ANGULAR TOPICS

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

LOGISTICS

- Schedules
- Lunch & breaks
- Other questions ?

PRIMENG

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

PRIMENG - INSTALLATION

Enhance your web applications with PrimeNG's comprehensive suite of customizable, feature-rich UI components

- Install core library

```
npm install primeng @primeng/themes
```

PRIMENG - CONFIGURATION

- Add `providePrimeNG` (and Angular animations) to the application config providers

```
import { ApplicationConfig } from '@angular/core';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';
import Aura from '@primeng/themes/aura';
import { providePrimeNG } from 'primeng/config';

export const appConfig: ApplicationConfig = {
  providers: [
    provideAnimationsAsync(),
    providePrimeNG({
      theme: {
        preset: Aura,
        options: { darkModeSelector: '.dark-theme' },
      },
    }),
  ],
};
```

PRIMENG - USAGE

- Import the desired module from `primeng/*` and use it in your component templates

```
import { Component } from '@angular/core';
import { CardModule } from 'primeng/card';

@Component({
  selector: 'app-root',
  imports: [CardModule],
  template: `
    <p-card header="Simple Card">
      Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    </p-card>
  `,
})
export class AppComponent {}
```

PRIMENG - ICONS

- Install icons library

```
npm install primeicons
```

- Import the icons set in your `styles.scss` file

```
@import 'primeicons/primeicons.css';
```

- Use icons in your component templates

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

@Component({
  selector: 'app-root',
  template: `
    <i class="pi pi-check"></i>
    <i class="pi pi-times"></i>
  `,
})
export class AppComponent {}
```

PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

Lab 1

TRANSLOCO

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

TRANSLOCO - INSTALLATION

Transloco allows you to define translations for your content in different languages and switch between them easily in runtime

- Run the following schematics and specify the list of expected languages

```
ng add @jsverse/transloco
```

- The schematics will
  - install the package `@jsverse/transloco`
  - create a file `src/app/transloco-loader.ts`
  - create json files for each specified language in `src/assets/i18n/*.json` (*)
  - configure Transloco in your `src/app/app.config.ts` file

(*) But you should move the `src/assets/i18n` directory to `public/assets/i18n`

TRANSLOCO - APPLICATION CONFIG

- The schematics adds `provideTransloco` to your application config providers

```
import { ApplicationConfig, isDevMode } from "@angular/core";
import { provideTransloco } from "@jsverse/transloco";
import { TranslocoHttpLoader } from "./transloco-loader";

export const appConfig: ApplicationConfig = {
  providers: [
    provideTransloco({
      config: {
        availableLangs: ["en", "fr"],
        defaultLang: "en",
        reRenderOnLangChange: true,
        prodMode: !isDevMode(),
      },
      loader: TranslocoHttpLoader,
    }),
  ],
};
```

TRANSLOCO - TRANSLATION FILES

- `public/assets/i18n/en.json`

```json
{
  "appTitle": "User posts",
  "copyright": "Copyright"
}
```

- `public/assets/i18n/fr.json`

```json
{
  "appTitle": "Messages des utilisateurs",
  "copyright": "Tous droits réservés"
}
```

TRANSLOCO - DIRECTIVE

- Import `TranslocoDirective` directive in your components

```
import { Component, RouterOutlet } from '@angular/core';
import { TranslocoDirective } from '@jsverse/transloco';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, TranslocoDirective],
  template: `
    <ng-container *transloco="let t">
      <header>{{ t('appTitle') }}</header>

      <router-outlet />

      <footer>{{ t('copyright') }}</footer>
    </ng-container>
  `,
})
export class AppComponent {}
```

TRANSLOCO - PLUGINS | LOCALE 1/3

- Install the following plugin to add localization support to Transloco

```
npm install @jsverse/transloco-locale
```

TRANSLOCO - PLUGINS | LOCALE 2/3

- Configure the plugin by mapping Transloco languages to locales

```
import { ApplicationConfig } from '@angular/core';
import { provideTranslocoLocale } from '@jsverse/transloco-locale';

export const appConfig: ApplicationConfig = {
  providers: [
    provideTranslocoLocale({ langToLocaleMapping: { en: 'en-US', fr: 'fr-FR' } }),
  ],
};
```

TRANSLOCO - PLUGINS | LOCALE 3/3

- Use directives provided by the plugin

