SENG 365 Week 7

Single Page Applications and JavaScript Frameworks

Outline

- Background on Single Page Applications
- Design patterns
- Document Object Model (DOM)
- Introduction to JavaScript frameworks

Timeline of JavaScript Frameworks

- **1995 -** JavaScript introduced
  - Poor browser compatibility for several years
- **2004 -** standardized AJAX (Google Gmail)
  - Beginning of single-page applications (SPAs)
- **2006 -** jQuery
  - Write JavaScript code without worrying about the browser version
  - AJAX support
- **2010s -** MVC* Data-view binding frameworks introduced
  - Client-side rendering
- **2010** - AngularJS, Ember.js, Backbone
- **2013** - React (*2015 - Redux*)
- **2014** - Vue.js
- **2016 -** Svelte (compilation to JS)

The move to Single Page Applications

- Users want **responsiveness & interactivity**; improved user experience
  - Compare user experience with native apps & stand-alone apps

- **Managing the interactions** with a user is ***much*** **more complex** than managing communication with server
  - Think of all those events (e.g. `onClicks`) to handle

Single Page Applications (SPAs)

“Single page apps are distinguished by their ability to redraw any part of the UI without requiring a server round trip to retrieve HTML. This is achieved by separating the data from the presentation of data by having a model layer that handles data and a view layer that reads from the models.”

http://singlepageappbook.com/goal.html

Some features of Single Page Applications

Separate:

- **Data**
- '**Content**' & Presentation: HTML & CSS (and other resources)

Reduce communication with the server/s by:

- **Occasional download** of resources e.g. HTML, CSS and JavaScript
- Asynchronous 'background' fetching of data: **AJAX / XHR or web sockets**
- **Fetch data only** e.g. JSON
- Fetch data from different servers: **CORS**

'Page' navigation

- **Client-side** JavaScript handles the **routing** instead of the browser itself
  - Managing **page history**
- Routing within the SPA

Reconceive web application

Re-balance workload across:

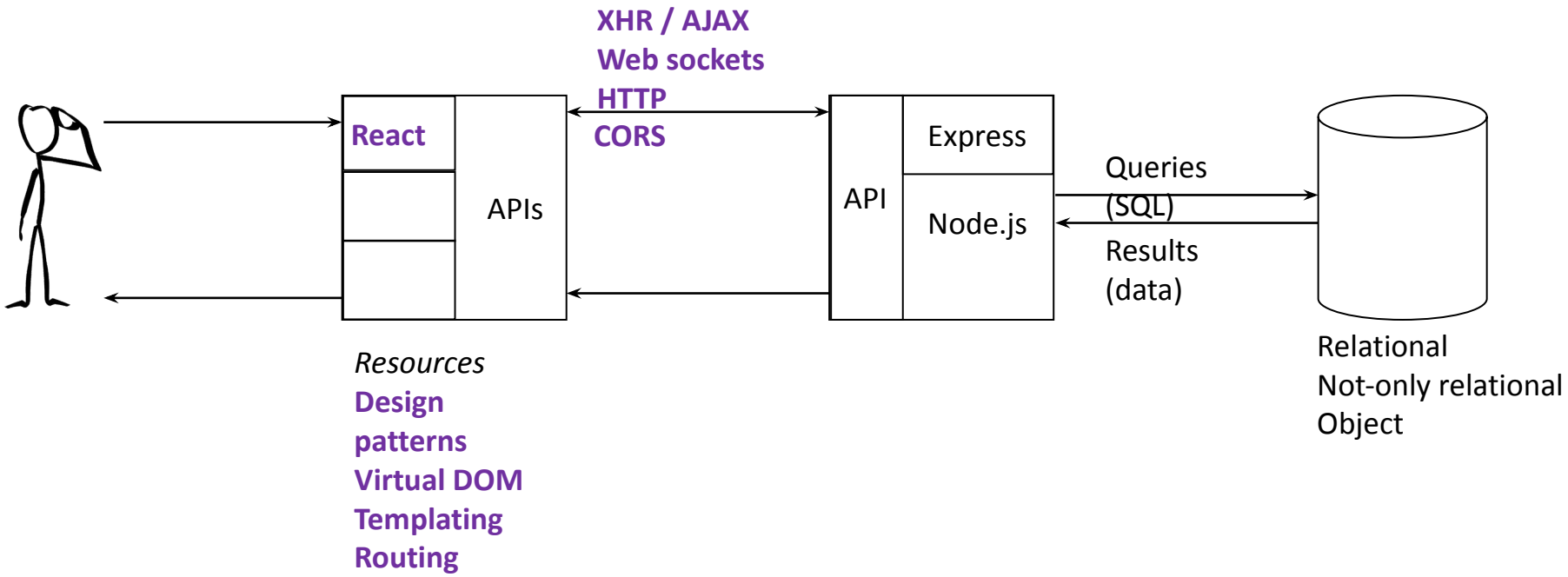
- **Server-side** application
  - e.g. Node.js
- **Client-side** application/
  - Front-end libraries and frameworks e.g. React

  Communication between client & server
  - API-driven/specified
  - e.g. AJAX & JSON

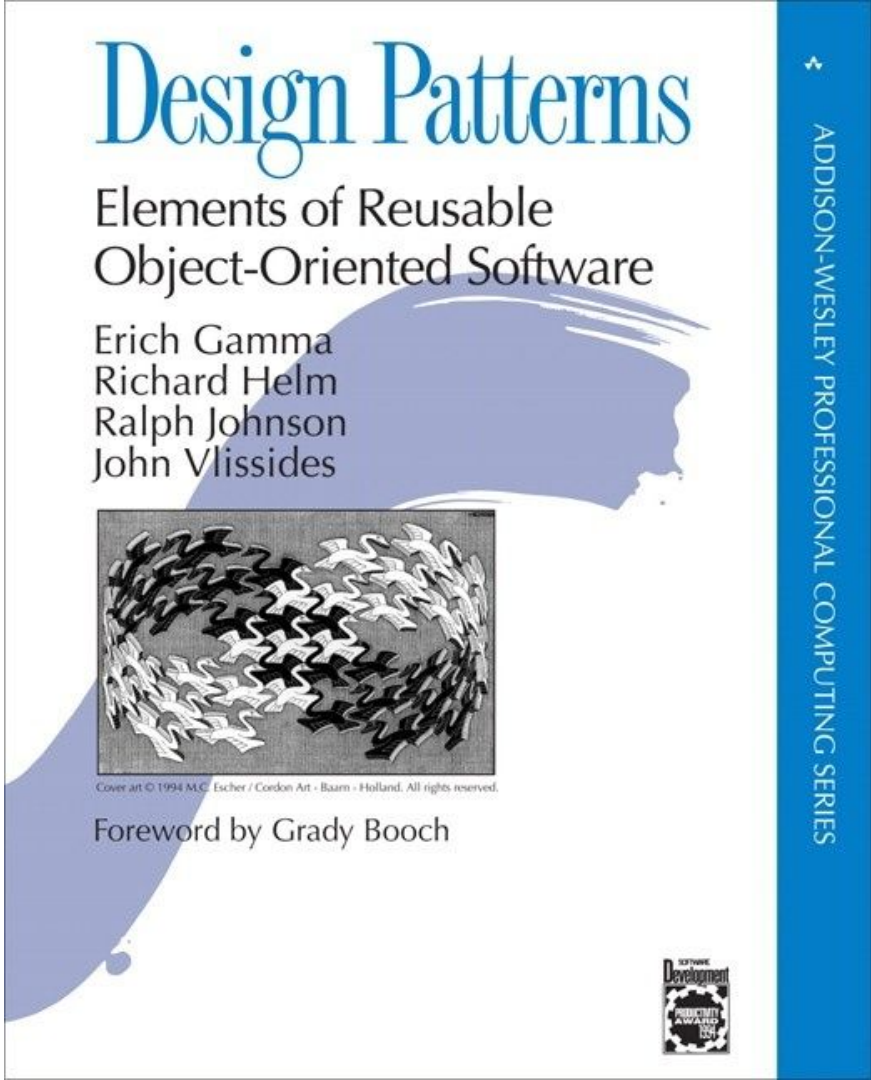
Considerations:

