OSP-Abgabe Technischer Leiter

Grundlegendes

Prinzipien:

- Schichtenarchitektur
- Single Responsibility Principle
- Modularisierung
- Dependency Injection
- Vererbung (In Angular z.B. Base Store)
- API
- Authentication
- Observer Pattern (In Angular)

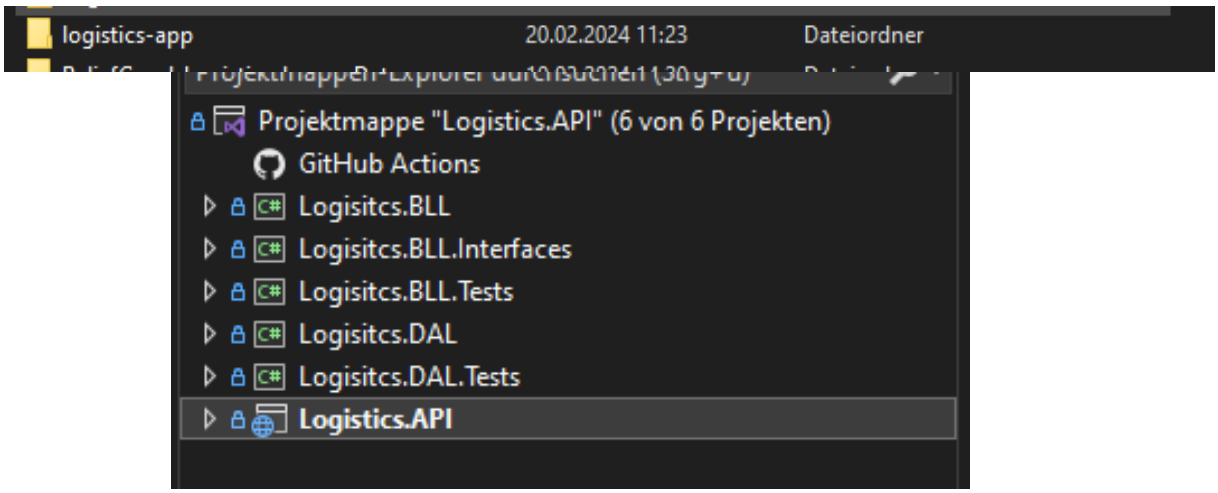
Methoden, Werkzeuge, Techniken:

- Dependency Injection über C# Startup

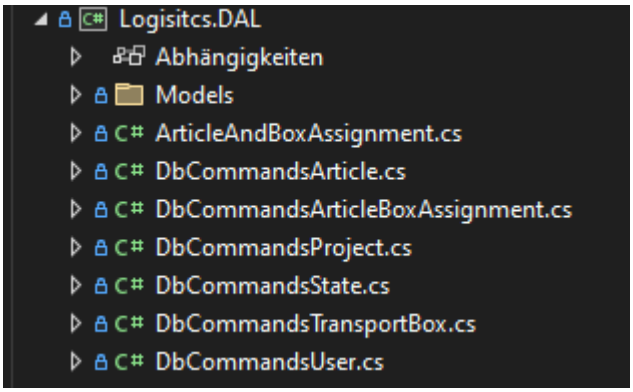
```
//DAL
services.AddTransient<IProjectDataFactory>(provider => new ProjectDataFactory());
services.AddTransient<IProjectFactory>(provider => new ProjectFactory());
services.AddTransient<ITransportBoxDataFactory>(provider => new TransportBoxDataFactory());
services.AddTransient<ITransportboxFactory>(provider => new TransportboxFactory());
services.AddTransient<IArticleAndBoxAssignmentFactory>(provider => new ArticleAndBoxAssignmentFactory());
services.AddTransient<IArticleDataFactory>(provider => new ArticleDataFactory());

//Helper
services.AddTransient<PdfHelper>(provider => new PdfHelper());

//BLL
services.AddTransient<IProjectBll>(provider => new ProjectBll(provider.GetService<IProjectDataFactory>(),
services.AddTransient<ITransportboxBll>(provider => new TransportboxBLL(provider.GetService<ITransportBox
services.AddTransient<IArticleBll>(provider => new ArticleBll(provider.GetService<IArticleAndBoxAssignmer
services.AddTransient<IPDFBLL>(provider => new PDFBLL(provider.GetService<PdfHelper>()));
services.AddTransient<ILoginBll>(provider => new LoginBll());
```

  Die Klassen werden einmal erzeugt und in den Service gespeichert, dann bei der Instanziierung der Bll-Klassen werden die Instanzen der Factories injectet. Dadurch wird nur eine Instanz erzeugt und es hat keine Direkte Abhängigkeit.

