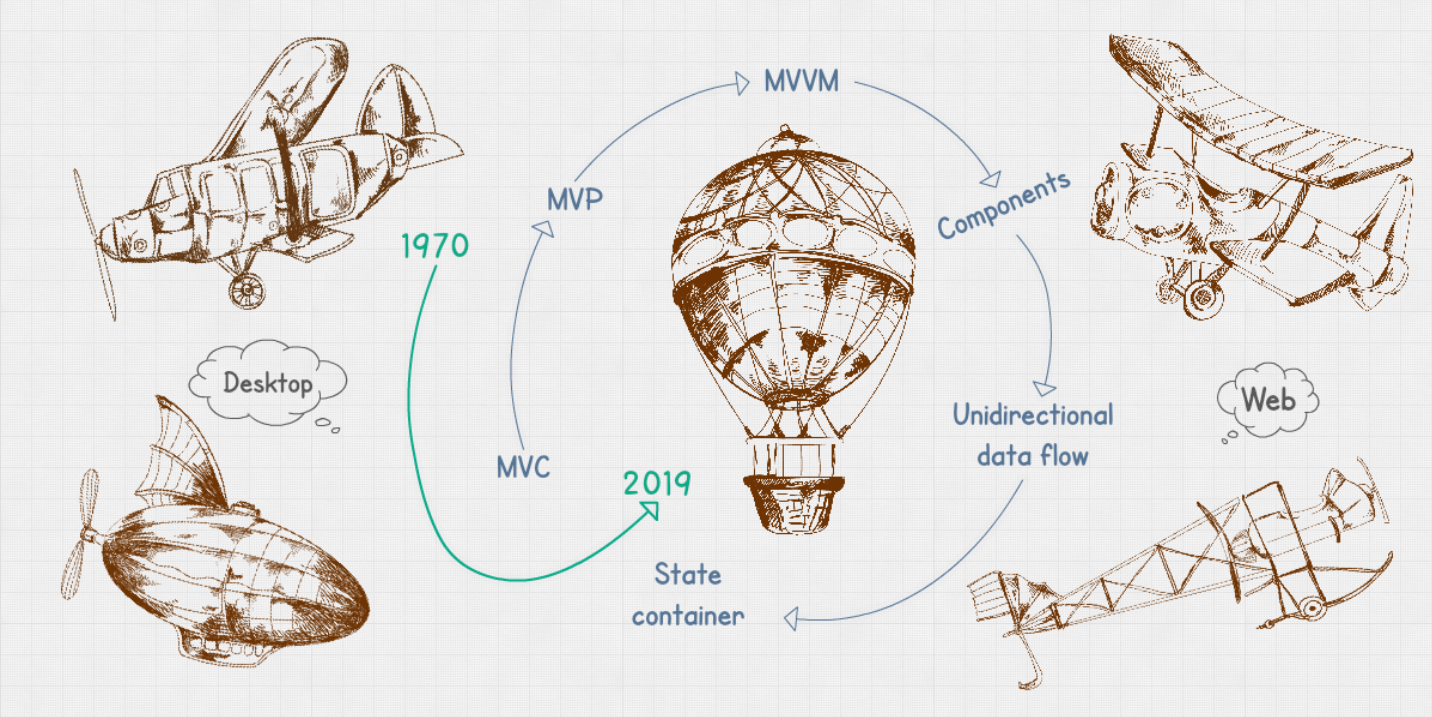
Contemporary Front-end Architectures

How we got here? A front-end engineer's perspective!

Harshal Patil

Sep 5, 2019 · 39 min read



Contemporary Front-end Architectures

Reasoning about the Data Flows within different components of a software system is the central idea of software architecture.

> *The quality of the architecture is the measure of ease with which you can justify this reasoning!*

Exploring these data flows and ultimately architectures in today's **web applications** is the motivation behind this article. Web applications have evolved from simple static websites (two-tiered architecture) into complex multi-layered SPA and SSR driven API first systems. CMS systems have grown into Headless content-first systems.

Front-end community has changed rapidly in recent times. It started with DOM infused algorithms introduced by jQuery, which was quickly succeeded by MVC based Backbone.js. And, in no time, we found ourselves in the jungle of bidirectional and unidirectional data flow architecture. Somewhere, we lost the track of how we got here. How did the world that was so drenched in MVC suddenly got into React pioneered unidirectional data flow? What is the correlation? As we progress, we will attempt to unlock this puzzle.

Though aimed at front-end engineers, the article should help any web developer seeking a general understanding of modern web application architecture. Software Architecture underpins large number of activities — process, requirement gathering, deployment topology, technology stack, etc. However, that is outside the scope of this article.

. . .

A Necessary Prelude — What is a Computer?

A prelude is necessary here. A computer is a machine that collects data/information from user and provide it back to the user after processing it, either instantly or later. How does computer collect and show this data? It uses software application to achieve this.

> The struggle of software architecture is to provide reasonable means to compose the software without losing the sanity.

The key thing to remember is that the data, which software application is processing, is known as **The Model** or **The Application State**. Few brave souls might call it **Domain Model** or **Business Logic** of the application. An application could be a desktop or web application.

Throughout the article, our struggle is to understand reasonable ways of presenting this application state to user (of web front-ends) without losing our sanity. We will explore how data flows from model to the view layer. There is nothing else we should talk.

. . .

Father's MVC — The original

Separating data from presentation is the core theme of Graphical User Interfaces (both web-based or desktop-based). With MVC — Model View Controller, separating presentation (View) from domain concerns (Model) was the primary design motivation. And without doubt, MVC was a seminal work which would influence generations to come.

> If there could be first principle of software development, then it is SoC — Separation of Concern. And probably, MVC pattern is its first bold manifestation.

MVC was introduced for Smalltalk-80. In MVC, **View object** displays the data held by a **Model object**. Before we can fully study data flows in MVC, we must understand software application environments of that time (circa 1970's):

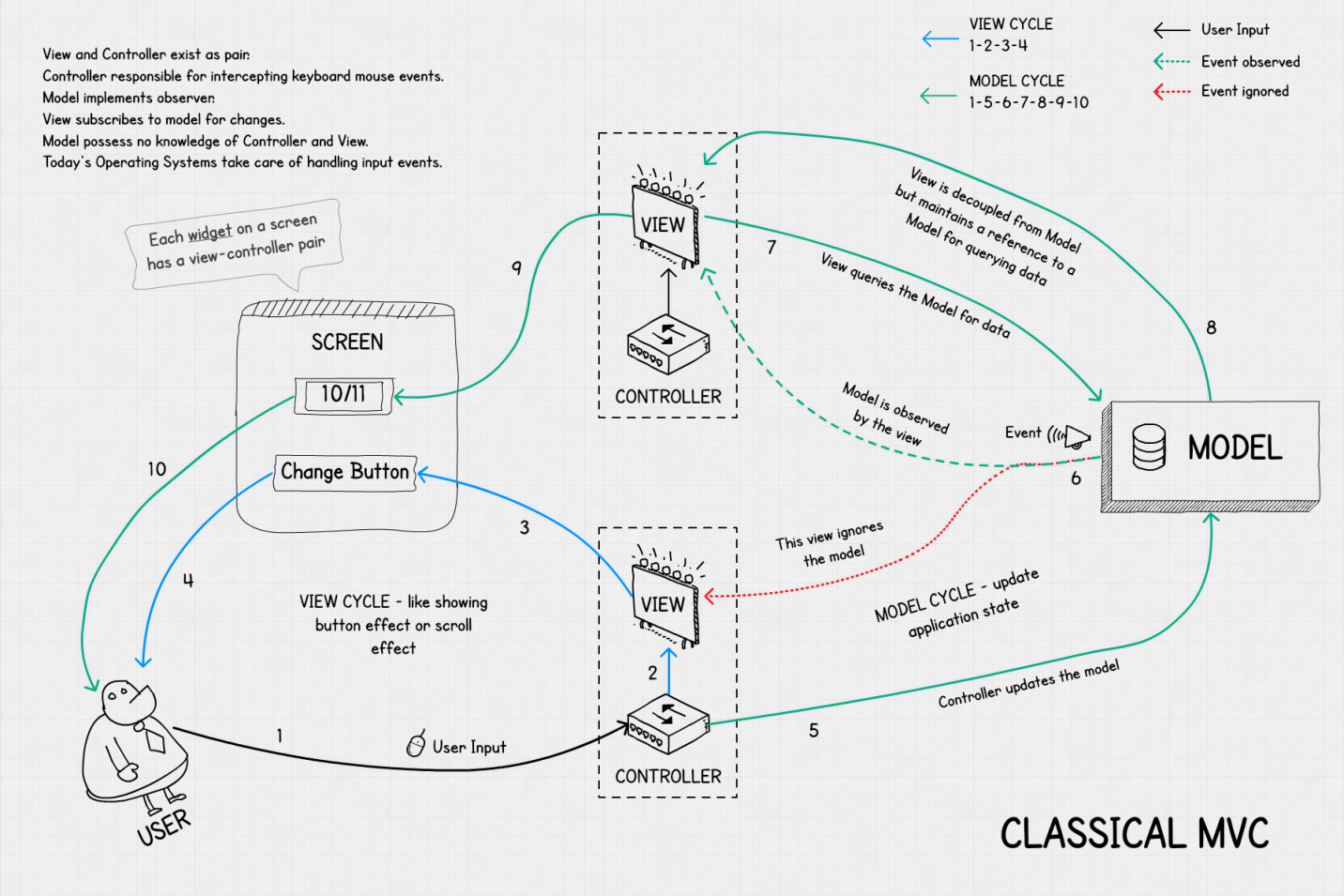
- This MVC was only meant for desktop application. Web was not born. We are talking about a decade prior to that.
- Forget Web, sophisticated GUI driven operating systems did not exist.
- It means that application software was very close to underlying hardware.

Above constraints had important implications on MVC. It became the responsibility of the **Controller object** to respond to user inputs like keyboard or mouse and translate into actions on the model. Also, lack of GUI widgets by operating systems means the view does not correspond to screen.

> *Rather view and controller would exist as a pair. View part of the pair would show user output and controller part of the pair would receive inputs from the user. It should be noted that view-controller pair would exist for each control on the screen giving us an early* ***concept of widget.***

> Today, in React, Vue or Angular this view-controller pair is conceptually same as component although exact mechanics are different with respect to handling state.

With all said and done about MVC, following image should illustrate data flows in MVC. In this example, we have a simple counter with Increment and Decrement button. The counter state is maintained the **Model**. Also, we have replaced two buttons with one button to keep it simple.



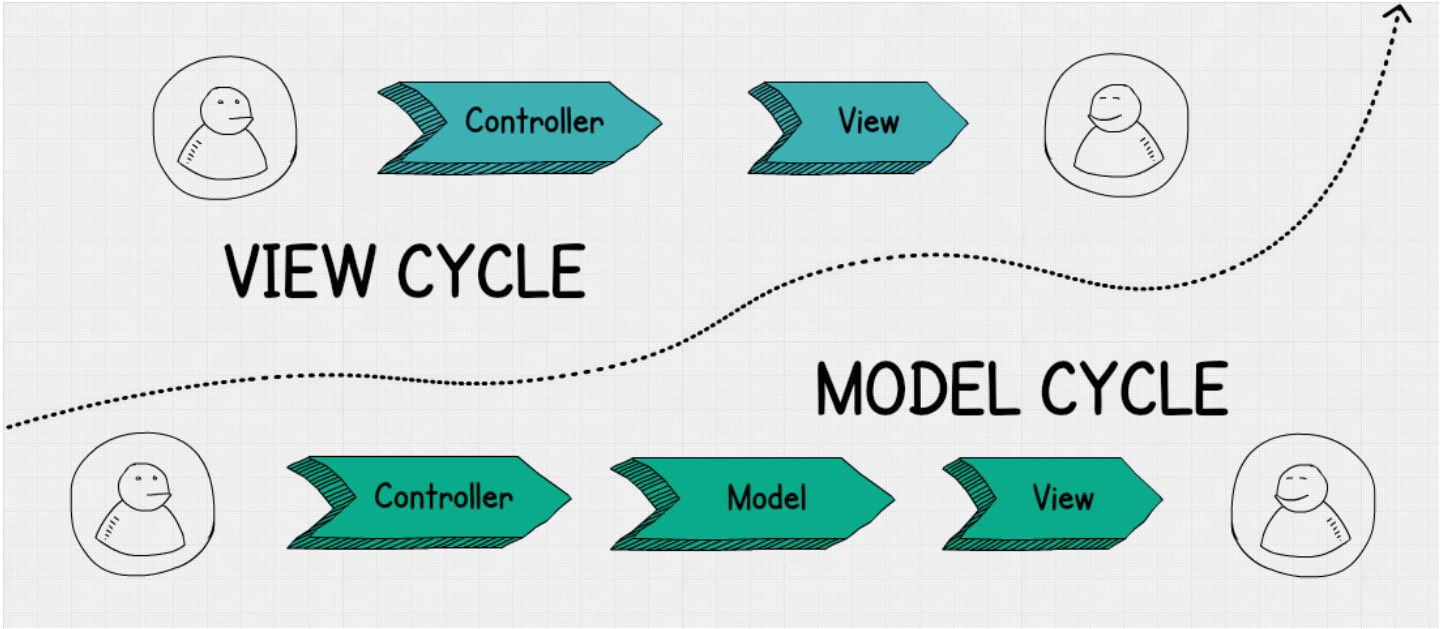
Classical MVC or Father's MVC

In terms of collaboration:

1. **View** and **Controller** contains a direct reference to **Model** but not vice versa. It means Model is not dependent on UI and can change without worrying about UI concerns.

