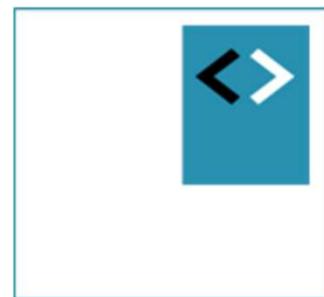




Gemeente
Haarlem

Angular - Maatwerk



Peter Kassenaar
info@kassenaar.com

Peter Kassenaar

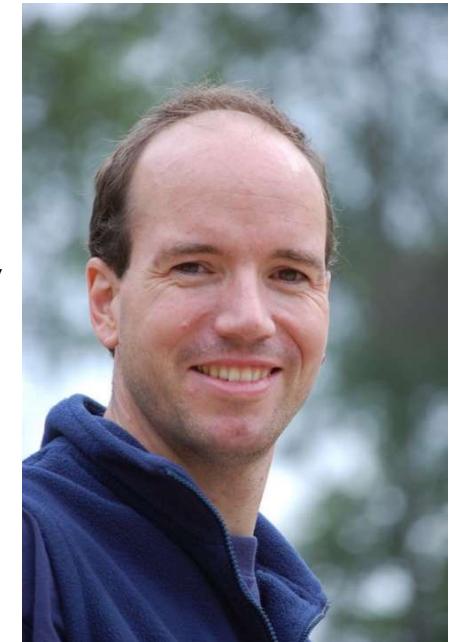


- Trainer, author, developer – since 1996
- Specialty: "*Everything JavaScript*"
- JavaScript, ES6, Angular, NodeJS, TypeScript, jQuery, Vue.js, React

www.kassenaar.com

info@kassenaar.com

VANDUUREN
MEDIA



ING

OHRA

zenito
BETERE ZEKERHEID
VOOR ONDERNEEMERS

Atos

euricom
A DIMENSION DATA COMPANY

woonbron
OBERON INTERACTIVE

sanoma

ROC West-Brabant

delta lloyd

**the eforum
FACTORY**

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



The screenshot shows the ProductIP website homepage. At the top is a navigation bar with links: Home (highlighted in orange), News, Events, Services, Pricing, Contact, Jobs, Log In, Register Now (highlighted in green), and icons for user profile and search. Below the navigation is a decorative section featuring five silver spheres hanging from thin black lines. To the right of this section is a sidebar with a yellow star icon, the number '10', and a blue square icon. The main content area contains the following text:

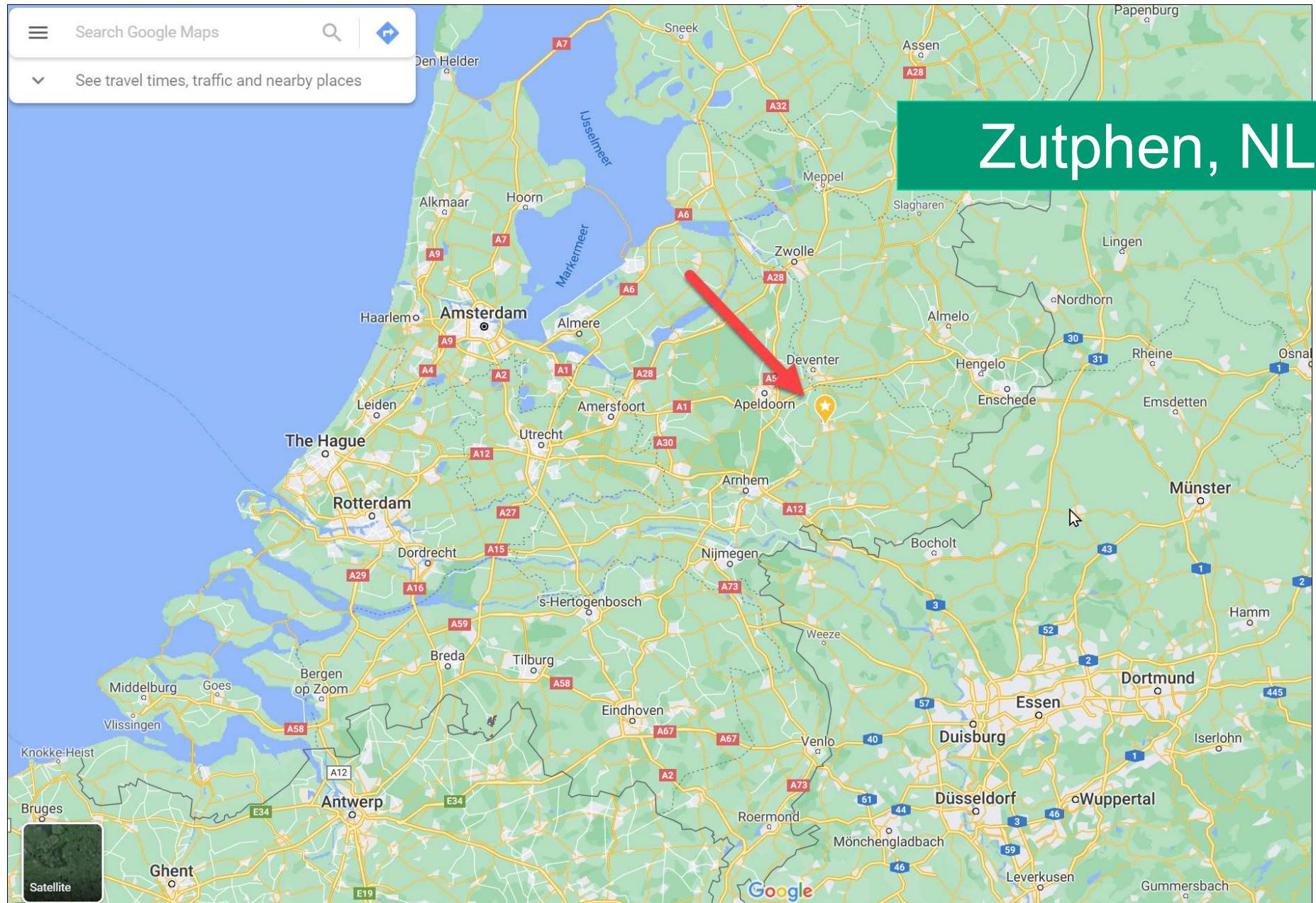
ProductIP provides relevant regulatory and compliance information for you to make mission critical product and supply chain business decisions

