



Acceso a Datos

Tema 4 Bases de Datos NoSQL

José Luis González Sánchez



Contenidos

¿Qué voy a aprender?



Contenidos

1. Introducción
2. Diseño NoSQL
3. Bases de Datos Orientadas a Objetos
4. JPA con ObjectDB
5. JSON
6. MongoDB
7. Java con MongoDB
8. JPA con MongoDB
9. Servicios web REST
10. Servicios web GraphQL

Bases de Datos NoSQL

Más allá del modelo relacional



Bases de Datos NoSQL

- Las bases de datos NoSQL son aquellas que no siguen el modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS). Se caracterizan porque no usan el SQL como lenguaje principal de consultas, y además, en el almacenamiento de los datos no se utilizan estructuras fijas de almacenamiento.
- El término NoSQL surge con la llegada de la web 2.0, ya que hasta ese momento solo subían contenidos a la red aquellas empresas que tenían un portal, pero con la llegada de aplicaciones como Facebook, Twitter o Youtube, en las que el usuario interactúa en la web, cualquier usuario podía subir contenido, provocando así un crecimiento exponencial de los datos.
- Surgen así los problemas para gestionar y acceder a toda esa información almacenada en bases de datos relacionales. Una de las soluciones, propuestas por las empresas para solucionar estos problemas de accesibilidad, fue la de utilizar más máquinas, sin embargo, era una solución cara y no terminaba con el problema. La otra solución fue la de crear nuevos sistemas gestores de datos pensados para un uso específico, que con el paso del tiempo han dado lugar a soluciones robustas, apareciendo así el movimiento NoSQL.
- Así pues, hablar de bases de datos NoSQL es hablar de estructuras que nos permiten almacenar información en aquellas situaciones en las que las bases de datos relacionales generan ciertos problemas, debido principalmente a problemas de escalabilidad y rendimiento de las bases de datos relacionales, donde se dan cita miles de usuarios concurrentes y con millones de consultas diarias. Por tanto, las bases de datos NoSQL intentan resolver problemas de almacenamiento masivo, alto desempeño, procesamiento masivo de transacciones (sitios con alto tránsito) y, en términos generales, ser alternativas NoSQL a problemas de persistencia y almacenamiento masivo (voluminoso) de información para las organizaciones.



Bases de Datos NoSQL. ACID vs BASE

- Las bases de datos relacionales focalizan su interés en la fiabilidad de las transacciones bajo el conocido principio ACID, acrónimo de Atomicity, Consistency, Isolation and Durability (Atomicidad, Consistencia, Aislamiento y Durabilidad):
 - Atomicity: Asegurar de que la transacción se complete o no, sin quedarse a medias ante fallos
 - Consistency: Asegurar el estado de validez de los datos en todo momento
 - Isolation: Asegurar independencia entre transacciones
 - Durability: Asegurar la persistencia de la transacción ante cualquier fallo
- El principio ACID aporta una robustez que colisiona con el rendimiento y operatividad a medida que los volúmenes de datos crecen.
- Cuando la magnitud y el dinamismo de los datos cobran importancia, el principio ACID de los modelos relacionales queda en segundo plano frente al rendimiento, disponibilidad y escalabilidad, las características más propias de las bases de datos NoSQL



Bases de Datos NoSQL. ACID vs BASE

- Hoy en día, los modernos sistemas de datos en internet se ajustan más al también conocido principio BASE, acrónimo de Basic Availability (disponibilidad como prioridad) Soft state (la consistencia de datos se delega a gestión externa al motor de la base de datos) Eventually consistency (intentar lograr la convergencia hacia un estado consistente)
 - Basic Availability: Prioridad de la disponibilidad de los datos
 - Soft State: Se prioriza la propagación de datos, delegando el control de inconsistencias a elementos externos
 - Eventauly Consistency: Se asume que inconsistencias temporales progresen a un estado final estable



Bases de Datos NoSQL. Ventajas

- La forma de almacenamiento de información en este tipo de bases de datos ofrece ciertas ventajas sobre los modelos relacionales, a destacar las siguientes:
 - Se ejecutan en máquinas con pocos recursos: estos sistemas no requieren mucha programación, por lo que se pueden instalar en máquinas de un coste más reducido.
 - Escalabilidad horizontal: para mejorar el rendimiento de estos sistemas simplemente se consigue añadiendo más nodos, con la única operación de indicar al sistema cuáles son los nodos que están disponibles.
 - Pueden manejar gran cantidad de datos: esto es debido a que utiliza una estructura distribuida, en muchos casos mediante tablas Hash.
 - No genera cuellos de botella: el principal problema de los sistemas SQL es que necesitan transcribir cada sentencia para poder ser ejecutada, y cada sentencia compleja requiere, además, de un nivel de ejecución aún más complejo, lo que constituye un punto de entrada en común, que ante muchas peticiones puede ralentizar el sistema.



Bases de Datos NoSQL. Diferencias

- No utilizan SQL como lenguaje de consultas. La mayoría de las bases de datos NoSQL evitan utilizar este tipo de lenguaje o lo utilizan como un lenguaje de apoyo. Por poner algunos ejemplos, Cassandra utiliza el lenguaje CQL, MongoDB utiliza JSON o BigTable hace uso de GQL.
- No utilizan estructuras fijas como tablas para el almacenamiento de los datos. Permiten hacer uso de otros tipos de modelos de almacenamiento de información como sistemas de clave—valor, objetos o grafos.
- No suelen permitir operaciones JOIN. Al disponer de un volumen de datos tan extremadamente grande suele resultar deseable evitar los JOIN. Esto se debe a que, cuando la operación no es la búsqueda de una clave, la sobrecarga puede llegar a ser muy costosa. Las soluciones más directas consisten en desnormalizar los datos, o bien, realizar el JOIN mediante software en la capa de aplicación.
- Arquitectura distribuida. Las bases de datos relacionales suelen estar centralizadas en una única máquina o bien en una estructura máster—esclavo, sin embargo en los casos NoSQL la información puede estar compartida en varias máquinas mediante mecanismos de tablas Hash distribuidas.



Bases de Datos NoSQL. Diferencias

- Relacional: En estas bases de datos cada objeto o entidad vive en una fila de una tabla. Cada una de estas tablas están estrictamente definidas, es decir, cada una de las columnas define un tipo de dato específico como varchar, int o bool. Esto es el **schema de la tabla y es inviolable**, si intentas añadir un tipo de dato que no corresponde con el schema, encontrarás un error. Aquí enfatizamos la parte de estructurada porque hay una estructura que no puedes variar sobre la marcha, es decir, todos los registros de una tabla tienen la misma estructura o esquema. Esto es lo que se llama normalización de datos.
- Cuando queremos relacionar elementos de dos o varias tablas, utilizamos una foreign key o clave foránea para trazar esa relación. Luego, a través de SQL expresamos una consulta donde se unen esas claves diferentes tablas, devolviendo el resultado.
- Teniendo esto claro, una gran diferencia al pasar al lado **NoSQL** es que los datos no se almacenan en tablas per se, sino en **colecciones de Objetos o a través de documentos y colecciones de los mismos o pares clave valor**.
- Otra gran diferencia es que en las bases de datos **NoSQL no existe el schema**. En su lugar hay una convención pero sin obligación y esto es importante ya que **permite iterar o variar el diseño de la base de datos sin tener que recurrir a las migraciones**. Esta característica también tiene sus desventajas, como que nada ni nadie te garantiza esa normalización de datos del mundo SQL ya que no sabes con seguridad lo que te estás descargando. Por eso se suele chequear en el cliente que lo que hemos recibido es lo que esperábamos. **Lo que perdemos en estructura lo ganamos en flexibilidad**.



Bases de Datos NoSQL. Diferencias

- En el mundo NoSQL para obtener datos relacionados existen varias técnicas pero todas **se basan en hacer el menor número de peticiones posible**. Algo muy común es colocar la información relacionada en lugares donde se pueda descargar en conjunto en una única petición. Ojo, esto quiere decir que quizás tengas que duplicar alguna pieza de información. Otro shock si vienes del mundo SQL.
- Esta libertad a la hora de estructurar nuestros datos **nos permite optimizar la base de datos para lecturas (ocurren muchas más veces que las escrituras) siendo tremadamente rápidas**. Además, por esa de-normalización, es más sencillo escalar la base de datos en caso de que sea necesario, de forma horizontal (en más servidores) en lugar de verticalmente (con más recursos como ancho de banda, RAM, etc).
- Reflexiona, si en una red social en cada post incluimos el autor, trayéndonos los post ya tendríamos los datos del autor, pero ¿Y si cambia el nick? ¿Cada cuánto pasa esto? Es un claro ejemplo de datos embebidos.
- ¿Y en un chat? ¿Nos podemos traer todos los mensajes con sus autores? de nuevo, ¿cada cuánto cambia los datos de un autor?
- ¿Y de un repositorio? ¿Y si nos treamos embebidos su issues y commit relacionados?



Bases de Datos NoSQL. Tipos

- Clave/Valor. Los datos son almacenados y se localizan e identifican usando una clave única y un valor (un dato o puntero a los datos). Ejemplos de este tipo son: DynamoDB, Riak, o Redis. Amazon y Best Buy entre otros utilizan esta implementación. Se caracterizan por ser muy eficientes tanto para las lecturas como para las escrituras.
- Columnas. Parecido al modelo clave/valor, pero la clave se basa en una combinación de columna, fila y marca de tiempo que se utiliza para referenciar conjuntos de columnas (familias). Es la implementación más parecida a bases de datos relacionales. Ejemplos: Cassandra, BigTable, Hadoop/HBase. Compañías como Twitter o Adobe hacen uso de este modelo.
- Documentos. Los datos se almacenan en documentos que encapsulan la información en formato XML, YAML o JSON. Los documentos tienen nombres de campos auto contenidos en el propio documento. La información se indexa utilizando esos nombres de campos. Este tipo de implementación permite, además de realizar búsquedas por clave-valor, realizar consultas más avanzadas sobre el contenido del documento. Ejemplos: MongoDB, Firebase Firestore de Google CouchDB, eXist. Un caso de uso de esta tecnología lo tenemos con Netflix, empresa que proporciona contenidos audiovisuales online.
- Grafos. Se sigue un modelo de grafos que se extiende entre múltiples máquinas. En este tipo de bases de datos, la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos para recorrerla. Es un modelo apropiado para datos cuyas relaciones se ajustan a este modelo, como, por ejemplo, redes de tránsito, mapas, etc. Ejemplos: Neo4J, GraphBase o Virtuoso
- Orientadas a Objetos. Se basa en plasmar las relaciones existentes en un modelo de dominio basado en las relaciones entre clases y los objetos existentes en el sistema. Un ejemplo es ObjectDB

Diseño NoSQL

Abre tu mente y ve más allá de tablas, filas y columnas

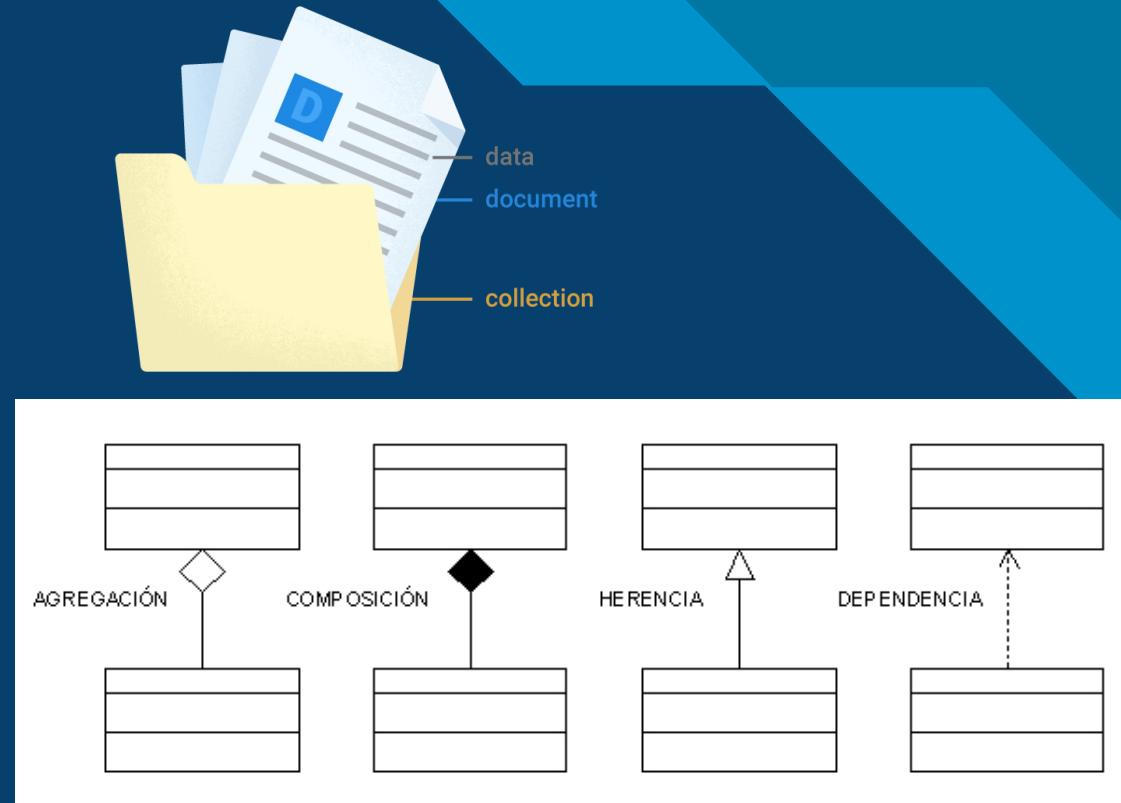
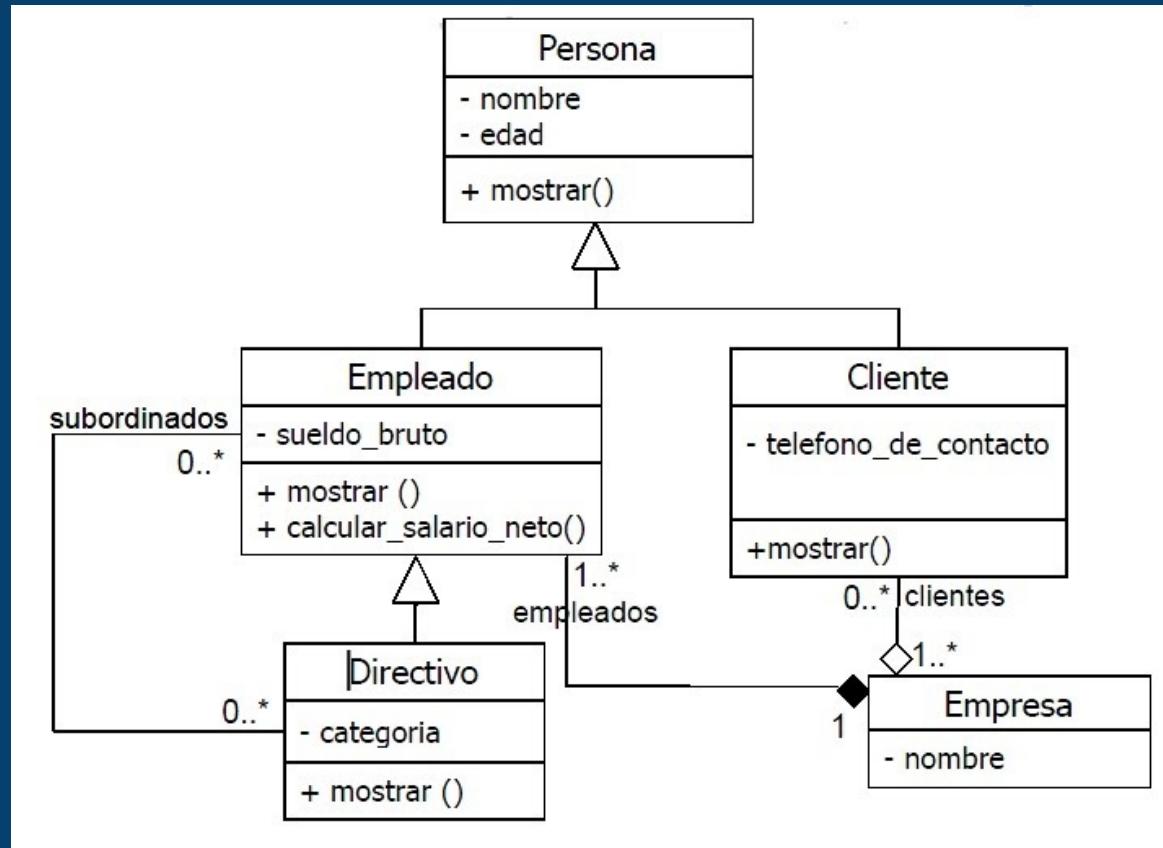


