

Introducción a Arquitectura Web

Franco Bulgarelli

Versión 1.3

Septiembre 2020

[1 Introducción](#)

[2 Estilos arquitectónicos](#)

[3 Sistemas centralizados y descentralizados](#)

[4 Arquitectura Web](#)

[4.1 Introducción](#)

[4.2 Un poco de historia](#)

[4.2.1 La Web tradicional](#)

[4.2.2 Web 2](#)

[4.3 Arquitectura en detalle](#)

[4.3.1 Procesamiento](#)

[4.3.2 Respuesta](#)

[4.3.3 Pedido](#)

[4.3.4 Presentación](#)

[4.4 Diseño en Arquitecturas Web](#)

[4.4.1 Declarativas vs programáticas](#)

[4.4.2 Aplicación Web MVC Server Side](#)

[4.4.3 Aplicación Web MVC Server Side “ajaxiana”](#)

[4.4.4 Aplicación Web MVC Client Side](#)

[4.5 Y el estado, ¿donde está?](#)

[4.6 Material complementario](#)

[4.7 Para el que quiera investigar](#)

1 Introducción

Si vamos a hablar de arquitectura Web, vale la pena primero repasar qué entendemos por arquitectura. Hay distintos enfoques, complementarios, que nos dan diferentes definiciones de *arquitectura de software*:

1. el diseño de los **componentes lógicos de alto nivel**, tales como los módulos, los paquetes, los procesos lógicos y las bibliotecas.
2. la toma de decisiones sobre **aspectos difíciles de cambiar** a lo largo del ciclo de vida de un sistema, tales como la elección de lenguajes de programación, el software de base, el vocabulario e idioma y las tecnologías de persistencia y presentación.
3. el diseño centrado en satisfacer **requerimientos no funcionales**, tales como lograr cierta velocidad de respuesta, minimizar el uso de memoria, procesamiento o almacenamiento y garantizar un acceso concurrente o la transferencia segura de información
4. el diseño de los **aspectos físicos del sistema**, tales como la selección de los componentes de hardware, el despliegue (instalación) del software y la topología de las redes de comunicación.

Como vemos, la arquitectura sigue siendo una arista particular del diseño, y trata de idear componentes, asignarles roles y determinar sus formas de comunicación. Pero la novedad es que ahora incluiremos en nuestros análisis y decisiones conocimientos mucho más detallados sobre la tecnología.

Ya sea para pensar el sistema en términos de alto nivel o para diseñar en términos de requerimientos no funcionales, al hablar de arquitectura elementos tales como los lenguajes, las plataformas, la distribución a través de la red de los componentes, nos impactarán aún más que cuando sólo nos dedicamos a construir el modelo de dominio.

Y esto será particularmente cierto en arquitectura Web, un mundo en el que conviven decenas de tecnologías diferentes.

2 Estilos arquitectónicos

De la misma forma que existen patrones de diseño (es decir soluciones probadas a problemas frecuentes) también existen patrones arquitecturales, a veces llamados *estilos arquitectónicos*. Y de igual forma, no existe una lista exhaustiva y definitiva, sino que son el resultado mutable de décadas de experiencias de construcción de software real.

Existen varios libros que intentan enumerar y describir los estilos más comunes, algunos de ellos son:

- *Patterns of Enterprise Application Architecture*, de Martin Fowler
- *Fundamentals of Software Architecture: An Engineering Approach*, de Mark Richards

3 Sistemas centralizados y descentralizados

De entre todos los estilos arquitectónicos, los dos primeros que nos servirán para entender las arquitecturas web son el **centralizado (o monolítico)** y **descentralizado (o distribuido)**: mientras que en los sistemas monolíticos todo el código se ejecuta en un mismo ambiente o máquina virtual, en un sistema descentralizado habrá partes que se ejecuten en otros ambientes, quizás incluso en otras computadoras¹.

Clásicamente los programas más básicos con los que contamos en una computadora de escritorio, como una calculadora o un editor de texto, presentan arquitecturas centralizadas, dado que toda su lógica de modelo, presentación y persistencia se resuelve en la misma computadora, e incluso en un mismo ejecutable.