Knowledge • SaaS • Impact

Legal advice is everywhere. Now you can translate this into impact

In the bottom left corner is a blue circular icon with a white swirl pattern. The bottom of the page features a yellow banner with the text "Building a future proof business with an operation and supply chain in".

www.productip.com

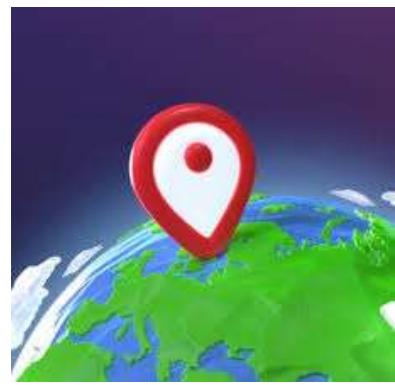




peterkassenaar



PeterKassenaar



PeterKassenaar



pkas06

github.com/PeterKassenaar/haarlem

The screenshot shows the GitHub repository page for `haarlem`. The repository is public and contains the following details:

- Code**: The active tab, showing a main branch and 1 branch.
- Issues**, **Pull requests**, **Actions**, **Projects**, **Wiki**, **Security**, **Insights**, **Settings**: Other navigation tabs.
- haarlem** (Public): The repository name and visibility.
- Pin** and **Unwatch 1**: Watch status and pinning options.
- main**: The main branch, with 1 Branch and 0 Tags.
- Go to file** search bar and **Add file** button.
- Code** dropdown menu.
- PeterKassenaar**: Initial commit, dated b8ea053 · now, with 1 Commit.
- .gitignore**, **LICENSE**, and **README.md**: Files listed with their commit history.
- README** and **MIT license**: Links to the respective files.
- haarlem**: A large header with the repository name.
- Slides and example code on the the training Angular - Gemeente Haarlem, spring 2025**: A note at the bottom of the repository page.

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

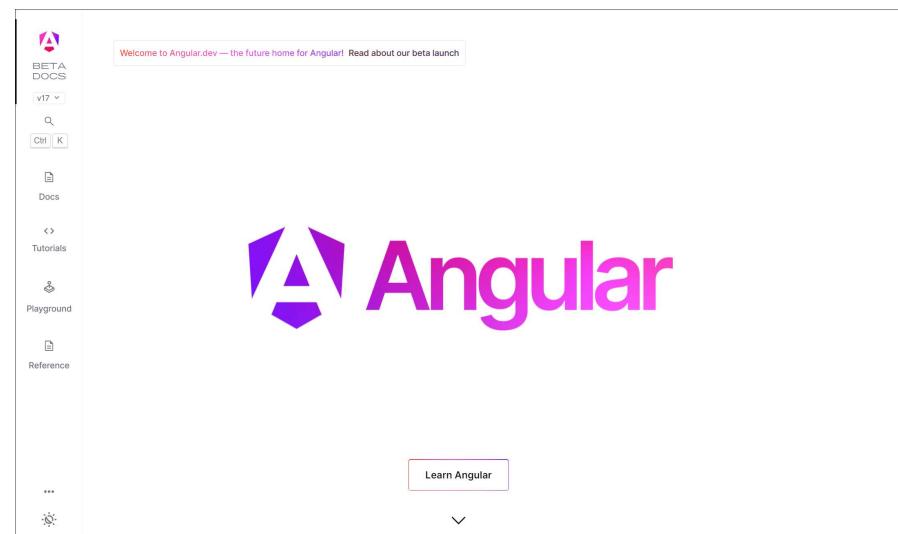
~16:00-16:15 - end



Material



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



angular.dev



Agenda - day #1

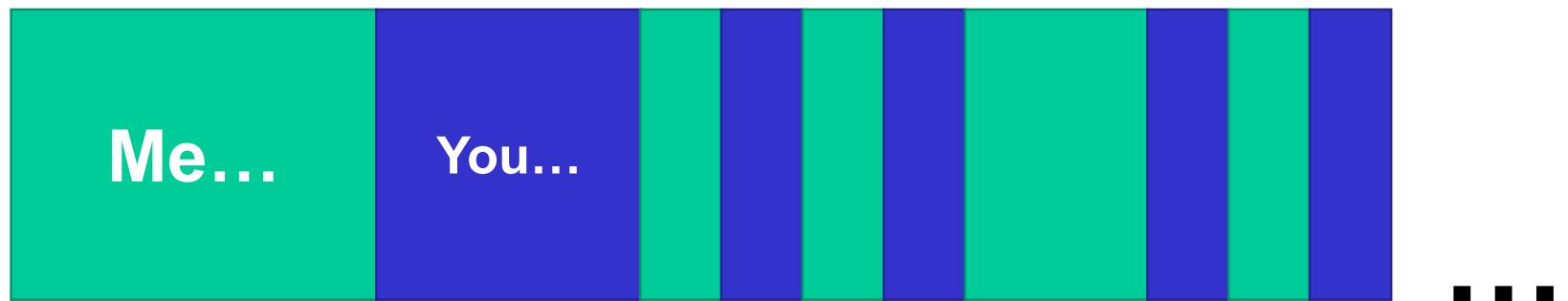
- **Introduction**
- **Angular New Features + refresher**
 - NPM, NPX, node_modules, webpack, Vite and more
 - @for, @if-then-else, @switch()
 - Standalone components and module-less defaults
 - Updating Angular versions
 - Dependency Injection
 - New provideHttpClient()
 - ...
- **RxJS**
 - Goals, structure & 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
- ...