- Manage application **assets / resources**
  - e.g. dependency management
- Application **design and implementation**
  - Modularisation
  - Design patterns
- **Templating**

The balance of work between client and server changes. And there are new technologies



Design patterns

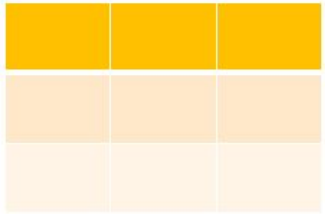


Background: *Separation of concerns*

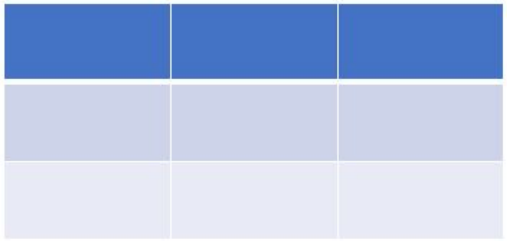
- A **design principle** for separating software code into distinct ‘sections’, such that each section addresses a separate concern.
- A **strategy** for handling complexity
  - Of the problem
  - Of the solution e.g. software code
- **Examples** of separation of concerns:
  - Object-oriented programming
    - Classes, objects, methods
  - Web computing
    - HTML: structure/organisation of content
    - CSS: presentation
    - JavaScript: functionality

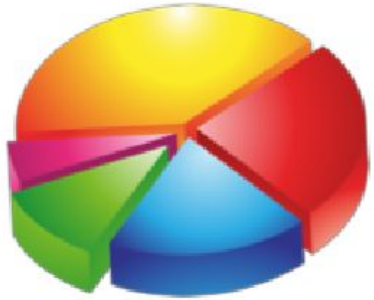
Example: data and views of data

Visualisation of data



Data (models) e.g. array, object, NoSQL

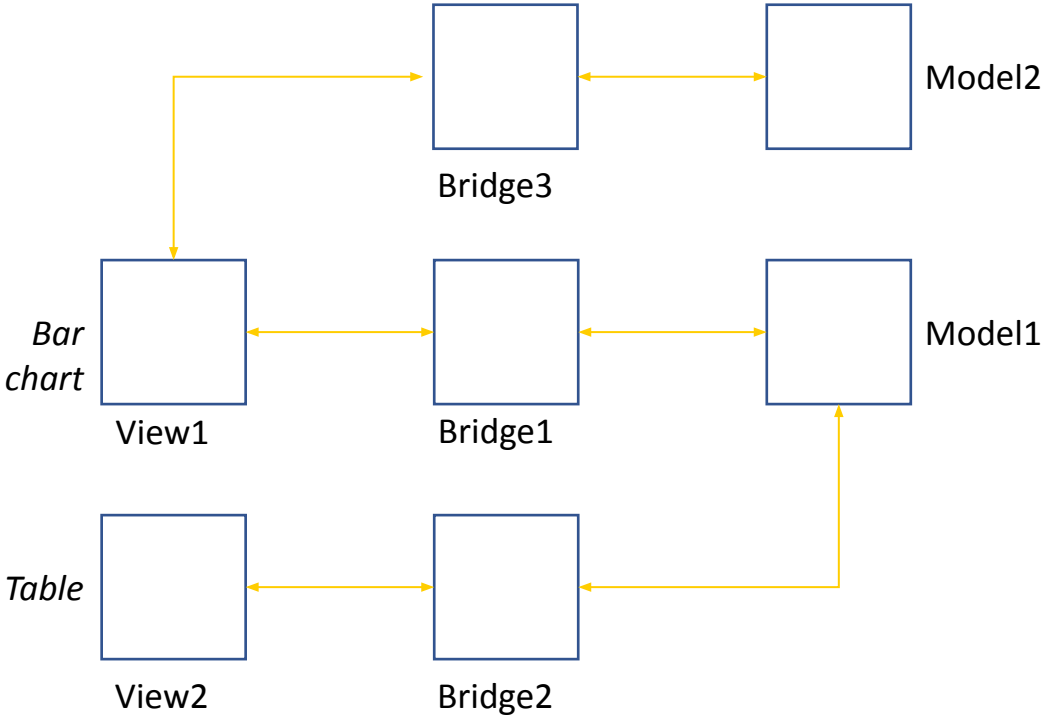




Separation of concerns:

- **The management of the data e.g. CRUD**
- **The visualisation of the data**

A generic approach



- We want to **display data** to the user **in different ways** e.g. as a bar chart or as a table of data.
- We want to keep our **data independent** of the **way it is presented**.
- We may want to **compute additional values** depending on how and why we present data e.g. add a total in our table.
- We need some **connection** – some 'bridge' – **between views and data**.
- We can *reuse*:
  - **models** and
  - **views**.

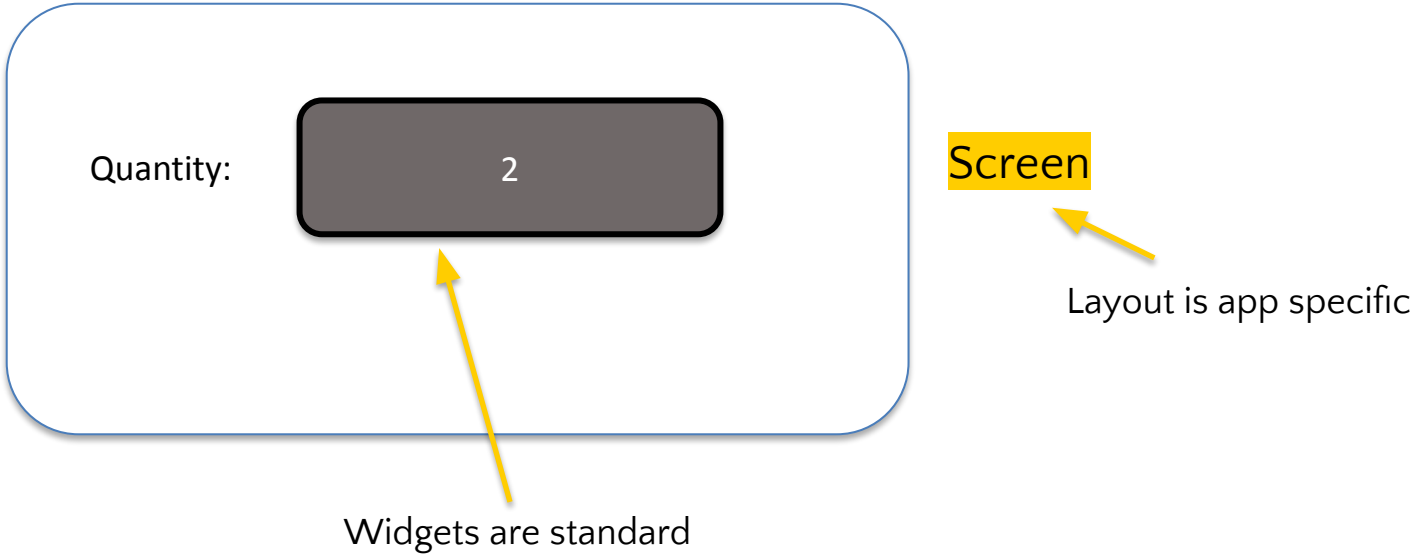
Variations on a theme of Model-View-{Bridge}*

MV...

