Putting it all together on the client

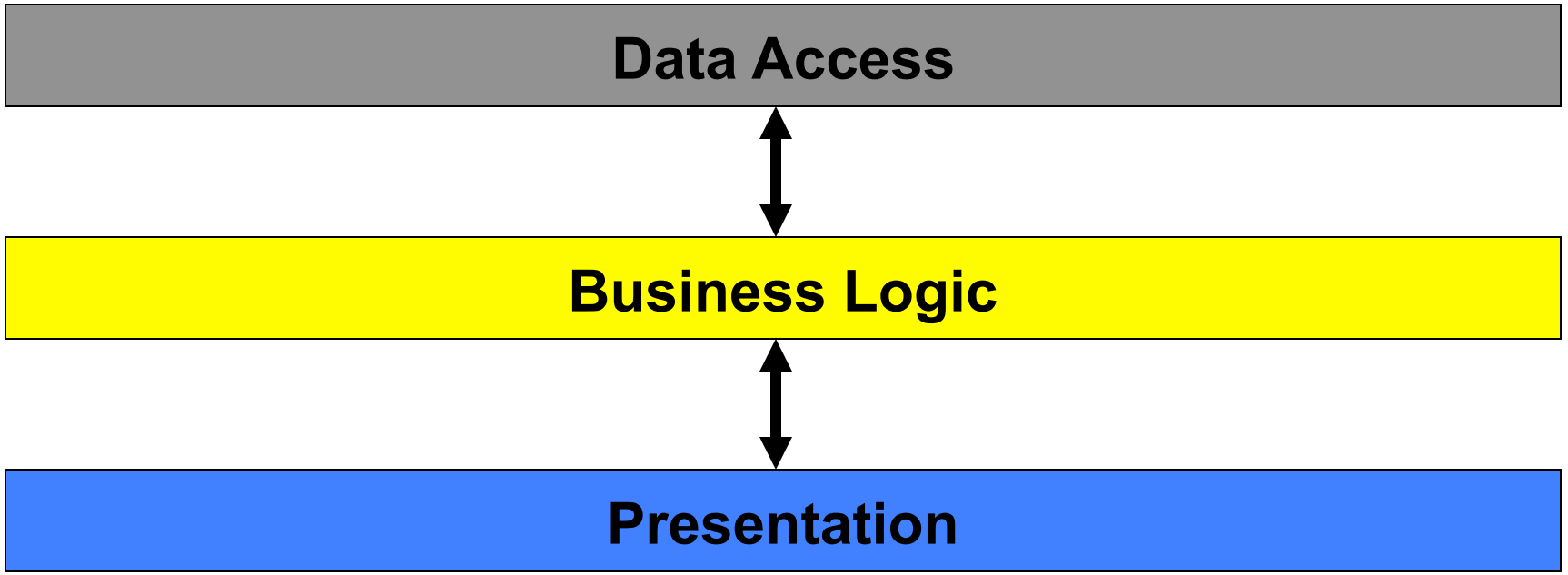
Alphabet Soup:

MVC, REST, AJAX, SOC, MVP, MVVM

Yum! Yum!

déjà vu: Separation of Concerns (SoC)

Web applications have 3 important aspects, or *concerns*



It is important to have *Separation of Concerns* (SoC):

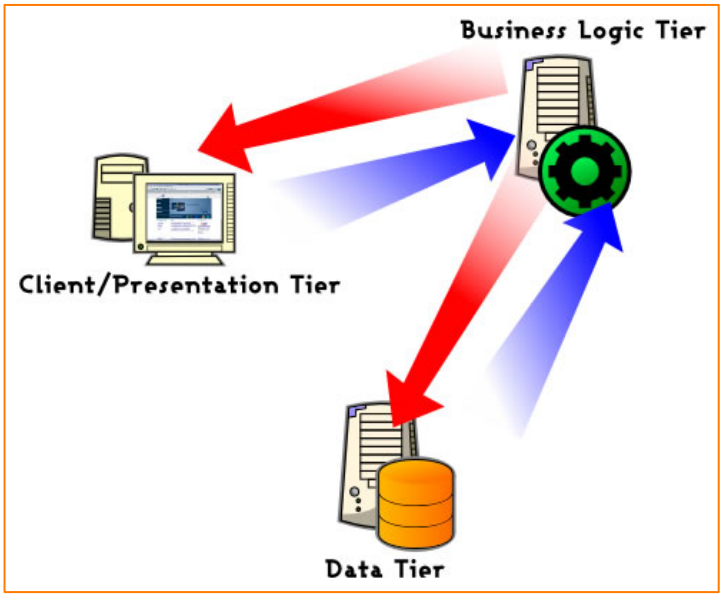
- Allows custom expertise and frameworks to be applied to each concern
- Allows for greater run-time deployment flexibility & optimizations

déjà vu: Separation of Concerns

Allows for greater run-time (deployment) flexibility

- Businesses had (still have) a wealth of proprietary (and poorly understood) business rules
- Businesses had (still have) a wealth of proprietary (and poorly understood) datasources
- Often these two layers were quite incestuous
  - Business rules coded as stored procedures, database triggers, long SQL queries, or (Blech!) COBOL

