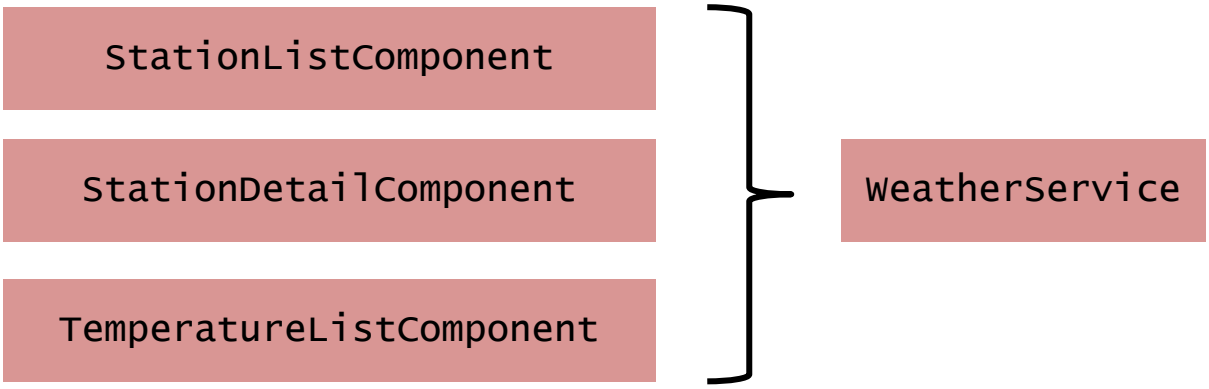
Angular: Services, Routing und Reaktive Programmierung

- **Was ist ein Service und wieso Services?**
  - Wie wird Klasse zum Service?
  - Wie wird Service in der gesamten Anwendung bekannt gemacht?
  - Wie wird durch *Constructor Injection* Service einer Komponente zur Verfügung gestellt?
  - Über einen Service der Anwendung Konfigurationswerte bereitstellen können
- **Was bedeutet Routing und wieso Routing?**
  - Wie werden Probleme beim Verlinken der Seiten gelöst?
  - Wie wird das Routing-Modul erstellt?
  - Wie wird dieses der Anwendung zur Verfügung gestellt?
  - Wie werden Routen verlinkt und Paramater an Routen übergeben?
  - Wie werden Parameter in Komponenten ausgelesen?
  - Wie werden Routen programmtechnisch gewechselt?
- **Was ist reaktive Programmierung und wieso?**
  - Was kann ein `Observable`?
  - Publisher und Observer an Beispielen erklärt

Was ist ein Service und wieso Services?

ist eine Funktion oder Klasse die für andere Funktionen oder Klassen Funktionalitäten bereitstellen



MÖGLICHKEIT: Objekt `WeatherService` dort instanziieren wo man es benötigt

PROBLEM: Anpassungen am Konstruktor führen zu weitreichenden Abhängigkeiten in gesamter Anwendung

LÖSUNG: Verantwortung für die Erzeugung von Abhängigkeiten wird an übergeordnete Stelle delegiert. Prinzip *Inversion of Control*

Trennung zw. Darstellungs- und Anwendungslogik

Obiges Prinzip wird in Angular durch *Dependency bzw. Constructor Injection* realisiert

```
@Injectable()
export class WeatherService {
  …
}
```

Dadurch wird normale Klasse zum Service, und es kann mit dieser Klasse *Constructor Injection* betrieben werden

```
import { HttpClientModule } from '@angular/common/http';
import { WeatherService } from './shared/weather-service';
@NgModule({
  imports: [HttpClientModule, … ],
  providers: [ WeatherService* ]
})
export class AppModule { }
```

Dadurch wird Service der gesamten Anwendung bekannt gemacht. Angular erzeugt von diesem eine Instanz

