

illionx



Global Knowledge.®

Angular Advanced Monorepo's - introduction

Peter Kassenaar –
info@kassenaar.com

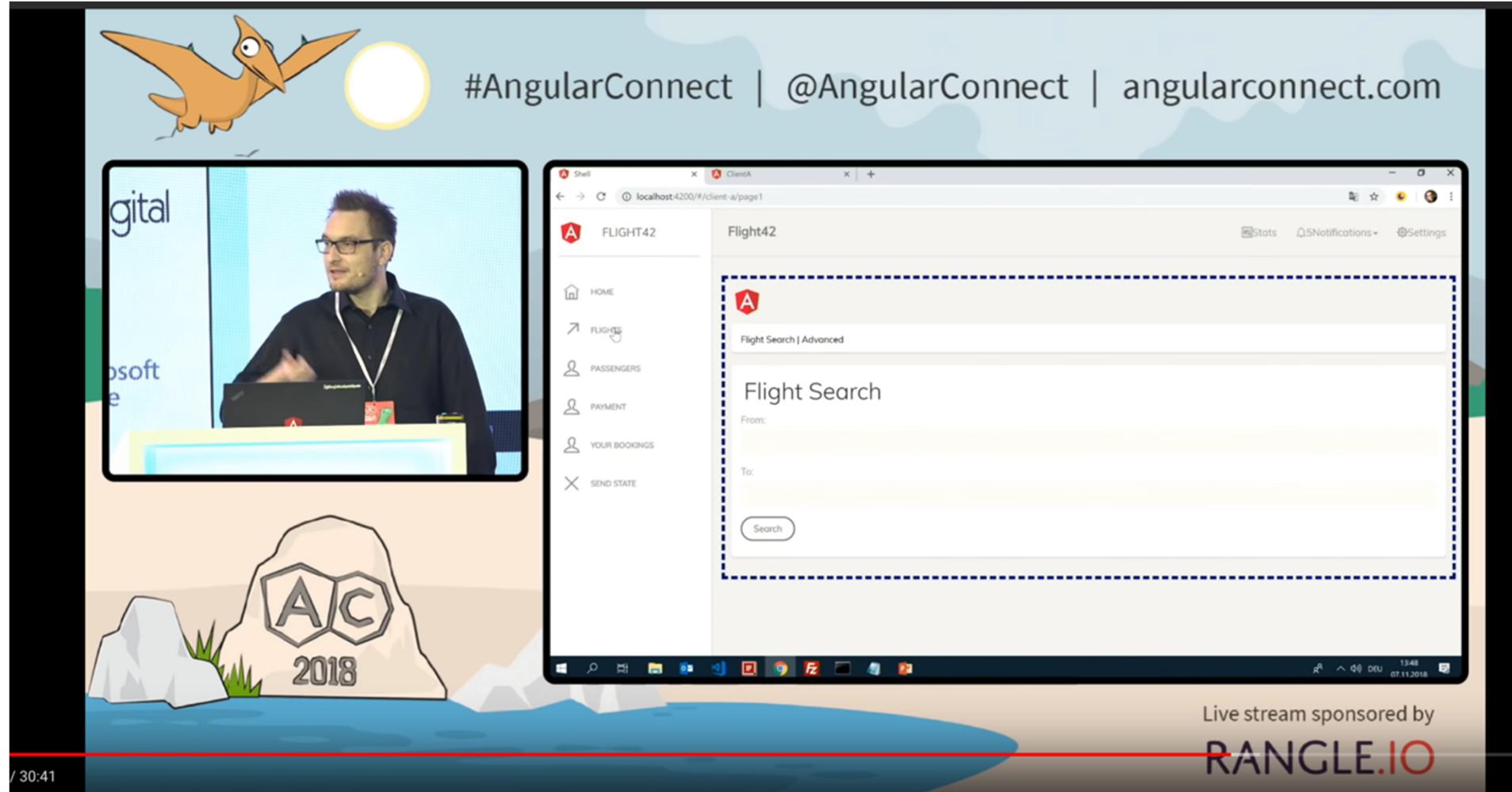
WORLDWIDE LOCATIONS

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



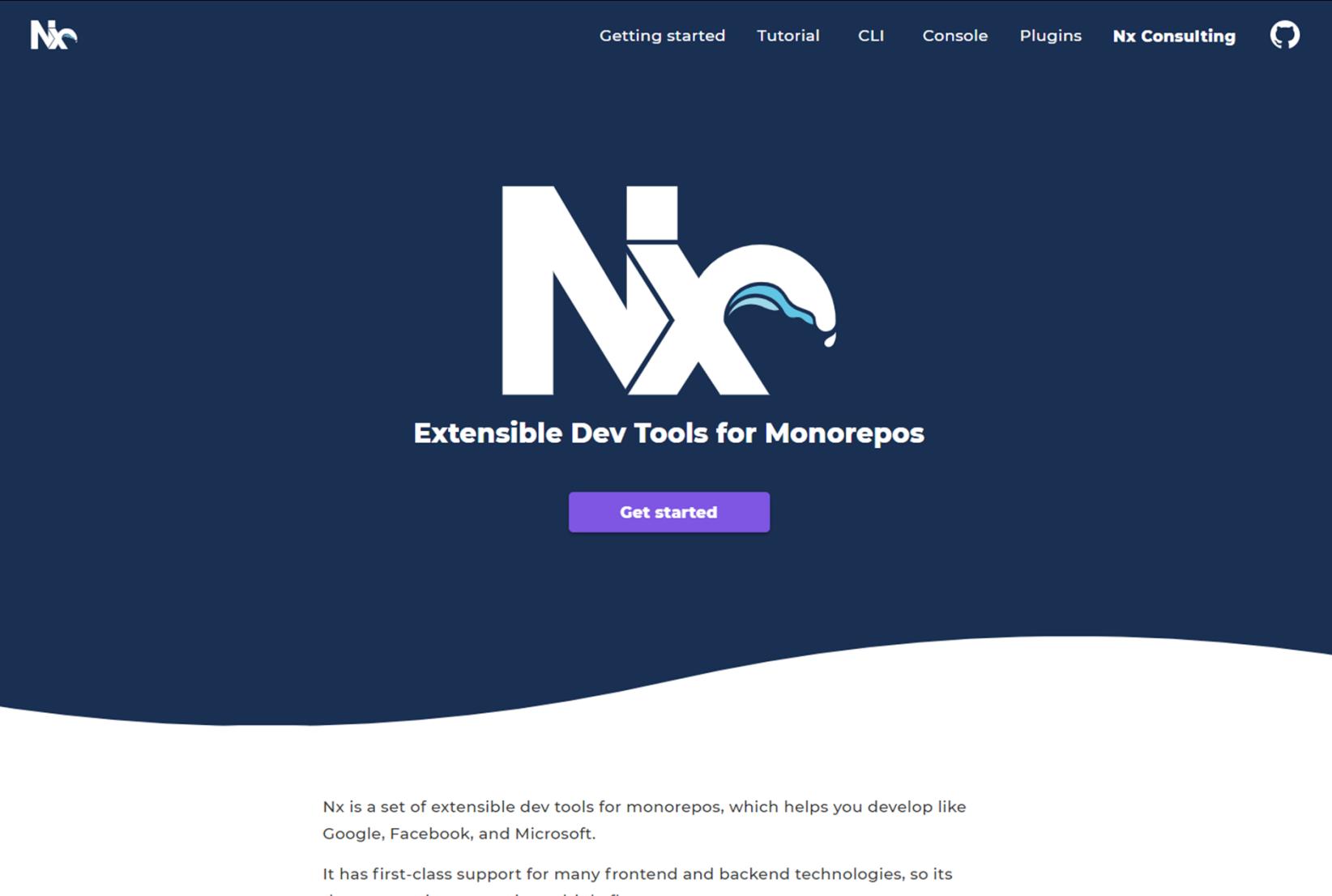
Monorepos

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 (opiniated) 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