- Schichtenarchitektur durch verschiedene C# Projekte und ein Client (Angular-Projekt)



- Modularisierung und Single Responsibility Principle werden durch getrennte Dateien umgesetzt.



Z.b. wie hier: Die DbCommands sind für jedes Objekt getrennt, damit man schnell potenzielle Fehlerquellen findet und keine Überladenen Klassen hat.

- Vererbung wird in Angular als Form einer Basisklasse genutzt.

```
export abstract class BaseStoreService<T>
{
  public errorReceived: EventEmitter<any> = new E
```

Die abstrakte Klasse BaseStoreService wird von anderen Klassen erweitert und ihr wird eine Klasse (hier T) übergeben, damit die Base-Funktionen den richtigen Typ zurückgeben. Außerdem gibt der Service vor, welche Funktionen implementiert werden müssen.

- Die API wird mithilfe von dem MVC-Controller Pattern von Microsoft umgesetzt.

```
namespace Logistics.API.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    1 Verweis | Artur Sartison, Vor 16 Stunden | 2 Autoren, 10 Änderungen
    public class ArticleController : ControllerBase
    {
        private readonly IArticleBll articleBll;

        0 Verweise | Artur Sartison, vor 5 Tagen | 1 Autor, 2 Änderungen
        public ArticleController(IArticleBll articleBll)
        {
            this.articleBll = articleBll;
        }
```

Der Controller wird von dem bereitgestellten ControllerBase abgeleitet und implementiert mithilfe der Decoratoren die Funktionen bzw. die Endpunkte. Die Route des Controllers ist der Name ohne den Controllerpart (also hier Article).

- Die Authentication wird durch das von Microsoft bereitgestellten „Microsoft.AspNetCore.Authorization;“-Paket umgesetzt. Jeder abgesicherte Controller wird mit dem [Auhorize]-Decorator dekoriert, damit man nur mit einem gültigen Token den Endpunkt erreicht. (Siehe 1. Screenshot davor)

  In der Startup-Datei wird die Authentication implementiert:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = Configuration["Jwt:Issuer"],
        ValidAudience = Configuration["Jwt:Issuer"],
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
    };
});

services.AddMvc();
```

  Aus der app.settings.json werden die Issuer geholt und in den Parametern gespeichert. Bei erfolgreichem Login wird ein Token mit diesem Schema erstellt und an das Frontend zurückgegeben.

```
private string GenerateJSONWebToken(IUserData userInfo)
{
    var claims = GetValidClaims(userInfo);
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(_config["Jwt:Issuer"],
       _config["Jwt:Issuer"],
       claims,
       expires: DateTime.Now.AddMinutes(120),
       signingCredentials: credentials);
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

  Es werden verschiedene „Claims“ in das Token geschrieben (bei uns UserId, UserMail, UserRole), damit wir diese im Frontend auslesen und verwerten können. Unsere Tokens sind immer 120 Minuten gültig, danach muss man sich erneut einloggen.

- Das Observer Pattern wurde mithilfe von Behaviour-Subjects umgesetzt. Bei Veränderung der Werte werden alle Subscriber über die Veränderung benachrichtigt.

```
// behaviour subject das die Referenzen speichert und alle subscriber benachrichtigt
private readonly _source = new BehaviorSubject<T[]>([]);

// Read only Observable woran die Subscriber sich dran hängen können
public readonly items$ = this._source.asObservable();

// Gibt die letzen Änderungen des Observerables zurück.
public getItems(): T[] {
  return !!this._source.getValue() ? this._source.getValue() : [];
}
```

