Web Application Architectures

Keith Lancaster, Ph.D.

In Today's Episode

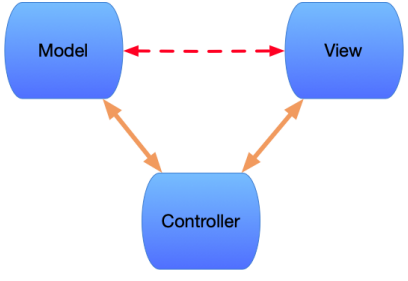
The MVC Design Pattern

In days of old...

- Early web application designs were one-offs, meaning very little commonality between apps in terms of their architecture
- No *separation of concerns*
  - Very common to mix database calls, HTML, Javascript in single files
- Overall development process was onerous
  - In the original ASP.NET, you had to compile the application and then move it to IIS to test. This worked, but was very slow

The Model-View-Controller (MVC) Design Pattern

- Trygve Reenskaug introduced MVC into Smalltalk-79 (circa 1978) while visiting the Xerox Palo Alto Research Center .[1]
- The goal was to *separate concerns*
  - The *model* managed data and business logic
  - The *view* was responsible for the user interface
- The *controller* served as an intermediary between the model and view
  - Until the mid-2010s, used predominatley for desktop applications



Enter Ruby on Rails (RoR)

- Around 2003, David Heinemeier Hansson (DHH) began work on Basecamp, a collaboration application, using the Ruby programming language
- Ruby had been developed by Yukihiro Matsumoto (Matz) in the late 90s as a successor to Perl
  - Heavily influenced by Perl and Lisp
  - *Everything* is an object
- DHH ended up developing a framework for Basecamp, and then released it as Ruby on Rails
- Yours truly used version 0.13 for a startup in 2005 after having a friend mention that it might be a better choice than Java or C#/.NET

