Angular Foundation day 2

Alliander
Frank van Wijk & Albert Brand

Xebia

Who am I?

Check-in

Course content

Day 1:

- Quick intro to Angular
- Components part 1
- Templates
- Directives
- Dependency injection
- Observables

Day 2:

- Components part 2
- State management
- Modules, Project structure
- Angular CLI
- Angular Router, Forms, HTTP client
- Testing
- Development tools
- Good practices

Components part II

Components topics

We'll dive a bit deeper into the following component topics:

- component interaction strategies
- the component lifecycle
- component change detection
- advanced content projection

Component input interception

Let's return to input bindings.

`@Input` can decorate a getter/setter pair instead of a property. This makes it possible to intercept values that are passed in and act upon them.

Note: you can use a custom trim template pipe instead of doing this.

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

@Component({
  selector: 'app-heading-name',
  template: '<h3>{{ name }}</h3>'
})
export class HeadingNameComponent {
  private _name = '';

  @Input()
  get name(): string { return this._name; }
  set name(name: string) {
    this._name = name.trim();
  }
}
```

Xebia

Component input interception

If there are multiple input properties and you want to update your component once on any change, it's better to implement the `ngOnChanges` method from the `OnChanges` interface.

```
import { Component, Input, OnChanges,
  SimpleChanges } from '@angular/core';

@Component({
  selector: 'app-sum-two',
  template: './app-sum-two.html',
})
export class SumTwoComponent implements OnChanges {
  @Input() a = 0;
  @Input() b = 0;
  sum: number;

  ngOnChanges(changes: SimpleChanges) {
    this.sum = this.a + this.b;
  }
}
```

Component interaction

Let's go a bit deeper in component communication.

For example, here is a `MyCounterComponent` that has a public `count` value and `increase` method. Can we access the internal state without adding `@Input` or `@Output`?

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

@Component({
  selector: 'app-my-counter',
  template: '<p>Counter: {{ counter }}</p>'
})
export class MyCounterComponent {
  count = 0;

  increase() {
    this.count += 1;
  }
}
```

Xebia

Component interaction

There is a way!

You can read any public property and call any public method via template variable access.

This can be an escape hatch for interaction between components.

Rule of thumb: use this sparingly! Keep internals of your component private.

```
<app-my-counter #counter></app-my-counter>
<button (click)="counter.increase()">Increase</button>
<div class="count">{{counter.count}}</div>
```

Xebia

Component interaction

Remember the `@ViewChild` decorator? You can also use it to inject a component based on other kinds of selectors.

Instead of selecting a template variable name, you can pass in the class of the component you want to inject.

Be wary: the component is only resolved after init!

```
import { Component } from '@angular/core';
import {
  MyCounterComponent } from './my-counter.component';

@Component({
  selector: 'app-counter-parent',
  template: `
    <app-my-counter></app-my-counter>
    <button (click)="start()">Increase</button>
  `
})
export class CounterParentComponent {
  @ViewChild(MyCounterComponent)
  private myCounter!: MyCounterComponent;

  start() { this.myCounter.start(); }
}
```

Component interaction

Feels quite brittle right? And what if two components do not have a direct parent-child relation?

A good practice is to let two or more components interact using observables from a shared service.

First: create a separate service that exposes the shared data as observables and the methods to manipulate it.

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable()
export class UserService {
  private loggedInUserSubject = new Subject<string>();

  loggedInUser$ =
    this.loggedInUserSubject.asObservable();

  logIn(user: string) {
    this.loggedInUserSubject.next(user);
  }
}
```

Component interaction

Second: provide the service in the component tree. In this case we only want to expose it to all descendents of the parent.

You can inject it immediately as well as seen here.

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-user-parent',
  template: `
    <h2>Parent</h2>
    <app-user-child></app-user-child>
    <button (click)="logIn()">Log in user</button>
  `,
  providers: [UserService]
})
export class UserParentComponent {
  constructor(private userService: UserService) { }

  logIn() {
    this.userService.logIn('John Doe');
  }
}
```

Component interaction

Third: inject it in the other component(s) and use the exposed service observable.

Your internal state is now only a reference to an observable which is managed by Angular.

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-user-child',
  template: `<div>
    Logged in user:
    {{ user$ | async }}
  </div>`,
})
export class UserChildComponent {
  user$ = this.userService.loggedInUser$;

  constructor(private userService: UserService) {}
}
```

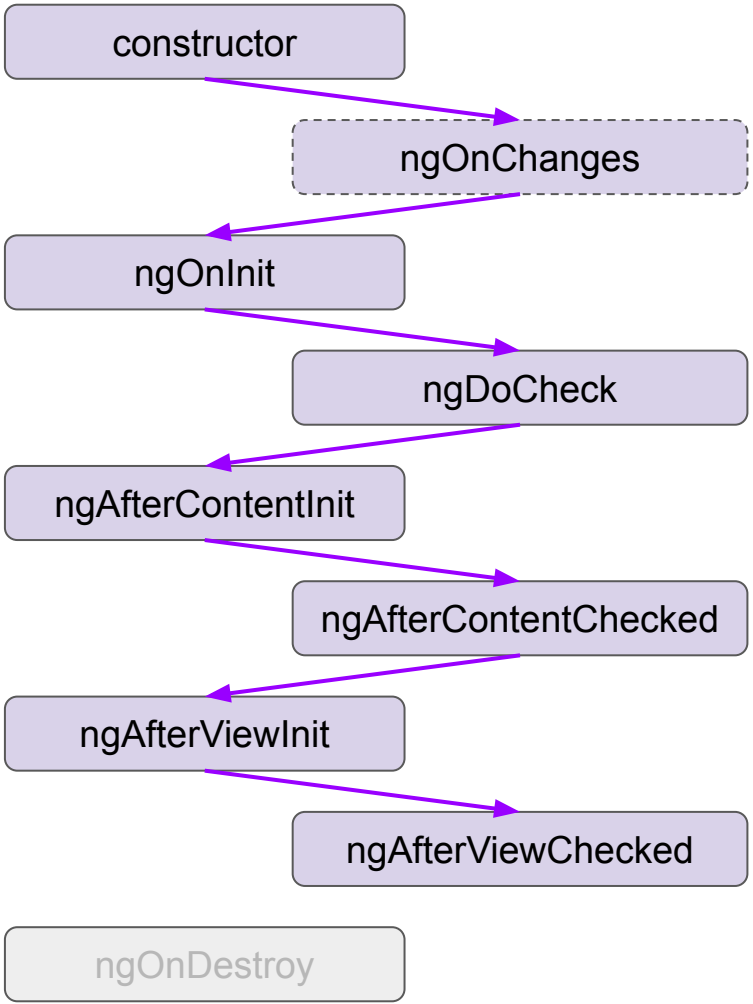
Component lifecycle

So far we have seen a couple of component lifecycle interfaces. These allow you to tap into the lifecycle of components and directives as they are created, updated and destroyed. Each interface defines one method, prefixed with `ng`.

Let’s go over them!

Component lifecycle

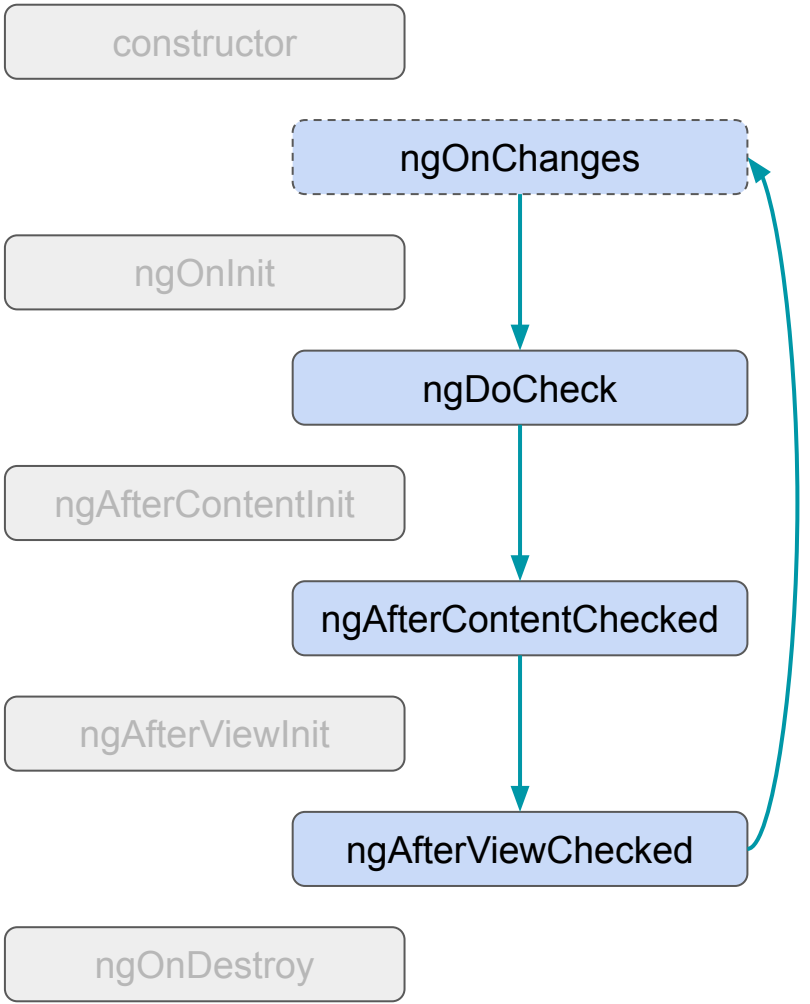
When a component is created, all purple lifecycle methods are called in order.



Xebia

Component lifecycle

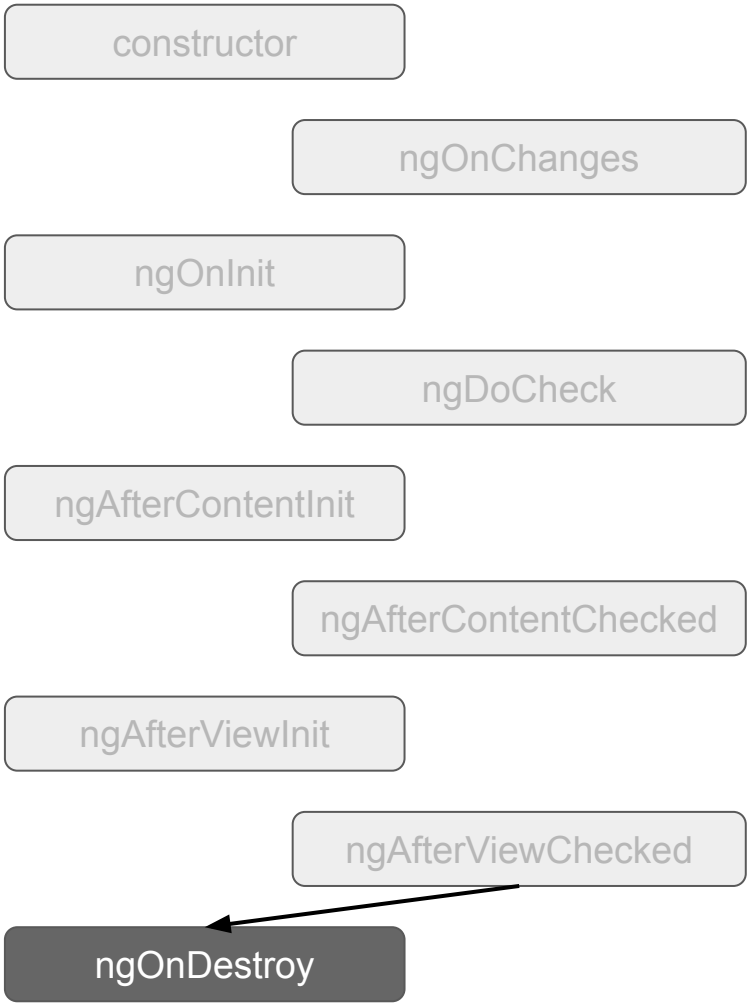
After the first `afterViewChecked`, Angular returns to `ngOnChanges`. From now on only the blue lifecycle methods are called.



Xebia

Component lifecycle

Just before the component is destroyed and after the last `afterViewChecked`, `ngOnDestroy` is called.



Xebia

ngOnChanges

`ngOnChanges` is called when Angular sets `@Input` properties. The method receives a `SimpleChanges` object that can be used to compare previous with current input.

The method is called frequently, so any operation you perform here can quickly become a performance issue.

If there is no `@Input`, then `ngOnChanges` will not be called.