HINWEIS

```
@Injectable({ providedIn: 'root' })
```

sorgt dafür, dass ohne Angabe in `providers` (*) Service allen Komponenten des `AppModule` bereitgestellt wird

```
@Component( … )
export class StationListComponent implements OnInit {
  stations!: Array<StationValley>;
  constructor(private ws: WeatherService) { }
  ngOnInit() {
    this.ws.getAll().subscribe(res => this.stations = res);
  }
}
```

Dadurch wird über private Eigenschaft der Service der Komponente zur Verfügung gestellt, welche ihrerseits die Funktionalitäten des Services nutzen kann

Alle Klassen die Decorator haben können auf diese Weise Service anfordern

Konfigurationskonstanten über Service bereitstellen

```
providers: [
  { provide: 'config', useValue: { showErrors: true, url: '…' } }
]
```

Verwendung:

```
constructor(@Inject('config') private conf: any) {
  console.log(`${conf.showErrors} ${conf.url}`);
}
```

Was bedeutet Routing und wieso Routing?

Laden von Bereichen bzw. Komponenten der Anwendung abhängig vom Zustand

PROBLEME

- Angular stellt *Single-Page-Applikationen* bereit
- Wenn über Link andere Inhalte angezeigt werden sollen, darf Seite nicht neu vom Server geladen sondern lediglich Komponenten ausgetauscht werden
- Vor-/Zurück-Knopf im Browser muss mit Navigation synchronisiert werden
- Alle Ansichten sollen über URLS aufrufbar sein, denn Suchmaschinen sollen über Links Seiten erreichen können
- Browser-Verlauf und -Lesezeichen sollen korrekt gesetzt werden

LÖSUNG

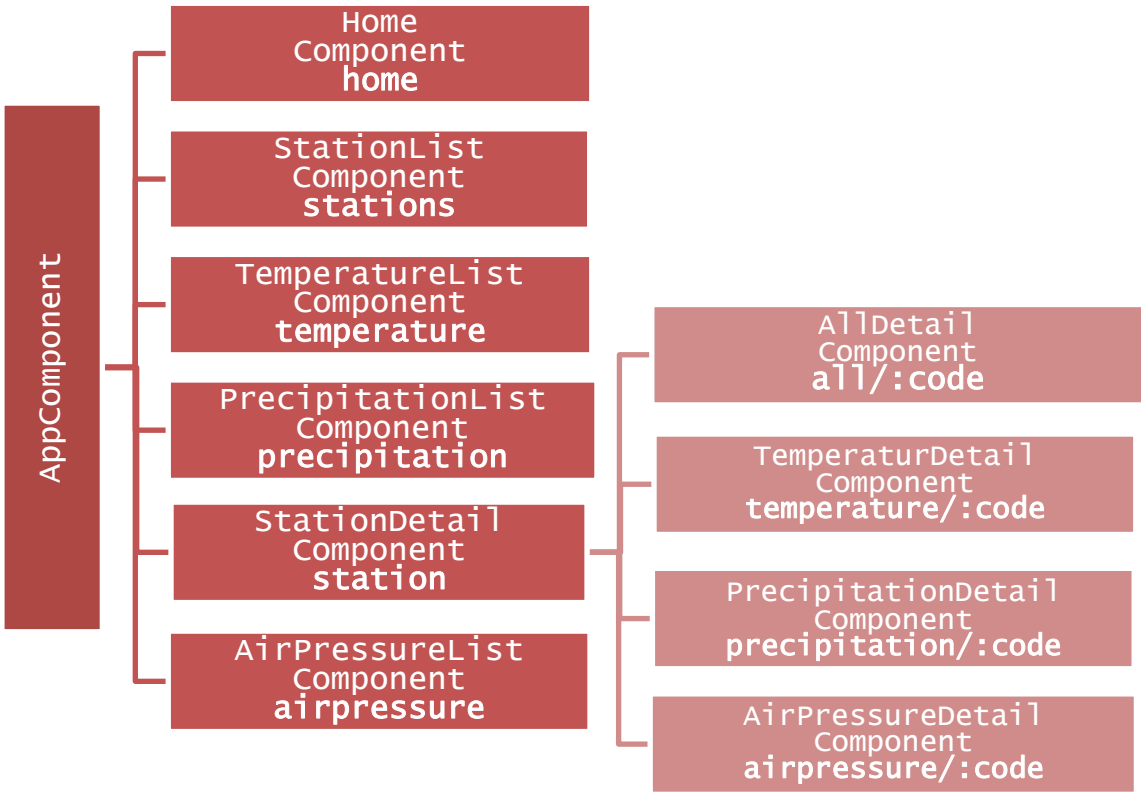
Angular-Router nutzt Möglichkeiten der *HTML5 History API* welche in allen modernen Browsern implementiert ist und ändert Browser-Verlauf ohne Seite neu zu laden

