How Ruby on Rails Works: (all notes taken from Professors Armando Fox and Dave Patterson's Saas Text Book Summer 2013):

WEBrick is a mini-server that has just enough functionality to let 1 user at a time interact with your web app client running in the browser. A real web site would use a production server (with production level clients – millions of people) which can either be deployed on 100's of computers serving many copies of the same site to millions of users.

SaaS – Software as a Service – are client server systems built to operate using the open standards of the world wide web. The Fox and Patterson book and part 1 of the SaaS class I took in summer 2013, focused on server-centric apps, rather than building complex in-browser client apps (such as Google Docs).

Note that things like BitTorrent are peer-to-peer protocols, in that every participant is both a client and a server, and anyone can ask anyone else for information. Fox and Patterson said that such a system where a single program must behave as both client and server, it's harder to specialize the program to do either job really well.

The Web Browser is the “Universal Client” and web apps use open standards that anyone can implement royalty free, in contrast to proprietary standards used by older client-server apps.

The primary difference between a client and a server in SaaS is that the Saas client is optimized for allowing the user to interact with information, and the Saas server is optimized for serving many clients simultaneously.

Network Protocol: a set of communication rules on which agents participating in a network agree – in this case – the agents are the web clients (like Firefox, any browser) and web servers (like WebBrick or Apache). The browsers and web servers communicate using the HTTP (hyper-text-transfer-protocol). HTTP relies on TCP/IP (Transmission Control Protocol) / (Internet Protocol) which allows a pair of agents to communicate with an ordered sequence of bytes or arbitrary character strings. In the TCP/IP network, each computer has an IP address of 4 bytes (IPv4) or 16 bytes (IPv6). The Domain Name Service (DNS) is used if you do not want to use the 4 byte number IP address. The browser clients automatically contact the DNS server to look up the site name typed in the address bar. For our debug situation in this class, we ran the rails server (WEBrick server) on our local computer, so we used “localhost” so that the browser client knew that the server it was contacting was running locally on our own computer, rather than having to contact it over the internet. Each network interface device has its own IP address. Note that the difference between HTTP and HTTPS is that HTTPS is the secure version of HTTP in that it uses public key cryptography to encrypt (encode) communication between HTTP clients and servers, so that an eavesdropper only sees gibberish. Our WEBrick rails mini-server does not use HTTPS.

The port number :3000 after localhost represents a port number where multiple agents (clients, servers) on a network can be running at the same IP address.

TCP/IP uses port numbers from 1 to 65535 to distinguish different network agents at the same IP address. All protocols based on TCP/IP including HTTP must specify both the host and the port number when opening a connection.

Example: localhost:3000/movies

locahost = local computer instead of remote host

3000 = server is listening on port 3000, waiting for browsers to contact it

Note: if port number not specified, defaults to port 80 for HTTP

IANA (Internet Assignment Numbers Authority) assigns default port numbers for various protocols and manages the top-level or “root” zone of DNS.

URL = Uniform Resource Locator

URI = Uniform Resource Identifier

URL and URI: begin with **communication scheme** by which information should be retrieved

followed by **hostname**

optional **port number**

and a **resource** on host user wants to retrieve - anything that can be delivered to the browser (image, list of movies in HTML format, form submission that creates a new movie).

Each Saas App: has its *own rules* for *interpreting* the *resource* *name -* **“REST”** - a proposal of SaaS app rules that strive for simplicity and consistency in resource naming across different SaaS apps.

Example HTTP Requests:

GET [http://srch.com:80/main/search?q=cloud&lang=en#to](http://srch.com:80/main/search?q=cloud&lang=en" \l "top)p

GET = HTTP method

http = scheme

srch.com = hostname

80 = port number

/main/search = resource path

?q=cloud&lang=en ----- these are the query terms: “key = value” separated by and or colon

#top = fragment

POST <http://localhost:3000/movies/3>

POST = HTTP method

http = scheme

localhost = host name

3000 = port number

/movies/3 = resource path

HTTP = stateless protocol which means every request is independent of every previous request

Applications keep track of state (such as have you logged in yet?, what step in the check out process are you on?) and must have its own mechanisms for keeping track of this state, since HTTP protocol is stateless and does not do it.

Cookies associate a particular browser client user with information held at the server corresponding to that user's session, but it's the browser's responsibility, not HTTP's or the Saas apps responsibility, to make sure the right cookies are included with each HTTP request.

Stateless protocols simplify server design at the expense of application design.

Rails framework shields you from this complexity.

Cookies are used to establish that 2 independent requests actually originated from the same user's browser, and can therefore be thought of as part of a session.

During the first site visit, the server includes a long string (up to 4 kbytes) with set-cookie: “HTTP response header”. The brower's responsibility (client) is to include this string with the cookie: “HTTP request header” on subsequent requests to that site.

The cookie string has enough information in it for the server to associate the request with the same user\_session.

For our Saas assignments, the RottenPotatoes Rails Server activates WEBrick, and this program waits for clients to make a request and then provides a reply. WEBrick waits to be contacted by the client web browser such as FireFox. It routes the browser's requests to the RottenPotatoes application. The web client FireFox requests RottenPotatoes home page from a web server (WEBrick). WEBrick obtains content from the RottenPotatoes app and sends the content back to FireFox. FireFox displays the content and closes the HTTP connection.

The web is primarily “client pull”, client server architecture because the client initiates all interactions – HTTP servers can only wait for clients to contact them. Because HTTP was designed as a request-reply protocol, only clients can initiate anything.

The evolving standard (including web sockets and HTML5) have some support for allowing the server to push updated content to the client. In contrast, true server push architectures such as text messaging on cell phones, allow the server to initiate a connection to the client to “wake it up” when the new information is available, but these cannot use HTTP. Early web criticisms was that the web's architecture was pure request-reply protocol, ruling out such push based applications. In practice, high efficiency server software supports creating web pages that frequently poll (check in with) the server to receive updates, giving the user the illusion of a push-based application even without the features proposed in websockets and HTML5.