Diseño NoSQL

- En este apartado veremos las claves de cómo diseñar NoSQL.
- Para ello es fundamental que sepamos en qué Sistema Gestor NoSQL vamos a trabajar, porque no es lo mismo estructurar la información para un sistema clave-valor, que para un sistema basado en documentos u objetos.
- Además debemos pensar muy cuidadosamente las operaciones que vamos a tener para manipular la información. Pues podemos premiar lecturas sobre escrituras o modificaciones. Podemos tener valores repetidos y no tienen que estar normalizadas.
- Mucha de la gestión de la integridad recaerá en nuestros servicios o software que hagan uso de ellas.
- Ten esto en cuenta, porque no hay un lenguaje de consulta “válido” para todas y debes pensar en qué modelo de almacenamiento NoSQL es el adecuado para tu problema.
- Muchos de los patrones de diseño del mundo relacional te servirán. Relaciones, sistemas de propagación de claves. Pero puedes enriquecerlos con otros nuevos.

Diseño NoSQL

- Si trabajamos en un lenguaje OO realiza tu diagrama de clases dejando claro la relación entre ellas, navegabilidad, cardinalidad y dependencias y semántica asociada a la relación. Te recuerdo que parte de la integridad de la información la vas a tener que realizar tú mismo mediante software.





Diseño NoSQL

- Si el diseño es OO solo deberás “transpasar” los elementos del diagrama indicado a modelo de negocio con clases que lo repliquen plasmando con las herramientas del lenguaje la semántica asociada al diagrama.
- Algunos consejos que las difieren al diseño Relacional y que puedes usar dependiendo del problema y modelo de negocio a parte de los conocidos ya en el modelo relacional.
- Relaciones 1-1 y 1-M. Puedes usar clases anidadas o documentos embebidos.

```
{  
  _id: 'joe',  
  name: 'Joe Bookreader',  
  address: {  
    street: '123 Fake Street',  
    city: 'Faketown',  
    state: 'MA',  
    zip: '12345'  
  }  
}
```



Diseño NoSQL

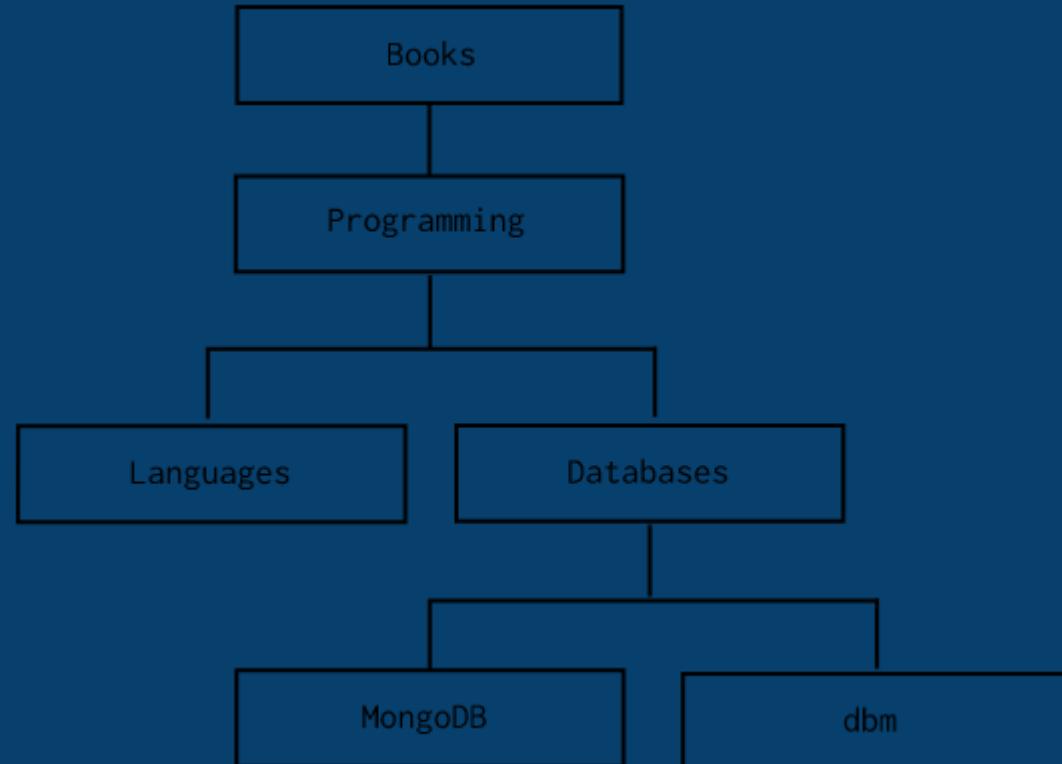
Relaciones 1 - M. Puedes usar referencias propagando objetos o claves. También puedes usar conjunto de documentos y objetos embebidos pudiendo o no elegir entre replicar elementos repetidos dependiendo si primas la lectura o lectura/escritura. También puedes usar referencias y datos embebidos. Ejemplo un editor tiene muchos libros / un libro tiene un solo editor

```
{  
    title: 'MongoDB: The Definitive Guide',  
    author: [ 'Kristina Chodorow', 'Mike Dirolf' ],  
    published_date: ISODate('2010-09-24'),  
    pages: 216,  
    language: 'English',  
    publisher: {  
        name: 'O'Reilly Media',  
        founded: 1980,  
        location: 'CA'  
    }  
}  
{  
    title: '50 Tips and Tricks for MongoDB Developer',  
    author: 'Kristina Chodorow',  
    published_date: ISODate('2011-05-06'),  
    pages: 68,  
    language: 'English',  
    publisher: {  
        name: 'O'Reilly Media',  
        founded: 1980,  
        location: 'CA'  
    }  
}  
  
{  
    name: 'O'Reilly Media',  
    founded: 1980,  
    location: 'CA',  
    books: [123456789, 234567890, ...]  
}  
{  
    _id: 123456789,  
    title: 'MongoDB: The Definitive Guide',  
    author: [ 'Kristina Chodorow', 'Mike Dirolf' ],  
    published_date: ISODate('2010-09-24'),  
    pages: 216,  
    language: 'English'  
}  
{  
    _id: 234567890,  
    title: '50 Tips and Tricks for MongoDB Developer',  
    author: 'Kristina Chodorow',  
    published_date: ISODate('2011-05-06'),  
    pages: 68,  
    language: 'English'  
}
```



Diseño NoSQL

Relaciones basadas en árboles/grafos con referencia al padre. Cada nodo se almacena con referencia al padre



```
{ _id: 'MongoDB', parent: 'Databases' },
{ _id: 'dbm', parent: 'Databases' },
{ _id: 'Databases', parent: 'Programming' },
{ _id: 'Languages', parent: 'Programming' },
{ _id: 'Programming', parent: 'Books' },
{ _id: 'Books', parent: null }
```



Diseño NoSQL

Relaciones basadas en árboles/grafos con referencia a los hijos. Por cada padre habrá una lista a los hijos que tiene

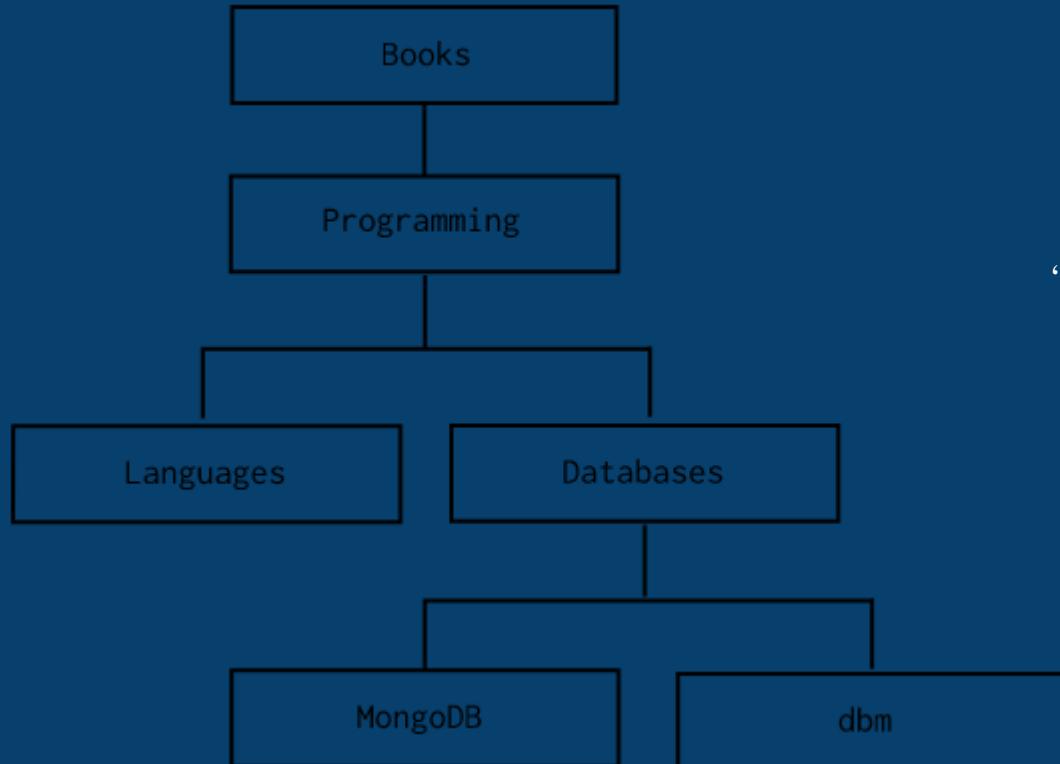


```
{ _id: 'MongoDB', children: [] },  
{ _id: 'dbm', children: [] },  
{ _id: 'Databases', children: [ 'MongoDB', 'dbm' ] },  
{ _id: 'Languages', children: [] },  
{ _id: 'Programming', children: [ 'Databases', 'Languages' ] },  
{ _id: 'Books', children: [ 'Programming' ] }
```



Diseño NoSQL

Relaciones basadas en árboles/grafos con referencia a los ancestros. Por cada nodo habrá una relación a los ancestros y padre



```
{ _id: 'MongoDB', ancestors: [ 'Books', 'Programming', 'Databases' ], parent: 'Databases' },
{ _id: 'dbm', ancestors: [ 'Books', 'Programming', 'Databases' ], parent: 'Databases' },
{ _id: 'Databases', ancestors: [ 'Books', 'Programming' ], parent: 'Programming' },
{ _id: 'Languages', ancestors: [ 'Books', 'Programming' ], parent: 'Programming' },
{ _id: 'Programming', ancestors: [ 'Books' ], parent: 'Books' },
{ _id: 'Books', ancestors: [ ], parent: null }
```



Diseño NoSQL

Relaciones basadas en árboles/grafos con referencia al camino. Se describe el camino para llegar a dicho objeto en base a las relaciones existentes

