

illionx



Global Knowledge.®

# Angular Advanced Monorepo's - introduction

Peter Kassenaar –  
[info@kassenaar.com](mailto:info@kassenaar.com)

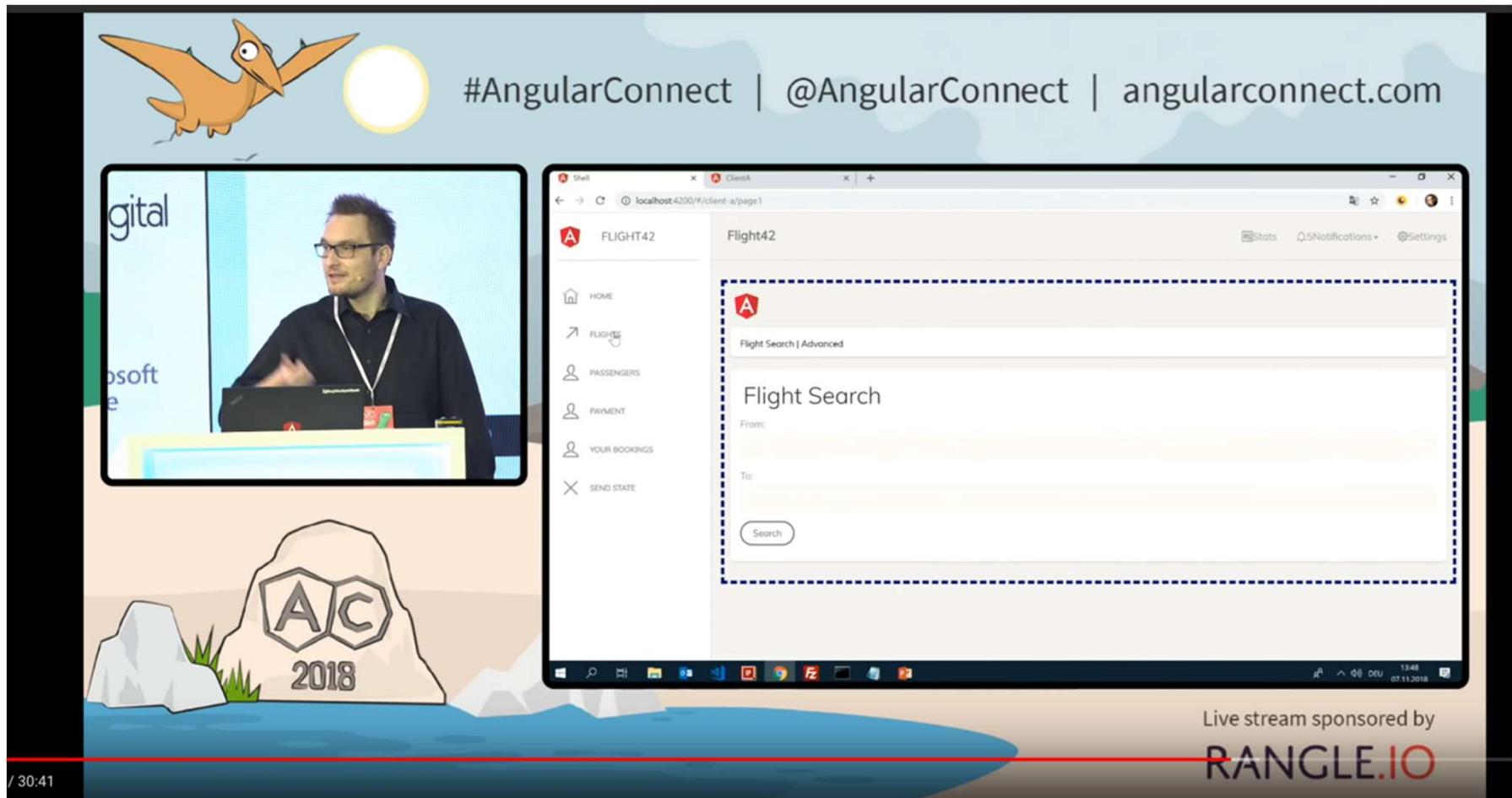
WORLDWIDE LOCATIONS

BELGIUM CANADA COLOMBIA DENMARK EGYPT FRANCE IRELAND JAPAN KOREA MALAYSIA MEXICO NETHERLANDS NORWAY QATAR  
SAUDI ARABIA SINGAPORE SPAIN SWEDEN UNITED ARAB EMIRATES UNITED KINGDOM UNITED STATES OF AMERICA

# Angular in the Enterprise

- When? With (really) big(ger) applications
- Multiple solutions, examples:
  - Monorepo approach:  
<https://github.com/PeterKassenaar/ng-monorepo>
  - Micro-app approach:  
<https://github.com/PeterKassenaar/ng-microfrontends>

# Manfred Steyer – Angular Connect



<https://www.youtube.com/watch?v=YU-fMRs-ZYU>

Code: <https://github.com/PeterKassenaar/angular-microapp>

# Enterprise applications – multiple options

- There are always *multiple solutions*
- There is **NOT** one solution that is 'the best'
- Options:
  1. **NPM packages**

Publish your own packages/libraries to npm, so others can npm install them
  2. **Monorepo**

Multiple projects in one code base, optionally sharing code
  3. **Micro-apps / micro-frontends**

Multiple applications, not sharing code, optionally different techniques/frameworks

# **So basically...**

**NPM  
packages**

**Monorepo**

**Micro apps**



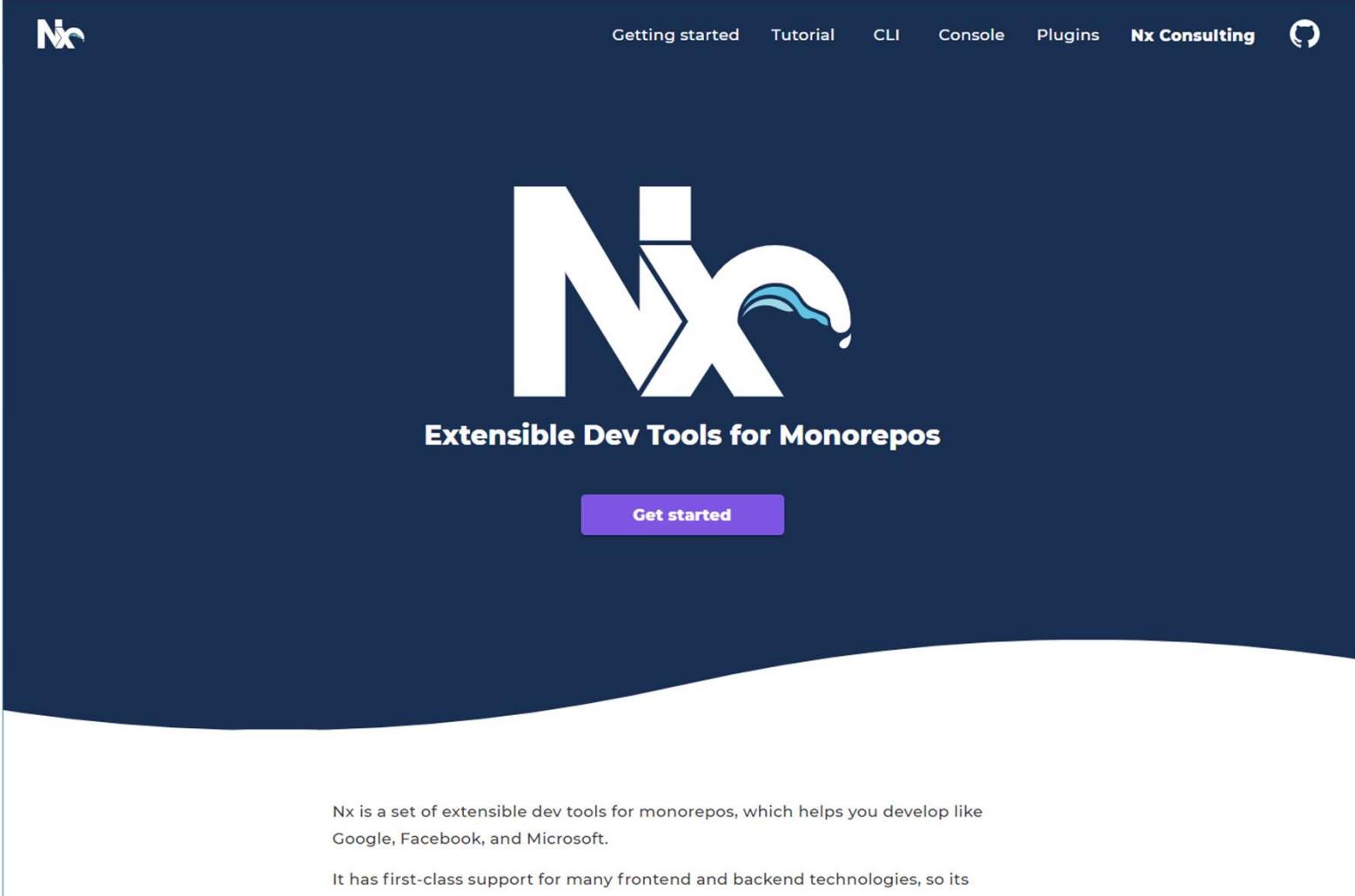
# Monorepos

Multiple projects in one solution, optionally sharing code

# What is a monorepo

- Often, bigger applications are not only split up into **modules**, but also in **other applications**
- Applications generally share a lot of code
  - Sharing components
  - Sharing services
  - Sharing pipes, other logic, etc...
- One (opinionated) solution – create a so called *monorepo*
- There are tools that also cover this
  - Nrwl Extensions – free, open source, <https://nrwl.io/nx>
  - Angular CLI as of V6.0.0+, <https://cli.angular.io/>