```
export class LogInputsComponent implements OnChanges {
  @Input() a = 0;
  @Input() b = 0;

  ngOnChanges(changes: SimpleChanges) {
    for (const propName in changes) {
      const changedProp = changes[propName];
      const from = changedProp.previousValue;
      const to = changedProp.currentValue;
      console.log(
        `${propName} changed from ${from} to ${to}`);
    }
  }
}
```

ngOnInit

The `ngOnInit` method is called the moment the first input properties have been set. If there are no inputs, `ngOnInit` is still called.

It allows for (asynchronous) initialization outside of the constructor. For instance, let a service fetch initial data.

You can use input properties as they are sure to be set.

```
@Component({ ... })
class DataComponent implements OnInit {
  @Input id: number;

  constructor(
    private externalService: ExternalService) {}

  ngOnInit() {
    this.externalService.byId(this.id)
      .subscribe(() => { ... })
  }
}
```

ngDoCheck

`ngDoCheck` can be used to check for changes that Angular does not detect.

For instance, if you use a non-Angular component in your app, you can use the `ngDoCheck` moment to reflect its change in one or more properties of your component.

```
@Component({
  template: `<div></div>`
})
class GalleryComponent implements DoCheck {
  gallery: Gallery;
  pageNumber = 1;

  constructor(el: ElementRef) {
    this.gallery = new Gallery(el.nativeElement);
  }

  ngDoCheck() {
    this.pageNumber = this.gallery.getPageNumber();
  }
}
```

ngAfterContentInit

`ngAfterContentInit` is called when the content of the component is initially projected to an `ng-content` element.

The example uses `@ContentChild` to select a specific `ChildComponent` inside the projected content and calls a method on it.

```
@Component({
  template: `
    <div>projected content:</div>
    <ng-content></ng-content>
  `
})
export class MyComponent implements AfterContentInit {
  @ContentChild(ChildComponent)
  contentChild!: ChildComponent;

  ngAfterContentInit() {
    contentChild.doSomething()
  }
}
```

ngAfterContentChecked

`ngAfterContentChecked` is used to perform an action after Angular checked if the projected content has changed.

In the example, the `contentChild` property is updated every time based on the result of the query.

```
export class MyComponent implements AfterContentChecked {
  @ContentChild(ChildComponent)
  contentChild!: ChildComponent;

  ngAfterContentChecked() {
    if (this.contentChild) {
      console.log('ChildComponent available!')
    }
  }
}
```

ngAfterViewInit

The `ngAfterViewInit` method is called when Angular finished to initialise the component view and child views, or the view the directive is part of.

At this moment you can safely use the components queried by `@ViewChild`.

```
@Component({
  template: `
    <div>child content:</div>
    <app-child-component></app-child-component>
  `
})
export class MyComponent implements AfterViewInit {
  @ViewChild(ChildComponent)
  viewChild!: ChildComponent;

  ngAfterViewInit() {
    viewChild.doSomething()
  }
}
```

ngAfterViewChecked

`ngAfterViewChecked` is used to perform an action after Angular has checked the component view and child views, or the view the directive is part of.

In the example, the `viewChild` property is updated based on the result of the query.

```
export class MyComponent implements AfterViewChecked {
  @ViewChild(ChildComponent)
  viewChild!: ChildComponent;

  ngAfterViewChecked() {
    if (this.viewChild) {
      console.log('ChildComponent available!')
    }
  }
}
```

ngOnDestroy

When the component is going to be destroyed, you can hook into `ngOnDestroy` to perform cleanup.

It is essential to unsubscribe from observables, stop interval timers and detach custom event handlers here.

Free anything that should be freed when the component is destroyed.

```
export class IntervalComponent implements OnDestroy {
  intervalId: number;

  constructor(private userService: UserService) {
    this.intervalId = setInterval(
      () => console.log('hi!'), 1000);
  }

  ngOnDestroy() {
    clearInterval(this.intervalId);
  }
}
```

Change detection

As you have seen in the lifecycle overview, Angular regularly performs change detection.

This is often the cause for slow performance of your app, and you should know several things about it to optimally implement an app.

Change detection

Angular uses the Zone.js library to detect when application state might have changed. It captures all asynchronous operations like `setTimeout`, `fetch` and event listeners to achieve this.

If you instantiate a third-party component inside your Angular app, it's internals will also be captured by Zone.js!

Prevent this by calling `runOutsideAngular`.

```
import { Component, NgZone, OnInit } from '@angular/core';
import * as Plotly from 'plotly.js-dist-min';

@Component(...)
class AppComponent implements OnInit {
  constructor(private ngZone: NgZone) {}

  ngOnInit() {
    this.ngZone.runOutsideAngular(() => {
      Plotly.newPlot('chart', data);
    });
  }
}
```

Xebia

Change detection

A cause of slowdowns can be that one or more components have slow computations. Remember: all template expressions are re-evaluated to determine changes.

The general solution is to optimize component code hotspots, for instance by debouncing inputs, adding caching, or by reducing the amount of updated components.

```
export class TypeaheadComponent {
  search = (text$: Observable<string>) =>
    text$.pipe(
      debounceTime(200),
      map(term => expensiveSearch(term))
    )
}
```

Change detection

When an application has a large component tree, running change detection can cause performance problems.

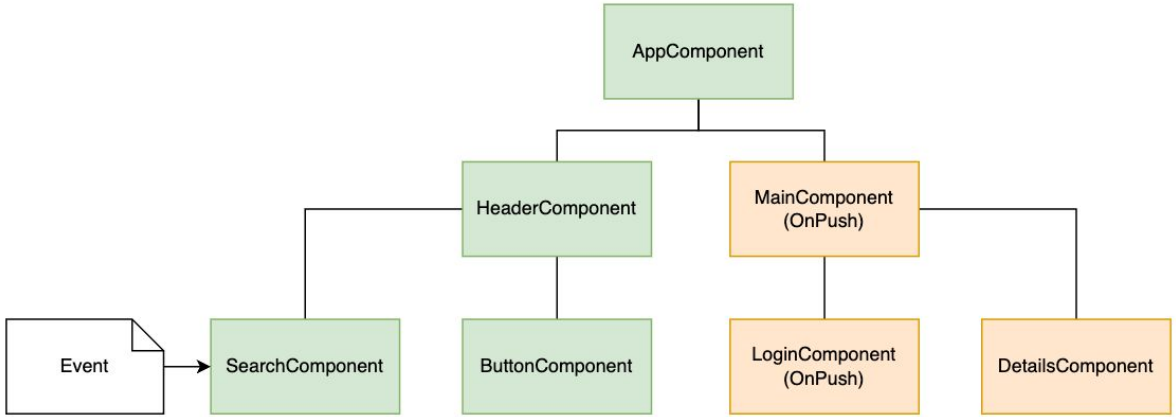
You can use the `changeDetection` property in `@Component` to adjust the strategy for specific component trees.

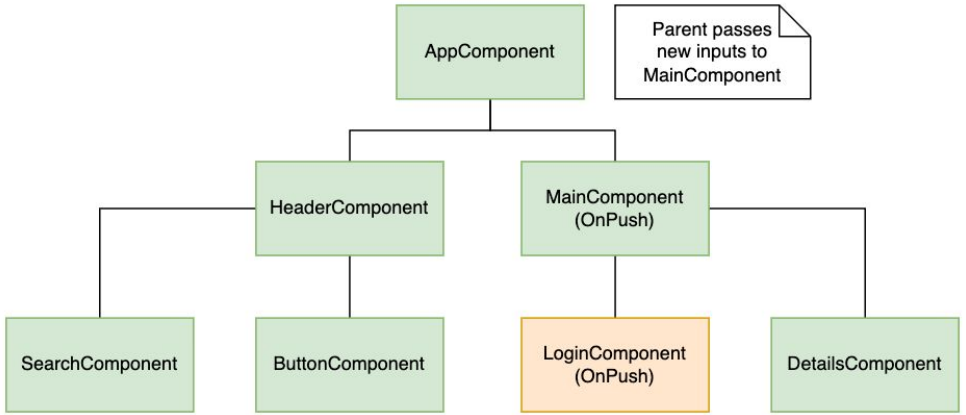
```
import {
  ChangeDetectionStrategy,
  Component
} from '@angular/core';

@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class MyComponent {}
```

Change detection

When `OnPush` is set on a root component of a subtree, change detection will only run in that subtree when:

- the root component receives new `@Input` values.
- a DOM event happens in the subtree.
- a component is explicitly marked for change detection, for instance triggered by the `async` pipe





Xebia

Content projection

The last component topic is content projection. You have seen the ‘single slot’ scenario in day 1. This can already reduce duplicate HTML by a lot, but there is more to explore.

Content projection

On day 1 you learned about single slot content projection. But you can have more than one slot!

Just add more `ng-content` elements with a `select` property. Angular supports selectors for any combination of tag name, attribute, CSS class, and the `:not` pseudo-class.

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

@Component({
  selector: 'app-zippy-multislot',
  template: `
    <h2>Multi-slot content projection</h2>

    Default:
    <ng-content></ng-content>

    Question:
    <ng-content select="[question]"></ng-content>
  `
})
export class ZippyMultislotComponent {}
```

Content projection

You can now render the component with the following content.

```
<app-zippy-multislot>
  <p question>Is content projection cool?</p>
  <p>Let's learn about content projection!</p>
</app-zippy-multislot>
```

The content will render as follows. Note that the `ng-content` element without a selector only receives content that does not match with any of the other `ng-content` elements.

```
<app-zippy-multislot>
  <h2>Multi-slot content projection</h2>

  Default:
  <p>Let's learn about content projection!</p>

  Question:
  <p question>Is content projection cool?</p>
</app-zippy-multislot>
```

Xebia

Content projection

A use case can be that you want to conditionally render content provided to the component.