Sin embargo, las aplicaciones más complejas contarán con módulos que se ejecutarán en otras computadoras: aplicaciones de mensajería, redes sociales, editores de texto colaborativos o un juegos en línea son sólo algunos ejemplos. Es más: existe una creciente tendencia a que aplicaciones que otrora se ejecutaban completamente en el entorno **local** (es decir, en la propia computadora de quien las usara) hoy en día ejecuten algunas funciones de forma **remota**, tales como gestión de complementos, actualizaciones periódicas, publicidad y recopilación de datos de uso.

4 Arquitectura Web

4.1 Introducción

- Los sistemas Web nos proponen una arquitectura distribuida
 - Los componentes están distribuidos en dos tipos de nodos: clientes (muchos) y servidores (uno, al menos a nivel lógico)
- Los clientes se comunican con el servidor siguiendo un protocolo de pedido-respuesta:
 - Un cliente hace un pedido, el servidor lo procesa y responde.
 - El cliente se encarga de presentar (renderizar) la respuesta al usuario final
 - La comunicación ocurre a través de redes Intranets o la misma Internet, empleando un protocolo llamado HTTP
- Los clientes son responsables de presentar la información de la respuesta al

¹ Pese a su aparente contraposición, en realidad las categorías de monolítico y distribuido no son taxativas y su significado puede cambiar según el contexto. Por ejemplo, en muchas ocasiones nos encontraremos con sistemas que se ejecutan en un mismo entorno al estudiarlos con más detalle veremos que no son referidos como monolitos dado que sus partes gozan de cierto grado de independencia al compararlas con sistemas similares.

Así mismo, nos encontraremos también con sistemas cuyas partes se ejecutan en diferentes entornos, pero al mirarlos más de cerca observaremos que cada una de esas partes están tan fuertemente acopladas que en ciertos contextos se lo llamará lo llamará monolítico.

usuario, empleando tecnologías específicas de la Web.

4.2 Un poco de historia

4.2.1 La Web tradicional

Las tecnologías que se emplean para la construcción de aplicaciones Web no son las mejores para hacer aplicaciones "dinámicas" (aplicaciones con un razonable nivel de interactividad y responsividad de la GUI), porque no fueron diseñadas para tal fin.

¿Por qué decimos esto? La Web, en su concepción original, era un servicio concebido para compartir contenido estático (textos académicos, por ejemplo) entre usuarios de computadoras personales con capacidad de procesamiento muy limitada: alguna organización (como una universidad) publicaba estos contenidos, y los usuarios se limitaban a consultarlos. Es decir, estas aplicaciones de la primera Web (a veces renombrada luego como Web 1.0) eran fundamentalmente:

- no interactivas
- operaban sobre contenido poco mutable
- de sólo texto
- de comunicación unidireccional (la información siempre viaja desde el servidor al cliente, a pedido de un cliente)

Por esto, HTTP, el protocolo de comunicación empleado en la Web es:

- Pedido-Respuesta (se abre una conexión por cada pedido, que surge del cliente, y el servidor la cierra cuando ha enviado la respuesta)
- Stateless (el protocolo per-sé no maneja ninguna noción de memoria de pedidos anteriores)
- Textual (se intercambian mensajes de sólo texto)
- Basado en códigos de respuesta (incluso para los flujos de error; no hay memoria compartida, continuaciones, excepciones ni eventos)

Esto es esquemático y en su forma más básica, el protocolo es complejo y presenta muchos aspectos como tipos de contenido, métodos, cachés y conexiones persistentes, etc, que modifican sustancialmente la visión anterior.

4.2.2 Web 2

El protocolo HTTP nos impone una forma de comunicación mucho menos rica que el paso de mensajes entre objetos, con lanzamiento de excepciones, en un ambiente con estado y efectos.

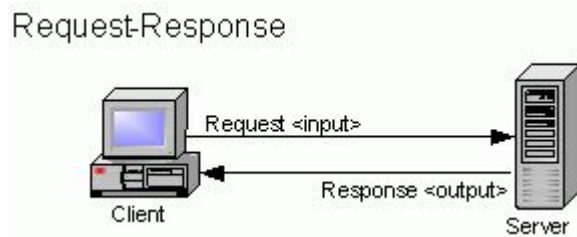
Además el advenimiento de las llamadas aplicaciones dinámicas, o RIAs (Rich Internet Applications), en la década del 2000, llevó a que las mismas sean cada vez más grandes, complejas, con una creciente demanda de interactividad no planificada inicialmente, empleando en general lenguajes orientados a objetos en el servidor para su desarrollo. La masificación de Internet dio también mayor necesidad de soportar más usuarios y mayores niveles de concurrencia, y clientes con hardware y software cada vez más heterogéneo (por ejemplo, diversidad de sistemas operativos, navegadores, dispositivos móviles), hace más compleja la presentación del contenido.

