Scalable Angular Application Architecture

18.07.2017

Currently, one of the most popular frameworks among the Web community is Angular (or Angular 2 as some prefer). The main reason why we in GFT have decided to use it in our projects, is its comprehensive character, and a strong push it gives towards consistent project structure and architecture.

Unfortunately, even a framework as opinionated as Angular can only enforce the basics of application architecture. That's sufficient for small or medium applications, however, in GFT we usually have to deal with big applications, composed of dozens of modules and filled with complex data collections and complicated user flows. What is more, our projects are often developed by a team scattered across different continents and time zones.

In order to maintain high quality of delivery and prevent technical debt from being created, we had to agree to a series of guidelines and good practices of how to plan, structure and write applications in Angular.

These architecture principles can be divided into three main categories:

1. Project structure – how to organize you project files, define and work with Angular modules and their dependencies
2. Data flow architecture –devise a guide on how to define that way the data flows through your application layers
3. State management – how to manage the state of GUI and propagate it between different application parts

This article is a combination of community–inspired guidelines and the experience that we've gathered working in our projects.

What is a scalable architecture ?

First of all, what does it mean that GUI application is scalable? GUI always runs as a single, separate application for every user, so there is no "high number of users" challenge that is typical for backend applications. Instead, GUI has to deal with the following scalability factors: increasing size of data loaded to the application, growing complexity and size of the project, usually followed by longer loading times.

There is also a part of the problem that is not visible from the outside, namely how an application scales from the programmer's point of view. An application with bad, or not scalable architecture, tends to be very hard to develop after some time. Increasing complexity, technical debt and simply code smell, has a direct impact on project estimates, costs and the quality of the overall solution.

Good architecture does not guarantee that above problems will not occur in your application, however, it gives the development team a powerful tool to reduce and even eliminate most of the issues that they might encounter.

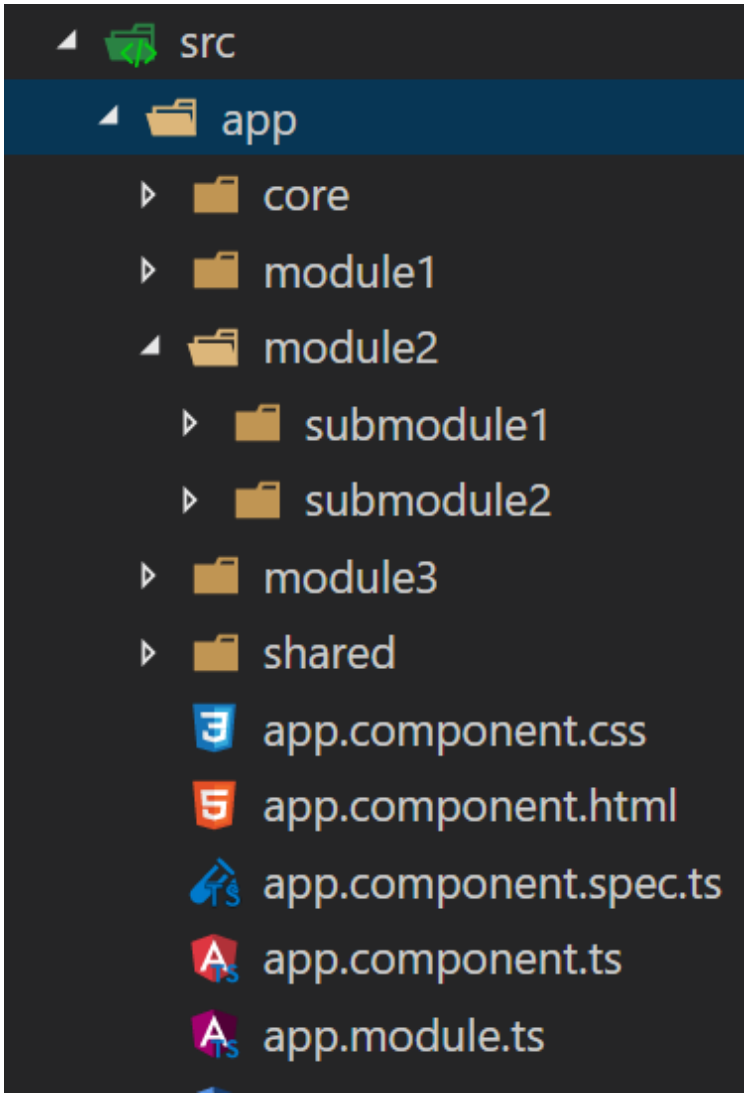
In short, well designed architecture should work equally good for small and big applications, should provide very good user experience regardless of the application's size and amount of processed data. Additionally, it should provide a set of clear and easy to follow rules for developers in order to sustain the quality of the project. And finally, it should be simple and preferably based on widely accepted design patterns. We want to keep the learning curve of our applications as small as possible.

