AJAX, API, Tests

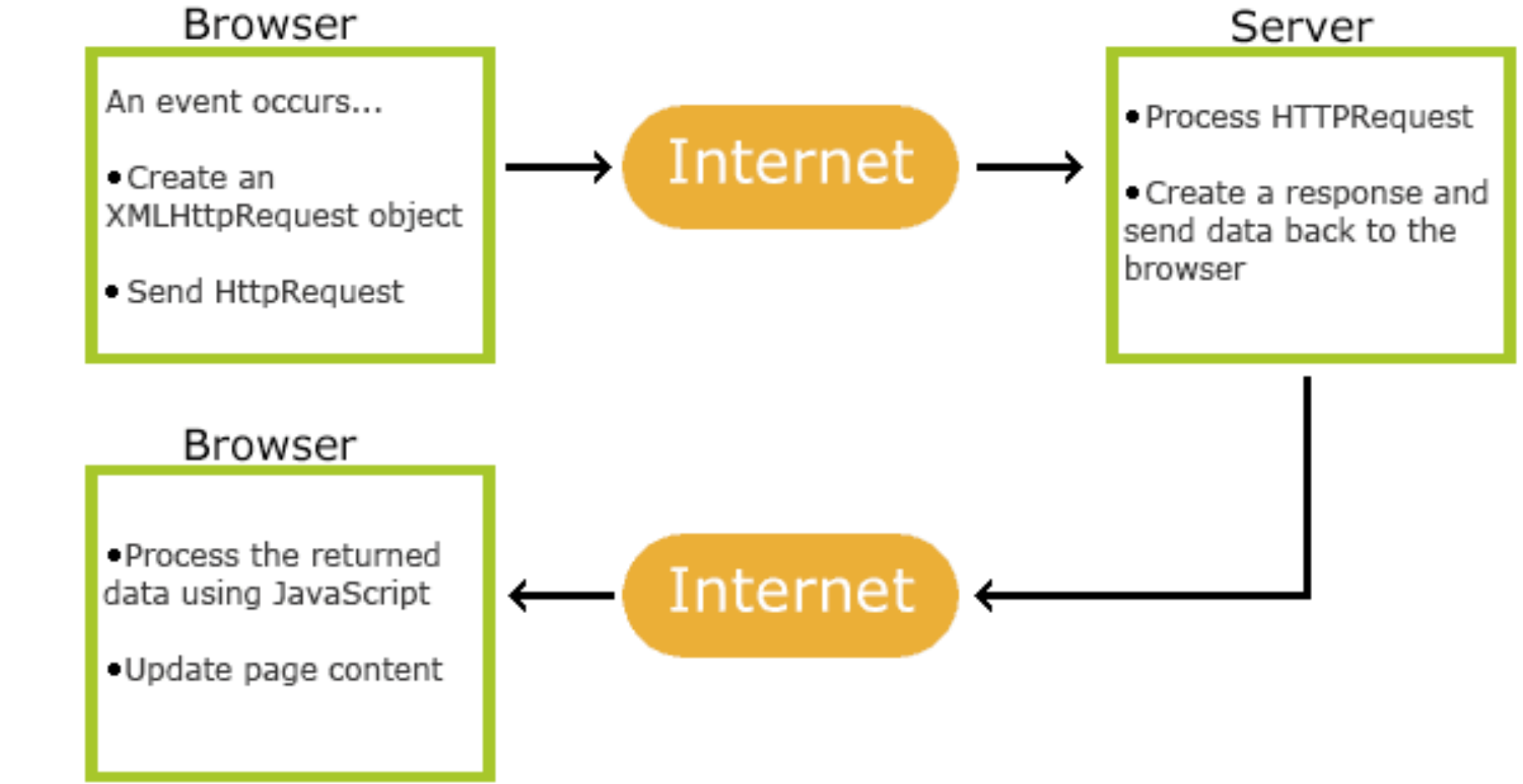
Rodrigo Rodriguez
rorodr@gmail.com
Skype: rarodriguezr

AJAX

- AJAX = Asynchronous JavaScript And XML.
- AJAX no es un lenguaje de programación, sino es una manera distinta de utilizar los estándares de desarrollo Web
- AJAX es el arte de intercambiar información con un servidor y actualizar partes de un sitio Web, sin la necesidad de recargar toda la página.

http://www.w3schools.com/ajax/ajax_intro.asp

AJAX: funcionamiento



AJAX en Rails

- Aplicar AJAX requiere de cambios tanto en el HTML (navegador), como en el lado del server
- En ambas secciones, Rails provee las herramientas necesarias para facilitar la interacción entre las partes.
- En helpers, se utiliza la opción “remote: true”, en controllers se efectúan “vistas” de tipo JS.

