

# Technische Dokumentation

## „Übungskasten“

---

### Inhalt

1	Einleitung.....	2
2	API.....	2
2.1	Presentation Layer .....	2
2.1.1	Einleitung.....	2
2.1.2	Authentifizierung.....	2
2.1.3	Beispiel eines Controllers .....	4
2.2	Application Layer .....	5
2.2.1	Einleitung.....	5
2.2.2	Abstraktion Ordner.....	5
2.2.3	Service Ordner .....	5
2.2.4	UseCases.....	5
2.3	Database Layer.....	6
2.3.1	Einleitung.....	6
2.3.2	Kommunikation nach Außen .....	6
2.3.3	Repository Pattern.....	6
2.3.4	Struktur.....	7
3	UI .....	7
3.1	Einleitung .....	7
3.2	Dependency .....	7
3.3	Services .....	7
3.3.1	Notification.service.....	7
3.3.2	API-Services .....	8
3.3.3	Login & Logout.....	9
3.3.4	Auth- und PasswordGuard .....	9
3.3.5	App Module .....	10
3.3.6	App Routing Module.....	10

# 1 Einleitung

Dies ist die technische Dokumentation für die Plattform „Übungskasten“. „Übungskasten“ ist eine umfassende Anwendung zur Verwaltung und Bereitstellung von Lern- und Übungsinhalten. Diese Dokumentation soll Entwicklern und technischen Anwendern einen detaillierten Überblick über die Architektur, die verschiedenen Schichten der Anwendung und die Implementierung geben. Sie beschreibt die verwendeten Technologien und Frameworks, die Konfigurationen der einzelnen Komponenten sowie die Schnittstellen und Dienste, die zur Interaktion mit der Plattform verwendet werden. Ziel der Dokumentation ist es, den Aufbau und die Funktionsweise der Anwendung verständlich zu machen und die Erweiterung und Wartung zu erleichtern.

## 2 API

### 2.1 Presentation Layer

#### 2.1.1 Einleitung

Die API-Controller im Backend von **Übungskasten** dienen zur Verwaltung und Bereitstellung von verschiedenen Bildungs- und Übungsinhalten. Jeder Controller ist darauf spezialisiert, bestimmte Daten zu handhaben und mit dem MediatR-Mediator zu kommunizieren, um die gewünschten Aktionen auszuführen. Diese Controller bieten Endpunkte für CRUD-Operationen (Erstellen, Lesen, Aktualisieren, Löschen) und andere spezifische Funktionen, die in der Anwendung benötigt werden.

#### 2.1.2 Authentifizierung

Die Authentifizierung wurde mithilfe von Cookies abgewickelt. Bei einem Login werden E-Mail und Passwort an die API gesendet, diese werden dann geprüft, ob es den Lehrer gibt und ob das Passwort stimmt. Wenn dies passt, wird ein Cookie erstellt mit dem Namen „AuthCookie“

```
private async Task CreateCookiesAsync(string Role, string Name)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, Name),
        new Claim(ClaimTypes.Role, Role)
    };

    var claimsIdentity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme);

    var authProperties = new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(60)
    };

    await HttpContext.SignInAsync(
        CookieAuthenticationDefaults.AuthenticationScheme,
        new ClaimsPrincipal(claimsIdentity),
        authProperties);
}
```

Das Cookie hält für eine Stunde, danach läuft er ab.

Die allgemeinen Einstellungen für Cookies wurden in der „Program.cs“ gesetzt.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.Name = "AuthCookie";
        options.LoginPath = "/api/Authentication/Login";
        options.LogoutPath = "/api/Authentication/Logout";
        options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
        options.SlidingExpiration = true;
        options.Cookie.HttpOnly = false;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
        options.Cookie.SameSite = SameSiteMode.None;
    });
```

#### 2.1.2.1 Options Erklärung

- **CookieAuthenticationDefaults.AuthenticationScheme:** Definiert das Standard-Authentifizierungsschema als Cookie-Authentifizierung.
- **options.Cookie.Name:** Legt den Namen des Authentifizierungs-Cookies fest.
- **options.LoginPath:** Gibt den Pfad zur Anmeldeseite an.
- **options.LogoutPath:** Gibt den Pfad zur Abmeldeseite an.
- **options.ExpireTimeSpan:** Setzt die Lebensdauer des Cookies auf 60 Minuten.
- **options.SlidingExpiration:** Ermöglicht eine gleitende Ablaufzeit, verlängert die Lebensdauer des Cookies bei Aktivität.
- **options.Cookie.HttpOnly:** Wenn auf false gesetzt, kann der Cookie auch durch JavaScript zugänglich gemacht werden. (Wird benötigt in der UI)
- **options.Cookie.SecurePolicy:** Erfordert, dass Cookies nur über HTTPS gesendet werden.
- **options.Cookie.SameSite:** Legt fest, dass Cookies in allen Kontextszenarien gesendet werden (None), was die Kompatibilität erhöht, aber möglicherweise die Sicherheit verringert.

#### 2.1.2.2 App Authentication

Ebenfalls muss in der API bzw. im „Program.cs“ File die Einstellung UseAuthentication gesetzt werden.

