ilionx

Global Knowledge®

Angular Advanced Monorepo's - introduction

Peter Kassenaar –
info@kassenaar.com

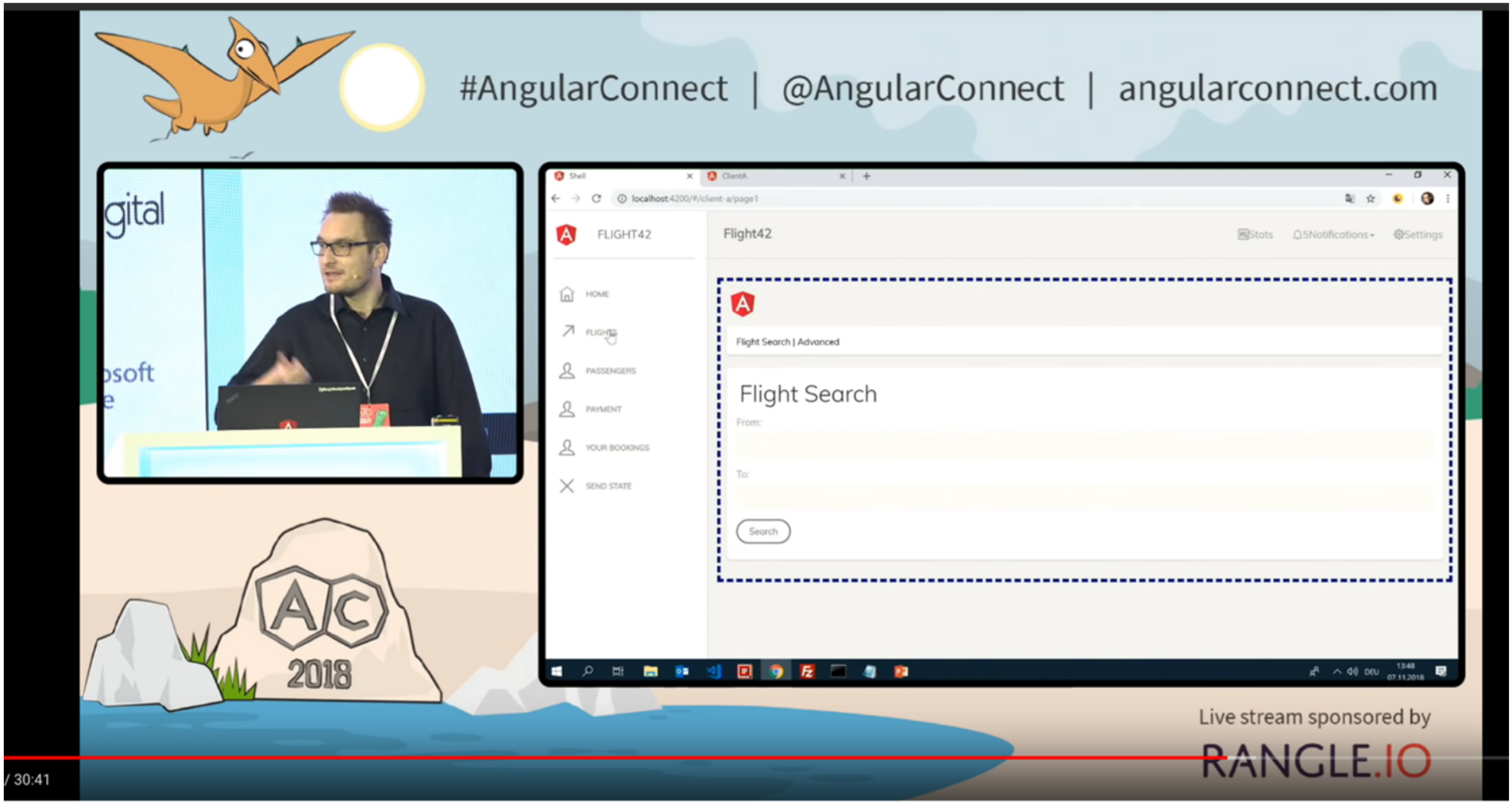
Angular in the Enterprise

- 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

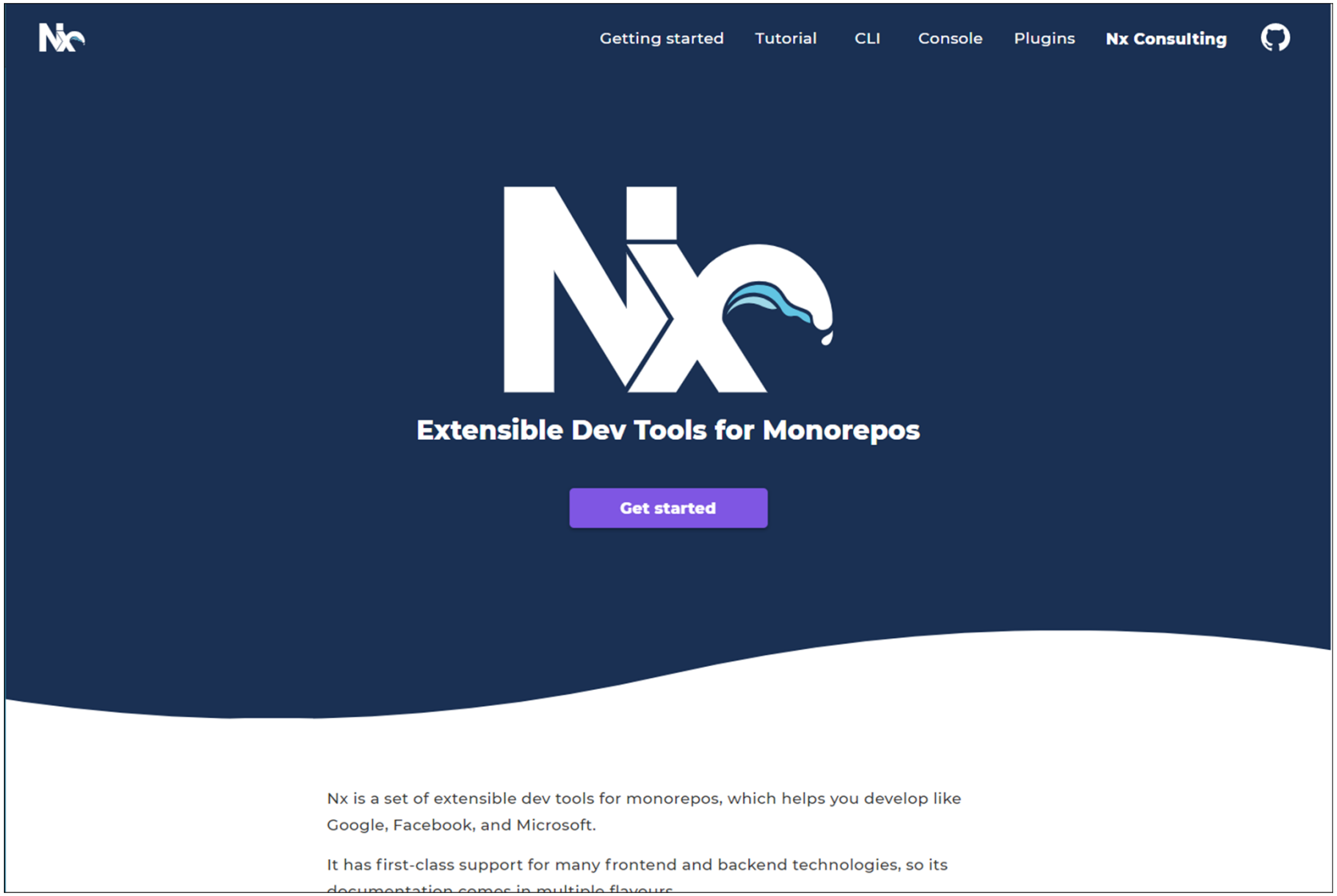
Monorepos

Multiple projects in one solution, optionally sharing code

What is a monorepo

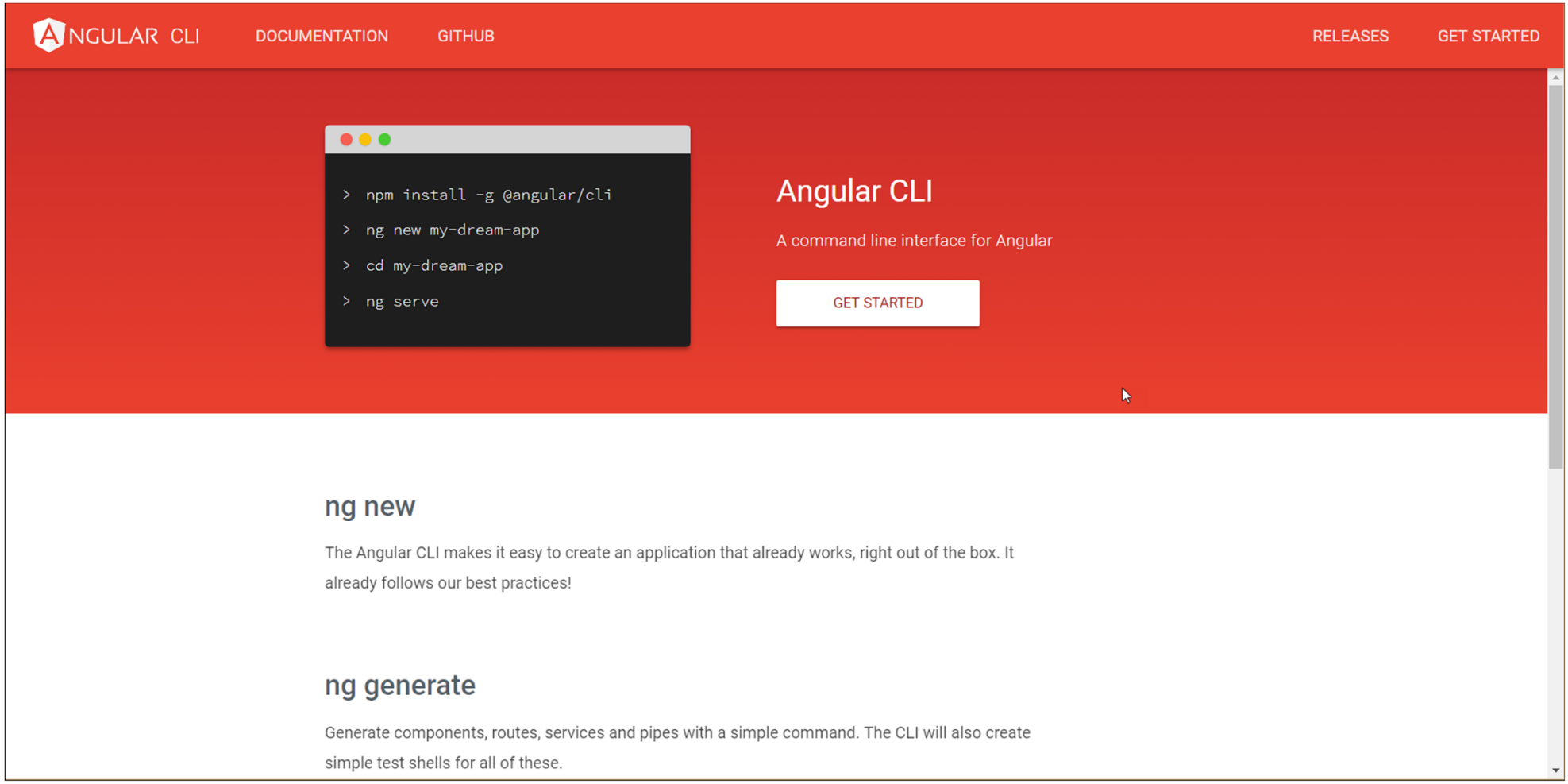
- Often, bigger applications are not only split up into modules, but also in other applications
- Applications generally share a lot of code
  - Sharing components
  - Sharing services
  - Sharing pipes, other logic, etc…
- One (opiniated) solution – create a so called *monorepo*
- There are tools that 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



