



Java Academy Xideral 2022

Alumno: Moisés Vidal Hernández

Maestro: Miguel Angel Rugerio

Java Bootcamp Training

Examen: Semana 4

Diciembre 16 del 2022

1. Explica en qué consiste:

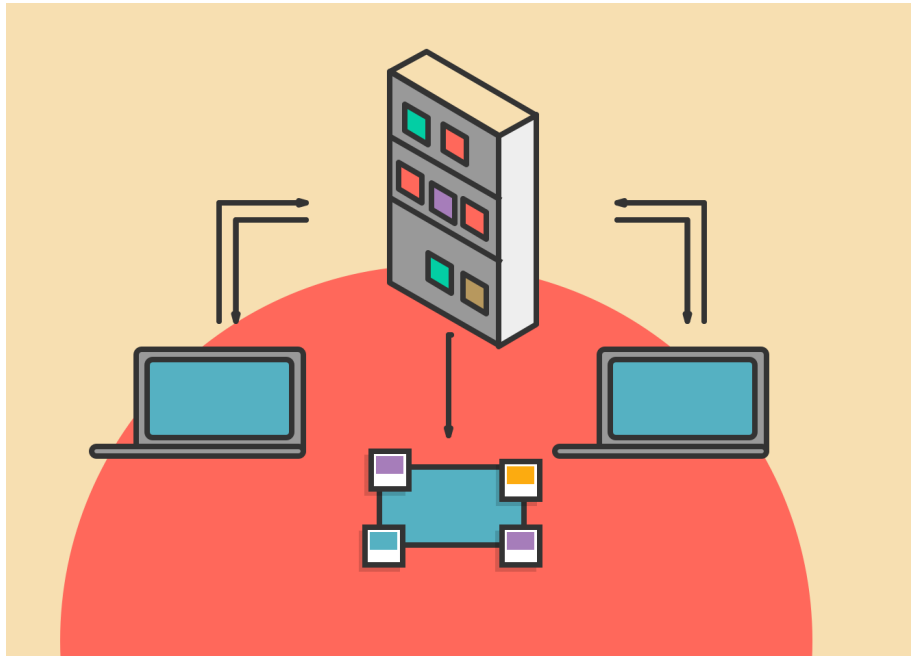
Trabajar con repositorios en GitHub

Un repositorio es todo el conjunto de archivos, que a su vez permiten la construcción de un proyecto.

Al subir nuestro repositorio en GitHub, lo que hacemos es subir todos los archivos de nuestro proyecto a una plataforma en la nube, que nos permite tener un respaldo y control del seguimiento a todos los cambios que se le hacen a dichos archivos, creación y borrado de estos.

El flujo de trabajo de los repositorios en GitHub, son dependientes del sistema de control de versiones GIT, ya que este nos permite visualizar una especie de rastro de los cambios. Algo así como un diario.

Al tener un repositorio en la nube en este caso GitHub, tenemos una copia global y cada miembro del equipo tiene una copia local en su computadora, para entenderlo mejor veamos la siguiente imagen.



El proyecto general está en un servidor, y cada miembro del equipo tiene una copia local en su computadora personal, cada miembro puede trabajar haciendo cambios dentro de una rama que esté basada en el repositorio global.

Los cambios se verán reflejados hasta que el miembro del equipo realice un Pull Request.

Branches

Las ramas en git, son una especie de camino en el cual se concentran los cambios de cada característica a desarrollar.

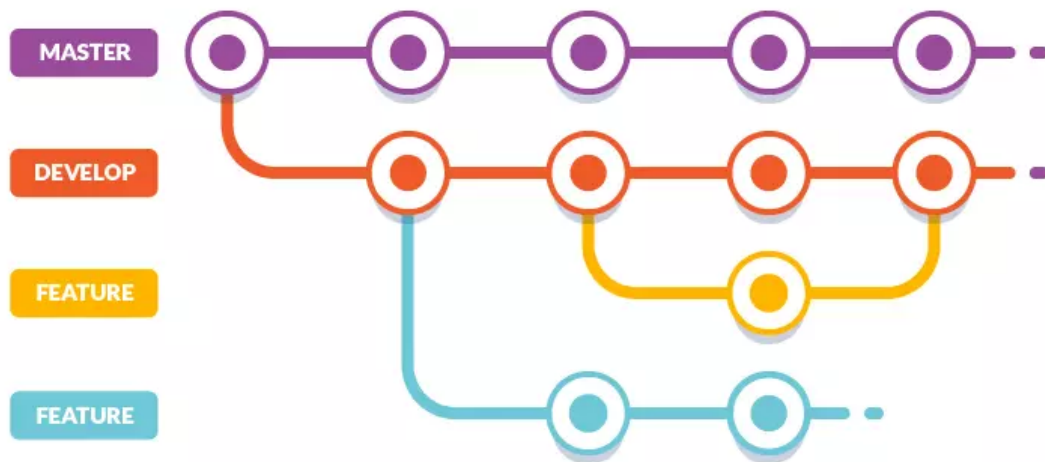
Obligatoriamente se tendrá al menos una rama base (main / master), esto depende del equipo de desarrollo y se pueden tener hasta 3 ramas base

- Develop
- QA
- Producción

Las ramas son creadas por cada desarrollador.