```
import { Component } from '@angular/core';
import { TranslocoDatePipe } from '@jsverse/transloco-locale';

@Component({
  selector: 'app-root',
  imports: [TranslocoDatePipe],
  template: `<p>{{ now | translocoDate }}</p>`,
})
export class AppComponent {
  now = Date.now();
}
```

TRANSLOCO - PLUGINS | PERSIST LANG 1/2

- Install the following plugin to persist selected language

```
npm install @jsverse/transloco-persist-lang
```

TRANSLOCO - PLUGINS | PERSIST LANG 2/2

- Configure the plugin to use `localStorage`
- Pre-load cached language (optional)

```
import { ApplicationConfig, inject, provideAppInitializer } from '@angular/core';
import { TranslocoService } from '@jsverse/transloco';
import {
  provideTranslocoPersistLang, TranslocoPersistLangService
} from '@jsverse/transloco-persist-lang';

export const appConfig: ApplicationConfig = {
  providers: [
    provideTranslocoPersistLang({ storage: { useValue: localStorage } }),
    provideAppInitializer(() =>
      inject(TranslocoService).load(inject(TranslocoPersistLangService).getCachedLang() ?? 'en'),
    ),
  ],
};
```

PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

Lab 2

NGRX SIGNALS

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

NGRX SIGNALS - INSTALLATION

NgRx Signals is a standalone library that provides a reactive state management solution and a set of utilities for Angular Signals

- Install library

```
npm install @ngrx/signals
```

NGRX SIGNALS - SIGNALSTORE WITHSTATE

- Create a store using the `signalStore` function
- Define initial state using `withState` function

```
import { signalStore, withState } from '@ngrx/signals';
import { Post } from '../shared/api/api.types';

interface UserPostsState {
  posts: Post[] | undefined;
  selectedPostId: number | undefined;
}

const initialState: UserPostsState = {
  posts: undefined,
  selectedPostId: undefined,
};

export const UserPostsStore = signalStore(
  withState(initialState),
);
```

NGRX SIGNALS - WITHCOMPUTED

- Computed signals can be added to the store using the `withComputed` feature

```
import { computed } from '@angular/core';
import { signalStore, withComputed } from '@ngrx/signals';
import { Post } from '../shared/api/api.types';

export const UserPostsStore = signalStore(
  withComputed(({ posts, selectedPostId }) => ({
    selectedPost: computed<Post | undefined>(() => {
      const postId = selectedPostId();
      return posts()?.find(({ id }) => id === postId);
    }),
  })),
);
```

NGRX SIGNALS - WITHMETHODS

- Methods can be added to the store using the `withMethods` feature
- Use `patchState` function to update the store state

```
import { inject } from '@angular/core';
import { patchState, signalStore, withMethods } from '@ngrx/signals';
import { firstValueFrom } from 'rxjs';
import { ApiService } from '../shared/api/api.service';
import { Post } from '../shared/api/api.types';

export const UserPostsStore = signalStore(
  withMethods((store, apiService = inject(ApiService)) => ({
    async loadPosts(userId: number) {
      const posts = await firstValueFrom(apiService.getUserPosts(userId));
      patchState(store, (state) => ({ ...state, posts }));
    },

    setPostId(selectedPostId: number | undefined) {
      patchState(store, (state) => ({ ...state, selectedPostId }));
    },
  })),
);
```

NGRX SIGNALS - PROVIDING

- Signal store can be provided globally…

```
import { signalStore } from '@ngrx/signals';

export const UserPostsStore = signalStore(
  { providedIn: 'root' },
);
```

- …or locally

```
import { Component } from '@angular/core';
import { UserPostsStore } from './user-posts/user-posts.store';

@Component({
  selector: 'app-root',
  providers: [UserPostsStore],
  template: `...`,
})
export class AppComponent {}
```

NGRX SIGNALS - INJECTING

- Consume the signal store in your components and services

```
import { Component, inject } from '@angular/core';
import { UserPostsStore } from './user-posts/user-posts.store';

