OBJECTIVES

- Describe where MVC goes in a Rails application structure
- Follow an http request from browser to router to controller to model to view to response to browser

MODEL VIEW CONTROLLER OVERVIEW

In order to ensure that applications have a specific organizational structure, the creators of Rails utilized the model-view-controller architecture. This concept can be a little abstract if you've never used it before, so let's give it a real world analogy.

In a restaurant kitchen we have three key components:

- A chef that makes the food
- A waiter that takes the order and brings out the food
- A table where the food is served to the customer

Using this as an analogy, we can assign the following roles to each of our restaurant components:

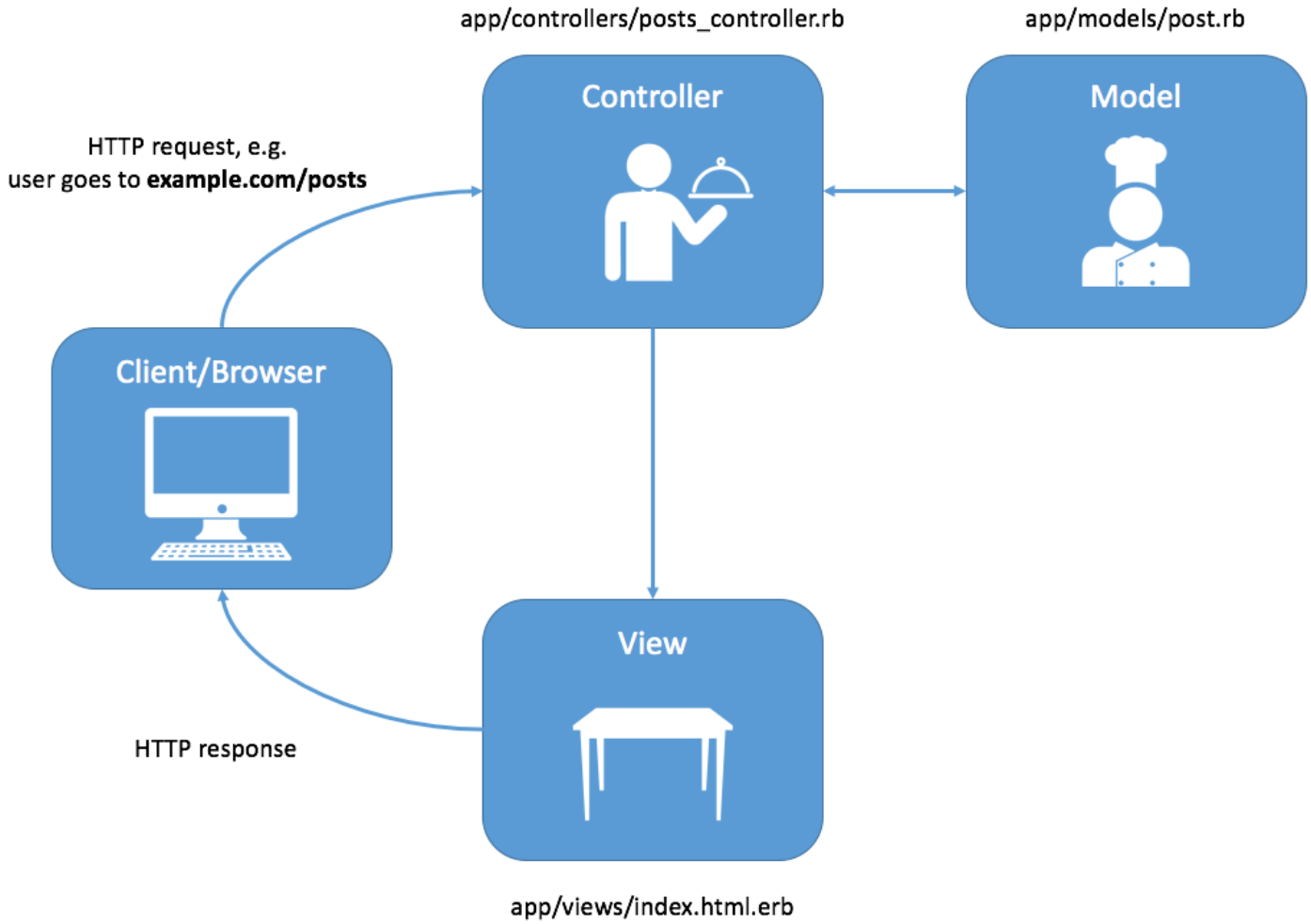
- Model - The model is the chef, it should manage the critical aspects of the application. One of my favorite tasks in a Rails application is working with the model files. This is where you can be very expressive with the custom algorithms that you want to utilize and you also have direct access to the specific database record. The logic of your application should mainly reside in the model files.
- Controller - The controller is the waiter. In the same way that the waiter takes the order from the customer to the chef and then brings the food to the table, the controller transmits data requests from the user to the model, and then delivers data that is rendered in the view to the user.
- View - The view is the table. A table shouldn't do anything besides sit there and hold the food when it is delivered. In like manner the views should not contain any programming logic, they should simply render what the controller sends it. One of the top issues I discover when I review a Rails application that has a large number of bugs is that the developer integrated too much logic directly into the view.