# Nrwl extensions to create a monorepo



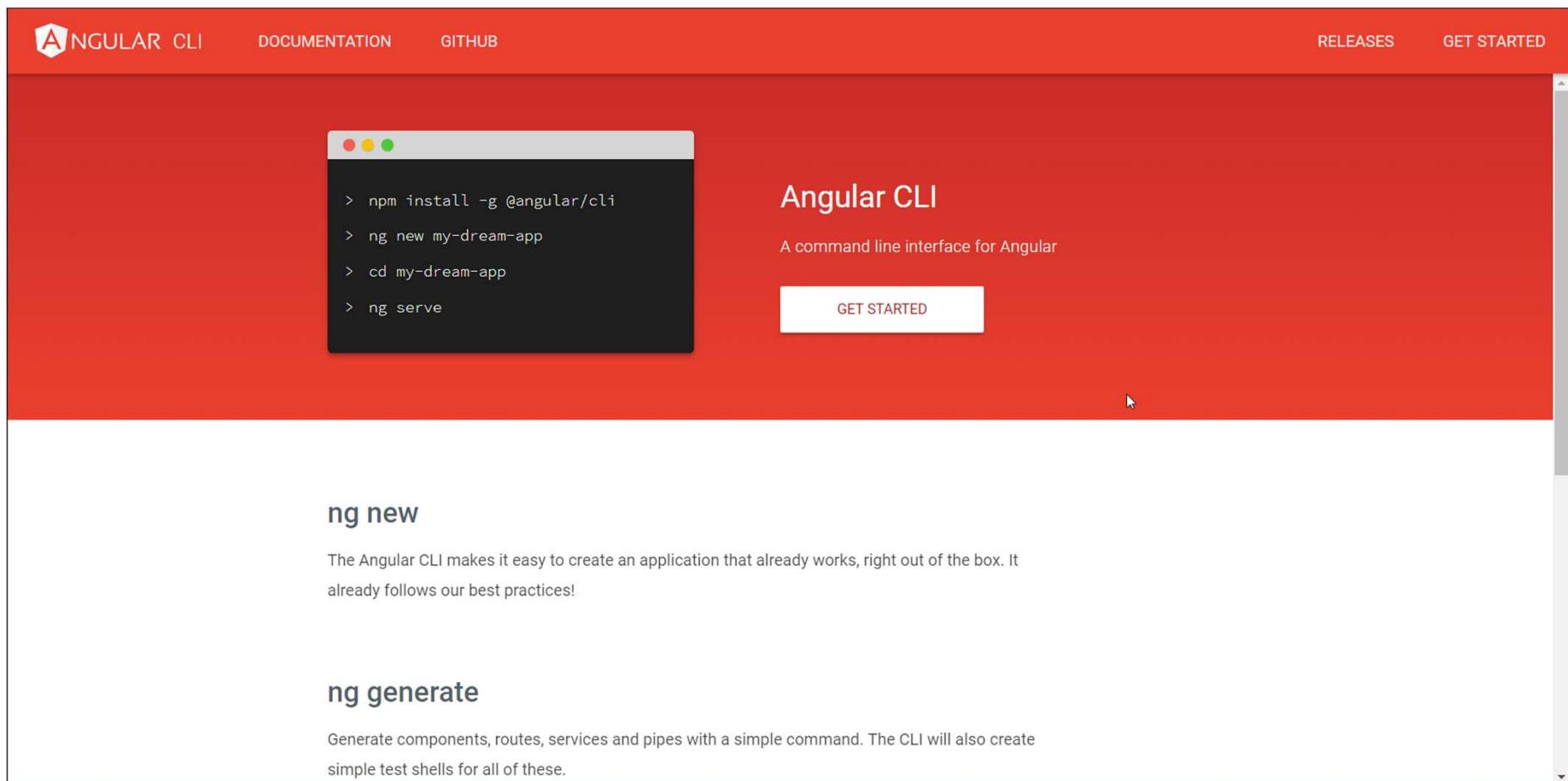
The screenshot shows the Nx website homepage. The header features the Nx logo and navigation links for Getting started, Tutorial, CLI, Console, Plugins, Nx Consulting, and a GitHub icon. The main section has a dark blue background with the Nx logo and the text "Extensible Dev Tools for Monorepos". A purple "Get started" button is centered. Below the main section, a white sidebar contains text about Nx being a set of extensible dev tools for monorepos, supporting Google, Facebook, and Microsoft, and having first-class support for many technologies. At the bottom, there is a red link to <https://nx.dev/>.

Nx is a set of extensible dev tools for monorepos, which helps you develop like Google, Facebook, and Microsoft.

It has first-class support for many frontend and backend technologies, so its documentation comes in multiple flavours.

<https://nx.dev/>

# Angular CLI – as of V6.0+



The screenshot shows the Angular CLI homepage with a red header. The header contains the Angular CLI logo, navigation links for 'DOCUMENTATION' and 'GITHUB', and buttons for 'RELEASES' and 'GET STARTED'. Below the header, there's a large callout box with a terminal window showing command-line steps to set up a new Angular application. To the right of the callout, the text 'Angular CLI' and 'A command line interface for Angular' is displayed, along with a 'GET STARTED' button. The main content area features sections for 'ng new' and 'ng generate'.

ANGULAR CLI DOCUMENTATION GITHUB RELEASES GET STARTED

> npm install -g @angular/cli  
> ng new my-dream-app  
> cd my-dream-app  
> ng serve

## Angular CLI

A command line interface for Angular

[GET STARTED](#)

### ng new

The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows our best practices!

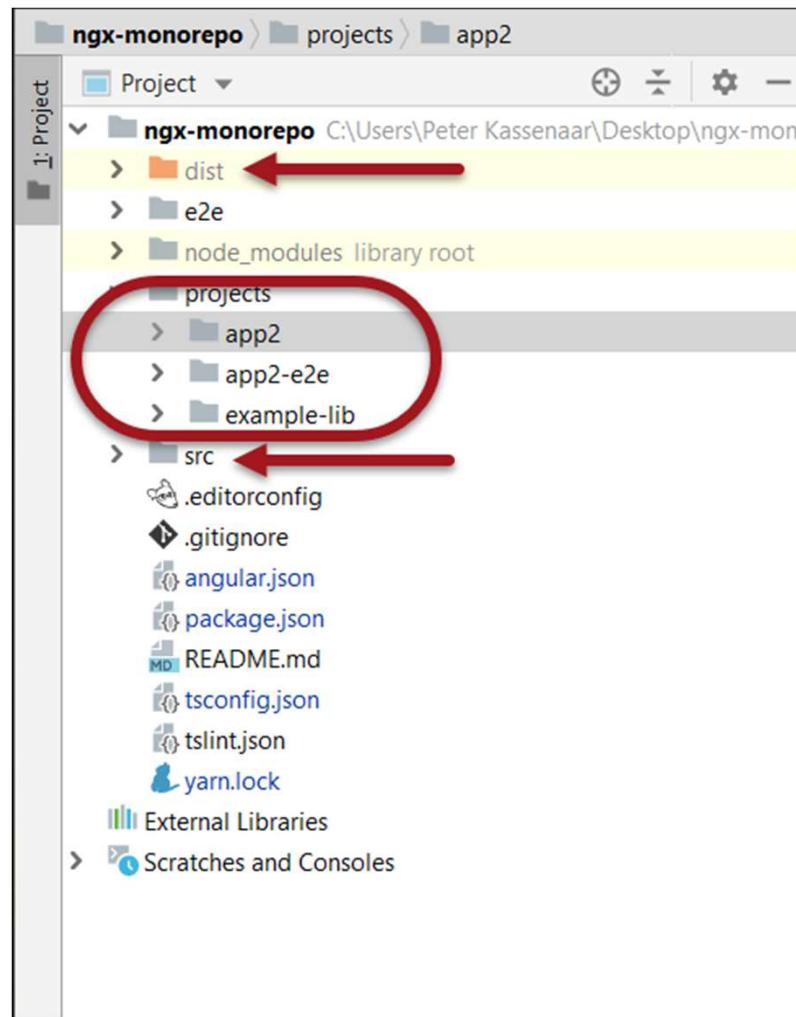
### ng generate

Generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these.

# Inside an Angular monorepo

- One `/node_modules`
- One main `package.json`
- `angular.json`, describing all the projects in the workspace
- Optional - One root app, in `/src` folder
- One `/projects` folder, containing:
  - A folder for every project
  - Libraries
  - Applications
  - This folder is created upon the first creation of a library or application
- `/dist` folder, holding the compiled Angular Packages that needs to be shared

# Structure/architecture



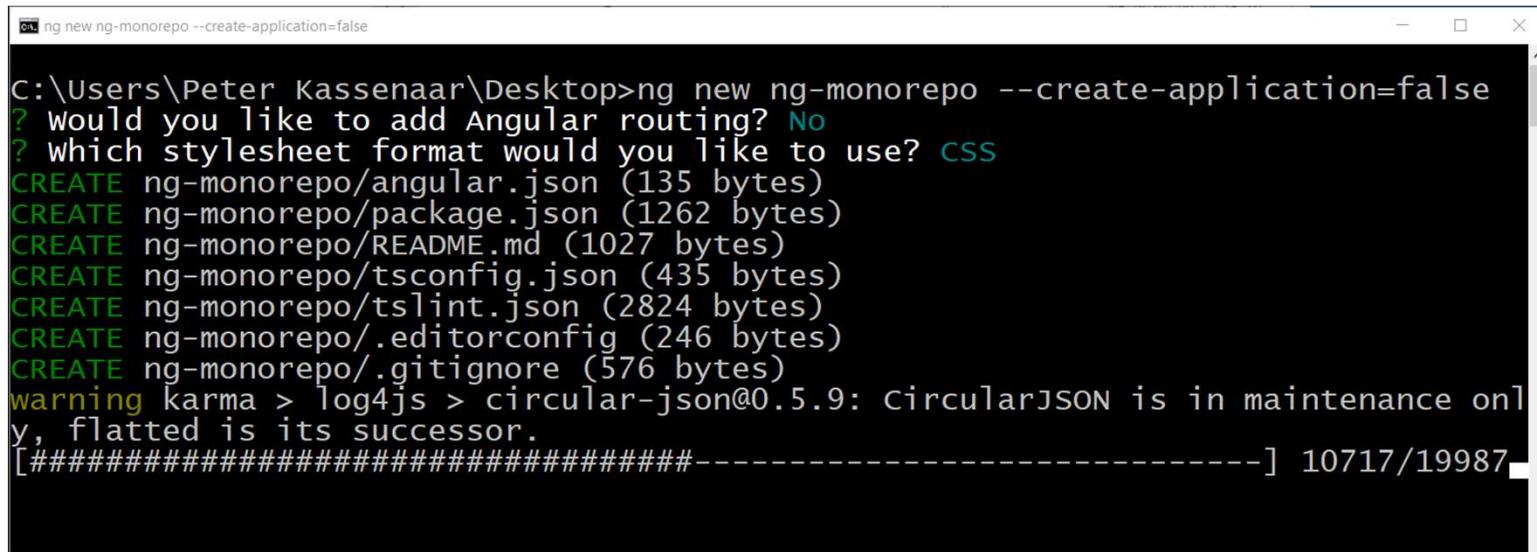
# Credits: Angular in Depth

The screenshot shows a blog post on the Angular In Depth website. The header features a red navigation bar with links for HOME, ANGULAR, RXJS, NGRX, ABOUT, and SUPPORT US, along with a note about AG-GRID. The main title is "The Angular Library Series - Creating a Library with Angular CLI". Below the title is a subtitle: "Angular libraries just got a lot easier to create thanks to the combination of Angular CLI and ng-packagr.". The author's profile picture is shown, along with the author's name, Todd Palmer, and a "Follow" button. The post was published on May 28, 2018, and has a 11 min read time. The background of the page features a large, abstract, curved architectural image.