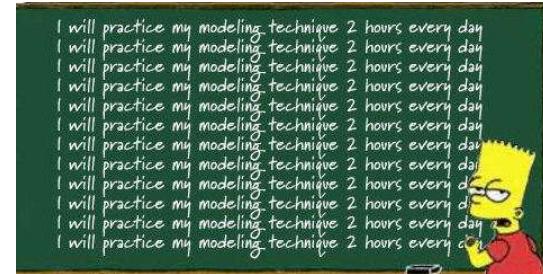
Overall process



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

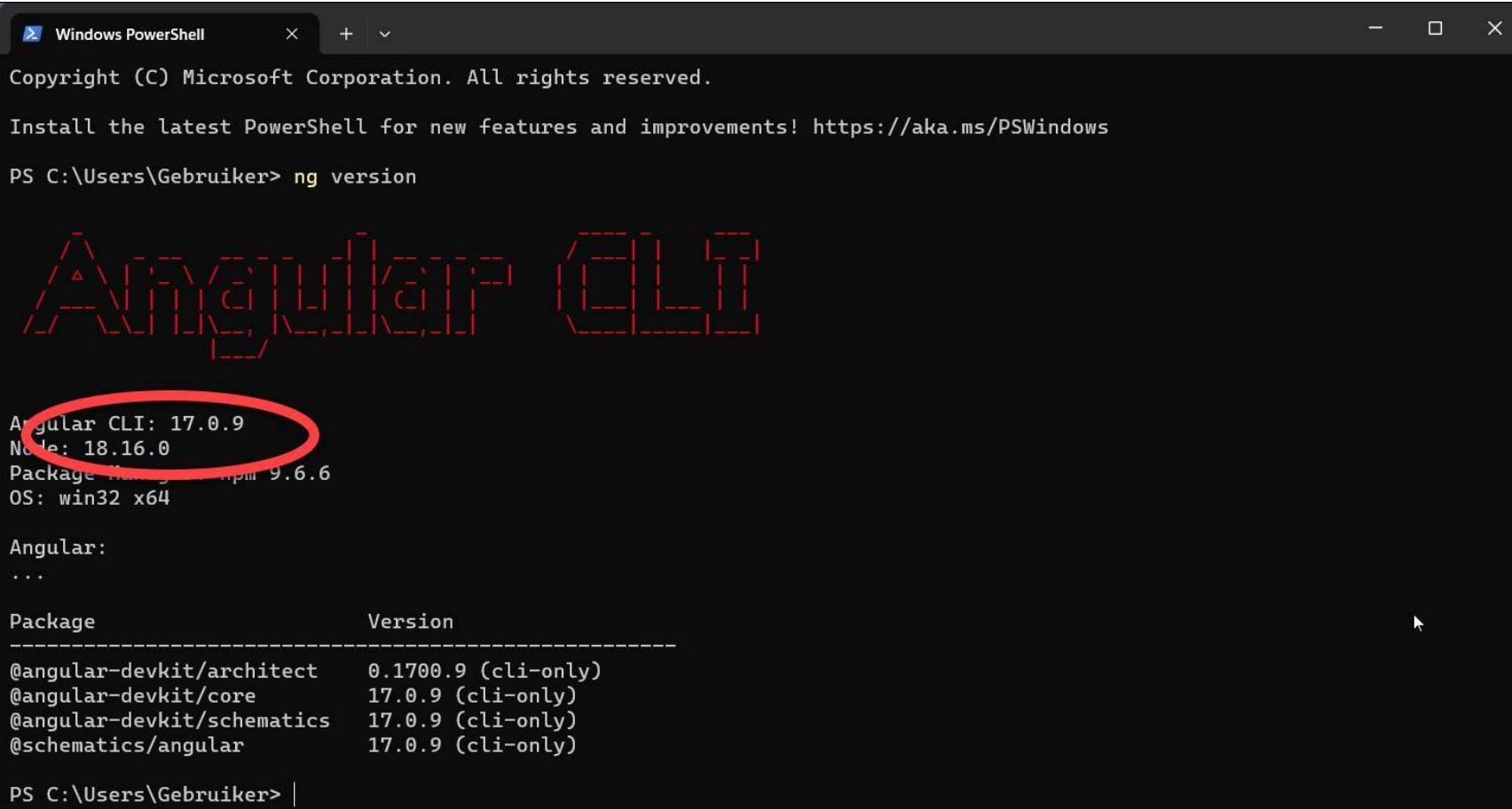


- Command Line Interface for
 - Scaffolding (`ng new`),
 - Developing (`ng generate`)
 - Testing (`ng test`)
 - Deploying applications (`ng deploy`)
- Local installation
 - SO: no online environment like Stackblitz or CodeSandbox

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

The screenshot shows the Angular CLI documentation page. At the top left is the Angular logo and the text "BETA DOCS". To its right is a sidebar with navigation links: "v17", "Ctrl K", "Docs", "Tutorials", "Playground", "Reference", and three dots at the bottom. The main content area has a breadcrumb navigation bar with "Developer Tools > Angular CLI" and a red circle highlighting it. Below this is the title "The Angular CLI". A paragraph explains what the Angular CLI is: "The Angular CLI is a command-line interface tool which allows you to scaffold, develop, test, deploy, and maintain Angular applications directly from a command shell." It also mentions that the Angular CLI is published on npm as the `@angular/cli` package and includes a binary named `ng`. A note says "Commands invoking `ng` are using the Angular CLI." Below this is a section titled "Try Angular without local setup" with a checked checkbox. A paragraph describes this as a standalone tutorial for new users, mentioning `Try it now!`, `StackBlitz`, and the lack of local setup required. The main content area is divided into four sections: "Getting Started" (with a "Get Started" button), "Command Reference" (with a "Learn More" button), "Schematics" (with a description of generating and modifying source files), and "Builders" (with a description of performing complex transformations). There is also a small edit icon in the top right corner.

