Angular

Two-way Binding

Data binding in AngularJS is the synchronization between the model and the view.

When data in the *model* changes, the *view* reflects the change, and when data in the *view* changes, the *model* is updated as well.

A spy directive can provide insight into a DOM object that you cannot change directly. Obviously you can't touch the implementation of a native . You can't modify a third party component either. But you can watch both with a directive.

```
// Spy on any element to which it is applied.
// Usage: <div mySpy>...</div>
@Directive({selector: '[mySpy]'})
export class SpyDirective implements OnInit, OnDestroy {
  constructor(private logger: LoggerService) { }
  ngOnInit()    { this.logIt(`onInit`); }
  ngOnDestroy() { this.logIt(`onDestroy`); }
  private logIt(msg: string) {
    this.logger.log(`Spy #${nextId++} ${msg}`);
  }
}
```

```
<div *ngFor="let hero of heroes" mySpy class="heroes">
  {{hero}}
</div>
```

Adding a hero results in a new hero . The spy's ngOnInit() logs that event. The Reset button clears the heroes list. Angular removes all hero elements from the DOM and destroys their spy directives at the same time. The spy's ngOnDestroy() method reports its last moments.

The ngOnInit() and ngOnDestroy() methods have more vital roles to play in real applications.

Use ngOnInit() for two main reasons:

To perform complex initializations shortly after construction.

To set up the component after Angular sets the input properties.

Use the DoCheck hook to detect and act upon changes that Angular doesn't catch on its own.

```
ngDoCheck() {
  if (this.hero.name !== this.oldHeroName) {
    this.changeDetected = true;
    this.changeLog.push(`DoCheck: Hero name changed to "${this.hero.name}" from "${this.oldHeroName}"`);
    this.oldHeroName = this.hero.name;
  }
```

AfterView

The AfterView sample explores the AfterViewInit() and AfterViewChecked() hooks that Angular calls after it creates a component's child views.

Here's a child view that displays a hero's name in an <input>:

ChildComponent

content_copy

```
@Component({
  selector: 'app-child-view',
  template: '<input [(ngModel)]="hero">'
})
export class ChildViewComponent {
  hero = 'Magneta';
}
```

The AfterViewComponent displays this child view within its template:

AfterViewComponent (template)

content_copy

```
template: `
  <div>-- child view begins --</div>
    <app-child-view></app-child-view>
  <div>-- child view ends --</div>`
```

The following hooks take action based on changing values within the child view, which can only be reached by querying for the child view via the property decorated with @ViewChild.

AfterViewComponent (class excerpts)

content_copy

```
export class AfterViewComponent implements  AfterViewChecked, AfterViewInit {
  private prevHero = '';
  // Query for a VIEW child of type `ChildViewComponent`
  @ViewChild(ChildViewComponent, {static: false}) viewChild:
ChildViewComponent;
  ngAfterViewInit() {
    // viewChild is set after the view has been initialized
    this.logIt('AfterViewInit');
    this.doSomething();
  }
  ngAfterViewChecked() {
    // viewChild is updated after the view has been checked
    if (this.prevHero === this.viewChild.hero) {
      this.logIt('AfterViewChecked (no change)');
    } else {
      this.prevHero = this.viewChild.hero;
      this.logIt('AfterViewChecked');
      this.doSomething();
    }
  }
  // ...
}
```

AfterContent

The AfterContent sample explores the AfterContentInit() and AfterContentChecked() hooks that Angular calls after Angular projects external content into the component.

Content projection

Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot. AngularJS developers know this technique as transclusion.

Consider this variation on the previous AfterView example. This time, instead of including the child view within the template, it imports the content from the AfterContentComponent's parent. Here's the parent's template:

AfterContentParentComponent (template excerpt)

content_copy

```
`<after-content>
  <app-child></app-child>
 </after-content>`
```

Notice that the <app-child> tag is tucked between the <after-content> tags. Never put content between a component's element tags unless you intend to project that content into the component.

Now look at the component's template:

AfterContentComponent (template)

content_copy

```
template: `
  <div>-- projected content begins --</div>
    <ng-content></ng-content>
  <div>-- projected content ends --</div>`
```

The <ng-content> tag is a placeholder for the external content. It tells Angular where to insert that content. In this case, the projected content is the <app-child> from the parent.

AfterContent hooks

AfterContent hooks are similar to the AfterView hooks. The key difference is in the child component.

The AfterView hooks concern ViewChildren, the child components whose element tags appear within the component's template.

The AfterContent hooks concern ContentChildren, the child components that Angular projected into the component.

The following AfterContent hooks take action based on changing values in a content child, which can only be reached by querying for them via the property decorated with @ContentChild.

AfterContentComponent (class excerpts)

content_copy

```
 export class AfterContentComponent implements AfterContentChecked,
AfterContentInit {
  private prevHero = '';
  comment = '';
  // Query for a CONTENT child of type `ChildComponent`
  @ContentChild(ChildComponent, {static: false}) contentChild: ChildComponent;
  ngAfterContentInit() {
    // contentChild is set after the content has been initialized
    this.logIt('AfterContentInit');
    this.doSomething();
  }
  ngAfterContentChecked() {
    // contentChild is updated after the content has been checked
    if (this.prevHero === this.contentChild.hero) {
      this.logIt('AfterContentChecked (no change)');
    } else {
      this.prevHero = this.contentChild.hero;
      this.logIt('AfterContentChecked');
      this.doSomething();
    }
  }
  // ...
}
```

No unidirectional flow worries with AfterContent

This component's doSomething() method update's the component's data-bound comment property immediately. There's no need to wait.

Recall that Angular calls both AfterContent hooks before calling either of the AfterView hooks. Angular completes composition of the projected content before finishing the composition of this component's view. There is a small window between the AfterContent... and AfterView... hooks to modify the host view

BrowserModule – The browser module is imported from @angular/platform-browser and it is used when you want to run your application in a browser.

CommonModule – The common module is imported from @angular/common and it is used when you want to use directives - NgIf, NgFor and so on.
FormsModule – The forms module is imported from @angular/forms and it is used when you build template driven forms.
RouterModule – The router module is imported from @angular/router and is used for routing RouterLink, forRoot, and forChild.
HttpClientModule –The HttpClientModule is imported from @angular/common/http and it used to initiate HTTP request and responses in angular apps. The HttpClient is more modern and easy to use the alternative of HTTP.
best practices for angular
1. Use of Angular CLI
Angular comes equipped with a command line interface called Angular CLI, which helps you scaffold(burden) your code in minutes.
Angular CLI can be installed easily via terminal by typing in the following command:
npm install -g @angular/cli
To generate and start a new project locally, us
ng new PROJECT-NAME
cd PROJECT-NAME
ng serve -o
2. Develop Angular apps in modular fashion using core, shared and feature modules
When you start developing in Angular, you may be tempted to disregard creation of modules for the sake of using components solely. That approach might be fine for smaller apps, but as your app starts to grow, development will become cumbersome. That's when separation of concerns steps in, which is fueled by a modular Angular app.
Splitting your app into core, shared and multiple feature modules will make your life much easier. Each module can have its own components, services, directives and pipes.
2. Develop Angular apps in modular fashion using core, shared and feature modules

When you start developing in Angular, you may be tempted to disregard creation of modules for the sake of using components solely. That approach might be fine for smaller apps, but as your app starts to grow, development will become cumbersome. That's when separation of concerns steps in, which is fueled by a modular Angular app.

Splitting your app into core, shared and multiple feature modules will make your life much easier. Each module can have its own components, services, directives and pipes.

Core module should be created of components (i.e. header, main navigation, footer) that will be used across the entire app.

Shared module can have components, directives and pipes that will be shared across multiple modules and components, but not the entire app necessarily.

Last but not least are feature modules. Per Angular's official documentation, feature module is:

an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

3. Lazy loading a feature module

Lazy loading a feature module is the best approach when it comes to accessing a module via Angular's routing. That way we will be able to make our Angular app faster. In other words, a feature module won't be loaded initially, but when you decide to initiate it. Therefore, making an initial load of the Angular app faster too! It's a nifty feature.

Here is an example on how to initiate a lazy loaded feature module via app-routing.module.ts file.

```
const routes: Routes = [
 {
   path: 'dashboard',
   loadChildren: 'app/dashboard/dashboard.module#DashboardModule',
```

```
    component: CoreComponent
  }
];
```

4. Use of smart vs. dummy components

Most common use case of developing Angular's components is a separation of smart and dummy components. Think of a dummy component as a component used for presentation purposes only, meaning that the component doesn't know where the data came from. For that purpose, we can use one or more smart components that will inherit dummy's component presentation logic.

Here is a useful read that will give you more insight on the subject, because the complexity of the matter requires a new post/article.

5. Proper use of dependency injection in Angular 6

Come Angular 6, there have been some changes for the better when it comes to providing services in Angular. The old way would be to state services explicitly inside providers inside of NgModule. Not anymore. If you have been using the latest version of Angular and Angular CLI, you will notice that the CLI generates the following piece of code whenever you generate a new service:

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

It is the new way of providing services in Angular. Also, please note that root can be replaced with any module where you wish to provide your service with. Your old code will still work, but consider replacing it with the new one as soon as possible.