```
app.UseAuthentication();
```

So wird sichergestellt, dass die API die Cookies verwendet. Ebenfalls haben wir bei dem BaseController – der von jedem Controller vererbt wird – das Attribut „Authorize“ gesetzt, um so sicherzugehen, dass Controller Endpoints nur mit gesetztem Cookie erreicht werden können.

```
[Authorize]
16 references
public abstract class BaseController : Controller
{
```

### 2.1.3 Beispiel eines Controllers

Der QuestionController verwaltet alle Aktionen rund um Fragen. Hier sind die Hauptmethoden:

#### 1. AddQuestion

- **Pfad:** POST /addQuestion
- **Beschreibung:** Fügt eine neue Frage hinzu. Der Inhalt der Frage wird im Request Body als JSON-Objekt übergeben.
- **Parameter:** QuestionDto (im Request Body)
- **Antwort:** HTTP 200 bei Erfolg, HTTP 500 bei Fehler

#### 2. GetPublicQuestions

- **Pfad:** GET /publicQuestions
- **Beschreibung:** Ruft alle öffentlichen Fragen ab.
- **Antwort:** Eine Liste von QuestionDto Objekten.

#### 3. GetFolderQuestions

- **Pfad:** GET /folderQuestions/{folderId}
- **Beschreibung:** Ruft alle Fragen ab, die zu einem bestimmten Ordner gehören.
- **Parameter:** folderId (im Pfad)
- **Antwort:** Eine Liste von Fragen in einem JSON-Objekt, HTTP 500 bei Fehler

```
[HttpPost("addQuestion")]
0 references
public async Task<IActionResult> AddQuestion([FromBody] QuestionDto question)
{
    try
    {
        var validationContext = new ValidationContext(question);
        Validator.ValidateObject(question, validationContext);