Se tiene que respetar cierta nomenclatura para nombrar nuestras ramas (esto depende del equipo de desarrollo)

- Para agregar una nueva característica:
"feature/#num_NombreCaracteristica". Ejemplo:
feature/#12_RegistroUsuarios
- Para solucionar un bug con prioridad moderada o baja:
"bugfix/#num_NombreBug"
- Para solucionar un bug con prioridad alta: **"hotfix/#num_NombreBug"**
- Para el entregable o versión con avances para mostrar al usuario:
"release/01"



Merge

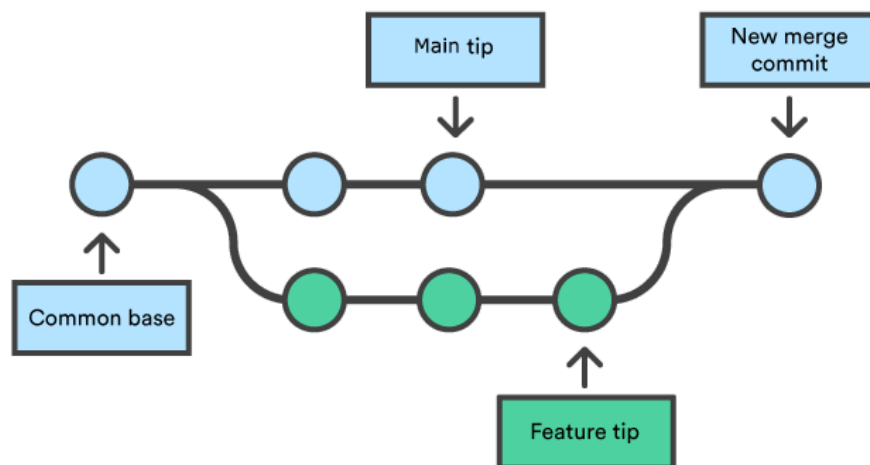
Básicamente es la fusión de cambios entre 2 ramas.

Si aplicamos un Merge, solo se encontraran los caminos del registro de cambios de nuestras ramas

Por ejemplo si estamos en una rama con una característica nueva y queremos unir los cambios de otra rama activa. Debemos jalar o traer la rama activa siguiendo los siguientes pasos.

- Primero nos situamos en la rama en la cual queremos traer los cambios: `git checkout feature/#23_LoginFrontend`
- Despues identificamos la rama de la cual queremos traer los cambios en este caso la rama que jalaeremos sera: `feature/#54_ApiLogin`
- Una vez identificadas las ramas, procedemos a hacer el merge con el comando: `git merge feature/#54_ApiLogin`
- Si surgen conflictos tendremos que resolverlos, en caso de que no surja ninguno, la rama sobre la que estamos pocisionados (`feature/#23_LoginFrontend`) jalara los cambios de la rama activa (`feature/#54_ApiLogin`)

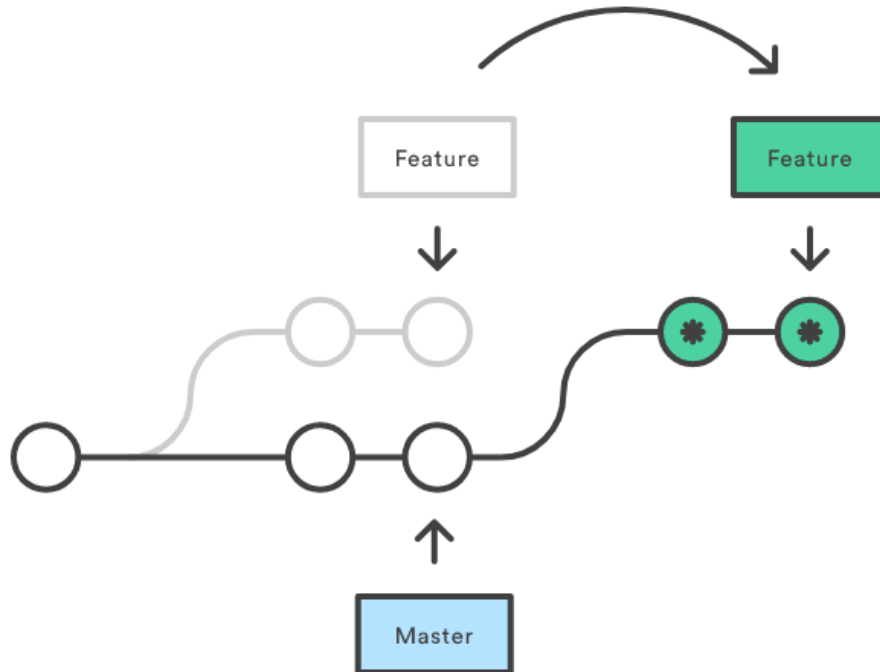
Un ejemplo más simple es cuando se hace un PR (Pull Request), github por detras hace un Merge, jalando los cambios de nuestra rama (feature) dentro de la rama base (main / master)



Rebase

El rebase hace algo similar al merge, pero con la excepción de que por detrás hace la unión de 2 ramas en 1 sola

Si aplicamos un rebase, el camino o registro de cambios de la rama en la que estamos trabajando se perderá una vez hecho el PR