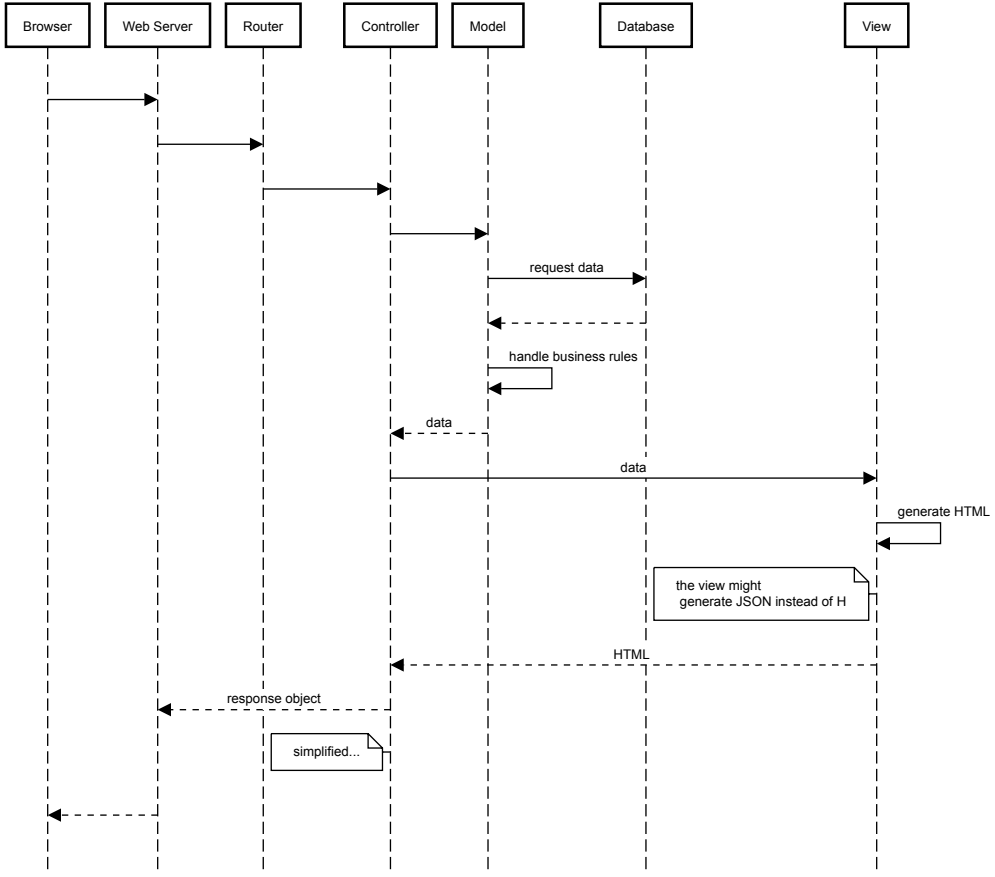
RoR and MVC

- Rails was the first framework to incorporate the MVC approach
- The combination of the MVC approach, the dynamic Ruby language (no compile stage), and the use of code generators made Rails astonishingly fast to use for development
- Yours truly (that is me again) gave one of the first demonstrations in Houston, developing a blog live in front of a large audience of web developers in 15 minutes. There were literally gasps from the audience.

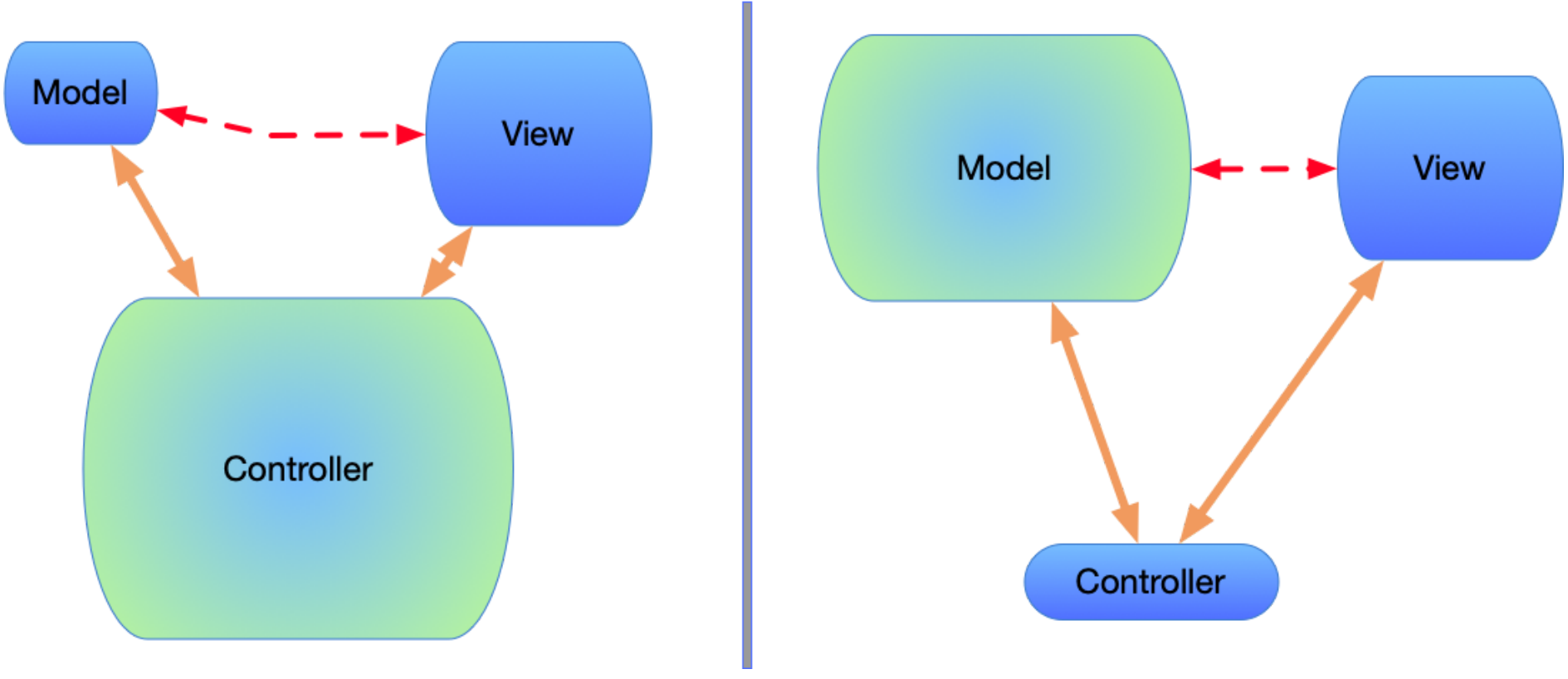
The Spread of MVC

- The MVC approach used by Rails became so popular that other frameworks followed suit
  - MVC .Net
  - Laravel (PHP)
  - Django (Python)
  - etc.
- The MVC pattern is subject to interpretation, so implementations of the pattern are not necessarily identical
- Even within an implementation, the approach to using the pattern changes over time as developers gain experience with the good (and the bad) of the pattern