Using `ng-content` in such a case does not work as the content provided is initialised before the component itself (regardless of any `*ngIf` statements in the component template).

```
@Component({
  selector: 'app-conditional-content',
  // BAD: projected content always initialises!
  template: `
    <h2>Conditional content projection</h2>
    <div *ngIf="show">
      <ng-content></ng-content>
    </div>
  `
})
export class ConditionalContentComponent {}
```

```
<app-conditional-content> (2)
  <app-child></app-child> (1)
</app-conditional-content>
```

Content projection

But there is a nice solution! Instead of passing in content, you can pass in a template of the content, which by design is only initialised when it is rendered.

You first wrap the provided content in an `ng-template` element. This makes sure that `app-child` is not initialised.

```
<app-conditional-content>
  <ng-template #content>
    <app-child></app-child>
  </ng-template>
</app-conditional-content>
```

Xebia

Content projection

Then, you use a `@ContentChild` query to get a reference to the template in the component.

Last, you replace the `ng-content` element with an `ng-container`. That component has another function: it can render any template you provide it via `ngTemplateOutlet`.

```
@Component({
  selector: 'app-conditional-content',
  template: `
    <h2>Conditional content projection</h2>
    <div *ngIf="show">
      <ng-container
        [ngTemplateOutlet]="contentTemplate">
      </ng-container>
    </div>
  `
})
export class ConditionalContentComponent {
  @ContentChild('content')
  contentTemplate!: TemplateRef<unknown>;
}
```

Xebia

Hands-on 07: communicate between components

https://github.com/xebia/xebia-angular-training-exercises

State Management

State management

So far we've seen a couple of different ways to manage state inside of your Angular application. Let's do a quick refresher and talk about when each case is appropriate.

State in a component

If you keep state in a component, it's main use is in the component template. State in a component should 'fit' logically with the component purpose.

If more components need to use this data, you can choose to **move state up** and pass it down via `@Input`.

```
@Component({
  selector: 'app-todos',
  template: `
    <app-todo-list
      [todos]="todos"
      (toggle)="toggleTodo($event)"
    ></app-todo-list>`
})
export class TodosComponent {
  todos: Todo[] = [
    { id: 0, title: 'Buy milk', done: true },
    { id: 1, title: 'Pay bills', done: false },
  ];
  toggleTodo(id: number) {} // handle toggle
}
```

State in a component

Beware of “property drilling”!

Here `todo-list` is just passing down inputs and passing up events. It has local state but there is no other benefit than connecting inputs and outputs.

```
@Component({
  selector: 'app-todo-list',
  template: `
    <h1>Todo list:</h1>
    <app-todo-item
      *ngFor="let todo of todos"
      [todo]="todo"
      (toggle)="toggle.emit($event)"
    ></app-todo-item>`
})
export class TodoListComponent {
  @Input() todos!: Todo[];
  @Output() toggle = new EventEmitter<number>();
}
```

State in a component

And to make it worse, `todo-item` does again the same. It renders the todo title, but there's a lot of boilerplate starting to build up.

```
@Component({
  selector: 'app-todo-item',
  template: `
    <div>
      <app-toggle-todo
        [todo]="todo"
        (toggle)="toggle.emit($event)"
      ></app-toggle-todo>
      <span>{{ todo.title }}</span>
    </div>`
})
export class TodoItemComponent {
  @Input() todo!: Todo;
  @Output() toggle = new EventEmitter<number>();
}
```

State in a component

Finally, `toggle-todo` renders the checkbox toggle and handles the actual event.

All these components have a reference to some part of the state even when they dont use it. All create new event emitters, which negatively impacts performance.

Can we do better?

```
@Component({
  selector: 'app-toggle-todo',
  template: `
    <input
      type="checkbox"
      [checked]="todo.done"
      (change)="toggle.emit(todo.id)"
    />
  `
})
export class ToggleTodoComponent {
  @Input() todo!: Todo;
  @Output() toggle = new EventEmitter<number>();
}
```

State in a component

A generic solution to this problem is **component composition**.

Instead of passing data to components, you pass **content**. In the components you render the content using `ng-content`.

In this way, most components become ‘dumb’ containers that only contribute surrounding elements to its children.

```
@Component({
  selector: 'app-todos',
  template: `
    <app-todo-list>
      <app-todo-item *ngFor="let todo of todos">
        <app-toggle-todo
          [todo]="todo"
          (toggle)="toggleTodo($event)"
        ></app-toggle-todo>
        <span>{{ todo.title }}</span>
      </app-todo-item>
    </app-todo-list>`
})
export class TodosComponent {
  // ...
}
```

State in a service

The Angular solution is to provide a service to extract the data and its logic, and provide it to the subtree via dependency injection.

You move both state and manipulation methods to the service.

Passing down the `todos` is now implicit via dependency injection.

```
export class TodoState {
  todos = [
    { id: 0, title: "Buy milk", done: true },
    { id: 1, title: "Pay bills", done: false }
  ];
  toggleTodo(id: number) {} // handle toggle
}


@Component({
  selector: 'app-todos',
  template: `<app-todo-list></app-todo-list>`,
  providers: [TodoState]
})
export class TodosComponent { }
```

Xebia

State in a service

You inject the service via the constructor and use the state service where needed.

The `todo-list` injects the state to render each `todo-item`.

```
@Component({
  selector: 'app-todo-list',
  template: `
    <h1>Todo list:</h1>
    <app-todo-item
      *ngFor="let todo of todoState.todos"
      [todo]="todo"
    ></app-todo-item>`
})
export class TodoListComponent {
  constructor(public todoState: TodoState) {}
}
```

State in a service

The `app-todo-item` has a bound `todo` property as local state, but it does not have to pass up events anymore.

Also, there is no need to inject the whole `todoState` here.

```
@Component({
  selector: 'app-todo-item',
  template: `
    <div>
      <app-toggle-todo [todo]="todo"
      ></apptoggle-todo>
      <span>{{ todo.title }}</span>
    </div>`
})
export class TodoItemComponent {
  @Input() todo!: Todo;
}
```

State in a service

Finally, `app-toggle-todo` uses the exposed method on the state service to toggle the todo based on its id.

There is no need to create an `EventEmitter` in this class because you can directly use the manipulation method from the state service.

```
@Component({
  selector: 'app-toggle-todo',
  template: `
    <input
      type="checkbox"
      [checked]="todo.done"
      (change)="todoState.toggleTodo(todo.id)"
    />
  `
})
export class ToggleTodoComponent {
  constructor(public todoState: TodoState) {}
}
```

State rules of thumb

State should be in a component when:

- it is for presentational use only
- it logically belongs to a single component

State should be in a service when:

- it is shared between multiple components
- it logically belongs to a subtree of your app

State should be in a global service when:

- it is of use to any component in your app
- the lifetime of the state is as long as the lifetime of your app

Hands-on 08:
extract state from component

https://github.com/xebia/xebia-angular-training-exercises

Modules

NgModule

An `NgModule` helps to organize related things together.

It is similar to a `Component` definition: it's a class with metadata in its decorator. In the metadata you specify module components, directives and pipes (together called *declarables*) and optionally providers for injection. You can import other modules, and use declarables from them. Also, you can export any of the declarables to other modules.

All Angular libraries are NgModules, such as `FormsModule` and `RouterModule`.

NgModule versus JS module

A JS module is simply an individual file that you import using the `import` syntax, from which you make classes, functions and variables available using the `export` syntax.

An `NgModule` brings together multiple declarables into cohesive blocks of functionality. The combined metadata of all imported `NgModule`s is used by the Angular compiler to build the final JS bundle of your application.

NgModule configuration

`declarations`: The declarables that belong to the `NgModule`.

`imports`: Other `NgModule`s that you want to import, to use their declarables.

`exports`: The declarables and modules that you want to export. All exported declarables and the declarables from the exported modules are made available in the importing module.

`providers`: Providers of services that components in this `NgModule` can use.

`bootstrap`: The entry component(s) that Angular creates and inserts into the `index.html` host web page, bootstrapping the application.

Root module

By convention and by default, the root `NgModule` is named `AppModule`.

The declarations contain the single component that this module can render. It imports the `BrowserModule` to be able to run the app in the browser.

By default, the `AppComponent` should be bootstrapped (i.e. rendered by the bundle as the root component).

```
import {
  BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Browser module

However, a module decorator is ‘just’ metadata. How does the app start in the browser?

By default, when you create an Angular app, a `main.ts` is created with the following content.

The call to `bootstrapModule` provided from the browser module schedules the rendering of the bootstrapped `AppComponent`.

```
import { platformBrowserDynamic } from
  '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Frequently used modules

`BrowserModule`	To run your application in a browser.
`CommonModule`	To use `*ngIf` and `*ngFor`. Exported by `BrowserModule`.
`FormsModule`	To build template driven forms.
`ReactiveFormsModule`	To build reactive forms.
`RouterModule`	To set up the router and use the router directives.
`HttpClientModule`	To communicate with a server over HTTP.

Custom module

To make a custom module, a good first start is to use the Angular CLI.

```
ng generate module CustomerDashboard
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

Custom module

When you generate a component using the CLI, it is automatically added to the new module if you specify it's path.

You can also use the `--module` flag to explicitly specify the target module.

```
ng generate component customer-dashboard/CustomerDash
```

```
import { CustomerDashComponent } from
  './customer-dashboard/customer-dash.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CustomerDashComponent
  ],
})
export class CustomerDashboardModule { }
```

Xebia

Custom module

By default, nothing is exported from the module. You want to be able to render the dashboard after importing this module, so you export the component.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CustomerDashComponent
  ],
  exports: [
    CustomerDashComponent
  ]
})
export class CustomerDashboardModule { }
```

Xebia

Custom module

Now you need to import the module in the `imports` configuration property in the root module.

Even though we did not import the `CustomerDash` component in the root module declarations, you can now add an `app-customer-dash` element in the `AppComponent` template.

```
// ..
import { CustomerDashboardModule } from
  './customer-dashboard/customer-dashboard.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    // ..
    CustomerDashboardModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Feature modules

What you just did created a so-called feature module. You grouped code related to a specific functionality together. If the customer dashboard is extended with more components and directives, you add them to that module.

It’s up to you to define a logical separation of the codebase. Angular supports this with NgModules.

A rule of thumb: start with a minimal amount of modules, and separate later.

Hands-on 09:
create and add a module

https://github.com/xebia/xebia-angular-training-exercises

Project Structure

Workspace

An Angular app is a set of files that is called a project. However, the project lives in an Angular workspace, which is comprised of one or more projects.

By default, when you generate the app using the CLI, code is placed under `src/app`.

The workspace root contains various configuration files, of which `angular.json` describes the workspace setup.

