IHK Nord Westfalen

Westfälische Hochschule

# Projektdokumentation Friends and Places


vorgelegt von: **Tim Gövert**
**Jonathan Janzen**
**Maximilian Bär-Bietenbeck**
**Luca Fasselt**

Studiengang: Duales Studium - IT Betreibswirt/ -in VWA
Bachelor of Arts FR Wirtschaftsinformatik,
Gruppe 17

Themensteller: Prof. Dr. Jürgen Priemer
Betreuer: Prof. Dr. Jürgen Priemer



Abgabedatum: 07.07.2025

# Inhaltsverzeichnis

**1. Einleitung....1**
**2. Aufgabenstellung....1**
**3. Verwendete Techniken und Werkzeuge....2**
3.1 Auswahl des Technologie-Stacks....2
3.2 Docker und Containerisierung....2
3.3 GitHub und CI/CD-Pipeline....2
4.1 Einsatz von ASP.NET Core....3
4.2 Integration von Swagger / OpenAPI....3
4.3 Vorteile der Dokumentation im Entwicklungsprozess....4
**5. Architektur der Friends and Places API....5**
**5.1 Modularer Aufbau: Controller – Service – Model....5**
5.2 Dependency Injection....6
**6. Implementierung einzelner Funktionsbereiche....7**
6.1 Registrierung neuer Nutzer....7
6.2 Benutzer-Login und Session-Verwaltung....8
6.3 Standort teilen und abfragen....9
6.4 GeoService....11
**7. UML-Diagramme und Architekturmuster....12**
7.1 Klassendiagramm....12
7.2 Sequenzdiagramm für typische Abläufe....13
**8. Schnittstellenkonformität mit den Vorgaben....14**
**9. Qualitätssicherung und Tests....14**
**10. Herausforderungen bei der Umsetzung....15**
10.1 Eigenheiten der FAP-API....15
10.2 Probleme mit Geo-APIs....15
10.3 Hosting-Alternativen: Bewertung und Fazit....15
**11. Ausblick und Weiterentwicklung....16**
**12. Persönliche Erfahrungen und Bewertung....17**

# Abkürzungsverzeichnis

| | |
|---|---|
| API | Application Programming Interface |
| CI/CD | Continuous Integration / Continuous Deployment |
| DI | Dependency Injection |
| DTO | Data Transfer Object |
| FAP | Friends and Places |
| GUID | Globally Unique Identifier |
| HTTP | Hypertext Transfer Protocol |
| IDE | Integrated Development Environment |
| JSON | JavaScript Object Notation |
| PLZ | Postleitzahl |
| UML | Unified Modeling Language |
| UI | User Interface |

# Abbildungsverzeichnis


| | |
|---|---|
| Abb. 1 | Auszug aus den launchSettings.json |
| Abb. 2 | Auszug aus der swagger-ui |
| Abb. 3 | Auszug aus den swagger Kommentaren |
| Abb. 4 | Auszug Klasse UserModel und EmailModel |
| Abb. 5 | Auszug Klasse LoginModel und PasswordModel |
| Abb. 6 | Auszug Methode GetOrCreateToken |
| Abb. 7 | Auszug der Deklaration ConcurrentDictionary |
| Abb. 8 | Auszug der Klasse LocationModel und CoordinateModel |
| Abb. 9 | Auszug der Abfrage GetLocation |
| Abb. 10 | Auszug der Methode Task |
| Abb. 11 | Auszug der Klasse CoordinateModel |
| Abb. 12 | Auszug der Geoapify API |
| Abb. 13 | Auszug des Konstruktors LocationService |
| Abb. 14 | UML-Klassendiagramm |
| Abb. 15 | Sequenzdiagramm |
| Abb. 16 | Auszug Github deploymen |

# 1. Einleitung

Im Rahmen der Veranstaltung Fortgeschrittene Internettechnologien bestand das Ziel des Projektes darin, eine funktionsfähige Webanwendung zu entwickeln, die standortbezogene Informationen zwischen registrierten Nutzern austauscht. Die Anwendung „Friends and Places“ (FAP) soll es ermöglichen, nach Anmeldung den eigenen Standort zu melden und den Standort befreundeter Nutzer abfragen.

Die technische Umsetzung erfolgte mit einem Fokus auf moderne Technologien und Best Practices. Als Backend-Framework wurde ASP.NET Core in Kombination mit C# gewählt. Die gesamte Anwendung ist containerisiert und wird mithilfe von Docker bereitgestellt, wodurch eine flexible und reproduzierbare Ausführung möglich ist. Für die API-Dokumentation und Tests kommt Swagger zum Einsatz. Zusätzlich nutzen wir GitHub nicht nur zur Versionskontrolle, sondern auch für Continuous Integration und Deployment (CI/CD), um automatisierte Docker-Images zu erstellen und zu veröffentlichen.

Das Projekt verbindet praktische Erfahrungen in der Webentwicklung mit theoretischen Inhalten der Vorlesung. Es diente sowohl der technischen Vertiefung als auch dem Verständnis für saubere Schnittstellengestaltung, modulare Softwarearchitektur und den Einsatz moderner Entwicklungs- und Deployment-Werkzeuge.

# 2. Aufgabenstellung

Im Zentrum des Projekts steht die serverseitige Nachbildung der REST-Schnittstelle der „Friends and Places“-Anwendung. Ziel war es, ein vollständiges Backend zu entwickeln, das alle vorgegebenen Web Services bereitstellt und sich konform zur spezifizierten API-Schnittstelle verhält. Die Umsetzung des Frontends war nicht Bestandteil der Aufgabe.

