Angular best practices

Introduction to Angular 6

General coding best practices

- General coding rules

- https://www.makeuseof.com/tag/basic-programming-principles/
- https://techbeacon.com/35-bad-programming-habits-make-your-code-smell
- https://sourcemaking.com/refactoring/smells
- https://blog.codinghorror.com/code-smells/

Angular style guide

- https://angular.io/guide/styleguide

Develop in modular fashion

- For small apps, you do not need multiple modules
  - all the components for the app are declared in one module.
- For larger apps, splitting your app into core, shared and multiple feature modules is recommended.
  - Each module can have its own components, services, directives and pipes.
- Core module consists of components (i.e. header, main navigation, footer) that will be used across the entire app.
  - All services which have to have one and only one instance per application (singleton services) should be implemented here.
  - Typical example can be authentication service or user service.

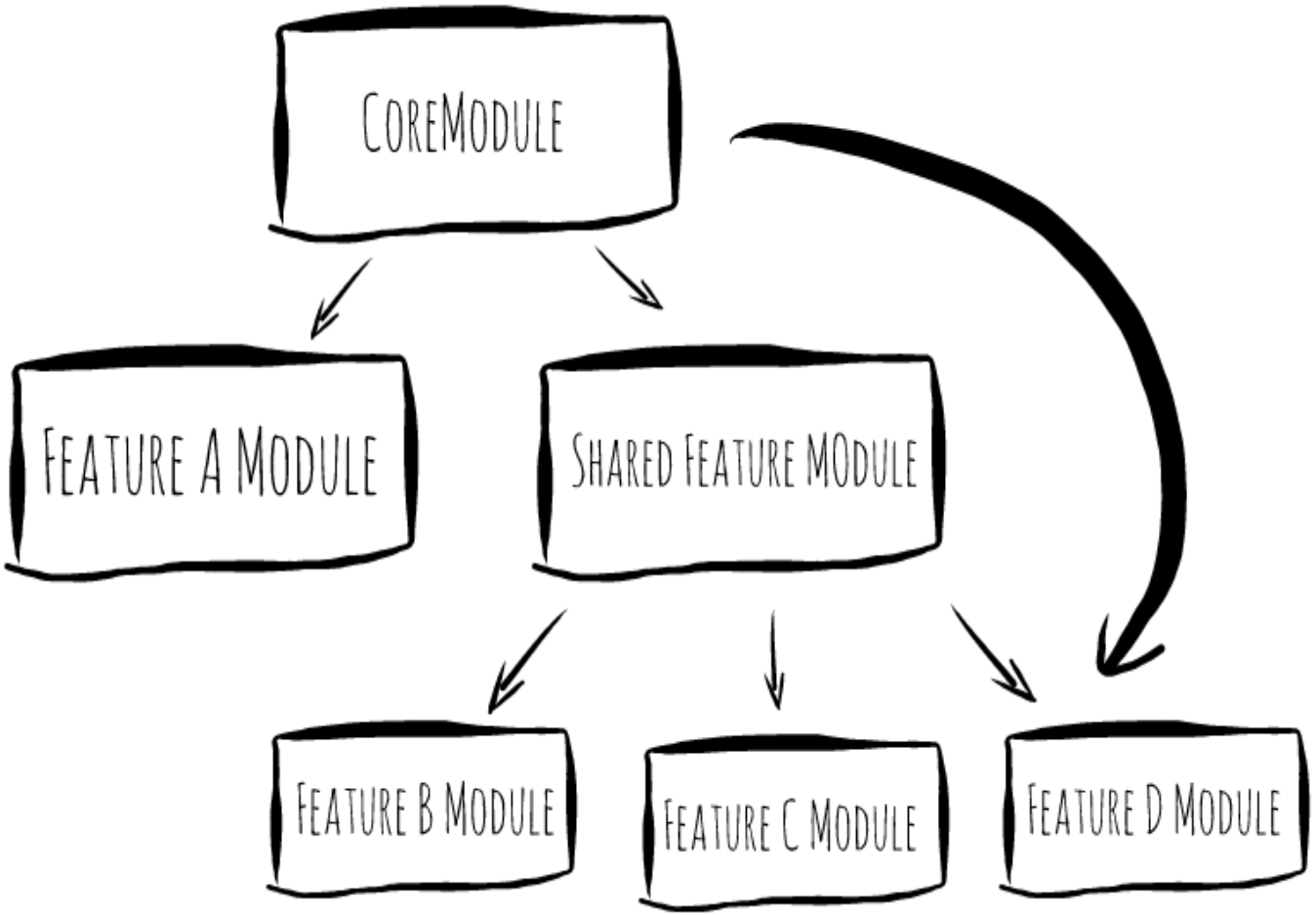
Develop in modular fashion

- Shared module can have components, directives and pipes that will be shared across multiple modules and components, but not the entire app necessarily.
  - These components don't import and inject services from core or other features in their constructors.
  - They should receive all data though attributes in the template of the component using them.
  - Shared module should not have any dependency with the rest of the application.
- Register the shared module inside the root module
  - It is important not to register services, that we want to use globally, in a shared module.
  - We should register such services inside its own feature module or in the root module.

Develop in modular fashion

- A feature module delivers a cohesive set of functionality focused on a specific application feature
  - E.g. a user workflow, routing, or forms.
- A feature module should only import services from the the root module.
  - Ideally they should also not depend on any other feature.