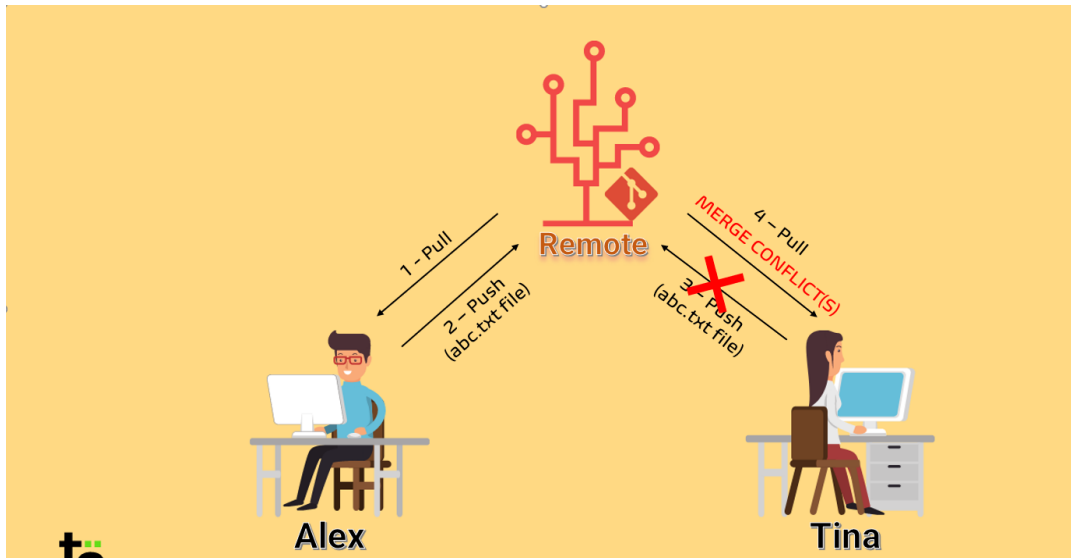


Por ejemplo si estamos en una rama con una característica nueva a desarrollar y queremos integrar con la rama base (máster o main), debemos hacer lo siguiente

- Primero nos situamos en la rama en la cual queremos traer los cambios: `git checkout feature/#23_LoginFrontend`
- Después identificamos la rama de la cual queremos traer los cambios en este caso la rama que jalaremos será: `main`
- Una vez identificadas las ramas, procedemos a hacer el merge con el comando: `git rebase main`
- Si surgen conflictos tendremos que resolverlos, en caso de que no surja ninguno, la rama sobre la que estamos posicionados (`feature/#23_LoginFrontend`) jalara los cambios de la rama activa (`main`)
- Para el caso de que surjan conflictos, debermos resolver conflictos y hacer un commit, una vez hecho esto continuamos con `git rebasse --continue`

Conflicts

Son problemas causados, principalmente cuando 2 desarrolladores o más están editando los mismos archivo o directorios



Estos conflictos forzosamente se deben resolver de manera manual
Por lo regular se hacen presentes cuando hacemos un pull, rebase o merge, y nuestro código puede verse como se muestra en la siguiente imagen.

```
colors.txt x
src > colors.txt
1 red
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2 <<<<<<< HEAD (Current Change)
3 green
4 =====
5 white
6 >>>>>>> his-branch (Incoming Change)
7 blue
```

- Los cambios que sean de tu rama se encontrarán en la sección del incoming. Si queremos tomar estos cambios debemos aceptar los que vengan de incoming, la desventaja de esto es que los cambios de la rama main no se considerarán.
- Los cambios que sean de la rama main o del repositorio global se encontrarán en la sección Head. Si queremos tomar los cambios de la rama main ,debemos aceptar los que vengan como current changes. El problema de esto es que los cambios que tengamos en nuestra rama serán descartados.
- La alternativa para no perder cambios entre 2 ramas es aceptar ambos con la opción Acep both changes, esta nos dejara tanto los cambios de la rama main y nuestra rama local, ahora solo será labor nuestra corregir el código manualmente.

Pull Request

Los Pull Request (PR), son aquellas peticiones o solicitudes para integrar los cambios que has realizado en tu rama y así integrarlos dentro de la rama base (main, master, develop, QA, etc.)



Para hacer un PR, es necesario que tu rama y tus cambios ya sean públicos (que figuren dentro del repositorio en GitHub).

El proceso de hacer un PR se hace desde la plataforma donde esté alojado el repositorio, ya sea GitHub, GitLab o BitBucket. Este proceso conlleva una revisión de los cambios realizados, una retroalimentación o feedback y posibles correcciones antes de unir los cambios realizados.

El PR puede involucrar la unión de todos los comits dentro de la rama base (lo que es un Merge) o la unión de todos los commits perder su rastro e integrarlos en uno solo (Rebase), pero esto ya depende de la consideración de cada desarrollador.

Fork

Es una especie de copia directa a tu perfil de un repositorio que se tenga en GitHub. Al hacer un fork estaremos haciendo una especie de seguimiento de todos los cambios que se realicen en el repositorio.

Es importante mencionar que al hacer un fork, es imprescindible estar haciendo sincronización constante para estar al día con el rastro de cambios, y luego estar haciendo un git pull en caso de que lo tengas clonado.

Como alternativa y mejor práctica recomiendo hacer un git clone y solo estar haciendo pull cuando se requiera, así estarás más al margen con el seguimiento de cambios.

Stash

Es un comando que nos permite ocultar nuestros cambios o quitarlos temporalmente de la pila de cambios que se tenga en ese momento.

Es usado comúnmente cuando se requiere hacer un pull desde una rama base.

Es recomendable usarlo cuando cambiamos de ramas, hacemos pull, merge o rebase.

Clean

Con git clean, descartamos aquellos archivos que están en la cola de cambios, pero aún no han sido agregados por nosotros como cambios significativos.

