What is memory leakage issue and how to overcome from that

- **Reference Link :**
  - https://medium.com/angular-in-depth/how-to-create-a-memory-leak-in-angular-4c583ad78b8b
  - https://www.twilio.com/blog/prevent-memory-leaks-angular-observable-ngondestroy
  - https://medium.com/@sub.metu/angular-how-to-avoid-memory-leaks-from-subscriptions-8fa925003aa9

- **Memory Management :**
  - Allocate the needed memory
  - Read and write the allocated memory
  - Release the memory as soon as it's not needed anymore.

- Lets say you have two routes in your website — **/home** and **/product.**
- Both show the homeComponent and the productComponent respectively. Let's assume that both the components are subscribing to some Observable. We don't care what Observable it is. When the user navigates to the product page from the home page, the homeComponent is destroyed and the productComponent initialises. The homeComponent was subscribed to the Observable and now its destroyed. We did not unsubscribe before destroying the component.

- **Handle the subscription :**
  - To avoid memory leaks, it's essential to unsubscribe from an Observable correctly when the subscription is not needed anymore, e.g. when our component is destroyed.
  - We implement the ngOnDestroy lifecycle hook on our component. Every time the component gets destroyed we call next and complete on our destroy$.
  - We then use the takeUntil operator and pass our destroy$ stream to it. This guarantees that the subscription is cleaned (unsubscribed) once our component gets destroyed.

```
@Component({
  selector:'app-sub',
  //...
})
export class SubComponent implements OnDestroy {

  private destroy$: Subject<void> = new Subject<void>();
  randomNumber = 0;

  constructor(private dummyService: DummyService) {
    dummyService.some$.pipe(
            takeUntil(this.destroy$)
        ).subscribe(value => this.randomNumber = value);
  }

  ngOnDestroy(): void {
      this.destroy$.next();
      this.destroy$.complete();
  }

}
```

- **Remember to Unsubscribe :**
  - npm install @angular-extensions/lint-rules --save-dev
  - and add it to your **tslint.json**

```
{
      "extends": [
            "tslint:recommended",
            "@angular-extensions/lint-rules"
      ]
}
```

Server Side Rendering

- **Reference Link :**
  - https://angular.io/guide/universal
  - https://www.youtube.com/watch?v=s6-G0TWZkvg&t=377s
  - https://trilon.io/blog/angular-universal-v9-whats-new
  - https://medium.com/jspoint/server-side-rendering-ssr-in-angular-5-the-simplest-and-quickest-ssr-approach-34cf53224f32

- **Why we use Server Side Rendering (SSR) :**
  - Rank Web Application through search engine optimization (SEO)
  - Improve performance on mobile and low-powered devices
  - Show the first page quickly with a first-contentful paint (FCP)

- **Steps :**
  - ng add @nguniversal/express-engine
  - ng run <app-name>:server-ssr
  - ng run <app-name>:prerender

- Search engines like Google & Bing & social media sites like Facebook & Twitter rely on web crawlers to index your application content and make that content searchable on the web.

- Taking a look at our **package.json**, we can see we now have a few new additional scripts / shortcuts to running some of the new features of the Angular Universal builders!

```
"scripts": {
        ...
        "dev:ssr": "ng run angular-universal-v9:serve-ssr",
        "serve:ssr": "node dist/angular-universal-v9/server/main.js",
        "build:ssr": "ng build --prod && ng run angular-universal-v9:server:production",
        "prerender": "ng run angular-universal-v9:prerender"
}
```

- In Angular, basically **index.html** page is served from express server for all the URL paths and that index.html page is passed through some **express view engine** which injects HTML into <app-root></app-root>, based on current route and component for that route.

Garbage Collection

- A garbage collection removes garbage.
- Its job is to clean up memory that is not needed anymore.
- JavaScript has automatic garbage collection, so your code only has to make sure it does not retain any references that it no longer needs
- Once an object is no longer referenced and therefore is not reachable by the application code, the **garbage** collector removes it and reclaims the unused memory.