Gemeente Haarlem

Angular - Maatwerk

Peter Kassenaar
info@kassenaar.com

Peter Kassenaar

- Trainer, author, developer – since 1996
- Specialty: *"Everything JavaScript"*
- JavaScript, ES6, Angular, NodeJS, TypeScript, jQuery, Vue.js, React

www.kassenaar.com

info@kassenaar.com

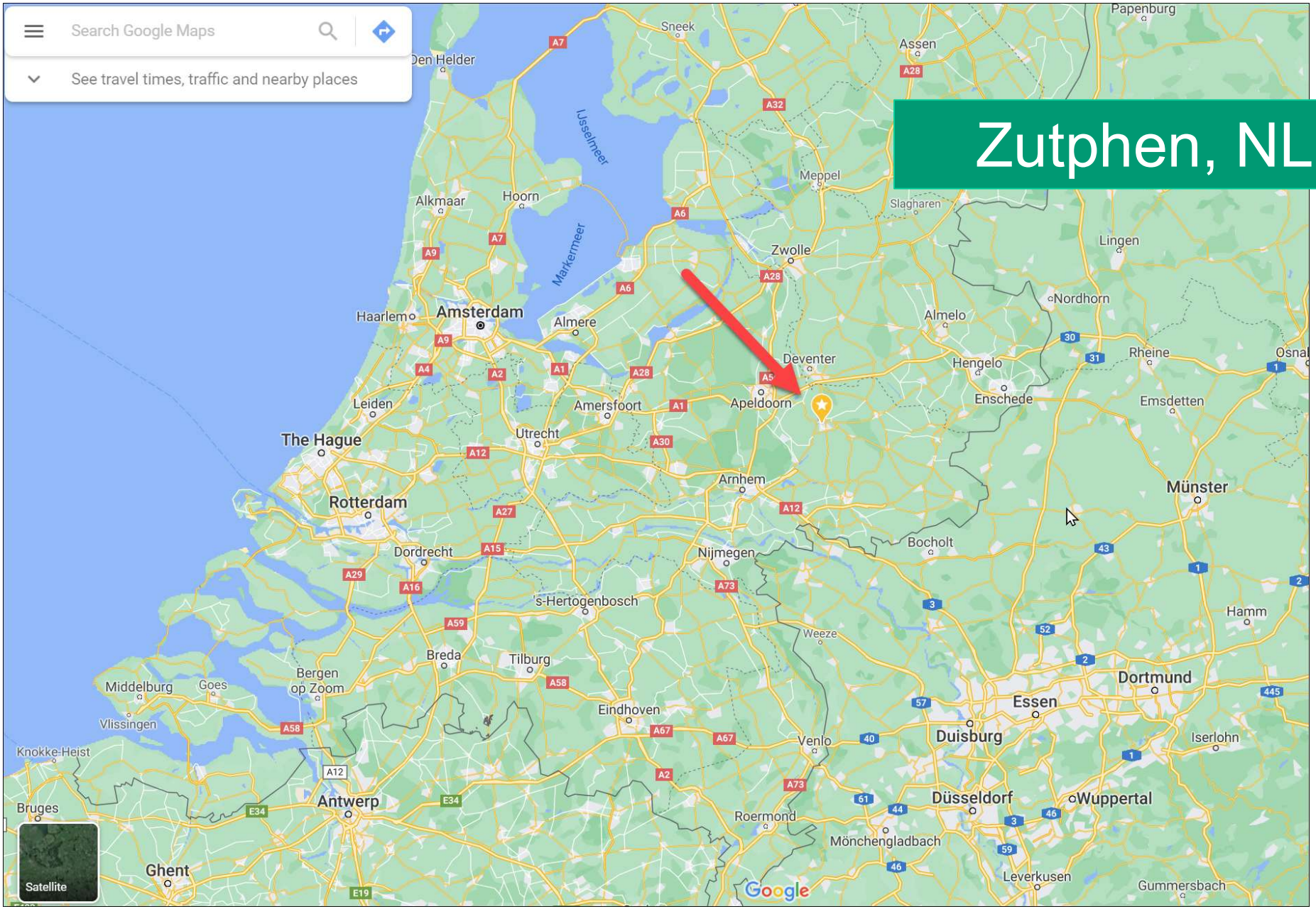
VANDUUREN MEDIA

ING OHRA euricom A DIMENSION DATA COMPANY sanoma delta lloyd

zenito BETERE ZEKERHEID VOOR ONDERNEMERS Atos woonbron OBERON INTERACTIVE ROC West-Brabant the eforum FACTORY

Sr. frontend developer ProductIP, Ede, NL (50%)

www.productip.com


Search Google Maps
See travel times, traffic and nearby places
Zutphen, NL
Den Helder
Sneek
Assen
Papenburg
IJsselmeer
Meppel
Alkmaar
Hoorn
Markermeer
Slagharen
Zwolle
Lingen
Haarlem
Amsterdam
Almere
Nordhorn
Almelo
Deventer
Hengelo
Rheine
Leiden
Amersfoort
Apeldoorn
Enschede
Emsdetten
The Hague
Utrecht
Arnhem
Münster
Rotterdam
Bocholt
Dordrecht
Nijmegen
s-Hertogenbosch
Hamm
Weeze
Breda
Tilburg
Bergen op Zoom
Middelburg
Goes
Dortmund
Eindhoven
Essen
Vlissingen
Duisburg
Knokke-Heist
Venlo
Iserlohn
Antwerp
Bruges
Roermond
Düsseldorf
Wuppertal
Mönchengladbach
Ghent
Leverkusen
Gummersbach
Satellite
Google

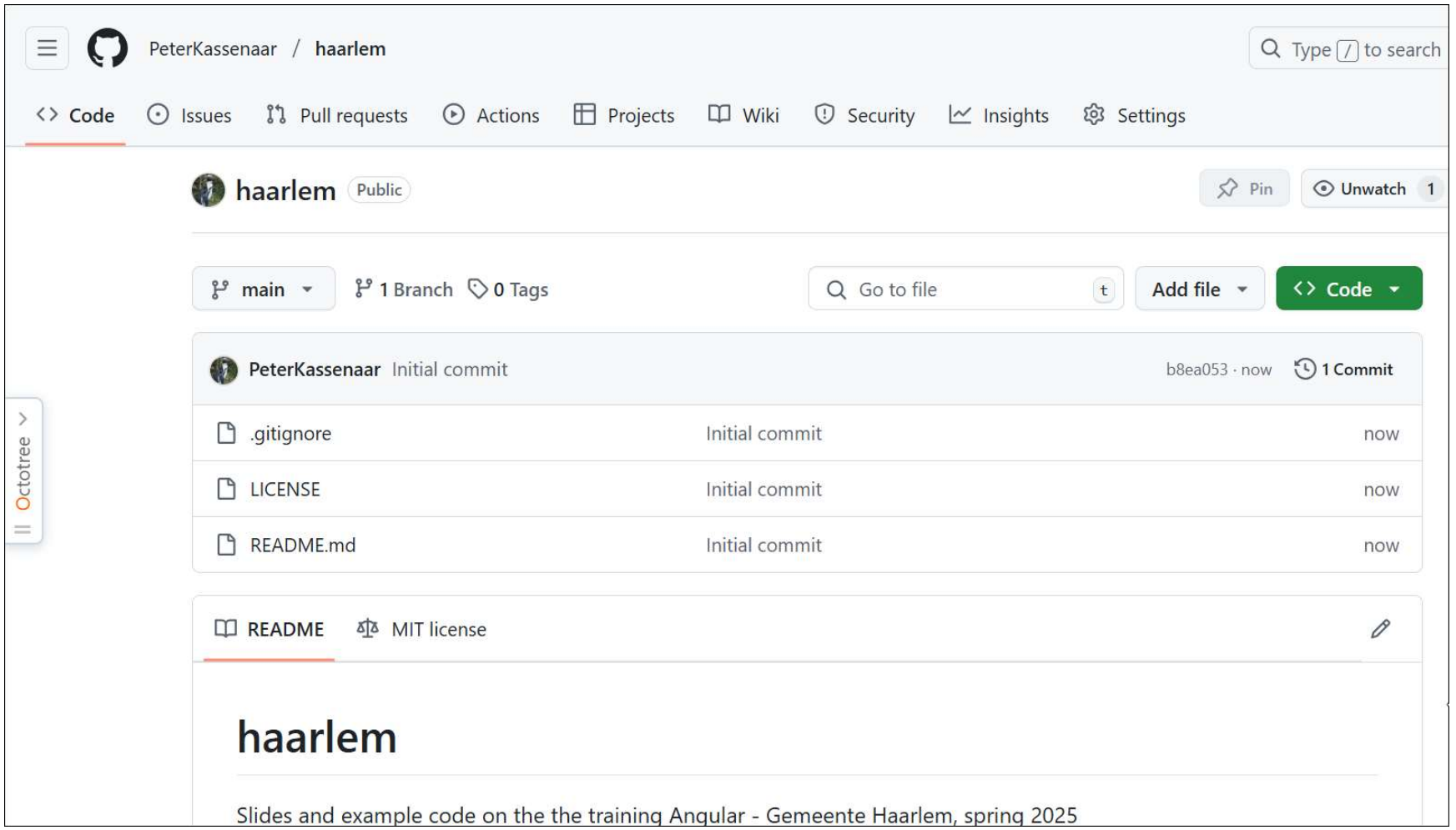
peterkassenaar

PeterKassenaar

PeterKassenaar

pkas06

github.com/PeterKassenaar/haarlem



About you…

Introduce yourself shortly

(Previous) Knowledge of Angular, (mobile/web-) apps?

How long have you worked with Angular yet?

Tell us a little bit about your projects.

What are your expectations of this course?

Agenda – 27-28 May 2025

~09:00 start – Morning session

~ 10:00, 11:00 Short Break

~12:00 Lunch

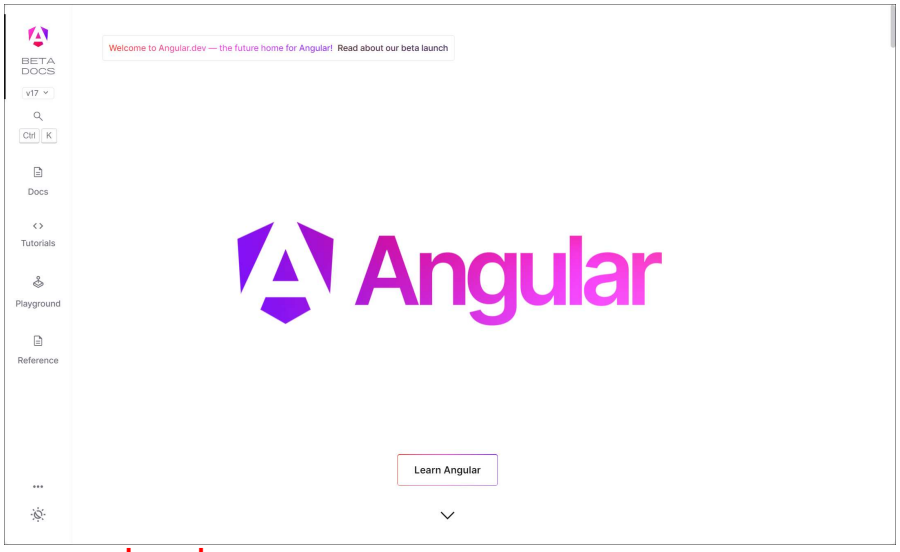
~12:45 Afternoon session

~ 14:00, 15:00 Break

~16:00-16:15 - end

Material

Software	(Angular + Editor + Browser + libraries)
Handouts	(PDF, Github)
Workshops	(in the presentations)
Websites	(online)



angular.dev

Agenda - day #1

- Introduction
- Angular New Features + refresher
  - NPM, NPX, `node_modules`, webpack, Vite and more
  - `@for`, `@if-then-else`, `@switch()`
  - Standalone components and module-less defaults
  - Updating Angular versions
  - Dependency Injection
  - New `provideHttpClient()`
  - …
- RxJS
  - Goals, structure & architecture, operators
  - Subscribing vs. Async pipe
- Routing & Guards
  - Creating & using guards, lazy loading

Agenda - day #2

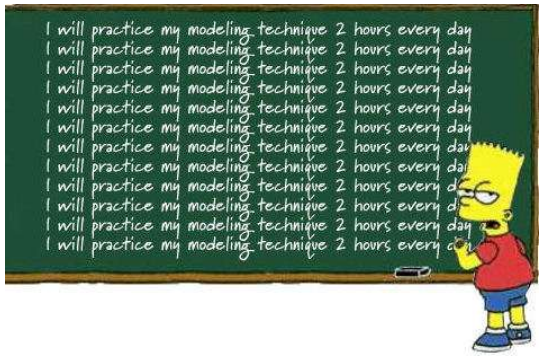
- State Management
  - General concepts, Redux, NgRX
  - NgRx Effects
- Reactive Form Controls
  - Control states: pristine, dirty, touched, etc.
  - Dynamic Form arrays
- Internationalization
  - Angular Package for i18n
  - Translation files
- …

Overall process

…

On Workshops…

- … are designated with a slide like this 👇
- Are between the talks with theory
- Are NOT written out line-by-line. You have to think for yourself
- The time during the training may be too short to finish all the workshops
  - Do them in your own time
  - At least you know the concepts the workshop is about
  - Choice – we have more to discuss, I let that prevail
- The example code often (but not always!) contains the ‘solution’ to the workshop. Use it! It’s there for you
  - But of course it would be nice if you can work with your own project/data



Questions?

