



# **PROYECTOS EXTRAORDINARIA**



**itv**

**Realizado por:**

Jorge Sánchez Berrocoso

15/06/2023

Proyecto	ITV		
Entregable	Especificación de Requisitos		
Autor	Jorge Sánchez Berrocoso		
Versión/Edición	V1.1	Fecha Versión	030/05/2023
Aprobado por		Fecha Aprobación	
		Nº Total de Páginas	X

<b>PROJECT MANAGER</b>
Jorge Sánchez Berrocoso
<b>PRODUCT OWNER</b>
Jorge Sánchez Berrocoso
<b>PROGRAMMERS</b>
Jorge Sánchez Berrocoso
<b>CLIENT</b>
José Luis Gonzalez Sánchez

## Contenido

Introducción .....	4
Descripción del problema propuesto .....	5
Diagrama .....	6
Restricciones .....	8
Requisitos .....	9
Requisitos Funcionales .....	9
Requisitos No Funcionales .....	11
Evaluación y Análisis.....	12
MODELOS .....	12
JPA Hibernate .....	14
Spring MongoDB .....	16
Spring JPA .....	17
Repositorios.....	18
Cache .....	24
Contraseña .....	25
Managers.....	26
Controllers .....	29
Test .....	31
Beneficios de utilizar MongoDB Reactivo.....	39
Beneficios de utilizar JPA Hibernate .....	40
Beneficios de utilizar Spring con Mongo Db.....	41
Beneficios de utilizar Spring con JPA .....	42
Patrones Utilizados.....	43
Notificaciones en Tiempo Real .....	44

# Introducción

Se ha creado un proyecto de 2DAM para resolver una práctica compuesta para el módulo de Acceso a Datos y Programación de Servicios y Procesos.

El profesor de ambos módulos nos ha dado los requisitos mínimos que tiene que cumplir dicha práctica y se tratan de puntos importantes a evaluar. Algunos de estos requisitos mínimos son los siguientes:

Implementación mediante SQL y SpringData, realizando el diagrama de clases y explicando el paso a tablas de la solución.

Implementación mediante NoSQL usando MongoDB y SpringData MongoDB, realizando el diagrama de clases y explicando la obtención de colecciones existentes.

Caché de entidades relevantes.

Enfoque Railway Oriented Programming.

Test de todos los elementos usados.

El tiempo estimado para realizar dicha práctica será de 30 días y se entregará y defenderá en clase el día 18 de junio de 2023.

# Descripción del problema propuesto

El problema se refiere a una central de Inspección Técnica de Vehículos (ITV) en Pepilandia. La central cuenta con un conjunto de trabajadores, de los cuales se necesita registrar la siguiente información: nombre, teléfono, email (único), nombre de usuario (único), contraseña (cifrada) y fecha de contratación. Cada trabajador tiene una especialidad, que puede ser ELECTRICIDAD, MOTOR, MECANICA o INTERIOR.

El salario de cada trabajador depende de su especialidad. Para ELECTRICIDAD el salario base es de 1800€, para MOTOR es de 1700€, para MECANICA es de 1600€ y para INTERIOR es de 1750€. Además, se agrega un bono de 100€ por cada tres años de antigüedad en la empresa.

Además de los trabajadores regulares, hay un responsable de la ITV que también es un trabajador, pero tiene un plus de dirección de +1000€.

El taller de la ITV gestiona las citas en intervalos de 30 minutos. Cada cita será atendida por un trabajador disponible en ese momento. Un trabajador no puede atender más de 4 citas en un intervalo y no puede haber más de 8 citas en el mismo intervalo.

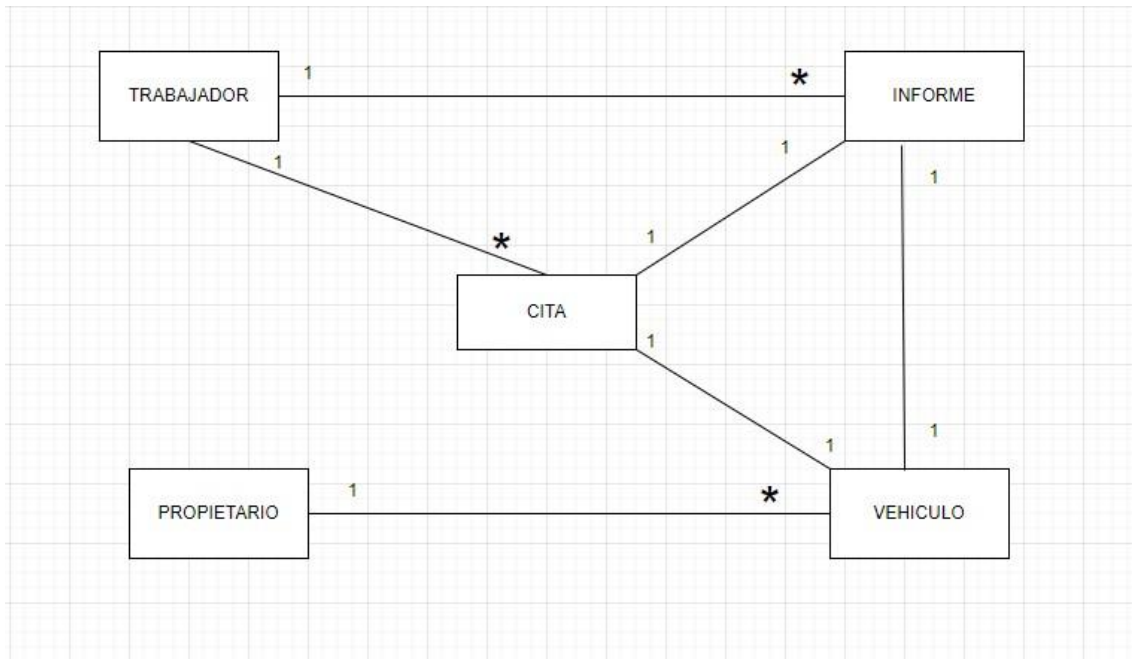
Para atender una cita, se requieren los datos del vehículo, como la marca, modelo, matrícula, fecha de matriculación y fecha de última revisión. Además, se necesitan los datos del propietario/a del vehículo, como el DNI, nombre, apellidos y teléfono.

La ITV emite un informe favorable o desfavorable después de la inspección. El informe incluye datos sobre el frenado (valor entre 1 y 10 con dos decimales), contaminación (valor entre 20 y 50 con dos decimales), si el frenado es apto o no y si las luces son aptas o no. El informe debe reflejar los datos del trabajador que realizó la inspección, del vehículo y del propietario/a.

Además de las operaciones básicas de registro y gestión de citas e informes, se requieren las siguientes funcionalidades adicionales:

- Encontrar al trabajador que gana el salario más alto sin ser el responsable.
- Calcular el salario medio de todos los trabajadores que no son responsables.
- Calcular el salario medio de todos los trabajadores agrupados por especialidad.
- Encontrar al trabajador con menos antigüedad.
- Listar los trabajadores ordenados por especialidad y luego por antigüedad.

# Diagrama



En la central de Inspección Técnica de Vehículos (ITV) en Pepilandia, existen diversas relaciones entre las entidades involucradas, las cuales son fundamentales para el funcionamiento y la gestión de la ITV. Estas relaciones se establecen a través de asociaciones entre las entidades y permiten el almacenamiento y la recuperación de la información de manera estructurada. A continuación, se describen las relaciones principales:

1. **Trabajador - Informe:** Existe una relación uno a muchos entre la entidad "Trabajador" y la entidad "Informe". Esto significa que un trabajador puede tener varios informes asociados, pero un informe solo puede estar relacionado con un trabajador en particular. Cada informe registrado en la ITV estará vinculado al trabajador que realizó la inspección correspondiente.
2. **Trabajador - Cita:** Existe una relación uno a muchos entre la entidad "Trabajador" y la entidad "Cita". Esto implica que un trabajador puede atender varias citas, pero una cita solo puede ser asignada a un trabajador específico. Cada cita registrada en la ITV estará asociada a un trabajador responsable de realizar la inspección.
3. **Cita - Vehículo:** Hay una relación uno a uno entre la entidad "Cita" y la entidad "Vehículo". Esto significa que cada cita programada en la ITV estará vinculada a un único vehículo, y a su vez, cada vehículo podrá tener asignada una sola cita. Esta relación permite el registro y la gestión de la información del vehículo durante la inspección.

4. **Informe - Vehículo:** Existe una relación uno a uno entre la entidad "Informe" y la entidad "Vehículo". Esto implica que cada informe generado después de la inspección estará asociado a un único vehículo, y cada vehículo tendrá un único informe correspondiente. Esta relación permite mantener un registro detallado de los resultados de la inspección para cada vehículo inspeccionado.
5. **Propietario - Vehículo:** Hay una relación uno a muchos entre la entidad "Propietario" y la entidad "Vehículo". Esto significa que un propietario puede tener varios vehículos registrados, pero cada vehículo estará asociado a un único propietario. Esta relación permite la gestión de la información del propietario del vehículo durante la inspección.

# Restricciones

Aquí están las restricciones del problema:

1. Cada trabajador debe tener un email único. No puede haber dos trabajadores con el mismo email.
2. Cada trabajador debe tener un nombre de usuario único. No puede haber dos trabajadores con el mismo nombre de usuario.
3. La contraseña de cada trabajador debe estar cifrada.
4. El salario de cada trabajador depende de su especialidad. Los valores son: ELECTRICIDAD (1800), MOTOR (1700), MECANICA (1600) e INTERIOR (1750). Además, se agrega un bono de 100€ por cada tres años de antigüedad.
5. El responsable de ITV, que es un trabajador, recibe un plus de dirección de +1000€.
6. En el taller, cada cita debe ser atendida por un trabajador disponible en ese momento.
7. Un trabajador no puede atender más de 4 citas por intervalo.
8. No se pueden tener más de 8 citas en el mismo intervalo.
9. Para cada cita, se requieren los datos del vehículo (marca, modelo, matrícula, fecha de matriculación y fecha de última revisión) y los datos del propietario/a (DNI, nombre, apellidos y teléfono).
10. Cada informe de inspección debe incluir los datos del trabajador que lo ha realizado, del vehículo y del propietario/a.



