

## Curso de Spring Framework

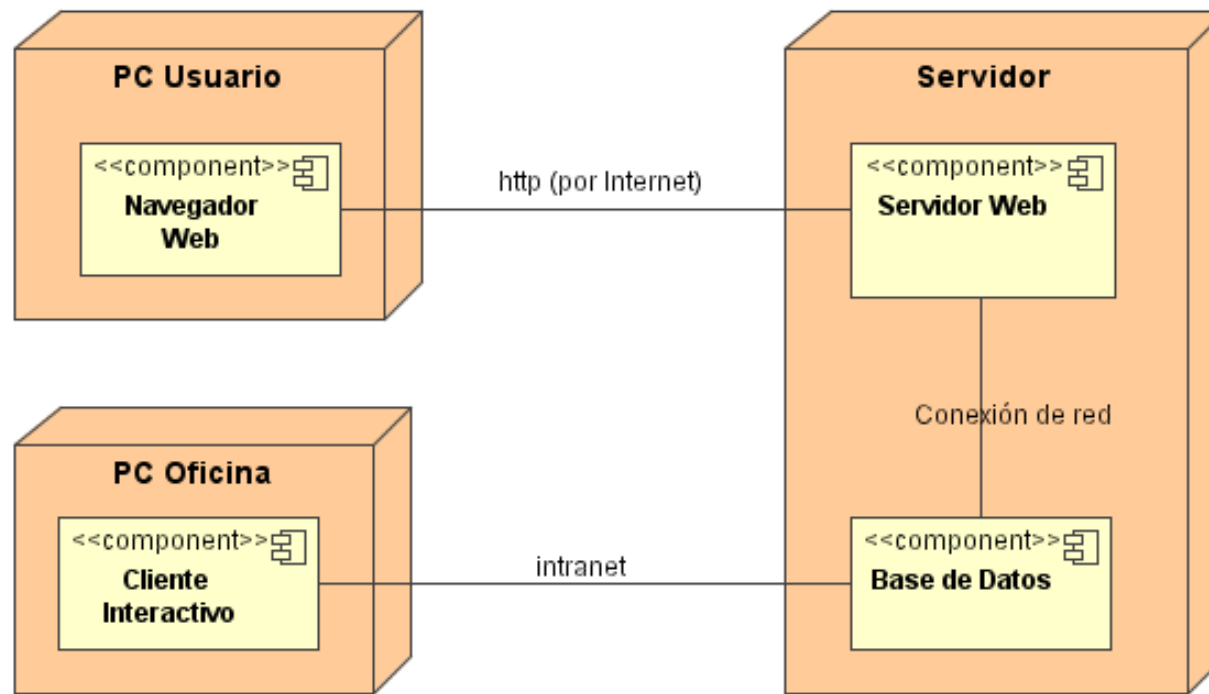
# Tema 3 Bases de datos con Spring

- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- Streams
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

- Bases de datos (*Database systems*)
  - Son programas utilizados para almacenar información y permitir un acceso posterior a ella
  - Varios programas (servidor web, aplicación gráfica, ...) pueden acceder a la información de forma **concurrente** a través de la red

- **Bases de datos (*Database systems*)**
  - La información está **centralizada, actualizada** y es más sencillo realizar actualizaciones y copias de seguridad
  - La información puede estar en forma de texto, números, ficheros, XML, etc...
  - Existen muchos tipos de bases de datos, pero las más usadas son las **Bases de datos SQL**

- Despliegue típico de una aplicación con BBDD



# Bases de datos SQL

- Una base de datos **relacional** almacena la información en tablas\* con filas y columnas (campo)

Tabla Libros

idLibro	titulo	precio
1	Bambi	3
2	Batman	4
3	Spiderman	2

Tabla Autores

idAutor	nombre	nacionalidad
1	Antonio	Español
2	Gerard	Frances

Tabla RelacionLibroAutor

idLibro	idAutor
1	1
2	2
3	2

\* A las tablas se las denominaba “relaciones”, de ahí el nombre de base de datos relacional

# Bases de datos SQL

- Una base de datos **relacional** almacena la información en tablas\* con filas y columnas (campo)

Tabla Libros

idLibro	titulo	precio
1	Bambi	3
2	Batman	4
3	Spiderman	2

Tabla Autores

idAutor	nombre	nacionalidad
1	Antonio	Español
2	Gerard	Frances

Tabla RelacionLibroAutor

idLibro	idAutor
1	1
2	2
3	2

La información se relaciona mediante identificadores (id)

- **SQL (*Standard Query Language*)**: Lenguaje de consulta estándar que permite:
  - Consulta de los datos seleccionados con diferentes criterios y realizando operaciones (medias, sumas, ...)
  - Inserción, Actualización y borrado de la información
  - Creación, alteración y borrado de las tablas y sus campos
  - Gestión de usuarios y sus privilegios de acceso



# Sentencia SQL SELECT

- También conocido como *statement* o *query* (consulta)
- Permite **recuperar** la información de una o varias tablas
- Especifica uno o más **campos**, una o más **tablas** y un **criterio** de selección
- La base de datos devuelve los campos indicados de aquellas **filas** que cumplan el **criterio** de selección

# Sentencia SQL SELECT

Situación en la  
base de datos

Tabla Libros

idLibro	titulo	precio
1	Bambi	3
2	Batman	4
3	Spiderman	2

Consulta

```
SELECT titulo, precio  
FROM Libros  
WHERE precio > 2
```



Conjunto de resultados (*ResultSet*)

titulo	precio
Bambi	3
Batman	4

# Cláusula WHERE

- Operador LIKE (Comparación de cadenas)

```
SELECT titulo, precio  
FROM Libros  
WHERE titulo LIKE 'Ba%'
```

- Operadores relacionales (<,<=,...) lógicos (AND, OR)

```
SELECT titulo, precio  
FROM Libros  
WHERE precio > 3 AND titulo LIKE '%Man'
```

# Uniones (JOINS)

- Se pueden unir varias tablas en una consulta

Tabla Libros

idLibro	titulo	precio
1	Bambi	3
2	Batman	4
3	Spiderman	2

Tabla Autores

idAutor	nombre	nacionalidad
1	Antonio	Español
2	Gerard	Frances

Tabla RelacionLibroAutor

idLibro	idAutor
1	1
2	2
3	2

# Uniones (JOINS)

- Se pueden unir varias tablas en una consulta

Tabla Libros

idLibro	titulo	precio
1	Bambi	3
2	Batman	4
3	Spiderman	2

Tabla Autores

idAutor	nombre	nacionalidad
1	Antonio	Español
2	Gerard	Frances

Tabla RelacionLibroAutor

idLibro	idAutor
1	1
2	2
3	2

# Uniones (JOINS)

```
SELECT titulo, precio, nombre  
FROM Libros, Autores, RelacionLibroAutor  
WHERE Libros.idLibro = RelacionLibroAutor.idLibro  
      AND Autores.idAutor = RelacionLibroAutor.idAutor
```



titulo	precio	nombre
Bambi	3	Antonio
Batman	4	Gerard
Spiderman	2	Gerard

- Existen muchos productos de bases de datos, comerciales y software libre
  - **MySQL** (Software Libre) – <http://www.mysql.org>
  - **PostgreSQL** (Software Libre) - <http://www.postgresql.org/>
  - **Oracle** (Comercial) - <http://www.oracle.com>
  - **MS SQL Server** (Comercial) - <http://www.microsoft.com/sql>
  - **H2** (Software libre) - <http://www.h2database.com/>



# MySQL



- <http://www.mysql.org/>
- Sistema gestor de base de datos multiplataforma
- Desarrollado en C
- Licencia código abierto GPL
- Soporte de un subconjunto de SQL 99
- Dispone de driver JDBC para Java
- Herramienta interactiva para hacer consultas y crear bases de datos
- Propiedad de **ORACLE®**



# H2 Database



- <http://www.h2database.com/>
- Sistema gestor de base de datos multiplataforma
- Implementado en **Java**
- Licencia **código abierto** MPL 2.0 y EPL 1.0
- Soporte de un subconjunto de SQL 99 y 2003
- Dispone de driver **JDBC** para Java
- Se puede usar **en memoria**, ideal para desarrollo y testing

- Bases de datos SQL
- **Bases de datos SQL con Spring**
- Spring Data y JPA
- Paginación
- Streams
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

# Bases de datos SQL con Spring

- El proyecto **Spring Data** ofrece mecanismos para el acceso a **Bases de datos SQL** y no relacionales
- Funcionalidades principales:
  - **Creación del esquema** partiendo de las clases del código Java (o viceversa)
  - **Conversión automática** entre objetos Java y el formato propio de la base de datos
  - **Creación de consultas** en base a métodos en interfaces

<http://projects.spring.io/spring-data/>  
<http://projects.spring.io/spring-data-jpa/>

# Bases de datos SQL con Spring

ejem1

- Objeto de la base de datos (entidad)

Anotaciones usadas para generar el esquema y para hacer la conversión entre objetos y filas

El atributo anotado con **@Id** es la clave primaria de la tabla. Lo habitual es que se genere de forma automática

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String firstName;
    private String lastName;

    // Constructor necesario para la carga desde BBDD
    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Getter, Setters and toString
    ...
}
```

# Bases de datos SQL con Spring

- Consultas a la base de datos

ejem1

El interfaz padre **JpaRepository** dispone de métodos para **consultar, guardar, borrar y modificar** objetos de la base de datos.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
    List<Customer> findByFirstName(String firstName);
}
```

Métodos que se traducen automáticamente en **consultas a la BBDD** en base al nombre y los parámetros.