2. **Model** implements Observer pattern and one or more **View** objects subscribe to it. When Model changes, it raises the event and **View** finally updates after reacting to the event.

There are two distinct data flows in MVC. In View only flow, Model is not involved. It is UI only change. Showing a button click effect or reacting to mouse scroll event is the example of View only flow.



Data cycles in MVC

Today, we no longer use this MVC and thus sometimes referred to as **classical MVC** or **Father's MVC**.

Get Going with Application Model

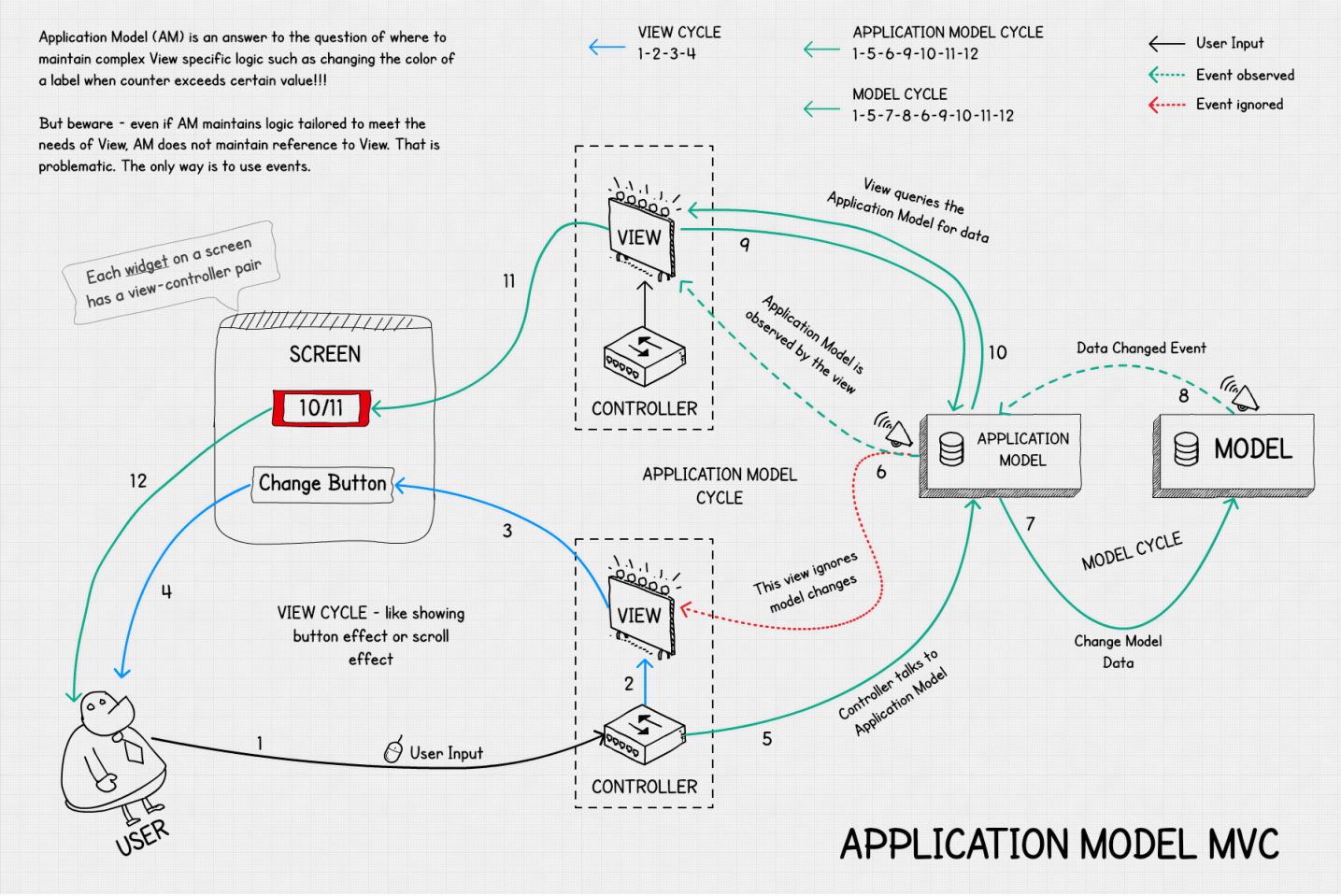
Soon it was realized that Application State cannot be fully isolated from the GUI. There always exists some sort of **Presentation Logic** or **View State** which must be maintained.

> *Complex graphical interfaces need additional state which exists only for helping UI widgets and enabling better user experience.*

But this can cause a problem. Let's take our earlier example of counter. When our counter hits 10, we must change the color of the label from black to red to indicate warning. This color change behavior is not really a business logic or concern. It is purely aesthetic part (user experience) which needs to be accommodated. The real question is — where? Should it be a **Model** or **View**?

Since this **Presentation Logic** or **View State** is basically a state derived from the **Domain Model**, it must be maintained in the **Model** object. But then Domain Model maintaining visual aspect i.e. red color feels awkward by the definition. So, if we put this in the View object, then it introduces another set of problems. Our label widget is no longer generic. We cannot really reuse it elsewhere. Further, putting condition with hard-coded number 10 in our View object means we are leaking some part of business logic.

To solve this problem, another entity was added to the original MVC — **Application Model (AM)**. With **AM** in picture, **view-controller** pair doesn't access Model directly. Instead, they register with **AM** events and use it to access the required data.



Application Model MVC

Data flows remain same as that of classical MVC. Of course, every pattern has its pros and cons and AM-MVC is no exception. Most prominent problem is that **AM** has not direct reference to **View** object and thus cannot manipulate it directly even if **AM** is designed to maintain **View** state.

In general, introduction of Application Model moves the **View** specific state away from the domain layer and helps simplifying **View** objects by reducing their complexity. This is very similar to **Presentation Model**, a concept coined by Martin Fowler in his seminal research:

> *The essence of a Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen. A view then simply projects the state of the presentation model onto the glass.*

. . .

Age of Modern Desktop Architectures

Fast forward into the time and things changed. New breed of Operating Systems was in full force. Applications were far away from the underlying hardware. Full blown kernel, OS drivers and utilities were present between them. GUI based operating systems like Windows provided UI widgets out-of-box.

> It was no longer necessary for Controllers to listen to input devices. The idea of View object changed.

Most of the **Controller** functionality was taken care by the Operating System. The idea of **View** changed. Earlier it was a single widget. Now it was a composition of widgets. A **View** can contain another view. **View** became bi-directional in a sense it responds to user actions as well as displays the **Model** data.

> The idea of View in a front-end world is very much analogues to this idea of View. In the context of Web, a View is an entire page.

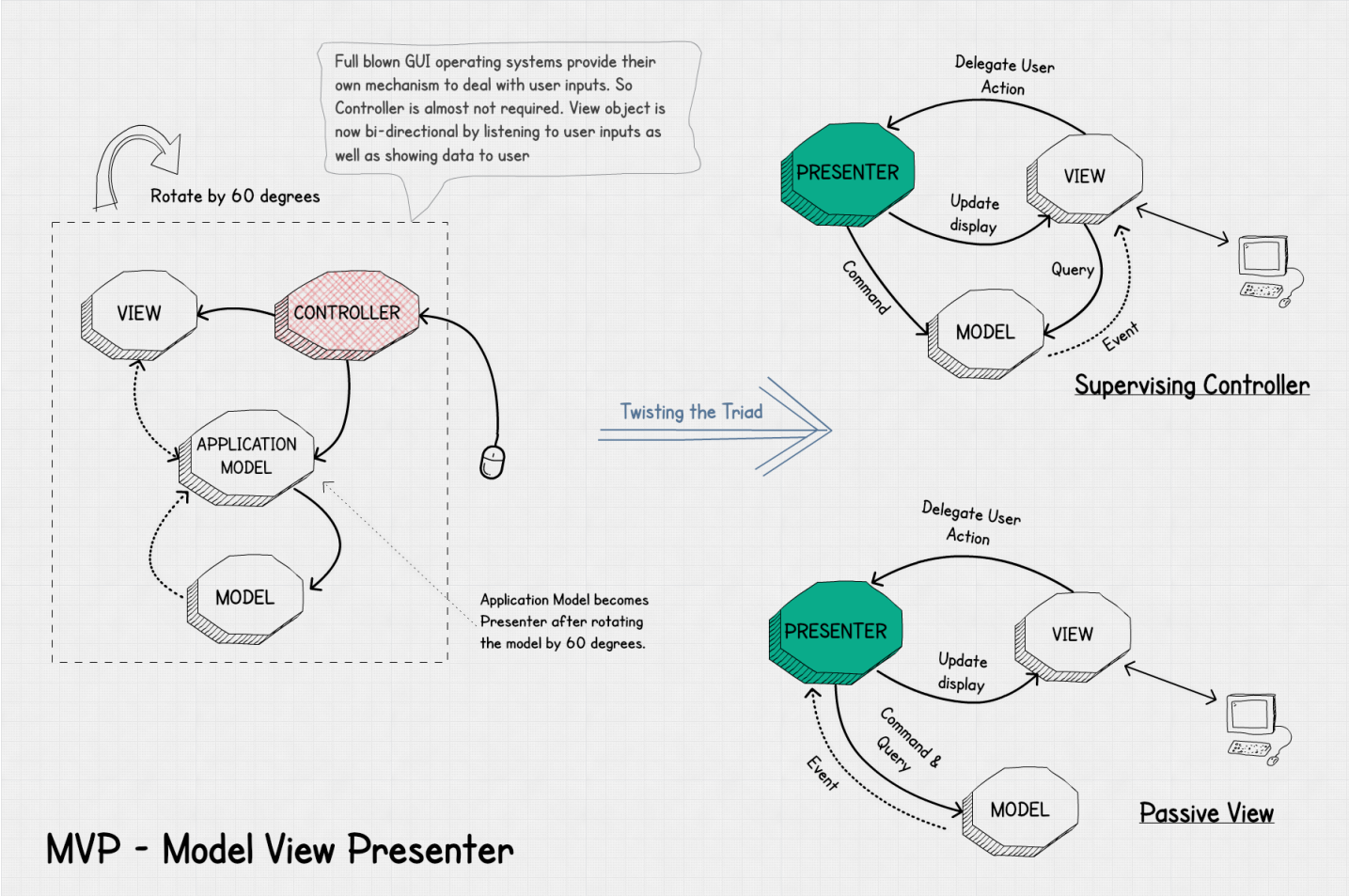
The classical MVC was becoming obsolete and awkward to use. To accommodate these changing environments, Dolphin team was looking for a new model of creating user interfaces. That was year 1995. The history of how Dolphin team reached a new design is very well documented here and here. We need not dwell into that.

> *In summary, the team ended up **rotating the MVC model by 60 degrees** which they called it as **Twisting the triad**. That's how we get MVP.*

In terms of collaboration:

1. The **Presenter** oversees the presentation logic. The presenter can change the view directly. **View** delegates user events to the **Presenter**.
2. Depending on the implementation, View subscribes to the Model and relies on Presenter for complex logic or in other case, **View** simply relies on **Presenter** for everything.