Let's now take a walk through the principles of well-designed Angular applications.

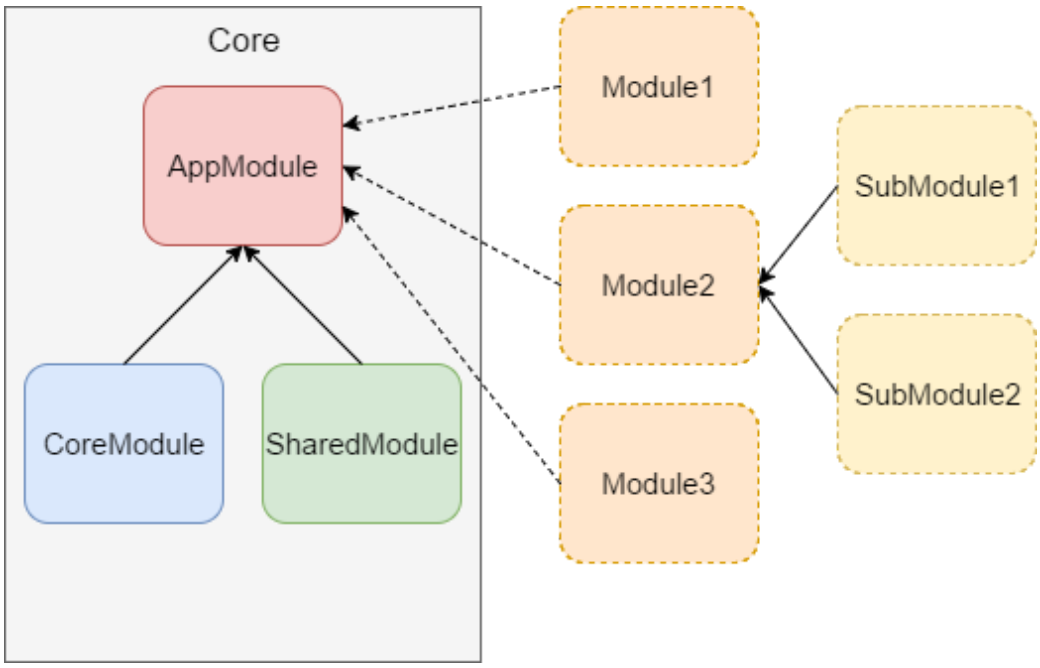
Structure of the application

Usually, one of the first things you do when you set up a new project is to define the structure of the application. There are many possible ways to do that, but the one that is considered to be officially recommended, and the one that we've decided to follow in GFT, is the module-oriented project structure.

In this approach, application modules are clearly visible in the file tree, as separate directories. Every module directory contains all files (code, styles, templates etc.) that are related to a given module. A very important element of this approach is isolation of modules. Simply speaking, it means that every module is self-contained and does not refer to files from different modules, so, theoretically, you can delete one of them from the application, and the rest will work without any problems.



Obviously, it’s not possible to strictly follow this rule in the real world. At least some services and components have to be reused across the whole application. Therefore, some parts of application functionality are stored in "Core" and "Shared" modules. Now our application structure looks like this:



As you can see, there are now three main modules in the project:

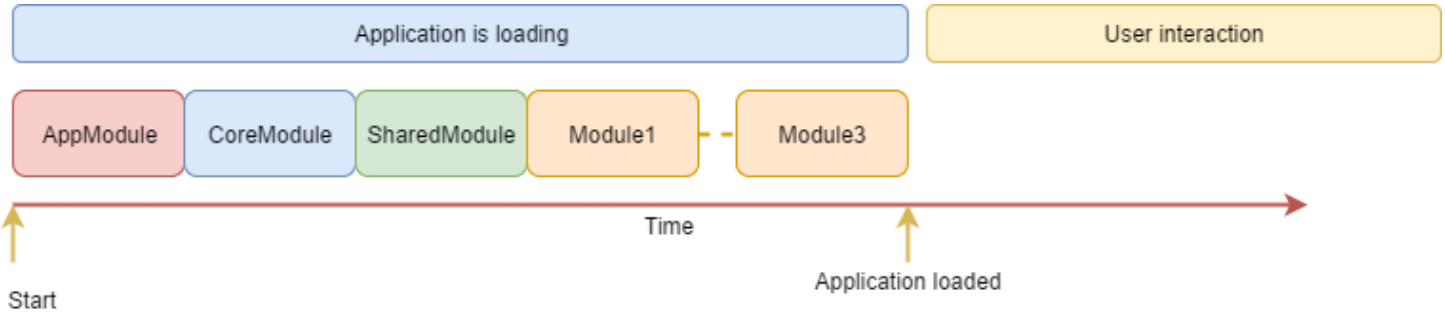
1. AppModule – the bootstrapping module, responsible for launching the application and combining other modules together
2. CoreModule – core functionalities, mostly global services, that will be used in the whole application globally. They should not be imported by other application modules
3. SharedModule – usually a set of components or services that will be reused in other application modules, not applied globally. They can be imported by feature modules.

All remaining modules (so-called feature modules) should be isolated and independent. Such a structure not only allows for clear concerns separation, but is also a convenient starting point for implementing lazy loading functionality, another crucial step in preparing a scalable application architecture.

Lazy Loading

One of the most visible performance problems that users may encounter is the loading time of the application. As the codebase gets bigger, the size of the application bundle increases as well. The bigger the bundle, the longer it takes for the browser to load and parse the source code.

There are multiple ways in which we can reduce the size of the application bundle, however, those are not within the subject matter of this article. Additionally, sooner or later, despite all efforts, the bundle will be big enough to cause visible delay in the application's start.

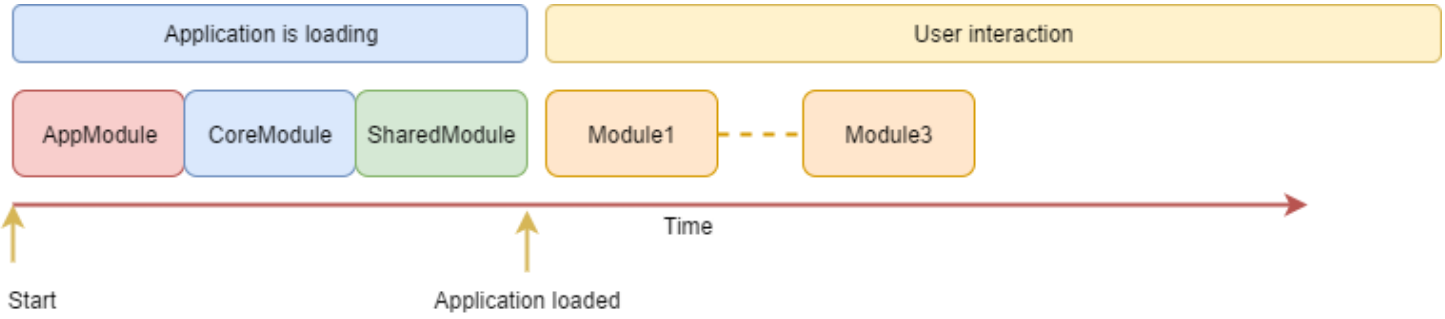


Fortunately, there is a pattern that is aimed to solve the loading time issue, that is lazy loading. It allows to deferthe loading of a particular part of the application until it's actually needed (usually when the user wants to access a particular screen of the application).

Angular comes with a built-in Lazy Loading capability, you just have to properly define the module route definition, so that it points to a module file that will be lazy-loaded.

```
const appRoutes: Routes = [
  { path: 'dashboard', loadChildren: 'path/to/dashboard.module#DashboardModule' },
  { path: 'transactions', loadChildren: 'path/to/transactions.module#TransactionsModule' },
];
```

If you followed the previously described project structure, your feature modules can now all be lazy loaded on demand, after the application is initialized. This vastly reduces the initialization time of the application, improving overall user experience. What is more, this solution scales with the application. When the application grows and more lazy loaded modules are added, the application core bundle and, therefore, its starting time remain the same.



If a default, load on demand approach is not sufficient, it's possible to define so-called preloading strategies that can be used to customize how the modules are loaded in a given application.

