Tomas Trajan 6.4K Followers About Follow

The Best Way To Architect Your Angular Libraries

Tomas Trajan Sep 15, 2020 · 14 min read ★

Sometimes it's just worth doing it right from start ☺ (📸 by Gabriel Sollmann)

The context

The ideas presented in this article are based on extensive experience from large enterprise environment (100 Angular SPAs and 30+ libs)…

> 😷 Wanna know how we can manage such a large environment without going full crazy 😵 Check out Omniboard! 😉

. . .

> *To be more precise, the initial impulse to explore this topic was creation of a second "component framework" (think internal Angular Material) following this approach and comparing it to the original one which does NOT use the* `ng-packagr` *sub-entries…*

The sub-entries facilitated tree-shaking means that when we import something, eg a `MyOrgDropdownModule` then we're only getting implementation of that sub-entry (and sub-entries it explicitly imports, eg it may use `MyOrgIconModule`) and *nothing else*!

Think 1.5 MB vs 50 KB in such scenarios…

More so, this also works wonders in the context of lazy loaded modules (so not just **main / eager** vs **lazy**) as the sub-entries enable Angular CLI to do

further optimization like extracting a sub-entry into virtual chunk that will only be loaded for particular set lazy loaded features that really use it!

. . .

What we're going to learn

- How to create Angular library implementing clean architecture from scratch (demo project included)
- How to implement sub-entry per feature (and how to simplify the process using `ng-samurai` schematics)
- How to define proper Typescript `"paths"` aliases
- **How this architecture and setup automatically saves us from using incorrect imports or introducing circular dependencies**
- How to analyze our library structure with `madge`
- How to consumer library in other applications
- How to make stuff private

- How to bring dependencies into consumer applications (dependencies vs peer dependencies)
- How to create demo application for a library in local workspace
- How to make this work with `jest`

As we can see it's quite some content so feel free to skip to the section that interests you the most based on your previous experience 😊

> Let's get started!

Step by step guide to create our library from scratch

In this section we're going to create new Angular library project from scratch mostly with the help of **Angular CLI Schematics** *(learn more)* so no worries, there won't be much manual typing 😉

1. Generate new Angular CLI workspace (without default application) `ng new angular-library-architecture-example --createApplication false --prefix my-org`

2. Enter the workspace `cd angular-library-architecture-example`
3. Generate new library in the workspace `ng g library some-lib --prefix my-org`
4. Rename `name` in the nested (lib) `package.json` file to `@my-org/some-lib` as this does not happen automatically when specifying prefix which only works for component and directive selectors
5. Adjust the root `tsconfig.json` file by replacing original `paths` content with the following…

```
"@my-org/some-lib/*": [
  "projects/some-lib/*",
  "projects/some-lib"
],
"@my-org/some-lib": [
  "dist/some-lib/*",
  "dist/some-lib"
]
```

The first entry (with the `@my-org/some-lib/*` is used during the development when referencing sub-entries from each other…

The second entry is used during the build to produce library artifacts that can be consumed by the applications in desired way…

6. Delete the content of the `projects/some-lib/src/lib/` folder and remove content of the root `public-api.ts` file so that it's empty.

> Great! Our initial setup is ready. Let's start creating features using sub-entries!

> *Now we will create our first sub-entry and we will do it manually to learn what it's all about. After that we will use great `ng-samurai` schematics to do it for us instead…*

1. Create `feature-a/` folder in the `projects/some-lib/src/lib/`
2. Create `index.ts`, `package.json` and `public-api.ts` files inside of our new `feature-a/` folder
3. In the `index.ts` file export everything using `export * from './public-api.ts';`

4. The `package.json` file should contain following snippet…

```
{
  "ngPackage": {
    "lib": {
      "entryFile": "public-api.ts",
      "cssUrl": "inline"
    }
  }
}
```

> *Please note that the index.ts and package.json files will have the same content for every created sub-entry…*

The `public-api.ts` will be used to export all the entities (like modules, components, services or directives) that belong to the particular sub-entry.

Currently we do not have such an entity yet so let's create one using standard Angular CLI schematics `ng g s feature-a/a`. This generates `a.service.ts` in the `feature-a/` folder.

Once our service is available, we are ready to export it from the `public-api.ts`

```
export * from './a.service';
```

> As we an see, the sub-entry setup is pretty straight forward, but a bit verbose so in the future we will use `ng-samurai` Angular Schematics collection by Kevin Kreuzer which can scaffold whole sub-entries using a single command!