As concluded in his papers on GUI architectures, Martin Fowler has divided MVP implementation into **Supervising Controller MVP** and **Passive View MVP**. Differences and data flows are explanatory from the diagram.

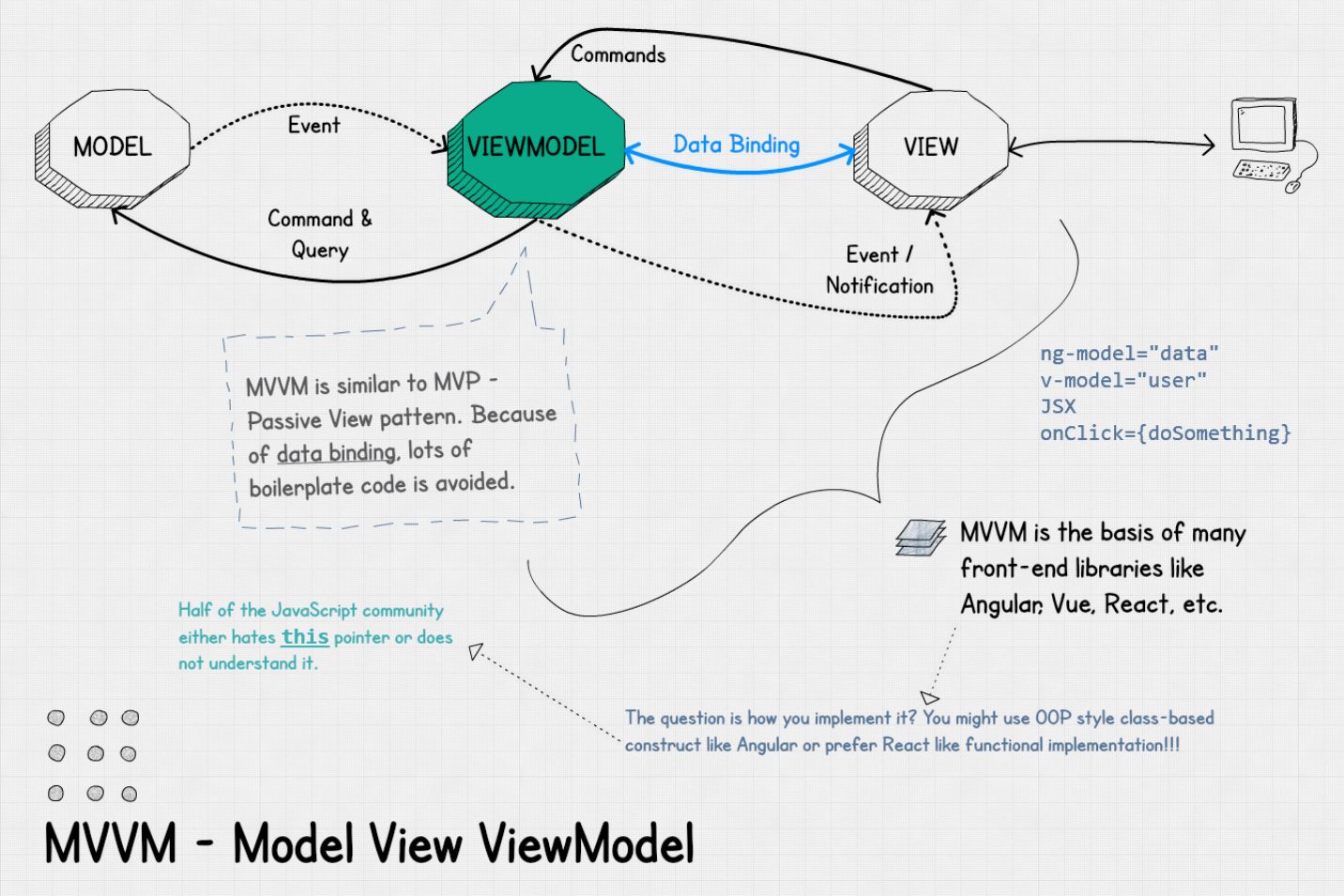


MVP — Model View Presenter

MVVM — Model View ViewModel

MVP is great and there are many possible variants and intricate details but from the front-end application perspective, MVVM really stands out. It is also known as **Model-View-Binder** in some implementations. MVVM is very similar to **Passive View MVP** but with an added twist of **data binding**. It is a programming technique that binds data sources from the provider and consumer together and synchronizes them. It gets rid of lots of boilerplate code that we traditionally need to keep **View** and **Model** in sync. This allows to work on much higher level of abstraction. In terms of collaborations:

1. **ViewModel** is an object that exposes bind-able properties and methods which are consumed by the **View**.
2. MVVM has additional **Binder** entity that is responsible for keeping **View** in sync with the **ViewModel**. Every time a property on **ViewModel** changes, **View** is automatically updated to reflect the changes on UI.



MVVM — Model View ViewModel

> Data Binding, inherent in MVVM, has been the basis of many Front-end libraries including

> Knockout, Angular, Vue.js, React and others.

We will revisit data binding again in web application section.

. . .

Into the Realm of Web Applications

Like the original **MVC** pattern, a pattern has emerged for use with Web applications. It is known as **Web MVC**. In fact, due the way web applications are built and deployed, MVC is more natural evolution compared to that of desktop application.

> *The major confusion in the community is because of the unawareness that **desktop MVC** and **web MVC** are two different patterns. Only if web MVC would have been named something else, things would have been much clearer.*

Web application is a sub category of distributed application. Though MVC feels more natural with web applications, in general, building distributed applications is hard. Some part of the code run on server while remaining on the client i.e. browser.

> The struggle of large scale web application architecture is determining what part of the code should execute where. Either we have server-driven apps or rich client-driven applications. Between the two, we can mix-match in endless ways.

While talking about web application in the context of MVC, there are three distinct data cycles and thus three MVC implementations: servers-side MVC, browser's internal MVC, and front-end MVC. The browser mediates between three types of interactions:

1. Between client-side (JS + HTML) code and server-side code.
2. Between user and server-side code.
3. Between user and client-side code.

Browser has its own **Model**, **View** and **Controller**. As a developer, we need not worry about browser MVC.

Server-side MVC a.k.a. Model 2

First well-known implementation of Server-side MVC is the **Model 2** by Sun Microsystems for Java web applications.

Server side MVC — A front-end side perspective

This MVC is very similar to the classical MVC but there are additional complications due to fact that data flow cycle time has shoot up exponentially when data crosses the client and server boundaries. Some things to note are:

- Desktop MVC has two **Data cycles** while Web MVC has three **Data cycles**.
- There are two **View cycles**. First is a **Client View Cycle** such as scroll-event, keyboard inputs, etc. Second is **Server View Cycle** such as page refresh, hyperlink activation, etc.

- Finally, we have a **Model Cycle** which has additional time complexity as it passes the client-server boundary.
- **Front Controller**: A component typically provided by the underlying technology stack to handle HTTP request dispatching. For example, `Servlet` container in Java web applications, `IHttpHandler` in ASP.NET or `HTTP.Server` class in Node.js.

Today, **SSR — Server Side Rendering** means altogether a different concept. However, that is not true. Since entire HTML/content is produced by server and no client-side JavaScript code is involved, web applications fully built with Server-side MVC are still considered as SSR.

Going beyond the Server-side

This is where it really gets interesting. Almost every browser started shipping JavaScript engines. In my opinion, it was the AJAX that changed the course of web applications. Google was the first to master it with its mail client and maps applications.

This is the world of server-side MVC generating HTML + JavaScript. JS code is sprinkled across the pages. JavaScript is mostly used to improve UX by reducing **Server View Cycles**. Things like form submission, input validation, etc. is handled by client-side code.

What happens when client gets its own Model?

It is the most prevalent architecture in the history of web applications. Most of the B2C applications, SEO friendly websites especially the one built with CMS — Content Management Systems are all using it. The amount of client-side code depends on the needs of the application.

This architecture as such was never really standardized and thus does not have a name. It has evolved in an incremental style and still considered an extension to **Web MVC**. ASP.NET MVC, Java Struts, Python Django, Ruby ROR, PHP CodeIgniter are some of the widely used frameworks that make extensive use of **Server-side MVC** or **Web MVC**.

Of course, there are many more variations to this standard pattern, but, they do not have any real influence on contemporary front-end architectures and can be omitted.

Essential RIA — Rich Internet Application Architecture

With all this background, we are now ready to discuss contemporary front-end architectures. Contemporary front-end architectures resolve around building **RIA — Rich Internet Application**. Exact definition of RIA is impossible as it means different things. But in general, **RIA** or **Rich Web Applications** are the category of applications where an application depends heavily on client-side code and their UX is very close to that of a desktop application. It is mostly built using frameworks that supports SPA — Single Page Application. The data flow of **Server View Cycle** from the Web MVC does not exists. There is usually just one initial HTML page and then client-side code and routing solutions are used to render subsequent pages.

Building RIA is a complex operation and has evolved from the learning of previous desktop-based GUI architectures. ViewModel, Observers, Components, etc. are some of the things that are borrowed from these architectures. **Oliver steel**, in his 15 years old blog post (believe me, it is one of the best article), has provided nice reference architecture for understanding RIA data flows.

Client side becomes heavy — The age of SPA

The most notable difference between RIA reference architecture and Web MVC is the absence of **Controller** and **View** from the top-level architecture. However, they are not gone in a literal sense. If we look under the surface, Controller and View are still present but thinned having their roles considerably slimmed down. Back-end is mostly API driven. **View** is confined to producing JSON and **Controller** is responsible for orchestration of the incoming requests and mapping them to appropriate business workflow.

GUI Patterns are hard?

If you have explored preceding patterns in depth, then you will realize that GUI patterns are different than other software engineering patterns. Take **Elements of Reusable Object-Oriented Software** for example. Most of the patterns are independent of technology or language. However, same is not true for GUI patterns. These patterns are applicable at the boundary of Human Computer Interaction. **User** and **Side Effect** are inherently part of the patterns. *(Note: Elements of Reusable Object-Oriented Software has described classical MVC in great depths)*

> GUI patterns are applicable at the boundary of HCI — Human Computer Interaction. User and Side

Effect are inherently part of the pattern.

Thus, it is almost impossible to discuss them in theory without considering underlying framework or language. So far, with reasonably higher level of abstraction, we were able to explore these patterns. But as we approach the crux of the article, we will rely on different libraries or frameworks to describe these patterns.

Most of the Web UI patterns can be classified into three phases, viz, Evolutionary, Revolutionary and Contemporary.

A Front-end Spectrum — Eagle's eye view

Evolutionary patterns are simply an extension of server-side MVC. They really do not try to break the wheel or invent something new altogether. They supplement existing applications by improving their UX one step at a time. Whereas, revolutionary patterns are those ideas that divorces front-end application development from server driven workflows. Their arrival marks the prominence of SPA applications. Contemporary patterns are like a second revision of these revolutionary patterns and a general trend where front-end community is heading.

. . .