Todo esto lleva a programadores, diseñadores gráficos, gente de infraestructura, etc más infelices :(

Entonces ¿por qué se siguió usando? Si bien no es tema de la materia analizar cuándo una arquitectura es la adecuada, a grandes rasgos diremos que:

- Ya existe, y está probada
- Emplea una infraestructura tecnológica (Internet, http, y sus tecnologías de presentación) que está ampliamente difundida y accesible

4.3 Arquitectura en detalle



Dijimos que a grandes rasgos el modelo Web es el siguiente:

1. El cliente hace un pedido
2. El servidor procesa el pedido
3. El servidor devuelve una respuesta
4. El cliente renderiza la respuesta

4.3.1 Procesamiento

¿A qué les suena el segundo punto? Recuerden que:

- El servidor procesa una entrada, y devuelve una salida
- No hay memoria del pedido anterior (no hay estado)
- Los resultados llevan un código de resultado (por ejemplo, para manejar errores)

¿A qué patrón de comunicación les suena? ¡Call & Return! ¡Call & Return!

¿Y cómo podríamos modelarlo? Con una función (de un parámetro), por supuesto!

```
procesar = proc { |request| response }2
```

¡El modelo de pedido-respuesta es tan simple como eso!

4.3.2 Respuesta

¿Y cómo se ve una respuesta en HTTP? Diremos que una respuesta es una terna

```
[codigoDeRespuesta, cabeceras, cuerpo]
```

Donde, por ejemplo, un código 200 (OK) significa que el procesamiento fue correcto. Las cabeceras son un conjunto de metadatos sobre la respuesta; por ejemplo el conjunto vacío

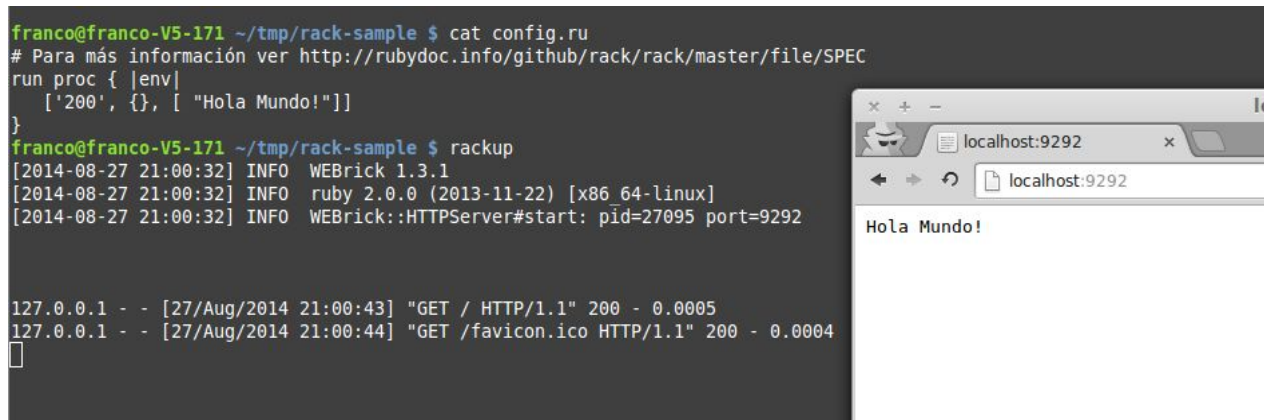
² ¿Se acuerdan de las lambdas de Haskell, o de los bloques de código de Smalltalk? Acá en lugar de escribir `\x -> y`, o `[|x| y]` escribiremos `proc { |x| y }`, en breve explicaremos por qué

[] es que no hay metadatos. Con todo esto, el Hola Mundo más simple es el siguiente bloque de código:

```
proc { |request| ['200', {}], [ "Hola Mundo!"] ] }
```