More sub-entries

Let's install `npm i -D ng-samurai` Angular Schematics collection and use it to generate additional sub-entry for feature B using `ng g ng-samurai:generate-subentry feature-b --gm false --gc false.`

> *The `--gm` and `--gc` flags stand for generate module and generate component respectively so depending on our feature we might want to generate default module and component but we might also skip them for service only sub-entries…*

Sub-entry generated using this command will come without any module or service so let's create new service using `ng g s feature-b/b`.

> ***Tip:** When implementing Angular libraries, we should almost always use `providedIn: 'root'` when creating new services (which is also a default when generating services using Angular CLI schematics). If our library **consisted only of such services** we would NOT need to use sub-entries as such services are perfectly tree-shakeable out of the box.*

That being said, it is very likely that we will need to add some modules (and components / directives) to our library as the requirements evolve and because of that it is always best to start using sub-entries

> straight from the beginning when implementing any non-trivial Angular library!

OK, let's create one more sub-entry for feature C using the approach described above together with its service, but this time we can also create module and component which we can use later when implementing library demo project…

> *Please make sure that we have exported every service / module / component from the appropriate* `public-api.ts` *file of the corresponding sub-entry.*

Top level exports

Once we have our features and their sub-entries ready, we still have to add their exports to top level `projects/some/lib/src/public-api.ts` file.

For that we will use Typescript `paths` aliases that we prepared previously so that the final export will look like `export * from '@my-org/some-lib/src/lib/feature-a';` . We will repeat this for every implemented sub-entry…

. . .

Checkpoint

Our initial library architecture was finished so let's try to build the library using `ng build` command.

We can explore the console output and see that there is an output for every implemented (and exported) sub-entry which will look like this…

```
---------------------------------------------------------------------------
Building entry point '@my-org/some-lib/src/lib/feature-b'
---------------------------------------------------------------------------
Compiling TypeScript sources through ngc
Compiling @angular/common : es2015 as esm2015
Bundling to FESM2015
Bundling to UMD
Minifying UMD bundle
Writing package metadata
Built @my-org/some-lib/src/lib/feature-b
```

Example of a console output when building Angular library implemented using ng-packgaer sub-entries

Let's also have a look in the `dist/` folder which contains generated artifacts.

> *As we can see, the build created separate file for every sub-entry and an additional main file which represents the top level* `public-api.ts` *and re-exports all the sub-entries bundles…*

Files generated by the Angular build process when building library that was implemented using sub-entries

> *This will enable us to import every exported entity from the library root* `@my-org/some-lib` *instead of using deep imports while preserving full tree-shake-ability!*

> Follow me on Twitter because you will get notified about new Angular blog posts and cool frontend stuff!😉

Composing sub-entry logic

Cool, we have our basic structure and the build is up and running so now it's the time to get more realistic…

In practice, it is pretty common that we would like to use some of the implemented features (sub-entries) inside of other library features (sub-entries).

> *A very common example of such use could be implementing* `logging/` *feature which might be used by the consumer application while at the same time it makes perfect sense to use it also inside of other library features like* `interceptors/`, `data-access/` *or* `tracing/` *…*

With our current setup we can simulate this by injecting service B in the service A. To do that we can use constructor injection and add

`constructor(private b: BService) {}` in the `a.service.ts` file.

Depending on the editor we're currently using it might resolve corresponding import automatically but we could always just do it ourselves and the correct import is…

```
import { BService } from '@my-org/some-lib/src/lib/feature-b';
```

We're referencing a service from another sub-entry and because of that we're using Typescript path alias instead of a relative import.

Let's try to build it using `ng build` and everything should work just fine!

> Now is the time to start exploring all the architectural benefits that we're getting for free just because we're using sub-entry approach!

Circular dependencies between sub-entries

Previously we were injecting service B in the service A so let's try to do the opposite and inject service A in the B to create a circular dependency and try to build the project again using `ng build` .

🅰➡🅱➡🅰

We're going to see an error in the build output…

Our build failed because we introduced circular dependency between our sub-entries

This is a very nice and helpful error which should help us pinpoint the real cause of our problem in no time. Unfortunately this is not the case under all the possible circumstances…

Let's say we would introduce circular dependency through feature C

🅰➡🅱➡©➡🅰 (such a letdown from emojis, no nice **C** 😿 is available)

Depending on our version of Angular CLI we might get nice descriptive error, Out of Memory crash or even nothing (eg build fails silently and does

not produce any Javascript files in the `dist/` folder, room for an improvement ?)

> *Any such occurrence signifies that we have introduced **circular dependency** and hence going in the direction of **tangled hard to maintain code base where everything depends on everything!** This represents a great opportunity to prevent it from happening!*

Let's fix our problem by reverting back to 🅰➡🅱➡© and it will work just fine! To do that, we have to remove injection of service A from service C and rebuild our library using `ng build`.