Die API sollte dabei folgende Funktionalitäten unterstützen:

- Registrierung neuer Benutzer mit Adress- und Standortdaten,
- Validierung von Benutzernamen und Rückgabe des zugehörigen Orts zur PLZ,
- Authentifizierung mit Sitzungsverwaltung,
- Standortübermittlung und Standortabfrage registrierter Benutzer,
- Verwaltung einer Beobachtungsliste,
- Rückgabe aller registrierten Nutzer und deren Daten,
- Geocoding von Adressen.

Zusätzlich war gefordert, die Schnittstelle als REST-konforme Anwendung im JSON-Format umzusetzen und die Anwendung auf GitHub bereitzustellen.

Die fertige API entspricht funktional exakt den Vorgaben und kann somit als Drop-in-Replacement für den originalen Server verwendet werden.

# 3. Verwendete Techniken und Werkzeuge

## 3.1 Auswahl des Technologie-Stacks

Für die Umsetzung des Projekts entschieden wir uns für .NET Core und C#. Diese Wahl basierte vor allem auf unserer beruflichen Vorerfahrung mit dem .NET-Ökosystem. Die Plattform ermöglicht eine saubere Trennung der Architektur (Controller, Services, Modelle) und unterstützt modernes API-Design. Gegenüber klassischen Java-Lösungen bietet .NET Core einen geringeren Setup- und Infrastruktur-Aufwand und ist besonders gut in Container-Umgebungen integrierbar.

Ein weiterer Vorteil ist der native Support für Dependency Injection (DI), der ohne zusätzliche Frameworks auskommt und zu einem sauberen, modularen Aufbau beiträgt. Zusätzlich ermöglicht .NET Core eine automatische Generierung der OpenAPI-Dokumentation mit Swagger, was die Entwicklung und das Testen der Schnittstellen stark vereinfacht. Auch in Bezug auf Performance bietet die Plattform eine solide Grundlage, gerade im Hinblick auf die Umsetzung effizienter REST-APIs.

## 3.2 Docker und Containerisierung

Die gesamte Anwendung wurde als Docker-Container umgesetzt. Dadurch können alle Abhängigkeiten in einem lauffähigen Image gebündelt werden – unabhängig vom Zielsystem.  Zwar haben wir keine geeignete Hosting-Plattform für die Ausführung gefunden, aber durch den Push in die Registry ist ein späteres Deployment jederzeit möglich. Der Fokus lag auf einer sauberen Container-Bereitstellung und nicht auf einer vollwertigen Produktionsumgebung.

## 3.3 GitHub und CI/CD-Pipeline

Der Quellcode wird in einem GitHub-Repository verwaltet. Über GitHub Actions wurde eine einfache, aber effektive CI/CD-Pipeline eingerichtet: Bei jedem Push auf den main-Branch wird das Projekt automatisch gebaut und ein neues Docker-Image erzeugt. Dieses wird direkt in die GitHub Container Registry geladen. Damit stellen wir sicher, dass stets eine aktuelle und einsatzbereite Version der Anwendung zur Verfügung steht – selbst wenn noch keine Hosting-Umgebung existiert. Ebenfalls haben wir die Dokumentation zum Projekt mithilfe der Readme.md Datei in Github umgesetzt. Diese enthält Informationen zum Entwickeln, Debuggen sowie zum Testen und Deployment der Anwendung.

Zuletzt ermöglicht Github eine konsequente Umsetzung des 4 Augen Prinzips durch Pull-Requests.

# 4. Framework und API-Dokumentation

## 4.1 Einsatz von ASP.NET Core

Die API wurde mit ASP.NET Core umgesetzt, einem modernen Web Framework, das speziell für den Aufbau schlanker und performanter REST-Services entwickelt wurde. Die klare Struktur und das integrierte Routing-Modell haben es erleichtert, die Anforderungen aus der Vorgabe in einzelne Endpunkte zu überführen.Ein weiterer Vorteil von ASP.NET Core ist die hohe Entwicklungsfreundlichkeit: Durch die **launchSettings.json** kann die Anwendung direkt aus der IDE (z. B. Visual Studio oder Rider) mit vordefinierten Umgebungsvariablen und Konfigurationen gestartet werden. So lässt sich sowohl die lokale Entwicklung als auch das Debugging komfortabel durchführen, ohne zusätzlichen Konfigurationsaufwand.

Die Kombination aus guter Performance, einfacher Testbarkeit und klarer Trennung von Verantwortlichkeiten (Controller, Services, Modelle) macht ASP.NET Core zu einer idealen Wahl für diese Art von Projekt.

```
{
  "profiles": {
    "http": {
      "commandName": "Project",
      "launchBrowser": false,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "GEOAPIFY_API_KEY": "-----DO NOT COMMIT----"
      },
      "dotnetRunMessages": true,
      "applicationUrl": "http://localhost:5227"
    }
  }
}
```

**Abb. 1 Auszug aus den launchSettings.json**

## 4.2 Integration von Swagger / OpenAPI

Zur automatischen Dokumentation der Schnittstellen kam Swagger (OpenAPI) zum Einsatz. Durch eine einfache Konfiguration wurde direkt aus dem Code eine interaktive API-Dokumentation generiert, die sowohl die Struktur als auch die Nutzbarkeit der Schnittstelle transparent macht.

