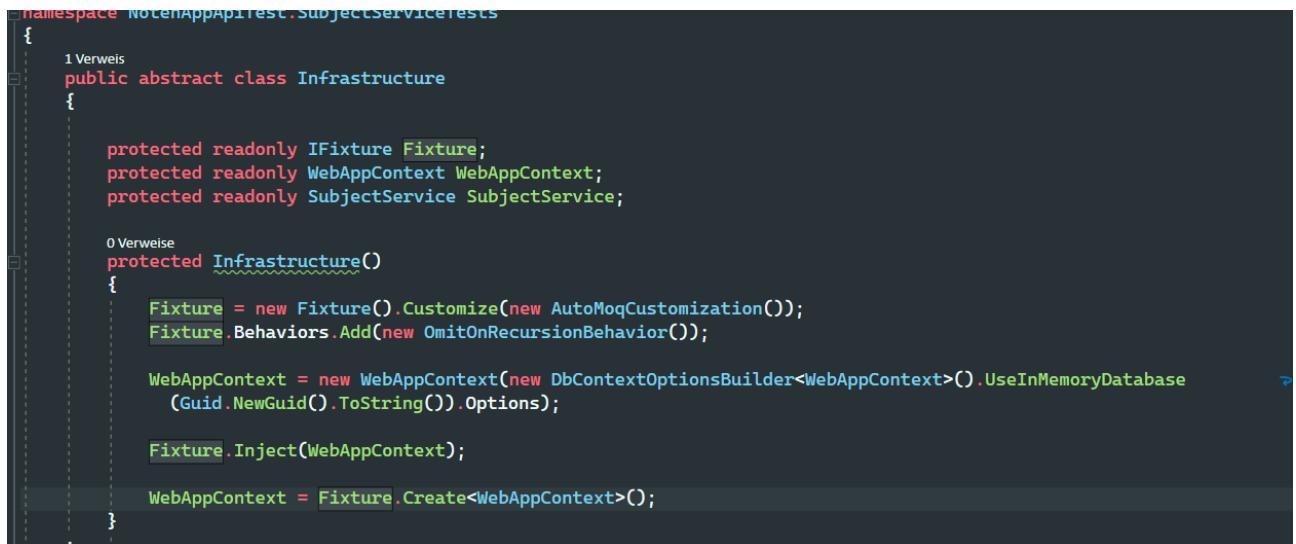
- Nun kommen wir zur Ordnerstruktur um ein einheitlich, übersichtliches Testumfeld für die Unit-Tests zu gewährleisten
- So bekommt jeder Service einen eigenen Ordner, welcher nach der Konvention [Name des Services + „Tests“] benannt ist
- Innerhalb dieser Strukturen werden nun die verschiedenen Methoden des Services getestet
- Diese Methoden-Tests werden wie die Methoden benannt, deren Funktionalität sie testen sollen
- Zusätzlich werden noch 2 weitere Konstrukte innerhalb dieser Ordner angelegt

Erforderliche Klassen innerhalb der Unit-Tests

- Die **Test-Klassen** an sich, benannt nach den Methoden die sie testen werden
- Die **Infrastructure-Klasse**, welche gewisse Voreinstellungen für den gesamten Service vornimmt. Dieser ist eine abstrakte Klasse, das heißt von dieser sollten keine (und können auch nicht) Objektinstanzen erzeugt werden. Seine Funktionen/ Methoden werden per Aufruf direkt in der erbenden Klasse zugänglich sein
- Die **Assertions** ==> Diese Klassen bilden eine Art Schnittstelle zwischen den Klassen und Entitäten der Datenbank. Wir haben durch die Architektur unserer Webanwendung das folgende Problem. Bekommen wir von einer Methode einen Datentyp zurück, so ist dieser meist ein DetailsModel. Jedoch muss überprüft werden ob dieses auch dem Model in der Datenbank entspricht. Nun kann aber ein DetailsModel nicht mit einem Model verglichen werden, zumindest nicht ohne weiteres. Für den Compiler sind dies zwei verschiedene Konstrukte, welche nur für uns Anwender eine Logikeinheit bilden. Denn sie erben weder voneinander, noch implementieren sie sich untereinander. Die Logik ihrer Zusammengehörigkeit entsteht nur beim Anwender selbst. Da sich ein Computer diesen Zusammenhang aber nicht vorstellen kann (denn ein Computer kann sich allgemein keine eigenen Urteile zusammen denken), würde ein Vergleich der Attributre zeitaufwendig und mit viel Code verbunden sein. Wir müssten quasi bei jedem Attribut dem Compiler mitteilen aus welchem Objekt es entspringt. Dies wäre zeitaufwendig und würde dutzende Zeilen Codes produzieren. Hier kommen die Assertions ins Spiel. Diese werden als Mittelklasse angelegt, welche Funktionalitäten bereitstellen um diese, doch von Grund auf, verschiedenen Klassen zu vergleichen.