> *You might have noticed that the sub-entries in the build log are ordered based on the dependency graph so that we first build our leaf sub-entry (feature C) all the way up to library root `public-api`.ts.*

. . .

Exploring the dependency graph

We have been speaking about the circular dependencies and a dependency graph in general. We have also provided simple visualization with the help of an emoji based dependency chains but that doesn't really scale for any real life project…

> Luckily there is a great open source tool called `madge` which can generate dependency graph as a nicely formatted picture!

Beware, it may be a bit hard to install it depending on your platform, cough Windows cough 😄…

Once we installed `madge` globally using `npm i -g madge` (and also all the necessary environment dependencies) we can run it using `madge projects/some-lib/src/public-api.ts --ts-config tsconfig.json --image graph.png` and the result should look something like this…

If we reverted back to the circular dependency described above 🅰➡🅱➡©➡🅰 and re-run the `madge` we're going to get picture that looks a lot different!

Entities marked in red are part of at least one circular dependency chain!

The rule is pretty simple, red = bad = circular dependency!

. . .

Guaranteed correct imports out of the box

Another great feature of this setup is that it forces us to use correct imports when consuming entities across the sub-entry boundaries…

Let's say that we try to import service C in service B. The correct way to do this would be using `import { CService } from '@my-org/some-lib/src/lib/feature-c';` but what if our IDE generates (or we by mistake write) `import { CService } from '../feature-c/c.service';` instead?

> *Such an import means we're trying to import something from other sub-entry using a relative import syntax*

If we tried to build our library using `ng build` we would end up with the following error…

The key part to focus on here is the presence of the `rootDir` keyword.

Whenever we get such an error we can be sure were trying to import something relatively across the sub-entry boundary and fix the problem!

> As we can see, using sub-entries come with many build in checks that ensure we're always doing the right thing!

How to consume library in the Angular applications

Our setup enables us to import everything from the root `@my-org/some-lib` regardless of which sub-entry is the location of the source file of given service, component or module.

This can be combined with a custom `import-blacklist` Tslint rule to prevent deep imports from our libraries eg `@my-org/some-lib/(?!testing).*.`

> *This will ban every deep import from our library besides `@my-org/some-lib/testing` which can be a great starting point to guarantee that you are consuming your library in a correct way!*

With all this in place we should be able to use something like service A using `import { AService } from '@my-org/some-lib';`.

How to make stuff private

One of the requirements to make this setup work is that we **have to export everything implemented in a sub-entry** in its `public-api.ts` file.

> *This can be problematic if we have a lot of internal (private) implementation which we do NOT want to expose to consumers as they might be tempted to use it just because it was available.*

Luckily there is a way to solve it. In the previous section we discussed way to prevent consumers from using deep imports with the help of Tslint `import-blacklist` rule.

We can make stuff private by creating dedicated “internal” sub-entries (eg `internal-utils/`) which **will NOT be exported** from the root `public-api.ts` file.

The only way to consume such sub-entry would be using deep import like `@my-org/some-lib/internal-utils` which is forbidden by the previously created Tslint rule and would lead to a linting error so we’re safe!

> *This approach works but seems a bit too much effort to achieve commonly needed thing like privacy so maybe another room for improvement?*

How to bring dependency into consumer

Sometimes our library uses some other 3rd party libraries like `date-fns`. In such situation we have two main options how to make sure that the consumer application will get such dependency and both options come with their own trade-offs.

1. Dependency will be brought by the library

Our dependency can be declared as a dependency in the **INTERNAL** library `packake.json` file (we also have to add it to the `whitelistedNonPeerDependencies` it in the `ng-package.json` file). That way whenever we install our library in some application we will also install that 3rd party dependency for that application.

> ***The good*** *thing about this approach is that it simplifies the setup for the consumer apps. Developers do not have to read and follow installation setup documentation or notice missing* `peerDependency` *warning during* `npm install`*...*

The problem with this approach is that consumer might already use different version of the same 3rd party library which might lead to variety of problems when the public API changes (breaks) between major releases of the 3rd party library…

> Another big potential problem with this approach is that even if everything works just find we might end up with multiple copies of such library in our

application bundles which would hurt startup time for our users — many thanks to Alan Agius for providing feedback to enhance the content of this article!

2. Dependency will be declared as a peerDependency

Another option is to declare 3rd party library in the `peerDependencies` in the internal `package.json` file.

> *The consumer application is then responsible for installing and providing that 3rd party dependency which is bit more work for the developers on the consumer side but **more safe in terms of life-cycle** management!*

. . .