<https://blog.angularindepth.com/creating-a-library-in-angular-6-87799552e7e5>

# High level overview of the steps

1. Create the library project using `ng new <mono-repo-name>`
  - We call this folder a *workspace*
2. The main app is mostly used for documentation, or example usage of the (shared) libs
  - Don't want the main app? Use `--create-application=false` flag!
  - <https://blog.angularindepth.com/angular-workspace-no-application-for-you-4b451afcc2ba>



The screenshot shows a terminal window with the following command and its execution:

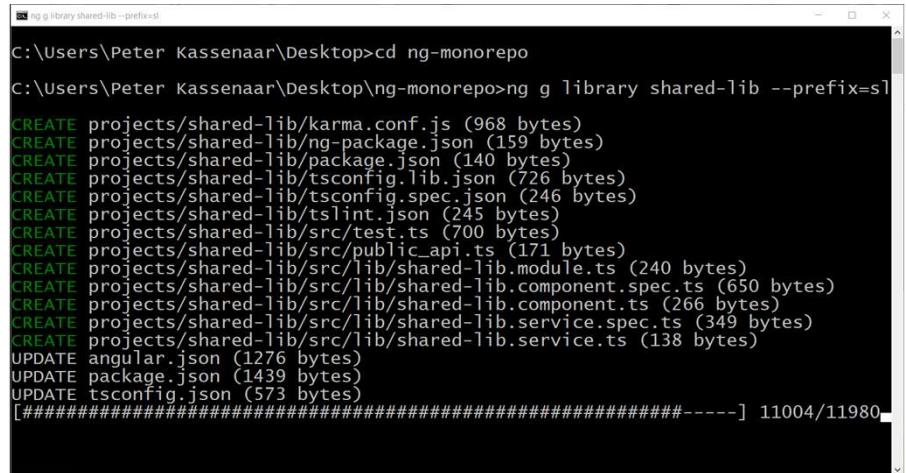
```
ng new ng-monorepo --create-application=false
```

Output:

```
C:\Users\Peter Kassenaar\Desktop>ng new ng-monorepo --create-application=false
? Would you like to add Angular routing? No
? which stylesheet format would you like to use? CSS
CREATE ng-monorepo/angular.json (135 bytes)
CREATE ng-monorepo/package.json (1262 bytes)
CREATE ng-monorepo/README.md (1027 bytes)
CREATE ng-monorepo/tsconfig.json (435 bytes)
CREATE ng-monorepo/tslint.json (2824 bytes)
CREATE ng-monorepo/.editorconfig (246 bytes)
CREATE ng-monorepo/.gitignore (576 bytes)
warning karma > log4js > circular-json@0.5.9: circularJSON is in maintenance only, flattened is its successor.
[########################################-----] 10717/19987
```

# Create (shared) lib

- Generate the (shared) library
  - cd ng-monorepo
  - ng generate library shared-lib --prefix=sl (or some other prefix)
  - Angular also updates the global angular.json, package.json and tsconfig.json
- Always use custom prefixes on libraries and projects
  - distinguish components and services!



```
ng g library shared-lib --prefix=sl
C:\Users\Peter Kassenaar\Desktop>cd ng-monorepo
C:\Users\Peter Kassenaar\Desktop\ng-monorepo>ng g library shared-lib --prefix=sl
CREATE projects/shared-lib/karma.conf.js (968 bytes)
CREATE projects/shared-lib/ng-package.json (159 bytes)
CREATE projects/shared-lib/package.json (140 bytes)
CREATE projects/shared-lib/tsconfig.lib.json (726 bytes)
CREATE projects/shared-lib/tsconfig.spec.json (246 bytes)
CREATE projects/shared-lib/tslint.json (245 bytes)
CREATE projects/shared-lib/src/test.ts (700 bytes)
CREATE projects/shared-lib/src/public_api.ts (171 bytes)
CREATE projects/shared-lib/src/lib/shared-lib.module.ts (240 bytes)
CREATE projects/shared-lib/src/lib/shared-lib.component.spec.ts (650 bytes)
CREATE projects/shared-lib/src/lib/shared-lib.component.ts (266 bytes)
CREATE projects/shared-lib/src/lib/shared-lib.service.spec.ts (349 bytes)
CREATE projects/shared-lib/src/lib/shared-lib.service.ts (138 bytes)
UPDATE angular.json (1276 bytes)
UPDATE package.json (1439 bytes)
UPDATE tsconfig.json (573 bytes)
[########################################] 11004/11980
```

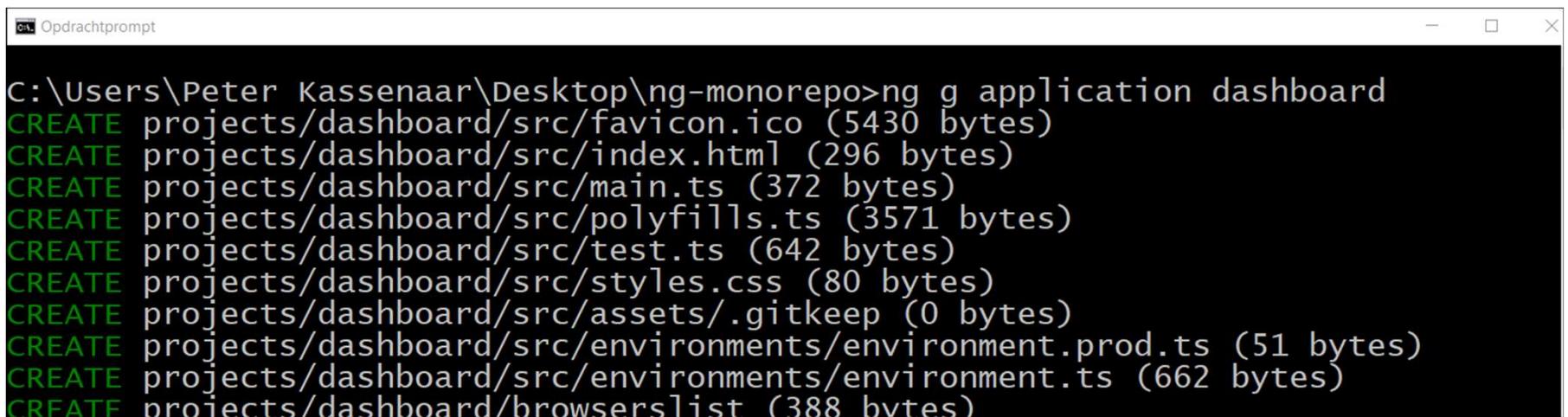
```
angular.json
1  {
2    "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
3    "version": 1,
4    "newProjectRoot": "projects",
5    "projects": {
6      "shared-lib": {"root": "projects/shared-lib"...}
41    },
42    "defaultProject": "shared-lib"
43 }
```

```
tsconfig.json
17  "lib": [
18    "es2018",
19    "dom"
20  ],
21  "paths": {
22    "shared-lib": [
23      "dist/shared-lib"
24    ],
25    "shared-lib/*": [
26      "dist/shared-lib/*"
27    ]
28  }
29 }
30 }
```

tsconfig.json – added paths, so we can import shared stuff easily later on

# Create first app in the monorepo

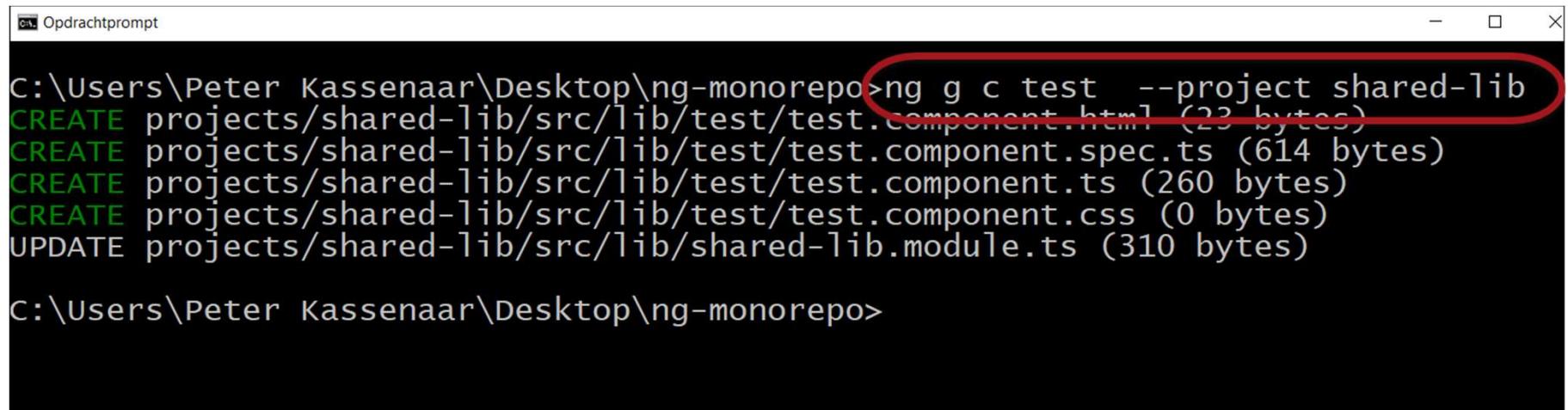
- Generate the (first) application
  - `ng generate application <application-name>`
  - Again, `angular.json` and `package.json` are updated with the new project



```
Opdrachtprompt
C:\Users\Peter Kassenaar\Desktop\ng-monorepo>ng g application dashboard
CREATE projects/dashboard/src/favicon.ico (5430 bytes)
CREATE projects/dashboard/src/index.html (296 bytes)
CREATE projects/dashboard/src/main.ts (372 bytes)
CREATE projects/dashboard/src/polyfills.ts (3571 bytes)
CREATE projects/dashboard/src/test.ts (642 bytes)
CREATE projects/dashboard/src/styles.css (80 bytes)
CREATE projects/dashboard/src/assets/.gitkeep (0 bytes)
CREATE projects/dashboard/src/environments/environment.prod.ts (51 bytes)
CREATE projects/dashboard/src/environments/environment.ts (662 bytes)
CREATE projects/dashboard/browserlist (388 bytes)
```