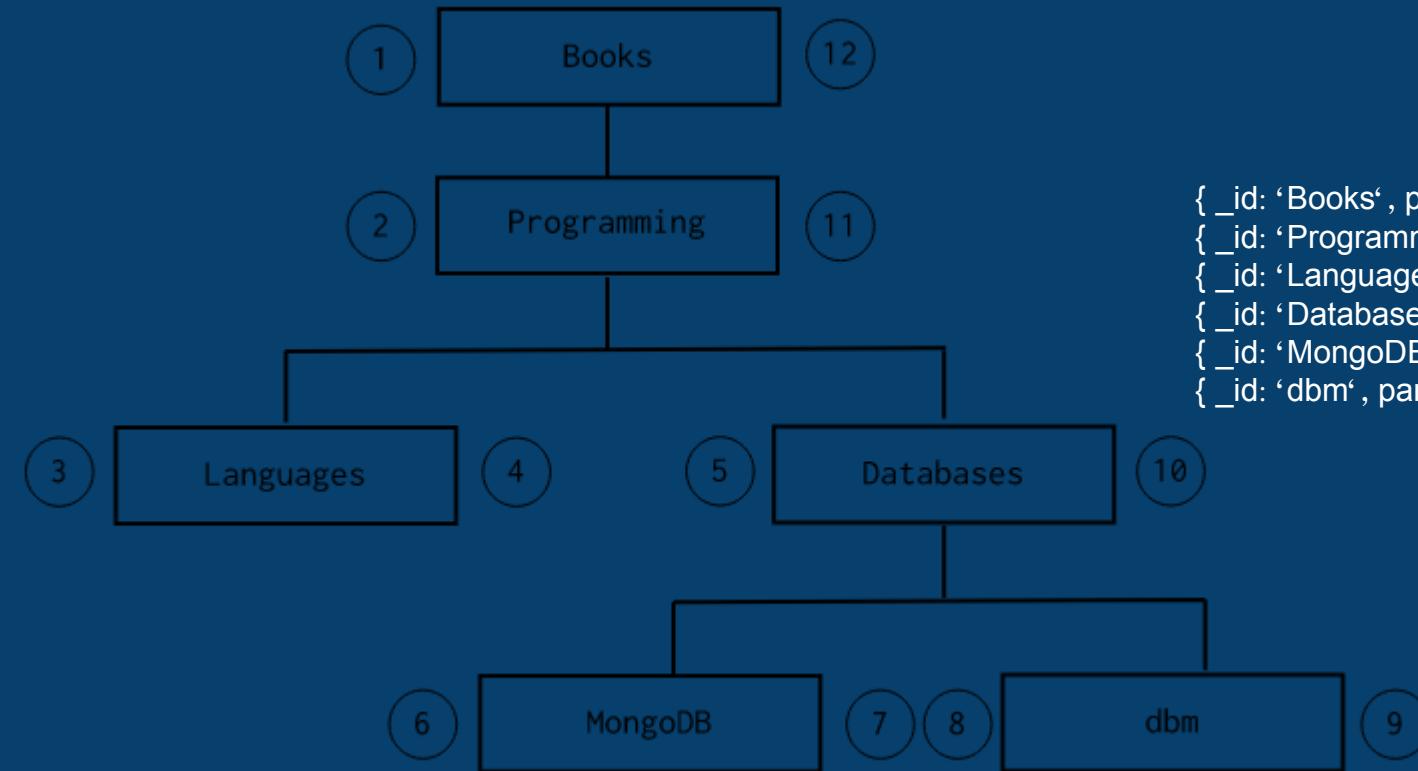


```
{ _id: 'Books', path: null },  
{ _id: 'Programming', path: 'Books', },  
{ _id: 'Databases', path: 'Books,Programming', },  
{ _id: 'Languages', path: 'Books,Programming, ' },  
{ _id: 'MongoDB', path: 'Books,Programming,Databases', },  
{ _id: 'dbm', path: 'Books,Programming,Databases,' }
```



Diseño NoSQL

Relaciones basadas en árboles/grafos con rconjuntos embebidos. Se almacenan subárboles o subgrafos



```
{ _id: 'Books', parent: null, left: 1, right: 12 },
{ _id: 'Programming', parent: 'Books', left: 2, right: 11 },
{ _id: 'Languages', parent: 'Programming', left: 3, right: 4 },
{ _id: 'Databases', parent: 'Programming', left: 5, right: 10 },
{ _id: 'MongoDB', parent: 'Databases', left: 6, right: 7 },
{ _id: 'dbm', parent: 'Databases', left: 8, right: 9 }
```



Diseño NoSQL

- **Datos anidados en documentos.** Puedes anidar objetos complejos como arrays o mapas dentro de los documentos. Esto tiene la ventaja de que la información es accesible con una única lectura de la base de datos.
 - Ventajas: Si tienes listas simples y fijas de datos que deseas conservar en tus documentos, esto es fácil de configurar y optimiza tu estructura de datos.
 - Limitaciones: No es tan escalable como otras opciones, especialmente si tus datos se expanden con el tiempo. Con listas más grandes o en crecimiento, el documento también crece, lo que puede tener como resultado que los tiempos de recuperación de los documentos sean más lentos.
 - ¿Cuál es un caso de uso posible? En una app de chat, por ejemplo, puedes almacenar las 3 salas de chat que un usuario visitó recientemente como una lista anidada en su perfil.



```
id: 321
username: "juanwmedia",
lastlogin: 1599040732,
posts: {
  post_111: {
    title: "Super post",
    username: "juanwmedia"
    content: ...
  },
  post_112: {
    title: "ABC of black cats",
    username: "juanwmedia"
    content: ...
  }
}
```

👍 Accesibilidad de la información.

👎 Nula escalabilidad.



Diseño NoSQL

- **Subcolecciones.** Puedes crear colecciones dentro de los documentos cuando tengas datos que podrían expandirse con el tiempo. De esta forma no aumentamos el tamaño del documento principal y creamos una relación padre/hijo
 - Ventajas: A medida que crecen las listas, el tamaño del documento principal no cambia. También obtienes capacidades completas de consulta en las subcolecciones y puedes emitir consultas de grupos de colecciones en todas las subcolecciones.
 - Limitaciones: No puedes borrar las subcolecciones con facilidad.
 - ¿Cuál es un caso de uso posible? En la misma app de chat, por ejemplo, puedes crear las colecciones de usuarios o mensajes dentro de los documentos de la sala de chat.



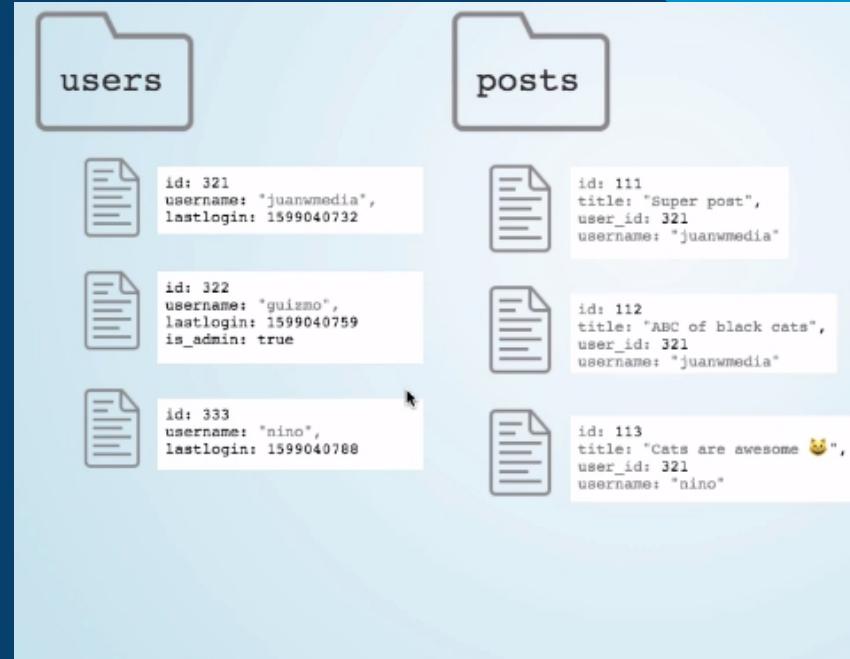
👍 Control del tamaño del documento principal.

👎 No se pueden eliminar de forma sencilla.



Diseño NoSQL

- **Colecciones de nivel de raíz.** Crea colecciones a nivel de raíz de la base de datos para organizar los conjuntos de datos dispares.
 - Ventajas: Las colecciones a nivel de raíz son buenas para las relaciones de varios a varios y proporcionan consultas eficaces dentro de cada colección.
 - Limitaciones: La obtención de los datos que son naturalmente jerárquicos puede llegar a ser cada vez más compleja a medida que crece la base de datos.
 - ¿Cuál es un caso de uso posible? En la misma app de chat, por ejemplo, puedes crear una colección para usuarios y otra para salas y mensajes.



👍 Genial para relaciones *many to many*.

👎 Mostrar la información de forma anidada o jerárquica.



Diseño NoSQL. Un ejemplo

Users table

id	username	last_login
321	"juanwmedia"	1599040732
322	"guizmo"	1599040759
333	"nino"	1599040788

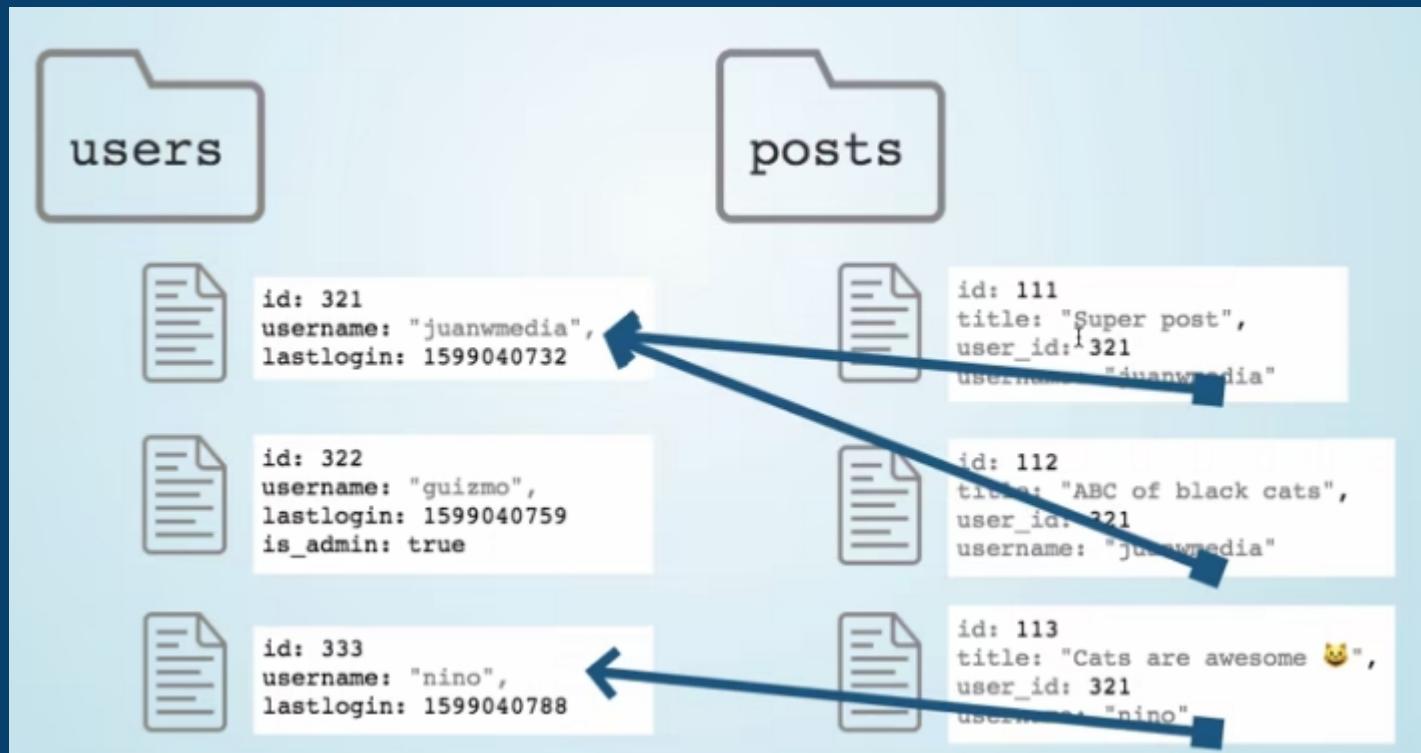
Posts table

id	title	user_id
111	"Super post"	321
112	"ABC of black cats"	321
113	"Cats are awesome 😺"	333

```
SELECT * FROM posts, users  
WHERE post.id = 112  
AND users.id = posts.user_id
```



Diseño NoSQL. Un ejemplo



```
db.users.aggregate([
  {
    $lookup:
      {
        from: "posts",
        localField: "username",
        foreignField: "username",
        as: 'my_posts'
      }
  }
])
```

```
db.posts.aggregate([
  {
    $lookup:
      {
        from: "users",
        localField: "username",
        foreignField: "username",
        as: 'user_info'
      }
  }
])
```

Bases de Datos OO

Del diagrama de clases a la persistencia



Bases de Datos OO

- Las Bases de Datos Orientadas a Objetos (BDOO) son aquellas cuyo modelo de datos está orientado a objetos, soportan el paradigma orientado a objetos almacenando métodos y datos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (P00) almacenando directamente los objetos en la BD y empleando las mismas estructuras y relaciones que los lenguajes de P00.
- Las características asociadas a las BDOO son las siguientes:
 - Los datos se almacenan como objetos.
 - Cada objeto se identifica mediante un identificador único u OID (Object Identifier), este identificador no es modificable por el usuario.
 - Cada objeto define sus métodos y atributos y la interfaz mediante la cual se puede acceder a él, el usuario puede especificar qué atributos y métodos pueden ser usados desde fuera.
 - En definitiva, un SGBDOO debe cumplir las características de un SGBD: persistencia, concurrencia, recuperación ante fallos, gestión del almacenamiento secundario y facilidad de consultas; y las características de un sistema orientado a objetos (OO): encapsulación, identidad, herencia y polimorfismo.



Bases de Datos OO

- Las **ventajas** que aporta un SGBDOO son las siguientes:
 - Mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan guardar.
 - Extensibilidad. Se pueden construir nuevos tipos de datos a partir de tipos existentes.
 - Existe una única interfaz entre el LMD (lenguaje de manipulación de datos) y el lenguaje de programación. Esto elimina el tener que incrustar un lenguaje declarativo como SQL en un lenguaje imperativo como Java o C.
 - Lenguaje de consultas más expresivo. El lenguaje de consultas es navegacional de un objeto al siguiente, en contraste con el lenguaje declarativo SQL.
 - Soporte a transacciones largas, necesario para muchas aplicaciones de bases de datos avanzadas.
 - Adecuación a aplicaciones avanzadas de bases de datos (CASE, CAD, sistemas multimedia, etc.)