SoC says to separate these layers & deploy independently



http://www.adobe.com/devnet/coldfusion/articles/ntier.html

MVC – Model-View-Controller

- *MVC != SOC*, and *MVC != n-tier*
- the **View** is the Presentation
- the **Model** comes from Datasources,
- the **Controller** is not the *Business Logic*
  - that also is in the **Model**
  - Controller **routes** requests

déjà vu: Model-View-Controller (MVC)

THE design pattern of the web

- What is a *design pattern* again?
  - A reusable solution in context
  - The *context* here is a web application (our 6 steps)
  - The reusable solution is the framework we will use to handle those common 6 steps according to the MVC structure
    - Yes MVC is a Structural pattern
    - The framework we will use is Express, but there are lots of them out there in lots of languages

History: MVC did not start with the web

- Started in the 70s in the Smalltalk community
- Yes, the same community that gave us Go4 Design Patterns
- See the “Thing-Model-View_Editor” reading

déjà vu: Implementing MVC

Steps:

1. HTTP request made to *Controller*
2. Controller (*routes* or *delegates*) to the *Model*
3. The Model executes the *business logic* on the *world model* (non-volatile state, typically a datasource)
4. The Controller identifies the appropriate *View template*
5. The View (*read-only)* accesses the Model to get the dynamic output
6. The View produces the final rendered response to the client browser.

Compare these 6 steps to the 6 steps of the Template + Strategy Pattern we discussed in the HTTP notes

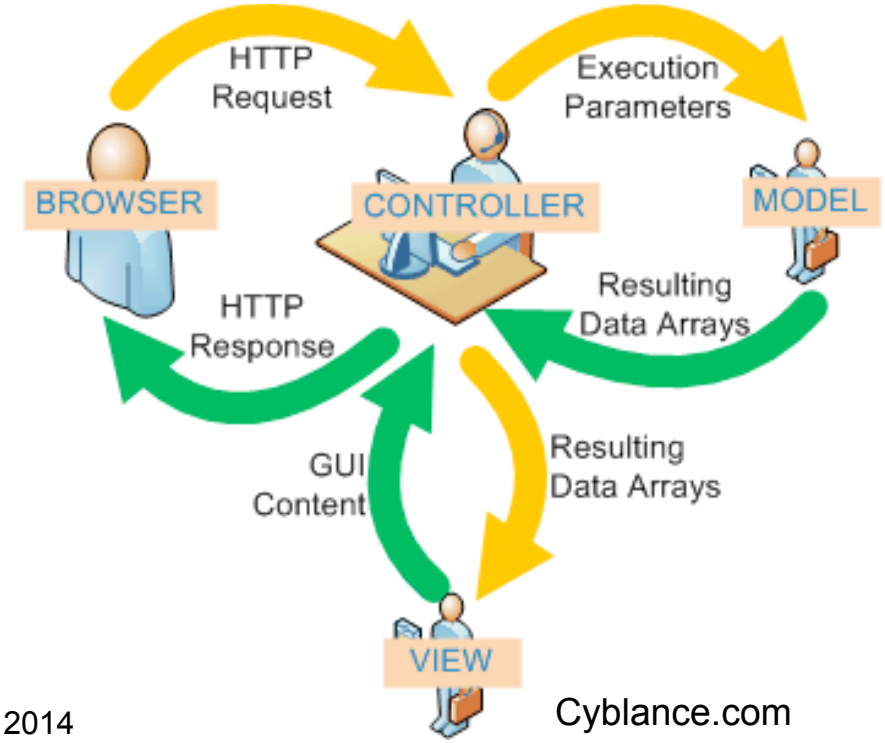
Key Concept Summary: MVC

MVC does not map directly to SoC

- the **View** is the Presentation and the
- the **Model** comes from Datasources,
- the **Controller** is not really the *Business Logic*
  - that also is in the **Model**
  - Controller routes/delegates requests and responses

We will see variations of MVC

- *Front Controller Pattern*
- *Model-View-Presenter (MVP)*
- *Model-View View-Model* (*MVVM*)
- Browser variants (Javascript)

