URL’s

* **Orientado a procedimientos**: cada ruta modela una acción.
  + /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.
  + /productos/4 5: 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
* Get **Read** /productos/45/ventasRecientes
* Put **Update/Replace** *PUT* [*http://www.example.com/buckets/secret\_stuff*](http://www.example.com/buckets/secret_stuff)
* Post: **Create** *POST* [*http://www.example.com/customers/12345/orders*](http://www.example.com/customers/12345/orders)**NO** devuelve modelAndView. Podes optar por hacer un Response res.redirect(/algo);
* Patch: **Update/Modify** *PATCH* [*http://www.example.com/buckets/secret\_stuff*](http://www.example.com/buckets/secret_stuff)

**POST** to a URL **creates a child resource** at a *server defined* URL.

**PUT** to a URL **creates/replaces the resource** in its entirety at the *client defined* URL.

**PATCH** to a URL **updates *part* of the resource** at that client defined URL.

* Delete: **Deletes** *DELETE* [*http://www.example.com/customers/12345*](http://www.example.com/customers/12345)

|  |  |  |
| --- | --- | --- |
| **MÉTODO** | **CONCEPTO** | **OBSERVACIONES** |
| **GET** | GET lleva los datos de forma "visible" al cliente (navegador web). El medio de envío es la URL. Los datos los puede ver cualquiera. | Los datos son visibles por la URL, por ejemplo:  www.aprenderaprogramar.com/  action.php?nombre=pedro&apellidos1= gomez |
| **POST** | POST consiste en datos "ocultos" (porque el cliente no los ve) enviados por un formulario cuyo método de envío es post. Es adecuado para formularios. Los datos no son visibles. | La ventaja de usar POST es que estos datos no son visibles al usuario de la web. En el caso de usar get, el propio usuario podría modificar la URL escribiendo diferentes parámetros a los reales en su navegador, dando lugar a que la información tratada no sea la prevista. |

Tags

* Input
  + **Text**

 First name:<br>  
  <input type="text" name="firstname"><br>

* + **TextArea** (es el que tiene para agrandar en la punta, solo apto para chupapijas)

<textarea rows="4" cols="50">

* + **Password**

 User password:<br>  
  <input type="password" name="psw">

* + **Submit** (Ejecuta la acción del form del que se encuentra, le podes poner una especifica formAction = “actionquequieras”)

 Last name:<br>  
  <input type="text" name="lastname" value="Mouse"><br><br>  
  <input type="submit" value="Submit">

* + **Reset**
  + **Radio Button** (Mutuamente Excluyentes y generalmente pocos elementos)(Son los circulitos)

<form>  
  <input type="radio" name="gender" value="male" checked> Male<br>  
  <input type="radio" name="gender" value="female"> Female<br>  
  <input type="radio" name="gender" value="other"> Other  
</form>

* + **Checkbox** (No mutuamente excluyentes, podes elegir de a muchos)(Son los cuadraditos)

<form>  
  <input type="checkbox" name="vehicle1" value="Bike"> I have a bike<br>  
  <input type="checkbox" name="vehicle2" value="Car"> I have a car   
</form>

* + **Button**

<input type="button" onclick="alert('Hello World!')" value="Click Me!">

* + **Color** (Seleccionar un color, abre una ventana similar a la de paint para seleccionar colores)
  + **Date**
  + **Datetime** (yyyy-mm-dd ‘t’ hh:mm:ss)
  + **Date-local** (dd/mm/yyyy hh:mm:ss)
  + **Email** (trabaja con una expresión regular)
  + **Month**
  + **Number** (spinner)

<form>  
  Quantity (between 1 and 5):  
  <input type="number" name="quantity" min="1" max="5">  
</form>

* + **Range** (barra deslizadora con numeritos) (Rango de números (min,max) <scale> )
  + **Hidden**

<input type="hidden" name="country" value="Norway">

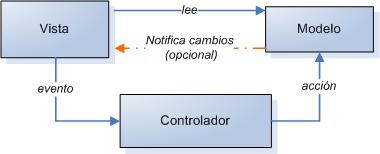
* + **Search**
  + **Tel**
  + **Time**
  + **Url**
  + **Week**
  + **Number** ( min, max, step)
* Values of input
  + **Disabled:** para no modificar
  + **Size:** cantidad de caracteres permitidos
  + **Autocomlete:** para que te autocomplete los valores si fueron previamente cargados
  + **Autofocus:** the first input shoud be focused when the page loads
  + **Pattern:** te deja generar un patrón al input (una ER)
  + **Required**
* Links
  + Se utiliza la etiqueta <a href=”referencia”>
  + Para imagenees <a href=”…”><img src=”…”/></a>
  + Links a otra etiqueta <a href="#id">
* **Data list: Version mas copada del combo box**
* Muestra información oninput="x.value=parseInt(a.value)+parseInt(b.value)">

