Organizing Features and Modules

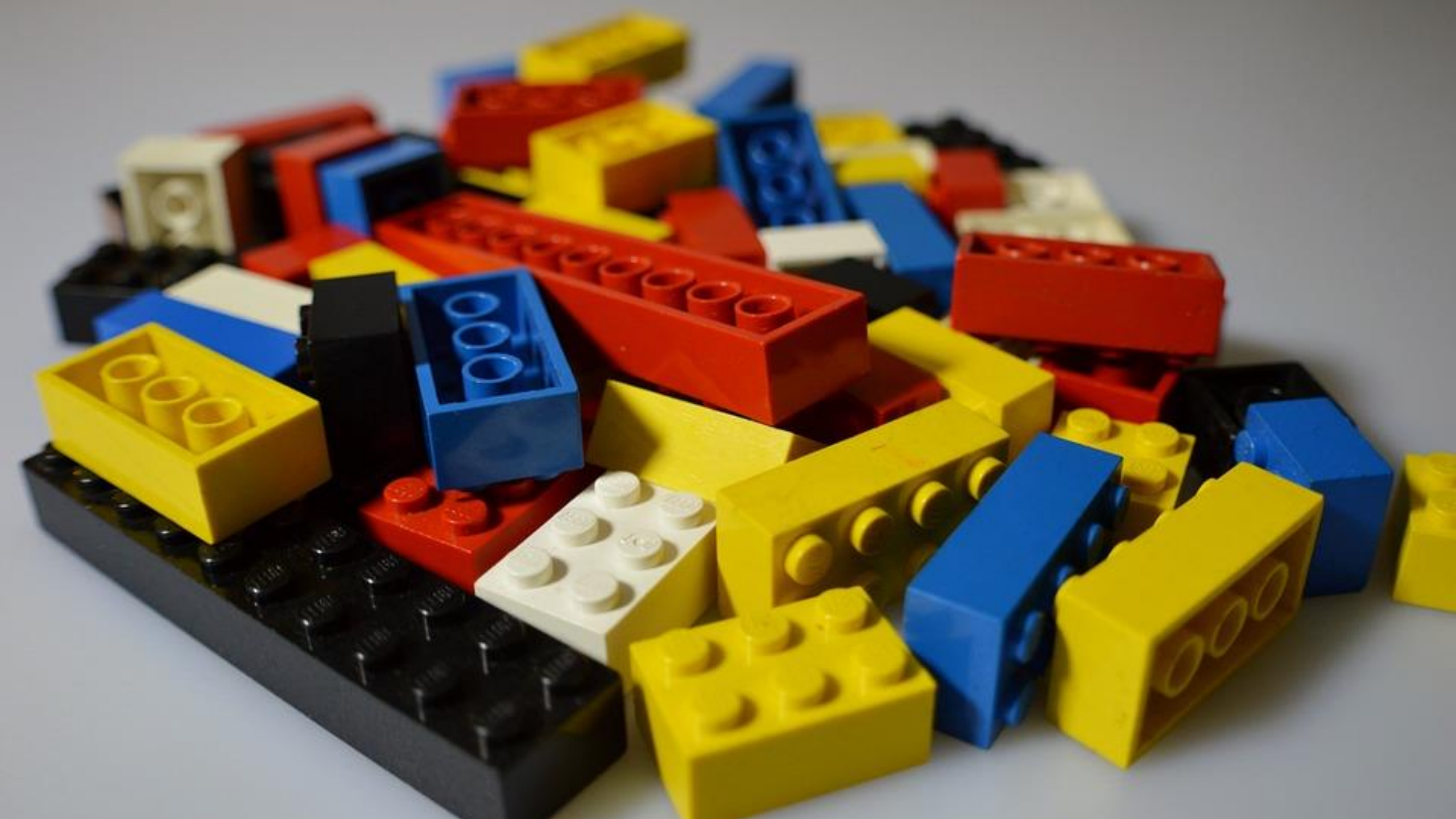
Dan Wahlin
WAHLIN CONSULTING

@danwahlin www.codewithdan.com

Module Overview

- Organizing Features
- Feature Modules
- Core and Shared Modules
- Creating a Custom Library
- Consuming a Custom Library

Organizing Features



Do structure the app such that you can Locate code quickly, Identify the code at a glance, keep the Flattest structure you can, and Try to be DRY.