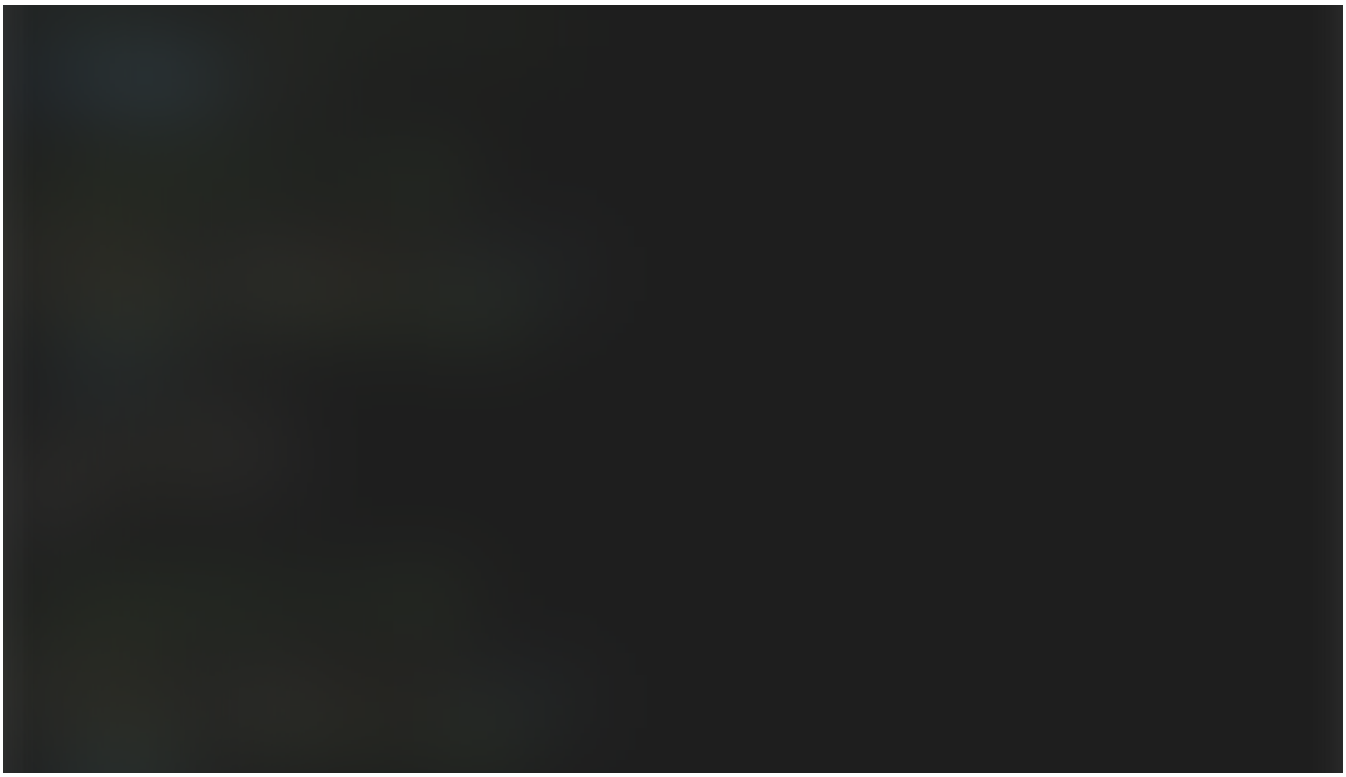
DOM-infused Algorithms

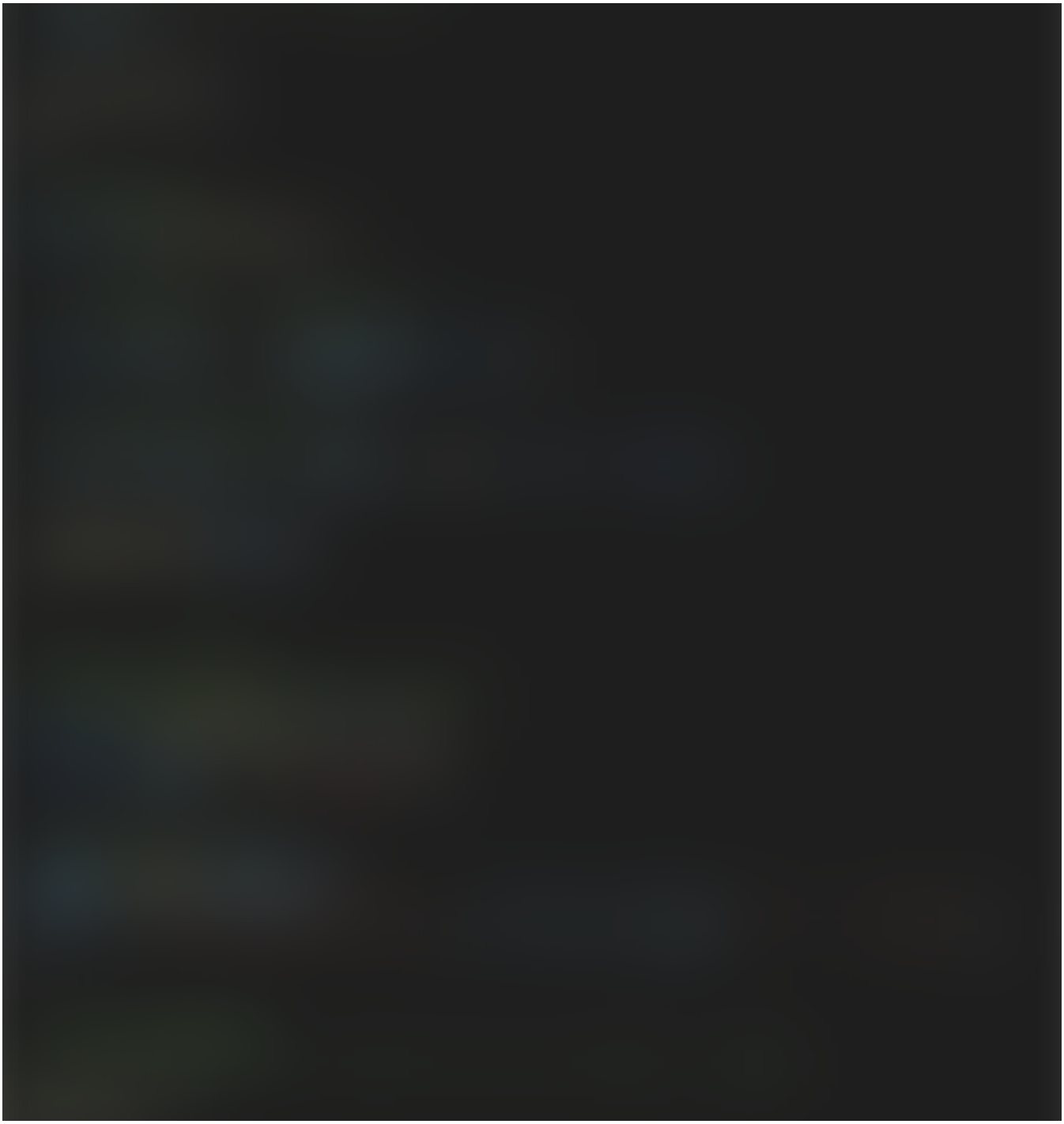
A technique introduced and mastered by jQuery was truly the beginning of writing large scale client-heavy applications even though jQuery was not really addressing architectural concerns. It was designed to simplify DOM manipulation when browsers had too many inconsistencies. It provided browser-agnostic API.

I do not think if it was intentional but jQuery simplified DOM APIs to such extent that it was hard to distinguish from the normal language APIs. This in turn allowed developers to literally mix DOM API (**View Layer**) with the business logic (**Model**).

One thing to point again is that it is still within the context of the same server-side MVC. It is just an extension. There is no real **Inversion of Control**. Overall control of the views/pages is still driven by the server-side code.

jQuery — HTML Code



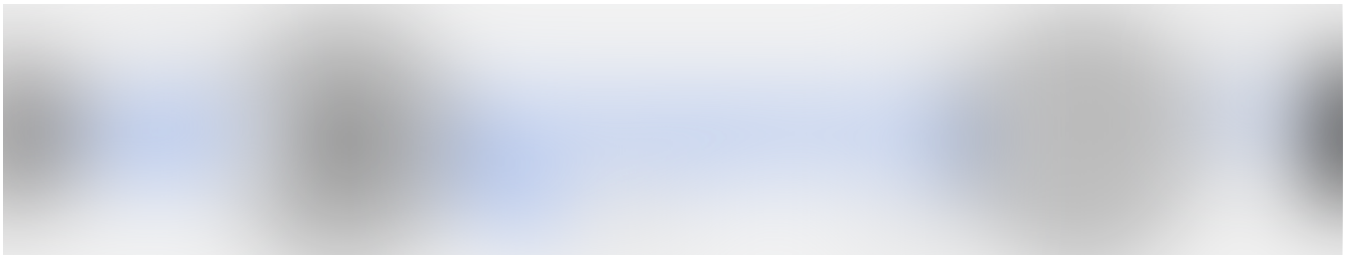


jQuery — DOM infused algorithm

In above code snippet, **Model**, **View** and **Presenter/Controller** are sort of mashed into one monolith code structure. This is the case when **Model** consists of only one property. Imagine, trying to build a web application without **Server View Cycle** (i.e. SPA). It would be impossible to handle it in any meaningful way. The code to interact with DOM is succinctly infused with rest of the application logic and that is why it is known as DOM-infused algorithm (Note: I am not sure if **DOM-infused** terminology is part of any nomenclature.)

Backbone.js — MV*

As we saw with jQuery, when developing applications, the way to structure and organize our code is clearly missing. That is exactly where **Backbone.js** comes into the picture as a next **Evolutionary Solution**. It was one of the first libraries to bring the benefits of MVC style to the client-side.



Backbone.js data flow

If we look at the **Backbone** data flow diagram, then we can clearly see **Model** and **View** but there is no **Controller** equivalent object. Patterns evolve and client-side MVC is merely an evolution of prior MVC architectures. During this evolution, most of the **JavaScript** community agreed on the definition of **Model** and **View** but there was no convergent opinion about **Controller**. Considering the client-side environment, the idea of **Controller** does not fit well. **Controller** is left open to interpretation.

In regard to Backbone, **Controller** is absent; so, where does it fit? Is it MVC, MVP or MVVM? Borrowing from the definition of server-side MVC, Controller has two responsibilities viz. Respond to user actions in the form of incoming HTTP requests and orchestrate **Model** to generate **View** (HTML page). In case of Backbone, these responsibilities are shared by **View** and **Router**. But an independent notion of a **Controller** or **Presenter** is absent.

> *Some developers feel Backbone is MVP where View is analogous to Presenter, **Template** is View whereas Backbone **Model** and **Collection** constitute **Model**.*
>
> *As Addy Osmani puts in his blog, it's ultimately best not to force Backbone to fit any specific design patterns. Design patterns should be considered flexible guides to how applications may be structured and, in this respect, Backbone fits neither MVC nor MVP. Instead, it borrows some of the best concepts from multiple architectural patterns and creates a flexible framework that just works well.*

That is how **MV*** or **Model-View-Whatever** is born. For elaborate discussion, a blog post by Addy Osmani is highly recommended:

Understanding MVC And MVP (For JavaScript And Backbone

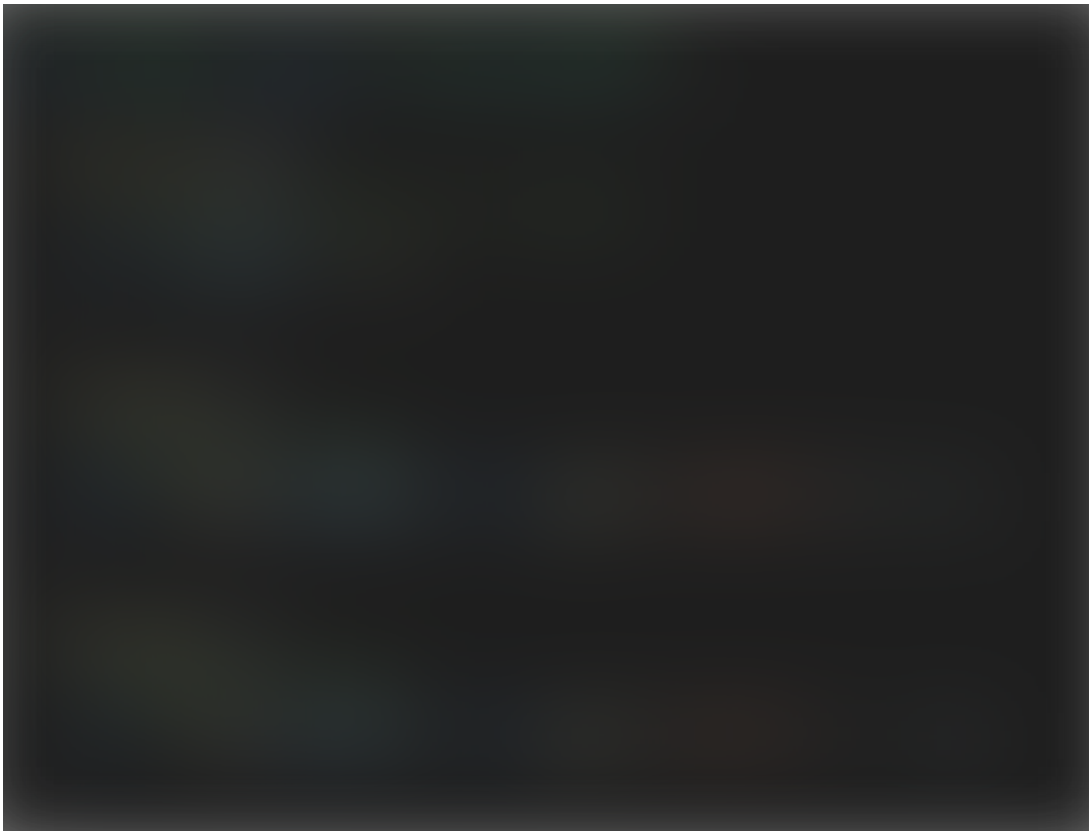
Understanding MVC And MVP (For JavaScript And Backbone Developers)

Before exploring any JavaScript frameworks that assist in structuring applications, it can be useful to gain a basic...