The manner in which HTML (Hyper-Text-Markup-Language) elements correspond to different things on a web page served by a Saas app is as follows. On the Web Developer tool bar, a FireFox add-on, “Make Visible Source” of: <http://localhost:3000/movies> followed by “View Source” shows the raw bytes that come back from the server when this page is requested.

The mark-up language is how the text fits into the page structure. The tags are mark-up of the text content. So for the example web page above:

<!DOCTYPE html>

<html>

<head>

<title>Rotten Potatoes!<\title>

<link href=”/assets/application.ces?body=1” media = “screen” rel=”stylesheet” type=”text/css”/>

<script src=”/assets/jquery.js?body=1” type = “text/javascript”></script>

<script src=”/assets/jquery ujs.js?body=1” type = “text/javascript”></script>

<script src=”/assets/application.js?body=1” type = “text/javascript”></script>

<meta content=”authenticity\_token” name=”cerf-param”/>

<meta content=”pc/1HZ5CSVqAtgEgmiHzFAH6HzEYIynONcZ7kUKP4=” name = “cerf-token”/>

<\head>

<body>

<h1 class='title'>Rotten Potatoes!/h1>

<div id='main'>

<h1>All Movies</h1>

<table id='movies'>

<thead>

<tr>

<th>Movie Title</th>

<th>Rating</th>

<th>Release Date</th>

<th>More Info</th>

</tr>

</thead>

<tbody>

<tr>

<td>Inception</td>

<td>PG-13</td>

<td>2010-07-16 00:00:00 UTC</td>

<td><a href=”/movies/4”>More About Inception</a></td>

</tr>

<tr>

<td>It's Complicated</td>

<td>R</td>

<td>2009-12-25 00:00:00 UTC</td>

<td><a href=”/movies/5”>More About It's Complicated</a></td>

</tr>

</tbody>

</table>

<a href = “/movies/new”> Add New Movie”</a>

</div>

</body>

</html>

How the Ruby app code corresponds to the HTML page:

In the above HTML page, “Rotten Potatoes!” is an HTML element with </title> which is not a self-closing tag. The <link href=..../> is an example of a self-closing tag. In Rails, tags can be nested and Rails automatically indents so it is not required. In the above HTML page, the nested tags are the nested table tags, table rows <tr>, table headers <th> for the table's column headers, and the nested table entries that fill that table up with the <tr> table row with <td> table data, that contains the strings to enter for each of those column header attributes for this table. <th> is a table head element and <tr> is a table row element. The <a href... element is a hyper-link, an attribute, and it's content is what the user sees. h1.title in the ruby app code is an element whose class element has a specific value and corresponds in the <body> section to <h1 class = 'title'>. In the <div id='main'> tag, div#main corresponds to this in the ruby app code. <h1> is a header element. <table id='movies'> is a table element which has an id value = 'movies', and id is the id attribute. In the ruby app code, table#movies corresponds to the tag <table id='movies'>, an element whose id is movies.

HTML (Hyper-text markup language): a markup language combines text with mark-up (annotations about text) in a way that makes it easy to syntactically distinguish between the two.

<p> <\p> - open tag, end tag

<br> <\> - open tag, self-closing tag

<a href = “http://....”> - opening tag with attribute

<brclear = “both”/> - line break that clears both left and right margins

The use of angle brackets for tags comes from SGML (Standardized Generalized Markup Language), a codified standardization of IBM's 1960's General Markup language used for encoding computer readable project documents.

HTML5 – the current version when I took this course in summer 2013 – is a confusing mess of terminology surronding the lineage of HTML. It includes features from both of its predecessors (HTML Version 1-4) and XHTML (eXtended HTML) which is a subset of XML, to both represent data and to describe other markup languages.

XML is a common data representation for exchanging information between 2 services in a service-oriented-architecture. This is used when we use Rotten Potatoes to retrieve movie information from a separate movie database service.

Note that not all browsers support all versions of HTML, and the text book we used for this class in summer 2013 used HTML5 except when otherwise noted.

The tags of particular interest to us in this class were:

HTML: id, class: figure heavily into connecting HTML structure of a page with its visual appearance, text and graphics.

Inspecting the ID and Class Attributes:

CSS uses div#name to indicate a div element whose id is name and div.name to indicate a div element with class name.

Objective: in a web browser, if you hover the mouse over the element with the “Display Element Info.” turned on in the Web Development Bar, you can see whether that element has class or id attribute attached to it. In Web Developer Information, use “Make Visible” and “View Source” so that bytes returned from the server can be viewed for browser to format and display. When you also select “Display element information”, Firefox draws a red box around the element, and hovering the mouse over these can show you the nesting of elements in the Web Development Bar.

Such as: <th>nesting: html> body> div#main > table#movies > thead > tr > th

would be what you see when you hovered the mouse over Movie Title, shown in the above HTML code, table header section of the web page.

Note: Only one element in an HTML document can have a given ID, where as many elements (even of different tag types) can share the same class.

All 3 aspects of an element, it's tag type, id (if it has one), and its class attributes (if it has any), can be used to identify an element as a candidate for visual formatting.

The Cascading-Style-Sheet (CSS) stylesheet we used gives more information about attributes. It allows us to associate “visual styling” instructions with HTML elements by using the elements' classes and ID's. The 4 basic mechanisms by which a selector in a CSS file can match an HTML element: tag name, class, ID, hierarchy. If multiple selectors match a given element, the rules for which properties to apply are complex, so most designers try to avoid such ambiguities by keeping CSS simple.

To see a web page without CSS styling, select in the Web Developer Tool Bar:

CSS > Disable styles > All Styles

and the web page will be displayed with all CSS formatting turned off, and shows the extent CSS can separate visual appearance from logical structure.