Angular Style Guide

https://angular.io/guide/styleguide

LIFT

Locate code quickly

Identify the code at a glance

Keep the flattest structure you can

Try to be DRY

Options for Organizing Code

Convention-Based

Follows strict naming conventions

Related code may be separated

Can result in a lot of files in a folder in larger applications

Feature-Based

Features are organized into their own folder

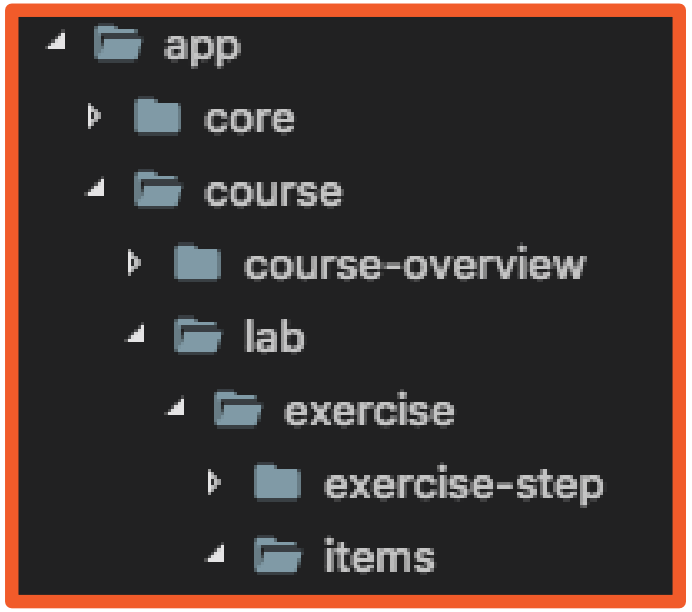
Features are self-contained

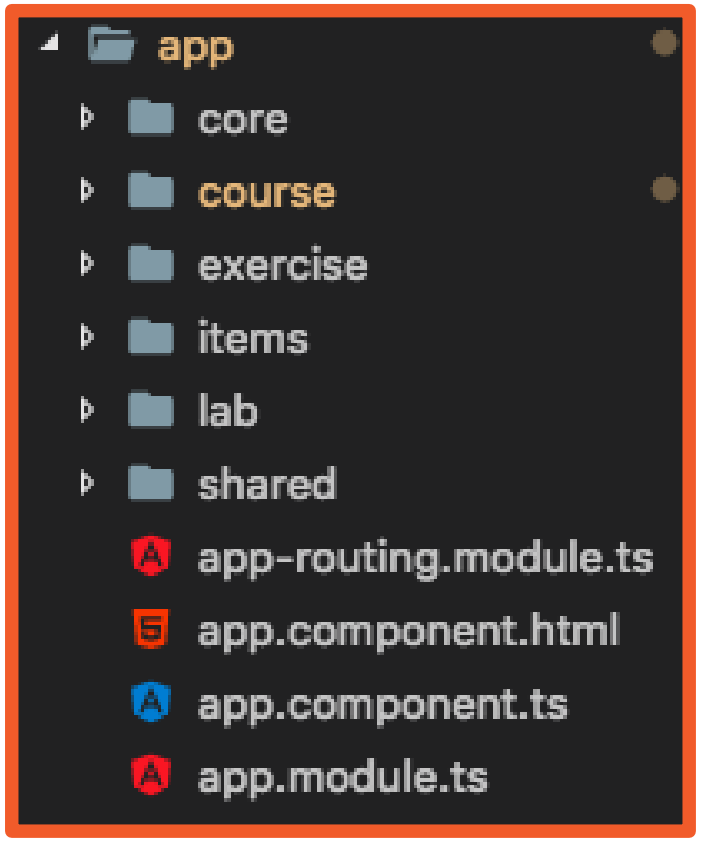
Easy to find everything related to a feature