<https://angular.dev/tools/cli>



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Gebruiker> ng version

Angular CLI: 17.0.9
Node: 18.16.0
Package Manager: npm 9.6.6
OS: win32 x64

Angular:
...
Package          Version
-----
@angular-devkit/architect    0.1700.9 (cli-only)
@angular-devkit/core         17.0.9 (cli-only)
@angular-devkit/schematics   17.0.9 (cli-only)
@schematics/angular          17.0.9 (cli-only)

PS C:\Users\Gebruiker> |
```

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 capabilities



ESBuild – extremely fast builder

» esbuild

Try in the browser

Getting Started

- Install esbuild
- Your first bundle
- Build scripts
- Bundling for the browser
- Bundling for node
- Simultaneous platforms
- Using Yarn Plug'n'Play
- Other ways to install

API

- Overview
- General options
- Input
- Output contents
- Output location
- Path resolution
- Transformation
- Optimization
- Source maps
- Build metadata
- Logging

Content Types

A red arrow points to the esbuild entry in the chart.

Tool	Time (s)
esbuild	0.39s
parcel 2	14.91s
rollup 4 + terser	34.10s
webpack 5	41.21s

An extremely fast bundler for the web

Above: the time to do a production bundle of 10 copies of the [three.js](#) library from scratch using default settings, including minification and source maps. More info [here](#).

Our current build tools for the web are 10-100x slower than they could be. The main goal of the esbuild bundler project is to bring about a new era of build tool performance, and create an easy-to-use modern bundler along the way.

Major features:

- Extreme speed without needing a cache

<https://esbuild.github.io/>

Vite.js – modern bundler



The screenshot shows the official Vite.js website. At the top, there's a navigation bar with links for Guide, Config, Plugins, Resources, Version, and a dark mode toggle. Below the header is a large, colorful graphic featuring a yellow lightning bolt inside a purple downward-pointing triangle against a blue-to-pink gradient background. To the left of the graphic, the word "Vite" is written in its signature purple font, followed by the text "Next Generation Frontend Tooling" in a large, bold, dark font. Below this, a subtext reads "Get ready for a development environment that can finally catch up with you." At the bottom of the main section are four buttons: "Get Started" (blue), "Why Vite?" (light gray), "View on GitHub" (light gray), and "ViteConf 23!" (blue). The page also features three callout boxes at the bottom: "Instant Server Start" (with a lightbulb icon), "Lightning Fast HMR" (with a lightning bolt icon), and "Rich Features" (with a wrench and screwdriver icon).

Vite

Next Generation Frontend Tooling

Get ready for a development environment that can finally catch up with you.

[Get Started](#) [Why Vite?](#) [View on GitHub](#) [ViteConf 23!](#)

Instant Server Start
On demand file serving over native ESM, no bundling required!

Lightning Fast HMR
Hot Module Replacement (HMR) that stays fast regardless of app size.

Rich Features
Out-of-the-box support for TypeScript, JSX, CSS and more.

<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 ..\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)
```





CLI reference

BETA DOCS v17 Ctrl K

Docs Tutorials Playground Reference ... ⚙️

← CLI Reference

- Overview
- ng add
- ng analytics
- ng build
- ng cache
- ng completion
- ng config
- ng deploy
- ng doc
- ng e2e
- ng extract-i18n
- ng generate
- ng lint
- ng new
- ng run
- ng serve
- ng test
- ng update
- ng version

CLI Reference

CLI Reference

Command	Alias	Description
add		Adds support for an external library to your project.
analytics		Configures the gathering of Angular CLI usage metrics.
build	b	Compiles an Angular application or library into an output directory named dist/ at the given output path.
cache		Configure persistent disk cache and retrieve cache statistics.
completion		Set up Angular CLI autocompletion for your terminal.
config		Retrieves or sets Angular configuration values in the angular.json file for the workspace.
deploy		Invokes the deploy builder for a specified project or for the default project in the workspace.
doc	d	Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.
e2e	e	Builds and serves an Angular application, then runs end-to-end tests.
extract-i18n		Extracts i18n messages from source code.
generate	g	Generates and/or modifies files based on a schematic.
lint		Runs linting tools on Angular application code in a given project folder.

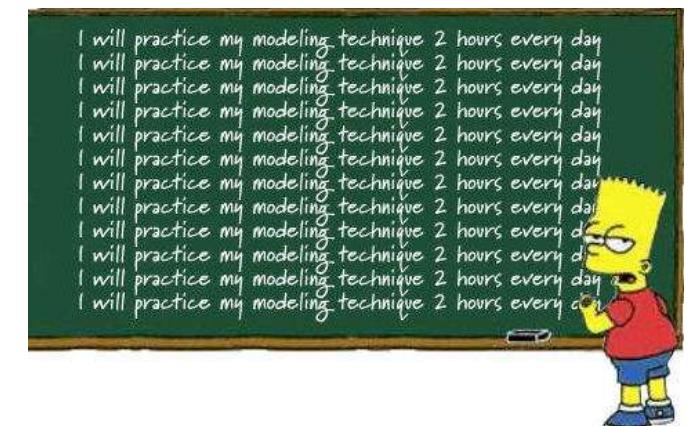
<https://angular.dev/cli>

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

The screenshot shows a web browser window with the URL `localhost:4200` in the address bar. The main content of the page is a large 'Hello, angular17' message. Above this message is the Angular logo. Below the main message is a smaller text: 'Congratulations! Your app is running.' followed by a small icon. To the right of the main content area, there is a sidebar with several links: 'Explore the Docs', 'Learn with Tutorials', 'CLI Docs', 'Angular Language Service', and 'Angular DevTools'. At the bottom of the sidebar are icons for GitHub, Twitter, and YouTube.





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.

The screenshot shows the official Node.js website at <https://nodejs.org/en>. The page features a green hexagonal background pattern. At the top, there's a navigation bar with links for Learn, About, Download, Blog, Docs, Contribute, Certification, and a search bar. The main headline reads "Run JavaScript Everywhere". Below it, a sub-headline states: "Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts." A prominent green button labeled "Download Node.js (LTS)" is visible. On the right side, there's a code editor window titled "Create an HTTP Server" with some sample code:

```
1 // server.mjs
2 import { createServer } from 'http';
3
4 const server = createServer((req, res) => {
5   res.writeHead(200, { 'Content-Type': 'text/plain' });
6   res.end('Hello World!');
7 });
8
9 // starts a simple http server
10 server.listen(3000, '127.0.0.1');
11 console.log('Listening on port 3000');
12
13 // run with `node server.mjs`
```

Below the code editor, the word "JavaScript" is displayed.



