iLink Digital
Thinking Beyond

Angular Architecture Best Practices & Guidelines

PREPARED BY
GURU KANDASAMY
LAKSHMANAKUMAR J

TABLE OF CONTENT

Contents

TABLE OF CONTENT 2

1.0 Introduction 4

1.1 Scalability Issues in Front-end 4

1.2 High Level Abstraction Layer 5

1.2.1 Presentation Layer 5

1.2.2 Abstraction Layer 5

1.2.2.1 Abstraction Interface 6

1.2.2.2 State 6

1.2.2.3 Synchronization Strategy 6

1.2.2.4 Caching 6

1.2.3 Core Layer 7

1.2.4 Unidirectional data flow and Reactive state management 7

1.2.5 Modular design 9

1.2.5.1 Core Module 10

1.2.5.2 Shared Module 11

1.2.5.3 Feature Module 11

1.3 Lazy Loading 12

1.4 Using index.ts 12

1.5 Using Scss 13

1.6 Use of smart vs. dummy components 13

2.0 Coding Conventions Guidelines 14

2.1 Class Names 14

2.2 File Naming 14

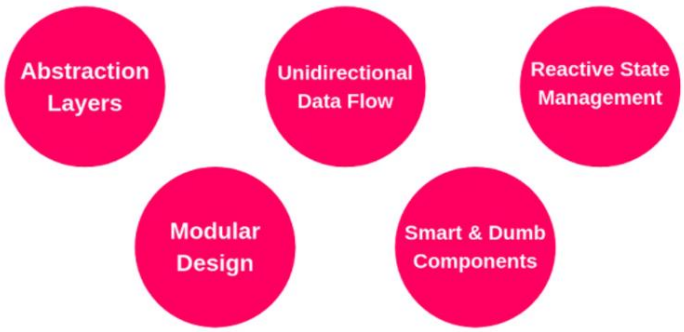
2.3 Constants 14

2.4 Single Responsibility Principle 14
2.5 Interfaces 15
2.6 Immutability 15
2.7 Safe Navigation Operator (?) 15
2.8 Prevent Memory Leaks in Angular Observable 15
2.9 Using Smart-dumb components 16
2.10 Using strict types instead of “any” 16
2.11 Change detection optimisations 16
2.12 Using trackBy in NgFor 16
2.13 Use Lint rules for Typescript and Scss 17
2.14 Always document 17
2.15 Cache API Calls 17
2.16 Properties and methods 17
2.17 Import line spacing 18
2.18 Components 18
2.19 Services 19
2.20 Lifecycle hooks 19
3.0 Angular Libraries 19
3.1 Angular Lib: User Interface: 19
3.2 Angular Lib: Routing utilities: 20
3.3 Angular Lib: DOM 20
3.4 Angular Lib: Pipes, Directives & Decorators 21
3.5 Angular Lib: Utilities 22
4.0 Performance 24
5.0 Security 25
6.0 Deployment 25
7.0 External References 26
8.0 About iLink Digital 27

1.0 Introduction

The goal of this document is to educate on how to design Angular application in order to maintain sustainable development speed and ease of adding new features in the long run. To achieve these goals, we will apply:

- Maintain abstractions between application layers,
- Ensure unidirectional data flow,
- Implement Reactive State Management,
- Follow Modular design,
- Make use of Smart and Dumb components pattern.



