

Angular Foundation day 1

Alliander

Frank van Wijk & Albert Brand

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
- Development tools
- Good practices

Quick intro to Angular

What is Angular?

“an application-design framework and development platform for creating efficient and sophisticated single-page apps”

Angular includes:

- A component-based framework for building scalable web applications
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more
- A suite of developer tools to help you develop, build, test, and update your code

Components

Components are the building blocks that compose an application. They consist of:

- a TypeScript class describing component behaviour
- + a selector to create components for elements
- + an HTML template to render contents
- + optional CSS styles

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

@Component({
  selector: 'app-hello-world',
  template: `
    <h2>Hello World</h2>
    <p>This is my first component!</p>
  `
})
export class HelloWorldComponent {
  // ...
}
```

Component usage

In another component template,
you write:

```
<app-hello-world></app-hello-world>
```


Component usage

When Angular renders this component, the resulting element structure looks like this:

```
<app-hello-world>  
  <h2>Hello World</h2>  
  <p>This is my first component!</p>  
</app-hello-world>
```

Component templates

Templates declare how a component is rendered.

The rendered elements are automatically updated when the input or state of your component changes.

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

@Component ({
  template: `<p>{{ message }}</p>`
})
export class HelloWorldInterpolationComponent {
  message = 'Hello, World!';
}
```

Template files

You can write templates in separate files.

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

@Component ({
  templateUrl:
    './hello-world-bindings.component.html'
})
export class HelloWorldBindingsComponent {
  canClick = false;
  message = 'Hello, World';

  sayMessage() {
    alert(this.message);
  }
}
```

Template bindings

You can bind component properties to elements in your template. You can also bind native and custom events to methods in the component class.

```
<button  
  type="button"  
  [disabled]="!canClick"  
  (click)="sayMessage()">  
  Trigger alert message  
</button>
```

Template directives

You make templates dynamic by adding directives such as `*ngIf` and `*ngFor`. You can also write your own directives.

```
<div *ngIf="showWarning; else noWarning">  
  <p>This is a warning.</p>  
</div>
```

```
<ng-template #noWarning>  
  <p>Nothing to see here!</p>  
</ng-template>
```

Dependency injection

Angular handles the dependencies of your components.

For instance, this logger service is provided in the root of the application, so it can be injected everywhere.

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

@Injectable({ providedIn: 'root' })
export class Logger {
  writeCount(count: number) {
    console.warn(count);
  }
}
```

Dependency injection

Here the dependency is injected and used in the component.

Angular handles the creation and injection of dependencies at runtime.

```
import { Component } from '@angular/core';
import { Logger } from '../logger.service';

@Component({ ... })
export class HelloWorldInjectionComponent {
  count = 0;

  constructor(private logger: Logger) { }

  onLogMe() {
    this.logger.writeCount(this.count);
    this.count++;
  }
}
```

Angular CLI

You can use a command line tool for frequent tasks.

```
ng add
```

```
ng build
```

```
ng serve
```

```
ng generate
```

```
ng test
```


First-party libraries

Angular is modular and comes with many optional libraries to add new functionality.

- Angular Router
 - Client-side navigation and routing
- Angular Forms
 - Both a template driven and reactive form system
- Angular HTTP Client
 - Enables client-server communication
- Angular Animations
 - Animations based on application state
- ... and others

Hands-on 01: generate your first Angular app

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

Components part I

What is a component?

“an encapsulation of a set of related functions and data”

Components are the main building block for Angular applications.

Each Angular component consists of:

- A TypeScript class that defines *behavior*
- An HTML template that declares *what renders on the page*
- A selector that defines *the query for component construction in other templates*
- Optionally, scoped CSS *styles applied to the template*

What is a component?

A component controls a “patch of screen” called a view.

Each application has one root component and is composed of a component hierarchy.

Angular creates, updates, and destroys components as the application state changes. For instance: by user interaction, API call results, etc.

What is a component?

Multiple components can be grouped together if they are related, to manage the complexity of an app. Angular provides `NgModules` to do so.

Each application has one root module, and modules are hierarchically organized as well. A component can be exported from a module and used in another one.

Create a component

A component is just an ordinary JavaScript class (written in TypeScript).

You use the `@Component` decorator to attach metadata to the class.

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

@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
})
export class HeroListComponent {
  // ...
}
```

Component decorator

Every component requires a selector. The full CSS selector syntax is allowed.

```
@Component({  
  selector: 'app-component-overview',  
})
```

This selector instructs Angular to instantiate this component any time the tag

```
<app-component-overview  
>
```

appears in another template.

Note: you can use any complex selector you like, including attribute and ancestor matching.

Component decorator

To define a template within the component, add a `template` property that contains the HTML you want to use.

```
@Component({  
  selector: 'app-component-overview',  
  template: '<h1>Hello World!</h1>',  
})
```

To define a template as an external file, add a `templateUrl` property.

```
@Component({  
  selector: 'app-component-overview',  
  templateUrl: './component-overview.component.html',  
})
```

Component decorator

To declare styles within the component, add a `styles` property that contains the styles you want to use.

```
@Component({  
  selector: 'app-component-overview',  
  template: '<h1>Hello World!</h1>',  
  styles: ['h1 { font-weight: normal; }']  
})
```

To declare the styles for a component in a separate file, add a `styleUrls` property.

```
@Component({  
  selector: 'app-component-overview',  
  templateUrl: './component-overview.component.html',  
  styleUrls: ['./component-overview.component.css']  
})
```

Note: both passed in as array!

Component state

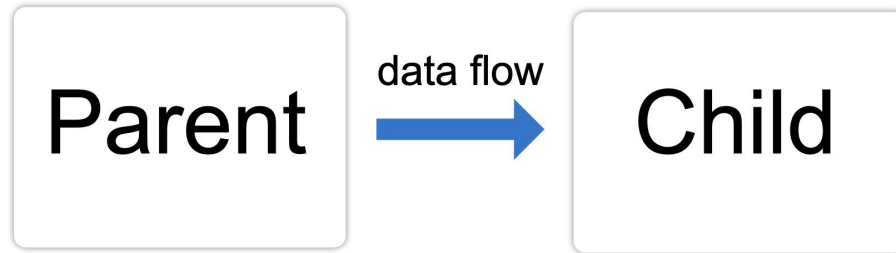
Component state can be contained in object properties. State can be manipulated via methods on the object.

Angular creates and destroys instances of the classes, so the memory is automatically claimed and freed as well.

```
export class HeroListComponent {  
  heroes: Hero[] = [];  
  selectedHero: Hero | undefined;  
  
  selectHero(hero: Hero) {  
    this.selectedHero = hero;  
  }  
}
```

Component data flow

@Input



Component data flow

You define inputs for a component by adding the `@Input()` decorator on a property.

You can provide a custom input name if needed, but it's hardly ever used.

Use the decorated property in templates just as other properties.

Use multiple inputs for multiple “parameters”.

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

@Component({
  selector: 'app-item-detail'
})
export class ItemDetailComponent {
  @Input() item = '';
  @Input('category') categoryName: string;
}
```

Component data flow

Pass in data in the *parent component template* via attributes. Plain attribute values are always passed as **strings**.