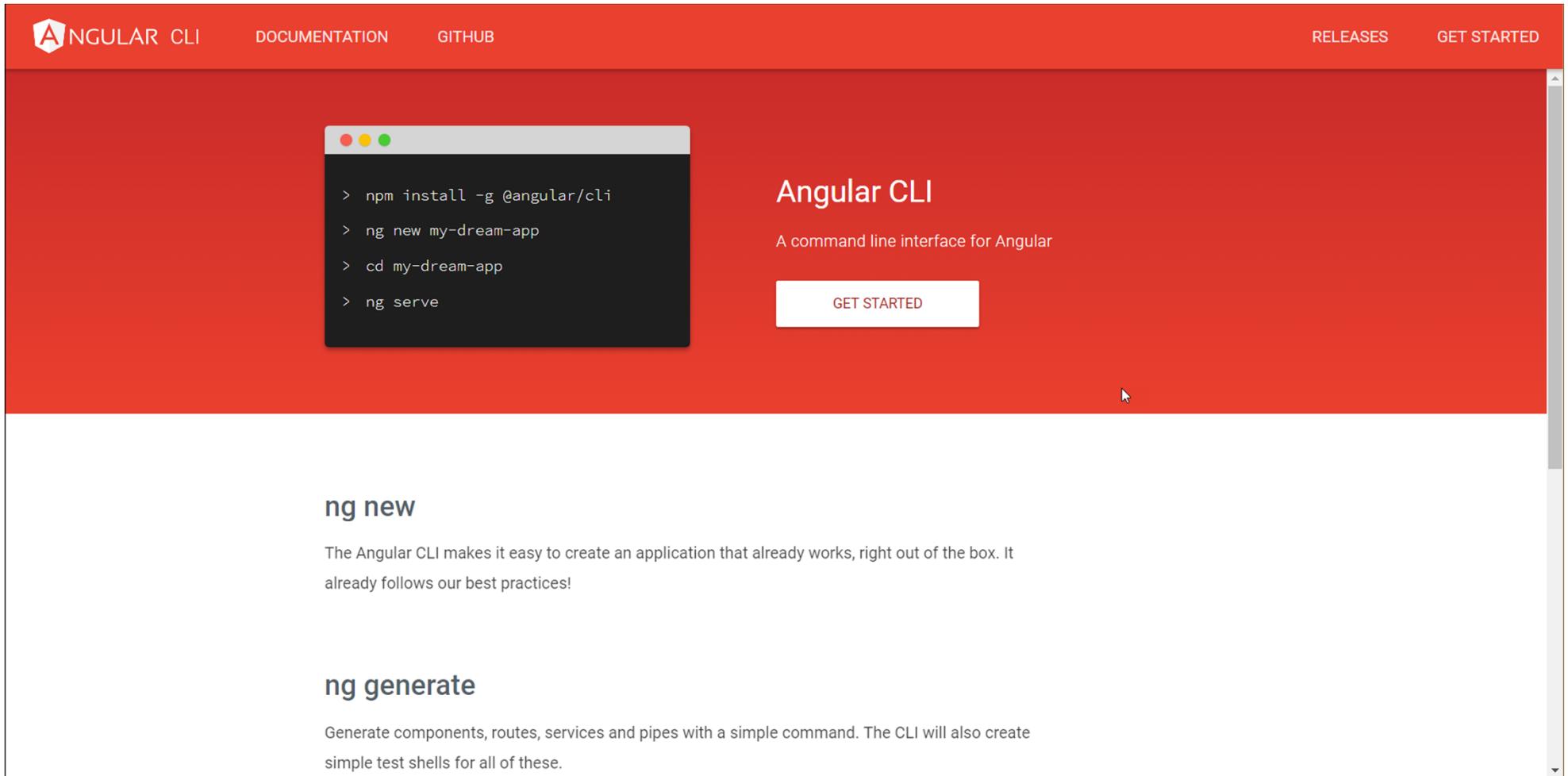
The screenshot shows the Nx website homepage. At the top, there's a dark blue header with the Nx logo on the left and navigation links for "Getting started", "Tutorial", "CLI", "Console", "Plugins", "Nx Consulting", and a GitHub icon on the right. The main content area has a dark blue background. In the center, there's a large white Nx logo with a blue wave graphic integrated into the letter "x". Below the logo, the text "Extensible Dev Tools for Monorepos" is displayed in white. A purple button labeled "Get started" is positioned below this text. At the bottom of the page, there's a white footer section containing two paragraphs of text.

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 links for DOCUMENTATION, GITHUB, RELEASES, and GET STARTED. Below the header, there's a large image of a terminal window showing command-line instructions for creating a new Angular application. To the right of the terminal image, the text "Angular CLI" is displayed in white, followed by a subtitle "A command line interface for Angular" and a "GET STARTED" button. The main content area below features two sections: "ng new" and "ng generate".

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!