CSS Selectors: identify elements to be styled:

Selector: What is Selected:

h1 any h1 element

div#message the div element whose ID is message

\_\_\_.red any element with class red

div.red, h1 div element with class red, or any h1

div#message h1 An h1 element that's a child of (inside of) div#message

a.lnk an element with class lnk

a.lnk:hover an element with class lnk, when hovered over

CSS Attributes: a few of the many attributes for elements:

Attribute: Example Values:

font-family “Times, serif”

font-weight bold

font-size 14pt, 125%, 12px

font-style italic

color black

margin 4px

background color red, #c2eed6 (RGB values)

border 1px solid blue

CSS Attributes Continued from above: a few of the many attributes for elements:

Attribute: Example Values:

text-align right

text-decoration underline

vertical-align middle

padding 1cm

In the above HTML page: In the Firefox browser Web Developer Menu, select “View Source” so source code is fed to us and “Information”->”Display Element Infomation” and move the window element menu out of the way. You can now hover over the web page element and see the element type tag, whether it has an id or class or attribute.

<link href=”/assets/application.ces?body=1” media = “screen” rel=”stylesheet” type=”text/css”/>

.title means an h1.title whose *class* attribute has the value title (when you hover the mouse on the web page, you see: html> body> h1.title)

If you hover the mouse over “Movie Title” on the web page you see:

table#movies is a table whose ID attribute has the value movies:

(html> body > div#main > table#movies > thead> tr> th)

How CSS uses this information to style a web page:

In the above <link href= HTML element, rel=stylesheet says to load stylesheet at href directory URL pointer, and use that information to control visual display of page. If you click on that URI / URL, it takes you to that stylesheet.

A CSS Style sheet from our above HTML web page example:

html, body {

margin : 0;

padding : 0;

background : white;

color : DarkSlateGrey;

font-family : Tahoma, Verdana, sans-serif;

font-size : 10 pt;

}

div#main {

margin : 0;

padding : 0 20px 20px;

}

a {

background : transparent;

color : maroon;

text-decoration : underline;

font-weight : bold;

}

h1 {

color : maroon;

font-size : 150%;

font-style : italic;

display : block;

width : 100%;

border-bottom : lpx solid DarkSlateGrey;

}

h1.title {

margin : 0 0 len;

padding : 10px;

background-color : orange;

color : white;

border-bottom : 4px solid gold;

font-size : 2cm

font-style : normal;

}

table#movies {

margin : 10px;

border-collapse : collapse;

width : 100%;

border-bottom : 2px solid black;

}

table#movies th {

border : 2px solid white;

font-weight : bold;

background-color : wheat;

}

table#movies th, table#movies td {

padding : 4px;

text-align : left;

}

#notice #warning {

background : rosybrown;

margin : 1cm 0;

padding : 4px;

}

form label {

display : block;

line-height : 25 px;

font-weight : bold;

color : maroon;

}

In the above HTML web page and the associated CSS stylesheet shown above:

“All Movies” on the web page will match all h1's in the document in CSS stylesheet.

h1 {

color : maroon;

font-size : 150%;

font-style : italic;

display : block;

width : 100%;

border-bottom : lpx solid DarkSlateGrey;

}

To make match more specific in other ways:

html > body> div main –demarcates a logical part of the page

One affect: unlike the “Rotten Potatoes” header that extends to the edge of a page, the main body of page has 20 pixels of padding on either side with this extra in the CSS stylesheet:

This is an example of how to use an ID attribute to make match more specific:

div#main {

margin : 0;

padding : 0 20px 20px;

}

<body>

<h1 class='title'>Rotten Potatoes!/h1>

<div id='main'>

<h1>All Movies</h1>

An example of how to use a class attribute to make match more specific:

h1.title in CSS stylesheet has different color and styling:

h1.title {

margin : 0 0 len;

padding : 10px;

background-color : orange;

color : white;

border-bottom : 4px solid gold;

font-size : 2cm

font-style : normal;

}

These are examples of nesting.

table#movies th, table#movies td {

padding : 4px;

text-align : left;

}

#notice #warning {

background : rosybrown;

margin : 1cm 0;

padding : 4px;

}

form label {

display : block;

line-height : 25 px;

font-weight : bold;

color : maroon;

}

Movie details match the table#movies th, table#movies td in web page.

More details of How Saas Works for our class environment:

The web client (FireFox browser for example) requests Rotten Potatoes home page from a web server (our class used WEBrick). FireFox constructs the HTTP request using the URI: [http://localhost:3000](http://localhost:3000/) to contact an HTTP server (WEBrick). WEBrick is listening on port 3000 (in test case conditions it was running on the same computer as client FireFox, which is why localhost was used in the HTTP request). WEBrick receives the HTTP request for the resource '\movies' (the list of all movies in Rotten Potatoes).

WEBrick obtains content from the Rotten Potatoes app and sends this content back to FireFox. WEBrick does this by using the Rack Middleware, written in Ruby, and using this it calls the Rotten Potatoes code in the app tier (within the 3-tier architecture of Saas). The Rotten Potatoes app generates the page content using movie info. stored in the persistence tier, which was implemented by a SQLite database using local files.

Rack supports 1 or more web app frameworks that simplify creation of a particular class of web apps in a particular language. Rack routes the request to the index action of the movies controller. The resource named by this route is the list of movies. The Ruby function implementing the index action in the movies controller asks the movie model for a list of the movies and associated attributes. If successful, the controller identifies a View that contains the HTML mark-up for presenting the list of movies, and passes it the movie information so that an HTML page can be constructed. If it fails, the controller identifies a View that displays an Error message. Rack passes the constructed View to WEBrick, which sends it back to FireFox as the HTTP reply.

WEBrick returns the content encoded in HTML, again using HTTP. The HTML may contain references to other kinds of media such as images to embed in the display page. The HTML may also contain a reference to a CSS stylesheet containing formatting information describing the desired visual attributes of the page (font sizes, colors, layout, and so on).

FireFox displays the content and closes the HTTP connection. Firefox fetches any referenced assets (CSS, images, and so on) by repeating the above steps as needed but providing the URI's of the desired assets as referenced in the HTML page. FireFox displays the page according to the CSS formatting directives and including any referenced assets such as embedded images.

The **3-tier architecture** and Horizontal Scaling:

**Presentation Tier:** usually consists of the HTTP server (web server) which accepts the requests from the outside world (i.e., users) and usually serves static assets (our class web server was WEBrick). Note that application servers sit between the web server (presentation tier, WEBrick in our case) and the actual app code and are sometimes referred to as middleware (Rack, in our case).