ROUTING, FILE NAMING CONVENTIONS, AND DATA FLOW

Rails was created with the concept of convention over configuration and this holds true for how the MVC structure was setup. View files correspond directly to controller files, which speak directly with models. As an example, imagine that you have a blog that has a database table called `posts`. You will have the following set of files:

- A `post.rb` model file that will contain: validations, database relationships, callbacks, and any custom logic for posts.
- A `posts_controller.rb` file that will have methods that will manage data flow for the Post behavior, this will include the full set of CRUD features, the standard methods are: index, new, create, show, edit, update, and destroy.
- A `views/` directory that will contain a corresponding view for each of the pages that the end user will access. For a CRUD based model, a few of the standard views would include: an index view to show all records, a show page that shows a specific record, and then new and edit pages that both render a form.

REQUEST FLOW



ROLES AND RESPONSIBILITIES

MODELS

At the end of the day the model file is a Ruby class. If it has a corresponding database table it will inherit from the `ActiveRecord::Base` class, which means that it has access to a number of methods that assist in working with the database. However, you can treat it like a regular Ruby class, allowing you to create methods, data attributes, and everything else that you would want to do in a class file. In a typical model file you will find your application's domain logic, extending the restaurant analogy the chef (your model) performs a number of tasks to create each meal that the waiter (controller) and especially the table (views) don't know anything about. Some of this domain logic would include items such as complex database queries, data relationships and custom algorithms.

It is important to remember to follow the single responsibility principle for your model class files. If any of the methods that you place in the model file perform tasks outside the scope of that specific model, they should probably be moved to their own class.

CONTROLLERS

As mentioned before, the controller is like the waiter in a restaurant. The controllers connect the models, views, and routes. To make the process more straightforward, think in terms of the following process:

- The view looks to the controller and only has access to the instance variables that the controller makes available. Those instance variables will contain any/all data coming in from the database.
- The routes file looks to the controller and ensures that the methods in the controller match the items in the routes file.

Remembering our restaurant analogy, the easiest way to think of the controller is that it manages data flow between the router, model, and views, in the same way that a waiter takes the order from a patron, relays the order details to the chef, and brings the meal out to the table.

VIEWS

In a Rails application, the view files should contain the least amount of logic of any of the files in the model-view-controller architecture. The role of the view is to simply render whatever it is sent from the database.

Rails also does a great job of supplying built in ActionView helper methods that you can implement to efficiently code the views. For example, if you wanted to create a `div` for a set of blog posts that you want to iterate over, you can implement the following code:

```
<%= div_for(@post, class: "post-index-page") do %>
  <p><%= @post.title %> <%= @post.summary %></p>
<% end %>
```

Which is translated to the following HTML markup:

```
<div id="post_42" class="post post-index-page">
  <p><strong>My Amazing Blog Post</strong> With an incredible
summary</p>
</div>
```

Notice how the Action View helper enabled us to dynamically set the HTML tags without having to write any HTML code at all? You will discover that this is a very helpful tool as your views grow in size, the more Ruby you can write and the less HTML the cleaner your views will be, which results in them being easier to manage and scale.