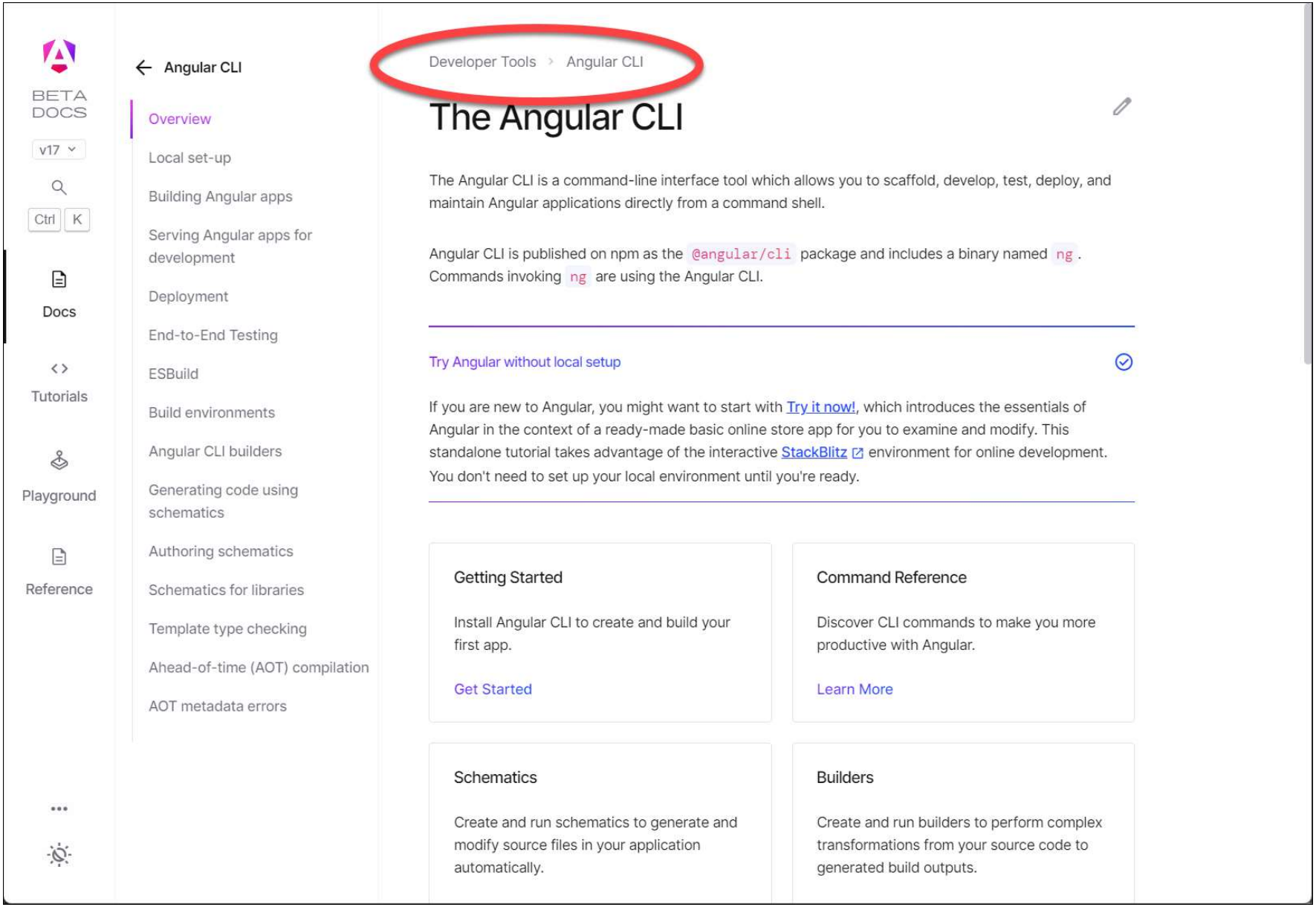
Angular CLI

Scaffolding new projects, new options

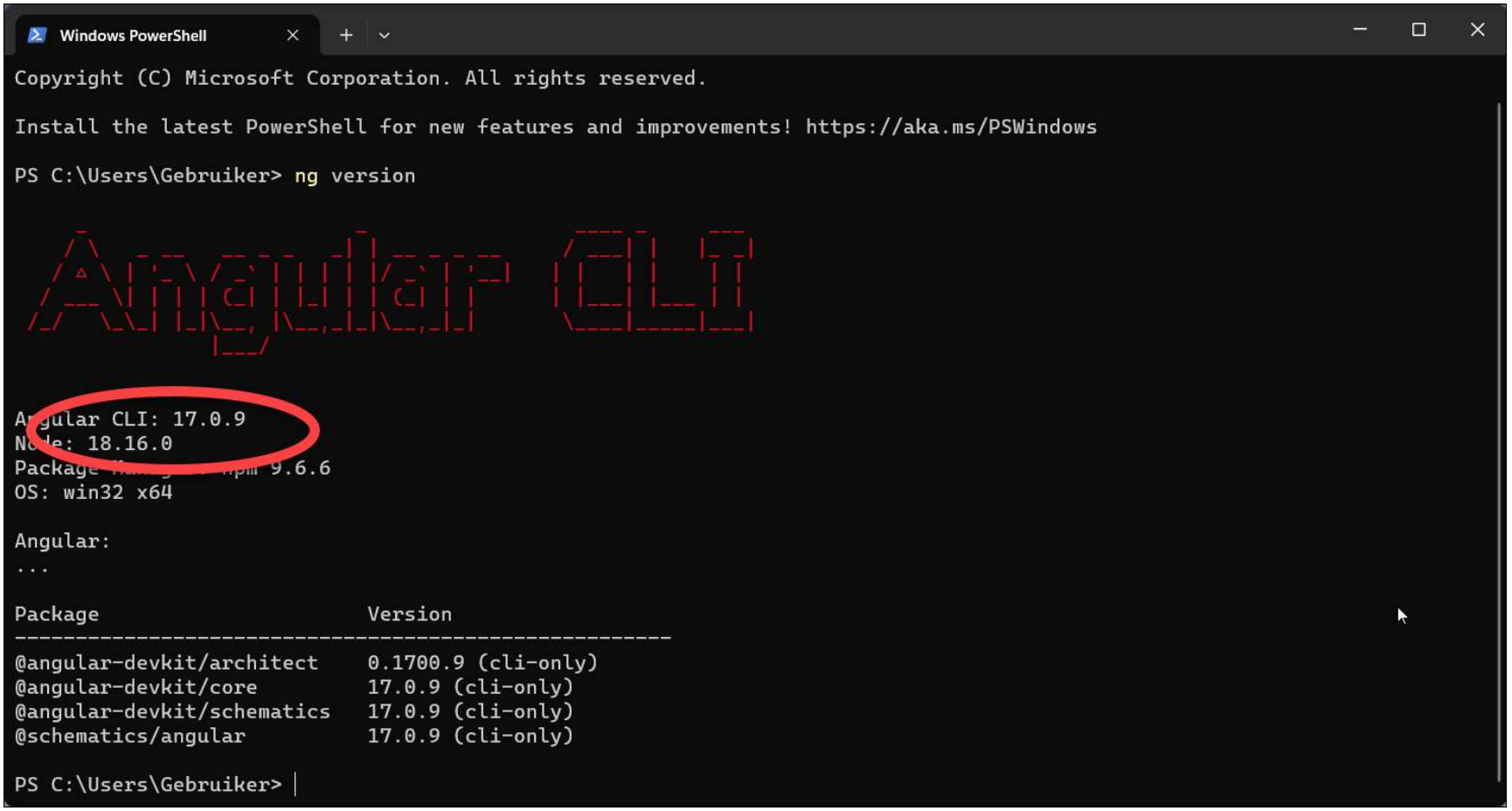
Angular CLI

- Command Line Interface for
  - Scaffolding (`ng new`),
  - Developing (`ng generate`)
  - Testing (`ng test`)
  - Deploying applications (`ng deploy`)
- Local installation
  - SO: no online environment like Stackblitz or CodeSandbox

