Angular Enterprise Architecture

Learn how to set up new Angular enterprise project in a scalable and maintainable way which guarantees long term success!

Tomas Trajan
@tomastrajan

ANGULAR EXPERTS

Table of content

About author

Tomas Trajan

Google Developer Expert for Angular & Web Technologies

@tomastrajan

A Google Developer Expert for Angular & Web Technologies working as a consultant and workshop teacher with focus on Angular, NgRx, RxJs and NX. Currently, empowering teams in enterprise organizations worldwide by implementing core functionality and architecture, introducing best practices, sharing know-how and optimizing workflows.

Tomas strives continually to provide maximum value for customers and the wider developer community alike. His work is underlined by a vast track record of publishing popular industry articles, leading talks at international conferences and meetups, and contributing to open-source projects.

Acknowledgements

This book is only possible because of the amazing Angular community whose members have been very generous with sharing all the know how and experience they have gained over the years.

I would like to thank all the people who have contributed to this book.

A special thanks to Kevin Kreuzer for long term collaboration and feedback on all things Angular.

A very big thanks to Chau Tran for great feedback, especially about using `npx` for Angular CLI and suggestion to switch examples to use upcoming `inject()` based dependency injection instead of older constructor based injection.

Foreword

> In this book we are going to learn how to scaffold new enterprise ready Angular application with **clean, maintainable and extendable architecture** in almost no time!

The content represents a clear reproducible step-by-step guide to achieve same outcome every time. This is especially beneficial in large enterprise environments where it is expected that there will be more than just one application. This is in stark contrast to an ad hoc per project approach which often leads to suboptimal outcomes which are hard to fix once they occur.

Besides many actionable tips, we'll also discuss guidelines about where we should implement most common building blocks like reusable services, feature specific components and other entities commonly found in most Angular applications.

The content of this book was created with Angular 14.2.0 but the approaches and tips are valid for any Angular version starting from version 6 to any future version in the foreseeable future!

Angular CLI & development tools

> The Angular CLI is a command-line interface tool that we use to initialize, develop, scaffold, and maintain Angular applications directly from a command shell — Official Docs

Angular CLI can be installed and runs in the command shell environment which provides Node.js runtime. We're going to learn how to prepare such environment in the following chapter.

Feel free to skim through following chapters in case we're already well acquainted with the Angular CLI and running its commands.

CLI — command line interface, in the context of this book, we're going to use terms like CLI, terminal and console interchangeably

Prepare development environment

Let's prepare our local development environment to be able to install and run Angular CLI.

Node.js

First, we need to make that Node.js was installed and is available in local CLI. Node.js can be downloaded and installed from the nodejs.org website which usually displays download options based on the operating system used to access the website.

Always prefer currently available **LTS** version of the Node.js when choosing which version to install.

Once installed, run `node --version` to validate that the installation was successful. We should see an outputted version in the CLI, for example...

```
1   v16.16.0
```

When working with multiple versions of Angular, we might need to figure out compatible version of Node.js and other dependencies. In that case we can always have a look at the compatibility table .

Install Angular CLI

The Node.js environment always comes together with NPM which allows us to install additional packages like `@angular/cli`.

> **NPM** — Node.js package manager (@angular/cli page)

Let's install Angular CLI by running `npm i -g @angular/cli` in our terminal.

The successful installation can be verified by running `ng version` command.

The output will show detailed information about the environment and versions of installed packages, for example, something like the following...

```
     _                      _                 ____ _     ___
    / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
   / △ \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
  / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
 /_/   \_\_| |_|\__, |\__,_|_|\__,_|_|      \____|_____|___|
                |___/

Angular CLI: 14.2.4
Node: 16.16.0
Package Manager: npm 8.11.0
OS: linux x64

Angular:
...

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.1402.4 (cli-only)
@angular-devkit/core         14.2.4 (cli-only)
@angular-devkit/schematics   14.2.4 (cli-only)
@schematics/angular          14.2.4 (cli-only)
```

Install Angular CLI

For environments with multiple Angular applications, often with different Angular versions, installing Angular CLI globally might not be the best option.

Such installation would lead to a noisy warning about the mismatch between the globally installed Angular CLI and the local version of Angular used in the project.

This warning can be disabled using the

```
1   ng config -g cli. warnings.versionMismatch.false
```

command, but such solution then also can lead to a confusion about which version of Angular CLI is used for the executed command.

NPX solution

Another way to solve this problem is to use `npx` instead of global Angular CLI installation. In such case we could either not install Angular CLI globally at all (or uninstall it with `npm un -g @angular/cli`).

Then, whenever we need to run Angular CLI command outside the already existing Angular workspace, we can use `npx @angular/cli <command>` instead!

The downside of this solution is that we always have to prefix the Angular CLI call inside our Angular workspaces with `npx` so it will look like this `npx ng <command>` ...

Following content of this book is written with global Angular CLI installation in mind, so it will use the `ng <command>` pattern instead.

Create new Angular workspace

Now is time to create our new Angular workspace. In the terminal of choice, we're going to navigate to a folder where we store our projects.

Run `ng new angular-architecture-example --create-application false` and wait until the process finishes.

Let's enter our newly created project workspace folder using `cd angular-architecture-example` and open the folder in the preferred editor (IDE).

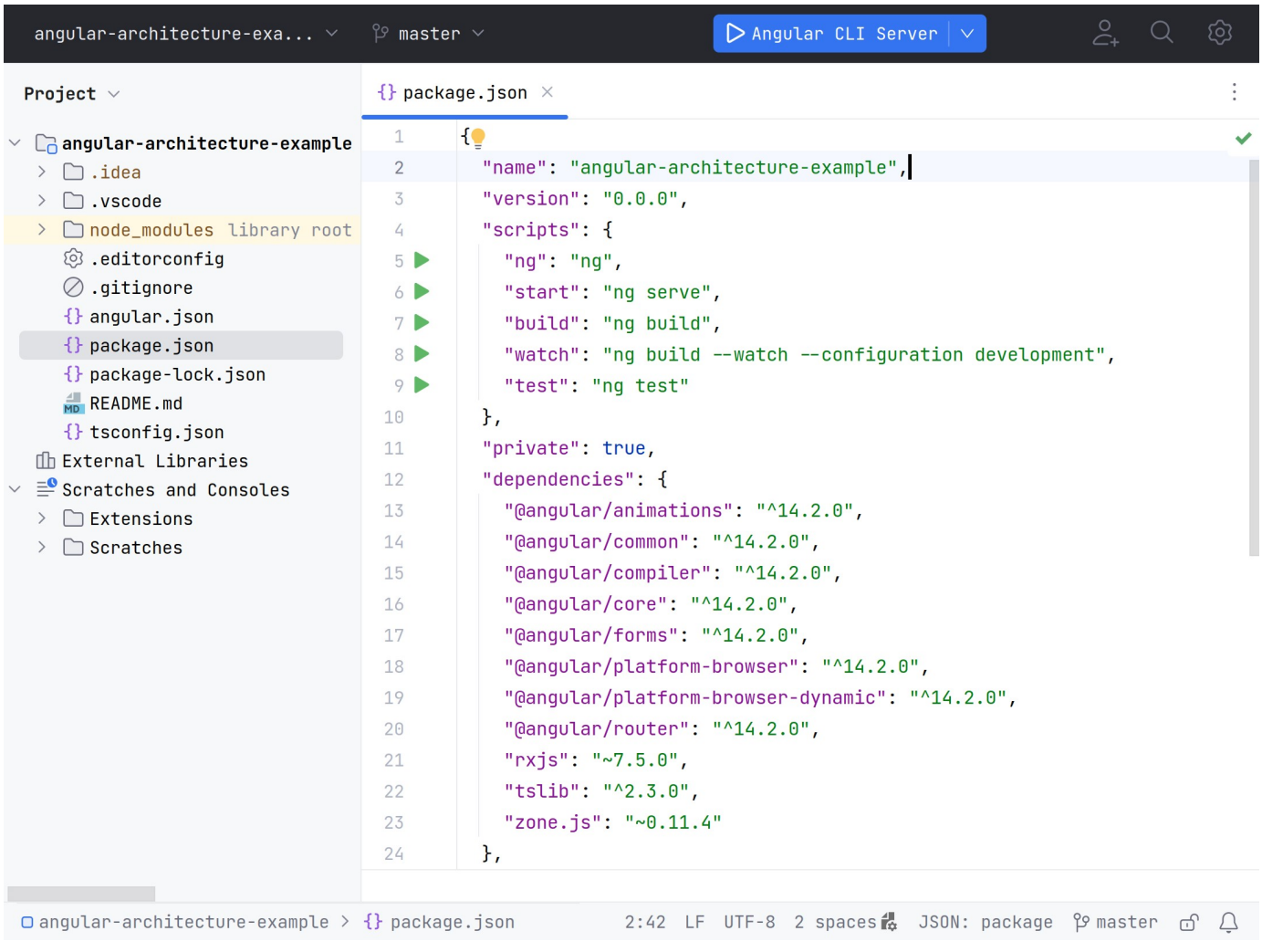
IDE — integrated development environment, usually a coding editor with extensive capabilities

When developing Angular application it is strongly recommended to use one of the two most popular IDEs:

- VS code — a great **free** open source editor developed by Microsoft which needs to be enhanced by the following essential extensions to support fully featured Angular development
- Intellij WebStorm — as o now the most powerful IDE with amazing capabilities and best developer experience out of the box (paid)

Create new Angular workspace

Let's explore our newly created empty Angular workspace



The workspace is currently empty but contains three important files which will be relevant for the whole lifetime of the project:

- `package.json` - list of all dependencies and **npm scripts**
- `angular.json` - workspace configuration for all the tasks provided by Angular CLI like `serve`, `build` or `test` and specification of Angular Schematics defaults
- `tsconfig.json` - root TypeScript configuration with possibility to override `angularCompilerOptions` to define various levels of strictness

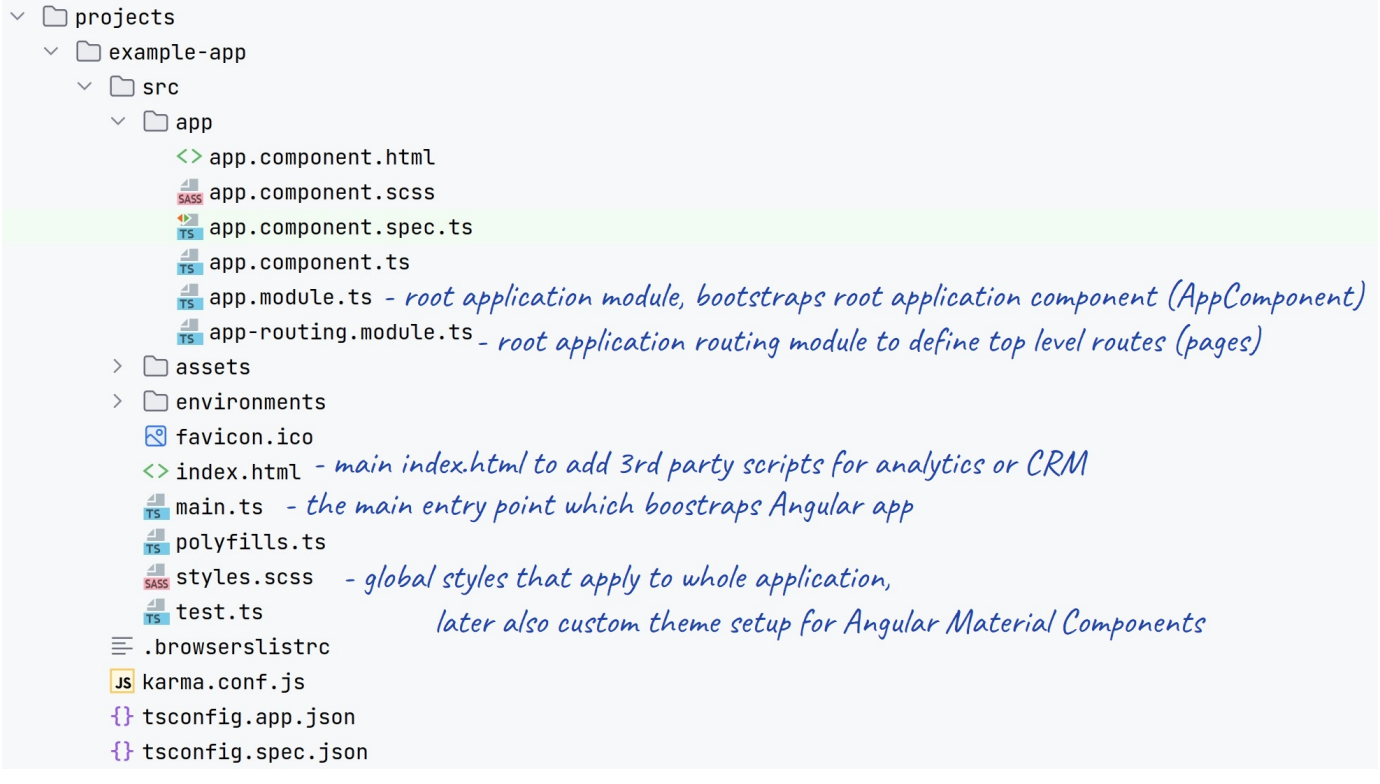
Create Angular application in the workspace

With the workspace ready, now it's time to create our first application by running following command in the workspace folder in our terminal.

```
ng g app example-app --prefix my-org --style scss --routing
```

The `example-app` will be created in the `projects/` folder. It will have routing capabilities and use integrate Angular `RouterModule` out of the box, and it will be using Sass styles (with `.scss` file extension).

> The `--prefix` will be used in the name of every generated component tag and directive selector so we will get `<my-org-user-list>` when we generate new user list component which is great because we can easily differentiate between our and 3rd party components!



Angular Schematics

> A schematic is a template-based code generator that supports complex logic. It is a set of instructions for transforming a software project by generating or modifying code. Schematics are packaged into collections and installed with npm — Official documentation

In previous steps, we have successfully generated whole Angular workspace and application with the help of Angular Schematics using just two commands!

This should illustrate how powerful these schematics are and give us a hint about how they are going to help us when implementing actual features for our users.

Angular CLI comes with the `@schematics/angular` collection out of the box.

Besides generating workspace and application, this schematics collection provides schematics to generate all Angular entities like `component`, `directive` or `service`, and even more advanced use cases like generating a whole lazy loaded `module` with a single command!

