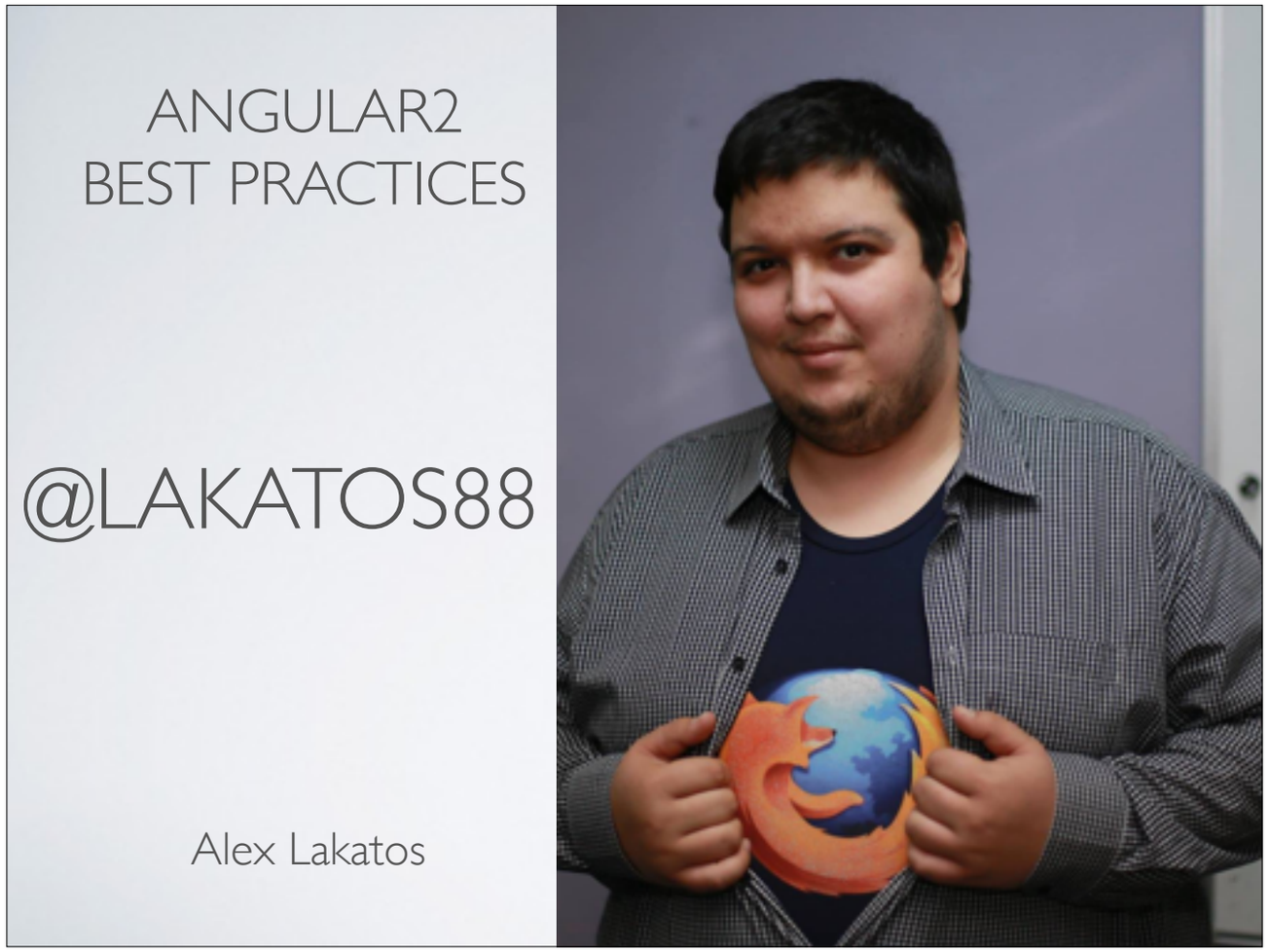

I’m Alex, and amongst other things I’m a Mozilla Reps Mentor, which basically means I’m a volunteer that helps other people volunteer.

ANGULAR2 BEST PRACTICES

@LAKATOS88

Alex Lakatos

I’m usually a pretty serious guy… except when I’m dancing on a table with money tied to my behind…

ANGULAR 2.0 STYLE GUIDE

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

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.

Why? Small functions are easier to read.

Why? Small functions are easier to 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. The recommended pattern is ***feature.type.ts***.
- Do use dashes to separate words in the descriptive name.
- 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.
- Do Use lower camel case for naming the selectors of our directives.
- Do use a custom prefix for the selector of our directives - **swa**

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.

CODING CONVENTIONS

- Do use upper camel case when naming classes and interfaces.
- Consider naming an interface without an I prefix.
- Do use uppercase with underscores when naming constants.
- Do use lower camel case to name properties and methods.
- Avoid prefixing private properties and methods with an underscore.

Why? Follows conventional thinking for class names.

Why? Classes can be instantiated and construct an instance. We often use upper camel case to indicate a constructable asset.

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).
- Avoid files with multiple components, multiple services, or a mixture.
- Do keep a flat folder structure as long as possible.
- Consider creating folders when we 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.
- Consider creating a file that imports, aggregates, and re-exports items. We call this technique a barrel.
- Consider naming this barrel file index.ts.

Why? LIFT Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. 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?

COMPONENTS

- Do use kebab-case for naming the element selectors of our components.
- Do define Components as elements via the selector.
- Do extract templates and styles into a separate file, when more than 3 lines.
- Do place properties up top followed by methods.
- Do place private members after public members, alphabetised.
- Do put presentation logic in the Component class, and not in the template.

DIRECTIVES

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

app/shared/highlight.directive.ts

```
@Directive({
  selector: '[tohHighlight]'
})
export class HighlightDirective {
  @HostListener('mouseover') onMouseEnter() {
    // do highlight work
  }
}
```

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.
- 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 our attention to misspellings.