AJAX in Rails: Helpers

- Los siguientes helpers cuentan con la opción de “remote: true”, la cual hace que el navegador haga una solicitud AJAX en vez de recargar la página:
  - **form_for**: Crear formularios que cargan AJAX
  - **form_tag**: Crear formularios que cargan AJAX
  - **link_to**: Crear enlaces que cargan AJAX
  - **button_to**: Crear botones que cargan AJAX

AJAX in Rails: Controllers

- Al utilizarse uno de los helpers anteriores (o hacer un llamado AJAX), el controller por defecto detecta que es un llamado asincrónico y establece que la petición es de formato “JS”
- Controller en vez de presentar el archivo “.html.erb”, busca el archivo “.js.erb”
- Archivos “.js.erb” deben de tener javascript (jquery) en vez de HTML

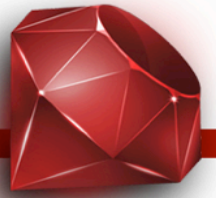
AJAX in Rails: JS files

- Permiten el uso de cualquier código Javascript/Jquery que se quiera, de modo que se pueda modificar la información de la página. Ejemplo:
  - `$("<%= escape_javascript(render @user) %>").appendTo("#users");`
- Al igual que en las vistas de HTML, se pueden usar variables de Rails, helpers, entre otras cosas.

ActionMailer

- Action Mailer permite enviar emails desde su aplicación utilizando clases de tipo Mailer y vistas.
- Todo Mailer funciona muy semejante a los controladores, así como las vistas que muestra permiten la existencia de layouts para un mailer específico.
- Para crear el mailer se puede usar el generador: `rails g mailer MiMailer`

ActionMailer: Configuración



- La configuración es simple, basta con definir el método de envío de correos (En un inicializador): Sendmail, SMTP o Files
  - `config.action_mailer.delivery_method= :smtp`
- Definir la configuración propia para el método de envío, por ejemplo la configuración de gmail
- Se pueden definir otros valores, como el URL por defecto para los enlaces, información de quien envía el correo, entre otros.

ActionMailer: Ejemplo

```
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer’
end
class MiMailer < ApplicationMailer
  default from: 'notifications@example.com’
  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome!')
  end
end
```

ActionMailer: Vistas

- En Action Mailer, se permite el envío de correos utilizando 2 tipos de vistas, en formato TEXT y en formato HTML.
- Las vistas funcionan igual que las de un controlador, se permite el uso de variables, métodos de Rails, así como HTML
- Si existe la vista HTML y la TXT, Rails envía las 2 vistas en cada correo enviado, para tener un mejor alcance de usuarios que lean el correo

ActionMailer: Ejemplo Vista

```
<!DOCTYPE html>
<html>
  <head>
   <meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
    <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br>
    </p>
    <p>
      To login to the site, just follow this link: <%= @url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

ActionMailer: Enviar correo

- Una vez configurado el mailer, así como definido su método y sus vistas. Se puede enviar un correo (sin importar donde se invoque), utilizando el comando:
  - MiMailer.welcome_email(user).deliver_now
- Por último es factible utilizar librerías externas para enviar correos de forma más eficiente: usando colas de mensajes, ejecutando correos en batches, entre otros métodos.

API RESTful

- Sin estado: El estado del cliente no debe de ser almacenado en el servidor entre una solicitud y otra
- Cada solicitud debe de identificar un único recurso a la vez: Al consultar se retorna una única dupla en la tabla, al modificar solamente se altera un registro
- Representación de Transferencia de estado: Los enpoints retornan una representación JSON o XML del recurso

Librerías para crear APIs

- **jBuilder**: Implementación nativa adaptada por Rails, permite generar una estructura de JSON de manera fácil
- **Grape**: Un framework que permite desarrollar todo el API basado en convenciones
- **Rabl**: Una librería que permite generar templates de ruby para JSON, BSON, XML y otros sin requerir cambios de código.
- ***Swagger***: Permite generar documentación para los Endpoints de una forma agradable

JBuilder

- Permiten crear JSONs de una manera simple y rápida.
- Al igual que en una vista de HTML, se genera un archivo en la carpeta views, con la extensión: “.json.jbuilder”
- Loa archivos de Jbuilder contienen toda la lógica requerida para generar el JSON

jBuilder: Ejemplo

```
json.content format_content(@message.content)
json.(@message, :created_at, :updated_at)
json.author do
  json.name @message.creator.name.familiar
  json.email_address
@message.creator.email_address_with_name
  json.url url_for(@message.creator, format: :json)
end
if current_user.admin?
  json.visitors calculate_visitors(@message)
end
json.comments @message.comments, :content, :created_at
```

Jbuilder: Ejemplo