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



https://angular.dev/tools/cli



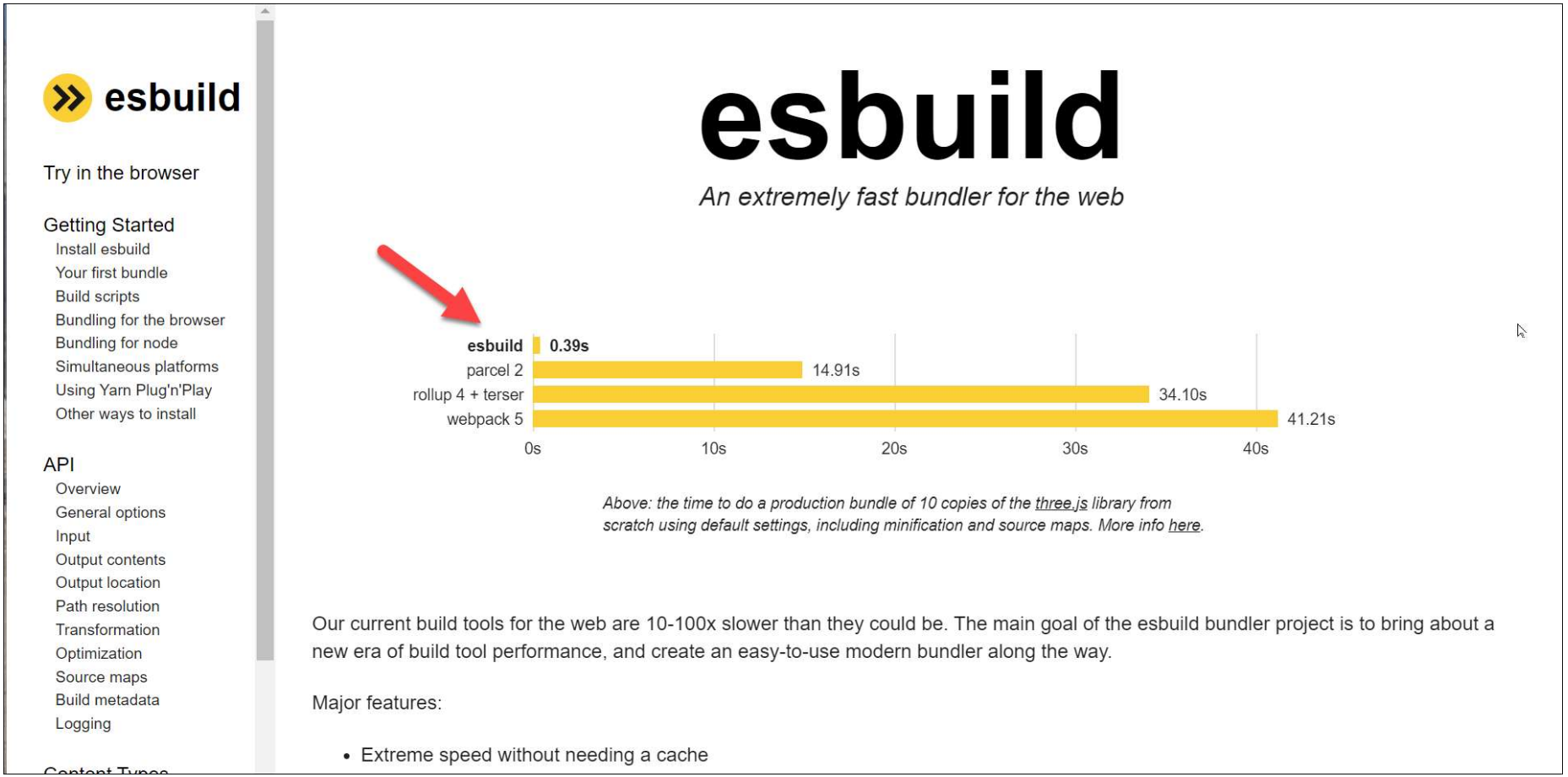
Current version? `ng version`

Make sure you have at least Node 18+

New in Angular CLI v.17+

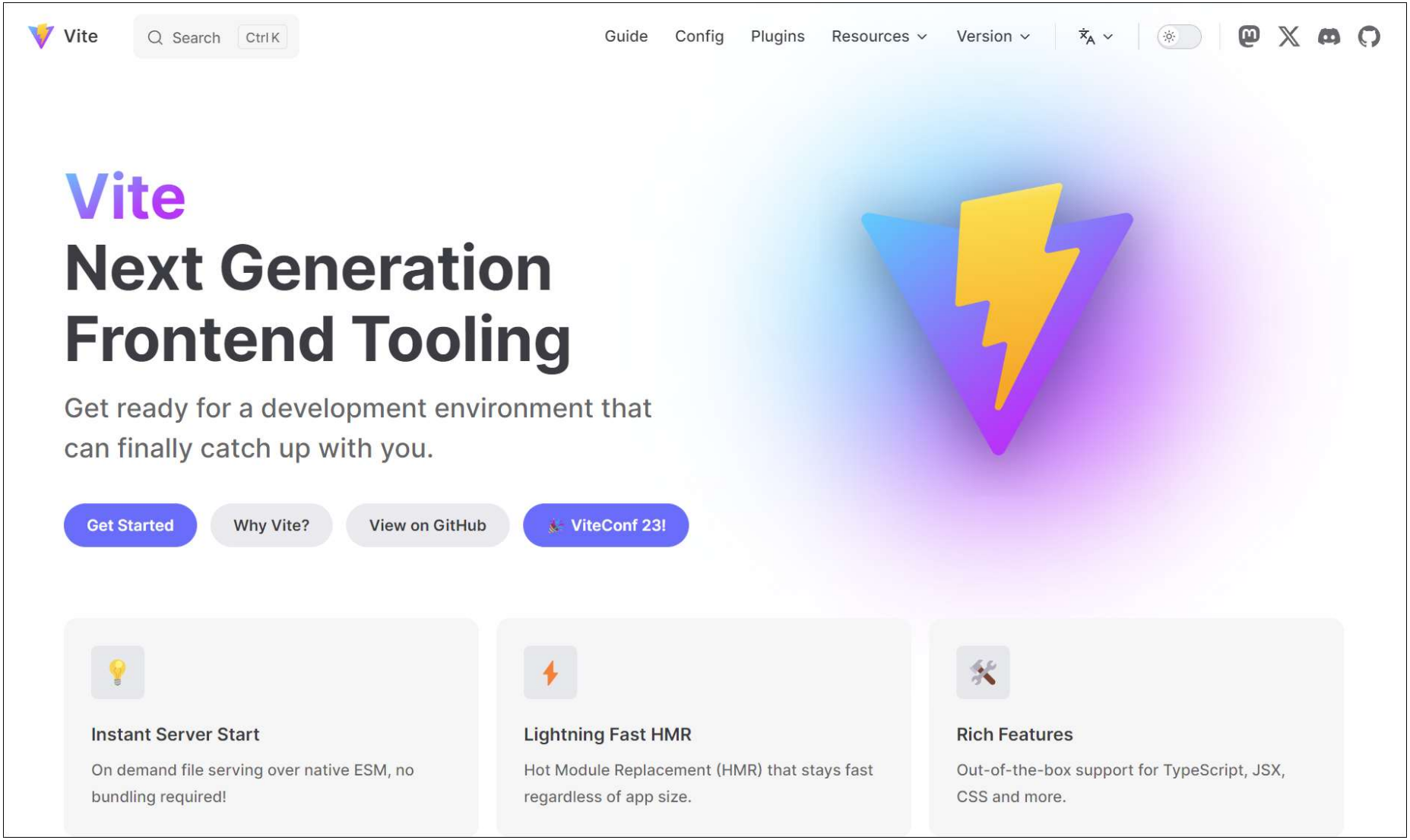
- Modern output format using ESM
  - dynamic import expressions to support lazy module loading.
- Faster build-time performance
  - both initial builds and incremental rebuilds.
- Newer JavaScript ecosystem
  - tools as `esbuild` and Vite.
- Integrated SSR and prerendering capabilites

ESBuild – extremely fast builder



https://esbuild.github.io/

Vite.js – modern bundler



https://vitejs.dev/

New in CLI

- Option syntax: Unix/POSIX conventions
- Boolean options:
  - `--some-option` sets `--some-option` to `true`.

    Alternative: `--some-option=true`
  - `--no-some-option` sets `--some-option` to false.

    Alternative: `--some-option=false`
  - Example: `ng new --no-create-application`
- Array options:
  - Space separated or repeated
  - `--option value1 value2 …`
  - `--option value1 --option value2 …`

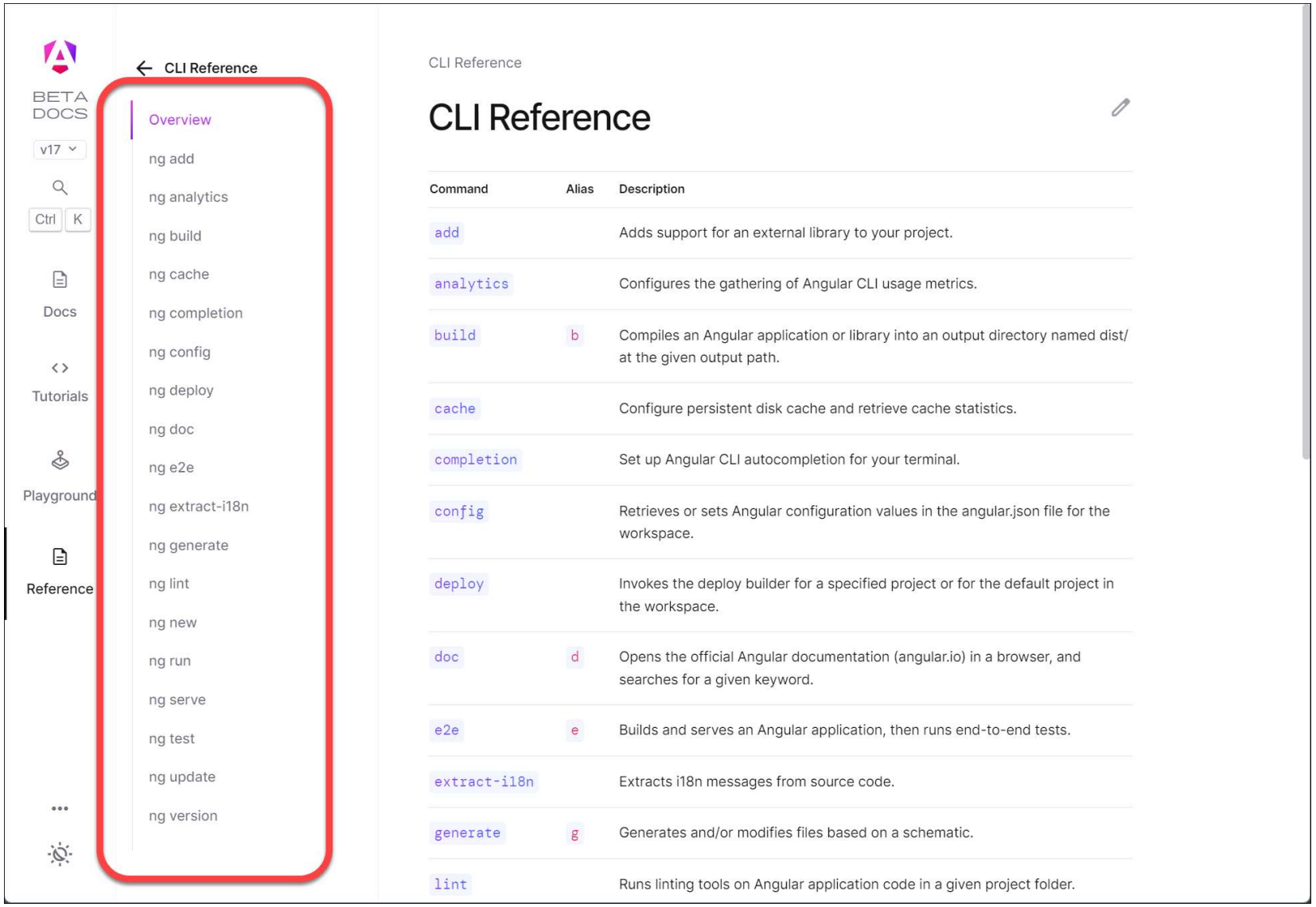
Creating a new application – `ng new`

- Creating an application: like before, using `ng new application-name`
- New options
  - NO more `Routing Y/N` – now enabled by default
  - Picking a `CSS variant` – same
  - `Enable pre-rendering SSG / SSR` (default: `No`)

```
PS C:\Users\Gebruiker\Desktop> ng new angular17
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
CREATE angular17/angular.json (2702 bytes)
CREATE angular17/package.json (1078 bytes)
CREATE angular17/README.md (1090 bytes)
CREATE angular17/tsconfig.json (936 bytes)
CREATE angular17/.editorconfig (290 bytes)
CREATE angular17/.gitignore (590 bytes)
CREATE angular17/tsconfig.app.json (277 bytes)
CREATE angular17/tsconfig.spec.json (287 bytes)
CREATE angular17/.vscode/extensions.json (134 bytes)
CREATE angular17/.vscode/launch.json (490 bytes)
CREATE angular17/.vscode/tasks.json (980 bytes)
CREATE angular17/src/main.ts (256 bytes)
CREATE angular17/src/favicon.ico (15086 bytes)
CREATE angular17/src/index.html (308 bytes)
```


Demo

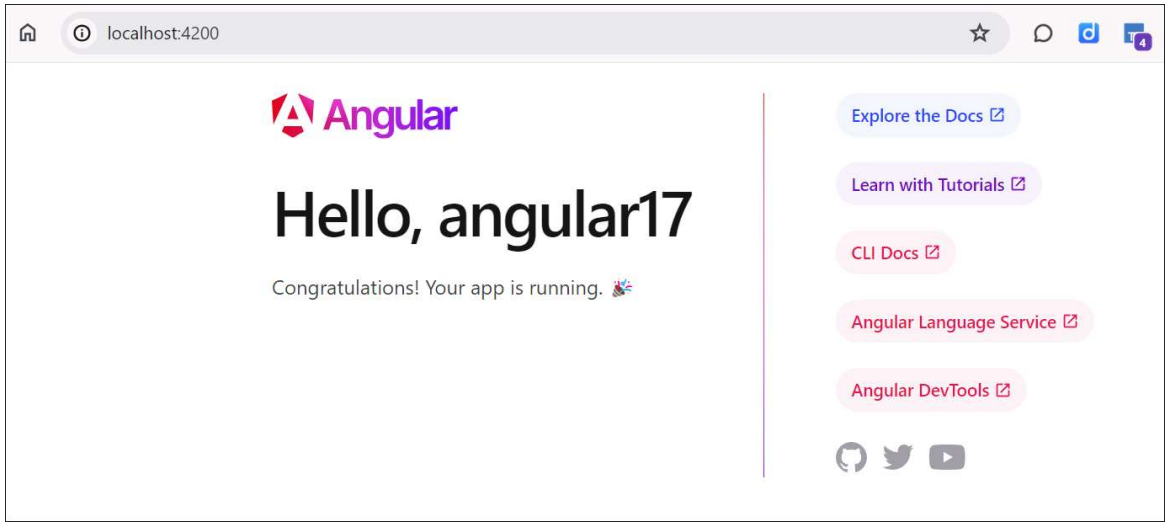
CLI reference

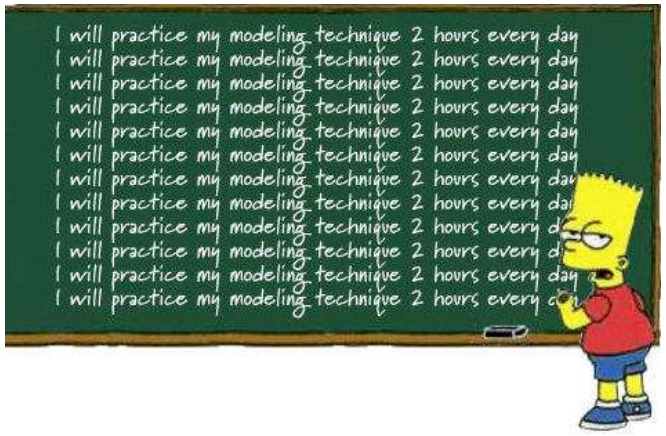


https://angular.dev/cli

Workshop

- Install angular CLI – make sure you have the latest version
  - `ng version`
- Create a new application, using the default values
- Open the application in your editor and run it
  - `ng serve –open`
- Check components, see what has changed already





NPM vs. NPX

What are the differences and when to use which?

Npm vs. npx

- After installing node.js, both are available.
- Main Difference:
  - npm *installs* and manages packages (locally or globally).
  - npx *runs* binaries from online or node modules *without* needing a global install.