# Requisitos

## Requisitos Funcionales

Los requisitos funcionales identificados son los siguientes:

1. Registro de trabajadores:
  - Capturar nombre, teléfono, email único, nombre de usuario único, contraseña cifrada y fecha de contratación de cada trabajador.
  - Asignar una especialidad (ELECTRICIDAD, MOTOR, MECANICA o INTERIOR) a cada trabajador.
2. Cálculo del salario de los trabajadores:
  - Calcular el salario de cada trabajador basado en su especialidad y antigüedad (100€ adicionales por cada tres años trabajados).
  - Añadir un plus de dirección de +1000€ al responsable de la ITV.
3. Gestión de citas:
  - Crear citas en intervalos de 30 minutos.
  - Asignar a un trabajador disponible para atender cada cita.
  - Limitar a un máximo de 4 citas por intervalo y 8 citas en total en el mismo intervalo.
4. Registro de datos del vehículo y propietario:
  - Registrar información del vehículo, incluyendo marca, modelo, matrícula, fecha de matriculación y fecha de última revisión.
  - Registrar información del propietario/a del vehículo, incluyendo DNI, nombre, apellidos y teléfono.
5. Generación de informes de inspección:
  - Emitir un informe favorable o desfavorable después de cada inspección.
  - Incluir en el informe los datos del trabajador que realizó la inspección, del vehículo y del propietario/a.
  - Registrar datos de frenado (valor entre 1 y 10 con dos decimales), contaminación (valor entre 20 y 50 con dos decimales), aptitud del frenado y aptitud de las luces.
6. Funcionalidades adicionales:

- Identificar al trabajador que gana el salario más alto, excluyendo al responsable de la ITV.
- Calcular el salario medio de todos los trabajadores que no son responsables.
- Calcular el salario medio de los trabajadores agrupados por especialidad.
- Encontrar al trabajador/a con menos antigüedad.
- Listar los trabajadores ordenados por especialidad y luego por antigüedad.

## Requisitos No Funcionales

Los requisitos no funcionales relevantes para este problema son:

1. Seguridad: El sistema debe garantizar la seguridad de los datos sensibles, como contraseñas y información personal. Debe implementar medidas de seguridad, como encriptación de contraseñas y acceso restringido a los datos.
2. Escalabilidad: El sistema debe ser capaz de manejar un volumen creciente de trabajadores, citas e informes sin degradar el rendimiento. Debe estar diseñado para ser escalable y poder soportar un aumento en la carga de trabajo.
3. Rendimiento: El sistema debe ser eficiente y ofrecer una respuesta rápida a las solicitudes de los usuarios. Las operaciones de registro, consulta y cálculos deben realizarse en tiempos aceptables, incluso con grandes cantidades de datos.
4. Usabilidad: El sistema debe ser intuitivo y fácil de usar para los usuarios. La interfaz de usuario debe ser clara y permitir una interacción fluida. Se deben proporcionar mensajes de error claros y guías de ayuda cuando sea necesario.
5. Disponibilidad: El sistema debe estar disponible y accesible para los usuarios en todo momento, evitando tiempos de inactividad prolongados. Se deben implementar medidas de respaldo y recuperación de datos para minimizar la pérdida de información en caso de fallas.
6. Mantenibilidad: El sistema debe ser fácil de mantener y actualizar. Debe seguir prácticas de desarrollo limpio y modular para facilitar la incorporación de nuevas funcionalidades y corrección de errores.
7. Integración: El sistema debe ser capaz de integrarse con otros sistemas o servicios externos, como bases de datos de vehículos o servicios de envío de correos electrónicos, si es necesario.

# Evaluación y Análisis

## MODELOS

Identificamos las siguientes entidades:

1. Trabajador:

- Atributos: nombre, teléfono, email, nombre de usuario, contraseña, fecha de contratación, especialidad.
- Descripción: Representa a un trabajador de la central de ITV. Cada trabajador tiene un conjunto de atributos que incluyen información personal y de contacto, así como detalles relacionados con su empleo, como la fecha de contratación y la especialidad en la que se desempeña.

2. Especialidad:

- Atributos: nombre.
- Descripción: Representa una especialidad en la que un trabajador puede tener experiencia. Las especialidades mencionadas en el problema son ELECTRICIDAD, MOTOR, MECANICA e INTERIOR.

3. Vehículo:

- Atributos: marca, modelo, matrícula, fecha de matriculación, fecha de última revisión.
- Descripción: Representa un vehículo sometido a inspección en la central de ITV. Se registran datos relevantes del vehículo, como la marca, modelo, matrícula, así como las fechas de matriculación y última revisión.

4. Propietario:

- Atributos: DNI, nombre, apellidos, teléfono.
- Descripción: Representa al propietario del vehículo sometido a inspección. Se almacena información personal y de contacto del propietario, como su DNI, nombre, apellidos y número de teléfono.

5. Cita:

- Atributos: trabajador asignado, fecha y hora, datos del vehículo y propietario.
- Descripción: Representa una cita programada en la central de ITV. Cada cita está asociada a un trabajador específico, tiene una

fecha y hora asignadas, y contiene los datos del vehículo y del propietario.

#### 6. Informe:

- Atributos: trabajador que realizó la inspección, datos de frenado, contaminación, aptitud de frenado y luces.
- Descripción: Representa el informe generado después de realizar la inspección de un vehículo. Contiene los resultados de la inspección, incluyendo los datos de frenado, contaminación, aptitud de frenado y luces, así como la referencia al trabajador que realizó la inspección.

## Mongo DB

En MongoDB, los modelos se representan utilizando documentos BSON (Binary JSON) que se almacenan en colecciones. Cada modelo se mapea a un documento BSON en la colección correspondiente.

Anotaciones utilizadas:

1. `@Document`: Esta anotación se utiliza para marcar una clase como un documento que se almacenará en una colección de MongoDB. Se utiliza para especificar el nombre de la colección y otras opciones de configuración.
2. `@BsonId`: Esta anotación se utiliza para marcar una propiedad como el identificador único de una colección.

Ejemplo:

```
@Serializable
data class Trabajador(
    @BsonId @Contextual
    val _id : String = newId<Trabajador>().toString(),
    val nombre : String,
    val telefono : Int,
    val email : String,
    val username : String,
    val contraseña : ByteArray,
    @Serializable(LocalDateSerializer::class)
    val fechaContratacion : LocalDate,
    val especialidad : String,
    val salario : Int,
    val responsable : Boolean
) {
    enum class Especialidad(){
        ELECTRICIDAD, MOTOR, MECANICA, INTERIOR
    }
}
```

## JPA Hibernate

Anotaciones que podemos ver en los modelos de JPA Hibernate:

- La anotación **@Entity** indica que la clase que es una entidad que se puede almacenar en la base de datos.
- La anotación **@Table(name = "")** especifica el nombre de la tabla en la base de datos donde se almacenarán los datos de la entidad.
- La anotación **@NamedQueries** define consultas con nombre que se pueden utilizar para realizar operaciones específicas en la entidad. En este caso, se define una consulta con nombre por ejemplo "Cita.findAll" que selecciona todas las instancias de la entidad **Cita**.
- La anotación **@Serializable** indica que la clase es serializable, lo que permite su representación en formatos de datos como JSON o XML.
- La anotación **@Id** marca el campo **uuid** como la clave primaria de la entidad **Cita**.
- La anotación **@GenericGenerator** especifica el generador de valores para la clave primaria. En este caso, se utiliza el generador **UUIDGenerator** de Hibernate para generar identificadores UUID únicos.
- La anotación **@Column(name = "uuid")** especifica el nombre de la columna en la tabla de la base de datos donde se almacenará el valor de **uuid**.
- La anotación **@Type(type = "uuid-char")** especifica el tipo de columna de la base de datos para el campo **uuid** como una cadena de caracteres.
- La anotación **@Serializable(with = UUIDSerializer::class)** indica que se utilizará un serializador personalizado llamado **UUIDSerializer** para serializar y deserializar el campo **uuid**.
- Las variables **fechaHora**, **idTrabajador**, **idVehiculo** e **idPropietario** representan los atributos de la entidad **Cita**. La anotación **@OneToOne** indica que estas variables están asociadas con una relación uno a uno con otras entidades (**Trabajador**, **Vehiculo** y **Propietario** respectivamente).
- La anotación **@JoinColumn** especifica el nombre de la columna en la tabla de la base de datos que se utilizará como clave foránea para la relación uno a uno.
- Las clases **Trabajador**, **Vehiculo** y **Propietario** deben ser entidades JPA definidas de manera similar a **Cita** para establecer las relaciones adecuadas.