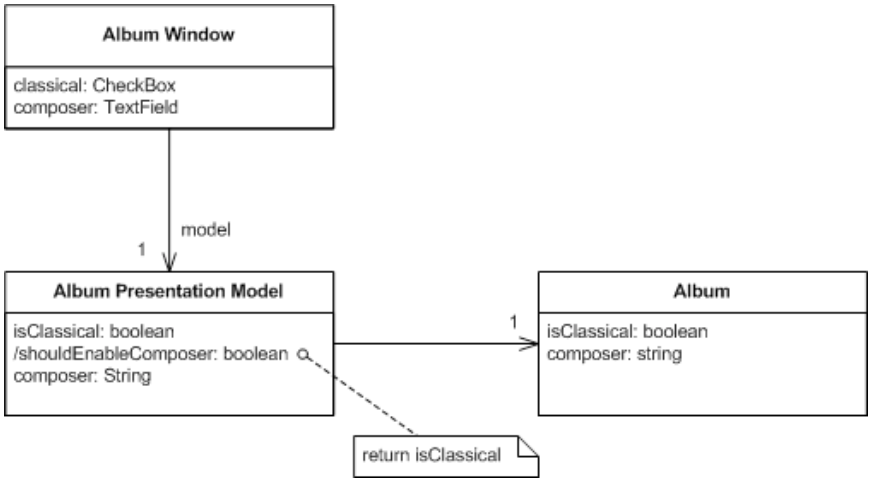


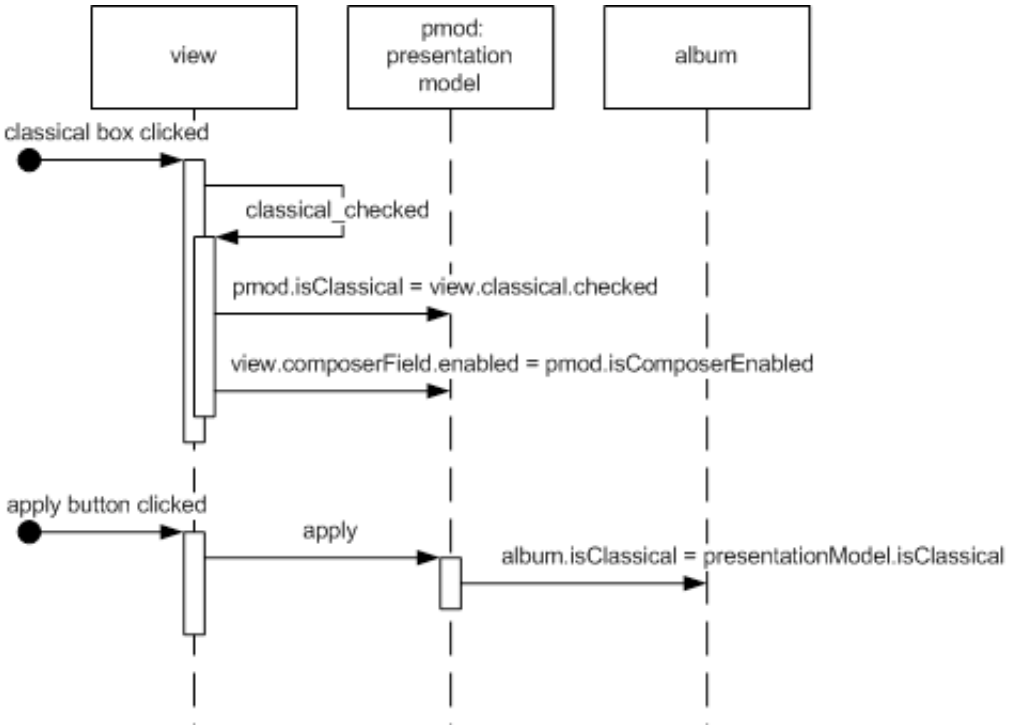
Have you coded a desktop GUI recently?