De hecho, el pseudocódigo anterior es código válido para el lenguaje Ruby cuando empleamos una biblioteca estándar llamada rack³; podemos ejecutar nuestro servidor en el puerto 9090 con un script como el siguiente⁴:

```
run proc { |env|  
  ['200', {}], [ "Hola Mundo!"]  
}
```



The screenshot shows a terminal window on the left and a web browser on the right. The terminal window displays the following commands and output:

```
franco@franco-V5-171 ~/tmp/rack-sample $ cat config.ru  
# Para más información ver http://rubydoc.info/github/rack/rack/master/file/SPEC  
run proc { |env|  
  ['200', {}], [ "Hola Mundo!"]  
}  
franco@franco-V5-171 ~/tmp/rack-sample $ rackup  
[2014-08-27 21:00:32] INFO WEBrick 1.3.1  
[2014-08-27 21:00:32] INFO ruby 2.0.0 (2013-11-22) [x86_64-linux]  
[2014-08-27 21:00:32] INFO WEBrick::HTTPServer#start: pid=27095 port=9292  
  
127.0.0.1 - - [27/Aug/2014 21:00:43] "GET / HTTP/1.1" 200 - 0.0005  
127.0.0.1 - - [27/Aug/2014 21:00:44] "GET /favicon.ico HTTP/1.1" 200 - 0.0004
```

The web browser on the right shows the address bar with 'localhost:9292' and the page content 'Hola Mundo!'.

4.3.3 Pedido

¿Cómo se ve un pedido en HTTP? Los pedidos tienen varios atributos, algunos de ellos son:

- Parámetros (existen distintos tipos de parámetros)
- Una URI/URL (la "dirección web", es decir, la identificación/localización del contenido al que quiero acceder)
- Un método, que le da cierta semántica al pedido (más adelante hablaremos de esto)

El primer punto no tiene mucho misterio: simplemente en ocasiones necesitaremos que nuestro pedido tenga información extra, parametrizada. Por ejemplo, si queremos imprimir nuestro Hola Mundo parametrizando el mensaje, podríamos reescribir nuestro bloque de código de procesamiento de la siguiente forma:

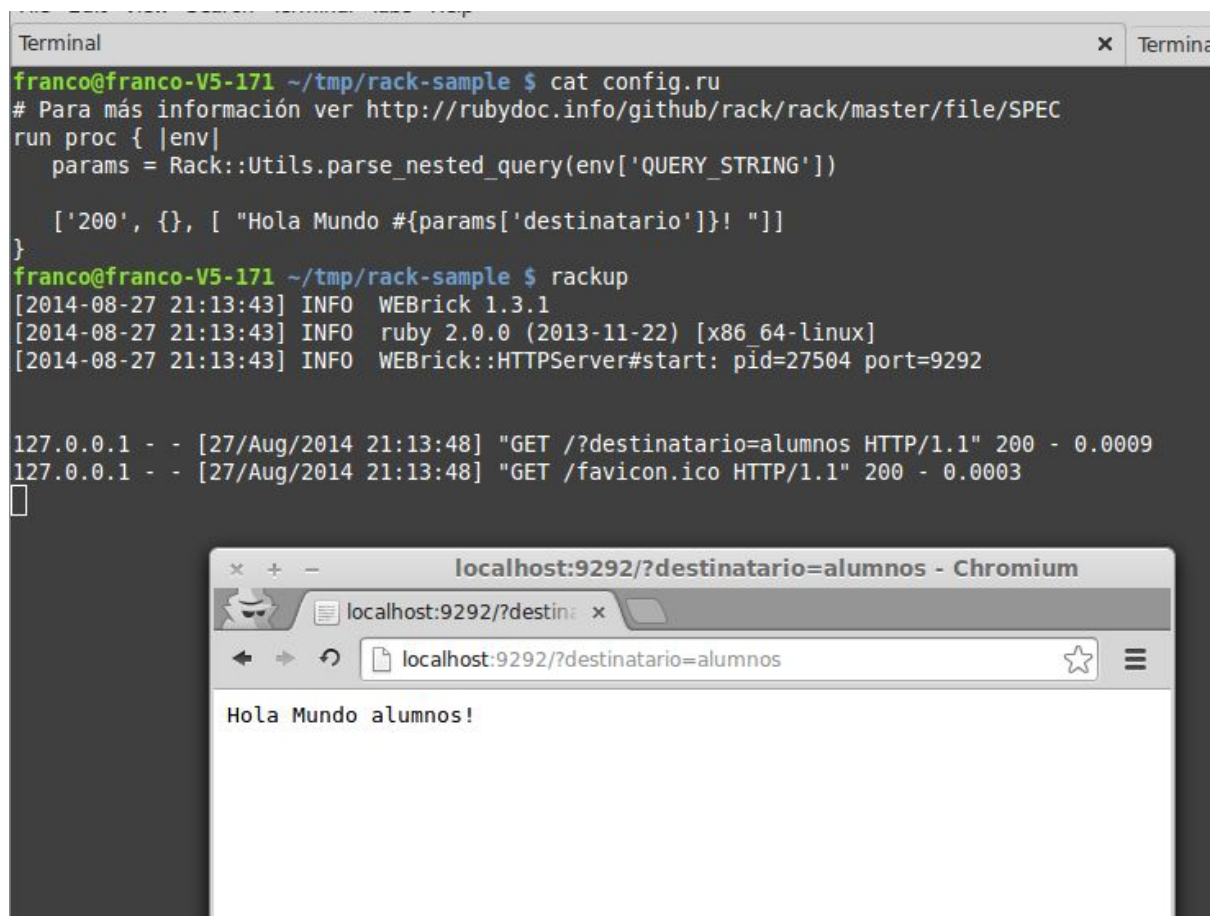
```
proc { |env|  
  params = Rack::Utils.parse_nested_query(env['QUERY_STRING'])  
  ['200', {}], [ "Hola Mundo #{params['destinatario']}! "]  
}
```