@Component({
  selector: 'app-root',
  template: `
    @for (post of userPostsStore.posts(); track post.id) {

      <button (click)="userPostsStore.setPostId(post.id)"> {{ post.title }} </button>
    }
  `,
})
export class AppComponent {
  readonly userPostsStore = inject(UserPostsStore);
}
```

PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

Lab 3

RX RESOURCE

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

RX RESOURCE - DEFINITION

Projects a reactive request to an observable defined by a loader function, which exposes the emitted values via signals

```
import { Component, inject, input } from '@angular/core';
import { rxResource } from '@angular/core/rxjs-interop';

@Component({
  selector: 'app-user-posts',
  template: './user-posts.component.html'
})
export class UserPostsComponent {
  user = input.required<User>();

  protected apiService = inject(ApiService);

  protected posts = rxResource({
    request: () => this.user().id,
    loader: ({ request }) => this.apiService.getUserPosts(request),
  });
}
```

RX RESOURCE - PROPERTIES

- `RxResource` provides useful properties
  - `value()`: The current value of the Resource, or undefined if there is no current value
  - `status()`: The current status of the Resource (Idle, Error, Loading, Resolved, …)
  - `reload()`: Instructs the resource to reload
  - `isLoading()`: Whether this resource is loading a new value (or reloading the existing one)
  - `set()`: Convenience wrapper for `value.set`
  - `update()`: Convenience wrapper for `value.update`
  - …

RX RESOURCE - USAGE

- In this example, we are taking advantage of `isLoading()` and `value()` properties

```
<!-- user-posts.component.html -->

@if (posts.isLoading()) {

  Loading...

} @else {

  @for (post of posts.value(); track post.id) {

    <button> {{ post.title }} </button>

  } @empty {

    No posts available.
  }
}
```

PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

Lab 4

HTTP RESOURCE

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

HTTP RESOURCE - DEFINITION

Makes a reactive HTTP request and exposes the request status and response value as a writable resource

```
import { httpResource } from '@angular/common/http';
import { Component, input } from '@angular/core';
import { Post, User } from '../shared/api/api.types';

@Component({
  selector: 'app-user-posts',
  templateUrl: './user-posts.component.html',
})
export class UserPostsComponent {
  user = input.required<User>();

  protected posts = httpResource<Post[]>(() => ({
    url: 'https://jsonplaceholder.typicode.com/posts',
    params: { userId: this.user().id },
  }));
}
```

HTTP RESOURCE - PROPERTIES AND USAGE

- The `HttpResource` properties and usage are almost the same as for `RxResource`
  - `value()`: The current value of the Resource, or undefined if there is no current value
  - `status()`: The current status of the Resource (Idle, Error, Loading, Resolved, …)
  - `reload()`: Instructs the resource to reload
  - `isLoading()`: Whether this resource is loading a new value (or reloading the existing one)
  - `set()`: Convenience wrapper for `value.set`
  - `update()`: Convenience wrapper for `value.update`
  - …

😉 *Note that `HttpResource` and `RxResource` are derived from the `Resource` feature*

PHP
CSS
HTML
<HTML>
<HEAD>
<TITLE> Lorem ipsum </TITLE>
</HEAD>
<BODY>
<H1> Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam</H1>
<P> <IMG SRC= "Lorem ipsum"></P>

Lab 5

RXJS

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

RXJS

- Refers to a **paradigm** called ReactiveX (http://reactivex.io/)
  - an API for asynchronous programming with observable streams
  - implemented in all major programming languages: RxJava, Rx.NET, ...
- Let's focus on the JavaScript implementation: *RxJS*

RXJS - IN A NUTSHELL

- Observables:
  - represent a stream of data that can be subscribed to
  - allowing multiple values to be emitted over time

RXJS - BUILDING BLOCKS

- To understand RxJS, you need to learn the following concepts:
  - `Observable` and `Observer`
  - `Subscription`
  - `Operators`
  - `Subjects`

RXJS - OBSERVABLE & OBSERVER 1/4