Ejemplo:

```
12 @Entity
13 @Table(name = "Cita")
14 @NamedQueries(
15     NamedQuery(name = "Cita.findAll", query = "select c from Cita c"),
16 )
17 @Serializable
18 data class Cita(
19     @Id
20     @GenericGenerator(
21         name = "UUID",
22         strategy = "org.hibernate.id.UUIDGenerator",
23     )
24     @Column(name = "uuid")
25     @Type(type = "uuid-char")
26     @Serializable(with = UUIDSerializer::class)
27     var uuid: UUID = UUID.randomUUID(),
28     @Serializable(with = LocalDateTimeSerializer::class)
29     var fechaHora: LocalDateTime,
30     @OneToOne
31     @JoinColumn(name = "trabajador_id", nullable = false)
32     var idTrabajador: Trabajador,
33     @OneToOne
34     @JoinColumn(name = "vehiculo_id", nullable = false)
35     var idVehiculo: Vehiculo,
36     @OneToOne
37     @JoinColumn(name = "propietario_id", nullable = false)
38     var idPropietario: Propietario
39 ) {
40 }
41 }
```

# Spring MongoDB

## Anotaciones :

1. **@Document**: Esta anotación se utiliza para marcar una clase como un documento en MongoDB. Permite especificar el nombre de la colección donde se almacenarán los documentos.
2. **@Field**: Esta anotación se utiliza para mapear una propiedad de la clase a un campo en el documento de MongoDB. Puedes especificar el nombre del campo y otras opciones, como requerido, único, etc.
3. **@Id**: Esta anotación se utiliza para marcar una propiedad como el identificador único del documento.
4. **@DBRef**: Esta anotación se utiliza para establecer una referencia a documentos en otras colecciones. Se utiliza para crear relaciones entre documentos en lugar de incrustarlos directamente. Spring Data MongoDB manejará automáticamente las referencias y permitirá cargar y guardar los objetos asociados.
5. **@DocumentReference**: Esta anotación es una alternativa a **@DBRef** que se utiliza para establecer una referencia a documentos en otras colecciones. Al igual que **@DBRef**, permite crear relaciones entre documentos. La diferencia radica en cómo se almacena la referencia. **@DocumentReference** almacena la referencia como un identificador simple en lugar de utilizar el concepto de referencia de MongoDB. Esto puede ser útil en ciertos casos de uso donde se requiere una referencia más sencilla.

## Ejemplo:

```
9  @Serializable
10  @Document(collection = "informe")
11  data class Informe(
12      @Id
13      val _id : String = ObjectId.get().toString(),
14      val frenado : Int,
15      val contaminación : Double,
16      val aptoFrenado: Boolean,
17      val luces : Boolean,
18      val apto : Boolean,
19      @DocumentReference
20      val idTrabajador: Trabajador,
21      @DocumentReference
22      val idVehiculo: Vehiculo,
23      @DocumentReference
24      val idPropietario: Propietario,
25  ) {
26
27  }
```



## Spring JPA

1. **@Entity**: Esta anotación se utiliza para marcar una clase como una entidad en el modelo de datos. Una entidad representa una tabla en una base de datos relacional. Cada instancia de la clase **@Entity** se guarda como una fila en la tabla correspondiente.
2. **@Table**: Esta anotación se utiliza para especificar el nombre de la tabla en la base de datos donde se almacenarán las instancias de la entidad. Puedes personalizar el nombre de la tabla y otros atributos, como el esquema y las restricciones.
3. **@Column**: Esta anotación se utiliza para mapear una propiedad de la entidad a una columna en la tabla. Puedes especificar el nombre de la columna, su tipo de datos, restricciones y otras propiedades.
4. **@Id**: Esta anotación se utiliza para marcar una propiedad como el identificador único de la entidad. La propiedad anotada con **@Id** se mapea a la clave primaria de la tabla.
5. **@GeneratedValue**: Esta anotación se utiliza junto con **@Id** para especificar cómo se generará automáticamente el valor del identificador.
6. **@ManyToOne** / **@OneToMany**: Estas anotaciones se utilizan para establecer relaciones entre entidades. **@ManyToOne** se utiliza en la propiedad de la entidad que representa el lado "muchos" de la relación, mientras que **@OneToMany** se utiliza en la propiedad que representa el lado "uno" de la relación. Estas anotaciones permiten establecer relaciones de uno a muchos entre entidades.
7. **@JoinColumn**: Esta anotación se utiliza para personalizar el nombre de la columna utilizada para la relación entre dos entidades. Puedes especificar el nombre de la columna y otras opciones relacionadas con la clave externa.

### Ejemplo:

```
11 @Entity
12 @Table(name = "cita")
13 @Serializable
14 data class Cita(
15     @Id
16     var uuid: UUID,
17     @Serializable(with = LocalDateTimeSerializer::class)
18     var fechaHora : LocalDateTime,
19     @ManyToOne
20     @JoinColumn(name = "trabajador_id", nullable = false)
21     var trabajador : Trabajador,
22     @ManyToOne
23     @JoinColumn(name = "vehiculo_id", nullable = false)
24     var vehiculo : Vehiculo,
25     @ManyToOne
26     @JoinColumn(name = "propietario_id", nullable = false)
27     var propietario : Propietario
28 )
29 {
30 }
```

## Repositorios

El repositorio es responsable de realizar operaciones de persistencia y recuperación de datos relacionados con las citas en el sistema. A continuación, se realiza una descripción de las operaciones implementadas:

1. `findAll()`: Recupera todas las citas almacenadas en la base de datos. Retorna un objeto `Result` que encapsula un flujo (`Flow`) de citas o una excepción en caso de error.
2. `findById(id: String)`: Recupera una cita específica por su ID. Retorna un objeto `Result` que encapsula la cita encontrada o una excepción en caso de error.
3. `save(entity: Cita)`: Guarda una nueva cita en la base de datos. Retorna un objeto `Result` que indica el éxito de la operación o una excepción en caso de error.
4. `update(entity: Cita)`: Actualiza una cita existente en la base de datos. Retorna un objeto `Result` que indica el éxito de la operación o una excepción en caso de error.
5. `delete(_id: String)`: Elimina una cita de la base de datos por su ID. Retorna un objeto `Result` que indica el éxito de la operación o una excepción en caso de error.
6. `findByTrabajadorAndIntervalo(trabajador: String, fechaHora: LocalDateTime)`: Recupera todas las citas asignadas a un trabajador dentro de un intervalo de tiempo específico. Retorna un objeto `Result` que encapsula una lista de citas encontradas o una excepción en caso de error.
7. `findByIntervalo(fechaHora: LocalDateTime)`: Recupera todas las citas dentro de un intervalo de tiempo específico. Retorna un objeto `Result` que encapsula una lista de citas encontradas o una excepción en caso de error.

Cada método realiza consultas a la base de datos utilizando el controlador adecuado y retorna los resultados encapsulados en objetos `Result`, que indican si la operación se realizó correctamente o si se produjo una excepción.

# Mongo Db

## Ejemplo con el Repositorio de Cita:

```
11 class CitaRepository : ICitaRepository {
12     override suspend fun findAll(): Result<Flow<Cita>> {
13         return try {
14             val citas = MongoDBManager.database.getCollection<Cita>().find().publisher.asFlow()
15             Result.success(citas)
16         } catch (e: Exception) {
17             Result.failure(CitaException("Error al obtener todas las citas"))
18         }
19     }
20
21     override suspend fun findById(id: String): Result<Cita?> {
22         return try {
23             val cita = MongoDBManager.database.getCollection<Cita>().findOneById(id)
24             Result.success(cita)
25         } catch (e: Exception) {
26             Result.failure(CitaException("Error al obtener la cita con ID: $id"))
27         }
28     }
29
30     override suspend fun save(entity: Cita): Result<Unit> {
31         return try {
32             MongoDBManager.database.getCollection<Cita>().save(entity)
33             Result.success(Unit)
34         } catch (e: Exception) {
35             Result.failure(CitaException("No se ha podido insertar la cita $entity"))
36         }
37     }
38 }
```

```
39     override suspend fun update(entity: Cita): Result<Unit> {
40         return try {
41             MongoDBManager.database.getCollection<Cita>().updateOneById(entity._id, entity)
42             Result.success(Unit)
43         } catch (e: Exception) {
44             Result.failure(CitaException("No se ha podido actualizar la cita $entity"))
45         }
46     }
47
48     override suspend fun delete(_id: String): Result<Unit> {
49         return try {
50             MongoDBManager.database.getCollection<Cita>().deleteOneById(_id)
51             Result.success(Unit)
52         } catch (e: Exception) {
53             Result.failure(CitaException("No se ha podido borrar la cita con el _id $_id"))
54         }
55     }
56 }
```

```

57 override suspend fun findByTrabajadorAndIntervalo(trabajador: String, fechaHora: LocalDateTime): Result<List<Cita>> {
58     return try {
59         val query = and(
60             eq( fieldName: "idTrabajador", trabajador),
61             gte( fieldName: "fechaHora", fechaHora),
62             lt( fieldName: "fechaHora", fechaHora.plusMinutes( minutes: 30))
63         )
64
65         val citas = MongoDBManager.database.getCollection<Cita>().find(query).toList()
66         Result.success(citas)
67     } catch (e: Exception) {
68         Result.failure(CitaException("Error al buscar las citas del trabajador $trabajador en el intervalo $fechaHora"))
69     }
70 }
71
72 override suspend fun findByIntervalo(fechaHora: LocalDateTime): Result<List<Cita>> {
73     return try {
74         val intervaloInicio = fechaHora
75         val intervaloFin = fechaHora.plusMinutes( minutes: 30)
76
77         val query = and(
78             gte( fieldName: "fechaHora", intervaloInicio),
79             lt( fieldName: "fechaHora", intervaloFin)
80         )
81
82         val citas = MongoDBManager.database.getCollection<Cita>().find(query).toList()
83         Result.success(citas)
84     } catch (e: Exception) {
85         Result.failure(CitaException("Error al buscar las citas en el intervalo $fechaHora"))
86     }
87 }