Organizing Features

- Use a feature-based approach to structure your code
- Use the Angular CLI to generate initial feature folder/component
- Minimum of one module per feature (as appropriate)
- Avoid deeply nested folders
- Flatten feature hierarchies

Flatten Feature Hierarchies

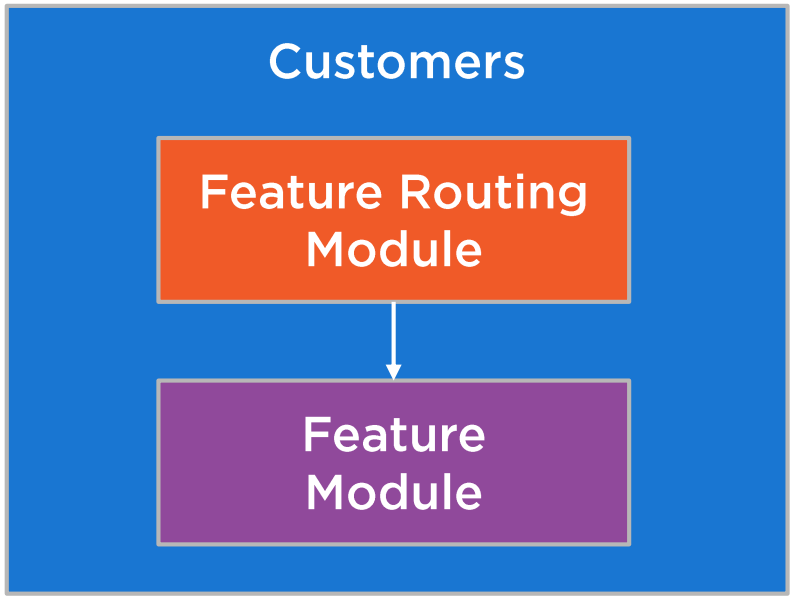


