Angular & .NET Cheatsheet

2025-12-01

Table of Contents

1. Frontend

1.1. Authentication

1.1.1. Interceptor

Beschreibung: Der HTTP-Interceptor ist eine Middleware-Komponente, die automatisch bei **jedem** ausgehenden HTTP-Request ausgeführt wird. Er fügt den JWT-Token aus dem SessionStorage als Bearer-Token in den Authorization-Header ein, sodass authentifizierte API-Anfragen möglich sind.

Wichtige Hinweise: - SessionStorage wird beim Schließen des Browser-Tabs gelöscht - für persistente Anmeldung localStorage verwenden - Der Interceptor muss in `app.config.ts` unter `provideHttpClient(withInterceptors([authInterceptor]))` registriert werden

```
export const authInterceptor: HttpInterceptorFn = (request, next) => {
  const jwt = sessionStorage.getItem('jwt');
  const isLoggedIn = jwt;
  const isApiUrl = request.url.startsWith(environment.apiBaseUrl);
  if (isLoggedIn && isApiUrl) {
    request = request.clone({
      setHeaders: { Authorization: `Bearer ${jwt}` }
    });
  }

  return next(request);
}
```

1.1.2. Auth Guard

Beschreibung: Ein Route Guard, der den Zugriff auf geschützte Routen kontrolliert. Er prüft vor dem Laden einer Route, ob ein gültiger JWT-Token im SessionStorage vorhanden ist. Ist dies nicht der Fall, wird der User automatisch zur Login-Seite umgeleitet.

Wichtige Hinweise: - Muss in den `app.routes.ts` bei den jeweiligen Routen mit `canActivate: [AuthGuard]` aktiviert werden - Der Guard muss in `app.config.ts` mit `provideRouter(routes)` registriert werden

```
import { Router, CanActivate } from '@angular/router';
import {inject, Injectable} from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  private router = inject(Router);

  canActivate(): boolean {
```

```
    const jwt = sessionStorage.getItem('jwt');
    if (!jwt) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }
}
```

1.2. Directives

1.2.1. Filter Directives

Beschreibung: Custom Validators für Template-Driven Forms. Diese Directive validiert Benutzereingaben **synchron** und gibt sofortiges Feedback. Ideal für einfache Validierungen wie Zahlenbereich, Format-Checks oder Pflichtfelder.

Wichtige Hinweise: - Muss im `imports` Array der Komponente registriert werden (Standalone Components) - Bei NgModule: Im `declarations` Array des Moduls registrieren - Die Directive muss das `Validator` Interface implementieren - Rückgabe `null` bedeutet valide, jedes andere Objekt ist ein Fehler - `ValidationErrors` Objekt kann mehrere Fehler enthalten: `{ invalid: true, outOfRange: true }` - Prüfe immer auf leere Werte (`!control.value || control.value === ""`) vor der Validierung - Verwende sprechende Error-Keys: `{ hourOutOfRange: true }` statt `{ invalid: true }` - Zeige Fehlermeldungen mit `@if(hourInput.errors?.['hourOutOfRange']) {<div>⋯</div>}`

```
@Directive({
  selector: '[appValidateHour]'
})
export class ValidateHour implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    if(!control.value || control.value === "") return null;
    const parsed = parseInt(control.value);
    if(isNaN(parsed)) return { notANumber: true};
    if(parsed > 23 || parsed < 0) {
      return { numberOutOfRange: true};
    }
    return null;
  }
}
```

1.2.2. Async Directives

Beschreibung: Async Validators für komplexe Validierungen die eine Backend-Abfrage erfordern, z.B. Prüfung ob Username bereits existiert, ob eine ID gültig ist, oder ob ein Wert in der Datenbank vorhanden ist. Die Validierung läuft asynchron und blockiert das UI nicht.

Wichtige Hinweise: - Muss `AsyncValidator` Interface implementieren (nicht `Validator`!) - Provider-

Konfiguration mit `NG_ASYNC_VALIDATORS` ist zwingend erforderlich - `multi: true` erlaubt mehrere Async Validators auf einem Input - Rückgabe muss `Observable<ValidationErrors | null>` sein - Während der Validierung hat das Control den Status `PENDING`

Best Practice: - Verwende `debounceTime(300)` um Requests zu verzögern bis User fertig getippt hat - Zeige Loading-Spinner während `control.pending === true` - Cache API-Responses um unnötige Requests zu vermeiden - Kombiniere mit synchroner Validierung: erst Format prüfen, dann Backend-Call

```
@Directive({
  selector: '[appRomanCheck]',
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: RomanCheck,
      multi: true
    }
  ]
})
export class RomanCheck implements AsyncValidator {
  private readonly dataService: DataService = inject(DataService);

  validate(control: AbstractControl): Observable<ValidationErrors | null> {
    const value: string = control.value;

    return from(this.dataService.isValid(value)).pipe(
      map((result: boolean) => (result ? null : { invalid: true }))
    );
  }
}
```

1.3. Routing

1.3.1. Routes

Beschreibung: Die zentrale Router-Konfiguration definiert alle verfügbaren Routen der Applikation. Jede Route mappt einen URL-Pfad auf eine Component und kann optional Guards, Resolver oder weitere Konfigurationen enthalten.