https://nx.dev/

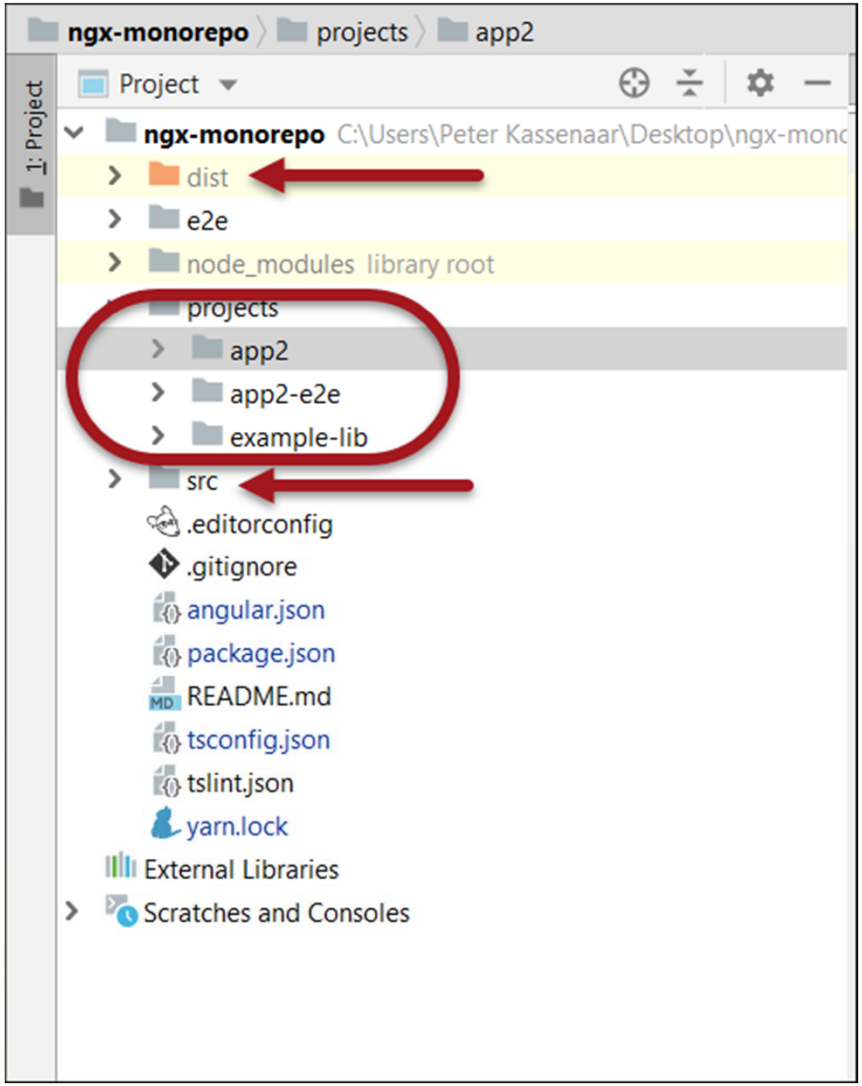
Angular CLI – as of V6.0+



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

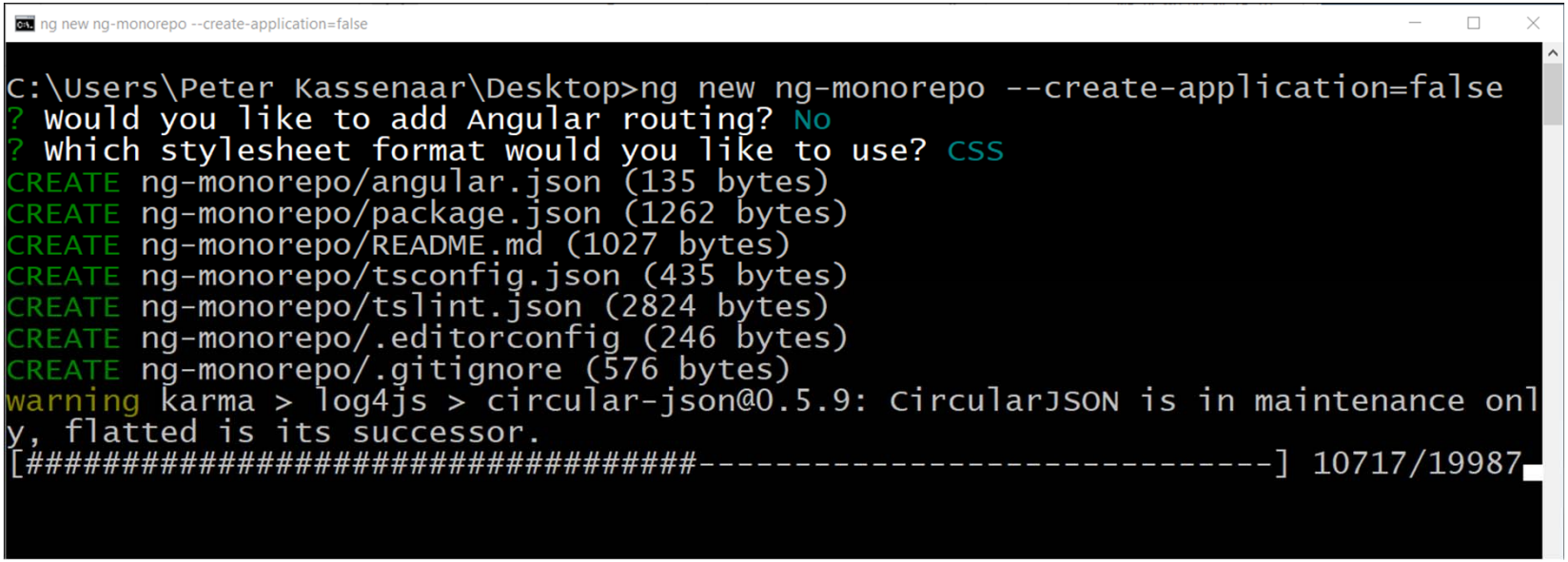


Credits: Angular in Depth

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

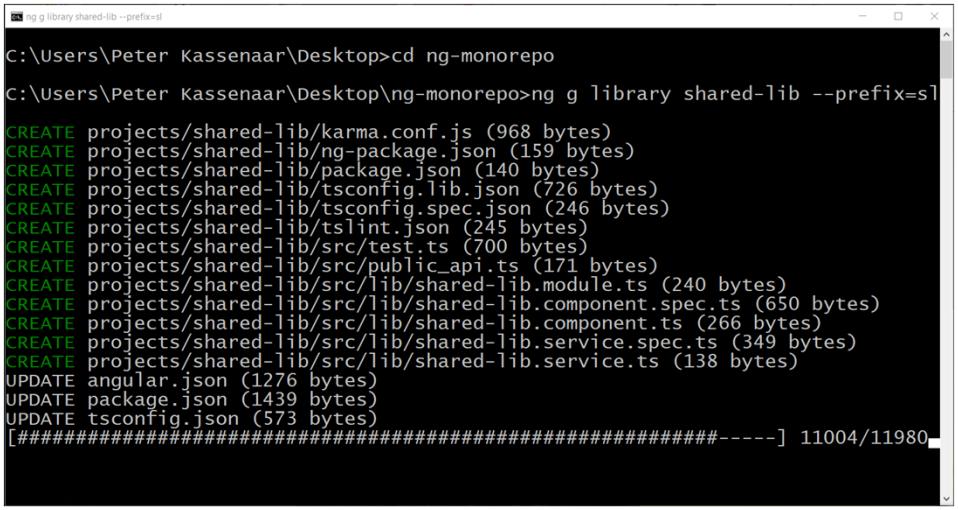
High level overview of the steps

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



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!



angular.json

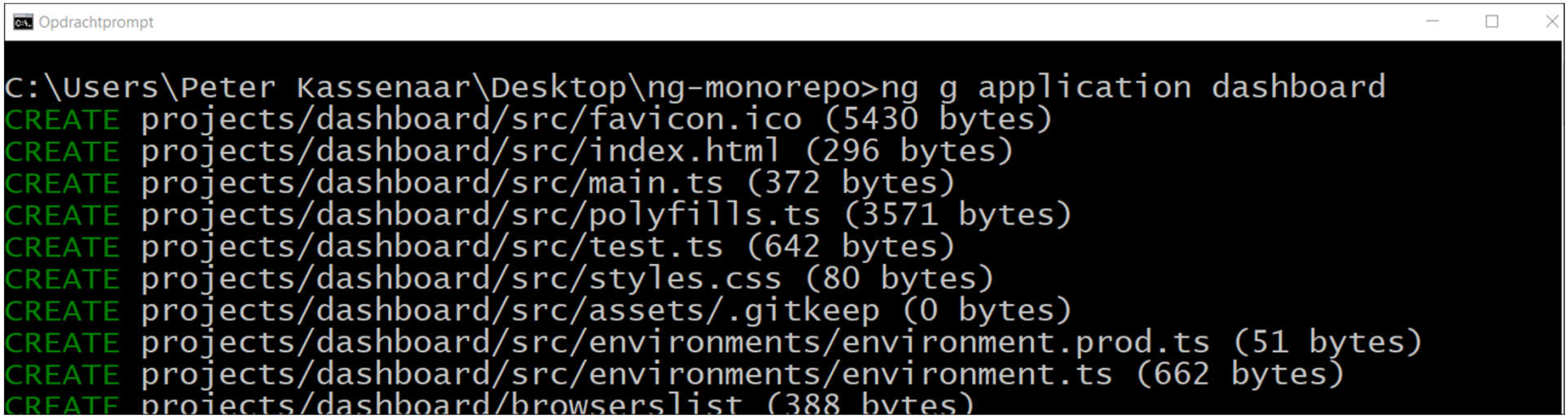
```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "shared-lib": {"root": "projects/shared-lib"...}
  },
  "defaultProject": "shared-lib"
}
```

```
    "lib": [
      "es2018",
      "dom"
    ],
    "paths": {
      "shared-lib": [
        "dist/shared-lib"
      ],
      "shared-lib/*": [
        "dist/shared-lib/*"
      ]
    }
  }
}
```