6. Aliases for imports

Creating aliases for imports is a plus. We may use imports three folders deep sometimes, so the following import is not the ideal solution:

```
import { LoaderService } from '../../../loader/loader.service';
```

Add the following piece of code into tsconfig.json file to make your imports short and well organized across the app:

```
{
  "compileOnSave": false,
  "compilerOptions": {
```

```
   removed for brevity,
   "paths": {
    "@app/*": ["app/*"],
    "@env/*": ["environments/*"]
   }
  }
}
```

When you're done, these imports

```
import { LoaderService } from '../../../loader/loader.service';
import { environment } from '../../environment';
```

should be refactored into these:

```
import { LoaderService } from '@app/loader/loader.service';
import { environment } from '@env/environment';
```

performance in angular

1) trackBy

When using ngFor to loop over an array in templates, use it with a trackBy function which will return an unique identifier for each item.

Why?

When an array changes, Angular re-renders the whole DOM tree. But if you use trackBy, Angular will know which element has changed and will only make DOM changes for that particular element.

For a detailed explanation on this, please refer to this article by Netanel Basal.

Before

```
<li *ngFor="let item of items;">{{ item }}</li>
```

After

```
// in the template
<li *ngFor="let item of items; trackBy: trackByFn">{{ item }}</li>
// in the component
trackByFn(index, item) {
  return item.id; // unique id corresponding to the item
}
```

2) const vs let

When declaring variables, use const when the value is not going to be reassigned.

Why?
Using let and const where appropriate makes the intention of the declarations clearer. It will also help in identifying issues when a value is reassigned to a constant accidentally by throwing a compile time error. It also helps improve the readability of the code.
Before

```
let car = 'ludicrous car';
let myCar = `My ${car}`;
let yourCar = `Your ${car};
if (iHaveMoreThanOneCar) {
  myCar = `${myCar}s`;
}
if (youHaveMoreThanOneCar) {
  yourCar = `${youCar}s`;
}
```

After

```
// the value of car is not reassigned, so we can make it a const
const car = 'ludicrous car';
let myCar = `My ${car}`;
let yourCar = `Your ${car};
if (iHaveMoreThanOneCar) {
  myCar = `${myCar}s`;
}
if (youHaveMoreThanOneCar) {
  yourCar = `${youCar}s`;
}
```

3) Pipeable operators
Use pipeable operators when using RxJs operators.
Why?
Pipeable operators are tree-shakeable meaning only the code we need to execute will be included when they are imported.
This also makes it easy to identify unused operators in the files.
Note: This needs Angular version 5.5+.

Before

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';
iAmAnObservable
  .map(value => value.item)
  .take(1);
```

After

```
import { map, take } from 'rxjs/operators';
iAmAnObservable
  .pipe(
    map(value => value.item),
    take(1)
  );
```

4) Isolate API hacks

Not all APIs are bullet proof — sometimes we need to add some logic in the code to make up for bugs in the APIs. Instead of having the hacks in components where they are needed, it is better to isolate them in one place — like in a service and use the service from the component.

Why?

This helps keep the hacks “closer to the API”, so as close to where the network request is made as possible. This way, less of your code is dealing with the un-hacked code. Also, it is one place where all the hacks live and it is easier to find them. When fixing the bugs in the APIs, it is easier to look for them in one file rather than looking for the hacks that could be spread across the codebase.

You can also create custom tags like API_FIX similar to TODO and tag the fixes with it so it is easier to find.

5) Subscribe in template

Avoid subscribing to observables from components and instead subscribe to the observables from the template.

Why?

async pipes unsubscribe themselves automatically and it makes the code simpler by eliminating the need to manually manage subscriptions. It also reduces the risk of accidentally forgetting to unsubscribe a subscription in the component, which

would cause a memory leak. This risk can also be mitigated by using a lint rule to detect unsubscribed observables.
This also stops components from being stateful and introducing bugs where the data gets mutated outside of the subscription.
Before

```
// // template
<p>{{ textToDisplay }}</p>
// component
iAmAnObservable
   .pipe(
     map(value => value.item),
     takeUntil(this._destroyed$)
   )
   .subscribe(item => this.textToDisplay = item);
```

After

```
// template
<p>{{ textToDisplay$ | async }}</p>
// component
this.textToDisplay$ = iAmAnObservable
   .pipe(
     map(value => value.item)
   );
```

6) Clean up subscriptions
When subscribing to observables, always make sure you unsubscribe from them appropriately by using operators like take, takeUntil, etc.
Why?
Failing to unsubscribe from observables will lead to unwanted memory leaks as the observable stream is left open, potentially even after a component has been destroyed / the user has navigated to another page.
Even better, make a lint rule for detecting observables that are not unsubscribed.
Before

```
iAmAnObservable
   .pipe(
```

```
      map(value => value.item)
    )
   .subscribe(item => this.textToDisplay = item);
```

After

Using takeUntil when you want to listen to the changes until another observable emits a value:

```
private _destroyed$ = new Subject();
public ngOnInit (): void {
   iAmAnObservable
   .pipe(
      map(value => value.item)
     // We want to listen to iAmAnObservable until the component is destroyed,
      takeUntil(this._destroyed$)
    )
   .subscribe(item => this.textToDisplay = item);
}
public ngOnDestroy (): void {
   this._destroyed$.next();
   this._destroyed$.complete();
}
```

Using a private subject like this is a pattern to manage unsubscribing many observables in the component.

Using take when you want only the first value emitted by the observable:

```
iAmAnObservable
   .pipe(
      map(value => value.item),
      take(1),
      takeUntil(this._destroyed$)
   )
   .subscribe(item => this.textToDisplay = item);
```

Note the usage of takeUntil with take here. This is to avoid memory leaks caused when the subscription hasn't received a value before the component got destroyed. Without takeUntil here, the subscription would still hang around until

it gets the first value, but since the component has already gotten destroyed, it will never get a value — leading to a memory leak.

7) Use appropriate operators

When using flattening operators with your observables, use the appropriate operator for the situation.

switchMap: when you want to ignore the previous emissions when there is a new emission

mergeMap: when you want to concurrently handle all the emissions

concatMap: when you want to handle the emissions one after the other as they are emitted

exhaustMap: when you want to cancel all the new emissions while processing a previous emisssion

For a more detailed explanation on this, please refer to this article by Nicholas Jamieson.

Why?

Using a single operator when possible instead of chaining together multiple other operators to achieve the same effect can cause less code to be shipped to the user. Using the wrong operators can lead to unwanted behaviour, as different operators handle observables in different ways.

8) Lazy load

When possible, try to lazy load the modules in your Angular application. Lazy loading is when you load something only when it is used, for example, loading a component only when it is to be seen.

Why?

This will reduce the size of the application to be loaded and can improve the application boot time by not loading the modules that are not used.

Before

```
// app.routing.ts
{ path: 'not-lazy-loaded', component: NotLazyLoadedComponent }
```

After

```
// app.routing.ts
{
  path: 'lazy-load',
```

```
  loadChildren: 'lazy-load.module#LazyLoadModule'
}
// lazy-load.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';
import { LazyLoadComponent }   from './lazy-load.component';
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
        {
            path: '',
            component: LazyLoadComponent
        }
    ])
  ],
  declarations: [
    LazyLoadComponent
  ]
})
export class LazyModule {}
```

9) Avoid having subscriptions inside subscriptions

Sometimes you may want values from more than one observable to perform an action. In this case, avoid subscribing to one observable in the subscribe block of another observable. Instead, use appropriate chaining operators. Chaining operators run on observables from the operator before them. Some chaining operators are: withLatestFrom, combineLatest, etc.

Before

```
firstObservable$.pipe(
  take(1)
)
.subscribe(firstValue => {
```

```
  secondObservable$.pipe(
    take(1)
  )
  .subscribe(secondValue => {
    console.log(`Combined values are: ${firstValue} & ${secondValue}`);
  });
});
```

After

```
firstObservable$.pipe(
  withLatestFrom(secondObservable$),
  first()
)
.subscribe(([firstValue, secondValue]) => {
  console.log(`Combined values are: ${firstValue} & ${secondValue}`);
});
```

Why?

Code smell/readability/complexity: Not using RxJs to its full extent, suggests developer is not familiar with the RxJs API surface area.

Performance: If the observables are cold, it will subscribe to firstObservable, wait for it to complete, THEN start the second observable's work. If these were network requests it would show as synchronous/waterfall.

10) Avoid any; type everything;

Always declare variables or constants with a type other than any.

Why?

When declaring variables or constants in Typescript without a typing, the typing of the variable/constant will be deduced by the value that gets assigned to it. This will cause unintended problems. One classic example is:

```
const x = 1;
const y = 'a';
const z = x + y;
console.log(`Value of z is: ${z}`
// Output
Value of z is 1a
```

This can cause unwanted problems when you expect y to be a number too. These problems can be avoided by typing the variables appropriately.