Bases de Datos OO

- Entre los **inconvenientes** hay que destacar:
 - Falta de un modelo de datos universal, la mayoría de los modelos carecen de una base teórica.
 - Falta de experiencia, el uso de los SGBDOO es todavía relativamente limitado.
 - **Falta de estándares, no existe un lenguaje de consultas estándar como SQL**, aunque está el lenguaje OQL (Object Query Language) de ODMG que se está convirtiendo en un estándar de facto. Pero cada fabricante opta por su propio lenguaje lo que implica poder cambiar entre ellas y aprovechar la experiencia.
 - Competencia con los SGBDR y los SGBDOR, que tienen gran experiencia de uso.
 - La optimización de consultas compromete la encapsulación: optimizar consultas requiere conocer la implementación para acceder a la BD de una manera eficiente.
 - Complejidad: el incremento de funcionalidad provisto por un SGBDOO lo hace más complejo que un SGBDR. La complejidad conlleva productos más caros y difíciles de usar.
 - Falta de soporte a las vistas: la mayoría de SGBDOO no proveen mecanismos de vistas.
 - Falta de soporte a la seguridad.
 - Sustituidas por otros tipos de BD NoSQL como Basadas en Documentos: MongoDB, Firestore, Casandra, etc.

JPA con ObjectDB

Utilizando una BBDDOO



ObjectDB

- ObjectDB es una base de datos orientada a objetos para Java. Se puede utilizar en modo cliente-servidor y en modo incrustado (en proceso).
- A diferencia de otras bases de datos orientadas a objetos, ObjectDB no proporciona su propia API propietaria. Por lo tanto, el trabajo con ObjectDB requiere el uso de una de las dos API estándar de Java - **JPA** o **JDO**. Ambas APIs están incorporadas en ObjectDB por lo que no es necesario un software ORM intermedio.
- ObjectDB es un software multiplataforma y se puede utilizar en varios sistemas operativos con Java SE 5 o superior. Se puede integrar en aplicaciones web Java EE y Spring y desplegado en contenedores de servlets (Tomcat, Jetty), así como en servidores de aplicaciones Java EE (GlassFish, JBoss).
- El tamaño máximo de la base de datos es de 128 TB (131.072 GB). El número de objetos en una base de datos es ilimitado (excepto por el tamaño de la base de datos).⁸
- Todos los tipos persistibles de JPA y JDO son soportados por ObjectDB, incluyendo las clases de entidad definidas por el usuario, **clases insertables (embeddable) definidas por el usuario**, colecciones de Java estándar, tipos de datos básicos (valores primitivos, wrappers, String, Date, Time, Timestamp) y cualquier otra clase serializable.
- Cada objeto en la base de datos tiene un identificador único. ObjectDB admite identificadores tradicionales de bases de datos orientadas a objetos, así como claves primarias como en sistemas de gestión de bases de datos relacionales, incluyendo claves primarias compuestas y generación y asignación automática de valores,⁸ como parte de su soporte de **JPA**, que es principalmente una API para sistemas de gestión de bases de datos relacionales.



ObjectDB

- Cada objeto en la base de datos tiene un identificador único. ObjectDB admite identificadores tradicionales de bases de datos orientadas a objetos, así como claves primarias como en sistemas de gestión de bases de datos relacionales, incluyendo claves primarias compuestas y generación y asignación automática de valores,⁸ como parte de su soporte de JPA, que es principalmente una API para sistemas de gestión de bases de datos relacionales.
- ObjectDB soporta dos lenguajes de consulta. JDO Query Language (JDOQL), que se basa en la sintaxis de Java, y **JPA Query Language (JPQL)**, que se basa en la sintaxis de SQL. Las consultas por criterios (criteria queries) de JPA 2 también están soportadas.
- La evolución de esquema automática de ObjectDB maneja la mayoría de los cambios en las clases de forma transparente, incluyendo agregar y quitar campos persistentes, cambiar los tipos de campos persistentes y modificar la jerarquía de clases. Cambiar el nombre de clases persistibles y campos persistentes también está soportado.

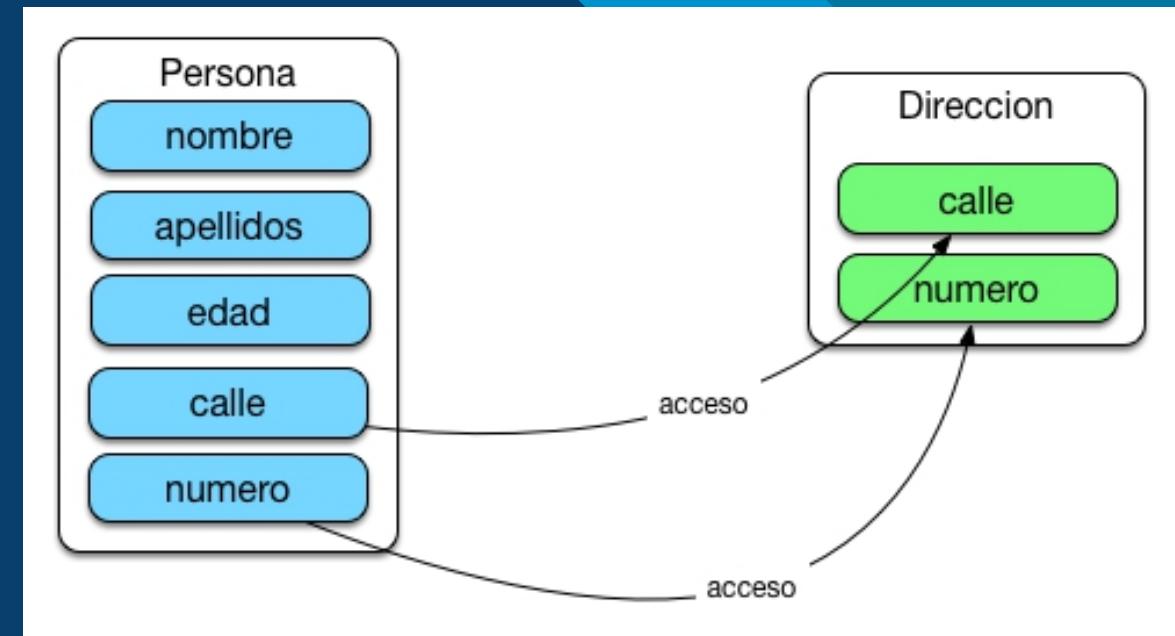
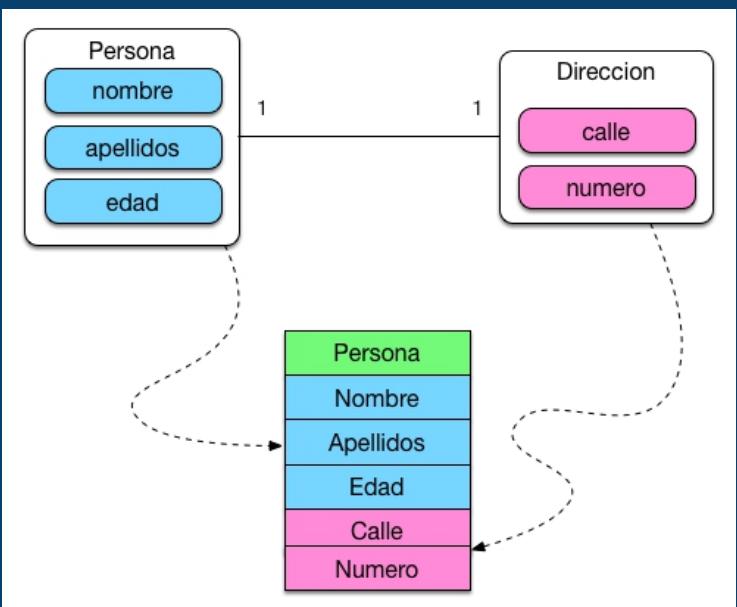
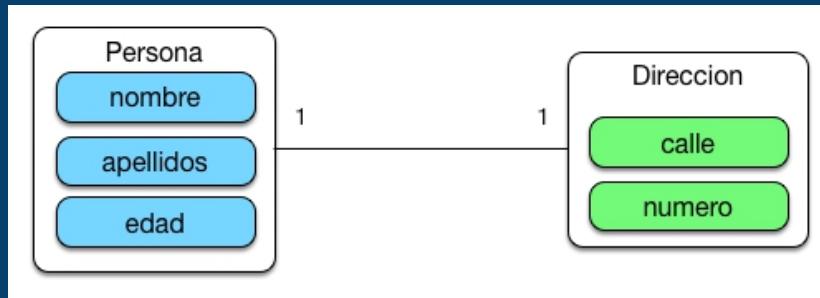


JPA y ObjectBD

- Todo lo que sabemos de JPA nos sirve y todo los tipos de datos, relaciones vista en temas anteriores, así como nuevas anotaciones para crear estructuras embebidas o jerarquizadas como hemos visto en diseño.
- También tendremos nuevas anotaciones que anteriormente (relacional no hemos usado)
 - `@Embeddable` : Con esta anotación, indicamos que la clase puede ser «integrable» dentro de una entidad. No es necesario clave primaria
 - `@Embedded` : Con esta anotación, indicamos que el campo o la propiedad de una entidad es una instancia de una clase que puede ser integrable. Es decir, para que funcione, el campo que hayamos anotado como `@Embedded`, debe corresponderse con una clase que tenga la anotación `@Embeddable`.
 - `@AttributeOverrides` : Con esta anotación, podemos sobreescribir o redefinir el mapeo de campos o propiedades de tipo básico. Esta anotación recibe un array de `@AttributeOverride`
 - `@AttributeOverride`: Con esta anotación, podemos redefinir el mapeo de tipos básicos. Su uso es `@AttributeOverride(name=>atributoDeLaClaseEmbebida», column = @Column(name=>nombreColumnaEnLaTabla»)),`
 - `@AssociationOverrides` : Con esta anotación, podemos sobreescribir o redefinir el mapeo de campos o propiedades complejos. Esta anotación recibe un array de `@AssociationOverride`
 - `@AssociationOverride`: Con esta anotación, podemos redefinir el mapeo de tipos básicos. Su uso es `@AssociationOverride(name=>atributoDeLaClaseEmbebida»,joinColumns=@JoinColumn(name=>columna_id»)),` donde el primer parámetro es el nombre del atributo de la clase embebida que estamos redefiniendo, y el segundo parámetro es un `@JoinColumn`, en donde podemos especificar por qué campo hacemos el join.



JPA y ObjectBD





JPA y ObjectBD

```
@Entity  
public class Persona {  
    @Id  
    private String nombre;  
    private String apellidos;  
    private int edad;  
  
    @Embedded  
    private Direccion direccion;  
}
```

```
transaccion.begin();  
Persona p1 = new Persona('maria', 'sanchez', 20, 'micalle', 5);  
em.persist(p1);  
transaccion.commit();  
em.close();
```

```
@Embeddable  
public class Direccion {  
    private String calle;  
    private int numero;  
}
```

```
TypedQuery<Personal> consulta=em.  
createQuery('select p from Persona p', Persona.class);  
List<Personal> lista=consulta.getResultList();  
  
for (Personal e: lista) {  
    System.out.println(e.getDireccion().getCalle());  
    System.out.println(e.getDireccion().getNumero());  
}
```



JPA y ObjectBD

- Si queremos embeber una colección tenemos dos opciones
- Embeber una colección completa de objetos:

`@ElementCollection`

```
private List<String> authors = new ArrayList<>();
```

- Embeber un colección de identificadores o índices a cada elemento embebido. Siempre que se marque como `@Indexed`
- `@IndexedEmbedded`

```
private List<Author> authors = new ArrayList<>();
```

Estas última opción tambien es aplicable para distintas relaciones: 1-1, 1-M, N-1, M-N (`@OneToOne`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`)

JSON

El formato estrella en NoSQL



JSON

- JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos.
- Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262 3rd Edition - Diciembre 1999.
- JSON es un formato de texto que es completamente independiente del lenguaje
- Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. (Fuente: <http://www.json.org/json-es.html>)
- JSON está constituido por dos estructuras:
 - Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un array asociativo.
 - Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arrays, vectores, listas o secuencias.
- Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.



JSON

- En JSON, se presentan de estas formas:

- Como un objeto, conjunto desordenado de pares nombre/valor. Un objeto comienza con { (llave de apertura) y termine con } (llave de cierre). Cada nombre es seguido por : (dos puntos) y los pares nombre/valor están separados por , (coma). En el ejemplo creo un objeto persona con nombre y oficio, y un objeto zona con su código y su nombre:

```
{ "persona": {"nombre": "Ana", "oficio": "Profesora" }}
```

```
{"zona": {"codzona": 10, "nombre": "Leganes" }}
```

- Un array, es decir, una colección de valores. Un array comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma). En el ejemplo creo el objeto persona, un array de dos elementos, no tienen por qué tener los mismos pares nombre/valor, y el objeto zona con dos zonas:

```
{"persona": [  
    {"nombre": "Alicia", "oficio": "Profesora", "ciudad": "Talavera"},
```

```
    {"nombre": "María Jesús", "oficio": "Profesora"}]]
```

```
{"zona": [  
    {"codzona": 10, "nombre": "Madrid"},
```

```
    {"codzona": 20, "nombre": "Toledo", "tasa": 15}]]
```



JSON

- Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un array. Estas estructuras pueden anidarse.
- Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape.
- Un carácter está representado por una cadena de caracteres de un único carácter. Una cadena de caracteres es parecida a una cadena de caracteres Java y JS.
- Un número es similar a un número Java y JS, excepto que no se usan los formatos octales y hexadecimales.