Data flow and state management

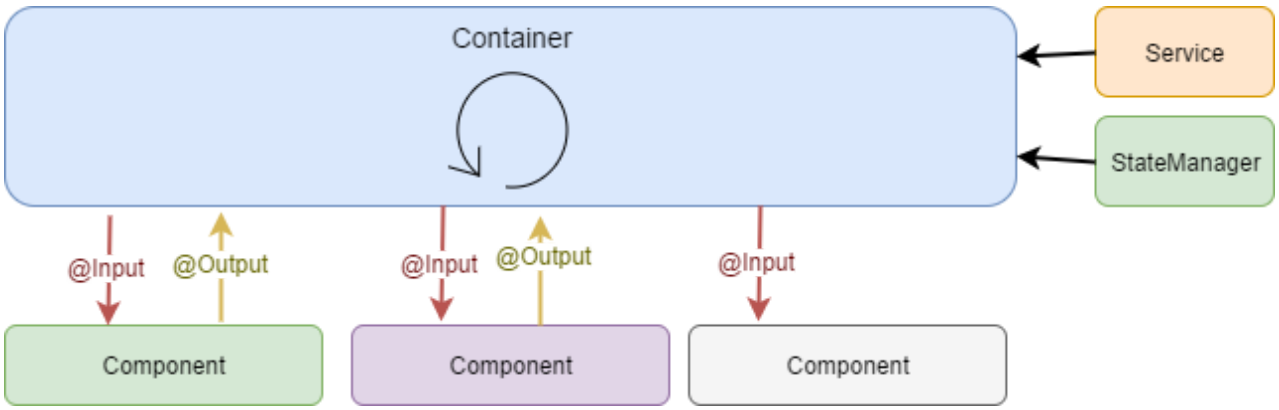
Architecture is not only about the structure, files and modules. In complex GUI applications, the biggest challenge we face is dealing with the problem of data and state information flow that occurs through layers of an application. This is usually the part of the application that gets complicated very quickly and the one that is frequently modified. Therefore, having a well designeddata and state management should be a crucial checkpoint on every architect's to-do list.

Data flow

Let's start with data flow. Angular 2+, unlike its predecessor, prefers a one way data flow. This kind of approach is much easier to maintain and follow than two-way data binding. It is more obvious what is the source of data in a given module and how the change is propagated through the system. In Angular, data flows from top to bottom. From the parent component to the child component and from the component to the template.

Still, it's a good idea to add some additional set of rules over the basic data flow principles.

