EVEN MORE
OPINIONATED

ANGULAR 2 STYLE GUIDE

@LAKATOS88

Alex Lakatos

I'm Alex, and amongst other things I'm a Mozilla Reps Council member, which basically means I'm a volunteer that helps other people volunteer. I do entertain the idea of a day job though.

EVEN MORE
OPINIONATED

ANGULAR 2 STYLE GUIDE

@LAKATOS88

Alex Lakatos

I'll be talking in some serious terms today (do, consider avoid)… but do take it with a pinch of salt, I'm not really a serious guy, I occasionally do things like dancing on a table with money tied to my behind…

EVEN MORE OPINIONATED
ANGULAR 2.0 STYLE GUIDE

- Single Responsibility
- Naming
- Coding Conventions
- Application Structure
- Components
- Directives
- Services
- Data Services
- Lifecycle Hooks

Why am I talking about this? Well, there are 119 rules in the Angular 2 style guide. Too many for my team to realistically follow. So we went through them, and trimmed them down to a minimum viable style guide for our team.

SINGLE RESPONSIBILITY

- Do define one thing (e.g. service or component) per file.
- Consider limiting files to 400 lines of code.

Why? One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.
Why? One component per file avoids hidden bugs that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.
Why? A single component can be the default export for its file which facilitates lazy loading with the Component Router.

SINGLE RESPONSIBILITY

- Do define small functions
- Consider limiting to no more than 75 lines.

Why? Small functions are easier to test, especially when they do one thing and serve one purpose.
Why? Small functions promote reuse, are easier to read & maintain.
Why? Small functions help avoid hidden bugs that come with large functions that share variables with external scope, create unwanted closures, or unwanted coupling with dependencies.

NAMING

- Do use consistent names for all symbols.
- Do follow a pattern that describes the symbol's feature then its type. ***feature.type.ts***.

Why? Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.
Why? The naming conventions should simply help us find our code faster and make it easier to understand.
Why? Names of folders and files should clearly convey their intent. For example, app/heroes/hero-list.component.ts may contain a component that manages a list of heroes.

NAMING

- Do use dashes to separate words in the descriptive name.
- Do try to stick with conventional type names: ***.service, .component, .pipe, .module, .directive***

Why? Type names provide a consistent way to quickly identify what is in the file.
Why? Make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.
Why? Provides pattern matching for any automated tasks.

NAMING

- Do put bootstrapping and platform logic for the app in a file named ***`main.ts`***.
- Avoid putting app logic in the ***`main.ts`***. Instead consider placing it in a *Component* or *Service*.

Why? Follows a consistent convention for the startup logic of an app.
Why? Follows a familiar convention from other technology platforms.

NAMING

main.ts

```
1.  import { platformBrowserDynamic } from '@angular/platform-
    browser-dynamic';
2.
3.  import { AppModule }              from './app/app.module';
4.
5.  platformBrowserDynamic().bootstrapModule(AppModule)
6.    .then(success => console.log(`Bootstrap success`))
7.    .catch(err => console.error(err));
```

Why? Follows a consistent convention for the startup logic of an app.
Why? Follows a familiar convention from other technology platforms.

NAMING

- Do use lower camel case for naming the selectors of your directives.
- Do use a custom prefix for the selector of your components and directives

Why? Keeps the names of the properties defined in the directives that are bound to the view consistent with the attribute names.
Why? The Angular HTML parser is case sensitive and will recognize lower camel case.
Why? Prevents element name collisions with components in other apps and with native HTML elements.

CODING CONVENTIONS

- Do use upper camel case when naming classes and interfaces.
- Do use lower camel case to name properties and methods.
- Consider spelling const variables in lower camel case.

Why? Follows conventional thinking for class names, properties and methods
Why? Classes can be instantiated and construct an instance. We often use upper camel case to indicate a constructable asset.

Why? lower camel case variable names (heroRoutes) are easier to read and understand than the traditional UPPER_SNAKE_CASE names (HERO_ROUTES).
Why? The tradition of naming constants in UPPER_SNAKE_CASE reflects an era before the modern IDEs that quickly reveal the const declaration. TypeScript itself prevents accidental reassignment.

CODING CONVENTIONS

- Consider naming an interface without an I prefix.
- Avoid prefixing private properties and methods with an underscore.