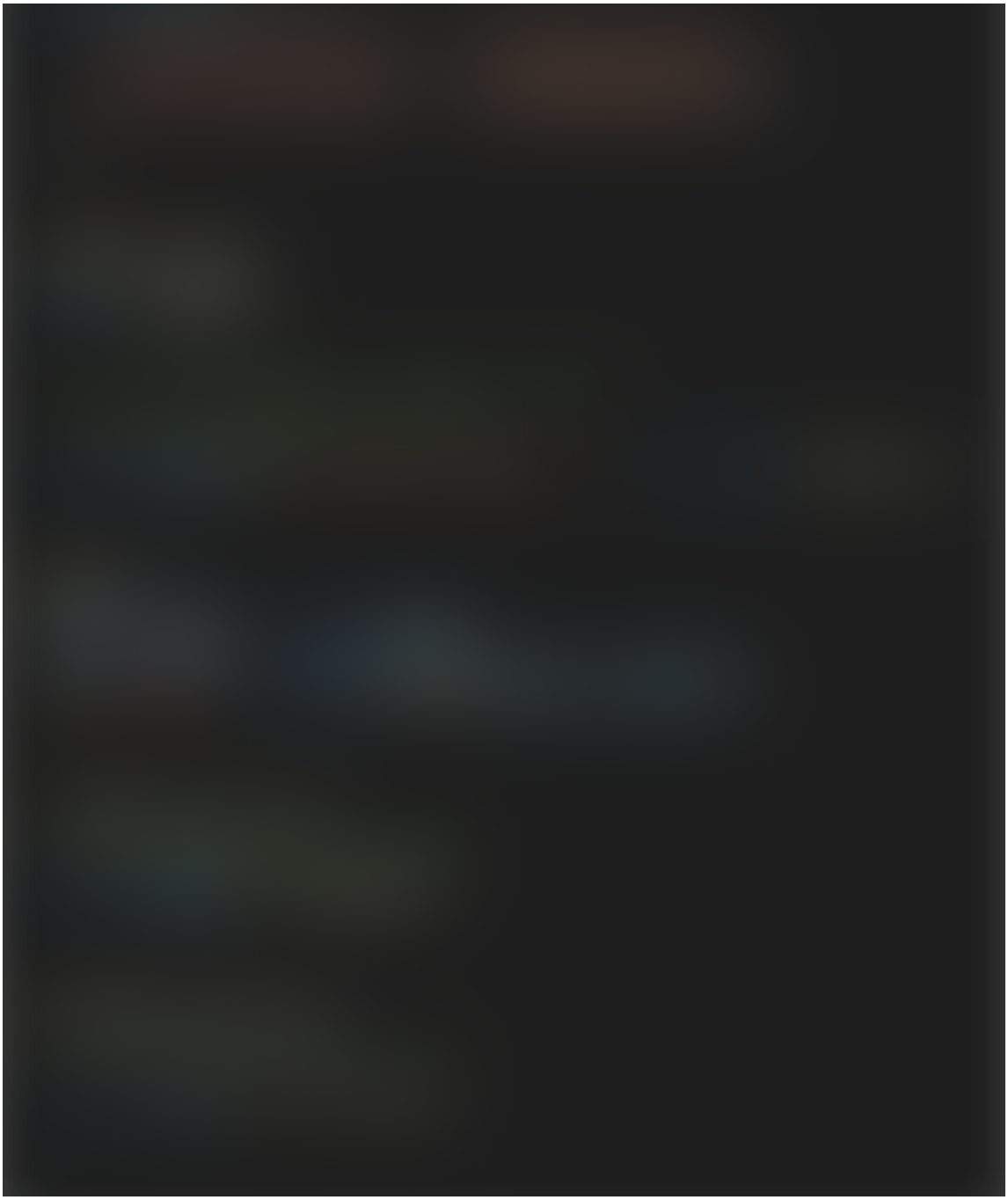
addyosmani.com

Compared to previous jQuery, Backbone helps produce more structured code.

View Template in Backbone.js



Creating a Model in Backbone.js



Creating an instance of View in Backbone.js

Earlier in the article, I called Backbone as next **Evolutionary Solution**. The reason behind is that it simply extended server-side MVC by complimenting it. For example, if our back-end is RESTful and it is implied that front-end code is simply a mean to represent server-side Model, then Backbone is pre-configured to sync with the API:

Automatically generated collection methods in Backbone.js

And, then there are many other small conventions baked into Backbone which simply feel like as an extension. On a closing note, Backbone may not have been the only solution at that time, but it was truly pioneering work in terms of code structure and organization. Like jQuery, it was adopted by many products.

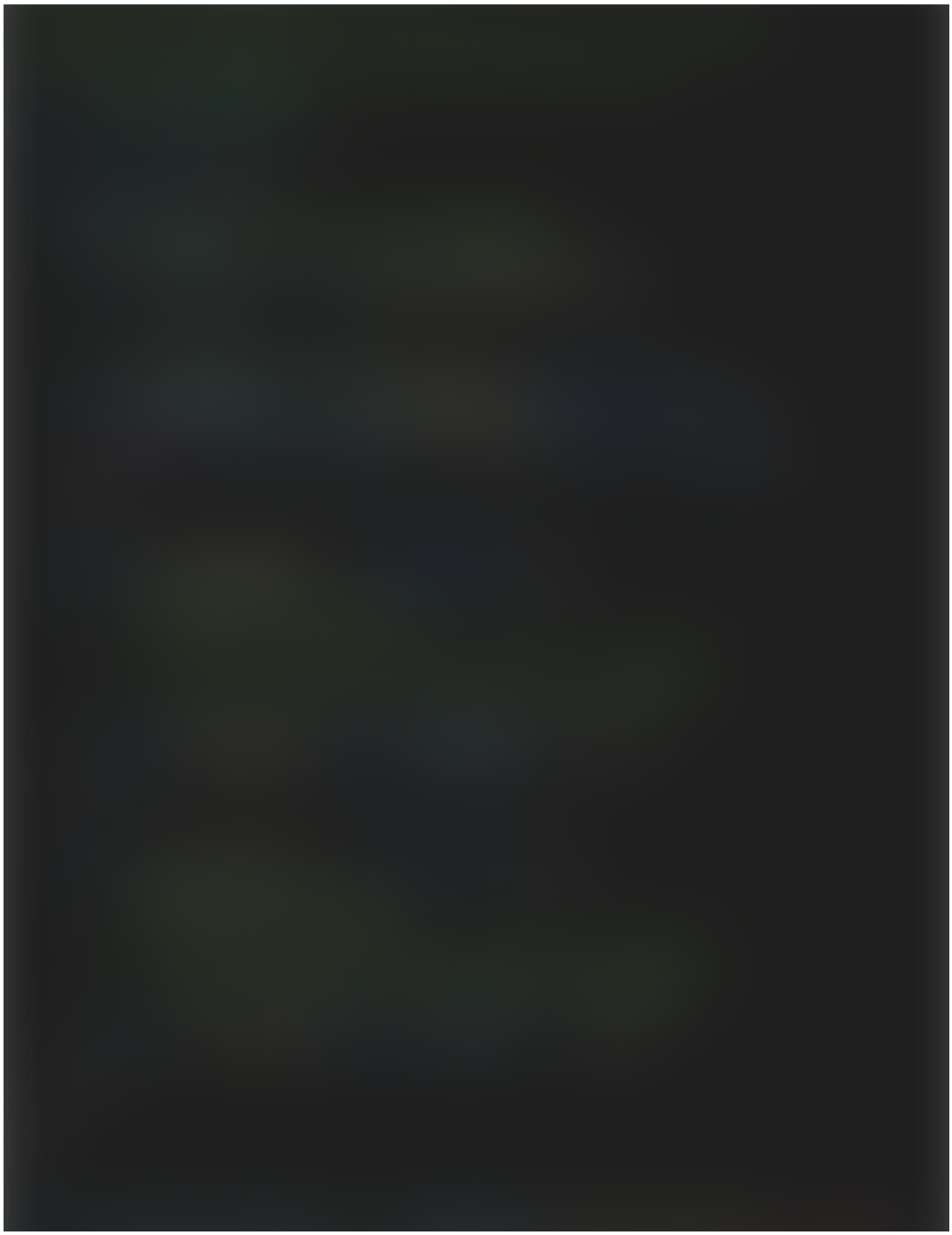
Knockout.js — Data Bindings for Front-end

Knockout.js is our final case study in basic patterns. It aims to implement **MVVM — Model-View-ViewModel** for JavaScript. It stays true to its words. While Backbone was about solving the problem of code organization and structure, Knockout is about implementing **View** layer efficiently with the help of **Declarative Data Bindings**. The advantages of declarative bindings are the same as that of any declarative programming constructs:

1. Easy to read: Declarative code aids in programming
2. Reducing boilerplate: Bindings allow us to automatically update DOM whenever **ViewModel** changes, as well as update the **ViewModel** whenever the **View** changes via user inputs.

3. Observable: Knockout provides higher level abstraction on top of events. This allows Knockout to automatically tracks dependencies between **ViewModel** props. If required, we can subscribe to Observable properties.

Knockout.js View — Two-way binding



ViewModel in Knockout.js — Reactivity using Computed properties

While Knockout provides well defined constructs for **View** and **ViewModel**, it is mute about how application's **Model** should be. This makes Knockout sharp focused and versatile as it can be used as a library instead of a framework. In my experience, I have seen it used to build mini SPA applications where web application has multiple pages and each page is a small Knockout app. This StackOverflow answer clearly defines the scope of Knockout's MVVM implementation.

> Often, with Knockout Model is assumed to be residing on server-side. ViewModel simply asks server-side Model using Ajax or equivalent.

It replaces jQuery and template solution like Handlebars for DOM updates but still uses jQuery for animations, Ajax and other utilities. Combined with Backbone, it can serve as a complete implementation of MVVM pattern. In theory this could have happened, but before that could happen, these concepts were already being taken forward in cohesive ways in next generation tooling.

> *This is where the **Revolutionary Journey** of Angular 1, Aurelia, Ember.js, etc. begins.*

Due to its close association with .NET world, it was widely used in ASP.NET MVC application. Like Backbone, it was another **Evolutionary Solution** to a slightly different problem. And again, the assumption, that client-side code was simply an extension for server-side MVC pattern, was intact. Server-side was still the dominant architecture. During those days, Backbone and Knockout were used be compared but in reality, they were complimentary solutions.

Both **Knockout** and **Backbone** are JavaScript libraries. Somehow, Backbone was viewed as a framework. Why? No certain answer but it probably has to be the

perspective. Backbone was always treated with higher level abstraction due to its emphasis on code structure. Further, Backbone was never meant replace ubiquitous jQuery *(even in 2019, 70% of the top 1 Million websites use jQuery)* whereas Knockout was overlapping with core jQuery i.e. DOM manipulation and that naturally made it difficult for Knockout. Thus, Knockout's adaptation was limited as compared to Backbone. But it was still one of the first implementation of **Declarative Data Bindings** for the front-end community and deserves a special mention.

. . .

Angular 1 — Let Me Have the Control

What jQuery did to DOM, **Angular 1** did to the front-end ecosystem in general. It forever changed the notion of building large-scale client-side applications. As a framework it introduced great many concepts — Module System, Dependency Injection, Inversion of Control, Simpler data binding, etc.