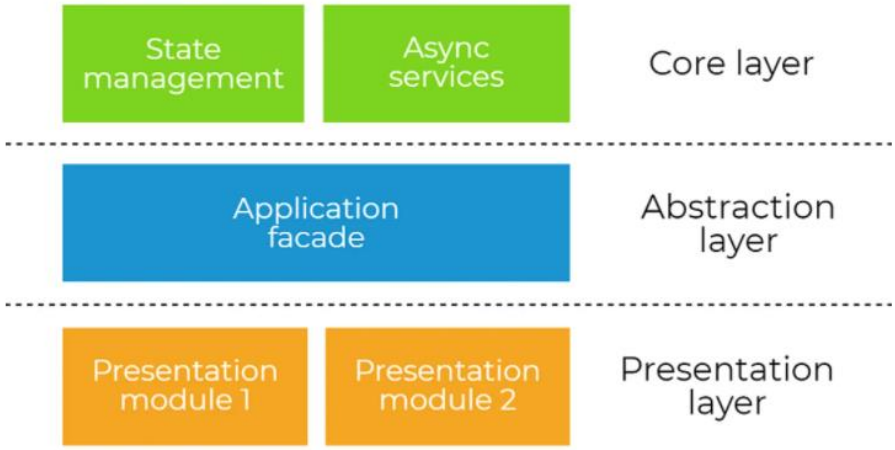
1.1 Scalability Issues in Front-end

- Front-end applications are not “just displaying” data and accepting user inputs but providing users with rich interactions and use backend mostly as a data persistence layer.
- Far more responsibility has been moved to the front-end part of software systems. This leads to a growing complexity of front-end logic, we need to deal with.
- One of the solutions to the problems described above is solid system architecture and this comes with the cost, the cost of investing in that architecture from day one.

1.2 High Level Abstraction Layer

Below diagram depicts general concept of the decomposition. This division of the system also dictates communication rules.

For example, the presentation layer can talk to the core layer only through the abstraction layer.



1.2.1 Presentation Layer

- Angular components reside in the Presentation Layer.
- The only responsibilities of this layer are to present the UI and delegates user's actions to the core layer, through the abstraction layer.
- It knows what to display and what to do, but it does not know how the user's interactions should be handled.

1.2.2 Abstraction Layer

- The abstraction layer decouples the presentation layer from the core layer and also has it's very own defined responsibilities.
- This layer exposes the streams of state and interface for the components in the presentation layer, playing the role of the facade.

- This kind of facade sandboxes what components can see and do in the system. We can implement facades by simply using Angular class providers. The classes here may be named with Facade postfix.

1.2.2.1 Abstraction Interface

- The main responsibilities for this layer; to expose streams of state and interface for the components.

- Public methods abstract away the details of state management and the external API calls from the components.

- Abstraction layer is not a place to implement business logic. we just want to connect the presentation layer to our business logic, abstracting the way it is connected.

1.2.2.2 State

- Abstraction layer makes our components independent of the state management solution. It gives us a lot of flexibility and allows to change the way we manage state not even touching the presentation layer

- Components are given Observables with data to display on the templates (usually with async pipe) and don't care how and where this data comes from.

- To manage our state we can pick any state management library that supports RxJS (like NgRx) or simple use BehaviorSubjects to model our state.

1.2.2.3 Synchronization Strategy

- **Optimistic update** changes the UI state first and attempts to update the backend state. This provides a user with a better experience, as he does not see any delays, because of network latency. If backend update fails, then UI change has to be rolled back.

- **Pessimistic update** changes the backend state first and only in case of success updates the UI state. Usually, it is necessary to show some kind of spinner or loading bar during the execution of backend request, because of network latency.

1.2.2.4 Caching

We can apply data caching in our facade. The easiest way to achieve it is to use shareReplay() RxJS operator that will replay the last value in the stream for each new subscriber.

To sum up, what we can do in the abstraction layer is to:

- expose methods for the components in which we:
- delegate logic execution to the core layer,
- decide about data synchronization strategy (optimistic vs. pessimistic),
- expose streams of state for the components:
- pick one or more streams of UI state (and combine them if necessary),
- cache data from external API.

1.2.3 Core Layer

- In the core layer, application logic is implemented. All data manipulation and outside world communication happen here.
- In the core layer, we also implement HTTP queries in the form of class providers. This kind of class could have Api or Service name postfix.
- API services have only one responsibility - it is just to communicate with API endpoints and nothing else. We should avoid any caching, logic or data manipulation here.
- In this layer, we could also place any validators, mappers or more advanced use-cases that require manipulating many slices of our UI state.

1.2.4 Unidirectional data flow and Reactive state management

- Reactive State management helps us to achieve data consistency in Angular applications.