5.2.2 Die Infrastructure anlegen

In dieser Klasse werden die Bedingungen gesetzt, damit die Pakete, welche wir installierten angewendet werden können. Beispielsweise instanzieren wir ein Object vom IFixture Interface, welches dafür verwendet wird einen Objekteintrag in der Datenbank zum simulieren. Fixture legt bei einer Objektinstanziierung Beispieldaten für die Attribute der Klassen an.



```
namespace NoteAppApiTest.SubjectServiceTests
{
    1 Verweis
    public abstract class Infrastructure
    {

        protected readonly IFixture Fixture;
        protected readonly WebApplicationContext WebApplicationContext;
        protected readonly SubjectService SubjectService;

        0 Verweise
        protected Infrastructure()
        {
            Fixture = new Fixture().Customize(new AutoMoqCustomization());
            Fixture.Behaviors.Add(new OmitOnRecursionBehavior());

            WebApplicationContext = new WebApplicationContext(new DbContextOptionsBuilder<WebApplicationContext>().UseInMemoryDatabase
                (Guid.NewGuid().ToString()).Options);

            Fixture.Inject(WebApplicationContext);

            WebApplicationContext = Fixture.Create<WebApplicationContext>();
        }
    }
}
```

Der Eintrag `Fixture.Behaviors.Add()` mit der Parameterübergabe `OmitRecursionBehavior()` verhindert dass es zu Rekursionen in der InMemory-Datenbank kommt, also Zirkelschlüssen. Außerdem wird ein Context auf die eigentliche Datenbankstruktur angelegt, damit Fixture weiß, welche Datenbankstruktur es simulieren soll.

Was ist eine InMemory-Database?

In diesem Code-Fragment taucht der Begriff InMemory-Database auf. Doch worum handelt es sich dabei eigentlich genau? Da wir in den UnitTests nur Bruchstücke, also die kleinsten Teile der Anwendungen auf ihre Funktionalität und die Richtigkeit ihrer liefernden Ergebnisse hin prüfen, interessiert uns Entwickler nicht, ob sämtliche andere Prozesse der Units, welche wir gerade nicht testen funktionieren. Zudem testet man auch nie gegen eine wirkliche Datenstruktur. Dieser

Aufwand ist für den Test unerheblich, würde zu viel Arbeit machen und stellt auch immer ein Risiko bereit die Arbeitsumgebung zu beschädigen. Also wird im Arbeitsspeicher des Rechners eine Datenbank simuliert, welche exakt nach der Datenbank aufgebaut ist, wie die, deren Kontext übergeben wird. Zudem wird der Arbeitsspeicher-Datenbank eine Guid, also eine eindeutige UUID gegeben, welche intern zugeordnet und verwaltet wird um die Anfragen an diese zu handeln.

```
0 Verweise
protected void RegisterModels()
{
    Fixture.Register(() => Fixture.Build<Subject>()
        .OmitAutoProperties()
        .With(s => s.Id)
        .With(s => s.Name)
        .With(s => s.ShortName)
        .With(s => s.Grads)
        .Create());

    Fixture.Register(() => Fixture.Build<Grad>()
        .OmitAutoProperties()
        .With(g => g.Id)
        .With(g => g.Name)
        .With(g => g.Subject)
        .With(g => g.Points)
        .With(g => g.Date)
        .Create());
}
```

