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

Ruby on Rails uses the MVC (Model-View-Controller) design pattern which associates a controller and a set of views with each model type. Models implement business logic. Views present information to the user and allow the user to interact with the app. Controllers mediate the interaction between views and models. Every user action such as clicking a link button, submitting a fill-in form, using drag and drop are eventually handled by some controller action, which will consult models as needed to obtain information and generate a view in response. 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 have a model for each type of entity manipulated by the app.

In the RottenPotatoes app for this class, there is only a movie model but others could be added later. Since models deal with the app's data, they contain 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 it's data. In Rotten Potatoes, you can list movies, add movies by clicking on links or buttons in the views. The one movie model in RottenPotatoes 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 something on a web page, a specific controller action corresponding to that user activity is invoked. Each controller corresponds to one model. 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 RottenPotatoes has only one model (Movies), it also has only one controller, the Movies controller. Actions defined in that controller can 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, which stores data in a database. Rails choice of MVC as underlying architecture seemed an obvious fit to the designers but other choices were possible.

Apps need to store and manipulate persistent data in a database, file or other persistent storage location. They need a way to convert between the data structures or objects manipulated by the app code and the way data is stored. For RottenPotatoes, the only persistent data is about movies. Movie attributes are: title, release date, MPAA rating, short “blurb”, and a summary. There are different issues with storing attributes. Attributes separated by commas in a file run into issues when movie title has a comma and is not fixed by double quoting it, since some titles have a double quote within it. It is tricky 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 systems allow you to simply specify 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 database management (RDBMS's) from the early 1970's are elegant structured storage systems whose design was based on formalism for representing structure and relationships. In brief, it stores collections of tables, each of which stores entities with common sets of attributes. One row equals one entity, columns equal the attributes for the entity, and the “id” column is the primary key, permanent and unique.

For Rails, primary keys use common convention of assigning integers in increasing order. Rails uses the “Acitve Record Architectural Pattern”. This pattern is a single instance of the model class, one movie. It corresponds to a single row in a 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.

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.

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.

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 localhose 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 Indentifier

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

Example HTTP Requests:

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

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.