Running schematics

Schematics are executed in the terminal. We can list all available schematics by running `ng g help` command in the root of our workspace.

```
ng generate

Generates and/or modifies files based on a schematic.

Commands:
  ng g <schematic>        Run the provided schematic.                    [default]
  ng g application [name] Generates a new basic application definition in the "projects"
  ng g class [name]       Creates a new, generic class definition.       [aliases: cl]
  ng g component [name]   Creates a new, generic component definition.   [aliases: c]
  ng g directive [name]   Creates a new, generic directive definition.   [aliases: d]
  ng g enum [name]        Generates a new, generic enum definition.      [aliases: e]
  ng g guard [name]       Generates a new, generic route guard definition.[aliases: g]
  ng g interceptor [name] Creates a new, generic interceptor definition.
  ng g interface [name]   Creates a new, generic interface definition.   [aliases: i]
  ng g library [name]     Creates a new, generic library project.        [aliases: lib]
  ng g module [name]      Creates a new, generic NgModule definition.    [aliases: m]
  ng g pipe [name]        Creates a new, generic pipe definition.        [aliases: p]
  ng g resolver [name]    Generates a new, generic resolver definition.  [aliases: r]
  ng g service [name]     Creates a new, generic service definition.     [aliases: s]
  ng g web-worker [name]  Creates a new, generic web worker definition.
```

With this in mind, we would run `ng g s core/auth` to generate new `AuthService` in the `projects/example-app/src/app/core/` folder.

```
CREATE projects/example-app/src/app/core/auth.service.spec.ts (347 bytes)
CREATE projects/example-app/src/app/core/auth.service.ts (133 bytes)
```

Schematics by default generate entity with respective test stub which saves even more time!

Configuring schematics

In previous step, we have created a new Angular application in our workspace. Doing so added its configuration to the `angular.json` file.

This configuration allows us to configure schematics for this application by specifying default options.

```
{
  // angular.json, simplified for brevity
  "projects": {
    "example-app": {
      "schematics": {
        "@schematics/angular:component": {
          "style": "scss",             // we used --style scss when generating app
          "displayBlock": true,        // add display: block styles to component by default
          "changeDetection": "OnPush"  // use OnPush change detection strategy
        },
        // configuration for other schematics like "@schematics/angular:service" ...
      },
    }
  }
}
```

It's always a good idea to predefine set of commonly used Angular schematics options in the `angular.json` file to achieve unified approach and code style in our project.

It's a good idea to enable options like `--change-detection OnPush` (performance) and `--display-block` (component styles) for the `@schematics/angular:component` schematics!

3rd party schematics and ng add

Besides the default `@schematics/angular` which is a schematics collection that is included out of the box when using Angular CLI, there are many other schematics collections provided by popular Angular libraries like Angular Material or NgRx.

These schematics collection then allow us to generate entities which are specific to the given library.

For example, we might generate `reducer` or whole state feature when working with NgRx, or we could scaffold app navigation when working with Angular Material.

Besides these, libraries often come with a special `ng-add` schematic which is used to automate the process of integration of the library into our Angular workspace and application.

We're going to see this in action in later chapter when we're going to add Angular Material Components library to our workspace!

Schematics IDE integration

Schematics are a great way to boost productivity and consistency of our development process be it for a single developer, team or even a whole engineering organization.

As our projects grow, they become more complex with deeper nested structure of folders to group features and related sub features into manageable chunks.

This nesting of the folder structure leads to a longer paths we would have to provide for a schematics.

For example, let's say we want to generate a new `user-item` component in the `user-admin` sub-feature of the `user` feature, with `ng g c features/user/user-admin/user-item`.

This is quite some typing on our side and has a plenty of opportunity to make a typo or forgetting an exact path and skipping a part of it which would then lead to need to clean up and try again.

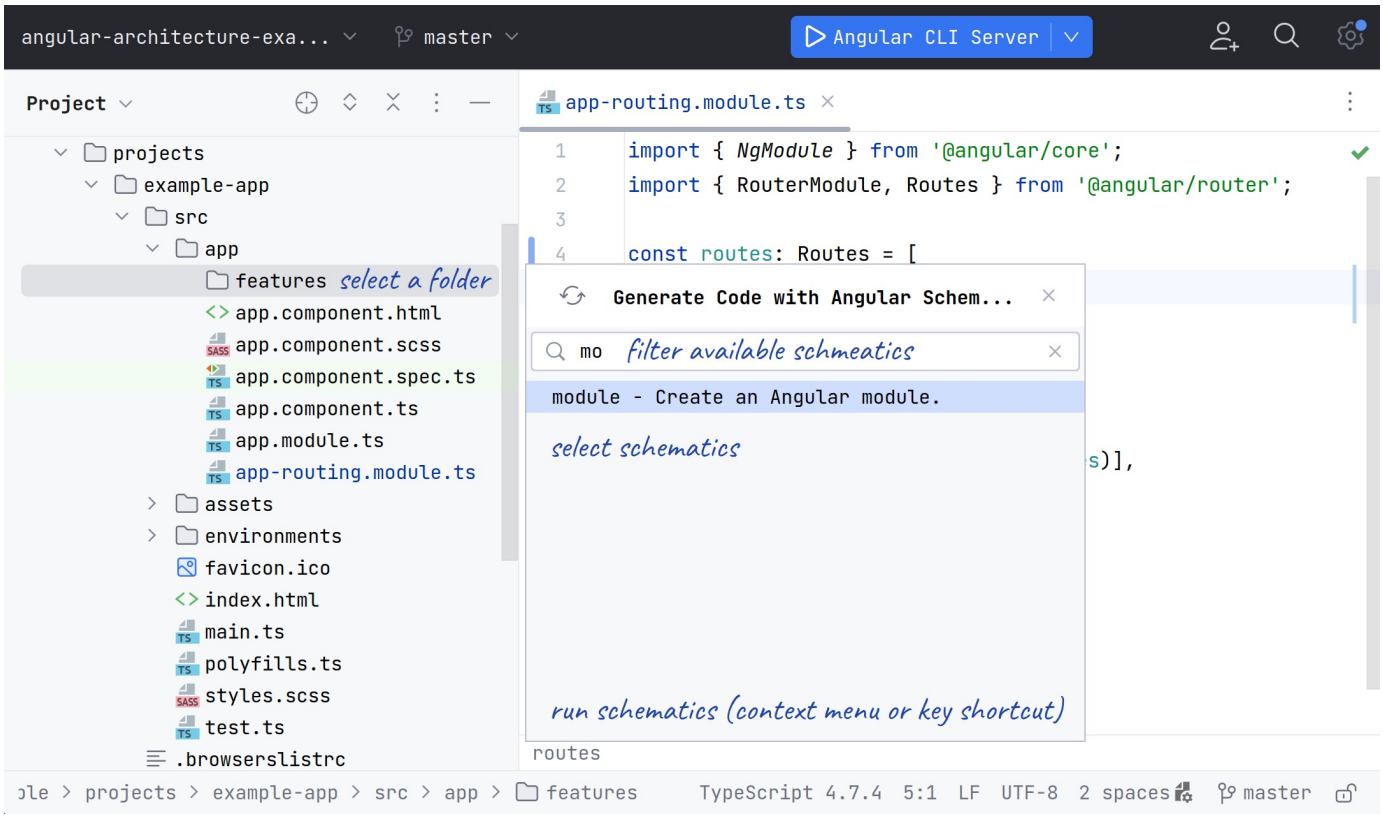
Running schematics manually in the terminal in large projects can become problematics because of the need to provide extensive folder paths manually.

Fortunately, there is a better way!

Schematics IDE integration

When working on an Angular projects, we tend to spend most of the time between our IDE where we write actual code and the browser to see our changes live.

Most popular IDEs like Webstorm or VS code have Angular schematics integration which allows us to run them on a selected folder directly in the IDE and hence remove the need to specify folder path manually.

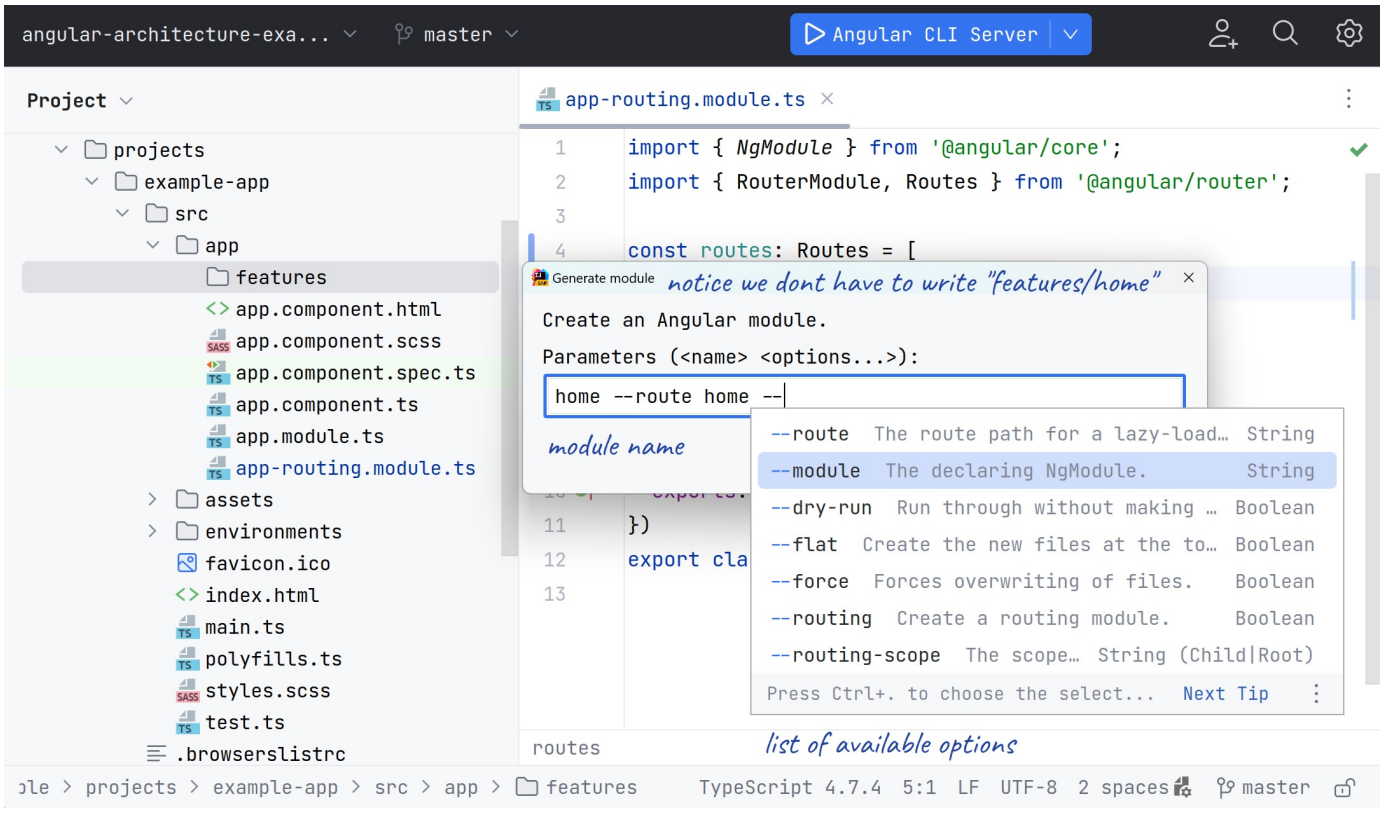


Make sure you add a key shortcut to run schematics on selected folder to streamline your workflow even further.

Schematics IDE integration

Once we selected desired schematics we can parametrize it further.

More so, this integration also provides code completion for all the available options for selected schematics increasing DX even further!



The schematics then run in selected folder path removing the need to specify long path manually.

> **This process scales well to projects of any size!**

> **DX** – developer experience

Code quality and developer experience tooling

In general, we always want to automate every possible task in our development! This is especially true when it comes to code quality and developer experience.

> Automating repetitive mundane tasks related to the development process allow us to **focus our attention on real problem-solving** and implementing useful features for our users and customers!

Angular makes this easy as it's one of the core tenets of its ecosystem of tools and libraries.

Besides Angular schematics which allow us to scaffold new entities like components, services or whole lazy loaded feature modules which follow all the latest best practices and code style there are additional areas which can be automated like code style and enforcement of best practices in regard to the code quality itself!

In the following chapter, we're going to explore most effort efficient tools at our disposal to achieve this goal!

Prettier

> Prettier is an opinionated code formatter that supports many languages and integrates with most editors and provides only a few options — Official documentation

Prettier is one of the most helpful things that happened to frontend development. It enables us to get **perfectly** and **consistently** formatted code just by pressing a couple of keys to trigger auto-formatting in our IDE.

We can even format whole project with help of a short npm script and automate the whole endeavor with so that the code is auto formatted on each commit to the repository!

Let's add it using `npm i -D prettier`. Once done, we will create a `.prettierrc` file which allows us to customize a couple of formatting options.

```
{
  "singleQuote": true,
  // you may consider ...
  "htmlWhitespaceSensitivity": "ignore"
  // prevents hard to read HTML template formatting
}
```

We can also add two useful npm scripts into the main `package.json` file that will help us format whole code base and test if it's properly formatted.

```
{
  scripts: {
    "format:test": "prettier \"projects/**/*.{ts,html,md,scss,json}\" --list-different",
    "format:write": "prettier \"projects/**/*.{ts,html,md,scss,json}\" --write"
  }
}
```

Eslint

> ESLint statically analyzes your code to quickly find problems. It is built into most text editors, and you can run ESLint as part of your continuous integration pipeline — Official documentation

Angular workspace doesn't come with ESLint out of the box, but it's easy to add it with the help of @angular-eslint/schematics collection.

We will be prompted to add ESLint support the first time we run `ng lint`.

```
Cannot find "lint" target for the specified project.
You can add a package that implements these capabilities.

For example:
  ESLint: ng add @angular-eslint/schematics

Would you like to add ESLint now? (Y/n)
```

Once we confirm, the ESLint will be installed and its configuration integrated with the project in our workspace!

After that we can run `ng lint` command to check if our code is properly formatted and fix all the linting errors!

The `@angular-eslint/recommended` preset is enabled by default and represents a good starting point for most projects. Feel free to explore and enable other rules as well!

Other popular libraries like NgRx provide their own ESLint rules sets like @ngrx/eslint-plugin.

Bundle size limits & analysis

Bundle size — the size of JavaScript files produced by the build process — is one of the most important metrics in frontend development as it directly impacts application startup performance.

> ⚠ Bundle size matters also for users who have access to high-speed internet connection.
>
> Even if users could download app bundles instantaneously (or have them cached in their browsers), **parsing and executing** of large JavaScript bundles still takes considerable amount of time and blocks UI interactivity!

Angular CLI provides concept of bundle size budgets which can be specified in the `angular.json` file and serve as a unit test for the size of bundles produced by the build process. Budgets can help us uncover cases where we accidentally added a new dependency which considerably increased the bundle size!

```
{
  "projects": { // inlined for brevity (would be nested otherwise)
    "example-app.architect.build.configurations.production": {
      "budgets": [
        {
          "type": "initial",
          "maximumWarning": "500kb",
          "maximumError": "1mb"
        }
      ]
    }
  }
}
```

Bundle size limits & analysis

Sometimes we might need to be able to analyze the bundle size in more detail to figure out which dependencies are responsible for the increase in size of our bundles.

