

Prueba AlMundo

- Entorno de desarrollo
 - Se desarrolló bajo el IDE de Eclipse STS(Spring tolos suit).
 - Se generó un proyecto SpringBoot con maven desde la página <https://start.spring.io/>
 - Systema operativo Windows 7 64 bits
 - Java 8
 - Spring MVC
 - SpringBoot
 - Maven
 - MY SQL
 - H2(DBMS)

Introducción

Se pensó en un proyecto SpringBoot dado que la recomendación fue que el mismo se realizara sobre Spring, java y maven y también por aprovechar las características de arranque, eficiencia, los repositorys CRUD sobre las interfaces y evitar la conexión concurrente a base de datos, además de la creación de los objetos en base de datos por medio de la integración de JPA.

El modelo de negocio está constituido por 3 tablas

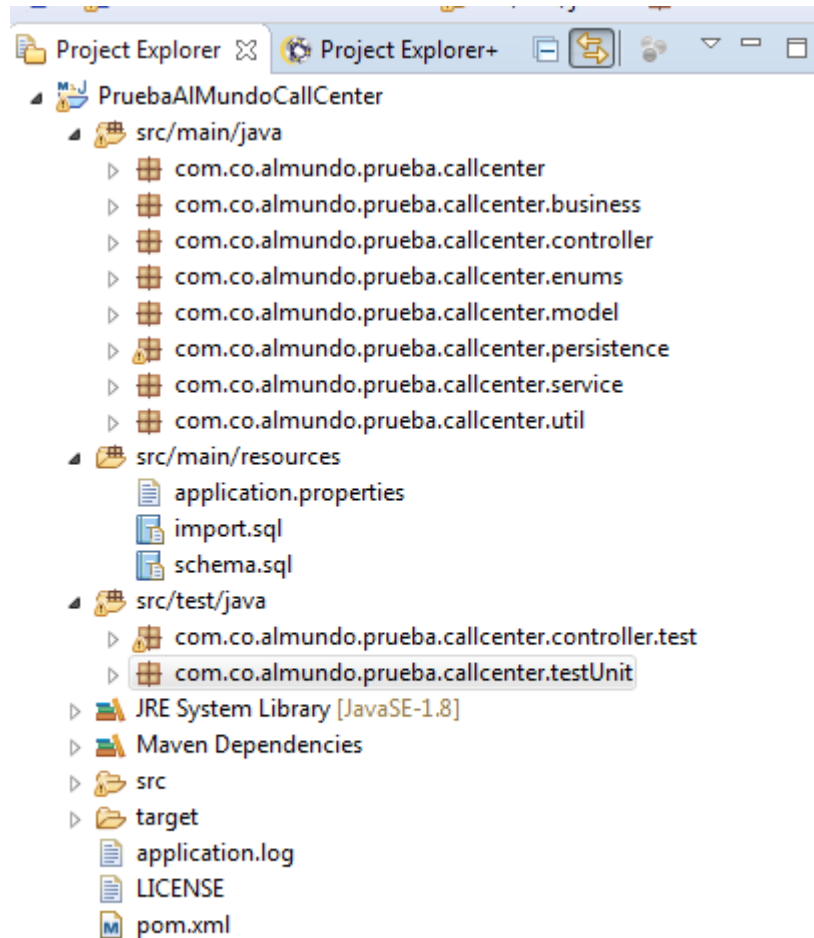
- **Call** (am_llamada)
- **Employe** (am_empleado)
- **EmployeeCall** (am_empleado_llamada),

Donde “**am**” es el sufijo de la compañía “**AlMundo**”, adicional a esto se crearon enumeradores para el manejo de los estados de las llamadas, los estados de los empleados y los cargos que ocupan cada uno de estos.

Con relación a la arquitectura se implementó el patrón **MVC**, en este caso no se creó interfaz gráfica, pero se relacionó el controlador (**Dispatcher**) con las clases contenedoras de los recursos necesarios, como consultas a bases de datos, procesamiento de llamadas, utilerías, hilos y tareas programadas, se implementó un singleton para el manejo de las colas las cuales se trabajaron con JMS y **Queue** de java útil, se trabajó el patrón **DAO**, pero no de la forma convencional si no que se aprovechó el apoyo de **SpringBoot**, y realizar las consultas JPQL directamente

