Who am I?

- My name is **Marius Oprea** a frontend developer, currently working with Angular(commonly referred as Angular 2+) since 2016, when the first version has been released.

What I will present today?

- Change detection in Angular and more specific the **OnPush strategy**.

What is change detection

https://angular.io/guide/change-detection

Change detection is the process through which Angular checks to see whether your application state has changed, and if any DOM needs to be updated.

Angular runs its change detection mechanism periodically so that changes to the data model are reflected in an app's view.

At a high level, Angular walks your components from top to bottom, looking for changes.

On every change detection cycle, Angular synchronously:

- **Evaluates all template expressions in all components**, unless specified otherwise, based on that each component's detection strategy

- **Executes** the `ngDoCheck`, `ngAfterContentChecked`, `ngAfterViewChecked`, and `ngOnChanges` **lifecycle hooks**.

A single slow computation within a template or a lifecycle hook can slow down the entire change detection process because Angular runs the computations sequentially.

Optimizing slow computations

Optimizing the underlying algorithm

Don't use inpure heavy computation methods in template

```
<div {{ inpureHeavyComputation(data) }}>
```

Use pure pipes for caching (impure pipes are called on each detection cycle)

```
<div {{  data | customPurePipe }}>
```

https://angular.io/guide/pipes

Pipes are simple functions to use in templates to accept an input value and return a transformed value

Minimize computations in lifecycle hooks methods

```
@Input() value: number;
let oldValue: number = 0;
ngDoCheck() {
   If (oldValue !== value) {
      // add functionality here
      oldValue = value;
   }
}
```

Use memoization for caching pure methods

Is an optimization technique used to speed up applications by storing the results of pure methods and returning the cached result when the method is called again with the same input.

```
@ memoize
pureheavyComputation(data) {
...
}
```

https://www.section.io/engineering-education/an-introduction-to-memoization-in-javascript/

Vanilla Javascript implementation

```
const memoize = (func) => {
  const results = {};
  return (...args) => {
    const argsKey = JSON.stringify(args);
    if (!results[argsKey]) {
      results[argsKey] = func(...args);
    }
    return results[argsKey];
  };
};


clumsysquare = memoize((num: any) => {
    let result = 0;
    for (let i = 1; i <= num; i++) {
      for (let j = 1; j <= num; j++) {
        result++;
      }
    }
    return result;
});
```

```
console.time("First call");
console.log(clumsysquare(9467));
console.timeEnd("First call");          //  First call: 53.9482421875 ms

console.time("Second call");
console.log(clumsysquare(9467));
console.timeEnd("Second call");     //  Second call: 0.02392578125 ms

console.time("Third call");
console.log(clumsysquare(9467));  //  Third call: 0.017822265625 ms
console.timeEnd("Third call");

console.time("Multiplication");
console.log(9467*9467);                 //  Multiplication:  0.015869140625 ms
console.timeEnd("Multiplication");

console.time("Math.pow");
console.log(Math.pow(9467, 2));    // Math.pow: 0.016357421875 ms
console.timeEnd("Math.pow");
```

Angular implementation

```
// memoization method
export function memoizee(func: any) {
  const results: any = {};
  return (...args: any) => {
    const argsKey = JSON.stringify(args);
    if (!results[argsKey]) {
      results[argsKey] = func(...args);
    }
    return results[argsKey];
  };
};
```

```
// @memoize decorator
export function memoize() {
  return function(target: any, key:
    any, descriptor: any) {
    const oldFunction = descriptor.value;
    const newFunction: any
    = memoizee(oldFunction)
    descriptor.value = function () {
       return newFunction.apply(this,
    arguments)
    }
  }
}
```

```
// method call
  @memoize()
  clumsysquare(num: any) {
    let result = 0;
    for (let i = 1; i <= num; i++) {
      for (let j = 1; j <= num; j++) {
        result++;
      }
    }
    return result;
  }
```

https://www.npmjs.com/package/memoizee

https://github.com/mgechev/memo-decorator

https://medium.com/angular-in-depth/how-to-improve-angular-performance-by-just-adding-just-8-characters-877bde708ddd

Use Zone.js and ngZones for further optimization

- **Zone.js is an execution context that persist across async tasks and a signaling mechanism, that Angular uses to detect when an application state might have changed**. It captures asynchronous operations like `setTimeout`, network requests, and event listeners. Angular **schedules change detection based on signals from Zone.js**

- A Zone is an execution context that persists across async tasks
- There are cases in which scheduled tasks or microtasks don't make any changes in the data model, which makes running change detection unnecessary. Common examples are:
  - `requestAnimationFrame`, `setTimeout` or `setInterval`
  - Task or microtask scheduling by third-party libraries

https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

https://github.com/angular/angular/blob/main/packages/zone.js/lib/zone.ts

https://github.com/angular/angular/tree/main/packages/zone.js/example

https://angular.io/guide/change-detection-zone-pollution

https://angular.io/guide/zone

https://www.w3schools.com/js/js_this.asp

https://www.youtube.com/watch?v=3IqtmUscE_U&t=150s&ab_channel=ng-conf

https://medium.com/swlh/what-is-zone-js-and-how-can-i-use-it-63ce08a55962

```
constructor(private ngZone: NgZone) {}
ngOnInit() {
    this.ngZone.runOutsideAngular(() => {
        // this code will not trigger change detection
    });
}
```

Skipping component subtrees

- **JavaScript, by default, uses mutable data structures** that you can reference from multiple different components. **Angular runs change detection over your entire component tree** to make sure that the most up-to-date state of your data structures is reflected in the DOM.
- Change detection is sufficiently fast for most applications. However, **when an application has an especially large component tree, running change detection across the whole application can cause performance issues**. You can address this by configuring change detection to only run on a subset of the component tree.
- If you are confident that a part of the application is not affected by a state change, you can use OnPush to skip change detection in an entire component subtree.

Using OnPush

- **OnPush change detection instructs Angular to run change detection for a component subtree only** when:
- The root component of the subtree receives new inputs as the result of a template binding. Angular compares the current and past value of the input with ==
- **Angular handles an event** *(e.g. using event binding, output binding, or* `@HostListener`*)* in the subtree's root component or any of its children **whether they are using OnPush change detection or not**.
- You can set the change detection strategy of a component to `OnPush` in the `@Component` decorator

Demo App

- https://github.com/marius-oprea/onPush

Thank you!