Prerequisites

1. Basic understanding of JavaScript and Ruby on Rails
2. Basic understanding of principles behind asynchronous programming

Goals of this tutorial

1. Understand the reasons why web applications use AJAX
2. Understand how to use AJAX in javascript and specifically in conjunction with RoR

Resources

- jQuery AJAX documentation: http://api.jquery.com/jQuery.ajax/
- Rails AJAX documentation: http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html
**Topic 1: AJAX introduction**

See slides posted under recitations to review

**Topic 2: jQuery**

A brief taste of asynchronous requests in javascript:

```javascript
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 1) { // call to open succeeded }
    if (xhr.readyState == 2) { // call to send succeeded }
    if (xhr.readyState == 3) { // response is coming in }
    if (xhr.readyState == 4) { // response completed }
}
xhr.open('GET', '/url/to/something', true);
xhr.send();
```

jQuery makes this much easier with `$ajax` (many more settings are listed at [http://api.jquery.com/jQuery.ajax/](http://api.jquery.com/jQuery.ajax/)):

```javascript
$.ajax("/url/to/something", {
    type: 'PUT',
    data: {
        param1: 'param1Value',
        param2: 'param2Value'
    },
    dataType: 'JSON' // Usually unnecessary, jQuery can guess the return type
    // etc - many more settings are available
});
```

Now, let’s look at AJAX callbacks. When you fire off an asynchronous event, you are usually looking for some response from a server which you will then use to edit the page you’re currently on:

```javascript
$.ajax("/url/to/something", {
    type: 'PUT',
    data: {
        param1: 'param1Value',
        param2: 'param2Value'
    },
    dataType: 'JSON' // Usually unnecessary, jQuery can guess the return type
    // etc - many more settings are available
}).done(function(data) {
```
```javascript
console.log("Request successful: " + data);
})
.fail(function(data) {
    console.log("Server gave an error: " + data);
})
.always(function() {
    console.log("Regardless, this request has finished.");
});
```

You can leave out the fail and always handlers if you’re only interested in the successful calls. There are a few other handlers which you can find on the jQuery ajax API, but these three are the ones you’ll see most commonly.

There’s also simplified `.post` and `.get` methods that are wrappers around `.ajax`:

```javascript
$.post("/url/to/something", {
    param1: 'param1Value',
    param2: 'param2Value'
}, function(data) {
    console.log("Success! data: " + data);
});
```

Using $.ajax and its variants have a few key benefits:

- More concise syntax, cleaner error handling
- Cross-browser compatible (XHR behaves badly in certain browsers like IE)
- Automatic data-parsing (JSON responses get detected and transformed into JS objects)

In general, if you’re already using jQuery on your website (for more reasons we’ll see below) there’s very few reasons to not use it’s AJAX functionality. Check out [https://github.com/jquery/jquery/blob/master/src/ajax.js](https://github.com/jquery/jquery/blob/master/src/ajax.js) to get some appreciation for the messy work handled by jQuery - try searching for “IE” for example.
Topic 3: Client-side AJAX in rails

Railsguide: [http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html#unobtrusive-javascript](http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html#unobtrusive-javascript)

Rails has a number of built-in helpers which makes writing AJAX-ready HTML very easy.

**Unobtrusive javascript**

This is a concept you were already introduced to, but we would like to re-emphasize due to its importance in understanding rails AJAX helpers.

Rails uses a technique called "Unobtrusive JavaScript" to handle attaching JavaScript to the DOM. This is generally considered to be a best-practice within the frontend community, but you may occasionally read tutorials that demonstrate other ways.