- Model View Controller (MVC)
- Model View Adaptor (MVA)
- Model View Presenter (MVP)
- Model View ViewModel (MVVM)

Commentary

(Some) disagreements on what *exactly* defines these design patterns



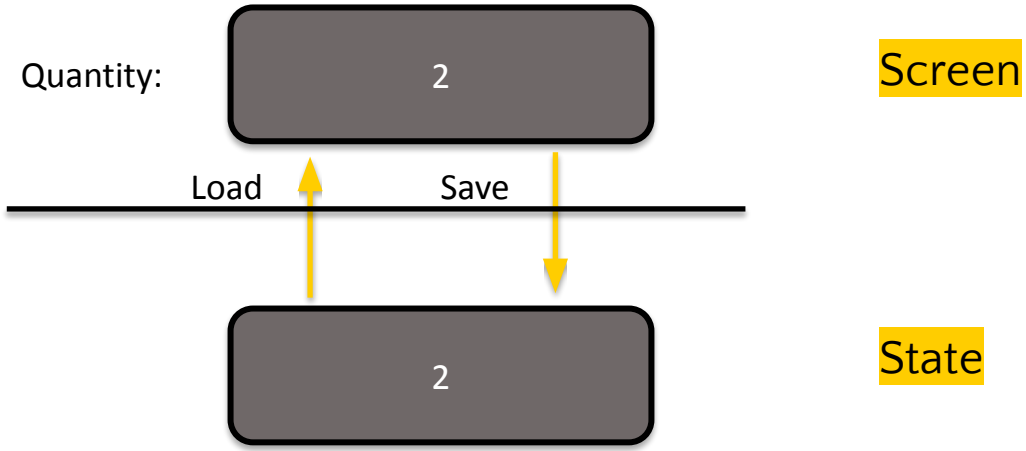
http://martinfowler.com/eaaDev/uiArchs.html