Develop in modular fashion



Lazy loading a feature module

- Lazy loading a feature module is the best approach when it comes to accessing a module via Angular's routing.
- A feature module won't be loaded initially, only when you decide to initiate it
  - This increases app speed
- Only one feature module should be loaded synchronously during the app startup to show initial content.
  - Every other feature module should be loaded lazily after user triggered navigation.

Avoid in-lining if possible

- For large applications, separate the component, template and CSS into separate files.

Separation of concerns

- We want our components to be as simple as possible to facilitate maintenance and testing
- This means if our component needs to do some complex logic that is non-UI related, delegate this to a service

Use services for data retrieval

- Based on the SRP: component should just focus on UI-related functionality.
- Data retrieval is typically done using HTTP requests
- We delegate the HTTP-related functionality from the component to the service which can subsequently be shared across multiple components

Service provision

- Make services as injectable through the @Injectable decorator.
  - This is necessary only when a service injects another service, but is recommended use because you never know when the service will need to inject another one and it will be hard to remember that the injectable decorator was not used.
- If we want to use our service as a singleton
  - We should provide that service at the application root where Angular creates a single, shared instance of our service, available for all components

```
@Injectable({
  providedIn: 'root',
})
```

Create reusable components

- Components should follow SRP.
- We can reuse those types of components inside any parent component and pass the data through the @Input decorator.
- Those components can emit events by using the @Output decorator and EventEmmiter class.

Aliases for imports

- Aliasing our app and environments folders will enable us to implement clean imports which will be consistent throughout our application.
- Add the following piece of code into tsconfig.json file to make your imports short and well organized across the app:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    removed for brevity,
    "paths": {
      "@app/*": ["app/*"],
      "@env/*": ["environments/*"]
    }
  }
}
```

Aliases for imports

- When you're done, these imports

```
import { LoaderService } from '../../../loader/loader.service';
import { environment } from '../../environment';
```

- Should be refactored into these:

```
import { LoaderService } from '@app/loader/loader.service';
import { environment } from '@env/environment';
```

Use Angular CLI if possible

- Angular CLI has its own set of commands for creating the Angular project, creating components, modules, services etc
- It will reference those components and services in the appropriate modules and will comply to the suitable naming convention

Appropriate access modifiers

- It is very important to distinguish the functions and properties that we are going to use only in our components from those that we are going to call from our template file.
- Those properties and functions which we are going to reference from the template should always have the public access modifier

Proper constructor use for DI

- We should use the constructor method signature to setup Dependency Injection for our services, rather than perform it within the body of the method.
- Instead of

```
private router: Router;

constructor(routerParam: Router) {
  this.router = routerParam;
}
```

- Do this:

```
constructor(private router: Router) {}
```

Avoid logic in templates

- If you have any sort of logic in your templates, it is good to extract it out into its component.
- Having logic in the template means that it is not possible to unit test it and therefore it is more prone to bugs

Avoid logic in templates

- Before

```
// template

<p *ngIf="role==='developer'">
    Status: Developer
</p>

// component
public ngOnInit (): void {
    this.role = 'developer';
}
```

- After

```
// template
<p *ngIf="showDeveloperStatus">
    Status: Developer
</p>

// component
public ngOnInit (): void {
    this.role = 'developer';
    if (role==='developer')
        this.showDeveloperStatus = true;
}
```

Safe Navigation Operator (?)

- Recommended to use the safe navigation operator while accessing a property from an object in a component's template
  - If the object is null and we try to access a property, we are going to get an exception.
  - If we use the save navigation (?) operator, the template will ignore the null value and will access the property once the object is not the null anymore.

```
<div class="cat">
  {{user?.name}}
</div>
```

Use Lifecycle Hooks

- We should use them when appropriate
- If we need to fetch some data from a database as soon as our component is instantiated, we should use ngOnInit() lifecycle hook and not the constructor.
- If we need to clean up some resources as soon as our component is destroyed, we should use ngOnDestroy() lifecycle hook.

Use Sass

- Sass is a styles preprocessor which brings support for additional features such as variables, functions, mixins.
- Sass is also required to effectively use official Angular Material Components library with it's extensive theming capabilities.
- To use Sass we have to generate our project using Angular CLI ng new command with --style scss flag.

Use trackBy

- When using ngFor to loop over an array in templates, use it with a trackBy function which will return an unique identifier for each item.
- By default, when all the elements references in an array changes, Angular re-renders the whole DOM tree.
- But if you use trackBy, Angular will know which element has changed and will only make DOM changes for that particular element.

Add caching mechanisms

- Having a caching mechanism means avoiding unwanted API calls.
- When making API calls, responses from some of them do not change often.
  - In those cases, you can add a caching mechanism and store the value from the API.
  - When another request to the same API is made, check if there is a value for it in the cache and if so, use it.
  - Otherwise, make the API call and cache the result.
- If the values change but not frequently, you can introduce a cache time where you can check when it was last cached and decide whether or not to call the API.
  - By only making the API calls when required and avoiding duplication, the speed of the application improves as eliminate redundant network traffic.

Angular Universal

- Angular Universal generates static application pages on the server through a process called server-side rendering (SSR).
- When Universal is integrated with your app, it can generate and serve those pages in response to requests from browsers.
- It can also pre-generate pages as HTML files that you serve later.
- This allows quick display of the first page to retain user attention.