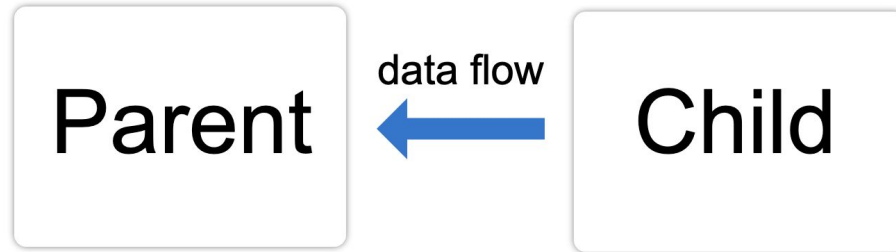
```
<app-item-detail item="tv" category="electronics">  
</app-item-detail>
```

You can pass in component state via bindings, and allow any type of data to be passed.

```
<app-item-detail  
  [item]="selectedItem"  
  [category]="selectedCategory">  
</app-item-detail>
```

Component data flow

@Output



Component data flow

You define outputs by adding the `@Output` decorator on a property that holds an `EventEmitter`.

You can then emit values in the component via the event emitter.

Note the explicit type of the emitted values!

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

@Component({
  selector: 'app-item-output'
})
export class ItemOutputComponent {
  @Output() newItemEvent = new EventEmitter<string>();

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


Component data flow

In the parent component template, you bind the output emitter via event binding using parentheses.

The `$event` variable contains the data that is emitted. Most of the time, it is passed as an argument to an object method.

You can combine both input and output into a single two-way binding expression, which is explained later.

```
<app-item-output (newItemEvent)="addItem($event)">
</app-item-output>
```

```
export class AppComponent {
  addItem(newItem: string) {
    console.log('New item:', newItem);
  }
}
```

Component styling

Use CSS to style any part of your component.

Any selectors, rules, and media queries can be used.

Styles only apply to the component they decorate! There is no inheritance or other passing through of styles by default.

```
@Component({  
  selector: 'app-root',  
  template: `  
    <h1>Tour of Heroes</h1>  
    <app-hero-main [hero]="hero"></app-hero-main>  
  `,  
  styles: ['h1 { font-weight: normal; }']  
})  
export class HeroAppComponent {  
  // ..  
}
```

Component styling

You can move styles to separate files with the `styleUrls` property.

This makes it possible to reuse common styles for multiple components.

You can combine both `styles` and `styleUrls` properties.

Angular supports Sass and Less out of the box: files with the `.scss` and `.less` extension are automatically preprocessed.

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styleUrls: ['./hero-app.component.css']
})
export class HeroAppComponent {
  // ..
}
```

Component styling

Another way of specifying styles is by using a `<style>` tag in the template.

Again, the style is processed in such a way that it is only applied to the current component.

```
<style>
  button {
    background-color: white;
    border: 1px solid #777;
  }
</style>
<h3>Controls</h3>
<button type="button" (click)="activate()">
  Activate
</button>
```

Component styling

Use the `:host` selector to select the root component element.

In this case, the `:host` selector selects the final rendered `<app-host-selector>` element.

This makes it redundant to add a wrapper element.

```
@Component({  
  selector: 'app-host-selector',  
  template: `  
    <h1>Hey!</h1>  
    <div>  
      Multiple elements in template  
    </div>  
  `,  
  styles: [':host { font-style: italic }']  
})  
export class HostSelectorExampleComponent { }
```

Component styling

Use the `:host` selector function form to apply host styles conditionally by including another selector inside parentheses.

The `<div>` element in the host's content becomes bold when the active CSS class is applied to the host element.

Note that custom elements by default are displayed inline. Set to `display: block` if necessary.

```
:host(.active) div {  
  font-weight: bold;  
}
```

```
<app-host-selector class="active">  
</app-host-selector>
```

Component styling

Use the `:host-context` selector function form to apply host styles conditionally based on an ancestor element condition.

The `<div>` element in the host's content becomes bold when the active CSS class is applied to an ancestor of the host element.

```
:host-context(.active) div {  
  font-weight: bold;  
}
```

```
<div class="active">  
  <div>  
    <app-host-selector></app-host-selector>  
  </div>  
</div>
```

Component styling

You can import other CSS files using the `@import` statement.

```
@import './hero-details-box.css';
```

For local imports, use the relative path indicator.

Component styling

In a default project setup, there is one global `styles.css` file where you can place global or reusable styles.

You can add more global style files (also, Sass and Less) via the `angular.json` project configuration file.

Content projection

You can insert, or *project*, the content you want to use inside another component.

Add a `<ng-content>` placeholder element where you want the projected content to appear.

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

@Component({
  selector: 'app-content-basic',
  template: `
    <h2>Single-slot content projection</h2>
    <ng-content></ng-content>
    <footer>The end!</footer>
  `
})
export class ContentBasicComponent {}
```

Content projection

You can now use this component in a template and provide it with child content.

```
<app-content-basic>  
  <p>Is content projection cool?</p>  
</app-content-basic>
```

The rendered component then replaces the `<ng-content>` placeholder and projects the given child content instead.

```
<app-content-basic>  
  <h2>Single-slot content projection</h2>  
  <p>Is content projection cool?</p>  
  <footer>The end!</footer>  
</app-content-basic>
```

Hands-on 02: pass data to a component

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

Templates

What is a template?

A template is a blueprint for a fragment of a user interface (UI).

Angular templates are written in HTML, and special syntax can be used in a template to use Angular-specific features.

Almost all HTML syntax is valid template syntax. However, due to security reasons, Angular ignores the `<script>` tag in templates.

Interpolation

With interpolation, you can embed expressions in templates.

By default, interpolation uses the double curly braces `{{` and `}}` as delimiters.

Here, Angular replaces `currentCustomer` and `itemImageUrl` with the string value of their corresponding component properties.

```
<h3>Current customer: {{ currentCustomer }}</h3>
```

```

```

Expressions

Many types of expressions are possible. Template expressions are written using a subset of JavaScript syntax.

The expressions are evaluated every time an external change has been detected by Angular. This is called data binding.

<p>The sum of 1 + 1 is {{ 1 + 1 }}</p>

<p>A subproperty is {{ hero.name }}</p>

<p>The value is {{ getVal() }}</p>

<p>Ternary: {{ ternaryResult ? 'yes!' : 'no...' }}</p>

<p>Safe path traversal: {{ hero?.name }}</p>

Attributes vs. properties

When pages are rendered in a browser, the HTML is parsed and produces a node tree. This is called the Document Object Model (DOM). **Attributes** in the HTML are transformed to known **properties** in the DOM.

In Angular, the role of HTML attributes is to initialize component state. After that, you deal with DOM properties and events of the target object.

```
<button id="button1" disabled>Save</button>
```



button: HTMLButtonElement

id: 'button1'
textContent: 'Save'
disabled: true
...

Document Object Model

```
document.getElementById('button1').disabled
```

Property binding

Create property bindings by enclosing the attribute name in square brackets.

Angular will evaluate the attribute value as a dynamic expression. Without the brackets, `src` would be set to the string literal `'itemImageUrl'`.

```
<img alt="item" [src]="itemImageUrl">
```

Property binding

Angular handles binding names with case sensitivity.