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

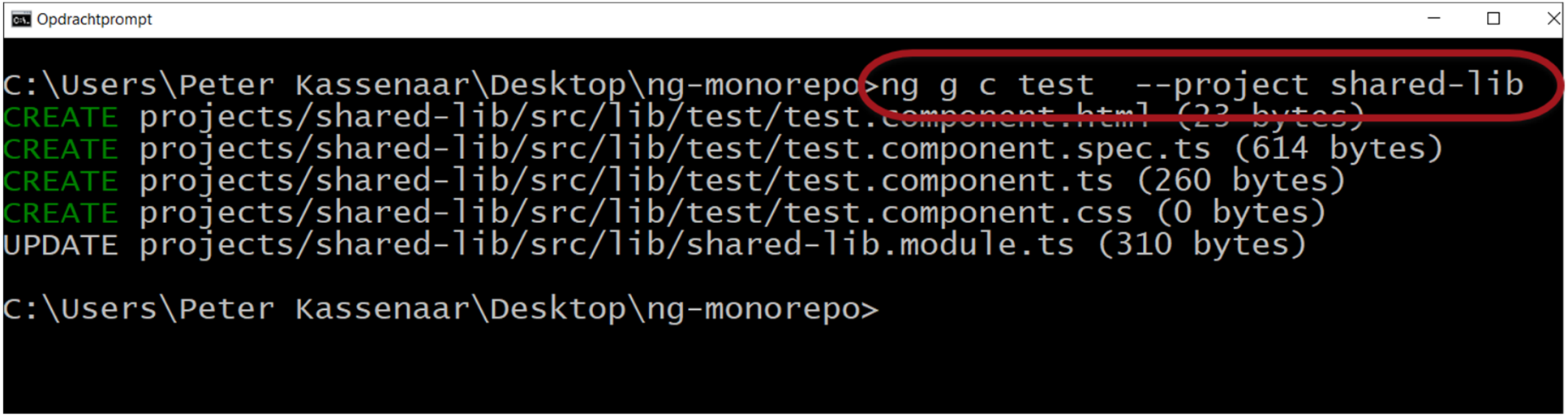
Create first app in the monorepo

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



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



Export the component

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

  `exports: […, TestComponent]`

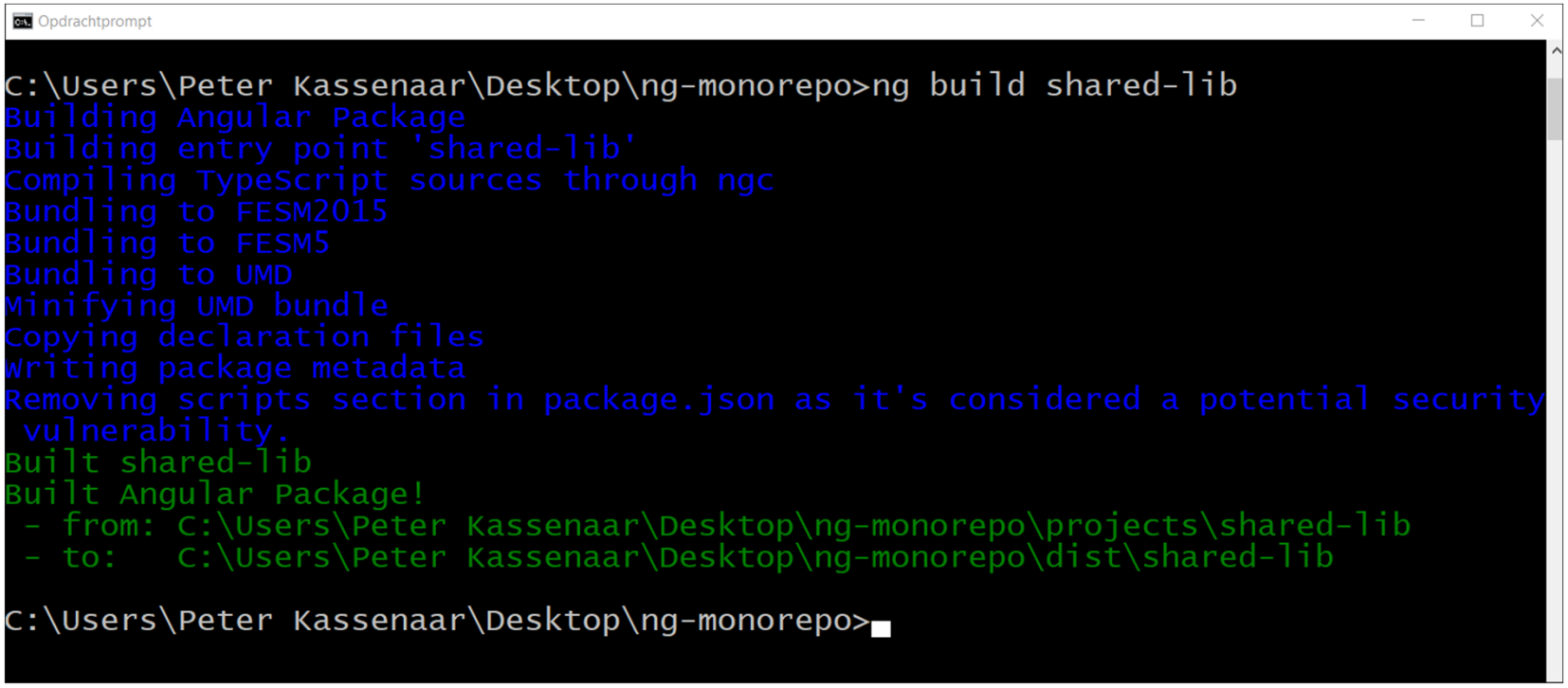
- Update the `public_api.ts` file to make the class available
  - Don’t forget!

```
/*
 * 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';
```

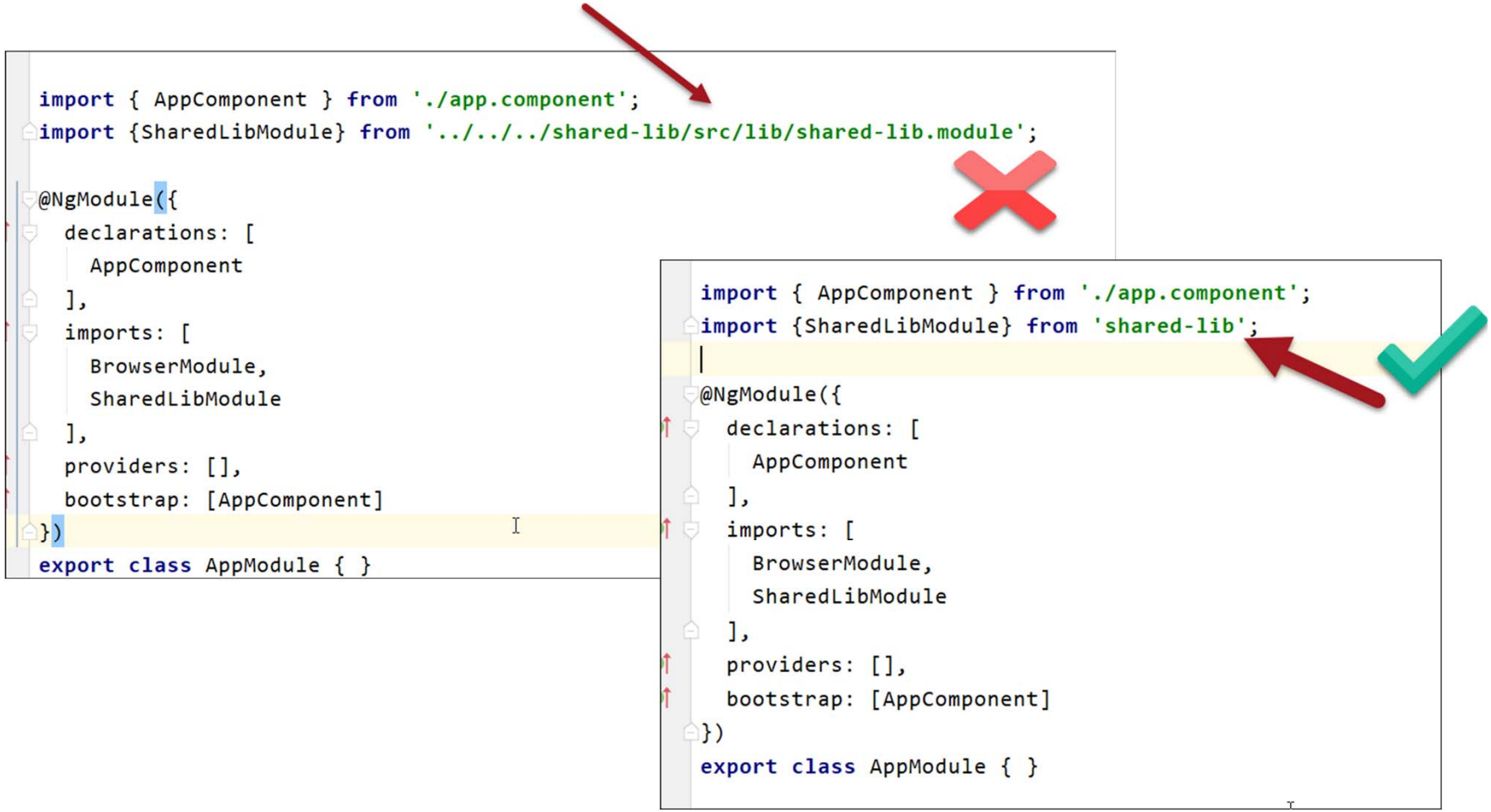
Build the library

- 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!



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 { }
```

```
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

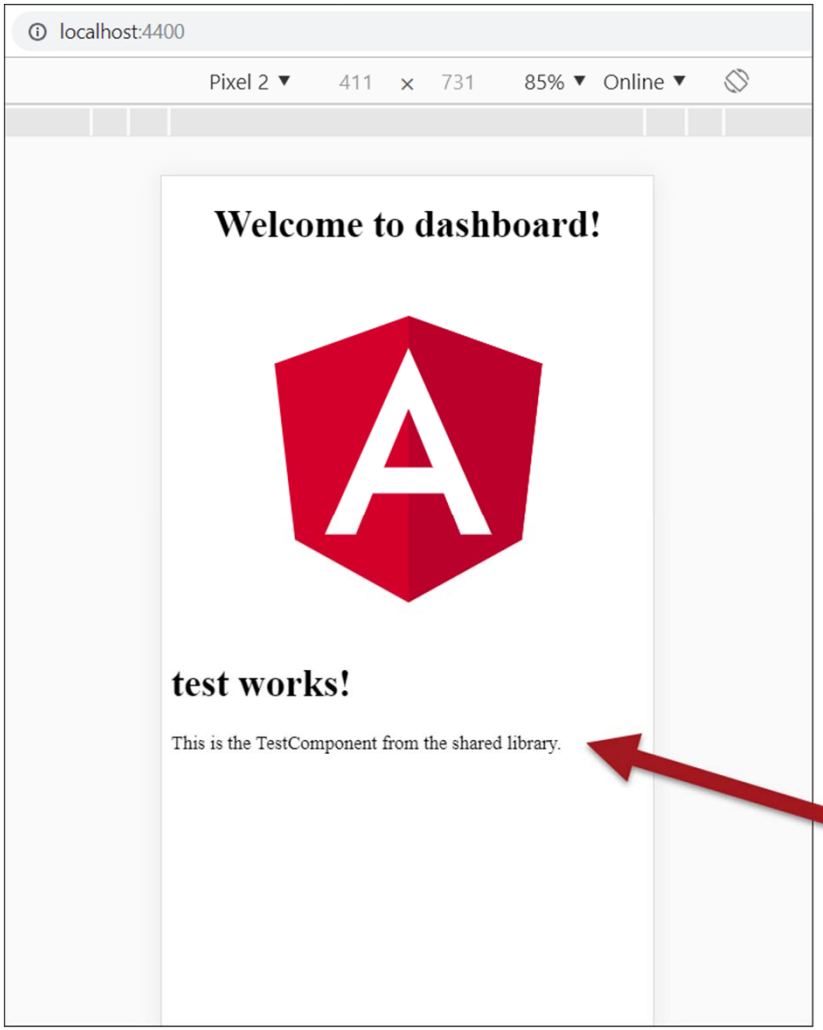
```
<div style="text-align:center">
   …
</div>
<sl-test></sl-test>
```