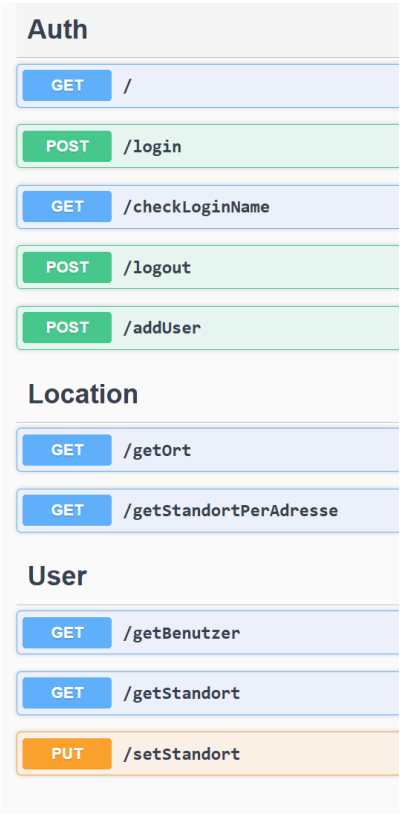


**Abb. 2 Auszug aus der swagger-ui**

Durch Kommentare im Code (doc-string) ermöglicht Swagger eine detaillierte Dokumentation aller API Routen im Code selbst. So wird auf eine externe Dokumentation verzichtet, was den Aufwand für Änderungen drastisch reduziert. Ebenfalls erleichtert diese Dokumentationsweise das Verständnis des Quellcodes, da Abschnitte der Doku nicht gesucht werden müssen.

```
//Service4
/// <summary>
/// Authenticates a user and returns an authentication token.
/// </summary>
/// <param name="model">The login credentials.</param>
/// <returns>Returns an Ok result with either a session ID or an empty object.</returns>
/// <response code="200">Always returns 200 OK. Returns a JSON object with 'sessionID' if login is successful
```

**Abb. 3 Auszug aus den swagger Kommentaren**

## 4.3 Vorteile der Dokumentation im Entwicklungsprozess

Die Verwendung von Swagger brachte klare Vorteile im Entwicklungs Alltag: Alle Endpunkte konnten direkt über das UI-Interface getestet werden – ohne externe Tools oder eigene Clients. Änderungen an der API waren sofort sichtbar und testbar, was die Kommunikation und Überprüfung deutlich vereinfachte.

# 5. Architektur der Friends and Places API

## 5.1 Modularer Aufbau: Controller – Service – Model

Zur besseren Strukturierung und Wartbarkeit wird der FAP Server Code nach dem Controller-Service-Model Pattern aufgebaut. Dabei wird die Anwendung als solche in die drei dem Namen entsprechenden Hauptbestandteile: Controller, Service, Model.
Ziel dieser Trennung ist es, Verantwortlichkeiten klar abzugrenzen und den Code übersichtlich sowie gut testbar zu gestalten.

Der Controller stellt die Schnittstelle zwischen der Anwendung und der Außenwelt dar - im FAP Server geschieht das über HTTP. Er nimmt Anfragen entgegen, prüft gegebenenfalls Eingaben und leitet diese an die zuständigen Services weiter. Darüber hinaus ist der Controller für die Rückgabe geeigneter Antworten zuständig, beim FAP Server ist dies stets im JSON Format. Die eigentliche Geschäftslogik ist jedoch nicht im Controller selbst, sondern in einem separaten Service untergebracht.

Der Service Layer kapselt die Geschäftslogik der Anwendung. Er verarbeitet Daten, führt Validierungen durch, koordiniert Abläufe und agiert als Vermittler zwischen Controller und Datenzugriffsschicht. Durch die Auslagerung der Logik in Services bleibt der Controller schlank und fokussiert, während gleichzeitig die Logik in einer zentralen und wiederverwendbaren Form zur Verfügung steht. Services werden beim FAP Service über Dependency Injection in die Controller eingebunden.

Die Model-Schicht umfasst Datenstrukturen wie Entity-Modelle und Data Transfer Objects (DTO’s), die zur Übertragung von Daten zwischen den Schichten verwendet werden. Die Models repräsentieren die Speicherobjekte des FAP Server, wie z.B. die Nutzerdaten oder die Session Tokens , während DTO’s zur Abstraktion der übertragenen Daten in den Controllern dienen.

Aufgrund einer sauberen Trennung von Verantwortlichkeiten und einer guten Test- sowie Wartbarkeit wird im FAP Server Projekt umfassend auf das Controller-Service-Model-Pattern gesetzt.

## 5.2 Dependency Injection

Durch Dependency Injection können Abhängigkeiten zwischen Komponenten (z. B. zwischen Controller, Service und Repository) flexibel und lose gekoppelt verwaltet werden. Anstatt Instanzen manuell zu erstellen, werden sie automatisch vom Framework zur Laufzeit bereitgestellt. Das ermöglicht es, konkrete Implementierungen jederzeit durch andere auszutauschen – etwa für Tests, alternative Datenquellen oder neue Anforderungen – ohne Änderungen am aufrufenden Code vornehmen zu müssen.

Ein weiterer entscheidender Vorteil ist die bessere Testbarkeit: Durch das Injizieren von Abhängigkeiten lassen sich einzelne Komponenten leicht isolieren und mittels Mock-Objekten gezielt testen. So kann z.B. die Datenbankschicht deterministisch simuliert werden, um Unit Tests stabil einsetzen zu können.

DI verbessert zudem die Lesbarkeit und Struktur des Codes. Die Konstruktoren zeigen klar auf, welche Abhängigkeiten eine Klasse benötigt, was die Wartung erleichtert.