The application server's job is to hide low-level mechanics of an HTTP from the app writer. An example would be when an app server can route incoming HTTP requests directly to appropriate pieces of code in the app., saving you from having to listen for and parse incoming HTTP requests.

Modern application servers support one or more web application frameworks that simplify creation of a particular class of web applications in a particular language. Saas uses the Rails framework and Rack Application Server, which comes with Rails. WEBrick is used for the Saas class only as the server and speaks to Rack directly. Other web servers such as Apache require additional software modules to do so.

If your app were written in PHP, Python or Java, you would use an app server that handles code written in those languages. The Google app engine runs Python and Java apps, has appropriate middleware that bridges your app's Python or Java code to the Google operated infrastructure that faces the outside world.

HTTP is stateless and app data must remain stored across HTTP requests, such as session data, user's login, and profile info. It is stored in the **Persistence Tier.** Popular choices are traditionally databases such as MySQL or PostgreSQL, which are open source databases, but prior to those, Oracle and IBM DB2 were popular choices.

Early Saas sites were created using Perl and PHP scripting languages, coinciding with open source OS Linux's success, and MySQL, an open source database.

**LAMP stack** stands for: **L**inux, **A**pache, **M**ySQL and **P**HP or **P**erl or **P**ython.

**Logic Tier:** the web server forwards the logic tier the requests for dynamic content

Low Traffic (low content): software in the Presentation, Logic and Persistence tier might run on one computer. Rotten Potatoes runs this way. WEBrick is our Presentation tier, SQLite is our persistence tier and stores the information on the computer. However, in production, it is more common for each tier to be on multiple computers.

A Typical Web Site:

So the Presentation Tier (Web Server(s), Asset Server)) sits between Load balancers: incoming HTTP requests are directed to one of several available application servers through the load balancer to handle dynamic content generation, allowing computers to be added or removed from each tier as needed to handle demand. The Presentation Tier web and asset servers render the views and interact with the User. The Logic Tier App servers run the Saas app code. The Saas Apps run by the Logic Tier App Server go through a load balancer to the Persistence Tier, a database master of 1 to n database slaves which store the app data.

Making the Persistence Tier “shared nothing” is much more complicated since it postpones the scaling problem rather than solves it. It is a master-slave approach which is used when the database is read more than it is written. The slaves perform the reads and the master is the only one who can perform writes. The Master updates the slaves with the results of writes as quickly as possible.

Heroku is an “app deployer” which we used in the Saas class. Adam Wiggins wrote it. It can use caching, sharding and other techniques to take the load off the database. In actuality though the load is NOT taken off the database. SQL databases are fundamentally non-scalable. There is no magic pixie dust we can sprinkle on them to make them scale.

A “shared nothing” architecture is called this because entities within a tier generally do not communicate with each other, allowing adding computers to each tier independent to match demand.

“Load balancers” distribute workload evenly, either through hardware appliances or specially configured web servers.

“HTTP statelessness” makes “shared nothing” possible, since all requests are independent, and any server in the Presentation or Logic Tier can be assigned to any request.

However—scaling the persistence tier is much more challenging than the Presentation or Logic tiers.

With cloud computing, computers can be added to each tier (Presentation or Logic tiers) as demand requires. Depending on the scale (size) of the deployment, more than 1 tier may be hosted on a single computer, or a single tier may require many computers.

The Reason for Databases:

Early web apps sometimes manipulated files directly for storing data and there are 2 reasons why databases overwhelmingly took over this role very early. Databases historically provided high durability for stored information. System crashes and transient data corruption would not cause data loss. This was especially important for web apps storing million's of user's data.

Databases store information in structured format. Relational databases are the most popular since each object is stored in a table whose rows represent object instances and whose columns represent object properties. This organization is a good fit for structured data that many web apps manipulate.

Note: Web apps such as FaceBook have grown so far beyond the scale for which relational databases were designed that they are being forced to look at alternatives to the long-reigning relational database.

Model-View-Controller Architecture:

Models: are concerned with the data manipulated by the application: how to store it, how to operate on it and how to change it. MVC apps typically has a model for each type of entity manipulated by the app. For this class, the Rotten Potatoes app, there is only a movie model. These models deal with the app's data and contain the code that communicates with the storage tier.

Views: Are presented to the user and contain information about the models with which users can interact. They serve as the interface between the system's users and its data. Rotten potatoes can list movies by clicking on links or buttons in the views. There is only one kind of model in Rotten Potatoes, the movie model, but it is associated with a variety of views. One view lists all movies, another view shows the details of a particular movie, and other views appear when creating new movies or editing existing ones.

Controllers: mediate the interaction in both directions – when the user interacts with a view by clicking on something on a web page, a specific controller action corresponding to that user activity is invoked. Each controller corresponds to 1 model. In Rails, each controller action is handled by a particular Ruby method within that controller. The controller can ask the model to retrieve or modify information. Depending on the results of doing this, the controller decides what view will be presented next to the user and supplies that view with any necessary information. Since Rotten Potatoes has only one model (movies), it also has one controller, the movies controller. Actions defined in that controller handle each type of user interaction with any movie view, (clicking on links or buttons for example), and contain necessary logic to obtain model data to render any of the movie views.

Saas Apps: have always been View-centric and have always relied on a persistence tier to store app data in a database. Rails choice of MVC as the underlying architecture seems like an obvious fit, but other choices are possible, such as “Martin Fowler's catalog of patterns of Enterprise Application Architectures.”

Static Content Apps: mostly have static content with a small amount of dynamically generated content, such as for a weather web site.

Page Controller Patterns: are used for an app that is easily structured as a small number of distinct pages – effectively giving each page its own simple controller that only knows how to generate that page.

Front Controller Patterns: are used by J2EE servlets, or an app that might take a user through a sequence of pages (for example, signing up for a mailing list), but has few models. A single controller handles all incoming requests rather than separate controllers handling requests for each model. The one controller relies on methods in a variety of models to generate one of a collection of views.

Template View Pattern: is used by PHP and emphasizes building the app around views, with logic in the models generating dynamic content in place of part of the views—the controller is implicit in the framework.