JSON y JAVA

- Para trabajar con JSON te recomiendo estas librerías y utilidades:
 - JSON Object de javax.json: <https://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html>
 - JSON Object de org.json: <https://www.baeldung.com/java-org-json>
 - GSon de Google: <https://github.com/google/gson>
 - Jackson de FasterXML: <https://github.com/FasterXML/jackson>
 - Kotlin JSON: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-j-s-o-n/>
- Te dejo algunos enlaces de interés
 - <https://www.baeldung.com/java-json>
 - <https://www.baeldung.com/jackson-vs-gson>
 - <https://www.baeldung.com/gson-serialization-guide>
 - <https://www.baeldung.com/gson-deserialization-guide>
 - <https://www.jsonschema2pojo.org/>
 - <https://json2csharp.com/json-to-pojo>
 - <https://www.baeldung.com/java-generate-class-from-json>

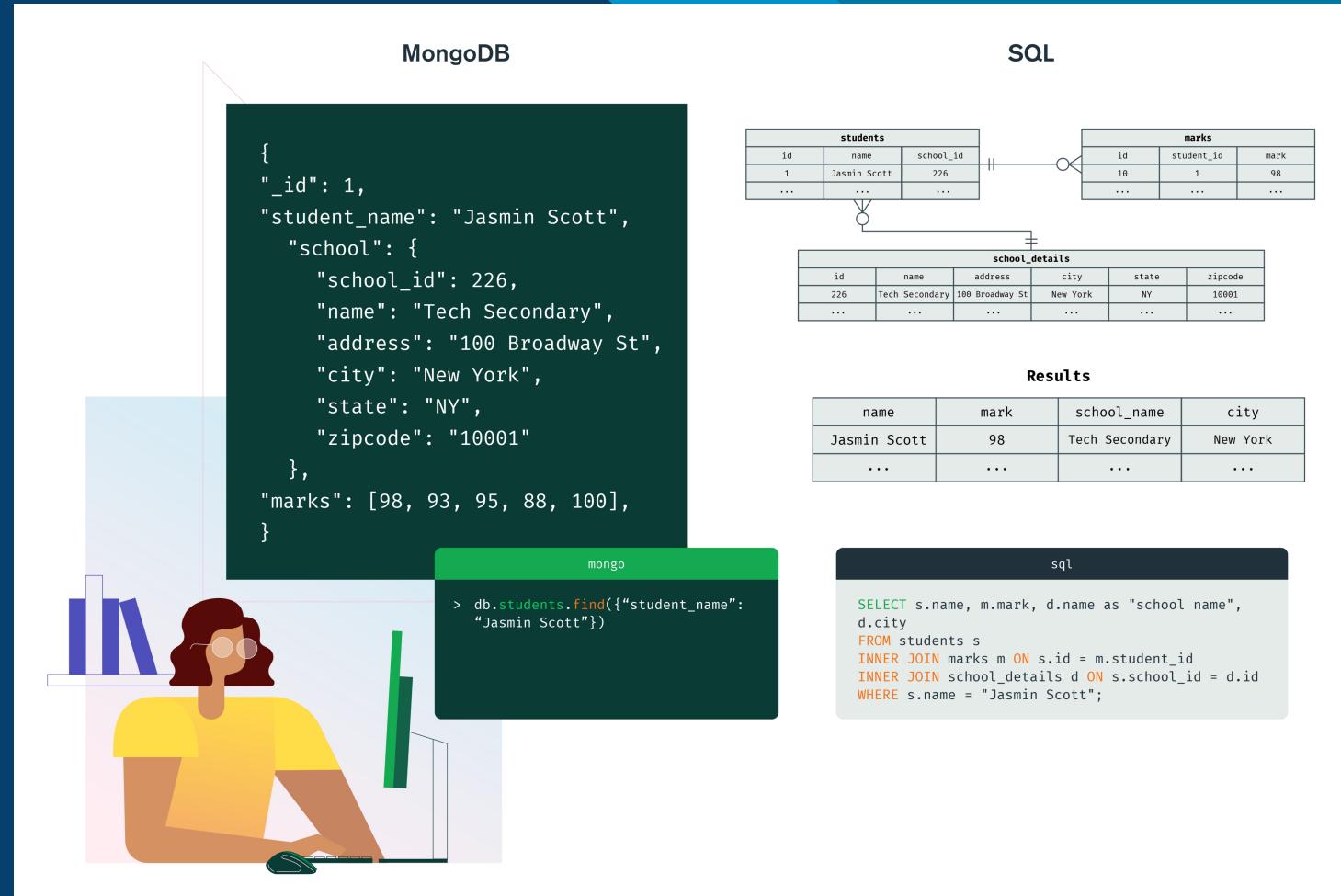
MongoDB

Nuestra BBDD NoSQL



MongoDB

- MongoDB (del inglés humongous, "enorme") es un sistema de base de datos NoSQL, orientado a **documentos** y de código abierto que nació en 2007.
- MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un **esquema dinámico**.
- Consultas ad hoc
- <https://university.mongodb.com/courses/catalog?level=Introductory>





MongoDB

- MongoDB (del inglés humongous, "enorme") es un sistema de base de datos NoSQL, orientado a **documentos** y de código abierto que nació en 2007.
- MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico.
- Consultas ad hoc
- <https://university.mongodb.com/courses/catalog?level=Introductory>



MongoDB. Características

- Consultas ad hoc: MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares. Las consultas pueden devolver un campo específico del documento pero también puede ser una función definida por el usuario para su mejor ocupación.
- Indexación: Cualquier campo en un documento de MongoDB puede ser indexado, al igual que es posible hacer índices secundarios. El concepto de índices en MongoDB es similar al empleado en base de datos relacionales..
- Replicación: MongoDB soporta el tipo de replicación primario-secundario. Cada grupo de primario y sus secundarios se denomina replica set. El primario puede ejecutar comandos de lectura y escritura. Los secundarios replican los datos del primario y sólo se pueden usar para lectura o para copia de seguridad, pero no se pueden realizar escrituras. Los secundarios tienen la habilidad de poder elegir un nuevo primario en caso de que el primario actual deje de responder.
- Balanceo de carga. MongoDB puede escalar de forma horizontal usando el concepto de shard. El desarrollador elige una clave de sharding, la cual determina cómo serán distribuidos los datos de una colección. Los datos son divididos en rangos (basado en la clave de sharding) y distribuidos a través de múltiples shard. Cada shard puede ser una réplica set. MongoDB tiene la capacidad de ejecutarse en múltiple servidores, balanceando la carga y/o replicando los datos para poder mantener el sistema funcionando en caso que exista un fallo de hardware. La configuración automática es fácil de implementar bajo MongoDB y se pueden agregar nuevas servidores a MongoDB con el sistema de base de datos funcionando.

- Almacenamiento de archivos

- MongoDB puede ser utilizado como un sistema de archivos, aprovechando la capacidad de MongoDB para el balanceo de carga y la replicación de datos en múltiples servidores. Esta funcionalidad, llamada GridFS15 e incluida en la



MongoDB. Características

- Almacenamiento de archivos: MongoDB puede ser utilizado como un sistema de archivos, aprovechando la capacidad de MongoDB para el balanceo de carga y la replicación de datos en múltiples servidores. Esta funcionalidad, llamada GridFS e incluida en la distribución oficial, implementa sobre los drivers, no sobre el servidor, una serie de funciones y métodos para manipular archivos y contenido. En un sistema con múltiple servidores, los archivos pueden ser distribuidos y replicados entre los mismos de forma transparente, creando así un sistema eficiente tolerante de fallos y con balanceo de carga.
- Agregación: MongoDB proporciona un framework de agregación que permite realizar operaciones similares al "GROUP BY" de SQL. El framework de agregación está construido como un pipeline en el que los datos van pasando a través de diferentes etapas en las cuales estos datos son modificados, agregados, filtrados y formateados hasta obtener el resultado deseado. Todo este procesado es capaz de utilizar índices si existieran y se produce en memoria. Asimismo, MongoDB proporciona una función MapReduce que puede ser utilizada para el procesamiento por lotes de datos y operaciones de agregación.
- **Ejecución de JavaScript del lado del servidor:** MongoDB tiene la capacidad de realizar consultas utilizando JavaScript, haciendo que estas sean enviadas directamente a la base de datos para ser ejecutadas.



MongoDB. Uso

- Todos los comandos para operar con esta base de datos se escriben en minúscula, los más comunes son los siguientes (<https://docs.mongodb.com/manual/tutorial/getting-started/>):
- De SQL a Mongo: <https://docs.mongodb.com/manual/reference/sql-comparison/>
- Listar las bases de datos: `show databases`
 - Mostrar la base de datos actual: `db`
 - Mostrar las colecciones de la base de datos actual: `show collections`
 - Usar una base de datos (similar a MySQL): use `nombrebasedatos`, si no existe no importa, la creará en el momento que añadamos un objeto JSON, con las funciones `.save` o `.insert`.
 - Si queremos saber el número de documentos dentro de las colecciones, utilizaremos la función `count`, escribiremos: `db .nombre_coleccidn.count ()`. También se utilizan las funciones `size()` y `length()`.
 - Para añadir comentarios utilizamos los caracteres `//` de comentario de JS.



MongoDB. Uso

- Insertar:
 - db.nombre colección.save(dato JSON)
 - db.nombre colección.insert(dato JSON)
- Consultar
 - db.nombre colección.find(filtro, campos) o findOne()
 - Si se desea que la salida sea ascendente por uno de los campos, utilizamos el operador .sort, por ejemplo, para obtener los datos de la colección ordenados por nombre escribimos: db.amigos.find().sort({nombre:1}); El número que acompaña a la orden indica el tipo de ordenación, 1 ascendente y - descendente.
 - En filtro indicamos la condición de búsqueda, podemos añadir los pares nombre: valor a buscar. Si omitimos este parámetro devuelve todos los documentos, o pasa un documento vacío (). En campos se especifican los campos a devolver de los documentos que coinciden con el filtro de la consulta. Si se desean devolver uno o más campos escribiremos (nombre campo1: 1, nombre campo2: 1,). Si no se desean que se seleccionen los campos escribimos. {nombre campo1: 0, nombre campo2: 0, ...} También podemos poner true o false en lugar de 1 o 0.
 - db.amigos.find({nombre : "Marleni"}); db.amigos.find({nombre : "Marleni."},{teléfono:1})
 - Si deseamos buscar el nombre y la nota de los alumnos de 1DAM escribiremos: db.amigos.find({curso : "1DAM"}, {nombre:1, nota:1})
 - Si queremos saber el número de registros que devuelve una consulta pondremos: db.nombre_colección.find({filtros}).count(),
 - Si queremos limitar, podemos usar limit()



MongoDB. Uso

- Selectores de búsqueda de comparación
 - \$eq, igual a un valor. Esta orden obtiene los documentos con nota = 6: db.amigos.find({ nota : { \$eq : 6 } })
 - \$gt, mayor que y \$gte mayor o igual que. Esta orden obtiene los documentos con nota >= 6: db.amigos.find({ nota : { \$gte : 6 } })
 - \$lt, menor que y \$lte, menor o igual que. El ejemplo muestra los amigos de 1DAM con notas entre 7 y 9 incluidas, preguntamos por un intervalo >=7 y <=9: db.amigos.find({curso : n1DAM", nota : { \$gte : 7, \$lte : 9} })
 - \$ne, distinto a un valor. El siguiente ejemplo obtiene los documentos con nota distinta de 7: db.amigos.find({ nota : { \$ne : 7 } })
 - \$in, entre una lista de valores y \$nin, no está entre la lista de valores. En el ejemplo se obtienen los documentos cuya nota sea uno de estos valores: 5,7 y 8: db.amigos.find({ nota : { \$in : [5, 7, 8] } })
 - Ahora añado el nombre y el curso: db.amigos.find({ nota : { \$in : [5, 7, 8] } },{nombre:1, curso:1})



MongoDB. Uso

- Selectores de búsqueda lógicos
 - \$or. La siguiente orden obtiene los documentos de los cursos 1DAM , o los que tienen nota > de 7: db.amigos.find({ \$or : [{ nota: { \$gt: 7 }}, {curso : "n1DAM"}] }). Esta consulta obtiene los amigos con nombre Ana o Marleni: db.amigos.find({ \$or: [{nombre : "Ana"}, {nombre: "Marleni"}] })
 - \$and. Este operador se maneja implícitamente, no es necesario especificarlo. Las siguientes órdenes hacen lo mismo, obtienen los amigos del curso 2DAM y con nota 6: db.amigos.find({ \$and : [{curso : "2DAM"}, {nota : 6} 1]}); db.amigos.find({curso : "2DAM", nota : 6}). Esta otra consulta devuelve el documento con nombre Marleni y con teléfono 3446500: db.amigos.find({nombre : "Marleni", teléfono : 3446500}); db.amigos.find({ \$and : [{nombre:"Marleni"},{teléfono:3446500}] })
 - \$not. Representa la negación, en el ejemplo obtengo los amigos con nota no mayor de 7. db.amigos.find({ nota : { \$not: { \$gt: 7 } } }). Esta otra consulta visualizará el nombre, el curso y la nota de los que su nota no mayor de 7: db.amigos.find({ nota : { \$not: \$gt: 7 } }),{_id:0, nombre:1, curso:1, nota:1})
 - \$exists, este operador booleano permite filtrar la búsqueda tomando en cuenta la existencia del campo de la expresión. Este ejemplo obtiene los registros que tengan nota. db.amigos.find({ nota : {\$exists:true} })



MongoDB. Uso

- Operaciones CRUD

- Insertar: insert(), insertOne(), insertMany()
- Actualizar: update(filtro, cambios): db.amigos.update({nombre:"Ana"},{nombre: "Ana María" }). Midificadores upsert — si asignamos true a este parámetro, se indica que si el filtro de búsqueda no encuentra ningún resultado, entonces, el cambio debe ser insertado como un nuevo registro. multi — en caso de que el filtro de búsqueda devuelva más de un resultado, si especificamos este parámetro a true, el cambio se realizará a todos los resultados, de lo contrario solo se cambiará al primero que encuentre, es decir, al que tenga menor identificativo de objeto, “_id”.
 - save({_id, data}), reemplaza el documento, por el nuevo
 - findOneAndUpdate(filtro, cambios)
 - updateOne(filtro, cambios)
 - Operadores de modificación:
 - \$set, permite actualizar con nuevas propiedades a un documento (o conjunto de documentos).
 - \$unset, permite eliminar propiedades de un documento. Por ejemplo, borro la edad 34 de
 - \$inc, incrementa en una cantidad numérica especificada en el valor del campo a incrementar.
 - \$rename, renombra campos del documento.