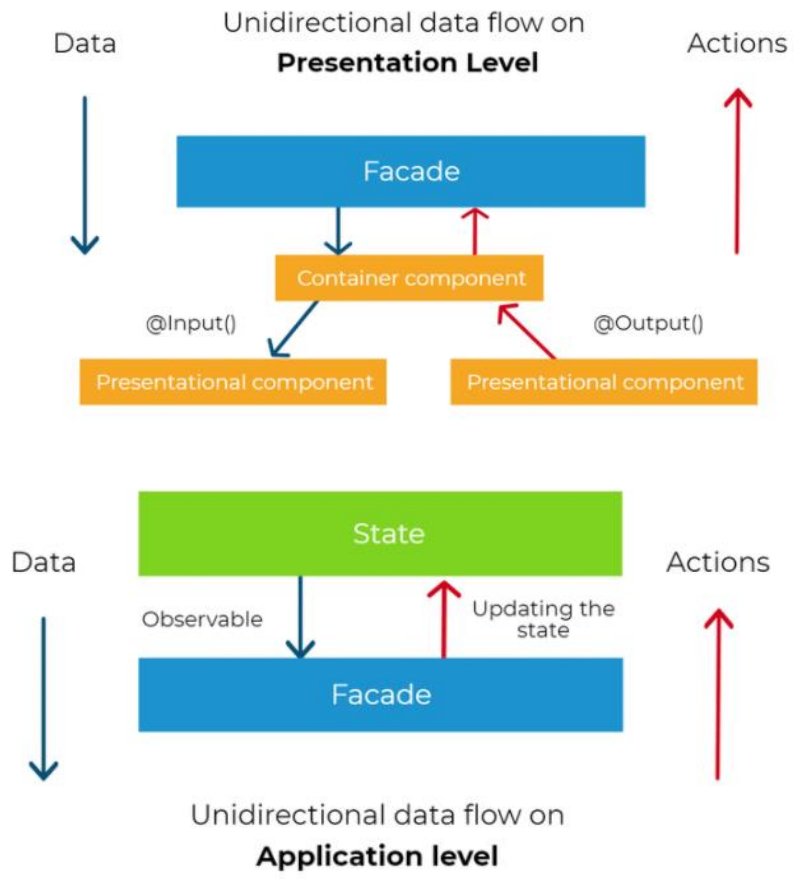
- Whenever any model value change in our application, Angular change detection system takes care of the propagation of that change. It does it via input property bindings from the top to bottom of the whole component tree.

NgRx

- We can place the application data (the state) in one place “above” the components and propagate the values down to the components via Observable streams (Redux and NgRx call this place a store).

- State can be propagated to multiple components and displayed in multiple places, but never modified locally. The change may come only “from above” and the components below only “reflect” the current state of the system.

- This gives us the important system’s property - data consistency - and the state object becomes the single source of truth.



- Our state object exposes the methods for the services in our core layer to manipulate the state.

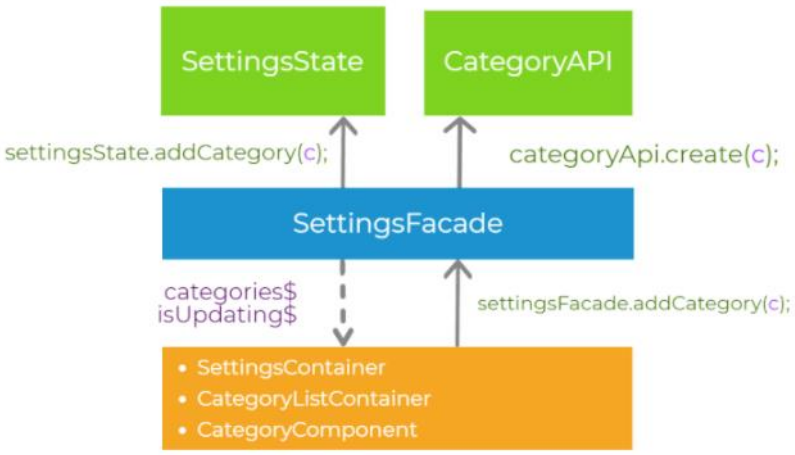
- Whenever there is a need to change the state, it can happen only by calling a method on the state object (or dispatching an action in case of using NgRx).

- Then, the change is propagated “down”, via streams, the to presentation layer (or any other service). This way, our state management is reactive.

Behaviour Subject

- The component delegates the execution to the abstraction layer, calling the method on the facade settingsFacade.addCategory().

- Then, the facade calls the methods on the services in the core layer - categoryApi.create() and settingsState.addCategory().



- The order of invocation of those two methods depends on synchronization strategy we choose (pessimistic or optimistic).