Insgesamt ermöglicht Dependency Injection eine saubere Trennung der Verantwortlichkeiten, hohe Flexibilität und eine nachhaltige Codebasis, die zukunftssicher erweitert und gepflegt werden kann.

# 6. Implementierung einzelner Funktionsbereiche

Im Folgenden wird exemplarisch auf einige Routen der API sowie auf einige Besonderheiten in der Implementierung eingegangen. Es werden Codestellen erläutert und Entscheidungen in der Umsetzung nachvollziehbar dargelegt.

## 6.1 Registrierung neuer Nutzer

Zur Registrierung eines neuen Benutzers wird ein **POST** HTTP-Request gegen die Route **/addUser** durchgeführt. Dabei wird im Body der Anfrage folgendes DTO übergeben, das vom Registrierenden mit Informationen zu füllen ist:

```
public class UserModel : LoginModel
{
    public string Vorname { get; set; }
    public string NachName { get; set; }
    public string Strasse { get; set; }
    public string Plz { get; set; }
    public string Ort { get; set; }
    public string Land { get; set; }
    public string Telefon { get; set; }
    public EmailModel Email { get; set; }
}

public class EmailModel
{
    public string Adresse { get; set; }
}
```

**Abb. 4 Auszug Klasse UserModel und EmailModel**

Bei Erhalt einer Anfrage wird zunächst der Nutzername auf Korrektheit und Regelkonformität geprüft. Der Nutzername darf nicht kürzer als 5 Zeichen lang sein und darf nicht schon registriert sein.

Ist diese Prüfung erfolgreich, wird der Nutzer vom **UserService** in die Liste der User aufgenommen.

Bei Erfolg, sowie bei Misserfolg der Anfrage wird ein 200er zurückgegeben, der im DTO das Ergebnis als boolean mitgibt.

## 6.2 Benutzer-Login und Session-Verwaltung

Als registrierter Benutzer kann ich mich einloggen, um die API in ihrer Vollständigkeit nutzen zu können. Dieser Login Prozess und die Authentifizierung danach werden über SessionTokens gelöst.
Um sich anzumelden, sendet der Nutzer (das Frontend) einen HTTP-Request an die **/login** Route. In dieser Anfrage muss in Form des folgenden DTO's der Nutzername sowie das Passwort mitgegeben werden:

```
public class LoginModel
{
    public string LoginName { get; set; }
    public PasswordModel Passwort { get; set; }
}

public class PasswordModel
{
    public string Passwort { get; set; }
}
```

**Abb. 5 Auszug Klasse LoginModel und PasswordModel**

Der FAP-Server überprüft dann mit Hilfe des UserServices, ob dieser Nutzer mit diesem Passwort existiert. Ist dies nicht der Fall, wird die Anfrage vorzeitig beendet und der Nutzer bekommt ein leeres Json zurückgegeben.

Bei erfolgreicher Validierung der Daten wird vom **AuthService** ein neues Session Token generiert:

```
public string GetOrCreateToken(string loginName)
{
    _authTokens.TryGetValue(loginName, out var t :string? );
    var token :string  = t ?? Guid.NewGuid().ToString();
    _authTokens.TryAdd(loginName, token);

    return token;
}
```

**Abb. 6 Auszug Methode GetOrCreateToken**

In der ersten Zeile der Methode wird überprüft, ob der User eventuell schon angemeldet ist, also sich schon ein Token als solches im Speicher befindet. Ist dies der Fall, wird korrekterweise das alte Token zurückgegeben und kein neues generiert.

Bei neuer Anmeldung wird vom .NET internen Tool **GUID** eine neue Unique ID erzeugt. Diese wird dann zusammen mit dem **loginName** des Nutzers in das **_authToken** Dictionary geschrieben.
Hierbei handelt es sich um ein Concurrent-Dictionary

```
private readonly ConcurrentDictionary<string, string> _authTokens
```

**Abb. 7 Auszug der Deklaration ConcurrentDictionary**

Dieses hat die Eigenschaft, im Gegensatz zum normalen Dictionary Thread Safe zu sein. So stellen wir sicher, dass auch bei Betrieb unter sehr viel mehr Last, wenn der Service horizontal zu skalieren beginnt, alle Zugriffe problemlos und korrekt verarbeitet werden können.

Race Conditions, also mehrfacher Zugriff auf dieselben Ressourcen zur gleichen Zeit, bei der Erstellung von Tokens sind somit ausgeschlossen.
Das erstellte Token wird dem Nutzer dann zurückgegeben.

## 6.3 Standort teilen und abfragen

Den eigenen Standort kann man mithilfe der **/setStandort** Route setzen, sobald man sich eingeloggt hat.
Dabei muss das folgende DTO vom Nutzer in der Anfrage mitgegeben werden:

```
public class LocationModel
{
    public string LoginName { get; set; }
    public string Sitzung { get; set; }

    public CoordinateModel Standort { get; set; }
}

public class CoordinateModel
{
    public float? Breitengrad { get; set; }
    public float? Laengengrad { get; set; }
}
```

**Abb. 8 Auszug der Klasse LocationModel und CoordinateModel**

Daraufhin wird die Sitzung mithilfe des **LoginName** validiert und bei erfolgreicher Validierung wird der Standort für den Nutzer gesetzt.

Dieser wird zweckmäßigerweise mit allen anderen Daten des Nutzers gespeichert und wird somit auch nach dem Logout und späteren Login persistiert.