Si no se necesita ningún tipo de consulta especial, el **interfaz puede quedar vacío**

Clase de la **entidad** y clase de su identificador (generalmente **Integer** o **Long**)

- **Uso de la base de datos**
  - Para acceder a la base de datos se **inyecta el repositorio** con `@Autowired` y se utilizan sus métodos
  - Métodos disponibles:
    - Consultar todos (**findAll**)
    - Consulta por id (**findById**)
    - Guardar un nuevo objeto o actualizarlo (**save**)
    - Borrar un objeto (**delete**)
    - Consultas por cualquier criterio (métodos añadidos al interfaz Repository)

# Bases de datos SQL con Spring

ejem1

- **Uso de la BBDD**

```
@Controller
public class DataBaseUsage implements CommandLineRunner {

    @Autowired
    private CustomerRepository repository;

    @Override
    public void run(String... args) throws Exception {

        repository.save(new Customer("Jack", "Bauer"));
        repository.save(new Customer("Chloe", "O'Brian"));

        List<Customer> bayers = repository.findByLastName("Bauer");
        for (Customer bauer : bayers) {
            System.out.println(bauer);
        }

        repository.delete(bayers.get(0));
    }
}
```

Controlador especial que se ejecuta cuando se **inicia la aplicación**. Puede leer los parámetros de la **línea de comandos**

Código de ejemplo que usa el **repositorio** para guardar, consultar y borrar datos de la **BBDD**

# Bases de datos SQL con Spring

ejem1

- pom.xml

```
<groupId>es.sidelab</groupId>
<artifactId>bbdd_ejem1</artifactId>
<version>0.1.0</version>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.3.RELEASE</version>
</parent>
...
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
  </dependency>
</dependencies>
```

Dependencia a **Spring Data** para BBDD relacionales

Dependencia a la **BBDD en memoria H2**. En las bases de datos en memoria, el esquema se genera de forma **automática** al iniciar la app



# Bases de datos SQL con Spring

ejem2

- Una aplicación web puede gestionar la información en una BBDD
  - Se añaden las dependencias de JPA y la BBDD en el **pom.xml**
  - En vez de usar una estructura en memoria se usa un **repositorio**
- Como ejemplo vamos a transformar la web de gestión de anuncios para que use BBDD

- pom.xml

```
...
<dependencies>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
  </dependency>

</dependencies>
```

Dependencias  
**Web**

Dependencia a la  
**BBDD H2**

# Bases de datos SQL con Spring

ejem2

- Objeto de la base de datos (entidad)

```
@Entity
public class Anuncio {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String nombre;
    private String asunto;
    private String comentario;

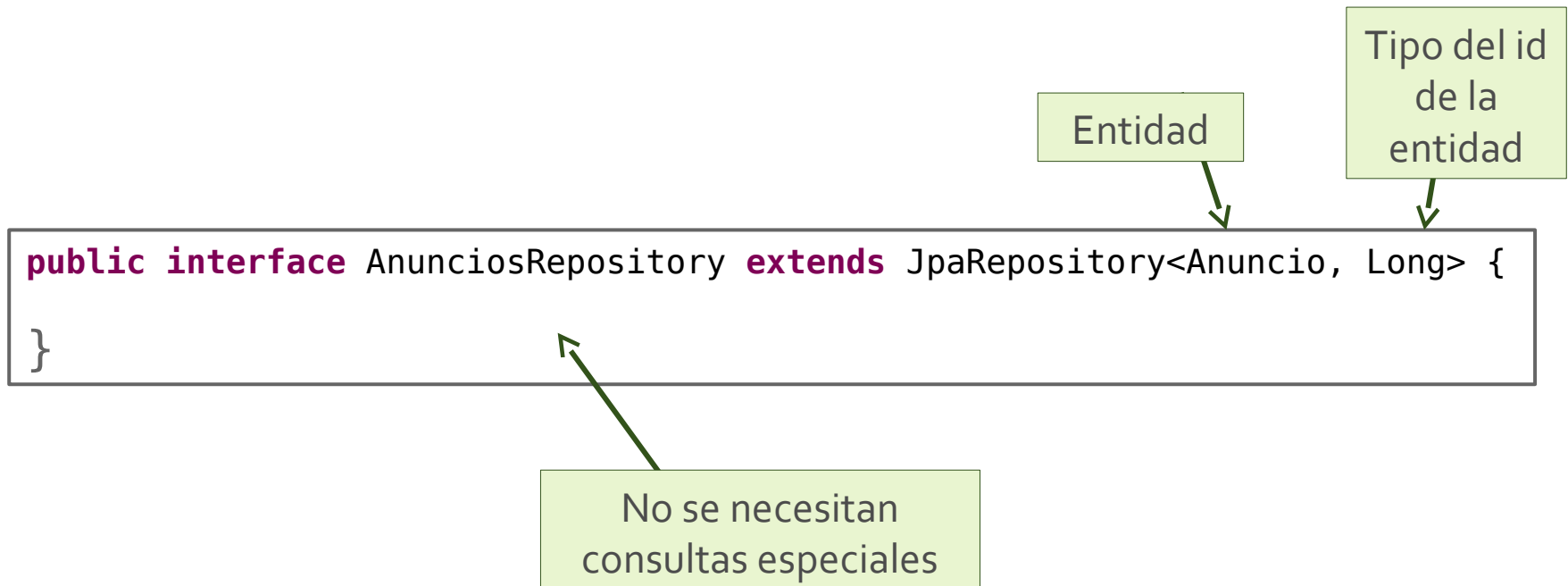
    public Anuncio() {}

    public Anuncio(String nombre, String asunto, String comentario) {...}

    public String getNombre() {
        return nombre;
    }

    ...
}
```

- Repositorio



# Bases de datos SQL con Spring

## • Controlador web

ejem2

Un método **@PostConstruct** se ejecutará automáticamente al iniciar la aplicación

```
@Controller
public class TablonController {

    @Autowired
    private AnunciosRepository repository;

    @PostConstruct
    public void init() {
        repository.save(new Anuncio("Pepe", "Hola..", "XXXX"));
        repository.save(new Anuncio("Juan", "Adios...", "XXXX"));
    }

    @RequestMapping("/")
    public String tablon(Model model) {

        model.addAttribute("anuncios", repository.findAll());

        return "tablon";
    }

    ...
}
```

Se pueden incluir tantos repositorios como sea necesario

Desde los métodos se usa el repositorio

# Bases de datos SQL con Spring

ejem2

- Gestión de *Optional* con *findById*

Cuando queremos obtener un elemento por su id debemos considerar si el elemento existe

```
@Controller
public class TablonController {

    @Autowired
    private AnunciosRepository repository;

    @RequestMapping("/anuncio/{id}")
    public String tablon(Model model, long id) {

        model.addAttribute("anuncio", repository.findById(id).get());

        return "tablon";
    }

    ...
}
```

Obtenemos el valor si lo hay. Si no, se lanza *NoSuchElementException*

Desde Spring 5  
findOne se llama  
findById y devuelve  
*Optional<Anuncio>*

# Bases de datos SQL con Spring

ejem5

- Gestión de *Optional* con *findByld*
  - Para devolver 404 cuando se produce la excepción NoSuchElementException se añade un @ControllerAdvice asociado a dicho tipo de excepción

```
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler{

    @ExceptionHandler(NoSuchElementException.class)
    protected ResponseEntity<Object> handleConflict(RuntimeException ex,
                                                    WebRequest request) {
        String bodyOfResponse = "Resource not found";
        return handleExceptionInternal(ex,
                                      bodyOfResponse,
                                      new HttpHeaders(),
                                      HttpStatus.NOT_FOUND,
                                      request);
    }
}
```

- **Consola web de base de datos H2**

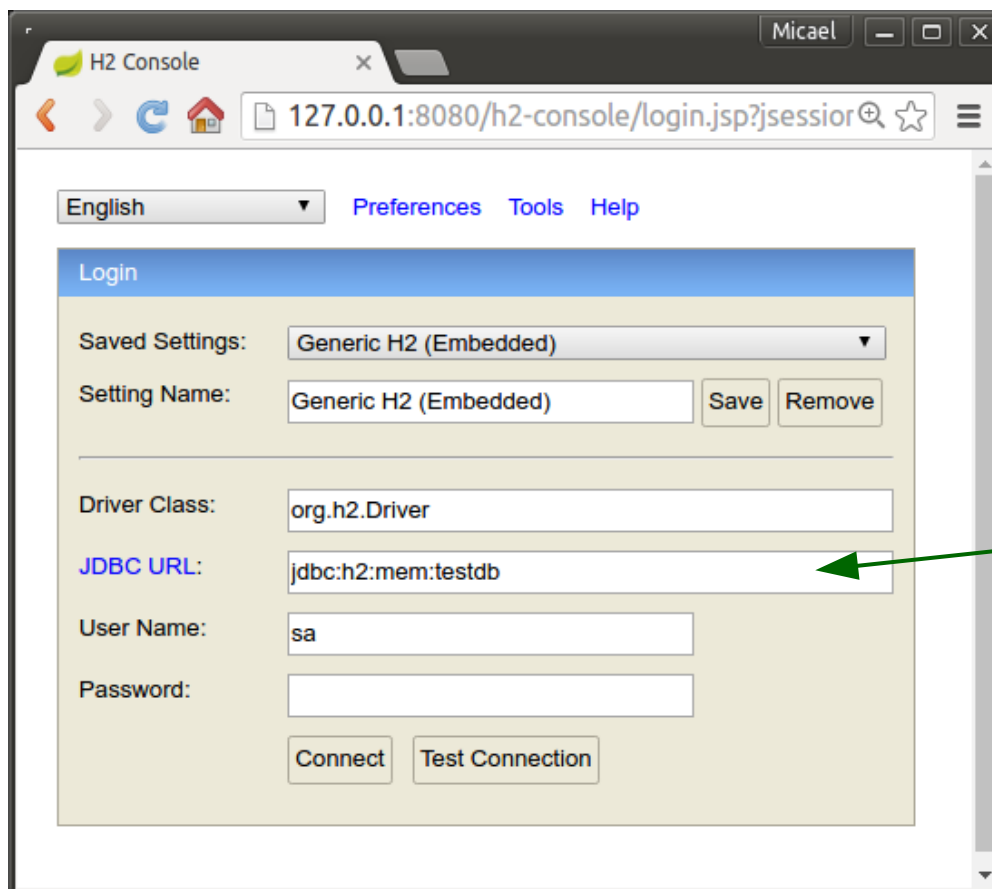
- Si desarrollamos una **aplicación web** que use una **base de datos H2** podemos acceder a una consola web de la propia BBDD
- La consola es accesible en  
<http://127.0.0.1:8080/h2-console>
- Para activar esa consola:
  - Usar las **devtools** en ese proyecto o bien...
  - Poner en el fichero **application.properties** la propiedad **spring.h2.console.enabled=true**



