

# Proyecto colaborativo en servidor Parte III

<b>Proyecto colaborativo en servidor Parte III</b>	<b>1</b>
<b>Conectándonos con una base de datos</b>	<b>1</b>
Montando un servidor de base de datos (en Heroku)	1
Instalando las dependencias necesarias	4
Programando el acceso a la base de datos desde el código	6
Indicando las credenciales de acceso	6
Creando el modelo (entity)	6
Creando el repositorio	8
Creando el servicio	8
Actualizando los controladores	9
Añadiendo las rutas para listar todos los gatitos	9
Añadiendo las rutas para insertar un gatito	10
Añadiendo las rutas mostrar el formulario de inserción	10

NOTA: El proyecto **completo** está subido al repositorio siguiente de github:  
<https://github.com/kant003/transformaloTu.git>

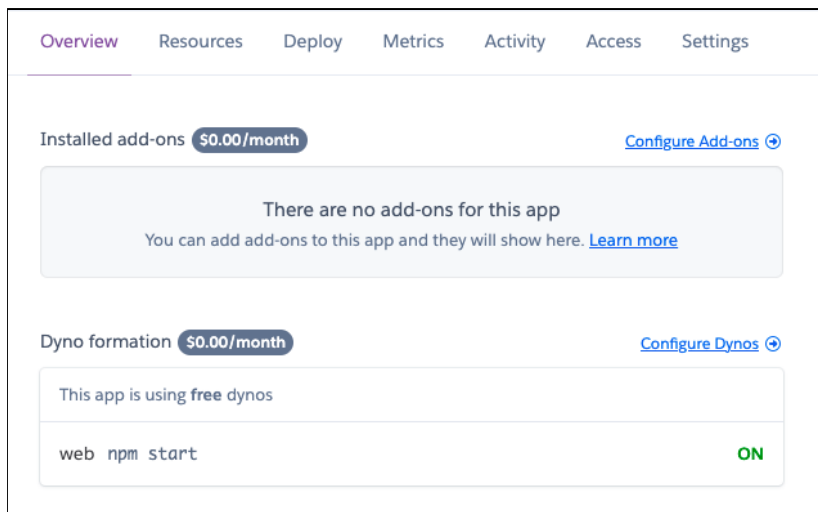
## Conectándonos con una base de datos

### Montando un servidor de base de datos (en Heroku)

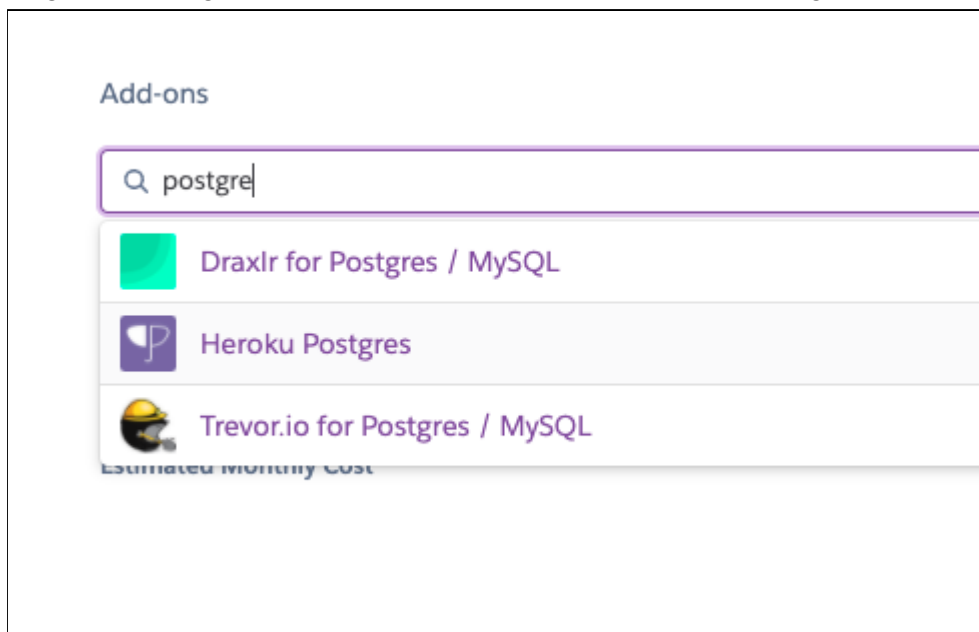
Logueate en heroku

Heroku no proporciona la posibilidad de montar también un servidor de base de datos (postgresql) (gratis). Vamos a instalarlo.