Why? TypeScript guidelines discourage the "I" prefix.
Why? JavaScript lacks a true private property or method & TypeScript tooling makes it easy to identify private vs public properties and methods.

APPLICATION STRUCTURE

- Do structure the app such that we can
  - **L**ocate our code quickly,
  - **I**dentify the code at a glance, keep the
  - **F**lattest structure we can, and
  - **T**ry to be *DRY* (Don't Repeat Yourself).

Why? LIFT Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. These principles are listed in order of importance. Another way to check our app structure is to ask ourselves: How quickly can we open and work in all of the related files for a feature?

APPLICATION STRUCTURE

- Do keep a flat folder structure as long as possible.
- Consider creating folders when you get to seven or more files
- Do put all of the app's code in a folder named app.
- Consider creating a folder for each component including its `.ts`, `.html`, `.css` and `.spec` file.
- Do create folders named for the feature they represent.

Why? No one wants to search for a file through seven levels of folders. A flat structure is easy to scan.
Why? Components often have four files (e.g. *.html, *.css, *.ts, and *.spec.ts) and can clutter a folder quickly.
Why? The LIFT guidelines are all covered.

APPLICATION STRUCTURE

- Do put the contents of lazy loaded features in a lazy loaded folder. A typical lazy loaded folder contains a routing component, its child components, and their related assets and modules.

Why? The folder makes it easy to identify and isolate the feature content.

COMPONENTS

- Do use dashed-case or kebab-case for naming the element selectors of components.
- Do define Components as elements via the selector.

Why? Keeps the element names consistent with the specification for Custom Elements.
Why? A component represents a visual element on the page. Defining the selector as an HTML element tag is consistent with native HTML elements and WebComponents.

COMPONENTS

- Consider keeping templates and styles in the same file, when under 3 lines.
- Do put presentation logic in the Component class, and not in the template.

Why? Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

COMPONENTS

- Do place properties up top followed by methods.
- Do place private members after public members, alphabetised.

Why? Placing members in a consistent sequence makes it easy to read and helps instantly identify which members of the component serve which purpose.

DIRECTIVES

- Do use attribute directives when you have presentation logic without a template.

app/shared/highlight.directive.ts

```
1. @Directive({
2.   selector: '[tohHighlight]'
3. })
4. export class HighlightDirective {
5.   @HostListener('mouseover') onMouseEnter() {
6.     // do highlight work
7.   }
8. }
```

app/app.component.html

```
<div tohHighlight>Bombasta</div>
```

Why? Attributes directives don't have an associated template.
Why? An element may have more than one attribute directive applied.

SERVICES

- Do use services as singletons within the same injector. Use them for sharing data and functionality.
- Do create services with a single responsibility that is encapsulated by its context.
- Do create a new service once the service begins to exceed that singular purpose.

Why? Services are ideal for sharing methods & stateful in-memory data.
Why? When a service has multiple responsibilities, every component or service that injects it now carries the weight of them all.

SERVICES

- Do provide services to the Angular 2 injector at the top-most component where they will be shared.
- Do use the *@Injectable* class decorator instead of the *@Inject* parameter decorator when using types as tokens for the dependencies of a service.

Why? The Angular 2 injector is hierarchical.
Why? When providing the service to a top level component, that instance is shared and available to all child components of that top level component.
Why? This is ideal when a service is sharing methods or state.
Why? The Angular DI mechanism resolves all the dependencies of our services based on their types declared with the services' constructors.
Why? When a service accepts only dependencies associated with type tokens, the @Injectable() syntax is much less verbose compared to using @Inject() on each individual constructor parameter.

DATA SERVICES

- Do refactor logic for making data operations and interacting with data to a service.
- Do make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.

Why? The component's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the component be simpler and more focused on the view.
Why? This makes it easier to test (mock or real) the data calls when testing a component that uses a data service.
Why? Data service implementation may have very specific code to handle the data repository. This may include headers, how to talk to the data, or other services such as Http. Separating the logic into a data service encapsulates this logic in a single place hiding the implementation from the outside consumers (perhaps a component), also making it easier to change the implementation.

LIFECYCLE HOOKS

- Do implement the lifecycle hook interfaces.

app/heroes/shared/hero-button/hero-button.component.ts

```
1. @Component({
2.   selector: 'toh-hero-button',
3.   template: `<button>OK</button>`
4. })
5. export class HeroButtonComponent implements OnInit {
6.   ngOnInit() {
7.     console.log('The component is initialized');
8.   }
9. }
```

Why? We get strong typing for the method signatures. The compiler and editor can call your attention to misspellings.