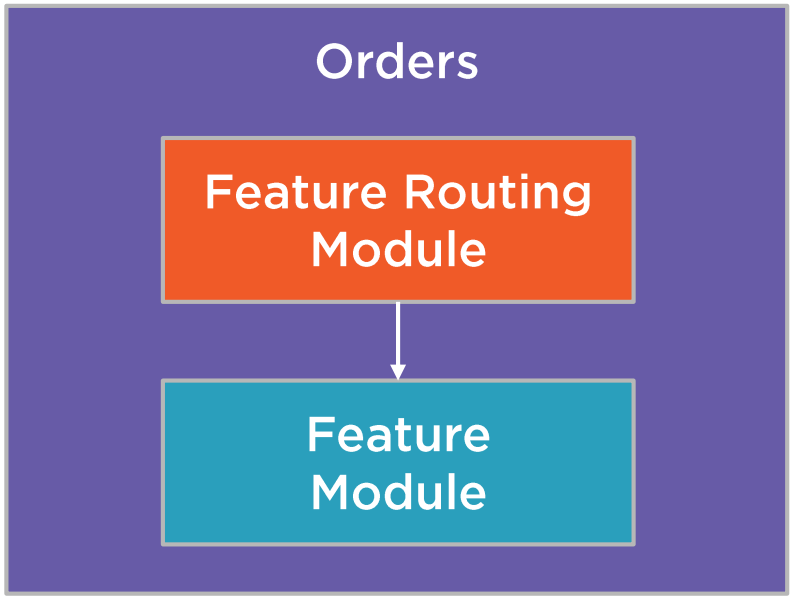


https://angular.io/guide/styleguide#flat

Feature Modules

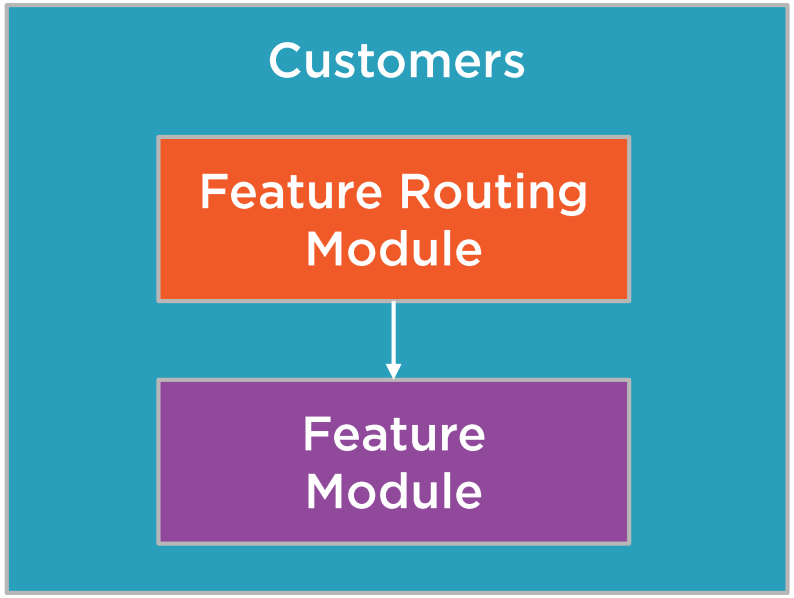
Feature Modules

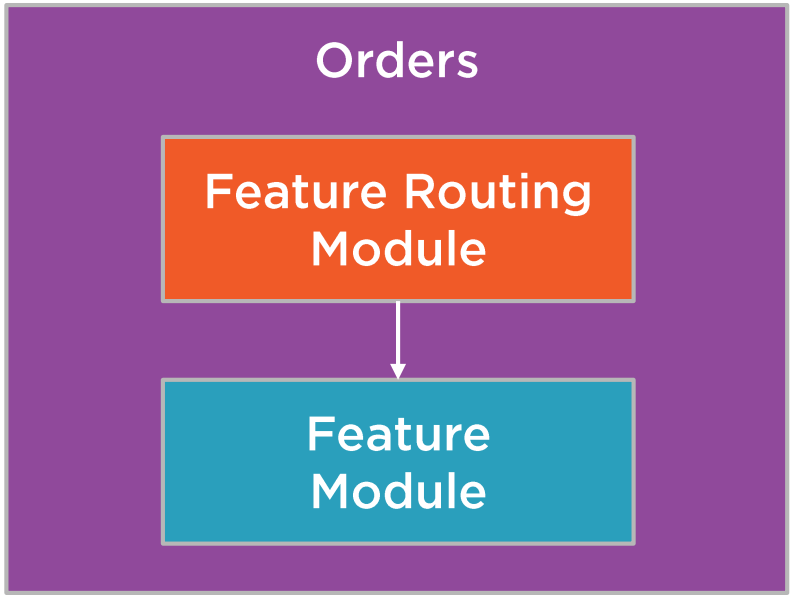




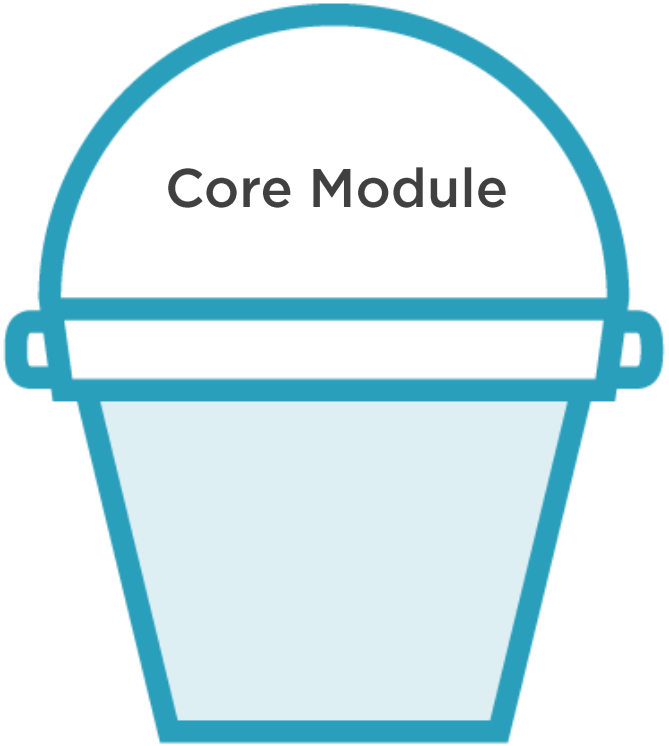
https://angular.io/guide/styleguide#feature-modules