En el menú Overview de Heroku pulsamos en **Configure Add - ons**



Elegimos el plugin que vamos a instalar, en nuestro caso PostgreSQL



Seleccionamos el plan gratuito. (por desgracia, este plan nos dejará usar la base de datos un cierto tiempo, transcurrido este las credenciales de acceso cambiarán)



Heroku Postgresmisensor2

By choosing "Online Order Form", this will add **Heroku Postgres** on your personal **misensor2** application.

Plan name


Hobby Dev – Free



[View add-on details in Elements Marketplace](#)

By submitting this order form, you agree that the Add-on is governed

Ya podemos acceder al servidor de base de datos



The add-on `heroku-postgresql` has been installed.

 Quickly add add-ons from Elements

 **Heroku Postgres** 

Estimated Monthly Cost

Puedes observar que la base de datos está vacía

OverviewDurabilitySettingsDataclips											
HEALTH											
 Available											
REGION	Europe	PRIMARY	Yes	VERSION	13.2	CREATED	a few seconds ago	MAINTENANCE	Unsupported ⓘ	ROLLBACK	Unsupported ⓘ
UTILIZATION											
0 of 20				0 of 10,000				0 B		0	
CONNECTIONS				ROWS  IN COMPLIANCE				DATA SIZE		TABLES	

Lo que nos va a interesar del servidor son las credenciales de acceso (host, usuario, password y base de datos). Podemos consultar estos datos en el menú Settings → Credentials

[Overview](#) [Durability](#) [Settings](#) [Dataclips](#)

## ADMINISTRATION

### Database Credentials

Get credentials for manual connections to this database.

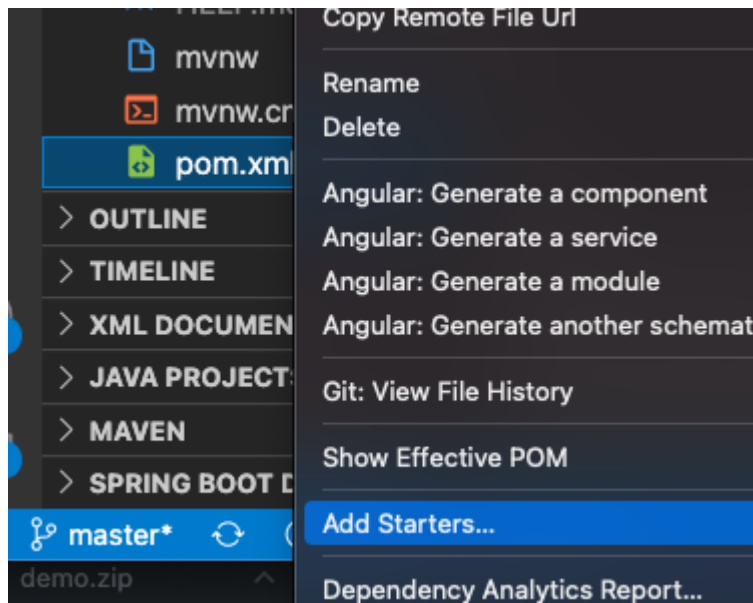
Please note that **these credentials are not permanent**.

Heroku rotates credentials periodically and updates applications where this database is a

Host	ec2-176-34-222-188.eu-west-1.compute.amazonaws.com
Database	dg3skk8l69j58
User	jvhuqiidazvukr
Port	5432
Password	123d7571caec4a43e6236619401ba1ed468a611c86800b4ec
URI	postgres://jvhuqiidazvukr:123d7571caec4a43e6236619401b
Heroku CLI	heroku pg:psql postgresql-triangular-74862 --app misensor2