Model-View-Controller: used by Rails and Java Spring – associates a controller and a set of views with each model type. The MVC design pattern distinguishes models that implement business logic. Views present information to the user to interact with the apps. Controllers mediate the interaction between views and models.

MVC Saas Apps: every user action that can be performed on a web page: clicking a link button, submitting a fill-in-form, using drag and drop. Every user action is eventually handled by some controller action which will consult model(s) as needed to obtain information and generate a view in response.

MVC apps are appropriate for interactive Saas apps with a variety of model types, where it makes sense to situate controllers and views along with each type of model. Other architectural patterns may be more appropriate for smaller apps with fewer models, or a smaller repertoire of operations.

The 3-tier Architecture:

Models – logic and persistence tiers

Controllers – logic and presentation tiers

Views – logic and presentation tiers

Active Record for Models: Apps need to store and manipulate persistent data (non-trivial) – in a database, as a plain file or other persistent storage location. A way is needed to convert between the data structures or objects manipulated by the app code and the way the data is stored.

For Rotten Potatoes – the only persistent data is about movies – movie attributes such as: title, release date, MPAA Rating, short “blurb” summary.

There are different issues with storing attributes: attributes are separated by commas in a file run into issues when the movie title has a comma, for example. It cannot be fixed by double quoting, since some titles have double quotes within them.

Examples show that it is tricky to do this and requires writing code to convert an in-memory object to our storage representation (called marshalling or serializing the object) and vice versa.

Structured storage are storage systems that allow you to simplify the desired structure of stored objects rather than writing explicit code to create structure, and in some cases specify relationships connecting objects of different types.

Relational databases (RDBMS's) started in the early 1970's and are an elegant structured storage system whose design was based on formalism for representing structure and relationships. It stores collections of tables, each of which stores entities with common sets of attributes. 1 row = 1 entity, columns = attributes for entity, and the “id” column = primary key, permanent and unique.

Rails Primary Keys: use common convention of assigning integers in increasing order.

Models: have the responsibility to manage app's data and some correspondance must be established between operations on model object in memory (movie object) and how it is represented by and manipulated in the storage tier.

Rails: uses Active Record Architectural Pattern. This pattern is a single instance of a model class (one movie for this class) that corresponds to a single row in the RDBMS table.

The model object has built-in **behaviors** that directly operate on the database representation of the object. These behaviors have the acronym **CRUD**. **C** is for “**C**reating a new row in table, representing the object”. **R** is for “**R**ead an existing row into a single instance.” **U** is for “**U**pdate an existing row with new attribute values from a modified object instance.” **D** is for “**D**elete a row (destroying the object's data forever).”

Adding an ability for movie goers to review their favorite movies will create a one-to-many relationship, or association between a moviegoer and her reviews.

Active Record exploits existing RDBMS mechanisms to make it easy to implement associations on the in-memory objects.

In regard to Routes, Controllers, and the REST principle, recall the MVC Pattern's Controller actions mediate the user's web browser interactions that cause CRUD requests, and in Rails, each controller action is handled by a particular Ruby method in a controller file.

The way Ruby on Rails works is: Each incoming HTTP request must be mapped to the appropriate controller method and this mapping is called a route. An HTTP request has both a URL (URI) and an HTTP Method. An HTTP method is either a function or the HTTP method of a request. For Ruby on Rails that uses a routes file to map the HTTP request to the appropriate controller method, method means HTTP verb associated with request (GET, POST, PUT, DELETE) and the controller action to the application code method or function that handles the request.

This “service-oriented-architecture” is a consistent way to map requests to actions. Roy Fielding's idea was to call it “**REST**” for “**Re**presentational **S**tate **T**ransfer”, which identified various entities manipulated by a web app as resources. It designs routes so any HTTP request would contain all information necessary to identify both a particular resource and the action to be performed on it.

So REST was a powerful organizing principle for Saas apps. The app designer must think carefully about exact conditions or assumptions each request depends on in order to be self-contained. They also had to decide how each type of entity manipulated by the app can be represented as a “resource” on which various operations can be performed. Apps designed with this guideline are said to expose RESTful APIs (Application Programming Interfaces). URLs (URIs) that map to particular actions are said to be RESTful APIs.

Rails' route mappings are generated by code in the file routes.rb. Rails does NOT mandate routes to be RESTful, but it's built-in support for routing assumes REST by default.

“Rake” runs maintenance tasks defined in RottenPotatoes' Rakefile.rake.

Example: route GET /movies/8

would for example, match 2nd row where :id = 8, the primary key of a model instance movie.

Example: route GET /movies

would for example match the first row, requesting a list of all the movies, an Index action.

Example: route POST /movies

would for example match 4rth row and create a new movie entry in the database. POST /movies route does not specify an 'id' because a new moview won't have an id until after it's created.

Index and Create actions have the same URL (URI) but different HTTP methods (GET, POST) which makes them distinct routes.

Summary of Rake Routes: show routes recognized by RottenPotatoes and the CRUD action reprsented by each route.

Operation on Resource Rails Controller Action (CRUD Method and URL (URI)

action represented by route) (when request matches URL and

HTTP method)

Index (list) movies Index GET /movies

Read (show) existing movie Show GET /movie/:id

Display fill-in form for New GET /movies/new

new movie

Create new movie from Create POST /movies

filled in form

Display form to edit Edit GET /movies/:id/edit

existing movie

Update movie from Update PUT /movies/:id

fill-in form

Destroy existing movie Destroy DELETE /movies/:id

This RESTful interface simplifies SOA (Systems-Oriented-Architecture) participation. If every request is self-contained, interactions between services don't need to establish an on-going session, as many web apps do when interacting with a web browser.

Modern practice suggests: when creating a user-facing Saas app designed to be used via a browser, you should think of the app primarily as a collection of resources accessible via RESTful API's that happen to be accessed via a web browser. This presents a minor problem, since routes make use of 4 different HTTP methods (GET, POST, PUT and DELETE), using different methods to distinguish routes for the same URI/URL.

Historically: web browsers only implemented GET for following a link and POST for submitting forms. Most browsers also implement HEAD which requests metadata about a resource.