Wichtige Hinweise: - Reihenfolge der Routes ist wichtig! Die erste passende Route wird verwendet - `pathMatch: 'full'` bei Redirects verwenden, sonst matched jede URL die mit dem Pfad beginnt - Optionale Wildcard-Route `{ path: '*', component: NotFoundComponent }` als *letzte Route, die als Fallback dient

```
export const routes: Routes = [
    { path: '', pathMatch: 'full', redirectTo: 'login'},
    { path: 'login', component: LoginComponent },
    { path: 'home', component: HomeComponent, canActivate: [AuthGuard]},
```

```
    { path: 'overview', component: InputDeviceOverviewComponent, canActivate:
[AuthGuard]}
];
```

1.3.2. Navigation (Programmatisch)

Beschreibung: Programmatische Navigation ermöglicht das Wechseln zwischen Routes im TypeScript-Code, z.B. nach erfolgreicher Formular-Submission, Login oder bei Button-Clicks. Der Router-Service stellt verschiedene Methoden für Navigation bereit.

Wichtige Hinweise: - `router.navigate(['/path'])` ist relativ zum Root (absoluter Pfad) - `router.navigate(['./path'], {relativeTo: route})` ist relativ zur aktuellen Route - Query Params separat mit Options-Object übergeben

```
this.router.navigate(["/overview"], {
  queryParams: { searchTerm: this.searchTerm() }
});
```

1.3.3. Path Parameter

Beschreibung: Path Parameter (Route Parameters) sind dynamische Segmente in der URL, z.B. `/details/42` wobei `42` die ID ist. Sie werden in der Route-Konfiguration mit `:paramName` definiert und ermöglichen das Laden von spezifischen Ressourcen.

Wichtige Hinweise: - Parameter sind **immer** strings! Konvertierung zu number erforderlich: `parseInt(id)`

```
// In app.routes.ts
export const routes: Routes = [
    { path: 'details/:id', component: InputDeviceDetails, canActivate: [AuthGuard]}
];

// Navigation
this.router.navigate(["/details", deviceId]);

// In Component
const id: string | null = this.activeRoute.snapshot.paramMap.get("id");
const parsed = parseInt(id || "");
```

1.3.4. Query Parameter

Beschreibung: Query Parameter sind optionale Parameter die nach dem `?` in der URL stehen, z.B. `/search?term=angular&page=2`. Sie werden für Filterung, Sortierung, Pagination oder optionale Konfiguration verwendet und überleben Route-Changes.

Wichtige Hinweise: - Query Params sind **immer** strings oder string arrays (`?id=1&id=2`) - `snapshot.paramMap.get("id")` liefert Path Params, nicht Query Params! - Für Query Params:

`snapshot.queryParamMap.get("searchTerm")` verwenden

```
// Navigation
this.router.navigate(["/overview"], {
  queryParams: { searchTerm: this.searchTerm() }
});

// In Component
const id: string | null = this.activeRoute.snapshot.queryParamMap.get("id");
const parsed = parseInt(id || "");
```

1.4. Forms

1.4.1. Reactive Forms

Beschreibung: Reactive Forms (Model-Driven) sind die empfohlene Methode für komplexe Formulare in Angular. Das Form-Model wird im TypeScript-Code definiert und das Template bindet sich daran. Bietet volle Kontrolle, Testbarkeit und Type-Safety.

Wichtige Hinweise: - `ReactiveFormsModule` muss importiert werden - `[formGroup]` Directive auf `<form>` Element binden - Verwende `FormBuilder` Service für sauberere Syntax

```
@Component({
  selector: 'app-login',
  template: `
    <form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
      <label for="username">Username:</label>
      <input type="text" id="username" formControlName="username" />
      <label for="password">Password:</label>
      <input type="password" id="password" formControlName="password" />
      <button type="submit" [disabled]="!loginForm.valid">Login</button>
    </form>
  `,
})
export class LoginComponent {
  loginForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.loginForm = this.fb.group({
      username: ['', Validators.required],
      password: ['', Validators.required],
    });
  }

  onSubmit() {
    if (this.loginForm.valid) {
      const { username, password } = this.loginForm.value;
      // Handle login logic
```

```
    }
  }
}
```

1.4.2. Template Driven Forms

Beschreibung: Template Driven Forms sind einfacher zu implementieren und ähneln AngularJS Formularen. Die Form-Logik liegt hauptsächlich im Template mit Directives. Gut geeignet für einfache Formulare mit wenigen Feldern.