# Bases de datos SQL con Spring

ejem2

- Consola web de base de datos H2

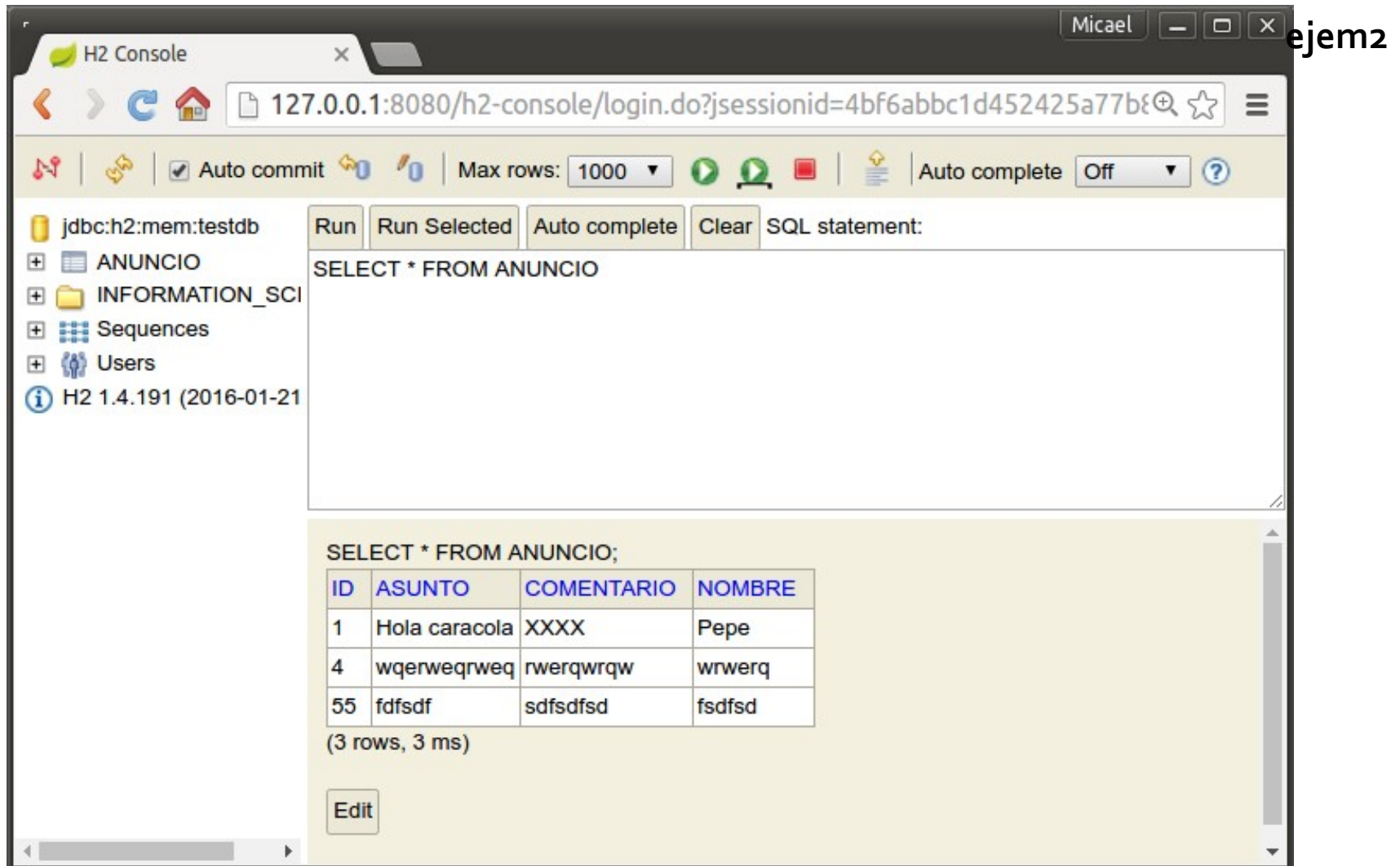


Hay que poner como  
JDBC URL

`jdbc:h2:mem:testdb`

# Bases de datos SQL con Spring

ejem2



The screenshot shows the H2 Console web interface in a browser window. The address bar displays the URL `127.0.0.1:8080/h2-console/login.do?jsessionId=4bf6abbc1d452425a77b8`. The interface includes a sidebar with a tree view of the database structure: `jdbc:h2:mem:testdb`, `ANUNCIO`, `INFORMATION_SCHEMA`, `Sequences`, and `Users`. The main area contains a SQL statement input field with the query `SELECT * FROM ANUNCIO`. Below the input field, the results of the query are displayed as a table with 4 columns: `ID`, `ASUNTO`, `COMENTARIO`, and `NOMBRE`. The table contains 3 rows of data. Below the table, it indicates `(3 rows, 3 ms)` and provides an `Edit` button.

SQL statement:

```
SELECT * FROM ANUNCIO
```

ID	ASUNTO	COMENTARIO	NOMBRE
1	Hola caracola	XXXX	Pepe
4	wqerweqrweq	rwerqwrqw	wnverq
55	fdfsd	sdfsdfsd	fsdfsd

(3 rows, 3 ms)

Edit

# Ejercicio 1

- Transforma el ejercicio de la **gestión de Items** para que utilice una base de datos en vez de guardar la información en memoria

- Bases de datos SQL
- Bases de datos SQL con Spring
- **Spring Data y JPA**
- Paginación
- Streams
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

- Las apps de base de datos en Spring usan dos librerías a la vez:
  - JPA (*Java Persistence API*)
    - Estándar de Java EE para acceso a base de datos relacionales
  - Spring Data
    - Facilidades propias de Spring que facilitan el acceso a base de datos con JPA

- **Spring Data**

- Permite la creación de **consultas** partiendo de **métodos** en los repositorios
- Se puede usar con **cualquier** base de datos (relacional o NoSQL)
- Los métodos pueden tener varios parámetros que forman **expresiones lógicas**: Y, O, No...
- Los métodos se pueden anotar para definir la consulta de forma más precisa con **JPQL**

<http://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

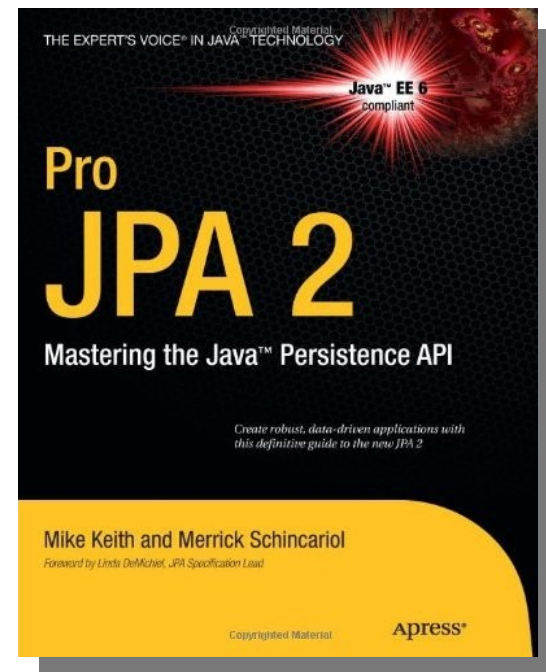
- **JPA (*Java Persistence API*)**

- Acceso a Bases de datos relacionales desde Java
- JPA forma parte del estándar **Java EE**
- **Funcionalidades**
  - Generación del **esquema** de la base de datos partiendo de las entidades (clases Java)
  - **Conversión automática** entre objetos y filas de tablas

- JPA (*Java Persistence API*)

## Libro

- Pro JPA 2 Mastering the Java™ Persistence API
- By Mike Keith , Merrick Schincariol
- Apress



## Tutorial

<https://www.javacodegeeks.com/2015/02/jpa-tutorial.htm>



- JPA (*Java Persistence API*)
  - Para usar JPA es necesario usar una **librería** concreta que lo implemente
  - En Spring por defecto se usa la librería **Hibernate**



- **JPA (*Java Persistence API*)**
  - La técnica para convertir entre el modelo de objetos y el modelo relacional se conoce como “**mapeo objeto relacional**” (*object relational mapping*) u ORM
  - Un ORM realiza las **conversiones** pertinentes entre **objetos/clases** y **filas/tablas**
  - Modos de funcionamiento:
    - **Generación de tablas partiendo de clases\***
    - **Generación de clases partiendo de tablas**