<output name="x" for="a b"></output> (lo completa dinámicamente)

Binding

* **El binding relaciona dos propiedades entre sí haciendo que se mantengan sincronizadas: si una cambia la otra también**. Con frecuencia se eligen una propiedad del modelo con una propiedad de la vista, pero no es indispensable. En nuestros primeros ejemplos la propiedad de la vista será el valor ingresado por el usuario en un control, más tarde incorporaremos otras propiedades.
* **El binding puede ser unidireccional o bidireccional**. En caso de un binding bidireccional, cualquiera de los dos extremos del vínculo puede ser originador de las modificaciones. Esta característica será de gran utilidad porque provee una forma muy sencilla que la vista refleje automáticamente cambios en el modelo.

MVC



Vamos a darle la responsabilidad a alguien (que llamaremos controller) para que se encargue de manejar el binding de atributos y acciones en forma bidireccional (o sea: hacia la vista y hacia el dominio). Para eso vamos a trabajar de la siguiente manera: la vista se va a registrar como interesada en el modelo. Entonces cada vez que alguien actualice el modelo, se va a disparar una notificación hacia determinados interesados (esta sería una implementación del *Observer* pattern).

* El **modelo** contiene el comportamiento de la aplicación. Como primera aproximación podemos asociar modelo con "modelo de dominio", pero más adelante vamos a ver que pueden aparecer otros tipos de modelo.   
    
  Para que el modelo logre cumplir su cometido de contener *toda* la lógica de la aplicación, hay dos situaciones que debemos evitar o minimizar:
  + La aparición de lógica propia del dominio en los demás componentes (vista o controlador).
  + La contaminación del modelo con elementos propios de la interacción del usuario y con cualquier cuestión relacionada con la tecnología.
* La **vista** tiene la responsabilidad de interactuar con el usuario. Para ello cuenta (por lo menos) con:
  + Elementos visuales que muestran información al usuario o que permiten al usuario ingresar información o interactuar con ellos (por ejemplo con el teclado o el mouse). A estos elementos visuales se los suele denominar ***controles*** o ***widgets***.
  + **Contenedores** para los controles, como ventanas y paneles.
  + Estrategias para organizar visualmente los controles dentro de un contenedor, denominadas ***layouts***.

Dado que elegimos una estrategia basada en objetos, tanto los controles básicos como los contenedores y layouts estarán modelados como objetos. Normalmente esos objetos no requieren de ser programados por nosotros, en cambio el framework nos provee de un conjunto de componentes primitivos listos para ser utilizados. Entonces nuestro código para la vista se limita a la parte creacional: crear los controles, agregarlos a algún contenedor, asignarle un layout al contenedor, etc.

* El **controlador** tiene la responsabilidad de comunicar a la vista con el modelo. Esta comunicación está basada en dos conceptos que ya mencionamos pero todavía no definimos con precisión: eventos y binding. Esta es la parte más compleja y es de lo que van a tratar las próximas secciones.

MONGO DB (documental)

Como podemos ver, acceder a los datos de este JSON (o cualquiera similar) es muy sencillo; basta solo realizar la query que queremos para obtener lo deseado. En cambio en ORM (y con un modelo de datos desnormalizado) podríamos tener que hacer muchos Joins a distintas tablas.

Mencionar (mostrando en el pizarrón)

* One to one (Empleados - CV’s / Paciente - Historial médico)
  + Fk vs. Embeber bidireccionalmente
    - Forma de accesso (tamaño)
    - Cuál crece más seguido?
    - Tamaño (tope de 16 Mb)
    - Atomicidad (de un solo documento)
* One to many (few) (Ciudades - Personas) / (Posts - comentarios)
  + Embeber en ciudad(tamaño!)
  + Embeber en pesona (problemas de consistencia)
  + True linking
* Many to many (Estudiantes - Profesores)
  + Embeber (a costa de duplicar información)
    - Puedo querer profesores antes de tener alumnos o alumnos sin profesores

# CRUD Operations Create Read Update Delete

## Creación de documentos

Lo que nos va a interesar contar en esta parte son los comandos necesarios para insertar documentos en nuestra base de datos, vamos a ver los comandos de InsertOne y el IntertMany. También podemos mostrar cómo insertar datos mediante la operación de update.

Podemos crear una colección on-the-fly de ayudantes con datos básicos. SI incluímos el campo intereses, podemos mostrar que puede ser tanto una colección como un valor.

Mostrar que \_id se autogenera, pero que se puede ser cualquier tipo de dato si está especificado. Es inmutable, no te lo deja cambiar (atómicamente, pero podés sacarlo cambiarlo y reinsertarlo).