1. Schritt: Routing-Modul erstellen

Messwerte

Home	Namen	Temperatur	Niederschlag	Luftdruck
Antholz Obertal		27.2 °C	0 mm	1009.3 hPa
Auer		38.7 °C	0 mm	1009.1 hPa
Barbian Kollmann		35.5 °C	0 mm	1009 hPa
Bozen		38.6 °C	0 mm	1010.3 hPa

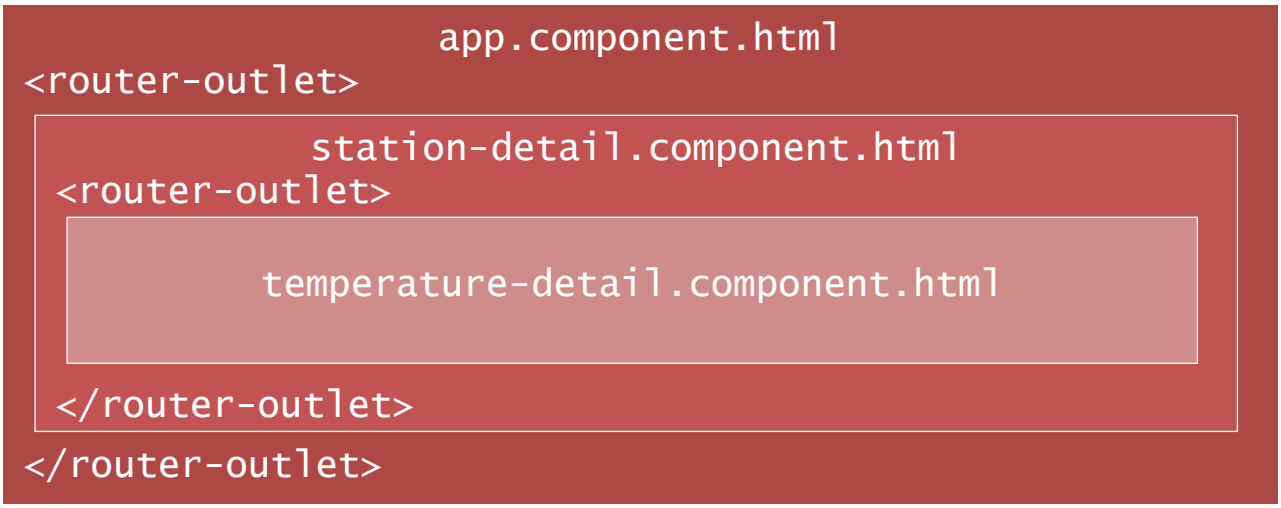
<router-outlet>




```
home
stations
temperature
precipitation
airpressure
station/all/:code
station/temperature/:code
station/precipitation/:code
station/airpressure/:code
```

/ (?) station (?)