To compensate: Rails routing mechanisms lets browsers use POST for requests that normally would require PUT or DELETE. Rails annotates web forms associated with such requests so that when the request is submitted, Rails can recognize it as special and can internally change the HTTP method “seen” by controller to PUT or DELETE as appropriate.

The advantage: The same set of routes and controller methods can be used to handle either requests coming from a browser (a human being, a client) and requests coming from another service in the SOA (service-oriented-architecture).

CREATE and UPDATE Rails Controller Actions (with Rotten Potatoes): Creating a new movie: There are 2 interactions with Rotten Potatoes MVC app. The reason is because before a user can submit information about the movie, he/she must be presented with a form in which to enter the information. The empty form is therefore the resource named by the route: GET /movies/new (new action to create a new movie – so empty form is the resource named by this route).

If you look at the page source of the Rotten Potatoes front page of the web app, you can scroll down to where the link is located in the HTML code: <a href = “ /movies/new”>Add new movie</a> to see the new action to create the new movie. On the web page, if you click the button “Add new movie”, this link click always triggers the GET action for HTTP.

If you type “rake routes” to see the table, you would see:

movies GET /movies (.:format) {:action => “index”, :controller => “movies”}

POST /movies (.:format) {:action=> “create”, :controller => “movies”}

new\_movie GET /movies/new (.:format) {:action=> “new”, :controller => “movies”}

edit\_movie GET /movies/:id/edit (.:format) {:action=>”edit”, :controller => “movies”}

movie GET /movies/:id (.:format) {:action=>”show”, :controller => “movies”}

PUT /movies/:id (.:format) {:action=>”update”, :controller => “movies”}

DELETE /movies/:id (.:format) {:action=>”destroy”, :controller => “movies”}

So clicking the “Add new movie” button gives the empty form, since it triggers in the above routes table, the “new\_movie GET” HTTP method.

In the HTML code, you see:

<form accept-charset = “UTF-8”

action = “/movies” method = “POST”

<div style = “margin : 0 : padding : 0;

display : inline”>

So that when you “Save form”, it knows to post the form at URI/URL: /movies.

This triggers the CREATE action (in the routes table) which then creates the actual movie.

This is analogous to the EDIT button, which triggers the route table “edit\_movie GET” action, and gives you an existing form to edit (rather than a new empty form), and then you click “Update Info”, which triggers the route table PUT action to “update”.

This shows: that we think of a web application as a collection of resources accessible via RESTful APIs, that happen to be accessed by a web browser, which only implements GET (for following links) and POST (for submitting forms), yet HTTP methods have 4 methods: GET, POST, PUT, DELETE.

Advantage: The same set of routes and controller methods can be used to handle either requests form the browser (human) or some other service in a SOA (service-oriented-architecture).

A user-facing web app: has 2 interactions required to create a new movie.

A SOA (service-oriented-architecture): remote service can create a single request containing all information needed to create the new movie, so it would neer need to use new route.

Comparing a NON-RESTful API with a RESTful API:

NON-RESTful site: RESTful site:

(3) Login to site:

POST /login/dave POST /login/dave

(4) Welcome page:

GET /welcome GET /user/301/welcome

(5) Add item ID 427 to cart:

POST /add/427 POST /user/301/add/427

(6) View cart:

GET /cart GET /user/301/cart

(7) Pay

POST /checkout POST /user/301/checkout

The NON-RESTful site: Every request after login line (3) relies on implicit information. Line (4) assumes site “remembers” who currently is logged in, and user is to show him/her their welcome page. Line (7) assumes site “remembers” who has been adding items to their cart for check-out.

The RESTful site: URI's/URL's contain enough information to satisfy the request without relying on implicit information—since after Dave logs in, his user id 301 is present in every request, and his cart is identified by his user id rather than implicitly based on notion of a currently-logged-in-user.

Non-RESTful Requests and Routes: are those that rely on results of previous requests.

In a SOA (service-oriented-architecture), a client of the RESTful site could immediately request to view cart (line 6), but a client of the NON-RESTful site would first have to perform lines 3-5 to set-up the implicit information on which line 6 depends.

Summary:

Route: consists of HTTP method (GET, POST, PUT, DELETE) and a URI/URL, which may include some parameters.

App. Frameworks (such as Rails): map routes to controller actions.

An app. Designed in accordance with REST (Representational State Transfer): can be seen from outside as a collection of entities on which specific operations can be performed, with each operation having a corresponding RESTful request that includes all the information necessary to complete the action.

RESTful Routes and Resources: the same controller logic can usually serve user-facing pages via a web browser or requests arriving from a SOA (service-oriented-architecture).

Web browsers: support GET and HOST methods

Framework (Rails): logic can compensate so that programmer can work under assumption that all HTTP methods (GET, POST, PUT, DELETE) are available.

REST verses SOAP verses WS-\*

Late 1990's: standards bodies created committees to develop standards for SOA(service-oriented-architecture) interoperation.

Web Services (WS) was one approach: a collection of elaborate protocols: WS-Discovery, WS-Description, etc, WS-\*, for jokes, WS-Deathstar-by David Heinemeier Hansson, creater of Rails

Competing Standard: SOAP (Simple Object Access Protocol) – simpler but more complex than REST

Practicing Developers: perceived SOAP and WS-\* as overdesigned committee-driven standards burdened by the archaic design of enterprise-based interoperation standards such as CORBA and DCOM which preceded them.

In contrast, REST: dramatically simpler and is more of a philosophy than a standard – so appealed to developers – and the majority of SOA apps are built this way today.

Optional :format in routes:

Raw output of “rake routes”: includes a token (.:format) in most routes – if present – format specifier allows a route to request resources in an output format other than default of HTML – for example:

GET /movies.xml

would request a list of all movies as an XML document rather than an HTML page.

Rotten Potatoes app: simple app does not include code to generate formats other than HTML – mechanism allows a properly designed existing app to be easily integrated into a SOA – by changing a few lines of code – allows all existing controller actions to become part of an external RESTful API.