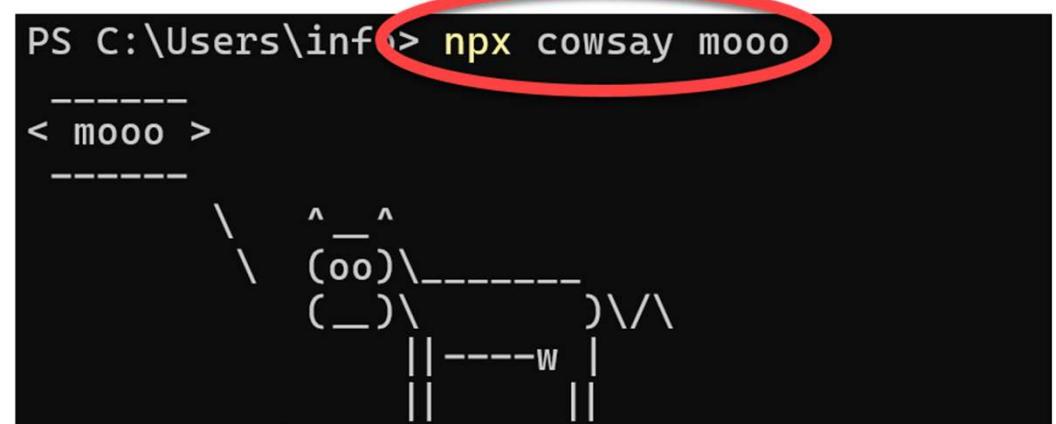
When to use npm?

- 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



```
PS C:\Users\info> npx cowsay moo
-----
< moo >
-----
      \  ^__^
       (oo)\_____
         (__)\       )\/\
             ||----w |
             ||     ||
```