```
ng new my-beautiful-app
```

```
my-beautiful-app/
  src/
  README.md
  angular.json
  node_modules
  package.json
  tsconfig.json
  ...
```

Xebia

Workspace setup

The JSON file describes the projects in the current workspace. You can update per-project configuration options, such as the default app prefix, loaded global styles and file size-budgets for the built bundles.

The file is validated against a schema, so it will inform you when you make a mistake when updating it.

```
{
  "projects": {
    "my-beautiful-app": {
      "architect": {
        "build": {
          "options": { ... }
          "configurations": {
            "production": { ... },
            "development": { ... },
          }
        },
      },
    }
  }
}
```

Xebia

Application structure

Every project is unique, so if there is some (folder) structure, make sure to follow it.

However, the Angular team provides the LIFT acronym about the best structure:

- ***L**ocate code quickly*
- ***I**dentify the code at a glance*
- ***F**lattest structure you can achieve*
- ***T**ry to DRY*

Application structure

The `src` folder is the root for all project code. Here you will find the `main.ts` file that is the entrypoint for the application.

Images and other static asset files are placed under `assets`.

The `AppModule`, components and other modules are placed in the `app` folder.

```
src/
  app/
  assets/
  favicon.ico
  index.html
  main.ts
  styles.css
  ...
```

Xebia

Application structure

The default structure in a module folder is to place the module class and the root component for that module in the root.

Other module components are placed in their own folder.

Services, directives, pipes can be placed where they make most sense: next to the component if only used there, else in the root of the module or in a shared folder.

```
app/
  my-comp/
    my-comp.component.css
    my-comp.component.html
    my-comp.component.spec.ts
    my-comp.component.ts
  my-other-comp/
    ...
  app.component.css
  app.component.html
  app.component.ts
  app.module.ts
  ...
```

Application structure

If you use feature modules, you'll often end up with some code shared by multiple features. A way to handle this is by introducing a `shared` module.

Following the LIFT principle, you can go for the following folder structure.

```
app/
  items/
    item-list/
    item-detail/
    items.module.ts
  users/
    user-list/
    user-detail/
    users.module.ts
  shared/
    tooltip.directive.ts
    truncate.pipe.ts
    shared.module.ts
  app-routing.module.ts
  app.component.ts
  app.module.ts
```

Standalone components

By default, Angular requires quite some boilerplate code to be generated before you can start coding. Especially the dance around the root `NgModule` causes frustration. Boilerplate has been one of the reasons why developers switch to alternative frameworks.

Recently this has been (more-or-less) addressed by the Angular team with standalone components.

Standalone components

A standalone component does not require a module to be bootstrapped. Instead, you set the `standalone` configuration to `true` for the app root component.

You can use `imports` to import both other standalone components, directives and pipes and non-standalone modules.

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

@Component({
  selector: 'app-component',
  standalone: true,
  imports: [CommonModule],
  template: `...`,
})
export class App {}
```

Standalone components

Then, in `main.ts` you import `bootstrapApplication` to bootstrap your application using the root component.

```
import { bootstrapApplication } from
  '@angular/platform-browser';
import { App } from './app.component';

bootstrapApplication(App);
```

Standalone components

If your application needs it, you can pass global providers to your app via the second argument of `bootstrapApplication`.

```
bootstrapApplication(App, {
  providers: [
    {
      provide: BACKEND_URL,
      useValue: 'https://backend.api/'
    },
    // ...
  ]
});
```

Xebia

Multi-project configuration

You can add multiple projects to a workspace, which gives you a **monorepo** (multiple projects in the same file version control system). However, the Angular out of the box configuration is a bit limited. So if you (think you) are going to need this, have a look at `Nx`:

https://nx.dev

Hands-on 10:
create a standalone component

https://github.com/xebia/xebia-angular-training-exercises

Angular CLI

What is the Angular CLI?

“The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications directly from a command shell.”

It’s an essential tool to productively develop Angular apps. Learn to embrace it!

Install the CLI

Node.js comes with the package manager `npm` which makes it easy to install the Angular CLI globally.

```
npm install -g @angular/cli
```

After installation, the tool is available using the `ng` command.

You can ask it for a global overview of commands, and for help per command.

```
ng help
ng generate --help
```

Start a new project

With `ng new` you create a new workspace with a single project as shown before.

A new folder with the given name is created and populated with default structure. The given name will be used in various places when creating a new app.

```
ng new my-first-project
```

Add an external library

`ng add` lets you add a published library to your workspace, including the configuration that is needed to use that library.

Often you will get asked questions to set up the library accordingly.

Using `ng add` for Angular libraries is preferred over installing directly via `npm`.

```
ng add @angular/material

ℹ️ Using package manager: npm
✔ Found compatible package version: @angular/material@13.0.0.
✔ Package information loaded.

The package @angular/material@13.0.0 will be installed and executed.
Would you like to proceed? Yes
✔ Packages successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme:
❯ Indigo/Pink
  Deep Purple/Amber
  Pink/Blue Grey
  Purple/Green
  Custom
```

Update an existing project

To update libraries in your current workspace to a more recent version, use `ng update`. This is possible for both the core Angular libraries and third-party ones.

You can control to which version the library will be updated.

Make sure you have committed all your changes in Git before starting an update!

```
// update to the stable release
ng update @angular/cli @angular/core

// update to a major version
ng update @angular/cli@^14 @angular/core@^14

// update an external lib
ng update @angular/material
```

https://update.angular.io

Build the project

With `ng build` you compile an Angular application into a distribution. This is the set of files that you need to serve to your users to show the app.

Underwater, it uses Webpack to bundle the compiled sources and process the assets.

Use the `--configuration` flag to specify the environment. By default, a build is for production.

```
ng build

ng build --configuration my-custom-env
```

Run the project

During local development, `ng serve` runs an HTTP server in the background and rebuilds the application every time you change a file.

```
ng serve
```

Using `--port` you can specify to which port it will listen.

```
ng serve --port 1337
```

If you are adventurous you can enable hot module replacement, which means Angular will push changed code to your browser, instead of refreshing the page.

```
ng serve --hmr
```

Xebia

Generate boilerplate

The most useful and largest command is `ng generate`. It can generate all Angular items for you, most of them with a sample unit test. Some items are by default automatically imported and wired in a module configuration.

The list of generation templates is quite long. You will probably encounter the following commands often.

```
ng generate component [name]
ng generate directive [name]
ng generate pipe [name]
ng generate service [name]
ng generate module [name]
```

ng generate component

This creates a new component definition.

It is by default added to the module declarations in the **parent** folder of the component. If no module exists there, it is not added.

You need to specify the component name in CamelCase, but modules in dash-case name (as their pathname).

```
// in project root:
ng generate component ItemList
// generates src/app/item-list/, added to AppModule

// in root/src/app (with items/ module present)
ng generate component items/ItemList
// generates ./items/item-list/, added to ItemsModule

// in root/src/app
ng generate component items/ItemList --module app
// generates ./items/item-list/, added to AppModule
```

ng generate directive

This command creates a new directive definition. It is by default added to the module declarations in the **same** folder.

The directive is by default placed in the app root folder. If you want to place it under a component or any other folder, prefix it with a path.

```
// in project root:
ng generate directive Glow
// src/app/glow.directive.ts

// in root/src/app/items
ng generate directive item-list/Glow
// src/items/app/item-list/glow.directive.ts
```

Xebia

ng generate pipe

You create a new pipe definition with this command. Again, it is by default added to the module declarations in the **same** folder.

Also, pipes are placed by default in the app root folder. Use a prefix path to place it under a different folder.

```
// in project root:
ng generate pipe Truncate
// src/app/truncate.pipe.ts

ng generate pipe pipes/Truncate
// src/app/pipes/truncate.pipe.ts
```

ng generate service

This creates a new service definition. It is by default **not** injected, you have to take care of this yourself.

```
// in project root:
ng generate service FileUpload
// ends up in src/app/file-upload.service.ts
```

ng generate module

This creates a new module definition. It is by default **not** added to the module hierarchy.

When creating modules, make sure to place it correctly.

```
// in project root:
ng generate module Items
// ends up in src/app/items/

// in root/src
ng generate module Items
// ends up in src/items/
```

Linting

If you work with multiple team members on the same codebase, it's a good practice to have some rules about code style.

Linting is the process to validate code style. By running the linter, all code files are checked against some rules.

```
ng lint

Linting "my-beautiful-app"...

my-beautiful-app/src/app/app.component.ts
  4:13  error  The selector should be kebab-case and
start with one of these prefixes: "app"
@angular-eslint/component-selector

✖ 1 problem (1 error, 0 warnings)

Lint errors found in the listed files.
```

Run tests

ng test

ng e2e

Hands-on 11:
create a production build

https://github.com/xebia/xebia-angular-training-exercises

Angular Router

What does a router do?

To mimic the transition of pages, a single page app shows or hide components instead of getting a new HTML page from the server. It should also update the browser URL to represent the current navigation state.

To achieve this in Angular, you use the Router module.

Setting up Angular Router

If you create a new app from scratch with the CLI, you can generate the necessary Router code by choosing to add the Router or by passing the `--routing` flag.

By default, Angular places the router in its own module. This module is added to the imported modules in the app root module.

```
ng new routing-app --routing
```

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from
  '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Define routes

You add route configuration objects to the `routes` array. Each route object specifies which URL path matches which component.

The order of this array is important. The router goes over the list from first to last entry and stops when a matching path is found.

```
import { FirstComponent } from
  './first.component';
import { SecondComponent } from
  './second.component';

const routes: Routes = [
  { path: 'first', component: FirstComponent },
  { path: 'second', component: SecondComponent },
];
```

Xebia

Define routes

You can match with route parameters. These parameters can be retrieved from the component constructor.

```
{ path: 'user/:id', component: UserComponent },
```

You can add redirects, to keep on supporting legacy paths while having a new one in place.

```
{ path: 'users/:id', redirectTo: 'user/:id' }
```

You can also match an empty path, and for instance redirect it to a default non-empty one.

```
{ path: '', redirectTo: 'home', pathMatch: 'full' },
```

Xebia

RouterLink directive

To render navigation links, you use the `routerLink` directive on an anchor element.