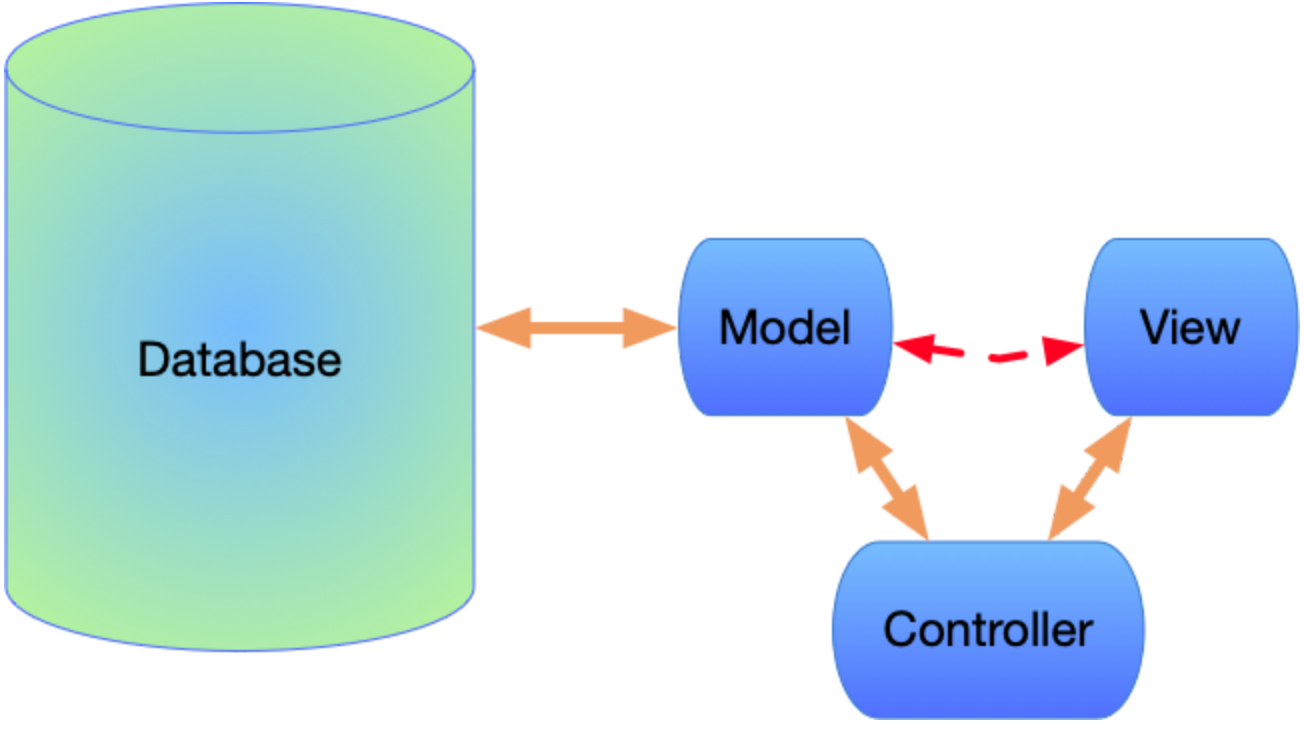
MVC Flow


Browser
Web Server
Router
Controller
Model
Database
View
request data
handle business rules
data
data
generate HTML
the view might
generate JSON instead of H
HTML
response object
simplified...