        var newQuestion = await _mediator.Send(new CreateQuestion { Question = question });
        await _mediator.Send(new AddQuestionToCreationFolder { Question = newQuestion });
        return Ok();
    }
    catch (ValidationException ex)
    {
        return BadRequest($"Validierungsfehler: {ex.Message}");
    }
    catch (Exception ex)
    {
        return StatusCode(500, $"Ein Problem ist aufgetreten: {ex.Message}");
    }
}
```

## 2.2 Application Layer

### 2.2.1 Einleitung

Die API-Controller im Backend von **exerciseBox** dienen zur Verwaltung und Bereitstellung von verschiedenen Bildungs- und Übungsinhalten. Jeder Controller ist darauf spezialisiert, bestimmte Daten zu handhaben und mit dem MediatR-Mediator und dem SessionCommunicator zu kommunizieren, um die gewünschten Aktionen auszuführen. Diese Controller bieten Endpunkte für CRUD-Operationen (Erstellen, Lesen, Aktualisieren, Löschen) und andere spezifische Funktionen, die in der Anwendung benötigt werden.

### 2.2.2 Abstraction Ordner

Dieser Ordner enthält alle Abhängigkeiten und Dienstregistrierungen.

- Extensions: Enthält Erweiterungsmethoden für die EF-Models. Dadurch wird das Mapping zwischen den Models und den DTOs vereinfacht
- Models: Enthält die Domänenmodelle und DTOs, die in der Anwendung verwendet werden.
- Repositories: Enthält die Repository-Schnittstellen und -Implementierungen.

### 2.2.3 Service Ordner

Dieser Ordner enthält die Dienste, die von der Anwendung verwendet werden, während die Daten für die Controller vorbereitet werden

### 2.2.4 UseCases

Dieser Ordner enthält die Anwendungsfälle, die die Geschäftslogik der Anwendung definieren.

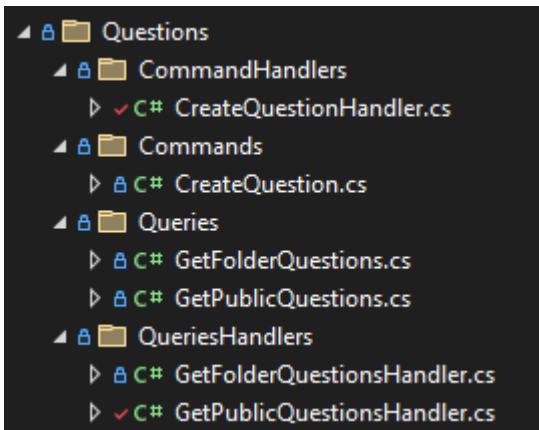
In diesem Sinne wurden Mediator in Kombination mit CQRS-Pattern verwendet

#### 2.2.4.1 Mediator und CQRS-Pattern

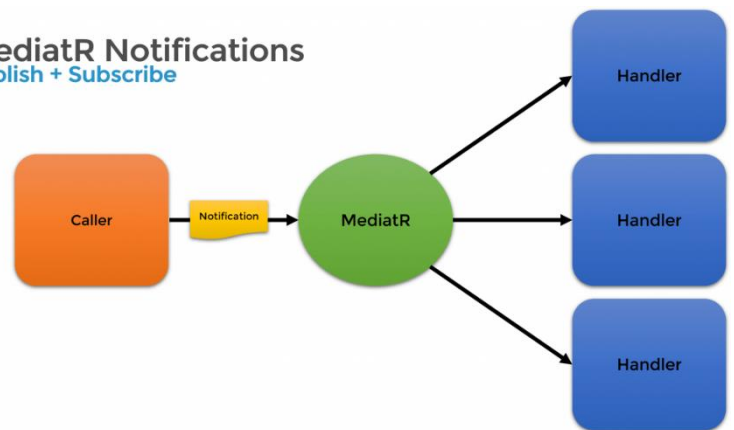
Um eine Schnittstelle zwischen Repositories und den Presentation Layer (z.B. API oder UI) zu haben wurde das Mediator Pattern zusammen mit dem CQRS-Pattern benutzt. Der Mediator dient hierbei als allgemeiner Kontrollpunkt. CQRS (Command Query Responsibility Segregation) trennt die Verantwortungen von Commands und Queries. Queries werden nur benutzt, um Daten zu beschaffen und Commands zum Manipulieren. Mithilfe des Mediators können so Commands und Queries angesteuert werden.

Der Mediator wurde mit dem Nugget Paket MediatR umgesetzt.

[NuGet Gallery | MediatR 12.3.0](#)



### MediatR Notifications Publish + Subscribe



## 2.3 Database Layer

### 2.3.1 Einleitung

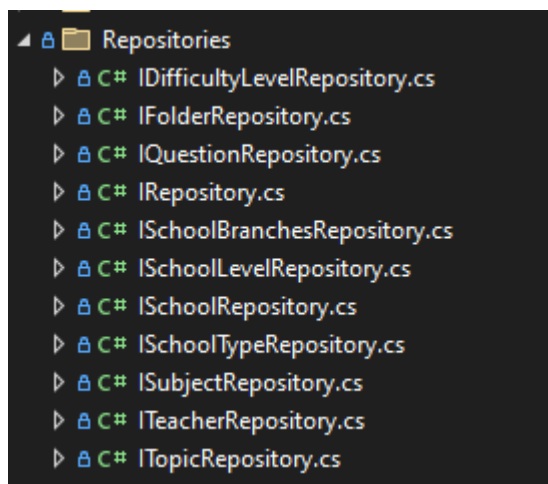