The best way to do this is to use source-map-explorer.

First, we need to install it with `npm i -D source-map-explorer`.

Then we can add the following npm scripts to analyze the bundle size in `package.json` file.

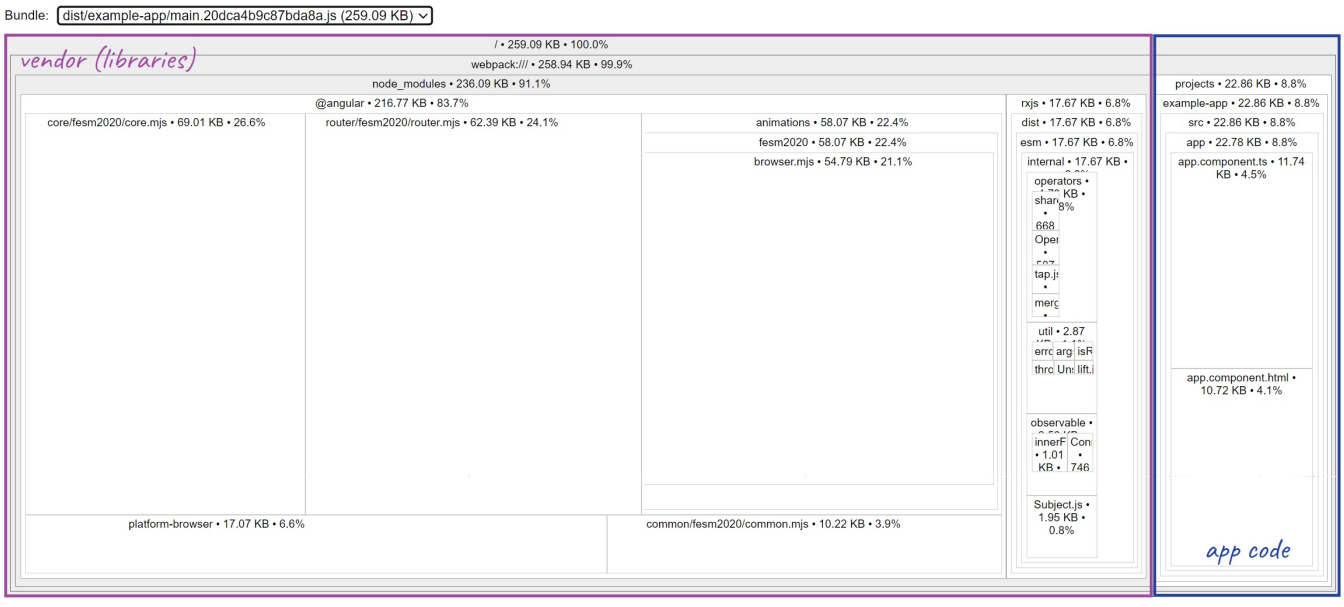
```
{
  scripts: {
    "analyze": "npm run analyze:build && npm run analyze:process && npm run analyze:serve",
    "analyze:build": "ng build --source-map",
    "analyze:process": "source-map-explorer dist/example-app/*.js --html report.html",
    "analyze:serve": "npx http-server report.html -o"
  }
}
```

The process was split into multiple steps to better understand what is going on but can be inlined into a single npm script with chain of multiple commands connected with `&&` operator.

Once prepared, we can execute our new npm script with `npm run analyze`. The script will build the application, analyze the bundle size and serve the report in the browser.

Bundle size limits & analysis

Let's explore the analysis report generated by `source-map-explorer`!



Our freshly generated application doesn't contain any business logic yet so the size of the "blue part" (application code) of the chart is relatively small compared to the "purple part" which represents the size of base dependencies like Angular framework and RxJs library.

As our application grows, this ratio will change and the application code will represent the majority of the bundle size.

> In the future chapters, we're going to explore a concept of lazy loading which will allow us to postpone loading of most of our business features until they are actually needed by the user which leads to smaller initial bundle size and faster application startup time! We're also going to explore what kind of impact this will have on the source map explorer chart!

Angular Material Components

Let's make our life easy by using already existing component framework. We can use Angular Material which comes with many high quality, **accessible** and beautiful components out of the box.

We will add it using Schematics by running `ng add @angular/material`.

This will install the library using npm, and we will be prompted to provide values for a couple of initial options:

- `custom theme` - will enable us to use our own brand colors with ease
- Material typography will make our app look consistent
- animations will make Angular Material look fabulous

Once finished we can start importing and using Angular Material components in our app but more on that later...

When developing apps with Angular it is very common that the heavy lifting is done with help of Angular Schematics which leaves us roo to focus on what really matters, **delivering features to our users!**

Angular Material Components

The terminal output will look something like the following...

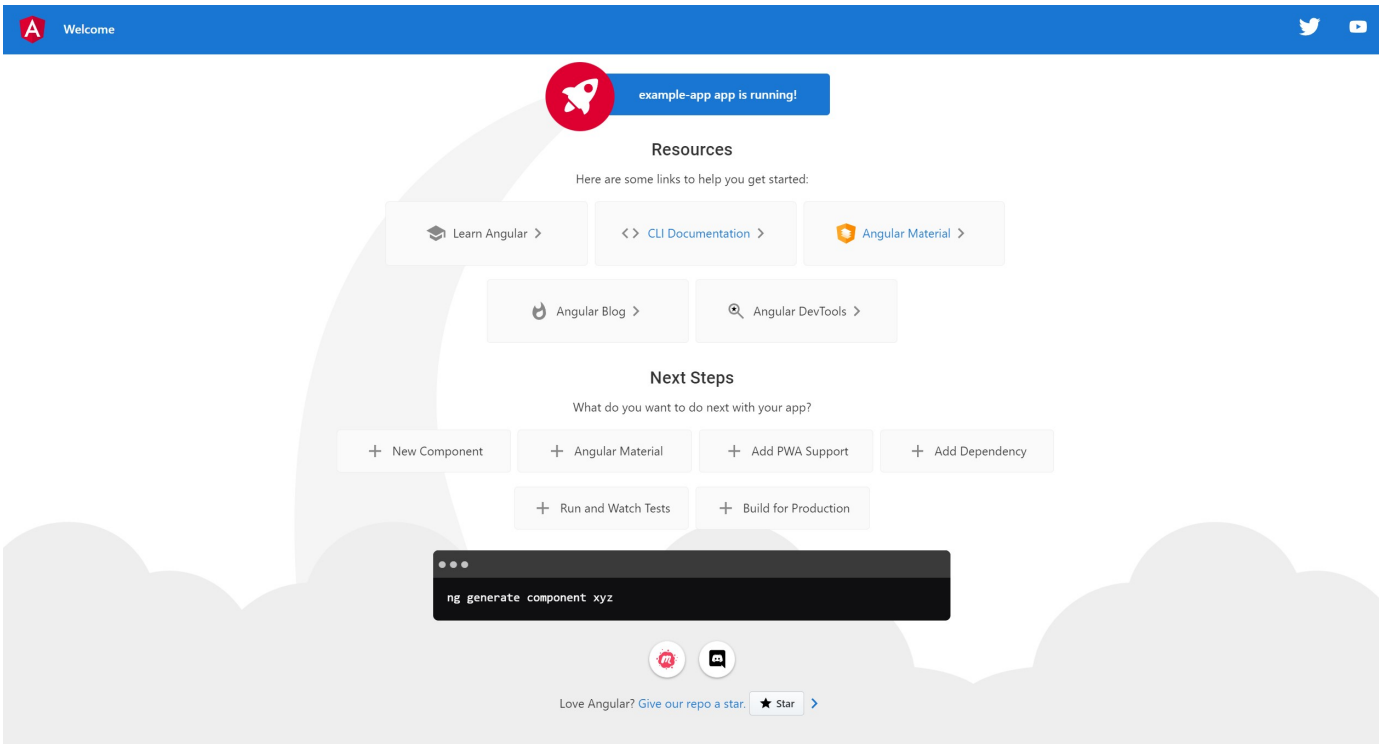
```
ℹ Using package manager: npm
✔ Found compatible package version: @angular/material@14.2.4.
✔ Package information loaded.

The package @angular/material@14.2.4 will be installed and executed.
Would you like to proceed? Yes
✔ Packages successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme: Custom
? Set up global Angular Material typography styles? Yes
? Include the Angular animations module? Include and enable animations
UPDATE package.json (1125 bytes)
✔ Packages installed successfully.
UPDATE projects/example-app/src/app/app.module.ts (502 bytes)
UPDATE projects/example-app/src/styles.scss (1644 bytes)
UPDATE projects/example-app/src/index.html (584 bytes)
```

The schematics have updated a couple of files which represents the Angular Material integration with our app.

- `package.json` - added `@angular/material` and `@angular/cdk` dependencies
- `app.module.ts` - added `BrowserAnimationModule`
- `style.scss` - add custom theme setup with lots of comments on how to adjust it
- `index.html` - added links to used fonts and enabled typography on the `<body>` tag

Final cleanup

Let's run our app using `npm start` or `ng serve -o`. The `-o` flag stands for "open" which will open browser with the correct URL automatically once the app is ready...



Angular CLI generated initial content with tips and links to documentation.

It is highly recommended to add these links into our bookmarks. Because of the large API surface of the Angular and related libraries, it is very common to keep coming back to these docs even if we're working with it on daily basis.

> Now we can delete whole initial layout from a single place in the `app.component.html` file where it was all inlined, so it can be removed with ease!

Initial setup wrap-up

> Our Angular workspace is now prepared including all the relevant productivity and code quality tooling to make development process a breeze!

Now we can start working on our application creating an enterprise grade architecture that will support us during the whole application lifecycle by allowing us to evolve and extend features to accommodate for ever-changing requirements and business needs!

Introduction

When speaking about enterprise grade or enterprise scale application development we need to consider a lot of different aspects that are not always necessary for smaller teams or organizations.

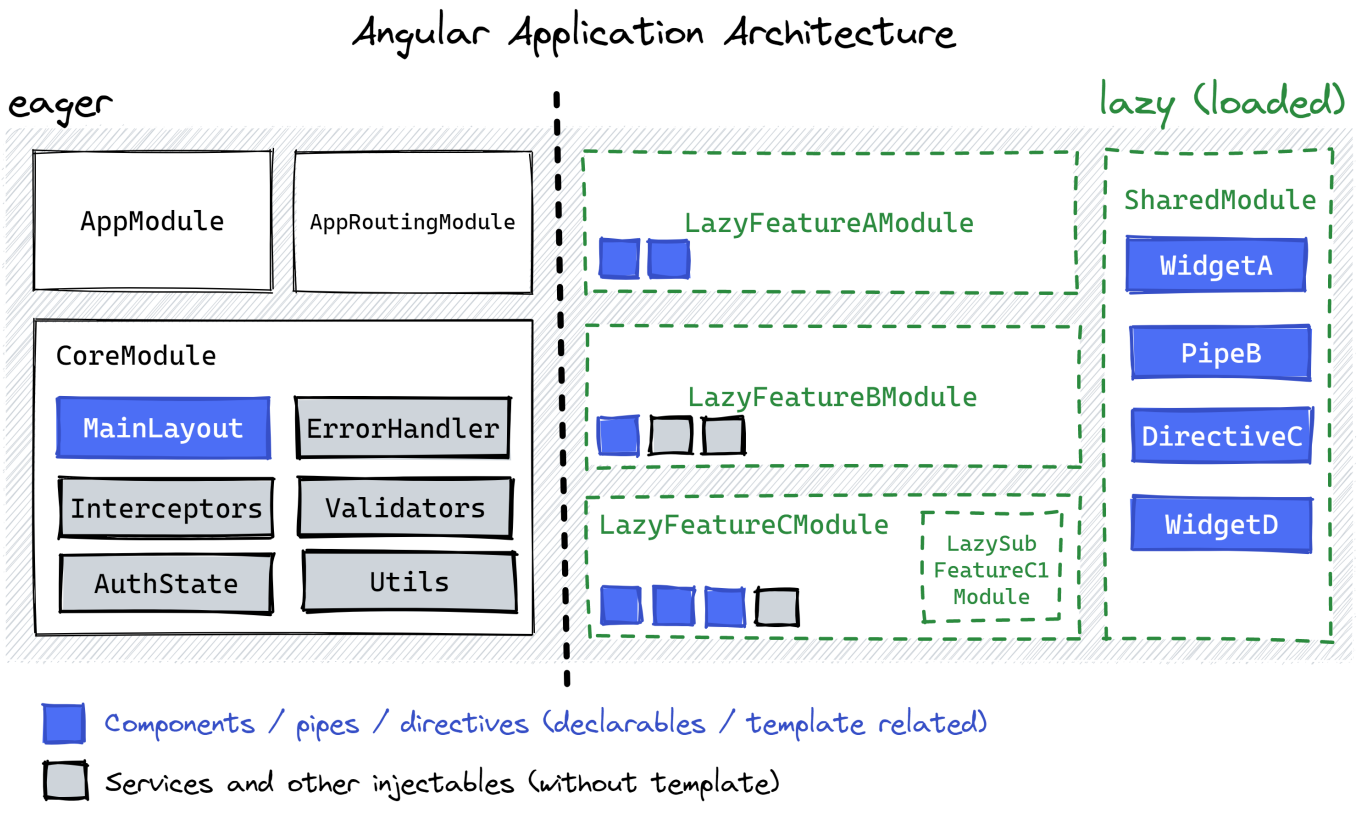
> In other words, developing ad hoc ERP system for a small to medium-sized company has a different set of constraints compared to developing a one of the multiple frontend applications for an underlying heterogeneous enterprise system of a large organization. With large organizations, we often find multiple teams taking care of different part of the business domain.

Proposed architecture is designed to fulfill following requirements:

- **(fully) isolated features** - each team should be able to work on their own features without stepping on each other toes
- **modularity** - ability to extend and evolve features without breaking existing ones
- **architecture scalability** - ability of application architecture to scale to arbitrary amount of features (and teams) without degrading performance for the end users and delivery speed for development of new features
- **fractal architecture** - architecture patterns which are repeatable across ever smaller pieces of logic, in our case features which can have arbitrary amount of nested sub features and nesting levels following same patterns
- **standardization** - set of best practices (vendor and custom schematics) to ensure consistency across the whole application
- **pragmatic reuse** - ability to reuse code and logic in a way which prevents excessive coupling and therefore guarantees future maintainability

The big picture

Before we start generating modules and components using Angular schematics and writing application implementation, let's take a step back and see the bigger picture, literally...



This architecture diagram expresses the high level overview of the proposed architecture which fulfills all the previously listed requirements.