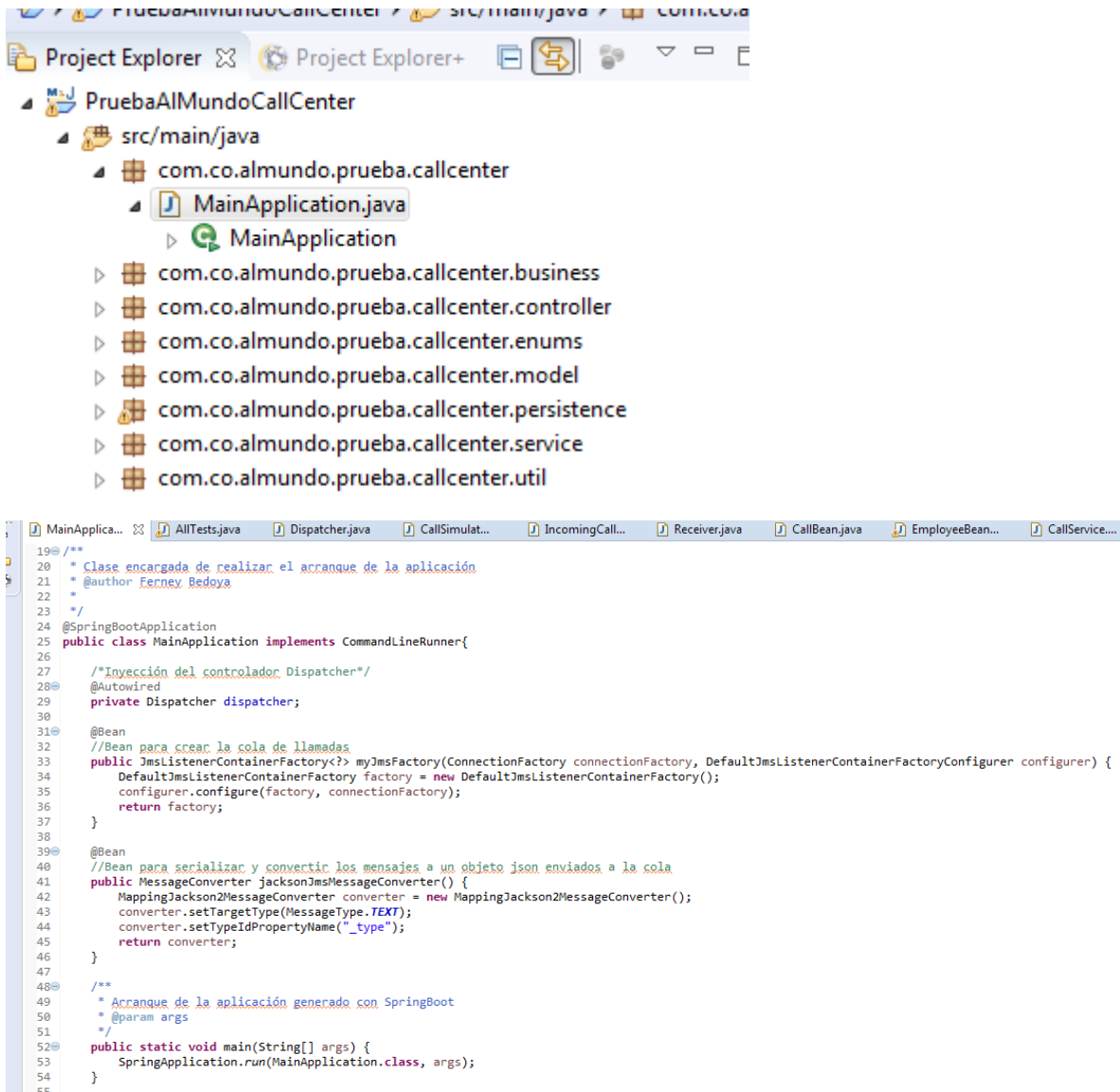
desde la interfaces y sin necesidad de implementar los métodos estándar de **inserción, update, merge, eliminación**, permitiendo ahorrar bastante tiempo.

La estructura de la aplicación quedo dividida en paquetes como es habitual y su presentación se puede ver a continuación.



Descripción de los paquetes y sus clases

Paquete `com.co.almundo.prueba.callcenter`: Contiene la clase principal de arranque para el aplicativo **`MainApplication.java`**, esta clase permite iniciar la aplicación y cargar los Beans necesarios para el llamado de las colas JMS.



Clase: **`MainApplication.java`**

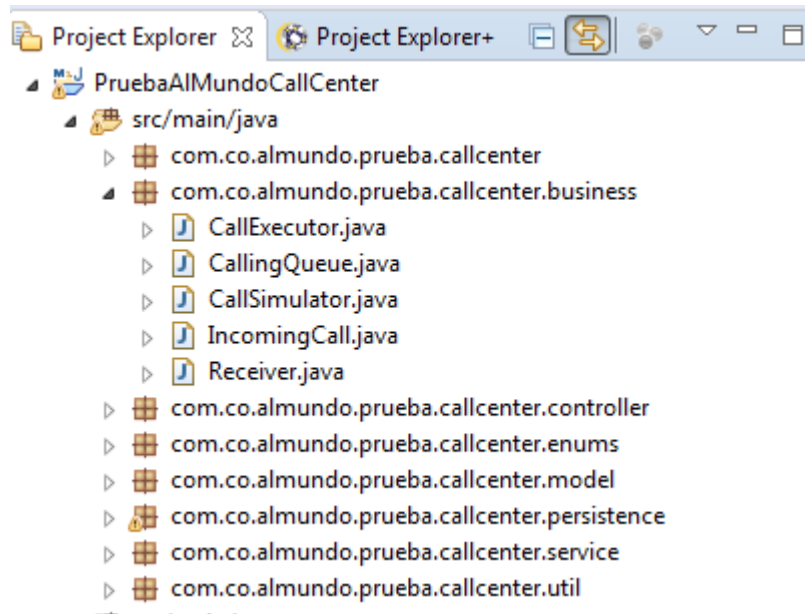
Paquete `com.co.almundo.prueba.callcenter.business`: Contiene las clases **`CallExecutor`**: Encargada de realizar el procesamiento de las llamadas.