```
const x: number = 1;
const y: number = 'a';
const z: number = x + y;
// This will give a compile error saying:
```

doneeeeeeeeeeeeeee

```
Type '"a"' is not assignable to type 'number'.
const y:number
```

This way, we can avoid bugs caused by missing types.

Another advantage of having good typings in your application is that it makes refactoring easier and safer.

Consider this example:

```
public ngOnInit (): void {
   let myFlashObject = {
      name: 'My cool name',
      age: 'My cool age',
      loc: 'My cool location'
   }
   this.processObject(myFlashObject);
}
public processObject(myObject: any): void {
   console.log(`Name: ${myObject.name}`);
   console.log(`Age: ${myObject.age}`);
   console.log(`Location: ${myObject.loc}`);
}
// Output
Name: My cool name
Age: My cool age
Location: My cool location
```

Let us say, we want to rename the property loc to location in myFlashObject:

```
public ngOnInit (): void {
   let myFlashObject = {
```

```
      name: 'My cool name',
      age: 'My cool age',
      location: 'My cool location'
   }
   this.processObject(myFlashObject);
}
public processObject(myObject: any): void {
   console.log(`Name: ${myObject.name}`);
   console.log(`Age: ${myObject.age}`);
   console.log(`Location: ${myObject.loc}`);
}
// Output
Name: My cool name
Age: My cool age
Location: undefined
```

If we do not have a typing on myFlashObject, it thinks that the property loc on myFlashObject is just undefined rather than that it is not a valid property.

If we had a typing for myFlashObject, we would get a nice compile time error as shown below:

```
type FlashObject = {
   name: string,
   age: string,
   location: string
}
public ngOnInit (): void {
   let myFlashObject: FlashObject = {
      name: 'My cool name',
      age: 'My cool age',
      // Compilation error
      Type '{ name: string; age: string; loc: string; }' is not assignable to type
'FlashObjectType'.
      Object literal may only specify known properties, and 'loc' does not exist in
type 'FlashObjectType'.
```

```
      loc: 'My cool location'
   }
   this.processObject(myFlashObject);
}
public processObject(myObject: FlashObject): void {
   console.log(`Name: ${myObject.name}`);
   console.log(`Age: ${myObject.age}`)
   // Compilation error
   Property 'loc' does not exist on type 'FlashObjectType'.
   console.log(`Location: ${myObject.loc}`);
}
```

If you are starting a new project, it is worth setting strict:true in the tsconfig.json file to enable all strict type checking options.

11) Make use of lint rules

tslint has various options built in already like no-any, no-magic-numbers, no-console, etc that you can configure in your tslint.json to enforce certain rules in your code base.

Why?

Having lint rules in place means that you will get a nice error when you are doing something that you should not be. This will enforce consistency in your application and readability. Please refer here for more rules that you can configure.

Some lint rules even come with fixes to resolve the lint error. If you want to configure your own custom lint rule, you can do that too. Please refer to this article by Craig Spence on how to write your own custom lint rules using TSQuery.

Before

```
public ngOnInit (): void {
   console.log('I am a naughty console log message');
   console.warn('I am a naughty console warning message');
   console.error('I am a naughty console error message');
}
// Output
```

No errors, prints the below on console window:

```
I am a naughty console message
I am a naughty console warning message
I am a naughty console error message
After
// tslint.json
{
   "rules": {
      .......
      "no-console": [
          true,
          "log",    // no console.log allowed
          "warn"    // no console.warn allowed
      ]
  }
}
// ..component.ts
public ngOnInit (): void {
   console.log('I am a naughty console log message');
   console.warn('I am a naughty console warning message');
   console.error('I am a naughty console error message');
}
// Output
Lint errors for console.log and console.warn statements and no error for
console.error as it is not mentioned in the config
Calls to 'console.log' are not allowed.
Calls to 'console.warn' are not allowed.
```

12) Small reusable components

Extract the pieces that can be reused in a component and make it a new one. Make the component as dumb as possible, as this will make it work in more scenarios. Making a component dumb means that the component does not have any special logic in it and operates purely based on the inputs and outputs provided to it.

As a general rule, the last child in the component tree will be the dumbest of all.

Why?
Reusable components reduce duplication of code therefore making it easier to maintain and make changes.
Dumb components are simpler, so they are less likely to have bugs. Dumb components make you think harder about the public component API, and help sniff out mixed concerns.
13) Components should only deal with display logic
Avoid having any logic other than display logic in your component whenever you can and make the component deal only with the display logic.
Why?
Components are designed for presentational purposes and control what the view should do. Any business logic should be extracted into its own methods/services where appropriate, separating business logic from view logic.
Business logic is usually easier to unit test when extracted out to a service, and can be reused by any other components that need the same business logic applied.
14) Avoid long methods
Long methods generally indicate that they are doing too many things. Try to use the Single Responsibility Principle. The method itself as a whole might be doing one thing, but inside it, there are a few other operations that could be happening. We can extract those methods into their own method and make them do one thing each and use them instead.
Why?
Long methods are hard to read, understand and maintain. They are also prone to bugs, as changing one thing might affect a lot of other things in that method. They also make refactoring (which is a key thing in any application) difficult.
This is sometimes measured as “cyclomatic complexity”. There are also some TSLint rules to detect cyclomatic/cognitive complexity, which you could use in your project to avoid bugs and detect code smells and maintainability issues.
15) DRY
Do not Repeat Yourself. Make sure you do not have the same code copied into different places in the codebase. Extract the repeating code and use it in place of the repeated code.

Why?
Having the same code in multiple places means that if we want to make a change to the logic in that code, we have to do it in multiple places. This makes it difficult to maintain and also is prone to bugs where we could miss updating it in all occurrences. It takes longer to make changes to the logic and testing it is a lengthy process as well.
In those cases, extract the repeating code and use it instead. This means only one place to change and one thing to test. Having less duplicate code shipped to users means the application will be faster.
16) Add caching mechanisms
When making API calls, responses from some of them do not change often. In those cases, you can add a caching mechanism and store the value from the API. When another request to the same API is made, check if there is a value for it in the cache and if so, use it. Otherwise, make the API call and cache the result.
If the values change but not frequently, you can introduce a cache time where you can check when it was last cached and decide whether or not to call the API.
Why?
Having a caching mechanism means avoiding unwanted API calls. By only making the API calls when required and avoiding duplication, the speed of the application improves as we do not have to wait for the network. It also means we do not download the same information over and over again.
17) Avoid logic in templates
If you have any sort of logic in your templates, even if it is a simple && clause, it is good to extract it out into its component.
Why?
Having logic in the template means that it is not possible to unit test it and therefore it is more prone to bugs when changing template code.
Before

```
// template
<p *ngIf="role==='developer'"> Status: Developer </p>
// component
public ngOnInit (): void {
   this.role = 'developer';
```

```
}
```

After

```
// template
<p *ngIf="showDeveloperStatus"> Status: Developer </p>
// component
public ngOnInit (): void {
   this.role = 'developer';
   this.showDeveloperStatus = true;
}
```

18) Strings should be safe

If you have a variable of type string that can have only a set of values, instead of declaring it as a string type, you can declare the list of possible values as the type.

Why?

By declaring the type of the variable appropriately, we can avoid bugs while writing the code during compile time rather than during runtime.

Before

```
private myStringValue: string;
if (itShouldHaveFirstValue) {
  myStringValue = 'First';
} else {
  myStringValue = 'Second'
}
```

After

```
private myStringValue: 'First' | 'Second';
if (itShouldHaveFirstValue) {
  myStringValue = 'First';
} else {
  myStringValue = 'Other'
}
// This will give the below error
Type '"Other"' is not assignable to type '"First" | "Second"'
(property) AppComponent.myValue: "First" | "Second"
```

Bigger picture

State Management

Consider using @ngrx/store for maintaining the state of your application and @ngrx/effects as the side effect model for store. State changes are described by the actions and the changes are done by pure functions called reducers.

Why?

@ngrx/store isolates all state related logic in one place and makes it consistent across the application. It also has memoization mechanism in place when accessing the information in the store leading to a more performant application. @ngrx/store combined with the change detection strategy of Angular leads to a faster application.

Immutable state

When using @ngrx/store, consider using ngrx-store-freeze to make the state immutable. ngrx-store-freeze prevents the state from being mutated by throwing an exception. This avoids accidental mutation of the state leading to unwanted consequences.

Why?

Mutating state in components leads to the app behaving inconsistently depending on the order components are loaded. It breaks the mental model of the redux pattern. Changes can end up overridden if the store state changes and re-emits. Separation of concerns — components are view layer, they should not know how to change state.

Jest

Jest is Facebook's unit testing framework for JavaScript. It makes unit testing faster by parallelising test runs across the code base. With its watch mode, only the tests related to the changes made are run, which makes the feedback loop for testing way shorter. Jest also provides code coverage of the tests and is supported on VS Code and Webstorm.

You could use a preset for Jest that will do most of the heavy lifting for you when setting up Jest in your project.

Karma

Karma is a test runner developed by AngularJS team. It requires a real browser/DOM to run the tests. It can also run on different browsers. Jest doesn't need chrome headless/phantomjs to run the tests and it runs in pure Node.

Universal

If you haven’t made your app a Universal app, now is a good time to do it. Angular Universal lets you run your Angular application on the server and does server-side rendering (SSR) which serves up static pre-rendered html pages. This makes the app super fast as it shows content on the screen almost instantly, without having to wait for JS bundles to load and parse, or for Angular to bootstrap.

It is also SEO friendly, as Angular Universal generates static content and makes it easier for the web crawlers to index the application and make it searchable without executing JavaScript.

Why?

Universal improves the performance of your application drastically. We recently updated our application to do server side rendering and the site load time went from several seconds to tens of milliseconds!!

It also allows your site to correctly show up in social media preview snippets. The first meaningful paint is really fast and makes content visible to the users without any unwanted delays.

Conclusion

Building applications is a constant journey, and there’s always room to improve things. This list of optimisations is a good place to start, and applying these patterns consistently will make your team happy. Your users will also love you for the nice experience with your less buggy and performant application.

Retry failed observable

Where the catchError operator provides a simple path of recovery, the retry operator lets you retry a failed request.

Use the retry operator before the catchError operator. It resubscribes to the original source observable, which can then re-run the full sequence of actions that resulted in the error. If this includes an HTTP request, it will retry that HTTP request.

The following converts the previous example to retry the request before catching the error:

retry operator

```
import { ajax } from 'rxjs/ajax';
import { map, retry, catchError } from 'rxjs/operators';
```

```
const apiData = ajax('/api/data').pipe(
  retry(3), // Retry up to 3 times before failing
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
  catchError(err => of([]))
);
apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

Lifecycle Hooks

A component has a lifecycle managed by Angular.