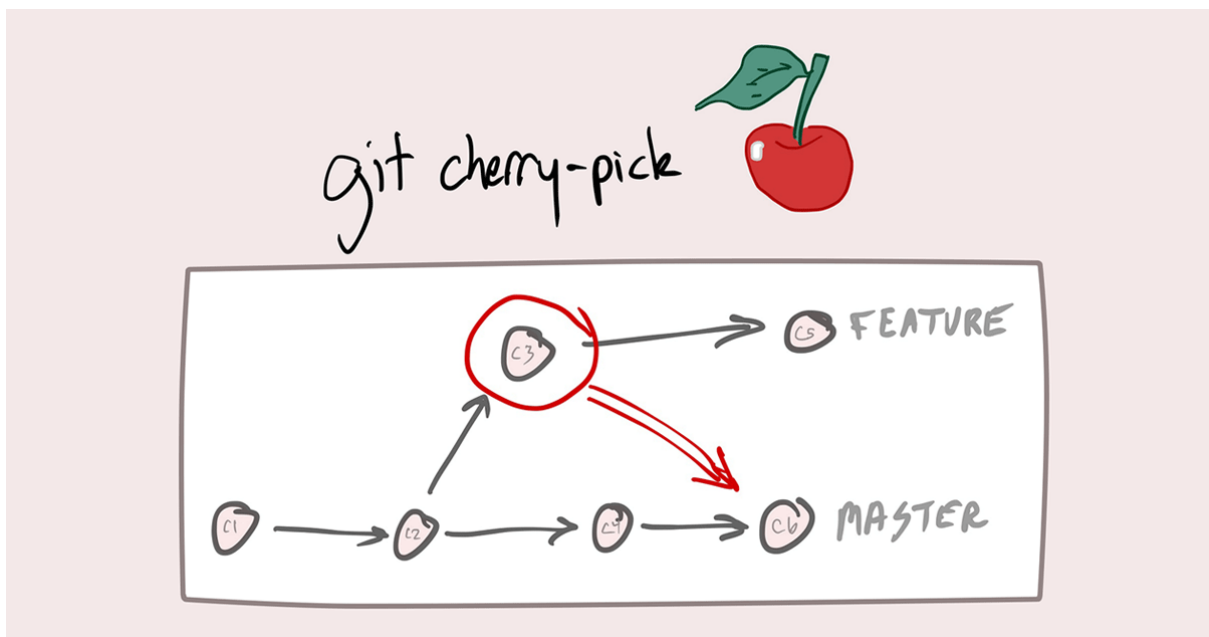
Siendo más específico, se limitarán o se descartarán los archivos con cambios y que no se han agregado con el comando “git add <archivo>”.

Cherry Pick.

Es un comando mediante el cual, podemos traer commits específicos desde otra rama, pero sin la necesidad de unir sus cambios o fusionar ambas ramas en 1 sola.

Es usado a menudo cuando surgen errores en ramas base alternas como develop o producción.

Es usado en casos muy específicos, pero igual depende del equipo de desarrollo.



2. Cuáles son las mejores prácticas de los Api REST.

Las API REST, son transferencias de estado representativo las cuales devuelven datos en formato JSON. Estas tiene verbos HTTP con los cuales se realizan distintas operaciones, dentro de los más comunes se encuentran:

- POST: Sirve para guardar nueva información o registros.
- GET: Solicitar (consultar) recursos que ya están guardados, la consulta a estos resultados se puede hacer de manera general o de manera específica con la siguiente nomenclatura `/peticion/{identificador}`. Por ejemplo si quisiéramos acceder a la información de un empleado específico sería la ruta seria asi: `/employees/3`
- PUT: Sirve para actualizar o modificar un recurso existente, para actualizar un registro se establecerá una ruta al registro específico respetando una nomenclatura parecida a la del Get, un ejemplo para actualizar un registro de empleado `/employees/12`.
- DELETE: Sirve para eliminar un registro existente, para eliminar un registro retomando nuestro ejemplo de empleados sería de la siguiente forma: `employees/12`.

A continuación dejamos una imagen con lo antes mencionado, para que se pueda tener un poco más de entendimiento.

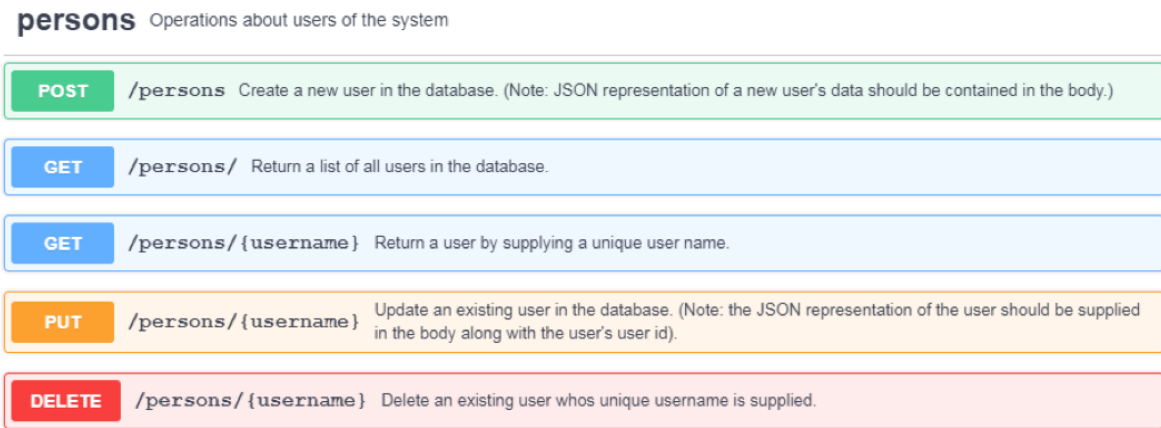


Diagrama de operaciones REST para personas. El título es "persons" con el subtítulo "Operations about users of the system". Se muestran cinco operaciones:

- POST** `/persons`: Create a new user in the database. (Note: JSON representation of a new user's data should be contained in the body.)
- GET** `/persons/`: Return a list of all users in the database.
- GET** `/persons/{username}`: Return a user by supplying a unique user name.
- PUT** `/persons/{username}`: Update an existing user in the database. (Note: the JSON representation of the user should be supplied in the body along with the user's user id).
- DELETE** `/persons/{username}`: Delete an existing user whos unique username is supplied.

3. Desarrolla una web app que permita exponer servicios REST en JSP's y JSON.

Justo como en ejemplos anteriores, este ejercicio fue proporcionado por el instructor, para que lo tomemos de base y le realicemos las actualizaciones correspondientes.