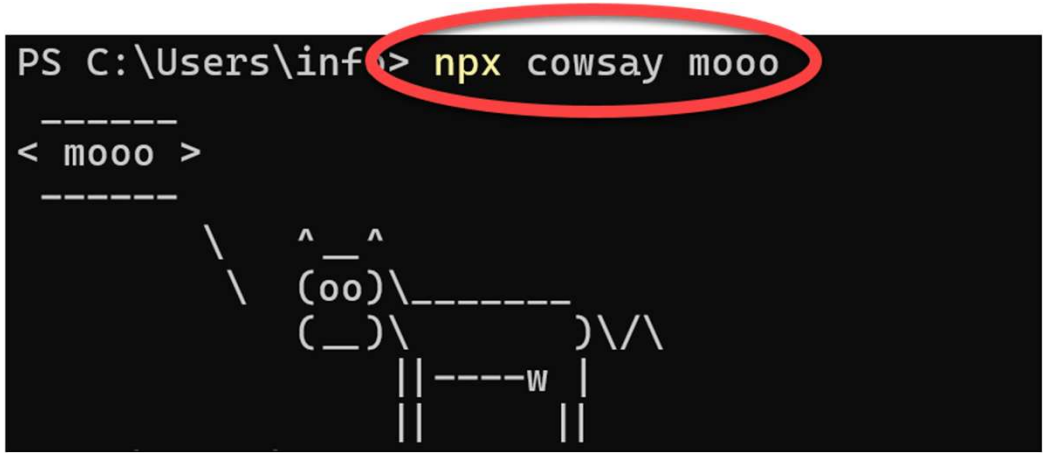
https://nodejs.org/en

When to use npm?

- Use npm when:
  - You want to *install* a package
    - For instance: `npm install eslint --save-dev` – will be installed on your machine
  - You want to add to `package.json` dependencies.
  - You want to *update/remove* packages.
  - You want to *use scripts* from your project's node_modules/.bin via npm run.

When to use npx?

- Use npx when:
  - You want to run a CLI tool once, without installing it globally.
    - For instance: `npx create-react-app my-new-app` or
    - npx eslint – run eslint once, without installing it.
  - You want to try something temporarily (like a specific version of a tool).
  - Package is installed temporarily and then removed



Important files

Often considered ‘boilerplate’, but necessary to run and configure angular apps.

Boilerplate files #1/3 - `package.json`

```
{
  "name": "hello-angular",
  "description": "Voorbeeldproject bij de training Angular (C) - info@kassenaar.com",
  "version": "0.0.1",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "16.0.0",
    "@angular/common": "16.0.0",
    "@angular/compiler": "16.0.0",
    "@angular/core": "16.0.0",
    "@angular/forms": "16.0.0",
    "rxjs": "^7,8.0",
    "zone.js": "^0.14.3"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.6.0",
    "@angular/cli": "6.3.7",
    "typescript": "~5.4.2"
  },
  "author": "Peter Kassenaar <info@kassenaar.com>"
}
```

Boilerplate files #2/3 - **`tsconfig.json`**

```
{
  "compileOnSave"  : false,
  "compilerOptions": {
   "outDir"                 : "./dist/out-tsc",
   "baseUrl"                : "src",
   "sourceMap"              : true,
   "declaration"            : false,
   "moduleResolution"       : "node",
   "emitDecoratorMetadata" : true,
   "experimentalDecorators": true,
   "target"                 : "es5",
   "typeRoots"              : [
     "node_modules/@types"
   ],
   "lib"                    : [
     "es2022",
     "dom"
   ]
  }
}
```

Boilerplate files #3/3 – **angular.json**

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "helloworld": {
      "root": "",
      "sourceRoot": "src",
      "projectType": "application",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist",
            "index": "src/index.html",
            "main": "src/main.ts",
            "tsConfig": "src/tsconfig.app.json",
      …
}
```

Additional files – Modern Angular apps

```ts
// app.config.ts
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
      provideZoneChangeDetection({ eventCoalescing: true }),
      provideRouter(routes)
    ]
};
```

```ts
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

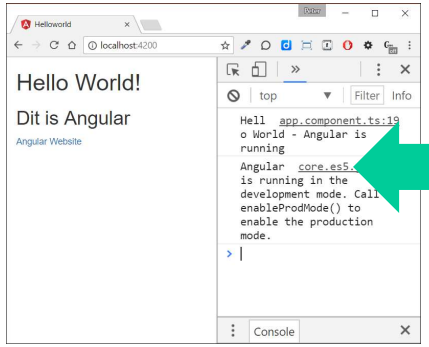
Standalone Components

Towards an `ngModule`-less future

Standalone components

- Traditionally: ngModule-based components
  - All components belong to an `@ngModule()`.
  - An `@ngModule()` acts as a container for similar functionality (Customers, Products, Login, and so on)
- Modern applications: standalone components
  - Components don’t have to belong to a module
  - You can mix & match!
  - Better performance
  - More component-level imports
  - In preview since Angular 13+, default in Angular 17+

Structure: existing Angular apps



main.ts / bootstrapper

ngModule / root module

AppComponent

Other components

Other components

Other components

Services

3rd party modules

Other modules

What are standalone components?

Regular Angular Components with the `standalone : true` option

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

@Component({
  selector: 'app-hello',
  standalone: true,
  imports: [],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {

}
```

Angular 19+ - standalone as default!

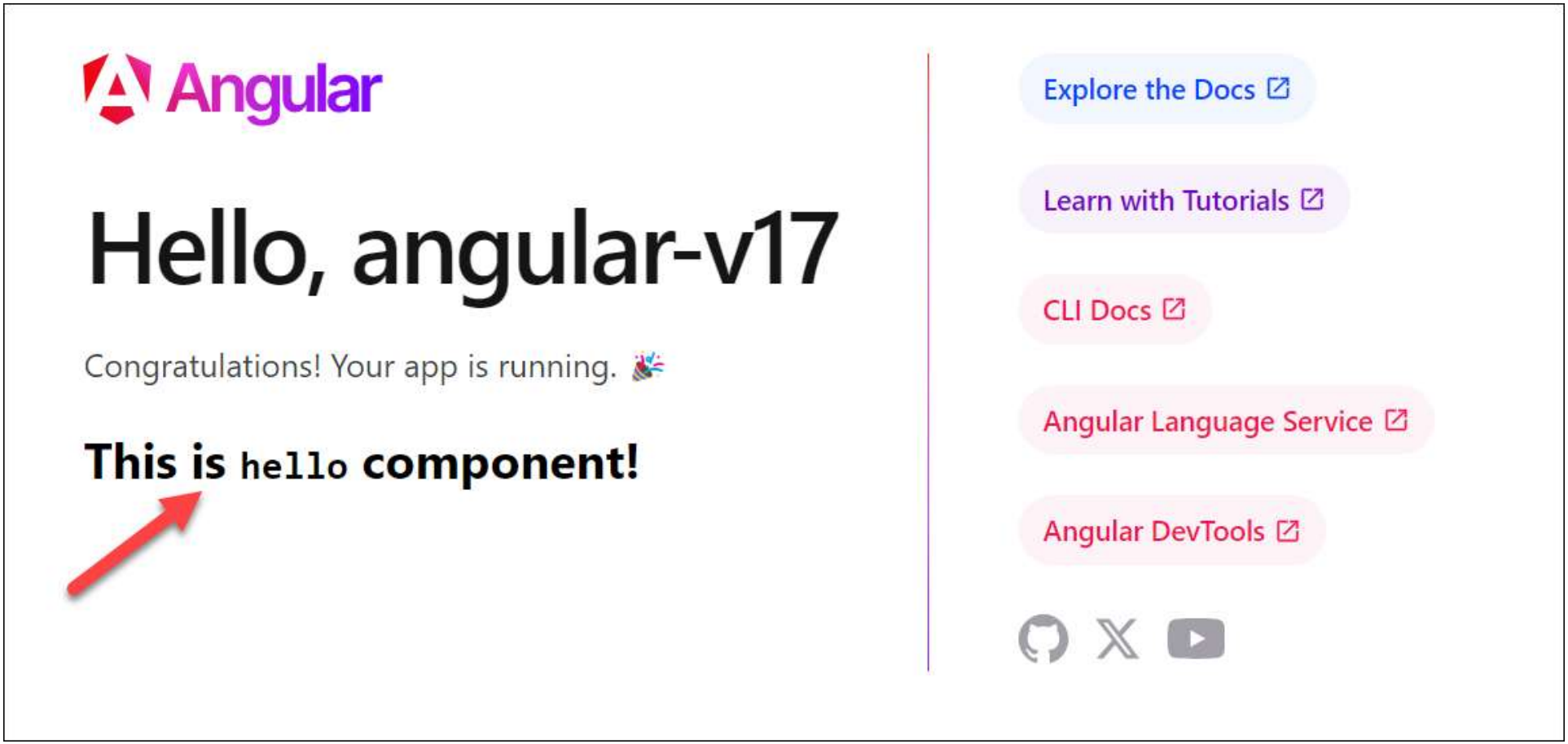
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
  // NOTE: No more standalone: true.
  // Annotate 'standalone: false' if this component belongs to a module!
})
export class AppComponent {
  //...

}
```

Using standalone components

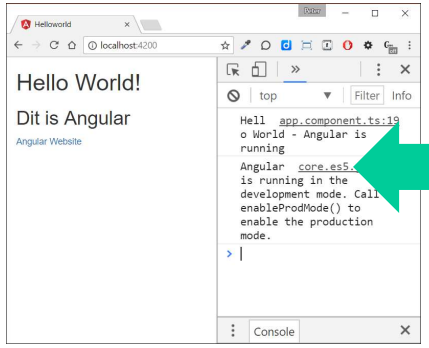
Simply import them in the component where you want to use it

```
import { Component } from '@angular/core';
import {HelloComponent} from "./hello/hello.component";

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ HelloComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
}
```

```
<h1>Hello, {{ title }}</h1>
<p>Congratulations! Your app is running. 🎉</p>
<app-hello/>
```


Angular
Hello, angular-v17
Congratulations! Your app is running.
This is hello component!
Explore the Docs
Learn with Tutorials
CLI Docs
Angular Language Service
Angular DevTools

Structure: modern Angular apps



main.ts / bootstrapper

AppComponent

Services

3rd party libs

Other components

Other components

Other components

Other components

(Note: no Module anymore, as this is now handled by standalone components)

Note

- Use `import {…} from …`
  - Otherwise it will NOT work – but NO ERROR will be thrown!
  - The browser just sees an unknown tag and renders nothing
  - Most IDE's handle this automatically / show warning

```
1   import { Component } from '@angular/core';
2   import {HelloComponent} from "./hello/hello.component";
3
    1+ usages   • PeterKassenaar
```

Using Standard Directives

- Standard directives are made available as standalone directives
- We have to `import` them in the component

```
<h2 *ngIf="username">
  Username: <code>{{ username }}</code>
</h2>
```

```
import { Component } from '@angular/core';
import {NgClass, NgFor, NgIf} from "@angular/common";

@Component({
  selector: 'app-hello',
  standalone: true,
  imports: [NgIf, NgClass, NgFor],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {
  username = 'info@kassenaar.com';
}
```

Standalone Pipes, Directives

- The same is true for standalone Pipes, Directives and more
  - Just add the `standalone: true` flag

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

@Pipe({
  name: 'capitalize',
  standalone: true
})
export class CapitalizePipe implements PipeTransform {

  transform(value: string):string {
    return value.toUpperCase();
  }
}
```

```
@Component({
  selector: 'app-hello',
  standalone: true,
  imports: [CapitalizePipe],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
```

Why Standalone components?

- Removing the need for `NgModules` – no extra complexity
  - Handy for beginners
- But: "I have to import everything in that component. Really?"
  - Yes – but IDE's handle that automatically for you
- Now, components can be lazy loaded instead of complete modules
  - More modularization, less monolithic
  - Performance!

> *The main benefit of standalone components is that they make it trivial to develop a fully lazy-loaded application, or migrate an existing application and make it fully lazy-loaded.*

Mixing and matching

- No need for a 'big bang' when refactoring
- Using standalone components in NgModule based applications
- Standalone components act as other modules, so `import` them:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {HelloComponent} from "../hello/hello.component";

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

Using NgModules in standalone components

- Export the component from the module as usual
- Import the module in the component as (now) usual

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

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [HelloComponent, CustomerModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
```

Combined NgModules/Standalone

Lazy loading with standalone components

No more boilerplate overhead

Classic lazy loading – with NgModules

Previously: without standalone components, lazy loading was done via modules

```
export const routes: Routes = [
  {
    path: "customer",
    loadChildren: () =>
      import("./customer/customer.module").
          then((m) => m.CustomerModule),
  },
];
```

boilerplate:

- creating the lazy loaded module
- import all the dependencies manually
- configure the module in the router

Lazy loading with standalone components

- Use `loadComponent` in routing config to point to the top-level standalone component
- Also import RouterOutlet in `<app-root>` component (this is done by default)

```
{
  path: "hello",
  loadComponent: () =>
    import("./hello/hello.component").then(m => m.HelloComponent)
}
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, …],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {…}
```