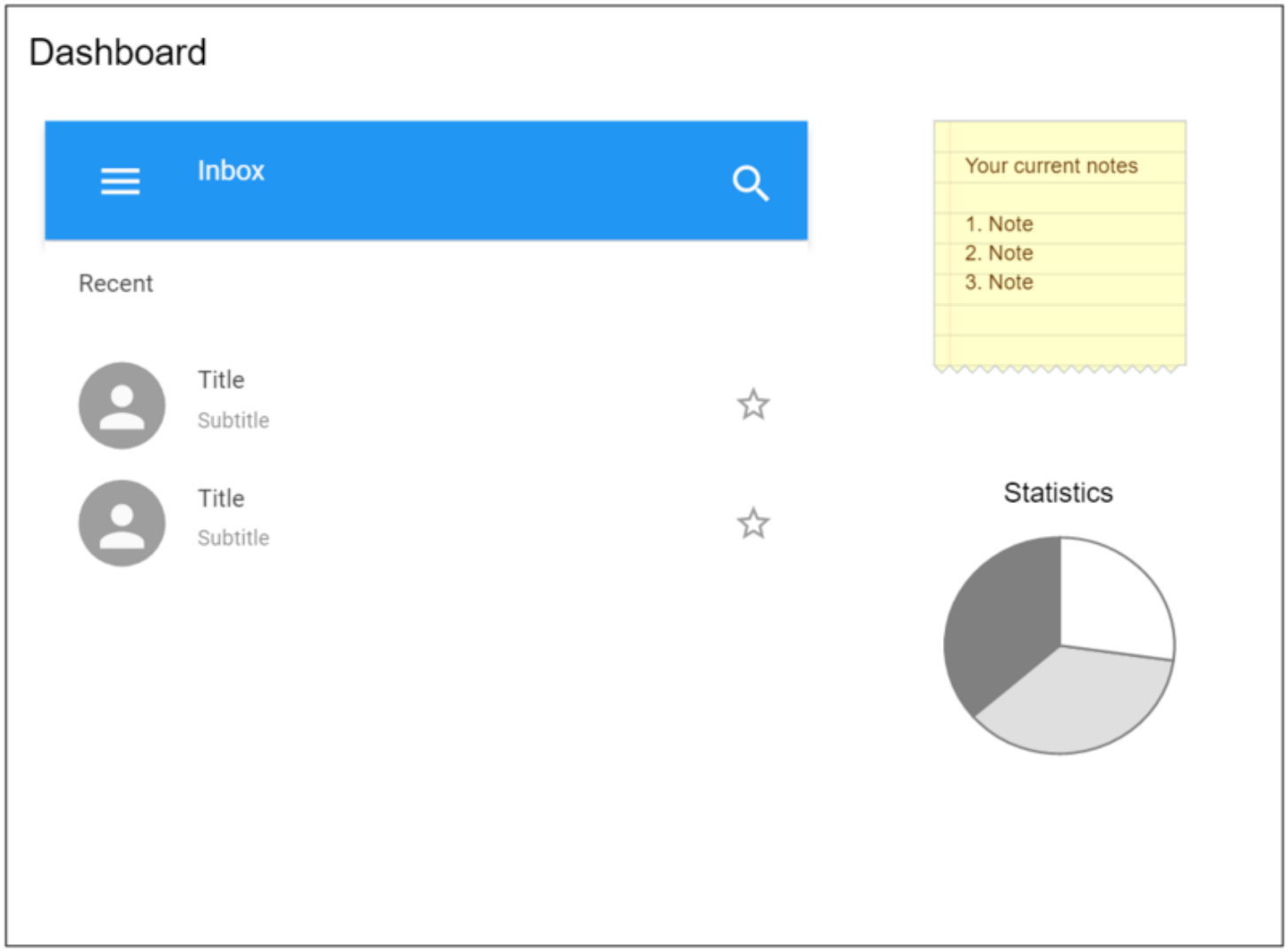
In our application we've introduced the idea of "smart" and "dummy" components. The smart components are also called "Containers". The idea behind this division is to clearly define the parts of the application that containsome logic, communicate with services and cause side effects (like service calls, state updates etc.). Every such action is implemented only in Containers. On the contrary, "stupid" components have very little or no logic at all. All the data they need is passed by @Input parameters. If a component wants to communicate with the outside word, it has to emit an event (via @Output attribute).



Such architectural approach is intended to keep the number of Containers as small as possible. The more components in the application are "dummy", the simpler is the data flow and the easier it is to work with it. Deciding which component should take the role of a Container and which should be just a plain component is not a trivial task and needs to be resolved per particular case. However, usually the first step we take is assuming that a main screen component should be the smart one, as in the example below, when the container is marked with blue color and simple components are gray.

After that, it's a matter of finding a good balance between a small number of containers and keeping the single responsibility principle.

A dashboard screen is a perfect example of multiple smart containers displayed on a single screen. Every dashboard tile would be smart and responsible for its own behavior, data and logic.



Such an approach to architecture is not only aboutreadability of code and organized data flow. Dummy component are much easier to test. Their state is entirely induced by the Input they are provided with, they cause no side effect and the result of any component action is visible as a proper event being fired.

What is more, such behavior nicely corresponds with performance optimization of Angular's change detection process. The change detection strategy for dummy components can be set to "onPush" which will trigger the change detection process for the component only when the input properties have been modified. It's an easy and very efficient method of optimizing Angular applications.

State management

There is one special type of data that flows through the application, namely state. A GUI is all about the state. Everything that the user sees on the screen is a reflection of the state of the application. When they perform a certain action, some data is loaded, or any other event occurs - application logic is executed and the GUI state is modified. Such modification usually triggers the rendering process and the view is updated so the user can see the results of the performed action.

Such a cycle occurs multiple times in an application's lifetime and is basically the backbone of all interactions in the GUI application.