# Create a shared component

- Of course you can create multiple components
  - ng generate component <component-name> --project shared-lib
  - Use the --project flag to tell the CLI what project you want to add the component to
- Give the component some UI



A screenshot of a Windows Command Prompt window titled "Opdrachtprompt". The window shows the output of the command "ng g c test --project shared-lib". The command and its output are highlighted with a red oval. The output includes several "CREATE" entries for files like test.component.html, test.component.spec.ts, test.component.ts, and test.component.css, along with an "UPDATE" entry for shared-lib.module.ts.

```
c:\Users\Peter Kassenaar\Desktop\ng-monorepo>ng g c test --project shared-lib
CREATE projects/shared-lib/src/lib/test/test.component.html (23 bytes)
CREATE projects/shared-lib/src/lib/test/test.component.spec.ts (614 bytes)
CREATE projects/shared-lib/src/lib/test/test.component.ts (260 bytes)
CREATE projects/shared-lib/src/lib/test/test.component.css (0 bytes)
UPDATE projects/shared-lib/src/lib/shared-lib.module.ts (310 bytes)

c:\Users\Peter Kassenaar\Desktop\ng-monorepo>
```

# Export the component

- Add it to the exports array of the `shared-lib.module.ts`

```
exports: [..., TestComponent]
```

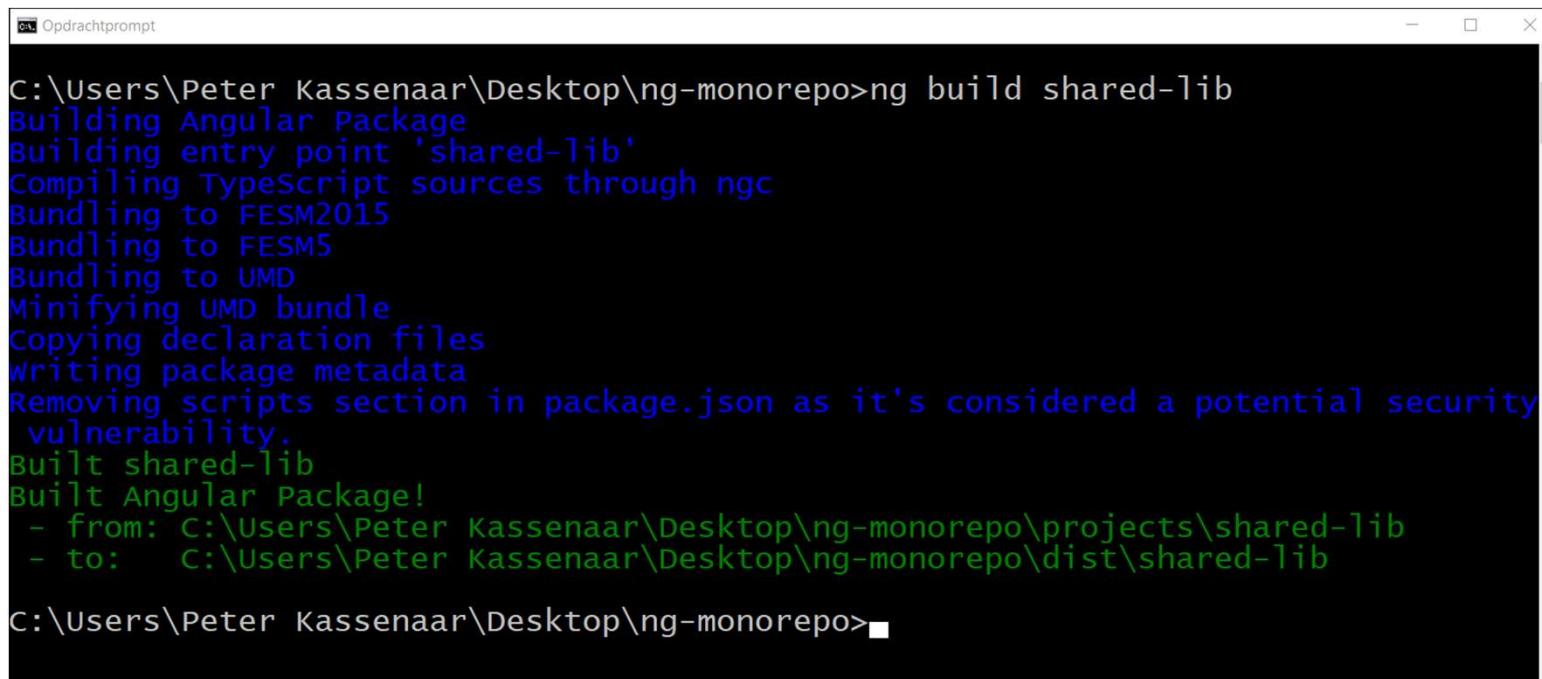
- Update the `public_api.ts` file to make the class available
  - Don't forget!

```
/*
 * Public API Surface of shared-lib
 */

export * from './lib/shared-lib.service';
export * from './lib/shared-lib.component';
export * from './lib/shared-lib.module';
export * from './lib/test/test.component';
```

# Build the library

- In order to use the component(s) from a shared library, it must be build
  - `ng build shared-lib`
  - A `\dist` folder is created
  - This folder is already mentioned in `tsconfig.json`. Verify this!



```
C:\Users\Peter Kassenaar\Desktop\ng-monorepo>ng build shared-lib
Building Angular Package
Building entry point 'shared-lib'
Compiling TypeScript sources through ngc
Bundling to FESM2015
Bundling to FESM5
Bundling to UMD
Minifying UMD bundle
Copying declaration files
Writing package metadata
Removing scripts section in package.json as it's considered a potential security
vulnerability.
Built shared-lib
Built Angular Package!
- from: C:\Users\Peter Kassenaar\Desktop\ng-monorepo\projects\shared-lib
- to:   C:\Users\Peter Kassenaar\Desktop\ng-monorepo\dist\shared-lib

C:\Users\Peter Kassenaar\Desktop\ng-monorepo>
```

# Using the library in the application

- Import the library module in the application `app.module.ts`
  - in our example: `dashboard.module.ts`
  - Remove the path your IDE might add to the AutoImport

```
import { AppComponent } from './app.component';
import {SharedLibModule} from '../../../../../shared-lib/src/lib/shared-lib.module';

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

X

```
import { AppComponent } from './app.component';
import {SharedLibModule} from 'shared-lib';

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

✓

# Using the shared component

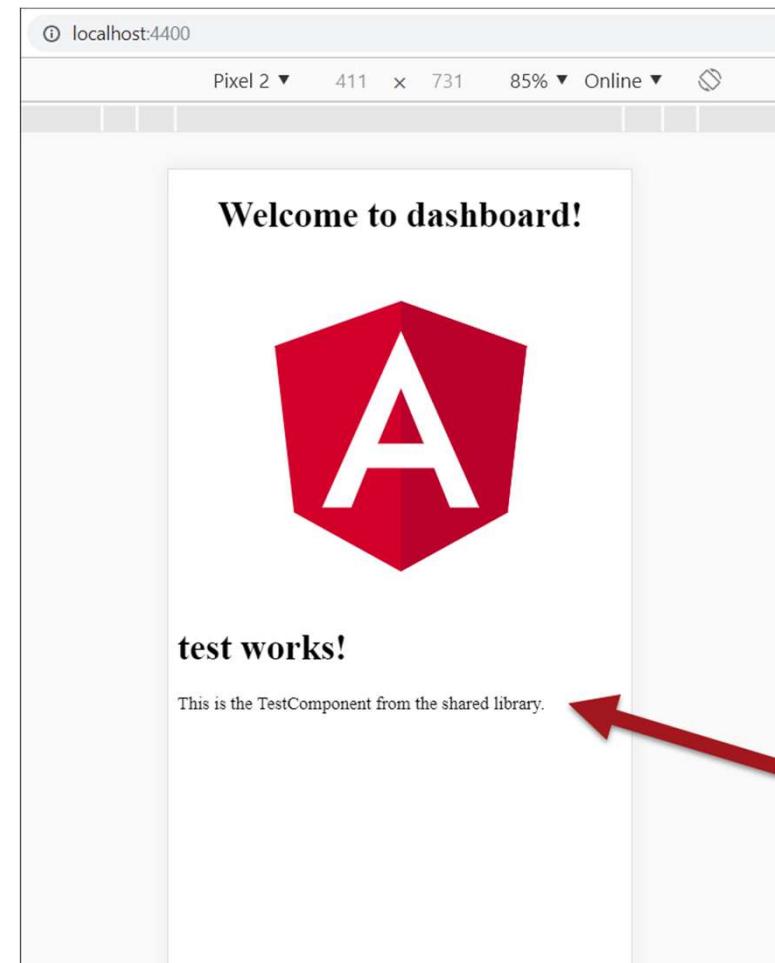
- Use the selector of the component from the shared library as normal

```
<div style="text-align:center">  
  ...  
</div>  
<sl-test></sl-test>
```

# Running the application

- Use the `ng serve` command
  - Use the `--project=<application-name>` to open the correct project
  - You can use additional flags as needed
- If you update your library contents, you need to rebuild it!
  - Write a script for that. For example

```
"scripts": {  
  ...  
  "build_lib": "ng build shared-lib"  
},
```



# Exporting a service

- Create a service the regular way
  - `ng generate service <service-name> --project=<project-name>`
- You need to make sure to export your service from the module
- But it only needs to be loaded once!
  - Use a `.forRoot()` method on the module



```
C:\Users\Peter Kassenaar\Desktop\ng-monorepo>ng g s shared/services/user --project shared-lib
CREATE projects/shared-lib/src/lib/shared/services/user.service.spec.ts (323 bytes)
CREATE projects/shared-lib/src/lib/shared/services/user.service.ts (133 bytes)
C:\Users\Peter Kassenaar\Desktop\ng-monorepo>
```