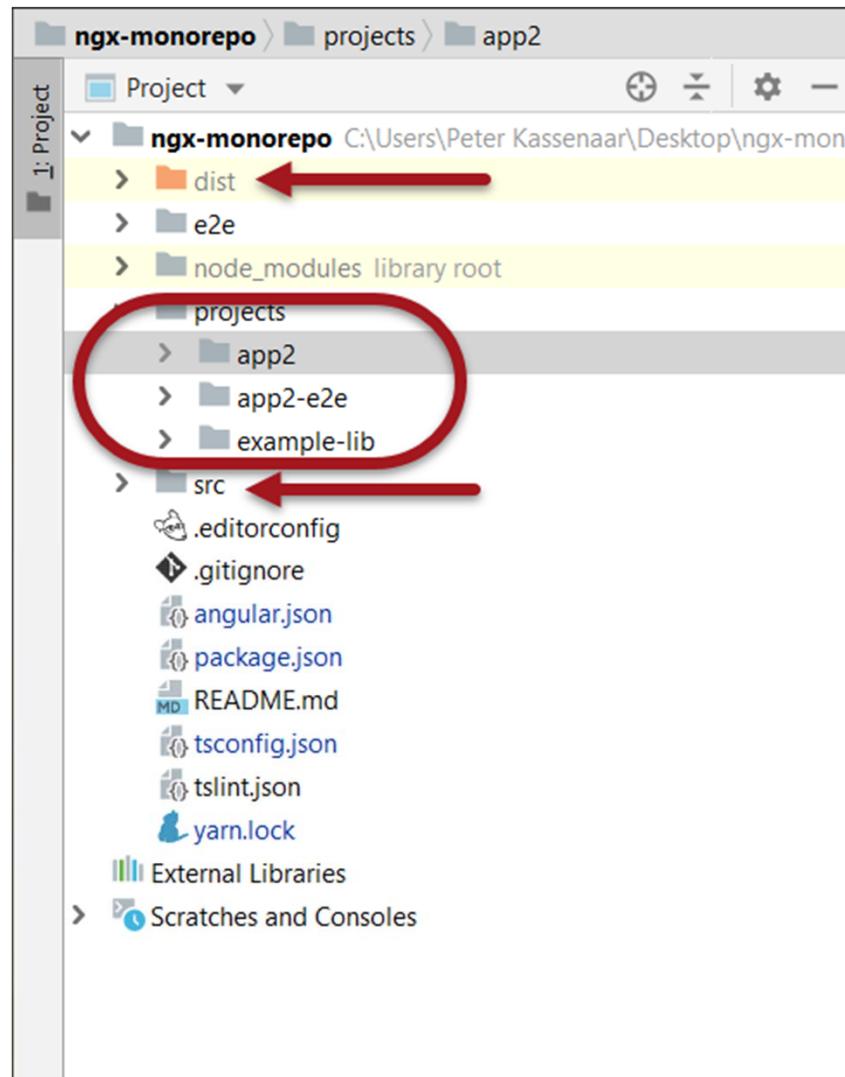
ng generate

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



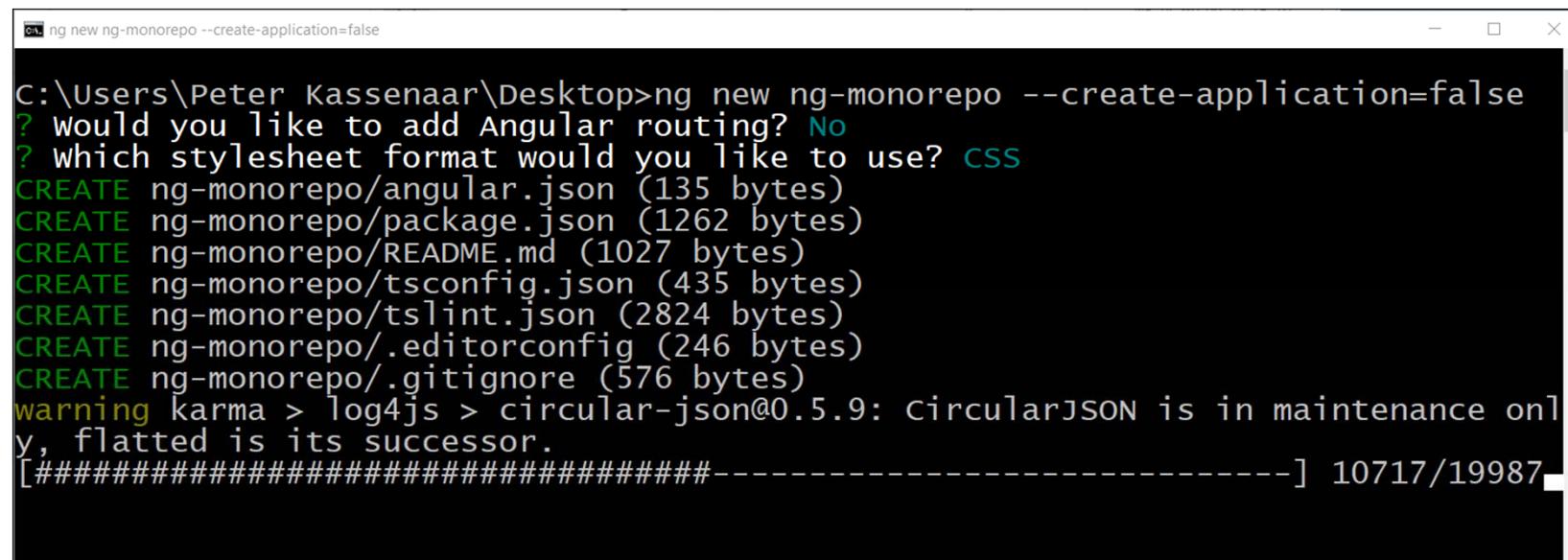
Credits: Angular in Depth

The screenshot shows a blog post on the Angular In Depth website. The header features the site's logo (a red 'M' icon), the title 'Angular In Depth by ag-Grid', a 'Follow' button, and a navigation bar with links for HOME, ANGULAR, RXJS, NGRX, ABOUT, SUPPORT US, and a tagline 'AG-GRID: THE BEST ANGULAR GRID IN THE WORLD'. The main content area has a large, bold title 'The Angular Library Series - Creating a Library with Angular CLI'. Below it is a subtitle: 'Angular libraries just got a lot easier to create thanks to the combination of Angular CLI and ng-packagr.' A profile picture of Todd Palmer, a follow button, and a timestamp ('May 28, 2018 · 11 min read') are visible. The background of the main content area is a photograph of a modern building's interior with curved, light-colored walls.

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

High level overview of the steps

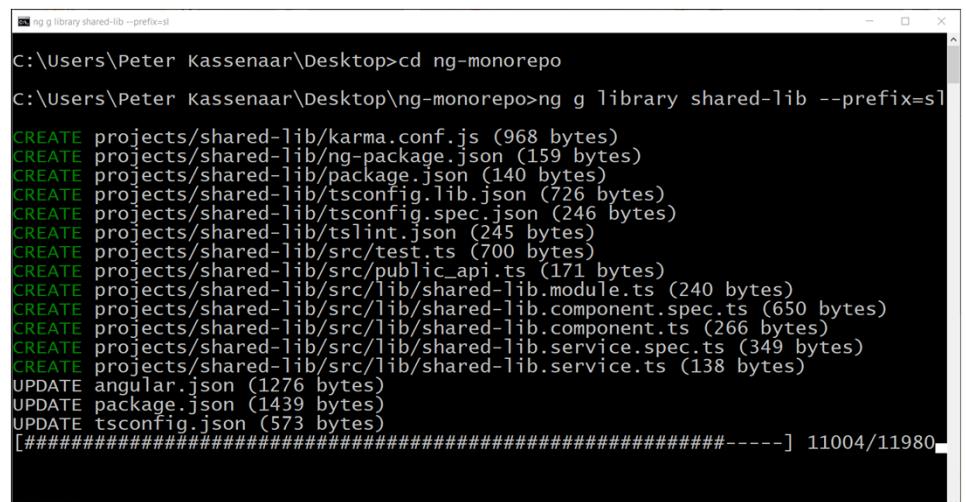
1. Create the library project using `ng new <mono-repo-name>`
 - We call this folder a *workspace*
2. The main app is mostly used for documentation, or example usage of the (shared) libs
 - Don't want the main app? Use `--create-application=false` flag!
 - <https://blog.angularindepth.com/angular-workspace-no-application-for-you-4b451afcc2ba>



```
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 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
```

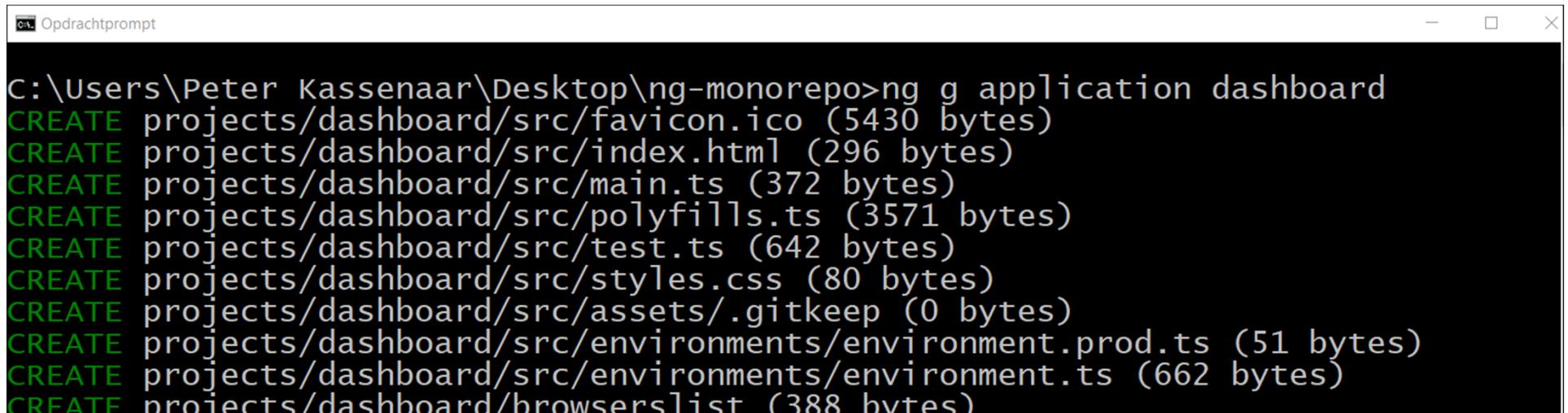
```
1  {
2      "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
3      "version": 1,
4      "newProjectRoot": "projects",
5      "projects": {
6          "shared-lib": {"root": "projects/shared-lib"...}
41     },
42     "defaultProject": "shared-lib"
43 }
```

```
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

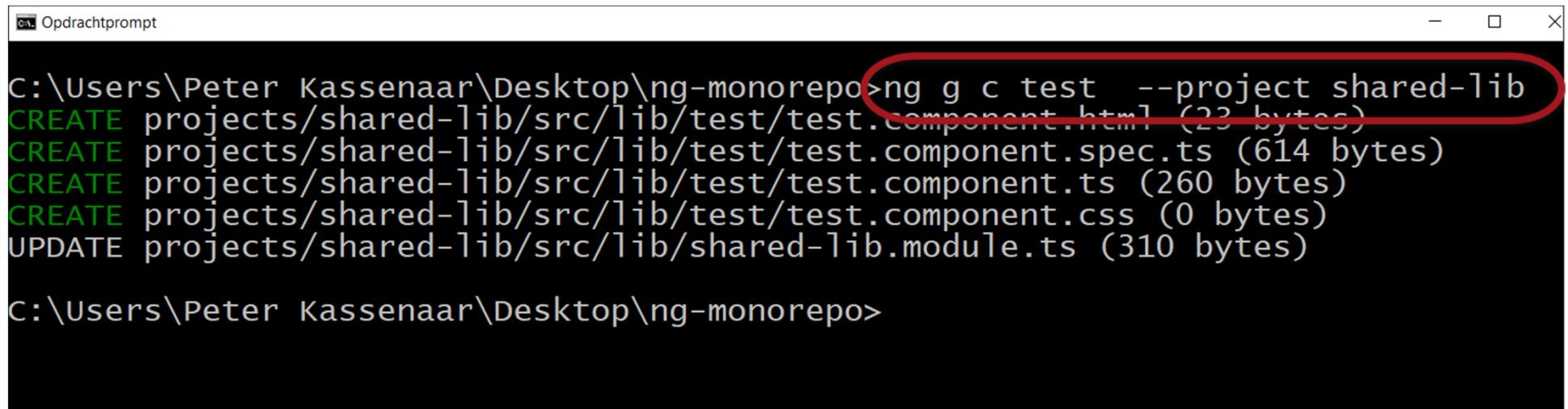
- Generate the (first) application
 - ng generate application <application-name>
 - Again, angular.json and package.json are updated with the new project



```
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 output is as follows:

```
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>
```

The command and its output are highlighted with a red oval.