```

## JPA Hibernate

### Ejemplo:

```
16 class CitaRepository : ICitaRepository {
17     private val logger = KotlinLogging.logger {}
18
19     override suspend fun findAll(): Result<Flow<Cita>> {
20         logger.info { "Buscando todas las citas" }
21         return try {
22             var lista = mutableListOf<Cita>()
23             HibernateManager.query {
24                 val query: TypedQuery<Cita> = HibernateManager.manager.createNamedQuery(
25                     name: "Cita.findAll",
26                     Cita::class.java
27                 )
28                 lista = query.resultList
29             }
30             Result.success(lista.asFlow())
31         } catch (e: Exception) {
32             Result.failure(CitaException("No se ha podido obtener la lista de citas"))
33         }
34     }
35 }
```

```
36 override suspend fun findById(id: UUID): Result<Cita?> {
37     logger.info { "Buscando la cita con id $id" }
38     return try {
39         var encontrado: Cita? = null
40         HibernateManager.query {
41             encontrado = HibernateManager.manager.find(Cita::class.java, id)
42         }
43         Result.success(encontrado)
44     } catch (e: Exception) {
45         Result.failure(CitaException("No se ha podido encontrar la cita con el ID $id"))
46     }
47 }
48
49 override suspend fun save(entity: Cita): Result<Unit> {
50     logger.info { "Insertando la cita $entity" }
51     return try {
52         HibernateManager.transaction {
53             val cita = HibernateManager.manager.merge(entity)
54         }
55         Result.success(Unit)
56     } catch (e: Exception) {
57         Result.failure(CitaException("No se ha podido insertar la cita $entity"))
58     }
59 }
```

```

61 override suspend fun update(entity: Cita): Result<Unit> {
62     logger.info { "Actualizar la cita $entity" }
63     return try {
64         HibernateManager.transaction {
65             val existing = HibernateManager.manager.find(Cita::class.java, entity.uuid)
66             if (existing != null) {
67                 existing.fechaHora = entity.fechaHora
68                 existing.idTrabajador = entity.idTrabajador
69                 existing.idPropietario = entity.idPropietario
70                 existing.idVehiculo = entity.idVehiculo
71
72                 HibernateManager.manager.merge(existing)
73                 logger.info { "Cita actualizado correctamente" }
74                 Result.success(Unit)
75             } else {
76                 logger.info { "Cita no encontrado" }
77                 Result.failure(CitaException("No se encontró ninguna cita con el UUID: ${entity.uuid}."))
78             }
79         }
80         Result.success(Unit)
81     } catch (e: Exception) {
82         Result.failure(CitaException("No se ha podido actualizar la cita con el UUID: ${entity.uuid}"))
83     }
84 }
85

```

```

87 override suspend fun delete(_id: UUID): Result<Unit> {
88     logger.info { "Eliminando informe" }
89     return try {
90         HibernateManager.transaction {
91             val informe = HibernateManager.manager.find(Informe::class.java, _id)
92             if (informe != null) {
93                 HibernateManager.manager.remove(informe)
94                 logger.info { "Informe eliminado correctamente" }
95                 Result.success(Unit)
96             } else {
97                 logger.info { "Informe no encontrado" }
98                 Result.failure(InformeException("No se encontró ningún informe con el UUID: $_id."))
99             }
100         }
101         Result.success(Unit)
102     } catch (e: Exception) {
103         Result.failure(InformeException("No se ha podido eliminar el informe con el UUID: $_id"))
104     }
105 }
106
107 suspend fun deleteAll(): Result<Unit> {
108     logger.info { "Eliminando todos los informes" }
109     return try {
110         HibernateManager.transaction {
111             val query = HibernateManager.manager.createQuery( qString: "delete from Informe")
112             query.executeUpdate()
113         }
114         Result.success(Unit)
115     } catch (e: Exception) {
116         Result.failure(InformeException("No se ha podido borrar los informes"))
117     }
118 }

```

## Spring MongoDB

Ejemplo :

```
@Repository
interface TrabajadorRepository : CoroutineCrudRepository<Trabajador, String> {
    suspend fun findTrabajadorByEmail(email:String):Trabajador?
    suspend fun findTrabajadorByUsername(username:String):Trabajador?
}
```

## Spring JPA Hibernate:

Ejemplo:

```
@Repository
interface CitaRepository : JpaRepository<Cita, UUID> {
    suspend fun findCitasByTrabajadorAndFechaHoraBetween(trabajador: Trabajador, fechaInicio: LocalDateTime, fechaFin: LocalDateTime): List<Cita>
    suspend fun findCitasByFechaHoraBetween(fechaInicio: LocalDateTime, fechaFin: LocalDateTime): List<Cita>
}
```

## Cache

He utilizado la **cache 4k** porque mi objetivo en este proyecto es reducir los tiempos de acceso a datos y mejorar el rendimiento del sistema, por eso he cacheado las entidades más relevantes como Vehículo, Cita y Informe.

### Ejemplo:


```
8
9     private const val STOP = 6 * 10000L
10
11     class CitaCache {
12
13         val cache = Cache.Builder()
14             .expireAfterWrite(1.minutes)
15             .expireAfterAccess(1.minutes)
16             .build<String, Cita>()
17
18         suspend fun refresh() {
19             withContext(newSingleThreadContext( name: "cache")) { this: CoroutineScope
20                 launch { this: CoroutineScope
21                     println("ACTUALIZANDO CACHE")
22                     cache.invalidateAll()
23                     delay(STOP)
24                 }
25             }
26         }
27     }
28 }
```



## Contraseña

He utilizado Bcrypt para almacenar las contraseñas de forma segura, porque a parte de ser fácil de usar es recomendable usarlo debido a su resistencia a los ataques de fuerza bruta.

### Ejemplo:

```
4      
5      object Contraseña {
6
7          fun encriptarContraseña(contraseña : String) : ByteArray {
8              return Bcrypt.hash(contraseña, saltRounds: 12)
9          }
10     }
```

# Managers

## Mongo Db Manager

MongoDBManager es una clase de gestión de acceso a la base de datos MongoDB. Proporciona métodos y funcionalidades para interactuar con la base de datos MongoDB desde una aplicación. Se encarga de administrar la conexión a una base de datos MongoDB utilizando la biblioteca `kotlinx.coroutines` y la biblioteca `KMongo`.

Ventajas:

- Permite realizar operaciones CRUD en la base de datos MongoDB.
- Proporciona métodos para consultar, insertar, actualizar y eliminar documentos.
- Ofrece soporte para consultas avanzadas y agregaciones.
- Automatiza gran parte del proceso de mapeo objeto-documento.

```
8 object MongoDBManager {
9
10     private lateinit var mongoClient: MongoClient
11     internal lateinit var database: CoroutineDatabase
12
13     private const val MONGO_TYPE = "mongodb://"
14     private const val HOST = "localhost"
15     private const val PORT = 27017
16     private const val DATABASE = "itv"
17     private const val USERNAME = "mongoadmin"
18     private const val PASSWORD = "mongopass"
19     private const val OPTIONS = "?authSource=admin"
20
21
22     private const val MONGO_URI =
23         "$MONGO_TYPE$HOST/$DATABASE"
24
25     //println("mongodb://localhost/tenistas?authSource=admin")
26
27
28
29
30     init {
31         println("MongoDB -> $MONGO_URI$OPTIONS")
32         mongoClient =
33             KMongo.createClient( connectionString: "$MONGO_URI$OPTIONS")
34                 .coroutine
35         database = mongoClient.getDatabase( name: "itv")
36     }
37 }
```

## Hibernate Manager

El Hibernate Manager es una clase de gestión de acceso a bases de datos relacionales utilizando el framework Hibernate. Hibernate es una implementación de JPA (Java Persistence API) y proporciona un conjunto de herramientas para interactuar con bases de datos relacionales utilizando el mapeo objeto-relacional.

Ventajas:

- Simplifica el acceso y la manipulación de datos en bases de datos relacionales.
- Proporciona una capa de abstracción para trabajar con bases de datos a través de objetos y consultas en lugar de SQL directamente.
- Permite realizar operaciones CRUD en la base de datos, como inserción, actualización, eliminación y consulta de registros.
- Ofrece un mecanismo de mapeo objeto-relacional flexible y potente.
- Proporciona transparencia de persistencia, lo que significa que los objetos se guardan automáticamente en la base de datos cuando se realizan cambios en ellos.

Ejemplo:

```
6
7 object HibernateManager : Closeable {
8     // Creamos la EntityManagerFactory para manejar las entidades y transacciones
9     private var entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "default")
10    lateinit var manager: EntityManager
11    private lateinit var transaction: EntityTransaction
12    private var session: Session? = null
13
14
15    init {
16        // Creamos la EntityManagerFactory
17        // Creamos la EntityManager
18        manager = entityManagerFactory.createEntityManager()
19        // Creamos la transacción
20        transaction = manager.transaction
21    }
22
23    fun open() {
24        logger.debug { "Iniciando EntityManagerFactory" }
25        manager = entityManagerFactory.createEntityManager()
26        transaction = manager.transaction
27        session = manager.unwrap(Session::class.java)
28    }
29
30    override fun close() {
31        logger.debug { "Cerrando EntityManager" }
32        session?.close()
33        manager.close()
34    }
35
36 }
```

```

46 fun query(operations: () -> Unit) {
47     open()
48     try {
49         operations()
50     } catch (e: SQLException) {
51         logger.error { "Error en la consulta: ${e.message}" }
52     } finally {
53         close()
54     }
55 }
56
57 /**
58  * Realiza unas operaciones en una transacción
59  * @param operations operaciones a realizar
60  * @throws SQLException si no se ha podido realizar la transacción
61  */
62 fun transaction(operations: () -> Unit) {
63     open()
64     try {
65         logger.debug { "Transaction iniciada" }
66         transaction.begin()
67         // Aquí las operaciones
68         operations()
69         transaction.commit()
70         logger.debug { "Transaction finalizada" }
71     } catch (e: SQLException) {
72         transaction.rollback()
73         logger.error { "Error en la transacción: ${e.message}" }
74         throw SQLException(e)
75     } finally {
76         close()

```

## Controllers

### Mongo Db, JPA Hibernate, Spring JPA, Spring MongoDB

En los cuatro proyectos, independientemente de la base de datos utilizada, se sigue un principio común de diseño: el principio de separación de responsabilidades. En este enfoque, el controlador es una capa de la arquitectura que se encarga de recibir las solicitudes del cliente, procesarlas y enviar las respuestas adecuadas.