Y una forma de pasarle el argumento sería la siguiente:

³ <http://rack.github.io/>

⁴ El ejemplo funcionando se encuentra acá <https://github.com/dds-utn/rack-sample>

⁵ Nótese que renombramos la variable de request a env. Esto es mas bien un tecnicismo de rack



The image shows a terminal window and a web browser. The terminal window, titled 'Terminal', shows the following commands and output:

```
franco@franco-V5-171 ~/tmp/rack-sample $ cat config.ru
# Para más información ver http://rubydoc.info/github/rack/rack/master/file/SPEC
run proc { |env|
  params = Rack::Utils.parse_nested_query(env['QUERY_STRING'])

  ['200', {}, [ "Hola Mundo #{params['destinatario']}! "]]
}

franco@franco-V5-171 ~/tmp/rack-sample $ rackup
[2014-08-27 21:13:43] INFO WEBrick 1.3.1
[2014-08-27 21:13:43] INFO ruby 2.0.0 (2013-11-22) [x86_64-linux]
[2014-08-27 21:13:43] INFO WEBrick::HTTPServer#start: pid=27504 port=9292

127.0.0.1 - - [27/Aug/2014 21:13:48] "GET /?destinatario=alumnos HTTP/1.1" 200 - 0.0009
127.0.0.1 - - [27/Aug/2014 21:13:48] "GET /favicon.ico HTTP/1.1" 200 - 0.0003
```

The web browser, titled 'localhost:9292/?destinatario=alumnos - Chromium', shows the URL 'localhost:9292/?destinatario=alumnos' in the address bar. The page content displays 'Hola Mundo alumnos!'.

Los métodos son algo invisible al usuario final del sistema web. HTTP indica que cuando se haga un pedido al servidor, este llevará una indicación del método que emplea, que dota de cierta semántica al pedido.

En la Web 1, los únicos pedidos posibles eran de consulta. Sin embargo, a partir de la versión 0.9 del protocolo de mediados de los noventa, se soportan explícitamente otros pedidos:

- GET: las consultas tradicionales. El procesamiento de tales pedidos no debería tener efecto.
- PUT, DELETE: operaciones con efecto destinadas a, respectivamente, actualizar y eliminar información el servidor
- POST: una operación con efecto "genérica", empleada normalmente para crear información nueva en el servidor, y para cualquier operación con efecto que no caiga en PUT o DELETE.

Es muy recomendable emplear los métodos con la semántica más próxima a lo que necesitamos. Los detalles sobre como indicar el método, quedan para el curioso.