Template Views: an architecture where at run time, variable values or results of code execution are substituted or interpolated into a page – user facing Saas apps primarily deliver HTML pages – so most frameworks provide a way to create a page of static mark-up (HTML or otherwise) interspersed with variables or very brief snippets of code.

The template view is the basis of many Saas frameworks including: Rails, Django and PHP.

The authors of my Saas course text book (Fox and Patterson) perfer HAML's conciseness to Rails built-in erb templating system, so HAML is pre-installed with the code given in this class to use. An “erb” is an object that works by building a chunk of Ruby code that will output the completed template when run. If safe-level is set to a non-nil value, ERB code will be run in a separate thread with $SAFE set to the provided level.

In trim-mode, if a string is passed containing 1 or more of the following modifiers, ERB will adjust its code generation as listed:

% } ruby processing enalbled for lines beginning with %

<> } omit newline for lines starting with <% and ending in %>

> } omit newline for lines ending in %>

eoutvar } can be used to set name of variable ERB will build up its output in – useful when you need to run multiple ERB templates through the same binding and/or when you want to control where output ends up. Pass the name of the variable to be used inside a string.

new (str, safe\_level = nil, trim\_mode = nil, eoutvar = '\_erbout') – constructs a new ERB object with the template specified in str.

HAML (that Fox and Patterson's book uses): uses templating system called HAML (for HTML Abstraction Markup Language, pronounced HAM-ell) to streamline the creation of HTML template views.

To visit the “moving parts” of a Rails App – open: /app/views/movies/index.html.haml in the Rotten Potatoes directory. This is the view used by the Index Controller action on movies; by convention over configuration – suffixes .html.haml indicate view should be processed using HAML to create index.html, and the location and name of the file identify it as the view for the index action in the movies controller.

Interpolation into Views Using HAML:

--View the source of the Rotten Potatoes web page – to compare the HTML of Rotten Potatoes to HAML – which is actually part of an application:

#files is app/views/movies/index.html.haml

%h1 All Movies

%table#movies

%thead

%tr

%th Movie Title

%th Rating

%th Release Date

%th More Info.

%tbody

[-@movies.each](mailto:-@movies.each) do |movie| -->result thrown away

%tr

%td = movie.title → result put in HTML doc

%td = movie.rating

%td = movie.release\_date

%td = link\_to “more about #{movie.title}”, movie\_path(movie)

= link\_to 'Add new movie', new\_movie\_path

HTML:

<!DOCTYPE html>

<html>

<head>

<title>Rotten Potatoes!</title>

<link href=”/assets/application.css?body=1” media=”screen” />

<script src=”/assets/jquery.js?body=1” type=”text/javascript” />

<script src=”/assets/jquery\_ujs.js?body=1” type=”text/javascript” />

<script src=”/assets/application.js?body=1” type=”text/javascript” />

<meta content=”authenticity\_token” name=”cerf-param”/>

<meta content=”pc/xxxxxx............................................”/>

</head>

<body>

<h1 class='title'>Rotten Potatoes!</h1>

<div id='main'>

<h1>All Movies</h1>

<table id='movies'>

<thead>

<tr>

<th>Movie Title</th>

<th>Rating</th>

<th>Release Date</th>

<th>More Info.</th>

</tr>

</thead>

<tbody>

<tr>

<td>Inception</td>

<td>PG-13</td>

<td>2010-07-16 00:00:00 UTC</td>

<td><a href=”/movies/4”>More about Inception</a></td>

</tr>

<tr>

<td>It's complicated</td>

<td>R</td>

<td>2009-12-25 00:00:00 UTC</td>

<td><a href=”/movies/5”>More about It's Complicated</a></td>

</tr>

</tbody>

</table>

<a href=”/movies/new”>Add new movie</a>

</div>

</body>

</html>

HAML: saves clutter space in code since it does not require end tags as HTML does. Instead, it nests and indents (with a scheme similar to Python) to show elements used.

**Element ID in HAML: (in bold)**

%table**#movies** → chosen to look like CSS

(could also use .string to indicate class of element)

Interpreting results in HAML code:

If a line starts with a hypen → HAML will throw result away (see above example)

NOTE: For the line: [-@movies.each](mailto:-@movies.each) do |movie|

-->there is no output on that line anyway – since starts a “each.do iteration loop”

If a line starts with an equals sign → HAML will insert the result into the HTML code

HAML Template:

--lines beginning with % expand into the corresponding HTML opening tag, with no closing tag needed, since HAML uses indentation to determine structure.

--Ruby-like hashes following a tag become HTML attributes

--Lines beginning with dash “-” are executed as Ruby code with any result discarded

--Lines beginning with “=” equals sign are executed as Ruby code with the result interpolated into HTML output.

Basics of HAML, Summarized, Compared to HTML:

HAML HTML

%br {:clear =>'left'} <br clear=”left”/>

%p.foo Hello <p class=”foo”>Hello</p>

%p#foo Hello <p id=”foo”>Hello</p>

<div class=”foo”> <div id=”foo” class=”bar”> #foo.bar

(indent)….....(.foo) (indent)….......................

</div> </div>

HAML:

%tags: must either contain tag and all its content on a single line.

--or they must appear by itself on line in which case all of tags content must be indented 2 spaces on subsequent lines.

--note that HAML specifies “class” and “id” attributes using a notation deliberately similar to CSS selectors.

MVC:

Views: should contain as little code as possible

HAML: Technically permits arbitrarily complex Ruby code in a template – it's syntax for including a mult-line piece of code is deliberately awkward—so this discourages programmers from doing so.

The only *computation* in Index view for Rotten Potatoes is limited to iterating over a collection (provided by model via the controller) and generating an HTML table row to display each element.

In contrast: apps written using PHP framework: apps often mingle large amounts of code into view templates – disciplined PHP programmers separate views from code, but PHP framework does not support it or reward it.

MVC advocates: distinguishes controller from view – makes it easier to think first about structuring app as set of RESTful actions – and later about rendering results of actions in separate view step.

SOA: means components of app act as interoperable services, and can be used independently and recombined in other apps.