Independientemente de si el proyecto utiliza MongoDB, una base de datos relacional a través de JPA o Hibernate, el controlador se mantiene igual. Esto se debe a que el controlador no está directamente relacionado con la implementación de la base de datos.

### Ejemplo:

#### Cita Controller

En el controlador de citas se verifica el límite de 4 citas por intervalo para el trabajador y también verificamos el límite de 8 citas en el mismo intervalo.

```
14 class CitaController(  
15     private val citaRepository: ICitaRepository,  
16     private val cache : CitaRepositoryCached  
17 ) {  
18  
19  
20     suspend fun findAllCita() : Flow<Cita>? {  
21         return citaRepository.findAll().getOrNull()?.flowOn(Dispatchers.IO)  
22     }  
23  
24
```

```
25     suspend fun saveCita(entity: Cita) {  
26         withContext(Dispatchers.IO) { this: CoroutineScope  
27             val trabajador = entity.idTrabajador // Obtén el trabajador asignado a la cita  
28  
29             // Verificar el límite de 4 citas por intervalo para el trabajador  
30             val citasIntervaloTrabajador = citaRepository.findByTrabajadorAndIntervalo(trabajador, entity.fechaHora)  
31  
32             if (citasIntervaloTrabajador.getOrDefault(listOf()).size >= 4) {  
33                 throw CitaControllerException("El trabajador no tiene hueco disponible en este intervalo de 30 minutos")  
34             }  
35  
36             // Verificar el límite de 8 citas en el mismo intervalo  
37             val citasIntervalo = citaRepository.findByIntervalo(entity.fechaHora)  
38             if (citasIntervalo.getOrDefault(listOf()).size >= 8) {  
39                 throw CitaControllerException("No hay disponibilidad de citas en este intervalo de 30 minutos")  
40             }  
41  
42             // Guardar la cita en la base de datos y en el caché  
43             launch { this: CoroutineScope  
44                 citaRepository.save(entity)  
45             }  
46             launch { this: CoroutineScope  
47                 cache.save(entity)  
48             }  
49         }  
50     }  
51
```

```

55 suspend fun findByIdCita(id : String) : Cita?{
56     val citaCached = cache.findById(id)
57     if(citaCached == null){
58         val cita = citaRepository.findById(id)
59         if (cita.getOrNull() == null){
60             throw CitaControllerException("La cita con el id: $id no existe")
61         }else{
62             return cita.getOrNull()
63         }
64     }else{
65         return citaCached
66     }
67 }
68 }
69
70 suspend fun borrarCita(id : String){
71     withContext(Dispatchers.IO){ this: CoroutineScope
72         launch { this: CoroutineScope
73             citaRepository.delete(id)
74         }
75     }
76     cache.delete(id)
77 }
78 }
79

```

```

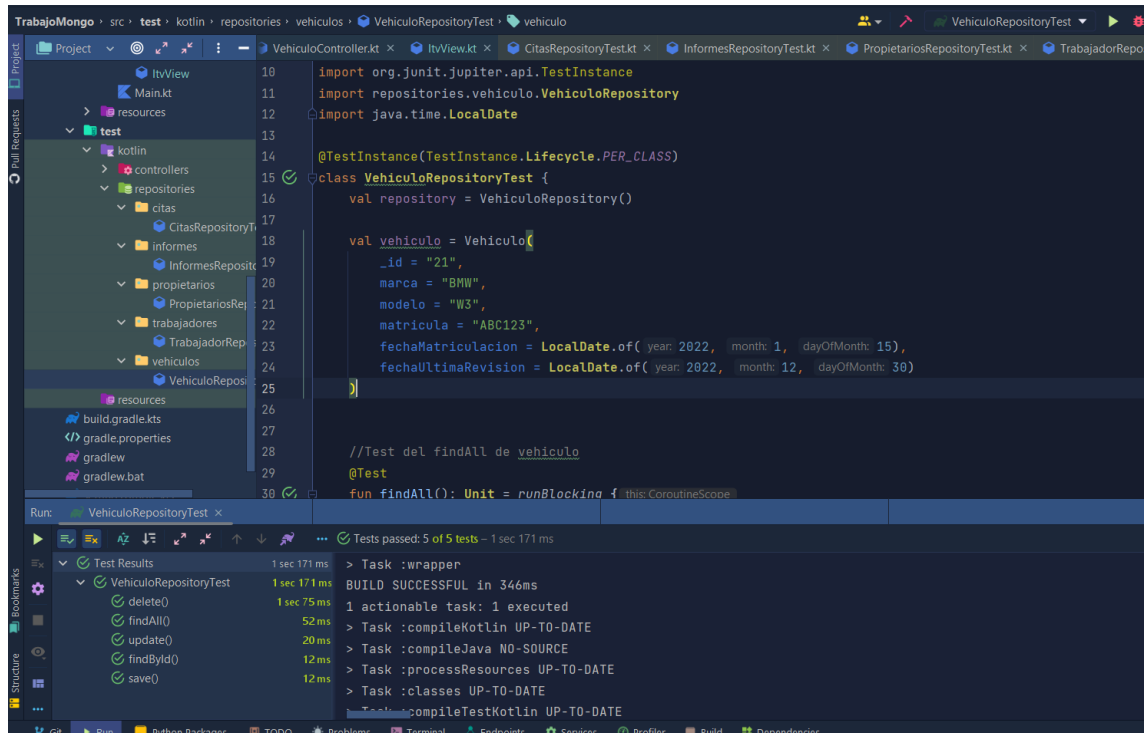
80
81 suspend fun updateCita(entity: Cita){
82     withContext(Dispatchers.IO){ this: CoroutineScope
83         launch { this: CoroutineScope
84             citaRepository.update(entity)
85         }
86     }
87     cache.update(entity)
88 }
89
90
91
92

```

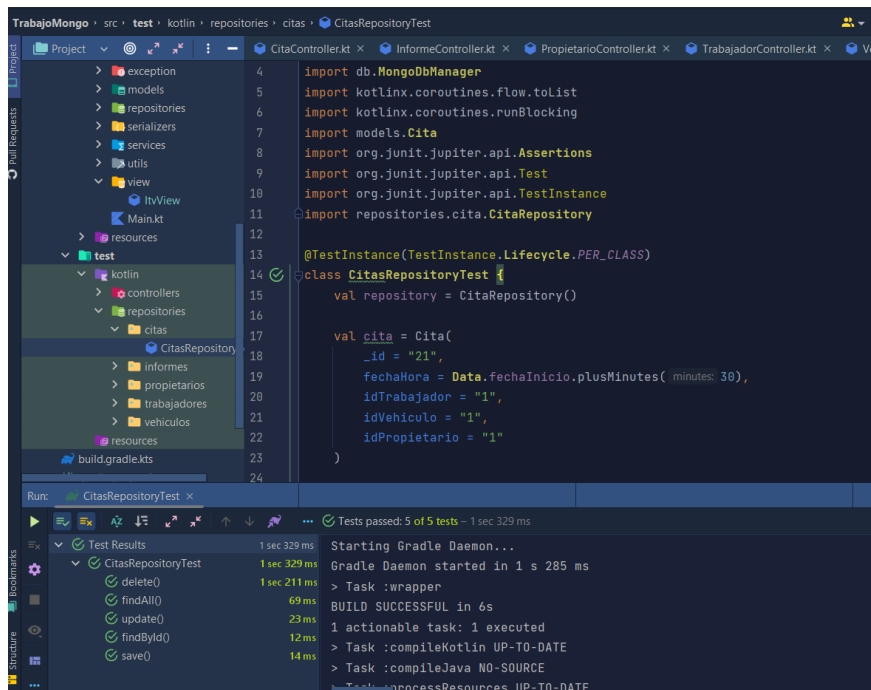
# Test

## Repositorios Test

### VehiculoRepositoryTest



### CitaRepositoryTest



## PropietarioRepositoryTest

The screenshot shows an IDE with the following components:

- Project Structure:** A tree view on the left showing the project hierarchy: `Project` > `resources` > `test` > `kotlin` > `repositories` > `proprietarios` > `PropietariosRepositoryTest`.
- Code Editor:** The main editor displays the `PropietariosRepositoryTest` class. It includes a `@TestInstance` annotation, a `Propietario` object with specific values, and a `findAll` test method that uses `MongoDbManager` to interact with the database.
- Run Panel:** Below the code editor, the `Run` panel shows the execution of the `PropietariosRepositoryTest` class. It indicates that all 5 tests passed successfully within 1 second and 220 milliseconds.
- Test Results:** A detailed view of the test results is shown at the bottom, listing the methods `delete()`, `findAll()`, `update()`, `findById()`, and `save()` with their respective execution times.