Export the component

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

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

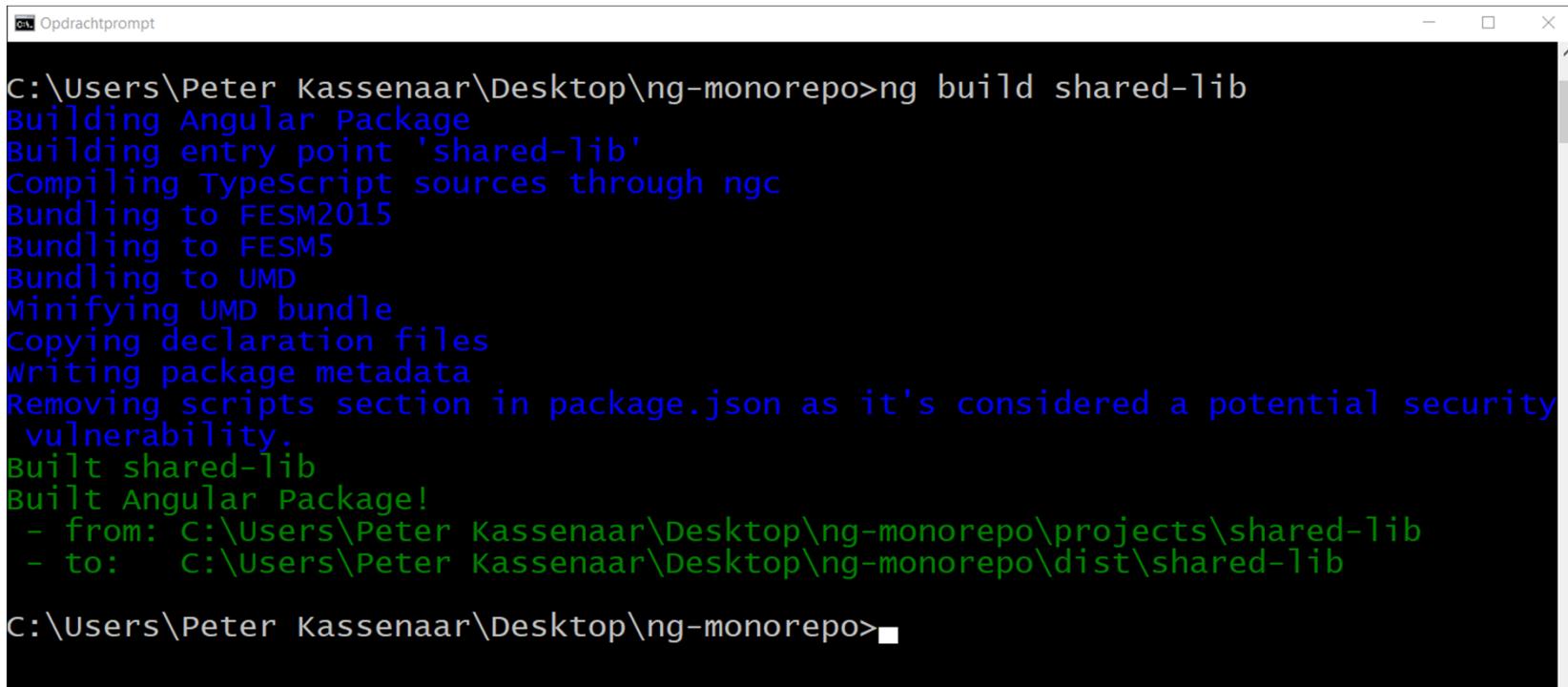
- Update the public_api.ts file to make the class available
 - Don't forget!

```
/*
 * 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'; X

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



```
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

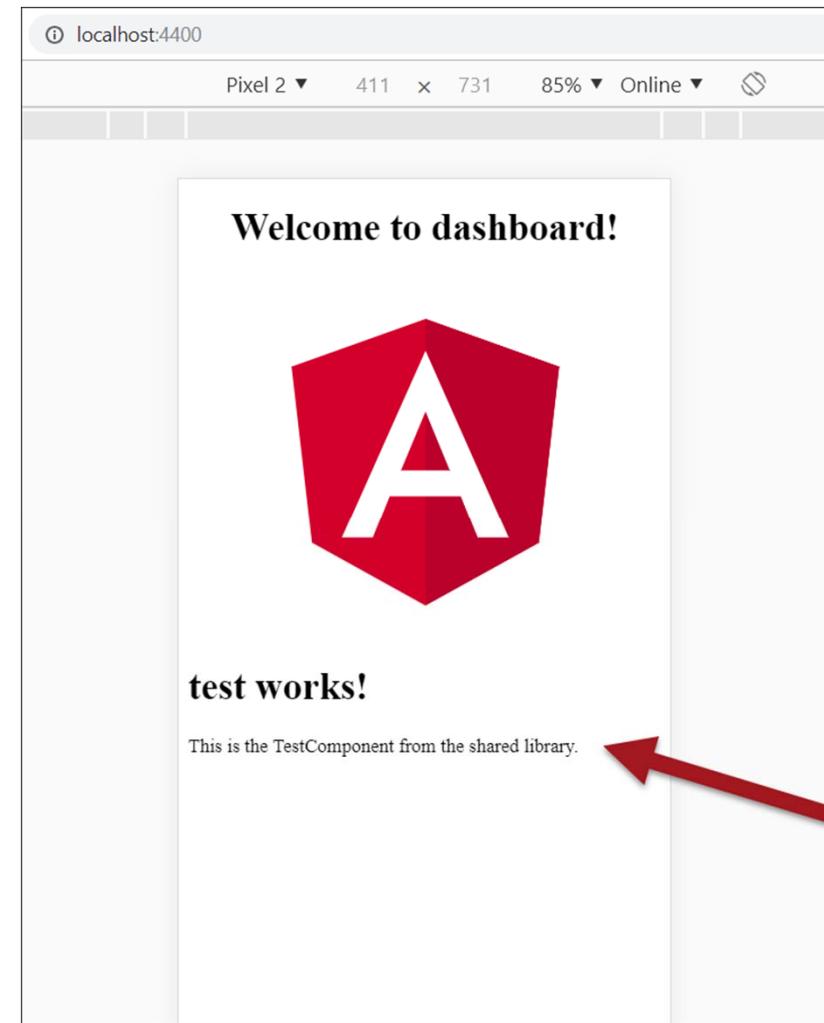
- Use the selector of the component from the shared library as normal

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

Running the application

- Use the `ng serve` command
 - Use the `--project=<project-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"  
},
```



library

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



```
Opdrachtprompt

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

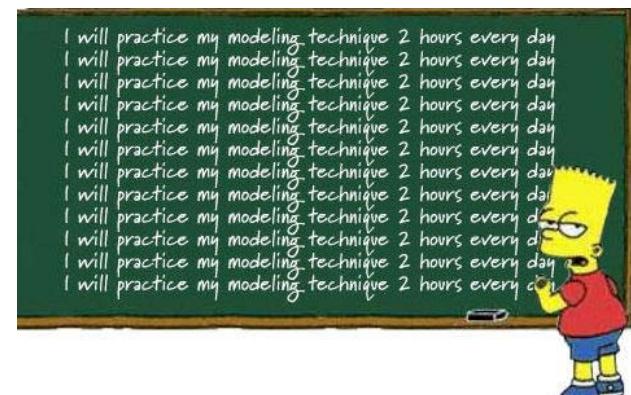
```
[ { "name": "Peter", "email": "test@test.com" }, { "name": "Sandra", "email": "sandra@sandra.com" } ]
```