```
import { Observable, Observer } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);                                  // <-- Emit next value
  subscriber.next(2);                                  // <-- Emit next value
  subscriber.complete();                               // <-- Mark as complete
});

const observer: Partial<Observer<number>> = {
  next: (data: number) => console.log(data),           // <-- Listen to "next" events
  complete: () => console.log('Done'),                 // <-- Listen to "complete" event
};

data$.subscribe(observer);                             // output: 1, 2, Done
```

- Use the *subscriber* to shape the behavior of the observable
- Use the *observer* to specify which events you want to listen to
- Subscriber and observer methods match: `next`, `complete` (and also `error`)

RXJS - OBSERVABLE & OBSERVER 2/4

```
import { Observable, Observer } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);                                   // <-- Emit next value
  subscriber.next(2);                                   // <-- Emit next value
  subscriber.error('Oops!');                            // <-- Mark as in error
});

const observer: Partial<Observer<number>> = {
  next: (data: number) => console.log(data),            // <-- Listen to "next" events
  error: (err: unknown) => console.error(err),          // <-- Listen to "error" event
};

data$.subscribe(observer);                              // output: 1, 2, Oops!
```

- Example of `error` event instead of `complete` event

RXJS - OBSERVABLE & OBSERVER 3/4

```
import { Observable, Observer } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
  subscriber.next(3);                                 // <-- Value NOT emitted
});

const observer: Partial<Observer<number>> = {
  next: (data: number) => console.log(data),          // <-- Object property as "next" observer
};

data$.subscribe(observer);                            // output: 1, 2
```

- Once the observable completes (or is in error), further calls to `next` are ignored

RXJS - OBSERVABLE & OBSERVER 4/4

```
import { Observable, Observer } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
  subscriber.next(3);                            // <-- Value NOT emitted
});


const next = (data: number) => console.log(data); // <-- Function as "next" observer


data$.subscribe(next);                            // output: 1, 2
```

- You can use a function as observer to simply listen to `next` events

RXJS - SUBSCRIPTION 1/3 (NOT YET...)

- Example of an observable that completes itself properly (without memory leak)

```
import { Observable } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  let data = 0;

  const interval = setInterval(() => {
    subscriber.next(++data);                // <-- Emit next value every second

    if (data === 3) {                       // <-- Until this value
      clearInterval(interval);              // <-- Cleanup interval to prevent memory leak
      subscriber.complete();                // <-- Then mark as complete
    }
  }, 1000);
});

data$.subscribe({
  next: (data: number) => console.log(data),
  complete: () => console.log('Done'),
}); // output: 1, 2, 3, Done
```

RXJS - SUBSCRIPTION 2/3

- Example of an observable that never completes and have a *memory leak*! 😱

```
import { Observable, Subscription } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  let data = 0;

  setInterval(() => {
    subscriber.next(++data);                // <-- Emit next value every second ad infinitum
    console.log('tick');
  }, 1000);


});

const subscription: Subscription = data$.subscribe((data: number) => {
  console.log(data);
  if (data === 3) {
    subscription.unsubscribe();             // <-- Unsubscribe from data$
                                            //     but the observable still ticking...
  }
}); // output: 1, tick, 2, tick, 3, tick, tick, tick, ...
```

RXJS - SUBSCRIPTION 3/3

- Example of an observable that never completes but cleans up itself properly

```typescript
import { Observable, Subscription } from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  let data = 0;

  const interval = setInterval(() => {
    subscriber.next(++data);                    // <-- Emit next value every second ad infinitum
    console.log('tick');
  }, 1000);

  return () => clearInterval(interval); // <-- Return the resource cleanup function
});

const subscription: Subscription = data$.subscribe((data: number) => {
  console.log(data);
  if (data === 3) {
    subscription.unsubscribe();                 // <-- Unsubscribe from data$ and execute
                                                //     the resource cleanup function
  }
}); // output: 1, tick, 2, tick, 3, tick
```

RXJS - OBSERVABLE SOURCE 1/4

- Observable can be created using `of` function:

```
import { of } from 'rxjs';

const source$ = of('hello', 123);

source$.subscribe(console.log); // output: hello, 123
```

RXJS - OBSERVABLE SOURCE 2/4

- Observable can be created from existing value (like `Array` or `Promise`) using `from` function:

```
import { from } from 'rxjs';

const fromArray$ = from(['hello', 123]);

fromArray$.subscribe(console.log); // output: hello, 123

const fromPromise$ = from(new Promise((resolve) => resolve('Done!')));

fromPromise$.subscribe(console.log); // output: Done!
```

RXJS - OBSERVABLE SOURCE 3/4

- Observable can be created using `fromEvent` function:

```
import { fromEvent } from 'rxjs';

const fromDocumentClick$ = fromEvent(document, 'click');

fromDocumentClick$.subscribe((event: Event) => console.log(event));
```

RXJS - OBSERVABLE SOURCE 4/4

- Observable that emits an error event can be created using `throwError` function:

```
import { throwError } from 'rxjs';

const error$ = throwError(() => new Error('Oops!'));

error$.subscribe({
  error: (err: Error) => console.error(err.message) // output: Oops!
});
```

RXJS - OPERATORS | SYNCHRONOUS 1/2

```
import {
  Observable, filter, map // <-- "filter" and "map": synchronous transformations
} from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.next(4);
  subscriber.complete();
});

data$.pipe(/* no operator */).subscribe(console.log);                    // output: 1, 2, 3, 4

data$.pipe(filter((data) => data % 2 === 0)).subscribe(console.log);// output: 2, 4

data$.pipe(map((data) => data * 10)).subscribe(console.log);          // output: 10, 20, 30, 40

data$.pipe(
  filter((data) => data % 2 === 0),
  map((data) => data * 10)
).subscribe(console.log);                                             // output: 20, 40
```

RXJS - OPERATORS | SYNCHRONOUS 2/2

```
import {
  Observable, map, tap // <-- "map" and "tap": synchronous transformations
} from 'rxjs';

const data$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.next(4);
  subscriber.complete();
});

let evenValuesCount = 0;                                      // <-- Defined out of the
stream

data$.pipe(
  tap((data) => {
    if (data % 2 === 0) evenValuesCount += 1;                 // <-- Handle side effect
    return 'ignored value';                                   // <-- Return value is
ignored
  }),
  map((data) => data * 10)
).subscribe(console.log);                                     // output: 10, 20, 30, 40
```

RXJS - OPERATORS | ASYNCHRONOUS 1/4

```typescript
import { Observable, concatMap } from 'rxjs'; // <-- "concatMap": asynchronous transformation

const todoId$ = new Observable<number>((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.complete();
});

const fetchTodoFactory$ = (id: number) => new Observable((subscriber) => {
  fetch(`https://jsonplaceholder.typicode.com/todos/${id}`)
    .then((response) => response.json())
    .then((todo) => {
      subscriber.next(todo);                       // <-- Emit "next" event
      subscriber.complete();                       // <-- Emit "complete" event
    })
    .catch((err) => subscriber.error(err));    // <-- Emit "error" event
});

todoId$.pipe(concatMap((id) => fetchTodoFactory$(id))).subscribe(console.log);

// output: { id: 1, title: 'delectus aut autem', completed: false }
// output: { id: 2, title: 'quis ut nam facilis et officia qui', completed: false }
```

RXJS - OPERATORS | ASYNCHRONOUS 2/4

```
import { Observable, fromEvent, map, switchMap } from 'rxjs';

const input = document.createElement('input');
input.type = 'number';
document.body.appendChild(input);

fromEvent(input, 'input').pipe(
  map((event) => (event.target as HTMLInputElement).value),
  switchMap((id) => new Observable((subscriber) => {
    const controller = new AbortController();
    fetch(`https://jsonplaceholder.typicode.com/todos/${id}`, { signal: controller.signal })
      .then((response) => response.json())
      .then((todo) => {
        subscriber.next(todo);
        subscriber.complete();
      })
      .catch((err) => subscriber.error(err));
    return () => controller.abort();
  }))
).subscribe(console.log);
```

RXJS - OPERATORS | ASYNCHRONOUS 3/4

- The `catchError` operator should:
  - return another observable
  - throw again to be handled by another `catchError` or the observer's `error` handler