Demo app to showcase or test library in the same workspace

Sometimes it can be valuable to showcase library functionality in the locally included demo application. Such application can be generated in the same

workspace using Angular Schematics `ng g application some-lib-demo`.

> *The demo application then has to consume entities provided by the library by importing its modules, components and services…*

Based on previously defined Typescript `paths` aliases we have two main options for importing of the library code in the demo app.

We can use standard `@my-org/some-lib` imports (same as in normal consumer apps) but for that to work we would have to first pre-build our library using `ng build` (so that its bundles are available in the `dist/`)

Second option is to use `@my-org/some-lib/src/lib/<sub-entry-name>` in which case running or building of the demo app is possible also without pre-building of the library.

> *This approach will also live reload on any changes to the library code so depending on your development style it might be preferable to the need to pre build library (eg if you are building component heavy library and use demo as an development enhancement tool).*

Bonus: Make it work with Jest

> *Jest is unfortunately implemented in a way that prevents its direct integration as a part of Angular CLI build pipeline. It's also the main reason why there is no official Angular CLI Jest support as the CLI team would have to support two separate pipelines…*

Of course, we can use Jest to test our library which was implemented using the architecture described in this article. All we have to do is to add `moduleMapper` property in the `jest.config.js` which closely follows aliases that we defined in the main `tsconfig.json` file!

```
moduleNameMapper: {
    '@my-org\\/some-lib\\/(.*)': '<rootDir>/$1',
}
```

The exact value after `<rootDir>` depends on the location of your `jest.config.js` file. In the example above our `jest.config.js` file is

located inside `projects/some-lib/` folder which means that our paths of `src/lib/feature-a` will map correctly as `$1` to the real file location.

Bonus: In-depth view of what is going on

One of the issues that we have faced led us to discussion with fellow GDEs and even Angular team members which mentioned various amazing resources which can help us to get even better understanding of how and why sub-entries (and tree-shaking) works in Angular / Typescript…

Many thanks to George Kalpakas @gkalpakas, Alan Agius @AlanAgius4, Pete Bacon Darwin @petebd and Joost Koehoorn for all the input!

- Angular Optimization Pipeline (+ `check-side-effects`)
- Typescript explicit vs `*` imports from Webpack tree-shaking perspective
- Angular CLI tree-shaking (Webpack + Terser)

. . .

Great, we have made it to the end! 🔥

I hope you enjoyed learning about the best way how to architect your Angular libraries and what benefits you can get by implementing ideas described in this article! Check out the demo project!

Please support this guide with your 👏👏👏 to help it spread to a wider audience, it would be very appreciated 🙏.