Beim Abfragen des Standortes eines anderen Nutzers ist kein DTO erforderlich, da alle benötigten Informationen über die Query der Anfrage mitgeliefert werden.

```
GetLocation([FromQuery] string login, [FromQuery] string session, [FromQuery] string id)
```

**Abb. 9 Auszug der Abfrage GetLocation**

Zunächst wird der Nutzer durch den **AuthService** validiert. Ist diese Validierung nicht erfolgreich, bekommt der Nutzer ein leeres JSON als Antwort.

Nach erfolgreicher Validierung wird der Standort des angefragten Nutzers aus dem **UserService** abgefragt. Die Methode im **UserService** sieht dabei wie folgt aus:

```
public async Task<CoordinateModel>? GetLocation(string loginName)
{
    var user = GetUser(loginName);
    if (user == null)
    {
        return null;
    }
    if (_authTokenService.HasAuth(loginName))
    {
        if (user.Breitengrad != null && user.Laengengrad != null)
        {
            return new CoordinateModel { Breitengrad = user.Breitengrad, Laengengrad = user.Laengengrad };
        }
    }

    return await _locationService.GetLocationFromAddress(country: user.Land, postalCode: user.Plz, city: user.Ort, user.Strasse);
}
```

**Abb. 10 Auszug der Methode Task**

Zunächst wird hier aus dem Speicher der Nutzer abgefragt und gespeichert. Dieser hat als solcher seine Heimatadresse hinterlegt.

Nun wird geprüft, ob der Nutzer online ist. Dies geschieht über die **HasAuth** Methode. Ist dies der Fall, wird überprüft, ob der Nutzer einen Standort gesetzt hat und dieser wenn möglich zurückgeben.
Ist der Nutzer nicht angemeldet oder hat er keinen Standort angegeben, wird die Heimatadresse mithilfe des **LocationService** aufgelöst und zurückgegeben.

Der Standort wird immer als Koordinatenpaar in Form des folgenden DTO's an den Nutzer übermittelt:

```
public class CoordinateModel
{
    public float? Breitengrad { get; set; }
    public float? Laengengrad { get; set; }
}
```

**Abb. 11 Auszug der Klasse CoordinateModel**

## 6.4 GeoService

Zur Ermittlung der Koordinaten mithilfe einer Adresse wurde der Service **Geoapify** genutzt. Bei diesem handelt es sich um einen OpenSource GeoCoding / Reverse Geocoding Service.
Zur Nutzung dieses Services benötigt man jedoch einen ApiKey. Dieser muss beim Request als Parameter mitgegeben werden:

```
$"https://api.geoapify.com/v1/geocode/search?text={street}%2C%20{postalCode}%20{city}%2C%20{country}&apiKey={_apiKey}"
```

**Abb. 12 Auszug der Geoapify API**

Einen Api Key in ein Github Repository zu pushen stellt sich aber als eine denkbar schlechte Idee heraus. Deshalb wird der ApiKey aus einer Umgebungsvariable ausgelesen:

```
public LocationService(HttpClient httpClient)
{
    _httpClient = httpClient;
    _apiKey = Environment.GetEnvironmentVariable("GEOAPIFY_API_KEY");
}
```

**Abb. 13 Auszug des Konstruktors LocationService**

Dies ermöglicht ein einfaches setzen des API Keys, vor allem im Docker Umfeld, was die bevorzugte Laufzeitumgebung des FAP Services ist.
Zum lokalen Debugging kann der APIKey in der launchsettings.json gesetzt werden und muss nicht hart im Code hinterlegt werden.
Allerdings setzt dieser APIKey voraus, dass jeder Host des FAP Servers sich einen solchen erstellt.

# 7. UML-Diagramme und Architekturmuster

## 7.1 Klassendiagramm

WIe bereits in 5.1 erwähnt, ist die Architektur nach dem Controller-Service-Muster aufgebaut, wobei jede wichtige Komponente für die App jeweils einen Controller hat, der die verschiedenen API-Endpunkte für Operationen bereitstellt und logische Operationen mithilfe des Services bearbeitet. Im Kern sind die wichtigsten für die Anwendung die Authentifizierung, der User und die Location. Wie die unten stehende Abbildung verdeutlicht, sind die einzelnen Controller nicht auf den Service ihres Objekts beschränkt, sondern müssen auch mit anderen Services interagieren, damit  zum Beispiel beim User-Login auch direkt der Standort mit abgefragt werden kann. In der Darstellung werden Controller und Service farbig unterschieden.