The directive also takes care of cancelling the normal browser navigation behavior.

```
<a routerLink="first">First Component</a>
<a routerLink="second">Second Component</a>
```

RouterLink directive

If you need to pass in a route params, bind `routerLink` with a array and pass in each path part of the URL you want to match.

```
<a [routerLink]="['user', user.id]">
  User {{ user.name }} detail
</a>
```

RouterLinkActive directive

You can conditionally add a class to the active router link using the `routerLinkActive` directive.

```
<a routerLink="users" routerLinkActive="active">
  Users
</a>
```

Xebia

Router outlet

But where is the matching or ‘active’ component made visible?

It is rendered by the `router-outlet` component. This component is exported by the `RouterModule`.

```
<router-outlet></router-outlet>
```

Wildcard route

What happens if none of the routes match? Then, nothing renders in the router outlet.

If you want to prevent this, you can either match a redirect or a ‘404’ component as the last route.

```
{ path: 'first', component: FirstComponent },
{ path: 'second', component: SecondComponent },
{ path: '**', redirectTo: 'first' },
```

```
{ path: 'first', component: FirstComponent },
{ path: 'second', component: SecondComponent },
{ path: '**', component: PageNotFoundComponent },
```

Page title

You can provide a static page title to each route config.

If you need to have a dynamic title, you can implement a resolver function that is called just before the new component is activated.

```
{
  path: 'first/:id',
  title: 'First Component',
  component: FirstComponent,
},
{
  path: 'second',
  title: ({ params }) => `ID: ${params['id']}`,
  component: SecondComponent
}
```

Xebia

Nested routes

You can put a router outlet in a component rendered in a router outlet.

For this to work, you define route children in a parent route config.

In this example, navigating to `first`, `first/child-a` and `first/child-b` is allowed.

```
{
  path: 'first',
  component: FirstComponent,
  children: [
    {
      path: 'child-a',
      component: ChildAComponent,
    },
    {
      path: 'child-b',
      component: ChildBComponent,
    },
  ],
},
```

Lazy loading route modules

To show how lazy loading works, let's use the CLI to generate a new project with routing and then generate lazy-loading modules using the `--route` and `--module` flags.

```
ng new lazy-loading-app --routing
cd lazy-loading-app
ng generate module alpha --route alpha --module app
ng generate module beta --route beta --module app
```

Lazy loading route modules

The generate CLI adds the `alpha` and `beta` route to the `AppRoutingModule`.

Note that it import the modules dynamically using `import()`. This way Angular loads the module at runtime when the user navigates to those routes.

```
const routes: Routes = [
  {
    path: 'alpha',
    loadChildren: () =>
      import('./alpha/alpha.module').then((m) => m.AlphaModule),
  },
  {
    path: 'beta',
    loadChildren: () =>
      import('./beta/beta.module').then((m) => m.BetaModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Lazy loading route modules

The app component template does not require any different syntax for lazy loading. Just use the `routerLink` directive to navigate to the module and Angular loads it.

```
<button type="button" routerLink="alpha">Alpha</button>
<button type="button" routerLink="beta">Beta</button>
<button type="button" routerLink="">Home</button>

<router-outlet></router-outlet>
```

Router forRoot / forChild

To make sure that the `RouterModule` only provides a single `Router` to all components, two static helper methods are available on the `RouterModule`.

Make sure to import the `forRoot` result in the root module and use `forChild` for all non-root modules.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

```
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AlphaRoutingModule { }
```

Xebia

Component route info

Now, let's look at the component side.

If you inject `ActivatedRoute` then you can subscribe to route parameter changes.

Angular reuses a component if only the route parameter changes, so you need to observe the change instead of relying on the component lifecycle.

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

@Component({ ... })
export class FirstComponent {
  constructor(private route: ActivatedRoute) {}

  id$ = this.route.params.pipe(
    map(params => params['id'])
  );
}
```

Hands-on 12: use the router

https://github.com/xebia/xebia-angular-training-exercises

Angular Forms

A tale of two forms

Angular provides two flavors of form development. One is template driven, where you declare your form model implicitly in a template. The other is reactive, where you build your form model explicitly in code and bind a template to it.

You can use both as they have feature parity. However, reactive forms are more scalable than template-driven forms. Template-driven forms are less suited for complex data models and limit reusability and testing.

Rule of thumb: for simple data models, consider template-driven forms. Else use reactive forms.

Template-driven form: setup

To use template-driven forms, import the `FormsModule` first and add it to the root module imports.

```
import { FormsModule } from '@angular/forms';

@NgModule({
  // ...
  imports: [BrowserModule, FormsModule],
})
export class AppModule {}
```

Template-driven form: data model

Then, create a data model. This can be a quite complex model, but for this example let's keep it simple.

You can use any Java/TypeScript concept. Here an object literal is used as a data store, and the possible types are stored in a string array.

```
type Item = {
  name?: string;
  type?: string;
};

@Component({ ... })
export class ExampleFormComponent {
  model: Item = {};

  types = ['solid', 'liquid', 'gas'];
}
```

Template-driven form: template

Now, add some form components. Both `input` and `select` value and event listeners are bound via the `ngModel` two-way binding. It requires a `name` attribute.

Note that you still need to write HTML form elements, Angular does not generate forms for you.

A good form component library can help you reduce a lot of boilerplate code!

```
<input
  name="name"
  required
  [(ngModel)]="model.name"
/>

<select
  name="type"
  required
  [(ngModel)]="model.type"
>
  <option value="">Choose an option</option>
  <option *ngFor="let type of types" [value]="type">
    {{ type }}
  </option>
</select>
```

Template-driven form: show validation

If you want to show validation errors, you can use a template variable to refer to the `ngModel` behind a form input and pull the necessary data from it.

The `valid` property tells you if the input is valid, and the `pristine` property is true as long as the user has not modified the input data.

```
<input
  id="name"
  required
  [(ngModel)]="model.name"
  name="name"
  #nameModel="ngModel"
/>
<div
  *ngIf="!nameModel.valid && !nameModel.pristine"
  class="alert alert-danger"
>
  Name is required
</div>
```

Template-driven form: submit

Finally, you can wrap the input elements in a form element. To intercept the submit, bind the `ngSubmit` event property with your own method.

You can introduce a template variable to refer to the `ngForm` model and check if the form is completely valid.

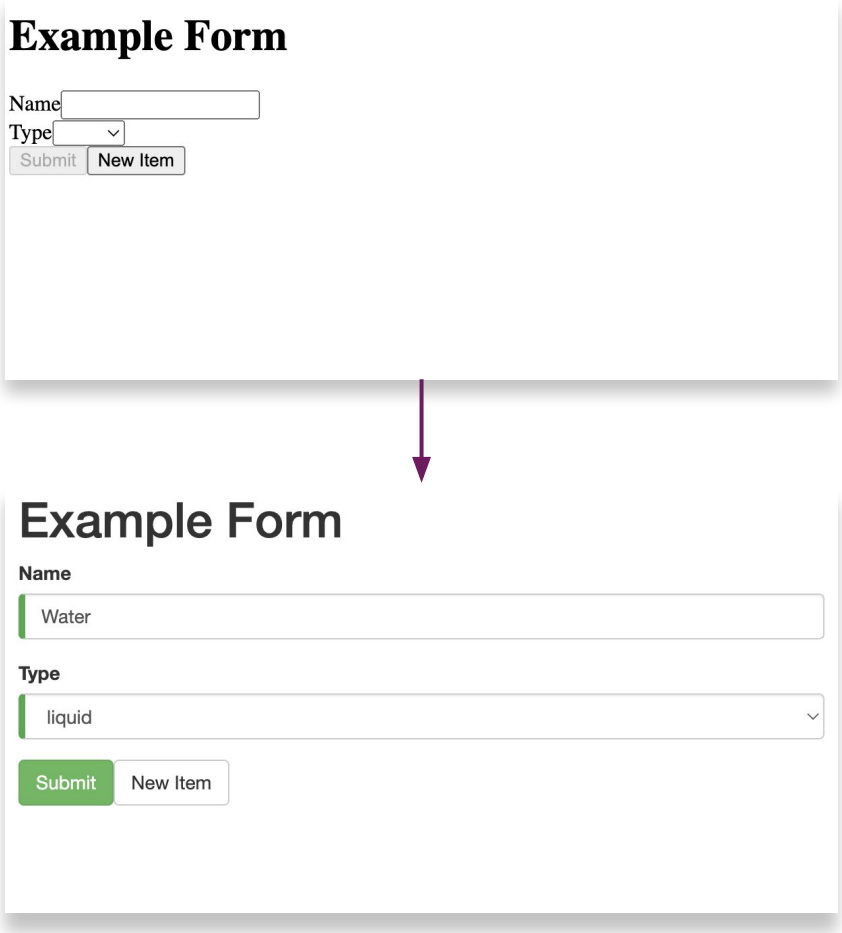
```
<form (ngSubmit)="onSubmit()"
      #exampleForm="ngForm">
  <!-- form elements -->
  <button
    type="submit"
    [disabled]="!exampleForm.form.valid"
  >
    Submit
  </button>
</form>
```

Xebia

Template-driven form

Needless to say, the resulting form does not look very enticing. By adding (lots of) CSS styles you can improve the form appearance.

Note that *how* components are styled is not an Angular concern. You can use a pre-styled component library (such as Angular Material) or add a separate CSS style library (such as Bootstrap).



Xebia

Reactive form: setup

To use reactive forms, import the `ReactiveFormsModule` first and add it to the root module imports.

Your app can import both form modules at the same time, but it's not preferred due to the increase of bundle size.

```
import {
  ReactiveFormsModule } from '@angular/forms';

@NgModule({
  // ...
  imports: [BrowserModule, ReactiveFormsModule],
})
export class AppModule {}
```

Reactive form: data model

With reactive forms, you create a separate form model in code.

A `FormControl` holds a single value and optionally has one or more validators attached.

With a `FormGroup`, you group multiple `FormControl`s together. You can nest `FormGroup`s as well.

```
@Component({ ... })
export class ExampleFormComponent {
  exampleForm = new FormGroup({
    name: new FormControl('', Validators.required),
    type: new FormControl('', Validators.required),
  });

  types = ['solid', 'liquid', 'gas'];
}
```

Reactive form: template

To hook up the view, you use the `formGroup` directive and bind it to the `exampleForm` property.

Now you can bind inputs in the form using `formControlName`. As you can see, reactive form templates are slightly easier to read!

```
<form [formGroup]="exampleForm">
  <input type="text" formControlName="name" />

  <select formControlName="type">
    <option value="">Choose an option</option>
    <option *ngFor="let type of types" [value]="type">
      {{ type }}
    </option>
  </select>
</form>
```

Xebia

Reactive form: show validation

If you want to show explicit validation messages, you do this in a similar fashion as template-driven forms.

To get a reference to the backing control, instead of using a template variable, you create a getter and use the `FormGroup` `get` method.

```
<input type="text" formControlName="name" />
<div
  *ngIf="nameControl.invalid && nameControl.dirty"
  class="alert alert-danger"
>
  Name is required
</div>
```

```
get nameControl() {
  return this.exampleForm.get('name')!;
}
```

Reactive form: submit

To catch the submit event, again bind a method to the `ngSubmit` property.