- Al estar usando Spring Boot, tenemos nuestra clase main la se ejecutará como una aplicación java, pero por detrás estará levantando el servidor de tomcat y realizará las configuraciones correspondientes.

```
CruddemoApplication.java X
1 package com.springboot.moridosSAdeCV;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class CruddemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(CruddemoApplication.class, args);
10    }
11 }
12
13
```

- Dentro del mismo paquete (com.luv2code.springboot.cruddemo) en el que se encuentra nuestra clase principal, tenemos otra clase que lleva el nombre de JpaConfig, dentro de esta clase especificamos el nombre de nuestra conexión, el nombre del usuario por el cual estaremos accediendo, password, puerto y base de datos.

```
@Bean(name = "mySqlDataSource")
@Primary
public DataSource mySqlDataSource()
{
    DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
    dataSourceBuilder.url("jdbc:mysql://localhost:3306/funeraria_bellakoza?useSSL=false&serverTimezone=UTC");
    dataSourceBuilder.username("springstudent");
    dataSourceBuilder.password("springstudent");
    return dataSourceBuilder.build();
}
}
```

- Dentro del paquete `com.luv2code.springboot.cruddemo.dao`, tenemos archivos escritos bajo el patrón de datos de acceso (DAO), el primer archivo es una interfaz, la cual contiene los métodos abstractos que se estarán desarrollando en otras clases que emplean Hibernate y Jdbc.

```

6 public interface MoridoDAO {
7
8     List<Morido> findAll();
9
10    Morido findById(int theId);
11
12    void save(Morido elMorido);
13
14    void deleteById(int theId);
15
16 }

```

- El segundo archivo contenido es una clase que implementa la interface antes mencionada, dentro de esta clase se integra la api JDBC, esto nos ayuda a conectarnos a la BD y realizar las operaciones básicas del crud, hasta cierto punto ya teníamos el método de listar y seleccionar por ID desarrollados. Por lo tanto nuestra labor era desarrollar el método para guardar un nuevo registro y eliminar.
- Para el caso de guardar nuevos registros y actualizar, estas acciones se encuentran dentro de un mismo método, dentro del cual se maneja el try with resources, internamente se evalúa que si el id tiene algún valor diferente a 0 se realizará una modificación en caso contrario se realizará un nuevo registro.

```

@Override
public void save(Morido theMorido) {
    // TODO Auto-generated method stub
    String sql = null;
    if (theMorido.getId() == 0) {
        sql = "insert into moridos (nombre, apellidos, edad, fecha_moricion, hora_moricion, lugar_moricion, causa_moricion) VALUES (?, ?, ?, ?, ?, ?, ?)";
    } else {
        sql = "update moridos set nombre=?,apellidos=?,edad=?,fecha_moricion=?,hora_moricion=?,lugar_moricion=?,causa_moricion=? where id=?";
    }
    System.out.println("sql = " + sql);
    try ( Connection myConn = dataSource.getConnection(); PreparedStatement myStmt = myConn.prepareStatement(sql); ) {
        myStmt.setString(1, theMorido.getNombre());
        myStmt.setString(2, theMorido.getApellidos());
        myStmt.setInt(3, theMorido.getEdad());
        myStmt.setString(4, theMorido.getFechaMoricion());
        myStmt.setString(5, theMorido.getHoraMoricion());
        myStmt.setString(6, theMorido.getLugarMoricion());
        myStmt.setString(7, theMorido.getCausaMoricion());
        if(theMorido.getId()!=0) {
            myStmt.setInt(8,theMorido.getId());
        }
        System.out.println("myStmt = " + myStmt);
        myStmt.executeUpdate();
        // retrieve data from result set row
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

- En esta clase tenemos el método de eliminar un registro mediante un id.

```
@Override
public void deleteById(int theId) {
    // TODO Auto-generated method stub
    String sql = "delete from moridos where id=?";
    try ( Connection myConn = dataSource.getConnection();
        PreparedStatement myStmt = createPreparedStatement(myConn, theId, sql);) {

        System.out.println("myStmt = " + myStmt);
        myStmt.executeUpdate();
        // retrieve data from result set row
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

- Dentro del mismo paquete tenemos otra clase que implementa Hibernate, básicamente esta clase hace lo mismo que en la que usamos JDBC, solo que lo hace de una manera más simplificada.
- Dentro del paquete “com.luv2code.springboot.cruddemo.entity”, tenemos una clase sobre la cual se basa el sentido y contexto de nuestra aplicación, esta clase se contiene atributos propios que hacen referencia a los datos de un empleado.
- Esta clase no lleva métodos de acceso ni constructores, ya que las notaciones de las 2 primeras líneas la hacen esa implementación, esto se puede ver en el ejemplo de como se usa con jdbc.

```
6 @Data
7 @NoArgsConstructor
8 @AllArgsConstructor
9 @Entity
10 @Table(name="moridos")
11 public class Morido {
12
13     @Id
14     @GeneratedValue(strategy=GenerationType.IDENTITY)
15     @Column(name="id")
16     private int id;
17
18     @Column(name="nombre")
19     private String nombre;
20
21     @Column(name="apellidos")
22     private String apellidos;
23
24     @Column(name="edad")
25     private int edad;
26
27     @Column(name="fecha_moricion")
28     private String fechaMoricion;
29
30     @Column(name="hora_moricion")
31     private String horaMoricion;
32
33     @Column(name="lugar_moricion")
34     private String lugarMoricion;
35
36     @Column(name="causa_moricion")
37     private String causaMoricion;
38 }
```