Angular creates and renders components along with their children, checks when their data-bound properties change, and destroys them before removing them from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks.

Lifecycle sequence

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

Hook Purpose and Timing

ngOnChanges()

Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Called before ngOnInit() and whenever one or more data-bound input properties change.

ngOnInit()

Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.
Called once, after the first ngOnChanges().
ngDoCheck()
Detect and act upon changes that Angular can't or won't detect on its own.
Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
ngAfterContentInit()
Respond after Angular projects external content into the component's view / the view that a directive is in.
Called once after the first ngDoCheck().
ngAfterContentChecked()
Respond after Angular checks the content projected into the directive/component.
Called after the ngAfterContentInit() and every subsequent ngDoCheck().
ngAfterViewInit()
Respond after Angular initializes the component's views and child views / the view that a directive is in.
Called once after the first ngAfterContentChecked().
ngAfterViewChecked()
Respond after Angular checks the component's views and child views / the view that a directive is in.
Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().
ngOnDestroy()
Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.
Called just before Angular destroys the directive/component.
Interfaces are optional (technically)
The interfaces are optional for JavaScript and Typescript developers from a purely technical perspective. The JavaScript language doesn't have interfaces. Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.

Fortunately, they aren't necessary. You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselves. Angular instead inspects directive and component classes and calls the hook methods if they are defined. Angular finds and calls methods like ngOnInit(), with or without the interfaces.

Nonetheless, it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

observable and promise check

In order to show how subscribing works, we need to create a new observable. There is a constructor that you use to create new instances, but for illustration, we can use some methods from the RxJS library that create simple observables of frequently used types:

of(...items)—Returns an Observable instance that synchronously delivers the values provided as arguments.

from(iterable)—Converts its argument to an Observable instance. This method is commonly used to convert an array to an observable.

Observables provide support for passing messages between publishers and subscribers in your application. Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values.

Observables are declarative—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe. An observable can deliver multiple values of any type—literals, messages, or events, depending on the context. The API for receiving values is the same whether the values are delivered synchronously or asynchronously. Because setup and teardown logic are both handled by the observable, your application code only needs to worry about subscribing to consume values, and when done, unsubscribing. Whether the stream was keystrokes, an HTTP response, or an interval timer, the interface for listening to values and stopping listening is the same.

Because of these advantages, observables are used extensively within Angular, and are recommended for app development as well.

To execute the observable you have created and begin receiving notifications, you call its subscribe() method, passing an observer. This is a JavaScript object that defines the handlers for the notifications you receive. The subscribe() call returns a Subscription object that has an unsubscribe() method, which you call to stop receiving notifications.

hat actually the difference is?

Promise emits a single value while Observable emits multiple values. So, while handling a HTTP request, Promise can manage a single response for the same request, but what if there are multiple responses to the same request, then we have to use Observable. Yes, Observable can handle multiple responses for the same request.

Let’s implement this with an example.

Promise

```
const promise = new Promise((data) =>
{ data(1);
  data(2);
  data(3); })
.then(element => console.log(‘Promise ‘ + element));
```

Output

```
Promise 1
```

Observable

```
const observable = new Observable((data) => {
data.next(1);
data.next(2);
data.next(3);
}).subscribe(element => console.log('Observable ' + element));
```

Output

```
Observable 1
Observable 2
Observable 3
```

So, in the above code snippet, I have created promise and observable of Promise and Observable type respectively. But, promise returns the very first value and

ignore the remaining values whereas Observable return all the value and print 1, 2, 3 in the console.

Promise is not lazy while Observable is lazy. Observable is lazy in nature and do not return any value until we subscribe.

home.component.ts (Observable)

```
1
getMenu() {
2
this.homeService.getFoodItem();
3
}
4
}
```

In above example we are not subscribing the observable, so we do not receive the data and even there would be a no network call for this service.

home.component.ts (Observable)

```
1
getMenu() {
2
this.homeService.getFoodItem().subscribe((data  => {
3
this.foodItem = data;
4
}),
5
error => console.log(error));
6
}
7
}
```

Here, we have subscribed our Observable, so it will simply return the data. But Promise returns the value regardless of then() method.

home.component.ts (Promise)

```
1
getMenu() {
2
this.homeService.getFoodItem()
3
.then((data) => {
4
this.foodItem = data;
5
});
6
}
```

Observable is cancellable in nature by invoking unsubscribe() method, but Promise is not cancellable in nature.

Hope this is helpful and give you a basic understanding of how Promise differs from Observable. Please feel free to provide your suggestions

The RxJS library

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change (Wikipedia). RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code. See (RxJS Docs).

Observable creation functions

RxJS offers a number of functions that can be used to create new observables. These functions can simplify the process of creating observables from things such as events, timers, promises, and so on.

Create an observable from a promise

content_copy

```
import { from } from 'rxjs';
// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
```

```
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
import { interval } from 'rxjs';
// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));

  Create an observable from an event

  import { fromEvent } from 'rxjs';
const el = document.getElementById('my-element');
// Create an Observable that will publish mouse movements
const mouseMoves = fromEvent(el, 'mousemove');
// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);
  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
Create an observable that creates an AJAX request
import { ajax } from 'rxjs/ajax';
// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

Operators are functions that build on the observables foundation to enable sophisticated manipulation of collections. For example, RxJS defines operators such as map(), filter(), concat(), and flatMap().

Operators take configuration options, and they return a function that takes a source observable. When executing this returned function, the operator observes the source observable's emitted values, transforms them, and returns a new observable of those transformed values. Here is a simple example:

Map operator

content_copy

```
import { map } from 'rxjs/operators';
const nums = of(1, 2, 3);
const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);
squaredNums.subscribe(x => console.log(x));
```

You can use pipes to link operators together. Pipes let you combine multiple functions into a single function. The pipe() function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

A set of operators applied to an observable is a recipe—that is, a set of instructions for producing the values you're interested in. By itself, the recipe doesn't do anything. You need to call subscribe() to produce a result through the recipe

```
import { filter, map } from 'rxjs/operators';
const nums = of(1, 2, 3, 4, 5);
// Create a function that accepts an Observable.
const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);
// Create an Observable that will run the filter and map functions
const squareOdd = squareOddVals(nums);
// Subscribe to run the combined functions
squareOdd.subscribe(x => console.log(x));
```

The pipe() function is also a method on the RxJS Observable, so you use this shorter form to define the same operation:

```
import { filter, map } from 'rxjs/operators';
const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );
// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

Common operators

RxJS provides many operators, but only a handful are used frequently. For a list of operators and usage samples, visit the RxJS API Documentation.

Error handling

In addition to the error() handler that you provide on subscription, RxJS provides the catchError operator that lets you handle known errors in the observable recipe.

For instance, suppose you have an observable that makes an API request and maps to the response from the server. If the server returns an error or the value doesn’t exist, an error is produced. If you catch this error and supply a default value, your stream continues to process values rather than erroring out.

Here's an example of using the catchError operator to do this:

```
import { ajax } from 'rxjs/ajax';
import { map, catchError } from 'rxjs/operators';
// Return "response" from the API. If an error happens,
// return an empty array.
const apiData = ajax('/api/data').pipe(
  map(res => {
    if (!res.response) {
      throw new Error('Value expected!');
    }
    return res.response;
  }),
```

```
  catchError(err => of([]))
);
apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

Naming conventions for observables

Because Angular applications are mostly written in TypeScript, you will typically know when a variable is an observable. Although the Angular framework does not enforce a naming convention for observables, you will often see observables named with a trailing “$” sign.

This can be useful when scanning through code and looking for observable values. Also, if you want a property to store the most recent value from an observable, it can be convenient to simply use the same name with or without the “$”.

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {
  stopwatchValue: number;
  stopwatchValue$: Observable<number>;
  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

directive

At the core, a directive is a function that executes whenever the Angular compiler finds it in the DOM. Angular directives are used to extend the power of the HTML by giving it new syntax. Each directive has a name — either one from the Angular

predefined like ng-repeat, or a custom one which can be called anything. And each directive determines where it can be used: in an element, attribute, class or comment.

Components

As we saw earlier, components are just directives with templates. Under the hood, they use the directive API and give us a cleaner way to define them.

The other two directive types don't have templates. Instead, they're specifically tailored to DOM manipulation.

Attribute directives

Attribute directives manipulate the DOM by changing its behavior and appearance.

We use attribute directives to apply conditional style to elements, show or hide elements or dynamically change the behavior of a component according to a changing property.

like ngclass

Structural directives

These are specifically tailored to create and destroy DOM elements.

Some attribute directives — like hidden, which shows or hides an element — basically maintain the DOM as it is. But the structural Angular directives are much less DOM friendly, as they add or completely remove elements from the DOM. So, when using these, we have to be extra careful, since we're actually changing the HTML structure.