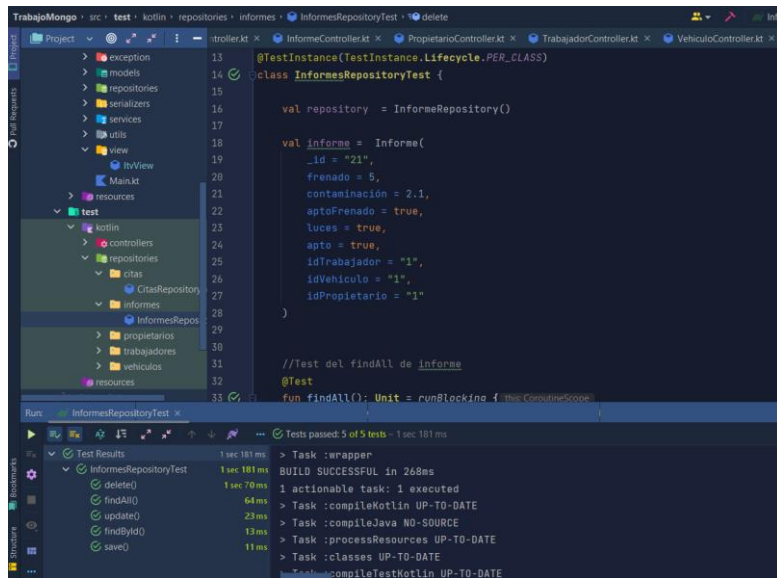
## TrabajadorRepositoryTest

The screenshot shows an IDE with the following components:

- Project Structure:** A tree view on the left showing the project hierarchy: `Project` > `resources` > `test` > `kotlin` > `repositories` > `trabajadores` > `TrabajadorRepositoryTest`.
- Code Editor:** The main editor displays the `TrabajadorRepositoryTest` class. It includes a `@TestInstance` annotation, a `Trabajador` object with specific values, and a `findAll` test method that uses `MongoDbManager` to interact with the database.
- Run Panel:** Below the code editor, the `Run` panel shows the execution of the `TrabajadorRepositoryTest` class. It indicates that all 5 tests passed successfully within 1 second and 241 milliseconds.
- Test Results:** A detailed view of the test results is shown at the bottom, listing the methods `delete()`, `findAll()`, `update()`, `findById()`, and `save()` with their respective execution times.



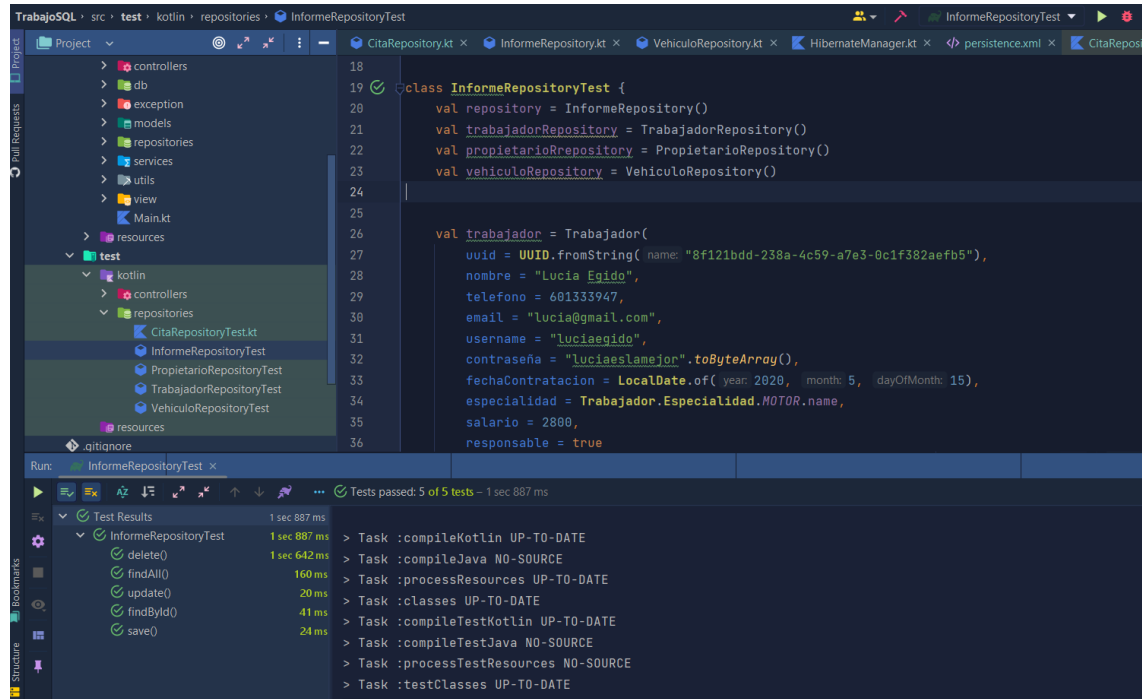
## InformeRepositoryTest



# JPA Hibernate

## Repositorios:

### InformeRepositoryTest



The screenshot shows the IntelliJ IDEA IDE with the `InformeRepositoryTest` class open. The class is located in the `test` directory under `repositories`. The code defines a test class with several test methods and a setup for a `Trabajador` entity.

```
18
19 class InformeRepositoryTest {
20     val repository = InformeRepository()
21     val trabajadorRepository = TrabajadorRepository()
22     val propietarioRepository = PropietarioRepository()
23     val vehiculoRepository = VehiculoRepository()
24
25
26
27     val trabajador = Trabajador(
28         uuid = UUID.fromString("8f121bdd-238a-4c59-a7e3-0c1f382aefb5"),
29         nombre = "Lucia Egido",
30         telefono = 601333947,
31         email = "lucia@gmail.com",
32         username = "Luciaegido",
33         contraseña = "Luciaeslamejor".toByteArray(),
34         fechaContratacion = LocalDate.of(year = 2020, month = 5, dayOfMonth = 15),
35         especialidad = Trabajador.Especialidad.MOTOR.name,
36         salario = 2880,
37         responsable = true
38     )
```

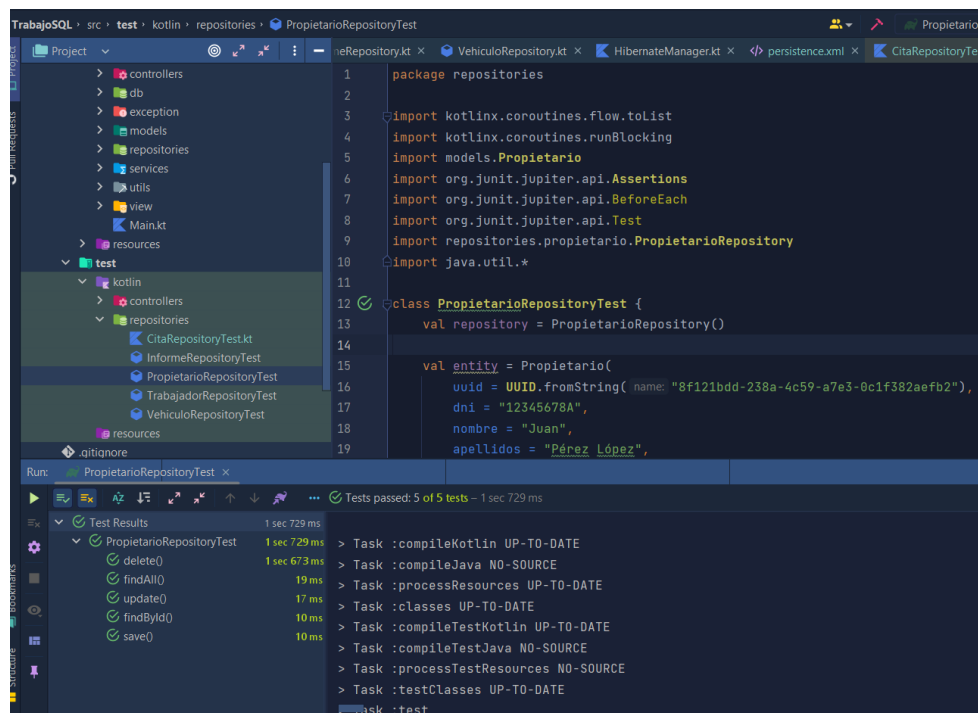
The Run tab shows the test results for `InformeRepositoryTest`. The tests passed are:

- `delete()` (1 sec 887 ms)
- `findAll()` (1 sec 642 ms)
- `update()` (160 ms)
- `findById()` (20 ms)
- `save()` (41 ms)
- `save()` (24 ms)

The Run tab also shows the tasks performed during the test run:

- `Task :compileKotlin UP-TO-DATE`
- `Task :compileJava NO-SOURCE`
- `Task :processResources UP-TO-DATE`
- `Task :classes UP-TO-DATE`
- `Task :compileTestKotlin UP-TO-DATE`
- `Task :compileTestJava NO-SOURCE`
- `Task :processTestResources NO-SOURCE`
- `Task :testClasses UP-TO-DATE`
- `Task :test`

### PropietarioRepositoryTest



The screenshot shows the IntelliJ IDEA IDE with the `PropietarioRepositoryTest` class open. The class is located in the `test` directory under `repositories`. The code defines a test class with several test methods and a setup for a `Propietario` entity.

```
1 package repositories
2
3 import kotlinx.coroutines.flow.toList
4 import kotlinx.coroutines.runBlocking
5 import models.Propietario
6 import org.junit.jupiter.api.Assertions
7 import org.junit.jupiter.api.BeforeEach
8 import org.junit.jupiter.api.Test
9 import repositories.propietario.PropietarioRepository
10 import java.util.*
11
12 class PropietarioRepositoryTest {
13     val repository = PropietarioRepository()
14
15     val entity = Propietario(
16         uuid = UUID.fromString("8f121bdd-238a-4c59-a7e3-0c1f382aefb2"),
17         dni = "12345678A",
18         nombre = "Juan",
19         apellidos = "Pérez López",
20     )
```

The Run tab shows the test results for `PropietarioRepositoryTest`. The tests passed are:

- `delete()` (1 sec 729 ms)
- `findAll()` (1 sec 673 ms)
- `update()` (19 ms)
- `findById()` (17 ms)
- `save()` (10 ms)
- `save()` (10 ms)

The Run tab also shows the tasks performed during the test run:

- `Task :compileKotlin UP-TO-DATE`
- `Task :compileJava NO-SOURCE`
- `Task :processResources UP-TO-DATE`
- `Task :classes UP-TO-DATE`
- `Task :compileTestKotlin UP-TO-DATE`
- `Task :compileTestJava NO-SOURCE`
- `Task :processTestResources NO-SOURCE`
- `Task :testClasses UP-TO-DATE`
- `Task :test`