Here's the simplest way to write JavaScript. You may see it referred to as 'inline JavaScript':

```html
<a href="#" onclick="this.style.backgroundColor='#990000'">Paint it red</a>
```

The problem with this approach is that as your javascript gets longer and more complex, inline JavaScript quick becomes very difficult to read and maintain. Let's separate the javascript out into a separate function:

```javascript
paintIt = function(element, backgroundColor, textColor) {
  element.style.backgroundColor = backgroundColor;
  if (textColor != null) {
    return element.style.color = textColor;
  }
};
```

```html
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
```

This code is a little better, but imagine we have multiple elements on which we want to run the `paintIt` function on click. This approach isn't very DRY, and doesn't fully separate concerns because you are still writing a javascript function call into your HTML. This is where unobtrusive javascript comes in:

```javascript
$(function() {
  return $('a[data-background-color]').click(function() {
    var backgroundColor,.textColor;
    backgroundColor = $(this).data("background-color");
    textColor = $(this).data("text-color");
    return paintIt(this, backgroundColor, textColor);
  });
});
```

Once we’ve done this, we can write our HTML as follows:
The benefits to this approach are many; check the rails guide to see a list. For now, now that we understand unobtrusive javascript, rails AJAX helpers will be much more clear. Surprise surprise: let's move on to rails AJAX helpers.

Form helpers
As you know, Rails provides a bunch of view helper methods written in Ruby to assist you in generating HTML. Sometimes, you want to add a little Ajax to those elements, and Rails has got your back in those cases. Because of Unobtrusive JavaScript, the Rails "Ajax helpers" are actually in two parts: the JavaScript half and the Ruby half.

First, we'll see how the Ruby:

```ruby
<%= form_for(@post, remote: true) do |f| %>
  ...
<% end %>
```

This code should seem pretty familiar as you've already used form helpers in your views. The only difference is "remote: true". The following HTML will be generated from the above Ruby:

```html
<form accept-charset="UTF-8" action="/posts" class="new_post" data-remote="true" id="new_post" method="post">
  ...
</form>
```

Note the `data-remote='true'`. Similar to how we wrote a function that selects all anchor tags with `data-background-color`, Rails has javascript which will select any element with `data-remote='true'`. Now, the form will be submitted by Ajax rather than by the browser's normal submit mechanism.

There are more helpers which Rails provides for Ajax:

```ruby
<%= form_tag('/posts', remote: true) %>
<%= link_to "a post", @post, remote: true %>
<%= button_to "A post", @post, remote: true %>
```
Controllers

Let's move on from the client side. Ajax isn't just client-side, you also need to do some work on the server side to support it. Often, people like their Ajax requests to return JSON rather than HTML.

Imagine you have a series of users that you would like to display and provide a form on that same page to create a new user. The index action of your controller looks like this:

class UsersController < ApplicationController
  def index
    @users = User.all
    @user = User.new
  end
  # ...
end

The index view (app/views/users/index.html.erb) will contain the following:

<b>Users</b>

<ul id="users">
  <% @users.each do |user| %>
    <li><%= user.name %></li>
  <% end %>
</ul>

<br>

<%= form_for(@user, remote: true) do |f| %>
  <%= f.label :name %><br>
  <%= f.text_field :name %>
  <%= f.submit %>
<% end %>

The form on the bottom of the page will call the create action on the Users controller. Because <i>data-remote</i> is set to true in the produced HTML, the request will be posted to the users controller as an Ajax request. Here's what the create method of the users controller will look like:

def create
@user = User.new(params[:user])
respond_to do |format|
  if @user.save
    format.js  {}
  end
end
end

Notice the format.js in the respond_to block; this allows the controller to respond to your Ajax request.

Views
Similar to how format.html will look for a create.html.erb view, format.js will look for a create.js.erb view. The contents of the file will generate the actual Javascript code that will be sent and executed on the client side:

```javascript
$(
  "<li>%<%= escape_javascript(@user.name) %></li>"
).appendTo("#users");
```

This code will append the name of the new user to the list with ID users that we wrote in the view. It’s recommended to use this method in conjunction with partial views. We’ll see a live example of this now.

**Topic 5: Sample application**

The sample application shown in recitation will be hosted in a Github repo soon, and the link will be updated here when that happens.