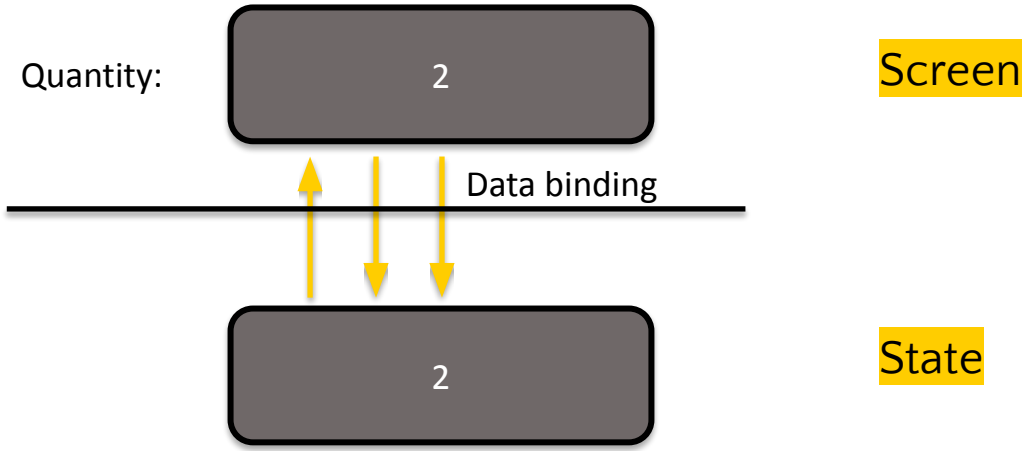
Quantity: 2

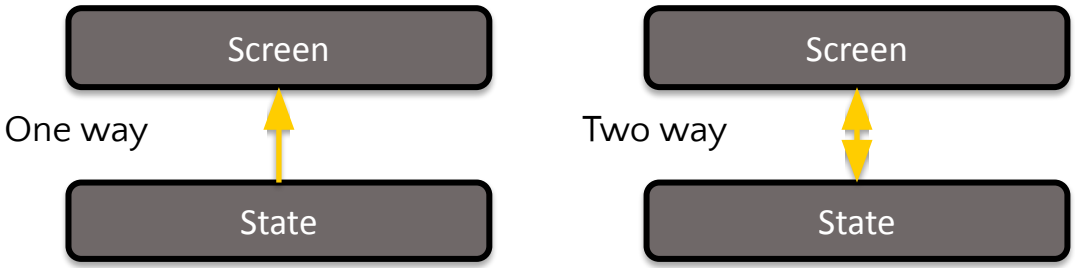
Screen

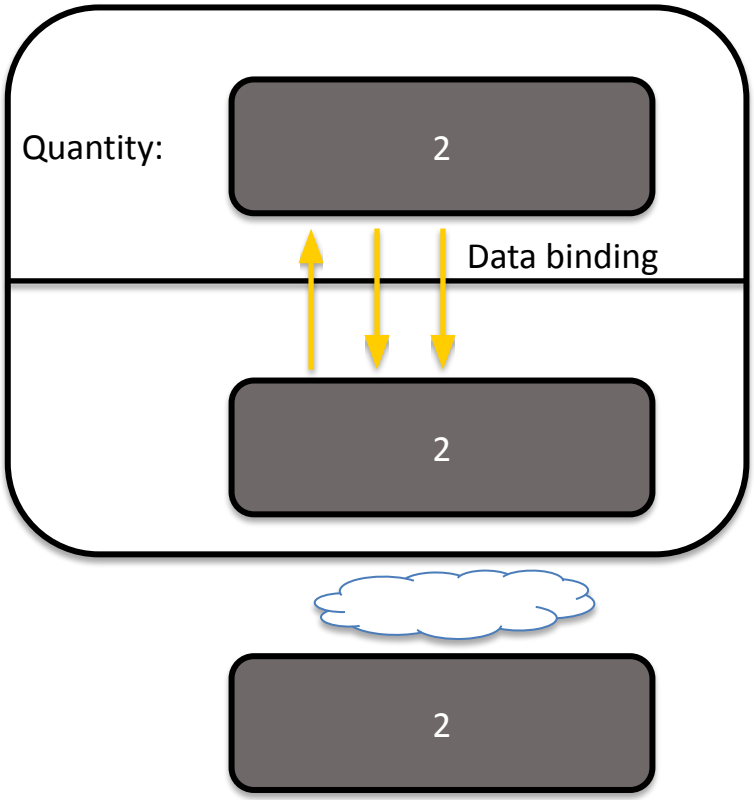
2

State


Quantity:
2
Load
Save
2
Screen
State


Quantity:
2
Screen
Data binding
2
State

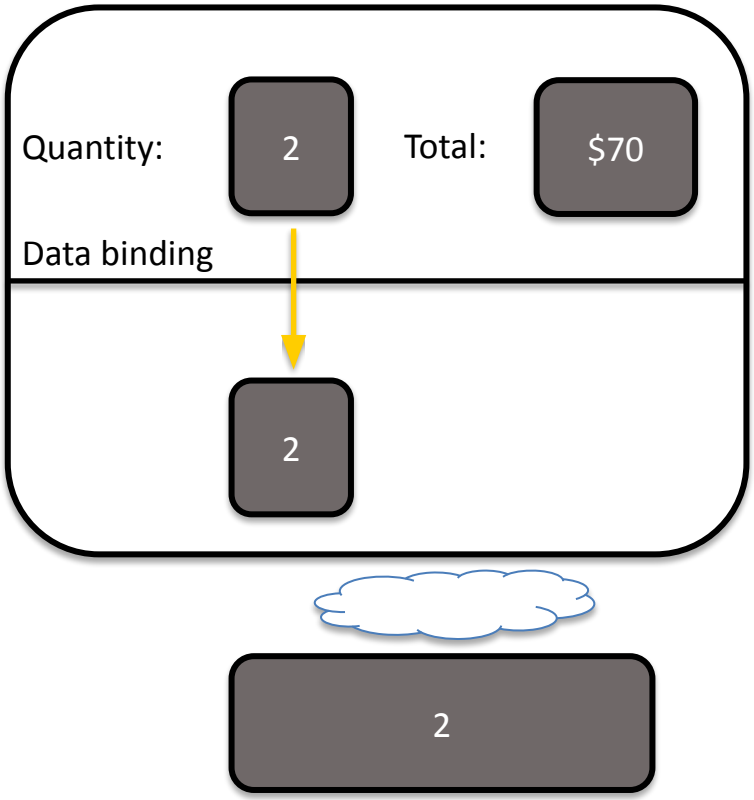

Screen
One way
State
Screen
Two way
State



Screen

Session state

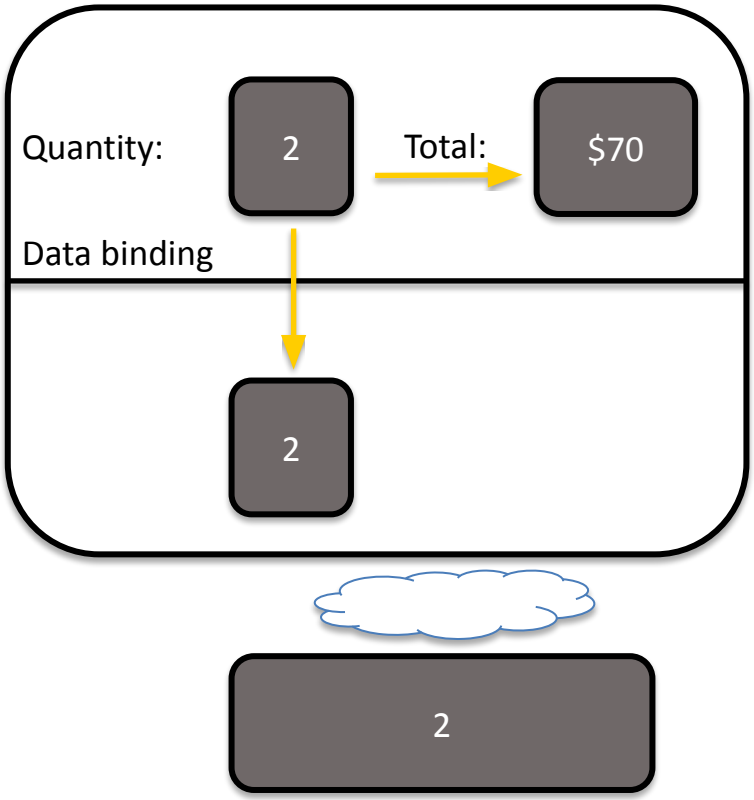
Record state



Screen

Session state

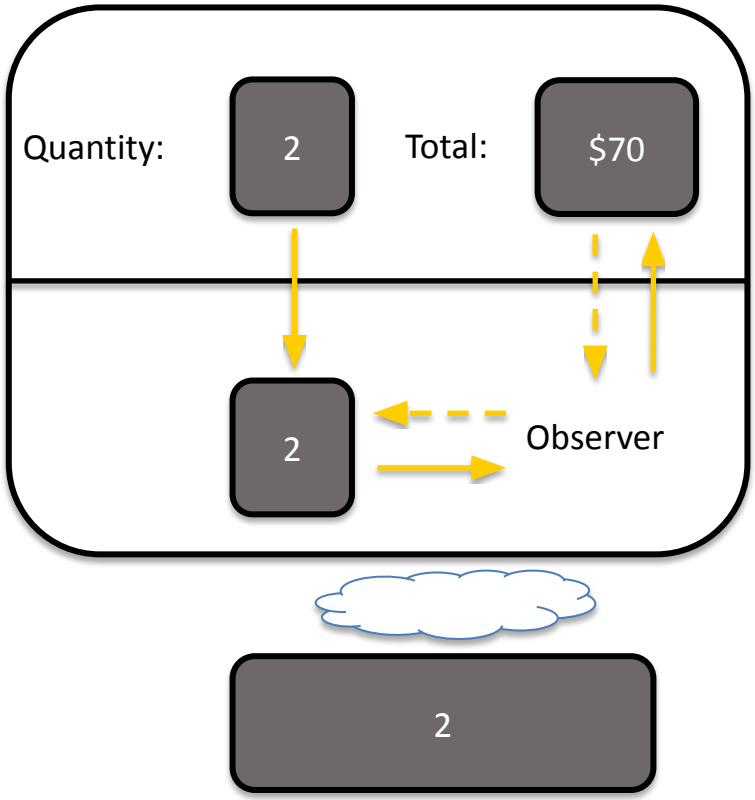
Record state



Screen

Session state

Record state



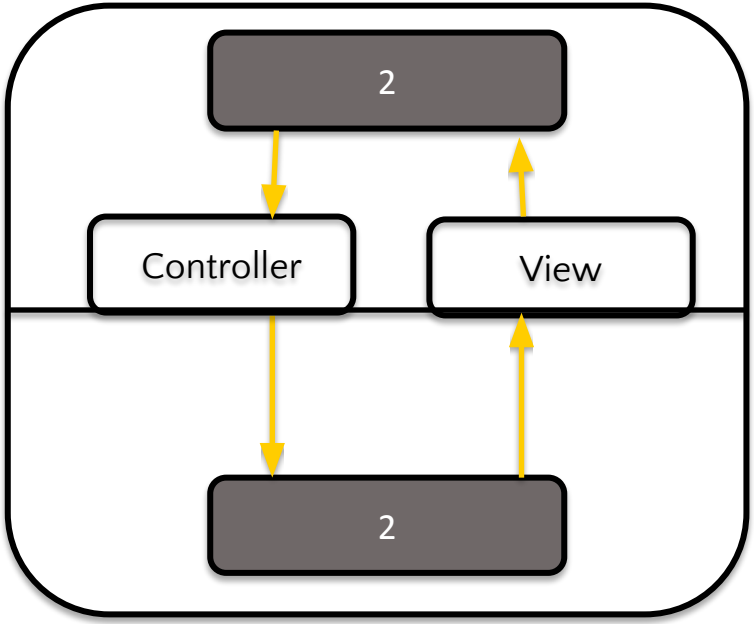
Screen = Presentation

Specific to the UI
Independent of Domain

Session state = Model

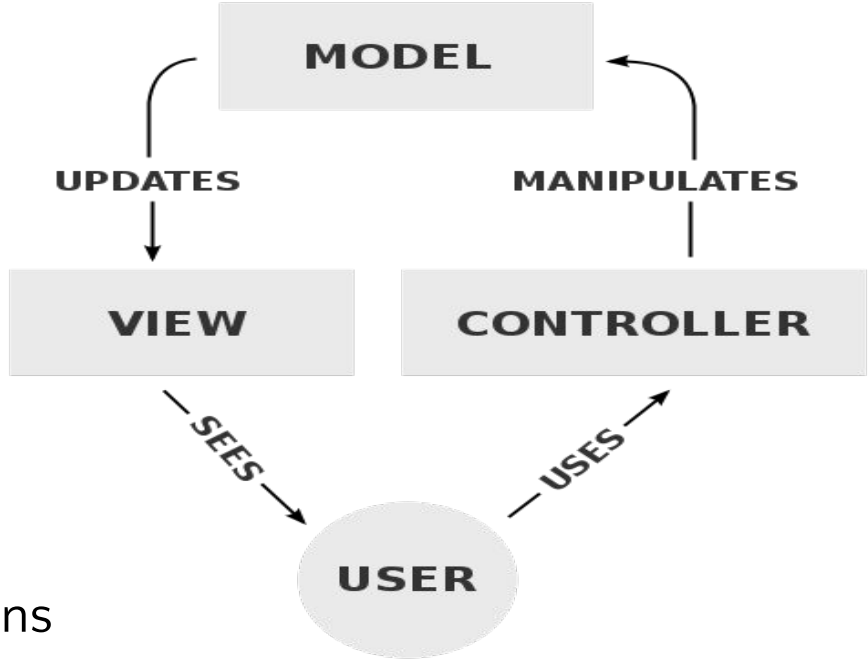
Independent of UI
Specific to the Domain