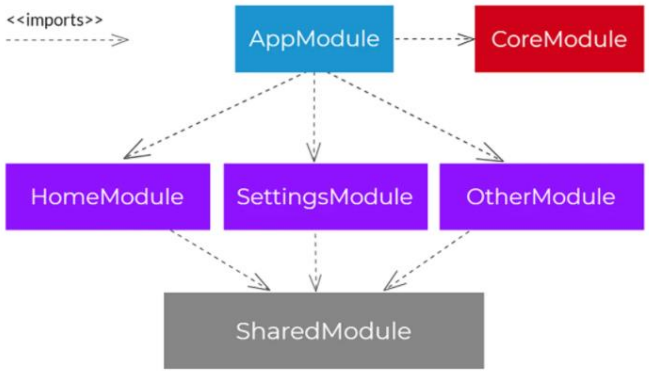
- Finally, the application state is propagated down to the presentation layer via the observable streams. This process is well-defined.

1.2.5 Modular design

- The idea of Modular design is to slice the application into feature modules representing different business functionalities.

- Each of the features modules share the same horizontal separation of the core, abstraction, and presentation layer.

- It is important to note, that these modules could be lazily loaded (and preloaded) into the browser increasing the initial load time of the application.



1.2.5.1 Core Module

- Create a CoreModule with providers for the singleton services you load when the application starts.

- Import CoreModule in the root AppModule only. Never import CoreModule in any other module.

- Consider making CoreModule a pure services module with no declarations.

```
|-- core
        |-- [+] authentication
        |-- [+] footer
        |-- [+] guards
        |-- [+] http
        |-- [+] interceptors
        |-- [+] mocks
        |-- [+] services
        |-- [+] header
        |-- core.module.ts
        |-- ensureModuleLoadedOnceGuard.ts
        |-- logger.service.ts
```

1.2.5.2 Shared Module

- Build a SharedModule with the components, directives, and pipes that you use throughout your app. This module should consist completely of declarations, most of them exported.

- The SharedModule may re-export other widget modules, such as CommonModule, FormsModule, and modules with the UI controls that you use most widely.

- The SharedModule should not have providers. Nor should any of its imported or re-exported modules have providers.

- Import the SharedModule in your feature modules, both those loaded when the app starts and those you lazy load later.

```
|-- shared
      |-- [+] components
      |-- [+] directives
      |-- [+] pipes
```

1.2.5.3 Feature Module

- Create multiple feature modules for every independent feature of our application.

- Feature modules should only import services from CoreModule.

- To attempt to build features which don't depend on any other features just on services provided by CoreModule and components provided by SharedModule

FEATURE MODULE	CAN BE SOME IMPORTED BY	EXAMPLES
Domain	Feature, AppModule	ContactModule (before routing)
Routed	Nobody	ContactModule, DashboardModule,

Routing	Feature (for routing)	AppRoutingModule, ContactRoutingModule
Service	AppModule	HttpModule, CoreModule
Widget	Feature	CommonModule, SharedModule

1.3 Lazy Loading

- Lazy Loading helps in reducing the size of the application by abstaining from unnecessary file from loading.

- The feature module should be placed synchronously when the app startup to show initial content. Each other feature module should be loaded lazily after user-triggered navigation. That way we will be able to make our Angular app faster.

- Utilizing lazy load the modules can enhance productivity.

1.4 Using index.ts

index.ts helps us to keep all related things together so that we don't have to be bothered about the source file name. This helps reduce the size of the import statement.

For example, we have article/index.ts as

//articles/index.ts

export * from './article.model'

export * from './article.config'

We can import all things by using the source folder name.

Import { Article, ArticleConfig } from '..article';

1.5 Using Scss

Scss is a styles preprocessor which brings support for fancy things like variables (even though css will get variables soon too), functions, mixins etc

The global styles for the project are placed in a scss folder under assets.

```
|-- scss
    |-- partials
        |-- _layout.vars.scss
        |-- _responsive.partial.scss
    |-- _base.scss
|-- styles.scss
```

The scss folder does only contain one folder — partials. Partial-files can be imported by other scss files. In my case, styles.scss imports all the partials to apply their styling.

1.6 Use of smart vs. dummy components

We want to divide components into two categories, depending on their responsibilities. First, are the smart components (aka containers). These components usually:

- have facade/s and other services injected,
- communicate with the core layer,
- pass data to the dumb components,
- react to the events from dumb components,
- are top-level routable components (but not always!).