- En la clase MoridoRestController, hacemos lo mismo solo que aquí usamos servicios rest y la respuesta o información que se retornara sera mediante objetos JSON.

```
@RequestMapping("/rest")
public class MoridoRestController {

    private MoridoService employeeService;

    @Autowired
    public MoridoRestController(MoridoService theMoridoService) {
        employeeService = theMoridoService;
    }

    // expose "/employees" and return list of employees
    @GetMapping("/moridos")
    public List<Morido> findAll() {
        return employeeService.findAll();
    }

    // add mapping for GET /employees/{employeeId}
    @GetMapping("/moridos/{moridoId}")
    public Morido getMorido(@PathVariable int moridoId) throws Exception {

        Morido theEmployee = employeeService.findById(moridoId);

        if (theEmployee == null) {
            throw new Exception("Employee id not found - " + moridoId);
        }

        return theEmployee;
    }

    // add mapping for POST /employees - add new employee
    @PostMapping("/moridos")
    public Morido addMorido(@RequestBody Morido theMorido) {

        // also just in case they pass an id in JSON ... set id to 0
        // this is to force a save of new item ... instead of update

        employeeService.save(theMorido);

        return theMorido;
    }
}
```

- Adicionalmente tenemos el paquete “com.luv2code.springboot.cruddemo.service”, dentro de este paquete tenemos:
- Una interface con el nombre de MoridoService, la cual lleva métodos abstractos, mismos que serán desarrollados dentro de la clase “MoridoServiceImpl”

```
public interface MoridoService {

    public List<Morido> findAll();

    public Morido findById(int theId);

    public void save(Morido theEmployee);

    public void deleteById(int theId);

}
```

- Dentro de la clase “MoridoServiceImpl”, tenemos la implementación de la interface antes descrita, aquí se ejecutan las acciones del crud. Pero lo más importante se encuentra en la línea 20 justo después de la anotación @Qualifier, ya que ahí se define que tipo de implementación usar, si Jdbc o Hibernate.

```
public class MoridoServiceImpl implements MoridoService {

    private MoridoDAO moridoDAO;

    @Autowired
    public MoridoServiceImpl(@Qualifier("moridoDAOJdbcImpl") MoridoDAO theMoridoDAO) {
        moridoDAO = theMoridoDAO;
    }

    @Override
    @Transactional
    public List<Morido> findAll() {
        return moridoDAO.findAll();
    }

    @Override
    @Transactional
    public Morido findById(int theId) {
        return moridoDAO.findById(theId);
    }

    @Override
    @Transactional
    public void save(Morido theMorido) {
        moridoDAO.save(theMorido);
    }

    @Override
    @Transactional
    public void deleteById(int theId) {
        moridoDAO.deleteById(theId);
    }

}
```

4. Desarrolla una web app que permita consumir el servicio Rest mediante una JSP's y JSON del proyecto anterior.

Para este proyecto debemos consumir los servicios REST expuestos con el proyecto anterior. Para ello tuvimos que desarrollar un proyecto con Spring y que explicamos a continuación.

- Primero tenemos a nuestro modelo que consiste en una clase llamada Morido, la cual tiene sus atributos y su respectivos métodos de acceso, con los cuales manejaremos la información y la presentaremos en la jsp.

```
3 public class Morido {
4
5     private int id;
6     private String nombre;
7     private String apellidos;
8     private int edad;
9     private String fechaMoricion;
10    private String horaMoricion;
11    private String lugarMoricion;
12    private String causaMoricion;
13
14    public Morido(){
15
16    }
17    public int getId() {
18        return id;
19    }
20
21    public void setId(int id) {
22        this.id = id;
23    }
24
25    public String getNombre() {
26        return nombre;
27    }
28
29    public void setNombre(String nombre) {
30        this.nombre = nombre;
31    }
32
33    public String getApellidos() {
34        return apellidos;
35    }
36
37    public void setApellidos(String apellidos) {
38        this.apellidos = apellidos;
39    }
40
41    public int getEdad() {
42        return edad;
43    }
```

- Tenemos una interface que lleva por nombre CustomerService, dentro de esta interface se tiene métodos abstractos, los cuales serán desarrollados en la clase CustomerServiceClientImpl.

```

7 public interface CustomerService {
8
9     public List<Morido> getMoridos();
10
11     public void saveMorido(Morido theCustomer);
12
13     public Morido getMorido(int theId);
14
15     public void deleteMorido(int theId);
16
17 }

```

- Dentro de la clase CustomerServiceClientImpl, implementamos los metodos abstractos de la interface antes descrita, dentro de cada metodo se define la lógica para cada acción que se desea realizar mediante llamadas Rest.

```

17 public class CustomerServiceRestClientImpl implements CustomerService {
18
19     private RestTemplate restTemplate;
20
21     private String crmRestUrl;
22
23     private Logger logger = Logger.getLogger(getClass().getName());
24
25     @Autowired
26     public CustomerServiceRestClientImpl(RestTemplate theRestTemplate,
27         @Value("${crm.rest.url}") String theUrl) {
28
29         restTemplate = theRestTemplate;
30         crmRestUrl = theUrl;
31
32         logger.info("Loaded property: crm.rest.url=" + crmRestUrl);
33     }
34
35     @Override
36     public List<Morido> getMoridos() {
37
38         logger.info("***OBTENER LISTA DE CLIENTES DESDE EL SERVICE REST CLIENTE");
39         logger.info("in getCustomers(): Calling REST API " + crmRestUrl);
40
41         // make REST call
42         ResponseEntity<List<Morido>> responseEntity
43             = restTemplate.exchange(crmRestUrl, HttpMethod.GET, null,
44                 new ParameterizedTypeReference<List<Morido>>() {
45                 });
46
47         // get the list of customers from response
48         List<Morido> customers = responseEntity.getBody();
49
50         logger.info("in getMorido(): customers" + customers);
51
52         return customers;
53     }