Record state = Data



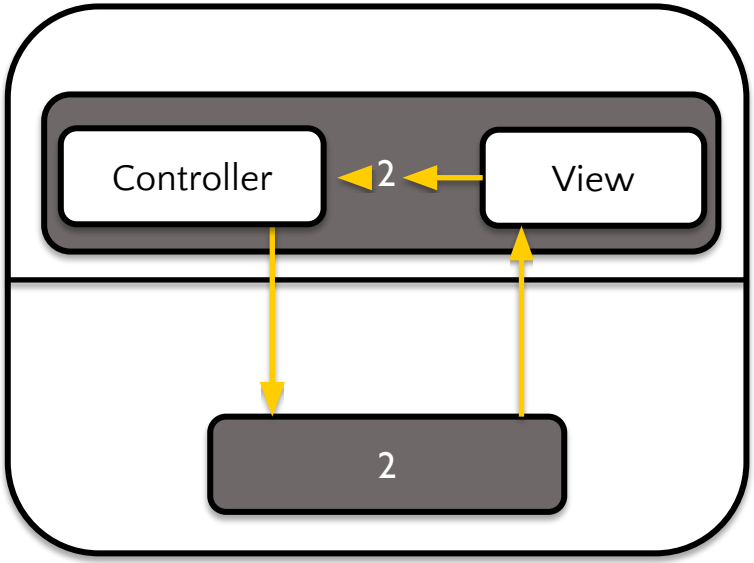
Model

MVC:

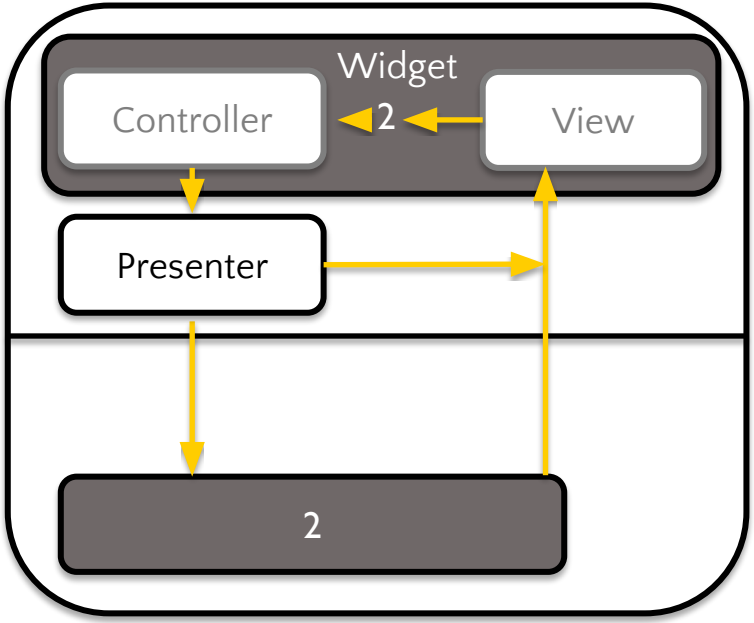


+ Multiple views
+ Synchronized views
+ Pluggable views and controllers
+ Exchangeable look and feel

- Complexity
- Can have lots of updates/notifications
- Links between controller and view
- Coupling of controller/view and model
- Mix of platform-dependent/independent code within controller and view (porting)