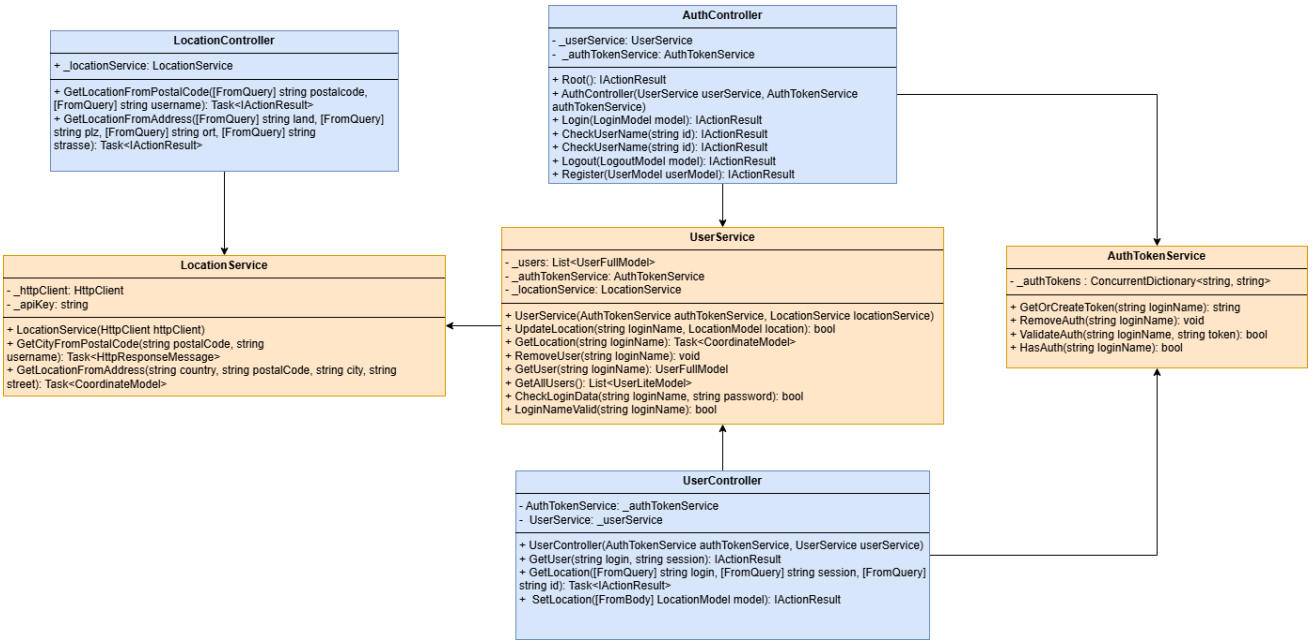


**Abb. 14 UML-Klassendiagramm**

## 7.2 Sequenzdiagramm für typische Abläufe

Die folgende Abbildung zeigt den Ablauf der Funktion GetUser aus dem UserController.

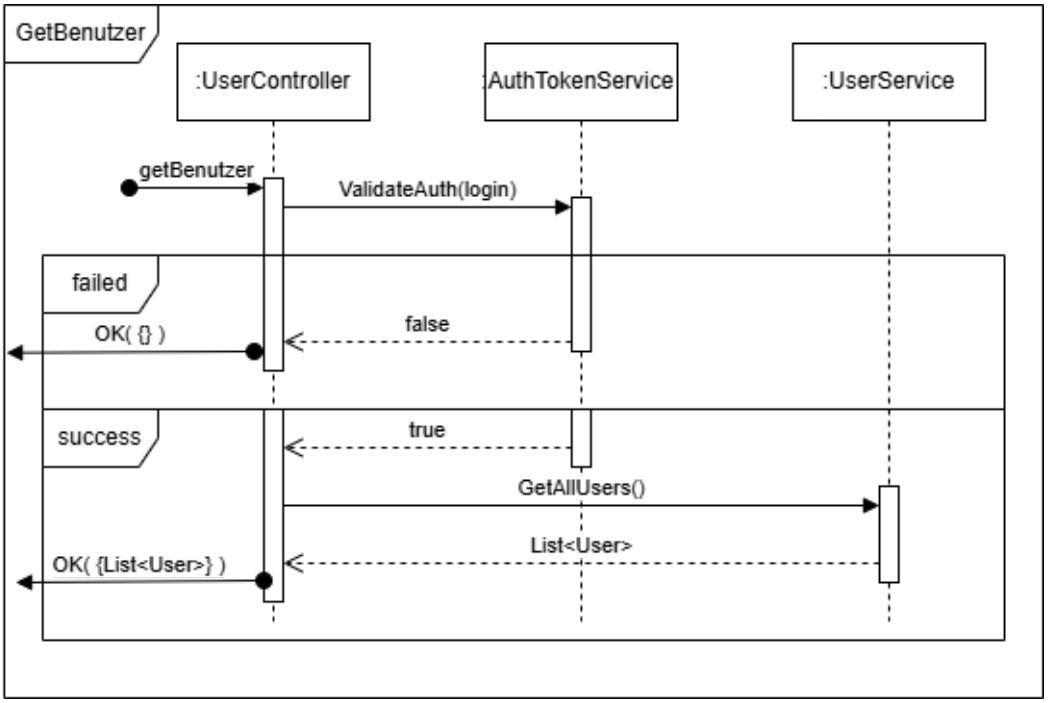


**Abb. 15 Sequenzdiagramm**

Das Sequenzdiagramm veranschaulicht den Ablauf des Methodenaufrufs **GetBenutzer**, bei dem eine Liste aller Benutzer abgerufen werden soll. Der Prozess beginnt mit einem Aufruf der Methode **getBenutzer** im UserController. Dieser leitet zunächst eine Anfrage zur Authentifizierungsprüfung an den **AuthTokenService** weiter, konkret durch den Aufruf der Methode **ValidateAuth(login)**.

Abhängig vom Ergebnis dieser Überprüfung unterscheidet sich der weitere Verlauf: Fällt die Authentifizierung negativ aus (der **AuthTokenService** liefert **false** zurück), antwortet der **UserController** mit einem leeren Ergebnis (**OK {}**), ohne weitere Schritte einzuleiten.

Im Falle einer erfolgreichen Authentifizierung (der Rückgabewert ist true) ruft der UserController die Methode **GetAllUsers()** beim UserService auf, um die Benutzerdaten zu laden. Der UserService gibt daraufhin eine Liste von Benutzern (List<User>) zurück, die schließlich vom UserController in der HTTP-Antwort (**OK(List<User>)**) übermittelt werden.