MongoDB. Uso

- Operaciones CRUD
 - Modificaciones para arrays
 - \$push, añade un elemento a un array. Este ejemplo añade el tema MongoDB al libro con db.libros.update({ codigo:1 },{ \$push : {temas: "MongoDB" } })
 - \$addToSet, agrega elementos a un array solo si estos no existen. En el ejemplo se añade el tema Base de datos a todos los libros que no lo tengan. Primero preguntamos si el libro tiene el campo temas. Para que se añada a todos los libros indicamos multi:true: b.libros.update({ temas : { \$exists:true } },{ \$addToSet: {temas:"Base de datos" } }, {multi:true})
 - \$each, se usa en conjunto con \$addToSet o \$push para indicar que se añaden varios elementos al array.
db.libros.update({codigo:1},{push:{temas:{Seach:["JSON","XML"]}}}); db.libros.update({codigo:2},{ \$addToSet :{temas: { \$each: ["Eclipse","Developper"] }}})
 - \$pop, elimina el primer o último valor de un array. Con valor -1 borrar el primero, con otro valor el último. En el ejemplo se borra el primer tema del libro con código 3: db.libros.update({codigo:3},{\$pop: {temas:-1}})
 - \$pull, elimina los valores de un array que cumplan con el filtro indicado. En el ejemplo se borran de todos los libros los elementos 'Base de datos' y "JSON", si los tienen: db.libros.update({}, {\$pull:{ temas: { \$in: ["Base de datos","JSON"] }}}), { multi: true })
 - Eliminación de datos. remove(filtro): db.amigos.remove({nombre : "Marleni"});



MongoDB. Uso

- La agregación pipeline o tuberías de agregación se basa en someter una colección a un conjunto de operaciones o etapas, estas etapas irán convirtiendo y transformando el conjunto de documentos pertenecientes a la colección, hasta obtener un conjunto de documentos con el resultado deseado.
- Se le llama tubería ya que cada etapa irá modificando, moldeando y calculando la estructura de los documentos para pasarlos a la etapa que le sigue. Las etapas son las siguientes:
 - \$project – Cambia la forma del documento. La proyección permite modificar la representación de los datos, por lo que en general se emplea para darles una nueva forma con la que resulte más cómodo trabajar. 1-1.
 - \$match – Filtra los resultados. La etapa match permite filtrar los documentos para que en el resultado de la etapa solo estén aquellos que cumplen ciertos criterios. Se puede filtrar antes o después de agregar los resultados, en función del orden en que definamos esta etapa. n-1
 - \$group – Agrupación. Permite agrupar distintos documentos según compartan el valor de uno o varios de sus atributos, y realizar operaciones de agregación sobre los elementos de cada uno de los grupos. Se utilizan las funciones sum, max, min, avg, etc. 1-1
 - \$sort – Ordenación de documentos. 1-1
 - \$skip – Salta n documentos. n-1
 - \$limit – Elige n elementos para el resultado. N-1.
 - \$unwind – Normaliza arrays 1-n.



MongoDB. Uso

- Algunas de las muchas funciones de agregado
 - \$sum: Suma el elemento indicado en toda las instancias de las colecciones: db.mycol.aggregate([{\$group : {_id : "\$by_user", numTutorial : {\$sum : "\$likes"}}}])
 - \$avg: calcula la media del elemento indicado: db.mycol.aggregate([{\$group : {_id : "\$by_user", numTutorial : {\$avg : "\$likes"}}}])
 - \$min: obtiene el mínimo de la colección del elemento indicado: db.mycol.aggregate([{\$group : {_id : "\$by_user", numTutorial : {\$min : "\$likes"}}}])
 - \$max: obtiene el máximo de la colección del elemento indicado: db.mycol.aggregate([{\$group : {_id : "\$by_user", numTutorial : {\$max : "\$likes"}}}])
 - \$push: inserta el valor en el array: db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push: "\$url"}}}])
 - \$addToSet: inserta el valor en el array evitando duplicados: db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}])
 - \$first: obtiene el primero: db.mycol.aggregate([{\$group : {_id : "\$by_user", firstUrl : {\$first : "\$url"}}}])
 - \$last: obtiene el último: db.mycol.aggregate([{\$group : {_id : "\$by_user", lastUrl : {\$last : "\$url"}}}])



MongoDB. Uso

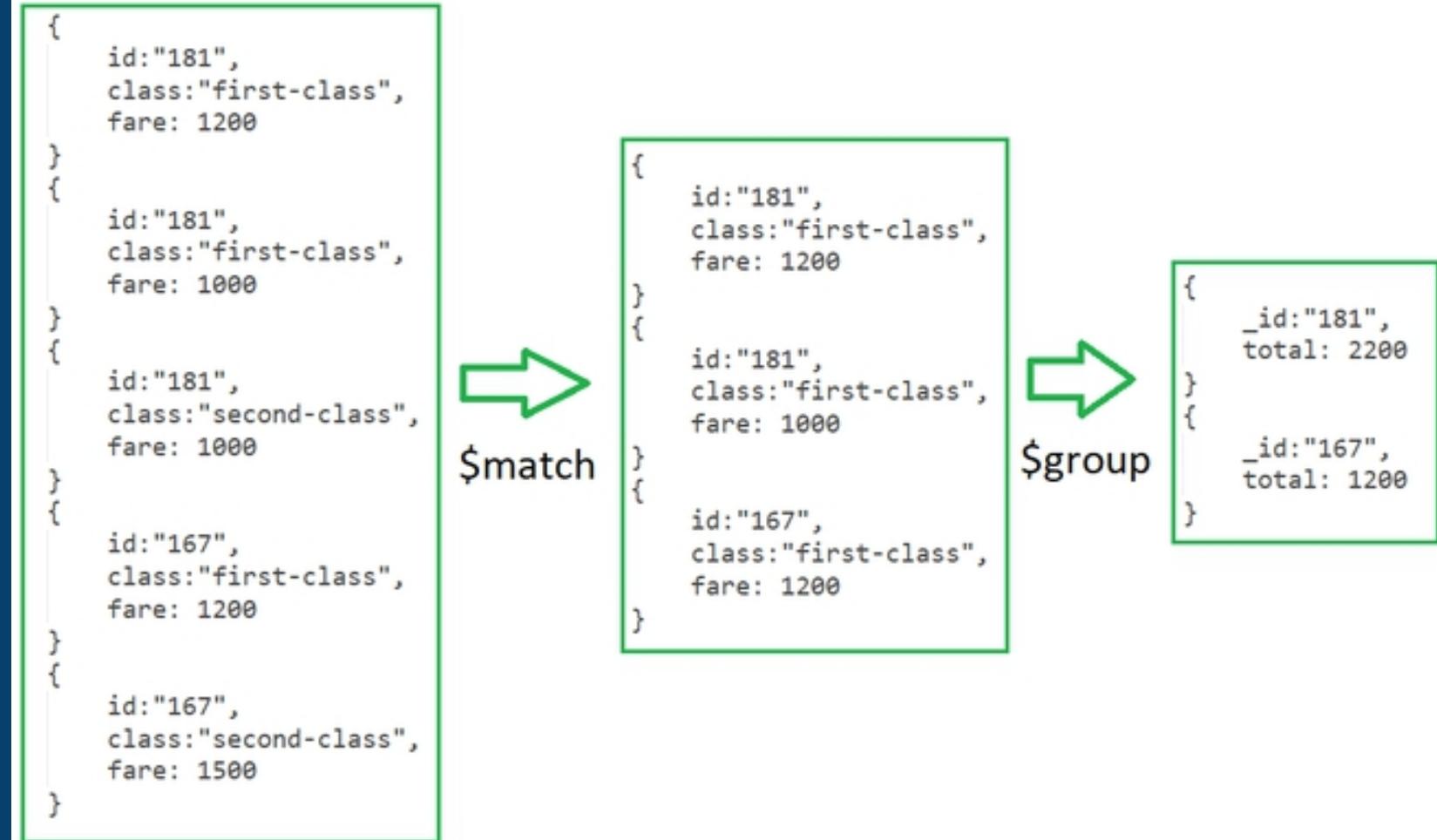
- De SQL a Agregaciones Mongo: <https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/>
 - WHERE -> \$match
 - GROUP BY -> \$group
 - HAVING -> \$match
 - SELECT -> \$project
 - ORDER BY -> \$sort
 - LIMIT -> \$limit
 - SUM() -> \$sum
 - COUNT() -> \$sum / \$sortByCount
 - join -> \$lookup
 - SELECT INTO NEW_TABLE -> \$out
 - MERGE INTO TABLE -> \$merge
 - UNION ALL -> \$unionWith (Available starting in MongoDB 4.4)

MongoDB. Uso

- Agregaciones

```
db.train.aggregate( [  
    { $match:{class:"first-class"}},  
    { $group:{_id:"id",total:{$sum:"$fare"}}}  
])
```

} pipeline stages





MongoDB. Uso

- En la siguiente consulta obtenemos por cada categoría el número de artículos, el total unidades vendidas de artículos, y el total importe, la suma de los pvp*unidades. Es como una select con group by. En este caso se utiliza la etapa \$group, cuando se utiliza esta etapa se debe añadir el identificador de objeto _id, en este caso como agrupamos por categoría lo indicamos en el _id. Que a su vez será el identificador del resultado. Para contar artículos se utiliza la función \$sum, sumando 1:
 - db.articulos.aggregate([{ \$group: { id: "\$categoría", contador: { \$sum: 1 }, sumaunidades: { \$sum: "\$uv"}, totalimporte: { \$sum: { \$multiply: ["\$pvp", "\$uvl"} } } }])
- En la siguiente consulta obtenemos el número de documentos de la categoría Deportes, el total de unidades vendidas de sus artículos, el total importe y la media de unidades vendidas. Se utilizan las etapas \$match para seleccionar la categoría, y luego \$group para obtener resultados agrupados, en este caso en _id ponemos cualquier valor:
 - db.articulos.aggregate([{ \$match: { categoría: "Deportes" } } , { \$group: { id: "deportes", contador: { \$sum: 1 }, sumaunidades: { \$sum: "\$uv"}, media: { \$avg: "\$uv"}, totalimporte: { \$sum: { \$multiply: ["\$pvp", "\$uv"] }}}}])
- En la siguiente consulta obtenemos por cada categoría el artículo con el precio más caro. Para ello primero ordenamos descendente por pcategoría, pvp y denominación, utilizando la etapa \$sort. Y el resultado obtenido se agrupa con \$group para luego obtener el primero de cada categoría con la función \$first:
 - db.articulos.aggregate({ \$sort: { categoría: -1, pvp: -1, denominación: -1 } }, { \$group: { _id: "\$categoría", mascaro: { \$first: "\$denominación" }, precio: { \$first: "\$pvp" } }}})



MongoDB. Uso

- Joins.

- Manuales, por un campo: `emplesdep = db.emple.find({ _id: { $in : departrabajo.emple } })`
- Manuales, si existe en un array: `emplesdep = db.emple.find({ _id: { $in : departrabajo.emple } }).toArray()`
- Usando `$lookup`

```
{  
  lookup:  
  {  
    from:leccionDestino,  
    localField: campoLocal,  
    foreignField: campoReferenciado,  
    as: campoSalida  
  }  
}
```

```
SELECT *, campoSalida  
FROMleccionLocal  
WHERE campoSalida IN (  
  SELECT *  
  FROM destino  
  WHERE campoReferencia =  
 leccionLocal.campoLocal  
);
```



MongoDB. Uso

- Joins.

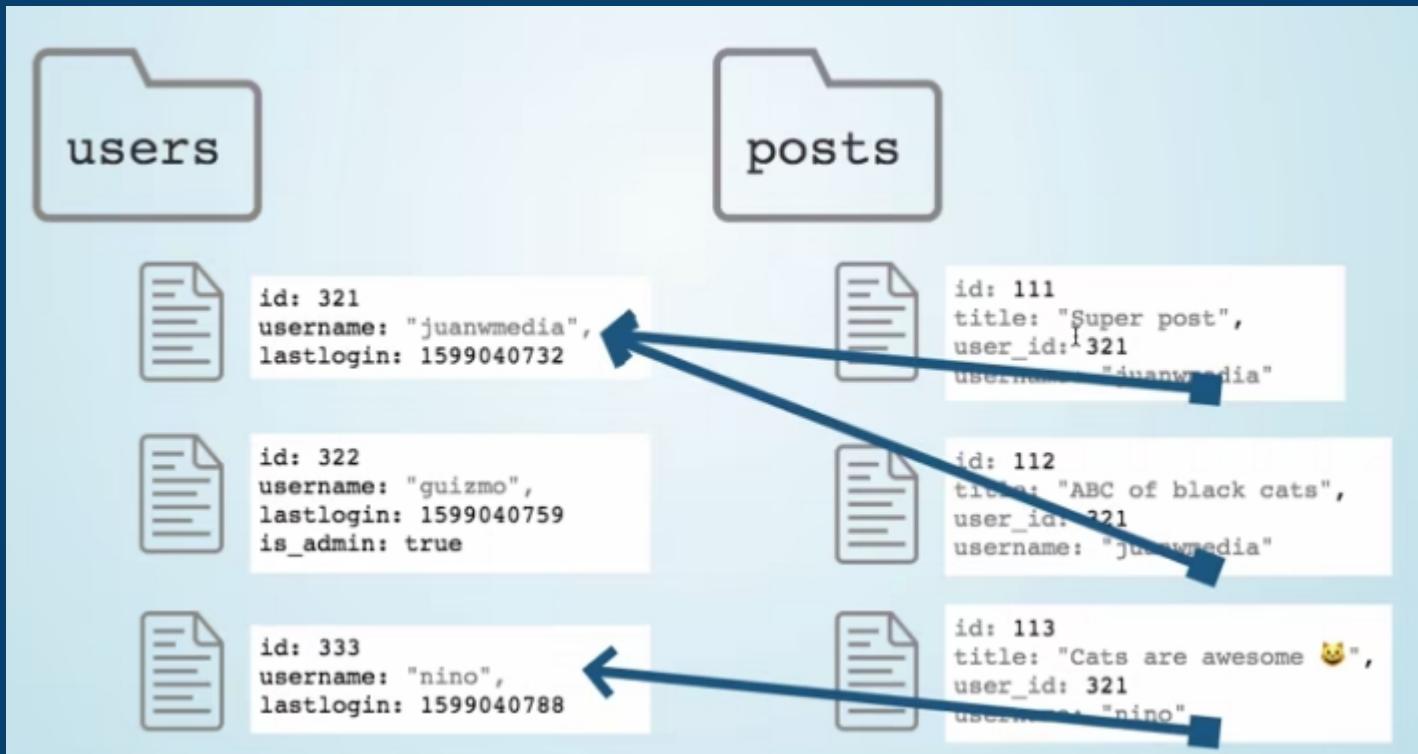
- Manuales, por un campo: `emplesdep = db.emple.find({ _id: { $in : departrabajo.emple } })`
- Manuales, si existe en un array: `emplesdep = db.emple.find({ _id: { $in : departrabajo.emple } }).toArray()`
- Usando `$lookup`: <https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/>

```
{  
  lookup:  
  {  
    from: colecciónDestino,  
    localField: campoLocal,  
    foreignField: campoReferenciado,  
    as: campoSalida  
  }  
}
```

```
SELECT *, campoSalida  
FROM colecciónLocal  
WHERE campoSalida IN (  
  SELECT *  
  FROM destino  
  WHERE campoReferencia =  
  colecciónLocal.campoLocal  
);
```