# Creating a .forRoot()

```
@NgModule({
  declarations: [SharedLibComponent, TestComponent],
  imports: [
  ],
  exports: [SharedLibComponent, TestComponent]
})
export class SharedLibModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: SharedLibModule,
      providers: [ UserService ]
    };
  }
}
```

- Remember to export the service from `public_api.ts`
- Remember to rebuild the library

# Using the shared service

- In your application, update the module to use `SharedLibModule.forRoot()`
- Inject the Service in the component where you want to use it
- Use the `servicemethods` as normal

```
@NgModule({
  ...
  imports: [
    SharedLibModule.forRoot()
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Component

```
import {Component, OnInit} from '@angular/core';
import {User, UserService} from 'shared-lib';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'dashboard';
  users: User[];

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.users = this.userService.getUsers();
  }
}
```



```
<hr>
<h2>Users from our shared service</h2>
{{ users | json }}
```



**test works!**

This is the TestComponent from the shared library.

### Users from our shared service

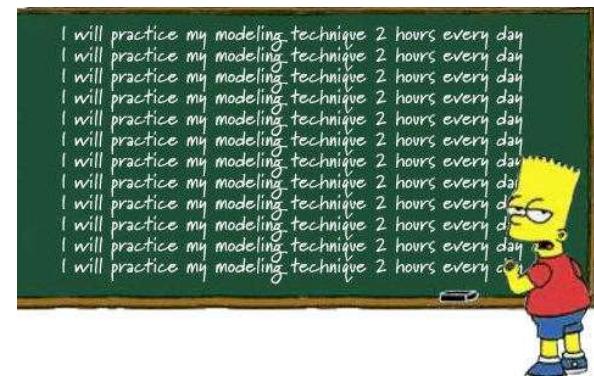
```
[ { "name": "Peter", "email": "test@test.com" }, { "name": "Sandra", "email": "sandra@sandra.com" } ]
```



# Workshop

- Create a new application inside the Monorepo
- Import the shared library
- Use/show the <sl-test> component from the shared lib inside your new project
- Build and run the new project
- Optional: create a new shared component in the lib.
  - Export / use the new component inside your project
  - Use the exported shared service from the library

Example: [github.com/PeterKassenaar/ng-monorepo](https://github.com/PeterKassenaar/ng-monorepo)



# Verdict on monorepo

- PRO:
  - Same version of Angular
  - Clear responsibilities per project/application
  - Share code and share services
- CON
  - Harder to use shared code / store
  - All the pro's, are actually also cons! (depending on your project)
  - Checkout: you always get the complete monorepo- unless working with *Github subtrees*

# Shorten TypeScript imports



PRO

Search

Lessons

## The Problem - Super Long Import Statements

Let's arbitrarily create a deeply nested component in Angular and a service to go along with it.

```
ng g component shared/deeply/nested/hello-world  
ng g service core/my
```

When you have a large Angular project with deeply nested files and folders it leads to import statements that look like this...

```
// import from component.ts  
import { MyService } from '../../../../../my.service';  
  
// import from service.ts  
import { HelloWorldComponent } from '../shared/deeply/nested/hello-world/hello-world.component';
```

That's might be fine occasionally, but it becomes very annoying and difficult to maintain for each new component you build. Furthermore moving a file will require every path to the new location.

## The Solution - TypeScript Config

Fortunately, we can configure TypeScript to make our files behave more like which should be the `src` directory in an Angular CLI project.

You can then point to any directory in your project and give it a custom name `@angular`, `@ngrx`, etc. The end result looks like...

```
// tsconfig.json in the root dir  
  
{  
  "compileOnSave": false,  
  "compilerOptions": {  
    // omitted...  
  
    "baseUrl": "src",  
    "paths": {  
      "@services/*": ["app/path/to/services/*"],  
      "@components/*": ["app/somewhere/deeply/nested/*"],  
      "@environments/*": ["environments/*"]  
    }  
  }  
}
```

You are receiving a free preview of 3 lessons

## Contents

[The Problem - Super Long Import Statements](#)

[The Solution - TypeScript Config](#)

[The End](#)

## Recent Posts

[Object Oriented Programming With TypeScript](#)

[Angular Elements Advanced Techniques](#)

[TypeScript - the Basics](#)

[The Real State of JavaScript 2018](#)

[Cloud Scheduler for Firebase Functions](#)

[Testing Firestore Security Rules With the Emulator](#)

[How to Use Git and Github](#)

[Infinite Virtual Scroll With the Angular CDK](#)

<https://angularfirebase.com/lessons/shorten-typescript-imports-in-an-angular-project/>



# Micro frontends

Having multiple, independent apps in your solution.  
Possibly using different techniques / frameworks

# **AKA Micro apps, Micro services, Portals, ...**

NPM  
packages

Monorepo

Micro apps

<https://github.com/PeterKassenaar/ng-microfrontends>

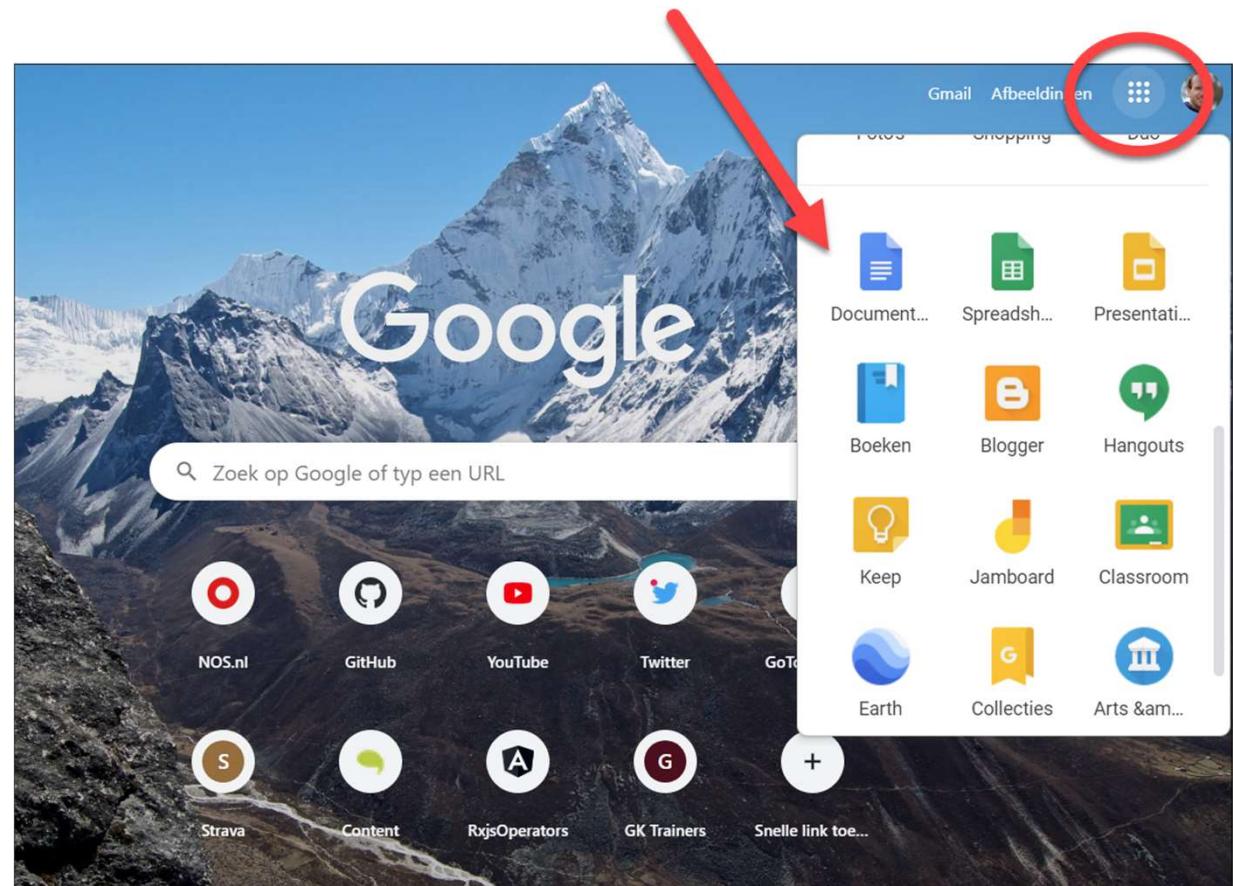
# Micro apps or micro frontend - When?

- When you **don't need a lot of interaction** between your apps
- When each app fulfills a **specific, standalone role**

Classical example:

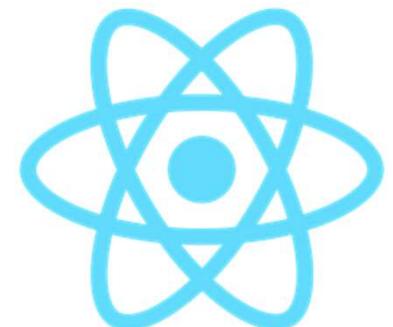
Google Suite or  
Microsoft Office.

Every application  
works on its own. No  
(or: barely) interaction  
needed.



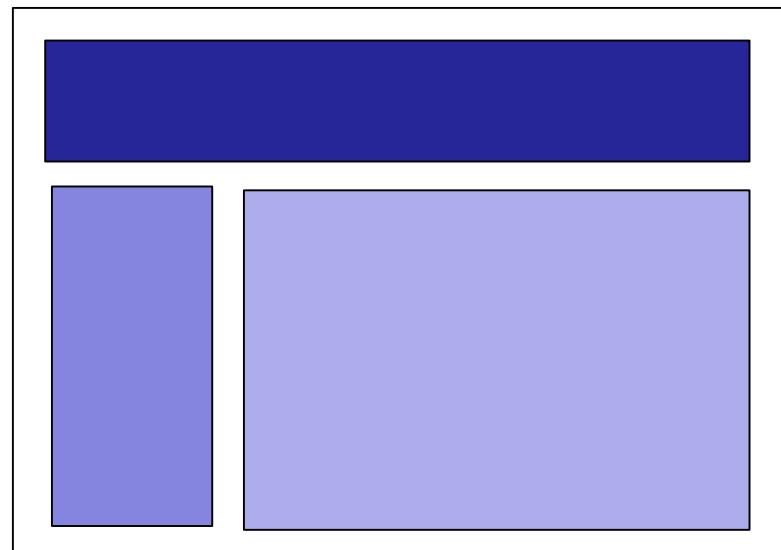
# Advantages

- No contract between apps
- Separate development of apps
- Separate deployment of apps
- Mixing technologies
- Mixing architectures
- Choosing the best solution per app
- Different teams, different skill sets



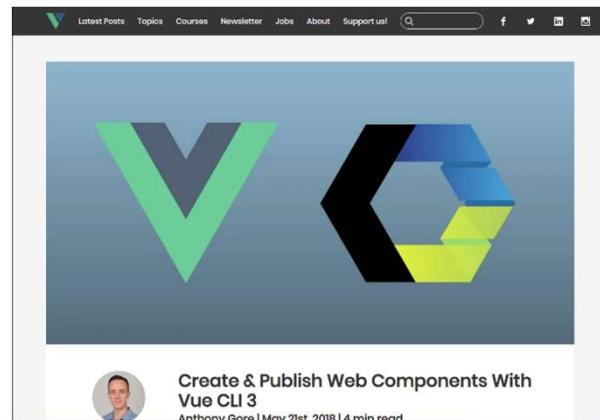
# Disadvantages

- **Loose state** when switching between applications
- Load a new application (and **overhead**) when switching between apps
- **Authorization/Authentication** is more difficult



# Approach 1 – Do it all yourself

- Angular – create **Angular Elements** from applications, load them in an Angular SPA application
  - (<https://github.com/PeterKassenaar/angular-microapp>)
- React, Vue, idem – export app as **Web Component**, load in SPA dashboard



A screenshot of the React documentation page for "Web Components". The page has a sidebar with navigation links like "INSTALLATION", "MAIN CONCEPTS", and "ADVANCED GUIDES". The main content area is titled "Web Components" and discusses the differences between React and Web Components. It includes a code snippet for a "HelloMessage" component.

# Approach 2 – use a library to create micro-apps

The screenshot shows the official documentation site for single-spa.js. The header includes the logo, navigation links for 'single-spa 5.x', 'FAQ', 'Docs' (which is active), 'Help', 'Blog', 'Donate', 'GitHub', language selection, a toggle switch, and a search bar.

The main content area features a sidebar with a dropdown menu for 'Overview' containing links like 'Overview of single-spa', 'Examples', 'Tutorials', 'CLI', 'Concept: Microfrontend', 'Concept: Root Config', 'Concept: Application', 'Concept: Parcel', 'The Recommended Setup', 'API', 'Ecosystem', 'Dev Tools', and 'Contributing'. A 'Version: 5.x' badge is visible above the main title.

## Getting Started with single-spa

### JavaScript microfrontends

Join the chat on Slack

single-spa is a framework for bringing together multiple javascript microfrontends in a frontend application. Architecting your frontend using single-spa enables many benefits, such as:

- Use multiple frameworks on the same page without page refreshing (React, AngularJS, Angular, Ember, or whatever you're using)
- Deploy your microfrontends independently.
- Write code using a new framework, without rewriting your existing app
- Lazy load code for improved initial load time.

### Demos and examples

See [our examples page](#).

### Architectural Overview

single-spa takes inspiration from modern framework component lifecycles by applying lifecycles to

The right sidebar contains a vertical list of links: 'JavaScript microfrontends', 'Demos and examples', 'Architectural Overview', 'The Recommended Setup', 'How hard will it be to use single-spa?', 'Isn't single-spa sort of a redundant name?', 'Documentation', 'Simple Usage', 'API', 'Contributing', 'Code of Conduct', 'Contributing Guide', and 'Who's Using This?'. A vertical scrollbar is visible on the right side of the page.

<https://single-spa.js.org/docs/getting-started-overview>

# frint.js (for JavaScript and React)

The screenshot shows the Frint.js website interface. At the top, there's a navigation bar with links to Documentation, Blog, REPL, and About. A search bar is also present. Below the navigation, a large banner features the text "Modular JavaScript for building Scalable & Reactive UIs". It includes a "LEARN MORE" button and a GitHub icon. To the right of the banner, a diagram illustrates the concept of modular loading. It shows a "Root App" represented by a green rectangular box with rounded corners. Inside the Root App, there's a "Region" represented by a white rectangular box with rounded corners. Within the Region, two purple rectangular boxes labeled "App 1" and "App 2" are shown. On the left side of the diagram, four files are listed in a vertical stack: "vendors.js" (blue), "root.js" (green), "app-1.js" (purple), and "app-2.js" (purple). Lines connect each of these four files to the corresponding components within the Root App and Region. At the bottom of the diagram, there's a link that says "Learn more about Apps and Regions, and Code Splitting."

<https://frint.js.org/>

Learn more about [Apps](#) and [Regions](#), and [Code Splitting](#).

# Demo / Result

The screenshot shows three browser tabs illustrating a micro-frontend architecture:

- localhost:4200**: The dashboard homepage. It features a navigation bar with "Angular Micro Frontends", "Home", "App 1", and "App 2". Below the nav, the text "This is the Dashboard homepage" is displayed. A list of instructions follows:
  - This is the file `./home.component.ts|html`.
  - Pick one of the items above to route to different micro-apps.
  - You can add files and components to the home application (e.g. the `./navbar` application) and adjust the routing table in `./app-routing.module.ts`.A small image of a laptop displaying a dashboard interface is shown on the right.
- localhost:4200/app1**: The App 1 micro-app. It has a navigation bar with "Angular Micro Frontends", "Home", "App 1", and "App 2". The "App 1" tab is active. The content area displays "Welcome to app1!" and a note: "This is `app1.component.ts|html` in the folder `./app1`". A green Yoshi character is on the right.
- localhost:4200/app2**: The App 2 micro-app. It has a navigation bar with "Angular Micro Frontends", "Home", "App 1", and "App 2". The "App 2" tab is active. The content area displays "Welcome to app2!" and a note: "This is `app2.component.ts|html` in the folder `./app2`". A large red Angular logo is on the left, and a running Mario character is on the right.

Red arrows point from the "App 1" and "App 2" tabs in the dashboard to their respective tabs in the child browser windows, indicating the modular nature of the application.

## **Our choice: single-spa**

*"single-spa is a framework for  
bringing together **multiple javascript  
microfrontends** in a frontend  
application"*

# Contents of a single-spa application

1. A **single-spa-config**, which is the root html page and the JavaScript that registers applications with single-spa.
2. One or more **applications**, which are registered with three things:
  - A name
  - A function to load the application's code
  - A function that determines when the application is active/inactive

# 1 single-spa config: the root index.html

In our example: ./dashboard-app/index.html:

```
<meta name="importmap-type" content="systemjs-importmap">
<script type="systemjs-importmap">
{
  "imports": {
    "app1": "http://localhost:4201/main.js",
    "app2": "http://localhost:4202/main.js",
    "navbar": "http://localhost:4300/main.js",
    "single-spa": "https://cdnjs.cloudflare.com/.../single-spa.min.js"
  }
}
</script>
```

Required!

# More scripts

Single-spa is built on top of system.js, so also load that:

```
<link rel="preload" href="https://cdnjs.cloudflare.com/ajax/libs/single-spa/4.3.5/system/single-spa.min.js"
      as="script" crossorigin="anonymous" />
<script src='https://unpkg.com/core-js-bundle@3.1.4/minified.js'></script>
<script src="https://unpkg.com/zone.js"></script>
<script src="https://unpkg.com/import-map-overrides@1.6.0/dist/import-map-overrides.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/4.0.0/system.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/4.0.0/extras/amd.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/4.0.0/extras/named-exports.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/4.0.0/extras/named-register.min.js"></script>
```

# Register all applications

```
<script>
System.import('single-spa').then(function (singleSpa) {
  singleSpa.registerApplication(
    'navbar',
    function () {
      return System.import('navbar');
    },
    function (location) {
      return true;
    }
  )

  singleSpa.registerApplication(
    'app1',
    function () {
      return System.import('app1');
    },
    function (location) {
      return location.pathname.startsWith('/app1');
    }
  );
  ..
  singleSpa.start();
})
</script>
```

A diagram illustrating the flow of the application registration process. Two red arrows point from text annotations to specific lines of code. The top arrow points to the first registration block, labeled "Register one or more apps". The bottom arrow points to the final line of the script, labeled "Start the application".

# Applications to be loaded:

- Each application is an entire **SPA itself**.
- You can **build them** with Angular, React, Vue or vanilla JS.
- Each application can respond to url **routing** events
- Each application must know how to **bootstrap**, **mount**, and **unmount** itself from the DOM.

The main difference between a traditional SPA and `single-spa` applications is that they must be able to **coexist** with other applications, and they do not each have their own html page (as they are loaded **inside** the main application).

# Application dependencies

Applications must have a single-spa-angular|react|vue dependency as a wrapper to register the app with single-spa:

```
"dependencies": {  
  "@angular-builders/custom-webpack": "^8",  
  "@angular/common": "~8.1.0",  
  "@angular/compiler": "~8.1.0",  
  "@angular/core": "~8.1.0",  
  ...  
  "single-spa-angular^3.0.1",  
  "tslib": "^1.9.0",  
  "zone.js": "~0.9.1"  
},
```



# Bootstrapping via main.ts