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

```
// 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)
  ]
};
```

```
// 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

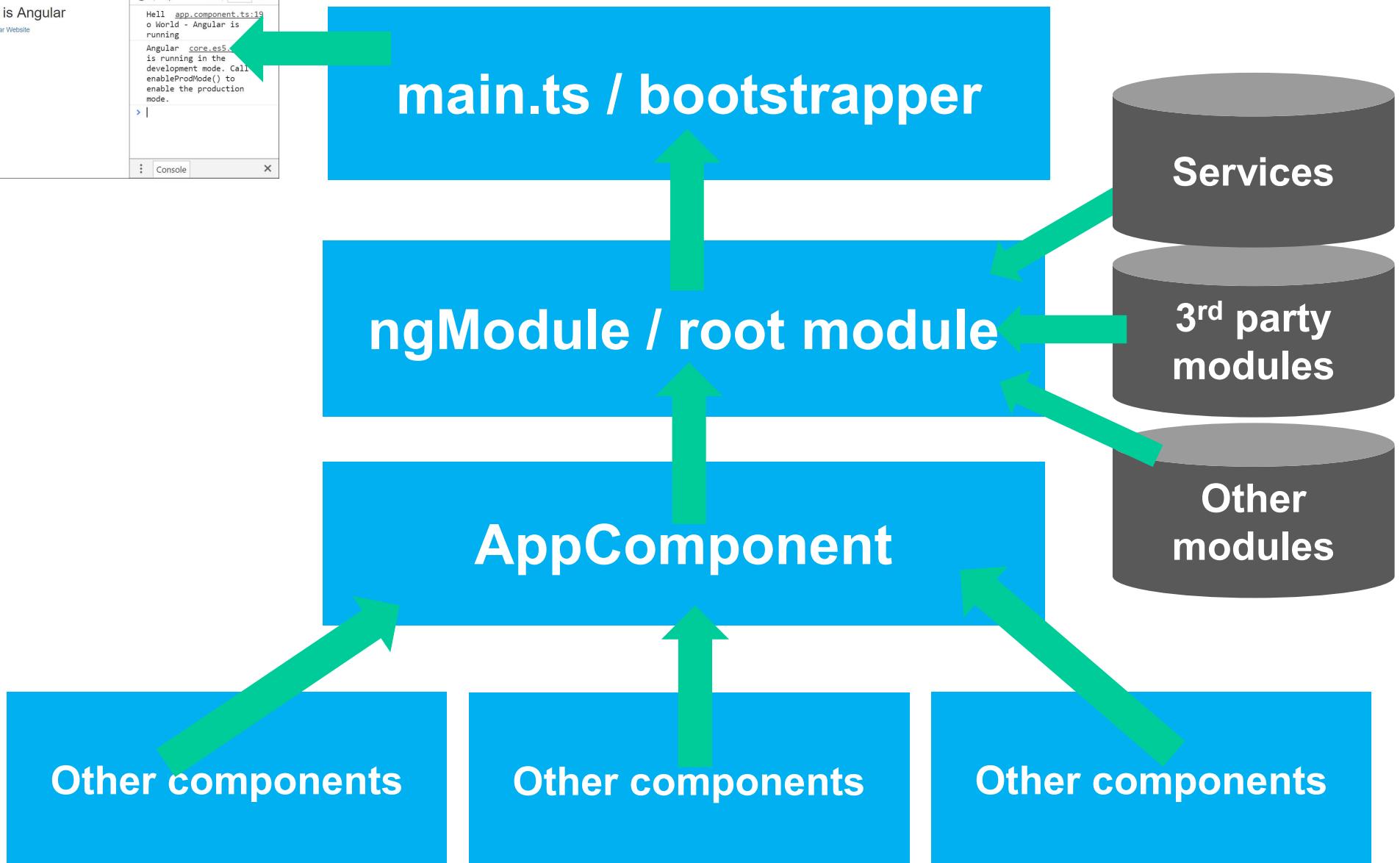
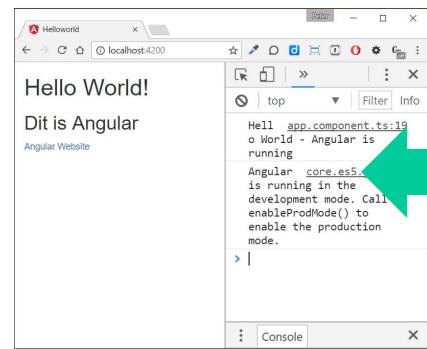
Towards an `NgModule`-less future



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





What are standalone components?

Regular Angular Components with the `standalone : true` option

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

@Component({
  selector: 'app-hello',
  standalone: true,           ← Red arrow pointing here
  imports: [],
  templateUrl: './hello.component.html',
  styleUrls: ['./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',
  styleUrls: ['./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',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

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



Hello, angular-v17

Congratulations! Your app is running.

This is hello component!



[Explore the Docs](#)

[Learn with Tutorials](#)

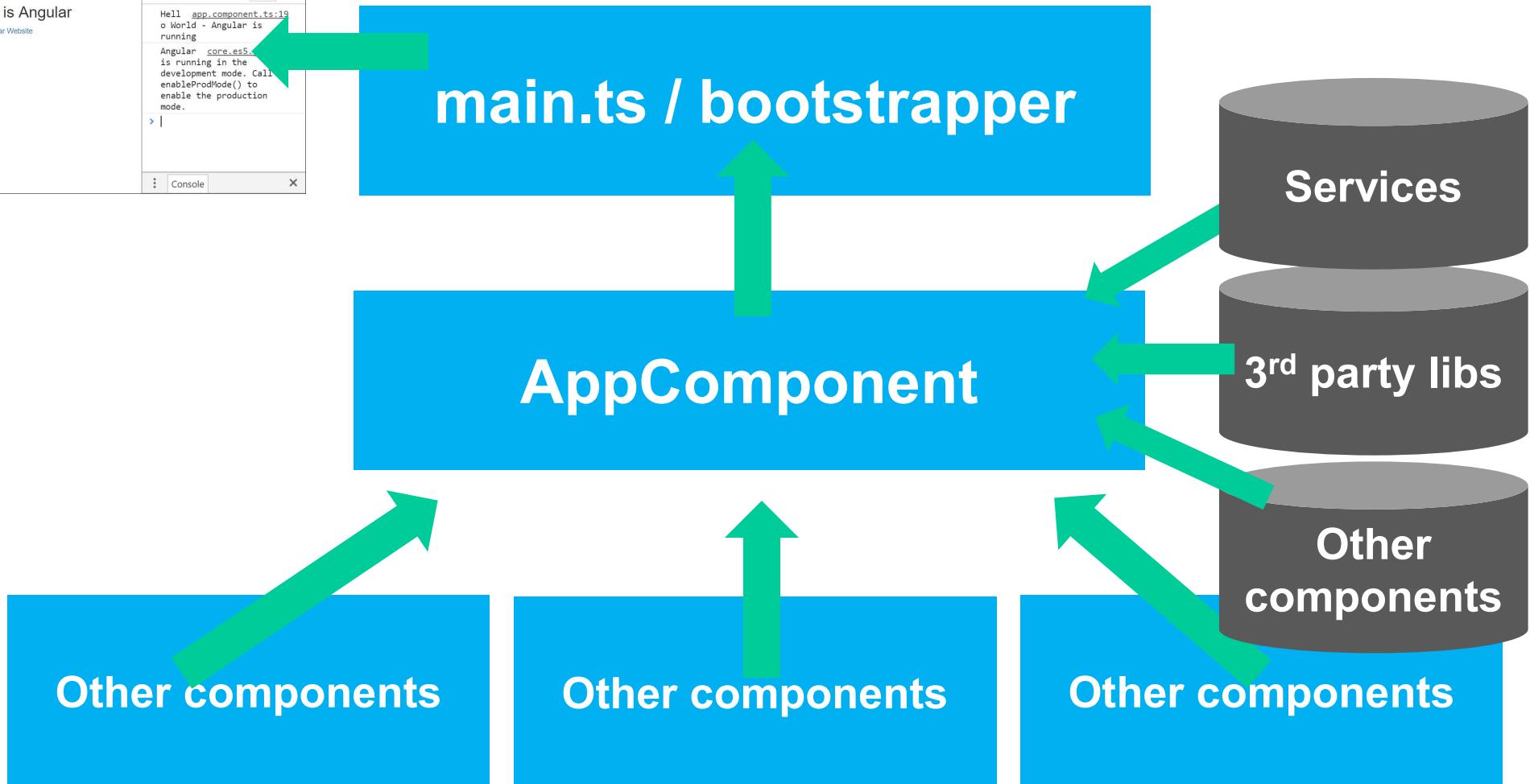
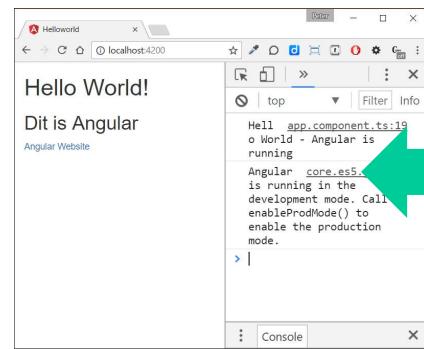
[CLI Docs](#)

[Angular Language Service](#)

[Angular DevTools](#)



Structure: modern Angular apps



(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
```