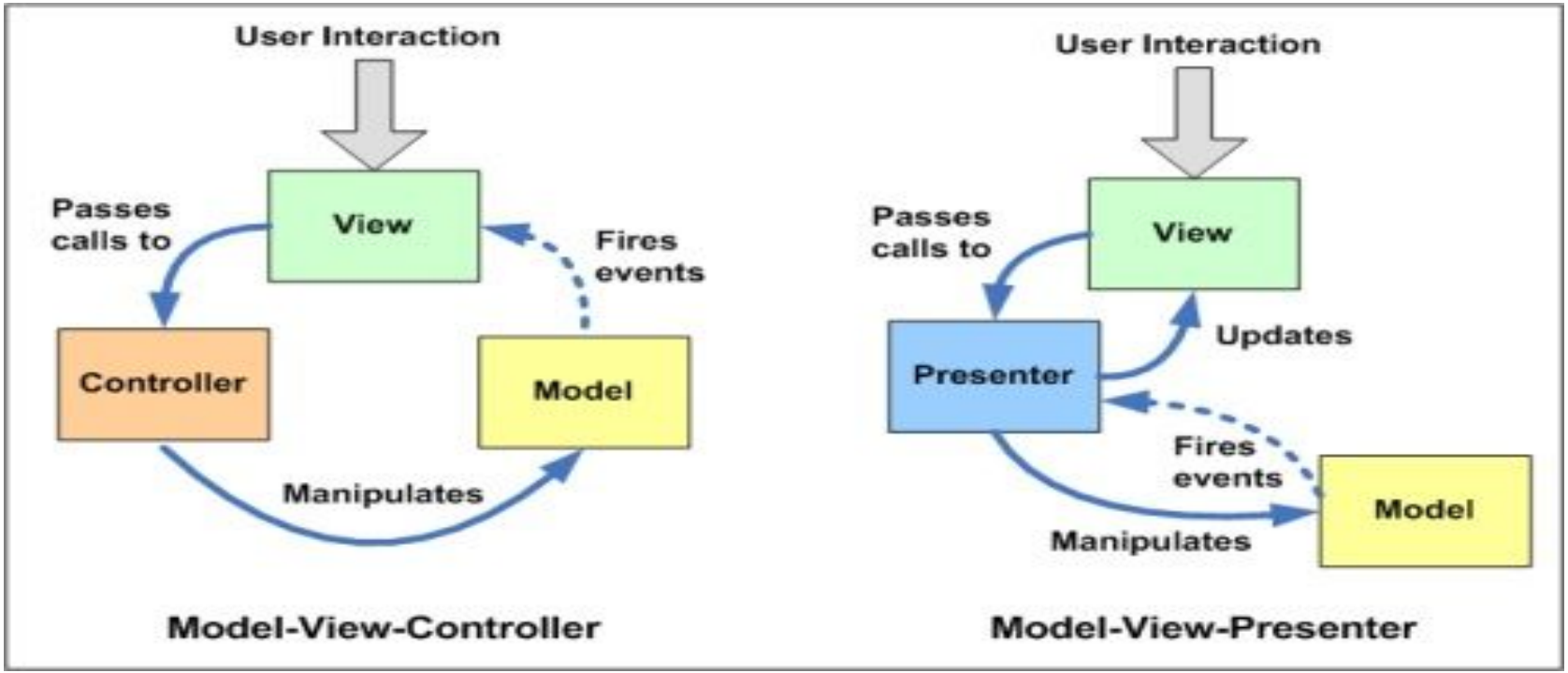


Model



Model

MVC & MVP



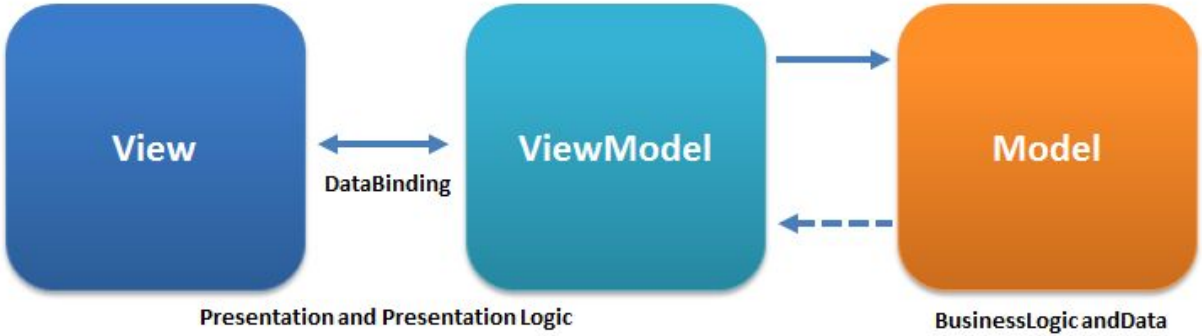
http://joel.inpointform.net/software-development/mvvm-vs-mvp-vs-mvc-the-differences-explained/

MVC versus MVP

- **MVC** is Model-View-Controller
- **MVP** is Model-View-Presenter
- Much hot air expended in defining / comparing / contrasting these.
  - **MVC**: view is stateless with not much logic. Renders a representation of model(s) when called by controller or triggered by model. Gets data directly from model.
  - **MVP**: view can be completely isolated from model and rendering data from presenter, or can be a MVC view, or somewhere in between
- But many variants.
- Categorisation not very important; you mostly just use whatever tools the framework gives you.
- The term MV* may be safer to avoid arguments!

MVVM

Model/View/ViewModel



- ViewModel is just the data currently required by the view
  - In a web context, Model may be on server, View and ViewModel on client
- *Data binding* synchronises view and viewModel bidirectionally
  - Uses lower-level "hidden" mechanism

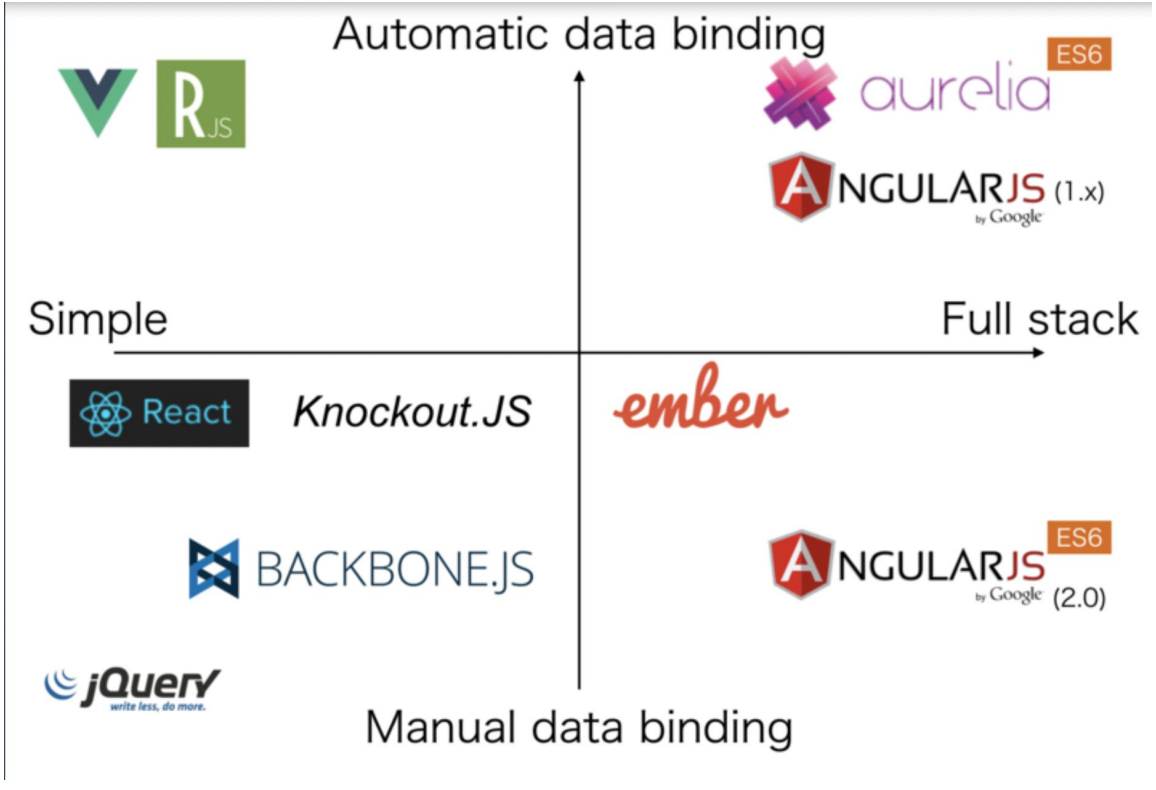
Design pattern summary

- **Developing web applications is challenging**
- **One strategy** to development, and to integration, is to **'divide and conquer'**
- **Separate** out **concerns**
  - **3-tier architecture** of browser, server and data store
- Separate out concerns
  - MV* as a **design concept**
  - **Differences of opinion** on what the M, the V and the * were
  - **Differences on how** M, V and * **interact** with each other
  - Looked at MVC, MVP, MVVM and MVW

Data-binding frameworks

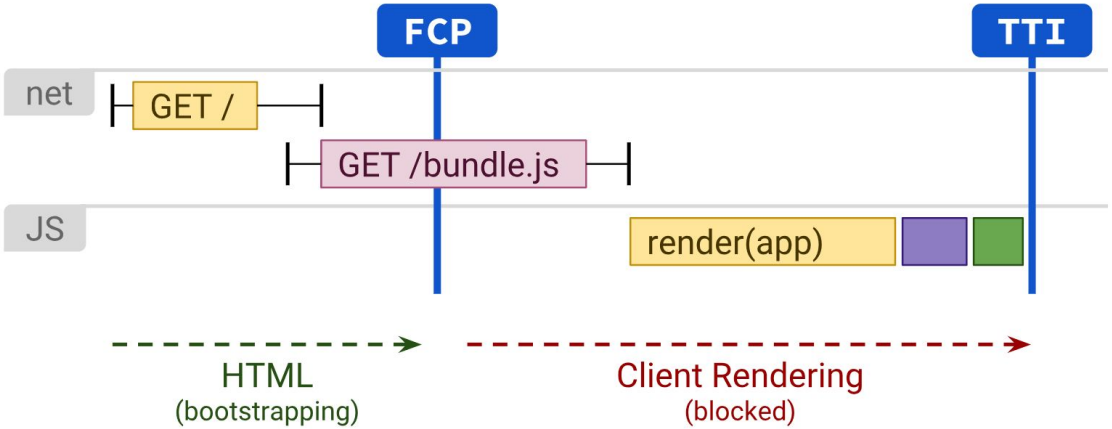
Angular, React, Vue.JS etc.