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"
},
```



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 is only needs to be loaded once!
  - Use a `.forRoot()` method on the module

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

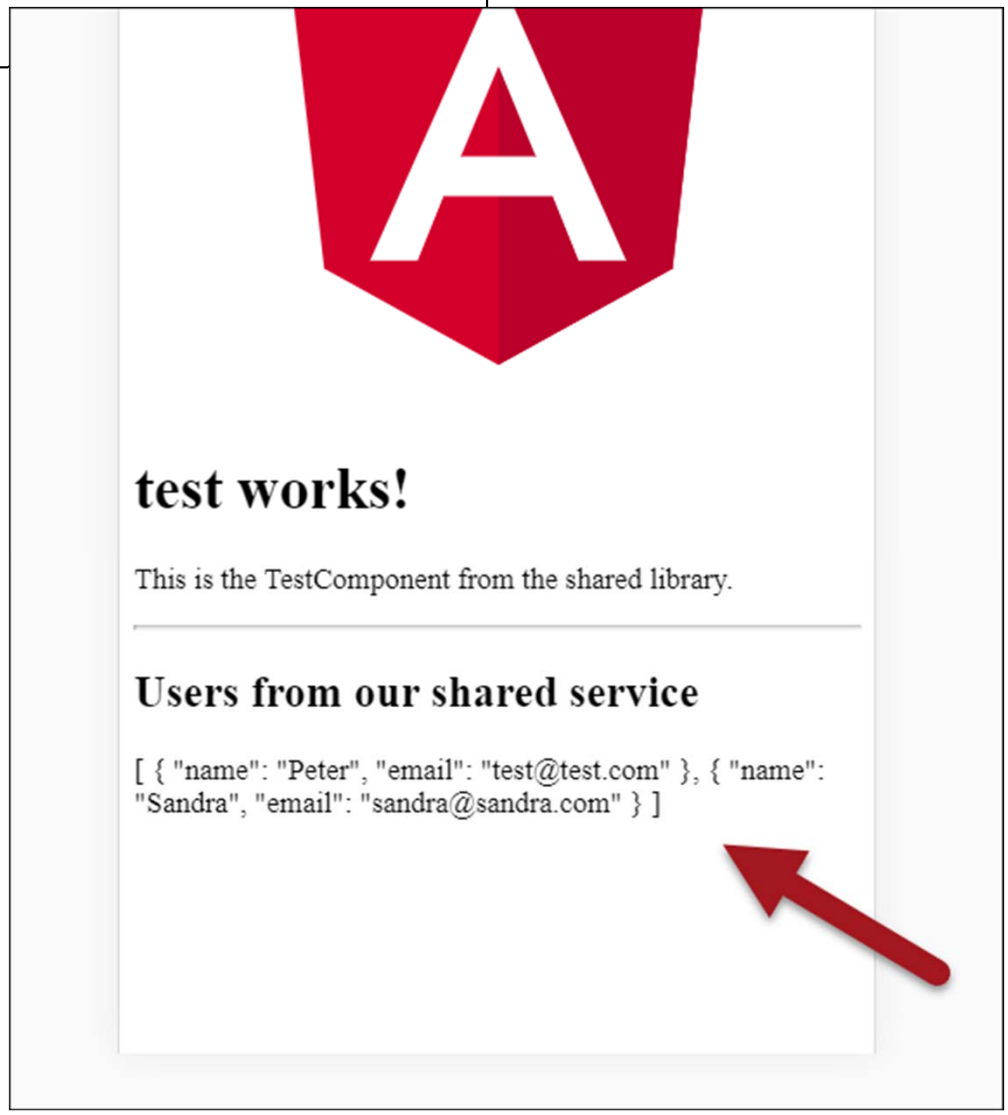
```
import {Component, OnInit} from '@angular/core';
import {User, UserService} from 'shared-lib';