Komponenten können verschachtelt werden:



<router-outlet> bestimmt die Stelle im Template an der Komponente geladen werden soll (siehe hinten)

Datei app-routing.module.ts

```
import { Routes, RouterModule } from '@angular/router';
…
const routes: Routes = [
  { path: '', component: HomeComponent, pathMatch: 'full'[1] },
  { path: 'home', component: HomeComponent },
  { path: 'stations', component: StationListComponent },
  { path: 'temperature', component: TemperatureListComponent },
  { path: 'precipitation', component: PrecipitationListComponent},
  { path: 'airpressure', component: AirPressureListComponent },
  { path: 'station', component: StationDetailComponent,
    children: [
      { path: 'all/:code',
        component: AllDetailComponent },
      { path: 'temperature/:code',
        component: TemperatureDetailComponent },
      { path: 'precipitation/:code',
        component: PrecipitationDetailComponent },
      { path: 'airpressure/:code',
        component: AirPressureDetailComponent },
      { path: '', redirectTo: '/stations', pathMatch: 'full' }
    ]
  }
];
@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ],
})
export class AppRoutingModule { }
```

`RouterModule` wird importiert. `forRoot()` wird aufgerufen und erstellt ein Modul mit den obigen Routen das in das `AppRoutingModule` importiert und wieder nach außen exportiert wird

[1] `full` bedeutet, dass exakt keine Route (`''`) angegeben werden darf, damit zur `HomeComponent` gesprungen wird. Sonst würde jede Route die nicht in `routes` angegeben ist, zur `HomeComponent` springen

2. Schritt: Routing-Modul zur Verfügung stellen

```
@NgModule({
  imports: [ AppRoutingModule, … ],
  …
})
export class AppModule { }
```

3. Schritt: Routen verlinken über Angular Material `mat-tab-nav-bar`

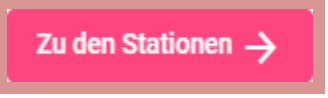
Datei `app.component.html`

```
<nav mat-tab-nav-bar [tabPanel]="tabPanel"
  backgroundColor="primary" color="accent">
  <a mat-tab-link
    routerLink="/home"
    routerLinkActive #homeTab="routerLinkActive"
    [active]="homeTab.isActive">
    Home
  </a>
  <a mat-tab-link
    routerLink="/stations"
    routerLinkActive #stationsTab="routerLinkActive"
    [active]="stationsTab.isActive">
    Stationen
  </a>
  …
</nav>

<mat-tab-nav-panel #tabPanel>
  <router-outlet></router-outlet>
</mat-tab-nav-panel>
```

Mit `routerLinkActive` wird ausgewählter Link hervorgehoben

```
<a mat-raised-button color="accent"
  routerLink="/stations">
  Zu den Stationen
  <mat-icon>arrow_forward</mat-icon>
</a>
```



- Anstelle von `href` muss `routerLink` verwendet werden, denn `href` würde gesamte Seite vom Server (!) neu laden, `routerLink` lädt nur Komponenten
- Navigation erfolgt immer relativ zur aktuellen URL. Befände man sich z.B. in `home` so wäre auch `routerLink='../stations'` korrekt

Parameterübergabe in Route

```
<a routerLink="/station/all/{{ station.code }}">
  {{ station.name }}
</a>
```

Auslesen des Parameters in Komponente

```
import { ActivatedRoute } from '@angular/router';

export class AllDetailComponent implements OnInit {

  measurements!: Array<Measurement>;

  constructor(
    private route: ActivatedRoute,
    private ws: WeatherService) { }

  ngOnInit() {
    this.ws
      .getAllMeasurements(this.route.snapshot.params['code'])
      .subscribe(measurements=> this.measurements = measurements);
  }
}
```

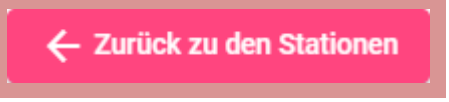
Zugriff von der übergeordneten Komponente `StationDetailComponent` aus:

```
this.route.snapshot.firstChild?.params['code'];
```

Mit `this.route.params.subscribe()` wird ein `Observable` auf die Parameter definiert, welches über Änderungen an den Parametern informiert und Komponente nicht neu lädt sondern dynamisch anpasst (siehe hinten)