Configuring the router

Use `app.config.ts` to configure the app and router

```
// app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

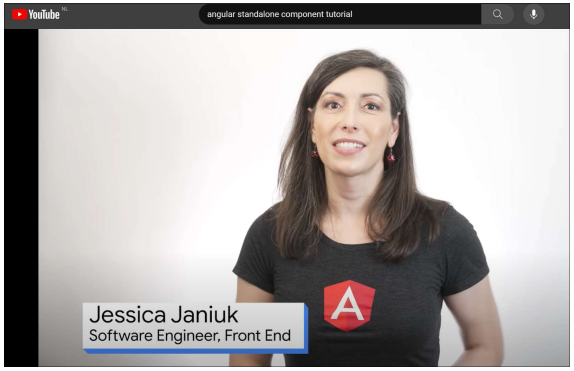
Conclusion

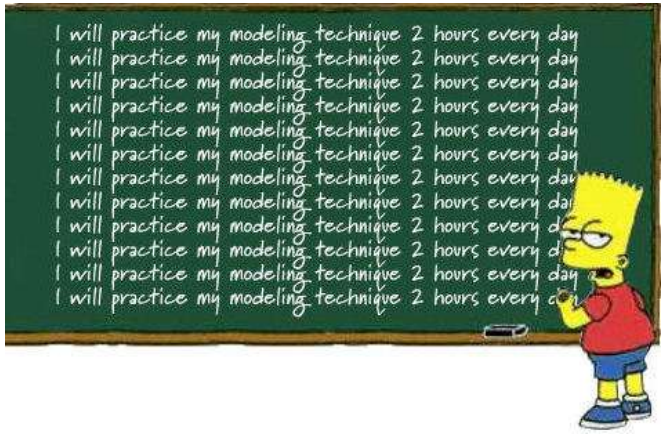
- No more (e.g. a lot less) boilerplate in configuring routing
- Use the Angular standalone API's to bootstrap the application instead of using NgModules.
  - `bootstrapapplication (topLevelComponent, configOptions)`

Workshop

- In your application, create some new (standalone) components
- Nest the components in each other – see how they are imported
- Optional: Create a lazy loaded route to the new component(s)

- Further reading: https://blog.angular-university.io/angular-standalone-components/
- https://www.youtube.com/watch?v=x5PZwb4XurU





Flow Control statements

New options for Template Syntax and Conditional Rendering

Control Flow in components: `@if`

- Previously: `*ngIf="…"`
- New: `@if (condition) { … }`

Invalid

```
<div @if="showTitle">
  {{  title }}
</div>
```

Valid

```
@if (showTitle){
  <div>
    {{  title }}
  </div>
}
```

Alternative `@else`

- Conditional Control Flow: `@if-@else`
- Note: no nesting!

Invalid

```
@if (showTitle){
  <div>
    {{  title }}
  </div>
  @else{
    <div>...</div>
  }
}
```

Valid

```
@if (showTitle) {
  <div>
    {{ title }}
  </div>
} @else {
  <div>Please set a title...</div>
}
```

Repeating items: `@for`

- Previously: `*ngFor="let item of items"`
- New: `@for (item of items; track item.id) {…}`

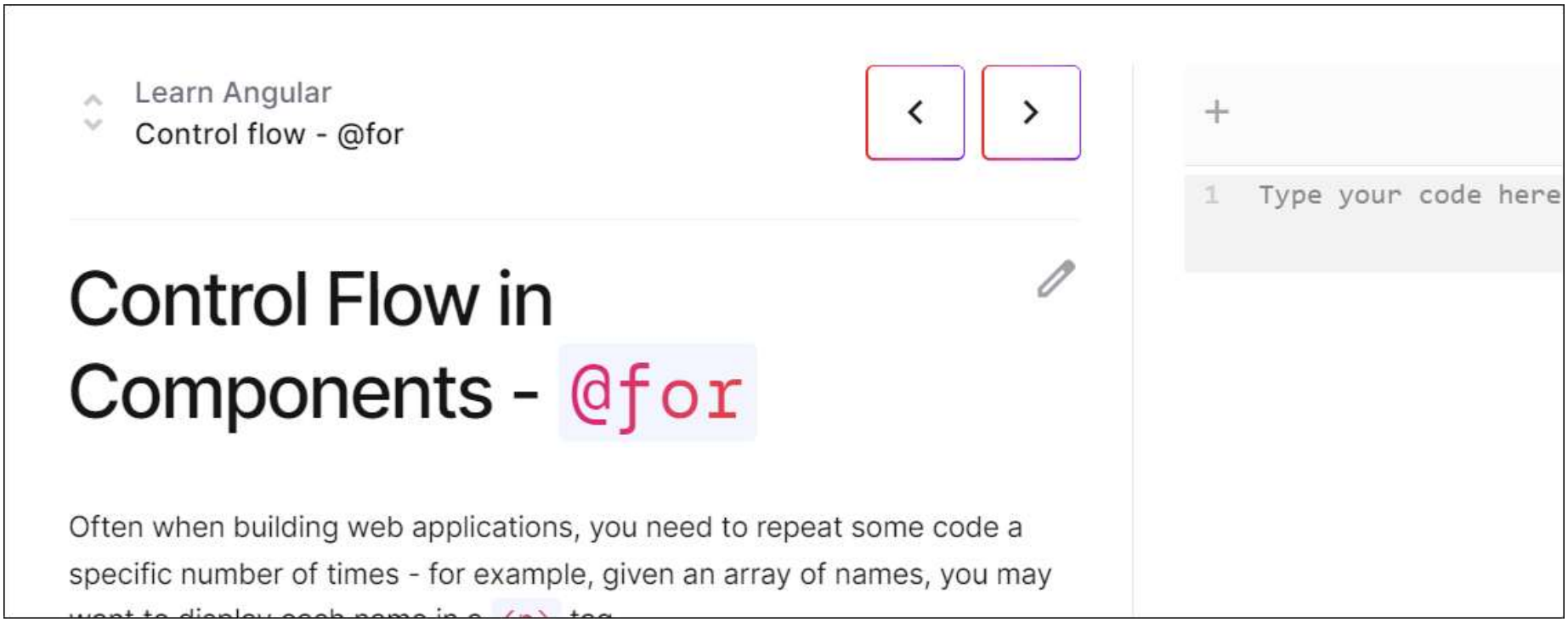
```
cities : City[] = [
  {id: 1, name: 'Amsterdam', country: 'NL'},
  {id: 2, name: 'Berlin', country: 'GER'},
  {id: 3, name: 'Tokyo', country: 'JAP'},
]
```

```
<ul>
  @for (city of cities; track city.id) {
    <li>{{ city.id }} - {{ city.name }}</li>
  }
</ul>
```

- 1 - Amsterdam
- 2 - Berlin
- 3 - Tokyo

Mandatory – tracking property

- Set a unique tracking property.
  - Optional in Angular 16 and lower (function `track by`)
  - Mandatory in Angular 17+ (=better rendering performance)
  - SO, for instance `@for (os of operatingSystems; track os.id)`



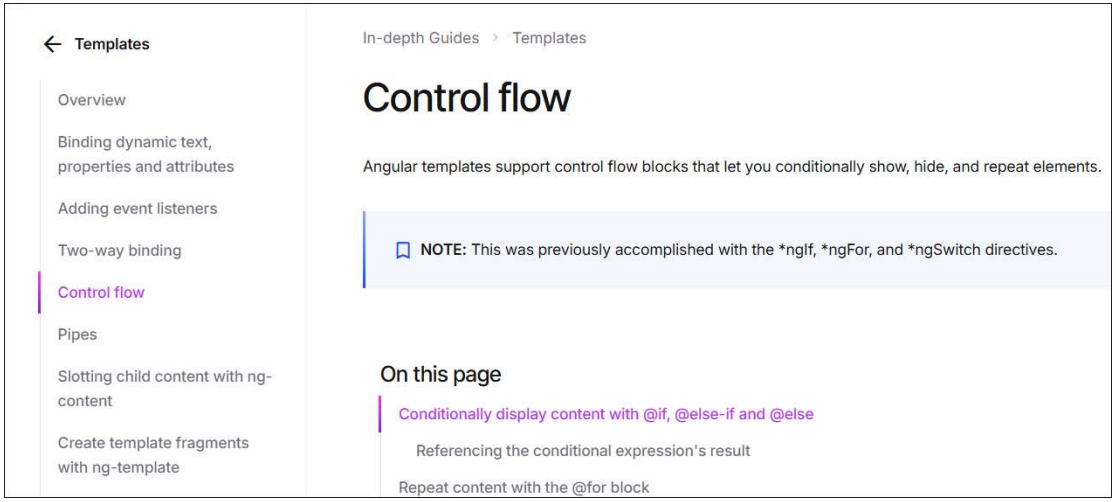
https://angular.dev/tutorials/learn-angular/control-flow-for

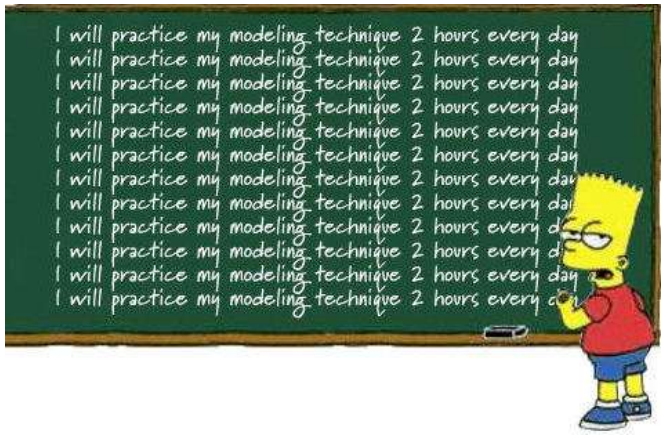
Unchanged in Angular 17+

- Property binding
  - `<img [src]="imgUrl" alt="… >`
- Event binding
  - `<button (click)="clickHandler()">`
- Input binding
  - `@Input() userID="…"`
- Output binding
  - `@Output() increment = new EventEmitter<number>();`

Workshop

- In your app, create a component with some variables and an array with data
- Show them conditionally (`@if`) and in a loop (`@for`) in the UI
- Check out the documentation on (new) template syntax on control flow at https://angular.dev/guide/templates/control-flow





Dependency Injection

No more using the constructor() for DI

Traditional – constructor based DI

- Not Wrong! This is still valid in Angular 17+

```
export class InjectComponent {

  // using classic constructor based Dependency Injection:
  constructor(public userService: UserService) {
  }

  //…
}
```

Alternative - using the `inject()` function

- Already available since Angular 14
  - but now a lot more useful!
  - Ditching *Constructor based Dependency Injection*
- Technical benefits
  - Readability benefits – a lot
  - It's a lot more clearer what dependencies a component needs – even if you don't fully understand DI.
  - Inheritance becomes much simpler. No need to use `super(service1, service2)` … on `extends BaseComponent`

```
export class InjectComponent {
  // using inject() based Dependency Injection
  userService = inject(UserService);
}
```

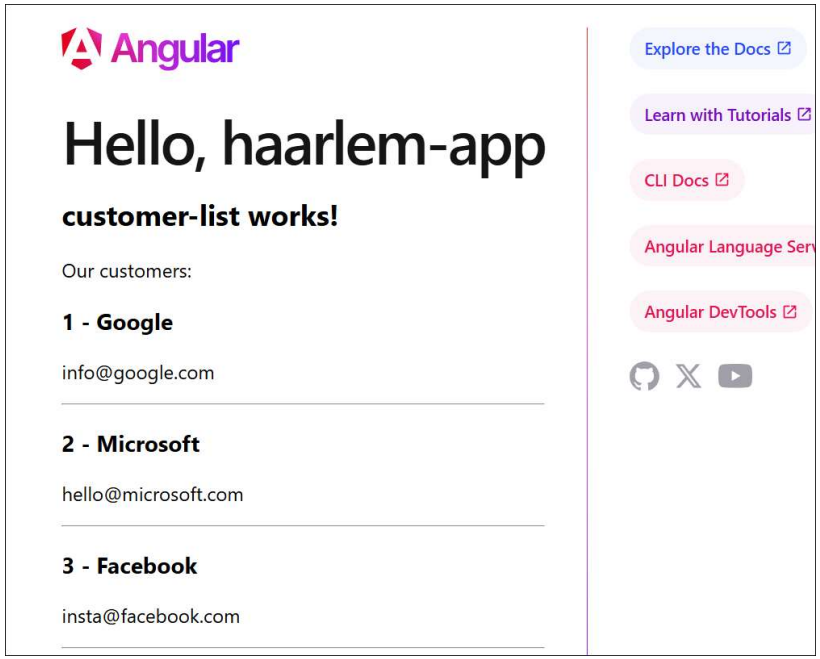
Inject only during constructor-time

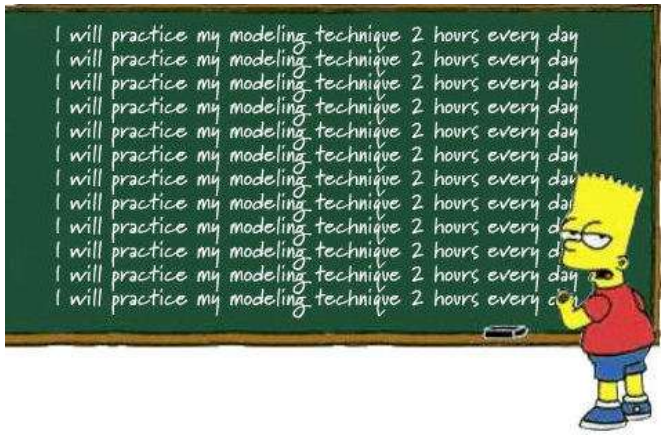
- Note: the `inject()` function only works when constructing the component
- So you CAN NOT use `inject()` in other methods