In die Infrastructure-Klasse, welche sogeschen als Bauplan für alle weiteren Testklassen gelten wird, werden wir außerdem die Models angeben, welche in der Datenbank wie genau simuliert werden. Würden wir diese Funktion nicht schreiben und sie ebenfalls nicht am Anfang jeder Testmethode aufrufen, würde das Programm diese Daten selbst festlegen. Das würde unnötig viel Zeit und Speicher kosten, denn so würden auch, für die Tests unrelevante Codepfade erstellt. Dazu ein kleines Beispiel: Wenn wir in einem unserer Unit-Tests darauf prüfen wollen, ob eine List aus Schulen von einer Methode zurückgegeben wird und dies akkurat geschieht, so interessiert uns während des Tests nicht, ob jede Schule Fächer und Noten und so weiter besitzt, wir legen die Schule nur als Entität mit Namen und mit Id an um ihre Existenz in der Datenbank zu prüfen. Genau dafür ist diese Hilfsfunktion vorhanden. Sie registriert das Aussehen der Models an der simulierten Datenbank. Oben als Bild: Ein Ausschnitt aus der RegisterModels() Methode für die Models „Subject“ und „Grad“.

5.2.3 Die Assertions anlegen

Wie oben bereits erklärt wird es in unserem Testprogramm nötig sein, dass wir zurückgegebene Werte direkt mit unseren abgegebenen Werten (zum Beispiel Parameter, oder Datenbankeinträge) vergleichen und überprüfen. Um dies nicht händisch für jedes Attribut innerhalb der Klassen zutun, werden wir sogenannte Assertions benutzen, Hilfsklassen mit Hilfsmethoden, welche diese Arbeit für uns übernehmen.

```
namespace NotenAppApiTest.SubjectServiceTests.Assertion
{
    4 Verweise
    public sealed class SubjectDetailsAssertions : ReferenceTypeAssertions<SubjectDetails, SubjectDetailsAssertions>
    {
        1 Verweis
        public SubjectDetailsAssertions(SubjectDetails subject) : base(subject) { }

        0 Verweise
        protected override string Identifier => "subjectdetails";

        0 Verweise
        public AndConstraint<SubjectDetails> Be(Subject subject)
        {
            Subject.Id.Should().Be(subject!.Id);
            Subject.Name.Should().Be(subject!.Name);
            Subject.ShortName.Should().Be(subject!.ShortName);

            return new AndConstraint<SubjectDetails>(Subject);
        }
    }
}
```

Für diesen Zweck muss für jedes Model, welches auf seine Eigenschaften mit einem baugleichen Model überprüft werden soll eine sogenannte Assertion angelegt werden. In einer hierarchisch übergeordneten Superklasse werden wir sämtliche dieser Assertions bündeln und später an die Testklassen übergeben. Denn diese Assertion können nicht nur simpel Daten miteinander vergleichen, sondern können die Assert-Befehle der FluentAssertions-Packages überschreiben, sodass deren Befehle wie `Should()` oder `Be()` und weitere verwendet werden können um Daten miteinander zu vergleichen. Wie dies funktioniert wird an späterer Stelle erläutert.

Hier sehen wir eine dieser Assertions-Klassen und ihren Inhalt einmal genauer. Der Name `SubjectDetailsAssertions` gibt an, dass wir mit diesen vor allem `SubjectDetails` vergleichen wollen (dementsprechend mit `Subjects` in der Datenbank). In dieser Klasse passiert viel, was nicht alles ausführlich erklärt werden muss, denn eine genaue Anleitung wie man diese Assertions schreibt findet sich auch in der Dokumentation der FluentAssertion-Packages. Wichtig hingegen ist die `AndConstraint`-Methode, welche besagte Überprüfung der Member vornimmt.

Wichtig: Es werden nur diejenigen Member und Attribute gegeneinander überprüft, welche man in dieser Methode unterbringt. Alles was hier nicht aufgeführt ist und dennoch ein Attribut einer der Klassen die verglichen werden sollen darstellt, wird nicht überprüft und wird auch nicht Bestandteil einer solchen Überprüfung (schmeißt also bei Nichterwähnung auch keinen Fehler).

```
0 Verweise
public static class Assertions
{
    0 Verweise
    public static SubjectDetailsAssertions Should(this SubjectDetails subjectDetails) => new SubjectDetailsAssertions
        (subjectDetails);

    0 Verweise
    public static SchoolYearDetailsAssertions Should(this SchoolYearDetails schoolYearDetails) => new ..
        SchoolYearDetailsAssertions(schoolYearDetails);

    0 Verweise
    public static GradDetailsAssertions Should(this GradDetails gradDetails) => new GradDetailsAssertions(gradDetails);

    1 Verweis
    public static SchoolDetailsAssertions Should(this SchoolDetails schoolDetails) => new SchoolDetailsAssertions
        (schoolDetails);

    0 Verweise
    public static HolidayDetailsAssertions Should(this HolidayDetails holidayDetails) => new HolidayDetailsAssertions
        (holidayDetails);
}
```