```
import {Directive, ElementRef} from '@angular/core';
@Directive({
  selector:'[my-error]'
})
export class MyErrorDirective{
  constructor(elr:ElementRef){
    elr.nativeElement.style.background='red';
  }
}
```

After importing the Directive from @angular/core we can then use it. First, we need a selector, which gives a name to the directive. In this case, we call it my-error.

Best practice dictates that we always use a prefix when naming our Angular directives. This way, we're sure to avoid conflicts with any standard HTML attributes. We also shouldn't use the ng prefix. That one's used by Angular, and we don't want to confuse our custom created Angular directives with Angular predefined ones. In this example, our prefix is my-.

We then created a class, MyErrorDirective. To access any element of our DOM, we need to use ElementRef. Since it also belongs to the @angular/core package, it's a simple matter of importing it together with the Directive and using it.

We then added the code to actually highlight the constructor of our class.

To be able to use this newly created directive, we need to add it to the declarations on the app.module.ts file:

```
import { MyErrorDirective } from './app.myerrordirective';
```

Creating a structural directive

In the previous section, we saw how to create an attribute directive using Angular. The approach for creating a structural behavior is exactly the same. We create a new file with the code for our directive, then we add it to the declarations, and finally, we use it in our component.

For our structural directive, we'll implement a copy of the ngIf directive. This way, we'll not only be implementing a directive, but also taking a look at how Angular directives handle things behind the scenes.

Let's start with our app.mycustomifdirective.ts file:

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[myCustomIf]'
})
export class MyCustomIfDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }
  @Input() set myCustomIf(condition: boolean) {
```

```
      if (condition) {
         this.viewContainer.createEmbeddedView(this.templateRef);
      } else {
         this.viewContainer.clear();
      }
   }
}
```

As we can see, we're using a couple of different imports for this one, mainly: Input, TemplateRef and ViewContainerRef. The Input decorator is used to pass data to the component. The TemplateRef one is used to instantiate embedded views. An embedded view represents a part of a layout to be rendered, and it's linked to a template. Finally, the ViewContainerRef is a container where one or more Views can be attached. Together, these components work as follow

for component

```
import { Component, OnInit, Input } from '@angular/core';
import { Hero } from '../hero';
@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: ['./hero-detail.component.css']
})
```

Architecture overview

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.

Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.

The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Modules

Angular NgModules differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

Feature Modules

Feature modules are NgModules for the purpose of organizing code.

For the final sample app with a feature module that this page describes, see the live example / download example.

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

Feature modules vs. root modules

A feature module is an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

How to make a feature module

Assuming you already have an app that you created with the Angular CLI, create a feature module using the CLI by entering the following command in the root project directory. Replace CustomerDashboard with the name of your module. You can omit the "Module" suffix from the name because the CLI appends it:

```
ng generate module CustomerDashboard
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

Importing a feature module

To incorporate the feature module into your app, you have to let the root module, app.module.ts, know about it. Notice the CustomerDashboardModule

export at the bottom of customer-dashboard.module.ts. This exposes it so that other modules can get to it. To import it into the AppModule, add it to the imports in app.module.ts and to the imports array:

Types of Feature Modules

There are five general categories of feature modules which tend to fall into the following groups:

Domain feature modules.

Routed feature modules.

Routing modules.

Service feature modules.

Widget feature modules.

Feature Module Guidelines

Domain Domain feature modules deliver a user experience dedicated to a particular application domain like editing a customer or placing an order.

They typically have a top component that acts as the feature root and private, supporting sub-components descend from it.

Domain feature modules consist mostly of declarations. Only the top component is exported.

Domain feature modules rarely have providers. When they do, the lifetime of the provided services should be the same as the lifetime of the module.

Domain feature modules are typically imported exactly once by a larger feature module.

They might be imported by the root AppModule of a small application that lacks routing.

Routed Routed feature modules are domain feature modules whose top components are the targets of router navigation routes.

All lazy-loaded modules are routed feature modules by definition.

Routed feature modules don’t export anything because their components never appear in the template of an external component.

A lazy-loaded routed feature module should not be imported by any module. Doing so would trigger an eager load, defeating the purpose of lazy loading.That means you won’t see them mentioned among the AppModule imports. An eager

loaded routed feature module must be imported by another module so that the compiler learns about its components.
Routed feature modules rarely have providers for reasons explained in Lazy Loading Feature Modules. When they do, the lifetime of the provided services should be the same as the lifetime of the module. Don't provide application-wide singleton services in a routed feature module or in a module that the routed module imports.
Routing
A routing module provides routing configuration for another module and separates routing concerns from its companion module.
A routing module typically does the following:
Defines routes.
Adds router configuration to the module's imports.
Adds guard and resolver service providers to the module's providers.
The name of the routing module should parallel the name of its companion module, using the suffix "Routing". For example, FooModule in foo.module.ts has a routing module named FooRoutingModule in foo-routing.module.ts. If the companion module is the root AppModule, the AppRoutingModule adds router configuration to its imports with RouterModule.forRoot(routes). All other routing modules are children that import RouterModule.forChild(routes).
A routing module re-exports the RouterModule as a convenience so that components of the companion module have access to router directives such as RouterLink and RouterOutlet.
A routing module does not have its own declarations. Components, directives, and pipes are the responsibility of the feature module, not the routing module.
A routing module should only be imported by its companion module.
Service
Service modules provide utility services such as data access and messaging. Ideally, they consist entirely of providers and have no declarations. Angular's HttpClientModule is a good example of a service module.
The root AppModule is the only module that should import service modules.
Widget

A widget module makes components, directives, and pipes available to external modules. Many third-party UI component libraries are widget modules.
A widget module should consist entirely of declarations, most of them exported.
A widget module should rarely have providers.
Import widget modules in any module whose component templates need the widgets.
How is change detection implemented?
Angular can detect when component data changes, and then automatically re-render the view to reflect that change. But how can it do so after such a low-level event like the click of a button, that can happen anywhere on the page?
How does this low-level runtime patching work?
This low-level patching of browser APIs is done by a library shipped with Angular called Zone.js. It's important to have an idea of what a zone is.
A zone is nothing more than an execution context that survives multiple Javascript VM execution turns. It's a generic mechanism which we can use to add extra functionality to the browser. Angular uses Zones internally to trigger change detection, but another possible use would be to do application profiling, or keeping track of long stack traces that run across multiple VM turns.
yes, Zone and NgZone is used to automatically trigger change detection as a result of async operations. But since change detection is a separate mechanism it can successfully work without Zone and NgZone. In the first chapter I will show how Angular can be used without zone.js. Second part of the article explains how Angular and zone.js interact together through NgZone. In the end I'll also show why automatic change detection sometimes doesn't work with 3rd party libraries like Google API Client Library (gapi).
it's expected since NgZone is not used and hence change detection is not triggered automatically. Yet it still works fine if we trigger it manually. This can be done by injecting ApplicationRef and triggering tick method to start change detection:

```
export class AppComponent  {
  name = 'Angular 4';
  constructor(app: ApplicationRef) {
    setTimeout(()=>{
```

```
        this.name = 'updated';
        app.tick();
      }, 1000);
  }
```

To summarize, the point of the above demonstration is to show you that zone.js and NgZone in particular are not part of change detection implementation. It's a very convenient mechanism to trigger change detection automatically by calling app.tick() instead of doing it manually at certain points. We will see in a minute what those points.

How NgZone uses Zones

In my previous article on Zone (zone.js) I described in depth the inner working and API that Zone provides. There I explained the core concepts of forking a zone and running a task in a particular zone. I'll be referring to those concepts here.
I also demonstrated two capabilities that Zone provides — context propagation and outstanding asynchronous tasks tracking. Angular implements NgZone class that relies heavily on the tasks tracking mechanism.
NgZone is just a wrapper around a forked child zone:
bootstrap in angular
Method 1: Using Angular CLI (npm install). From the command line interface install bootstrap and references it in angular.json
npm install bootstrap --save
https://angular.io/guide/bootstrapping
An NgModule describes how the application parts fit together. Every application has at least one Angular module, the root module that you bootstrap to launch the application. By convention, it is usually called AppModule.
If you use the Angular CLI to generate an app, the default AppModule is as follows:
The @NgModule decorator identifies AppModule as an NgModule class.
@NgModule takes a metadata object that tells Angular how to compile and launch the application.
declarations—this application's lone component.

imports—import BrowserModule to have browser specific services such as DOM rendering, sanitization, and location.
providers—the service providers.
bootstrap—the root component that Angular creates and inserts into the index.html host web page.
The default application created by the Angular CLI only has one component, AppComponent, so it is in both the declarations and the bootstrap arrays.
The declarations array
The module's declarations array tells Angular which components belong to that module. As you create more components, add them to declarations.
You must declare every component in exactly one NgModule class. If you use a component without declaring it, Angular returns an error message.
The declarations array only takes declarables. Declarables are components, directives and pipes. All of a module's declarables must be in the declarations array. Declarables must belong to exactly one module. The compiler emits an error if you try to declare the same class in more than one module.
These declared classes are visible within the module but invisible to components in a different module unless they are exported from this module and the other module imports this one.
An example of what goes into a declarations array follows:
content_copy

```
declarations: [
  YourComponent,
  YourPipe,
  YourDirective
],
```

A declarable can only belong to one module, so only declare it in one @NgModule. When you need it elsewhere, import the module that has the declarable you need in it.
Only @NgModule references go in the imports array.
https://angular.io/guide/bootstrapping
An Attribute directive changes the appearance or behavior of a DOM element.
ngclass

There are three kinds of directives in Angular:
Components—directives with a template. like popup
Structural directives—change the DOM layout by adding and removing DOM elements. if else
Attribute directives—change the appearance or behavior of an element, component, or another directive. ngclass
ng generate directive highlight
what is diff b/w directive and component
what is metadata with eg

```
import { Directive, ElementRef } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

```
<p appHighlight>Highlight me!</p>
```

```
import { Directive, ElementRef, HostListener } from '@angular/core';
```

Then add two eventhandlers that respond when the mouse enters or leaves, each adorned by the HostListener decorator.