It was and is still a pain to pick right JavaScript libraries and build a perfect technology stack for front-end. **Angular 1** simply provided a simpler yet cohesive alternative. Same can be said true for **Ember.js** and other similar framework but adaptation of Angular 1 was way in a different league that the alternatives of that time.

> *Angular 1 is **Revolutionary Solution** in a sense that it clearly marked the departure from the idea of simply extending server-side MVC with client-side code sprinkled across pages. Angular 1 made SPA a first-class, almost de-facto solution for building next generation user experiences.*

A Framework or a Library?

Prior solutions were more libraries than framework. Angular 1 is, without doubt, a well-defined framework. A necessary distinguishing factor between a framework and library is **IOC — Inversion of Control**. Further to qualify as a framework, Angular 1 exhibit:

1. Well-defined MVVM: Angular 1 has clear **Model**, **View** and **ViewModel** objects.
2. Dependency injection (DI): Angular 1 has first-class support for DI which provided managed lifecycle for **Model** objects. In Angular 1, application's **Model** is

implemented using **Service**. Service is a **singleton** entity by default which is often desirable for **Model** objects.

3. Data binding & change detection: Data bindings are declarative and change detection is automatic. Data binding the necessary component to qualify it as MVVM. Bindings were bidirectional. Further change detection was almost automatic (Angular 1 was truly magical). It reduced lots of boilerplate code for developers. Use of events and overall subscription mechanism generally reduced. This is very prevalent in MVC based systems. Events also reduce code readability since data flow for certain workflow is spread across the code.

4. Module system: Angular 1 introduced module system specific to the framework. Modules are the basis of code organization for virtually every language. JavaScript did not have module system till 2015 *(Browsers did not support it till 2018)*. Angular was way ahead of its time in terms of organization.

At the same time, Angular 1 was also criticized for the complexity it introduced. **Most important criticism is that it was modeled after server-side constructs.** It is not idiomatic for front-end developers. Few things were simply bad:

1. Namespace collision: Though DI was great, but it was implemented using **Service Locator** pattern which used global name space. That made it almost mandatory to prefix services.

2. Bidirectional data bindings: It might have been a great idea at that time, but over the period, it is realized that it doesn't really scale well. Especially the rise of React made two-way data binding optional. Bi-directional bindings can create lots of spaghetti code if not careful.

3. Use of confusing terminology: Some of the terminology used by Angular 1 was clearly confusing. For example, Angular 1 has `$scope` which acts as a **ViewModel**, but it also had **Controller** which would augment `$scope` object. Probably right name could have been **VMFactory** or equivalent. Also, Angular 1 has three types of **Service** recipes, out of which one is named **Service**.

There were many other small problems. Additionally, **Angular 2** or simply **Angular** was a complete breaking change to an extent that it is like a completely new framework. I do not find anything common other than name and few concepts.

Angular.js — The big picture

Over the years Angular 1 had minor releases which fixed many little intricacies of using Angular 1. Most significant being the addition of **Component Model** which is where most of the front-end world has converged on.

Angular 1 had a long-lasting legacy on the front-end community. With all its pro and cons, it helped community understand the importance of software architecture and provided a baseline for writing scalable applications. Its cons/disadvantages became the basis of problem solving for future architectures.

. . .

Contemporary Front-end Architectures

Modern day applications are very close to the way desktop applications are. Some of the notable changes in today's environment are:

1. Web is no longer confined to desktop machines. We have host of gadgets like tablets, mobiles, smart watches, glasses, etc. In fact, mobile phones drive more web traffic than computers.

2. JavaScript, the language powering the Web, has evolved into a full-blown language and continues to evolve with many new features.
3. New API like File System, Camera, PWA, Hardware sensors, etc. are available for web applications.
4. User Experience is the deciding factor for users choosing between the competing services.
5. Network infrastructure has improved but at a same time, next billion under-privileged users are joining Web. Streaming, video-on-demand is a daily affair.
6. Same JS technology stack is being used to build native mobile applications using various tools and cross-compilers.
7. Many new frameworks and tools use JavaScript as a target language instead of using it as a source language. Few good examples are Elm, PureScript, ReasonML, etc.

Contemporary architectures are the reflection of these changing needs. As the name suggests, they are built on the experiences learned from the past by the entire front-end community.

> Earlier Software Architectural Patterns modeled and respected limited hardware capability. Today, that is not the case. Computing capacity is ever increasing. And, software architecture reflects that perspective.

Above hypotheses can be linked to how today's front-end architectures are. These architectures have converged on three core principles.

1. Data flow takes the center stage

Considering the large scope and the varied environment in which front-end code needs to run, business logic takes a center stage. As engineers, we spend more time reading and debugging than writing the code, it must be intuitive for us to trace code.

Whenever fixing a bug or adding a new feature to the existing code base, it is vital to understand its impact and regression that can possibly result.

> *The only way to do that in a large code base is to ensure that **we understand how data is flowing in the application**. This is the most important goal of the software architecture.*

Today, any framework is built around this critical aspect. The emphasis on clearly segregating **State** and **View** is very high. This is evident from the shift from simple event-based to more sophisticated state containers like **Redux**, **Vuex**, **Ngrx**, etc. Due to heavy reliance on **Events** or **Pub/Sub** systems, the data (or the control flow) would flow across the application in bits and pieces. The emphasis on data flow is no longer localized. Instead, being the central idea, data flow is now considered across entire application.

Angular 1 already showed that two-way data flow, even localized to a view or component, can create untangled mess of control flow. Also, **React has proved that uni-directional data flow is easy to reason about** and as such, modern frameworks including Angular 2 have followed the suit.

2. Component-based Architecture

Shift to component-based user interfaces is the corollary to the first principle of data flow. There are three aspects to consider when we talk about components.

First is the Data Flow emphasis: Traditional architectures focused on splitting responsibilities **horizontally**. However, data-flow based thinking demands that the responsibility be split **vertically**. In agile world, that is how **User Stories** are envisioned. MVC based architecture cannot do this easily. There is no easy way to share or reuse a UI code by a feature (Question here is — how we can really slice a feature in isolation when a feature is spread throughout the code due to horizontal organization!). But a component encapsulating one complete feature can be easily configured as a share-able and package-able entity.

Second is the Ever-growing View State: In past, the ratio of **View State** to **Business State** was low. But as applications get more interactive, this ratio has gone up exponentially. This **View State** needs to be close to that actual view. **View State** is complex and often represents imperative time-dependent data like animations, transitions, etc. MVC like architectures does not provide good means to encapsulate this state. Here, components as a core unit of encapsulation can help greatly.

Third aspect has to do with the finding the atomic unit of UI development. It begins with a question:

> *What is the best way to share a UI functionality? Sharing a UI functionality means it could have some or all the four pieces:*
>
> *Structure (HTML — **View**), Style (CSS — **View**), Behavior (JavaScript — **ViewModel**) and a Business Logic (**Model**).*
>
> *That is where the idea of **Component** is born. And we realize that component is simply a good implementation of **MVVM pattern** or **MV* pattern** to be precise…*

But in concrete terms, how to represent a **UI Component**? Closest one I could find is this blog post by Deric Baily. On a language level, typically module or package is used to represent a building block of reusable functionality. JavaScript too has modules. But that is not enough.

> *This whole **abstract idea of components** allowed framework authors to define concrete implementations as per their needs.*

Additionally, **good components are highly composable** and enable fractal architecture. For example, a login form component could be part of standard login page or be shown as a dialog box when session times out. A component once defined in terms of properties it accepts and events it emits could be reused anywhere as long as it meets the underlying framework requirement.

> A component could be a function, class, module or bundled code generated by a bundler like Webpack.

As said earlier, a component is simply a **well implemented MVVM pattern**. And, because of the component composability, we get MVVM pattern implemented as a fractal (A fractal is a self-repeating never-ending pattern). It means we are dealing with multiple self-contained MVVM control flows at multiple levels of abstraction.

An endless fractal of components implementing MVVM — Loop within loop! Does it sound like Inception?

And, that is exactly what we must expect from an architecture.

> A good architecture enables multiple levels of abstractions. It allows us to view one abstraction (level of details) at a time without worrying about the other levels. That is a key ingredient to make a Testable and Readable Solution.

3. Let framework take care of the DOM

DOM is expensive and tedious to work with. It is better if DOM can somehow automatically update itself whenever **State** (**Local** or **Global**) changes. Further, it should be as efficient as possible without disturbing other elements. When it comes to sync DOM with State, there are things that needs be taken care of.

1. **Representing DOM as a function of Model**: Combined with components, **declarative data bindings** are a great solution to represent View in terms of Model. Angular has templates. React uses JSX. Vue supports both JSX and templates. Combining data binding with component, we get a perfect MVVM.
2. **Change detection:** A framework needs a mechanism to identify what all data of the State has changed. Angular 1 used expensive digest cycle. React favors **immutable data structures**. With immutable data, detecting a state change is

simply an **equality check**. No need for dirty checking! Vue.js relies on its reactivity systems built on top of getters/setters. Angular uses **Zones** for change detection. Vue 3 will use ES2015 proxies for reactivity.

3. **Updating the DOM:** After actual changes are detected, a framework needs to update the DOM. Many frameworks like React, Vue, Preact, etc. use **Virtual DOM (time optimized)** while Angular uses **Incremental DOM (memory optimized)**.

. . .

Defining a Contemporary Front-end Architecture

Scope of earlier GUI architectures was focused on code structure and organization. It was about relationship between the **Model** and its **Representation** i.e. **View**. That was the need of that time.

Today, landscape has changed. Modern front-end architecture needs have gone beyond simple code organization. As a community, we already carry that tribal knowledge with us.

> Most of the modern frameworks have already standardized the notion code structure and organization with the help of Components. They are the atomic units of today's architecture.

It should also be noted that an architecture is closely associated with its implementation framework. This goes back to the fact that GUI patterns are complex and cannot be discussed independently of its implementation due to its vicinity to user. We can go on to say that a framework is its architecture.

Envisioning a good component-system is the most fundamental need for any front-end architecture or framework. Beyond that it must address many other issues as discussed earlier like declarative DOM abstraction, explicit data flows, change detection, etc. Most of these elements are highlighted in the following diagram:

Elements of Contemporary Front-end Architecture

With all those considerations in mind, we might be tempted to create a reference architecture, but **it is simply impossible**. Every framework or library has its own library peculiarities when it comes to implementation. For example, React and Vue, both favor unidirectional data flow but React prefers immutable data while Vue embraces mutability. So, it is better to refer individual framework to get a complete sense of its architecture.

Having said that, we can still attempt to create an approximation of the reference architecture as show below:

The reference architecture for contemporary age

All the traits or elements that we described so far are present in this architecture. It cleanly maps to a three-tiered architecture, though middle-ware at the 2nd tier is optional. The second tier represents a **UI Server which is less understood aspect.** UI Server is simply a server that is designed to serve modern client-heavy UI applications. It takes care of the orthogonal aspects like SSO integration, authentication & authorization, server-side rendering, session management, serving UI assets, caching. Further, it acts as a reverse proxy for making API calls to avoid CORS issues when API server is deployed elsewhere in a large-scale application. The de-facto choice here is Node.js as many frameworks have their SSR renderer written in Node and Node excels at handling large I/O requests due to its asynchronous nature. We will discuss more about UI Server in another article about web application topology.

> *The most important change in contemporary architecture is the notion of **Model**.*

Model is no longer visualized as a single black box. It is segregated into application wide global state and component wide local state. Global state is typically managed using sophisticated state containers like Redux, Mobx, Vuex, etc. Local state for each component is an union of three things — slice of global state, component's private local state (async data, animation data, UI state) and finally state passed as **props** by the parent component. We can think of local state as a better abstraction of **Model** and **ViewModel**. When we add GraphQL to this equation, state management changes. *(explained in GraphQL section)*

Data flow unidirectional as it flows top-down from parent to child component. While frameworks allow data to flow directly in reverse direction but that is discouraged. Instead, events are raised from children components. Parent component can listen or ignore them.

An incomplete picture

This reference architecture does not really capture the entire essence of contemporary architectures. Majority of the web traffic is driven by static websites and CMS's — Content Management Systems. Modern tooling has considerably changed the way we develop and deploy these applications. In case of CMS, they are becoming **Headless** by decoupling their front-end from the back-end. Rise of **Strapi**, **Contentful**, etc. is evident. At a same time, we are no longer building static website using plain HTML and CSS. **Static site builders and generators** have become extremely popular to do the same. We can now use the same front-end framework to build static websites aided by sophisticated build tools. With **React.js**, we can use **Gatsby.js** and with **Vue.js**, we have **Nuxt.js**. When we compile our code, it produces a static website which can be fully deployed to any static web server. This technique is known as **Pre-rendering** as opposed to **Server Side Rendering**.