Die Database-Schicht in unserer API ist für den Datenzugriff und die Datenverwaltung zuständig. Diese Schicht verwendet das Repository Pattern, um eine saubere und flexible Struktur für den Datenzugriff zu bieten. Dadurch wird die Geschäftslogik von der Datenzugriffsschicht getrennt, was die Wartung und Erweiterung der Anwendung erleichtert.

### 2.3.2 Kommunikation nach Außen

Das Repository Pattern hilft dabei, eine Abstraktionsebene zwischen der Datenzugriffsschicht und der Geschäftslogik zu schaffen. Jedes Repository ist für eine bestimmte Entität zuständig und bietet spezifische Methoden für CRUD-Operationen (Create, Read, Update, Delete).

### 2.3.3 Repository Pattern

Das Repository Pattern ist ein Design Pattern, das hilft, eine komplexe Segregation aufzubauen für die einzelnen Entities. So wird für jedes Entity ein spezifisches Repository (Interface) aufgebaut, das für die Datenschnittstelle zu Verfügung gestellt wird.



Das "IRepository.cs" Interface dient als Base Repository, wo die CRUD-Funktionen deklariert werden. Diese werden mit zwei generischen Typen versehen, die in den einzelnen Repositories einen Typen zugewiesen bekommen.

Grundsätzlich ist zu sagen, dass das Pattern benutzt wird, um auf unsere Datenbank zuzugreifen. Ebenfalls wird dazu Entity Framework benutzt.

#### 2.3.4 Struktur

- **exerciseBox.Domain**
  - **Entities:** Hier werden die automatisch erstellten EF-Models definiert, die die Datenmodelle der Anwendung darstellen.
- **Database**
  - Diese Ebene ist für die eigentliche Datenbankkonfiguration und -verwaltung zuständig

## 3 UI

### 3.1 Einleitung

Die UI wurde mit dem Typescript Framework Angular17 abgewickelt. ([Angular - Introduction to the Angular docs](#)). Angular ist ein Framework das auf einer Singlepage Application, die selbst erstellen Komponenten aufbaut.

Für die Installation wird Node.js benötigt.

[Node.js — Run JavaScript Everywhere \(nodejs.org\)](#)

### 3.2 Dependency

### 3.3 Services

#### 3.3.1 Notification.service

Der NotificationService ist eine zentrale Klasse zur Anzeige von Toast-Benachrichtigungen in Angular-Anwendungen. Dieser Service ermöglicht es, Fehler- und Erfolgsmeldungen anzuzeigen, die konsistent gestaltet sind und leicht in verschiedenen Teilen der Anwendung verwendet werden können.

##### 3.3.1.1 Schritte zur Verwendung des NotificationService:

#### Import des NotificationService:

Importiere den NotificationService in die Datei, in der du ihn verwenden möchtest. Zum Beispiel in einer Komponente oder einem anderen Service.