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}
@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}
private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

The @HostListener decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the <p> in this case.

src/app/highlight.directive.ts (constructor)
content_copy
constructor(private el: ElementRef) { }
Pass values into the directive with an @Input data binding
import { Directive, ElementRef, HostListener, Input } from '@angular/core';
@Input() highlightColor: string;
<p appHighlight [highlightColor]="'orange'">Highlighted in orange</p>
<p appHighlight [highlightColor]="color">
what is use of main.ts file
https://angular.io/guide/architecture
main.ts is the entry point of your application , compiles the application with just-in-time and bootstraps the application .Angular can be bootstrapped in multiple environments we need to import a module specific to the environment. in which angular looks for which module would run first.
// The browser platform with a compiler
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
// The app module
import { AppModule } from './app/app.module';
// Compile and launch the module
platformBrowserDynamic().bootstrapModule(AppModule);
APP.MODULE.TS
This is root module that tells Angular how to assemble the application . Every Angular app has a root module class
@NgModule — takes a metadata object that tells Angular how to compile and launch the application.
Imports — the BrowserModule that this and every application needs to run in a browser.
Declarations — the application's component.
Bootstrap — this is the root component tells which component to run first.
enableProdMode - Disable Angular's development mode, which turns off assertions and other checks within the framework.
platformBrowserDynamic - To boostrap your app for browsers

AppModule - The root module which inform Angular about various files and codes.
environment - Environment Variables in Angular read this
app.module.ts:
Component
It is also a type of directive with template,styles and logic part which is most famous type of directive among all in Angular2. In this type of directive you can use other directives whether it is custom or builtin in the @Component annotation like following:
The Complete Guide to Angular Performance Tuning
https://christianlydemann.com/the-complete-guide-to-angular-performance-tuning/
Improving change detection
Change detection can be the most performance heavy in Angular apps and therefore it is necessary to have some awareness of how to render the templates most effectively, so you are only rerendering a component if it has new changes to show.
OnPush change detection
The default change detection behavior for components is to re-render every time an asynchronous event has happened in the app such as click, XMLHttpRequest, setTimout. This can become a problem because this will cause many unnecessary renderings of the templates, that may not have been changed
OnPush change detection fixes this by only re-rendering a template if either:
One of its input properties has gotten a new reference
An event from the component or one of its children eg. click on a button in the component
Explicit run of change detection
To apply this strategy you just need to set the change-detection strategy in the component's decorator:

```
@Component({
  selector: 'app-todo-list',
  templateUrl: './todo-list.component.html',
  styleUrls: ['./todo-list.component.scss'],
```

```
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class TodoListComponent implements OnInit {
```

Design for immutability (unchanging over time)

To leverage this method you need to make sure, that all state changes are happening immutably because we need a new reference provided to a component's input to trigger change detection with onPush. If you are using Redux for state management, then you would naturally get a new instance every time the state changes, which will trigger change detection for onPush components when provided to a component's inputs. With this approach you want to have container components, that is responsible for getting the data from the store and presentation component, which will only interact with other components using input and output.

presentation component <-> container component <-> redux store

The easiest way to provide the store data to the template is using the async pipe. This will look like having the data outside of an observable and will make sure to clean up the stream when the component gets destroyed automatically.

```
<div class="todo-list-wrapper" *ngIf="(todoList$ | async) as todoList">
```

Make onPush the default change detection strategy

Using schematics you can make onPush the default changeDetection strategy when generating new components with Angular CLI. You simply add this to the schematics property in Angular.json:

```
 "changeDetection": "OnPush"
```

Using pipes instead of methods in templates

Methods in a template will get triggered every time a component gets rerendered. Even with onPush change detection, that will mean that it gets triggered every time there is interaction with the component or any children of the component (click, type). If the methods are doing heavy computations, this will make the app slow as it scales as it keeps recomputing every time there is interaction with the component.

What you can do instead is using a pure pipe to make sure, that you are only recomputing when the input to the pipe changes. async pipe, as we looked at before, is an example of a pure pipe.

A pure function

There's so much information on the web about functional programming that probably every developer knows what a pure function is. For myself I define a pure function as a function that doesn't have an internal state. It means that all operations it performs are not affected by that state and given the same input parameters and produces the same deterministic output.

```
const addPure = (v1, v2) => {
  return v1 + v2;
};
const addImpure = (() => {
  let state = 0;
  return (v) => {
    return state += v;
  }
})();
```

If I call both functions with the same input, say number 1, the first one will produce the same output 2 on every call:

```
addPure(1, 1);  // 2
addPure(1, 1);  // 2
addPure(1, 1);  // 2
```

while the second one produces different output:

```
addImpure(1);  // 1
addImpure(1);  // 2
addImpure(1);  // 3
```

So the key takeaway here is that even if the input doesn't change the impure function can produce different output. It means that we cannot use the input value to determine if the output will change.

Pure:

input parameters value determine the output so if input parameters don't change the output doesn't change

can be shared across many usages without affecting the output result
Impure:
cannot use the input value to determine if the output will change
cannot be shared because the internal state can be affected from outside
This has the consequence of triggering the method every time a button is clicked inside of the component that is even using onPush change detection:
With pipe
We fix this by converting the method to a pipe, as a pipe as default is pure it will rerun the logic if the input changes (reference change).
Now, this pipe is only being triggered when the input (todolist) has changed.
Cache values from pure pipes and functions
Even when using pure pipes, we can optimize this further by remembering/caching previous values, so we don't need to recompute if we already run the pipe with the same input in the past. Pure pipes don't remember the previous values, but will just make sure that if the input hasn't changed the reference, it will not recompute. To do the caching of previous value we need to combine it with something else.
An easy way to do this is to use Lodash memorize method. In this case, this is not very practical as the input is an array of objects. If the pipe was taking a simple data type, such as number as input, it could be beneficial to use this as a key to cache results and thus avoid recomputation.
Using trackBy in ngFor
When using ngFor and updating the list, Angular will by default remove the whole list from the DOM and create it again, because it has no way, by default, to know which item has been added or removed from the list. The trackBy function is solving this by allowing you to provide Angular with a function used for evaluating, which item has been updated or removed from the ngFor list, and then only rerender that.

```
 public trackByFn(index, item) {
   return item.id;
 }
```

This will track changes in the list based on the id property of the items (todo items).
The trackBy function is used in the template like this:

```
<ul class="list-group mb-3">
  <app-todo-item-list-row *ngFor="let todo of todos; trackBy: trackByFn"
[todoItem]="todo" (todoDelete)="deleteTodo($event)"
(todoEdit)="editTodo($event)"></app-todo-item-list-row>
</ul>
```

Improving page load

The page load time is an important aspect of user experience today. Every millisecond a user is waiting, potentially means a loss in revenue, because of a higher bounce rate and worse user experience, so this is a place you should optimize. Page load time also has an impact on SEO, as faster websites are rewarded by search engines.
For improving page load we want to use caching using Angular PWA, lazy loading and bundling.
Cache static content using Angular PWA
Caching the static content will make your Angular app load faster as it will already be in the browser. This is easily done using Angular PWA which will use service workers to cache the static content, that is the js, css bundles, images and static served files, and present them without making a call to the server.
I have already created a guide to how to setup caching with Angular PWA you can read here.
Cache HTTP calls using Angular PWA
With Angular PWA you can easily set up caching rules for HTTP calls to give a faster user experience without cluttering your app with a lot of caching code.
Either you can optimize for freshness or performance, that is, you can either choose to only read the cache if the HTTP call times out or first check the cache and then only call the API then the cache expires.
I have a guide with a video showing you how to do this here.
Lazy load routes

Lazy loading routes will make sure that a feature will be bundled in its own bundle and that this bundle can be loaded when it is needed.
To set up lazy loading we simply create a child route file like this in a feature:
Optimizing bundling and preloading
To optimize page load even further you can choose to preload the feature modules, so navigation is instant when you want to render a lazily loaded feature module.
This can be done by setting the: preloadingStrategy to PreloadModules as:

```
RouterModule.forRoot(routes, {
  preloadingStrategy: PreloadAllModules
})
```

Server-side rendering with Angular Universal
For Angular apps that are containing indexed pages, it is recommended to server-side render the app. This will make sure the pages are being fully rendered by the server before shown to the browser which will give a faster page load. This will require that the app is not dependent on any native DOM elements, and you should instead inject
Improving UX
Performance tuning is all about optimizations at the bottleneck, that is the part of the system that is affecting your user experience the most. Sometimes the solution could just be to handle actions more optimistically and thus less waiting for the user.