MongoDB. Uso

Joins



```
db.users.aggregate([
  {
    $lookup:
      {
        from: "posts",
        localField: "username",
        foreignField: "username",
        as: 'my_posts'
      }
  }
])
```

```
db.posts.aggregate([
  {
    $lookup:
      {
        from: "users",
        localField: "username",
        foreignField: "username",
        as: 'user_info'
      }
  }
])
```

MongoDB con Java

Usando MongoDB con Java



MongoDB y Java

- Usamos las librerías oficiales de MongoDB para Java: mongodb-driver
- Podemos trabajar con los objetos BSON
- Podemos elegir si trabajar de una manera “Tipada con Pojos” vs “No Tipada con BSON”
- Funciones y métodos vistos en anterioridad con Mongo: insert, inserMany, update, find, agregados, etc.
- Referencias:
 - <https://www.mongodb.com/developer/language/java/>
 - <https://www.mongodb.com/developer/quickstart/java-setup-crud-operations/>
 - <https://www.mongodb.com/developer/quickstart/java-mapping-pojos/>
 - <https://www.mongodb.com/developer/quickstart/java-aggregation-pipeline/>
- Ver proyectos de ejemplos con un controlador propio de Mongo

MongoDB con JPA

Trabajando con MongoDB y Java



MongoDB y JPA

- Para trabajar con MongoDB usando JPA haremos uso de la implementación Hibernate OGM
- De nuevo tenemos las anotaciones conocidas y otras nuevas
- Importante no confundir las anotaciones del modelo relacional con las anotaciones NoSQL, aunque pueden llegar a confusión.
- Para muchas consultas es mejor usar Native Query con las consultas MongoDB que JPQL
- Es importante aplicar un buen diseño para justificar las anotaciones, relaciones, dependencias y navegación más adecuadas a nuestro problema.
- Referencias:
 - https://docs.jboss.org/hibernate/ogm/5.4/reference/en-US/html_single/#ogm-gettingstarted
 - <https://hibernate.org/ogm/documentation/getting-started/>
 - <https://www.baeldung.com/hibernate-ogm>
 - <https://itexpertsconsultant.wordpress.com/2016/02/27/mongodb-with-hibernate-orm/amp/>
- Ver proyectos de ejemplos usando Mongo con JPA

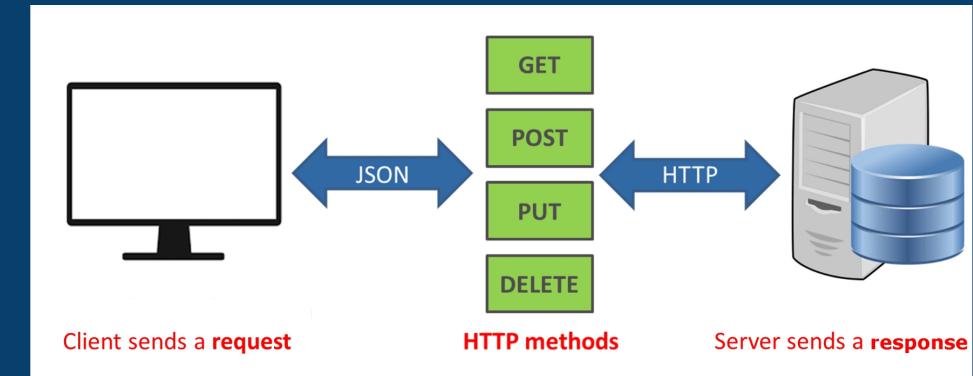
Servicios Web REST

Accediendo a datos sobre HTTP



Servicios Web REST

- Un servicio web es un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático, a través de la que es posible interactuar con el mismo mediante el **intercambio de mensajes sobre HTTP**.
- **Servicios Web REST.** En este caso, el servicio expone una serie de recursos, a los que se accede con una simple petición HTTP. Tanto los datos enviados en la llamada como los generados en la respuesta, pueden estar en cualquier formato, siendo JSON el más habitual.
- **REST transporta datos por medio del protocolo HTTP**, pero este permite utilizar los diversos métodos que proporciona HTTP para comunicarse, como lo son los HTTP Verbs GET, POST, PUT, DELETE, PATCH y a la vez, utiliza los códigos de respuesta nativos de HTTP (404,200,204,409).
- Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: POST (crear), GET (leer y consultar), PUT (editar) y DELETE (eliminar).
- Trabajaremos con las URIs para redireccionar y seleccionar el recurso con el que vamos a operar
- <http://api.example.com/users/>
- <http://api.example.com/users/{id}>





Servicios Web REST

- **HTTP verbs:** Si realizamos CRUD, debemos utilizar los HTTP verbs de forma adecuada para cuidar la semántica.
 - *GET*: Obtener datos. Ej: GET /v1/empleados/1234
 - *PUT*: Actualizar datos. Ej: PUT /v1/empleados/1234
 - *POST*: Crear un nuevo recurso. Ej: POST /v1/empleados
 - *DELETE*: Borrar el recurso. Ej: DELETE /v1/empleados/1234
 - *¿PATCH?*: Para actualizar ciertos datos
- **Nombre de los recursos.** Plural mejor que singular, para lograr uniformidad: Uri cortas sin espacios y evitar guiones y que represente el recurso
 - Obtenemos un listado de clientes: GET /v1/clientes
 - Obtenemos un cliente en particular: GET /v1/clientes/1234
- **Siempre se debe devolver un código de estado HTTP con los requests.**
- **Formato de salida.** Nos fijaremos en el ACCEPT HEADER. Preferiblemente JSON. XML no es nuestro amigo: schemas, namespaces...



Servicios Web REST

- Otro componente de un RESTful API es el «HTTP Status Code», que le informa al cliente o consumidor del API que debe hacer con la respuesta recibida. Estos son una referencia universal de resultado, es decir, al momento de diseñar un RESTful API toma en cuenta utilizar el «Status Code» de forma correcta. Los código de estado más utilizados son:
 - 200 OK
 - 201 Created (Creado)
 - 304 Not Modified (No modificado)
 - 400 Bad Request (Error de consulta)
 - 401 Unauthorized (No autorizado)
 - 403 Forbidden (Prohibido)
 - 404 Not Found (No encontrado)
 - 422 (Unprocessable Entity (Entidad no procesable)
 - 500 Internal Server Error (Error Interno de Servidor)



Servicios Web REST

- A la hora de consultar/manipular los datos a través de REST es importante conocer qué queremos conseguir y cómo están estructurados los datos, así como endpoint a atacar.
- También podemos apoyarnos en herramientas como Postman

The image shows two side-by-side screenshots of API documentation and a testing tool.

Left Side (Swagger UI):

- productos-rest-controller** Products REST Controller
- API Endpoints:
 - GET /productos findAll
 - POST /productos create
 - GET /productos/{id} findById
 - PUT /productos/{id} update
 - DELETE /productos/{id} delete
 - GET /test hola
- Models:
 - Producto >

Right Side (Postman):

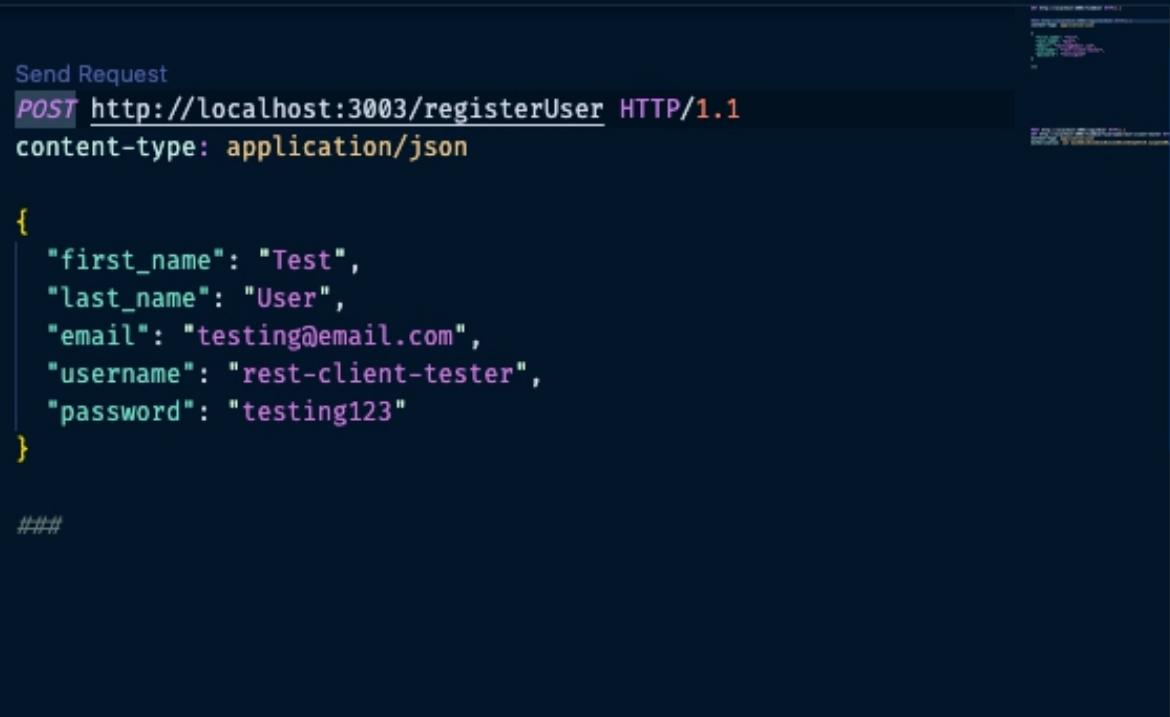
- GET Listar Todos**
- Request URL: http://localhost:8080/productos
- Method: GET
- Params tab (selected):
 - Query Params:

KEY	VALUE	DESCRIPTION
Key	Value	Description
 - Headers (7): [List of headers]
 - Body
 - Pre-request Script
 - Tests
 - Settings
- Body tab:
 - Pretty
 - Raw
 - Preview
 - Visualize BETA
 - JSON

```
1: [
2:   {
3:     "id": 1,
4:     "nombre": "producto 1"
5:   },
6:   {
7:     "id": 2,
8:     "nombre": "producto 2"
9:   },
10:  {
11:    "id": 3,
12:    "nombre": "producto Actualizado"
13:  }
14: ]
```
- Status: 200 OK Time: 459ms Size: 423 B Save Response



Servicios Web REST



```
Send Request
POST http://localhost:3003/registerUser HTTP/1.1
content-type: application/json

{
  "first_name": "Test",
  "last_name": "User",
  "email": "testing@email.com",
  "username": "rest-client-tester",
  "password": "testing123"
}

###
```

```
1  HTTP/1.1 200 OK
2  Access-Control-Allow-Origin: *
3  X-DNS-Prefetch-Control: off
4  X-Frame-Options: SAMEORIGIN
5  Strict-Transport-Security: max-age=15552000; includeSubDomains
6  X-Download-Options: noopen
7  X-Content-Type-Options: nosniff
8  X-XSS-Protection: 1; mode=block
9  Content-Type: application/json; charset=utf-8
10 Content-Length: 26
11 ETag: W/"1a-ASMfoApUJ3y9JnZZqDmv+/C5hog"
12 Date: Tue, 13 Oct 2020 20:45:10 GMT
13 Connection: close
14
15 {
16   "message": "user created"
17 }
```



Servicios Web REST

- Para consumir un servicio REST podemos apoyarnos en cualquier librería que procese peticiones HTTP, pues solo son eso, no hay nada extraño.
- También podemos hacer uso de librerías como Retrofit: <https://square.github.io/retrofit/>
- Ejemplo de Clase: <https://github.com/joseluisgs/api-client-acceso-datos>