Contrasting Implementation: “software silo” - rarely has API's to internal components

Cost consideration: If you mis-estimate what the customer wants, cost is lower with SOA than with “siloed” software to recover from a mistake and try something else, or to produce a similar-but-not-identical variant to please a subset of users.

Example: Amazon – started 1995 – with siloed software for online retailing – by 2002 – Steve Yegge (former CEO, founder, former Amazonian, and Jeff Bezos, current CEO) mandated change that would today be called SOA—Broadcast email to all employees:

1. All teams expose data and functionality through service interfaces.

2. Team must communicate with each other through these interfaces.

3. No other form of interprocess communication allowed:

--no direct linking

--no direct reads of another team's data store

--no shared memory model

--no back-doors what-so-ever

\*\*Only communication allowed is via service interface calls over the network.

4. Does not matter technology they use:

--HTTP

--Corba

--Pub/Sub

--Custom Protocols

5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

6. Anyone who does not do this will be fired.

7. Thank you. Have a nice day!

Example: FaceBook 2007 – 3 years after company went online – FaceBook Platform launched – SOA – allowed 3rd party developers to create apps that interact with core features of FaceBook, such as what people like, friends, tagged photos of people, etc.

Example: New York Times – readers could see which articles their friends were reading and liked

Contrasting Example: Google+ - had no API's when launched in 2011 – used software silo – had one heavyweight API 3 months later – following the complete stream of everything a Google+ user sees.

Hypothesized Bookstore Example: create 1st as silo—then as SOA—both contain same 3 subsystems: reviews, user profiles, and buying.

Silo Version: subsystems can get user profile information out of users subsystem – however – all subsystems are inside a single external API (“the bookstore”).

Subsystems: Review Subsystem, User Profile Subsystem, Order and Buying Subsystem – are all inside a single external API (“the bookstore”), and “the bookstore” API interfaces to the human.

SOA Version: all 3 subsystems independent and available via API's

Example:

Reviews Subsystem: User Reviews, Editor Reviews, Review Service

User Subsystem: Users, User Profile Service

Buyer Subsystem: Buyers, Buying Service – interfaces to Credit Card Processing

Bookstore Service: Sits outside the Reviews Subsystem, User Subsystem and Buyer Subsystem, and interfaces to the Human

Bookstore Data Center Boundary: includes the Review Subsystem, User Subsystem, Buyer Subsystem and the Bookstore Service.

Favorite Books Service: sits outside the Bookstore Data Center boundary and also interfaces to a Human on one side, and a Social Network Service on the other side, containing users.

In this example, the reviews subsystem wants information about the user. It cannot just reach directly into the user's database. Instead, it has to ask the user's service via whatever API is provided for that purpose, and a similar restriction is true for buying.

The Bookstore: is one combination of services – it creates new apps and others can recombine them.

Example of recombining services: “My Favorite Books” app: might combine users service and reviews service with a social network, so you can see what your social network friends think about the books you have reviewed.

Critical Distinction of SOA: no service can name or access another service's data – it can only make requests for data through an external API.

NOTE: It does NOT match the traditional layered model of software, in which each higher layer, is built directly from the primitives of the immediately lower layer as in siloed software.

SOA Implies: vertical slices through many layers, and those slices are connected to form a service.

SOA: more work to build compared to siloed service – but payback is tremendous for re-usability. Explicit APIs make testing easier.

SOA Downside: high cost of wading through software stack of network interface – performance hit—also--dependability planning more challenging—since siloed system completely down on failure—SOA must deal with the sticky case of partial failures.

This is why MVC advocates argue distinguishing controller from view makes it easier to think first about structuring app as a set of RESTful actions, and later about rendering results of actions in a separate view step.

Given the case stated for SOA—now clear how the separation of models, views, controllers and adherence to RESTful controller style, naturally leads to an app whose actions are easy to “externalize” as standalone API actions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* notes on the Ruby syntax omitted for now \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Common Rails Application Directory Structure:

Gemfile - list of Ruby gems (libraries) this app uses

Rakefile - commands to automate maintenance and deployment

app - your application (app. Logic)

app/models - model code (app.'s data – storage and manipulation)

app/controllers – controller code (mediates interaction between model data and view)

app/view - view templates (info. about model data with which user can interact, for ex., click)

app/view/layouts – page templates used by all views in the app

app/helpers – helper methods to streamine view templates

app/assets - static assets (Javascript, images, stylesheets)

config - basic configuration information

config/environments – development verses production – start-up instructions for app

config/environments/production.rb – specify production mode options, such as performance optimization, which you would not do in development, or debug time

config/environments/development.rb – specify development and debug mode options

config/environments/test.rb – stub-out external interactions – example being to save outgoing emails to a file rather than send

config/database.yml – database configuration for development verses production – it maintains separate copies of database in the db directory

config/routes.rb – mappings of URIs to controller actions

db - database schema

db/development.sqlite3 – data storage for development database

db/test.sqlite3 – database used for running tests

db/migrate - migrations (descriptions of changes to database schema)

doc - generated documentation

lib - additional app code shared among M, V, C

log - log files

public - error pages served by the web server

script - development tools, not part of app

tmp - temporary data maintained at runtime

gemfile\_edits.rb:

#use Haml for templates

gem 'haml' → specifies using Haml templating system rather than the built-in “erb”

#use Ruby debugger

group : development, :test do

gem 'ruby-debug19' →use interactive ruby debugger ruby-debug19 for development and testing, but not production

end

“run bundle install” --without production

–> this checks if any gems specified in our Gemfile are missing and need installation

→ bundler is an example of “automation for repeatability” – rather than manually installing the gems your app needs – listing them in Gemfile and letting bundler install them automatically ensures that the task can be repeated consistently in a variety of environments, eliminating mistakes in dong such tasks as a possible source of app errors.

\* important because you deploy your app and info. used to make the deployment environment match your development environment

Start app rails server – point browser to URI: http://localhost:3000

localhost = hostname

3000 = port fetches home page – index.html – ours is: public/index.html (generic welcome page for Rails app)

<http://localhost:3000/movies>

→ will get an error for URI unless there is a mapping in “routes” for URI's app methods – to check, run “rake routes”.

log/development.log → see error message