# 8. Schnittstellenkonformität mit den Vorgaben

Ziel der Implementierung war es, die vorgegebene Implementierung des FAP-Server in Bezug auf die API identisch nachzubauen.
Bei Beginn des Projekts war geplant, die Status Codes und Antworten des FAP Servers zu modernisieren und dem aktuellen Standard entsprechend zu gestalten. Davon wurde zuletzt abgesehen, da dies den Austausch mit anderen Backend Implementierungen unmöglich machen würde, welche genau auf die bestehende API angepasst sind.
Die gegebene REST-Schnittstelle wurde mit dem neuen Backend vollumfänglich mit den bestehenden Funktionen umgesetzt und erfolgreich getestet. Wie schon gesagt, wurde diese identisch nachgebaut und hat damit auch keine großen Auswirkungen auf das bestehende Verständnis der Nutzbarkeit, da die Eingaben und Ausgaben weiterhin dieselben sind.

# 9. Qualitätssicherung und Tests

Die Qualitätskontrolle des FAP Service unterteilt sich in 3 Grundlegende Stages, welche sicherstellen sollen, dass der ausgelieferte Service, der bei potenziellen Kunden landet, korrekt funktioniert.

Unit Tests sind im Code umfangreich enthalten und Testen alle deterministischen Logiken ab. Dabei fokussieren sich Unit Tests eher auf einzelne kleine Teilbereiche und sind somit nicht in der Lage, komplexe Zusammenhänge wie “Login + SetStandort + getStandort “auszuführen.
Dennoch haben Unit Tests die wichtige Aufgabe, sicherzustellen, dass Änderungen im Code bestehende Logiken nicht verändern.

Die Github Action (CI/CD) hat die Aufgabe, den Code zu kompilieren, sowie die Unit Tests auszuführen. Die Pipeline läuft bei jedem Commit auf den **main** Branch und sorgt somit für eine konstante Überprüfung des Codes.

Zusätzlich zur Testausführung ist die Pipeline dafür zuständig, den FAP Server als Docker-Image zu bauen und zu kompilieren und dieses dann in die Registry hochzuladen. Somit ist jeder Stand, zu dem eine Pipeline läuft, auch direkt deploybar.

Zuletzt werden nach dem Deployment oder teilweise auch während des Entwicklungsprozesses manuelle Tests durchgeführt. Diese ermöglichen es, Randfälle sowie zusammenhängende Logiken zu testen. Allgemein sollte alles getestet werden, was nicht durch Unit Tests erfasst ist. Manuelle Tests wurden hauptsächlich mithilfe des Swagger Tools durchgeführt, was vorgefertigte Requests und den Aufbau von DTO’s vorgibt und somit das Testen deutlich vereinfacht.

# 10. Herausforderungen bei der Umsetzung

## 10.1 Eigenheiten der FAP-API

Ein Problem bei der Umsetzung waren die Eigenheiten der FAP-API. Zunächst war es verwirrend, dass die Responses immer mit einem 200 zurückkamen, auch wenn diese nicht dem Standard entsprechen. Hier war es wichtig, dass der Body der Antwort mit betrachtet wird, da bei falscher Anfrage ein **false** zurückgegeben wird. Initial wurde davon ausgegangen, dass mit den üblichen HTTP-Response-Codes gearbeitet wird und diese auch passend verwendet werden, was jedoch nicht der Fall war. Ebenso war nicht immer klar, was für Eingaben bei der Anfrage erforderlich waren, was das Testen erschwert hat.

## 10.2 Probleme mit Geo-APIs

Auch bei den Geo-APIs gab es vermehrt Probleme. Zunächst hat die vorgeschlagene API nur für die Umwandlung von Postleitzahlen in Ortsnamen funktioniert. Damit waren die restlichen benötigten Funktionen für die App nicht verfügbar und es musste eine neue und geeignete API für Geodaten gesucht werden.
Hierbei wurde sich dann für eine Kombination aus zwei APIs entschieden. Dabei handelt es sich um **geoapify** und **geonames**.
Geonames ist dazu da, um die Stadtnamen anhand der PLZ zu bekommen. Geoapify ist dazu da, um die Adresse in Koordinaten umzuwandeln und in einer weiteren Ausbaustufe eine Karte anzuzeigen.

## 10.3 Hosting-Alternativen: Bewertung und Fazit

Zur Bereitstellung des Backends wurden verschiedene Hosting-Anbieter recherchiert und getestet. Dabei stellte sich die Auswahl jedoch als herausfordernd dar, da jeder Anbieter spezifische Eigenheiten oder Einschränkungen aufwies, die nicht mit den Anforderungen oder Möglichkeiten der Projektgruppe vereinbar waren.

Zunächst wurden Google Cloud und Amazon Web Services evaluiert, da diese in den Projektanforderungen genannt wurden. Beide Optionen schieden jedoch frühzeitig aus, da für die Registrierung zwingend eine Zahlungsmethode hinterlegt werden musste – in der Regel eine Kreditkarte, über die keines der Gruppenmitglieder verfügte.

Im Anschluss wurde die Plattform Vercel getestet. Hier bestand die Schwierigkeit darin, dass C# nicht als unterstützte Sprache für das Hosting angeboten wurde. Ein Versuch, dennoch ein Docker-Image zu deployen, blieb leider ohne Erfolg.