There are no mappings (such as `className` in React).

There are no implicit camelCase to dash-case conversions here.

```
<tr><td [colspan]="columnCount">...</td></tr>
```

```
<tr><td [colSpan]="columnCount">...</td></tr>
```

Property binding

To use a Boolean value to disable a button, bind the `disabled` attribute to a Boolean property in the component.

```
<button type="button" [disabled]="isUnchanged">  
  Disabled Button  
</button>
```

Property binding

You can use the same syntax to pass any type of data to a custom child component.

Here, `selectedItem` is a property that holds an object.

```
<app-item-detail  
  [item]="selectedItem">  
</app-item-detail>
```

Attribute binding

You create an attribute binding by prefixing the binding name with `attr.` and wrapping it in square brackets.

This is mainly used when HTML attributes have no corresponding DOM properties (such as all `aria-*` attributes).

If you want to remove the attribute from HTML, set its value to `null` or `undefined`.

```
<button type="button" [attr.aria-label]="actionName">  
  With ARIA  
</button>
```

Style binding

Create a style binding by prefixing the binding name with `style.` and providing an expression. The style will apply when the expression is truthy.

For style names, you can choose to use either camelCase or dash-case to refer to a style.

```
<nav [style.background-color]="expression"></nav>
```

```
<nav [style.backgroundColor]="expression"></nav>
```

Style binding

You can add a unit in the binding name.

In this example, `navWidth` is of type number. If it contains for instance `16`, then the resulting width value will be `16px`.

This makes it a little easier to handle unit values.

```
<nav [style.width.px]="navWidth"></nav>
```


Style binding

You can bind multiple styles by binding `style` to:

- a string with a list of styles
- an object with style names as key and style values as value

```
@Component({
  selector: 'app-nav-bar',
  template: `
    <nav [style]="navStyle">
      <a [style]="linkStyle">Login</a>
    </nav>`
})
export class NavBarComponent {
  navStyle = 'font-size: 1.2rem; color: blue;';
  linkStyle = {
    width: '400px',
    height: '100px'
  };
}
```

Class binding

Class bindings make it possible to add or remove classes dynamically. Create a class binding with the `class .` prefix analogous to the `style` syntax.

In this example, Angular adds the class when the bound expression, `onSale` is truthy, and it removes the class when the expression is falsy—with the exception of `undefined`.

```
<div [class.sale]="onSale">On sale now!</div>
```

Class binding

You can bind to multiple classes by binding `class` to:

- a space-delimited string of class names
- an array of class names
- an object with class names as the keys and truthy or falsy expressions as the values

```
@Component({
  selector: 'app-nav-bar',
  template: `
    <nav [class]="navClass">
      <a [class]="linkClass">Login</a>
      <div [class]="divClass">Text</div>
    </nav>`
})
export class NavBarComponent {
  navClass = 'class-1 class-2';
  linkClass = ['class-3', 'class-4']
  divClass = {
    'class-5': true, 'class-6': false
  };
}
```

Event binding

Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

Wrap a target event name within parentheses as attribute name and set its value to a *template statement*.

```
<button (click)="onSave()">Save</button>
```

Template statements

Template statements are blocks of template expressions, chained together with semicolons (;).

It extends the expression syntax with basic assignment (=). However, the following syntax is not allowed:

- `new`
- Increment and decrement operators, `++` and `--`
- Operator assignment, such as `+=` and `-=`
- The bitwise operators, such as `|` and `&`
- The Angular template pipe operator
- referring to anything in the global namespace such as `window` or `global`

Template statements

Rule of thumb:

keep template statements minimal!

- Use method calls for complex logic
- Only perform basic property assignments

Long template statements are harder to test, harder to debug, harder to understand in general.

Event binding

Here is an example of good use of a template statement.

The complexity is mainly offloaded to the `getValue` method and there's only a basic assignment happening.

```
<input  
  [value]="currentItem.name"  
  (input)="currentItem.name = getValue($event)">
```

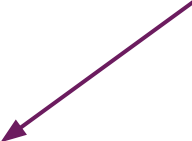
Event binding

The event object, `$event`, is available in event bindings and can be passed to a method to be read.

Native elements emit native DOM events. So, in this case, an `<input>` element emits DOM event objects as `$event`.

```
<input
  [value]="currentItem.name"
  (input)="currentItem.name = getValue($event)">

getValue(event: Event): string {
  return (event.target as HTMLInputElement).value;
}
```



Event binding

If the event belongs to a directive or component, `$event` has the shape that the directive or component produces.

```
<app-item-output (newItemEvent)="addItem($event)">  
</app-item-output>
```

```
addItem(newItem: Item) {  
  console.log('New item:', newItem);  
}
```

Two-way binding

You can listen for events and update values simultaneously between parent and child components.

This two-way binding syntax is a combination of square brackets and parentheses: `[()]`, popularized as ‘bananas in a box’.

```
<app-counter [(count)]="myCounter"></app-counter>  
<div>{{ myCounter }}</div>
```

```
export class AppComponent {  
  myCounter = 1;  
}
```

Two-way binding

To make two-way binding work, the `@Output()` property is connected to the `@Input()` property by adding the `Change` postfix to its name.

For example, if the `@Input()` property name is `count`, the `@Output()` property must be `countChange`.

```
@Component({
  selector: 'app-counter',
  template: `
    <button (click)="inc()">increase</button>
  `
})
export class CounterComponent {
  @Input() count: number;
  @Output() countChange = new EventEmitter<number>()

  inc() {
    this.count += 1;
    this.countChange.emit(this.count);
  }
}
```

Two-way binding

Two-way binding is a shorthand syntax for the following pattern.

HTML form elements don't follow the `Change` postfix convention. To make two-way binding work in forms you need to use the `NgModel` directive.

But that is a topic for day 2!

```
<app-counter [(count)]="myCounter"></app-counter>
```



```
<app-counter  
  [count]="myCounter"  
  (countChange)="myCounter=$event">  
</app-counter>
```

Pipes

- Pipes are simple functions to transform an input value to a transformed value.
- Use pipes to transform strings, currency amounts, dates, and other data for display in a template.
- Pipes can accept parameters to change their behavior.
- Angular provides built-in pipes for typical data transformations, including transformations for internationalization, which use locale information to format data.

Pipes

Use the pipe operator (|) within a template expression along with the name of the pipe.

This example uses one of the built-in pipes.

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

@Component({
  selector: 'app-birthday',
  template: "<p>{{ birthday | date }}</p>"
})
export class BirthdayComponent {
  birthday = new Date(1979, 11, 21);
}
```

Pipes

You can chain pipes so that the output of one pipe becomes the input to the next.

```
<p>The uppercased birthday is  
  {{ birthday | date | uppercase }}  
</p>
```

Pipes

Here, `dateObj` is a JavaScript `Date` object set and the Angular locale is `en-US`.

The `date` pipe is highly configurable and can be extended with other locales as well.

```
{{ dateObj | date }}  
// 'Jun 15, 2015'
```

```
{{ dateObj | date:'medium' }}  
// 'Jun 15, 2015, 9:43:11 PM'
```

```
{{ dateObj | date:'shortTime' }}  
// '9:43 PM'
```

```
{{ dateObj | date:'mm:ss' }}  
// '43:11'
```


Pipes

Here, `balance` holds the number `0.259`.