Codelyzer

HTTPS://ANGULAR.IO/STYLEGUIDE

STYLE A-01

> **Do** use codelyzer to follow this guide.

> **Consider** adjusting the rules in codelyzer to suit your needs.

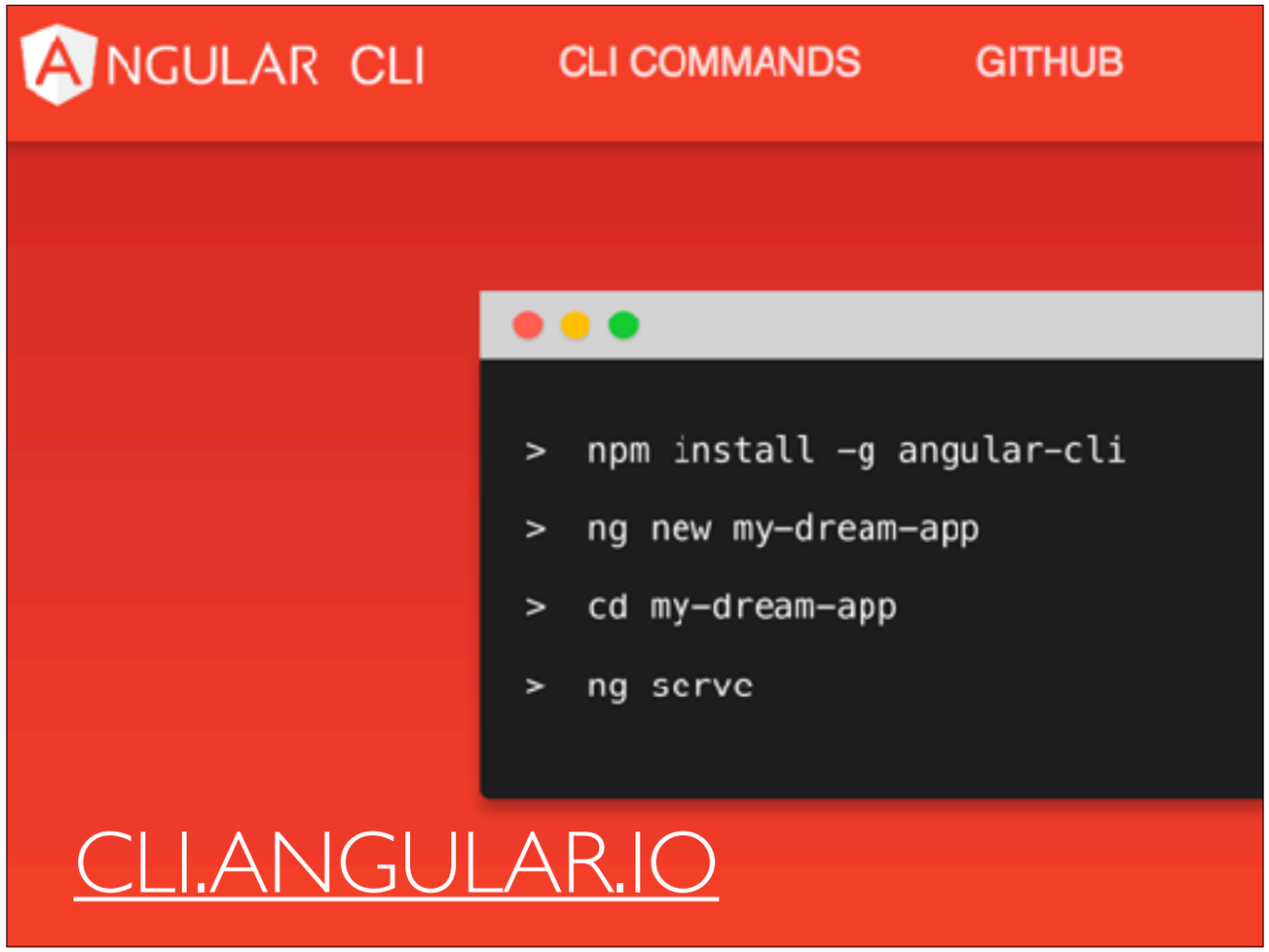
Back to top

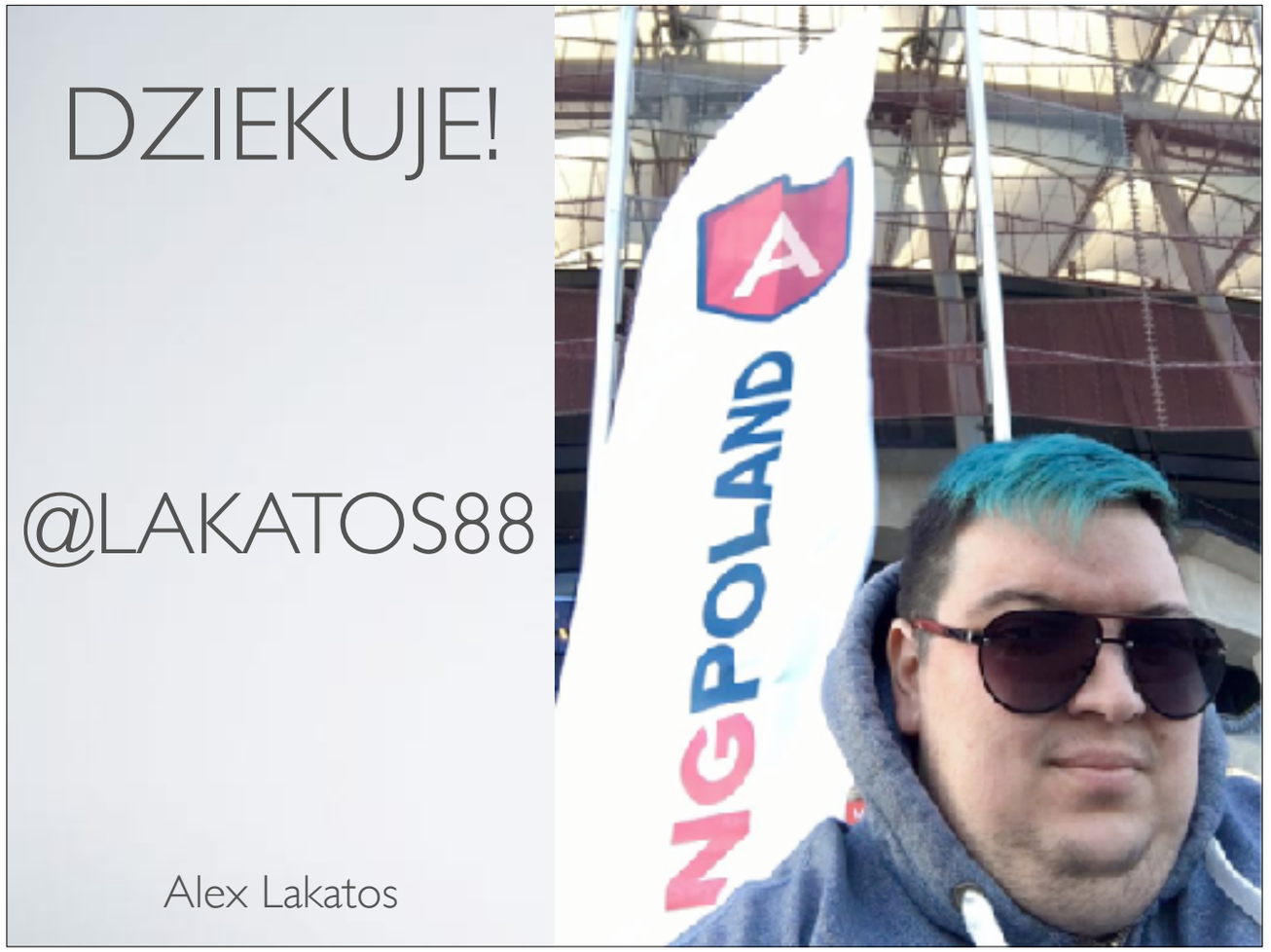
File Templates and Snippets

STYLE A-02

> **Do** use file templates or snippets to help follow consistent styles and patterns. Here are te
> of the web development editors and IDEs.

> **Consider** using snippets for Visual Studio Code that follow these styles and guidelines.


ANGULAR CLI
CLI COMMANDS
GITHUB
> npm install -g angular-cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
CLI.ANGULAR.IO



gien cuie ng-poland.

Thank you ng-poland.