- There are many Design Patterns for GUIs, Fowler gave us the *Presentation Model* pattern

  "Represent the state and behavior of the presentation independently of the GUI controls used in the interface"

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





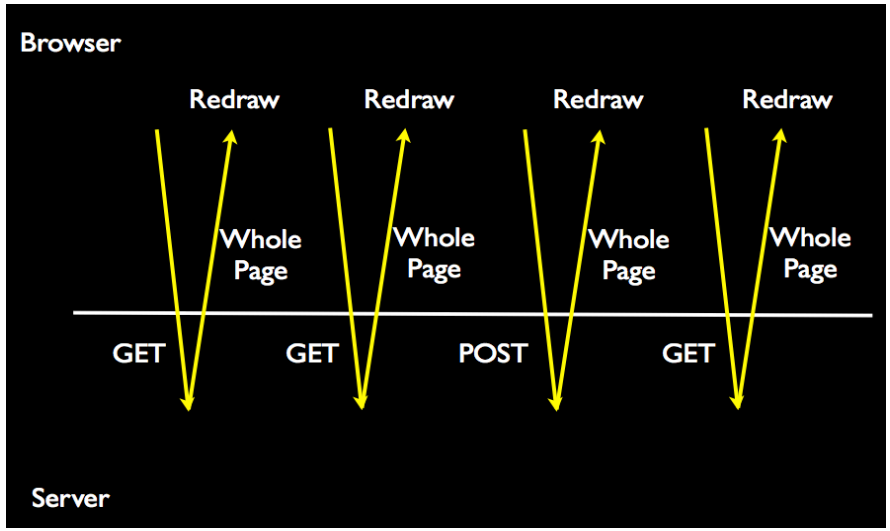
Hats off to Microsoft…

…they gave us XMLHttpRequest

But weren't we doing so well on the server-side?

- Um, well, "kinda", "sorta"
- Presentation management on the server is a pain
  - Attempts to create a desktop-like GUI model in the View are awkward (JSF)
  - Round-trips to the server for many things simply are not necessary

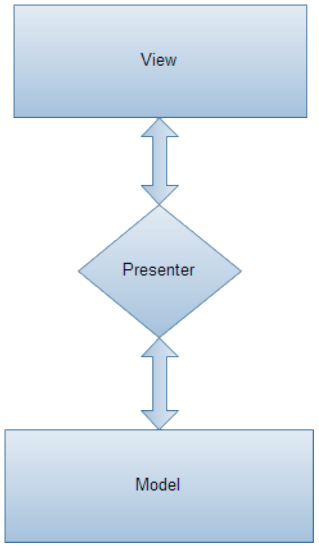
- AJAX got us started
  - Seen as a way to cut down on server roundtrips and full page refreshes
  - Evolved into new design patterns for the client



Hats off to Microsoft (again) …

They gave us MVP

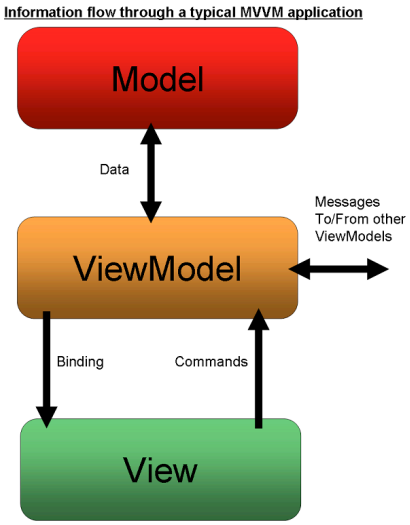
- *Model-View-Presenter* (or really, *Model-Presenter-View*)
  - A realization that the passive View in MVC with the active event handling of the Controller was actually a bad decoupling
  - The View could now handle interactions, but delegates them to the Presenter, which in turn interacts with the Model
    - Note the View does not interact with the Model (no object thing)
    - Being more active now instead of passive allows the Javascript/ AJAX approach a cleaner design



Images from codeproject.org

They gave us MVVM too

- *Model View ViewModel* (yeah that is written correctly)
- An adaptation of the Presentation Model from Fowler:
  - Separate content/data a widget acts upon from the widget itself
  - This new factored out object is called the "ViewModel"
  - Interactions with ViewModel can be handled locally; the magic is knowing when the Domain Model ("the" Model) changes
- Ever hear the term "code behind"? (no I don't like it either)



Going for it all (well, almost all)

MVP and MVVM are patterns based on desktop UIs

- Play well on the web given increased bandwidth, HTML5 and Javascript popularity, and the ability of AJAX to update UI state dynamically through server interactions.
- But the Controller is still on the server
  - Changes to the Domain Model are still the result of actions routed to business logic invocations on the server, resulting in a page refresh