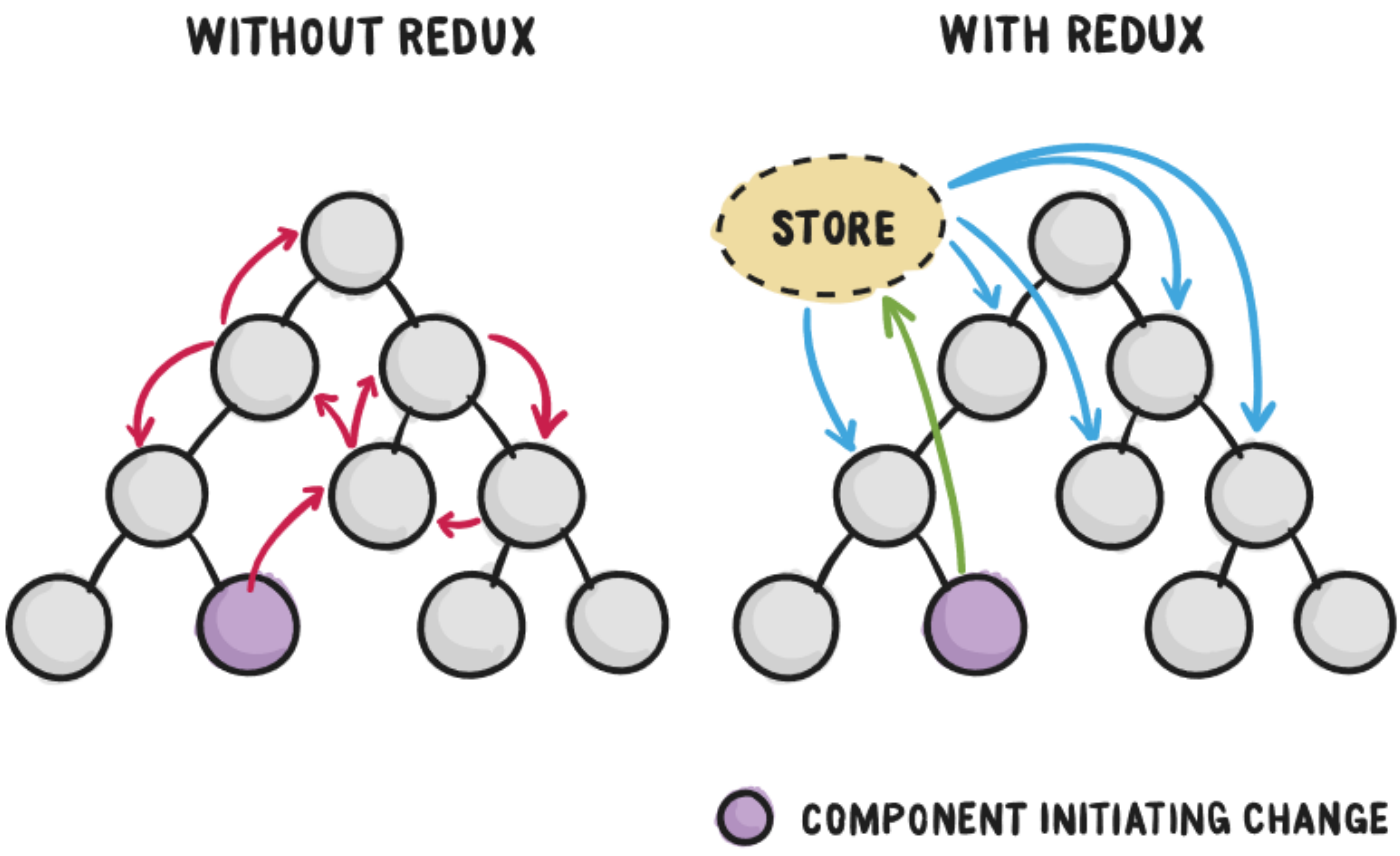
While the process itself is fairly simple, complex applications suffer from many issues related to state management. This is caused by the fact the typical interaction goes through all layers of an application, which makes it hard to follow and debug. A given state is usually shared and information stored there affects multiple components and even screens at once. As it was already mentioned, state operations are also usually the most complicated ones and the ones that are modified frequently.

Single store to rule them all

One of the ways to deal with those issues is to leverage the unidirectional data flow on an application-wide level. The Angular community has widely adopted the Redux architecture pattern, originally created for React applications.

The idea behind Redux is that the whole application state is stored in one single Store, the object that represents the current state of the application. A Store is immutable, it cannot be modified, every time a state needs to be changed, a new object has to be created.

One, single point of reference for the entire application state simplifies the problem of synchronization between different parts of the application. You don't have to look for a given piece of information in different modules or components, everything is available in the store.