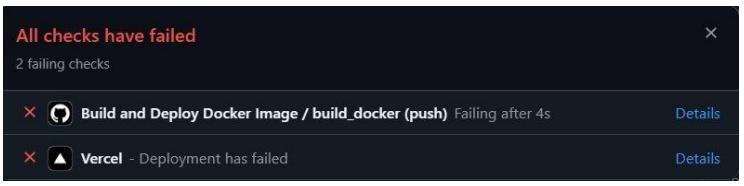


**Abb. 16 Auszug Github deployment**

Fazit des Ganzen ist, dass es schwer ist für ein UNI-Projekt, sei es durch technische Limitierungen oder durch Barrieren, wie Zahlungsbedingungen, eine geeignete Hosting-Alternative für die entwickelte App zu finden, die nicht gerade damit verbunden ist, einen Server selbst zu hosten und diesen von Grund auf zu konfigurieren.

# 11. Ausblick und Weiterentwicklung

Auch wenn die grundlegenden Anforderungen der FAB-API erfolgreich umgesetzt wurden, haben sich im Verlauf der Entwicklung sowie bei der finalen Präsentation mehrere Ansatzpunkte für eine sinnvolle Weiterentwicklung und Verbesserung ergeben. Ein zentraler Punkt ist dabei der aktuell nicht standardkonforme Umgang mit HTTP-Statuscodes. Die API gibt derzeit bei jeder Antwort den Statuscode **200 OK** zurück, unabhängig davon, ob die Anfrage erfolgreich oder fehlerhaft war. Für eine realitätsnahe und robuste Anwendung wäre es jedoch sinnvoll, korrekte HTTP-Statuscodes zu verwenden, z.B. **400 Bad Request** bei fehlerhaften Nutzereingaben, **401 Unauthorized** bei fehlender Authentifizierung oder **404 Not Found**, wenn angefragte Ressourcen nicht existieren. Dadurch würde die Schnittstelle deutlich aussagekräftiger und besser wartbar.
Ein weiterer Punkt betrifft die Art und Weise der Authentifizierung. Aktuell wird das Session-Token im Body des Requests übertragen. In modernen REST-APIs ist es jedoch üblich, Authentifizierung über den Authorization-Header abzuwickeln, z.B. per **Bearer-Token**. Diese Änderung würde nicht nur den gängigen Konventionen besser entsprechen, sondern auch die Lesbarkeit und Konsistenz der Schnittstelle verbessern. Ebenfalls entspricht das Senden des Session Tokens im Body nicht den heutigen Sicherheitsanforderungen und Standards, sodass ein Umbau auf Header als zwingend erforderlich angesehen wird.
Auch in Bezug auf die Sprache im API-Design besteht Verbesserungspotenzial. Derzeit werden teils deutsche Feldnamen in den Rückgaben verwendet. Um die API international anschlussfähig und einheitlich verständlich zu gestalten, sollte konsequent auf englischsprachige Feldnamen umgestellt werden.
Neben diesen Verbesserungen im Detail ist auch eine technische Weiterentwicklung denkbar. Die Einführung einer Version Struktur in den Routen, beispielsweise **/api/v1/…** würde ermöglichen, bei späteren Änderungen oder Erweiterungen der API abwärtskompatible Versionen beizubehalten. Außerdem könnte die bislang speicherbasierte Datenhaltung durch eine persistente Speicherung in einer Datenbank abgelöst werden. Dies würde nicht nur die Daten langfristig sichern, sondern auch den Grundstein für zukünftige Ausbaustufen legen.
Auf architektureller Ebene ist die bestehende Containerisierung mit Docker bereits eine gute Grundlage für Skalierung und Deployment. Eine Weiterentwicklung könnte in Richtung Cloud-native Infrastruktur führen, um automatische Skalierung, resilientes Hosting und Lastverteilung zu ermöglichen.

# 12. Persönliche Erfahrungen und Bewertung

Rückblickend war das Projekt nicht nur technisch erfolgreich, sondern auch aus persönlicher Sicht eine sehr gewinnbringende Erfahrung. Der Gesamtaufwand war aus unserer Sicht gut machbar und angemessen verteilt. Durch die unterschiedlichen Schwierigkeitsgrade einzelner Aufgaben konnten sich alle Teammitglieder nach ihren individuellen Stärken einbringen. Einige Komponenten wie die grundlegenden API-Routen oder die Dockerisierung waren vergleichsweise schnell umzusetzen, während andere Bereiche insbesondere Authentifizierung, Geodaten und Testing, mehr Tiefgang und Recherche erforderten. Genau diese Mischung machte den Lernaspekt besonders hoch.
Besonders positiv war die Teamarbeit. Die Zusammenarbeit lief reibungslos, Abstimmungen erfolgten regelmäßig und effizient. Bei größeren Herausforderungen, wie etwa einem hartnäckigen Bug, der erst am Abend vor der Präsentation endgültig behoben werden konnte, zeigt sich, wie wertvoll eine gute Kommunikation und gegenseitige Unterstützung im Team sind. Die Nutzung von GitHub als zentrales Tool für Code-Verwaltung, CI/CD und Aufgabenverteilung hat sich sehr bewährt und eine moderne Form der Zusammenarbeit ermöglicht.
Gleichzeitig war über den Projektverlauf hinweg nicht immer ganz klar, welcher Umfang und welche inhaltliche Tiefe für Präsentation und Ausarbeitung tatsächlich erwartet wurden. Dennoch konnten wir das Projekt zielgerichtet umsetzen, die funktionalen Anforderungen vollständig erfüllen und das Ziel effizient und erfolgreich erreichen. Insgesamt war das Projekt eine wertvolle Gelegenheit, theoretisches Wissen in einem praxisnahen Kontext anzuwenden und moderne Technologien sinnvoll miteinander zu verknüpfen.