Benefits of Self-Contained Features





Core and Shared Modules

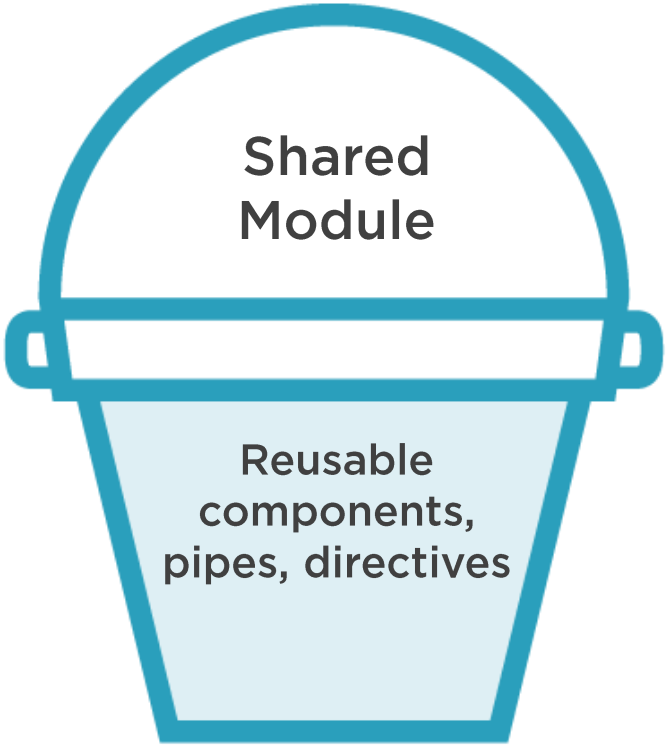


Core folder should contain singleton services shared throughout app

Services that are specific to a feature can go in the feature's folder

Example: LoggingService, ErrorService, DataService

https://angular.io/guide/styleguide#core-feature-module

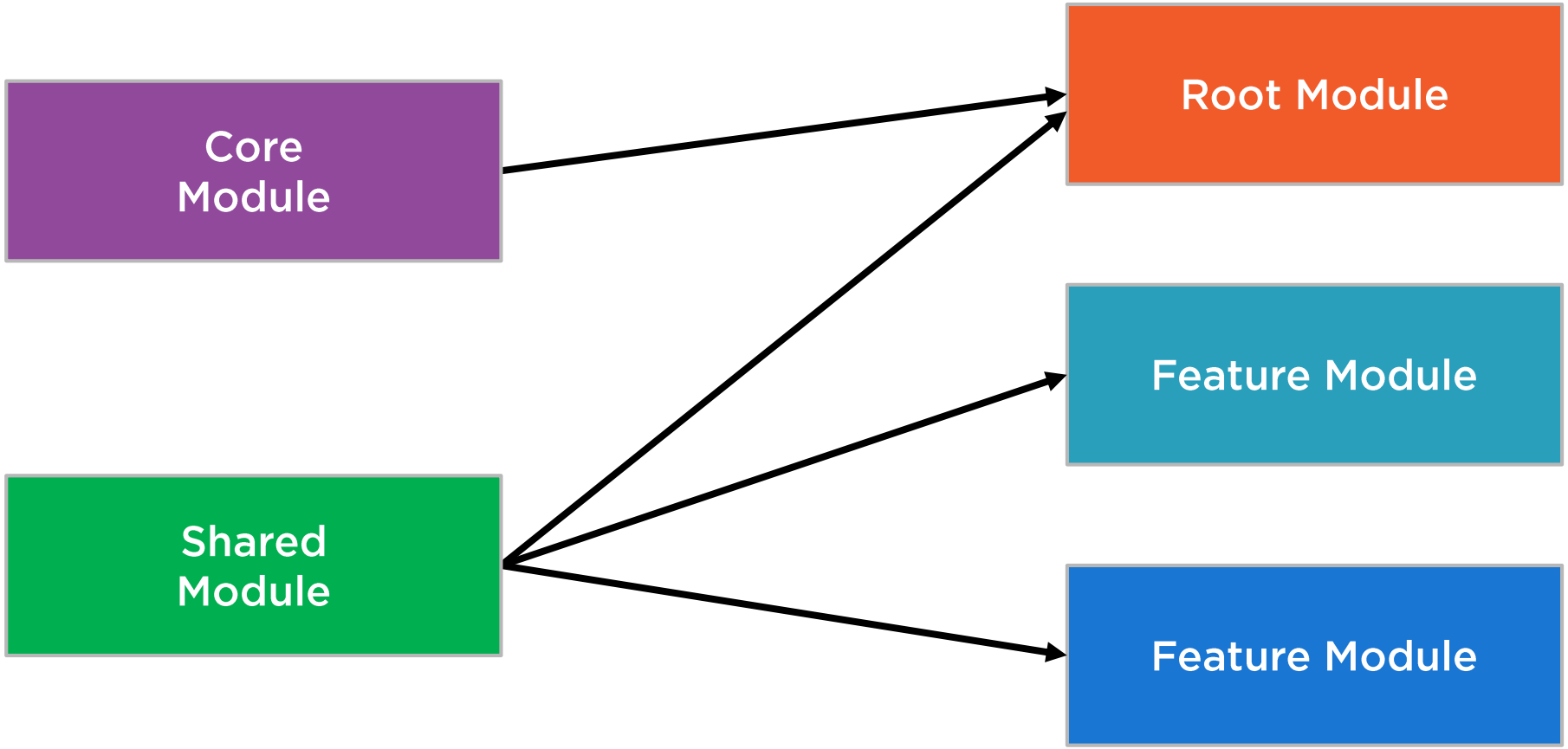


Shared folder should contain reusable components, pipes, directives

Example: CalendarComponent, AutoCompleteComponent

https://angular.io/guide/styleguide#shared-feature-module

Importing Core and Shared



Core and Shared in Action

```
export function throwIfAlreadyLoaded(parentModule: any, moduleName: string) {
  if (parentModule) {
    throw new Error(`${moduleName} has already been loaded. Import Core modules in
     the AppModule only.`);
  }
}

import { throwIfAlreadyLoaded } from './import.guard';

export class CoreModule {
  constructor( @Optional() @SkipSelf() parentModule: CoreModule) {
    throwIfAlreadyLoaded(parentModule, 'CoreModule');
  }
}
```

Preventing Reimport of Core

Core should only be imported into the root/app module

```
export class EnsureModuleLoadedOnceGuard {

  constructor(targetModule: any) {
    if (targetModule) {
      throw new Error(`${targetModule.constructor.name} has already been loaded.
        Import this module in the AppModule only.`);
    }
  }

}
```

Alternative Approach – Base Class

An alternative to throwIfAlreadyLoaded is to convert it into a class

```
import { EnsureModuleLoadedOnceGuard } from './ensure-module-loaded-once.guard';

export class CoreModule extends EnsureModuleLoadedOnceGuard {

  constructor(@Optional() @SkipSelf() parentModule: CoreModule) {
    super(parentModule);
  }

}
```

Using the Base Class

CoreModule can derive from EnsureModuleLoadedOnceGuard

Creating a Custom Library

```
$ ng new my-project

$ ng generate library my-lib

$ ng build my-lib
```

Creating a Custom Library

Angular CLI provides library functionality

The tsconfig file will be updated to look for library reference

Build the library before trying to use it in the workspace

```
$ ng build my-lib

$ cd dist/my-lib

$ npm publish
```

Publishing a Custom Library

Build your library for production

Publish to npm or to an internal npm

https://docs.npmjs.com/packages-and-modules/contributing-packages-to-the-registry