Every Angular application has two main parts:

- **eager** - the part of the application which is loaded on application startup
- **lazy** - the part of the application which is loaded on demand as a result of user interaction, in Angular the laziness is usually implemented on a route (page) level, so it's result of user navigating to a specific feature

Declarables vs injectables

Before dwelling into the details of the eager part, let's take a look recapitulate the difference between *declarables* and *injectables* which represents a corner stone concepts of Angular DI.

> **DI** — Dependency Injection

In Angular there are two main types of entities:

- **declarables** - entities which have template (components) or can be used in the templates of components (directives, pipes) and any arbitrary combination of thereof
- **injectables** - entities like services, route guards or interceptors which can be injected into other entities using constructor injection (older) or using `inject()` (newer), their key distinctive feature being absence of the own template and being unrelated to the concept of template in general

Template context

Template context is a very useful mental model when working with declarable. It boils down to the need to "collect" all the declarable that are used in the template of a given component and make them available in the module where the component is declared.

This can be achieved either **directly**, the declarable are part of the `declarations: []` array of the same module or **transitively** when declarable are **exported** by some other module which is then added to the `imports: []` array of the module where the component is declared.

Declarables vs injectables

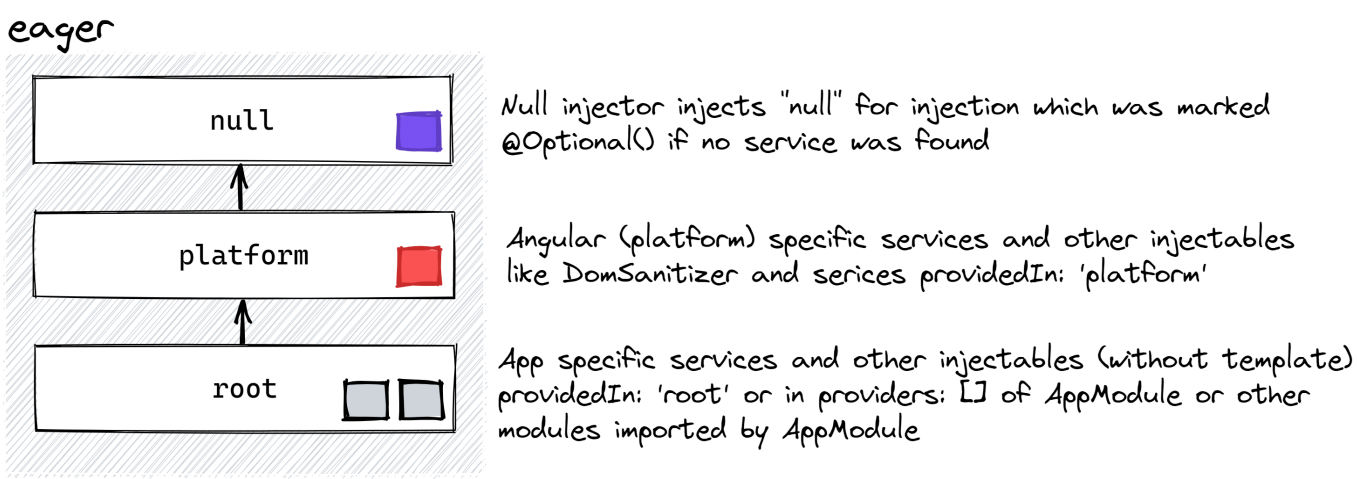
A component can be declared only in a single module in a single `declarations: []` array. To use this component in other template context it would be necessary to add it to `exports: []` of the owning module and then import the owning module in a target module.

Injector hierarchy

In Angular, injectables follow their own DI system implemented with the concept of **injector hierarchy**.

> From the practical point of view, injector hierarchy determines how many instances of a given injectable (e.g. service) will be created and which instance will be injected into specific consumer like a component.

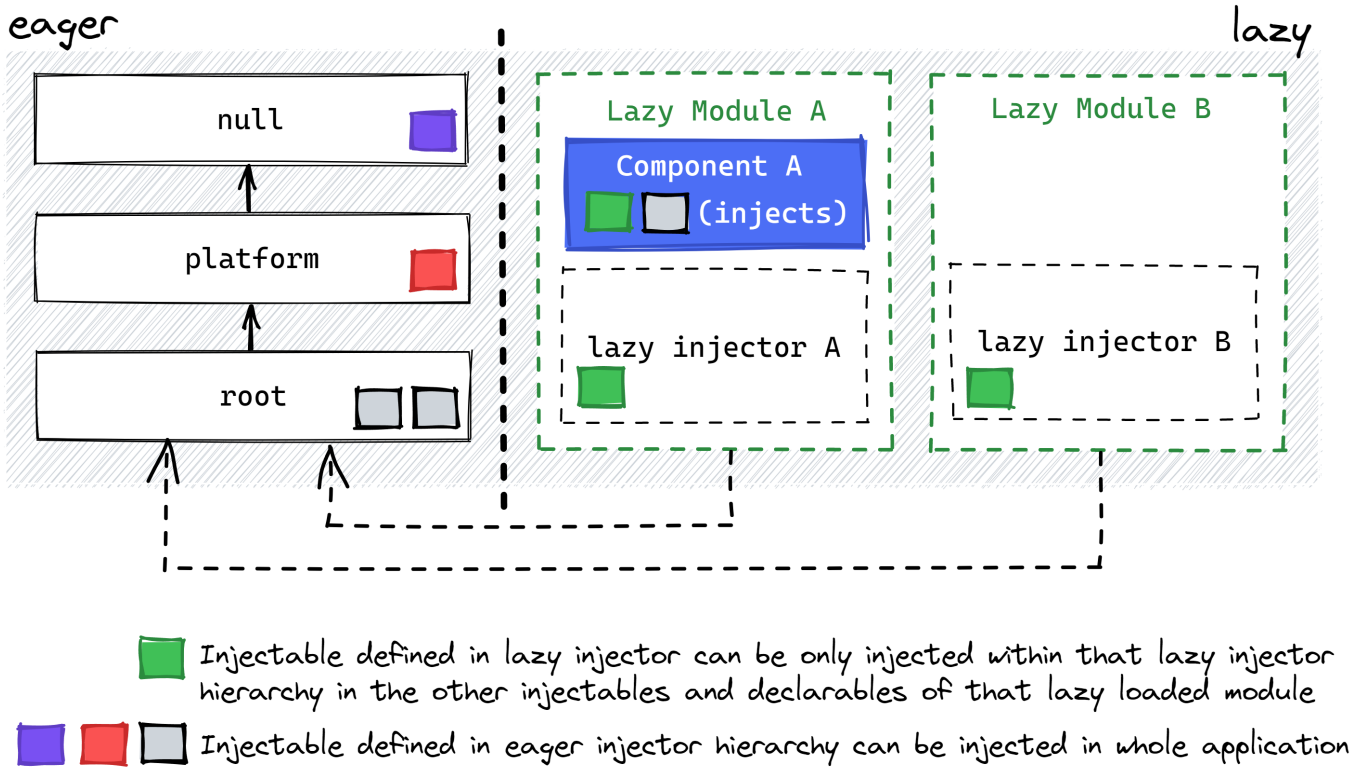
The eager part of Angular comes with three main injectors, two of which are hidden behind the main one, the `root injector`.



Declarables vs injectables

All three injectors in the eager part of the application represent single **eager injector hierarchy** and the injectables registered in these injectors will be **application wide singletons** or in other words, there will be **only one instance** of the given injectable so whoever (including lazy loaded features) injects it will get the same instance.

Let's explore what happens when we start adding lazy loaded modules.



Adding a lazy loaded module creates its own **isolated lazy injector hierarchy**.

This has couple of very important consequences:

1. lazy loaded modules (features) can be **fully isolated** from each other and the eager part of the application, this is they key benefit of the architecture
2. lazy loaded modules (features) can access injectables which belong to eager injector hierarchy which enables access to core services

Declarables vs injectables

3. providing the same injectable in multiple lazy loaded injectors will result in multiple instances of the given injectable (one per injector)
4. providing the same injectable in both lazy and eager injector hierarchy will result in multiple instances of the given injectable (one per injector)

In practice, we're going to provide our injectables in two ways:

1. **application wide singletons** - in the eager injector hierarchy using `@Injectable({ providedIn: 'root' })`, in `core/` folder (could be further structured by feature)
2. **lazy feature scoped injectable** - in the `providers: []` array of the lazy loaded module

Other injectors

There are other even more fine-grained injectors like `ElementInjector` which allow us to create unique instance of a given injectable for each instance of component on which it was declared.

These solve other, more specific use cases and are therefore not as relevant for the topic of overall Angular application architecture.

Declarables vs injectables

Constructor vs inject based injection

With version 14, Angular exposed new `inject()` function as an alternative to standard constructor based dependency injection. The main benefit of `inject()` is that it enables full type safety when working with Angular injection tokens.

Besides that, there has been a recent mismatch between the direction of evolution of the JavaScript and TypeScript,especially for specifying of the class fields. Angular constructor injection uses TypeScript based syntax sugar for declaring and initializing of the class fields which is not supported by the new JavaScript standard version.

```
// constructor based injection
export class UserComponent {
  constructor(private userService: UserService) {}
}

// is a shorthand version of
export class UserComponent {
  private userService: UserService;

  constructor(userService: UserService) {
    this.userService = userService;
  }
}
```

To make the code compatible with the new JavaScript standard, we would need to set `"useDefineForClassFields": false` flag in the `tsconfig.json`.

On the other hand, `inject()` based injection doesn't have this problem.

```
// inject() based injection
export class UserComponent {
  private userService = inject(UserService);
}
```

Eager vs lazy parts of the application

Eager part

Eager part of Angular application always contains root `AppModule` and corresponding `AppComponent` which is the entry point of the application which is bootstrapped by Angular at startup.

Besides that, the eager part contains `AppRoutingModule` which defines to level lazy routes of the application.

Route configuration can be also defined in the `AppModule` but is often extracted to a separate file to improve readability and maintainability of the application.

The main body of the eager part of the application is the `CoreModule` (often implemented in the `core/` folder) which will contain implementation of the core features of the application like:

- **main layout** - the layout which is used by all the features of the application usually consisting of header, footer, main navigation, signed-in user avatar and menu etc...
- **core infrastructure** - any infrastructure related code like interceptors, guards, error handlers etc...
- **app wide services** - any services, utilities, transformers which are used by whole application (eager and lazy parts)
- **eager state** - application state that needs to be present from application startup, such state is often consumed also by the lazy features

Eager vs lazy parts of the application

- **"shared" state** - any state that needs to be shared between multiple lazy features (even if it's not directly used in the eager part of the application)

The `CoreModule` is necessary only for registering declarations of the components related to main layout (and navigation) and various other modules and providers which need to be overridden or parametrized. But we will **not** use it to provide app wide singleton services!

Always use `providedIn: 'root'` for registration of app wide singleton services instead of providing them in the `providers: []` array of the `CoreModule`. The service files should be stored in `core/<feature-name>/` folder to provide better organization of the code.

Lazy (loaded) part

This is the part of the application which implements actual business (or user) related features and functionality.

Every such feature will be loaded as a result of user navigating to a specific route as the lazy loading is implemented on the route (or page) level.

It is also possible to lazy load components in Angular but this solves another set of challenges and is not relevant for the overall application architecture!

Eager vs lazy parts of the application

Such navigation, for example as a result of user clicking on a button in the top level navigation, will then trigger loading of the separate lazy JavaScript bundle which contains implementation of the feature.

It is always a good idea to implement **all your features as lazy loaded features** including initial feature (like dashboard or home page) can be then activated automatically using router config without the need for user to manually navigate!

The unit of lazy loaded part of application is a **lazy loaded module**. From here on we will refer to such module as **lazy feature** or **lazy module**.

The lazy module is a regular Angular module which is **not imported** using a direct import and reference in the `imports: []` array of another application Angular module (like `AppModule` or `CoreModule`).

Instead, the lazy module is **imported** using a dynamic `import()` statement in the `loadChildren` property of a respective route configuration.

The `import()` function is a native dynamic JavaScript import which is then processed during the build process by the bundler and represents a lazy boundary which tells bundler to bundle such module in its own separate JavaScript bundle (file).

Eager vs lazy parts of the application

The lazy loaded module then contains components, directives, pipes, services and other available Angular entity types which implement the actual business feature.

More so, **lazy loaded module also contains its own routing module** which defines sub-routes of the feature which allows us to implement simple sub-pages like "edit" or "details" or even full-blown nested navigation to reach other lazy sub features of the main lazy loaded feature.

> The fact that the lazy loaded modules have their own routing modules which allow them to have lazy loaded submodules gives us a hint that the proposed architecture is **fractal** and can be repeated for as many nested level as necessary to fulfill any business requirements!

Every lazy feature is **completely isolated** from other features which is the key property which makes this architecture so powerful! We're going to discuss the benefits of full isolation from every relevant angle as we go through the rest of the book.

In practice, we will always create top level lazy loaded feature modules using Angular schematics called `module` by running `ng g m features/<feature-name> --route <route-name> --module app` (or even better using schematics integration in our IDE).

Notice that the lazy feature modules are located in the `features/` folder to provide better organization of the code.

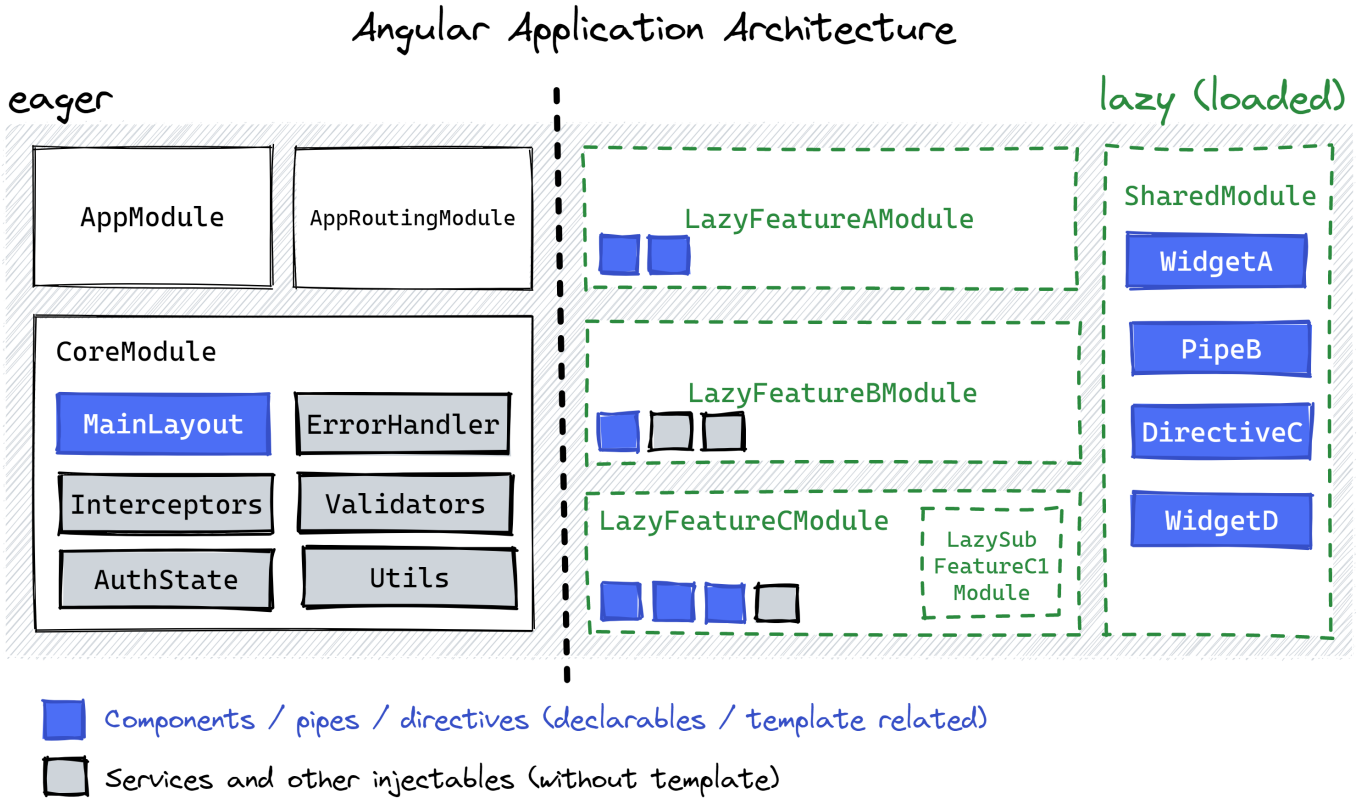
Hands on architecture

Now we should have a good understanding about the main parts of the architecture we're going to implement it in practice!