`CallingQueue`: Encargada de Almacenar las llamadas en una cola.

`CallSimulator`: Simulador de llamadas.

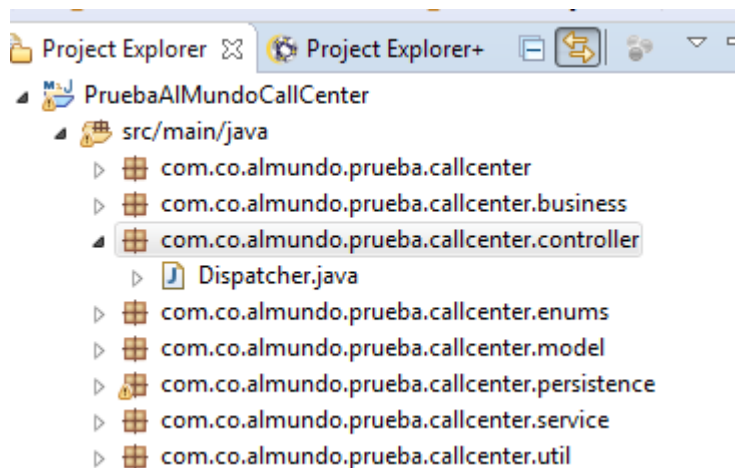
IncomingCall: Realizar llamadas y almacenarlas en una cola JSM.

Receiver: Encargada de recibir la llamada.



Paquete `com.co.almundo.prueba.callcenter.controller`:

Contiene la clase más importante del aplicativo, **Dispatcher** en esta se realiza la lógica de negocio, búsqueda de empleados disponibles, procesamiento de llamadas, asignación de llamadas, puesta de llamada en cola.



```

Dispatcher.java MainApplication.java CallExecutor.java application.properties AllTests.java CallTest.java
58
59  /*Random para la generación aleatoria de los segundos */
60  private Random random = new Random();
61
62  /**
63   * Metodo encargado de insertar llamada, calcular la duración de la llamada
64   * y asignar la llamada a un empleado disponible
65   * @param employee
66   * @param call
67   */
68  public void dispatchCall(Employee employee, Call call){
69
70      //Se inserta la llamada
71      callBean.save(call);
72      LOGGER.info("----- LA LLAMADA :"+ call.getId()+" HA SIDO ALMACENADA EN BASE DE DATOS -----");
73
74      //Se crea la duración de la llamada en un rango de 5 a 10 segundos
75      Long duration = (long) (MINIMUM_CALL + random.nextInt(MINIMUM_CALL));
76
77      //Se le asigna la llamada a un empleado disponible
78      EmployeeCall employeeCall = new EmployeeCall(employee, call, duration);
79      callService.processCall(employeeCall);
80  }
81

```

Método dispatchCall: encargado de insertar llamada, calcular la duración de la llamada y asignar la llamada a un empleado disponible.

```

~/
@Override
public void run() {

    Employee freeEmployee = null;
    Call call = null;
    /*se consultan los empleados disponibles*/
    employeesAvailable();

    while (running) {
        try {
            //Se valida si hay empleados libres
            if(!employeesAvailable.isEmpty()) {

                call = CallingQueue.listCallInQueue();

                if(call != null) {
                    try {
                        freeEmployee = searchFreePerson();
                    } catch (SQLException e) {
                        LOGGER.info("----- ERROR CONSULTANDO EL LISTADO DE EMPLEADOS -----");
                    }
                    //asigna llamada al empleado
                    dispatchCall(freeEmployee, call);
                }
            } else {
                LOGGER.info("----- LOS EMPLEADO NO ESTAN DISPONIBLES, POR FAVOR ESPERE -----");
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            LOGGER.error("----- ERROR EN LA ASIGNACIÓN DE LA LLAMADA -----");
        }
    }
}
}

```

Método RUN: verifica las llamadas que van llegando para poder Actualizarles el estado.

```

/**
 * Metodo que consulta los empleados disponibles
 * @return List<Empleado>, lista de empleados disponibles
 */
private void employeesAvailables(){
    employeesAvailable = Collections.synchronizedList(employeeBean.listEmployeeAvailable());
    employeeBusy();
}

/**
 * Metodo que consulta los empleados ocupados para temas se pruebas
 */
private void employeeBusy() {
    ArrayList<Empleado> listEmployee = new ArrayList<>();
    listEmployee = (ArrayList<Empleado>) Collections.synchronizedList(employeeBean.listEmployeeBusy());

    if(listEmployee != null && !listEmployee.isEmpty()) {
        for(Empleado employee : employeesAvailable) {
            LOGGER.info("----- EL EMPLEADO OCUPADO ES: "+employee.getName()+" -----");
        }
    }
}