Und in diesem Bild sieht man letzten Endes die Superklasse der Assertions, welche zur Bündelung der einzelnen DetailsAssertions vorhanden ist. Wir bemerken, dass es sich um eine abstrakte Klasse handelt, also eine Klasse von der keine Instanz generiert werden muss um auf ihre Eigenschaften zuzugreifen. Wir schreiben die Assertion nach folgendem Muster:

- Der Zugriffsmodifikator `public static` um die Assertions außerhalb der Klasse zugänglich zu machen
- Die Assertion die überprüft werden soll
- **Should ==>** sollte sein + Parameter was wir auf der einen Seite des Vergleiches erwarten. Also `WAS` sollte wie sein (Dies hier wird eine Lambda-Expression, eine verkürzte Schreibweise einer Funktion um nicht einen gesamten Funktionskörper definieren zu müssen)
- Dann ein FatArrow um zu sagen was mit dem Parameter in der Lambda-Expression geschieht + eine Instanz der Assertion mit Übergabe des erzeugten Parameters

Nun ist unser Setup soweit eingerichtet, dass wir damit starten können unsere ersten Software-Tests für das Modul SubjectService zu entwickeln. Im nächsten und letzten Schritt werden wir also die Tests anlegen und überprüfen.

5.2.3 Den Test schreiben - Unit-Test

Nun sind wir also am wichtigsten Punkt der Softwareentwicklung angelangt, dem Testen der Software. Dieser Schritt geschieht parallel zur Entwicklung und wird immer und immer wieder wiederholt. Immer wieder werden Codepfade angepasst werden, neue Tests entwickelt und vorangegangene Tests werden angepasst werden. Bis die Software schließlich veröffentlicht werden kann. Genug aber der Vorrede, schauen wir uns den ersten Test an, welchen wir schreiben werden.

```
[Fact]
● | Verweise
public void GetAllSubjects()
{
    // Arrange
    RegisterModels();

    for(int i = 0; i < 3; i++)
    {
        var subject = Fixture.Build<Subject>().Create();
        WebApplicationContext.Subjects.Add(subject);
    }

    WebApplicationContext.SaveChanges();
    // Act
    var allSubjects = Call();

    // Assert
    allSubjects.Should().NotBeNull();
    allSubjects.Should().HaveCount(3);
    WebApplicationContext.Subjects.Should().HaveCount(3);
}

[Fact]
● | Verweise
public void ReturnsEmptyListIfNoSubjectFound()
{
    // Arrange
    RegisterModels();

    // Act
    var result = Call();

    // Assert
    result.Should().BeEmpty();
    WebApplicationContext.Subjects.Should().BeEmpty();
}

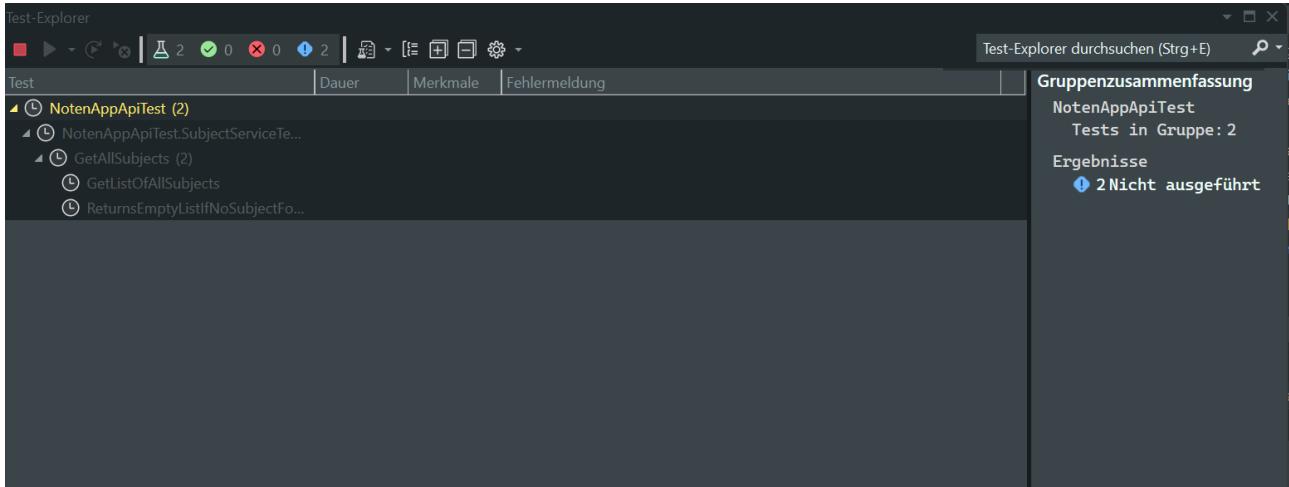
[debuggerStepThrough]
2 Verweise | ● 0/2 bestanden
public List<SubjectDetails> Call() => SubjectService.GetAllSubjects();
}
```