Route programmtechnisch wechseln

```
<button mat-raised-button color="accent"
  (click)="backToStations() bzw. back()">
  <mat-icon>arrow_back</mat-icon>
  Zurück zu den Stationen
</button>
```




```
import { ActivatedRoute, Router } from '@angular/router';
import { Location } from '@angular/common';
export class StationDetailComponent implements OnInit {
  constructor(
    private route: ActivatedRoute,
    private router: Router,
    private location: Location,
  ) { }
  backToStations() {
    this.router.navigate(['/stations']);
  }
  back() {
    this.location.back();
  }
}
```

- [] damit bei Bedarf Parameter übergeben werden können
- Location muss von @angular/common importiert werden

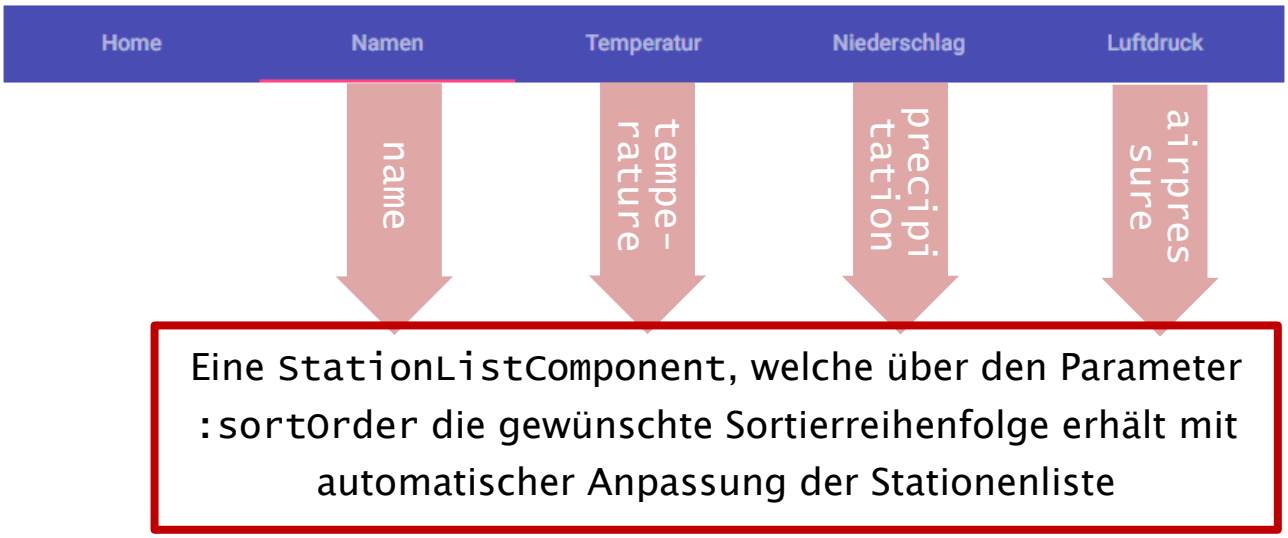
Was ist reaktive Programmierung und wieso?

Damit können Datenflüsse und deren Veränderungen deklarativ modelliert und asynchrone und eventbasierte Aufgaben stark vereinfacht werden

ReactiveX oder *Reactive Extensions* oder kurz *RxJS* ist die Implementierung für JavaScript

`Observer`	Beobachter Entwurfsmuster: Subjekt (*publisher*) bietet Mechanismus für Beobachter (*observer*, *subscriber*) an, der über Änderungen informiert wird. Es wird *Abonnement* auf den Publisher gemacht
+	
`Iterator`	Bietet Methoden an, um Daten aus einer Datenstruktur zu lesen ohne dass interne Datenstruktur offengelegt wird
=	
`Observable`	Sind neue Daten vorhanden, dann werden diese ohne Aufruf von `next()` automatisch „gepusht“

Beispiel: Abfrage der Parameter über Observable




```
export class StationListComponent implements OnInit {

  stations!: StationValley[];
  sortOrder!: string;

  constructor(
    private ws: WeatherService,
    private route: ActivatedRouter) { }

  ngOnInit() {
    this.route.params.subscribe(params => {
      this.sortOrder = params.sortOrder;

      this.ws.getAll(params.sortOrder)
        .subscribe(stations => this.stations = stations);
    });
  }

  …
}
```