Source: https://css-tricks.com/learning-react-redux/

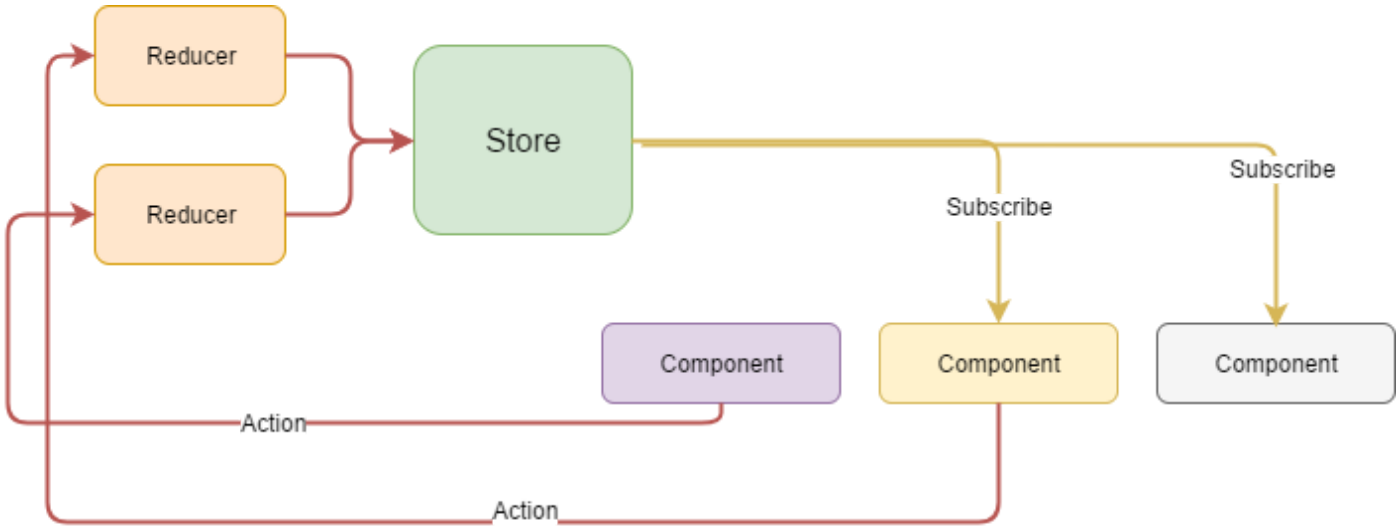
Actions and Reducers

A Store cannot be accessed and modified directly: the so called reducers are responsible for creating a new state with modified data. A Reducer is usually a simple function that takes an action as an argument, and based on the provided information, it returns a new state value.

```
let reducer = (oldState: AppState, action: Action): AppState => {
    switch(action.type) {
        case 'INCREMENT':
            return {
                counter: oldState.counter + 1
            };
        case 'DECREMENT':
            return {
                counter: oldState.counter - 1
            }
        default:
            return {
                counter: 0
            };
    }
};
```

State propagation

There are multiple implementations of Redux pattern available in the Angular ecosystem, one of the most popular ones is definitely *ngrx/store*. We will not go into details of this particular implementation; what we are interested in is how the state is modified and propagated through the application.

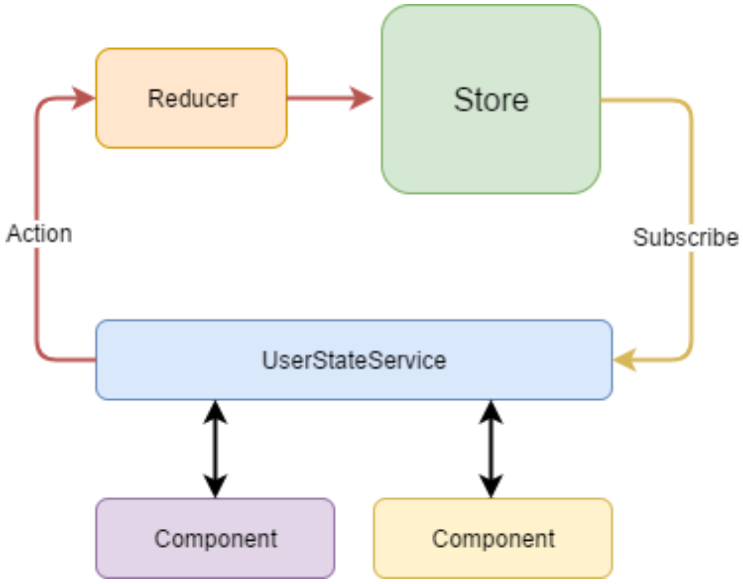


Once the modification to the store is applied, all subscribers are notified about the change. Thanks to the fact that a state is immutable, you don’t have to perform any deep checking - if the state has been modified, the subscriber will receive a new object instance.