```
import { interval, tap, catchError, of } from 'rxjs';

const source$ = interval(1000).pipe(
  tap((value) => {
    if (value > 3) throw new Error('Oops!');
  }),
  catchError(() => of('Fallback'))             // <-- Trigger "next" event
);

source$.subscribe({
  next: console.log,
  error: console.error,                         // <-- Never called
  complete: () => console.log('Done!')
});

// Output => 0, 1, 2, 3, Fallback, Done!
```

RXJS - OPERATORS | ASYNCHRONOUS 4/4

- `concatMap`
  Projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next.
- `mergeMap`
  Projects each source value to an Observable which is merged in the output Observable.
- `switchMap`
  Projects each source value to an Observable which is merged in the output Observable, emitting values only from the most recently projected Observable.
- *a lot more…*

RXJS - SUMMARY SO FAR

- By convention, a variable representing an observable ends with the symbol `$`
- The `Observable` implementation is a function that use the `Subscriber` methods to emit the stream events
    - `.next()`, `.complete()` and `.error()`
- The `.subscribe()` method activates the observable to emit its data stream
    - It accepts an object (`Partial<Observer>`) or a function as `Observer` to listen to the stream events
    - It returns a `Subscription` allowing the consumer to `.unsubscribe()` from the activated observable
- Unsubscription is necessary to avoid memory leaks when the consumer is no longer interested in the data
    - Unless the observable is already in "complete" (or "error" state)
- The `Operators` allow to transform the emitted values and make the observables very powerful

RXJS - SUBJECT 1/2

- A `Subject` implements both `Observable` and `Observer` interfaces

```
import { Subject } from 'rxjs';

const subject$ = new Subject();

// Act as Observable
subject$.subscribe(/* ... */);
subject$.pipe(/* ... */);

// Act as Observer
subject$.next(/* ... */);
subject$.error(/* ... */);
subject$.complete(/* ... */);

// Can be converted into a simple Observable...
const observable$ = subject$.asObservable();

// ...hidding the Observer interface
observable$.next(/* ... */); // ❌ Property 'next' does not exist on type 'Observable'
```

RXJS - SUBJECT 2/2

- Unlike observable:
  - subject implementation lives outside its instantiation (calling `next`, `error`, `complete`)
  - subject can emit stream events even before any subscription ("*hot*" observable)
  - subject is "*multicast*" (all subscribers share the same stream events)

```
const data$ = new Subject<string>();

data$.next('A');                               // <-- value is lost

data$.subscribe((data) => console.log(`#sub1(${data})`));

data$.next('B');                               // <-- value recieved by subscriber 1

data$.subscribe((data) => console.log(`#sub2(${data})`));

data$.next('C');                               // <-- value recieved by subscribers 1 and 2
data$.next('D');                               // <-- value recieved by subscribers 1 and 2
data$.complete();
// output: #sub1(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D)
```

RXJS - OBSERVABLE COMPARED TO SUBJECT

- Unlike subject:
  - observable implementation lives inside its instantiation (calling `next`, `error`, `complete`)
  - observable emits stream events only when subscribing ("*cold*" observable)
  - observable is "*unicast*" (each subscriber receive a new data stream)

```
import { Observable } from 'rxjs';

const observable$ = new Observable<string>((subscriber) => {
  // This is where implementation takes place...
  subscriber.next('A');
  subscriber.next('B');
  subscriber.complete();
});

data$.subscribe((data) => console.log(`#sub1(${data})`));
data$.subscribe((data) => console.log(`#sub2(${data})`));

// output: #sub1(A), #sub1(B), #sub2(A), #sub2(B)
```

RXJS - SUBJECT | BEHAVIORSUBJECT

A variant of Subject that requires an initial value and emits its current value whenever it is subscribed to.

```
import { BehaviorSubject } from 'rxjs';

const data$ = new BehaviorSubject<string>('A');            // <-- Initial value

data$.subscribe((data) => console.log(`#sub1(${data})`)); // <-- #sub1 receive 'A'

data$.next('B');                                           // <-- #sub1 receive 'B'

data$.subscribe((data) => console.log(`#sub2(${data})`)); // <-- #sub2 receive 'B'

data$.next('C');
data$.next('D');

console.log(`#snapshot(${data$.value})`); // <-- and you have access to the instant value!

data$.complete();

// output: #sub1(A), #sub1(B), #sub2(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D), #snapshot(D)
```

RXJS - SUBJECT | REPLAYSUBJECT

A variant of Subject that "replays" old values to new subscribers by emitting them when they first subscribe.

