Architekt

Grundlegendes

Prinzipien:

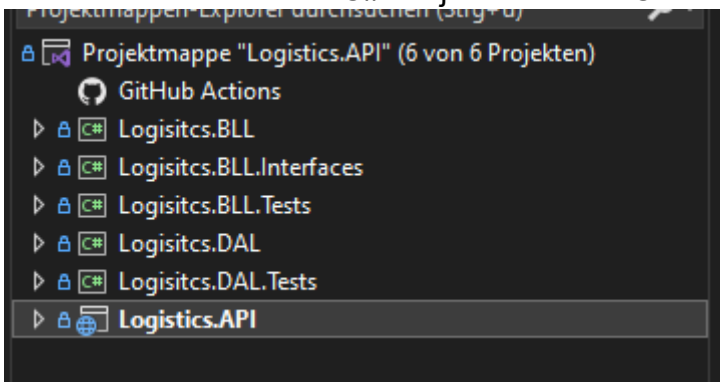
- Schichtenarchitektur
- Single Responsibilty Principle
- Modularisierung
- Dependency Injection
- Vererbung (Angularseitig z.B. BaseStore)
- API
- Authentication
- Observer Pattern (Angularseitig)

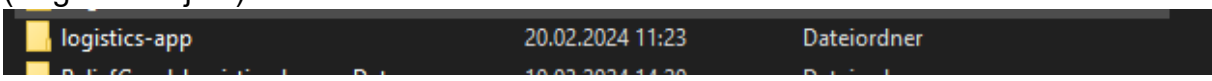
Methoden, Werkzeuge, Techniken:

- Dependency Injection über c# Startup

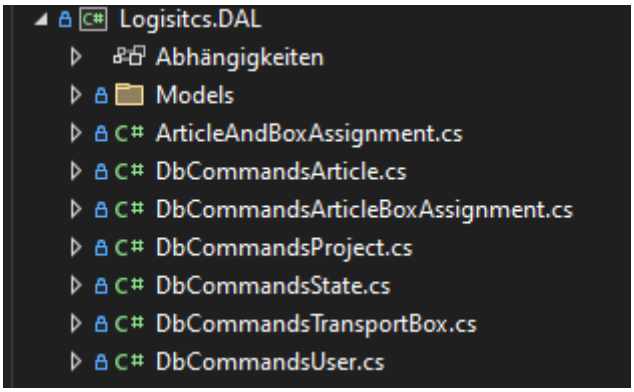
```
//DAL
services.AddTransient<IProjectDataFactory>(provider => new ProjectDataFactory());
services.AddTransient<IProjectFactory>(provider => new ProjectFactory());
services.AddTransient<ITransportBoxDataFactory>(provider => new TransportBoxDataFactory());
services.AddTransient<ITransportboxFactory>(provider => 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<ITransportBo
services.AddTransient<IArticleBll>(provider => new ArticleBll(provider.GetService<IArticleAndBoxAssignme
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)



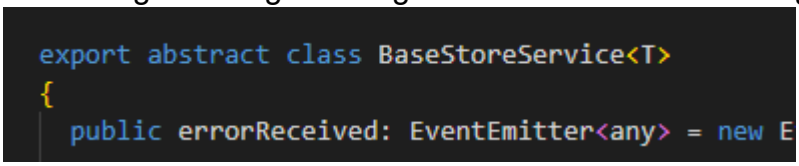


- Modularisierung und Single Responsibility Principle werden durch getrennte Dateien umgesetzt.



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

- Vererbung wird Angularseitig in 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 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

```
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    1 Verweis | Rene Müller, Vor 45 Minuten | 2 Autoren, 9 Änderungen
    public class ArticleController : ControllerBase
    {
        7 Verweise | Artur Sartison, Vor 1 Tag | 1 Autor, 1 Änderung
        protected IArticleBll articleBll { get; }

        0 Verweise | Artur Sartison, Vor 1 Tag | 1 Autor, 2 Änderungen
        public ArticleController(IArticleBll articleBll)
        {
            this.articleBll = articleBll;
        }