@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();
  }
}
```

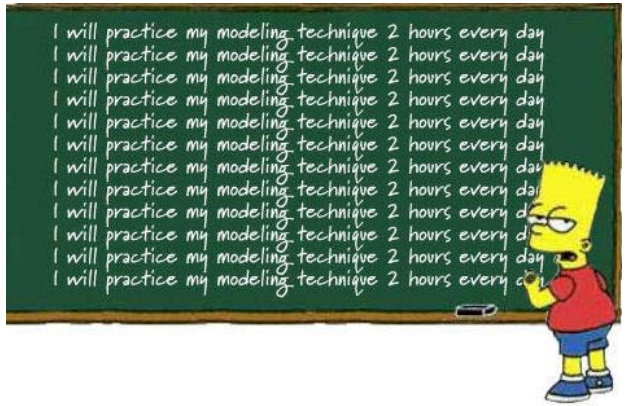
```
<hr>
<h2>Users from our shared service</h2>
{{ users | json }}
```



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



Shorten TypeScript imports

AngularFirebase

PRO Search Lessons

You are receiving a free preview of 3 lessons

Contents

Recent Posts

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 lik
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 nar
`@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

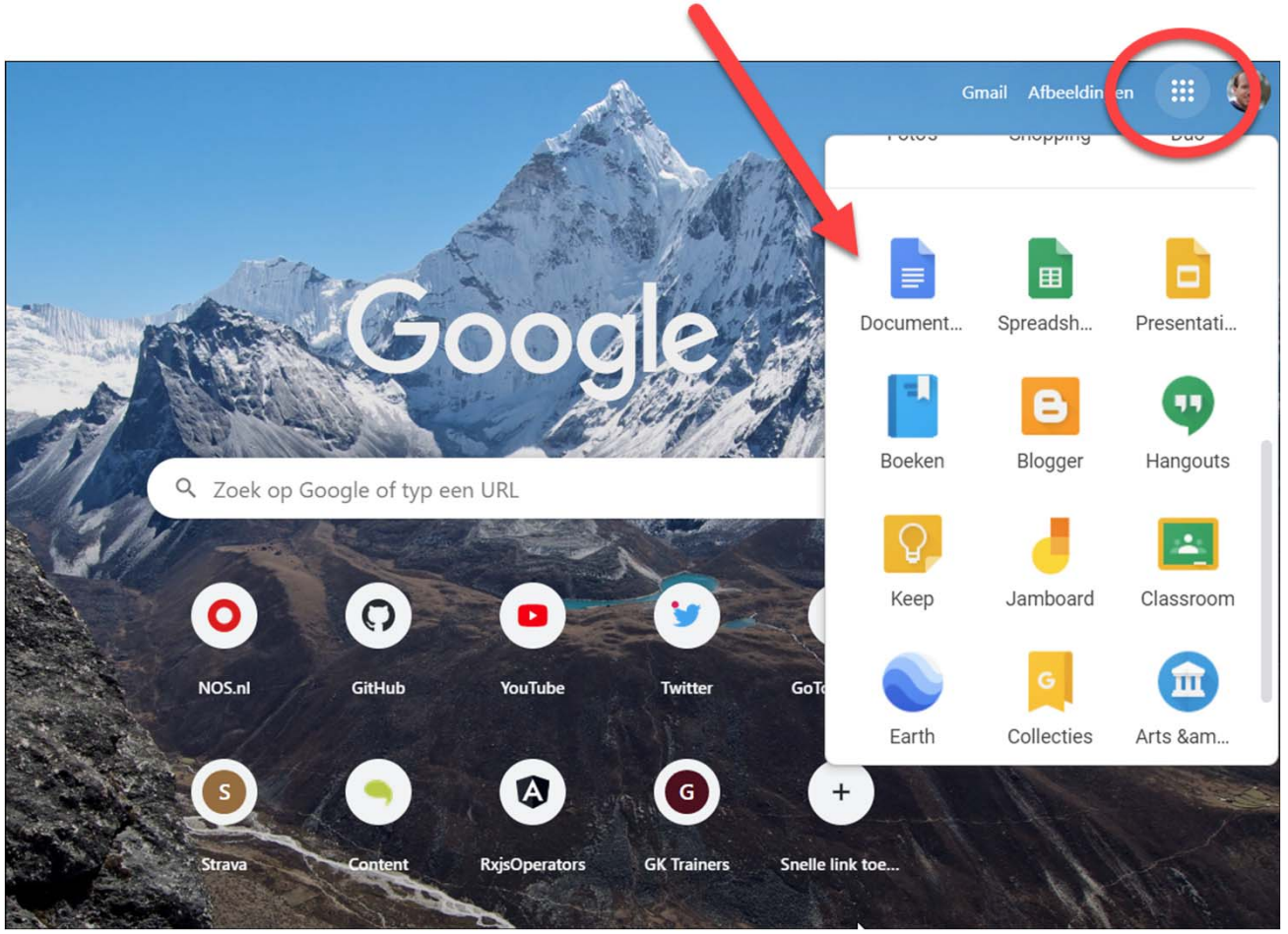
Having multiple, independent apps in your solution.
Possibly using different techniques / frameworks

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

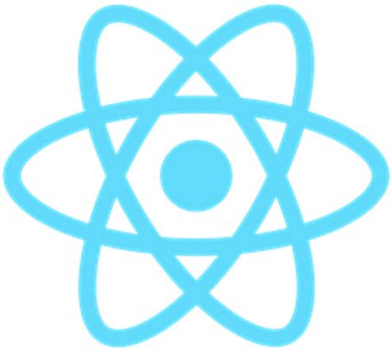
Classical example: Google Suite or Microsoft Office.

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



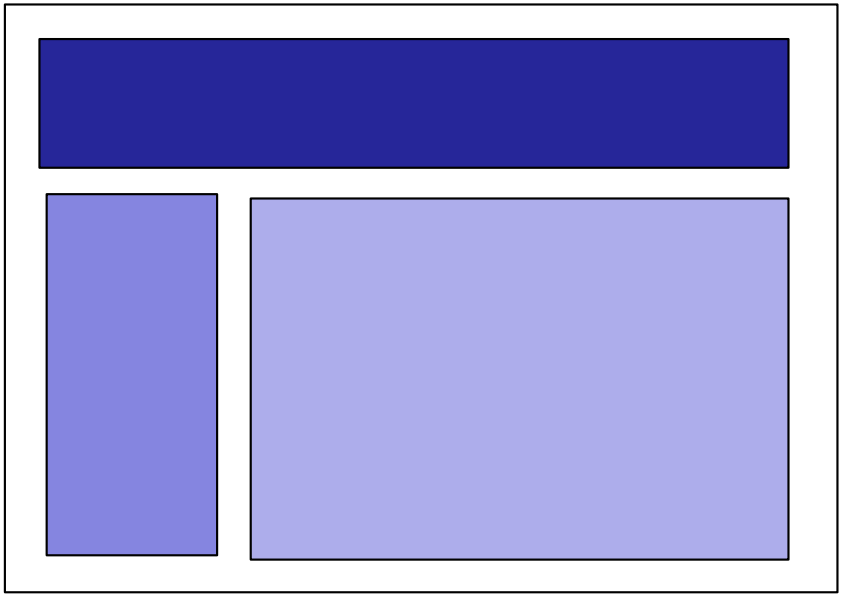
Advantages

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



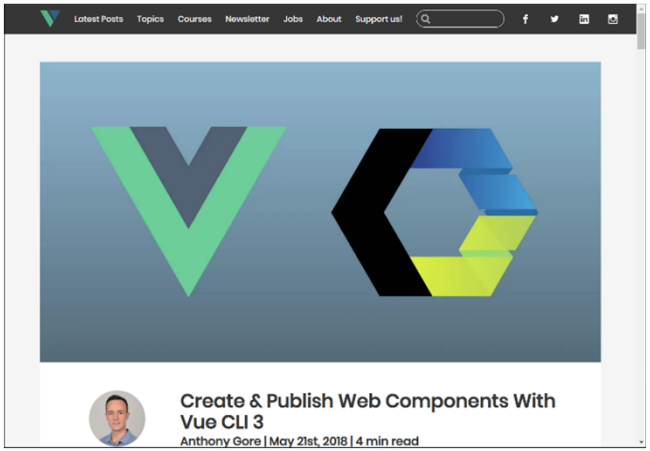
Disadvantages

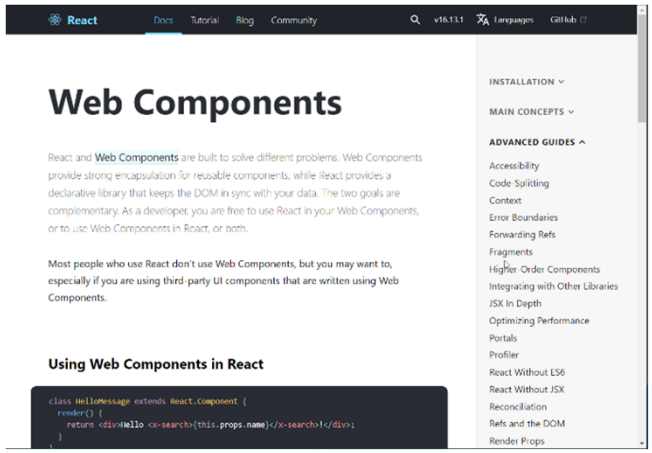
- Loose state when switching between applications