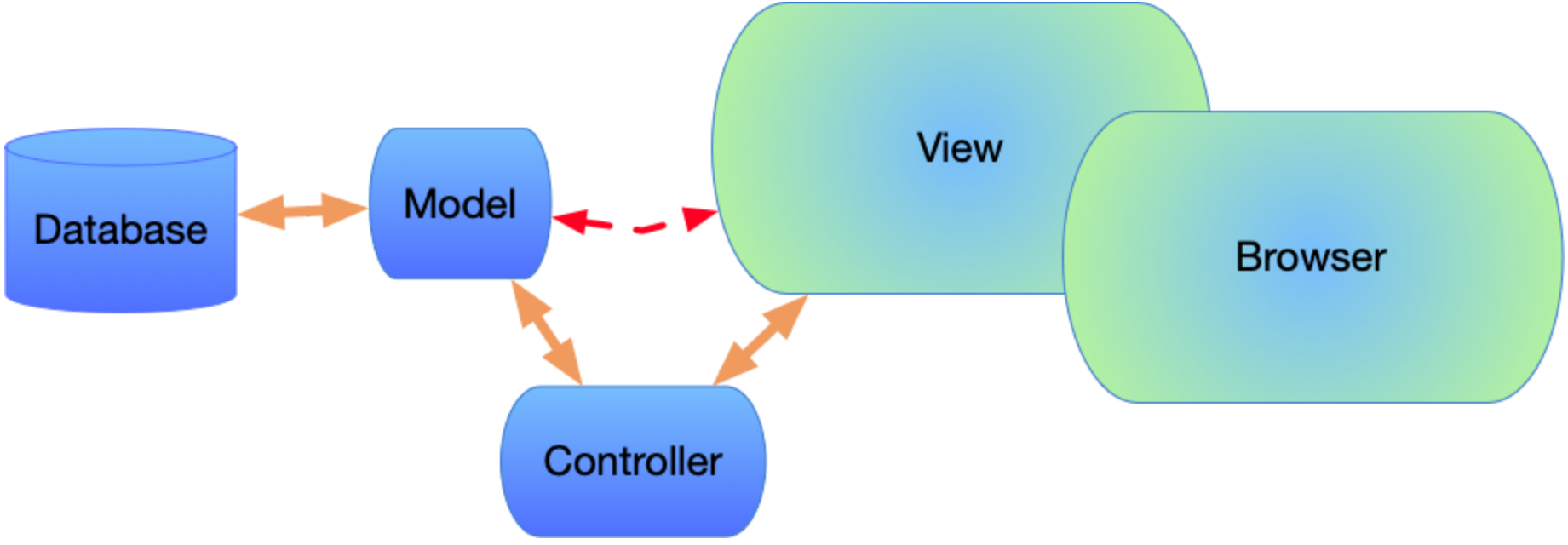
Where Should the Business Logic Go?



An Alternative?



Or Maybe Even...



So Which Approach is "Best"?

- Frameworks tend to be *opinionated*
  - Following Rails yet again, many frameworks have guidelines for where files go, naming conventions, and where to put business logic
- Most have moved away from the fat controller approach
- Increasingly, there is some movement away from the fat model as well
- The idea of services (and micro services) has become popular, however…
  - Microservice architectures can be a nightmare as well compared to monolith applications

The bottom line: This is not a settled issue

One More Contribution by DHH and Rails..

- Prior to Rails, most configuration for an application was performed via XML files
- Rails introduced the concept of *convention over configuration*
  - *Convention over configuration* means designating system defaults, such as where files should be located relative to other files, rather than forcing the end user to make decisions with each new application
  - If you have worked on a Rails app, for example, you will be able to navigate the structure of a different Rails app right away (as long as they followed conventions)
  - Phoenix, which was developed by ex-Rails developers, follows the same convention

What about Phoenix?

- This appears to be a still evolving issue
- Phoenix began with essentially the fat model approach, but then added *contexts*
- Contexts, as we will see, provide an interface into a logical (and data) area
  - Example: An `Accounts` context might contain the `User` data model
- How much business logic goes in the context, as opposed to putting it in a separate space or even separate application is a subject of discussion in the community

Summary

- The MVC pattern (and its variations) is for now the dominate architecture for web application frameworks
- Pros
  - The pattern provides *relatively* clear guidance for how an app should *separate areas of concern*
  - Separation of concerns in general leads to easier
    - code maintenance and bug fixing
    - onboarding of new developers
    - extension of functionality

Summary (2)

- Cons
  - There *are* variations in the pattern implementations (is this really pro?)
  - Issues such as where business logic, field validations, etc. are still in flux. Again, not really a con exactly, but something that makes it a bit confusing when trying to sort out the best approach to an application’s architecture
    - Example: Version 1.1 - do it this way
    - Version 1.2 - Nope, lets do this the other way
    - Version 1.3 - Uh, we were right the first time
  - Multiple people saying "This is the way"

Wait a Tick - What about Phoenix?

- Prior to 1.7, Phoenix had an explict View module
- Phoenix 1.7 *removes* the view directory, but the concept of a view remains the same