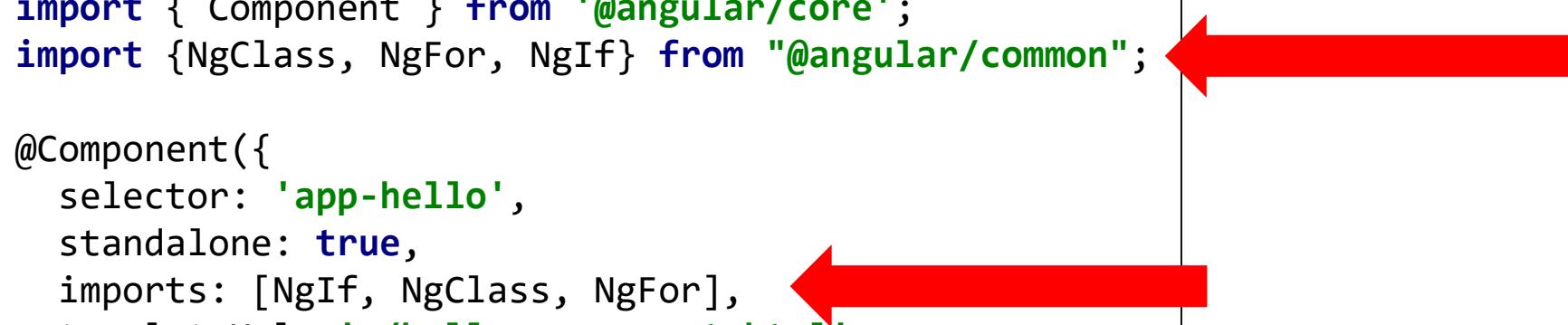
1 usages • PeterKoenig



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',
  styleUrls: ['./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',
  styleUrls: ['./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',
  styleUrls: ['./app.component.css'
})
```



Combined NgModules/Standalone



 Angular

Hello, angular-v17

Congratulations! Your app is running.

customer component

customer details...

Explore

Learn wi

CLI Docs

Angular

Angular





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',  
  styleUrls: ['./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



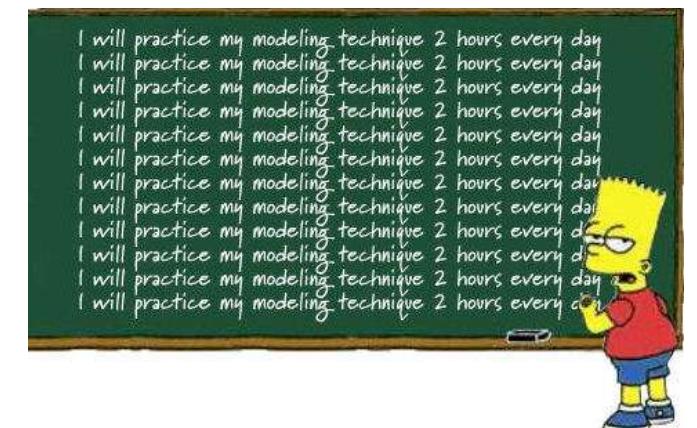
- No more (e.g. a lot less) boilerplate in configuring routing
- Use the **Angular standalone API's** to bootstrap the application instead of using NgModules.
 - `bootstrapApplication (topLevelComponent, configOptions)`

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)`

The screenshot shows a web-based Angular tutorial interface. On the left, there's a sidebar with a tree view: 'Learn Angular' is expanded, showing 'Control flow - @for'. To the right of the sidebar are navigation buttons ('<' and '>') and a code editor area with a '+' button and a placeholder 'Type your code here'. Below the sidebar, the main content area has a title 'Control Flow in Components - **@for**' with a pencil icon. The main text in the content area reads: 'Often when building web applications, you need to repeat some code a specific number of times - for example, given an array of names, you may want to display each name in a `</>` tag'. A small portion of the text at the bottom is obscured by a red redaction box.

Unchanged in Angular 17+



- Property binding
 - ``
- 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>

[← Templates](#)

In-depth Guides > Templates

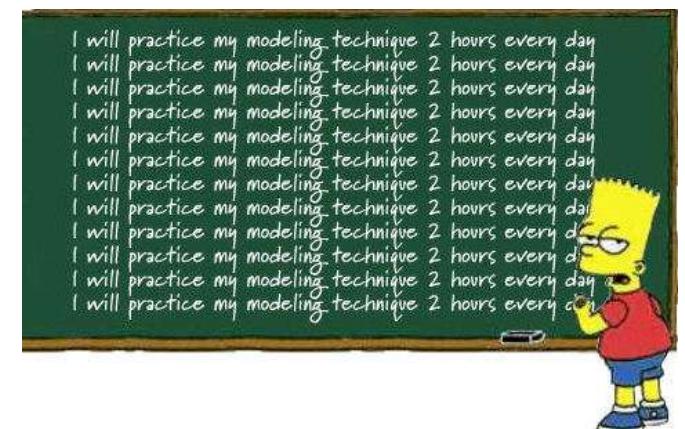
Control flow

Angular templates support control flow blocks that let you conditionally show, hide, and repeat elements.

NOTE: This was previously accomplished with the `*ngIf`, `*ngFor`, and `*ngSwitch` directives.

On this page

- Conditionally display content with `@if`, `@else-if` and `@else`
- Referencing the conditional expression's result
- Repeat content with the `@for` block





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 your 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:

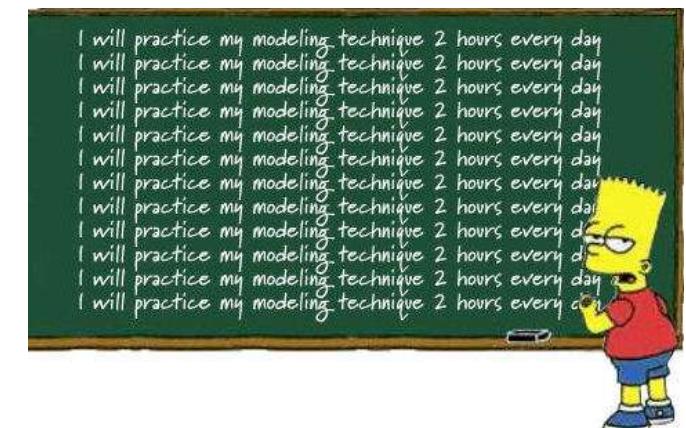
The screenshot shows a web page with the Angular logo at the top left. The main heading is "Hello, haarlem-app". Below it, a bold text says "customer-list works!". A section titled "Our customers:" lists three entries:

- 1 - Google**
info@google.com
- 2 - Microsoft**
hello@microsoft.com
- 3 - Facebook**
insta@facebook.com

On the right side of the page, there is a sidebar with several links:

- Explore the Docs
- Learn with Tutorials
- CLI Docs
- Angular Language Services
- Angular DevTools

Below these links are icons for GitHub, X (Twitter), and YouTube.





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	<code>constructor</code>	Standard JavaScript class constructor 🔗 . Runs when Angular instantiates the component.
Change	<code>ngOnInit</code>	Runs once after Angular has initialized all the component's inputs.
Detection	<code>ngOnChanges</code>	Runs every time the component's inputs have changed.
	<code>ngDoCheck</code>	Runs every time this component is checked for changes.
	<code>ngAfterContentInit</code>	Runs once after the component's <i>content</i> has been initialized.
	<code>ngAfterContentChecked</code>	Runs every time this component content has been checked for changes.
	<code>ngAfterViewInit</code>	Runs once after the component's <i>view</i> has been initialized.
	<code>ngAfterViewChecked</code>	Runs every time the component's view has been checked for changes.
Rendering	<code>afterNextRender</code>	Runs once the next time that all components have been rendered to the DOM.
	<code>afterRender</code>	Runs every time all components have been rendered to the DOM.
Destruction	<code>ngOnDestroy</code>	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<string, SimpleChange>` 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.

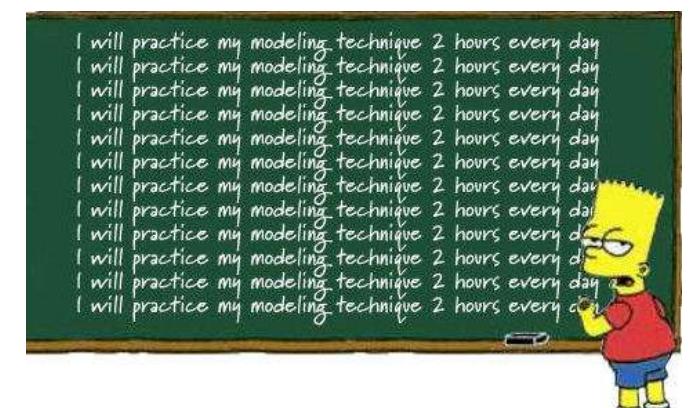
```
@Component({
  ...
})
export class UserProfile {
  @Input() name: string = '';