```
export class InjectComponent {

  // works - executed in construction phase of component
  userService = inject(UserService);

  ngOnInit(){
    // INVALID: won't work, this function is executed later
    inject(SomeService).someValue
  }
  someMethod(){
    // INVALID: won't work, this function is executed later
    inject(SomeService).someValue
  }
}
```

Workshop

- Create a service in you app, providing some functionality
- Use the `inject()` function to inject the function in a component.
- Show data from the service in the component
  - Sample output:





Lifecycle hooks

Perform certain actions during the component lifecycle, from creation until destruction

Not mandatory

- Lifecycle hooks are `interfaces`. They are never mandatory.
- Lifecycle hooks are annotated as `ngSomeAction()`…
- It is best practice to `implements` the lifecycle hooks in the component class (though Angular will work without `implements`)

```
export class CustomerListComponent implements OnInit, OnDestroy {
  //...

  ngOnInit(): void {
    //...
  }

  ngOnDestroy(): void {
    //...
  }
}
```

Execution Order

Phase	Method	Summary
Creation	`constructor`	Standard JavaScript class constructor. Runs when Angular instantiates the component.
Change Detection	`ngOnInit`	Runs once after Angular has initialized all the component's inputs.
	`ngOnChanges`	Runs every time the component's inputs have changed.
	`ngDoCheck`	Runs every time this component is checked for changes.
	`ngAfterContentInit`	Runs once after the component's *content* has been initialized.
	`ngAfterContentChecked`	Runs every time this component content has been checked for changes.
	`ngAfterViewInit`	Runs once after the component's *view* has been initialized.
	`ngAfterViewChecked`	Runs every time the component's view has been checked for changes.
Rendering	`afterNextRender`	Runs once the next time that **all** components have been rendered to the DOM.
	`afterRender`	Runs every time **all** components have been rendered to the DOM.
Destruction	`ngOnDestroy`	Runs once before the component is destroyed.

https://angular.dev/guide/components/lifecycle#

See documentation

- Most used:
  - `ngOnInit()`
  - `ngOnChanges()`
  - `ngOnDestroy()`
- But: Your mileage may vary!
- See documentation on their exact behavior

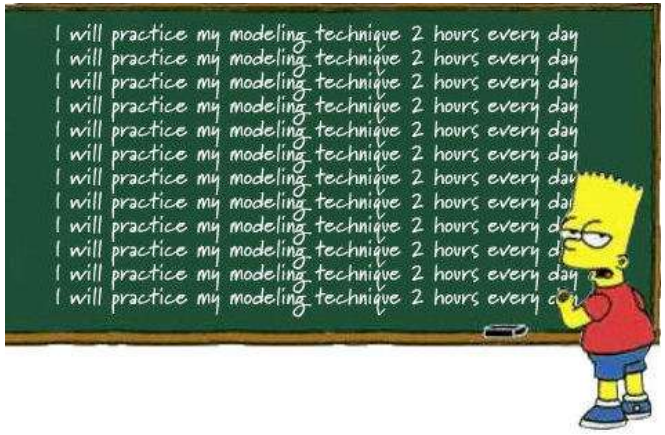
Inspecting changes

The `ngOnChanges` method accepts one `SimpleChanges` argument. This object is a `Record` mapping each component input name to a `SimpleChange` object. Each `SimpleChange` contains the input's previous value, its current value, and a flag for whether this is the first time the input has changed.

```
@Component({
  /* ... */
})
export class UserProfile {
  @Input() name: string = '';

  ngOnChanges(changes: SimpleChanges) {
    for (const inputName in changes) {
      const inputValues = changes[inputName];
      console.log(`Previous ${inputName} == ${inputValues.previousValue}`);
      console.log(`Current ${inputName} == ${inputValues.currentValue}`);
      console.log(`Is first ${inputName} change == ${inputValues.firstChange}`);
```

Workshop

- Implement the three most used lifecycle hooks in a component.
  - `ngOnInit()`
  - `ngOnChanges()` – when is this one called? Demonstrate this!
  - `ngOnDestroy()`
- Add `console.log()` statements to them, so you can see they are actually called.



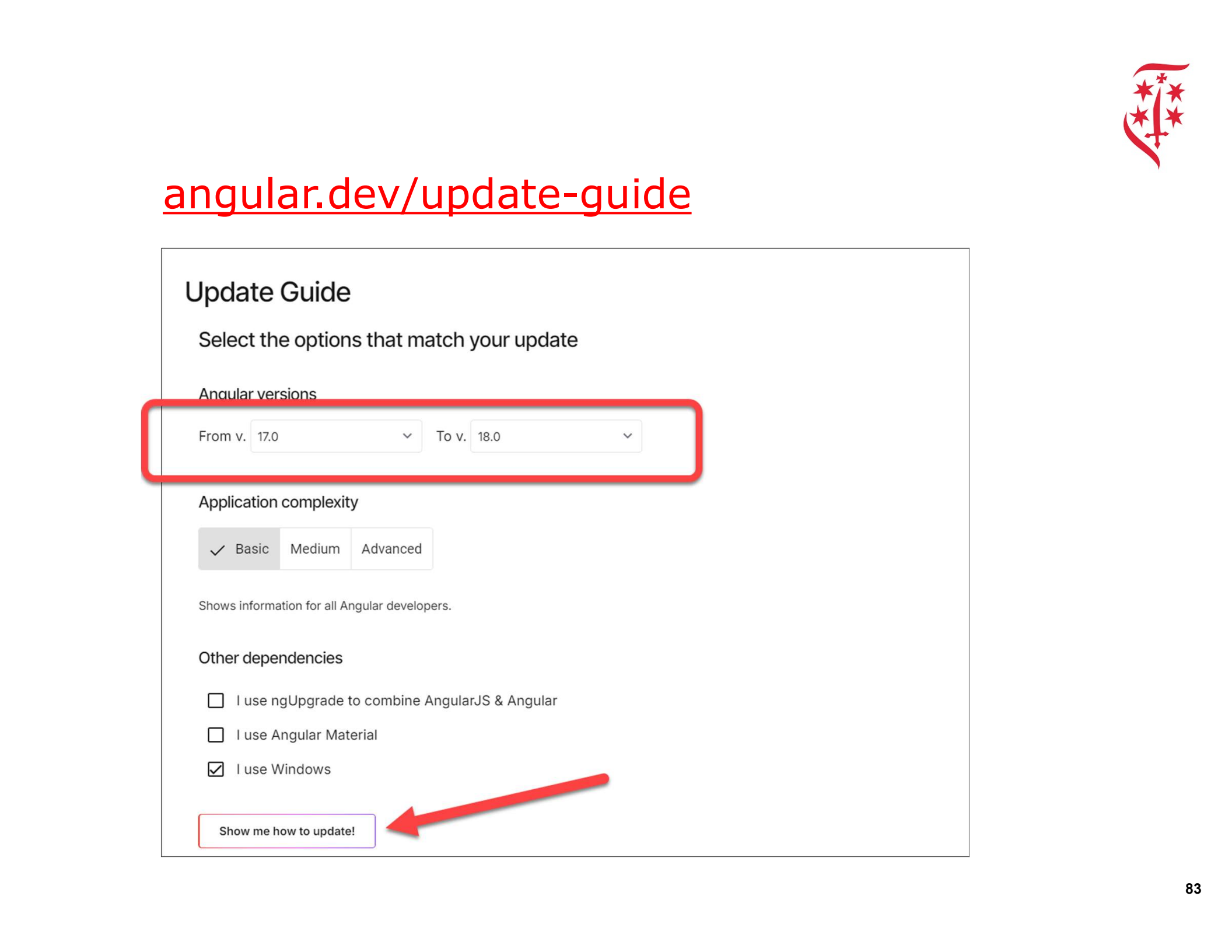
Updating Angular

Angular is moving fast. New major release every 6 months

@IgorMinar, Nov. 9, 2018

Angular updates

- Important milestones
  - Angular 2 – breaking change with Angular JS (1.x).
    - Syntax, structure, architecture
  - Angular 6 – Introduction modern Angular CLI
  - Angular 9 – Ivy Compiler
  - Angular 14 – Standalone components
  - Angular 17 – ‘Modern’ Angular, ‘renaissance’
  - Angular 19 – Current



angular.dev/update-guide

Update Guide

Select the options that match your update

Angular versions

From v. 17.0 To v. 18.0

Application complexity

✓ Basic | Medium | Advanced

Shows information for all Angular developers.

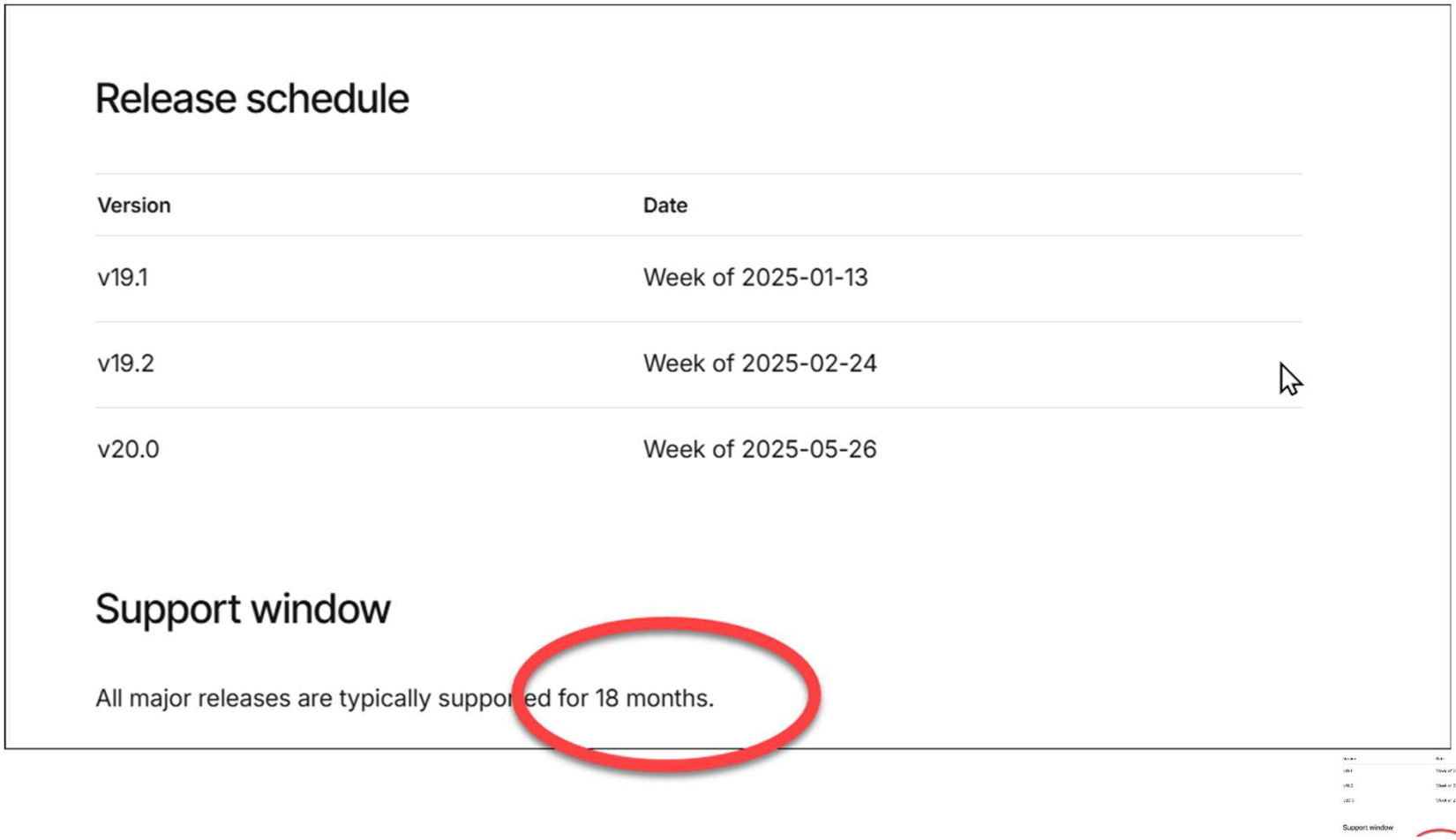
Other dependencies

- ☐ I use ngUpgrade to combine AngularJS & Angular
- ☐ I use Angular Material
- ☑ I use Windows

Show me how to update!

Angular Versies en -Long Time Support

→ https://angular.dev/reference/releases



Release schedule

Version	Date
v19.1	Week of 2025-01-13
v19.2	Week of 2025-02-24
v20.0	Week of 2025-05-26

Support window

All major releases are typically supported for 18 months.

On updating – summary

- Use the Official Update Guide website
- Read the breaking changes
- Create a new branch(!)
- Print out the update process, follow step-by-step

Getters vs methods

What are similarities/differences, why would you choose one over the other?

Getters and methods

- Both enable you to get the value of a variable
- *Functionally* they are interchangable.
  - Which one to use? Personal preference, or corporate decision
- *Syntactically* they are different
- Getter - Is just a *property accessor*
  - **Pro** – clean syntax, semantically signals that you are *accessing a property*, not calling a function
  - **Con** – re-evaluated on every change detection-cycle (performance)