We're going to create empty application skeleton with all the parts we've discussed so far.

1. `CoreModule` - with main layout and navigation
2. `LazyFeature<X>Module` - couple of lazy loaded features to illustrate the concept
3. `SharedModule` - with shared components, directives, pipes (declarables only) which are only used by lazy features

> Let's refresh our memory about the overall architecture, so we see how these parts fit together!



The CoreModule

Let's start by creating a new `CoreModule` in the `core/` folder by running `ng g m core` or right-clicking the `app/` folder and running `module` schematics in our IDE.

In both terminal or IDE we are going to see the following output:

```
CREATE projects/example-app/src/app/core/core.module.ts (190 bytes)
Done
```

The generated `CoreModule` will have the following content:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common'

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class CoreModule {}
```

and we're going to adjust it by:

1. add `BrowserModule` and `BrowserAnimationsModule` to the `imports: []` array
2. remove `CommonModule` from the `imports: []` array

> The `BrowserModule` re-exports everything that is provided by the `CommonModule` and hence we can remove it as it becomes redundant.

The CoreModule

The adjusted `CoreModule` will look like this:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  declarations: [],
  imports: [
    // vendor
    BrowserModule,
    BrowserAnimationsModule
  ]
})
export class CoreModule { }
```

Notice the `vendor` comment in the `imports: []` array. It can be a good practice to group imports by type, for example vendor, shared, core, local to be able to visually scan and get a good overview of what is used by a module, service or component!

Importing the `CommonModule` (and hence also `BrowserModule`) allows components that will be declared in the `CoreModule` to use many of the directives like `*ngIf`, `*ngFor` or `ngClass` provided by Angular itself.

There is nothing special about the Angular built in components and directives. This means we have to import the module which exports them in order to be able to use them in our own template context.

The CoreModule

Next up, we have to open `AppModule` and import and add `CoreModule` to the `imports: []` array of the `AppModule`.

```
1   import { NgModule } from '@angular/core';
2
3   import { CoreModule } from './core/core.module';
4   import { AppComponent } from './app.component';
5   import { AppRoutingModule } from './app-routing.module';
6
7   @NgModule({
8     declarations: [AppComponent],
9     imports: [
10      CoreModule, // we have removed BrowserModule, BrowserAnimationsModule
11      AppRoutingModule,
12    ],
13    providers: [],
14    bootstrap: [AppComponent]
15  })
16  export class AppModule { }
```

The `AppModule` will stay virtually empty for the whole life-time of the project, everything that needs to be available from start will be added to the `CoreModule` instead.

That way we will always know that there just one place where to find all the eager logic of the application instead of splitting in randomly between the `AppModule` and the `CoreModule`.

> Great, our `CoreModule` is set up and ready to be used. Now we can continue by focusing on implementation of the main application layout which will be visible for every lazy feature and right from the application startup!

Creating main layout

Now it's time to create the main layout which will consist of header with main navigation, footer and the content where we are going to render currently active lazy feature.

Navigation toolbar implemented using components provided by Angular Material component library

We will implement it inside the `CoreModule`, specifically inside the nested `core/layout/` folder. The good thing is that we do not have to create these folders manually because they will be created automatically when running Angular schematics.

Let's create a "main-layout" component using `ng g c core/layout/main-layout --export`. It will be automatically added to both the `declarations: []` and `exports: []` the arrays of the `CoreModule`.

Once done, we are going to use it in the `app.component.html` by using its html tag `<my-org-main-layout></my-org-main-layout>` which is the same as the value of the selector inside the `main-layout.component.ts` file.

> The same could have been achieved by right-click `core/` folder and running `component` schematics in our IDE with the component name `layout/main-layout` which would create the component in the `core/layout/` folder automatically. Such approach is more convenient as we do not have to leave the IDE and we need to provide less information to achieve the same result.

Creating main layout

With our `main-layout` component in place, we can start implementing its content in its `main-layout.component.html` file.

```
<!-- component from @angular/material, MatToolbarModule -->
<mat-toolbar color="primary">

  My Epic App

  <span class="spacer"></span>
  <!-- mat-flat-button is a directive from @angular/material, MatButtonModule -->
  <button mat-flat-button color="primary" routerLink="home">Home</button>
  <button mat-flat-button color="primary" routerLink="admin">Admin</button>
  <!-- routerLink is a directive from @angular/router, RouterModule -->
</mat-toolbar>

<main> <!-- standard HTML element (semantic HTML) -->
  <router-outlet></router-outlet> <!-- component from @angular/router, RouterModule -->
</main>

<footer>Awesome Inc.</footer>
```

We're using quite some new components and directives like `<mat-toolbar>` or `routerLink` which are not yet part of the local template context (have not been imported in the `CoreModule`) so they are going to be highlighted as errors in the IDE.

Let's fix this error by importing the `MatToolbarModule` and `MatButtonModule` in out `CoreModule` and adding them to the `imports: []` array.

> This importing usually can be achieved automatically by running a "quick fix" feature of our IDE. It will automatically import the missing module and add it to the `imports: []` array.

Creating main layout

The resulting `CoreModule` should look like this:

```
@NgModule({
  declarations: [MainLayoutComponent],
  imports: [
    // vendor
    BrowserModule,
    BrowserAnimationsModule,
    RouterModule,

    // material       // added
    MatToolbarModule, // added
    MatButtonModule,  // added
  ],
  exports: [MainLayoutComponent]
})
export class CoreModule { }
```

Please notice helpful import grouping and comments like `// vendor` and `// material`. They are not mandatory but nice to have because they let your colleagues (or even you in the future) get a quick overview of the module structure compared to long randomly sorted list of imports!

Now we can improve the layout by adding some styles to the `main-layout.component.scss` file.

The styles are going to expand the `main` part to take the whole available screen height so the footer is going to be always at the bottom of the screen.

Creating main layout

```
:host {             /* the main-layout component itself */
  display: flex;
  flex-direction: column;
  height: 100vh;
}

mat-toolbar {
  .spacer {
    flex: 1 0 auto;
  }
  button {
    margin: 0 10px;
  }
}

main {
  flex: 1;
  display: flex;
  flex-direction: column;
  overflow: auto;
}
```

> Great, we have imported modules that export all the components and directives that we use inside our `main-layout.component.html` template and the application should be up and running again!

Introducing lazy features

With our main layout and main navigation in place, we can now focus on implementing the lazy features.

> We will generate lazy feature for every top level route of our application. This is a reasonable starting point but don't forget that it is possible to lazy load nested routes if the feature gets too big!

First, we will generate module for our home feature using `ng g m features/home --route home --module app` (or using IDE schematics) which should output following in the terminal and will do two things:

```
CREATE projects/example-app/src/app/features/home/home-routing.module.ts (335 bytes)
CREATE projects/example-app/src/app/features/home/home.module.ts (343 bytes)
CREATE projects/example-app/src/app/features/home/home.component.scss (28 bytes)
CREATE projects/example-app/src/app/features/home/home.component.html (19 bytes)
CREATE projects/example-app/src/app/features/home/home.component.spec.ts (585 bytes)
CREATE projects/example-app/src/app/features/home/home.component.ts (347 bytes)
UPDATE projects/example-app/src/app/app-routing.module.ts (346 bytes)
Done
```

1. generate module, routing module and component files in `/features/home` folder
2. add lazy route config for the `/home` to the main `app-routing.module.ts` file

Let's have a look in the `app-routing.module.ts` file. We should see our newly generated `home` route.

Introducing lazy features

```
const routes: Routes = [
  {
    path: 'home',
    loadChildren: () =>
      import('./features/home/home.module').then((m) => m.HomeModule),
  },
];
```

Let's add first "empty" route config which will redirect to the home lazy loaded feature in case now route is matched to the current URL.

```
const routes: Routes = [
  { // added empty route config
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  },
  {
    path: 'home',
    loadChildren: () =>
      import('./features/home/home.module').then((m) => m.HomeModule),
  },
];
```

This approach makes it possible to implement **all** of our features as lazy loaded features.

It's a good practice to implement every feature as a lazy loaded feature also in case when our current requirements are only a single feature (page or screen). This approach will make our application easy to extend when the requirements change and there is virtually no cost or downside to implement it this way.

Introducing lazy features

We can also add last “catch all” `**` route. This route will redirect to home in case user navigates to a route which does not exist.

```
const routes: Routes = [
  // other routes...
  // notice catch all route is last route in the config
  {
    path: '**',
    redirectTo: 'home',
  }
];
```

It is also possible to implemented dedicated “not found” lazy feature with more sophisticated info for the user based on what might have gone wrong (e.g. record was removed or user does not have access to it anymore) but for now, we are going to keep it simple.

The route configs are resolved in order. Always make sure that the "catch all" route is the last route in the config. Otherwise, as it will match all routes, the other routes that are configured after it will never be matched!

Now we can generate our second lazy feature (admin) using `ng g m features/admin --route admin --module app` (or in IDE). Notice that it was correctly added before “catch all” route by the schematics.

Introducing lazy features

Our application is now running, and we can navigate to the home and admin features using a top level navigation in the header. Let's see it in action!

Clicking on the navigation buttons should switch the displayed content to that of the currently active lazy feature.

> It is considered good UX to show user which feature is currently active by highlighting the corresponding navigation button.

Let's open `main-layout.component.html` file and add `routerLinkActive="active"` directive on both navigation buttons.

Then we can implement `.active` class in the `main-layout.component.scss` file, for example using `filter: brightness(<amount>);` css rule which is great because it does not force use to choose a specific color, so it will work with any theme.

```
button {
    /* previous styles... */
    &.active {
        filter: brightness(120%);
    }
}
```

Once done, it will look like the following...

Fractal nature of lazy features (nesting)

We have now implemented two **top level** lazy features (home and admin) and we can navigate between them using the top level navigation in the header.

The generated lazy features have their own routing module and their own routing config. This means that we can add nested routes to the lazy features.

For example, the generated routing module of our home feature looks like this.

```
const routes: Routes = [
  { path: '', component: HomeComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HomeRoutingModule { }
```

It comes with a single route config which will match the empty path and will therefore be activated when the user navigates to `/home` route as every route ends with an empty route in Angular.

As our feature grow, we might add additional **eager** routes, for example we might add a route for displaying a list of tasks and task details...

```
const routes: Routes = [
  {
    path: '',
    component: HomeComponent,
    children: [
      { path: 'task', component: TaskListComponent },
      { path: 'task/:id', component: TaskDetailComponent }
    ]
  }
];
```

Fractal nature of lazy features (nesting)

Such nesting works great and enables scaling of the individual feature to support new use cases.

At some point the feature might grow so much that it makes sense to split it further into multiple lazy sub-features. For example, we might want to split the home feature into two lazy sub-features: `task` and `dashboard`.

We can do it by generating new lazy features using the same approach as before. The only difference being the path of the sub features would contain parent feature folder.

In this case, we would generate `ng g m features/home/task --route task --module features/home` or just `task --route task --module .` if we ran schematics in the `home/` folder in our IDE.

Running schematics in IDE becomes more and more ergonomic as our application grows because it allows us to run schematics directly from a nested folder like `features/home/` and hence reduces the need to specify long path manually which is often an error-prone process.

Similar, we could then proceed with respective schematics for the dashboard feature, for example using `dashboard --route dashboard --module .` if we ran schematics in the `home/` folder in our IDE. Notice that the module points to the lazy loaded parent feature `--module features/home` instead of the previously used root module `--module app`.

Fractal nature of lazy features (nesting)

Once done, we will have two new lazy loaded **sub-features** of the home lazy loaded feature and the routing config will look like the following.

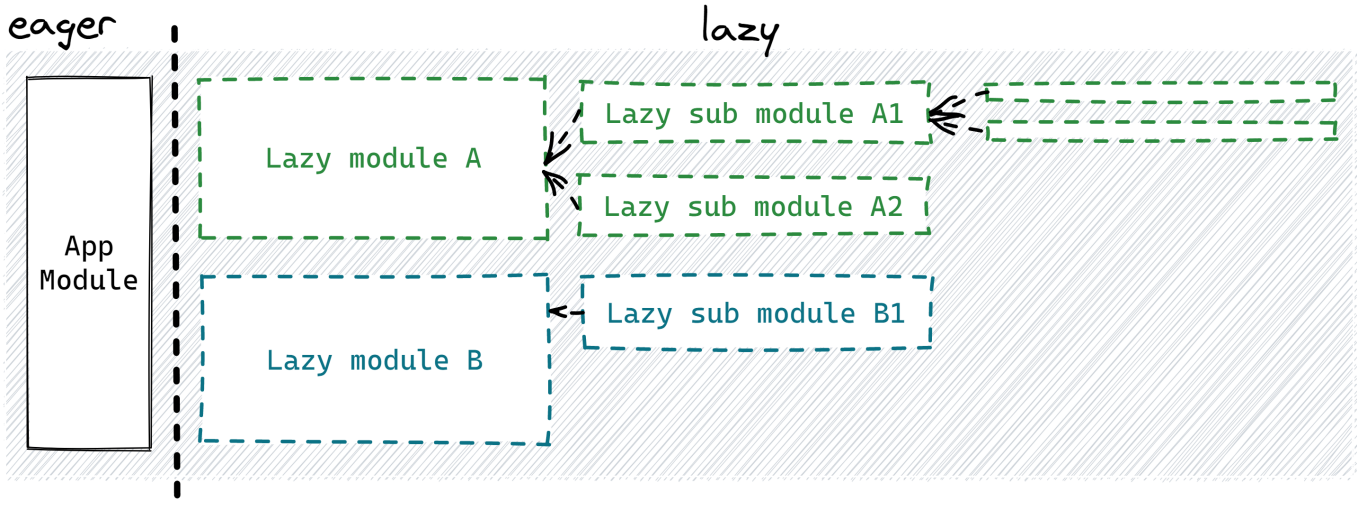
```
const routes: Routes = [
  {
    path: '',
    component: HomeComponent,
    children: [
      // lazy sub features as children of parent lazy route
    ]
  },

  // lazy sub features as siblings of parent lazy route
  {
    path: 'task',
    loadChildren: () => import('./task/task.module').then((m) => m.TaskModule),
  },
  {
    path: 'dashboard',
    loadChildren: () =>
      import('./dashboard/dashboard.module').then((m) => m.DashboardModule),
  },
];
```

Depending on our use case we have in general two main options how to handle this.