```
{{ balance | currency }}  
// '$0.26'
```

Again, the `currency` built-in pipe is highly configurable and can be extended with other locales as well.

```
{{ balance | currency:'EUR' }}  
// '€0.26'
```

```
{{ balance | currency:'EUR':'code' }}  
// 'EUR0.26'
```

Pipes

The `async` pipe is a utility pipe to render asynchronous data.

For instance, it waits for a `Promise` to resolve and renders its value.

This is a really important pipe that does a lot for you under the hood! We'll revisit it in the `Observable` chapter and on day 2 as well.

```
{{ userNamePromise | async }}  
// renders 'Joe' after 300ms
```

```
userNamePromise = new Promise(resolve => {  
  setTimeout(() => {  
    resolve("Joe");  
  }, 300);  
});
```

Pipes

A few other built in pipes:

- `DecimalPipe`: transforms any number into a formatted string
- `PercentPipe`: transforms any number into a percentage string.
- `UpperCasePipe`: transforms text to upper case.
- `LowerCasePipe`: transforms text to lower case.
- `TitleCasePipe`: transforms text to title case (capitalized first letters).
- `JsonPipe`: converts a value into its JSON representation.

Custom pipe

You can create your own pipes by implementing the `PipeTransform` interface and decorating the class with `@Pipe`.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'truncate'})
export class TruncatePipe implements PipeTransform {

  transform(value: string, len: number, sym: string) {
    return value.slice(0, len) + sym;
  }

}
```

```
{{ 'This is a very long title!' | truncate:10:'...' }}
```

Template variables

Template variables help you use data from one part of a template in another part of the template.

A template variable can refer to the following:

- a rendered DOM element
- a component
- an `ng-template`

Template variables

Declare a variable with an attribute prefixed with a hash (#). Depending on where it is placed, it refers to a DOM node, a component or an `ng-template`.

You can then use the variable anywhere in the template, for instance by passing it to a method or interpolating its value.

```
<input #phone placeholder="phone number" />

<button (click)="callPhone(phone.value)">
  Call
</button>
```

Template variables

If you want to explicitly target a component instead of a DOM node, specify an attribute value in the variable declaration.

In this example, `itemForm` points to the `ngForm` property that exposes the `NgForm` component on `<form>` elements.

```
<form #itemForm="ngForm"  
      (ngSubmit)="onSubmit(itemForm)">  
  <input name="name" />  
  <button type="submit">Submit</button>  
</form>
```

Template variables

Directives create new template scope. Variables can only be accessed from an inner to an outer template scope.

This is analogous with code block scope.

```
<input #ref1 type="text" />
<span *ngIf="true">Value: {{ ref1.value }}</span>
```

```
<input *ngIf="true" #ref2 type="text" />
<span>Value: {{ ref2.value }}</span>
```


Template variables

Inside the component, you can get a hold of the element that a variable points to using the `@ViewChild` decorator.

The `@ViewChild` decorator is quite versatile. We'll come back to it later!

```
@Component({
  template: `
    <div #var>I'm a dividing element!</div>
  `,
})
export class ViewChildComp implements AfterViewInit {
  @ViewChild('var') tmpEl: ElementRef;

  ngAfterViewInit() {
    console.log('element:', this.tmpEl.nativeElement)
  }
}
```

Hands-on 03: create a template

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

Directives

What is a directive?

Directives are classes that add additional behavior to DOM elements in your Angular applications.

They can change either the structure (visibility and order) or the appearance (attributes) of DOM elements.

ngIf

To conditionally render a part of the template, use the `*ngIf` directive.

The asterisk (*) prefix is a convention for a shorthand notation. Angular converts it into the longer form shown here.

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```



```
<ng-template [ngIf]="hero">  
  <div class="name">{{hero.name}}</div>  
</ng-template>
```

ngIf

You can expand the shorthand form with `else` or `then` and `else`.

By default the `else` template is blank.

```
<div *ngIf="condition; else elseBlock">True.</div>  
<ng-template #elseBlock>False.</ng-template>
```

```
<div *ngIf="condition;  
  then thenBlock else elseBlock"></div>  
<ng-template #thenBlock>True.</ng-template>  
<ng-template #elseBlock>False.</ng-template>
```

ngIf

You can use `as` to store the condition result as a template variable in the if-scope.

```
<div *ngIf="condition as value">
  {{ value }}
</div>
```

ngIf

You can combine the pipe (|) operator with storing the value which helps to write concise templates.

```
<div *ngIf="userPromise | async as user">  
  Hello {{ user.last }}, {{ user.first }}!  
</div>
```


ngFor

To render a collection of items, use the `*ngFor` directive.

Again, the shorthand form is converted into the longer form as shown here.

Pretty confusing syntax! Just remember: using an `*ngFor` results in an `ng-template` which is reused to render each item in a collection.

```
<div *ngFor="let hero of heroes">
  {{hero.name}}
</div>
```



```
<ng-template ngFor [ngForOf]="heroes" let-hero>
  <div>
    {{hero.name}}
  </div>
</ng-template>
```

ngFor

You can pull more variables into the template via `ngFor`:

- `index`: the index of the current item in the iterable
- `count`: the length of the iterable
- `first / last`: true when item is the first / last in the iterable
- `even / odd`: true when item has an even / odd index

```
<li *ngFor="
  let user of users;
  index as i;
  first as isFirst;
  count as count;
  odd as odd"
  [class.odd]="odd"
>
  {{i}}/{{count}}. {{user}}
  <span *ngIf="isFirst">default</span>
</li>
```

ngFor

It's perfectly fine to render a collection as a list of custom components.

You can pass the `ngFor` variables to the component input with ordinary property binding.

```
<app-item-detail  
  *ngFor="let item of items"  
  [item]="item">  
</app-item-detail>
```

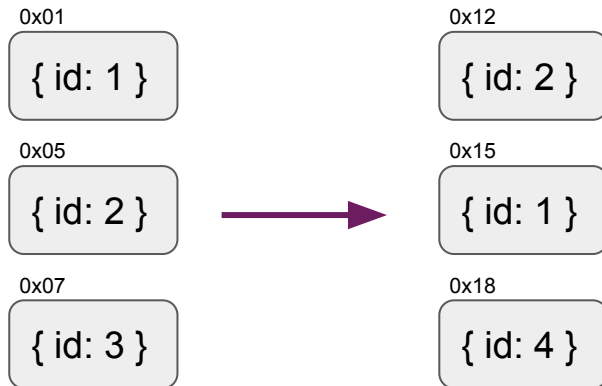
ngFor

By default, Angular compares item in the collection using `Object.is` (which means reference equality for objects).

You can change this behavior to prevent unnecessary template deletion and creation by providing a `trackBy` function.

```
<div *ngFor="let item of items; trackBy: trackByItems">  
  {{item.id}} - {{item.name}}  
</div>
```

```
trackByItems(index: number, item: Item): number {  
  return item.id;  
}
```



Multiple directives

Sometimes, you want to show a collection conditionally.

However, structural directives cannot be used on the same element by design.