Como estuvimos hablando recién, podemos insertar un documento en MongoDB mediante la utilización de un comando:

|  |
| --- |
| db.libros.insertOne({“título”: “Juego de Tronos”, “saga”: “Canción de hielo y fuego”, “anio\_publicacion”: 1996}); |

Esto lo que va a hacer es crear el documento dentro de la colección libros y, en el caso de que la colección no exista, creará la misma.

\*Acá podemos mostrar que, al dar enter, el documento se va a insertar exitosamente. Después podemos hacer un “db.libros.find().pretty()” para mostrar que el documento que acabamos de insertar si existe. :) Se puede mostrar que al crear, podemos nosotros proporcionar un “\_id:” creado por nosotros, el cual tiene que ser único.\*

También, podemos querer insertar muchos documentos… En ese caso Mongo nos provee de una herramienta similar a la anterior, insertMany.

Si bien estos son los dos comandos más importantes (o más usados) para la creación de documentos, también vamos a poder crearlos con los comandos de actualización; los cuales se conocen por su terminología en inglés upserts.

Hablando en grandes rasgos este último término se utiliza para cuando queremos actualizar documentos que no existen y, por lo tanto, mongo procederá a crearlos.

## Lectura de documentos

A la hora de leer documentos (o trabajar en las búsquedas dentro de estos), podemos requerir ver todo el documento, alguno que satisfaga la búsqueda deseada o, también, podemos pedir realizar una lectura sobre un elemento específico dentro del documento.

\*Acá estaría bueno tener una pequeña base en donde hacer las consultas. Podemos hacer mostrar lo que dije arriba con:\*

* Find - Para demostrar que nos va a dar EXACTAMENTE lo que escribamos dentro. Cuando utilizamos el find, podemos valernos de otros “campos” (por decirlo de alguna manera) para hacer más específica nuestra query:
  + Un filtro, de manera que pueda especificar lo que quiero ver:
    - Podemos filtrar por un solo campo o varios, a su vez podemos usar condiciones a la hora de filtrar y utilizar ands y ors.

\*Acá mostramos cómo podemos buscar por un campo en específico, todavía no hay una base hecha; cuando esté terminada hay que poner un ejemplito.\*

* Tenemos muchos más métodos de colecciones, podemos ver cuales nos interesan mostrar porque son un montón.

Tipos de mapeo de herencia

Single Table

* *A favor*
  + *Cuando las subclases comparten muchos atributos en común*
  + *Cuando quiero evitar muchos JOINs al hacer las consultas*
  + *Buena performance general*
  + *Soporta todo tipo de relaciones olimórficas*
  + *Evita generar muchas tablas*
  + *Para trabajar con colecciones de objetos polimórficos tengo que hacer consultas diferentes, no soporta relaciones polimórficas many-to-one ni one-to-one*
  + *Más simple*
* *En contra* 
  + *Campos no utilizados que deben aceptar valores nulos*
  + *Tentación de “reutilizar” atributos para cosas distin*





Table per class

* *A favor*
  + *Es la técnica utilizada para las entidades independientes (las clases que heredan de Object en la jerarquía: en el ejemplo de la página 1 serían Pedido, Item y Producto)*
  + *Cuando las subclases comparten muy pocos atributos entre sí*
  + *Permite establecer campos no nulos para cada subclase*
  + *No requiere tener un campo discriminador (Ej: TIPO\_PEDIDO para el PEDIDO)*
* *En contra*
  + *Para trabajar con colecciones de objetos polimórficos tengo que hacer consultas diferentes, no soporta relaciones polimórficas many-to-one ni one-to-one*
  + *Cada subclase repite atributos “heredados” de la superclase*



Joined

Cuando hay muchos atributos comunes entre las subclases pero también muchos atributos propios en cada subclase. Si no hay una cantidad significativa de datos y sabemos que no afectará la performance.

* *A favor*
  + *Es el modelo “ideal” según las reglas de normalización (evita redundancia de definiciones en la estructura de las tablas)*
  + *Permite establecer campos no nulos para cada subclase*
  + *Soporta todo tipo de relaciones polimórficas*
* *En contra*
  + *Es la opción que más entidades requiere crear*
  + *La que más cantidad de accesos a la base requiere (mayor costo de performance)*





**Interfaz del EntityManager**

|  |  |
| --- | --- |
| find(entityClass, entityPK) | Busca una entidad en el contexto de persistencia |
| remove(entity) | Elimina una entidad del contexto de persistencia |
| persist(entity) | Agrega una entidad al contexto de persistencia |
| createQuery(string) | Crea una consulta en JQL |

**Embeber por medio de enums**



**Embeber usando annotations de forma común**