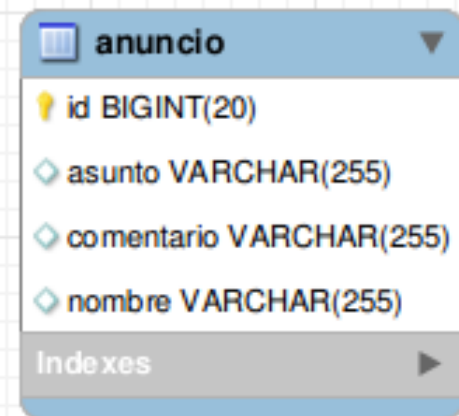
- Entidad con atributos básicos

- Se genera una tabla por cada entidad
- Por cada atributo de la clase de un tipo simple (entero, float, String, boolean...), se crea un campo en la tabla

```
@Entity
public class Anuncio {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String nombre;
    private String asunto;
    private String comentario;
}
```



- **Relaciones entre entidades**
  - Existe una relación cuando una entidad tiene como **atributo otra entidad**
  - En memoria, un **objeto** apunta a uno o más **objetos**
  - También es posible tener una colección de entidades (**List, Set, Map...**)
  - Las relaciones pueden ser **1:1, 1:N, M:N**

- **Relación 1:1 unidireccional**
  - Una entidad tiene una referencia a otra entidad
  - El atributo que apunta a otro es anotado con **@OneToOne**
  - Cada objeto tiene que ser guardado en la base de datos de forma **independiente**
  - Al **cargar un objeto** de la BBDD podemos acceder al objeto **relacionado**

- Relación 1:1 unidireccional

```
@Entity
public class Student {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int year;

    @OneToOne
    private Project project;
}
```

```
@Entity
public class Project {

    @Id
    @GeneratedValue(...)
    private long id;

    private String title;
    private int calification;
}
```

Entidades

- Relación 1:1 unidireccional

```
@Entity
public class Student {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int year;

    @OneToOne
    private Project project;
}
```

```
@Entity
public class Project {

    @Id
    @GeneratedValue(...)
    private long id;

    private String title;
    private int calification;
}
```

Entidades

# Relación 1:1 unidireccional

ejem4

```
@Entity
public class Student {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int year;

    @OneToOne
    private Project project;
}
```

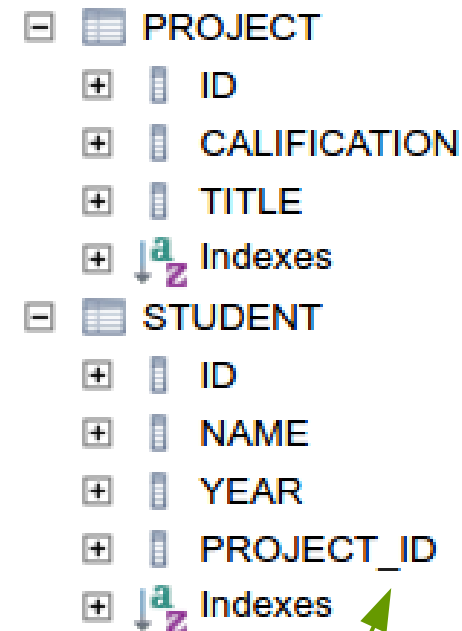
```
@Entity
public class Project {

    @Id
    @GeneratedValue(...)
    private long id;

    private String title;
    private int calification;
}
```



## Generación de tablas



La tabla **STUDENT** tiene un campo **PROJECT\_ID** que apunta al ID de la tabla **PROJECT**



# Generación de tablas desde entidades

ejem4

- Relación 1:1 unidireccional

```
Project p1 = new Project("TFG1", 8);
projectRepository.save(p1);

Student s1 = new Student("Pepe", 2010);
s1.setProject(p1);

Student s2 = new Student("Juan", 2011);

studentRepository.save(s1);
studentRepository.save(s2);
```

Cada objeto se tiene  
que guardar en su  
repositorio

Inicialización de datos

- Relación 1:1 unidireccional

Consulta de datos

```
studentRepository.findAll();
```



```
[
  {
    "id" : 1,
    "name" : "Pepe",
    "year" : 2010,
    "project" : {
      "id" : 1,
      "title" : "TFG1",
      "calification" : 8
    }
  },
  {
    "id" : 2,
    "name" : "Juan",
    "year" : 2011
  }
]
```

- Operaciones en cascada
  - Hay veces que un objeto **siempre está asociado** a otro objeto.
  - Por ejemplo, un proyecto siempre está asociado a un estudiante
  - Cuando se **crea el estudiante, se crea el proyecto**, cuando se borra el estudiante, se borra el proyecto

- Operaciones en cascada

- Si la anotación `@OneToOne` se configura con **`cascade = CascadeType.ALL`** entonces ambos objetos de la relación tienen el mismo ciclo de vida
- Al **guardar** el objeto principal, se **guarda** el asociado
- Al **borrar** el objeto principal, se **borra** el asociado

- Operaciones en cascada

```
//Deleting a student delete her associated project
@DeleteMapping("/students/{id}")
public Student deleteStudent(@PathVariable Long id) {
    Student student = studentRepository.findById(id).get();
    studentRepository.deleteById(id);
    return student;
}

//A project only can be deleted if it has no associated student.
@DeleteMapping("/projects/{id}")
public Project deleteProject(@PathVariable Long id) {
    Project project = projectRepository.findById(id).get();
    projectRepository.deleteById(id);
    return project;
}
```

# Generación de tablas desde entidades

ejem5

- Operaciones en cascada

```
@Entity
public class Student {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int year;

    @OneToOne(cascade=CascadeType.ALL)
    private Project project;
}
```

Como la relación en **cascade=ALL**, no es necesario guardar el objeto project explícitamente.

El **project** se guarda cuando el **student** se guarda

```
Student s1 = new Student("Pepe", 2010);
s1.setProject(new Project("TFG1", 8));
studentRepository.save(s1);
```

- **Relación 1:1 bidireccional**

- Para facilitar el desarrollo, los **dos objetos de una relación** pueden tener un atributo al otro objeto
- Ambas entidades tienen que anotarse con **@OneToOne**
- Aunque en memoria ambos objetos se apunten entre sí, en la base de datos la relación se guarda en la **tabla de una entidad**

- **Relación 1:1 bidireccional**
  - La entidad cuya tabla guarda la información se anota con **@OneToOne**
  - Sus objetos **deben tener la información** al guardar en la BBDD
  - La otra entidad se anota con **@OneToOne(mappedby="attr")** indicando el nombre del atributo de la otra entidad
  - Este atributo sólo se usa en lecturas



- Relación 1:1 bidireccional

```
@Entity
public class Student {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int year;

    @OneToOne(cascade=CascadeType.ALL)
    private Project project;
}
```

```
@Entity
public class Project {

    @Id
    @GeneratedValue(...)
    private long id;

    private String title;
    private int calification;

    @OneToOne(mappedBy="project")
    private Student student;
}
```

```
Student s1 = new Student("Pepe", 2010);
s1.setProject(new Project("TFG1", 8));
studentRepository.save(s1);
```

Al guardar, la entidad principal debe apuntar a la otra entidad

- **Relación 1:N**

- Cuando existe una relación 1:N entre entidades se usan las anotaciones **@OneToMany** y **@ManyToOne**
- Puede ser **unidireccional** o **bidireccional**
- Cualquier sentido de la relación puede ser la **entidad principal** (la que se usa para guardar los datos)
- También se puede usar **cascade**

- Relación 1:N unidireccional

```
@Entity
public class Team {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int ranking;

    @OneToMany
    private List<Player> players;

}
```

```
@Entity
public class Player {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int goals;

}
```

- Relación 1:N unidireccional

```
Player p1 = new Player("Torres", 10);  
Player p2 = new Player("Iniesta", 10);  
  
playerRepository.save(p1);  
playerRepository.save(p2);  
  
Team team = new Team("Selección", 1);  
  
team.getPlayers().add(p1);  
team.getPlayers().add(p2);  
  
teamRepository.save(team);
```

# Generación de tablas desde entidades

ejem7

- **Relación 1:N unidireccional**
  - Si usamos cascada hay que tener cuidado en el borrado para que se eliminen todos los elementos asociados de la entidad secundaria

```
//Deleting a team doesn't delete its associated players
@DeleteMapping("/teams/{id}")
public Team deleteTeam(@PathVariable Long id) {
    Team team = teamRepository.findById(id).get();
    //Force loading players from database before deleting team
    team.getPlayers().size();
    teamRepository.deleteById(id);
    return team;
}
```

Al llamar a *size()* se cargan los elementos de la lista, que normalmente sólo se cargan de forma perezosa

# Generación de tablas desde entidades

ejem8

- Relación 1:N unidireccional (cascade)

```
@Entity
public class Blog {

    @Id
    @GeneratedValue(...)
    private long id;

    private String title;
    private String text;

    @OneToMany(cascade=CascadeType.ALL)
    private List<Comment> comments;
}
```

```
@Entity
public class Comment {

    @Id
    @GeneratedValue(...)
    private long id;

    private String author;
    private String message;
}
```

```
Blog blog = new Blog("New", "My new product");
blog.getComments().add(new Comment("Cool", "Pepe"));
blog.getComments().add(new Comment("Very cool", "Juan"));
repository.save(blog);
```

# Generación de tablas desde entidades

ejem9

- Relación 1:N bidireccional

```
@Entity
public class Team {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int ranking;

    @OneToMany(mappedBy="team")
    private List<Player> players;

}
```

```
@Entity
public class Player {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int goals;

    @ManyToOne
    private Team team;

}
```

# Generación de tablas desde entidades

ejem9

- Relación 1:N bidireccional

```
Team team = new Team("Selección", 1);
teamRepository.save(team);

Player p1 = new Player("Torres", 10);
Player p2 = new Player("Iniesta", 10);

p1.setTeam(team);
p2.setTeam(team);

playerRepository.save(p1);
playerRepository.save(p2);
```

La entidad principal ahora es **Player**, por eso se configura el Team en el objeto Player antes de guardar (y no al revés)