```

Metodo employeesAvailables: consulta los empleados disponibles

employeeBusy : Método que consulta los empleados ocupados para temas se pruebas

```

/**
 * Metodo encargado de buscar el personal libre
 * @return Empleado
 * @throws SQLException
 */
private Employee searchFreePerson() throws SQLException {
    Employee employee = null;

    //Se realiza la búsqueda de operadores libres como primer filtro
    List<Empleado> operators = employeeBean.listEmployee();

    //Se valida que la consulta listar empleados haya retornado registros
    if(operators != null && !operators.isEmpty()){
        //Se recorre el listado de registros discriminando por Operador, Supervisor y Director
        //respectivamente
        for(Empleado freeEmployee: operators) {
            if(freeEmployee.getCharge().equals(TypeEmployee.OPERADOR) && freeEmployee.getState().equals(EmployeeStatus.DISPONIBLE)) {
                LOGGER.info("----- BUSCANDO EMPLEADO OPERADOR -----");
                employee = freeEmployee;
                this.employeesAvailable.remove(employee);
                LOGGER.info("----- SE HA ENCONTRADO EL EMPLEADO: "+employee.getName()+" CARGO: "+employee.getCharge()+" -----");
                break;
            }else if(freeEmployee.getCharge().equals(TypeEmployee.SUPERVISOR) && freeEmployee.getState().equals(EmployeeStatus.DISPONIBLE)) {
                LOGGER.info("----- BUSCANDO EMPLEADO SUPERVISOR -----");
                employee = freeEmployee;
                this.employeesAvailable.remove(employee);
                LOGGER.info("----- SE HA ENCONTRADO EL EMPLEADO: "+employee.getName()+" CARGO: "+employee.getCharge()+" -----");
                break;
            }else if(freeEmployee.getCharge().equals(TypeEmployee.DIRECTOR) && freeEmployee.getState().equals(EmployeeStatus.DISPONIBLE)) {
                LOGGER.info("----- BUSCANDO DIRECTOR -----");
                employee = freeEmployee;
                this.employeesAvailable.remove(employee);
                LOGGER.info("----- SE HA ENCONTRADO EL EMPLEADO: "+employee.getName()+" CARGO: "+employee.getCharge()+" -----");
                break;
            }
        }
    }
}

```

Método searchFreePerson encargado de consultar el personal libre

Paquete com.co.almundo.prueba.callcenter.enums: se alojaron los enumeradores para la aplicación solo son 3.

CallState: Estados de las llamadas.

EmployeeStatus: Estados de los empleados.

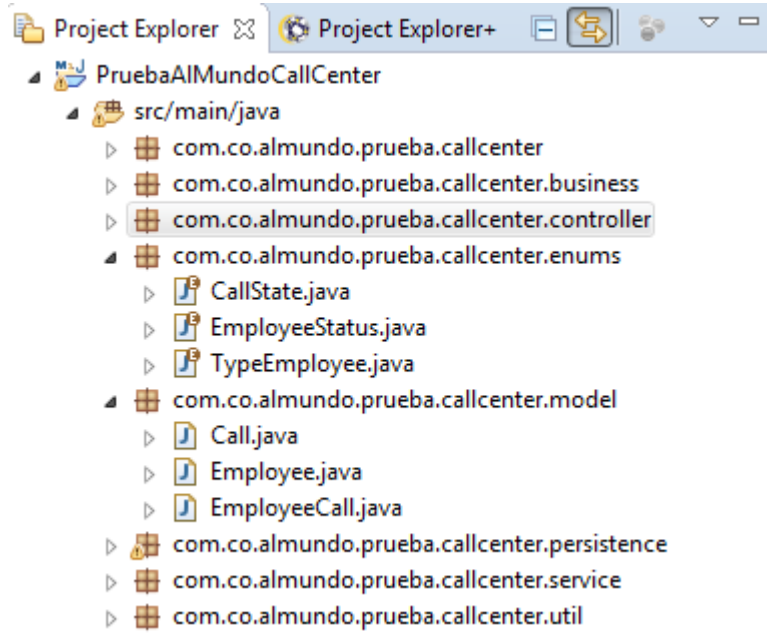
TypeEmployee: Cargo de los empleados.

Y en el **paquete com.co.almundo.prueba.callcenter.model**, se crearon las entidades en este paquete también tres.

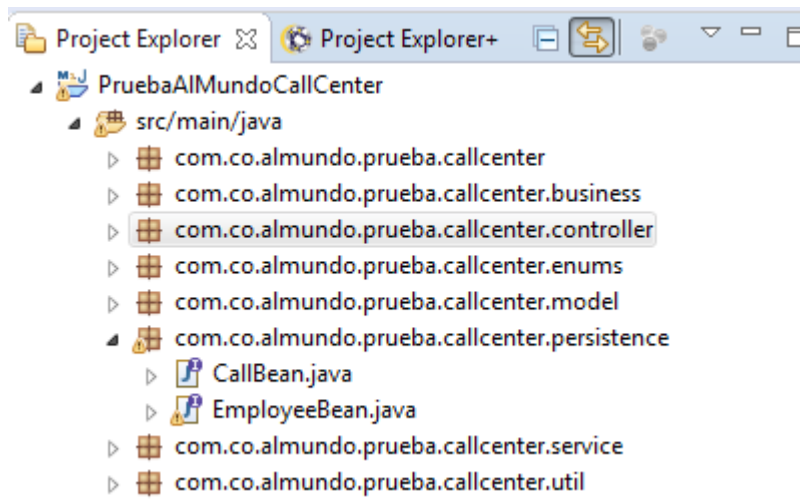
Call: Llamada(am_llamada).