- Load a new application (and overhead) when switching between apps
- Authorization/Authentication is more difficult



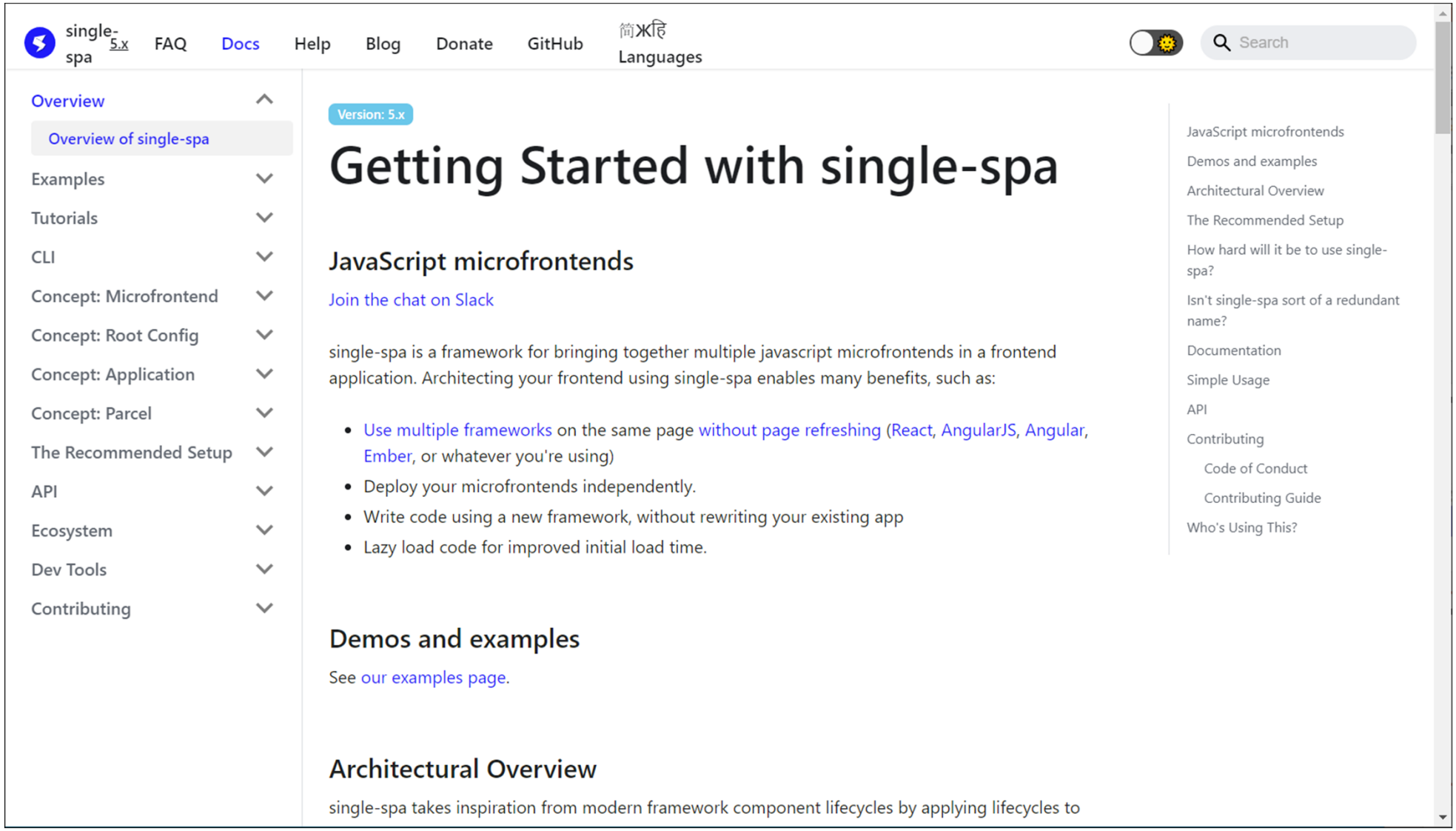
Approach 1 – Do it all yourself

- 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



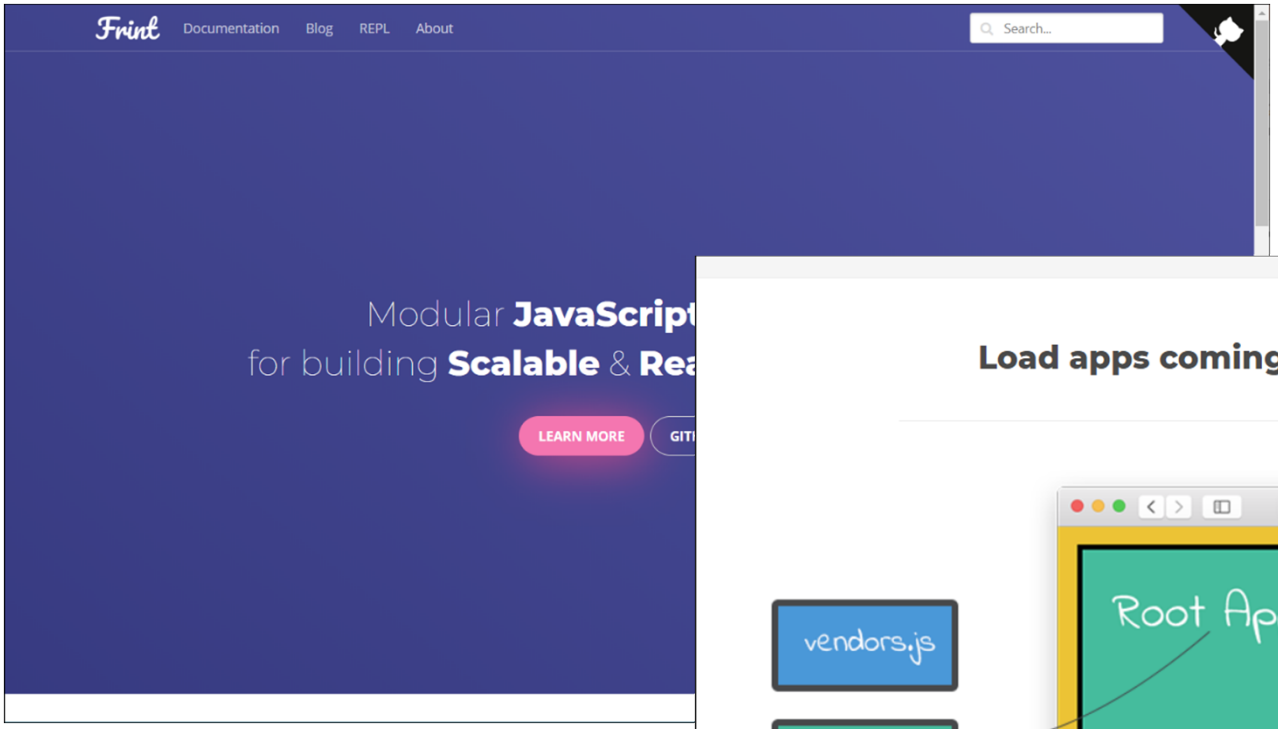


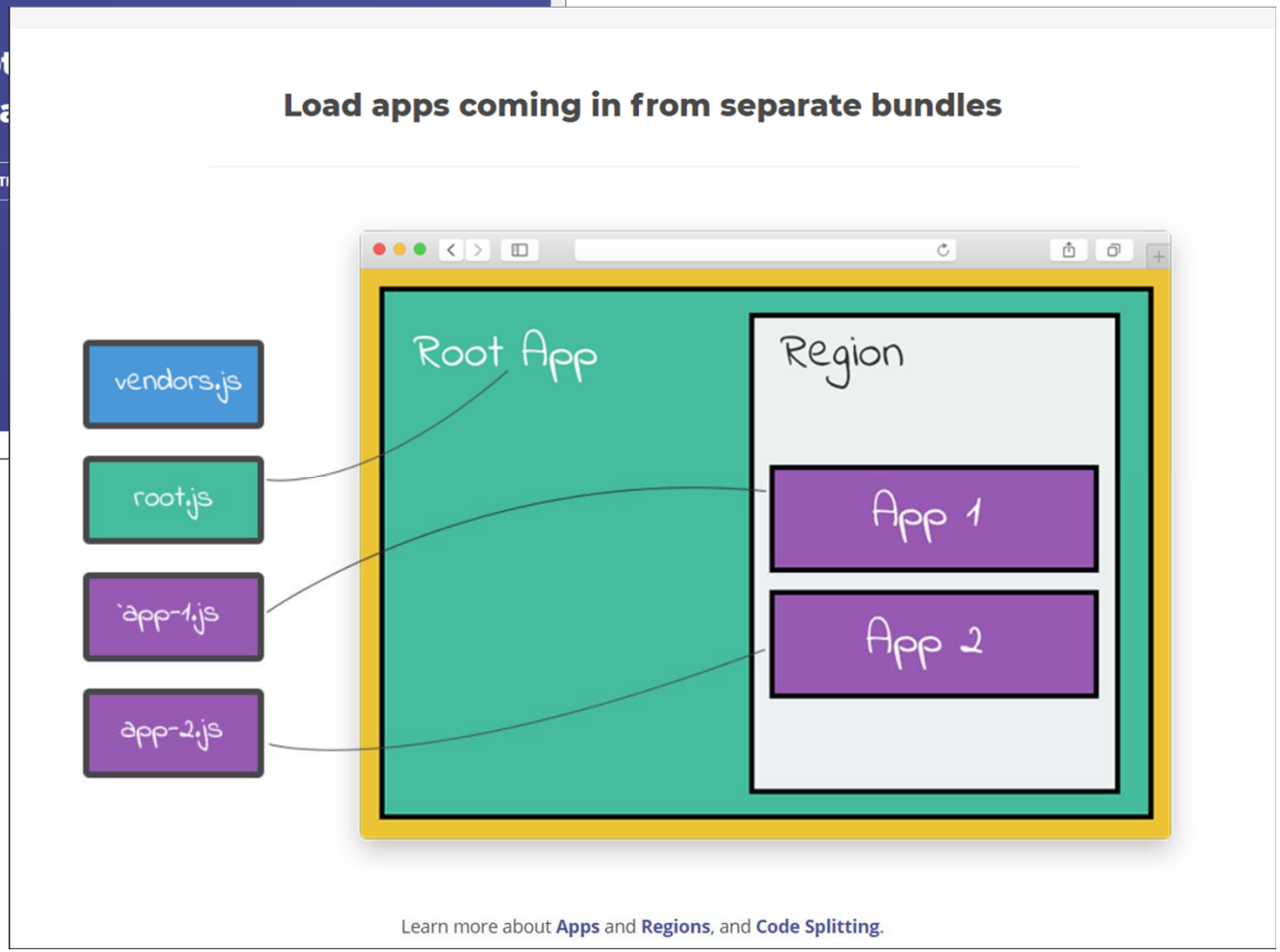
Approach 2 – use a library to create micro-apps



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

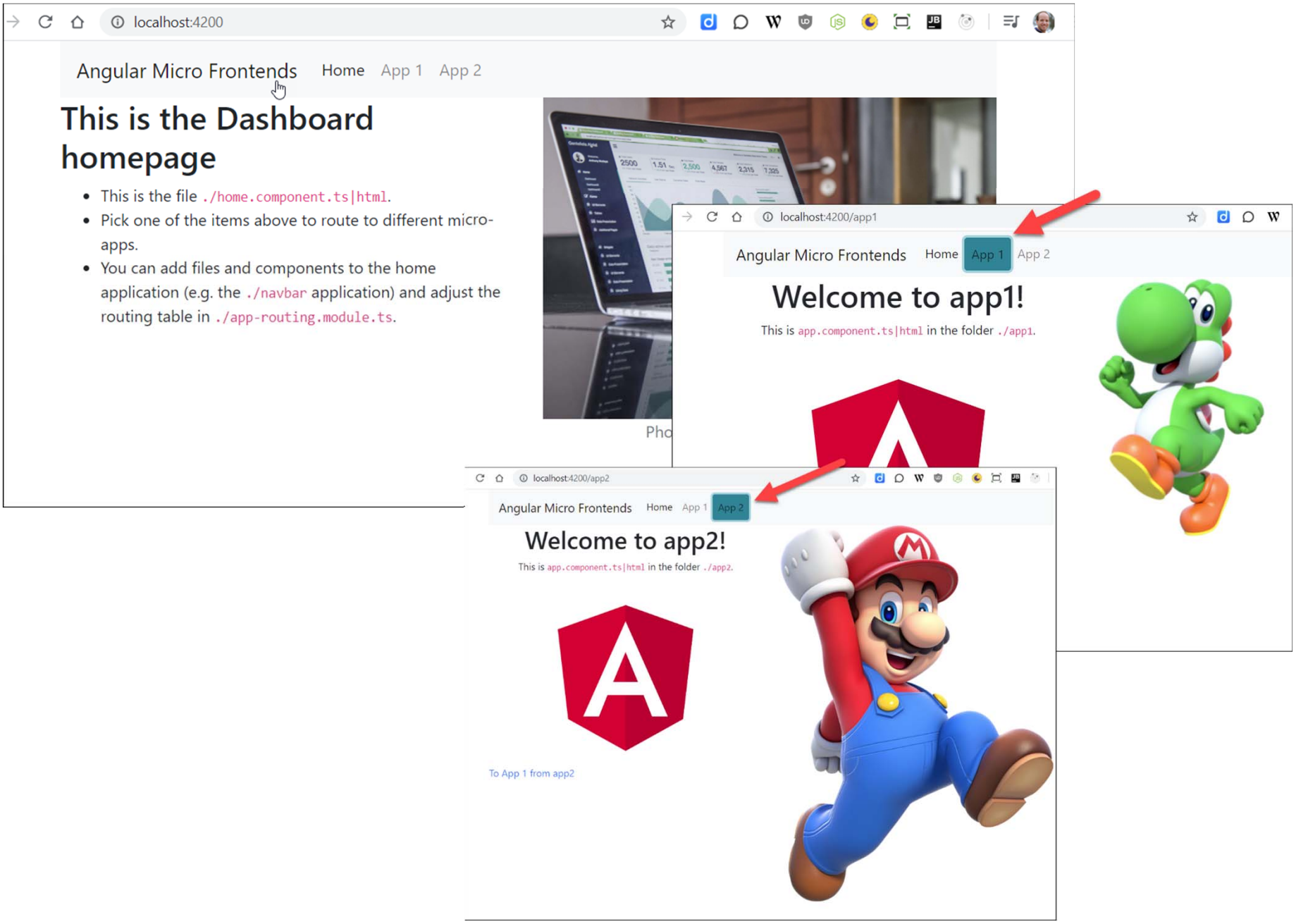
frint.js (for JavaScript and React)





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

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 scrips

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:

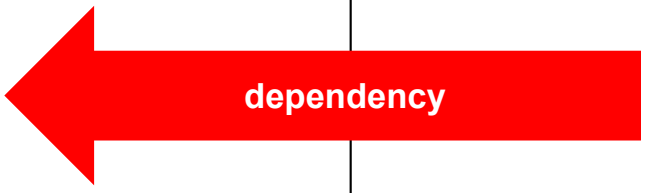
- Each application is an entire SPA itself.
- You can build them with Angular, React, Vue or vanilla JS.
- Each application can respond to url routing events
- Each application must know how to bootstrap, mount, and unmount itself from the DOM.

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

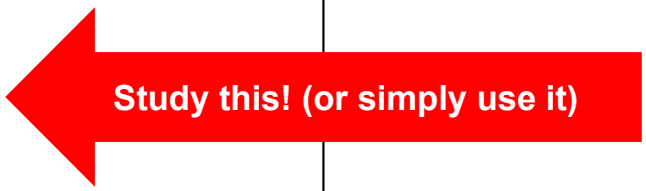
```
// main.single-spa.ts
import { enableProdMode, NgZone } from '@angular/core';