You can use the `ng-container` component to wrap the collection and render it conditionally. The `ng-container` component does not create a DOM node.

```
<ng-container *ngIf="showItems">
  <li *ngFor="let item of items">
    {{item.name}}
  </li>
</ng-container>
```

ngSwitch

This is more esoteric but it has its uses. `ngSwitch` is like the JavaScript switch statement.

On a parent element, provide `ngSwitch` with an expression that returns the switch value. Then, add `*ngSwitchCase` expressions for each case. The default case is handled by `*ngSwitchDefault`.

```
<div [ngSwitch]="feature">
  <ng-container *ngSwitchCase="'counter'">
    Add a counter to the list.
  </ng-container>
  <ng-container *ngSwitchCase="'scroller'">
    Make the list scrollable.
  </ng-container>
  <ng-container *ngSwitchCase="'slider'">
    Show a slider next to the list.
  </ng-container>
  <ng-container *ngSwitchDefault>
    No feature enabled.
  </ng-container>
</div>
```

ngClass & ngStyle

Mentioned here for legacy reasons: the `ngClass` and `ngStyle` attribute directives. In older versions of Angular, you used these to set multiple classes or styles at once.

Recent versions started to support the same syntax with `class` and `style` bindings.

Always use the modern binding if possible.

```
<div [ngClass]="currentClasses">Content</div>
```

```
currentClasses = {  
  saveable: true,  
  modified: false,  
  special: true,  
}
```

```
<div [class]="currentClasses">Content</div>
```

ngModel

For template-based forms, the `ngModel` attribute binding makes it easy to add two-way binding on form elements and perform validation.

`ngModel` is part of the forms module and is not included by default.

More on `ngModel` in the Forms chapter.

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

@Component({
  selector: 'example-app',
  template: `
    <input
      [(ngModel)]="name"
      #ctrl="ngModel"
      required
    >
    <p>Value: {{ name }}</p>
    <p>Valid: {{ ctrl.valid }}</p>
  `,
})
export class SimpleNgModelComp {
  name: string = '';
}
```


Custom attribute directive

You can create both your own structural and attribute directives.

Here is an example of a custom attribute directive that highlights the node it was added to.

Angular provides the reference to the element it was set on by injecting `ElementRef` in the constructor.

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor
      = 'yellow';
  }
}
```

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

Custom attribute directive

We can extend the behavior of the directive by listening to user events.

Instead of adding event listeners on the native DOM, you should use the `@HostListener` decorator. The decorator also takes care of event listener removal after the component is destroyed.

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

Custom attribute directive

Directives are specialised components. You can provide inputs to the directive as well.

Here we extend the directive with two inputs named `appHighlight` and `enabled`.

Just as with components, you can use property binding to pass data to the directive.

```
export class HighlightDirective {
  @Input() appHighlight = '';
  @Input() enabled = false;

  @HostListener('mouseenter') onMouseEnter() {
    if (this.enabled) {
      this.highlight(this.appHighlight);
    }
  }
  //...
}

<p appHighlight="yellow"
  [enabled]="highlightEnabled">
  Highlight me?
</p>
```

Directive composition

Angular lets you apply directives to a component's host element from within the component class.

When the framework renders a component, Angular also creates an instance of each host directive and applies it on the host element.

You can add multiple directives if needed.

```
@Component({  
  selector: 'admin-menu',  
  template: 'admin-menu.html',  
  hostDirectives: [ExpandBehavior],  
})  
export class AdminMenu {  
  // ...  
}
```

<admin-menu></admin-menu>



Hands-on 04: render a list of items

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

Dependency injection

What is dependency injection?

Dependency Injection (DI), is a design pattern and mechanism for creating and delivering some parts of an application to other parts of an application that require them.

In Angular, dependencies are typically services, but they also can be values, such as strings or functions. A runtime injector instantiates dependencies when needed, using a configured provider of the service or value.

Provide a dependency

You provide a dependency in two steps: first you add the `@Injectable` decorator to a class to mark that it can be injected.

In this example, the logger class can now be provided, but you still need to configure *where* it is provided.

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

@Injectable()
export class Logger {
  log(msg: any) { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any) { console.warn(msg); }
}
```


Provide a dependency

You can provide the dependency on the **component level** using the `providers` config on the component decorator.

You get a new instance of the class with each new instance of that component.

The root and its descending components get the same instance if they inject.

```
@Component({  
  selector: 'some-component',  
  template: '...',  
  providers: [Logger]  
})  
class SomeComponent {}
```

Provide a dependency

You can provide the dependency on the **module level** using the `providers` config on the module decorator.

The same instance of the class is made available to all components in that `NgModule`.

```
@NgModule({  
  declarations: [SomeComponent]  
  providers: [Logger]  
})  
class SomeModule {}
```

Provide a dependency

However, you can also provide a dependency on the **module level** via the dependency decorator of the injectable class.

```
@Injectable({  
    providedIn: SomeModule  
})  
class Logger {}
```

This is the preferred way for specifying module dependencies, as this enables the compiler to create smaller bundle size.

Provide a dependency

You can provide the dependency on the **application level** by setting `providedIn` to `root` on the dependency decorator.

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

Angular creates a single, shared instance of the class and injects it into any class that asks for it.

This is often the start scenario if you have not split your app up into multiple modules.

Inject a dependency

Now you want to inject the dependency. The most common way to inject a dependency is to declare it as a parameter of a constructor.

It's not necessary to assign the dependencies to fields, Angular generates code to do this for you.

Note that you still have to `import` the `Logger` class as it is used to determine what needs to be injected!

```
import { Component } from '@angular/core';  
import { Logger } from '../logger.service';  
  
@Component({ ... })  
class SomeComponent {  
    constructor(private logger: Logger) {}  
}
```

Create an injectable service

A service is just a class, so there is no difference in how it is made injectable!

However, Angular does makes a distinction between components and services. It does not enforce any split. So for instance, you could write 'heavy' components that perform fetching as well.

The rule of thumb is: business logic in a component should only concern about its rendering. Extract any other concern as a service and inject it.

Create an injectable service

You can use the command line to quickly create a service.

```
ng generate service heroes/hero
```



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

@Injectable({
  providedIn: 'root',
})
export class HeroService {}
```

Inject a service in your service

When a service depends on another service, follow the same pattern as injecting into a component.

```
import { Injectable } from '@angular/core';
import { Logger } from '../logger.service';

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor(private logger: Logger) { }

  getHeroes() {
    this.logger.log('Getting heroes ...');
    return [];
  }
}
```


Class provider shorthand

Each provider is of a certain type and can be looked up by a *token*.

What you have seen so far is a shorthand for a class provider that uses the same class as its token.

Based on where the provider is set up, the class is instantiated by the dependency injector using the `new` operator.

```
providers: [Logger]
```



```
providers: [{ provide: Logger, useClass: Logger }]
```

Alias providers

You can create aliases for providers to map one token to another by specifying `useExisting`.

In practice this is not used that often.

```
providers: [  
    NewLogger,  
    { provide: OldLogger, useExisting: NewLogger }  
]
```

Factory providers

You can provide a dynamic value as a dependency by using a factory provider.