```
// main.single-spa.ts
import { enableProdMode, NgZone } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { Router } from '@angular/router';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
import singleSpaAngular from 'single-spa-angular';
import { singleSpaPropsSubject } from './single-spa/single-spa-props';

if (environment.production) {
  enableProdMode();
}

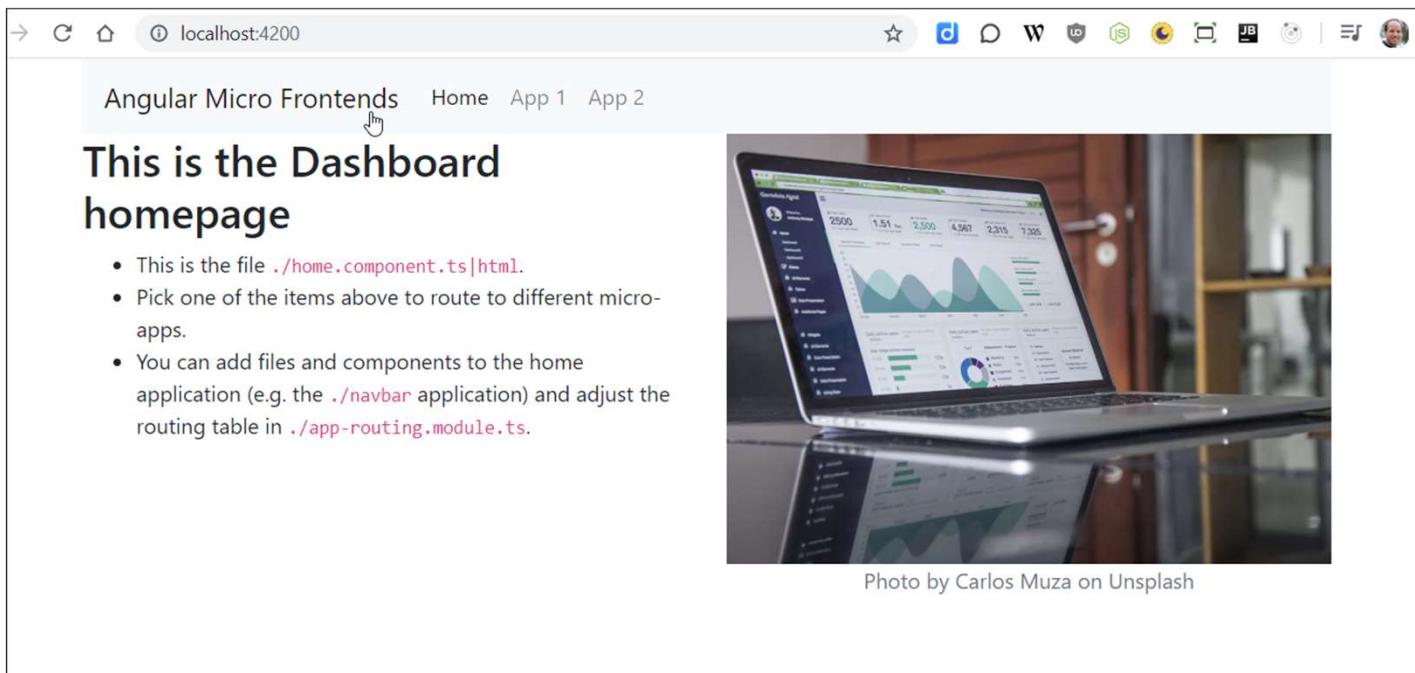
const lifecycles = singleSpaAngular({
  bootstrapFunction: singleSpaProps => {
    singleSpaPropsSubject.next(singleSpaProps);
    return platformBrowserDynamic().bootstrapModule(AppModule);
  },
  template: '<navbar-root />',
  Router,
  NgZone,
});