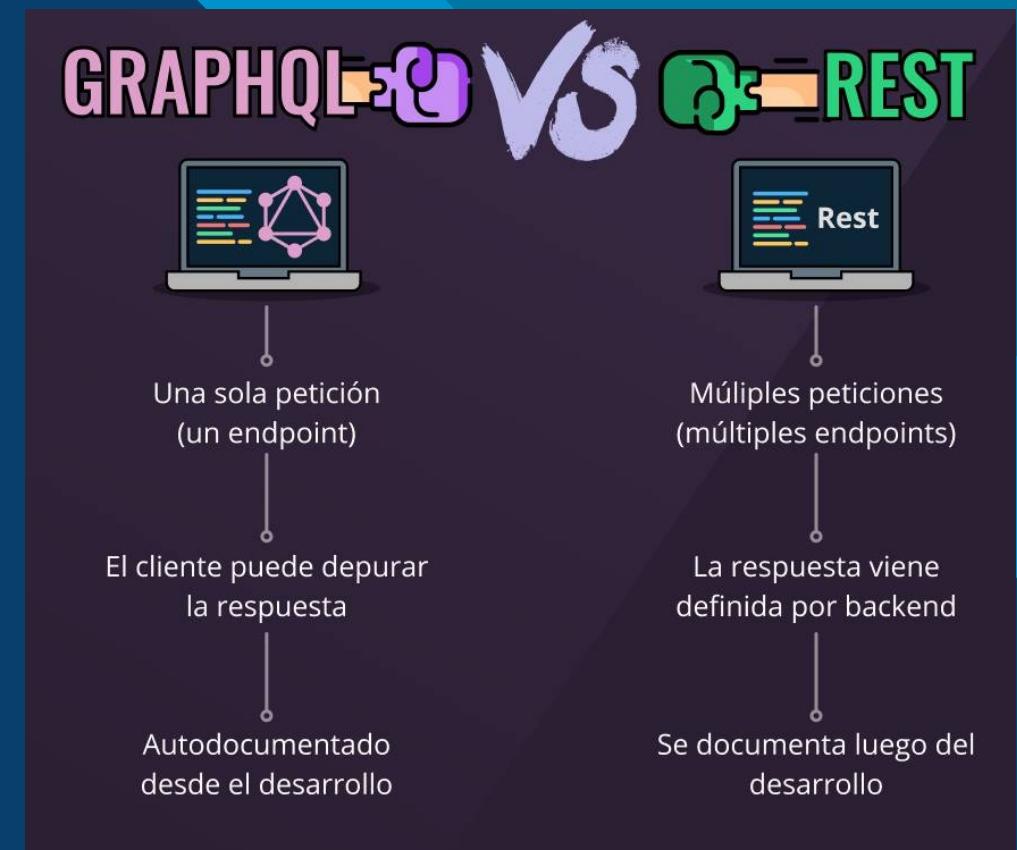
Servicios Web GraphQL

Accediendo a datos sobre HTTP



Servicios Web GraphQL

- Como bien sabemos es un servicio Web que usa HTTP, y es una alternativa o complemento a REST para consultar tus APIs. Pues GraphQL (de Query Languaje) te propone explorar los datos y sus relaciones a traves de una simple petición en base a los esquemas existentes.
- GraphQL existe un único endpoint o ruta el cual suele recibir el nombre /graphql. apartir de aqui solo lanzamos peticiones POST usando el lenguaje de consulta transportado en JSON.
- Importante, siempre nos devuleve como código de estado 200. Es decir, si hay error, debemos explorarlo en el cuerpo de la respuesta.
- Con GraphQL solventa Over Fetching hay situaciones en las que no necesitamos toda la información y acabamos ignorando muchos de los datos, lo cual es indicativo de que realmente no estamos siendo del todo eficientes. Este problema hace que tengamos una carga de datos más lenta y consumamos más ancho de banda, o en el caso de los móviles consumamos más datos. El Under Fetching se da cuando nos faltan datos y requiere de otra petición adicional que nos devuelva los datos que nos falta. En todo caso consultamos lo que necesitemos y de la manera que lo necesitemos.





Servicios Web GraphQL

- Dos conceptos claves: Consultas y Mutaciones.
- **Las Consultas o Queries**, nos sirven para consultar. Con un JSON indicamos la entidad, los campos que necesitemos y podemos explorar las relaciones de los mismos. De la misma manera, en una sola petición podemos pedir los datos que queramos, distintas consultas o todas las relaciones que necesitemos al nivel de detalles que necesitemos.

```
{  
  user(id: 4802170) {  
    id  
    name  
    isViewerFriend  
    profilePicture(size: 50) {  
      uri  
      width  
      height  
    }  
    friendConnection(first: 5) {  
      totalCount  
      friends {  
        id  
        name  
      }  
    }  
  }  
}  
  
{  
  "data": {  
    "user": {  
      "id": "4802170",  
      "name": "Lee Byron",  
      "isViewerFriend": true,  
      "profilePicture": {  
        "uri": "cdn://pic/4802170/50",  
        "width": 50,  
        "height": 50  
      },  
      "friendConnection": {  
        "totalCount": 13,  
        "friends": [  
          {  
            "id": "305249",  
            "name": "Stephen Schwink"  
          },  
          {  
            "id": "3108935",  
            "name": "Nathaniel Roman"  
          }  
        ]  
      }  
    }  
  }  
}
```



Servicios Web GraphQL

- Dos conceptos claves: Consultas y Mutaciones.
- **Las Mutaciones** nos sirven para realizar cambios en las entidades o en nuestro sistema.
- Mas info en: <https://graphql.org/learn/>

The screenshot shows a GraphQL playground interface with the following components:

- Header:** Includes a play button, a "Logout" button, and a "Docs" link.
- Left Panel (Query Editor):** Contains the GraphQL mutation code and its variables.
- Right Panel (Results):** Displays the JSON response from the server.
- Bottom Panels:** Includes sections for "LOGS" and "QUERIES".

GraphQL Mutation (Left Panel):

```
1 mutation mutateMeeting(
2   $companyId: String!,
3   $meeting: MeetingInput!
4 ) {
5   mutateMeeting(
6     companyId: $companyId
7     meeting: $meeting
8   ) {
9     id,
10    start,
11    end
12  }
13}
```

Query Variables (Left Panel):

```
1 {
2   "companyId": "Company-Test",
3   "meeting": {
4     "start": "start",
5     "end": "end",
6     "agreements": [
7       {
8         "name": "agreement 1"
9       },
10      {
11        "name": "agreement 2"
12      }
13    ]
14  }
15}
```

JSON Response (Right Panel):

```
{
  "data": {
    "mutateMeeting": {
      "id": "Meeting-f065730c-221f-4edb-9427-c83a96383bb2",
      "start": "start",
      "end": "end"
    }
  }
}
```



Servicios Web GraphQL

- De nuevo al ser solo peticiones sobre HTTP, no necesitamos nada más que una librería para enviar y recibir JSON.
- También podemos usar Postman para consumirlas.
- Ejemplo de Clase: <https://github.com/joseluisgs/api-client-acceso-datos>

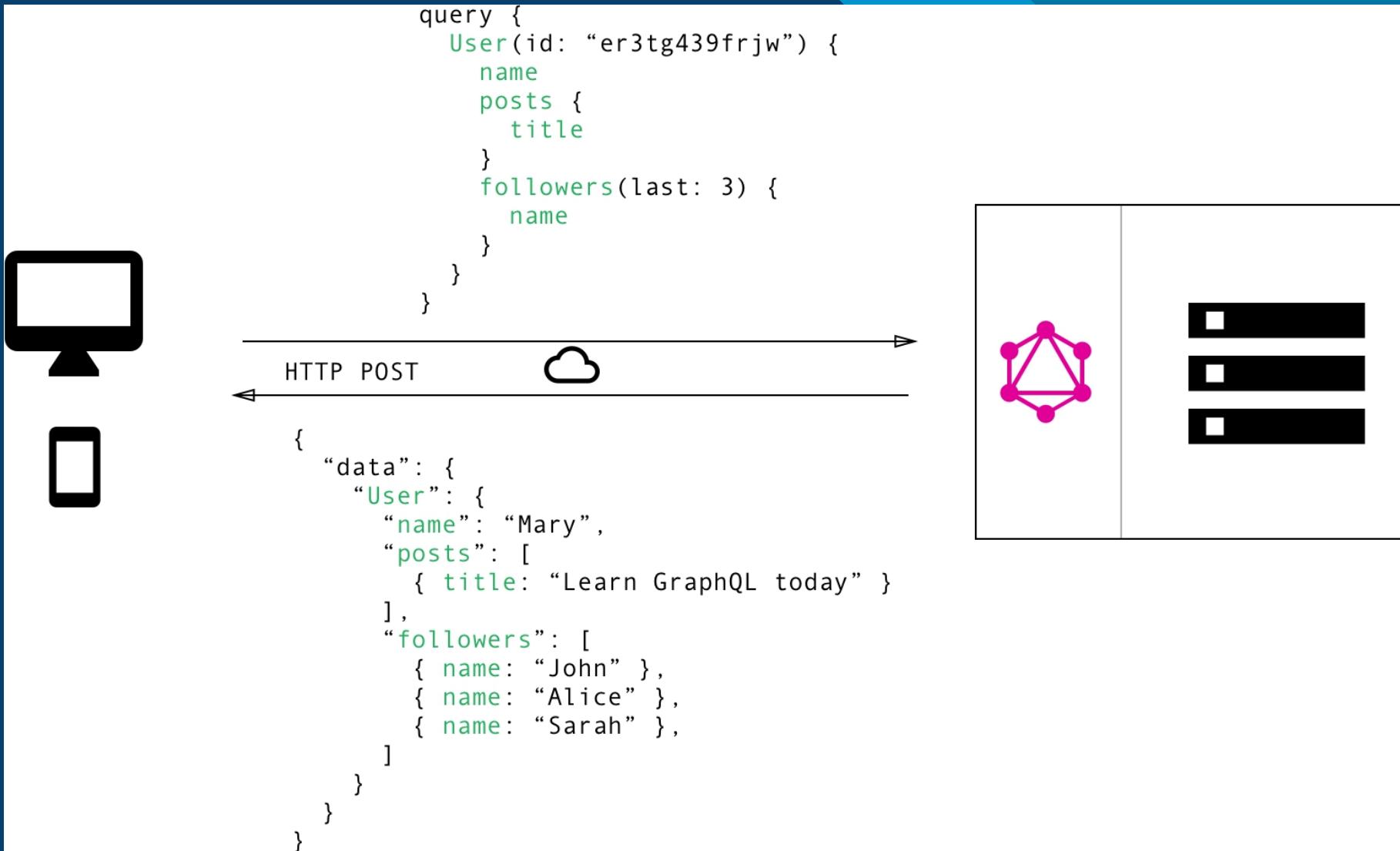


REST vs GraphQL



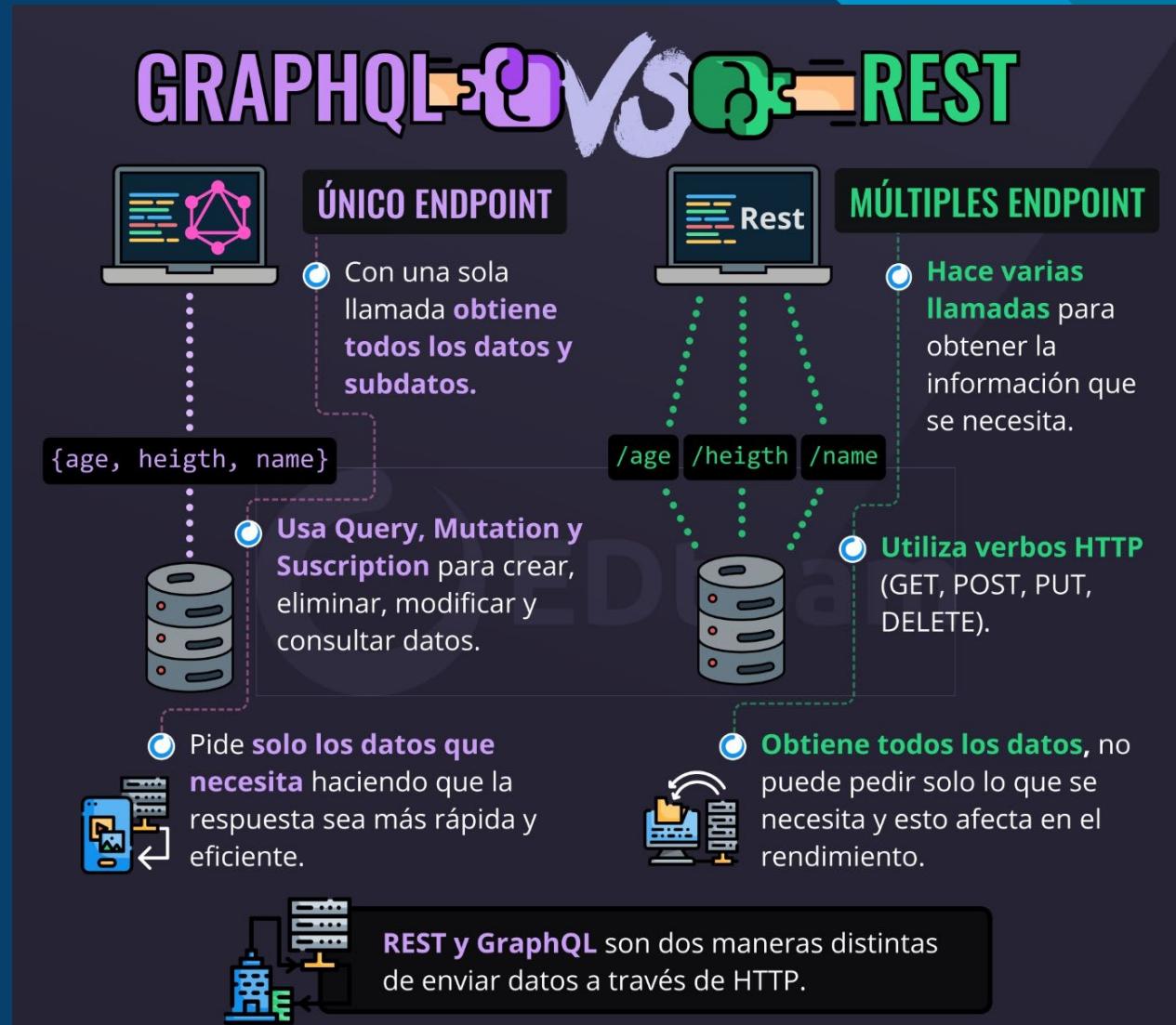


REST vs GraphQL





REST vs GraphQL



“

"Lo realmente necesario es saberlo todo sobre los cambios en la información. Nadie quiere o necesita que le recuerden 16 horas al día que tiene sus zapatos puestos"

- David Hubel

”



Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>
- Discord: <https://discord.gg/eFMKxfPY>
- Aula Virtual: <https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=247>



Gracias

José Luis González Sánchez