Workshop

- Create a new application inside the Monorepo
- Import the shared library
- Use/show the <s1-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



Shorten TypeScript imports

AngularFirebase 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 like `@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/*"]  
    }  
  }  
}
```

<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

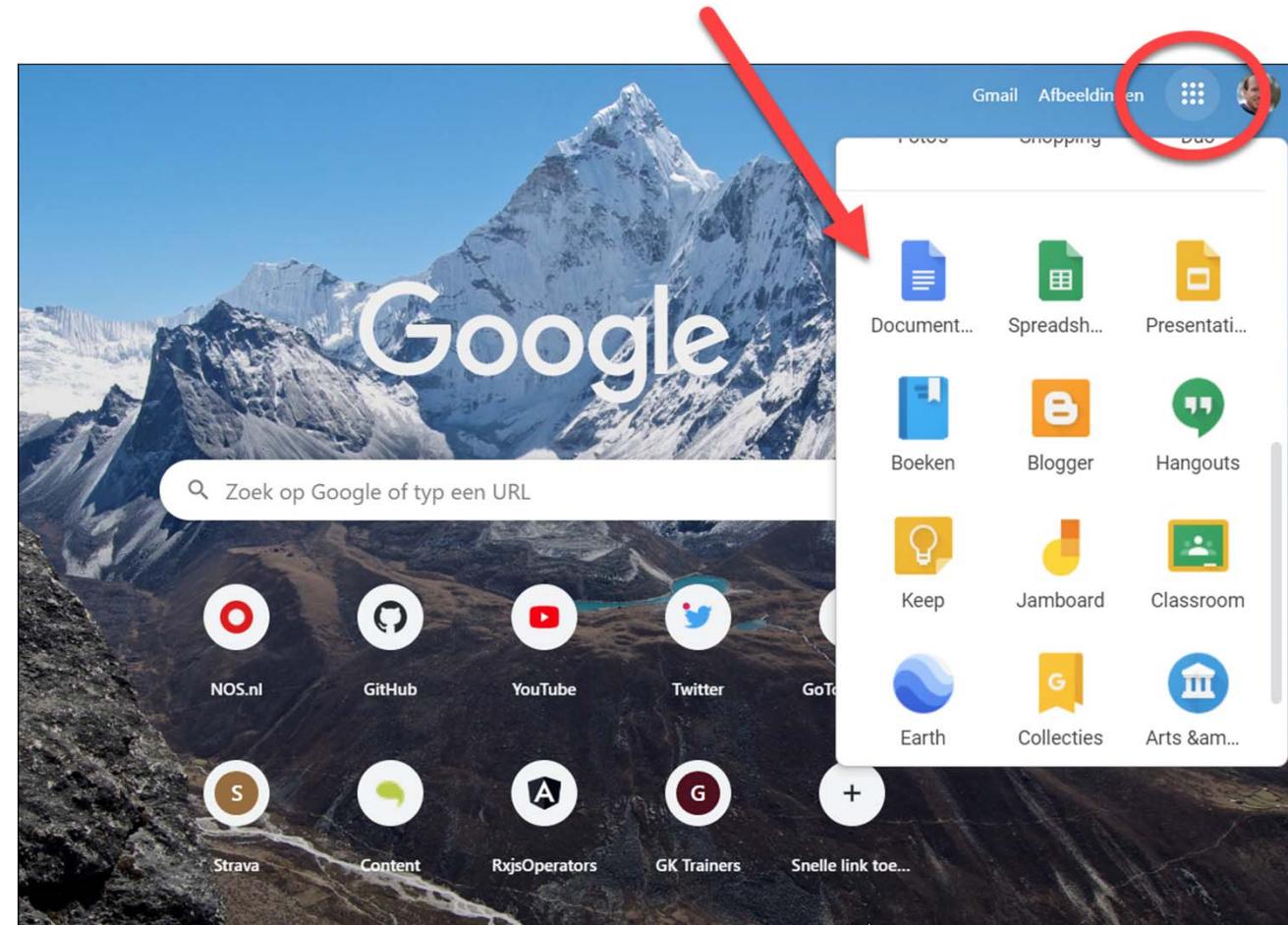
Micro apps or micro frontend - When?

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

Classical example:

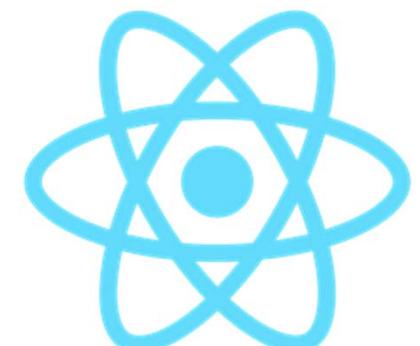
Google Suite or
Microsoft Office.

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



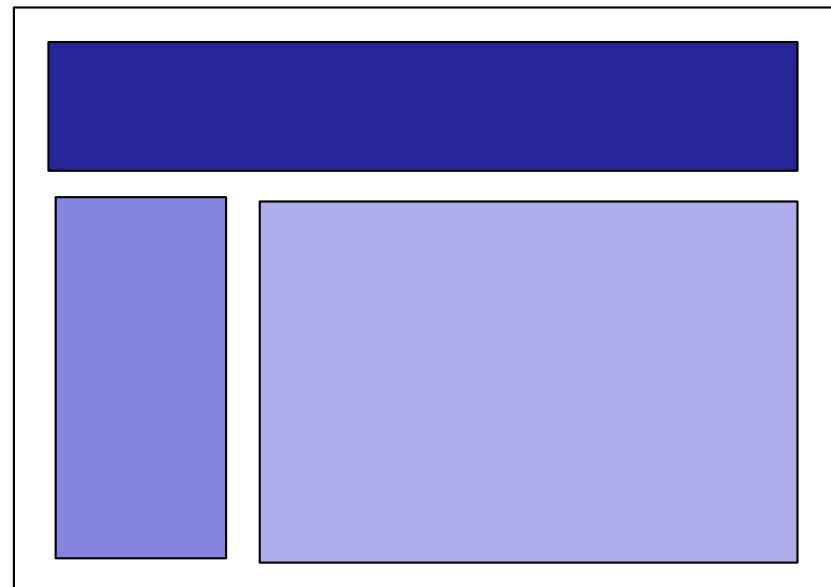
Advantages

- 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



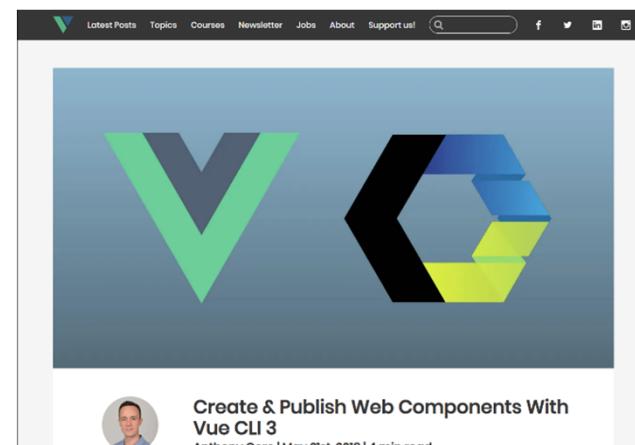
Disadvantages

- 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
- React, Vue, idem – export app as **Web Component**, load in SPA dashboard

A screenshot of the React documentation. The main heading is "Web Components". It discusses the difference between React and Web Components and provides a code snippet for "Using Web Components in React":

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search></div>;
  }
}
```

Approach 2 – use a library to create micro-apps

The screenshot shows the homepage of the single-spa.js.org website. The header includes a logo, navigation links for 'single-spa 5.x', 'FAQ', 'Docs' (which is highlighted), 'Help', 'Blog', 'Donate', 'GitHub', and language selection ('简体中文'). A search bar and a toggle switch are also present.

The main content area features a sidebar with a tree view of topics like 'Overview', 'Examples', 'Tutorials', 'CLI', 'Concept: Microfrontend', etc. The 'Overview' section is expanded, showing 'Overview of single-spa' as the active page.

The main content area has a heading 'Getting Started with single-spa' and a sub-section 'JavaScript microfrontends'. It includes a link to 'Join the chat on Slack' and a description of what single-spa is and its benefits. A bulleted list details these benefits:

- 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.

Below this, there's a 'Demos and examples' section with a link to the examples page. The footer contains a 'Architectural Overview' section and a note about lifecycles.