Employee: Empleado(am_employee)

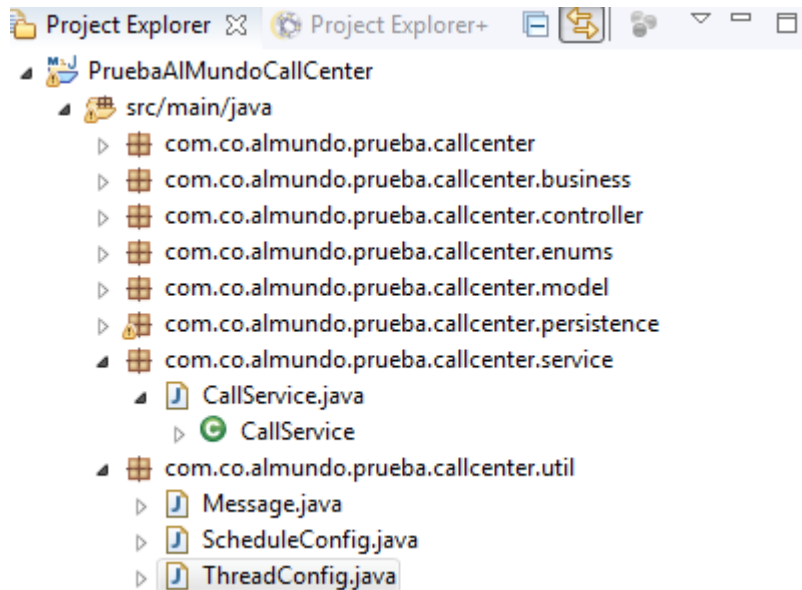
EmployeeCall: Empleado Llamada (am_employee_call)



Paquete com.co.almundo.prueba.callcenter.persistence: Contiene dos clases importantes pues ellas son las que conectan la aplicación con la base de datos por medio del repository de Spring, solo en una se implementaron métodos propios en **EmployeeBean**, para obtener la lista total de empleados, los ocupados, los disponibles y la clase **CallBean**, que solo se usó para obtener los métodos **merge**, **insert**, **remove** y **update** dentro de la clase **dispatcher**.



Paquete **com.co.almundo.prueba.callcenter.service** y el paquete **com.co.almundo.prueba.callcenter.util**, el primero almacena la clase **CallService**, que es la encargada de procesar las operaciones de negocio y el segundo las utilerías como Mensaje personalizado, componente para la creación de las tareas automáticas y la configuración del hilo de Spring.



Flujo de la aplicación

Una vez se ejecute la misma desde la clase principal **MainApplication.java**, spring comienza a realizar su trabajo y configura el contexto para comenzar a ejecutar la tarea programada de creación de llamadas **CallExecutor.java**, dando inicio al hilo de ejecución de la misma y comenzando a encolar las llamadas con las clases **IncomingCall.java**, **CallingQueue.java** y **CallExecutor.java**, se crea una tarea programada para que esté realizando llamadas constantemente, permitiendo que los usuarios disponibles se agoten y las llamadas puedan quedar encoladas hasta que un usuario termine la llamada que oscila entre 5 y 10 segundos, para ellos se creó un random que arroja esta duración, continuación se muestran cómo trabajan, cabe resaltar que las clases están debidamente documentadas para su mayor entendimiento.


```

/**
 * Clase encargada de realizar las llamadas y almacenarlas en una cola JMS
 * repite la llamada cada 5 segundos
 * @author fernbecr
 */
@Component
public class IncomingCall {

    private static final Logger LOGGER = LoggerFactory.getLogger(IncomingCall.class);

    @Autowired
    private ApplicationContext applicationContext;

    @Scheduled(fixedDelay = 5000)
    public void makeCall() {
        LOGGER.info("----- REALIZANDO LA LLAMADA Y LLAMANDO LA COLA JMS -----");
        JmsTemplate jmsTemplate = applicationContext.getBean(JmsTemplate.class);
        jmsTemplate.convertAndSend("user", new Employee(EmployeeStatus.DISPONIBLE, TypeEmployee.OPERADOR));
    }
}

```

IncomingCall.java

```

* Clase encargada de Almacenar las llamadas en una cola
* para su posterior procesamiento
* @author Ferney Bedoya
*/
public class CallingQueue {

    /*variables estaticas*/
    private static CallingQueue instance;

    /*Declaración de la cola para la llamada*/
    private Queue<Call> callQueue;

    /**
     * Conversión de la clase CallingQueue en un patron de diseño singleton
     */
    public static CallingQueue getInstance() {
        if(instance == null) {
            instance = new CallingQueue();
        }
        return instance;
    }

    /**
     * Se crea la cola de llamadas para mantener el orden de ingreso
     */
    private CallingQueue() {
        this.callQueue = new LinkedList<>();
    }

    /**
     * Metodo encargado de obtener de la lista de llamadas actuales en la cola
     * para su procesamiento
     * @return Lista de llamadas encoladas
     */
    public static Call listCallInQueue(){
        return getInstance().callQueue.poll();
    }
}