Wichtige Hinweise: - `FormsModule` muss importiert werden - `[(ngModel)]` bindet an Component-Property (Two-Way Binding)

```
@Component({
  selector: 'app-login',
  template: `
    <form (ngSubmit)="onSubmit()" #loginForm="ngForm">
      <label for="username">Username:</label>
      <input
        type="text"
        id="username"
        name="username"
        required
        [(ngModel)]="username"
      />
      <label for="password">Password:</label>
      <input
        type="password"
        id="password"
        name="password"
        required
        [(ngModel)]="password"
      />
      <button type="submit" [disabled]="!loginForm.form.valid">Login</button
    </form>
  `,
})
export class LoginComponent {
  username: string = '';
  password: string = '';

  onSubmit() {
    // Handle login logic
  }
}
```

1.5. Signals

Beschreibung: Signals sind Angulars neues Reaktivitäts-Primitiv (ab v16). Sie ermöglichen Fine-Grained Reactivity, Change Detection Optimierung und bessere Performance. Ein Signal ist ein Wrapper um einen Wert der automatisch Dependencies trackt.

Wichtige Hinweise: - `signal()` erstellt writable Signal, `computed()` für abgeleitete Werte - Signals müssen mit `()` aufgerufen werden um den Wert zu lesen: `count()` - Zum Schreiben: `count.set(5)` oder `count.update(v => v + 1)`

1.5.1. Models

Beschreibung: Two-Way Binding von Signals in Standalone Components. Das `model<T>()` Decorator erstellt ein Signal-Property das automatisch mit dem Template synchronisiert wird. Änderungen im Template aktualisieren das Signal und umgekehrt.

Wichtige Hinweise: - `model<T>(defaultValue)` initialisiert das Signal mit einem Standardwert - Im Template mit `[(ngModel)]="propertyName"` binden - Änderungen im Template aktualisieren automatisch das Signal

```
// Child Component
@Component({
  selector: 'app-device-form',
  template: `
    <form>
      <label for="name">Device Name:</label>
      <input id="name" type="text" [(ngModel)]="deviceName" />

      <label for="description">Description:</label>
      <input id="description" type="text" [(ngModel)]="deviceDescription" />
    </form>
  `,
})
export class DeviceFormComponent {
  deviceName = model<string>('');
  deviceDescription = model<string>('');
}

// Parent Component
@Component({
  template: `
    <app-device-form
      [(deviceName)]="name"
      [(deviceDescription)]="description"
    />
  `
})
export class ParentComponent {
  name = signal<string>('Initial Device');
```

```
  description = signal<string>('Initial Description');
}
```

1.5.2. Input

Beschreibung: Signal-basierte Inputs ersetzen das traditionelle `@Input()` Decorator und bieten Type-Safety, automatische Change Detection und bessere Integration mit dem Signal-System. Parent Components können reaktiv Daten an Child Components übergeben.

Wichtige Hinweise: - `input.required<T>()` wirft Compile-Error wenn Property nicht gesetzt wird - `input<T>(defaultValue)` für optionale Inputs mit Fallback-Wert - Signal Inputs sind **readonly** - Child kann Wert nicht direkt ändern

```
@Component({
  selector: 'app-device-card',
})
export class DeviceCardComponent {
  device = input.required<InputDevice>();
  isRequired = input<boolean>(false);
  title = input<string>('Default Title');
}

// Parent Component
@Component({
  template: `
    <app-device-card
      [device]="selectedDevice()"
      [isRequired]="true"
      [title]="'Device Details'"
    />
  `
})
export class ParentComponent {
  selectedDevice = signal<InputDevice>({ id: 1, name: 'Device 1' });
}
```

1.5.3. Output

Beschreibung: Signal-basierte Outputs ersetzen `@Output()` EventEmitter und ermöglichen typsichere Event-Kommunikation von Child zu Parent Component. Events werden mit `output<T>()` definiert und können mit oder ohne Daten emittet werden.