Por último, las URIs requieren un poco más de debate. Un servidor Web típico sirve varios contenidos diferentes/permite realizar múltiples computaciones. Por eso existen las URLs: nos permiten identificar y localizar de forma unívoca un recurso, y dentro del servidor,

encontraremos en distintas URLs relativas al mismo.

La URLs nos presenta una estructura jerárquica, donde la raíz es /.

La forma de diseñar la estructura de URLs nos lleva a dos estilos de organización:

- Orientado a procedimientos: cada ruta modela una acción, eventualmente parametrizada, donde la semántica de la operación es propia de cada modelo de dominio. Se suele emplear los métodos GET y POST, para computaciones con y sin efecto, respectivamente. Son ejemplos de rutas:
 - /comprar?idProducto=456&idZona=48
 - /buscarPorPrecio?maximo=100&minimo=10&incluirDescuentos=true
 - /buscarProducto?id=45
- Orientado a recursos: cada ruta apunta a un recurso bien definido, siguiendo algunas convenciones bien conocidas. La semántica de cada ruta está dada en parte por el modelo de dominio, pero en gran medida también por el método HTTP que se utilice y siguiendo las reglas PUT/POST/GET/DELETE antes comentadas (de hecho, una misma ruta puede tener semánticas diferentes según el método empleado). Ejemplos:
 - /productos/45: si se usa GET, se devuelve el producto con id 45. Si se usa DELETE, se lo borra.
 - /productos: si se usa GET devuelve todos los productos, si se usa PUT, se los actualiza en lote
 - /productos/45/ventasRecientes: si se usa GET, devuelve todas las ventas recientes del producto con id 45

Metáfora asociada: llamo a mi casa por teléfono, pido a quien atienda que vaya al dormitorio, y en la mesita de luz se fije si está el libro "Diseño Web Introductorio".
Matcheamos: teléfono -> protocolo http, mi casa -> 127.0.0.1, recurso (el libro) alojado en un "directorio" (dormitorio/mesita de luz) de ese host. Y así formamos una URL.

Aunque hay varios motivos para preferir la segunda por sobre la primera, en cierta medida ambas se complementan y ambos esquemas coexisten. Un debate profundo sobre el diseño de la estructura de de URIs escapa a la materia, pero *preferiremos* adaptarnos a la semántica HTTP y utilizar el estilo **Orientado a Recursos**

¿Cómo entonces hacemos que nuestro servidor reaccione a distintas URIs?

O bien hacemos horribles switches:

```
proc { |req|  
  if req.path == "unaUri"  
    procesarDeUnaForma  
  else if req.path == "otraUri"  
    procesarDeOtraForma  
  else if (... )  
}
```


O bien, le decimos al servidor que aplique distintas funciones de procesamiento según la URI, lo cual es más feliz.

4.3.4 Presentación

Si bien las respuestas tienen en principio cuerpos de texto, cualquiera que haya navegado en internet sabe que las presentaciones son mucho más ricas e interactivas que simple texto. Con el texto plano, ¡nos quedamos cortos!

El texto de la respuesta normalmente no es texto plano, sino que está codificado: lo que el servidor responde normalmente es el código fuente de un programa escrito usando una combinación de lenguajes, que es interpretado por un programa cliente, el mismo programa que también es responsable de crear las conexiones HTTP. ¿Como se llama esta aplicación? Adivinaron, ¡es el browser!

Los browsers modernos son capaces de entender los siguientes lenguajes sin necesidad de ningún complemento (plugin), por lo que constituyen el estándar de facto de la Web:

- HTML: lenguaje basado en marcas, primo hermano del XML, diseñado para estructurar información
- CSS: lenguaje para formatear información (estructurada en HTML)
- JS: lenguaje de propósito general, que en los navegadores es utilizado para desarrollar cualquier lógica de aplicación. En particular, algunos casos de uso concreto son:
 - Mutar, acceder a, y observar eventos del DOM (representación orientada a objetos de una estructura jerárquica XML, HTML o similar) del contenido HTML
 - Implementar efectos visuales complejos
 - Realizar pedidos al servidor en segundo plano
 - Implementar navegabilidad del lado del cliente
 - Implementar lógica de negocio del lado del cliente

Los browsers modernos son capaces de entender los siguientes lenguajes sin necesidad de ningún complemento estándar de facto de la Web:

- HTML: lenguajes basado en marcas, primo hermano del XML, diseñado para estructurar información
- CSS: lenguaje para formatear información (estructurada en HTML)
- JS: lenguaje de proposito general, que en los navegadores es utilizado para desarrollar cualquier lógica de de uso concreto son:
 - Mutar, acceder a, y observar eventos del DOM (representación orientada a objetos de una estructura contenido HTML)
 - Implementar efectos visuales complejos
 - Realizar pedidos al servidor en segundo plano
 - Implementar navegabilidad del lado del cliente
 - Implementar lógica de negocio del lado del cliente

DOM

window > li

attributes	[]
baseURI	"https://sites.google.co...enoch/introduccion-web"
childElementCount	0
childNodes	[<TextNode textContent="Mutar, acceder a, y obs...ar) del contenido HTML"]
0	<TextNode textContent="Mutar, acceder a, y obs...ar) del contenido HTML"
attributes	null
baseURI	"https://sites.google.co...enoch/introduccion-web"
childNodes	[]
constructor	Text { }
data	"Mutar, acceder a, y obs...ar) del contenido HTML "

4.4 Diseño en Arquitecturas Web

Si bien la abstracción de la función que devuelve ternas es útil para entender el problema, esta forma de trabajo es, como se imaginarán, bastante incómoda, porque hay que construir el texto de la respuesta a mano, lidiar a mano con varios aspectos del protocolo HTTP, y se pierden las abstracciones. Arriba de esta abstracción se construyen otras de más alto nivel: un MVC (limitado). Así aparecen los controladores (la C de MVC) que:

- toman un pedido,
- lo descomponen en sus argumentos y operan sobre los aspectos de más bajo nivel del protocolo,
- delegan el procesamiento en el modelo, y
- dejan la información que necesitan otros componentes para presentar la UI (modelo 2).

MVC model 2

- basado en plantillas (Rails, Spring MVC, Grails, Play, Struts)
- basado en componentes (Wicket)

4.4.1 Declarativas vs programáticas

Veamos el siguiente "pseudocódigo":

```
class LibrosController { //<--- un controlador

    public RepositorioLibros repositorio() {
        return RepositorioLibros.instance()
    } -- el modelo

    //<--- implementa métodos por cada ruta
    // GET /libros
    public Libros search(request, response) {
        return repositorio().buscar(request.queryParams("titulo"))
    }

    // GET /libros/:id
    public Libro detalle(request, response) {
        return repositorio().get(request.params("id").toInt())
    }
}
```

4.4.2 Aplicación Web MVC Server Side

<https://github.com/flbulgarelli/ejemplos-web-mvc-libros/tree/master/server-side>

- Si bien la aplicación es interactiva, sigue respetando el modelo tradicional pedido-respuesta
 - La lógica está fundamentalmente en el servidor
 - El cliente sólo se limita a renderizar la respuesta
- El código de presentación (HTML, CSS, JS) viaja completo en las respuestas del servidor
- La vista se renderiza de forma total
- El árbol DOM se mantiene inmutable

4.4.3 Aplicación Web MVC Server Side “ajaxiana”

<https://github.com/flbulgarelli/ejemplos-web-mvc-libros/tree/master/server-side-ajax>

Olvidándonos de siglas y frases hechas, decimos que nuestros clientes y servidor se comunican de forma "ajaxiana" cuando se apartan del modelo pedido-respuesta tradicional:

- Una vez obtenida la respuesta, los clientes continúan haciendo pedidos en segundo plano, y cuando tienen un resultado, no necesariamente recargan la página

completa, sino tan sólo una porción de la misma.

- Los clientes mantendrán eventualmente conexiones persistentes, por lo que el servidor eventualmente podría enviar información sin pedido previo.
- El servidor devuelve respuestas que describen información pura o una vista parcial: esa parte de vista serán luego renderizada dentro una vista más grande, que el cliente construyó previamente.

Algunas consecuencias son:

- El cliente ahora implementa más lógica de aplicación: empezamos a tener más código JavaScript
- Tendremos que también diseñar del lado del cliente. La estrategia de mezclar código HTML y JS sin organizar el código y generar abstracciones, no escala.