```

CallingQueue.java

```

@Override
public void run() {
    try {
        Call callCurrent = employeeCall.getCall();
        Employee employee = employeeCall.getEmployee();
        /*
         * Actualiza llamada a LLAMADA_EN_PROGRESO
         * Calcula la duracion de la llamada
         */
        Long idLlamada = callCurrent.getId();
        LOGGER.info("----- LA LLAMADA : " + idLlamada + " FUE ASIGNADA AL EMPLEADO: " + employeeCall.getEmployee().getId() + "-");
        employee.setState(EmployeeStatus.OCUPADO);
        checkAsyncService.updateStatusEmployee(employee);

        LOGGER.info("----- ACTUALIZANDO LLAMADA : " + idLlamada + " A LLAMADA EN PROGRESO -----");
        callCurrent.setState(CallState.LLAMADA_EN_PROGRESO);
        checkAsyncService.updateStateCall(callCurrent);

        //Se simula el tiempo de la llamada en el proceso
        Thread.sleep(employeeCall.getDuration() * MILLISECONDS);
        LOGGER.info("----- REGISTRANDO LLAMADA " + idLlamada + " AL EMPLEADO -----");
        checkAsyncService.assignedCallEmployee(employeeCall);

        /*
         * Actualiza el estado de la llamada a FINALIZADA
         */
        LOGGER.info("----- ACTUALIZANDO LLAMADA : " + idLlamada + " A LLAMADA FINALIZADA -----");
        callCurrent.setState(CallState.LLAMADA_FINALIZADA);
        checkAsyncService.updateStateCall(employeeCall.getCall());

        LOGGER.info("----- LA LLAMADA : " + idLlamada + " A FINALIZADO LA ATENDIO EL EMPLEADO : " + employee.getId() + "-----");
        employee.setState(EmployeeStatus.DISPONIBLE);
        checkAsyncService.updateStatusEmployee(employee);
        applicationEventPublisher.publishEvent(new Message(this, employee));
    }
}

```

CallExecutor.java

Una vez estas terminan su primera iteración el controlador comienza a desplegar su negocio permitiendo el procesamiento de las llamadas, búsqueda de empleados disponibles, asignación de llamadas, puesta de llamada en cola.

```

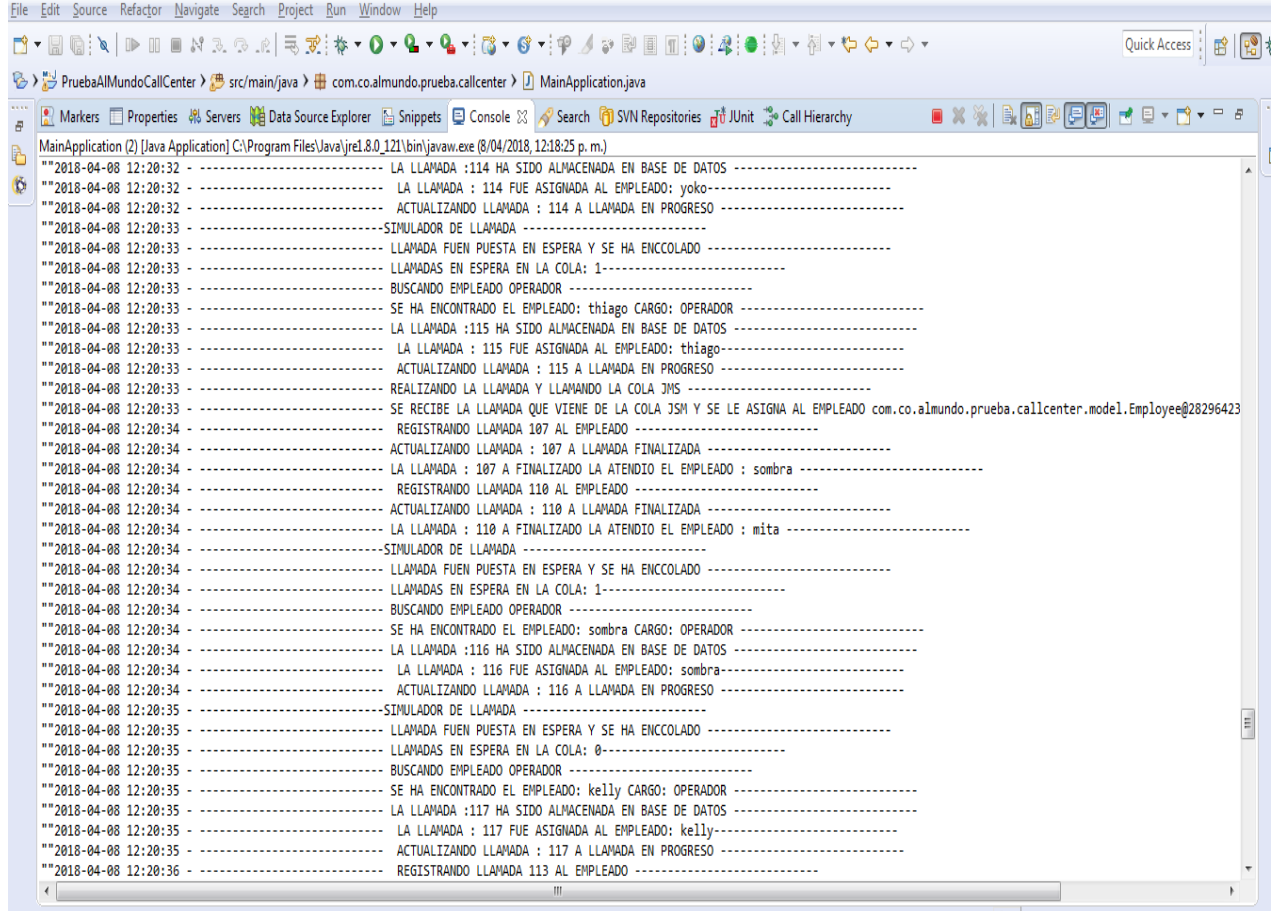
public void run() {
    Employee freeEmployee = null;
    Call call = null;
    /*se consultan los empleados disponibles*/
    empleadosDisponibles();

    while (running) {
        try {
            //Se valida si hay empleados libres
            if(!employeesAvailable.isEmpty()) {
                call = CallingQueue.listCallInQueue();

                if(call != null) {
                    try {
                        freeEmployee = searchFreePerson();
                    } catch (SQLException e) {
                        LOGGER.info("----- ERROR CONSULTANDO EL LISTADO DE EMPLEADOS -----");
                    }
                    //asigna llamada al empleado
                    dispatchCall(freeEmployee, call);
                }
            } else {
                LOGGER.info("----- LOS EMPLEADO NO ESTAN DISPONIBLES, POR FAVO ESPERE -----");
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            LOGGER.error("----- ERROR EN LA ASIGNACIÓN DE LA LLAMADA -----");
        }
    }
}