The form group can be used in a template expression to determine the form validity.

In the component, you can retrieve a snapshot of the form values via the `value` property.

```
<form [formGroup]="exampleForm"
      (ngSubmit)="onSubmit()">
  <!-- form elements -->
  <button
    type="submit"
    [disabled]="!exampleForm.valid"
  >
    Submit
  </button>
</form>
```

```
onSubmit() {
  console.log(this.exampleForm.value);
}
```

Xebia

Reactive form: using FormBuilder

You can use `FormBuilder` to reduce the amount of duplication in the form group construction.

Inject the dependency to use it. It especially shines when you have a longer, more complex form.

```
exampleForm = new FormGroup({
  name: new FormControl('', Validators.required),
  type: new FormControl('', Validators.required),
});
```

```
constructor(private fb: FormBuilder) { }

exampleForm = this.fb.group({
  name: ['', Validators.required],
  type: ['', Validators.required],
});
```

Xebia

Reactive form: validators

Angular comes out of the box with these useful reactive form validators.

If you need to, you can define your own validators as well.

```
min(min: number)
max(max: number)
required()
requiredTrue()
email()
minLength(minLength: number)
maxLength(maxLength: number)
pattern(pattern: string | RegExp)
```

Reactive form: custom validator

You can write your own validator by implementing the `ValidatorFn` type interface.

Such a function takes an `AbstractControl` (the base of all reactive control types) and returns either `null` or an object literal with errored values.

You can use the key in the error object to identify the error when showing an error message.

```
function forbiddenValidator(text: string): ValidatorFn {
  return (control: AbstractControl<string>) => {
    const invalid = control.value.includes(text);
    return invalid ? {
      forbidden: {
        value: control.value
      }
    } : null;
  };
}


new FormControl('', [
  Validators.required,
  forbiddenValidator('joe'),
]),
```

Xebia

Reactive form: dynamic forms

What if your form should be able to grow based on user input? For instance: adding new user rows.

Angular provides `FormArray` for this. You can dynamically add new recurring parts of your form by wrapping it in a `FormArray`.

```
exampleForm = new FormGroup({
  users: new FormArray([this.newUserFormGroup()]),
  // ... other fields
});

get users() {
  return this.exampleForm.get('users') as FormArray<FormGroup>;
}

addUser() {
  this.users.push(this.newUserFormGroup());
}

newUserFormGroup(): FormGroup {
  return new FormGroup({
    username: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
  });
}
```

Reactive form: dynamic forms

Now you can render each user row by looping over the `FormArray` entries.

Because it contains `FormGroup` instances, you need to pass it to the `formGroup` directive. Then you can use `formControlName` again just as you saw in the example earlier.

```
<div *ngFor="let userForm of users.controls"
     [formGroup]="userForm">
  <label>
    Username:
    <input type="text" formControlName="username" />
  </label>
  <label>
    Email:
    <input type="text" formControlName="email" />
  </label>
</div>
```

Hands-on 13: create a reactive form with validation

https://github.com/xebia/xebia-angular-training-exercises

Angular HTTP Client

What is the HTTP client?

Angular provides a client HTTP API for Angular applications.

The HTTP client service offers the following features:

- The ability to request typed response objects
- Streamlined error handling
- Testability features
- Request and response interception

Add the client module

To use the HTTP client, first add the `HttpClientModule` to the imports of your app.

Now your app is ready to use the client in your services.

```
import {
  HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Request data from server

You inject `HttpClient` in any service that needs it. Then you use the `get` method to fetch data from a server.

It returns an `Observable` that emits the requested data when the response is returned.

The first argument is the URL that you want to fetch, the second is a config object. By default, the client tries to fetch a JSON response and returns the parsed body.

```
export class PokemonService {
  constructor(private http: HttpClient) {}

  getPokemon(id: number) {
    return this.http.get<Pokemon>(
      `https://pokeapi.co/api/v2/pokemon/${id}`);
  }
}

export type Pokemon = {
  name: string;
  sprites: { front_default: string; };
};
```

Handling asynchronous data

Now let's look at how to handle the data using observables.

In a component, you introduce state in an observable `BehaviorSubject`. It holds the id that you want to pass to the `getPokemon` function. Every time the id changes, you want to refetch the data, so you use `switchMap` to 'switch' each id to a request.

Then you create 2 observables for the name and image URL.

```
export class PokemonComponent {
  private idSubject = new BehaviorSubject(1);
  private pokemon$ = this.idSubject.pipe(
    switchMap(id => this.ps.getPokemon(id))
  );
  name$ = this.pokemon$.pipe(map(p => p.name));
  sprite$ = this.pokemon$.pipe(map(
    p => p.sprites.front_default)
  );

  constructor(private ps: PokemonService) {}

  next() {
    this.idSubject.next(this.idSubject.getValue() + 1);
  }
}
```

Uh oh...

Xebia

Handling asynchronous data

Everything goes well, you use the `async` pipe to observe the name and image URL. But then you look at the network panel...

2 requests? For the same URL? Why?

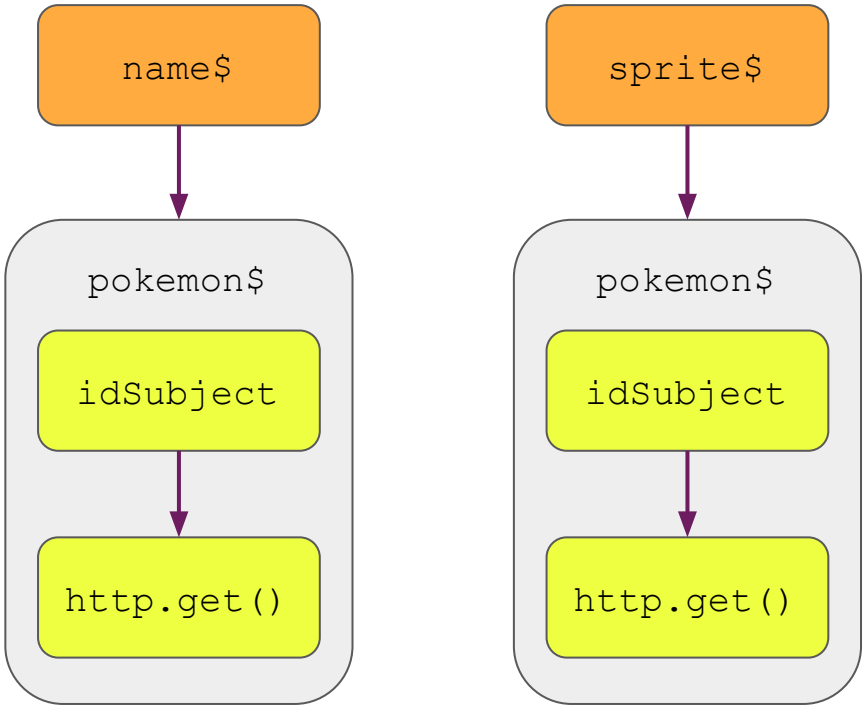
```
<h2>{{ name$ | async | titlecase }}</h2>
<img [src]="sprite$ | async" />

<button (click)="next()">Next</button>
```

Handling asynchronous data

Our `pokemon$` observable is being subscribed to twice, once by `name$` and once by `sprite$`.

However, there is no magic deduplication happening. Subscribing will create two separate paths.

How to fix this?

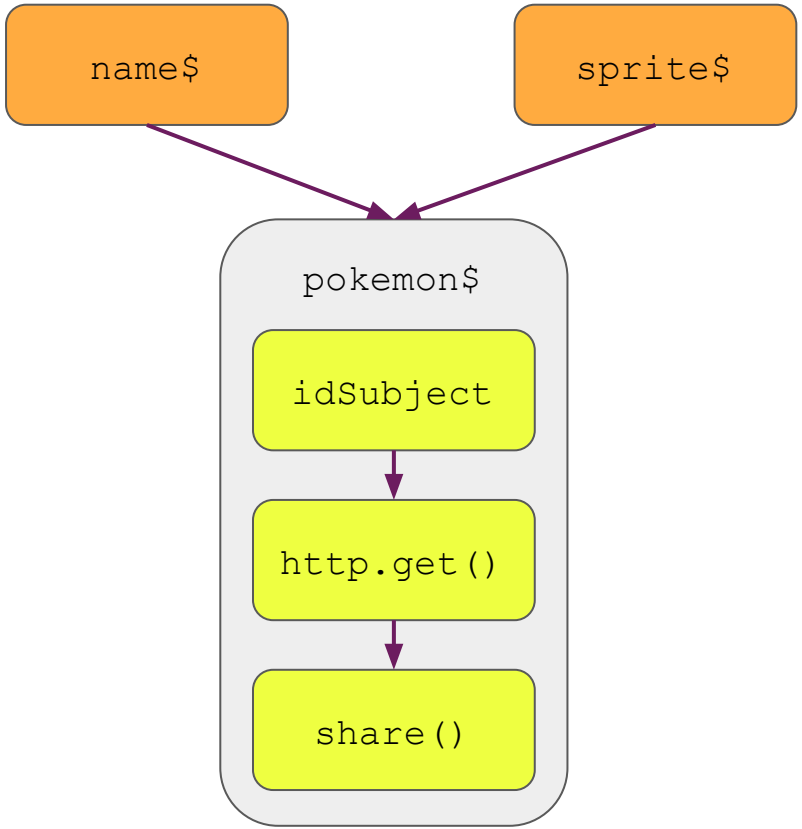


Xebia

Handling asynchronous data

Solution: share the observed response data!

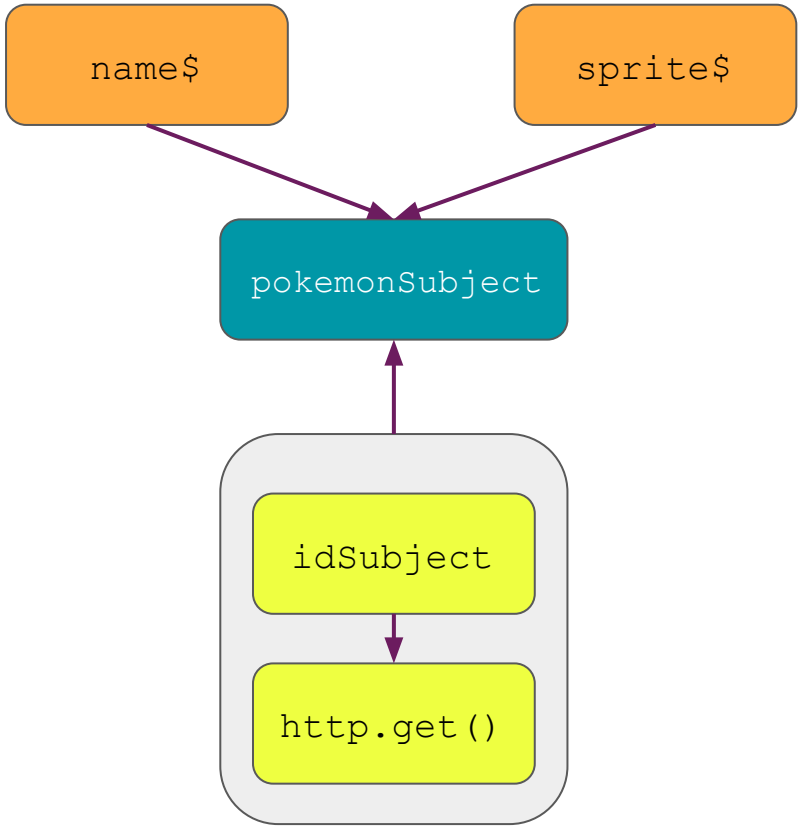
We can go for the RxJS `share` pipe operator. But on day 1 you learned the preferred way to deal with sharing data.



Handling asynchronous data

Instead of using the `share` operator you need to introduce a new `Subject` instance.

The `Subject` takes care of multiple subscriptions in the template. To emit data to the `Subject`, you subscribe the producer to it.



Xebia

Handling asynchronous data

To fix the double network requests, the `pokemonSubject` is placed between the `idSubject` and the `name$` and `sprite$` observables.

When you check your network panel, now only one request will happen on every change of `idSubject`.