Algo interesante de este modelo de programación es que, los pedidos ajax son modelados típicamente como la aplicación de una función de orden superior, que toma dos argumentos mínimos: el pedido, y una función que procesa la respuesta. En JS:

```
$.ajax({                                //<-- aplicamos la función ajax
  url: "" + nuevoId,                    //<-- el primer argumento es la url del
  request                               //<-- el segundo argumento es la función que
  complete: function(response) {       //<-- el segundo argumento es la función que
    procesa el resultado
    if (response.status == 200) {
      $("#libro").text(response.responseText);
    }
    else {
      alert("No se pudieron obtener los detalles del libro " + id + "\n" +
        "Por un error: " + response.status + response.statusText);
    }
  }
});
```

¿Por qué modelarla de esta forma? ¿No sería más natural hacerlo como la aplicación de la función de procesamiento del lado del servidor que modelamos al principio de esta clase?

```
var respuesta = procesarAjax(pedido)
```

Sucede que por su naturaleza, los pedidos Ajax son asíncronos. Entonces, una función como la anterior, aunque nos sirve del lado del cliente, no nos sirve del lado del servidor, porque call-and-return (al menos en su concepción más básica) nos fuerza a ser sincrónicos. La estrategia de reemplazar el retorno de una función por un argumento extra que lo procese, es otro patrón de comunicación, conocido como continuación, y es muy frecuente encontrarlo en este tipo de sistemas.

4.4.4 Aplicación Web MVC Client Side

- Basada en el modelo anterior. Además de implementar MVC del lado del servidor, lo

hace del lado del cliente.

- La gran mayoría del código de presentación se encuentra en el cliente y no en el servidor
- El servidor tan sólo se limita a entregar datos del modelo, y a servir (pero no evaluar) el código de la vista

¡Y siempre hay híbridos!

4.5 Y el estado, ¿donde está?

- En el servidor: Session
- En el medio: Cookie
- En el cliente: Local Storage

4.6 Material complementario

- <https://sites.google.com/site/utndesign/cursos/martes-manana-tarde/resumenes/resumenes-martes-2012/clase22-introduccionalasinterfacesweb>

4.7 Para el que quiera investigar

El diseño de sistemas Web involucra una cantidad enorme de tecnologías, que escapan a la materia y que en esta introducción hemos mostrado de forma muy simplificada, para proteger al inocente. El que esté más interesado en saber más sobre las mismas, puede investigar sobre:

- [HTTP](#): Hypertext Transfer Protocol. El protocolo utilizado entablar una comunicación cliente-servidor, siendo un cliente posible el Navegador.
- [HTML](#): el lenguaje utilizado para construir las vistas (originalmente pensadas como **Documentos**). Es interpretado por el navegador.
- [CSS](#): Cascading Style Sheets. El lenguaje utilizado para decorar (gráficamente hablando) una vista HTML.
- [Javascript](#): un lenguaje de programación de tipado dinámico. Su uso nació de la necesidad de tener un [cliente rico/pesado](#), es decir, de tener mayor carga en la computadora que corre el Navegador y muestra la página. Hoy en día, Javascript es muy utilizado también del lado del Servidor, siendo [Node.js](#) una implementación de Framework MVC Servidor.
- [DOM](#): Document Object Model. Es una convención para interpretar elementos en un documento. Los tipos de documento que se pueden interpretar son: HTML, XHTML y XML.
- Frameworks MVC Servidor. En este caso hemos utilizado [Grails](#), pero existe [una infinidad más](#).

- Interacción Web de bajo nivel. En este caso utilizamos la interfaz [Rack](#)
- Diseñar rutas orientada a recursos es conocido como [REST](#), que cubre muchísimos aspectos más.
- La función \$.ajax se trata en realidad de un mensaje al objeto \$, provisto por la biblioteca [jQuery](#) (\$ es, en realidad, un alias al objeto homónimo a la biblioteca).
- Frameworks MVC Cliente:
 - [Angular.js](#): Framework MV* que permite *extender* el lenguaje de marcado extendido de hipertexto (HTML) y trabajar con la **V**ista directamente en el HTML. Está libre de jQuery y usa Javascript puro.
 - [Backbone.js](#): Framework [MVVM](#) que utiliza jQuery por detrás y “compila” las vistas a partir de templates y Modelos de Vista (View Models).