You set the `useFactory` function and declare the factory dependencies on `deps`.

In this example, every time a component is created and requires the `HeroService`, the factory function is called.

This makes it possible to separate the authorization lookup from `HeroService`.

```
const heroServiceFactory =
  (log: Logger, userSrv: UserService) =>
    new HeroService(log, userSrv.user.isAuthenticated);

@NgModule({
  providers: [{
    provide: HeroService,
    useFactory: heroServiceFactory,
    deps: [Logger, UserService]
  }]
})

// hero.service
constructor(
  private logger: Logger,
  private isAuthenticated: boolean) { }
```

Value providers

If you want to inject non-class dependencies, such as runtime configuration, you can use a value provider.

To make it work, you have to instantiate a token from `InjectionToken` and pass in the data via `useValue`.

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

type AppConfig = {
  someValue: string;
}

const appConfig: AppConfig = {
  someValue: 'abcdef'
}

export const APP_CONFIG =
  new InjectionToken<AppConfig>('app.config');

@NgModule({
  providers: [
    { provide: APP_CONFIG, useValue: appConfig }
  ]
})
```

Value providers

To inject the value from the value provider, the current best practice is to use the `inject` function and pass in the instantiated `InjectionToken`.

A nice addition of the `inject` function is type inference. In the example, `appConfig` is of type `AppConfig` without explicitly typing it.

```
import { Component, inject } from '@angular/core';
import { APP_CONFIG } from '../app.module';

@Component({ ... })
class SomeComponent {
  private appConfig = inject(APP_CONFIG);
}
```

Injector resolving hierarchy

Angular resolves a token in two phases:

1. Against own and ancestor element injectors
2. Against own and ancestor module injectors

If Angular then still doesn't find a provider, it throws an error.

This makes it for instance possible to inject unique service instances per component subtree.

Resolution modifiers

When a service injection is decorated with `@Optional`, Angular considers it to be optional.

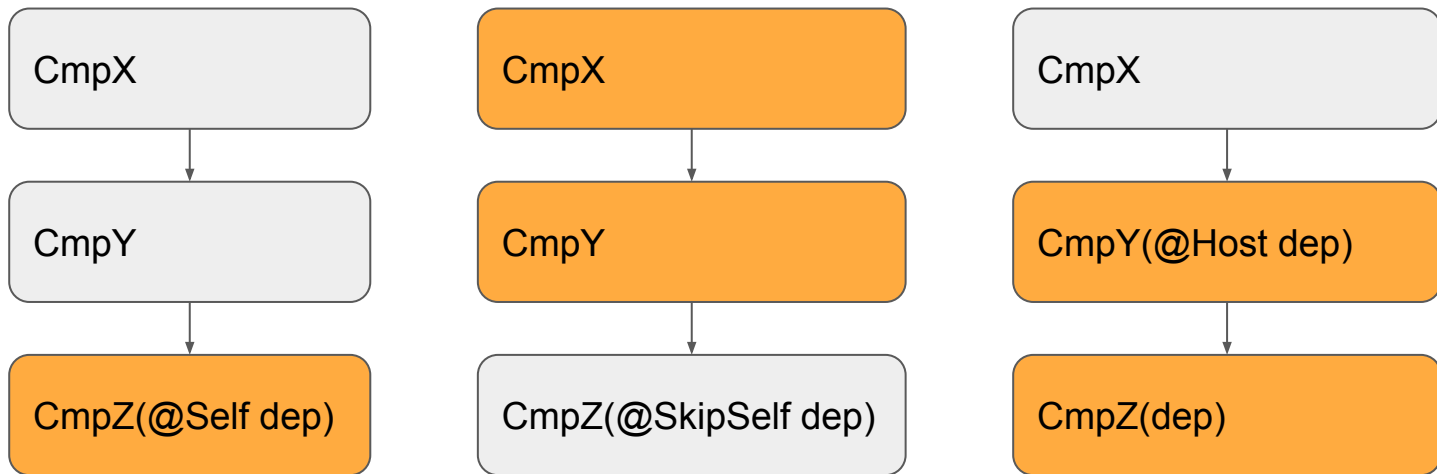
```
constructor(@Optional()  
    public optional?: OptionalService) {}
```

This can come in handy if you want the component to work regardless of the placement in the element / module hierarchy.

The service will be set to `null` if not found, so you need to add extra checks.

Resolution modifiers

- `@Self`: only look at the element injector of the current component
- `@SkipSelf`: skip this component, start search in the parent element injector
- `@Host`: this component is the last stop when searching for injectors



Special case: projected content

Remember projected content? In this example, the `MyService` dependency is available in both `some-component` and `app-child` which is projected by `app-parent`.

If you don't want to pass dependencies to projections, use `viewProviders` instead of `providers`.

```
@Component({  
  selector: 'app-parent',  
  template: `  
    <some-component></some-component>  
    <ng-content></ng-content>  
  `,  
  providers: [MyService]  
})  
export class ParentComponent {}
```

```
<app-parent>  
  <app-child></app-child>  
</app-parent>
```

Hands-on 05:

inject a service in a component

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

Observables

What is an observable?

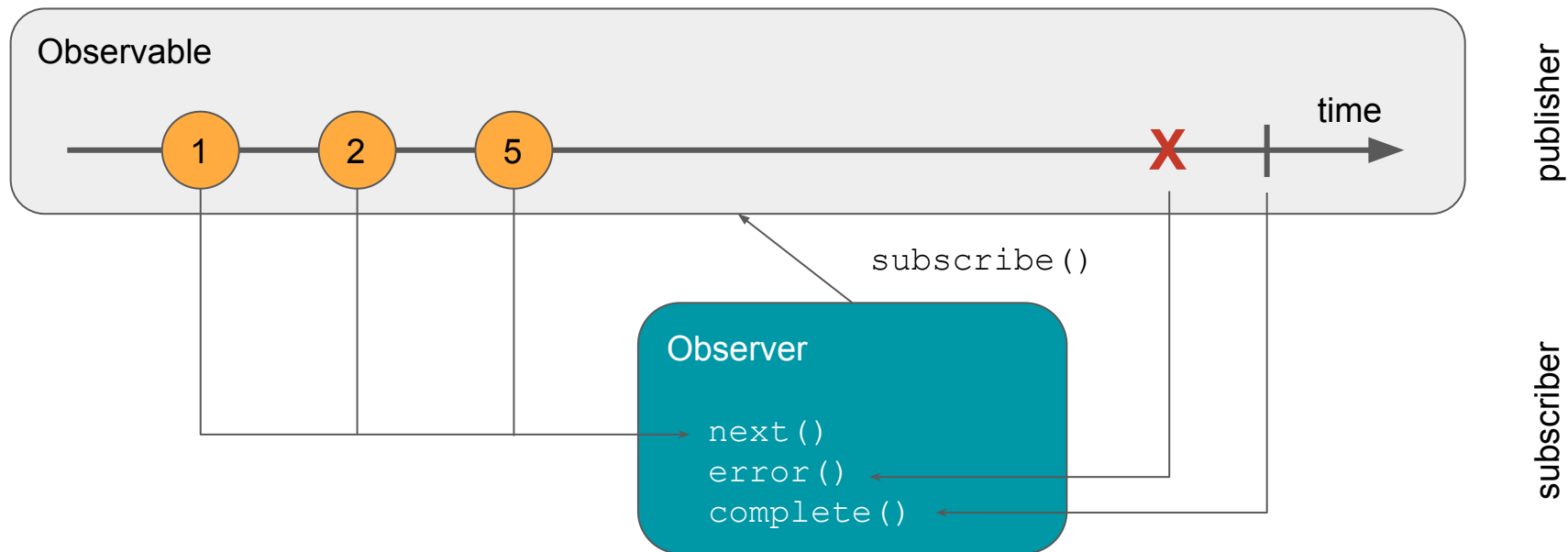
In essence, an observable is an object that accepts subscribers (called observers). The observable can produce any type of data, which notifies observers of this data.

The Angular team embraced the observable concept while developing Angular 2.0. A lot of the APIs of Angular have the observable signature, for instance:

- @Output event emitters
- HTTP client requests and interceptors
- the activated route in the router
- querying elements with @ViewChildren/@ContentChildren

You will learn how to use observables to handle these Angular APIs with ease, and you can use observables in your Angular application for any stream of data.

What is an observable?



Create observables

The publishing side creates `Observable` instances. You pass in an function that receives an `Observer` object.