2.0 Coding Conventions Guidelines

2.1 Class Names

- Do use upper camel case when naming classes.
- Classes can be instantiated and construct an instance.
- By convention, upper camel case indicates a constructible asset.

2.2 File Naming

While creating files, we should pay attention to the file names.

- Names of folders and files should clearly convey their intent.
- Names should be consistent with the same pattern in which we mention the file's feature first and then the type, dot separated.
- To add more descriptive names to our files we should use a dash(-) to separate the words in the name

2.3 Constants

- Do declare variables with const if their values should not change during the application lifetime.
- Consider spelling const variables in lower camel case.
- Do tolerate existing const variables that are spelled in UPPER_SNAKE_CASE. The tradition of UPPER_SNAKE_CASE remains popular and pervasive, especially in third party modules.

2.4 Single Responsibility Principle

- We should not to create more than one component, service, directive inside a single file.

- Every file should be responsible for a **single functionality**.

- Single responsibility helps us to keep our files clean, readable, and maintainable.

2.5 Interfaces

- To Create a contract for our class we should always use **interfaces**. By using them we can force the class to implement functions and properties declared inside the interface.

- Using interfaces is a perfect way of describing our object literals. If our object is an interface type, it is obligated to implement all of the interface's properties.

2.6 Immutability

- Objects and arrays are the reference types in javascript.

- In order to copy one object into another object or an array and to modify them, the best practice is to do that in an immutable way using **es6 spread operator (…)**

- This way, we are deep copying the user object and then just overriding the status property.

2.7 Safe Navigation Operator (?)

- Use safe navigation operator while accessing a property from an object in a component's template.

- When we use the save navigation **(?)** operator**,** the template will ignore the null value and will access the property once the object is not null anymore.

2.8 Prevent Memory Leaks in Angular Observable

Observables in Angular are very useful as it streamlines your data, but memory leak is one of the very serious issues that might occur if you are not focused. Below are few ways which can be followed to avoid leaks.

- *Using* async *pipe*
- Using take(1)

- Using takeUntil()

2.9 Using Smart-dumb components

- This pattern helps to use **OnPush change detection strategy** to tell Angular there have been no changes in the dumb components.
- **Smart components** are used in manipulating data, calling the APIs, focussing more on functionalities, and managing states.
- While **dumb components** are all about cosmetics, they focus more on how they look.

2.10 Using strict types instead of “any”

- Avoid using type any while declaring variables.
- If the variables and constants are not specified by any, they will be assumed by the value and as a result, will be assigned to it.

2.11 Change detection optimisations

- Use NgIf and not CSS - If DOM elements aren’t visible, instead of hiding them with CSS, it's a good practice to remove them from the DOM by using *ngIf.
- Move complex calculations into the ngDoCheck lifecycle hook to make your expressions faster.
- Cache complex calculations as long as possible
- Use the OnPush change detection strategy to tell Angular there have been no changes. This lets you skip the entire change detection step.

2.12 Using trackBy in NgFor

- When using ngFor to loop over an array in templates, use it with a trackBy function which will return a unique identifier for each DOM item.

- When an array changes, Angular re-renders the whole DOM tree. But when you use trackBy, Angular will know which element has changed and will only make DOM changes only for that element.

2.13 Use Lint rules for Typescript and Scss

- Linting forces the program to be cleaner and more consistent.

- It is widely supported across all modern editors and can be customized with your own lint rules and configurations.

2.14 Always document

- Always document the code as much as possible. It will help the new developer involved in a project to understand its logic and readability.

- It is a good practice to document each variable and method.

- For methods, we need to define it using multi-line comments on what task the method performs and all parameters should be explained.

2.15 Cache API Calls

- Caching the API calls especially on the website limits the number of server requests to fetch redundant information thus saving time and reduce the load on the server.

- Utilizing the caching, one needs to make an HTTP request and then store the results of that request in memory, which can be served once again whenever required without requesting to the server.

- This helps in user to make fewer HTTP requests to the server and on return had to wait less for the response every time it required.

2.16 Properties and methods

- Do use lower camel case to name properties and methods.

- Avoid prefixing private properties and methods with an underscore.

2.17 Import line spacing

- Consider leaving one empty line between third party imports and application imports.
- Consider listing import lines alphabetized by the module.
- Consider listing de-structured imported symbols alphabetically.