> *This is here we have next contemporary architecture for building static websites. Idea is to take headless CMS like Strapi and use a static site builder like Gatsby to build the front-end. At a build time, when we are generating static website, we pull all the data from the CMS and generate pages.*
>
> *When, the author changes the content in headless CMS, we re-trigger the build of our static website which builds the website with new content and immediately deploy to server or CDN.*

Building static website — The contemporary way

With this new workflow, it is as good as running a full-fledged dynamic website without the drawbacks of CMS like complicated deployment, slow load times, etc... We get fast load times and improved caching thanks to the fact that static websites can be directly deployed to CDN. We also get rid of all the problems of static websites like slow update cycles, lack of re-usability, etc. Quoting the vision from the Nuxt.js website —

> *We can go further by thinking of an e-commerce web application made with* `nuxt generate` *and hosted on a CDN. Every time a product is out of stock or back in stock, we regenerate the web app. But if the user navigates through the web app in the meantime, it will be up to date thanks to the API calls made to the e-commerce API. No need to have multiple instances of a server + a cache anymore!*

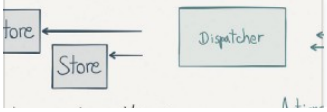
> To sum it up, contemporary front-end solutions are built on component-based unidirectional architectures.

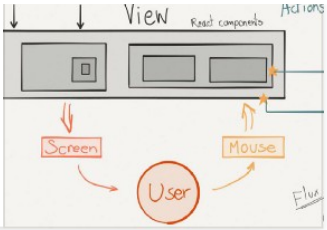
Taking unidirectional architectures further, there are many ways in which they can be implemented. Frameworks have their own ways of doing things. Few good examples to consider are:

1. Flux (Envisioned for React)
2. Redux (Mostly used with React, but view agnostic)
3. MVU — Model View Update (Used in Elm)
4. MVI — Model View Intent (Used in Cycle.js)
5. BEST, Vuex, Ngrx, etc.

These patterns are best described by Andre Staltz in his blog post:

André Staltz - Unidirectional User Interface Architectures





. . .

Contemporary or Modern???

So far, we have deliberately avoided the word **modern** and used **contemporary**. Though modern, but today's architectural practices are merely an evolution of our previous ideas. As a community, we have tried to blend into existing ecosystems by staying within the boundaries and rarely breaking the wheel. So, the word **contemporary** is an apt description to express this idea.

While completing our definition of contemporary, we must tie all the loose ends. We must link our past to the present and the upcoming future. I can think of three possible links —

1. Past — Relating today's component to historical MV*
2. Present — Scene with **Web Components**
3. Future — **Functional** components

Relating today's component to historical MV*?

It should be obvious by now but, a question might arise, that how these patterns relate to the earlier once. Isn't a **Component** already a better implementation of **MVVM** or **MV***?

As discussed previously, for a contemporary architecture, it is just one of the low-level concern. Whereas, contemporary patterns are about the reasoning of entire application as a whole. They are dealing at a much higher level of abstraction. **UI Component** is the atomic unit which takes the slice of **Global State** it receives from a parent, combines it with its **Local State** and displays the output to the user.

Unidirectional patterns are answer to a larger piece of puzzle. It is about communication between sibling components and maintaining application wide **State**.

If a **Component** enables vertical slicing of responsibilities, these patterns bring back the horizontal segregation of responsibilities for the entire application.

> *If this is still confusing, consider Vue.js for example. A **Vue Component** is a perfect implementation of MVVM while at a same time, we can use Vuex (Unidirectional State Container for Vue) to manage application-wide state. Architecture lives at multiple levels of abstraction.*

The Scene with Web Components

Component architecture being the basis of almost any framework, an attempt is being made to standardize the notion of components as an official **Web Standard**, though the reasoning behind them was entirely different. Also, I am referring to them as **an attempt** because even after being a standard, many framework authors have raised concerns about their feasibility.

In the context of this article, the most important concern is about the data flow. Tom Dale has rightly summarized this problem:

That would be nice, but in my experience, framework-agnostic components are a long way off.

Why? It all boils down to data flow. The way that data flows between components differs greatly between frameworks. How...

medium.com

As far as other problems are concerned, the blog post by Rich Harris is must:

Why I don't use web components

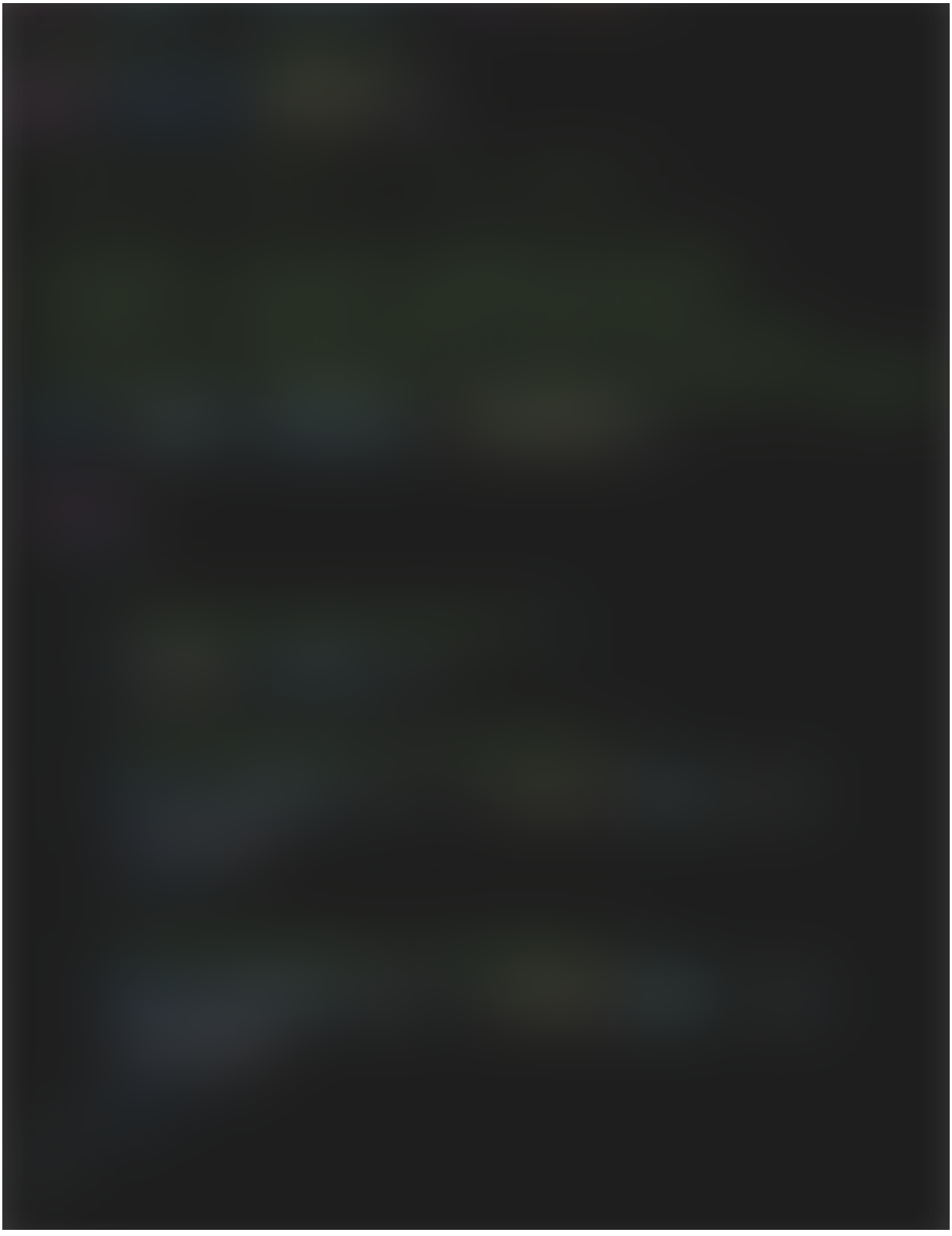
Edit description

dev.to

It doesn't mean that we should altogether avoid them when we are defining our own technology stack. The general advice is to start small with leaf components like button, checkbox, radio, etc. Always exercise caution.

Functional components, hooks, WTH?

When we refer to a Component as an implementation of **MVVM**, we generally expect a **ViewModel** object whose props and methods are used by **View** via bindings. In case of React, Vue, Angular, etc., it is generally the **Class instance**. But these frameworks also have a concept of **Functional Component** (component without any local state) where the instance simply does not exist. Also, React has now recently introduced a new way of writing components using Hooks which allows us to write **Stateful** component without class syntax.



The question here is — Where is the **ViewModel** object? In terms of ingenuity, hooks are simple but radically different idea of maintaining a local state across invocations. However, from architectural perspective, it is still the same idea. We can consider it as a simple syntax level change. We all agree that Classes in JavaScript language are terrible. They are often confusing. It puts extra onus on the developers to write clean code. Hooks solve that problem by getting rid of classes.

The only thing that changes is the notion of **ViewModel**. Be it a **Stateful Component with Hooks** or **Functional Component**, we can assume that **ViewModel** object for a component is its **Lexical Context** or **Closure**. Any variable, hook value or props received by that component together makes for its **ViewModel**. Same idea translates into other frameworks.

> *It looks like **Functional Components** are future. Though, I won't call **hooks** a fully-functional long-term solution (weird at best), but at a syntax level it is elegant and brings a relief to age old problem of classes. If I cannot convince you that why syntax matters, have a look at Svelte.*

. . .

Next phase of Contemporary Architectures

Every new technology that is associated with web applications will impact front-end applications one-way or another. Currently, there are three trends — GraphQL, SSR and Compilers, that we must discuss here to conclude this article.

GraphQL

GraphQL is a server-side query language. As you might have read, it replaces REST. But that is not true. When we talk about REST, it is a meta pattern. At a concept level, it embraces Resources-oriented architecture to define application **Domain Model** and at an implementation level, it uses semantics of HTTP protocol to exchange these resources so that it confers to the way Web is sharing the information.

Modern business needs are complex, and many workflows cannot be simply exposed as a resource in HTTP CRUD analogy. That is where REST feels awkward. GraphQL is designed to replace REST at messaging level of plain HTTP protocol. GraphQL provides

its own messaging envelope that is understood by GraphQL server and further allows to query server-side resources (Domain Model).

But as a side-effect of the way GraphQL clients are implemented, GraphQL has started encroaching on state container responsibilities. If we look at the essential fact that **Model** on client-side is simply a subset of **Model** on server-side specifically normalized for UI operations, then state container like Redux/Flux is simply about caching the data on client-side.

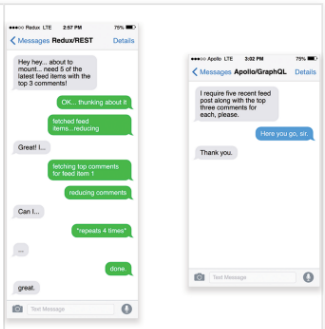
> GraphQL clients have a built-in support for caching across multiple requests.

This simple fact has allowed developers to kill lots of boilerplate code associated with **State Management**. On a very large scale how it shapes up is still to be seen. The exact mechanics of how that happens is very well documented in following posts:

How GraphQL Replaces Redux

"What?!", you say. "GraphQL is a server side query language. Redux is a client-side state management library. How could...

hackernoon.com



Reducing our Redux code with React Apollo

Switching to React Apollo eliminated a lot of complexity from our application — here's how we did it!