How should I prioritize performance tuning?
Start with low hanging fruits: onPush, Lazy loading and then PWA and then gain awareness of where your performance bottlenecks are in the system. Every improvement that is not at the bottleneck is an illusion as it will not improve the user experience with the app. Tuning methods like detaching the change detection should only be used if you have a specific problem with a component's change detection impacting performance.
OnPush
Lazy loading modules

Improve page load with Angular PWA
trackBy for ngFor
Pure pipes instead of methods (including async)
Cache values from pipes and pure functions
Cache HTTP requests better
Detach/manual change detection
Angular Universal
Why is Angular Universal the last one? Because introducing server-side rendering can cause big changes to the development setup (need to maintain another server, cannot reference DOM and need to maintain a server and a client bundle) and should be used either for performance reasons that can not be fixed with the previous steps or SEO purposes.
https://github.com/lydemann/angular-performance-tuning-guide
angular trainer
github.com/lydemann
christianlydemann.com/
git
https://christianlydemann.com/angular-architect-course-video/
AOT Compilation
When running a production build, Angular using JIT (just in time) compilation, which essentially means, Angular compiles your views in the browser at runtime. This has two downsides. First, the compilation process must run before your application can be used, and this can increase the time it takes for your site to load. Secondly, we have to ship the Angular compiler with your application, and it is not a small module!
By taking advantage of AOT (ahead of time) compilation, we move this step to build time so we do it once when building our application, and only ship the compiled templates. We can now remove the Angular compiler from our bundle (reducing our bundle size by ~1mb) and allows us to skip the compilation step making our pages load much quicker!
Minification

Code minification is the process tools like UglifyJS perform to optimize the code we have written. It performs many optimizations, for example, removing whitespace, renaming properties, dead code elimination and much much more. When developing having well named variables make development much easier, but when shipping our applications we don't need these names to be so helpful, so while averageUserAge might be useful when developing this could be renamed to a1 reducing the amount of code needed to be shipped.

12. Unsubscribe

As previously mentioned, Angular uses observables quite a lot, and if you make any HTTP requests or are listening to router events, you will too.

Observables are great, but you need to ensure once you are finished with them that you unsubscribe, otherwise memory leaks can occur and this can cause performance issues.

Unsubscribing is easy, you store the subscription, and then use the ngOnDestroy lifecycle hook to unsubscribe, eg:

```
export class AppComponent implements OnDestroy {
  private _subscription: Subscription;
  constructor(router: Router) {
    this._subscription = router.events.subscribe(event => {
      // do stuff here
    });
  }
  ngOnDestroy(): void {
    this._subscription.unsubscribe();
  }
}
```

14. Profiling

There are many things that I can list to improve performance and even if your application followed everything listed here you may still have performance issues. That is simply because each application is different, it will use different third party libraries and be architected differently. And this is where profiling comes in!

The developer tools for all modern browser come equipped with performance profiling tools to help identify code that is running slowly, which is great to help figure out how you can improve it further.

There are a few other tools you can use to help improve performance, first the Webpack Bundle Analyzer. This tool allows you to visually explore your bundle. It can let you see what modules or libraries are the largest, but more importantly it can help identify items that should not have been included in the bundle. For example if you are using an older version of RxJS, accidentally importing directly from rxjs would have included the whole library in your bundle. Tools like this can help spot this kind of mistake and allow you to easily rectify it.

Lighthouse testing is another great way to see how you application performs on a range of devices. It is a tool now built in to the Chrome Dev Tools. It will profile many aspects of your application, such a load performance, accessibility, PWA support, SEO optimization and Best Practices. It can also give you a good indication about how well your site will perform in regards to Google rankings as many of these criteria it tests for will affect the site ranking.

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

Of note however, is the fact that the hoisting mechanism only moves the declaration. The assignments are left in place.

If you've ever wondered why you were able to call functions before you wrote them in your code, then read on!

However, in contrast, undeclared variables do not exist until code assigning them is executed.

Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed. This means that, all undeclared variables are global variables.

```
function hoist() {
  a = 20;
  var b = 100;
```

```
}
hoist();
console.log(a);
/*
Accessible as a global variable outside hoist() function
Output: 20
*/
console.log(b);
/*
Since it was declared, it is confined to the hoist() function scope.
We can't print it out outside the confines of the hoist() function.
Output: ReferenceError: b is not defined
*/
```

The apply() method is an important method of the function prototype and is used to call other functions with a provided this keyword value and arguments provided in the form of array or an array like object.

But what if we want the elements to be pushed individually instead as an array? Sure there are literally n number of ways to do so, but as we are learning apply, let's use it:

In the given example we can see the use of apply in joining two given arrays. The arguments array is the elements array and the this argument points to the array variable.

The call() method is used to call a function with a given this and arguments provided to it individually.

This is very similar to apply, the only difference being that apply takes arguments in the form of an array or array-like objects, and here the arguments are provided individually.

https://www.freecodecamp.org/news/how-to-use-the-apply-call-and-bind-methods-in-javascript-80a8e6096a90/

those services dependency injection

to reduce bootstrap time modules are loaded only when they are required

loadchildren

path of components

module path and component path
app.module.js
app.routing
observable : are used to provide stream of values over time
promises return one value
promises are uncancellable
by using unsubscribe
observable are lazy
promises are eager
observable subscibe is needed
observable has rich set of operators.
in rxjs
map: map is used to modify the data given by observable
2 to b
3 to c change
where observable does not use
async await
promise use async await
observable
autocomplete case we use observable
async pipe
pipes are used to modify the value
pure pipe and impure pipe
pure pipe no inner state
impure pipe no same imput
async pipe

async direct values
async subscibe and unsubscribe automatically
directives
structural (dom modify), attribute(element behaviour), component(view screen)
memory leaks
if proper unsubscribe then can

flat map, join map, merge map
flat , merge : inner and outer observable can be merged.
outer observable: a,b,c,d
flat map :will subscribe automatically a,b,c,
flattening operator when there is case of nested observable then we use it to flatten the result
a:1,2,3
b:4,5,6
flat map return 1,2,3 can be veried and so on
switch map: a subscribe then b subscribe like autocomplete case
concat map:first a subscribe then b subscribe and then c subscribe
first operator: first value take and unsubscribe
take operator: value define then unsubscribe do
life cycle hook
ngoninit initialize the value
ngonchanges (change detection
view child queries resolved
abstract class: implementation detail definition not, some implementation can be , abstract method can be
interface not implementation
only abstract
view encapsulation: shadow-down,none,emulated
none: parent style can be taken, style can be leaked
emulated parent leak not
shadow down: parent se nahi lega and leak not
comp b in comp a
angular performance
lazy loading
memory leaking avoid
best practice of angular
set interval to clear interval
subscribe to unsubscribe
iv renderer : angular 9

tree shaking
the code which is not required will be removed from bundle
ng build prod
webpack mein
angular view engine was not tree shakable. ivrenderer se angular view engine became tree shakable
angular 11 features check
interceptor
layer make token add , response modify want to do
singleton services
whose one copy shared in the whole application
route module provider mein service declare kar di hai
singleton service having one object and broaden to all
parent child not use then subject se
comp
3 variations
1 subject and behavioral
subject: observable and observer at same time
observable value emit and observer value consume
behaviour subject: initial value can be assigned
The Promise.all() method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises. It rejects immediately upon any of the input promises rejecting or non-promises throwing an error, and will reject with this first rejection message / error.
Since the first, returns an array, we can wait for it to resolve and iterate over the array and run the other two promises subsequently i.e for each array item, we wait for the getProductId promise to be resolved then call the capitalizeId promise as follows:

```
const capitalizeProductsIds = async () => {
  const products = await getProducts()
  for (let product of products) {
```

```
    const productId = await getProductId(product);
    console.log(productId);
    const capitalizedId = await capitalizeId(productId);
    console.log(capitalizedId);
  }
  console.log(products);
}
capitalizeProductsIds()
```

Combining the async/await syntax with the for..of loop will give us the following output

```
product1
PRODUCT1
product2
PRODUCT2
product3
PRODUCT3
product4
PRODUCT4
(4) [{…}, {…}, {…}, {…}]
```

That's not the behavior that we are really looking to implement but instead we want to execute the first promise and wait for it to complete, then the second promise and finally the third promise when the second is fully completed.

Combining And Resolving all Promises with Promise.all(), map() and Async/Await

So instead of using the for loop with the async/await syntax, we need to use the Promise.all() and map() methods with async/await as follows:

```
const capitalizeProductsIds = async () => {
  const products = await getProducts()
  Promise.all(
    products.map(async (product) => {
      const productId = await getProductId(product);
      console.log(productId);
      const capitalizedId = await capitalizeId(productId)
      console.log(capitalizedId);
```

```
    })
  )
  console.log(products);
}
capitalizeProductsIds();
```

This is the output:

```
(4) [{…}, {…}, {…}, {…}]
product1
product2
product3
product4
PRODUCT1
PRODUCT2
PRODUCT3
PRODUCT4
```

The Promise.all() method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises. It rejects immediately upon any of the input promises rejecting or non-promises throwing an error, and will reject with this first rejection message / error.

Event bubbling allows a single handler on a parent element to listen to events fired by any of its children.

Angular supports bubbling of DOM events and does not support bubbling of custom events.

Mixing the names of custom and DOM events (click, change, select, submit) could be problematic. Use stopPropagation to avoid double event handler invocation.