  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



Time-based Releases

@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 interchangeable.
 - 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

```
// 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



v4.8.2 ▾ Run Export ▾ Share → .JS .D.TS Errors Logs Plugins

```
1  type Person = {
2    name: string;
3    age: number;
4  }
5
6  // Same properties, now in an interface.
7  interface IPerson {
8    name: string;
9    age: number;
10 }
11
12 // some variables, based on that type/interface
13 const person1: Person = {
14   name: 'Peter',
15   age: 10
16 }
17
18 const person2: IPerson = {
19   name: 'Sandra',
20   age: 20
21 }
```

```
"use strict";
// some variables, based on that type/interface
const person1 = {
  name: 'Peter',
  age: 10
};
const person2 = {
  name: 'Sandra',
  age: 20
};
console.log('I\'m a person with a type: ', person1);
console.log('I\'m a person with an interface: ', person2)
console.log('There is no difference....');
```



Works as expected

```
// some variables, based on that type/interface
const person1: Person = {
  name: 'Peter',
  age: '10'
}
TS2322: Type 'string' is not assignable to type 'number'.
40-types-vs-interface.ts(12, 9): The expected type comes from property 'age' which is declared here on type 'Person'
  Suppress with @ts-ignore Alt+Shift+Enter More actions... Alt+Enter
```

```
// some variables, based on that type
const person1: Person = {
  name: 'Peter',
  ag2e: 10
}
```

TypeScript is clever. Gives you some tips.

```
TS2322: Type '{ name: string; ag2e: number; }' is not assignable to type 'Person'.
Object literal may only specify known properties, but 'ag2e' does not exist in type 'Person'. Did you mean to write 'age'?
```

Suppress with @ts-ignore Alt+Shift+Enter More actions... Alt+Enter



Classes can implement both

// 1d. Classes can implement both a type and an interface. A

```
class Peter implements Person{  
    name: string;  
    age: number;  
}
```

```
class Sandra implements IPerson{  
    name: string;  
    age: number;  
}
```

```
const peter = new Peter();
```

```
peter.|
```

con	f	name	string
rfaces	f	age	number
		FunctionCall(expn)	

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;  
}
```

A screenshot of a code editor showing a TypeScript error. The code defines two separate `IPerson` interfaces and then creates a variable `harry` of type `IPerson`. The variable is initialized with an object that has properties `name` and `age`, but lacks the required `city` property. A red arrow points from the error message to the `city` property in the declaration. The error message is: "TS2741: Property 'city' is missing in type '{ name: string; age: number; }' but required in type 'IPerson'." Below the error message, it says: "40-types-vs-interface.ts(74, 9): 'city' is declared here." There are also options to "Remove unused constant 'harry'" and "More actions...".

```
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
```

```
// A type is therefore also called a type alias .  
  
type Teacher = boolean;  
  
// 3b, now you can use the type as an alias  
const peter: Teacher = 'true'; // <== error, wrong type (s)  
  
} 

TS2322: Type 'string' is not assignable to type 'boolean'. ⋮



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



---



const types_vs_interfaces.peter: boolean



---



3b, now you can use the type as an alias


```

TS



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  
}
```

TS



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

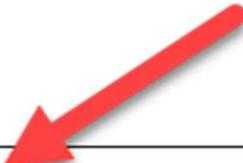
TS



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.

TS



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

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

```
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!");  
    }  
}
```

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



New provideHttpClient

In moduleless applications, 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



← HTTP Client

In-depth Guides > HTTP Client

On this page

- HTTP client service features
- What's next
- Back to the top

Overview

- Setting up HttpClient
- Making requests
- Intercepting requests and responses
- Testing

Understanding communicating with backend services using HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the `HttpClient` service class in `@angular/common/http`.

HTTP client service features

The HTTP client service offers the following major features:

- The ability to request [typed response values](#)
- Streamlined [error handling](#)
- Request and response [interception](#)
- Robust [testing utilities](#)

What's next

[Setting up HttpClient](#) [Making HTTP requests](#)

<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#providing-dependency-injection>

Hello, naariem-app

http-client works!

Name: Leanne Graham
Email: Sincere@april.biz

Name: Ervin Howell
Email: Shanna@melissa.tv

Name: Clementine Bauch
Email: Nathan@yesenia.net

Name: Patricia Lebsack