Thanks to the centralized nature of this pattern, every state modification is properly propagated to all parts of the application. The only rule you have to obey is to reflect all state changes in the store, in order to avoid keeping them in isolated components.

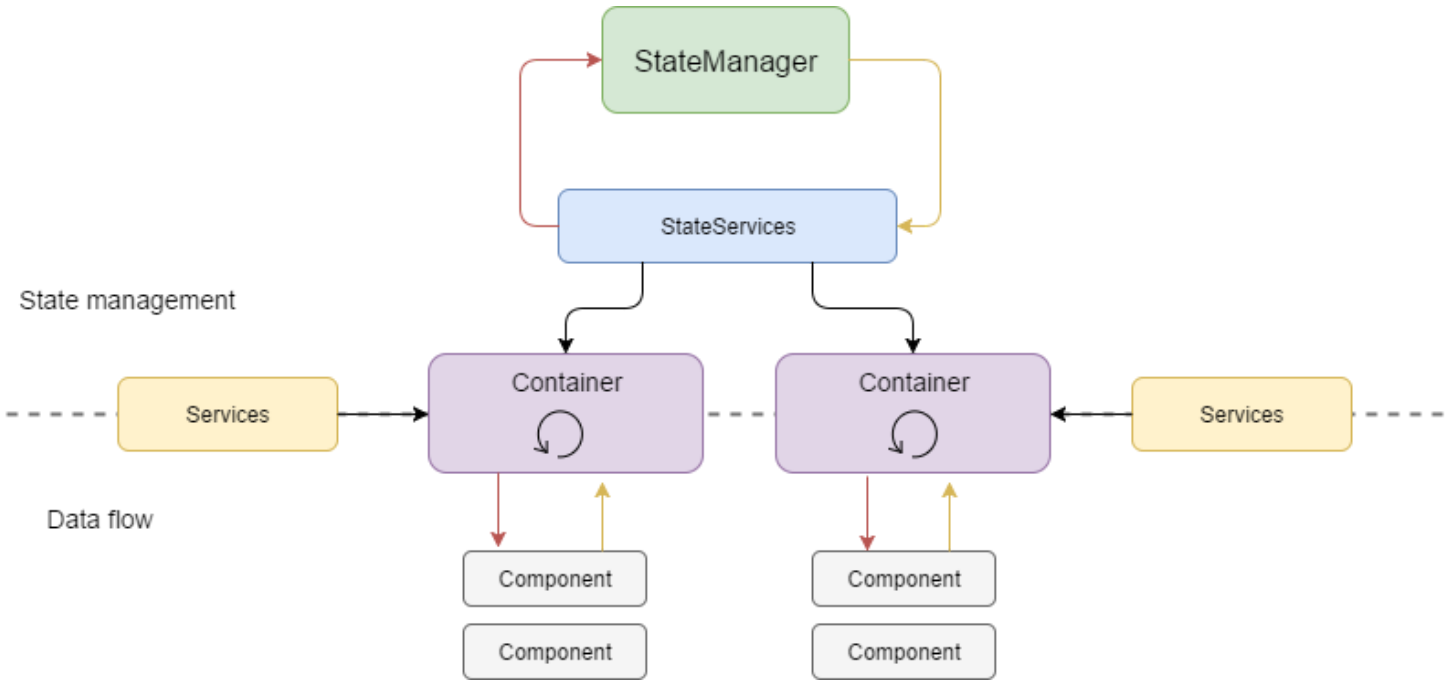
State services

As with other kinds of logic that are not directly related to the view layer, it's a good practice to encapsulate Store operations in dedicated services. Such services can then be reused across the module or application, and our components do not have be aware of the details of state operations.



Combining state management and data flow

The patterns described are really powerful when combined together. Redux-base state management provides bullet proof synchronization between various parts of the application and the idea of an immutable state. Unidirectional data flow and the idea of smart containers provides well-defined points of responsibility in the application and the pattern of how to propagate upcoming state modifications to the dummy components below. The immutable nature of shared data nicely fits in the optimization patterns of Angular change detection algorithms.



An application designed in this manner is composed of encapsulated, autonomous components thatsynchronize their state and data through the central state management system.

What next?

While this document describes high level architecture that we use in Angular applications created at GFT, it still does not address all architecture details and challenges you may encounter.

There are still pending discussions on the best approach to state management of lazy loaded modules, details regarding the performance optimizations and tooling used in the development process.

However, following the abovementioned rules will give you a very solid and scalable foundation for any applications you will create.