Wichtige Hinweise: - `output<T>()` erstellt einen Event-Emitter, **kein** Signal! - Events werden mit `.emit(value)` ausgelöst (wie bei EventEmitter) - Parent muss Event im Template mit `(eventName)="handler($event)"` abonnieren

```
@Component({
  selector: 'app-device-form',
```

```
  template: `
    <button (click)="handleSave()">Save</button>
    <button (click)="handleCancel()">Cancel</button>
  `
})
export class DeviceFormComponent {
  deviceSaved = output<InputDevice>();
  cancelled = output<void>();

  handleSave() {
    const device: InputDevice = { id: 1, name: 'Device 1' };
    this.deviceSaved.emit(device);
  }

  handleCancel() {
    this.cancelled.emit();
  }
}

// Parent Component
@Component({
  template: `
    <app-device-form
      (deviceSaved)="onDeviceSaved($event)"
      (cancelled)="onCancelled()"
    />
  `
})
export class ParentComponent {
  onDeviceSaved(device: InputDevice) {
    console.log('Device saved:', device);
  }

  onCancelled() {
    console.log('Form cancelled');
  }
}
```

1.6. HTTP Client

Beschreibung: Der HttpClient-Service ist Angulars zentrale API für HTTP-Kommunikation. Er bietet eine Observable-basierte API für REST-Calls, automatische JSON-Parsing, Request/Response-Interceptors und Type-Safety durch Generics.

Wichtige Hinweise: - `HttpClientModule` muss in `app.config.ts` mit `provideHttpClient()` registriert werden - Observables sind lazy - ohne `.subscribe()` wird kein Request gesendet - Error Handling mit `.pipe(catchError())` implementieren

```
import { Injectable } from '@angular/core';
```

```
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { Observable, observeOn } from 'rxjs';
import { Game } from '../models/game.model';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json'
  })
}
@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = "https://h-aitenbichler.cloud.htl-leonding.ac.at/restserver";

  public readonly header = new HttpHeaders({
    'Content-Type': 'application/json'
  });

  constructor(private http: HttpClient) { }

  // Optional - Hilfsmethode für Arrays als Query Parameter
  private toQueryParamArray(content: string[]): string {
    return "?sudoku=" + content.join("&sudoku=");
  }
}
```

1.6.1. GET

```
getGames(): Observable<Game[]> {
  return this.http.get<Game[]>(`${this.apiUrl}/game`);
}
```

```
getGame(id: number): Observable<Game> {
  return this.http.get<Game>(`${this.apiUrl}/game/${id}`);
}
```

1.6.2. POST

```
postGame(game: Game): Observable<Game> {
  return this.http.post<Game>(`${this.apiUrl}/game`, game);
}
```

1.6.3. PUT

```
putGame(id: number, game: Game) {
  return this.http.put(`${this.apiUrl}/game/${id}`, game,
    {
      headers: this.header,
      observe: 'response'
    });
}
```

1.6.4. PATCH

```
patchGame(id: number, partialGame: Partial<Game>) {
  return this.http.patch(`${this.apiUrl}/game/${id}`, partialGame,
    {
      headers: this.header,
      observe: 'response'
    });
}
```

1.6.5. DELETE

```
deleteGame(id: number) {
  return this.http.delete(`${this.apiUrl}/game/${id}`, {
    headers: this.header,
    observe: 'response'
  });
}
```

1.7. Templates

1.7.1. Table

```
<table>
      <thead>
        <tr>
          <th>Device</th>
          <th>Description</th>
          <th>Reservation</th>
          <th></th>
        </tr>
      </thead>
    @for (device of filteredDevices; track device) {
      <tbody>
        <tr class="border-bottom">
```

```
          <td>{{ device.name }}</td>
          <td>{{ device.description }}</td>
          <td>{{ formatReservationDates(device.reservations) }}</td>
          <td>
            <button class="btn btn-secondary" (click)="detailInputDevice(device.id)">
              Reserve
          </button>
        </td>
        </tr>
      </tbody>
    }
</table>
```

1.7.2. Input Form

1.7.3. Details View

2. Backend

2.1. CDI (Dependency Injection)

Beschreibung: Dependency Injection Container registriert Services die dann automatisch in Controllers/Endpoints injiziert werden können. Ermöglicht loose coupling, bessere Testbarkeit und zentrale Konfiguration.