ACHTUNG: Liste von Stationen wird nur dann in der View aktualisiert, wenn sich Listenobjekt (`stations`) ändert. Allein eine Änderung der Inhalte des Listenobjektes führen noch nicht zur Aktualisierung der Sicht

Beispiel: Observer horcht auf Komponenteneingaben

Publisher

search-station.component.html

```
<input type="text"
  #searchTerm
  (keyup)="handleKeyEvent(searchTerm.value)">
```

search-station.component.ts

```
export class SearchStationComponent {
  @Output() searchTermEmitted =
    new EventEmitter<string>();
  handleKeyEvent(searchTerm: string) {
    this.searchTermEmitted.emit(searchTerm);
  }
  …
}
```

Observer

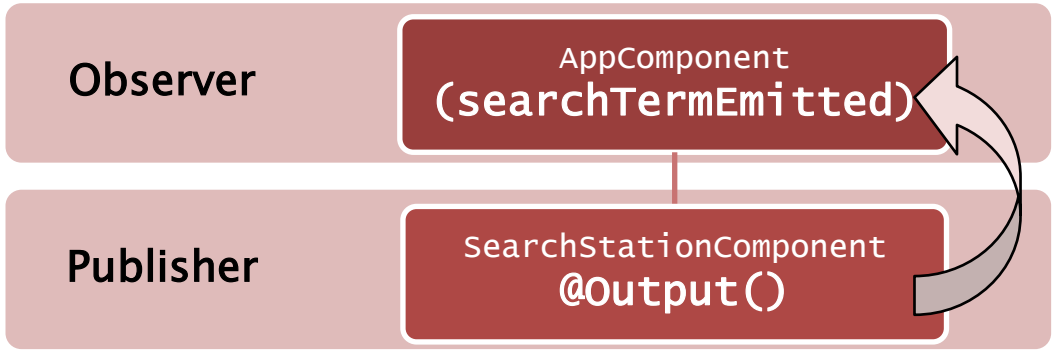
app.component.html

```
<tb-search-station
  (searchTermEmitted)=
    "handleEmittedSearchTerm($event)">
</tb-search-station>
{{ searchTerm }}
```

app.component.ts

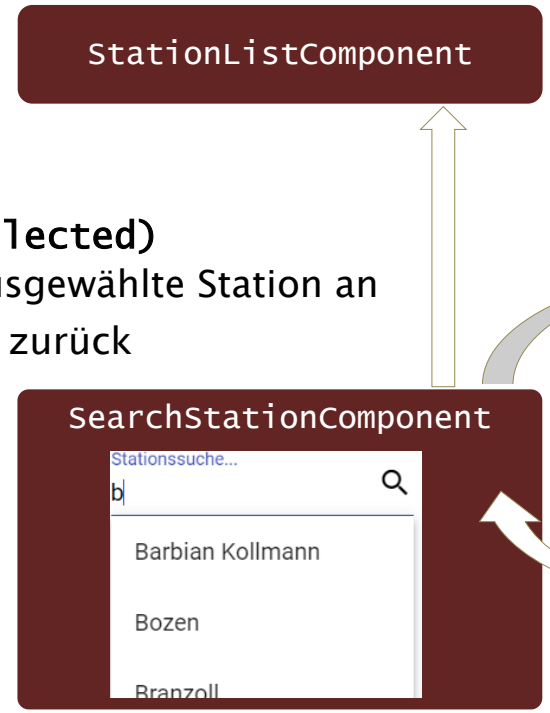
```
export class AppComponent {
  searchTerm!: string;
  handleEmittedSearchTerm(text: string) {
    this.searchTerm = text;
  }
  …
}
```

Tastendr
Tastendr



Bei jedem Tastendruck informiert *Publisher* den *Observer* und teilt ihm Änderung mit