```
import { Component } from '@angular/core';
import { NotificationService } from './notification.service';

@Component({
  selector: 'app-meine-komponente',
  templateUrl: './meine-komponente.component.html',
  styleUrls: ['./meine-komponente.component.css']
})
```

```

})
export class MeineKomponenteComponent {

  constructor(private notificationService: NotificationService) {}

  // Hier kannst du Methoden deiner Komponente implementieren, die den NotificationService verwenden
}

```

### Anzeigen einer Erfolgsmeldung:

Verwende die `showSuccess` Methode des `NotificationService`, um eine Erfolgsmeldung anzuzeigen.

```

this.notificationService.showSuccess('Erfolgreich gespeichert!');

```

### Anzeigen einer Fehlermeldung mit automatischem Schließen:

Verwende die `showError` Methode des `NotificationService`, um eine Fehlermeldung anzuzeigen, die nach einer bestimmten Zeit automatisch verschwindet.

```

this.notificationService.showError('Fehler beim Laden der Daten.');
```

### Anzeigen einer persistierenden Fehlermeldung:

Verwende die `showPersistentError` Methode des `NotificationService`, um eine Fehlermeldung anzuzeigen, die dauerhaft sichtbar bleibt, und nicht schließbar ist (z.b. bei fatalem Fehler)

## 3.3.2 API-Services

### 3.3.2.1 Verwendung der API-Services in Angular-Anwendungen

Die API-Services in unserer Angular-Anwendung befinden sich im Ordner `services/api-services` und sind jeweils in eigenen Dateien untergebracht. Diese Services vereinfachen die Kommunikation mit unseren Backend-APIs und kapseln die Logik für HTTP-Anfragen. Dadurch können Komponenten und andere Teile der Anwendung auf einfache Weise API-Aufrufe durchführen, ohne sich um die Details der HTTP-Kommunikation kümmern zu müssen.

### 3.3.2.2 Gemeinsame Merkmale der API-Services

1. **Abstraktion der HTTP-Kommunikation:** Alle API-Services verwenden den `HttpClient` von Angular, um HTTP-Anfragen zu senden und Antworten vom Server zu empfangen. Sie kapseln die Details der API-Endpunkte und bieten einfache Methoden zur Interaktion mit den APIs.
2. **Abhängigkeit von Basis-URL:** Die API-Services injizieren die Basis-URL (`API_BASE_URL`), die für den Aufbau der vollständigen URLs für die API-Endpunkte verwendet wird. Dies ermöglicht eine einfache Konfiguration und Änderung der Basis-URL, ohne den Code der Services ändern zu müssen.
3. **Fehlerbehandlung:** Die Services implementieren grundlegende Fehlerbehandlung, um Fehler während der HTTP-Anfragen abzufangen und zu verarbeiten. Dies hilft, potenzielle Probleme



zu erkennen und angemessen darauf zu reagieren, wie z.B. durch das Anzeigen von Fehlermeldungen.

4. **Observable- und Promise-basiertes Design:** Die meisten Methoden in den API-Services geben Observables oder Promises zurück, was eine asynchrone Verarbeitung von HTTP-Anfragen ermöglicht. Dies passt gut zur reaktiven Programmierung in Angular.

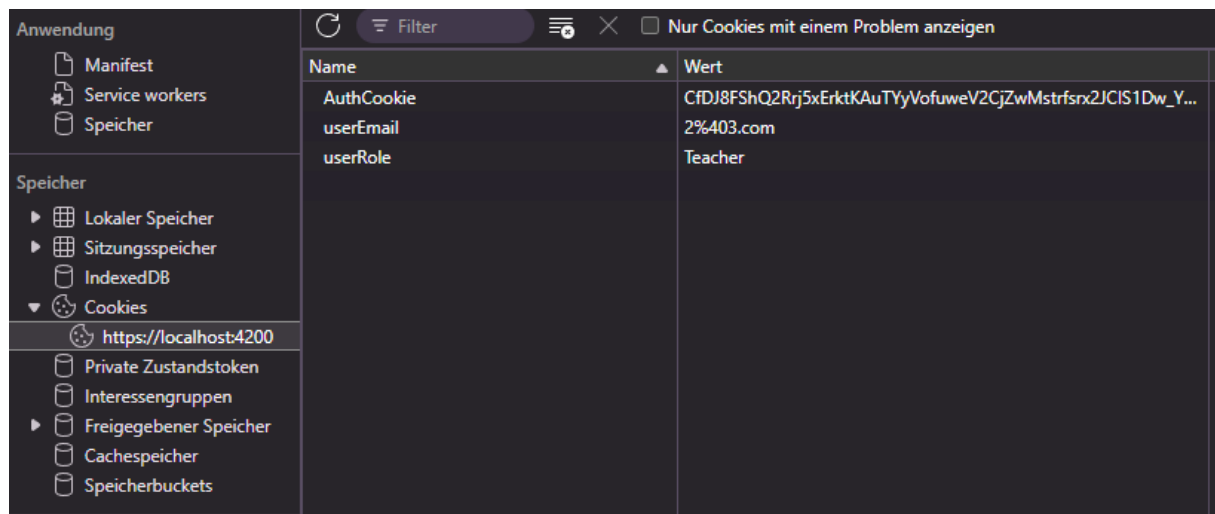
### 3.3.2.3 Beispielhafte Verwendung eines API-Services

Hier ist ein Beispiel, wie ein typischer API-Service verwendet wird:

```
this.difficultyLevelsService.getDifficultyLevels().subscribe(
  (levels) => {
    console.log('Schwierigkeitsgrade geladen:', levels);
  },
  (error) => {
    console.error('Fehler beim Laden der Schwierigkeitsgrade:', error);
  }
);
```

### 3.3.3 Login & Logout

Bei dem Login wird von der API ein Cookie gesetzt, ebenfalls werden von der UI zwei weitere Cookies gesetzt, einmal der userEmail Cookie, der die E-Mail des Users speichert und der userRole Cookie, der die Rolle des Users speichert.



Anwendung	
Manifest	
Service workers	
Speicher	
Speicher	
Lokaler Speicher	
Sitzungsspeicher	
IndexedDB	
Cookies	
https://localhost:4200	
Private Zustandstoken	
Interessengruppen	
Freigegebener Speicher	
Cachespeicher	
Speicherbuckets	

Filter		Nur Cookies mit einem Problem anzeigen	
Name	Wert		
AuthCookie	CfDJ8FShQ2Rrj5xErktKAuTYyVofuweV2CjZwMstrfsnx2JCIS1Dw_Y...		
userEmail	2%403.com		
userRole	Teacher		

### 3.3.4 Auth- und PasswordGuard

Um sicherzustellen das ein User angemeldet ist und welche Pages er mit seiner Rolle erreichen kann. Ebenfalls wird durch dem PasswordGuard sichergestellt das der User nicht mehr sein Standardpassword benutzt.

Die Rolle des User und seine Email wird nach dem Login als Cookie gesetzt. So wird mit der Hilfe des CookieService ([ngx-cookie-service - npm \(npmjs.com\)](https://www.npmjs.com/package/ngx-cookie-service)) auf die existstns des „AuthCookie“ – der von der API gesetzt wird – geschaut, um so sicherzugehen das der User sich eingeloggt hat.

```
public isLoggedIn(): boolean {  
    return this.cookieService.check('AuthCookie');  
}
```

### 3.3.5 App Module

Das AppModule ist das Hauptmodul einer Angular-Anwendung und stellt das zentrale Modul dar, das die verschiedenen Komponenten, Services und anderen Module einer Anwendung zusammenführt. Es definiert die grundlegende Struktur und Konfiguration der Anwendung. Hier sind die wichtigsten Funktionen des AppModule:

- **Deklarationen:** Im declarations-Array werden alle Komponenten, Direktiven und Pipes aufgelistet, die zu diesem Modul gehören. In deinem Fall sind das z.B. AppComponent, NavbarComponent und QuestionsPoolComponent.
- **Importe:** Im Imports-Array werden alle Module importiert, die das AppModule benötigt, um zu funktionieren, wie z.B. BrowserModule, AppRoutingModule und HttpClientModule. Diese importierten Module bieten grundlegende Funktionalität, die in der gesamten Anwendung verwendet werden kann.
- **Providers:** Das Providers-Array listet alle Services und Abhängigkeiten auf, die im gesamten Modul verfügbar sein sollen, wie z.B. TeacherService, AuthenticationService und SessionStorageProvider. Diese Services können dann durch Dependency Injection in verschiedenen Teilen der Anwendung verwendet werden.
- **Bootstrap:** Das Bootstrap Array enthält die Hauptkomponente, die beim Start der Applikation geladen wird. In unserem Fall ist dies AppComponent.

Das AppModule fungiert somit als zentrale Stelle für die Konfiguration und Initialisierung der gesamten Angular-Anwendung.

### 3.3.6 App Routing Module

Das AppRoutingModuleModule definiert die Routen (Pfadkonfigurationen) für die Angular-Anwendung. Es legt fest, welche Komponenten für bestimmte URL-Pfade angezeigt werden und ermöglicht die Navigation innerhalb der Anwendung. Hier sind die wichtigsten Funktionen des AppRoutingModuleModule:

- **Routenkonfiguration:** Das Array routes enthält eine Liste von Routen, die die verschiedenen Pfade der Anwendung und die zugehörigen Komponenten definieren. Jede Route besteht aus einem Pfad und einer Komponente, die angezeigt wird, wenn der Pfad aufgerufen wird. Zusätzliche Konfigurationen wie canActivate und data legen fest, welche Guards angewendet werden und welche Rollen benötigt werden.
- **RouterModule:** Im Import-Array wird das RouterModule mit der Methode forRoot(routes) importiert. Dies konfiguriert den Router mit den definierten Routen und macht ihn in der gesamten Anwendung verfügbar.
- **Export:** Das RouterModule wird auch exportiert, um sicherzustellen, dass es in anderen Modulen der Anwendung verwendet werden kann.
- **Routing-Komponenten:** Die Konstante routingComponents fasst alle Komponenten zusammen, die in den Routen verwendet werden. Diese werden im AppModule deklariert, damit Angular weiß, welche Komponenten zur Anwendung gehören.

```
{ path: "schoolView", component: SchoolViewComponent, canActivate: [AuthGuard], data: { expectedRole: Roles.School }},
```