Data binding frameworks



Client-side rendering

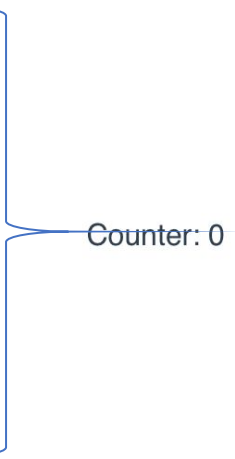
- Renders the page using JS in the browser
- Logic, data-fetching, templating, and routing handled by browser code
- First-contentful page (FCP) as JS bundle loaded
- Time-to-interactive (TTI) after the render function is executed



Templating

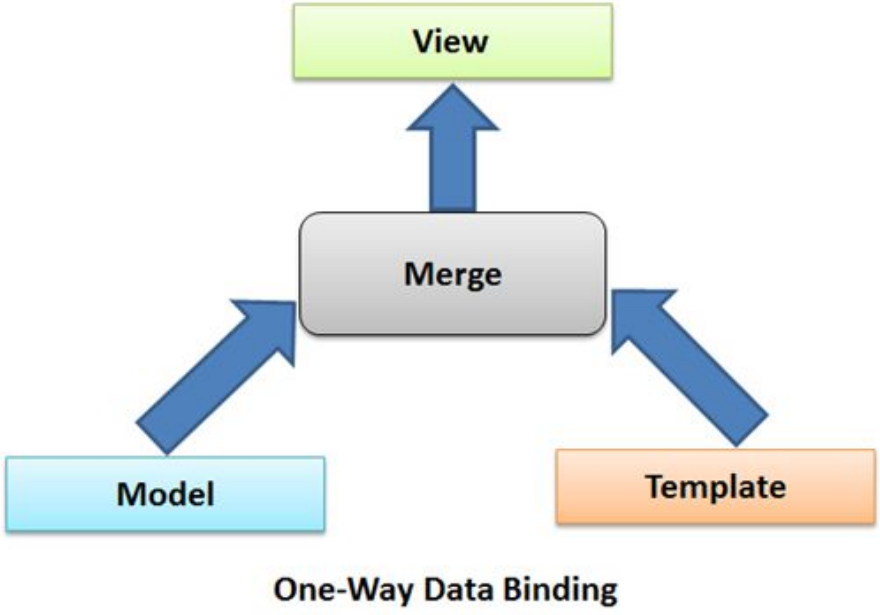
```html
<div id="counter">
  Counter: {{ counter }}
</div>
```

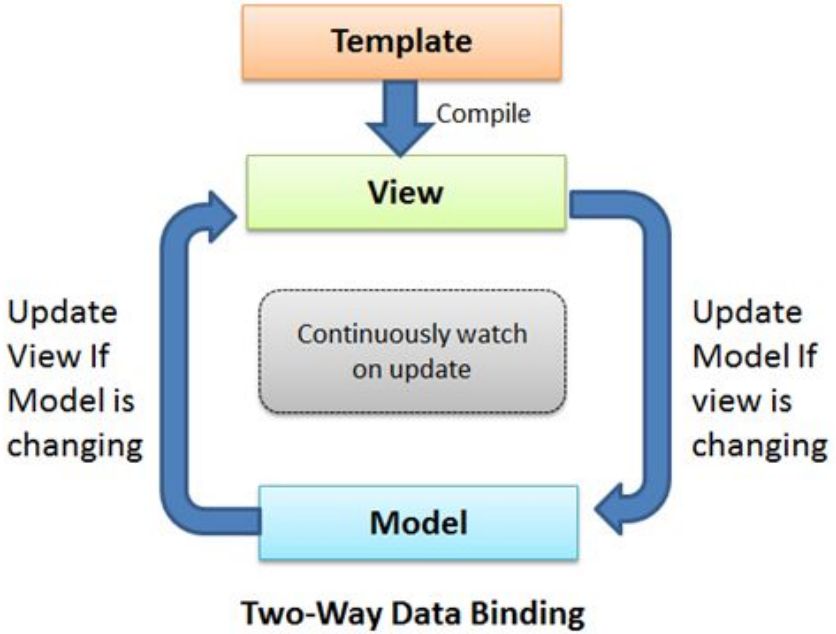
```js
const Counter = {
  data() {
    return {
      counter: 0
    }
  }
}

Vue.createApp(Counter).mount('#counter')
```



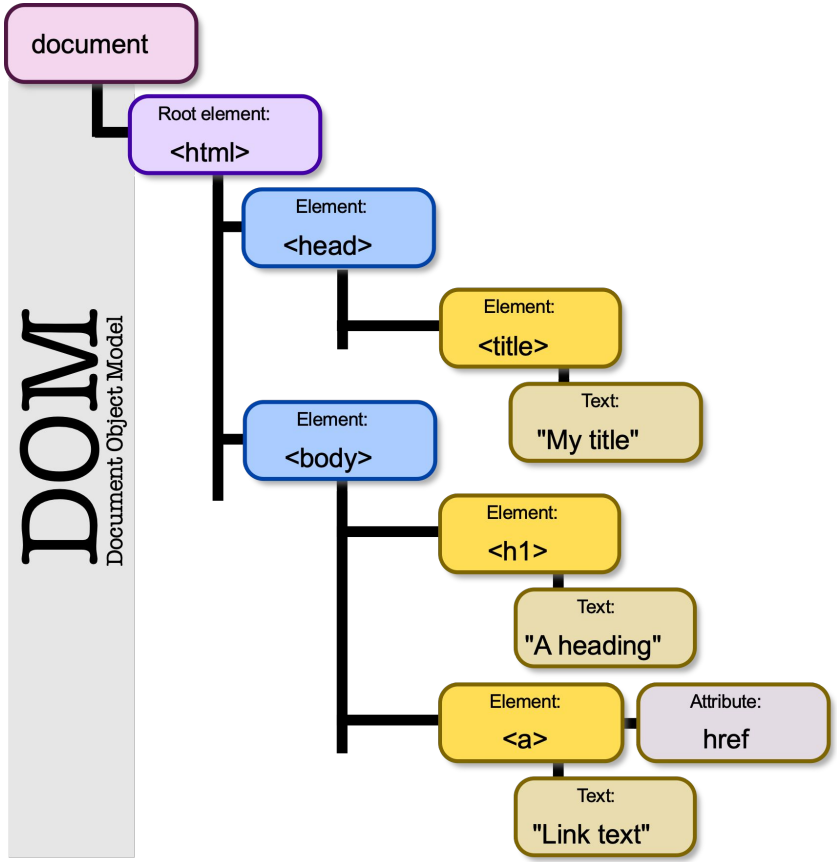
- Frameworks such as Angular and Vue use **templating** to map JS variables to HTML elements
- Data-binding in HTML using **“moustache” syntax**:
  - Most frameworks use curly braces: `{{data}}`
- JS, CSS, HTML separate files / sections

Types of data binding





One way binding can also go the other way: view -> data



“The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.” (W3C). The **HTML DOM :** standard model for HTML documents

Examples of DOM ‘sizes’

Website	DOM count
www.bbc.com/news	2298
Facebook.com	1570
Facebook.com with a few scrolls	8300
Yahoo performance site	700
Trademe.co.nz	3200

1. Load page
2. Open JavaScript console
3. `document.getElementsByTagName('*').length`

DOM performance

- The **DOM (Document Object Model)** can become **excessively large** in an application, e.g. Facebook, when you've scrolled down a bit
- A large number of **DOM nodes to traverse**
- **Performance impact** e.g. you have to modify a large number of nodes (even with a tree structure)

Virtual DOM

- The **Virtual DOM**: an abstraction of the DOM

  - Modify the virtual DOM; update the real DOM from the virtual DOM when needed
  - Libraries / frameworks (e.g. React, Vue.js) will use a virtual DOM in the background
  - You don’t need to work directly with the DOM

https://bitsofco.de/understanding-the-virtual-dom/

Updating the Virtual DOM

When

The data has changed and it needs to be updated: but how do we know that the data has changed?