```
const random$ = new Observable(observer => {  
  const value = Math.random();  
  observer.next(value);  
});
```

This function is run every time when an observer subscribes to the data.

In this function, you call `observer.next` one or multiple times with the values you want to send to that observer.

Create observables

If an error occurs in your observable, you can notify the observer by calling `observer.error`. You can pass in any error information.

```
const error$ = new Observable(observer => {  
  observer.error('Lame example error');  
});
```

Create observables

When the observable has completed producing data and you want to notify the observer about that, call `observer.complete`.

```
const vowels$ = new Observable(observer => {  
  const vowels = ['a', 'e', 'i', 'o', 'u'];  
  
  for (let letter of vowels) {  
    observer.next(letter);  
  }  
  
  observer.complete();  
});
```


Subscribe to an observable

The simplest way to subscribe to the observable is by calling the `subscribe` function on it and passing in a function that accepts the next value.

```
random$.subscribe(  
  value => console.log('Emitted:', value);  
);
```

```
// Emitted: 0.3646219186661961
```

Angular will convert the function into the `observer` object and calls the observable function with it.

Subscribe to an observable

If you want to handle observable errors or completion, you need to pass in an object that implements the `Observer` interface. Note that both `error` and `complete` are optional.

```
vowels$.subscribe({  
  next(value) { console.log('Emitted:', value) },  
  error(value) { console.error('Error:', value) },  
  complete() { console.log('Completed!') }  
});
```

```
// Emitted: a  
// Emitted: e  
// Emitted: i  
// Emitted: o  
// Emitted: u  
// Completed!
```

Unsubscribe

It makes sense to let the observer unsubscribe at some point.

This is done by returning an object with an `unsubscribe` function from the observable. Then, the observer can call it after subscription.

If the observable calls `error` or `complete`, the observer is automatically unsubscribed.

```
const interval$ = new Observable(obs => {
  const id = setInterval(() => {
    obs.next(Math.random());
  }, 100);

  return {
    unsubscribe: () => clearInterval(id),
  };
});

const subscription = interval$.subscribe(
  value => console.log(value)
);

setTimeout(() => subscription.unsubscribe(), 500);
```

AsyncPipe with observable

Let's revisit the `async` template pipe. It also works with observables! Just slap on the `async` pipe and you will get the latest value in your template.

The `async` pipe takes care of subscribing to the observable, notifying about changes and unsubscribing when the component is destroyed.

Also, in this example, the `time$` property does not change, so you can opt-out of default change detection.

```
@Component({
  template: `
    <div>Time: {{ time$ | async }}</div>
  `
})
export class TimeComponent {
  time$ = new Observable<string>((observer: Observer) => {
    setInterval(
      () => observer.next(new Date().toString()), 1000);
  });
}
```

RxJS

“RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.”

Angular comes with the `rxjs` dependency out of the box and uses it internally.

However, RxJS is a separate project and is completely separate from Angular. You don't need to use it, it's complementary.

RxJS of

Use `of` to convert the given arguments to an observable that emits each value separately.

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

const nums$ = of(1, 2, 3);

nums$.subscribe(x => console.log(x));

// 1
// 2
// 3
```

RxJS from

Use `from` to convert 'almost anything' to an observable.

You can for instance use it on an array. It will convert it to an observable that emits each value separately.

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

const result$ = from([10, 20, 30]);

result$.subscribe(x => console.log(x));

// 10
// 20
// 30
```

RxJS from

You can use `from` to convert a promise to an observable that emits the resolved value or the rejected error.

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

const result$ = from(
  new Promise((resolve) => {
    setTimeout(() => {
      resolve('Result!');
    }, 1000);
  })
);

result$.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```


RxJS from

Convert any iterable to an observable with `from`.

Generator functions and iterables has become available in JavaScript since 2015. They match quite nicely with observables!

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

function* generateDoubles(start: number) {
  let i = start;
  while (true) {
    yield i;
    i = 2 * i;
  }
}

const iterator = generateDoubles(2);
const result$ = from(iterator);

result$.subscribe(x => console.log(x));
```

RxJS fromEvent

You can use `fromEvent` to create observable events.

For this you need to have access to the native DOM element. The example on the right is simplified and does not use Angular.

Question: how did we get a reference to a native element in an Angular component?

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

const el = document.getElementById('my-element');

const mouseMoves$ = fromEvent<MouseEvent>(
  el, 'mousemove');

mouseMoves$.subscribe(evt =>
  console.log(
    'Coords:', evt.clientX, evt.clientY);
);
```

RxJS interval & timer

Use `interval` to emit a ascending number every X milliseconds.

The actual value is often not used, it's all about the timing!

If you only want to get a single value emitted after X milliseconds, use `timer` in a similar fashion.

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

const source$ = interval(1000);

source$.subscribe(val => console.log(val));

// 0 (after 1s)
// 1 (after 2s)
// 2 (after 3s)
// 3 (after 4s)
// ...
```

RxJS composition

So far we've seen a couple of 'simple' observable creation functions that take a non-observable source and create observables from it.

But what if we want to manipulate the observable in some way? This is where RxJS shines. RxJS provides lots of functions to compose new observables from existing ones.

RxJS map

With `map`, you can transform emitted values by calling a transformation function for each value.

`map` takes a transformation function. It returns a composable function that, given an observable, produces a new one.

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const nums$ = of(1, 2, 3);

const squared = map((n: number) => n * n);
const squaredNums$ = squared(nums$);

squaredNums$.subscribe(x => console.log(x));

// 1
// 4
// 9
```

RxJS filter

Using `filter` you can emit values when they pass a certain condition.

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

const nums$ = of(1, 2, 3);

const odd = filter((n: number) => n % 2 !== 0);
const oddNums$ = odd(nums$);

oddNums$.subscribe(x => console.log(x));

// 1
// 3
```

RxJS pipe

What if you have a more complex transformation with multiple steps?

You can of course use the output of one observable as an input for the next, but in practice that leads to long, unreadable expressions.