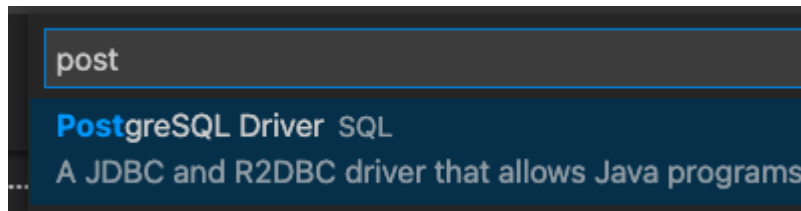
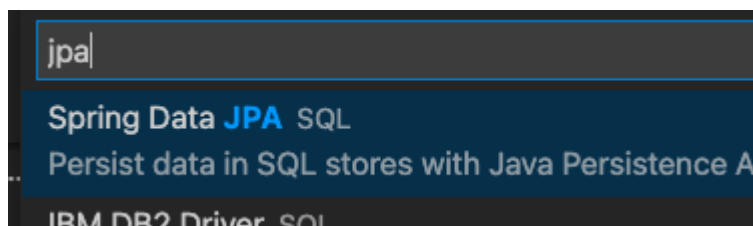
## Instalando las dependencias necesarias

Pulsa con el botón derecho en el fichero **pom.xml** y selecciona **Add Starters**



Vamos a añadir 2 dependencias:

- Spring Data JPA
- PostgreSQL Driver



Cuando las tengas añadidas pulsa en añadir dependencias

Comprueba que se han añadido correctamente a tu fichero pom.xml

```

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

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<scope>runtime</scope>
</dependency>
</dependencies>

```

## Problemas con la versión de postgresql

Si el pom falla...

```

<dependency>    org.postgresql:postgresql:jar:42.3.3 failed to transfer fr
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<scope>runtime</scope>
</dependency>

```

Cambia la versión a la 42.3.2

```

<dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<version>42.3.2</version>
<scope>runtime</scope>
</dependency>

```

## Programando el acceso a la base de datos desde el código

Indicando las credenciales de acceso

Edita el fichero resources → application.properties con el siguiente contenido

```
spring.datasource.url=jdbc:postgresql://XXXXXXX:5432/YYYYYY
spring.datasource.username=ZZZZZZZ
spring.datasource.password=MMMMMM
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

Donde XXXXXX = el host de tu servidor de base de datos

Importante solo coloca la url del servidor

YYYYYY= el nombre de la base de datos

ZZZZZZ= será el usuario

MMMMMM= será el password

## Creando el modelo (entity)

Con el modelo / Entidad vamos a poder **mapear** las tablas de nuestra base de datos con clases de nuestro programa escrito en java.

```
@Entity
@Table(name = "gatito")
public class GatitoModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(unique = true, nullable = false)
    private Long id;

    private String nombre;

    private Integer peso;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(columnDefinition = "TIMESTAMP DEFAULT CURRENT_TIMESTAMP")
    private Date fecha; // Si no pongo fecha, la fecha será la del sistema

    ///// aqui van los getters y setters (los omito para que sea más legible)
```

En este caso estamos creando una clase `GatitoModel` que tendrá los atributos `id`, `nombre`, `peso` y `fecha`.

Para indicar que esta clase mapea una tabla de nuestra base de datos la notamos con

```
@Entity
@Table(name="gatito")
public class GatitoModel {
```

Indicamos que el campo `id` de la tabla es autoincremental, único y no nulo con la notación

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(unique=true, nullable=false)
private Long id;
```

También indicamos que la fecha se rellena automáticamente con el día y hora actual usando la notación:

```
@Temporal(TemporalType.TIMESTAMP)
@Column(columnDefinition = "TIMESTAMP DEFAULT CURRENT_TIMESTAMP")
private Date fecha; // Si no pongo fecha, la fecha será la del sistema
```

## Creando el repositorio

En el repositorio vamos a indicar todas las consultas que se van a lanzar contra la base de datos.

Hemos optado por el uso de un ORM (en nuestro caso JPA) que nos va a facilitar muchísimo el trabajo con la base de datos.

Creamos un fichero llamado ***GatitoRepository*** (yo he optado por situarlo dentro de la carpeta `repositories`, para tener el código más ordenado)

```
@Repository
public interface GatitoRepository extends CrudRepository<GatitoModel, Long> {
}
```

Observa que la interface extiende de `CrudRepository`.

```
extends CrudRepository<GatitoModel, Long>
```

Esto permitirá que JPA haga todo el trabajo, pues nos va a ofrecer mecanismos para poder hacer `selects`, `updates`, `inserts` y `deletes` sin tener que escribir ni una sola línea de código SQL.



En los **generics** hay que indicarle el modelo con el que vamos a trabajar, en este caso **GatitoModel** y el tipo de la clave primara (id) que es un **Long**.

Si necesitáramos hacer consultas más complejas, podríamos añadirlas dentro de esta interfaz.