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;
```



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`

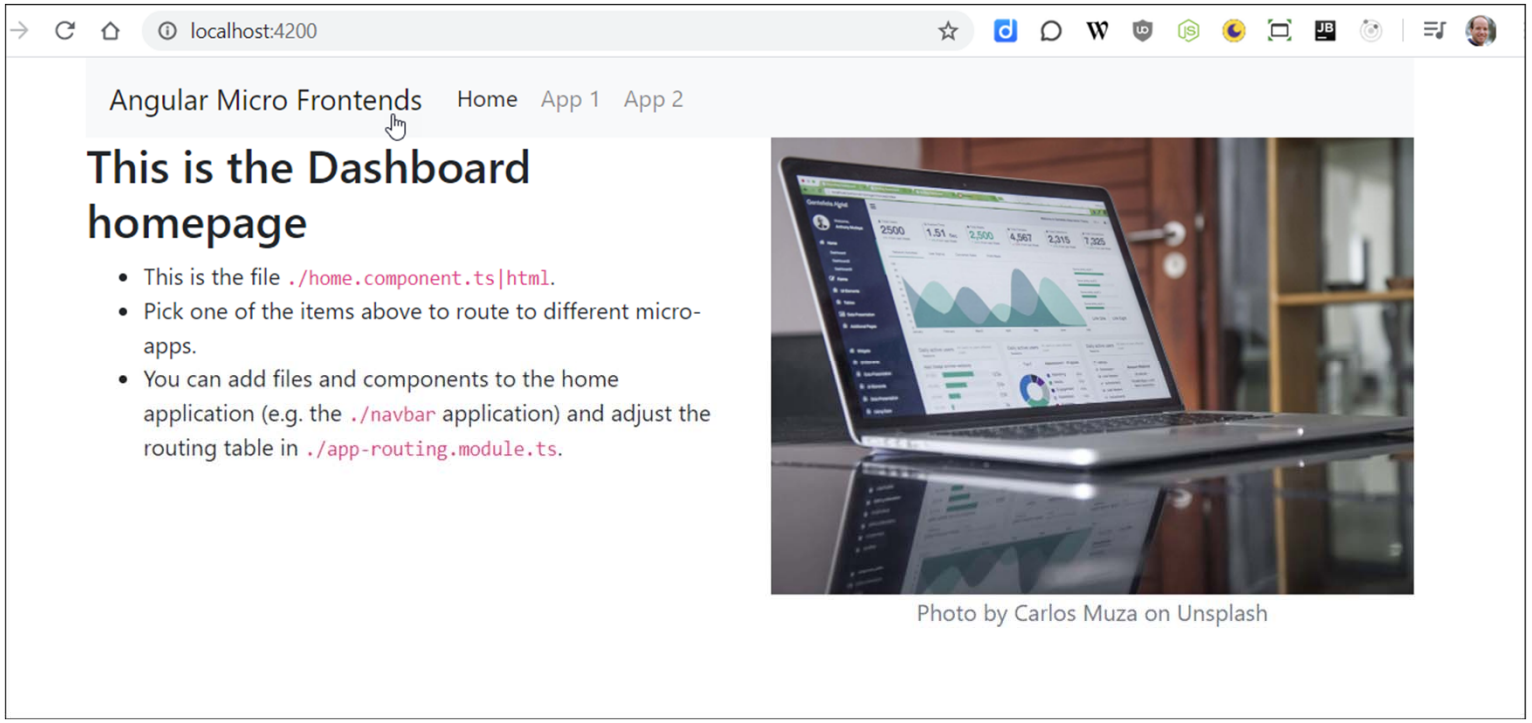


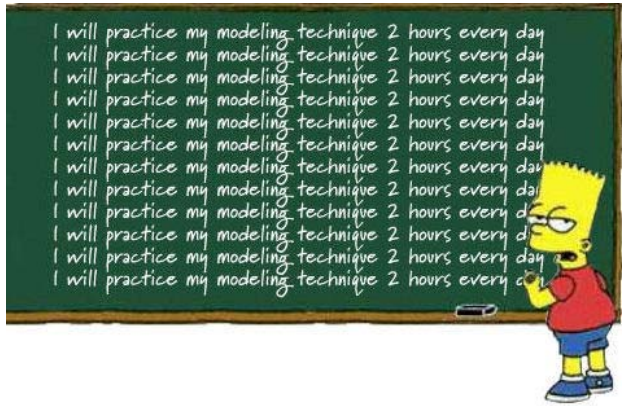
Photo by Carlos Muza on Unsplash

Verdict on single-spa

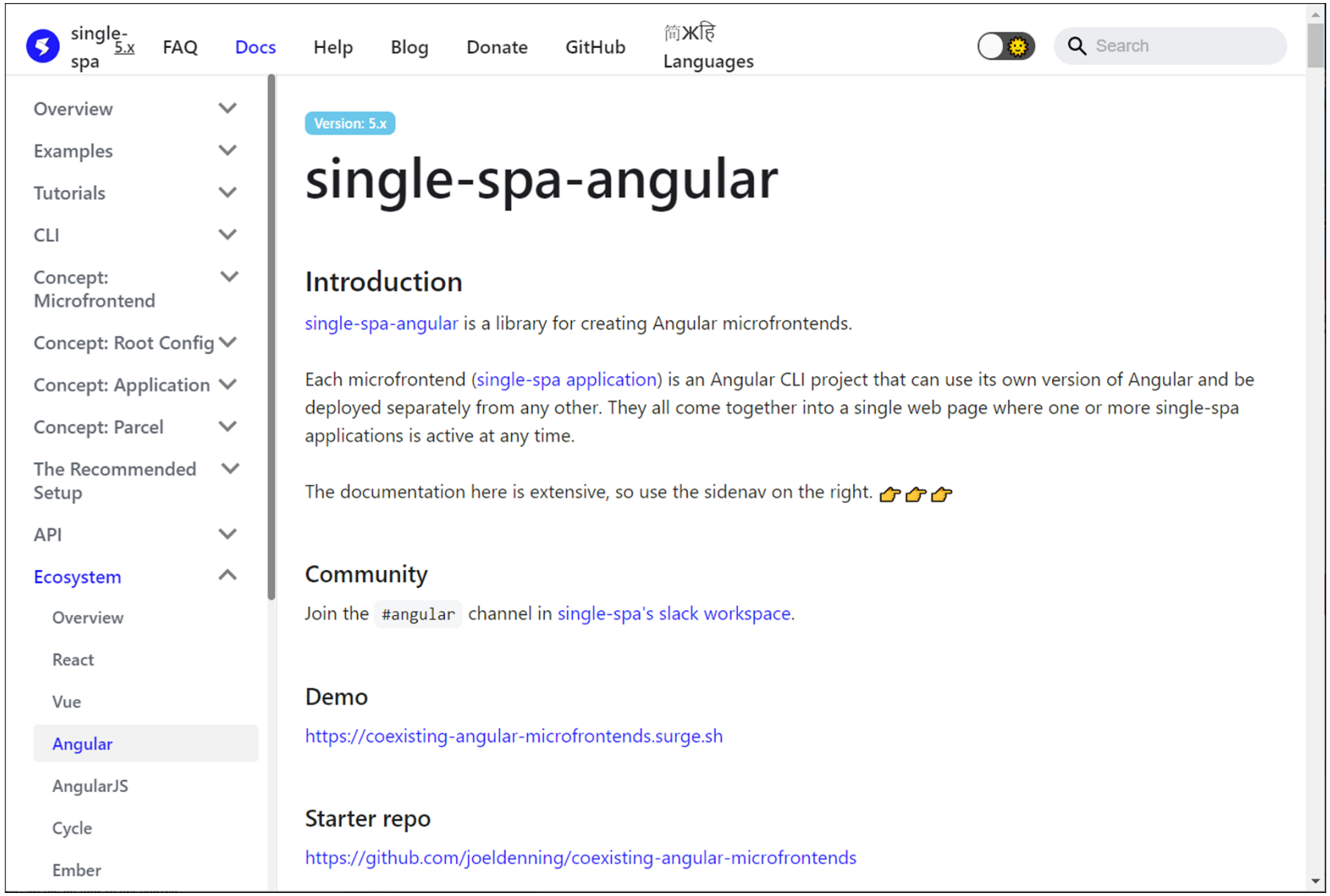
- Pros
  - Builds on proven technology like SystemJS
  - Lots of support for tech stacks, different UI frameworks and so on, so very flexible
  - Supports routing
  - Custom lifecycle events, very robust
- Cons
  - No version control
  - No component registry or viewer (apart from `<import-map-overrides-full>`)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

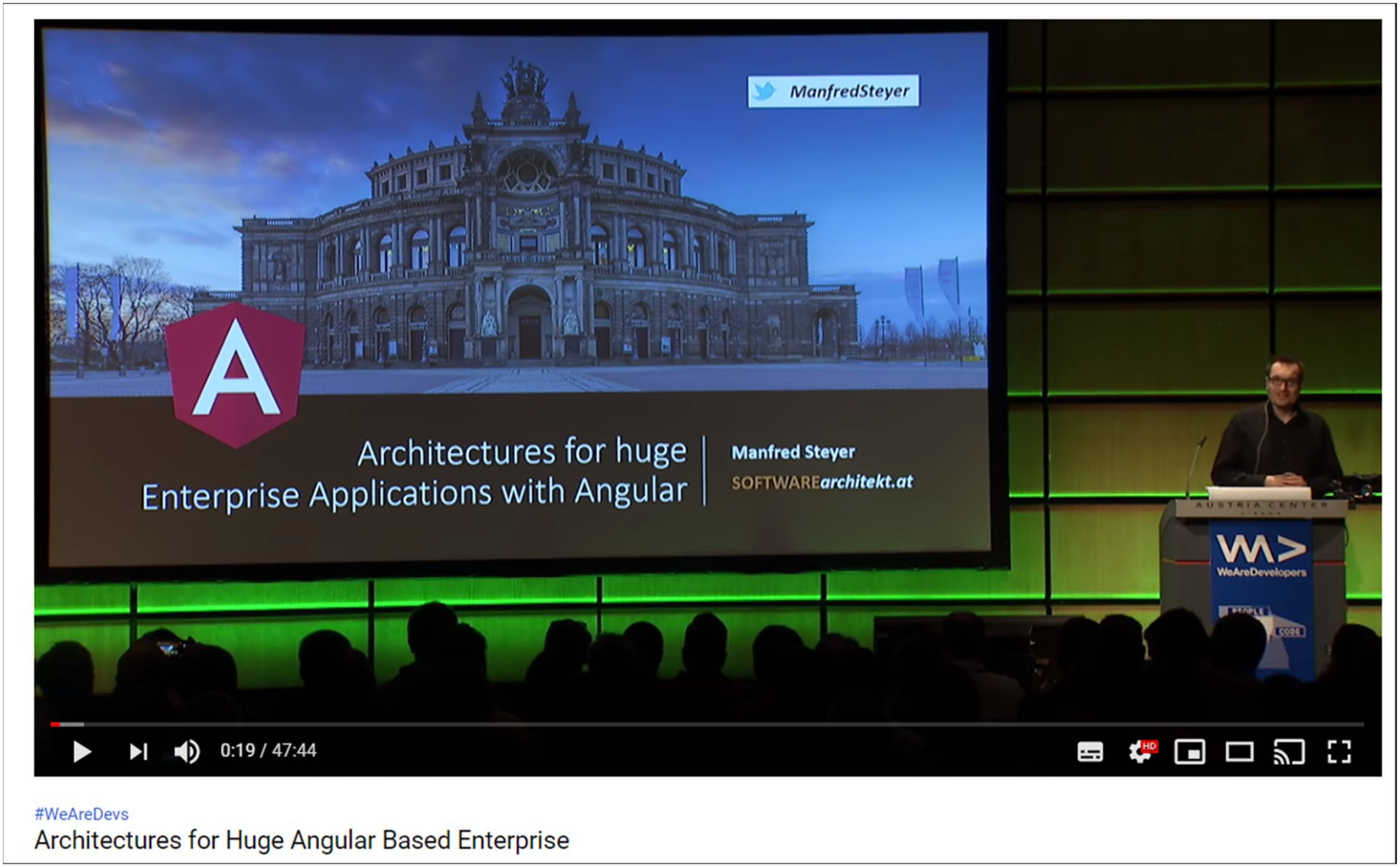


https://single-spa.js.org/docs/ecosystem-angular/