- Relación 1:N bidireccional (cascade)

```
@Entity
public class Blog {

    @Id
    @GeneratedValue(...)
    long id;

    private String title;
    private String text;
```

```
@OneToMany(cascade=CascadeType.ALL, mappedBy="blog")
private List<Comment> comments = new ArrayList<>();
}
```

```
@Entity
public class Comment {

    @Id
    @GeneratedValue(...)
    private long id;

    private String author;
    private String message;

    @ManyToOne
    Blog blog;
}
```

- **Relación M:N**

- Cuando existe una relación M:N entre entidades se usa la anotación **@ManyToMany**
- Puede ser **unidireccional** o **bidireccional**
- Cualquier sentido de la relación puede ser la **entidad principal** (la que se usa para guardar los datos)
- También se puede usar **cascade**

# Generación de tablas desde entidades

ejem10

- Relación M:N bidireccional

```
@Entity
public class Team {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int ranking;

    @ManyToMany(mappedBy="team")
    private List<Player> players;

}
```

```
@Entity
public class Player {

    @Id
    @GeneratedValue(...)
    private long id;

    private String name;
    private int goals;

    @ManyToMany
    private List<Team> teams;

}
```

# Generación de tablas desde entidades

ejem10

- Relación M:N bidireccional

```
Team team1 = new Team("Selección", 1);
Team team2 = new Team("FC Barcelona", 1);
Team team3 = new Team("Atlético de Madrid", 2);

teamRepository.save(team1);
teamRepository.save(team2);
teamRepository.save(team3);

Player p1 = new Player("Torres", 10);
Player p2 = new Player("Iniesta", 10);

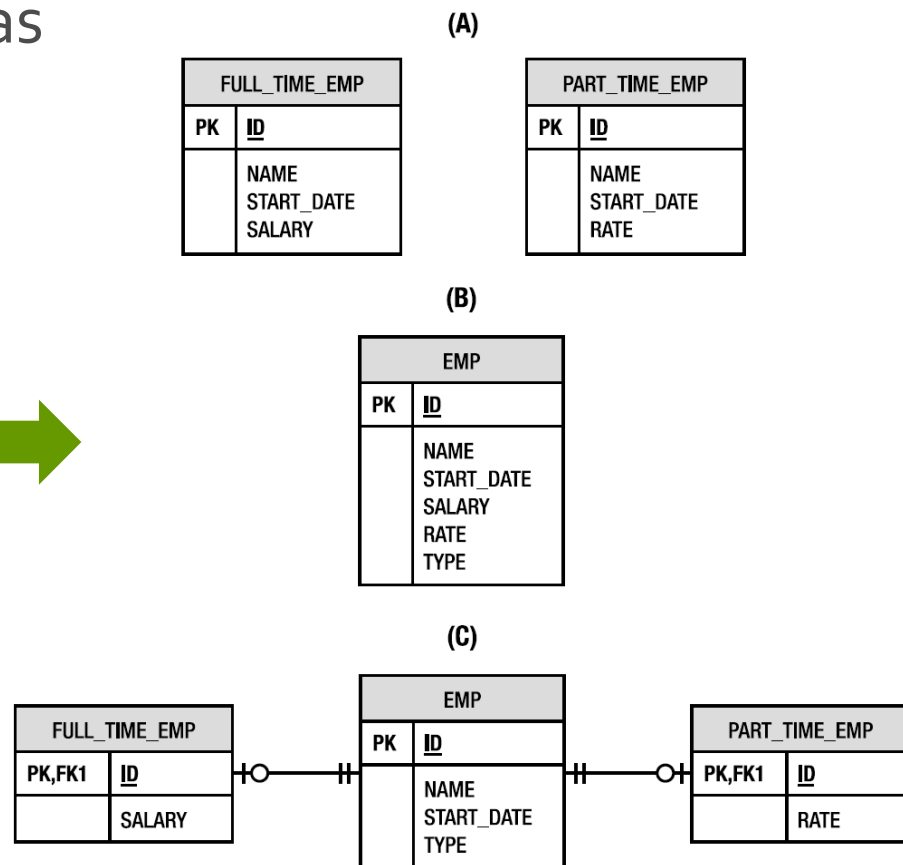
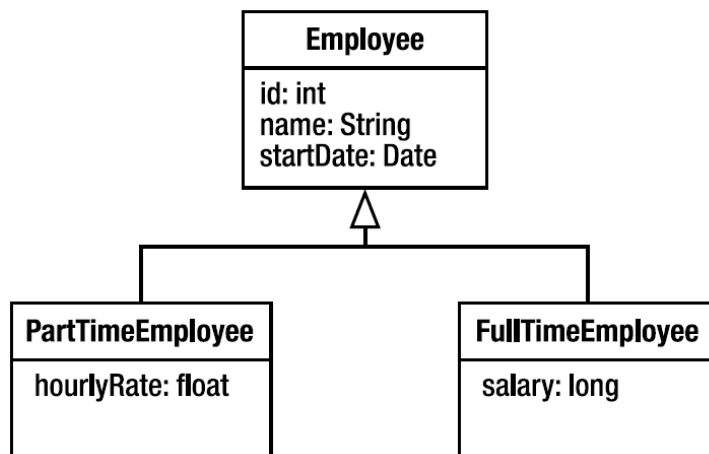
p1.getTeams().add(team1);
p1.getTeams().add(team3);

p2.getTeams().add(team1);
p2.getTeams().add(team2);

playerRepository.save(p1);
playerRepository.save(p2);
```

# JPA (*Java Persistence API*)

- Herencia entre clases
  - Existen varias estrategias



# Spring Data y JPA

- **Spring Data**
  - <http://projects.spring.io/spring-data/>
- **Spring Data Commons**
  - <http://docs.spring.io/spring-data/commons/docs/current/reference/html/>
- **Spring Data JPA**
  - <http://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
- **Bases de datos SQL en Spring Boot**
  - <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html>
- **Inicialización de bases de datos SQL en Spring Boot**
  - <http://docs.spring.io/spring-boot/docs/current/reference/html/howto-databases-initialization.html>
- **Tutorial Spring Data JPA**
  - <https://spring.io/guides/gs/accessing-data-jpa/>

## Ejercicio 2

- Implementar una **API REST** para un blog
- El blog contendrá **Entradas y Comentarios**
- Cada Entrada tiene los siguientes campos:  
Nombre del autor, Nickname del autor, Título, Resumen, Texto, Lista de comentarios
- Los comentarios tienen los siguientes campos:  
Nickname del autor del comentario, Contenido, Fecha del comentario

- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- **Paginación**
- Streams
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD



- Con SpringData es muy sencillo que los resultados de las consultas se devuelvan **paginados**
- Basta poner el parámetro **Pageable** en el método de la consulta
- Cambiar el resulta de List<Element> a **Page<Element>**
- Una Page contiene la información de una **página** (info, nº pág, nº total elementos)

# Paginación

ejem14

Devolvemos un objeto  
**Page<Anuncio>**

En nuestros métodos  
podemos añadir un  
parámetro **Pageable**

```
public interface AnuncioRepository extends JpaRepository<Anuncio, Long> {  
    Page<Anuncio> findByNombre(String nombre, Pageable page);  
    Page<Anuncio> findByAsunto(String asunto, Pageable page);  
}
```

```
Page<Anuncio> a = repository.findByNombre("pepe", PageRequest.of(0, 50));  
Page<Anuncio> a = repository.findAll(PageRequest.of(3, 20));
```

El método `findAll` también tiene  
versión **Paginable**

Creamos un objeto **PageRequest**  
para indicar en num página y tamaño

- En el controlador, podemos crear directamente el objeto Pageable desde la información de la petición

```
@RestController
public class AnuncioController {

    @Autowired
    private AnunciosRepository repository;

    @GetMapping("/anuncios/")
    public Page<Anuncio> anuncios(Pageable page) {

        return repository.findAll(page);

    }

    ...
}
```

- Con la URL habitual se devuelve la **primera página** con **20 elementos**

```
http://localhost/anuncios/
```

- En las peticiones se pueden incluir **parámetros** para solicitar cualquier página

```
http://localhost/anuncios/?page=1&size=3
```

- Válido para **webs** y para **APIs REST**

- En las APIs REST, el JSON de respuesta incluye información de paginación

```
{
  "content": [
    {
      "id": "572216b20c74f013533d113d",
      "nombre": "Juan",
      "asunto": "adios",
      "comentario": "anuncio 0"
    },
    {
      "id": "572216b20c74f013533d113e",
      "nombre": "Pepe",
      "asunto": "hola",
      "comentario": "anuncio 1"
    },
    {
      "id": "572216b20c74f013533d113f",
      "nombre": "Juan",
      "asunto": "adios",
      "comentario": "anuncio 1"
    }
  ],
  "last": false,
  "totalPages": 134,
  "totalElements": 402,
  "sort": null,
  "numberOfElements": 3,
  "first": false,
  "size": 3,
  "number": 1
}
```

- Añade paginación a la página web de gestión de **anuncios**, en la página principal

- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- **Streams**
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

- Con SpringData es muy sencillo que los resultados de las consultas se devuelvan como **Streams** de Java 8
- Las consultas que devuelven Stream son más **eficientes**
  - Usan **menos memoria**: no hay que traer todos los elementos de la lista en la consulta
  - Son **más rápidas**
- Internamente dependiendo de la implementación se usa
  - ScrollableResultSet (Hibernate)
  - ScrollableCursor (EclipseLink)



- Basta con cambiar el tipo de resultado devuelto de `List<Element>` a **`Stream<Element>`** en los métodos del repositorio
- ¿Y qué pasa con la paginación?
  - Puedes pasar un parámetro `Pageable`