That's where the `pipe` function comes in! It's used so often that it's accessible via the RxJS observable object.

```
import { of } from 'rxjs';
import { filter, map } from 'rxjs/operators';

const nums$ = of(1, 2, 3, 4, 5);

const oddSquaredNums$ = nums$.pipe(
  filter((n: number) => n % 2 !== 0),
  map((n: number) => n * n)
);

oddSquaredNums$.subscribe(x => console.log(x));

// 1
// 9
// 25
```

RxJS tap

When you're composing multiple pipe operations, it can be very useful to log or debug the in-between result.

Use `tap` to perform side effect code that does not transform the result.

Don't rely on side effects to correctly produce data! If there is a dependency, it should trigger its own observable value.

```
import { of } from 'rxjs';
import { tap, map } from 'rxjs/operators';

of(1, 2).pipe(
  tap(val => console.log(`BEFORE MAP: ${val}`)),
  map(val => val + 10),
  tap(val => console.log(`AFTER MAP: ${val}`))
).subscribe(console.log);

// BEFORE MAP: 1
// AFTER MAP: 11
// 11
// BEFORE MAP: 2
// AFTER MAP: 12
// 12
```


RxJS take

With `take` you can take the next N values from an observable and then unsubscribe.

You can use `take` just as in the previous examples, however wrapping *piping operators* in `pipe` is the preferred way.

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

const source$ = of(1, 2, 3, 4, 5);

const takeTwo$ = source$.pipe(take(2));

takeTwo$.subscribe(console.log);

// 1
// 2
```

RxJS skip

Instead of taking N values, you can also skip N values and return others by using `skip`.

```
import { of } from 'rxjs';
import { skip } from 'rxjs/operators';

const source$ = of(1, 2, 3, 4, 5);

source$.pipe(skip(2)).subscribe(console.log);

// 3
// 4
// 5
```

RxJS startsWith

Instead of skipping, you can also prepend values using `startsWith`.

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

const source$ = of(3, 4);

const prepended$ = source$.pipe(startsWith(1, 2));

prepped$.subscribe(console.log);

// 1
// 2
// 3
// 4
```

RxJS concat

You can concatenate the result of two or more observables into a new one with `concat`.

Pay attention what you concatenate: when a source never completes, any subsequent observables never run.

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

const source1$ = of(1, 2);
const source2$ = of(3, 4);

concat(source1$, source2$).subscribe(console.log);

// 1
// 2
// 3
// 4
```

RxJS combineLatest

With `combineLatest` you combine the latest values emitted from two or more observables into an array.

This is very useful to join multiple disjunct sources. The optional second parameter can be used as a shorthand to map the values into something else than an array.

```
import { interval, combineLatest, take } from 'rxjs';

const source1$ = interval(1000);
const source2$ = interval(1000);

combineLatest([
  source1$.pipe(take(3)),
  source2$.pipe(take(3))
]).subscribe(console.log);

// [ 0, 0 ]
// [ 1, 0 ] (after 1s)
// [ 1, 1 ] (after 1s)
// [ 2, 1 ] (after 2s)
// [ 2, 2 ] (after 2s)
```

RxJS switchMap

With `switchMap` you can create 'inner' observables that emit their own value or values for each emitted source value.

For instance, if you observe button events, you can 'switch' each event to an observable that does a HTTP request.

In this example, each source value leads to the production of two new values.

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

const source1$ = of(2, 5);

const switched$ = source1$.pipe(
  switchMap(x => of(x, x ** 2))
);

switched$.subscribe(console.log);

// 2
// 4
// 5
// 25
```

RxJS share

By design, each subscription kicks off a separate producer process from the observable.

In this example, both subscribers cause the `tap` side effect to trigger.

```
import { timer } from 'rxjs';
import { tap, map } from 'rxjs/operators';

const example$ = timer(1000).pipe(
  tap(() => console.log('SIDE EFFECT')),
  map(() => 'RESULT'),
);

example$.subscribe(console.log);
example$.subscribe(console.log);

// SIDE EFFECT (after 1s)
// RESULT (after 1s)
// SIDE EFFECT (after 1s)
// RESULT (after 1s)
```

RxJS share

The `share` operator makes it possible to change this to have one process for producing values, and sharing the outcomes with multiple observers!

This is also called multicasting, and is close to the generic publisher/subscriber pattern.

```
import { timer } from 'rxjs';
import { tap, map, share } from 'rxjs/operators';

const example$ = timer(1000).pipe(
  tap(() => console.log('SIDE EFFECT')),
  map(() => 'RESULT'),
  share(),
);

example$.subscribe(console.log);
example$.subscribe(console.log);

// SIDE EFFECT (after 1s)
// RESULT (after 1s)
// RESULT (after 1s)
```


RxJS Subject

However, there is another way of multicasting values. This is done using a special `Subject` class which is both an observable and an observer.

You subscribe the subject to one or more observers and you subscribe a source observable to the subject. See it as an 'in-between broadcast channel'.

In Angular this is the preferred way for broadcasting values.

```
import { Subject, of } from 'rxjs';
import { tap } from 'rxjs/operators';

const subject = new Subject();
subject.subscribe((v) => console.log(`observerA: ${v}`));
subject.subscribe((v) => console.log(`observerB: ${v}`));

of(1, 2).pipe(
  tap(() => console.log('SIDE EFFECT'))
).subscribe(subject);

// SIDE EFFECT
// observerA: 1
// observerB: 1
// SIDE EFFECT
// observerA: 2
// observerB: 2
```

RxJS BehaviorSubject

A `BehaviorSubject` is an extended version of `Subject` that holds the latest value which is always emitted when subscribed to. You need to provide it with an initial value.

This makes sure that an observer either gets the initial or the last-emitted value, regardless of when it started observing. This is often what you expect from a data perspective!

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

const subject = new BehaviorSubject(1);
subject.subscribe(console.log);
// 1
subject.subscribe(console.log);
// 1
subject.next(2);
// 2
// 2
subject.subscribe(console.log);
// 2
subject.next(3);
// 3
// 3
// 3
```

RxJS error handling

If an exception is thrown in any of the operator functions, this will cause the observer `error` function to be called with the result.

To catch such errors and provide a fallback, use `catchError`. You provide it with a new observable which replaces the current source.

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

const data$ = of(1, 2, 3).pipe(
  map(() => { throw new Error('Oops!'); }),
  catchError(() => of(-1))
);

data$.subscribe({
  next(x: any) { console.log('data: ', x); },
  error() { console.log('this will not run'); },
});

// data: -1
```

RxJS retry

Use `retry` to let RxJS restart the subscription process N times when it fails.

In the following scenario, the `ajax` function from `rxjs` performs a request. It will be retried 3 times before calling `error`.

You can use this operator with the Angular HTTP client as well.

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

ajax('/api/does-not-exist').pipe(
  map(() => {
    console.log('Error occurred.');
```

```
    throw new Error('Simulated error');
  }),
  retry(3)
).subscribe({
  error() { console.log('Error in subscription'); },
});
```

```
// Error occurred.
// Error occurred.
// Error occurred.
// Error in subscription
```

Hands-on 06: observe the mouse!

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

Check-out



Thank you!

See you next week!