1. **lazy sub features are siblings of parent lazy feature route** - keep the config as it is because the lazy sub features should replace the parent lazy feature (home) in the view when navigated to
2. **lazy sub features are children of parent lazy feature route** - add `children` array to the parent lazy feature route config and move the lazy sub feature route configs into it, then add `<router-outlet></router-outlet>` component in the parent lazy route component template to display the active lazy sub feature
3. **combination of both** - we can have some lazy sub features as children of parent lazy feature route and some as siblings

Fractal nature of lazy features (nesting)

> This **splitting of large features into smaller sub-features** is a common pattern which can be repeated in **as many levels as needed**. It will always work exactly the same way following exactly the same patterns which makes it so powerful!



In the next chapter, we're going to discuss benefits of lazy loading in more detail (especially isolation) which will shine more light on why the fractal nature of lazy features is such a powerful property of this architecture.

Lazy loading and its benefits

Most developers are nowadays aware that lazy loading decreases the size of initial JavaScript bundle and hence speeds up application startup time.

This is definitely great but lazy loading comes with many more important benefits which have **huge impact of the application maintainability and extensibility**.

Development feedback loop

Splitting the application into multiple lazy features will lead to much faster re-builds when running `ng serve` or Angular CLI dev mode.

This is the mode used by developers to *work* on the application. With lazy loading, when we make a change to some source file, Angular CLI has to **rebuild much smaller lazy bundle** to which that file belongs, usually in range pf tens to hundreds of KBs.

That's a significant improvement in comparison to a single monolithic main bundle which could easily balloon to several MBs.

In practice, this can reduce re-build times from a couple of seconds to less than a hundred milliseconds which is a noticeable improvement in developer experience.

Lazy loading and its benefits

Isolation

Lazy loading in practice means that each lazy feature is isolated from the rest of the application.

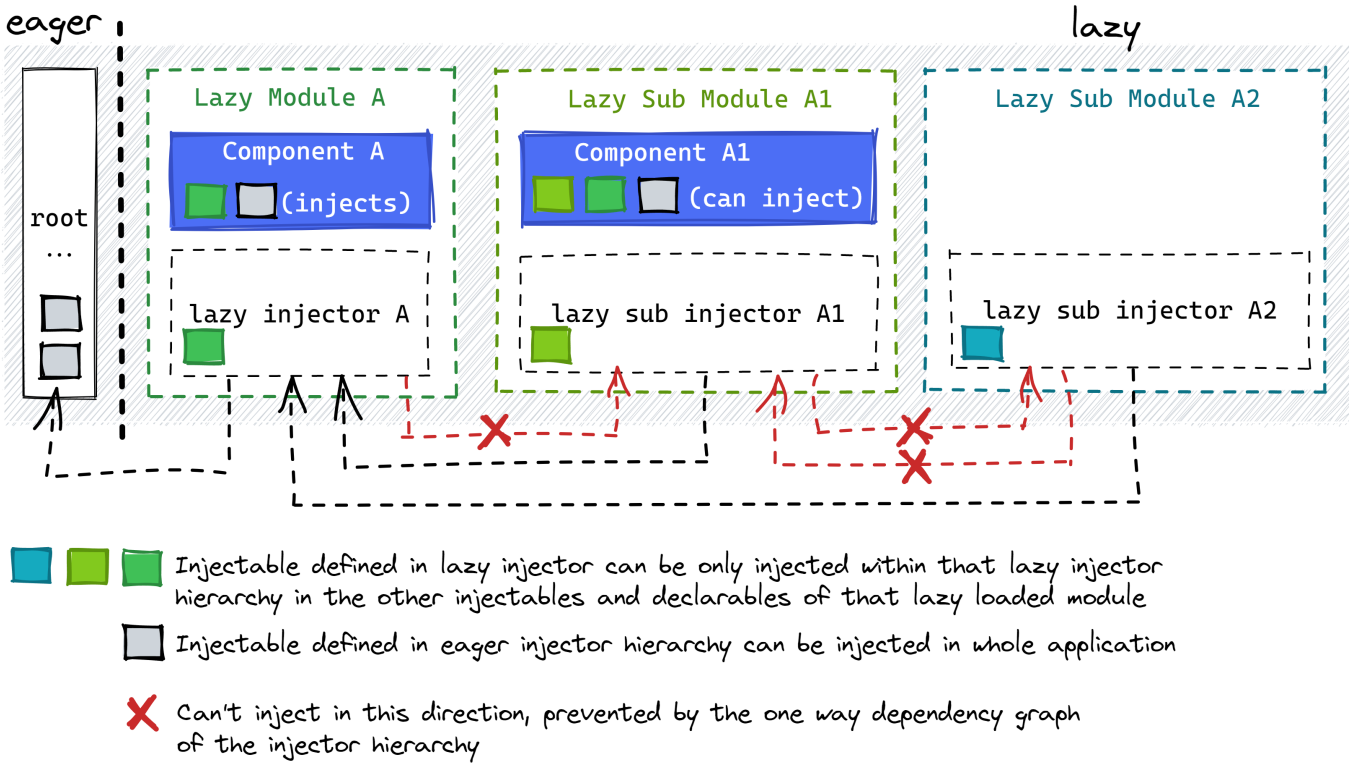
> If we think about it from the perspective of JavaScript and the browser, lazy loaded feature is an actual **separate JavaScript file** which is loaded only after user interaction during the application runtime.

This leads to a conclusion that if we tried to depend on some entity implemented as a part of that given lazy feature from another lazy feature, we would get an error at runtime because the JavaScript file with that feature would not be available until it was loaded.

⚠ Importing some service (or component) from lazy feature in another will cause Angular CLI build to bundle that implementation either in the main bundle or a virtual pre-bundle shared by both lazy features which is not what we want.

Luckily, **such mistake will get uncovered at runtime** because even though we imported and tried to inject such service (or use such component in template), it will not be a part of the target lazy loaded feature injector (or template context) and hence injection (or template usage) will fail!

Lazy loading and its benefits



Isolation guarantees

Using properly isolated lazy loaded features will make our application architecture much more robust and easier to maintain.

More so, it will provide us with a set of very useful guarantees which will make the development of new features and fixing bugs much easier and less error-prone.

Local impact

We are guaranteed that making changes to implementation of a given feature will have no impact on any other feature in the whole application.

Lazy loading and its benefits

Changes made to **core** and **shared** parts of the application can still have global impact and therefore reasonable test coverage is still required.

Local testing

In the same way, we are guaranteed that changing implementation of a given feature will not break tests of any other feature.

Faster startup time

And last but not least, lazy loading will lead to faster startup time of the application. Both initial content-full paint and first time to interactive will benefit from reduction of the size of the initial JavaScript bundle.

> In other words, the more implementation is moved into lazy loaded features, the less JavaScript will be loaded during the initial application startup and the application will load and become interactive faster!

Keep in mind that even when we could assume fast internet connection, the parsing and execution of large JavaScript bundle can still make the application startup take several seconds and make application feel slow and unresponsive.

Lazy loading heuristics

In the previous section we have discussed the benefits of lazy loading. Let's have a closer look what is usually considered a good candidate for lazy loaded feature and what is its basic anatomy.

In Angular, it is most common to implement lazy loaded features on the level of routes. While Angular allows to have more than one route active at the same time in parallel (multiple url segments separated by a delimiter in the url), such approach is pretty rare, so we can discard it for our purposes.

A lazy loaded feature usually corresponds to a single route or page or a sub-page of a given page. This can be done as an explicit nested navigation menu or an implicit flow as a result of user interaction like clicking on an edit button which switches the current subroute to editor or detail view.

Once we have decided that a given feature should be lazy loaded, the feature will then contain mix of components, services, pipes, directives, etc. which will implement the feature itself and will be scoped to the feature.

Feature specific services can be scoped to that feature by removing the out of the box generated `providedIn: 'root'` from their `@Injectable()` decorators and adding them to the `providers: []` array of the lazy feature module instead!

Sharing reusable components

Until now, our example application contains the core and two lazy feature modules.

As we start implement requirements of our lazy features we may encounter situation when some of them need to use the same or very similar component, directive or pipe (declarables).

This is a perfect use case for a `SharedModule` so let's create it using `ng g m shared` (or in our IDE). What can be provided by a `SharedModule`?

Declarables

The natural fit for a `SharedModule` are declarables like components, directives and pipes. The reason for that is that it's in their very nature to be instantiated multiple times. In fact, every time we use them in template we're getting a new instance.

Because of this, they represent the best fit for a `SharedModule` as every time we import it in a lazy loaded feature module. The module and its components can start using shared declarables in their templates without any issue!

The declarables will be part of the `declarations: []` and `exports: []` arrays of the `SharedModule`. That way the consumers lazy loaded features get access to them. This also allows us to create declarables which are private to the `SharedModule` and are not exported.

Sharing reusable components

> What is typically a part of the `SharedModule`?

Local custom shared components

Components (directives and pipe) which are useful to more than one lazy features which are implemented locally (within the application repository) and therefore not comping from a libraries or Angular itself.

Components from libraries

Another use case for a `SharedModule` is to re-export **most commonly** used components from vendor libraries like 3rd party (or internal) component libraries or a specific widget used by multiple lazy features.

Angular native modules like CommonModule or RouterModule

Typical use case is also to re-export Angular `CommonModule` as it's a module which exports Angular core components, pipes and directives like `ngClass`, `*ngIf` or `*ngFor` which are used by virtually every other component!

The next code example then also illustrates all core mechanics of the Angular template context mental model so please pay extra attention to the provided comments that should explain relationship between the `declarations: [ ]`, `imports: [ ]` and `exports: [ ]` arrays.

Sharing reusable components

The realistic implementation of a `SharedModule` in a typical Angular application will then look something like this...

```
@NgModule({
  // LOCAL TEMPLATE CONTEXT
  declarations: [
    SomeWidgetComponent,
    SomePrivateComponent // this component is not exported (not in exports: [])
  ],                     // and is therefore available only in the local
                         // template context of the SharedModule itself
                         // eg can be used in template of SomeWidgetComponent


  // brings declarables into LOCAL TEMPLATE CONTEXT
  imports: [
    // vendor
    CommonModule,   // *ngIf, *ngFor, ngClass, | date, ...
    RouterModule,   // <router-outlet>, routerLink, routerLinkActive, ...

    // material (or custom component framework) components
    // next two modules are used in both imports: [] and exports: []
    // therefore can be used in both LOCAL and CONSUMER TEMPLATE CONTEXTS
    MatCardModule,   // <mat-card>
    MatButtonModule  // mat-button, mat-raised-button, ...
  ],


  // delivers declarables to CONSUMER TEMPLATE CONTEXTS
  // (lazy feature modules which import the SharedModule)
  exports: [
    // vendor
    CommonModule, // *ngIf, *ngFor, ngClass, | date, ...
    RouterModule, // <router-outlet>, routerLink, routerLinkActive, ...
    ReactiveFormsModule, // (re-export only) formControl, formControlName, ...
                         // not used in local shared components templates

    // material (or custom component framework) components
    MatCardModule,
    MatButtonModule,     // commonly used components in most features ...
    MatDatepickerModule, // (re-export only)
                         // not used in local shared components templates

    // localy implemented shared components
    SomeWidgetComponent
  ]
})
export class SharedModule {}
```

Sharing reusable components

Understanding of the Angular template context mental model will allow us to make informed decisions about:

1. What needs to be part of `imports: [ ]` into the `SharedModule` (into local template context) to be able to use it in the local shared components templates
2. What needs to be part of `exports: [ ]` from the `SharedModule` (into consumer template context) to be available to be used in the templates of components which belong to the lazy feature modules which import the `SharedModule`
3. What can be private to the `SharedModule` and not exported (referenced only in the `declarations: []` array)
4. What can be re-exported from the `SharedModule` without being used in local shared components templates (referenced only in the `exports: [ ]` array)

The `SharedModule` can only be imported in the lazy feature modules and never in the `CoreModule` (or the `AppModule`) as that would significantly increase the eager bundle size of the application.

Sharing reusable components

Once we have `SharedModule` ready we can use it in our lazy loaded feature modules and remove `CommonModule` as it is now exported by the shared module itself.

```
@NgModule({
  declarations: [
    HomeComponent,
    TaskListComponent,
    TaskDetailComponent
  ],
  imports: [
    SharedModule,  // ⟵ SharedModule replaced original CommonModule
    HomeRoutingModule
  ]
})
export class HomeModule {}
```

The `SharedModule` will be imported by many lazy loaded features and because of that it should **never** implement any services (`providers: []`) and should only contain declarables (components, directives and pipes) and transitive re-exports of modules (which only contain declarables).

The reason for that is that every lazy loaded module would get its own service instance which is almost never what we want because in most cases we expect services to be singletons.

Let's clarify re-exporting of the `RouterModule` which might look suspicious as we might correctly assume that `RouterModule` contains services (for example the `Router` itself) which would be in direct conflict with our previous statement.

Sharing reusable components

Most Angular libraries from Angular team itself and also many other 3rd party open source libraries work around this problem by introducing a special convention when module in question only ever register `providers: []` when its called with the `.forRoot()` static method.

This means that importing and re-exporting of the plain `RouterModule` in the `SharedModule` is safe as it does not register any services and only contains declarables like `<router-outlet>` or `routerLink` !

As a final point of this topic, let's consider a situation where we want to create a "shared" services used in many parts of our application.

As we learned previously, we should implement such service in the `core/` folder and use `providedIn: 'root'` syntax without putting it in the `providers: []` array of the `CoreModule` .

> Great! Our application architecture skeleton is now finished! From now we are able to focus purely on delivering features for our users!