So könnte die Klasse GetAllSubjects aussehen. Doch was sehen wir auf diesem Bild nun genau?

- Ganz am Ende des Bildes sehen wir eine Anweisung **[DebuggerStepThrough]** also eine Anweisung für den Debugger, welche er überlesen soll. In dieser nachfolgenden Lambda-Funktion werden wir die Servicemethode vorbereiten. Dies ist kein notwendiger Schritt, erspart uns aber bei jedem Test die vollständige Funktion mit Parameterliste und Co. ausführlich abzuschreiben, indem wir diesen gesamten Ausdruck in die Funktion `Call()` speichern. In diesem Beispiel mag die Verwendung dieser Lambda-Expression mehr oder weniger sinnlos erscheinen, doch man stelle sich vor, wie es bei einer zu testenden Funktion aussehen würde, welche einen ganzen Menge an Parametern entgegennimmt. Hier würde massiv viel Schreibarbeit in den simplen Aufruf der eigentlichen Methode fallen, dass sich der Code unnötig aufblähen würde.
- Dann sehen wir die Anweisung **[Fact]**. Diese leitet für den Test-Explorer, welchen wir uns noch ansehen werden den Test ein.
- Der Aufbau des Tests aus *Arrange*, *Act* und *Assert*.
- Im **Arrange** wird die Testumgebung vorbereitet. Beispielsweise bei einer Abfrage von Daten aus einer Datenbank, werden hier die Daten, welche man abfragen möchte angelegt und in die InMemory-Datenbank gespeichert.
- Im **Act** folgt der Aufruf der Methode, welche man testen möchte. Hier verwenden wir unsere angelegte Call-Methode.
- Im **Assert** wird das Ergebnis überprüft, welches zurückkommt. Wir wollen beispielsweise eine leere Liste erhalten, weil wir keine Datenbankeinträge angelegt haben (um zu simulieren, dass keine vorhanden sind)? Also wird im Assert an den Compiler geschrieben, dass wir eine leere Liste erwarten und zuzüglich vielleicht, dass die Datenbank (in dem Fall die InMemory-Datenbank) ebenfalls leer ist. Dieser Ausdruck wird anschließend ausgewertet. Ist dieser true, so ist der Test bestanden.

Wichtig: Der Testausdruck muss um den Test zu bestehen immer true zurückliefern, ganz egal ob wir mit einem `Should()`-Befehl beispielsweise einen Datensatz auf false testen. Dies gilt nur für das Ergebnis. Der Test am Ende muss immer true sein um bestanden zu werden.

5.2.4 Der Test - Explorer



Nun da wir unsere ersten kleinen Tests geschrieben haben können wir diese auch direkt überprüfen. Dazu öffnen wir den Test-Explorer (Im Daten-Explorer in der IDE mit einem Rechtsklick auf die Testklasse und dann auf „Tests ausführen“). Dieser gibt uns einen Überblick über sämtliche geschriebenen Tests und ob diese fehlgeschlagen sind, oder ob diese geöffnet haben. Bei einem Fehlschlag werden Informationen zum Grund als Exception geworfenen, welche der Entwickler analysieren und aufarbeiten muss. Manchmal reicht es einen Test anzupassen, weil der Ausdruck des Tests, welcher überprüft werden sollte schlicht falsch oder unvollständig ist, doch in vielen Situationen muss der Programm-Code angepasst werden. Schließlich ist dies auch der Grund, weshalb diese Tests ausgeführt werden.

