**Software Architecture Patterns for Front-End Development**

Architecture Patterns for Front-End Development



Photo by [Luca Bravo](https://unsplash.com/@lucabravo?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

Different Software Architecture patterns are widely researched and applied in back-end development. Front-end, beside its relatively short history, lacks such kind of well-established techniques. Nevertheless, in last years with the growing popularity of different front-end frameworks like Angular, React etc. different architectural best practices are also developed. Let us have a look at some patterns that are inevitable part of nowadays’ front-end projects.

***MVC*** (Model-View-Controller) was one the first patterns that was introduced decades ago, where you would try to split different concerns such as Presentation, Data Handling to different actors. Over time derivatives of MVC: [***MVP***](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter) and [***MVVM***](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel) are developed. Modern Front-End Development is greatly influenced by MVC (even from above mentioned successors…) and some points did not change. e.g. Separation of Concerns (SoC).

*In computer science,***SoC***is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.*

SoC is the main trigger that following architecture patterns are initiated:

***Modular Architecture —***if we take Angular Framework as an example, we can see that application is splitted to different modules based on different domains. Each Module hast its own [*NgModule*](https://angular.io/guide/architecture-modules), which are independent containers, being gathered all in AppModule: mother of all modules. Also Core and Shared Module are important Modules to be considered.

***Micro Frontends*** — this is just extension of micro services architecture to front-end layer. As project grows with multiple functionalities it becomes unmanageable in terms of dependencies, builds deployments. Based on micro frontends concept front-end app will be separated to different use case areas. This use case area either have responsible front team members or in big applications separate teams handling end-to-end micro areas.

***Component Architecture —***this pattern is also derived from SoC, being more fine granular separation of different presentation units in components. This means if we have for example ui listing books, we would need to create book component having all necessary code for showing list of books. In book component folder we would have .html file for view, .ts file for component specific business logic, .scss (or .css) file for styling and .spec.ts file for testing.

***Dumb-Smart Components***— is the next pattern that SoC initiated. Here we have two kind of components: Dumb components that are only for presentation, Smart Components that are responsible for data flow, separating presentation from data injection. This happens mainly through two-way data binding based on @Input, @Output (EventEmitter<T>) in Dumb Component. With this annotations Dumb Component gets relevant data from Smart Component, or sends data to Smart Component. Smart Components generally injects Service or Facade and copes with data flow. Please See relevant gist [here](https://gist.github.com/alizeynalli90/b7aa7b23c74b86f25de0a610f51a4617).

***State Management —*** as one of the outcomes from SoC was separating data handling from presenting it, soon it was realised that all UI data can not be fully isolated in back-end. So there was a necessity to have an extra instance in front-end that keeps track of temporary data. Based on this idea, in nowadays front-end frameworks there is a data storage instance called State Management. In React it would be Redux, in Angular NgRx (Actually, both base on Redux Pattern). The Building blocks like Selector, Store, Reducer, Action, Effects work on event-based asynchronous data streaming. Check out documentary for more [info](https://ngrx.io/guide/store).

***Unidirectional Architecture*** — this mainly incorporates with reactive programming. The data flow in nowadays frontend frameworks are unidirectional, powered with data flow streams. The Data flows only in one direction and it is towards the view. The view will in turn activate different actions. This gives better control over data. Different libraries like RxJs, NgRx, Flux enables multiple functionalities when working with data streams.