Devolvemos un objeto  
**Stream<Anuncio>**

En nuestros métodos  
podemos añadir un  
parámetro **Pageable**

```
public interface AnuncioRepository extends JpaRepository<Anuncio, Long> {  
    Stream<Anuncio> findStreamAll(Pageable page);  
    Stream<Anuncio> findByAsunto(String asunto, Pageable page);  
}
```

- Cuando usamos Streams **debemos cerrar el Stream** cuando dejemos de usarlo
  - Llamando al método *close()*
  - Utilizando el *try-with-resources* (preferible)

```
try (Stream<Anuncio> anuncios =  
    repository.findStreamAll("pepe", PageRequest.of(0, 50)) {  
    ...  
}
```

Obtenemos el objeto **Stream**  
dentro de un **try-with-resources**

Creamos un objeto **PageRequest**  
para indicar en num página y tamaño


- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- Streams
- **Consultas**
- Gestión del esquema
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

- Con SpringData pueden definirse consultas a la base de datos de diversas formas:
  - Métodos en los repositorios
  - Java Persistence Query Language (JPQL)
  - QueryDSL

- Métodos en los repositorios

- En base al nombre del método y el tipo de retorno se construye la consulta a la BBDD

```
public interface AnuncioRepository extends JpaRepository<Anuncio, Long> {  
    Page<Anuncio> findByNombre(String nombre);  
    Page<Anuncio> findByAsunto(String asunto);  
}
```



Consultas con un único campo  
como criterio

## • Métodos en los repositorios

```
public interface PersonRepository extends Repository<Person, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
  
}
```

- **Métodos en los repositorios**

- **Consulta**

- `List<Elem> find...By...(...)`
    - `List<Elem> read...By...(...)`
    - `List<Elem> query...By...(...)`
    - `List<Elem> get...By...(...)`

- **Contar**

- `int count...By...(...)`



- **Métodos en los repositorios**
  - **Expresiones:** And, Or
  - **Comparadores:** Between, LessThan, GreatherThan
  - **Modificadores:** IgnoreCase
  - **Ordenación:** OrderBy...Asc / OrderBy...Desc

- **Métodos en los repositorios**
  - Propiedades de los objetos relacionados
    - No sólo podemos filtrar por un **atributo** de la propia entidad, también podemos filtrar por un **atributo de otra entidad** con la que esté relacionada la principal
      - **Person** con un atributo **address**
      - **Adress** tiene un atributo **ZipCode**

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

## • Métodos en los repositorios

### – Ordenación

- Podemos pasar un parámetro de tipo Sort que controla la ordenación
- Existe el método `findAll(Sort sort)`

```
repository.findAll(Sort.by("nombre"));  
repository.findAll(Sort.by(Order.asc("nombre")));
```

- Podemos añadir métodos personalizados

```
List<User> findFirst10ByLastname(String lastname, Sort sort);
```

- **Métodos en los repositorios**

- Ordenación

- El objeto Sort se puede crear desde la URL (como Pageable)

```
http://localhost/anuncios/?sort=nombre,desc
```

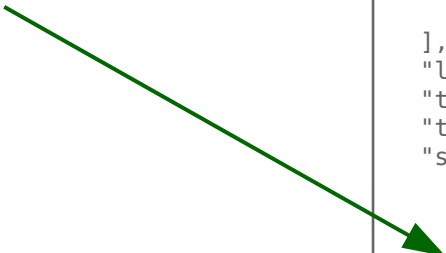
```
@RequestMapping("/anuncios/", method = RequestMethod.GET)
public Page<Anuncio> anuncios(Sort sort) {
    return repository.findAll(sort);
}
```

- **Métodos en los repositorios**
  - Ordenación
    - El objeto Pageable incluye la información de ordenación de la URL

```
http://localhost/anuncios/?page=1&size=3&sort=nombre,desc
```

```
@RequestMapping("/anuncios/", method = RequestMethod.GET)
public Page<Anuncio> anuncios(Pageable page) {
    return repository.findAll(page);
}
```

- En las APIs REST, el JSON de respuesta incluye información de paginación y ordenación



```
{
  "content": [
    {
      "id": "572216b20c74f013533d1140",
      "nombre": "Pepe",
      "asunto": "hola",
      "comentario": "anuncio 2"
    },
    {
      "id": "572216b20c74f013533d1142",
      "nombre": "Pepe",
      "asunto": "hola",
      "comentario": "anuncio 3"
    },
    {
      "id": "572216b20c74f013533d1144",
      "nombre": "Pepe",
      "asunto": "hola",
      "comentario": "anuncio 4"
    }
  ],
  "last": false,
  "totalPages": 134,
  "totalElements": 402,
  "sort": [
    {
      "direction": "DESC",
      "property": "nombre",
      "ignoreCase": false,
      "nullHandling": "NATIVE",
      "ascending": false
    }
  ],
  "numberOfElements": 3,
  "first": false,
  "size": 3,
  "number": 1
}
```

- **Métodos en los repositorios**
  - Limitar los resultados

```
User findFirstBy...();  
User findTopBy...();  
User findTopDistinctBy...();  
List<User> queryFirst10By...();  
List<User> findTop3By...();  
List<User> findFirst10By...();
```

- **Java Persistence Query Language (JPQL)**
  - JPQL es un lenguaje similar a SQL pero referencia conceptos de las entidades JPA

ejem11

```
public interface TeamRepository extends JpaRepository<Team, Long> {  
  
    @Query("select t from Team t where t.name = ?1")  
    List<Team> findByName(String name);  
  
}
```

NOTA: La query JPQL no sería necesaria porque el método del repositorio ejecuta justo esa consulta



- **Java Persistence Query Language (JPQL)**
  - Estructura de sentencia SELECT

```
SELECT ... FROM ...  
[WHERE ...]  
[GROUP BY ... [HAVING ...]]  
[ORDER BY ...]
```

- Actualización y borrado

```
DELETE FROM ... [WHERE ...]  
  
UPDATE ... SET ... [WHERE ...]
```

- **Java Persistence Query Language (JPQL)**
  - Ejemplos

```
SELECT e from Employee e  
WHERE e.salary BETWEEN 30000 AND 40000
```

```
SELECT e from Employee e WHERE e.ename LIKE 'M%'
```

```
SELECT DISTINCT b FROM Blog b JOIN b.comments c  
WHERE c.author=?1
```

- **Java Persistence Query Language (JPQL)**

ejem12

```
public interface BlogRepository extends JpaRepository<Blog, Long> {  
    @Query("SELECT DISTINCT b FROM Blog b JOIN b.comments c "+  
           "WHERE c.author=?1")  
    List<Blog> findByCommentsAuthor(String author);  
}
```

NOTA: La query JPQL no sería necesaria porque el método del repositorio ejecuta justo esa consulta

- **Structured Query Language (SQL)**
  - Incluso podemos usar SQL directamente

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query(  
        value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1",  
        nativeQuery = true)  
    User findByEmailAddress(String emailAddress);  
  
}
```

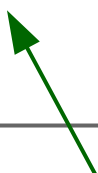
- **QueryDSL**

- Es una librería de generación de consultas en BD relacionales **genérica**
- Se puede usar con **SpringData**
- Permite implementar las consultas usando código Java (el **compilador** avisa de **errores** en la consulta)
- Se puede usar **autocompletar** y la **refactorización de atributos**

- QueryDSL

- Partiendo de las entidades se genera código que sirve para hacer las consultas
- Uso en SpringData

```
public interface TodoRepository extends  
    Repository<Todo, Long>, QueryDslPredicateExecutor<Todo> {  
  
}
```



Añadimos otro interfaz padre al repositorio

- QueryDSL

- Consultas

- Parten de las clases “Q” generadas partiendo del modelo
    - Se define el predicado (filtro)
    - Se usa el findAll(...)

```
@RequestMapping("/", method = RequestMethod.GET)
public Iterable<Todo> todos() {

    Predicate q = QTodo.todo.title.eq("Foo");

    return repository.findAll(page);
}
```

- QueryDSL

```
QTodo.todo.title.eq("Foo").and(QTodo.todo.description.eq("Bar"));
```

```
QTodo.todo.title.eq("Foo").or(QTodo.todo.title.eq("Bar"));
```

```
Qtodo.todo.title.eq("Foo").and(  
    QTodo.todo.description.eq("Bar").not());
```

```
QTodo.todo.description.containsIgnoreCase(searchTerm)  
    .or(QTodo.todo.title.containsIgnoreCase(searchTerm));
```



## • QueryDSL

ejem13

```
@RestController
@RequestMapping("/blogs")
public class BlogController {

    @Autowired
    private BlogRepository repository;

    @RequestMapping("/")
    public Iterable<Blog> getBlogs(@RequestParam(required=false) String author) {
        if(author == null){
            return repository.findAll();
        } else {
            //return repository.findByCommentsAuthor(author);
            return repository.findAll(QBlog.blog.comments.any().author.eq(author));
        }
    }
}
```

- QueryDSL
  - Dependencias

ejem13

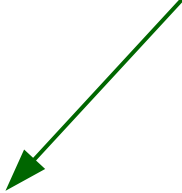
```
<dependencies>
  ...
  <dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-core</artifactId>
    <version>3.6.0</version>
  </dependency>
  <dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <version>3.6.0</version>
  </dependency>
</dependencies>
```

## • QueryDSL

ejem13

```
<build>
  <plugins>
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>apt-maven-plugin</artifactId>
      <version>1.1.3</version>
      <dependencies>
        <dependency>
          <groupId>com.mysema.querydsl</groupId>
          <artifactId>querydsl-apt</artifactId>
          <version>3.6.0</version>
        </dependency>
      </dependencies>
      <executions>
        <execution>
          <goals>
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources</outputDirectory>
            <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Tarea para generar las clases  
partiendo de las entidades



- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- Streams
- Consultas
- **Gestión del esquema**
- Bases de datos NoSQL
- Estructura de una aplicación con BBDD

# Gestión del esquema

- En las **Bases de datos SQL** es necesario **crear el esquema** antes de insertar los datos (crear tablas, relaciones, etc...)
- En las bases de **datos en memoria** (H2, Derby, HSQL...), el esquema siempre se construye de forma **automática** al iniciar la aplicación
- Cuando se usa una base de datos en producción los datos tienen que guardarse en disco y gestionar adecuadamente la **creación del esquema**
- Usaremos **MySQL** a modo de ejemplo

- **Instalación en ubuntu**

- **Servidor**

- ```
sudo apt-get install mysql-server
```

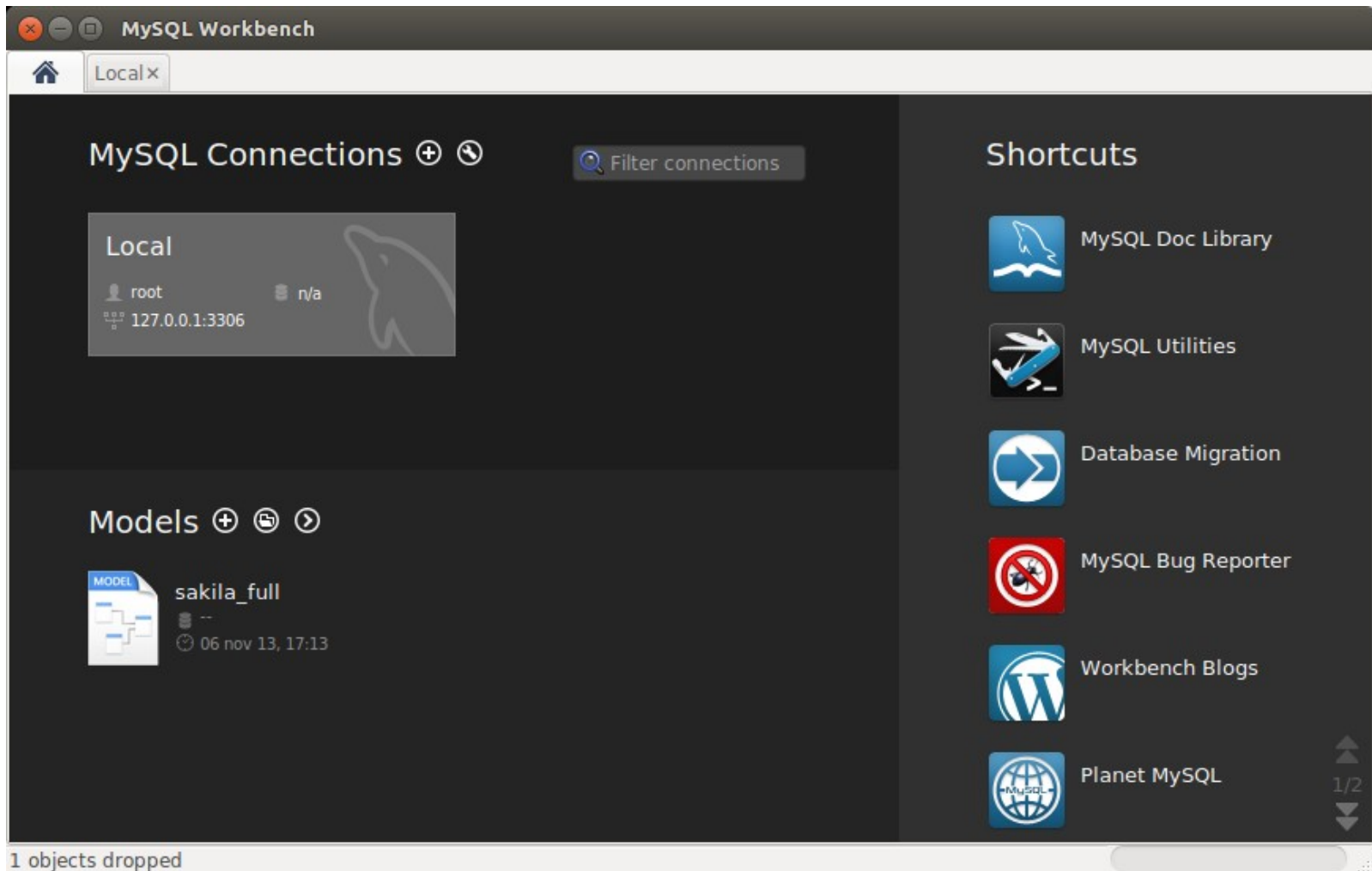
- **Herramienta interactiva**

- ```
sudo apt-get install mysql-workbench
```

- **Otros sistemas operativos**

- <http://dev.mysql.com/downloads/>

# MySQL



# MySQL

MySQL Workbench

Crear esquema

Local x Local x

MANAGEMENT

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE

- Startup / Shutdown
- Server Logs
- Options File

SCHEMAS

Filter objects

- opticom
- test

Object Info Session

No object selected

new\_schema - Schema x SQL File 5 x

Name: new\_schema

Rename References

Default Collation:

Comments:

Schema

Action Output

Added new script editor

The name of the schema. It is recommended to use only lowercase characters. Spaces should be avoided.

Refactor model, changing all references, triggers, stored procedures and functions to use the old schema name to the new one.

Specifies which charset/collation to use if they do not have an explicit set. Common choices are Latin1 or UTF8.



- pom.xml

```
<groupId>es.sidelab</groupId>
<artifactId>bbdd_ejem3</artifactId>
<version>0.1.0</version>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.3.RELEASE</version>
</parent>
...
<dependencies>
  ...
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
```

Dependencia a **Spring Data** para BBDD relacionales

Dependencia al driver **MySQL**

- **Datos de conexión a la BBDD**

Este fichero tiene que estar en el fichero **src/main/resources**

**Host** en el que está alojado el servidor MySQL

Nombre del **esquema**. Este esquema tiene que ser creado manualmente por el desarrollador usando herramientas de MySQL

**application.properties**

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=root
spring.datasource.password=pass
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=...
```

**Usuario y contraseña.**  
En algunos SO estos datos se configuran al instalar MySQL

- **Propiedad de generación del esquema**