## VehiculoRepositoryTest

The screenshot shows the IntelliJ IDEA interface with the `VehiculoRepositoryTest` class open. The class is located in the `repositories` package. It contains a `delete()` method and a `save()` method. The test results show that all tests passed.

```
1 package repositories
2
3 import ...
12
13 class VehiculoRepositoryTest {
14
15     val repository = VehiculoRepository()
16
17
18     val entity = Vehiculo(
19         uuid = UUID.fromString( name: "8f121bdd-238a-4c59-a7e3-0c1f382aefb",
20         marca = "Mercedes-Benz",
21         modelo = "E-Class",
22         matricula = "STU901",
23         fechaMatriculacion = LocalDate.of( year: 2021, month: 3, dayOfMonth: 1,
24         fechaUltimaRevisión = LocalDate.of( year: 2022, month: 7, dayOfMonth: 1,
25     )
26
27     @BeforeEach
```

Run: VehiculoRepositoryTest x

Tests passed: 5 of 5 tests - 1 sec 809 ms

Test Results

- VehiculoRepositoryTest 1 sec 809 ms
  - delete() 1 sec 751 ms
  - findAll() 21 ms
  - update() 19 ms
  - findById() 7 ms
  - save() 11 ms

Task :compileKotlin UP-TO-DATE

Task :compileJava NO-SOURCE

Task :processResources UP-TO-DATE

Task :classes UP-TO-DATE

Task :compileTestKotlin UP-TO-DATE

Task :compileTestJava NO-SOURCE

Task :processTestResources NO-SOURCE

Task :testClasses UP-TO-DATE

## TrabajadorRepositoryTest

The screenshot shows the IntelliJ IDEA interface with the `TrabajadorRepositoryTest` class open. The class is located in the `repositories` package. It contains a `delete()` method and a `save()` method. The test results show that all tests passed.

```
1 package repositories
2
3 import ...
12
13 class TrabajadorRepositoryTest {
14
15     val repository = TrabajadorRepository()
16
17
18     val entity = Trabajador(
19         uuid = UUID.fromString( name: "8f121bdd-238a-4c59-a7e3-0c1f382aefb",
20         nombre = "Lucia Egido",
21         telefono = 601333947,
22         email = "lucia@gmail.com",
23         username = "luciaegido",
24         contraseña = "Luciaeslamejor".toByteArray(),
25         fechaContratacion = LocalDate.of( year: 2020, month: 5, dayOfMonth: 1,
26         especialidad = Trabajador.Especialidad.MOTOR.name,
27         salario = 2800,
28         responsable = true
29     )
30
```

Run: TrabajadorRepositoryTest x

Tests passed: 4 of 4 tests - 1 sec 713 ms

Test Results

- TrabajadorRepositoryTest 1 sec 713 ms
  - delete() 1 sec 676 ms
  - findAll() 22 ms
  - findById() 6 ms
  - save() 9 ms

Task :compileKotlin UP-TO-DATE

Task :compileJava NO-SOURCE

Task :processResources UP-TO-DATE

Task :classes UP-TO-DATE

Task :compileTestKotlin UP-TO-DATE

Task :compileTestJava NO-SOURCE

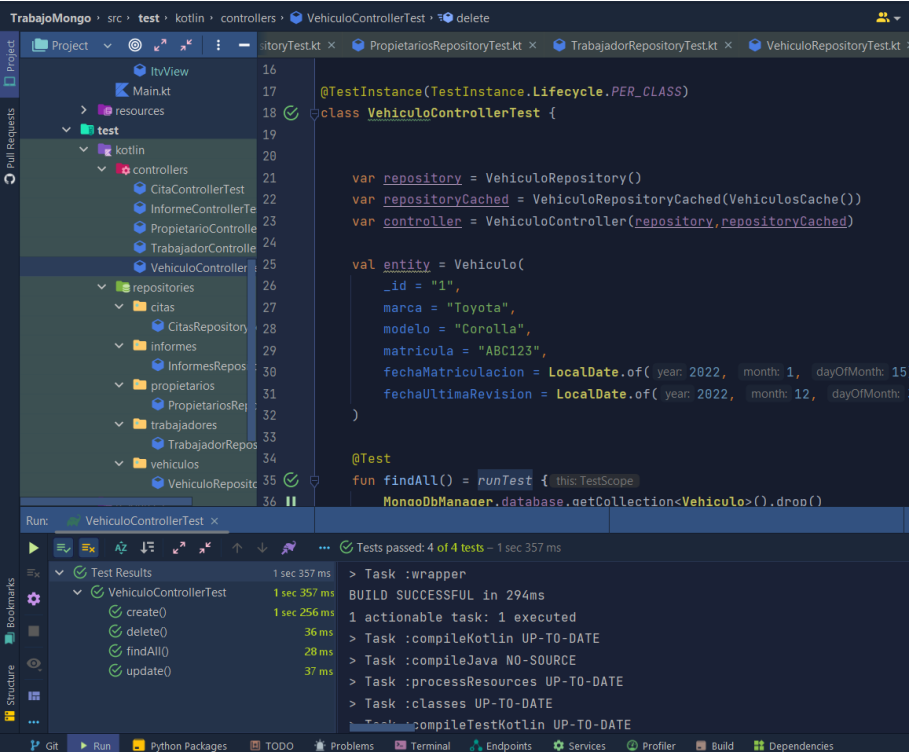
Task :processTestResources NO-SOURCE

Task :testClasses UP-TO-DATE

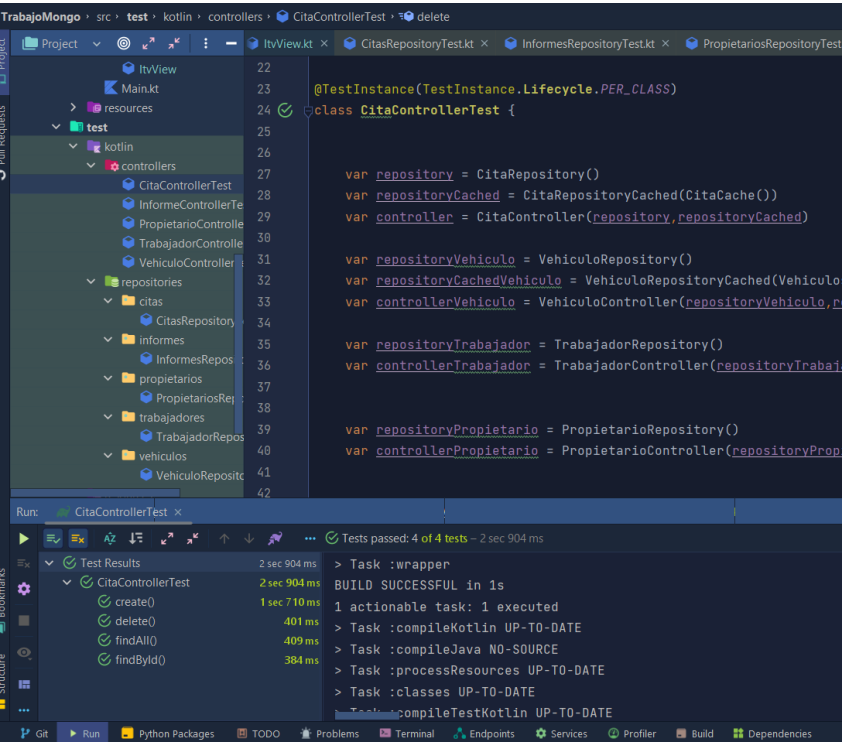
# Controlador Test

El controlador al ser el mismo en los 4 proyectos, lo pongo una vez de ejemplo.