SharedModule tradeoffs

> It is important to acknowledge there is no such thing as a silver bullet.

Approach to sharing of reused components, pipes and directives with the help of the `SharedModule` comes with its own set of pros and cons.

Presented architecture strives to strike a nice balance between a small bundle size and reasonable developer experience based on real life observations in large enterprise environments.

Main things to consider being the composition, size and tree-shake-ability of the libraries that are used by the project.

As with all the recommendations, personal preferences and especially specific use case of the project at hand should always be the main criterion when making decisions about what approach makes the most sense.

With all this out of the way, we are going to explore three main options.

1. Smallest bundle size

> No `SharedModule` at all

There is no `SharedModule` module at all and every lazy feature module imports **exactly** what it needs which is determined by what is used in the templates of the components which belong to the given lazy feature module.

SharedModule tradeoffs

In more detail, lazy feature imports **all** modules provided by the 3rd party (or own) component libraries which deliver components which are used in the templates of local lazy feature components.

As for the local custom shared components, they will need to be implemented using *SCAM* (Single Component Angular Module) pattern as a component can only be declared (be part of the `declarations: []` array) of a single module which prevents its direct reuse by multiple lazy features.

The *SCAM* module will then contain the component in both the `declarations: []` and `exports: []` arrays and will be imported by the all lazy feature modules which want to use this local reusable component.

Such shared component, eg `RatingComponent`, then would be implemented in `shared/rating/` folder and contain exactly one component and one module (could be inlined in single file).

```
@Component({ selector: 'my-org-rating', /* ... */ })
export class RatingComponent {}

@NgModule({
  declarations: [RatingComponent],
  imports: [CommonModule], // ... and others used in the widgets template (context)
  exports: [RatingComponent]
})
export class RatingModule {}
```

> Every single locally implemented shared component then would follow exactly the same pattern!

SharedModule tradeoffs

The locally implemented shared components will then be consumed by the lazy features explicitly by importing their respective *SCAM* modules.

```
@NgModule({
  declarations: [HomeComponent,],
  imports: [
    HomeRoutingModule,
    CommonModule,  // ⟵ SharedModule doesn't exist
    MatCardModule, // ⟵ add every 3rd party module explicitly
    RatingModule,  // ⟵ add SCAM for every custom shared component used in the feature
    // ... in practice a long list of used components and 3rd party modules
  ]
})
export class HomeModule {}
```

With this no `SharedModule` approach, we get the smallest possible bundles with most optimization but the developers will have to maintain **extensive lists** of imports which is a tedious error-prone task.

It is especially easy to forget to remove imports once the components, pipes and directives are no longer used in the templates of components which belong to given lazy feature as this will not result in any compilation errors.

Please notice that introducing arrays like `export const SHARED = [RatingModule, ... ]` which group all shared components defeats the purpose of using *SCAM*s and ends up with same results as using `SharedModule` but using more concepts and adding to the confusion.

SharedModule tradeoffs

Another downside of this approach is that **it will lead to a lot of extra effort when implementing tests for the components** which belong to a lazy feature because every component test has to build appropriate template context by importing what is necessary into the `TestBed` testing module.

This multiplies the effort needed to maintain the list of imports of the lazy loaded feature module by the amount of components such feature has while each of the test needing only a unique **subset** of the imports based on what is used in the template of the component under test.

> You might still opt-in for this approach if you want to get the smallest possible bundles and don't mind the extra effort needed to maintain the multiple lists of imports.

Besides *SCAM*s, the **no** `SharedModule` approach can be also implemented using Standalone components (*STAC*s), as they behave exactly the same as *SCAM*s when it comes to their interaction with template context but with additional benefit of reducing the amount of boilerplate code needed to implement them (we don't have to implement the *SCAM* `@NgModule` part).

SharedModule tradeoffs

2. Simplest possible DX

> No `SharedModule` and no lazy loaded feature modules

This example is just for illustration purposes only as it is not compatible with the architecture described in this book. This solution means that were going to have only a single module, the `AppModule` which will import **every** module which was used in atl east single template.

Developers do not have to burden themselves with maintaining of the small isolated template context of the lazy feature modules and can access everything everywhere.

The application has only one bundle as everything is part of the initial main bundle and the architecture will slowly converge to a *"big ball of mud"* which is very hard to maintain until it implodes under its own weight..

SharedModule tradeoffs

3. Small lazy bundles and reasonable DX

> `SharedModule` (with discriminate content) and lazy loaded feature modules

The application will use our architecture with lots of lazy loaded feature modules together with `SharedModule` as we described previously.

It will import and re-export **most commonly used** declarables from 3rd party (or local) component libraries together with re-usable local components.

Such `SharedModule` will then be imported by all lazy feature modules and will represent a good starting point for the template context of the component tests when we import it in the `TestBed` testing module.

The main concern of this final best approach is to be able to decide what represents "**most commonly used declarables**" which should be part of the `SharedModule` and what is used only by one (or few) lazy loaded feature modules and hence should be imported by them in an explicit way (and not be a part of the `SharedModule`).

SharedModule tradeoffs

As a general rule of thumb, if the component, pipe or directive is used only by few lazy loaded feature modules, it should be imported by them instead of the `SharedModule`.

If the declarable is especially *"heavy"* with its impact on the lazy feature module bundle size, it should be definitely imported by the lazy feature module instead of the `SharedModule`. Typical example of this later case can be a complex data table or a charting library.

> This final approach strikes a fine balance between reasonable bundle size together with productive DX. We won't have to add 50 lines of imports in every lazy feature module and component test files which is definitely a win! More so, this approach is flexible enough to be able to adjust it based on the specific needs of the project based on its unique requirements.

SharedModule tradeoffs

4. *Smaller* lazy bundles and *still reasonable* DX

> Multiple focused `SharedModule`s (with discriminate content) and lazy loaded feature modules

Similar to the previous approach, the application will use `SharedModule` with the most commonly used declarables from 3rd party (or local) component libraries like `CommonModule`, `MatCardModule` or `RatingComponent`, **but** this time we will introduce additional `SharedModule`s like `SharedFormsModule` or `SharedChartsModule` which will be focused on specific areas of the functionality delivering declarables which are:

- used together by the consumer lazy loaded feature, eg multiple form input types to implement form
- only used by a specific subset of lazy loaded feature modules (only some lazy features will implement forms)

This approach represents the closest thing to the **optimal solution under most circumstances** as it tries to minimize both the size of the lazy feature bundles and the effort needed to maintain the lists of imports while being easy to extend and adjust to the specific needs of the project at hand!

SharedModule tradeoffs

Example of such a use case specific shared module can be `SharedFormsModule` which could look like this:

```
@NgModule({
  declarations: [CustomFormInputComponent],
  imports: [
    CommonModule,
    ReactiveFormsModule,

    // material ...
    MatFormFieldModule,
    MatInputModule,
    MatDatepickerModule,
    MatCheckboxModule,
  ],
  exports: [
    CommonModule,
    ReactiveFormsModule,

    // material ...
    MatFormFieldModule,
    MatInputModule,
    MatDatepickerModule,
    MatCheckboxModule,

    // local
    CustomFormInputComponent
  ]
})
export class SharedFormsModule {}
```

And then it would be consumed together with the base `SharedModule` in the lazy feature module which needs to implement forms.

```
@NgModule({
  declarations: [HomeComponent],
  imports: [
    HomeRoutingModule,
    SharedModule,  // ⟵ base SharedModule
    SharedFormsModule, // ⟵ usecase specific shared module
    // ... other heavy one-of imports
  ]
})
export class UserModule {}
```

Sharing declarables between the eager and lazy loaded parts of the application

Till now, we have only discussed use case when we need to share our own local custom declarables (and declarables coming from 3rd party libraries) between multiple lazy loaded feature modules.

By now, it should be clear that we should never import `SharedModule` (or use case specific shared modules like `SharedFormsModule`) in the `CoreModule` (or `AppModule`) as it would mean that the declarables from the `SharedModule` would be part of the eagerly loaded main bundle and degrade startup and build time performance of the application significantly!

> Sometimes we really need to share some declarables between the eager and lazy loaded parts of the application!

A typical example could be something like:

- `AvatarComponent` - used in both header for currently logged-in user and a lazy loaded feature in the list of tasks (eg task was assigned to user)
- `MenuComponent` - used in both header (current user menu, eg settings and log out) and a lazy loaded feature in the list of tasks (task actions menu, eg edit and remove)

In this case we can create a new `SharedEagerModule` which will be:

1. imported by the `CoreModule` (eager part)
2. re-exported by the base `SharedModule` (which makes its declarables available in the lazy features)

Sharing declarables between the eager and lazy loaded parts of the application

Another option would be to implement all these declarables as *SCAM*s or (*STAC*s) which will be then

1. **individually** imported by the `CoreModule` (eager part)
2. **individually** re-exported by the base `SharedModule` (which makes its declarables available in the lazy features)

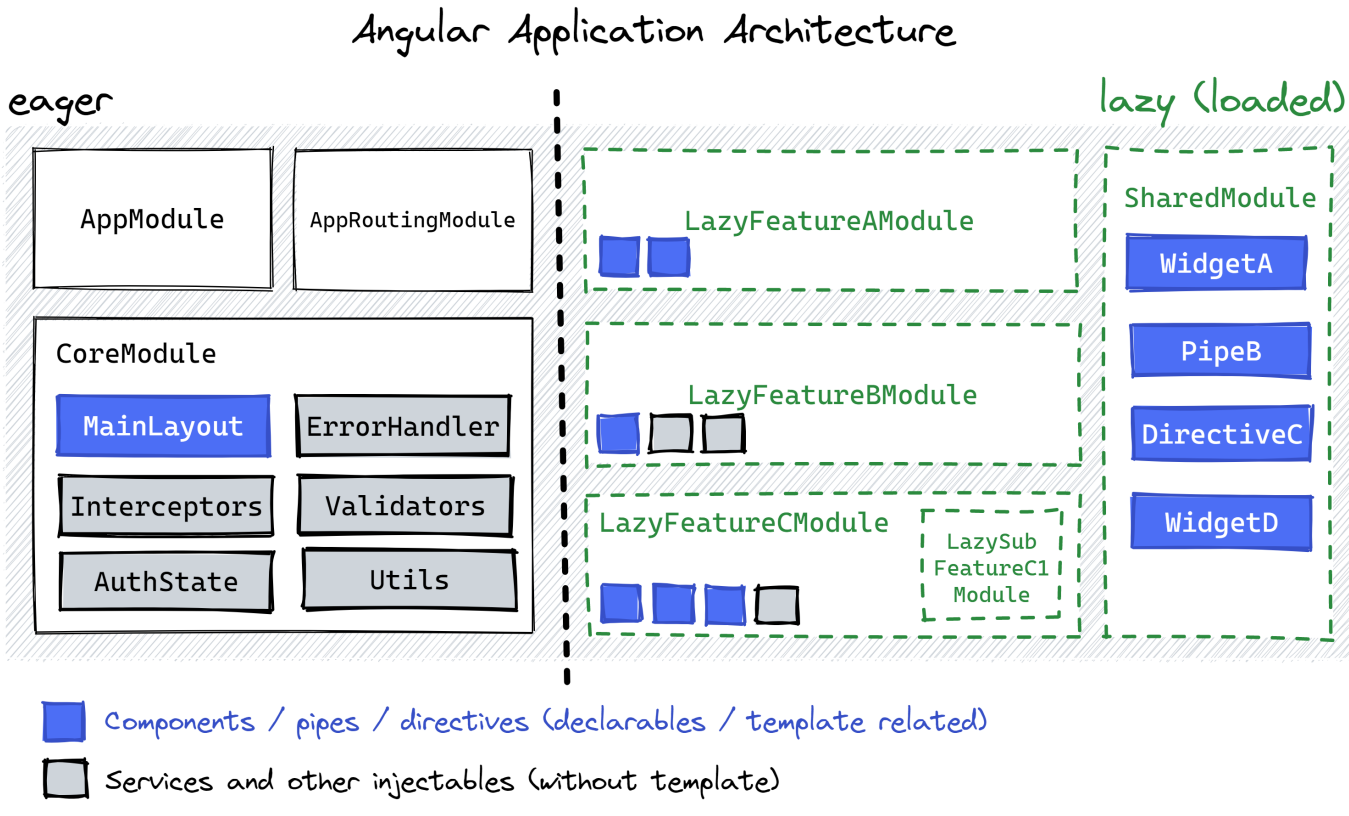
One way or another, all these declarables will end up in the both **eager** and **every lazy** template context and hence will be available in the template of every component in the application.

Always try to minimize amount of unique declarables which are used in the eager part of the application!

Wrap-up and future considerations

The `SharedModule` represented the last piece of the puzzle in our implementation of enterprise grade Angular application architecture.

> Let's revisit the architecture diagram one more time with our newly acquired knowledge!



In a case of really large application, consider reaching for NX monorepo as it provides more robust **automated tooling** to enforce rules and guidelines described in this book like preventing imports between the lazy features which must otherwise be upheld by the combination of developer discipline and rigorous code reviews.

Angular standalone components (STACs)

Angular team have introduced a new way of defining of the *template context* by introducing the concept of **standalone components** which were released in development preview in Angular v14 and fully in Angular v15.

This approach allows us to define a component which is not part of any module by specifying `standalone: true` in the `@Component` decorator.

Such component is the imported by the module which wants to use it in the `imports: []` array, and then it can be used in the template as before.

As such, this approach is very similar to the *SCAM* pattern we described previously while reducing the amount of boilerplate code needed to implement it.

Standalone components do not change the way the two core concepts - **template context** and **injection hierarchy** - work in Angular. They are just a new way of defining what should be part of the template context of the component and what should be instantiated in the lazy injector of the lazy loaded feature.

Because of this, approach described in this book is still valid and will continue to work in the future!

Angular application can easily mix and match the two approaches (`NgModule` and `standalone: true`) per lazy loaded feature. Shared standalone components can be imported by the `SharedModule` and then re-exported by it in the same way as declarables from 3rd party libraries.

Get in touch & feedback

Did you enjoy the content of this book and learned something new?

Let us know what was your experience get-in-touch@angularexperts.io or directly on Twitter by sending us a direct message.

> We would love to hear your feedback and suggestions (short feedback form / 30sec) for future improvements or on case you found any mistakes or typos in the book.

Check out our offer of Angular, NgRx and RxJs trainings and consulting services to empower your team with the right knowledge to deliver exceptional experiences to your customers.

> Always bet on Angular!