Erweiterte Stationssuche



`(stationSelected)`

- Liefert ausgewählte Station an Observer zurück

- Abonniert `WeatherService` zum Abfragen der Wetterstationen

- Liefert alle Stationen die `searchTerm` entsprechen

`searchTermEmitted`

- Beim Initialisieren der Komponente abonniert (*subscribe*)
- Bei „fast“ jedem Tastendruck
- Abonniert `WeatherService` zum Abfragen der Stationen

Zeigt gefundene Stationen an und reagiert auf Ereignis „Station ausgewählt“

```
export class SearchStationComponent implements
  OnInit, OnDestroy {

  foundStations!: StationValley[];

  @Output() stationSelected =
    new EventEmitter<StationValley>();

  private searchTermEmitted =
    new EventEmitter<string>();

  constructor(private ws: WeatherService) { }

  ngOnInit() {
    this.searchTermEmitted
      .pipe(
        debounceTime(500),
        distinctUntilChanged(),
        switchMap(searchTerm =>
          this.ws.getStations(searchTerm)))
      .subscribe(foundStations =>
        this.foundStations = foundStations);
  }

  ngOnDestroy() {
    this.searchTermEmitted.unsubscribe();
  }

  handleKeyEvent(searchTerm: string) {
    this.searchTermEmitted.emit(searchTerm);
  }

  handleSelectedStation(station: StationValley) {
    this.stationSelected.emit(station);
  }
}
```

```
<mat-form-field>
  <input type="text"
    placeholder="Stationssuche... "
    #searchTerm
    (keyup)="handleKeyEvent(searchTerm.value)"
    matInput [matAutocomplete]="auto">
  <mat-icon matSuffix>search</mat-icon>
</mat-form-field>

<mat-autocomplete #auto="matAutocomplete"
  (optionSelected)=
    "handleSelectedStation($event.option.value)">
  <mat-option *ngFor="let s of foundStations"
    [value]="s">{{ s.name }}
  </mat-option>
</mat-autocomplete>
```

`getStations(searchTerm)` liefert `Observable <Array<StationValley>>` zurück

`debounceTime()`, `distinctUntilChanged()`, `switchMap()` müssen von `rxjs/operators` importiert werden

`toString()` von `StationValley` muss Stationsname zurückliefern, damit Name der ausgewählten Station im Eingabefeld erscheint

Abonnement muss beim Zerstören der Komponente wieder aufgehoben werden (`OnDestroy`)

`handleKeyEvent()` benachrichtigt bei jedem Tastendruck Observer. `SearchStationComponent` ist gleichzeitig sein eigener Observer des Tastendrucks. Abonnieren erfolgt in `ngOnInit()`

Mithilfe von `pipe()` kann ein Observable-Stream manipuliert werden

Nicht bei jedem Tastendruck (`debounceTime()`) und nur bei geänderten Eingaben (`distinctUntilChanged()`) wird vom Wetterservice ein Observable angefordert, das alle Stationen sucht die dem `searchTerm` entsprechen

`switchMap()` abonniert innerhalb des äußeren Observables das innere Observable vom Wetterservice. Ergebnis ist neues Observable welches bei jedem Stimulus des äußeren Observable eine Liste von Stationen emittiert. Funktion liefert nur aktuellstes Ergebnis weiter, ältere Ergebnisse die später ankommen werden ignoriert.

Bei Auswahl einer Station (`handleSelectedStation()`) wird Observer (`StationListComponent`) darüber informiert

```
export class StationListComponent implements OnInit {
  private stations: Array<StationValley> = null;

  constructor(
    private router: Router,
    …
  ) { }

  ngOnInit() {
  …
  }

  handleStationSelected(station: StationValley) {
    this.router.navigate(
      ['/station', 'all', station.code]
    )[2];
  }
}
```

```
<tb-search-station (stationSelected)=
  "handleStationSelected($event)">
</tb-search-station>
```

[2] oder this.router.navigate([`/station/all/${station.code}`]);