export const bootstrap = lifecycles.bootstrap;
export const mount = lifecycles.mount;
export const unmount = lifecycles.unmount;
```

Study this! (or simply use it)

# Running locally

- Start every sub-application: they run at their own port
  - App1 on localhost:4201
  - App2 on localhost:4202
  - And so on...
- Start the dashboard-app, which runs on localhost:4200



The screenshot shows a web browser window with the URL `localhost:4200` in the address bar. The page title is "Angular Micro Frontends". Below the title, there is a navigation bar with links for "Home", "App 1", and "App 2". A mouse cursor is hovering over the "Home" link. The main content area displays the text "This is the Dashboard homepage" and a bulleted list:

- This is the file `./home.component.ts|html`.
- Pick one of the items above to route to different micro-apps.
- You can add files and components to the home application (e.g. the `./navbar` application) and adjust the routing table in `./app-routing.module.ts`.

To the right of the browser window, there is a photograph of a laptop displaying a complex dashboard with various charts and data visualizations. The laptop is resting on a reflective surface.

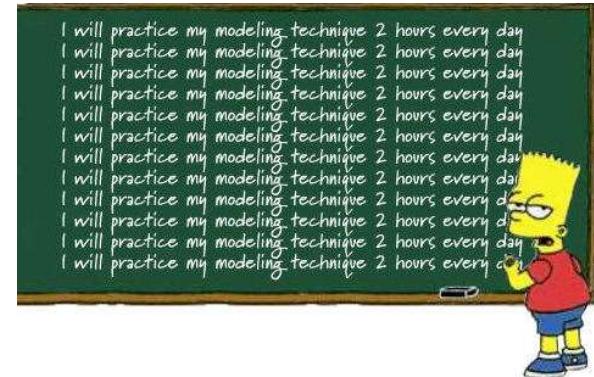
Photo by Carlos Muza on Unsplash

# Verdict on single-spa

- Pros
  - Builds on **proven technology** like SystemJS
  - Lots of support for tech stacks, different UI frameworks and so on, so **very flexible**
  - Supports **routing**
  - Custom **lifecycle events**, very robust
- Cons
  - No **version control**
  - No **component registry or viewer** (apart from `<import-map-overrides-full>`) debugging tool
  - Child projects **don't work "normally"** after conversion, since the default boot page doesn't understand how to launch a single child spa
    - Might be fixed with `environment variables`
  - **No live reloading** for Angular, React and Vue [out of the box]

# Workshop

- Work from the example: <https://github.com/PeterKassenaar/ng-microfrontends>
- See the `readme.md` file on how to get started
  - Perform an `npm install` for all separate apps
  - Perform an `npm start` for the separate apps
  - Open the dashboard at <http://localhost:4200>, see if you can get the example apps running
- Add a new app to the repo (app3). Think about:
  - Updating `./dashboard-app/index.html`
  - Updating `./navbar/` to add app3 to the navigation
  - Update/edit/add `main.single-spa.ts`
- <https://single-spa.js.org/docs/ecosystem-angular/>



# Angular documentation on single-spa

The screenshot shows the official documentation for the `single-spa` library. The top navigation bar includes links for `single-spa`, `FAQ`, `Docs` (which is currently selected), `Help`, `Blog`, `Donate`, and `GitHub`. There is also a language selection dropdown and a search bar.

The main content area displays the title `single-spa-angular` and a `Version: 5.x` badge. Below the title, there's a section titled `Introduction` which states: "single-spa-angular is a library for creating Angular microfrontends." It explains that each microfrontend is an Angular CLI project that can be deployed separately and will work together in a single web page.

The sidebar on the left contains a navigation menu:

- Overview
- Examples
- Tutorials
- CLI
- Concept: Microfrontend
- Concept: Root Config
- Concept: Application
- Concept: Parcel
- The Recommended Setup
- API
- Ecosystem
  - Overview
  - React
  - Vue
  - Angular
  - AngularJS
  - Cycle
  - Ember

The `Angular` item in the sidebar is highlighted with a light gray background. The main content area also has a section for the `Angular` ecosystem, which includes a link to the `#angular` channel in the slack workspace and a `Demo` section with a link to <https://coexisting-angular-microfrontends.surge.sh>.

<https://single-spa.js.org/docs/ecosystem-angular/>



# More info

Info on npm packages, monorepo's and micro apps

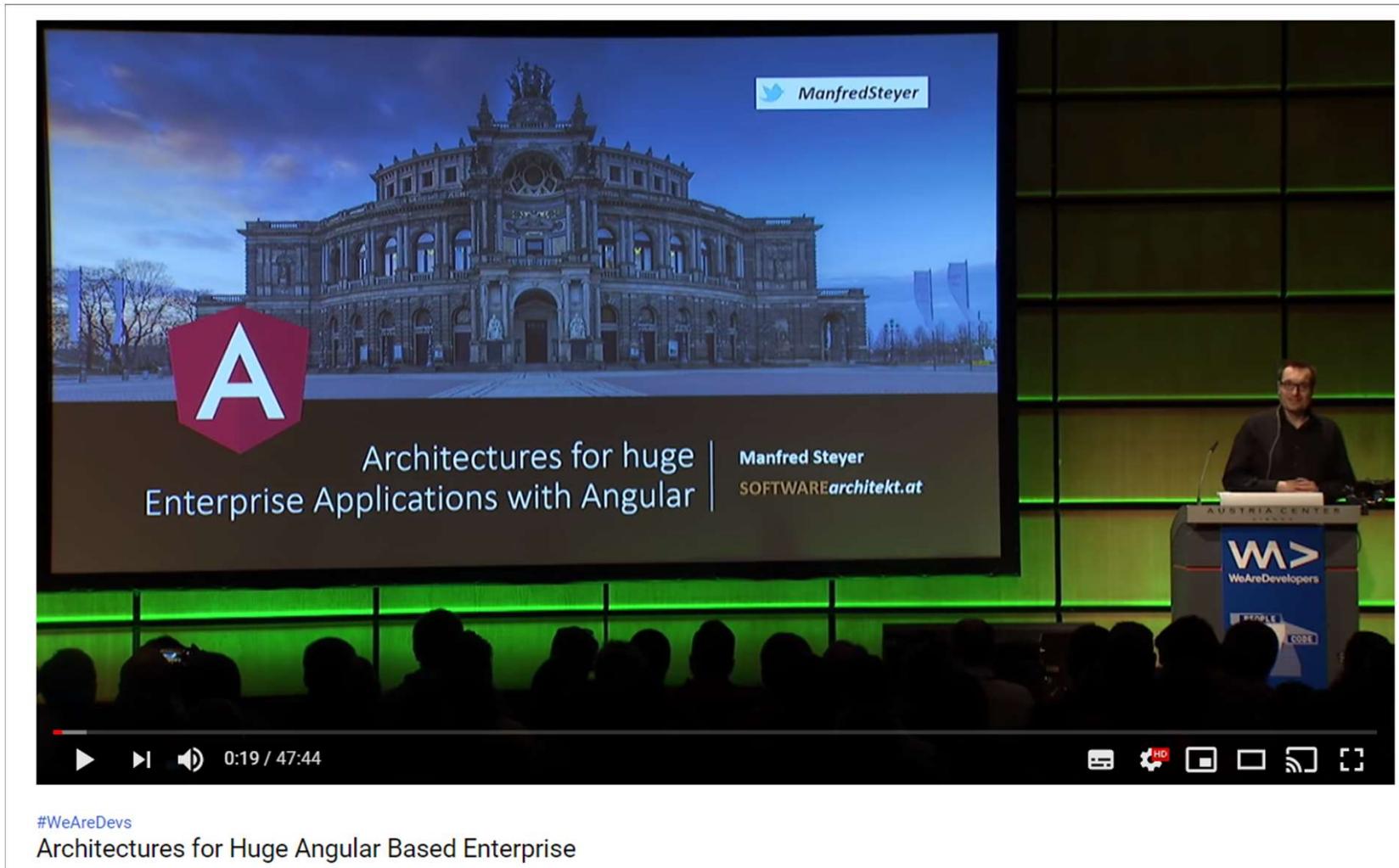
# Info on...

NPM  
packages

Monorepo

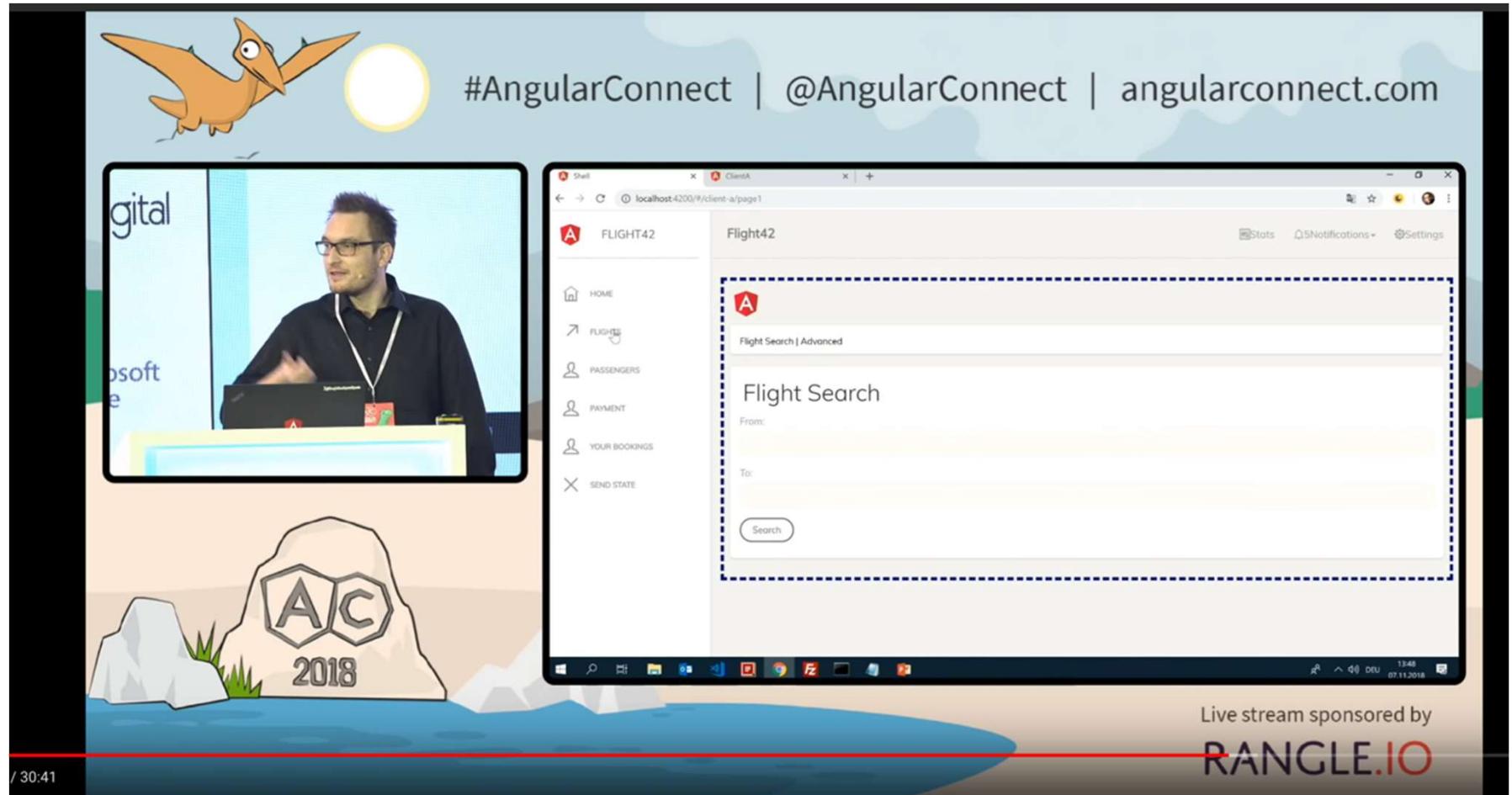
Micro apps

# Talks on Angular Monorepo's



- [https://www.youtube.com/watch?v=q4XmAy6\\_ucw](https://www.youtube.com/watch?v=q4XmAy6_ucw)

# Manfred Steyer – Angular Connect



<https://www.youtube.com/watch?v=YU-fMRs-ZYU>

Code: <https://github.com/PeterKassenaar/angular-microapp>

# Victor Savkin – creator of Nx



**Before**

- Googler on Angular team
- Blogged at [vsavkin.com](http://vsavkin.com)



**Now**

- Co-Founder at [nrwl.io](http://nrwl.io)
- Blog at [blog.nrwl.io](http://blog.nrwl.io)




Angular at Large Organizations - Victor Savkin

- <https://www.youtube.com/watch?v=piQ0EZhtus0>

# Publishing your library to npm

The screenshot shows a blog post on the Angular In Depth website. The header features the site's logo, navigation links for 'ABOUT' and 'SUPPORT US', and a call to action for 'AG-GRID: THE BEST ANGULAR GRID IN THE WORLD'. The main title of the post is 'The Angular Library Series — Publishing', with a subtitle 'Publishing your Angular Library to npm'. Below the title is a profile picture of Todd Palmer, a 'Follow' button, and the date 'Aug 29, 2018 · 6 min read'. The post content is a large image of a grand library interior with multiple levels of bookshelves and ornate columns.

<https://blog.angularindepth.com/the-angular-library-series-publishing-ce24bb673275>

# Implementing micro apps in Angular

M Sign in

BB Tutorials & Thoughts FRONTEND BACKEND REACT CLOUD COMPUTING DOCKER | K8S PYTHON CERTIFICATIONS

## How To Implement Micro-Frontend Architecture With Angular

Everything you need to know about microservice oriented architecture for the frontend from beginner to advanced

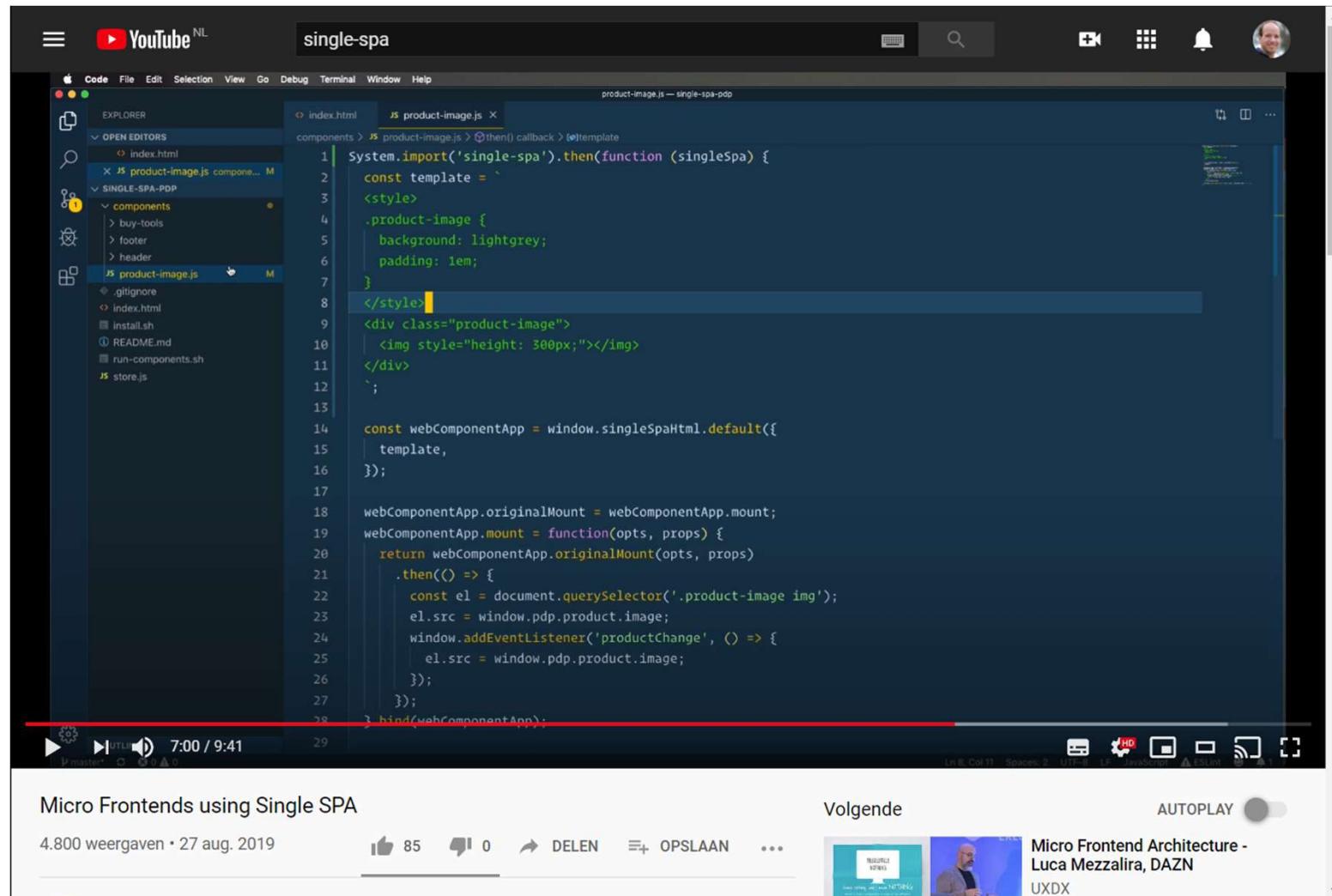
 Bhargav Bachina Follow  
Jan 10 • 8 min read ★





https://medium.com/bb-tutorials-and-thoughts/how-to-implement-micro-frontend-architecture-with-angular-e6828a0a049c

# Using single-spa with various tech stacks



<https://www.youtube.com/watch?v=wU06eTMQ6yl>

## More info

- <https://blog.angularindepth.com/creating-a-library-in-angular-6-87799552e7e5>
- <https://blog.angularindepth.com/creating-a-library-in-angular-6-part-2-6e2bc1e14121>
- <https://blog.angularindepth.com/angular-workspace-no-application-for-you-4b451afcc2ba>