On the right side, there's a sidebar with links to various documentation and community pages, such as '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?'.

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

frint.js (for JavaScript and React)

The image shows a screenshot of the frint.js website on the left and a conceptual diagram on the right.

Website Screenshot:

- Header:** Frint, Documentation, Blog, REPL, About, Search bar.
- Section:** Modular **JavaScript** for building **Scalable & Reactive** web apps.
- Buttons:** LEARN MORE, GIT, DOWNLOAD.

Diagram Description:

- Root App:** A large green window labeled "Root App".
- Region:** A white box labeled "Region" containing two purple boxes labeled "App 1" and "App 2".
- Bundles:** Four colored boxes on the left (blue, green, purple, purple) labeled "vendors.js", "root.js", "app-1.js", and "app-2.js".
- Connections:** Lines connect each bundle to the corresponding app within the Root App and Region.

Bottom Text: Learn more about **Apps** and **Regions**, and **Code Splitting**.

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

Demo / Result

localhost:4200

Angular Micro Frontends Home App 1 App 2

This is the Dashboard homepage

- 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`.

localhost:4200/app1

Angular Micro Frontends Home App 1 App 2

Welcome to app1!

This is `app.component.ts|html` in the folder `./app1`.

localhost:4200/app2

Angular Micro Frontends Home App 1 App 2

Welcome to app2!

This is `app.component.ts|html` in the folder `./app2`.