- Async/Await

```
2 Verweise | Rene Müller, Vor 21 Stunden | 1 Autor, 1 Änderung
public async Task<MemoryStream> GetStreamLogisticsDb()
{
    return await Task.Run(() =>
    {
        try
        {
            var databasePath = $"{Environment.CurrentDirectory}\\Fileserver\\logisticsDB.sqlite";

            MemoryStream ms = new MemoryStream();
            using (FileStream file = new FileStream(databasePath, FileMode.Open, FileAccess.Read, FileShare.ReadWrite))
                file.CopyTo(ms);

            return ms;
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex);
        }
```

  Außerdem nutzen wir async/await um unsere verschiedenen Aufrufe asynchron zu machen. Dadurch warten wir auf das Ergebnis, um Fehler durch gleichzeitige Operationen zu vermeiden.

Modularisierung

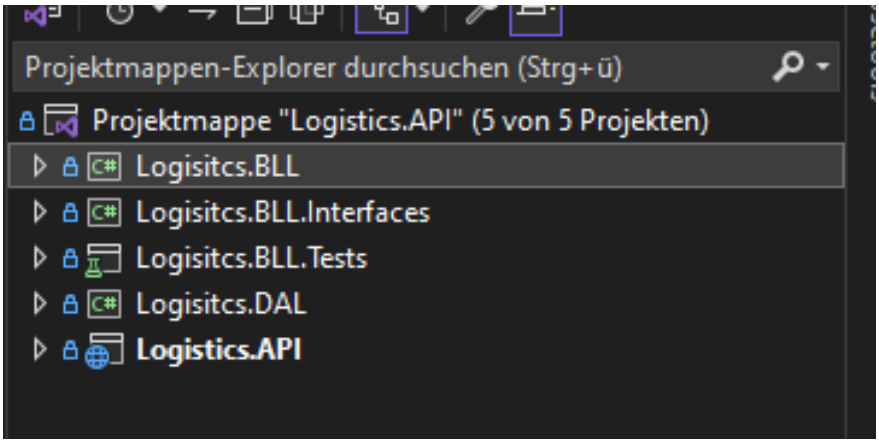
Frameworks, Patterns

Wie oben schon beschrieben haben wir die Patterns MVC, Dependency Injection und das Observer Pattern genutzt.

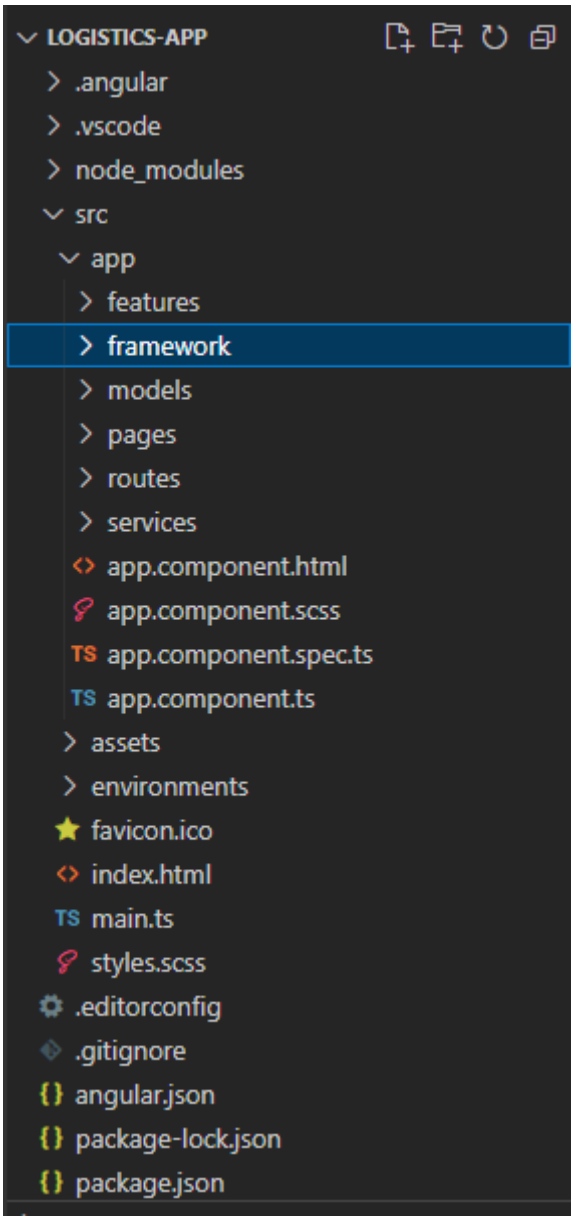
- Als Framework haben wir clientseitig Angular mit dem UI-Framework (Angular-) Material genutzt
- Als CSS-Framework haben wir Tailwind genutzt
- Und serverseitig haben wir ASP.Net Core genutzt