## Creando el servicio

El servicio nos va a permitir conectar nuestra aplicación con el exterior (en este caso la BD). Mediante el servicio vamos a lanzar las consultas usando el Repository que hemos creado en el paso anterior.

Creamos un fichero llamado **GatitoBDService** ( yo he optado por situarlo dentro de la carpeta services, para tener el código más ordenado)

```
@Service
public class GatitoBDService {
    @Autowired
    GatitoRepository gatitoRepository;

    public ArrayList<GatitoModel> obtenerTodosLosGatitos() {
        return (ArrayList<GatitoModel>) gatitoRepository.findAll();
    }

    public GatitoModel guardarGatito(GatitoModel gatito) {
        return gatitoRepository.save(gatito);
    }
}
```

Hemos añadido 2 métodos:

- **obtenerTodosLosGatitos**: que devuelve un ArrayList con todos los gatitos guardado en la base de datos. Observa como usamos el método findAll que nos provee JPA gracias a que estamos usando el repository que extiende de CrudRepository.
- **guardarGatito**: que recibe el gatito que vamos a guardar como parámetro y nos devuelve el gatito guardado (con el nuevo id generado). También usamos el repository para (con el método save) guardar el gatito en la base de datos de forma sencilla.

## Actualizando los controladores

Por último, tenemos que añadir las rutas adecuadas en los controladores, para que el cliente se pueda comunicar con el servidor indicando las acciones que desea ejecutar.

El controlador va a hechar mano de los servicios creado en los pasos anteriores, así que no olvides indicar a Spring que nos lo inyecten (por el mecanismo de inyección de dependencias)

```
@Autowired
GatitoBDService gatitoBDService;
```

## Añadiendo las rutas para listar todos los gatitos

En el fichero SaludaController.java, añadimos el siguiente código

```
@GetMapping("/listarGatitos")
public String gatitos() {
    return gatitoBDService.obtenerTodosLosGatitos().toString();
}
```

cuando el cliente acceda (por GET) a la ruta <http://localhost:8080/listarGatitos> le mostraremos en el navegador la lista de todos los gatitos almacenados en la base de datos

## Añadiendo las rutas para insertar un gatito

En el fichero SaludaController.java, añadimos el siguiente código

```
@PostMapping("/insertaGatito")
public String insertaGatito(@RequestParam Map<String, String> body) {
    System.out.println(body.get("nombre"));
    GatitoModel gatito = new GatitoModel();
    gatito.setNombre(body.get("nombre"));
    gatito.setPeso(Integer.parseInt(body.get("peso")));
    gatitoBDService.guardarGatito(gatito);
    return "he guardado los datos del gatito";
}
```

cuando el cliente nos lance una petición por POST a la ruta <http://localhost:8080/insertaGatito> guardaremos un nuevo gatito en la base de datos.

¿Cómo nos lanza el cliente una petición por POST? pues mediante un formulario html, donde colocaremos en su interior los datos que se van a enviar por el BODY de la petición http. (en nuestro caso el nombre y peso del gatito)

## Añadiendo las rutas mostrar el formulario de inserción

Dentro de la carpeta resources→ templates vamos a crear un nuevo fichero llamado formularioInsercionGatito.html con el siguiente contenido

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Formulario para insertar un gatito en la BD</h1>
  <form action="http://localhost:8080/insertaGatito" method="POST">
    <label for="nombre">Nombre:</label>
    <input type="text" name="nombre" value="miku">
    <br/>
    <label for="peso">Peso:</label>
    <input type="text" name="peso">
    <br/>
    <input type="submit" value="Insertar gatito">
  </form>
</body>
</html>
```

Como ves no es más que un simple formulario html con los inputs:

```
<input type="text" name="nombre">
<input type="text" name="peso">
```

El tag form posee como atributos

```
action="http://localhost:8080/insertaGatito"
method="POST"
```

El action será la ruta de nuestro servidor (backend) que va a recoger los parámetros (nombre y peso) y los va a guardar en la base de datos.

Para que el cliente pueda ver este formulario vamos a añadir la siguiente ruta:  
Edita el fichero WebController.java y añade el siguiente contenido:

```
@RequestMapping("/formularioGatito")
public String formularioGatito() {
    return "formularioInsercionGatito";
}
```

Cuando el cliente acceda a la ruta <http://localhost:8080/formularioGatito> le devolveremos (mediante thymeleaf) el formulario html que hemos creado en el paso anterior