More info

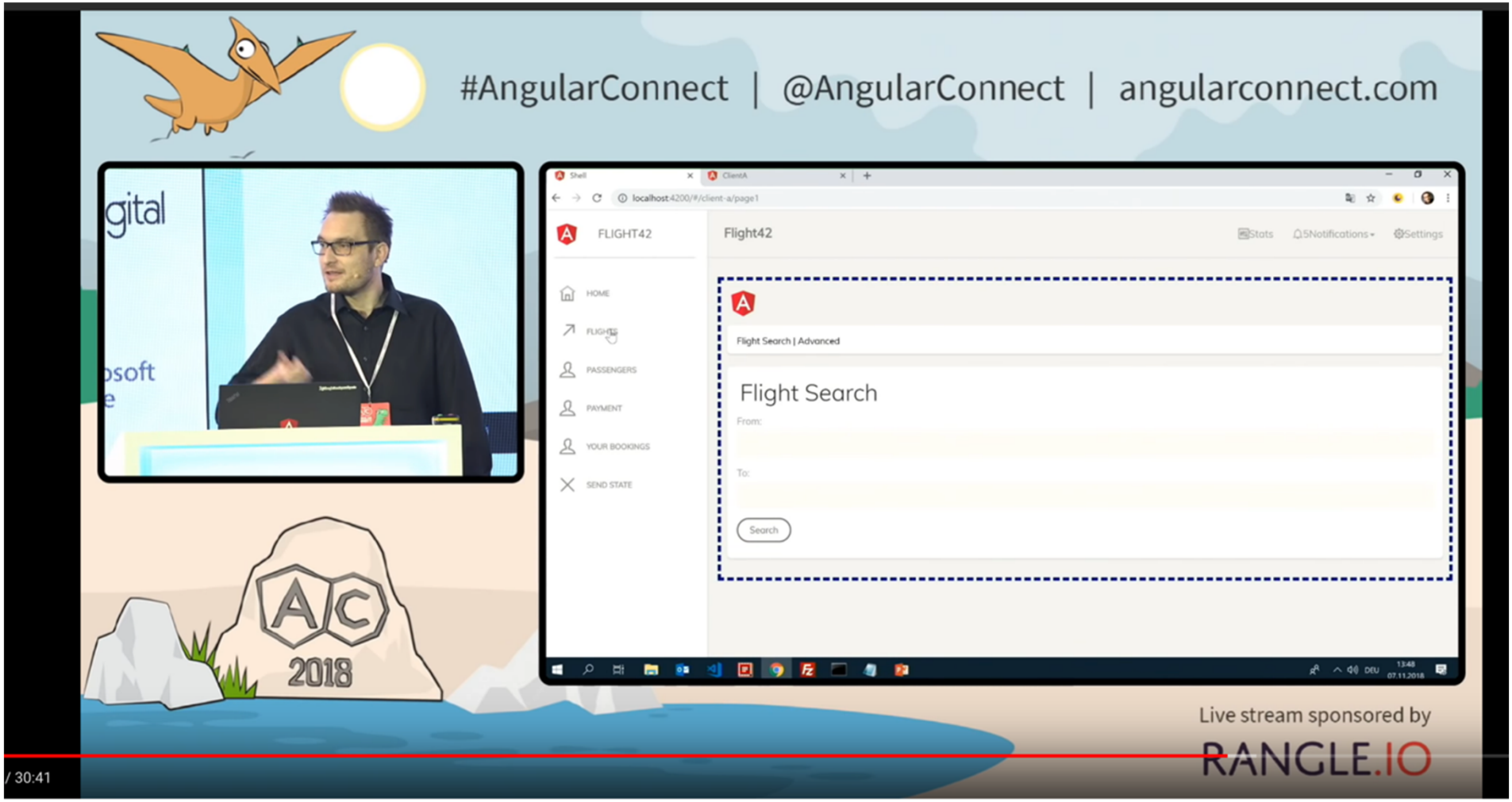
Info on npm packages, monorepo’s and micro apps

Talks on Angular Monorepo's



- https://www.youtube.com/watch?v=q4XmAy6_ucw

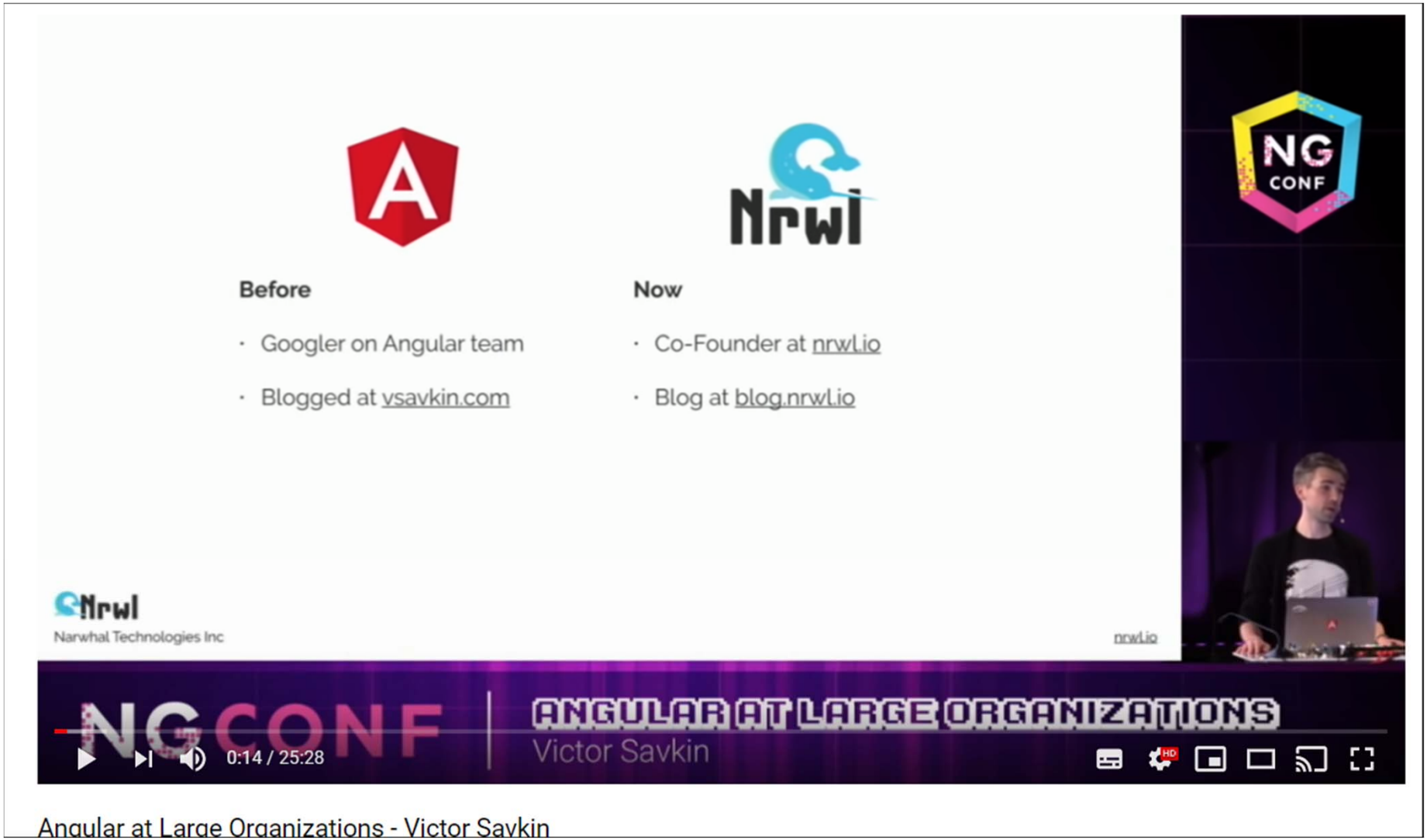
Manfred Steyer – Angular Connect



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

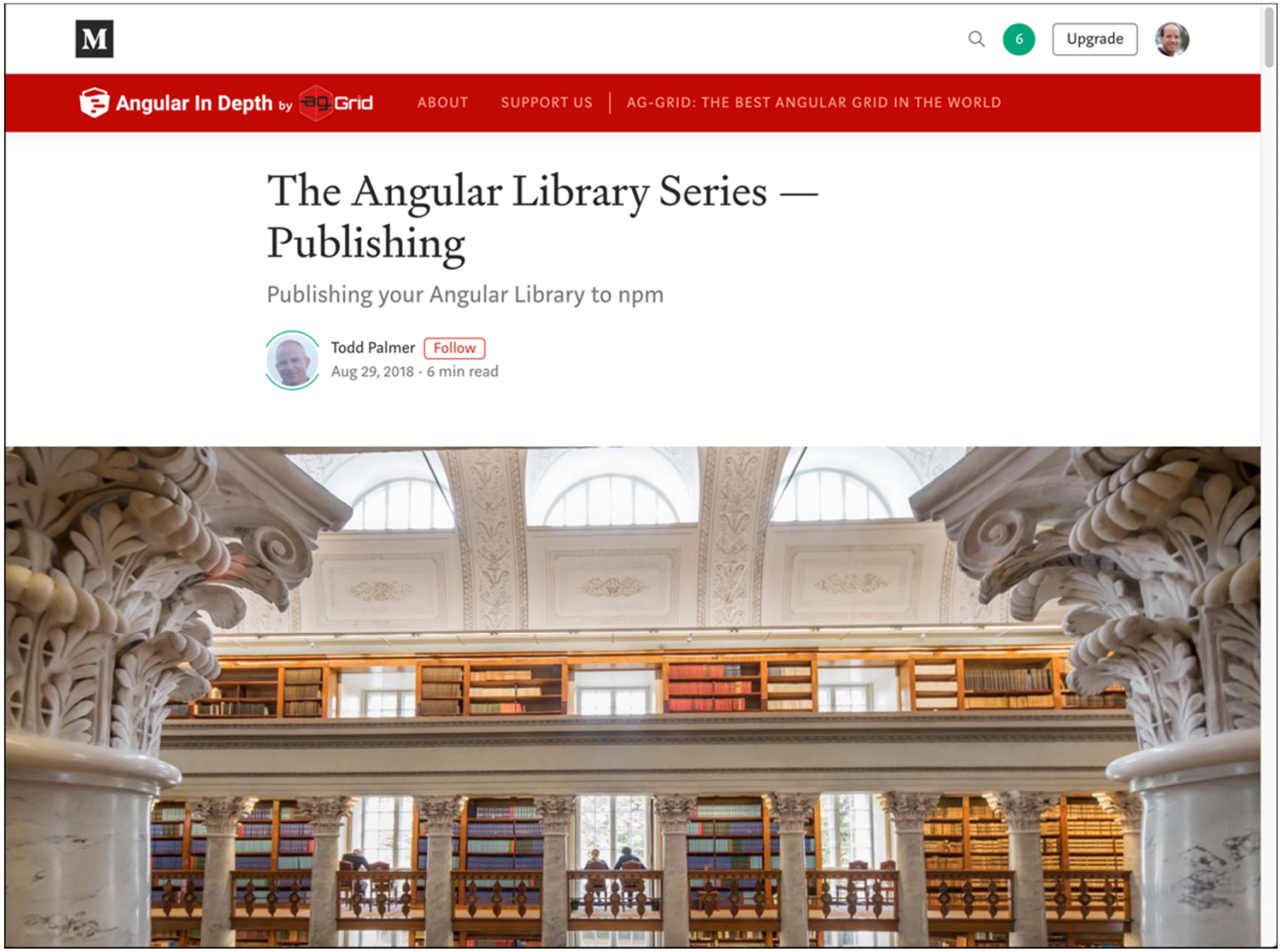
Code: https://github.com/PeterKassenaar/angular-microapp

Victor Savkin – creator of Nx



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

Publishing your library to npm

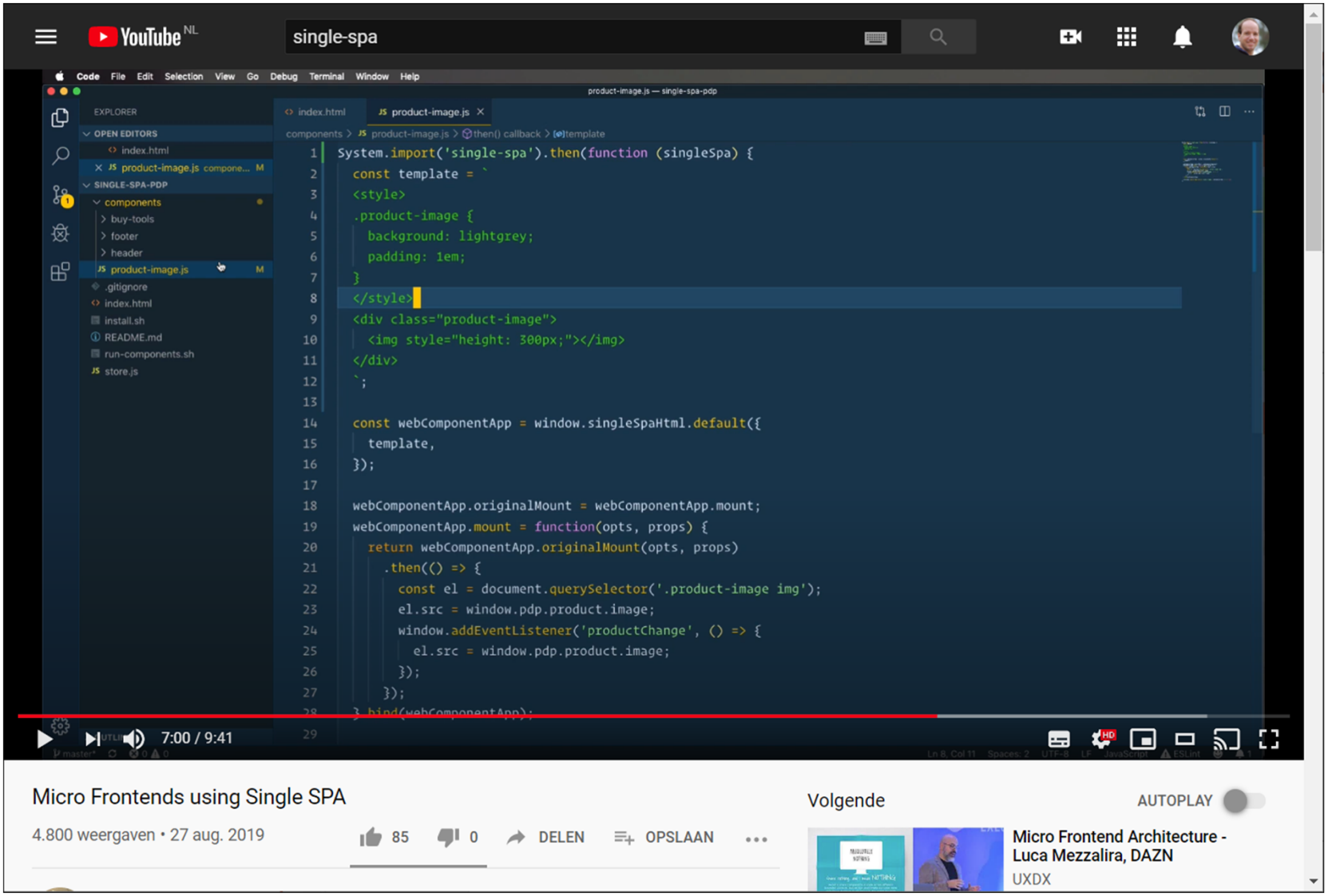


https://blog.angularindepth.com/the-angular-library-series-publishing-ce24bb673275

Implementing micro apps in Angular

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