6. Abschließende Worte zum Projekt

Ich habe mir in diesem Projekt die Aufgabe gestellt eine Webapplikation zu entwickeln, welche dafür verwendet werden soll Schulnoten von Schülern zu speichern. Jeder Schüler, welcher sich an der späteren Applikation anmelden würde, sollte so ein seine Noten aufzeichnen können, diese sollten auf Unterpunktung und Durchschnitt hin geprüft werden, sowie verwaltet werden können. Gleichzeitig sollte die Applikation als eine Art Terminplaner für anstehende Noten fungieren und beispielsweise bevorstehende Tests, Examen, Ausarbeitungen, oder auch Ferien und Feiertage anzeigen können.

Standpunkt nach der Projektarbeit ist, dass sich die Applikation weiter in der Entwicklungsphase befindet. Durch einige Rückschläge in den verschiedenen Stufen des Schaffensprozesses ist es nötig gewesen immer und immer wieder neu an das Projekt heranzugehen. Das notwendige Wissen darüber, wie man eine solche Applikation überhaupt umsetzen kann fehlte mir zu Beginn des Projektes und musste über die letzten Monate erst einmal aufgebaut und vertieft werden. Außerdem wurden Vorgängerversionen, welche nur auf eine Entwicklung der Applikation mit vanilla.JavaScript abzielten eingestampft werden, da diese Form der Entwicklung nicht ausgereicht hätte um eine agile Webanwendung zu erstellen und die Eingaben der Nutzer persistent speichern zu können. Die Art und Weise die Aufgabe nach der REST-Architektur zu entwickeln kam erst im Laufe der Entwicklungszeit und wurde seitdem nicht noch einmal verändert.

Als Abgabe des Projektes fungiert ein Datenträger des geschriebenen Codes, sowie diese Projektarbeit. Gleichzeitig möchte ich auf das Repository dieses Projektes verweisen, welche laufend aktualisiert werden wird und so immer den neuesten Stand des Programmes umfassen wird. Zu erreichen ist dieses Repository unter folgendem Link:

<https://github.com/robCode93/Facharbeit-Notenapp--Angular-und-C-Sharp->

Unter diesem Link befinden sich sämtliche Dateien dieses Projektes, welche per GitHub-GUILCient auf den jeweiligen PC geklont werden können und sich so auch die neustes Entwicklung angesehen werden kann.

7. Quellenverzeichnis

Quellenname	Webadresse/ ISBN/ ...s
GeeksForGeeks-Blog : Versionierungsnummer	https://www.geeksforgeeks.org/introduction-semantic-versioning/
Angular - Documentation	https://angular.io/docs
Jeberries: Syntaxfragen Angular	https://de.jberries.com/frage/wozu-dient-die-sternchen-syntax-f%C3%BCr-ngif-oder-ngfor-direktiven-in-angular-16
Happy-Angular Blog: Syntaxfragen zu Angular + Direktiven	https://happy-angular.de/direktive-nutze-die-macht-der-template-befehle-folge-6/
Refactoring Guru: DesignPatterns Übersicht	https://refactoring.guru/
IONIS: DesignPattern (Observer-Patterns)	https://www.ionos.de/digitalguide/websites/webentwicklung/was-ist-das-observer-pattern/
JavaBeginners-Blog: DesignPatterns	https://javabeginners.de/Design_Patterns/Observer_-Pattern.php
Wikipedia : Beobachtermuster (Softwarentwicklung)	https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)
RX.JS-Documentation	https://rxjs.dev/
StackOverflow: node	https://stackoverflow.com/questions/22343224/whats-the-difference-between-tilde-and-caret-in-package-json
npmjs: Dependecy-Injection	https://www.npmjs.com/package/node-dependency-injection
Microsoft-Documentation: LINQ	https://learn.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/
Microsoft-Documentation: ASP.NET	https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0