```

En la salida por consola se pueden evidenciar los procesos que se van realizando por medio de mensajería LOG, inyectada en la aplicación.



```

MainApplication (2) [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (8/04/2018, 12:18:25 p. m.)
**2018-04-08 12:20:32 - ----- LA LLAMADA :114 HA SIDO ALMACENADA EN BASE DE DATOS -----
**2018-04-08 12:20:32 - ----- LA LLAMADA : 114 FUE ASIGNADA AL EMPLEADO: yoko-----
**2018-04-08 12:20:32 - ----- ACTUALIZANDO LLAMADA : 114 A LLAMADA EN PROGRESO -----
**2018-04-08 12:20:33 - -----SIMULADOR DE LLAMADA -----
**2018-04-08 12:20:33 - ----- LLAMADA FUEN PUESTA EN ESPERA Y SE HA ENCOLADO -----
**2018-04-08 12:20:33 - ----- LLAMADAS EN ESPERA EN LA COLA: 1-----
**2018-04-08 12:20:33 - ----- BUSCANDO EMPLEADO OPERADOR -----
**2018-04-08 12:20:33 - ----- SE HA ENCONTRADO EL EMPLEADO: thiago CARGO: OPERADOR -----
**2018-04-08 12:20:33 - ----- LA LLAMADA :115 HA SIDO ALMACENADA EN BASE DE DATOS -----
**2018-04-08 12:20:33 - ----- LA LLAMADA : 115 FUE ASIGNADA AL EMPLEADO: thiago-----
**2018-04-08 12:20:33 - ----- ACTUALIZANDO LLAMADA : 115 A LLAMADA EN PROGRESO -----
**2018-04-08 12:20:33 - ----- REALIZANDO LA LLAMADA Y LLAMANDO LA COLA JMS -----
**2018-04-08 12:20:33 - ----- SE RECIBE LA LLAMADA QUE VIENE DE LA COLA JMS Y SE LE ASIGNA AL EMPLEADO com.co.almundo.prueba.callcenter.model.Employee@28296423
**2018-04-08 12:20:34 - ----- REGISTRANDO LLAMADA 107 AL EMPLEADO -----
**2018-04-08 12:20:34 - ----- ACTUALIZANDO LLAMADA : 107 A LLAMADA FINALIZADA -----
**2018-04-08 12:20:34 - ----- LA LLAMADA : 107 A FINALIZADO LA ATENDIO EL EMPLEADO : sombra -----
**2018-04-08 12:20:34 - ----- REGISTRANDO LLAMADA 110 AL EMPLEADO -----
**2018-04-08 12:20:34 - ----- ACTUALIZANDO LLAMADA : 110 A LLAMADA FINALIZADA -----
**2018-04-08 12:20:34 - ----- LA LLAMADA : 110 A FINALIZADO LA ATENDIO EL EMPLEADO : mita -----
**2018-04-08 12:20:34 - -----SIMULADOR DE LLAMADA -----
**2018-04-08 12:20:34 - ----- LLAMADA FUEN PUESTA EN ESPERA Y SE HA ENCOLADO -----
**2018-04-08 12:20:34 - ----- LLAMADAS EN ESPERA EN LA COLA: 1-----
**2018-04-08 12:20:34 - ----- BUSCANDO EMPLEADO OPERADOR -----
**2018-04-08 12:20:34 - ----- SE HA ENCONTRADO EL EMPLEADO: sombra CARGO: OPERADOR -----
**2018-04-08 12:20:34 - ----- LA LLAMADA :116 HA SIDO ALMACENADA EN BASE DE DATOS -----
**2018-04-08 12:20:34 - ----- LA LLAMADA : 116 FUE ASIGNADA AL EMPLEADO: sombra-----
**2018-04-08 12:20:34 - ----- ACTUALIZANDO LLAMADA : 116 A LLAMADA EN PROGRESO -----
**2018-04-08 12:20:35 - -----SIMULADOR DE LLAMADA -----
**2018-04-08 12:20:35 - ----- LLAMADA FUEN PUESTA EN ESPERA Y SE HA ENCOLADO -----
**2018-04-08 12:20:35 - ----- LLAMADAS EN ESPERA EN LA COLA: 0-----
**2018-04-08 12:20:35 - ----- BUSCANDO EMPLEADO OPERADOR -----
**2018-04-08 12:20:35 - ----- SE HA ENCONTRADO EL EMPLEADO: kelly CARGO: OPERADOR -----
**2018-04-08 12:20:35 - ----- LA LLAMADA :117 HA SIDO ALMACENADA EN BASE DE DATOS -----
**2018-04-08 12:20:35 - ----- LA LLAMADA : 117 FUE ASIGNADA AL EMPLEADO: keily-----
**2018-04-08 12:20:35 - ----- ACTUALIZANDO LLAMADA : 117 A LLAMADA EN PROGRESO -----
**2018-04-08 12:20:36 - ----- REGISTRANDO LLAMADA 113 AL EMPLEADO -----