```
private idSubject = new BehaviorSubject(1);
private pokemonSubject = new Subject<Pokemon>();
name$ = this.pokemonSubject.pipe(
  map(p => p.name)
);
sprite$ = this.pokemonSubject.pipe(
  map(p => p.sprites.front_default)
);

constructor(private ps: PokemonService) {
  this.idSubject
    .pipe(switchMap(id => this.ps.getPokemon(id)))
    .subscribe(this.pokemonSubject);
}
```

Sending data to server

Next to getting data from the server, you can create POST, PUT, PATCH and DELETE requests with their respective methods.

Make sure that you subscribe to the returned observer. The Angular HTTP client is *cold*, which means it will only do work if there is at least one subscriber.

```
postMetadata(meta: Metadata): Observable<Metadata> {
  return this.http.post<Metadata>('api/meta', meta);
}

deleteUser(user: User) {
  return this.http.delete<User>(`api/user/${user.id}`);
}
```

Handling errors

You can add error handling by adding the `catchError` operator.

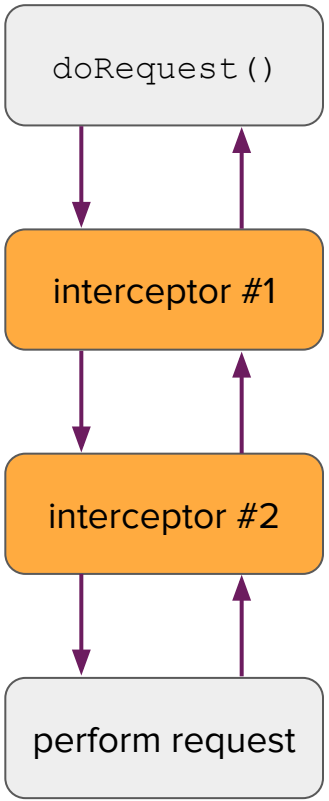
The error handler function gets detailed information about the error and lets you for instance log the response code. You can then return a user-facing error with an adjusted status message.

```
getPokemon(id: number) {
  return this.http
    .get<Pokemon>(
       `https://pokeapi.co/api/v2/pokemon/${id}`)
    .pipe(catchError(this.handleError));
}

handleError(error: HttpErrorResponse) {
  console.error(
    'statuscode:', error.status,
    'error:', error.error
  );
  return throwError(() => new Error('No.'));
}
```

HTTP interceptors

You can declare interceptors that inspect (and optionally transform) any HTTP requests coming from your application, and inspect (and optionally transform) the server response on their way back to the application.

The main usage of these interceptors is authentication and logging.



Xebia

HTTP interceptors

You create an interceptor by implementing the `HttpInterceptor` interface.

Here is a ‘no-op’ interceptor that does not change the request or response.

It receives the original request as a parameter and returns an observable (created by the `handle` call) that asynchronously emits the server response.

```
@Injectable()
export class NoopInterceptor implements HttpInterceptor {
  intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```

Xebia

HTTP interceptors

The `next` object is the next item in the chain of interceptors. The final one sends the request to the server and receives the response.

An interceptor can skip the call to `handle` and return its own observable with an artificial server response, as shown in this example.

```
export class CacheInterceptor implements HttpInterceptor {
  constructor(private cache: RequestCache) {}

  intercept(req: HttpRequest<unknown>, next: HttpHandler) {
    const cachedResponse = this.cache.get(req);
    if (cachedResponse) {
      return of(cachedResponse);
    }

    return next.handle(req).pipe(
      tap((event) => {
        if (event instanceof HttpResponse) {
          this.cache.put(req, event);
        }
      })
    );
  }
}
```

HTTP interceptors

A use case for interceptors is authorization.

Because interceptors are asynchronous, you can check for an authorization token first, and if not present, perform a separate request to get a token.

Because requests are immutable, use the `clone` method to make a copy. It comes with several helpers to for instance easily set a header.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(
    req: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    const authToken = this.auth.getAuthorizationToken();

    const authReq = req.clone({
      setHeaders: { Authorization: authToken },
    });

    return next.handle(authReq);
  }
}
```

HTTP interceptors

To add interceptors to the HTTP client, create an array of interceptor providers. This represents the chain of interceptors that each request will go through, starting from the first.

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { NoopInterceptor } from './noop-interceptor';
import { CacheInterceptor } from './cache-interceptor';

export const interceptorProviders = [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: NoopInterceptor,
    multi: true
  },
  {
    provide: HTTP_INTERCEPTORS,
    useClass: CacheInterceptor,
    multi: true
  },
];
```

HTTP interceptors

Then you add the providers to the AppModule providers.

```
import { interceptorProviders } from
  './interceptorProviders';

@NgModule({
  declarations: [...],
  imports: [BrowserModule, HttpClientModule],
  providers: [interceptorProviders],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Hands-on 14:
interact with external API

https://github.com/xebia/xebia-angular-training-exercises

Testing

Built in testing

Angular provides APIs to test your application. They are verbose, but they do the job.

You can reduce this ‘boilerplate’ code by writing reusable functions (and introduce another set of things to learn for your fellow coworkers)

Or...

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement as HTMLElement;
    expect(compiled.querySelector('.content')?.textContent)
      .toContain('is running!');
  });
});
```

Xebia

Introducing Angular Testing Library

The Angular Testing Library is a very lightweight solution for testing Angular components. It provides light utility functions on top of the DOM Testing Library in a way that encourages better testing practices.

Its primary guiding principle is:

The more your tests resemble the way your software is used, the more confidence they can give you.

Testing with ATL

Let's take a look at a simple component test.

ATL provides a `render` function that is very flexible and renders a single component. You can provide the property inputs, configure its dependencies and its environment.

Then, via `screen` you get DOM elements and perform certain expectations.

```
import { render, screen } from '@testing-library/angular'
import { CounterComponent } from './counter.component.ts'

test(renders counter', async () => {
  await render(CounterComponent, {
    componentProperties: {counter: 5},
  })

  expect(screen
    .getByText('Current Count: 5')).toBeInTheDocument()
})
```

Xebia

Firing events

Using `fireEvent` you can trigger all native events that could trigger, such as a click, focus and keystroke.

`fireEvent` does not let your code wait for its effect.

```
import {
  render,
  screen,
  fireEvent
} from '@testing-library/angular'

test('should increment the counter on click', async () => {
  await render(CounterComponent, {
    componentProperties: { counter: 5 },
  })

  fireEvent.click(screen.getByText('+'))

  expect(screen.getByText('Current Count:
6')).toBeInTheDocument()
})
```

Xebia

User actions

If you need to mimic user behavior, you can use another library from `@testing-library`.

With `user-event` you need to await the result of a click and then provide certain expectations. It performs actions closer to how they would happen in a browser.

```
import { render, screen } from '@testing-library/angular';
import userEvent from '@testing-library/user-event';

import { Some } from './some.component';

test('some test', async () => {
  const user = userEvent.setup();
  await render(SomeComponent);
  const incrementControl = screen.getByRole('button', {
    name: /increment/i
  });

  await user.click(incrementControl);

  // ... perform action after Angular render cycle ...
});
```

Xebia

Testing forms

The `userEvent` helper makes it straightforward to implement form tests.

```
const nameControl = screen.getByRole('textbox', {
  name: /name/i });
const errors = screen.getByRole('alert');

expect(errors).toContainElement(
  screen.queryByText('name is required'));

expect(nameControl).toBeInvalid();

await user.type(nameControl, 'Frank');

expect(screen.queryByText(
  'name is required')).not.toBeInTheDocument();
```

Remember: test how a user uses your software

The API does not allow you to:

- manipulate internal state (that’s not what a user does)
- manage Angular-specific internals and effects (the lib is cross-framework)
- manage the Angular component lifecycle directly

Angular + Cypress + Cypress Testing Library = 💖

A logical choice when using Angular Testing Library is to use the same style in your end-to-end tests. The Cypress Testing Library makes this possible!

Hands-on 15: Testing with ATL

https://github.com/xebia/xebia-angular-training-exercises

Angular 16: switch to Jest

- Open your `angular.json` and replace the builder in `test`: `@angular-devkit/build-angular:karma` to `@angular-devkit/build-angular:jest`
- Install the necessary dependencies with `npm install -D jest @types/jest jest-environment-jsdom`
- Edit the `tsconfig.spec.json` and replace the value `jasmine` in the property `types` with `jest`

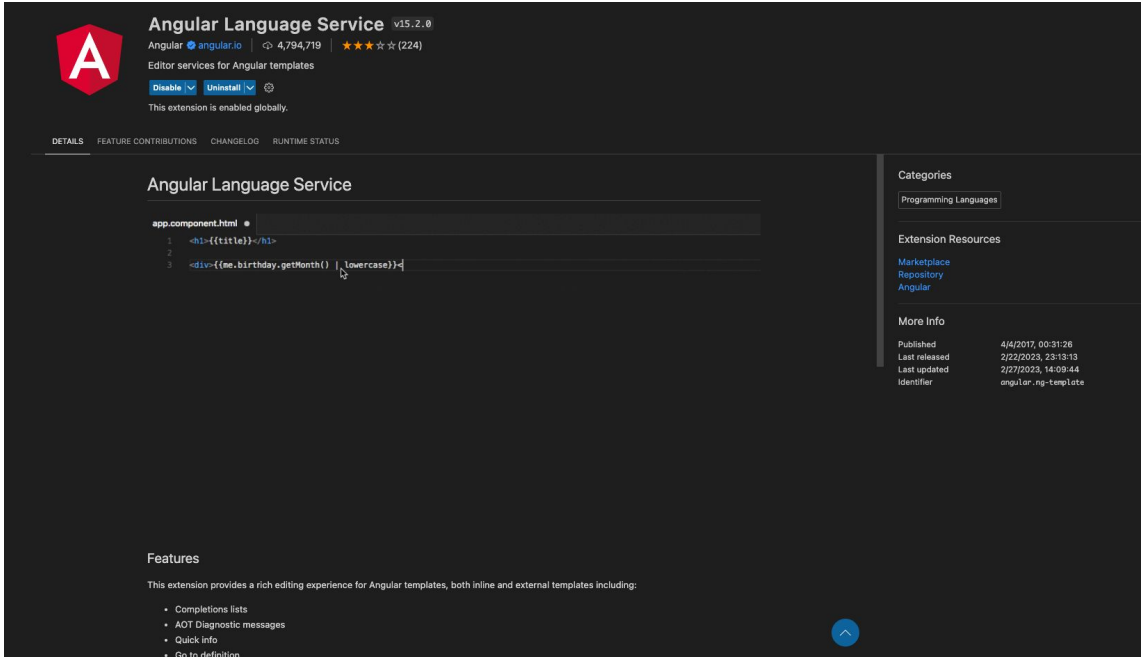
Future of testing with Angular

Development tools

VSCode Angular extension

For people using VSCode, make sure to install the [Angular Language Service](#) extension.