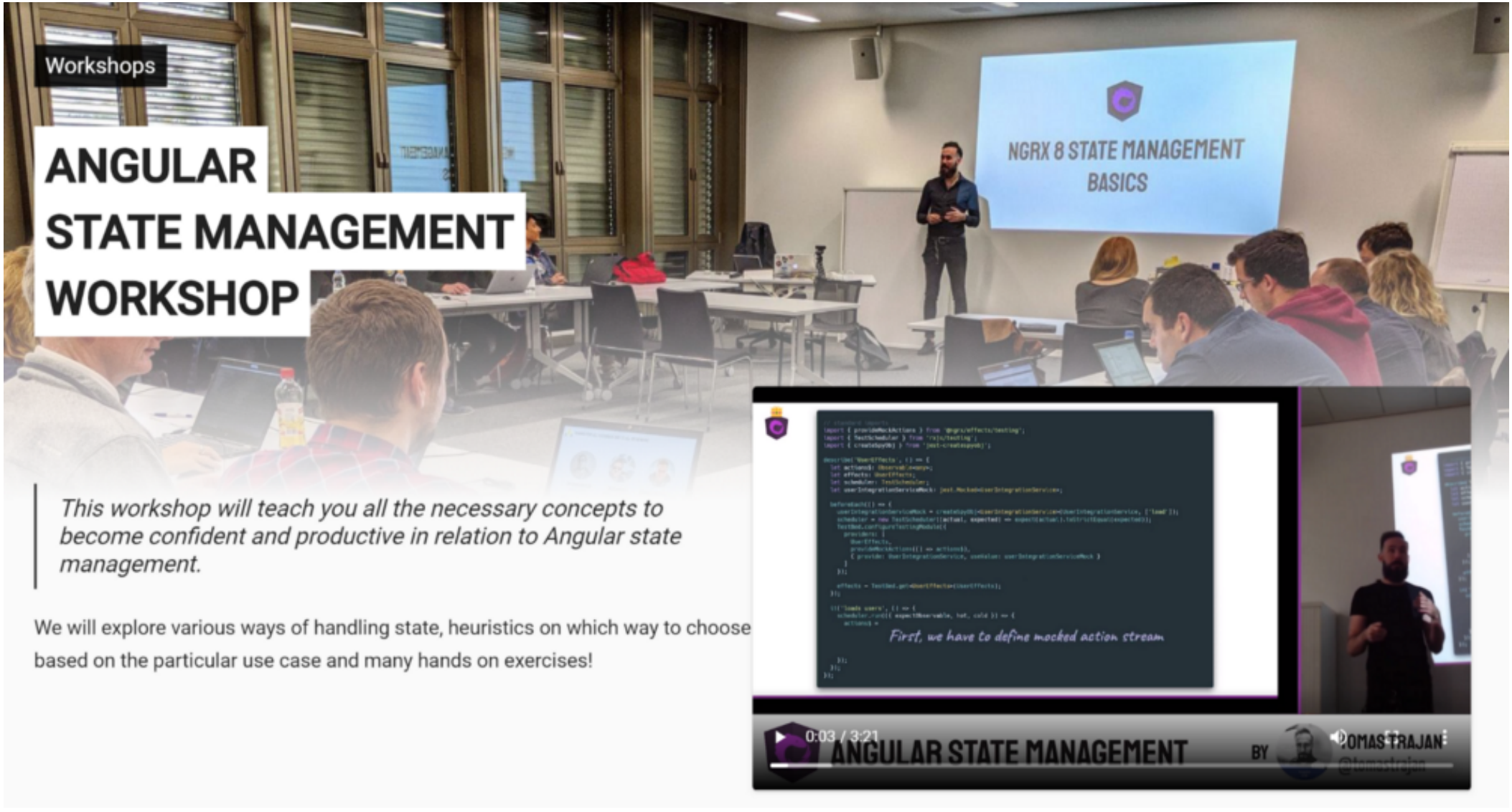
Also, don't hesitate to ping me if you have any questions using the article responses or Twitter DMs @tomastrajan.

. . .

And never forget, future is bright

Obviously the bright future! (📷 by Andreas Sjövall)

Wanna learn some Angular and you and your team are from Switzerland or surrounding countries? Check out my workshop offer!



> Starting an Angular project? Check out Angular NgRx Material Starter

Angular NgRx Material Starter with built in best practices, theming and much more!

> *Building Angular based microfrontends? Check out @angular-extensions/elements*

@angular-extensions/elements is a library that helps you lazy load Angular elements or any other webcomponents in your Angular applications

. . .

If you made it this far, feel free to check out some of my other articles about Angular and frontend software development in general…

How to architect epic Angular app in less than 10 minutes! ⏱😄

In this article we are going to learn how to scaffold new Angular application with clean, maintainable and extendable...

medium.com

The Best Way to build reactive sub-forms with Angular

Learn how to extract repetitive sub-form implementations into standalone, robust and type safe components with Angular...

medium.com

The Best Way To Lazy Load Angular Elements

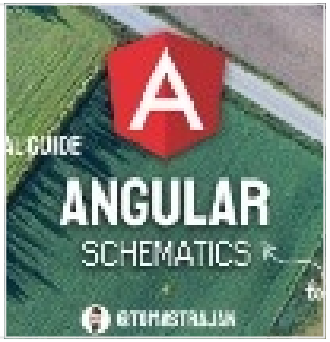
Or any other Web Components in your Angular applications!

medium.com

Total Guide To Custom Angular Schematics

Schematics are great! They enable us to achieve more in shorter amount of time! But most importantly, we can think less...

medium.com



1K 9

Angular Frontend Typescript Angularjs Software Architecture

More from Tomas Trajan

Follow

🅰 Google Developer Expert for Angular #GDE ❤ Typescript 🛠 Maker of the @releasebutler and Medium Enhanced Stats ☀ Obviously the bright Future

More From Medium

Manage forms with custom React hook

Bikram Bhattacharya

What are Controlled & Uncontrolled Component in React?

Chennan Mao

Recursive Functions in JavaScript

Ross Mawdsley in The Startup

Tech Stack and Initial Project Setup

Jon Meyers in Geek Culture

Cartoon Guide to Data Structures—Singly Linked Lists

Rajesh Pillai in Full Stack Engineering

Async, Defer and Dynamic Scripts

Daniel Movsesyan in The Startup

Agnostic Event Reporting

Micha Sherman in Fiverr Engineering

Integrating Firebase Cloud Functions with Google Calendar API

Scott McCartney in Zero Equals False