Prefer naming custom events after their intent and not cause.

```
<!-- custom event clashes with DOM event -->
<product-form (submit)="onSubmit($event)"></product-form>
<!-- custom event shows its intent and does not clash -->
<product-form (save)="onSaveProduct($event)"></product-form>
```

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed.

ngOnChanges()

Respond when Angular sets or resets data-bound input properties. The method receives a SimpleChanges object of current and previous property values.

Called before ngOnInit() (if the component has bound inputs) and whenever one or more data-bound input properties change.

ngOnInit()

Initialize the directive or component after Angular first displays the data-bound properties and sets the directive or component's input properties

Called once, after the first ngOnChanges(). ngOnInit() is still called even when ngOnChanges() is not (which is the case when there are no template-bound inputs).

ngDoCheck()

Detect and act upon changes that Angular can't or won't detect on its own. See details and example in Defining custom change detection in this document.

Called immediately after ngOnChanges() on every change detection run, and immediately after ngOnInit() on the first run

ngAfterContentInit()

Respond after Angular projects external content into the component's view, or into the view that a directive is in.

Called once after the first ngDoCheck().

ngAfterContentChecked()

Respond after Angular checks the content projected into the directive or component.

Called after ngAfterContentInit() and every subsequent ngDoCheck()

ngAfterViewInit()

Respond after Angular initializes the component's views and child views, or the view that contains the directive.

Called once after the first ngAfterContentChecked()

ngAfterViewChecked()
Respond after Angular checks the component's views and child views, or the view that contains the directive.
Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked()
ngOnDestroy()

Cleanup just before Angular destroys the directive or component. Unsubscribe Observables and detach event handlers to avoid memory leaks.
Called immediately before Angular destroys the directive or component.
https://angular.io/guide/lifecycle-hooks
ViewChild
Property decorator that configures a view query. The change detector looks for the first element or the directive matching the selector in the view DOM. If the view DOM changes, and a new child matches the selector, the property is updated.
View queries are set before the ngAfterViewInit callback is called.
Decorators are a design pattern that is used to separate modification or decoration of a class without modifying the original source code. In AngularJS, decorators are functions that allow a service, directive or filter to be modified prior to its usage.
A common pattern in Angular is sharing data between a parent component and one or more child components. Implement this pattern with the @Input() and @Output() decorators.
Sending data to a child component
The @Input() decorator in a child component or directive signifies that the property can receive its value from its parent component.

```
  @Input() item = '';
<p>
  Today's item: {{item}}
</p>
<app-item-detail [item]="currentItem"></app-item-detail>
```

src/app/app.component.ts
content_copy

```
export class AppComponent {
  currentItem = 'Television';
}
```

Sending data to a parent component

The @Output() decorator in a child component or directive lets data flow from the child to the parent.

Configuring the child component

The following example features an <input> where a user can enter a value and click a <button> that raises an event. The EventEmitter then relays the data to the parent component.

Import Output and EventEmitter in the child component class:

Import Output and EventEmitter in the child component class:

content_copy

```
import { Output, EventEmitter } from '@angular/core';
```

In the component class, decorate a property with @Output(). The following example newItemEvent @Output() has a type of EventEmitter, which means it's an event.

src/app/item-output/item-output.component.ts

content_copy

```
@Output() newItemEvent = new EventEmitter<string>();
```

The different parts of the preceding declaration are as follows:

@Output()—a decorator function marking the property as a way for data to go from the child to the parent

newItemEvent—the name of the @Output()

EventEmitter<string>—the @Output()'s type

new EventEmitter<string>()—tells Angular to create a new event emitter and that the data it emits is of type string.

For more information on EventEmitter, see the EventEmitter API documentation.

Create an addNewItem() method in the same component class:

src/app/item-output/item-output.component.ts

content_copy

```
export class ItemOutputComponent {
  @Output() newItemEvent = new EventEmitter<string>();
```

```
  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

The addNewItem() function uses the @Output(), newItemEvent, to raise an event with the value the user types into the <input>.

design patterns

Your structural patterns, they deal with structuring the relationship between different components (or classes) and forming new structures in order to provide new functionalities. Examples of structural patterns are Composite, Adapter and Decorator.

Your behavioral patterns, they help abstract common behavior between components into a separate entity which in turn, and your creational patterns. Examples of behavioral patterns are Command, Strategy, and one of my personal favorites: the Observer pattern.

Your creational patterns, they focus on class instantiation and making your life easier in order to create new entities. I'm talking about Factory method, Singleton and Abstract Factory.

Singleton

The singleton pattern is probably one of the most known design patterns out there. It is a creational pattern because it ensures that no matter how many times you try to instantiate a class, you'll only have one instance available.

This is a great pattern to handle things such as database connections, since you'll probably want to only handle one at a time, instead of having to re-connect on every user request.

```
class MyDBConn{
   protected static instance: MyDBConn | null = null
   private id: number = 0
   constructor() {
      //... db connection logic
      this.id = Math.random() //the ID could represent the actual connection to the
db
   }
```

```
  public getID(): number {
    return this.id
  }
  public static getInstance(): MyDBConn {
    if(!MyDBConn.instance) {
      MyDBConn.instance = new MyDBConn()
    }
    return MyDBConn.instance
  }
}
const connections = [
                MyDBConn.getInstance(),
                MyDBConn.getInstance(),
                MyDBConn.getInstance(),
                MyDBConn.getInstance(),
                MyDBConn.getInstance()
            ]
connections.forEach( c => {
  console.log(c.getID())
})
```

Factory method

As I've already mentioned, the Factory Method pattern is a creational pattern, just like Singleton. However, instead of directly working on top of the object we care about, this pattern only takes care of managing its creation.

Let me explain: imagine you have to write code that will move vehicles, they are very different types of vehicles (i.e a car, a bicycle and a plane), the movement code should be encapsulated inside each vehicle class, but the code that calls their move method can be generic.

The question here is how are you going to handle the object creation? You could have a single creator class with 3 methods, or one method that receives a parameter. In either case, extending this logic in order to support the creationg of more vehices requires you to keep growing the same class.

However, if you decided to use the factory method pattern, you could do something like below:
https://blog.bitsrc.io/design-patterns-in-typescript-e9f84de40449
Why use forkJoin?
This operator is best used when you have a group of observables and only care about the final emitted value of each. One common use case for this is if you wish to issue multiple requests on page load (or some other event) and only want to take action when a response has been received for all. In this way it is similar to how you might use Promise.all.
Be aware that if any of the inner observables supplied to forkJoin error you will lose the value of any other observables that would or have already completed if you do not catch the error correctly on the inner observable. If you are only concerned with all inner observables completing successfully you can catch the error on the outside.
It's also worth noting that if you have an observable that emits more than one item, and you are concerned with the previous emissions forkJoin is not the correct choice. In these cases you may be better off with an operator like combineLatest or zip.

```
import { ajax } from 'rxjs/ajax';
import { forkJoin } from 'rxjs';
/*
  when all observables complete, provide the last
  emitted value from each as dictionary
*/
forkJoin(
  // as of RxJS 6.5+ we can use a dictionary of sources
  {
    google: ajax.getJSON('https://api.github.com/users/google'),
    microsoft: ajax.getJSON('https://api.github.com/users/microsoft'),
    users: ajax.getJSON('https://api.github.com/users')
  }
)
  // { google: object, microsoft: object, users: array }
```

```
  .subscribe(console.log);
```

switch map

You'll notice some of those requests got cancelled before they finished. Much like takeLatest in Redux-Saga, switchMap in RxJS only cares about the latest value that the observable emitted, in this case cancelling any previous HTTP requests that were still in progress.

But that wasn't exactly what I needed.

Much like takeEvery in Redux-Saga, mergeMap in RxJS passes all requests through, even when a new request was made before a previous one had finished — exactly what I needed!

switchMap cancels previous HTTP requests that are still in progress, while mergeMap lets all of them finish.

In my case, I needed all requests to go through, as this is a metrics service that's supposed to log all actions that the user performs on the web page, so I used mergeMap.

A good use case for switchMap would be an autocomplete input, where we should discard all but the latest results from the user's input.

Components—directives with a template. This type of directive is the most common directive type.

Attribute directives—directives that change the appearance or behavior of an element, component, or another directive.

Structural directives—directives that change the DOM layout by adding and removing DOM elements.

Dependency injection in Angular

Dependencies are services or objects that a class needs to perform its function. Dependency injection, or DI, is a design pattern in which a class requests dependencies from external sources rather than creating them.

BehaviourSubject

BehaviourSubject will return the initial value or the current value on Subscription

```
var bSubject= new Rx.BehaviorSubject(0);  // 0 is the initial value
bSubject.subscribe({
  next: (v) => console.log('observerA: ' + v)  // output initial value, then new values on `next` triggers
```

```
});
bSubject.next(1);  // output new value 1 for 'observer A'
bSubject.next(2);  // output new value 2 for 'observer A', current value 2 for
'Observer B' on subscription
bSubject.subscribe({
  next: (v) => console.log('observerB: ' + v)  // output current value 2, then new
values on `next` triggers
});
bSubject.next(3);
```

Subject

Subject does not return the current value on Subscription. It triggers only on .next(value) call and return/output the value

```
var subject = new Rx.Subject();
subject.next(1); //Subjects will not output this value
subject.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
});
subject.next(2);
subject.next(3);
```