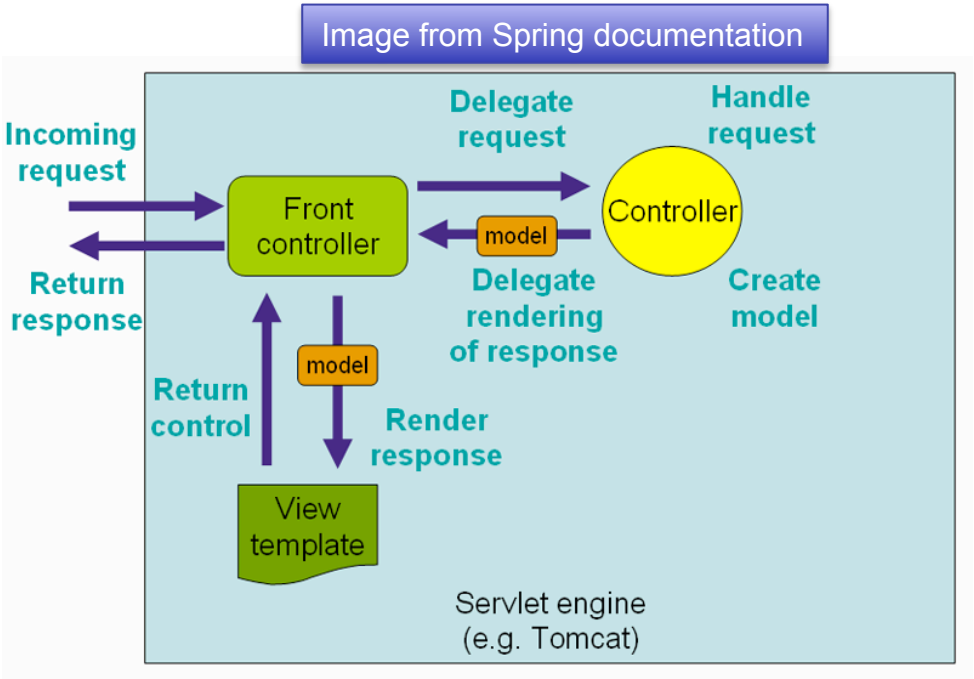
Can we (should we) move the Controller to the Client???

- Well, you *can* if you really want to
  - Javascript responds to all events with AJAX calls to RESTful services
  - The markup is a shell that the Javascript can manipulate over and over without a full page refresh from the server (Single Page Application “SPA”)
  - Your Javascript must know the application workflow and choose Views
- But *should* we?
  - Variants on the design pattern are still evolving, like the *Front Controller* pattern on the next slide

Front Controller Pattern

- A *front controller* is a web-specific pattern that provides a single URI entry point for all requests, then delegates those requests to the proper (true) Controllers.
  - Javascript can in fact implement a lot of what a FC does

- Figures out what actual Controller to invoke
- That Controller can be RESTful now
- Whether we AJAX or not depends more on View
- “Empty” Controller requests can stay local



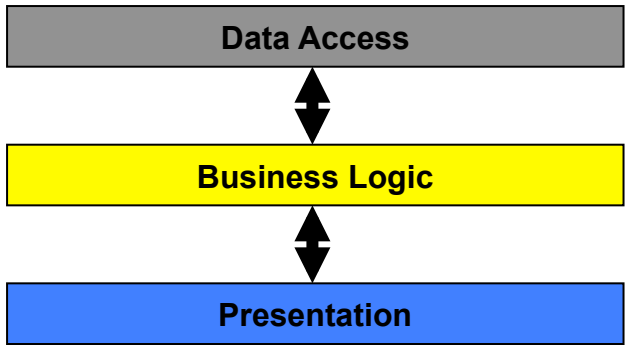
Discussion

So what we have is a “splitting” of our MVC pattern:

- The “model” is not part-UI state and behavior and part world-model state and behavior
- The “view” is mostly in the browser now (now we have templates and transforms and CSS) but not 100%
- The “controller” and it’s routing logic is split via variations like the Front Controller pattern
  - We will see this concretely with Angular

MVC became popular to maintain good SoC. How we doin’?

- Presentation is like View above
- Business logic needs to stay on server
  - REST helps with this
  - If you move to JS, you can lose control
- Data Access
  - Again, REST helps with world model data
  - Be careful not to leak data (privacy)



Summary

App architectures are moving toward our 6th style, “fat client”

Forces for this change:

- Desire for better user responsiveness
- Cost of maintenance of server-driven “pages”
- Tech advances in browser technologies (HTML5/JS/CSS)
- Evolving understanding (Fielding’s work on REST) and the advantages API-driven systems give on the server
- Mobile and gaming devices

Devs trying to hang onto the existing MVC-driven style

- Pattern extended to MVP, MVVM, Front Controller
- The “splitting” of the M, V, and C happens in different ways, and those ways can present their own issues

Will MVC remain the predominant web app design pattern?

- Well, it won’t go away, but now apps are evolving toward more p2p, interactive,and “push” oriented, stretching the limits