```


- En la clase CustomerController, tenemos una serie de métodos en donde ejecutamos acciones que interactúan directamente con la base de datos y en base a la acción a realizar redirigimos a cierta JSP.

```

18 @RequestMapping("/moridos")
19 public class CustomerController {
20
21     // need to inject our customer service
22     @Autowired
23     private CustomerService customerService;
24
25     @GetMapping("/list")
26     public String listMoridos(Model theModel) {
27
28         // get customers from the service
29         List<Morido> theCustomers = customerService.getMoridos();
30
31         // add the customers to the model
32         theModel.addAttribute("moridos", theCustomers);
33
34         return "list-moridos";
35     }
36
37     @GetMapping("/showFormForAdd")
38     public String showFormForAdd(Model theModel) {
39
40         // create model attribute to bind form data
41         Morido theCustomer = new Morido();
42
43         theModel.addAttribute("moridos", theCustomer);
44
45         return "moridos-form";
46     }
47
48     @PostMapping("/saveMorido")
49     public String saveCustomer(@ModelAttribute("moridos") Morido theCustomer) {
50
51         // save the customer using our service
52         customerService.saveMorido(theCustomer);
53
54         return "redirect:/moridos/list";
55     }
56

```

- En el archivo application-properties, definimos el enlace desde el cual estaremos consumiendo el servicio REST expuesto.

```

application.properties ×
1 #
2 # The URL for the CRM REST API
3 # - update to match your local environment
4 #
5 #crm.rest.url=http://localhost:8080/spring-crm-rest/api/custom
6 #crm.rest.url=http://localhost:8888/rest/moridos

```


- Para añadir y actualizar, tenemos una JSP que nos mostrará un formulario.

```

1 <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <title>Save Customer</title>
6 <link type="text/css" rel="stylesheet"
7     href="${pageContext.request.contextPath}/resources/css/style.css">
8
9 <link type="text/css" rel="stylesheet"
10     href="${pageContext.request.contextPath}/resources/css/add-customer-style.css">
11 </head>
12 <body>
13     <div class="container">
14         <div class="container-title">
15             <h2 class="title-page">Funeraria Bellakoza</h2>
16         </div>
17         <form:form class="form center" action="saveMorido"
18             modelAttribute="moridos" method="POST">
19             <!-- need to associate this data with customer id -->
20             <form:hidden path="id" />
21             <h3 class="title-form">Datos del difunto</h3>
22             <form:input path="nombre" placeholder="nombre" />
23             <form:input path="apellidos" placeholder="apellidos" />
24             <form:input path="edad" placeholder="edad" />
25             <form:input path="fechaMoricion" placeholder="Fecha de la defuncion" />
26             <form:input path="horaMoricion" placeholder="Hora en la que se quedo tiezo" />
27             <form:input path="lugarMoricion" placeholder="Lugar del desceso" />
28             <form:input path="causaMoricion" placeholder="Causa de muerte" />
29             <input type="submit" value="Save" class="save" />
30         <div class="container-back center">
31             <a class="btn-back"
32                 href="${pageContext.request.contextPath}/moridos/list">Back to
33                 List</a>
34         </div>
35     </form:form>
36 </div>
37 </body>
38 </html>

```

- Finalmente nuestra vista con los registros listados, los botones para añadir y actualizar, sería la siguiente.

The screenshot shows a web browser window with the URL `localhost:3030/crm-web-app-rest-client-demo/moridos/list`. The page title is "Funeraria Bellakoza". There is a button labeled "Agregar registro". Below it is a table with the following data:

Nombre del difunto	Apellidos	Edad	Fecha de defuncion	Hora de la Muerte	Lugar de la Muerte	Causa de la Muerte	Actualizar	Eliminar
David	Adams	63	2001-12-12	12:30 AM	su casita del maincra	lo balacaron los municipales	Actualizar	Eliminar
Samuel	Cornelio	24	2073-04-21	04:23 PM	en las miches	lo asalto el bryan	Actualizar	Eliminar
La llamada	Del google meet	63	2001-12-13	06:03 PM	en el gugle meet	gugle nos cobro	Actualizar	Eliminar
Moses	Vidal	63	2023-12-12	12:33 AM	en la cerveteria	lo levantaron por escuchar corrios tumbaos	Actualizar	Eliminar
Emmanuel	Reyes	63	2001-02-15	14:32 AM	En el lolito	no uso a garen xD	Actualizar	Eliminar

The background of the page features a decorative illustration of skeletons and candles.

- Para añadir o modificar un registro existente la vista sería la siguiente.

Funeraria Bellakoza

Datos del difunto

David

Adams

63

2001-12-12

12:30 AM

su casita del manicra

lo balaciaron los municipales

Save

Back to List

3. Desarrolla un programa Spring Batch que permita la carga por lotes a una base de datos SQL.

Como se ha venido manejando en ejercicios anteriores, para este de igual forma se nos proporcionó un proyecto base, el cual emplea Spring Batch para hacer transacciones por lotes (en grandes volúmenes) dentro de una base de datos.

Para ello explicaremos cada parte importante del proyecto.