Consuming a Custom Library

Putting All the Modules Together

Organizing Functionality

Widget

Feature/Child Component

Service

Routed Component

Pipes

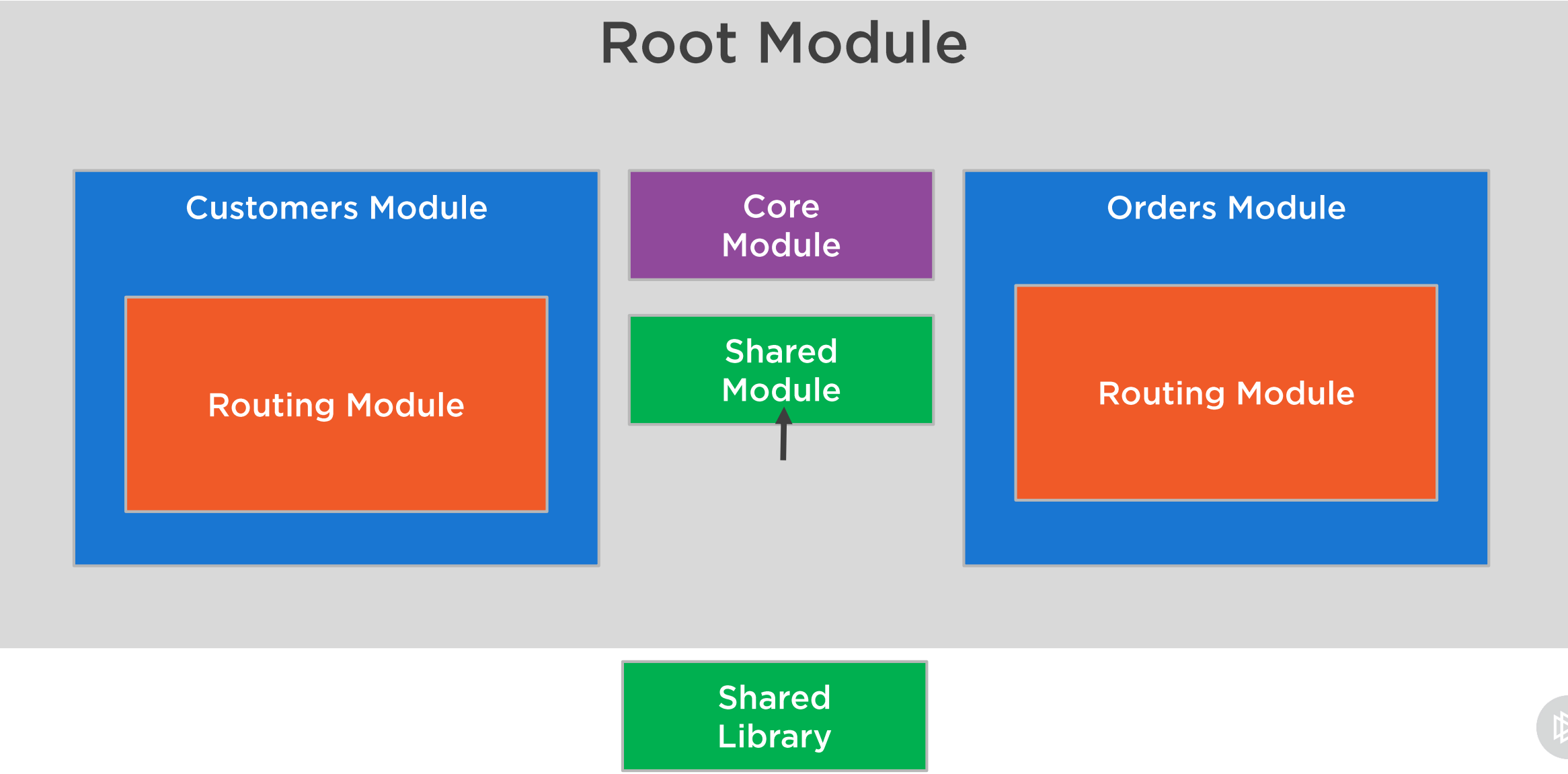
Feature Module

Routing Module

Core Module

Shared Module

Organizing Application Modules



Summary

- Use a feature-based approach
- Take time to plan feature modules
- One module per feature (as appropriate)
- Use core and shared modules
- Use the Angular CLI to create custom libraries as needed