CSCC09F
Programming on the Web

Client-Side Frameworks

SPAs, Backbone.js, Underscore

Web Apps, the 1st Generation

- The first generation of Web apps operated according to the Web's "document"-interaction model:
- A user entered a URL into their browser … the browser requested that document from a server … the browser rendered what the server sent back
- We can build apps on top of this model, but in doing so, we delegate most of the application logic to the server, and the client is mostly a rendering engine
- This early generation of Web apps had "thin clients"
    - considered a good thing, since clients weren't trustworthy environments didn't want to download business logic to them

Single Page Applications (SPAs)

- Single Page Applications (SPAs) work by loading a set of startup data into the browser from a server and then reacting to data changes and user interaction (state changes) on the client side without subsequent requests to the server for complete-page refreshes
  - startup data will typically include a mix of HTML, CSS, JavaScript, and possibly other resources like images
- When a client-side state-change necessitates data be sent to/from the server, an Ajax request is issued to a server API
- The client-side responds to an Ajax request by updating the client view and/or state accordingly

Single Page Applications (SPAs)

- A primary goal of SPAs is low-latency interactivity, so that Web apps can be more competitive with “desktop” and mobile “native” apps
- To accomplish this, business logic previously implemented by a server gets pushed out to clients as JavaScript
- With all this new logic happening on the client, JavaScript applications quickly start to get quite complicated
- Rather than just displaying information in the form of HTML/CSS, an app must now: respond to <u>events</u>, maintain state of <u>models</u>, explicitly initiate Web <u>requests</u> and define <u>callbacks</u> to process the results, and make sure client and server states remain <u>consistent</u>

Single Page Applications (SPAs)

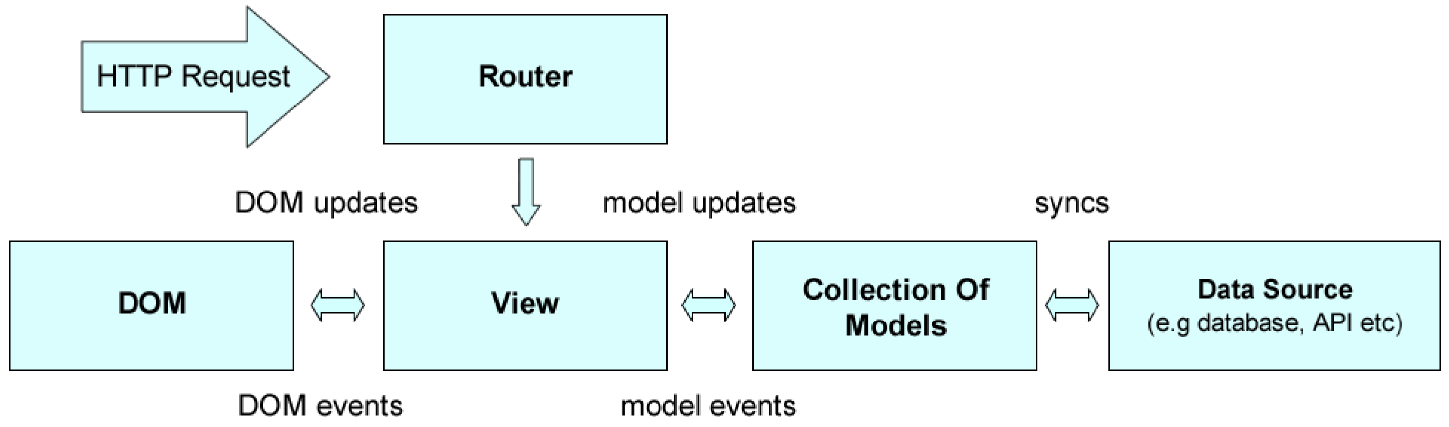
- Building sophisticated SPA's in plain JavaScript would be very expensive due to the low-expressive power of its DOM API and browser inconsistencies
- Moving up to a library like jQuery helps (a lot), but this approach often leads to "spaghetti" code with no overall architecture (structure)
- Keeping the HTML UI, JS DOM, JS models, and server-side datastore in sync is tricky, especially when you factor in the asynchronous nature of server-side updates
- Trying to keep views updated in response to user interactions and model changes can result in "jQuery callback soup"

Single Page Applications (SPAs)

- A client-side framework helps, by providing built-in support for common tasks, such as:
  - abstracting remote (server) resources into models and collections on the client
  - supporting model/collection event-triggers on state-change and enabling view-code to listen for these events
  - syncing of models with persistent (server) datastore
  - view templating
  - supporting addressable app-views through URLs
  - keeping the overall codebase organized in a manageable structure: MVC or MV*

Single Page Applications

- In a SPA with client-side MV*, a Router intercepts URLs and triggers corresponding client-side JS handlers (within a view) – without issuing an HTTP request to the server.



- In addition to URL routing, DOM events (e.g. UI change) and Model events (e.g. value changes) trigger View handlers
- These handlers in turn update DOM and/or Models, which may trigger other events
- Models are sync'd with data sources, possibly on servers

Not all Apps are SPA

- Not every app fits the SPA model
- For an app that relies on a server for most of its business logic and “heavy lifting” computation of page/view rendering, with a relatively small/simple code on the client to improve interactivity, the extra time to learn/use a client-side framework may not be justified
- A library like jQuery is still useful, for achieving browser independence, compact/readable/maintainable code

What is Backbone?

- Flexible, minimalist solution to improve client-side code structuring
- Well supported, with extensive documentation and an active plugin community solving specialized problems not addressed directly by Backbone
- Simplifies server-side persistence – don't have to manage low-level synchronization of server/client-side data-views
- Decouples the DOM from app data (model)
- Provides synchronization of DOM, models and collections
- Provides event-notification of data-state changes
- Expresses model, view, routing concepts succinctly and following a consistent pattern

Who Uses Backbone

- Linkedin
- Foursquare
- Airbnb
- Groupon
- Walmart Mobile
- Disqus
- Khan Academy
- Code School
- Trello
- SoundCloud
- Stripe
- Pandora
- Metalab
- Seatgeek

Why/not Backbone

- Why Backbone?
  - Backbone was one of the earliest JavaScript client-side MV* frameworks, and thus is now…
  - Mature/stable, with a large user base, an active developer community, and a good base of documentation, examples, and solution-models for common problems
  - Backbone is simpler than its main competitors, Angular.js and Ember, giving it a flatter “learning curve” and less behind-the scenes “magic” that we don’t have time to cover in 12 weeks
- Why not Backbone?
  - Backbone is not as prescriptive/sophisticated as competitors such as Angular – when you’re developing complex commercial apps, these more advanced frameworks may provide a stronger development base