```
spring.jpa.hibernate.ddl-auto=...
```

- **create-drop:** Crea el esquema al iniciar la aplicación y le borra al finalizar (ideal para programar como si la BBDD estuviera en memoria)
- **create:** Crea el esquema al iniciar la aplicación
- **update:** Añade al esquema actual las tablas y atributos necesarios para hacer el esquema compatible con las clases Java (no borra ningún elemento). Si el esquema está vacío, se genera completo.
- **validate:** Verifica que el esquema de la BBDD es compatible con las entidades de la aplicación y si no lo es genera un error.
- **none:** No hace nada con el esquema y asume que es correcto.

- **Estrategia de desarrollo con BBDD**
  - **Fase 1:** Desarrollo: Usar base de datos en memoria (H2)
  - **Fase 2:** Desarrollo: Usar base de datos persistente (MySQL) con create-drop
  - **Fase 3:** Instalación: Usar base de datos persistente con update
  - **Fase 4:** Actualización: Usar herramientas específicas

- **Estrategia de desarrollo con BBDD**
  - **Fase 1: Desarrollo: Usar base de datos en memoria (H2)**
    - La aplicación se puede ejecutar sin ninguna aplicación instalada
    - Los tests se ejecutan más rápido
    - Los datos de ejemplo son insertados por la propia aplicación.
    - Cualquier cambio en las entidades (clases) se refleja de forma automática en el esquema

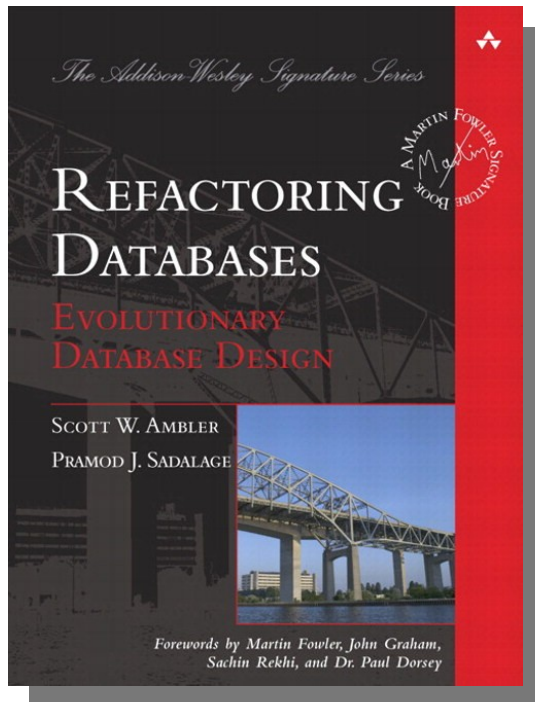
- **Estrategia de desarrollo con BBDD**
  - **Fase 2: Desarrollo: Usar base de datos persistente (MySQL) con create-drop**
    - Se verifica que la aplicación funciona correctamente con MySQL (puede haber ligeras diferencias con H2)
    - Cualquier cambio en las entidades se refleja de forma automática en la BBDD al arrancar la aplicación

- **Estrategia de desarrollo con BBDD**
  - **Fase 3: Instalación: Usar base de datos persistente con update**
    - Al iniciar la aplicación por primera vez se genera el esquema.
    - Al iniciar la aplicación por segunda vez el esquema no se modifica (porque la app no ha cambiado)
    - Si se quiere evitar el tiempo extra que requiere comprobar el esquema, se puede configurar la gestión del esquema a “none”

- **Estrategia de desarrollo con BBDD**
  - **Fase 4: Actualización: Usar herramientas específicas**
    - Si se desarrolla una nueva versión de una aplicación y esa nueva versión tiene algún cambio en las entidades no se puede instalar sin más
    - Es necesario hacer cambios en la BBDD para el nuevo esquema, pero la BBDD ya tiene datos
    - Hay que usar herramientas específicas



- **Migración del esquema de la BBDD**
  - Cuando una aplicación con BBDD se empieza a usar en **producción**, se empiezan a **guardar datos** en la misma
  - Es posible que la aplicación **evolucione**:
    - Incluyendo nuevas entidades
    - Modificando las entidades existentes
  - Para **actualizar** la aplicación en **producción**, antes hay que **migrar** la BBDD al nuevo esquema



## Refactoring Databases: Evolutionary Database Design

by Scott W. Ambler and  
Pramod J. Sadalage

Addison Wesley Professional  
ISBN#: 0-321-29353-3

<http://www.ambysoft.com/books/refactoringDatabases.html>

- Existen dos herramientas externas que facilitan la evolución del esquema:



<http://flywaydb.org/>

## Database Migrations Made Easy

Gestiona la migración con sentencias SQL específicas de la BBDD



<http://www.liquibase.org/>

## Source Control for your Database

Gestiona la migración con XML genérico independiente de la de la BBDD

# Gestión del esquema

- **Flyway y Liquibase** mantienen una **lista de cambios** que hay que hacer en el esquema para **evolucionar/migrar** de una versión a otra superior
- Cuando la aplicación **Spring** se **inicia**, se pueden ejecutar de forma **automática estos cambios** para convertir el esquema en la versión deseada
- **Ejemplos:**
  - <https://github.com/spring-projects/spring-boot/tree/1.3.x/spring-boot-samples/spring-boot-sample-flyway>
  - <https://github.com/spring-projects/spring-boot/tree/1.3.x/spring-boot-samples/spring-boot-sample-liquibase>

- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- Streams
- Consultas
- Gestión del esquema
- **Bases de datos NoSQL**
- Estructura de una aplicación con BBDD

- Las bases de datos NoSQL son bases de datos con **modelos de datos diferentes al relacional**
- **NoSQL** es un término que referencia a esta diferencia con las BBDD relacionales, pero no hay una definición formal
- Actualmente se entiende NoSQL como “**Not Only SQL**”

- **Características comunes**

- No se basan en SQL (aunque pueden tener lenguajes de consulta)
- Son escalables mediante clusterización
- Están pensadas para las necesidades de las aplicaciones web actuales
- No tienen esquema de datos. La estructura de los datos se define en cada inserción
- La mayoría son open-source

- ¿Por qué existen las bases de datos NoSQL?
  - Las bases de datos **relacionales no son escalables** porque no se pueden clusterizar
  - Algunos lenguajes requieren otros tipos de datos en vez de los usados en las BBDD relacionales (*impedance mismatch*)
  - La necesidad de **crear un esquema antes de guardar datos** dificulta el desarrollo



# Bases de datos NoSQL



**Cassandra**



**CouchDB**  
relax



**mongoDB**



**riak**

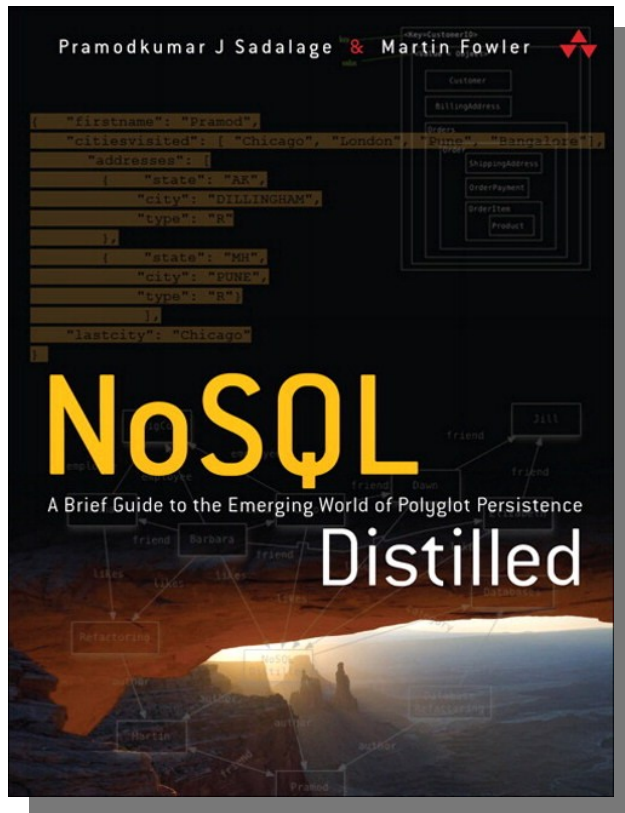


**redis**



**elastic**

- **Persistencia polígglota (*Polyglot persistence*)**
  - Hay que **decidir** la base de datos a utilizar en cada aplicación
  - Diferentes aplicaciones tienen **diferentes necesidades**
  - Hay aplicaciones que pueden tener varias bases de datos a la misma vez
    - **Redis** para sesiones o colas de mensajes
    - **ElasticSearch** para logs
    - **MongoDB** para documentos



## NoSQL Distilled

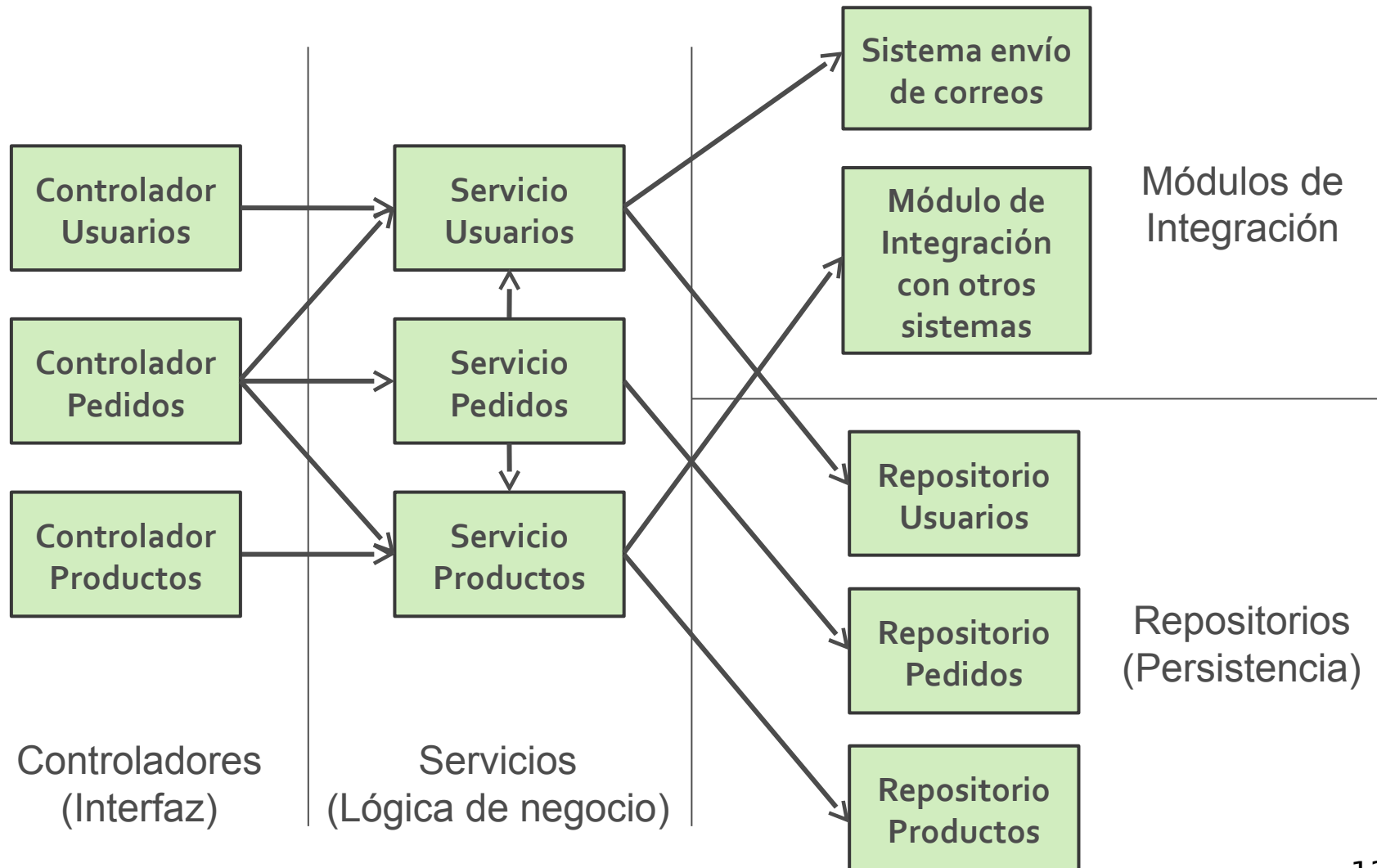
Pramod J. Sadalage &  
Martin Fowler

Addison-Wesley

- Bases de datos SQL
- Bases de datos SQL con Spring
- Spring Data y JPA
- Paginación
- Streams
- Consultas
- Gestión del esquema
- Bases de datos NoSQL
- **Estructura de una aplicación con BBDD**

- La mayoría de las aplicaciones web utilizan **Bases de datos** para guardar su información
- Todas esas aplicaciones tienen una **arquitectura similar** que facilita su desarrollo y mantenimiento

# Estructura de una aplicación con BBDD



- Los servicios y los módulos de integración suelen estar anotados con **@Service** para indicar su naturaleza
- **@Service** es similar a **@Component** (se pueden inyectar en otros componentes)