2.18 Components

- Do use dashed-case or kebab-case for naming the element selectors of components because it will Keeps the element names consistent with the specification for Custom Elements.
- Do give components an element selector, as opposed to attribute or class selectors because components have templates containing HTML and optional Angular template syntax.
- Do extract templates and styles into a separate file, when more than 3 lines.
- Do name the template file [component-name].component.html, where [component-name] is the component name.
- Do name the style file [component-name].component.css, where [component-name] is the component name.
- Do specify component-relative URLs, prefixed with ./. Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability.
- Do use the @Input() and @Output() class decorators instead of the inputs and outputs properties of the@Directive and @Component metadata.
- The metadata declaration attached to the directive is shorter and thus more readable.

- Avoid input and output aliases except when it serves an important purpose because Two names for the same property (one private, one public) is inherently confusing.

- You should use an alias when the directive name is also an input property, and the directive name doesn't describe the property.

2.19 Services

- Do provide services to the Angular injector at the top-most component where they will be shared.

- When providing the service to a top level component, that instance is shared and available to all child components of that top level component. This is ideal when a service is sharing methods or state and this is not ideal when two different components need different instances of a service.

- Do use the @Injectable() class decorator instead of the @Inject parameter decorator when using types as tokens for the dependencies of a service.

2.20 Lifecycle hooks

- Do implement the lifecycle hook interface because lifecycle interfaces prescribe typed method signatures.

- Use those signatures to flag spelling and syntax mistakes.

3.0 Angular Libraries

3.1 Angular Lib: User Interface:

- **ng-bootstrap** is the integration of the well know Bootstrap CSS framework with APIs designed for the Angular ecosystem. No dependencies on 3rd party JavaScript. ng-bootstrap

- **ngx-bootstrap** is another integration of Bootstrap with Angular components, so no needs to include the Bootstrap javascript. ngx-bootstrap

- **ngx-ui** is a UI library made by Swimlane which includes component and style for Angular. ngx-ui

- **primeng** is a collection of rich UI components for Angular developed by PrimeTek Informatics. primeng

- **angular/material** is the official components for Angular made by the Angular team. The team builds and maintains both common UI components and tools to help you build your own custom components. material

3.2 Angular Lib: Routing utilities:

- **angular-component/router** is a declarative component router for Angular applications inspired by Svelte and React routing libraries. angular-component/router

- **routerkit** is a very simple library but which brings great improvements to routing such as strong typing, autocompletion and centralization of routing it is thus easier to understand the application as a whole. routerkit

- **routeshub** is an add-on to Angular's router which provides state-based routing for medium to large applications. routeshub

- **ui-router/angular** is another routing framework for developing state-based routing in Angular. ui-router/angular

- **ngx-router** is an utility to get via dependency injection all the routing information such as the name and the different types of parameters: query params and route params... This utility will help you to avoid duplicate code and also will simplify your smart components testing. ngx-router

- **route-path-builder** is a a very simple yet powerful declarative route path management made with a single class called RoutePathBuilder. route-path-builder

3.3 Angular Lib: DOM

- **resize-observer** is a library for declarative use of Resize Observer API with Angular. It helps to easily observe and respond to changes in the size of an element’s content or border box in a performant way. resize-observer

- **intersection-observer** is a library for declarative use of Intersection Observer API with Angular. It allows to register a callback function that is executed whenever an element they wish to monitor enters or exits another element (or the viewport). intersection-observer

- **ng-resize-observer** is similar to the first library resize-observer above, however this one work is still not completed. ng-resize-observer

- **ngx-visibility** is a library for monitoring the visibility of elements in the DOM. It uses Intersection Observer API in a more performant way compared to others libraries because it keeps the number of observers to a minimum. ngx-visibility

- **ng-in-viewport** is a library similar to ngx-visibility since it also allows to check if an element is within the browsers visual viewport. ng-in-viewport

- **ng-event-plugins** is a tiny library for optimizing change detection cycles for performance sensitive HTML DOM events such as touchmove, scroll, drag... and also provides a declarative command for preventDefault() and stopPropagation(). ng-event-plugins

- **ng-click-outside** is an Angular directive for handling click events outside of an element. ng-click-outside