Wichtige Hinweise: - Services müssen **vor** `var app = builder.Build()` registriert werden - `AddTransient`: neue Instanz bei jeder Injection (stateless Services) - `AddScoped`: eine Instanz pro HTTP-Request (z.B. DbContext) - `AddSingleton`: eine Instanz für gesamte App-Lifetime (Caching, Configuration) - Interface-Registration (`<IService, Service>`) erlaubt Mocking in Tests

Füge Service in Backend hinzu:

```
builder.Services.AddTransient<IMessageService, MessageService>();
```

```
builder.Services.AddTransient<MessageService>();
```

2.2. Swagger

2.2.1. Swagger Builder

Beschreibung: Swagger/OpenAPI UI generiert automatisch interaktive API-Dokumentation. Endpoints können direkt getestet werden, Request/Response Schemas werden visualisiert. Essential für Frontend-Entwickler und API-Testing.

Wichtige Hinweise: - `AddEndpointsApiExplorer()` muss **vor** `AddSwaggerGen()` stehen

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddSingleton<IRomanNumerals, RomanNumerals>();

// Add cors here if needed
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAngular",
        policy => policy
            .WithOrigins("http://localhost:4200")
            .AllowAnyHeader()
            .AllowAnyMethod());
});
```

```
var app = builder.Build();

// Add cors here if needed
app.UseCors("AllowAngular");

app.UseSwagger();
app.UseSwaggerUI();
```

2.2.2. Cors

Beschreibung: CORS (Cross-Origin Resource Sharing) erlaubt Browser Requests von anderen Domains (z.B. Frontend auf localhost:4200 zu API auf localhost:5000). Ohne CORS blockiert Browser die Requests aus Sicherheitsgründen.

Wichtige Hinweise: - CORS-Policy muss **vor** `app.Build()` registriert werden - `UseCors()` muss **vor** `UseAuthorization()` stehen (Middleware-Reihenfolge!)

```
// Reihenfolge siehe oben
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAngular",
        policy => policy
            .WithOrigins("http://localhost:4200")
            .AllowAnyHeader()
            .AllowAnyMethod());
});

app.UseCors("AllowAngular");
```

2.3. Minimal API

2.3.1. GET

```
app.MapGet("/isValid", (string literal, IRomanNumerals service) =>
{
    try
    {
        service.ConvertFromRomanLiteral(literal);
        return true;
    }
    catch (Exception ex) when (ex is ArgumentOutOfRangeException or ArgumentException)
    {
        return false;
    }
});
```

2.3.2. POST

```
app.MapPost("/devices", (InputDevice device, IDeviceService service) =>
{
    try
    {
        var created = service.CreateDevice(device);
        return Results.Created($"/devices/{created.Id}", created);
    }
    catch (Exception ex)
    {
        return Results.BadRequest(ex.Message);
    }
});
```

2.3.3. PUT

```
app.MapPut("/devices/{id}", (int id, InputDevice device, IDeviceService service) =>
{
    try
    {
        var updated = service.UpdateDevice(id, device);
        return updated != null ? Results.Ok(updated) : Results.NotFound();
    }
    catch (Exception ex)
    {
        return Results.BadRequest(ex.Message);
    }
});
```

2.3.4. PATCH

```
app.MapPatch("/devices/{id}/status", (int id, string status, IDeviceService service)
=>
{
    try
    {
        var updated = service.UpdateDeviceStatus(id, status);
        return updated ? Results.NoContent() : Results.NotFound();
    }
    catch (Exception ex)
    {
        return Results.BadRequest(ex.Message);
    }
});
```

2.3.5. DELETE

```
app.MapDelete("/devices/{id}", (int id, IDeviceService service) =>
{
    try
    {
        var deleted = service.DeleteDevice(id);
        return deleted ? Results.NoContent() : Results.NotFound();
    }
    catch (Exception ex)
    {
        return Results.BadRequest(ex.Message);
    }
});
```

2.4. Authentication

Beschreibung: JWT (JSON Web Token) Authentication für stateless API-Authentifizierung. Token enthält Claims (User-ID, Roles) und wird bei jedem Request im Authorization-Header gesendet.

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

2.5. Services

Beschreibung: Services kapseln Business-Logic und werden via Dependency Injection in Endpoints/Controller injiziert. Ermöglicht Separation of Concerns, bessere Testbarkeit und Code-Reuse.

Wichtige Hinweise: - Services müssen in `Program.cs` registriert werden (siehe 2.1)

2.5.1. Service Interface

```
namespace YourProject.Services;

public interface IYourService
{
    // Definiere deine Service-Methoden hier
}
```

2.5.2. Service Implementation

```
namespace YourProject.Services;

public class YourService : IYourService
{
    // Implementiere deine Service-Methoden hier
}
```

2.5.3. Service Registration

```
builder.Services.AddTransient<IYourService, YourService>();
// oder
builder.Services.AddTransient<YourService>();
```