This extension provides a rich editing experience for Angular templates.



Xebia

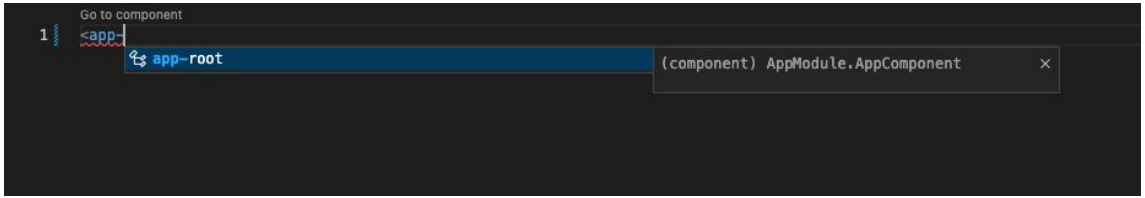
VSCode Angular extension

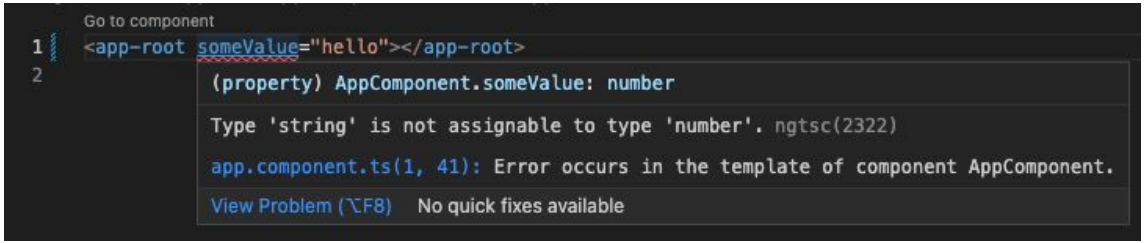
From a template, you get completion lists of components.

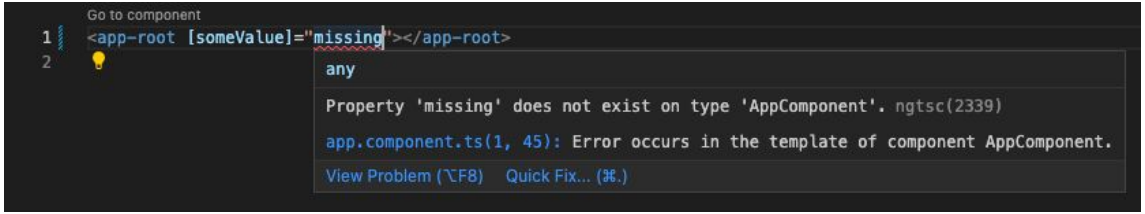
You can Ctrl/Cmd-click on any component or directive to jump to its definition.

The extension checks if the component inputs are of the correct type.

It also checks template expressions for any problems.



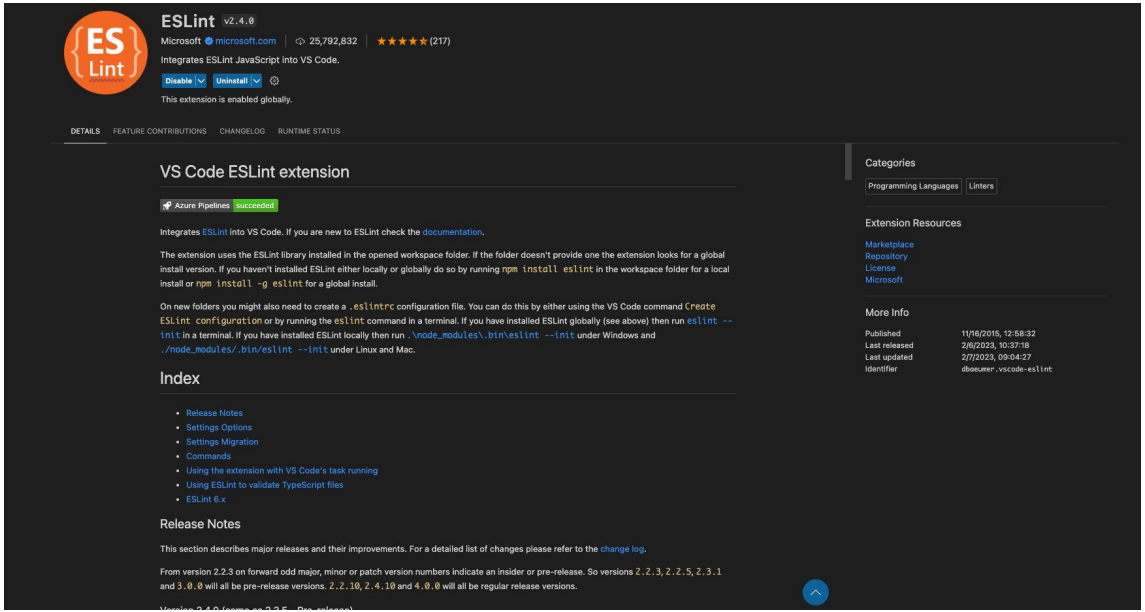




Xebia

VSCode ESLint extension

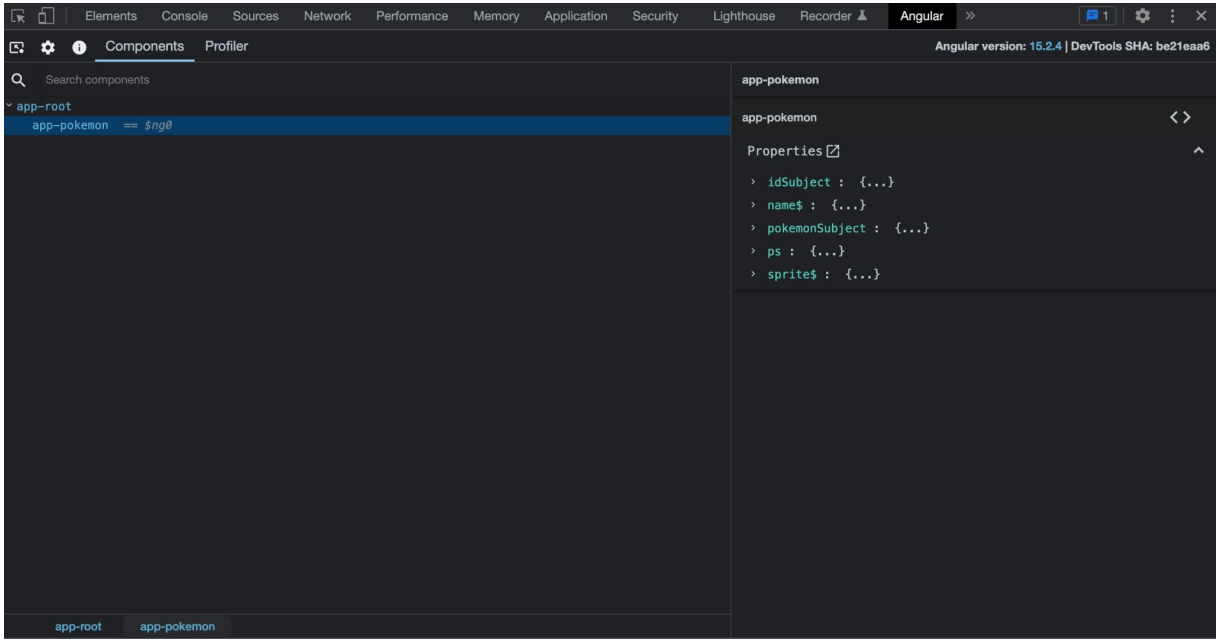
If your project is configured for linting (i.e. you ran `ng lint` and installed the dependencies), there's an extension for VSCode that shows linting issues inline.



Chrome DevTools Angular extension

Using the Chrome DevTools Angular extension, you can examine the component tree of your application, jump to source and check the current state of public properties.

Also, you can profile your Angular application and detect slow components by looking at different performance graphs.



Xebia

Hands-on 16:
spot the bug!
A detective story...

https://github.com/xebia/xebia-angular-training-exercises

Good practices

Keep Angular up to date

It is important to regularly update the Angular libraries:

- You get access to new features that might speed up your work
- Security defects are actively being fixed, not updating leaves your app potentially vulnerable
- Time spent on small upgrades often is less than one big upgrade in a while
- You can prevent breaking changes from becoming big blockers

Follow the Angular Style Guide

The Angular team created the Angular Style Guide to make it easier to develop Angular application following the same conventions and structure:

https://angular.io/guide/styleguide

It's quite lengthy, opinionated, and in some places mentions deprecated syntax. But it comes with a lot of good advice. Make sure someone in your team reads it!

Performance matters!

Make sure to use these performance improvements:

- use `trackBy` in `ngFor*` for lists that render objects
- use the `async` pipe and `OnPush` change detection where possible
- use lazily loaded feature modules
- use native ECMAScript where possible, instead of libraries such as Lodash

Test automation

Just to reiterate, automated tests:

- Lead to faster uncovering of bugs
- Are a proven method to produce better quality of software
- Make sure that application changes do not cause regressions over time
- Can be set up on function, class, feature or application-level to fit best
- Using Testing Library they have a familiar, similar API

Check-out

Thank you!

It’s been a pleasure!

Presentation tools

https://docs.google.com/presentation

https://romannurik.github.io/SlidesCodeHighlighter/