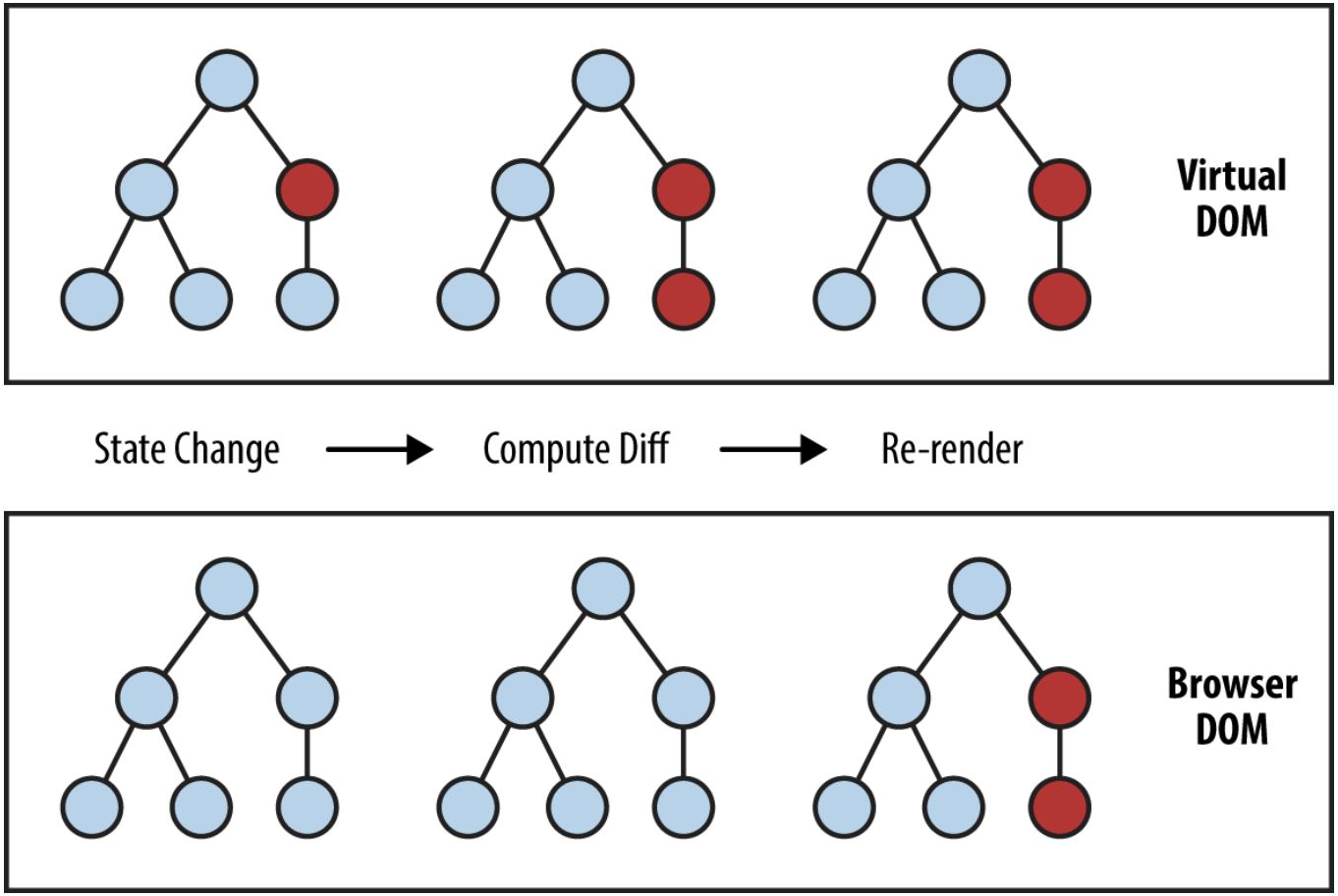
1. **Dirty checking**: poll the data at a regular interval to check the data structure recursively.
2. **Observable**: observe for state change. If nothing has changed, don't do anything. If something has changed, we know exactly what to update.

Updating the Virtual DOM

How

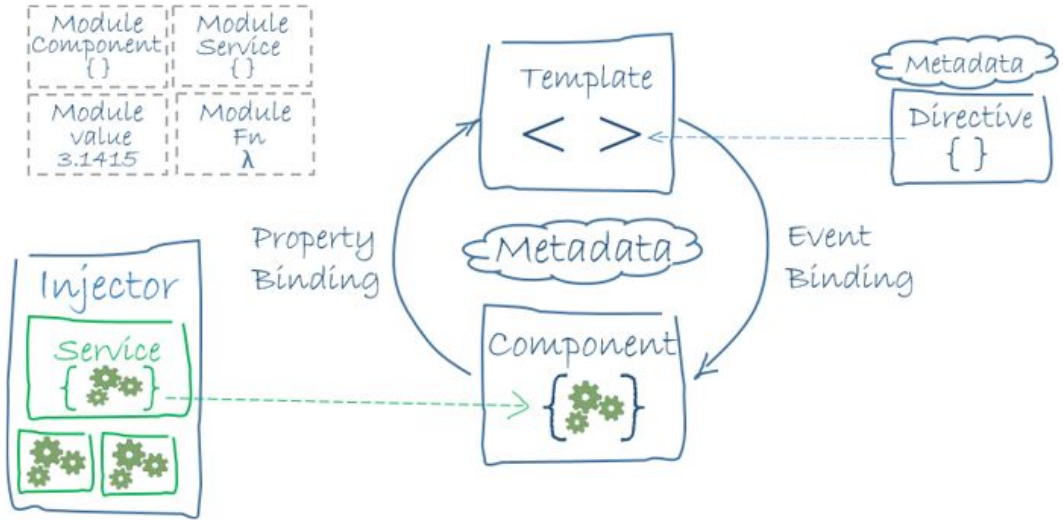
How do we make changes efficiently?

- Need efficient **diff** algorithms.
- **Batch** DOM read/write operations.
- **Efficient update** of sub-tree only.

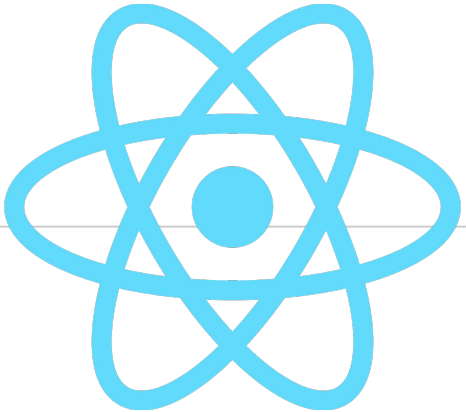

Virtual
DOM
State Change
Compute Diff
Re-render
Browser
DOM

Angular

- Platform for enterprise-scale web application development
- Typescript only (no JS)
- Relatively complex to learn
- Two-way binding
- Dirty-checking to know when DOM should change



React



- Unlike Angular **does not use templating**
- Rather defines reusable **components** in **JSX**
  - Brings the HTML inside the JavaScript
  - Each component has a lifecycle
- Most popular framework at present
  - Thousands of third-party libraries
- Uses a **virtual DOM**
- More about React in coming weeks

VueJS

- Designed by Google dev who found Angular too heavy-weight
- Uses component with lifecycles concept & virtual DOM from React
- But templating in HTML, instead of JSX
  - Closer to native HTML than React, whereas JS is the “starting point” in React

Svelte

- Svelte is different again, it parses .svelte files and compiles into JavaScript
  - Uses abstract syntax tree to generate JS and CSS
- Compiled JS mounts the component, handles events, and patches the DOM directly (no virtual DOM)
  - All HTML elements are created by JS
- No framework code, so small and fast code

https://lihautan.com/the-svelte-compiler-handbook/