- **ngx-favicon** is an Angular service to dynamically update the favicon on an app. ngx-favicon

- **ngx-page-scroll** is a small animated scrolling utility for Angular applications which allows you to trigger the scroll after loading, after navigation via the router or on click via a directive. ngx-page-scroll

- **ngx-scroll-to** is library that provides a way to smooth scroll to any element on the page and thus enhance scroll-based features in your app. ngx-scroll-to

- **ngx-ui-scroll** is a library that provides a very powerful Angular directive for unlimited bidirectional scrolling over limited viewport. ngx-ui-scroll

3.4 Angular Lib: Pipes, Directives & Decorators

- **ngx-pipes** is a collection of useful pipes for Angular with no external dependencies. The lib is divided in different modules so you can either add the whole or only the types of pipes you need.ngx-pipes

- **ngx-date-fns** is a library based on the modern date-fns library and offers an Angular pipe for displaying date and time in a convenient way. ngx-date-fns

- **ngx-moment** is a library that provides a set of useful moment.js pipes for Angular. ngx-moment

- **ngneat/helipopper** is a wrapper for tippyjs that provides a directive called tippy and let you create powerful and customizable tooltips, menus, dropdowns and popover for your applications. ngneat/helipopper

- **ng2-tooltip-directive** is another library to facilitate the creation of tooltip in Angular via a simple directive called tooltip. ng2-tooltip-directive

- **ng-directives** is a collection of directives that can be used in different scenarios for instance for showing a delta value arrow, a full-screen switch button, a conditionnal element based on permissions or a highlight a text. ng-directives

3.5 Angular Lib: Utilities

- **angularfire** is the official Angular library for Firebase. It offers a wrapper to use the various features of Firebase in a way that suits Angular conventions. With this library easily add complex modules such as authentication, databases, real-time, monitoring, advertising and more... angularfire

- **angular/cdk** is the official components infrastructure for Angular made by the Angular team. The CDK is a very useful library that helps you author custom UI components with common interaction patterns such as layout responsiveness, scrolling, accessibility... angular/cdk

- **ngx-feature-toggle** is an Angular module to manage feature toggles in applications using a normalized way. ngx-feature-toggle

- **ng-keyboard-shortcuts** is an Angular module that provides a declarative API using components/directive to manage keyboard shortcuts in scalable way. ng-keyboard-shortcuts

- **angular2-hotkeys** is another library that helps the integration of combination of keyboard shortcuts in Angular applications. angular2-hotkeys

- **ngneat/hotkeys** is another library that wants to facilitate the creation of shortcuts in your app thanks to a customizable hotkeys directive and service. ngneat/hotkeys

- **ngx-template-streams** is a library for declarative use of event, such as native DOM events, or component outputs in a reactive way using observable. ngx-template-streams

- **ng-dynamic-component** is a library which allows to use dynamic components with full life-cycle support for inputs and outputs. ng-dynamic-component

- **ngx-route-params-input** allows to use angular router params and query params as component input easily with less boilerplate code. ngx-route-params-input

- **ng-effects** is a library which allows to write more reactive code in Angular. ng-effects

- **ngx-responsive** is a library which contains a superset of responsive directive in order to show or hide items according to the size of the device screen or according to the device informations. ngx-responsive

- **ng-event-bus** is a RxJS-based message bus service for Angular. It helps to centralize communication between all the services and components of an app. ng-event-bus

- **ngx-dynamic-hooks** is a utility for automatically inserting live Angular components into a dynamic string of content and render the result in the DOM. ngx-dynamic-hooks

- **workers** is a small library with great advantage, in fact it offers a comfortable use of the Web Workers API. workers

- **flex-layout** is an official Angular library that provides HTML UI layout for Angular applications using Flexbox and a Responsive API. flex-layout

- **observable-webworker** is a library that offers a simplified reactive API for working with Web Workers. observable-webworker

- **ng-web-apis/speech** is a library for using Web Speech API with Angular. ng-web-apis/speech

- **ng-web-apis/permissions** is a library for using Permissions API in Angular applications. ng-web-apis/permissions

- **runtime-config-loader** is a library that provides an easy way to integrate a runtime configuration through a json file into your Angular application. runtime-config-loader

- **ngx-device-detector** is an utility library that provides a simple way to detect the device, OS, and browser details... ngx-device-detector

4.0 Performance

Following are some of the best practices to improve the performance of a web app.