To App 1 from app2

Mario

Yoshi

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>
```

Register one or more apps

Start the application

Applications to be loaded:

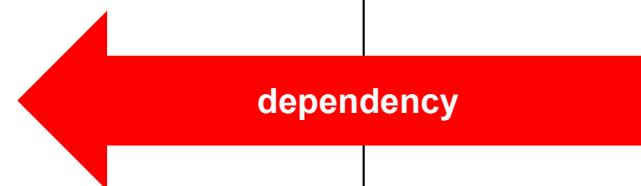
- Each application is an entire **SPA itself**.
- You can **build them** with Angular, React, Vue or vanilla JS.
- Each application can respond to url **routing** events
- Each application must know how to **bootstrap**, **mount**, and **unmount** itself from the DOM.

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

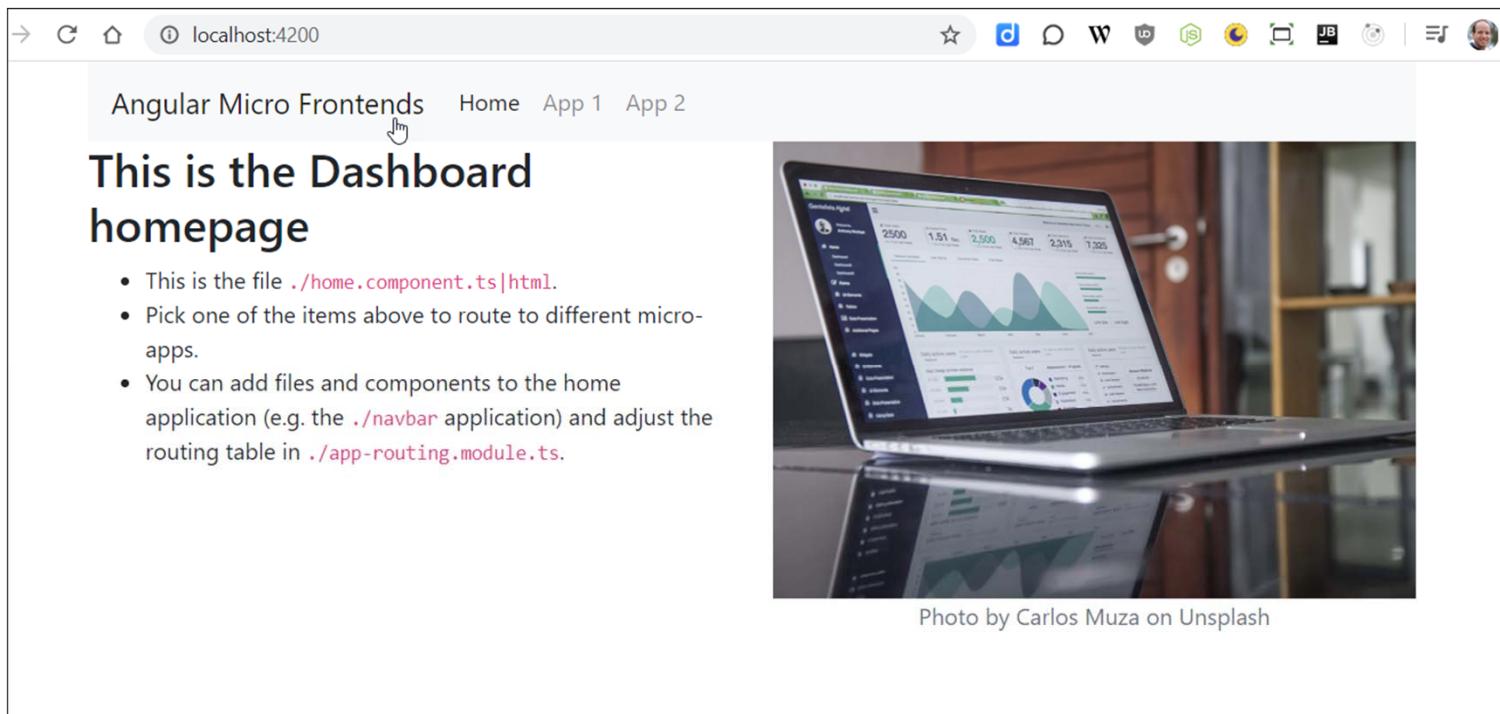


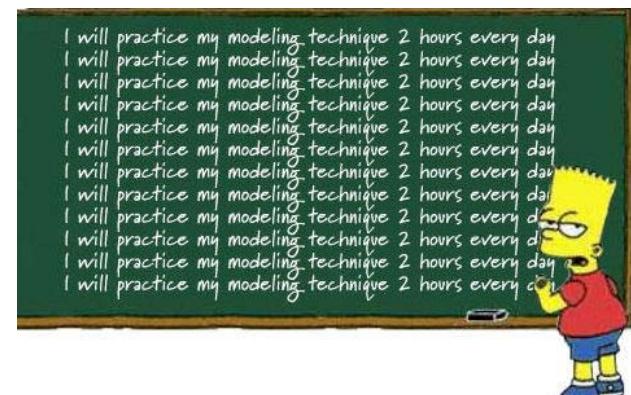
Photo by Carlos Muza on Unsplash

Verdict on single-spa

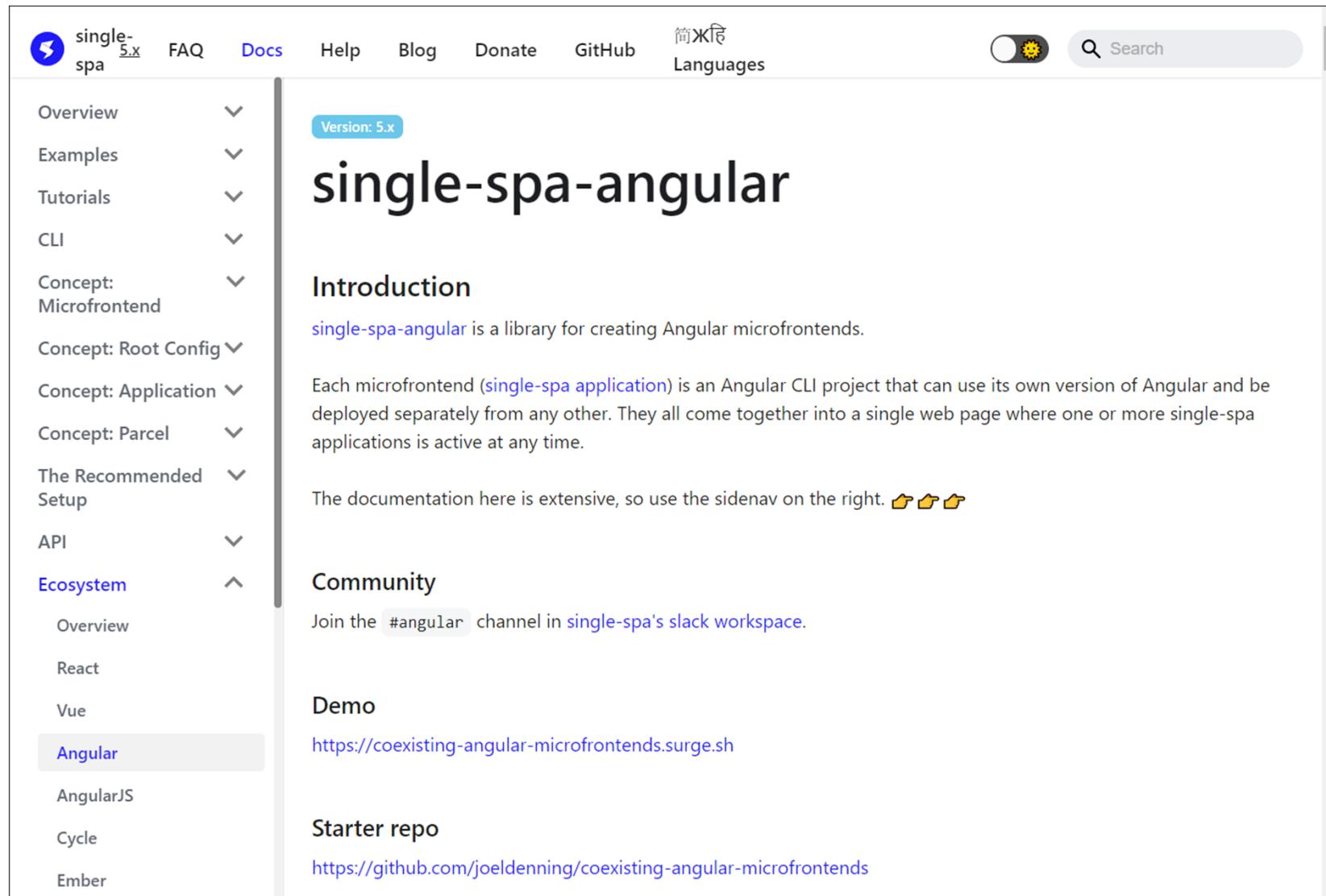
- Pros
 - Builds on **proven technology** like SystemJS
 - Lots of support for tech stacks, different UI frameworks and so on, so very **flexible**
 - Supports **routing**
 - Custom **lifecycle events**, very robust
- Cons
 - No **version control**
 - No **component registry or viewer** (apart from `<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
 - **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



The screenshot shows the documentation for the `single-spa-angular` library. The left sidebar has a tree view of documentation sections under the `Ecosystem` heading, with `Angular` selected. The main content area displays the `single-spa-angular` page, which includes an introduction, a note about the extensive documentation, a community section, a demo section with a link to a surge.sh site, and a starter repo section with a GitHub link.

single-spa 5.x FAQ **Docs** Help Blog Donate GitHub 简体中文
Languages

Version: 5.x

single-spa-angular

Introduction

`single-spa-angular` is a library for creating Angular microfrontends.