```
{
  "content": "<p>This is <i>serious</i></p>",
  "created_at": "2011-10-29T20:45:28-05:00",
  "updated_at": "2011-10-29T20:45:28-05:00”,
  "author": {
    "name": "David H.",
    "email_address": "'David Heinemeier Hansson'
<david@heinemeierhansson.com>",
    "url": "http://example.com/users/1.json"
  },
  "visitors": 15,
  "comments": [{ "content": "Hello everyone!",
"created_at": "2011-10-29T20:45:28-05:00" }, ...]
}
```

Diseño del API (1/3)

- Una de las partes principales en la creación de un API es su diseño, por ello acá algunas recomendaciones para la creación de un API
  - De ser posible, separar el API del controller (a no ser que se quiera el URL), para poder añadir algunas capacidades específicas como seguridad, versionamiento, entre otras.
  - Manejar versiones: Para ello basta con crear namespaces en los controllers con la version del api. Los routes también varían un poco al utilizar namespaces.

Diseño del API (2/3)

- Usar un BaseApiController (o semajante), donde se definen métodos genéricos para errores (idealmente con retorno de JSON), seguridad, entre otros
- Usar algún método de autenticación: Hay múltiples alternativas para esto, incluyendo librerías como Devise para verificar tokens, entre otras opciones.
- Utilizar módulos para almacenar métodos genéricos que sirvan en todo el API.

Diseño del API (3/3)

- Algunas veces el API tiene validaciones distintas que la Web, por lo que se recomienda utilizar validaciones condicionales en los modelos
- Para generar los JSON siempre utilizar los Jbuilder. A veces lo que tenemos que presentar es muy simple y nos vemos tentados a utilizar @modelo.to_json, esto no siempre es lo mejor a largo plazo.

Interactuando con un API

- Existen diversas maneras de interactuar con un API, siendo las 2 mejores:
  - **ActiveResource:** Implementa un mapeo de Objetos relacionales (como si fuera un modelo), el cual se conecta a un servicio REST (usando las rutas estándar de Rails).
  - **RestClient**: Permite interactuar fácilmente con cualquier URL (incluyendo RESTful API)

ActiveResource (1/3)

- Al utilizarse de manera semejante a los modelos, por lo general las clases que heredan de Active Resource se almacenan en “app/models”
- Un ejemplo de declaración de clase (y uso) es:

```
class Person < ActiveResource::Base
  self.site = http://api.people.com:3000
end
tyler = Person.find(1)
Person.exists?(1)
```

ActiveResource (2/3)

- Algunos métodos válidos son:
  - find: Permite obtener un elemento según su ID
    - `Person.find(1) =>` http://SERVER/people/1.json
  - all: Permite obtener todos los elementos
    - `Person.all =>` http://SERVER/people.json
  - Create/new: Permite crear un objeto nuevo
    - `person = Person.new(:first_name => 'Juan')`
    - `person.save`
  - Update: Actualiza un objeto existente
    - `person = Person.find(1)`
    - `person.first = "Alberto"`
    - `person.save`

ActiveResource (3/3)

- Algunos métodos válidos son:
  - delete: Permite eliminar un elemento según su ID
    - `person = Person.find(1)`
    - `Person.destroy`
- También es posible asignar asociaciones, lo cual se maneja igual que ActiveRecord
  - `has_many :comments`
- Las relaciones requieren que un post tenga un comentario asociado en su URL
  - `GET http://blog.io/posts/1/comments.json`

RestClient

- Librería que permite interactuar con cualquier tipo de solicitud de manera sencilla. Ej:

```
RestClient.get 'http://example.com/resource', {:params =>
{:id => 50, 'foo' => 'bar'}}
RestClient.post 'http://example.com/resource', :param1 =>
'one', :nested => { :param2 => 'two' }
RestClient.delete 'http://example.com/resource’
response = RestClient.get 'http://example.com/resource'
response.code ➔ 200
response.cookies ➔ {"Foo"=>"BAR", "QUUX"=>"QUUUUX"}
response.to_str ➔ \n<!DOCTYPE html PUBLIC \"-//W3C//DTD
HTML 4.01//EN\"\n   \"http://www.w3.org/TR/html4/
strict.dtd\">\n\n<html ....
```

Pruebas en Rails

- Rails provee una forma nativa para manejar pruebas de código, los cuales cuentan con diversos tipos de prueba:
  - Modelos: Pruebas de unidad para modelos
  - Controladores: Pruebas de unidad
  - Integración: Pruebas de integración de las distintas capas
  - Mailers: Pruebas de unidad para mailer

Datos para pruebas

- Considerando que las pruebas requieren de datos que permitan verificar el correcto funcionamiento del código, en Rails existen los Fixtures
- **Fixtures** son archivos YML que declaran los distintos elementos disponibles para pruebas
  - @contact = contacts(:one) => Siendo contacts(:one), un elemento declarado en fixtures/contacts.yml

Otras librerías para pruebas

- Rspec:
  - "Competencia" directa de las pruebas nativas
  - Cuenta con mucho soporte de la comunidad
  - Sintaxis es bastante distinta a las pruebas nativa, por lo que en algunos casos se facilita la lectura
- Factory Girl: Similar a Fixtures, con la diferencia que se declaran clases y objetos. Además se pueden guardar o no en la BD

Recursos adicionales

- ***Libro:*** *Agile Web Development with Rails 4*
- http://guides.rubyonrails.org/working_with_javascript_in_rails.html
- https://www.airpair.com/ruby-on-rails/posts/building-a-restful-api-in-a-rails-application
- http://guides.rubyonrails.org/action_mailer_basics.html