Name	Priority	Comments
Minification	High	All file should be minified (HTML, CSS, JS)
Lazy loading	High	Images, scripts and CSS need to be lazy loaded to improve the response time of the current page
DNS resolution	High	DNS of third-party services that may be needed are resolved in advance during idle time using dns-prefetch.
Pre-connection	Medium	DNS lookup, TCP handshake and TLS negotiation with services that will be needed soon is done in advance during idle time using preconnect.
pre load & pre fetch	Medium	Pre loading & prefetching improves in TTI & FCP
webp format for serving images	High	
CSS bundle optimisation	High	use purge css to optimise css bundle
JS bundle optimisation	High	
Compression	High	brotli
Caching static assets & using CDN with larger expiry	High	cache-control: max-age=2592000
Inlining Critical styles to serve initial content fast	Medium	Helps in FCP

Add inline styles for application fonts & Serve fonts from cdn	High	<link rel="preload" as="font" type="font/woff2" href="assets/fonts/base-fonts/Nunito_600.woff2" crossorigin />
		
		<style>
		@font-face {
		font-family: Nunito; src: url(assets/fonts/Material_icon_font.woff) format('woff');
		}
		</style>

Service Workers	Low	Widely used with PWAs

5.0 Security

Name	Priority
Regular maintenance of third party or angular libraries	Medium
x-xss-protection	High
Content-Security-Policy	High
X-Content-Type-Options "nosniff"	High
Cross-Origin Resource Sharing (CORS)	High
CSRF	High
strict-transport-security (HSTS)	High
x-frame-options	High
Referrer-Policy "no-referrer-when-downgrade";	High
Feature-Policy (Permissions-Policy)	High
Cookie Scope	High
Cookie flags set to HttpOnly, Secure and SameSite	High
Rate limiting	High

6.0 Deployment

- Remove All console.log and debuggers from Projects files.
- Enable Angular Production Mode before taking build.
- Use AOT compilation build.
- --output-hashing all — hash contents of the generated files and append hash to the file name to facilitate browser cache busting (any change to file content will result in different hash and hence browser is forced to load a new version of the file)

- --extract-css true — extract all the css into separate style-sheet file
- --sourcemaps false — disable generation of source maps
- --named-chunks false — disable using human readable names for chunk and use numbers instead

7.0 External References

https://dev-academy.com/angular-architecture-best-practices/

https://angular.io/guide/aot-compiler

https://angular.io/guide/styleguide

https://wikipedia.org/wiki/Single_responsibility_principle

https://docs.npmjs.com/cli/version

https://stackoverflow.com/questions/273695/git-branch-naming-best-practices https://cli.angular.io/

https://medium.com/beautiful-angular/angular-2-and-environment-variables-59c57ba643be

https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md

https://gist.github.com/digitaljhelms/4287848

https://www.npmjs.com/package/codelyzer

https://medium.com/@tomastrajan/6-best-practices-pro-tips-for-angular-cli-better-developer-experience-7b328bc9db81

https://medium.com/medialesson/why-and-how-to-structure-features-in-modules-in-angular-d5602c6436be

https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines

8.0 About iLink Digital

iLink Digital, a CMMI Level 3 and ISO 9001:2008 Certified Global Software Solution Provider and Systems Integrator, delivers next-generation technology solutions to help clients solve complex business challenges, improve organizational effectiveness, increase business productivity, realize sustainable enterprise value and transform your business inside-out. iLink integrates software systems and develops custom applications, components, and frameworks on the latest platforms for IT departments, commercial accounts, application services providers (ASP) and independent software vendors (ISV). iLink solutions are used in a broad range of industries and functions, including healthcare, telecom, government, oil and gas, education, and life sciences. iLink's expertise includes Cloud Computing & Application Modernization, Data Management & Analytics, Enterprise Mobility, Portal, collaboration & Social Employee Engagement, Embedded Systems and User Experience design etc.

What makes iLink Digital's offerings unique is the fact that we use pre-created frameworks, designed to accelerate software development and implementation of business processes for our clients. iLink has over 60 frameworks (solution accelerators), both industry-specific and horizontal, that can be easily customized and enhanced to meet your current business challenges.

For more information, please visit us at https://www.ilink-digital.com/