```

Se muestra la creación de la llamada (Simulador llamada), el estado de la misma ya sea en espera, en progreso, finalizada, la cantidad de llamadas encoladas, los cargos de los empleados y quien atendió la llamada, la búsqueda de los empleados, los disponibles y la asignación a estos.

Cuando no hay empleados libres la llamada se encola en una lista de espera, hasta el momento que exista alguno disponible, proceso que se invoca desde la clase Dispatcher.java

Cuando las llamadas son concurrentes están se van encolando y se van asignando con respecto a la disponibilidad de empleados.

Test unitarios

Se creó un test unitario para poder tener 10 llamadas en cola dándole un tiempo de espera antes de que sea lanzado a los clientes, este test se cumple si las llamadas encoladas después de ser asignadas es igual a cero.

```
@Autowired(required=true)
private CallBean callBean;

private static final Logger LOGGER = LoggerFactory.getLogger(AllTests.class);
private ExecutorService executor = Executors.newFixedThreadPool(1);

@Test
public void llamadasConcurrentes() throws Exception {

    //Se contruyen 10 llamadas temporales y se encolan
    int call = 10;
    for (int i = 1; i <= call; i++) {
        CallingQueue.glueCall(new CallTest().byIdEmployee((long) i ).build());
    }

    //Se envía tru al runing paa que comience el proceso de llamadas
    dispatcher.setRunning(true);
    executor.execute(dispatcher);

    //Se verifica si fue capaz de desencolar las 10 llamadas
    //se pausa el hilo para que las llamadas leguen al tiempo
    Thread.sleep(10000);
    dispatcher.setRunning(false);
    Assert.assertTrue(CallingQueue.callsQueue() == CERO);
}
```

Se creó un test unitario se inserción de usuarios en la base de datos, los cuales fueron creados de forma aleatoria.

```
@Test
public void insertEmployee() throws Exception {

    ArrayList<Employee>listEmployeeInsert = new ArrayList<>();
    for(long i = 50; i< 80; i++) {
        Employee employee = new Employee();
        employee.setId(i);
        employee.setEmail("email"+i+"@gmail.com");
        employee.setCharge(TypeEmployee.OPERADOR);
        employee.setState(EmployeeStatus.DISPONIBLE);

        listEmployeeInsert.add(employee);
    }

    for(long i = 80; i< 90; i++) {
        Employee employee = new Employee();
        employee.setId(i);
        employee.setEmail("email"+i+"@gmail.com");
        employee.setCharge(TypeEmployee.SUPERVISOR);
        employee.setState(EmployeeStatus.DISPONIBLE);

        listEmployeeInsert.add(employee);
    }

    for(long i = 90; i< 100; i++) {
        Employee employee = new Employee();
        employee.setId(i);
        employee.setEmail("email"+i+"@gmail.com");
        employee.setCharge(TypeEmployee.DIRECTOR);
        employee.setState(EmployeeStatus.DISPONIBLE);

        listEmployeeInsert.add(employee);
    }

    employeeBean.saveAll(listEmployeeInsert);
    listEmployeeInsert.clear();

    employeeBean.saveAll(listEmployeeInsert);
    listEmployeeInsert.clear();

    listEmployeeInsert = (ArrayList<Employee>) employeeBean.findAll();

    if(listEmployeeInsert != null && !listEmployeeInsert.isEmpty()) {

        for(Employee employee : listEmployeeInsert) {
            LOGGER.info("----- SE CONSULTA EL USUARIO "+ employee.getName());
        }
    }
}
```

Se creó un test de inserción de llamadas a la base de datos las cuales también son creadas de forma aleatoria.

```
@Test
public void insertCall() throws Exception {
    ArrayList<Call> listCall = new ArrayList<>();
    for(long i = 9000; i<1000; i++) {
        Call call = new Call();
        call.setId(i);
        call.setState(CallState.LLAMADA_EN_COLA);
        listCall.add(call);
    }

    callBean.saveAll(listCall);
    listCall.clear();
    listCall = (ArrayList<Call>) callBean.findAll();

    if(listCall != null && !listCall.isEmpty()) {
        for(Call call : listCall) {
            LOGGER.info("----- SE CONSULTA LA LLAMADA "+ call.getId());
        }
    }
}
```