        [HttpGet("all/{boxId}")]
        0 Verweise | Artur Sartison, Vor 1 Tag | 2 Autoren, 6 Änderungen
        public async Task<IActionResult> GetAll(string boxId)
        {
```

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 die von Microsoft bereitgestellten „Microsoft.AspNetCore.Authorization;“ Paket umgesetzt. Jeder abgesicherte Controller wird mit dem [Auhorize]-Decorator dekoriert,

damit man nur mit einem gültigen Token rein kommt.
In der Startup wird diese 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.

- Das Observer Pattern wurde mithilfe von Behaviorsubjects 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() : [];
}
```

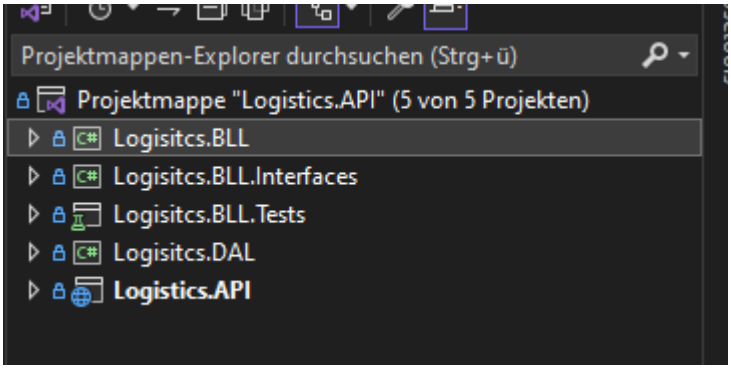
Modularisierung

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

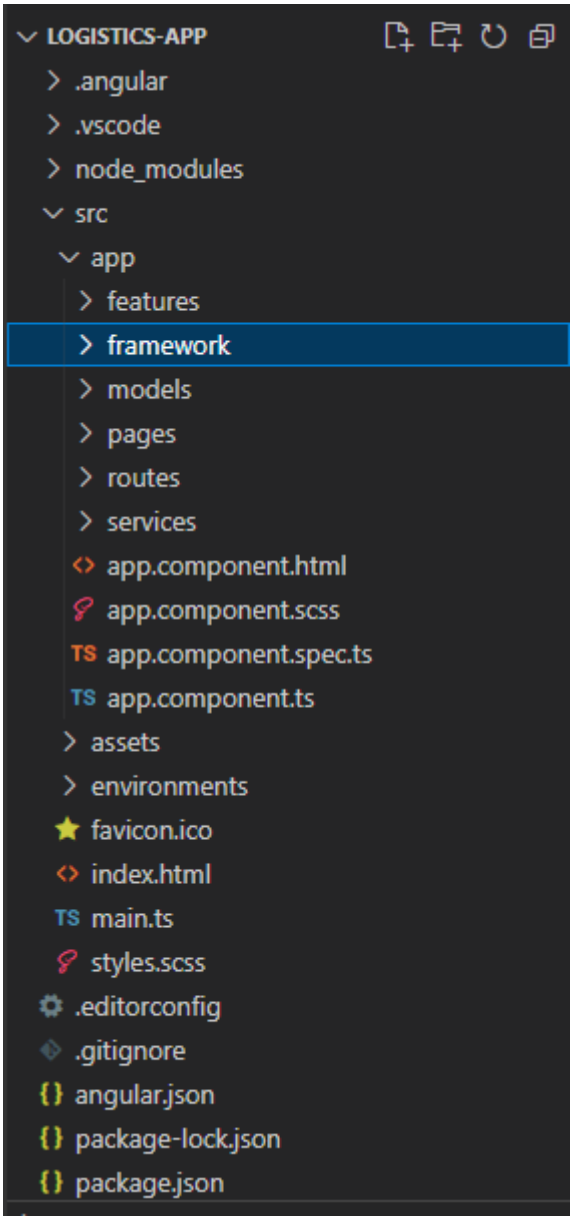
- Als Framework haben wir clientseitig Angular mit dem UI-Framework (Angular-) Material genutzt
- Als CSS-Framework haben wir Tailwind genutzt
- Und serverseitig haben wir ASP.Net Core genutzt

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

UML, Andere Formate (hier im Ordner)