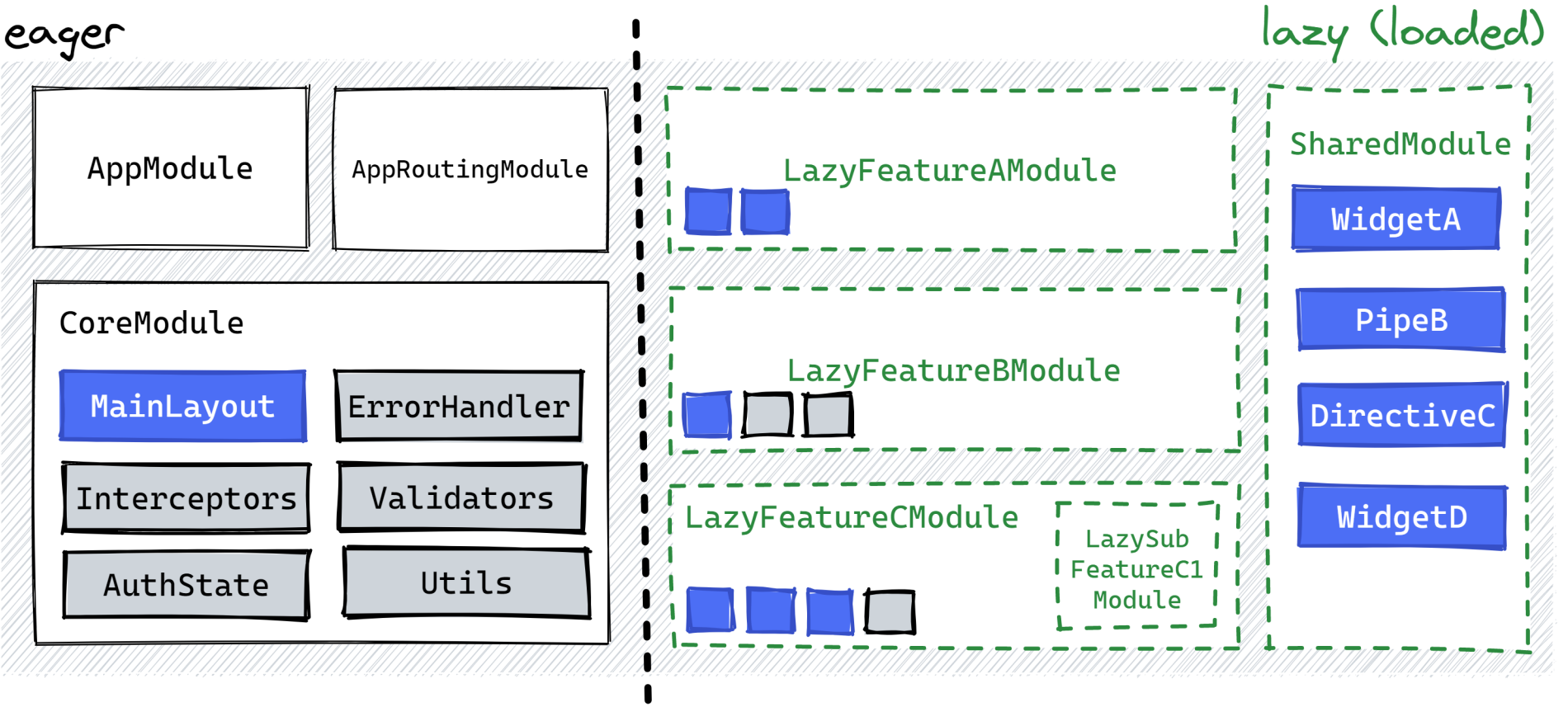
The v1.6.1 version was published on 09.01.2023

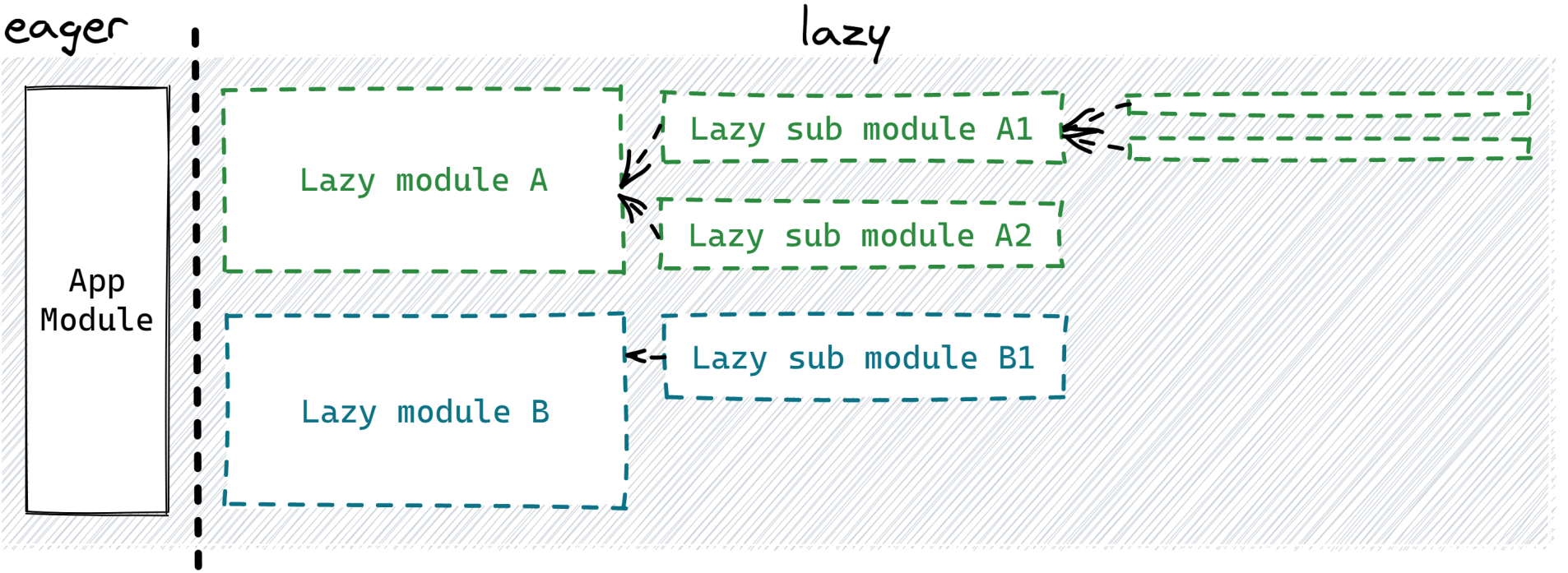
Resources

Following pages contain easy to print diagrams which can be used as a reference for the architecture described in this book.

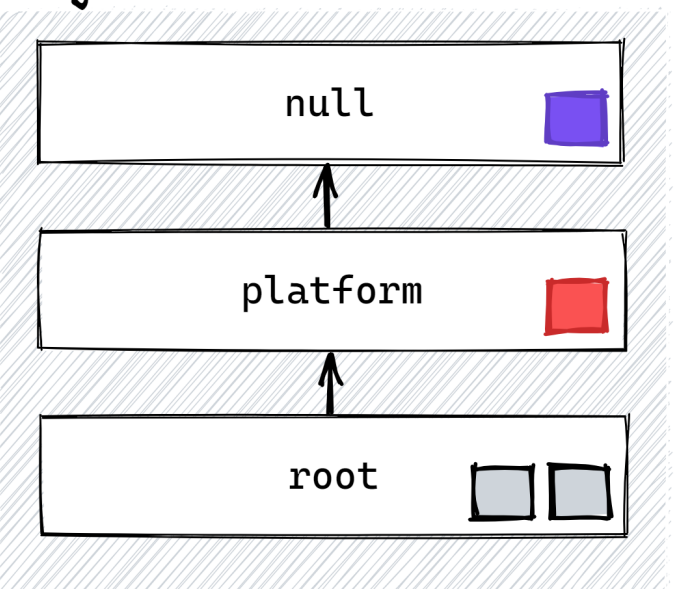
This can prove useful when first familiarizing yourself with the described architecture and can serve as a reference when presenting the architecture to your team or planing a new feature for your project!

Angular Application Architecture




eager
lazy
App Module
Lazy module A
Lazy module B
Lazy sub module A1
Lazy sub module A2
Lazy sub module B1

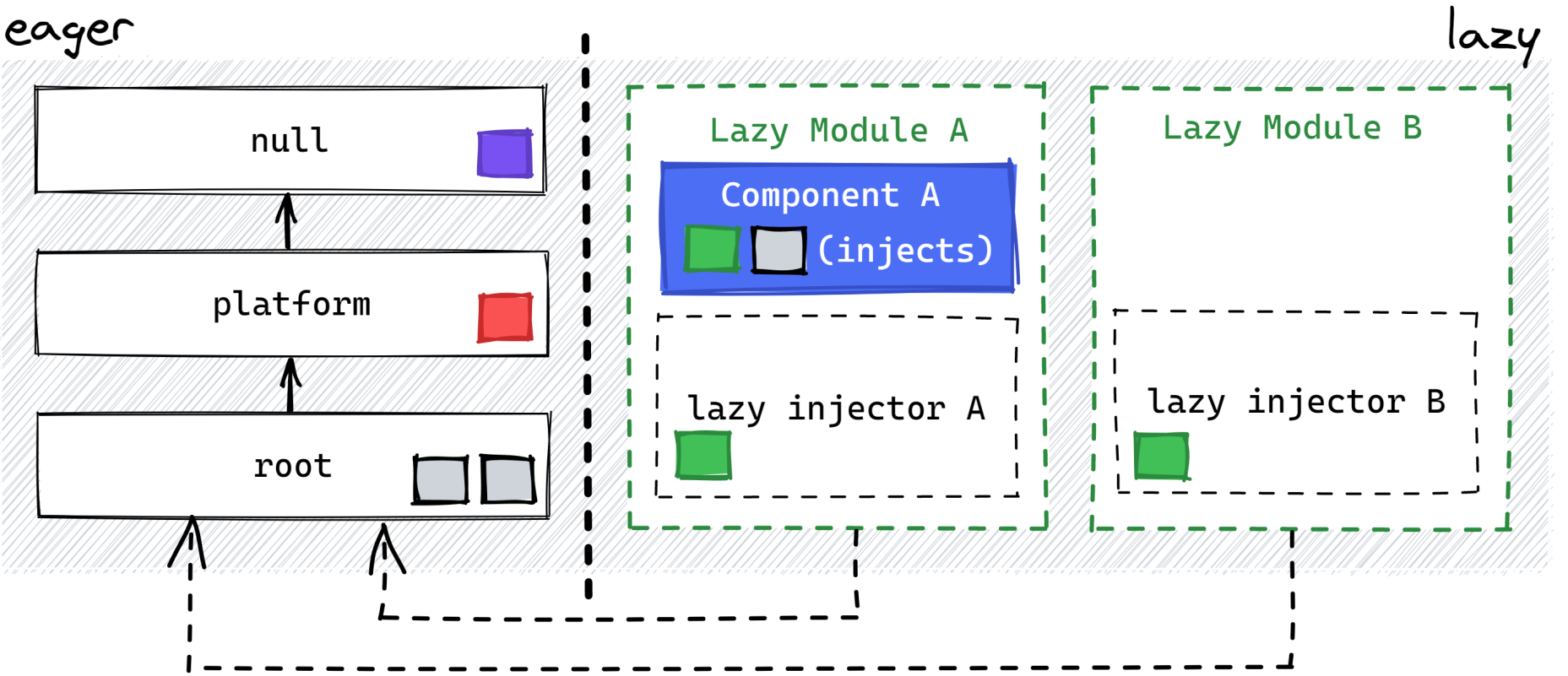
eager



Null injector injects "null" for injection which was marked @Optional() if no service was found

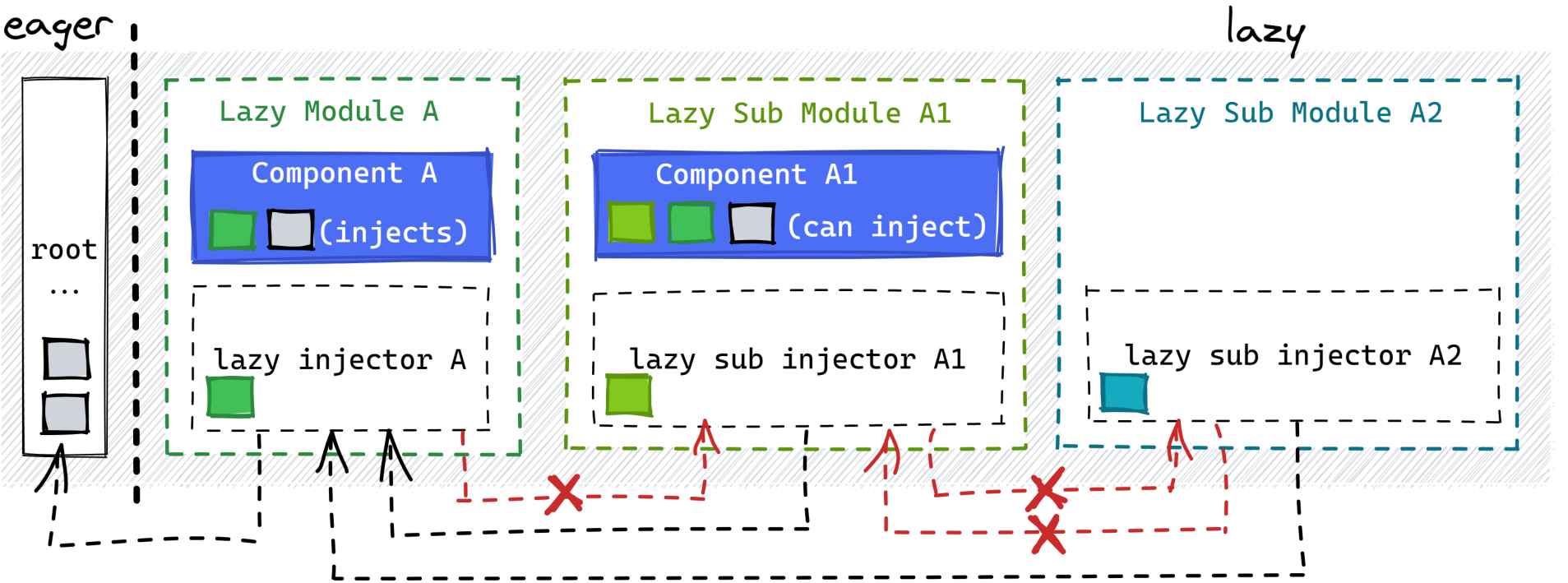
Angular (platform) specific services and other injectables like DomSanitizer and serices providedIn: 'platform'

App specific services and other injectables (without template) providedIn: 'root' or in providers: [] of AppModule or other modules imported by AppModule



Injectable defined in lazy injector can be only injected within that lazy injector hierarchy in the other injectables and declarables of that lazy loaded module

Injectable defined in eager injector hierarchy can be injected in whole application



Injectable defined in lazy injector can be only injected within that lazy injector hierarchy in the other injectables and declarables of that lazy loaded module

Injectable defined in eager injector hierarchy can be injected in whole application

Can't inject in this direction, prevented by the one way dependency graph of the injector hierarchy