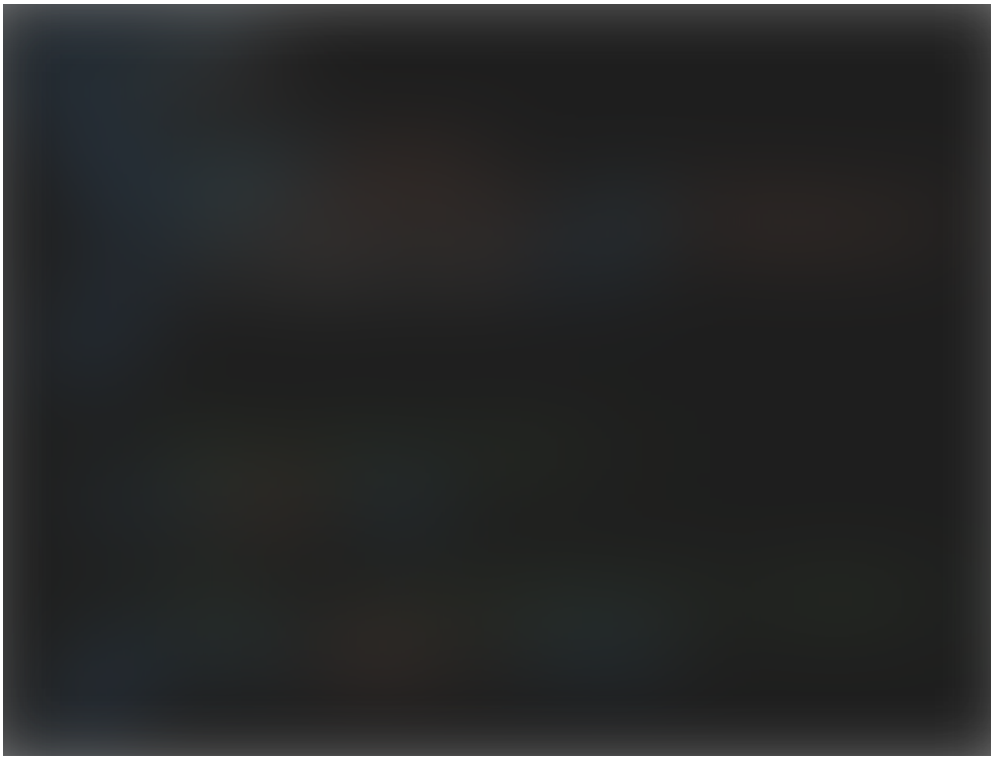
blog.apollographql.com

Be sure to explore GraphQL as it is the next big thing.

SSR — Server Side Rendering

Traditionally, server-side MVC was prominent and server would generate static or dynamic HTML page which was easily **crawl-able** by search engines. Due to the awesome user experience that is possible with client-side frameworks, we are slowly

rendering everything on browser. A typical HTML page returned by server when requesting fully client-side rendered application is almost an empty HTML page:



Initial HTML file for SPA applications — Almost empty!

Generally, this is fine but has problems when building eCommerce like websites which need fast-load times, SEO friendly crawl-able content. Adding to the woes, mobile phones have slow internet connection speed while some entry level devices have low end hardware. Search engines have limited capability to crawl fully client-rendered applications.

To mitigate this problem, old-school **SSR — Server Side Rendering** is back with a twist. With SSR, we can render a client-side SPA on the server, **hydrate the State** and then send a fully rendered page to the client. It increases the responsiveness of the website by reducing the initial load time of application pages.

> SSR is the next step in evolution as it abstracts away the Client and Server distinction.

Since, client-side code is JavaScript, we need an equivalent engine on server-side to execute JS code. **Node.js**, being the JavaScript engine, is the de-facto server-side stack for doing SSR. Although, SSR setup can get really ugly and complicated, many popular

frameworks have provided first-class tooling and higher-level framework for enabling SSR with an extremely smooth developer experience.

SSR completely changes our day to day development workflow. We are one step above the client-server abstraction. It introduces mandatory three-tiered architecture with **Node.js** based server being the key to the middleware. However, from an architectural perspective — in terms of data flow and segregation of responsibilities, all these things are same. SSR neither introduces any new data flows nor changes existing once.

Age of Compilers : Svelte — Compiler or diminishing Framework?

I do not know how to describe Svelte. What I can say is — Framework runs within browser when user launches the web application. There is a run-time cost of converting the abstractions provided by the framework. Svelte is different.

> Like static site builder, Svelte runs at build time, converting your components into highly efficient imperative code that surgically updates the DOM.

So Svelte is a component-based framework which exhibit all the characteristics of contemporary front-end frameworks but at a same, it is also a **compiler**. Compiler compiles the source into performant imperative JavaScript code. Being a compiler, it can do things other frameworks cannot do:

1. Giving accessibility warning at build times
2. Implementing code level optimizations
3. Generating Smaller bundles
4. Incorporate DSL into JavaScript without breaking syntax

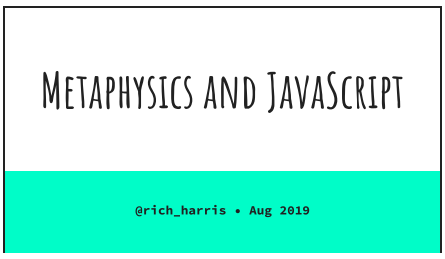
The explicit goal is to write less code. Svelte proves that compiler can achieve lot many things that were previously impossible with plain JavaScript. If Svelte is not enough, then we have Stencil.js which is a **TypeScript + JSX** compiler for writing **Web Components**.

Svelte 3: Rethinking reactivity

It's finally here After several months of being just days away, we are over the

Some of these ideas are already getting mainstream in some form — Angular AOT compiler, Vue single file components, etc. And then there are others who take this idea to the extreme:

This presentation by Rich Harris nicely puts forward the underlying philosophy of Svelte and **subjectively** compares with React:



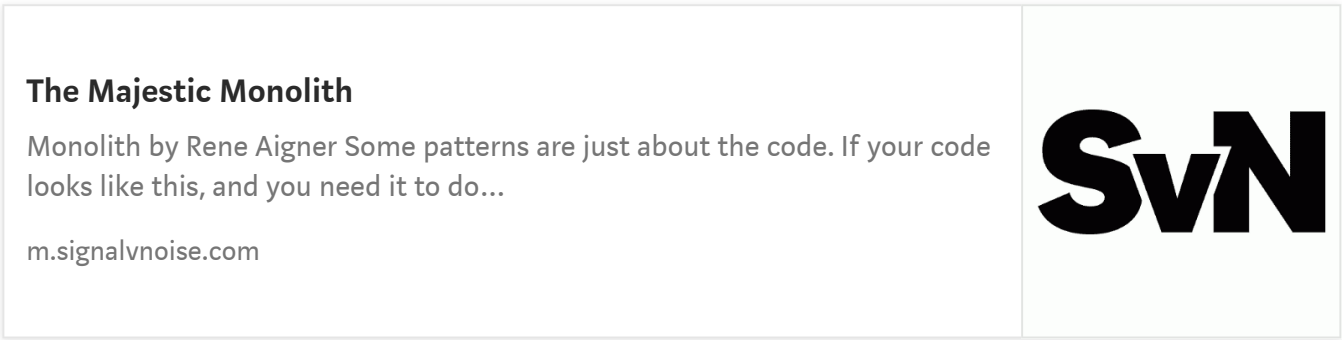
Metaphysics and JavaScript by Rich Harris on Svelte

> *As again, the future of front-end development is bright with compilers.*

There are other ways! The Majestic Monolith!!!

While full client-side frameworks are rage now-a-days, it is not the only way of doing things. The diversity of web still lives. There are still many applications that are heavily server-driven. They will continue to do so.

But does that mean they will continue to have inferior experience? Definitely not! Architecture is envisioned to support the product and Basecamp team has preciously done with their framework, Stimulus. Read more to understand their philosophy:



It is a modest framework that helps in aiding interactivity to back-end rendered pages through light JavaScript while embracing latest practices and latest standards. Stimulus is often used together with Turbolinks to create first-class SPA like user experience. *(I am a regular user of Basecamp and it find it a lot more polished than many other SPA applications.)*

Stimulus is different in a sense that it is driven via HTML instead of JavaScript. The **State** is maintained in HTML and not JavaScript objects. The data flow is pretty simple: Controller is attached to DOM and it exposes methods that can be attached to actions which can perform further actions.

As you might recall, it is very similar to the days of Backbone and Knockout. And indeed, it is. The goal is simple — An interactive front-end framework for the back-end

powered web. The only difference is adopting to modern community standards and practices.

Strudel.js is another modest framework on similar lines. In 2019, we can probably supplement using contemporary DOM library like RE:DOM.

While they may not address all the other problems expected from contemporary front-end frameworks, they bring a respite in the world where JavaScript fatigue is everyday reality.

. . .

Conclusion

There is only one word to describe GUI architectures —**Resplendent**. While MVC as a pattern is dead for front-end software development, but the principles are ageless, and they stay same.

We started with original MVC and explored famous desktop pattern. We then moved to Web application and applied same principles to arrive at popular patterns. Over the course, we moved to early independent client-side patterns before fully discussing SPA frameworks.

The important take-away is that today's front-end frameworks are component oriented and they abstract away MVC/MVVM concerns as just one of the aspect while addressing newer environments and challenges.

Finally, we glanced over newer accommodations to front-end architectures including JavaScript driven SSR and rise of GraphQL. While doing this, we have skipped many new exciting technologies like HTTP2, WebAssembly that can alter the course of front-end architectures and the way we look at the applications.

Since all these patterns have overlapping terminology and often developed independently by the community, it was hard to define a clear linear timeline in terms of evolution. At times it was problematic to relate different pattern together. Also, to keep things simple, we have taken liberty to describe certain concepts without the attention they deserve. Along the journey, due to non-availability of any nomenclature, we ended up inventing our own terms for few ideas.

Web Application Topology is another closely related aspect to the web application architecture. Topology usually has a deep impact on front-end development in terms of choosing technology stack, security constraints, performance, etc. And, thus, that will be the topic of next article.

. . .

We do hope that this article helps you understand elements of front-end architecture in a better perspective. Please consider sharing this article!!!

[WE NEED YOUR HELP]: *We are experimenting with different tools to draw illustrations and images. For this article, we have used completely new tool. Your feedback would be greatly appreciated — in terms of brevity, clarity and readability.*

. . .

References

If you have gotten this far, then following references are gold mines of GUI architectures. This article is incomplete without them.

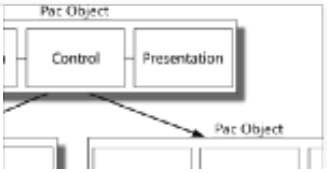
GUI Architectures

Graphical user interfaces have become a familiar part of our software landscape, both as users and as developers...

martinfowler.com

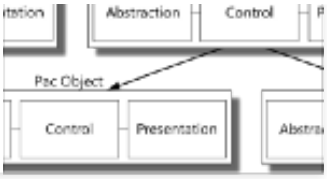
Interactive Application Architecture Patterns

The MVC, MVP, and PAC patterns are each intended to address the needs of



interactive applications by separating the...

aspiringcraftsman.com



Twisting the MVC Triad - Model View Presenter (MVP) Design Pattern

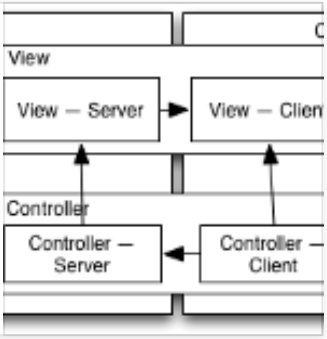
In this post we'll review the way in which MVP (Model View Presenter) design pattern evolved from Smalltalk's old...

aviadezra.blogspot.com

Web MVC

The Model-View-Controller (MVC) architecture is a standard architecture for interactive applications. In client-server...

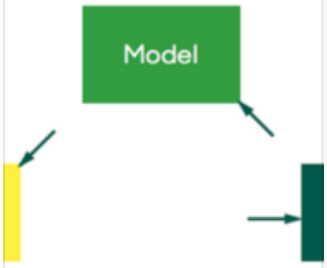
blog.osteele.com



Reactive MVC and the Virtual DOM - Futurice

Model-View-Intent is a unidirectional data flow architecture with Virtual DOM rendering for single-page web apps...

www.futurice.com



Slightly off the topic, but if you are into stream-based programming (Observable, Rx.js, Cycle.js, etc.), this classical lecture is highly recommended. There is no better in-depth discussion than this.

Streams — Harold Abelson, Structure and Interpretation of Computer Programs

. . .

Credits:

Cover image illustrations — Vintage vector created by Macrovector from www.freepik.com

JavaScript | Front End Development | Web Development | Software Development | Software Architecture