```
// getter
get fullName(): string {
  return this.firstName + ' ' + this.lastName;
}
```

```
<p>{{ fullName }}</p>
```

Methods

- Methods are just regular class functions
- They are called from the template – which was considered bad practice some time ago (=depends on what the function does)
  - Pro – can take parameters
  - Con – less clean in templates, also possible performance issues

```
// method
getFullName(): string {
  return this.firstName + ' ' + this.lastName;
}
```

```
<p>{{ getFullName() }}</p>
```

Verdict

Use `get` when exposing a value as if it were a property

— cleaner, more idiomatic, intention-revealing.

Use methods if parameters or side-effects are needed.

Avoid calling either in templates if they're expensive.

"What should I be using?"

The ongoing debate of `type` *vs* `interface`

What have we got?

- In TypeScript, we have the keyword `type` as well as `interface`
- `type`: use the equal sign (=)
- `interface`: use direct curly brace notation

```ts
// A type with some properties
type Person = {
    name: string;
    age: number;
}

// Same properties, now in an interface.
interface IPerson {
    name: string;
    age: number;
}
```

?? Any difference ??

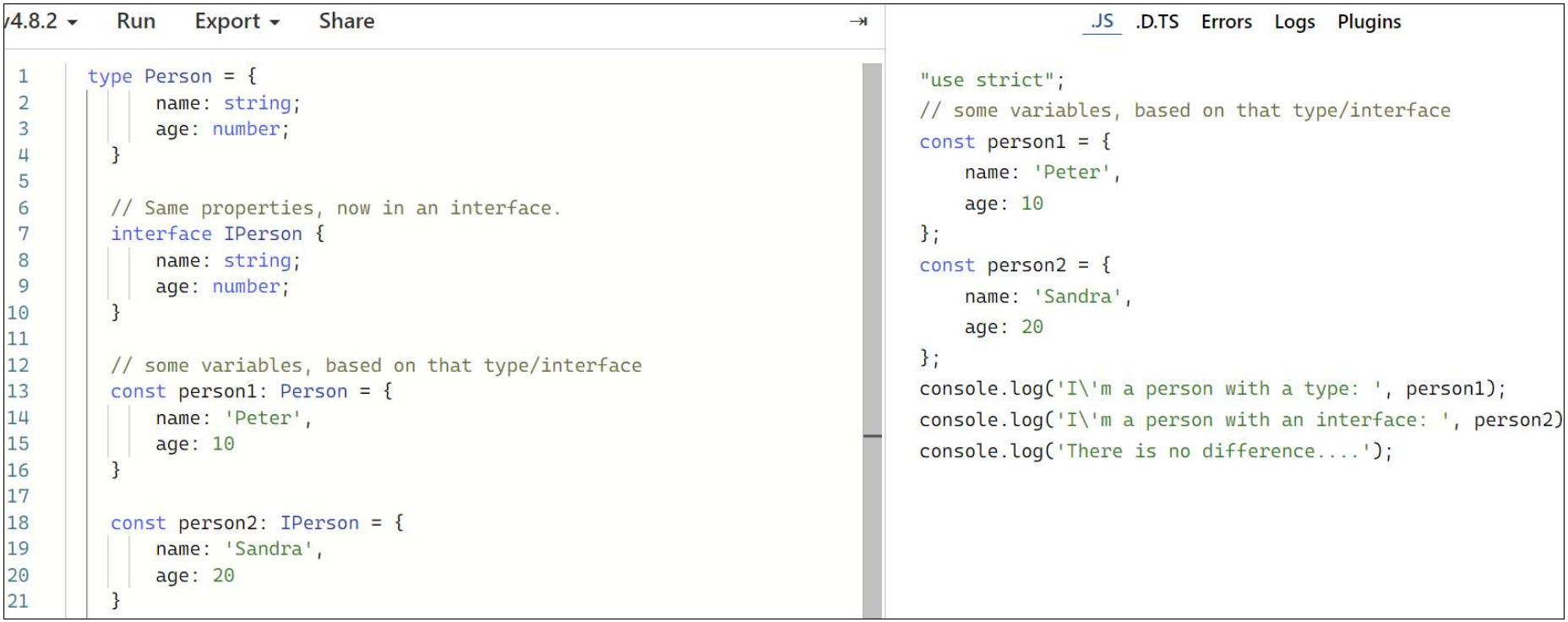
Example code: `../40-types-vs-interface.ts`

Short answer

There is no functional difference.

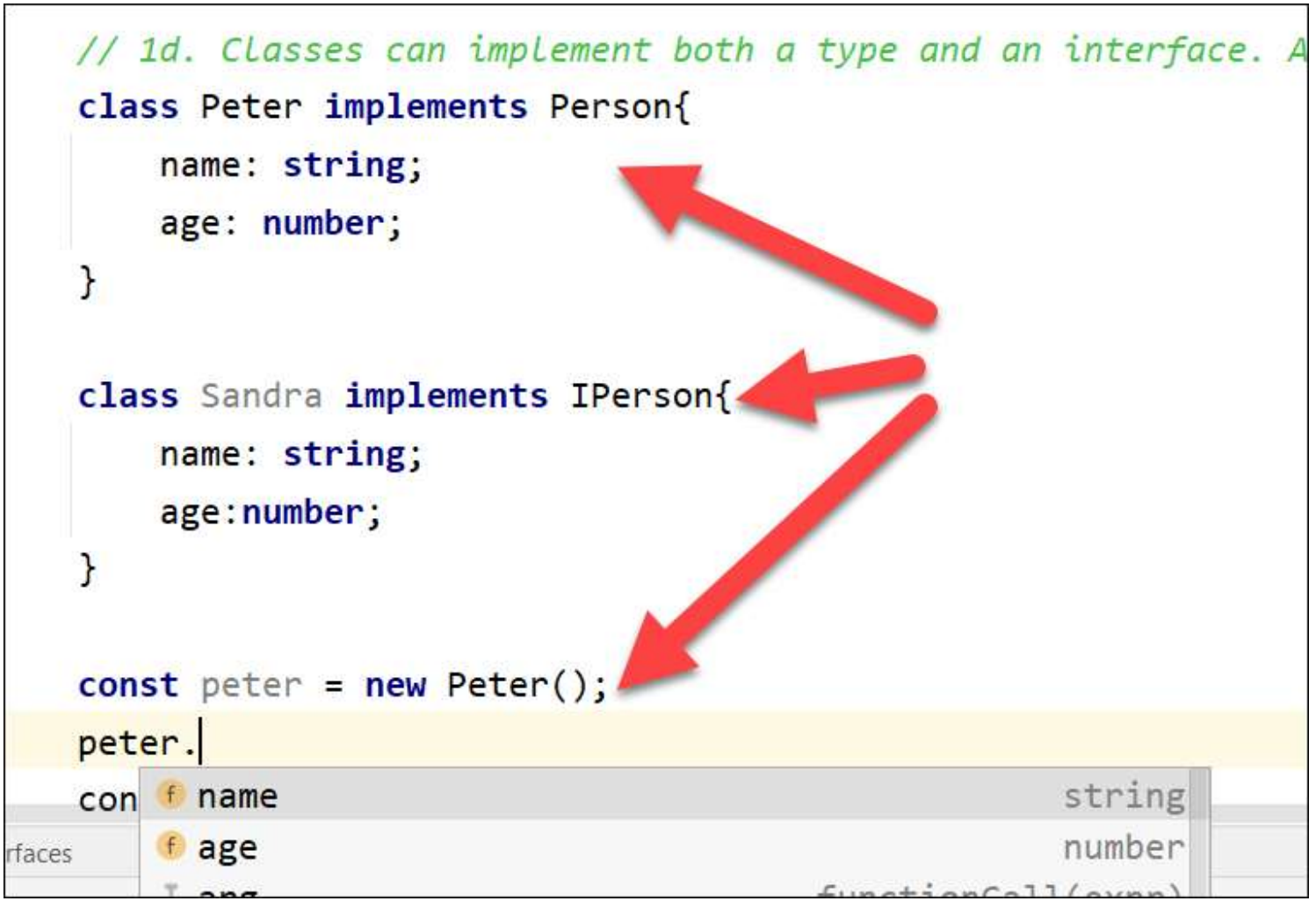
Use whatever floats your boat. In the compiled JavaScript both are gone!

TypeScript playground



Works as expected

Classes can implement both



So?

Use your *personal preference*, or company standard.

But:

Under the covers there are some differences in how TypeScript treats both.

Interfaces

Unique features of interfaces

Unique features of interfaces

- 1. Interfaces can `extend` other interfaces.
- With a `type` this is not possible

```
interface IPerson {
    name: string;
    age: number;
}
```

```
interface Peter extends IPerson{
    teacher: boolean;
}

const me: Peter ={
    name: 'Peter',
    age: 10,
    teacher: true
}
```

2. Declaration merging

- The code below is valid (‘Define an interface multiple times’)
- The resulting object needs to have the props of *both* interfaces

```
interface IPerson {
    name: string;
    age: number;
}
…
```

```
interface IPerson{
    city: string;
}
```

```
const harry: IPerson={
    nam
    age
}
// 2c.
```

TS2741: Property 'city' is missing in type '{ name: string; age: number; }' but required in type 'IPerson'.

40-types-vs-interface.ts(74, 9): 'city' is declared here.

Remove unused constant 'harry' Alt+Shift+Enter More actions... Alt+Enter

```
const types_vs_interfaces.harry: IPerson
```

3. Interfaces are geared to...

- Interfaces are more geared towards `objects`, `functions` and `classes`.
- They generally are more used in a `OOP-style` of programming.
- `Types` are more used in a functional way of programming, composing complex types from simple types

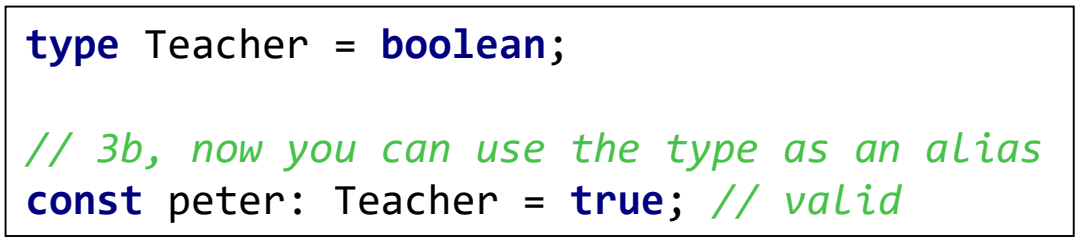
"Composition (types) over inheritance (interface)"

So: Your choice

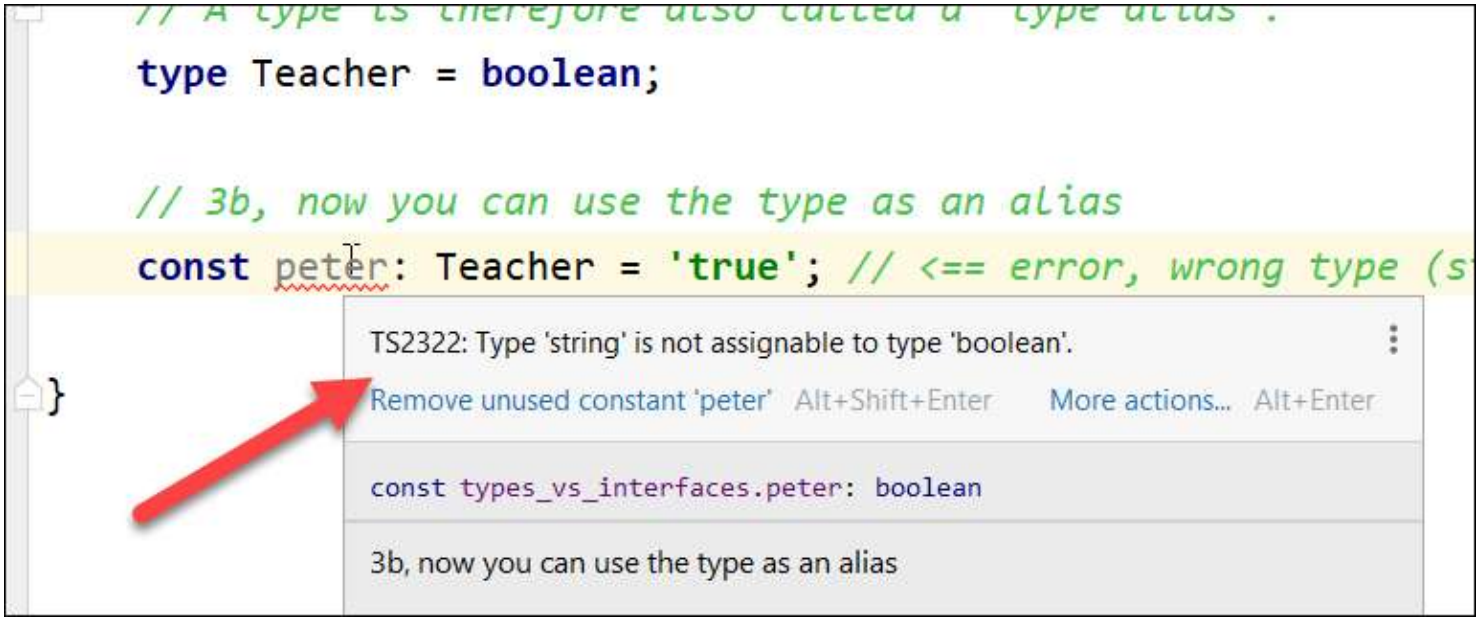
Types

Types are much like interfaces nowadays

Types are an alias for a shape of data