Struktur der Projekte:

C#:



Typescript (Angular)



Features: Die kleinsten Einheiten an Code (z.B. das Projektroulette zum Anzeigen der Projekte)

Framework: Alle erstellten Klassen, die so allgemein sind, dass man sie auch in deren Projekten nutzen könnte (z.B. eine Toolbar)

Models: Alle Interfaces und eine enum-Datei

Pages: Alle ansteuerbaren Seiten, dort werden die Features zu einer Seite zusammengeschnürt

Services: Angular-Service Klassen, die einfach in andere Klassen importiert werden können (Sind einfacher zu injecten, da sie nur eine Instanz haben und so geteilt genutzt werden können)

AppComponent: Unser entryPoint, dort sind die Framework Komponenten eingebunden und das RouterOutlet (dort wo die angerouteten Seiten angezeigt werden)

Visualisierung (mit hier im Ordner)

UML

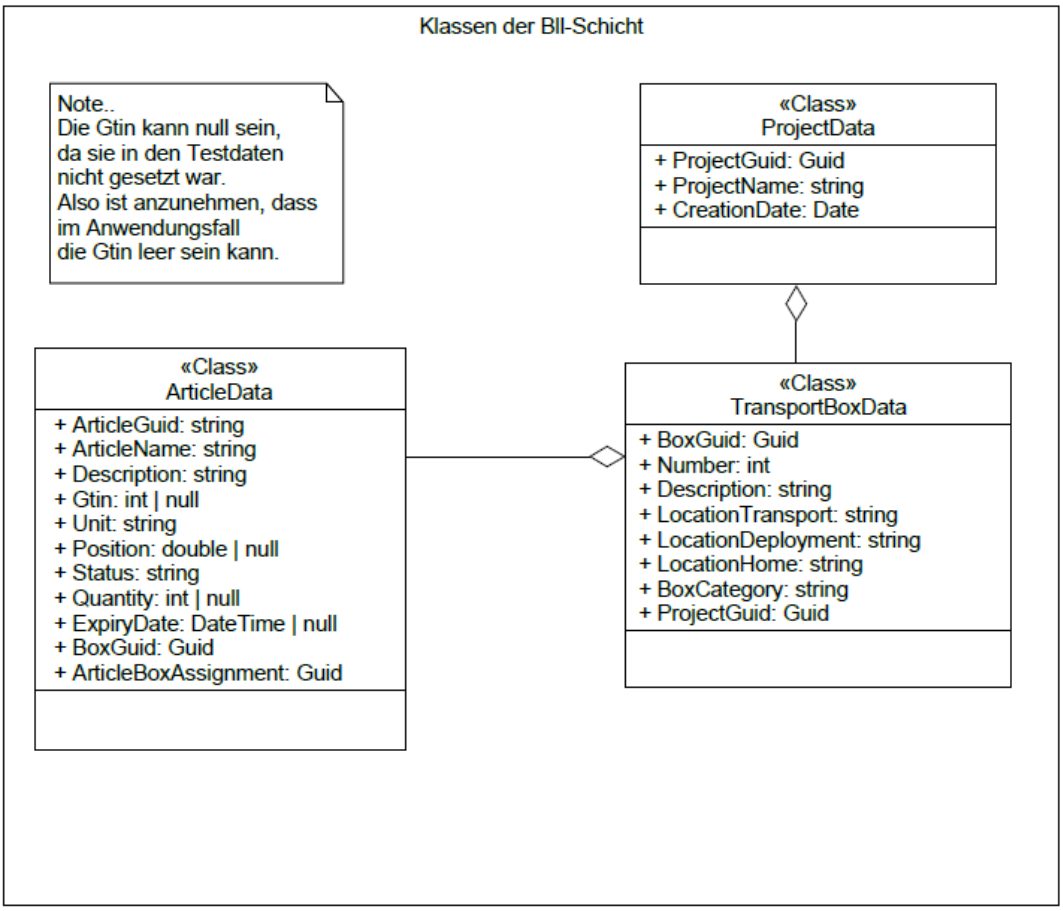


Abbildung 1- Klassendiagramm Bll

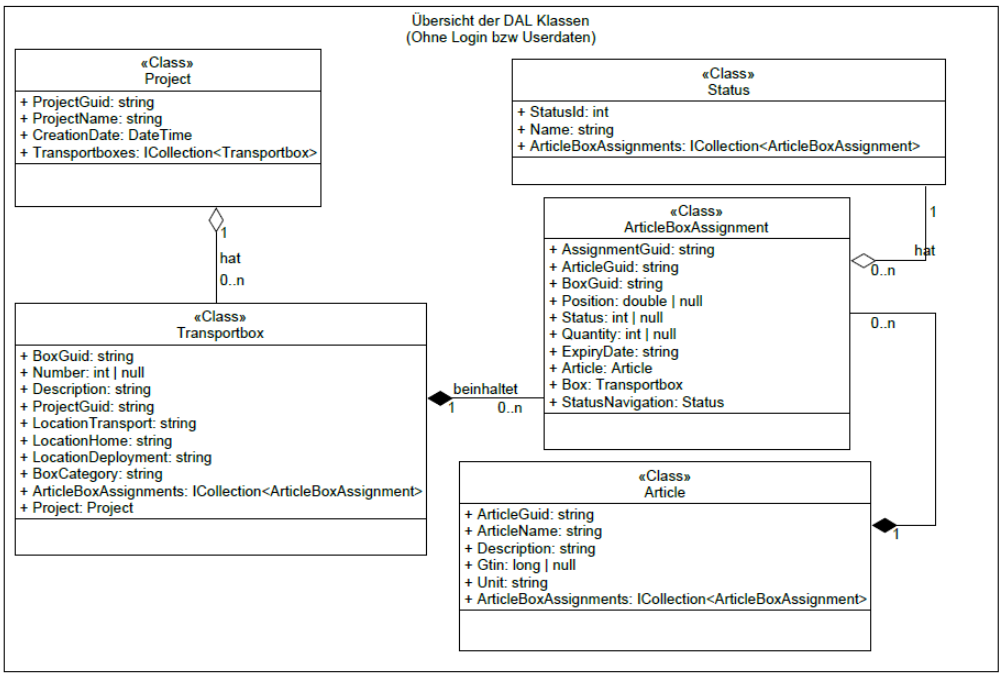


Abbildung 2 - Klassendiagramm Dal

Andere Formate

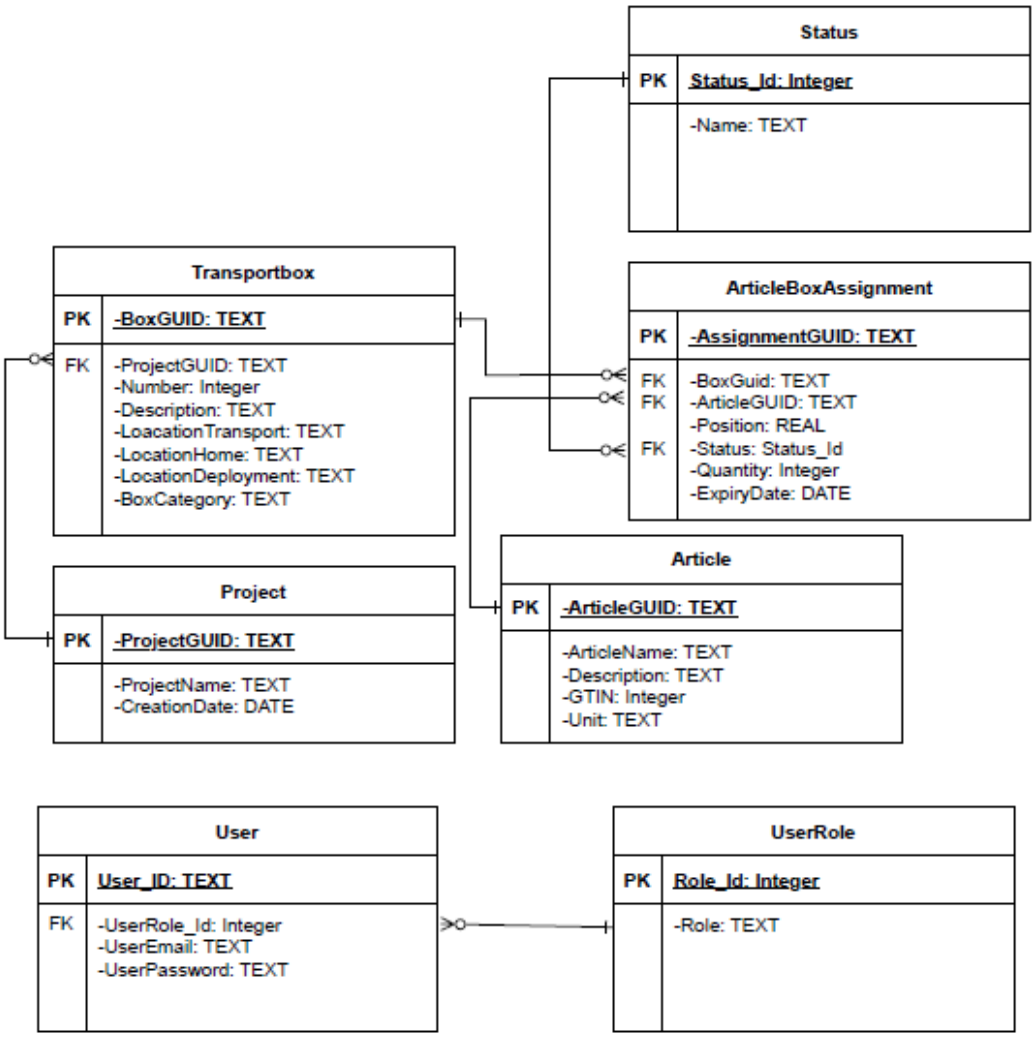


Abbildung 3 – Krähenfuß

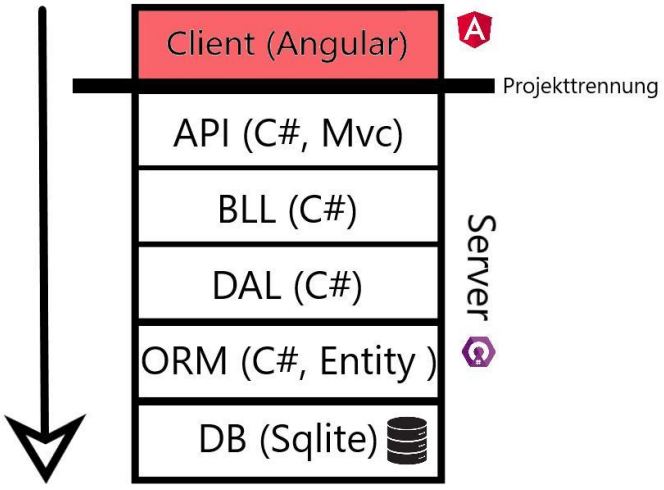


Abbildung 4 - Schichtenmodell

Sonstige Planung vom Anfang

Technische Umsetzung

1. **Datenbank**
   SQ-lite Datenbank

2. **ORM**
   Als ORM wird Entity Framework genutzt

3. **Erreichbarkeit**
   a. Die Datenbank wird über ein C#-Projekt erreichbar sein. Dabei wird eine API mithilfe des „Microsoft.AspNetCore.Mvc“-Pakets erstellt und über Visual Studio gehostet.
   b. Die Controller sollen mithilfe des „Microsoft.AspNetCore.Authorization“-Paketes abgesichert werden, damit nicht jeder Zugriff auf die Datenbank hat.

4. **Frontend**

   Das Frontend wird ein Angular Projekt, das bedeutet es wird nodeJs, npm-package benötigt. Als UI-Styling wird Angular Material verwendet. Das Projekt soll nach moderner Angular Struktur mit Standalone-Komponenten aufgebaut werden. Zusätzlich wird TailwindCss als Css Framework genutzt.