- Como lo hemos venido manejando en los últimos proyectos, partimos de nuestra entidad, dentro de esta clase definimos el nombre de nuestra tabla, dentro de la cual estaremos almacenando los registros, en esta misma clase especificamos el nombre de las columnas de nuestra tabla.

```
12 @Entity
13 @Table(name = "CUSTOMERS_INFO")
14 @Data
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class Customer {
18
19     @Id
20     @Column(name = "CUSTOMER_ID")
21     private int id;
22     @Column(name = "FIRST_NAME")
23     private String firstName;
24     @Column(name = "LAST_NAME")
25     private String lastName;
26     @Column(name = "EMAIL")
27     private String email;
28     @Column(name = "GENDER")
29     private String gender;
30     @Column(name = "CONTACT")
31     private String contactNo;
32     @Column(name = "COUNTRY")
33     private String country;
34     @Column(name = "DOB")
35     private String dob;
36
37 }
```

- Enseguida tenemos una interface que lleva por nombre CustomerRepository, la cual hace herencia de la clase JpaRepository, la función de esta interface es realizar las operaciones de un crud, pero con la novedad de que ya no es necesario declarar métodos ni instancias SQL, ya que jpa hace estas operaciones automáticamente, solo tendremos que especificar el tipo de objeto que estamos manejando y el tipo de dato que es el identificador de cada registro.

```

BatchProcessingDemoApplication.java  JobController.java  CustomerProcessor.java  CustomerRepository.java X
1 package com.springboot.batch.repository;
2
3 import com.springboot.batch.entity.Customer;
4
5
6 public interface CustomerRepository extends JpaRepository<Customer,Integer> {
7 }
8

```

- Ahora pasamos a nuestra clase que lleva por nombre JobController, esta clase nos ayuda a definir el enlace mediante el cual estaremos realizando las transacciones además, que desde esta clase corremos todo el proceso de la transacciones agregando sus instancias y pasando los parametros.

```

18 public class JobController {
19
20     @Autowired
21     private JobLauncher jobLauncher;
22     @Autowired
23     private Job job;
24
25     @PostMapping("/importCustomers")
26     public void importCsvToDBJob() {
27
28         long horaInicio = System.currentTimeMillis();
29
30         JobParameters jobParameters = new JobParametersBuilder()
31             .addLong("startAt",horaInicio).toJobParameters();
32
33         try {
34             jobLauncher.run(job, jobParameters);
35         } catch (JobExecutionAlreadyRunningException |
36             JobRestartException |
37             JobInstanceAlreadyCompleteException |
38             JobParametersInvalidException e) {
39             e.printStackTrace();
40         }
41     }
42 }

```

- Dentro de la clase SpringBatchConfig, tenemos una serie de métodos, mediante los cuales estaremos realizando las transacciones, a continuación explicaremos cada método y parte importante de esta clase.
- Primero tenemos la sección de anotaciones que nos permitirán configurar, agregar los argumentos y habilitar el procesamiento batch.
- Enseguida tenemos las instancias a las interfaces antes descritas, mediante las cuales realizaremos inyección de dependencias.

```

25 @Configuration
26 @EnableBatchProcessing
27 @AllArgsConstructor
28 public class SpringBatchConfig {
29
30     private JobBuilderFactory jobBuilderFactory;
31     private StepBuilderFactory stepBuilderFactory;
32     private CustomerRepository customerRepository;
33 }

```

- El primer método lleva por nombre FlatFileItemReader, este método nos permitirá leer un archivo csv el cual contiene los registros, especificamos el nombre del proceso, indicamos que salte la primera fila ya que en esta se encuentra la cabecera.

```

@Bean
public FlatFileItemReader<Customer> reader() {
    FlatFileItemReader<Customer> itemReader = new FlatFileItemReader<>();
    itemReader.setResource(new FileSystemResource("src/main/resources/customers.csv"));
    itemReader.setName("csvReader");
    itemReader.setLinesToSkip(1);
    itemReader.setLineMapper(lineMapper());
    return itemReader;
}

```

- En el método LineMapper, este método especifica los separadores del nuestro archivo, ya que al ser un CSV estos archivos están separados con comas, de igual forma especificamos el título de las cabeceras.

```

private LineMapper<Customer> lineMapper() {
    DefaultLineMapper<Customer> lineMapper = new DefaultLineMapper<>();

    DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
    lineTokenizer.setDelimiter(",");
    lineTokenizer.setStrict(false);
    lineTokenizer.setNames("id", "firstName", "lastName", "email", "gender", "contactNo", "country", "dob");

    BeanWrapperFieldSetMapper<Customer> fieldSetMapper = new BeanWrapperFieldSetMapper<>();
    fieldSetMapper.setTargetType(Customer.class);

    lineMapper.setLineTokenizer(lineTokenizer);
    lineMapper.setFieldSetMapper(fieldSetMapper);
    return lineMapper;
}

```

- El método RepositoryItemWriter, se encarga de guardar o escribir la información, dentro del método se crea una instancia, se nombra al proceso y se retorna al mismo.

```
@Bean
public RepositoryItemWriter<Customer> writer() {
    RepositoryItemWriter<Customer> writer = new RepositoryItemWriter<>();
    writer.setRepository(customerRepository);
    writer.setMethodName("save");
    return writer;
}
```

- En el método Step1, radica toda la serie de acciones y la llamada a los ítems, se indica primero que se realice una lectura al archivo, ejecutar el procesamiento de la información leída, escribir la información, ejecutar la tarea y finalmente construir todo el step que cargara los registros del csv en una tabla sql.

```
@Bean
public Step step1() {
    return stepBuilderFactory.get("csv-step").<Customer, Customer>chunk(10)
        .reader(reader())
        .processor(processor())
        .writer(writer())
        .taskExecutor(taskExecutor())
        .build();
}
```

- En el método runJob, llamamos al método step1, para que realice la carga de información por lotes.

```
@Bean
public Job runJob() {
    return jobBuilderFactory.get("importCustomers")
        .flow(step1()).end().build();
}
```

- Finalmente tenemos el CustomerProcesor, dentro de esta clase llamamos al método y especificando la condición con la cual controlamos qué datos insertar en la base de datos.

```
6 public class CustomerProcessor implements ItemProcessor<Customer, Customer> {
7
8     @Override
9     public Customer process(Customer customer) throws Exception {
10         if(customer.getCountry().equals("China") || customer.getCountry().equals("Ukraine")) {
11             return customer;
12         }
13         return null;
14     }
15 }
16 }
```