Each microfrontend (`single-spa application`) is an Angular CLI project that can use its own version of Angular and be deployed separately from any other. They all come together into a single web page where one or more single-spa applications is active at any time.

The documentation here is extensive, so use the sidenav on the right. 👉👉👉

Community

Join the `#angular` channel in `single-spa's slack workspace`.

Demo

<https://coexisting-angular-microfrontends.surge.sh>

Starter repo

<https://github.com/joeldenning/coexisting-angular-microfrontends>

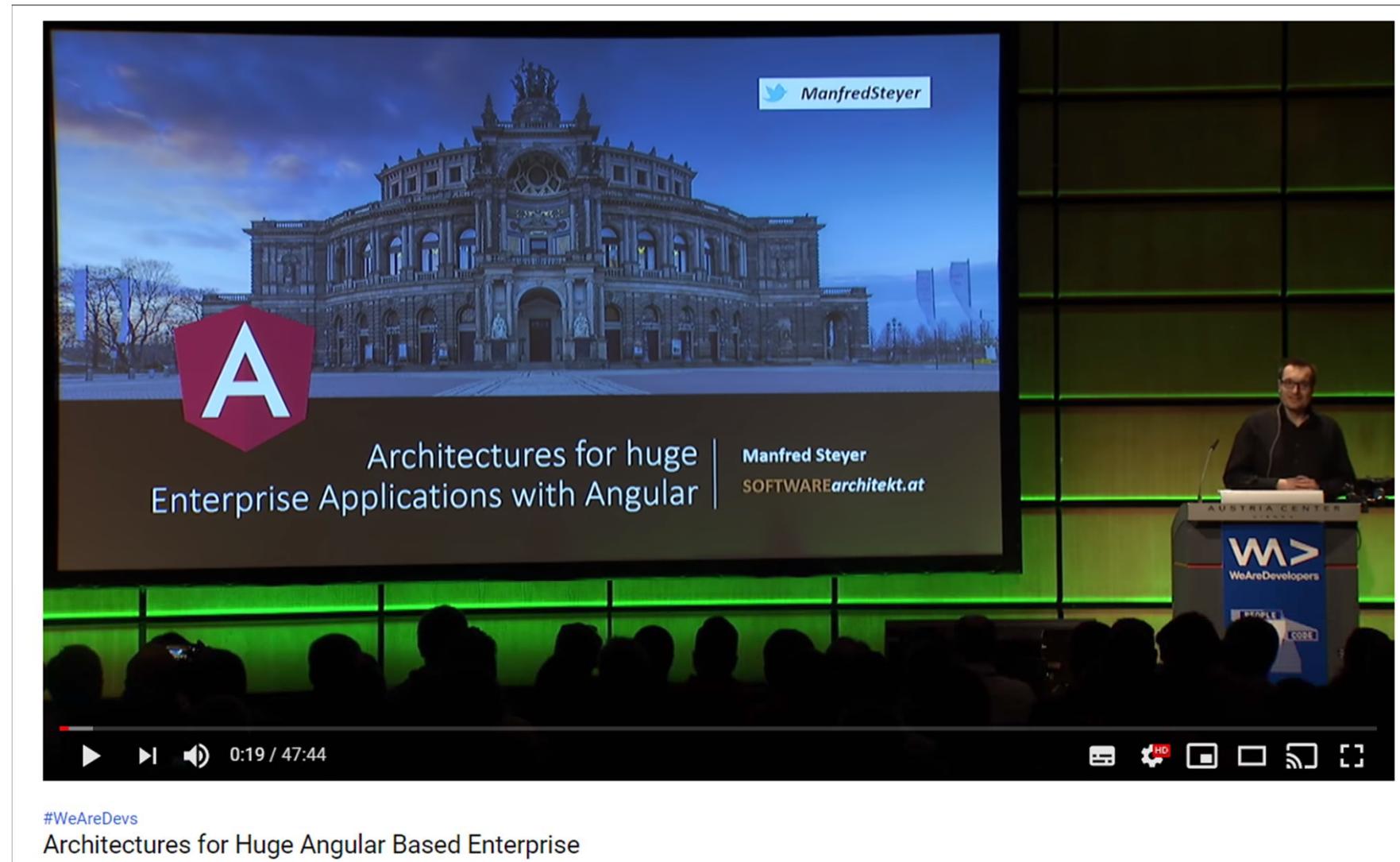
<https://single-spa.js.org/docs/ecosystem-angular/>



More info

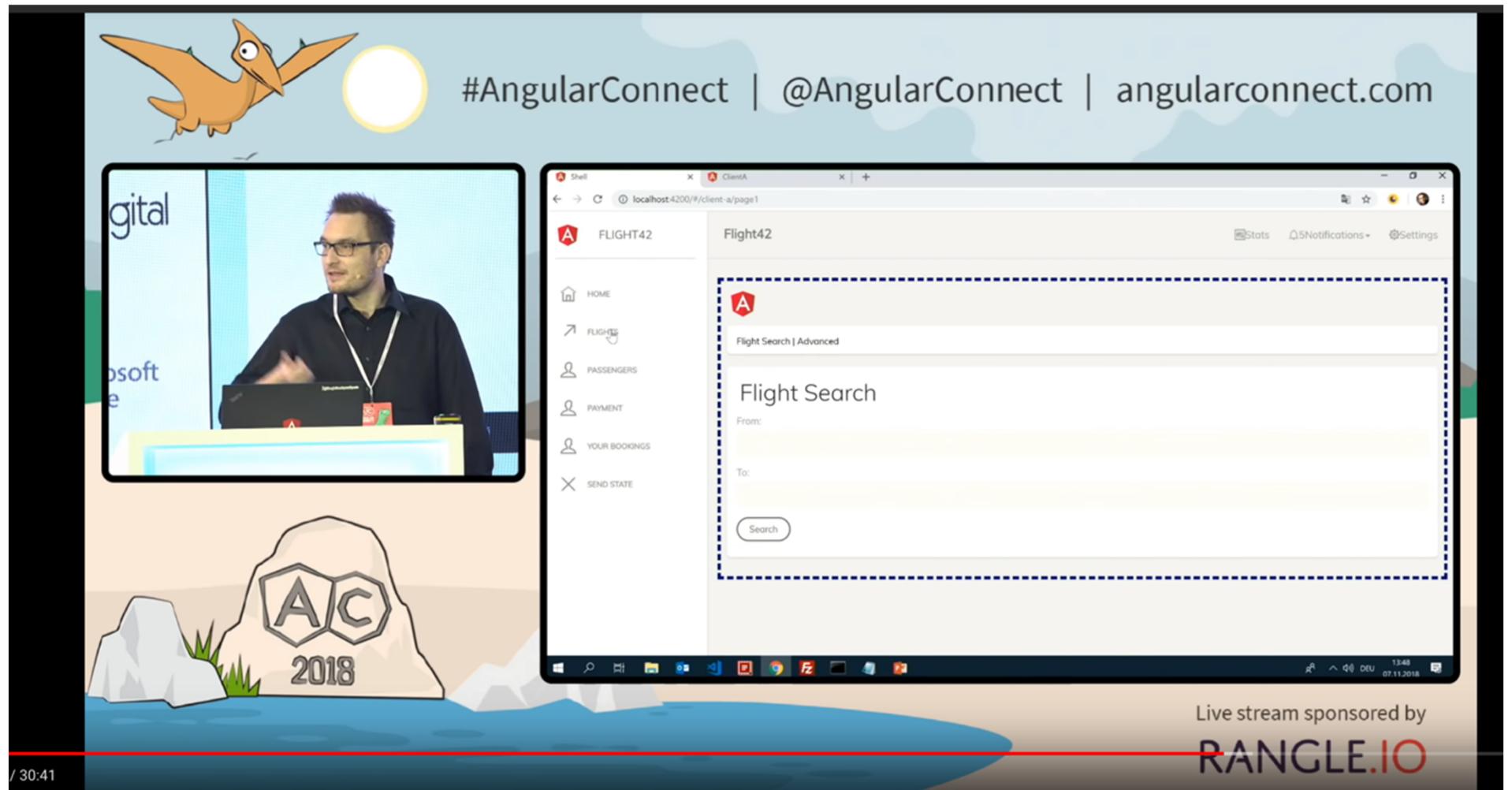
Info on npm packages, monorepo's and micro apps

Talks on Angular Monorepo's



- 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



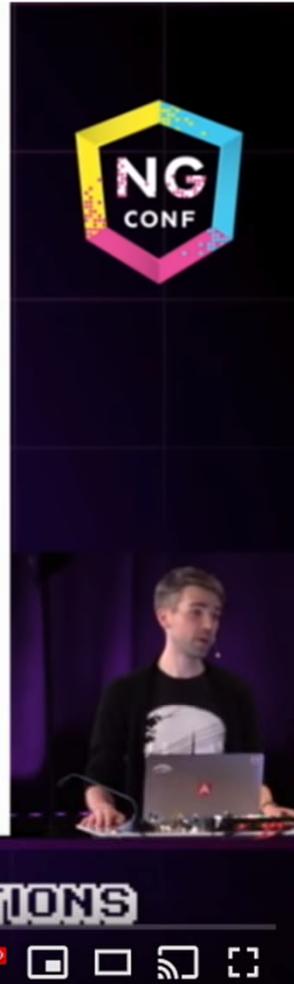
Before

- Googler on Angular team
- Blogged at ysavkin.com



Now

- Co-Founder at [nrwl.io](#)
- Blog at [blog.nrwl.io](#)



 **nrwl**
Narwhal Technologies Inc

nrwl.io



Angular at Large Organizations - Victor Savkin

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

Publishing your library to npm

M

Angular In Depth by ag-Grid

ABOUT SUPPORT US AG-GRID: THE BEST ANGULAR GRID IN THE WORLD

6 Upgrade

The Angular Library Series — Publishing

Publishing your Angular Library to npm

Todd Palmer Follow Aug 29, 2018 · 6 min read



<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 ★

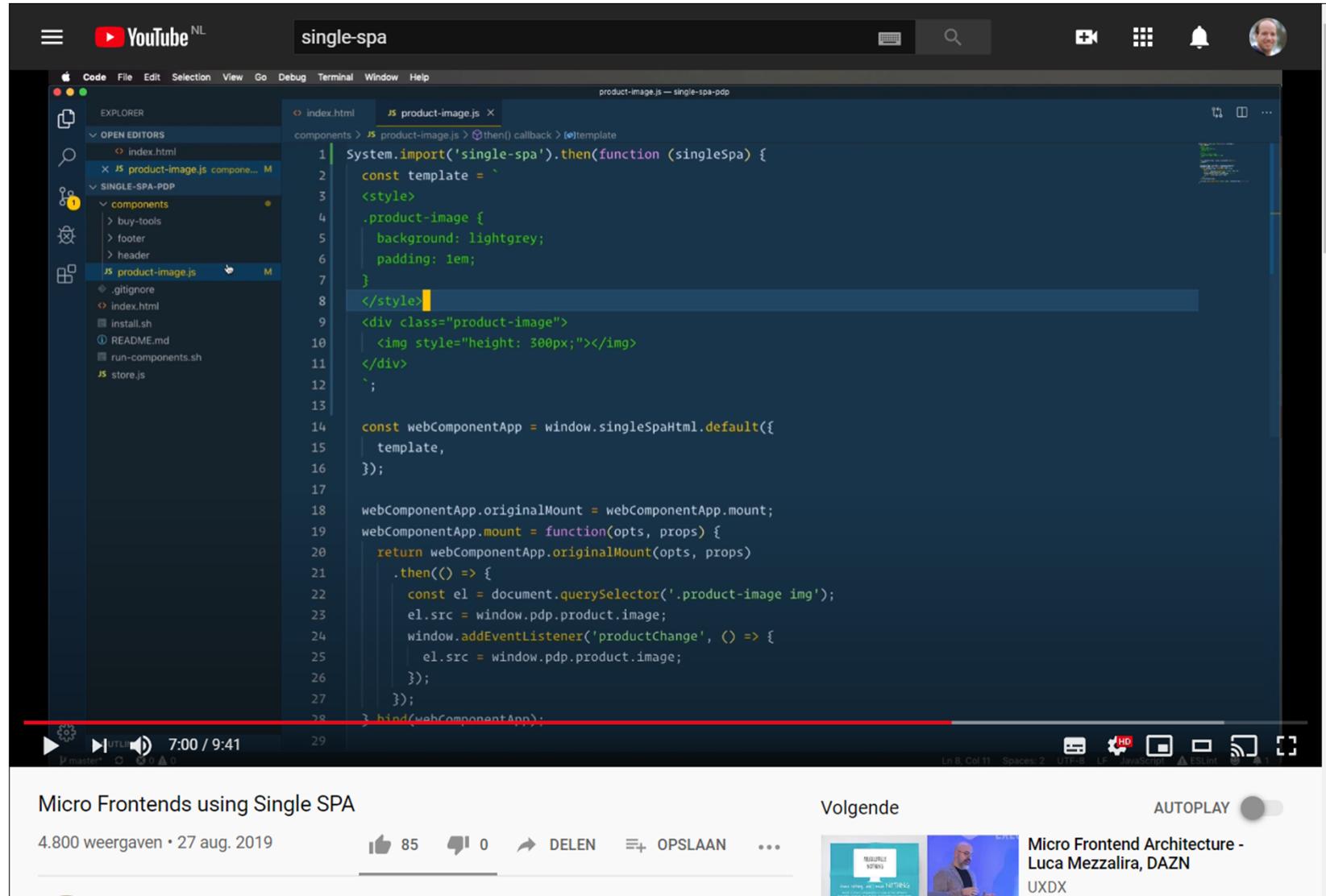
Twitter LinkedIn Facebook Bookmark



A cursor arrow is visible in the bottom right corner of the image.

<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=wU06eTMQ6yI>

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>