```
import { ReplaySubject } from 'rxjs';

const data$ = new ReplaySubject<string>(2);                 // <-- Number of events to replay

data$.next('A');

data$.subscribe((data) => console.log(`#sub1=${data}`));  // <-- #sub1 receive 'A'

data$.next('B');                                            // <-- #sub1 receive 'B'

data$.subscribe((data) => console.log(`#sub2=${data}`));  // <-- #sub2 receive 'A' and 'B'

data$.next('C');
data$.next('D');
data$.complete();

// output: #sub1(A), #sub1(B), #sub2(A), #sub2(B), #sub1(C), #sub2(C), #sub1(D), #sub2(D)
```

RXJS - STATE MANAGEMENT 1/3

- Expose application data through service facade and observables

```
import { BehaviorSubject, Observable, tap, map } from 'rxjs';

export class TodoService {
  private _todos$ = new BehaviorSubject<Todo[] | undefined>(undefined);

  todos$ = this._todos$.asObservable();

  get todosSnapshot() { return this._todos$.value; }

  dispatch(): Observable<void> {
    return from(fetch<Todo[]>('https://jsonplaceholder.typicode.com/todos')).pipe(
      tap((todos) => {
        this._todos$.next(todos);       // <-- Using `tap` operator for "side-effects"

      }),
      map(() => undefined),             // <-- Force the consumer to use the `todos$` property
    );
  }
}
```

RXJS - STATE MANAGEMENT 2/3

- Same example but using a `ReplaySubject` instead of a `BehaviorSubject`

```
import { ReplaySubject, Observable, tap, map } from 'rxjs';

export class TodoService {
  todosSnapshot?: Todo[];

  private _todos$ = new ReplaySubject<Todo[]>(1); // <-- `undefined` no longer required

  todos$ = this._todos$.asObservable();

  dispatch(): Observable<void> {
    return from(fetch<Todo[]>('https://jsonplaceholder.typicode.com/todos')).pipe(
      tap((todos) => {
        this.todosSnapshot = todos;
        this._todos$.next(this.todosSnapshot);  // <-- Using `tap` operator for "side-effects"
      }),
      map(() => undefined),               // <-- Force the consumer to use the `todos$` property
    );
  }
}
```

RXJS - STATE MANAGEMENT 3/3

- Determine the appropriate place to trigger data fetching
- Don't forget to handle errors!
- Consume the data anywhere

```
// app.component.ts
const todoService = new TodoService();

let showError = false;

todoService.dispatch().subscribe({ error: () => (showError = true) });

// todo-list.component.ts
todoService.todos$.subscribe((todos) => console.log(todos));

// todo-count.component.ts
todoService.todos$.pipe(map(({ length }) => length)).subscribe((length) => console.log(length));
```

RXJS - CONCLUSION

- Now you know the main concepts of RxJS:
  - `Observable` and `Observer`
  - `Subscription`
  - `Operators`
  - `Subjects`
- But your journey has just begun
- And there's so much more to learn:
  - `combineLatest`, `debounceTime`, `delay`, `pairwise`, `reduce`, `share`, `shareReplay`, `skip`, `skipUntil`, `skipWhile`, `startWith`, `take`, `takeUntil`, `toArray`, `withLatestFrom`, `zip`, …

IN-DEPTH RESOURCES

SUMMARY

- PrimeNG
- Transloco
- NgRx signals
- RxResource
- HttpResource
- RxJS
- In-depth resources

IN-DEPTH RESOURCES

- **PrimeNG:** https://primeng.org/
- **Transloco:** https://jsverse.gitbook.io/transloco
- **NgRx signals:** https://ngrx.io/guide/signals
- **Resource:** https://angular.dev/guide/signals/resource
- **RxResource:** https://angular.dev/api/core/rxjs-interop/rxResource
- **HttpResource:** https://angular.dev/api/common/http/httpResource
- **RxJS:** https://rxjs.dev/

THANK YOU