```
type Teacher = boolean;

// 3b, now you can use the type as an alias
const peter: Teacher = true; // valid
```



Types have Intersections and Unions

- Types cannot extend other types, as interfaces can
- But, types have subtypes, intersections and unions
    - More or less the same outcome, but different syntax.

```
// 3d. Intersection Types. Use the ampersand:
type tStudent = {
    student: boolean
}

// type tPerson is *intersected* with type tStudent
type tPerson ={
    name: string;
    age: number;
} & tStudent

// we need to use all the properties here, b/c of intersection type
const sandra : tPerson={
    name: 'Sandra',
    age: 20,
    student: true
}
```

Intersection Types

- Use the ampersand, pronounced as AND

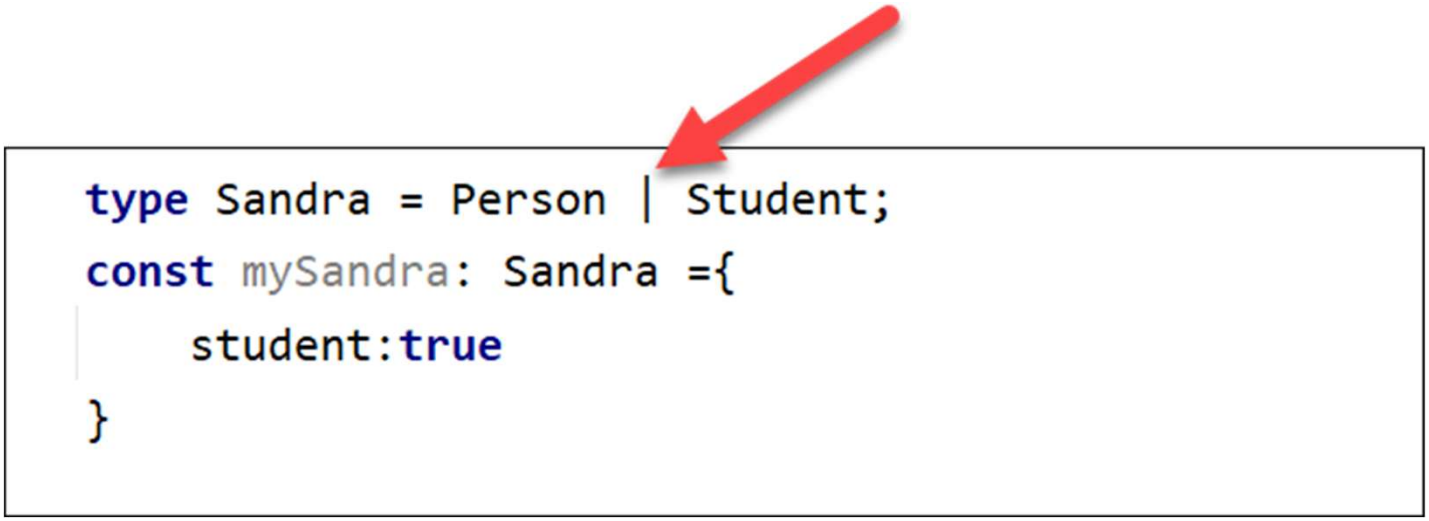
```
// 3e. Of course you can create additional types like so:
type Sandra = tPerson & Student;
```

```
const mySandra: Sandra = {
    name: 'Sandra',
    age: 20,
    // student: true // <== ERROR when omitted.
}
```

This can come in handy if you have data coming in from multiple endpoints and want to combine (‘compose’) it in one frontend type, but retain type safety

Union Types

- A `type` is of one `type` OR the other
- The pipe symbol | is pronounced OR
- They can also be merged together (using ALL the properties of given types)

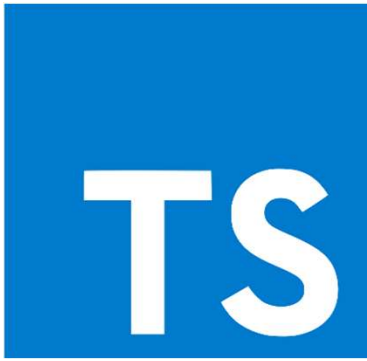


```
type Sandra = Person | Student;
const mySandra: Sandra ={
    student:true
}
```

So, unique features of types :

- Types are aliases for the shape of the data. They are static.
- Types can be intersected (&) or united (|)
- Sometimes easier to use than interfaces
  - In a composition/functional programming style environment
  - Personal preference!

- So again, types and interfaces are MOSTLY the SAME



Types and interfaces “In the wild”:

Using a lot of objects and/or classes? ➔ Use `interfaces`

Not? ➔ use `types`

But then again, it mostly doesn’t matter!

However: be consistent with

Yourself

Your team

Your company

`Public`, `private`, `protected` and `abstract`

- They are all keywords for TypeScript
  - Meaning you can use them on properties, functions, classes, and so on
  - They are developer aids – NOTHING remains in the compiled JavaScript code.
- `public`
  - Accessible from anywhere—inside the class, outside the class, subclasses, etc.
  - If you use nothing, public is used. It is the default value
- `private`
  - Only accessible within the class where it's defined. Not visible to subclasses or external code.
- `protected`
  - Like `private`, but accessible within subclasses as well. Not accessible from outside the class hierarchy.

Example on keywords

- `abstract`
  - Used in `abstract class` or abstract method.
  - Cannot be instantiated directly.
  - Forces subclasses to implement the abstract methods.

- You can then instantiate classes based on the (`abstract`) base classes, providing the (`protected`) members as parameters

```
abstract class Animal {
  protected name: string;

  constructor(name: string) {
    this.name = name;
  }

  abstract makeSound(): void;
  public move() {
    console.log(`${this.name} moves`);
  }
}

class Dog extends Animal {
  private breed: string;
  constructor(name: string, breed: string) {
    super(name);
    this.breed = breed;
  }
  makeSound() {
    console.log("Woof!");
  }
```

```
// usage of the classes above.
public myDog = new Dog("Max", "Labrador");
```

../abstract-public-private.component.ts

New `provideHttpClient`

In moduleless appllications, provide `http` via `app.config.ts`

Most common usage of observables: `http`

Modern applications are moduleless;

provide the imported `httpClient()`

```
// app.config.ts
import …;
import {provideHttpClient} from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
    …
  ]
};
```

Then: inject http in Component / service

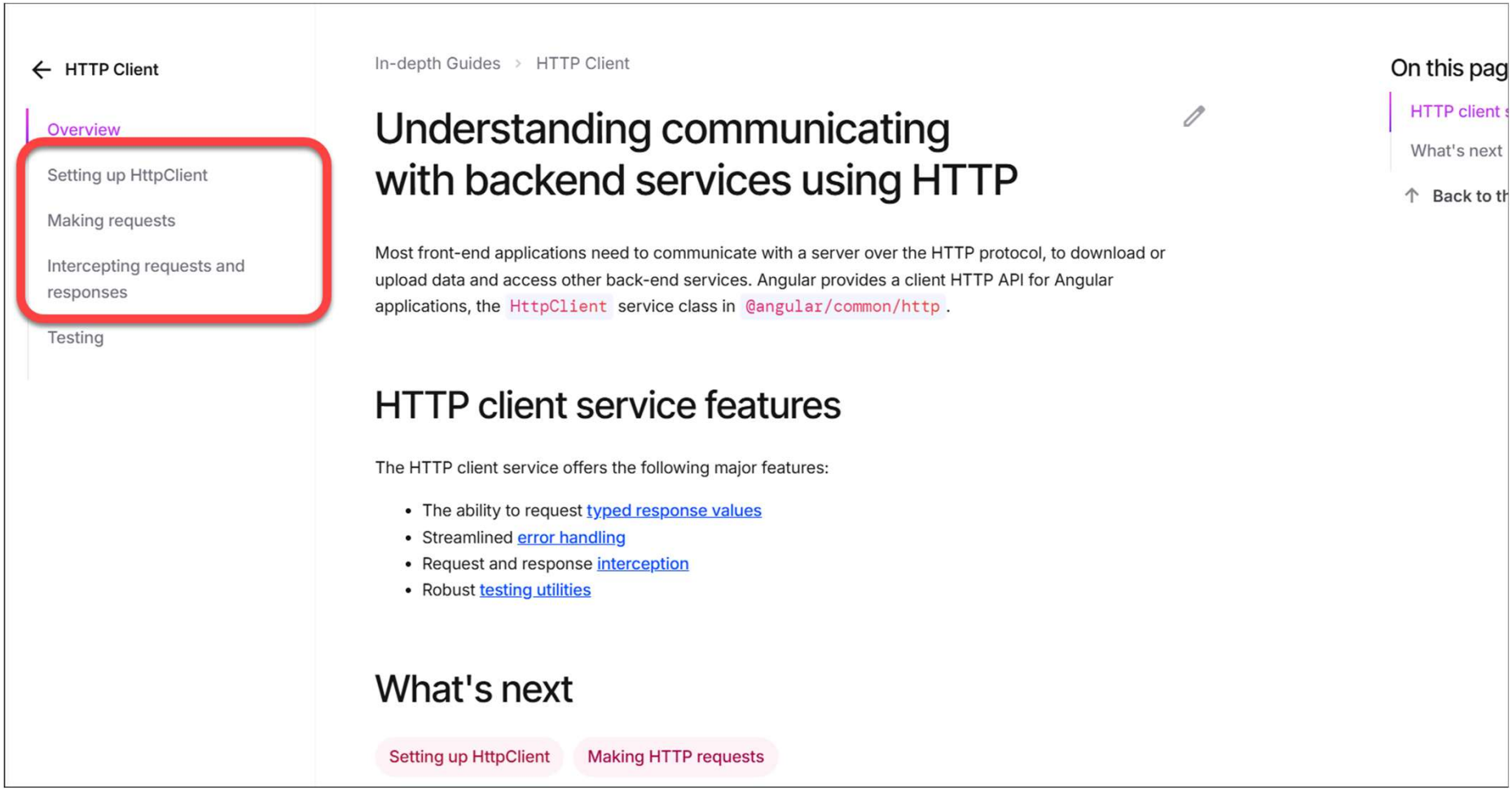
- Create an `http` variable, of type `HttpClient`
- Use Constructor Injection, or `inject()` function

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

}
```

```
export class AppComponent {
  // using inject()
  private http = inject (HttpClient);
}
```

https://angular.dev/guide/http

Documentation on `httpClient`



https://angular.dev/guide/http

Workshop

- Create a small component fetching (dummy) users from https://jsonplaceholder.typicode.com/users
  - Use the `provideHttpClient()` provider in `app.config.ts`
  - Inject the `HttpClient` in the component
  - Fetch and show the users in the UI,
  - Example output:
- Study options like `withFetch()` and `withInterceptors()` for yourself:
- https://angular.dev/guide/http/setup#providin dependency-injection