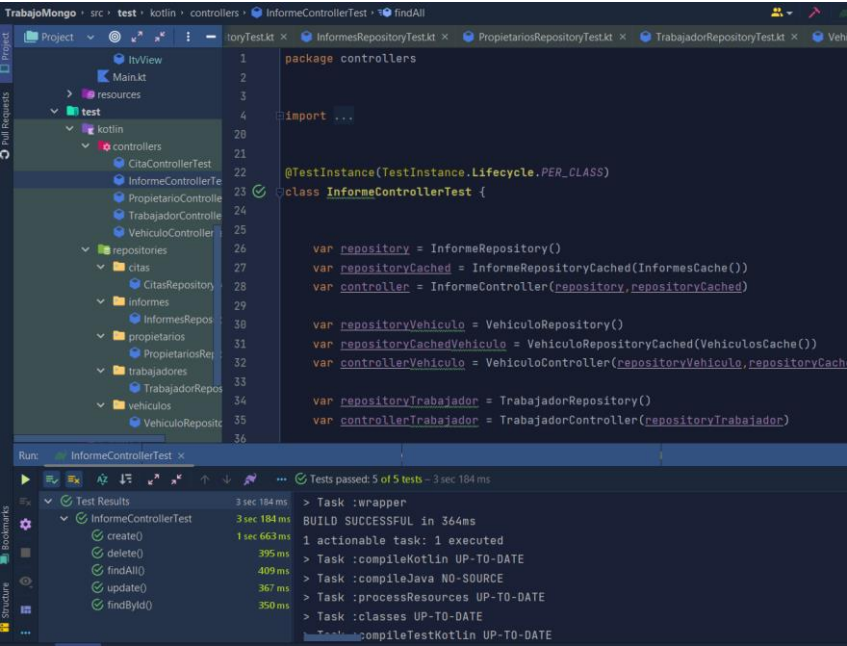
## VehiculoControllerTest



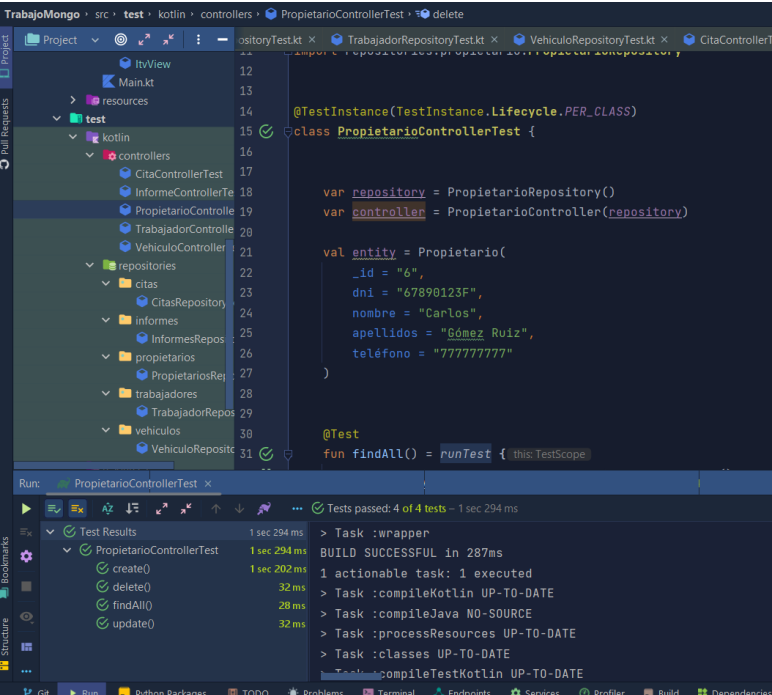
## CitaControllerTest



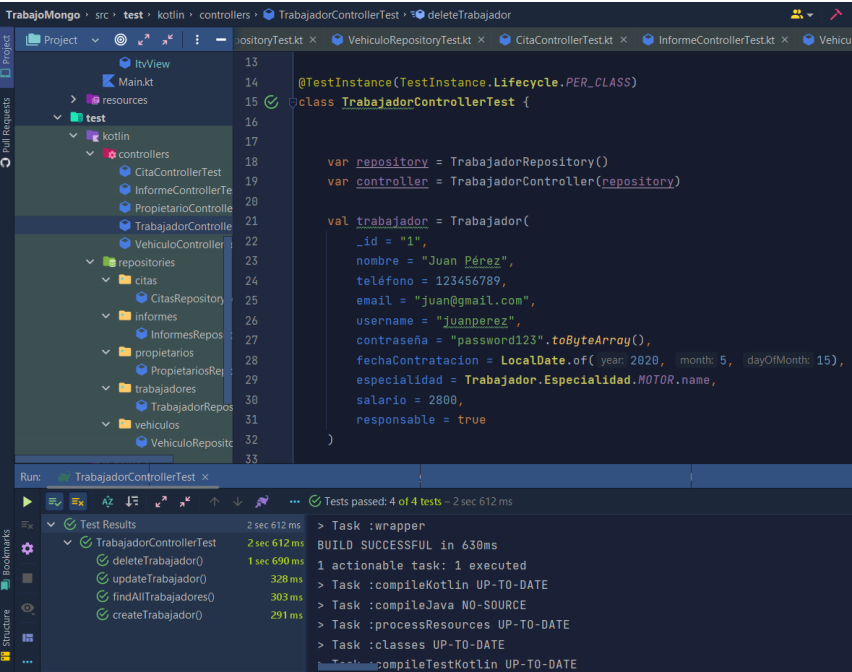
# InformeControllerTest



# PropietarioControllerTest



# TrabajadorControllerTest



## Beneficios de utilizar MongoDB Reactivo

1. Mejora del rendimiento: Al adoptar la programación reactiva y las corrutinas, se logra un mejor rendimiento en las aplicaciones. La ejecución asíncronica evita bloqueos y tiempos de espera innecesarios, permitiendo que la aplicación siga siendo receptiva y eficiente mientras espera las respuestas de la base de datos.
2. La capacidad de respuesta rápida y la capacidad de gestionar múltiples solicitudes simultáneamente mejoran significativamente la experiencia del usuario. Al evitar bloqueos, la aplicación proporciona una experiencia fluida y sin interrupciones, lo que resulta en una mayor satisfacción del usuario.
3. Gestión eficiente de errores: La programación reactiva y las corrutinas proporcionan mecanismos efectivos para manejar errores. Se pueden utilizar operadores de manejo de errores para capturar y gestionar excepciones de manera más elegante, garantizando una respuesta adecuada y una recuperación adecuada en caso de fallas.



**Reactive  
Mongo**

## Beneficios de utilizar JPA Hibernate

1. **Abstracción del acceso a la base de datos:** JPA Hibernate proporciona una capa de abstracción que simplifica el acceso y la manipulación de datos en la base de datos. Esto significa que no tienes que preocuparte por escribir consultas SQL complejas y te permite trabajar directamente con objetos Java, lo que agiliza el desarrollo y mejora la legibilidad del código.
2. **Mapeo objeto-relacional automático:** Hibernate permite mapear automáticamente los objetos Java a las tablas de la base de datos y viceversa. Con la configuración adecuada, puedes generar automáticamente las sentencias SQL necesarias para crear y actualizar las tablas, lo que reduce la carga de trabajo manual y evita errores comunes.





## Beneficios de utilizar Spring con Mongo Db

1. Operaciones CRUD simplificadas: Spring Data MongoDB proporciona una interfaz de repositorio que facilita las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. No necesitas escribir código repetitivo para realizar operaciones comunes, ya que Spring Data MongoDB genera automáticamente las consultas correspondientes según las convenciones y reglas definidas en tu código.
2. Consultas personalizadas: Aunque Spring Data MongoDB maneja muchas operaciones CRUD automáticamente, también te permite definir consultas personalizadas cuando sea necesario. Puedes utilizar anotaciones o consultas basadas en el lenguaje de consultas de MongoDB (Mongo Query Language) para acceder a funcionalidades más avanzadas de MongoDB.



## Beneficios de utilizar Spring con JPA

1. Simplificación del acceso a datos: Spring proporciona una capa de abstracción sobre JPA, lo que simplifica el acceso y la manipulación de datos en la base de datos. Puedes trabajar directamente con objetos Java en lugar de escribir consultas SQL complejas, lo que agiliza el desarrollo y mejora la legibilidad del código.
2. Configuración flexible y fácil: Spring facilita la configuración de JPA en tu aplicación. Puedes utilizar anotaciones y archivos de configuración XML para definir entidades, relaciones y propiedades de persistencia. Además, Spring ofrece una integración sencilla con herramientas de construcción y entornos de desarrollo, lo que simplifica aún más la configuración y puesta en marcha de tu aplicación.

A green rectangular graphic with a vertical gradient from light green at the top to dark green at the bottom. Centered in the middle is the text 'Spring JPA' in white, with '@Query' in yellow below it.

Spring JPA  
@Query

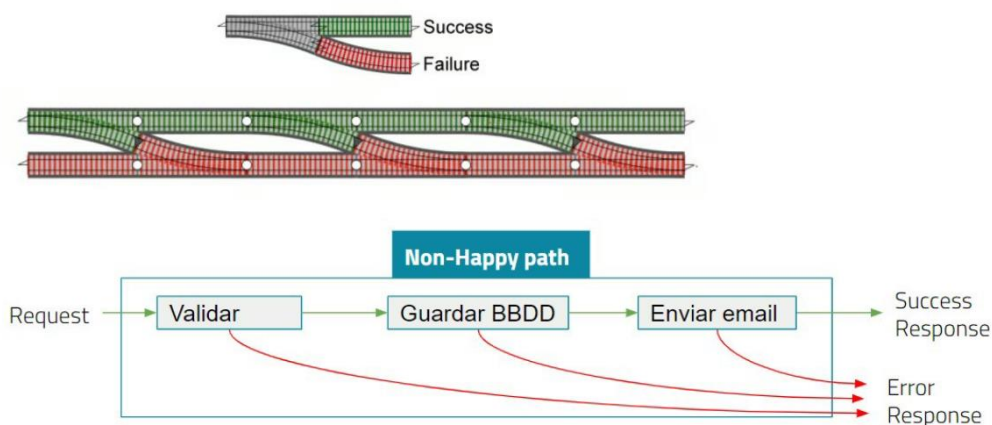
# Patrones Utilizados

**Patrón MVC (Modelo-Vista-Controlador):** Este patrón se utiliza para separar la lógica de presentación de los datos y las reglas de negocio. El modelo representa los datos y la lógica subyacente, la vista se encarga de la presentación visual y el controlador maneja las solicitudes del usuario y coordina la interacción entre el modelo y la vista.

**Patrón Repositorio:** Este patrón se utiliza para abstraer la capa de acceso a datos y proporcionar una interfaz común para realizar operaciones CRUD en la base de datos. Un repositorio actúa como una colección de objetos y proporciona métodos para recuperar, insertar, actualizar y eliminar entidades.

**Patrón Inyección de Dependencias:** Este patrón se utiliza para desacoplar las dependencias entre los componentes de una aplicación. En lugar de que un objeto cree o busque directamente sus dependencias, estas se inyectan en el objeto desde el exterior. Esto permite una mayor flexibilidad y facilidad de prueba, así como una mejor modularidad y reutilización de componentes.

**Railway Oriented Programming:** Este patrón de diseño que nos permite escribir código más limpio y mantenible. Es una técnica de programación funcional que nos permite manejar errores de forma más sencilla y segura. En lugar de usar excepciones, se usan valores de retorno para indicar si una operación ha tenido éxito o no. En el caso de que la operación haya fallado, se devuelve un valor que indica el error. Se van encadenando operaciones que pueden fallar, y en caso de que alguna de ellas falle, se devuelve el error. De esta forma, se evita el uso de excepciones, que pueden ser difíciles de manejar.



## Notificaciones en Tiempo Real

Para obtener notificaciones en tiempo real de las citas existentes y reaccionar a los cambios realizados posteriormente, he utilizado un flujo caliente llamado `SharedFlow` con la configuración de `replay = 1`. Esta configuración me permite recibir la última modificación realizada en las citas, perdiendo las modificaciones anteriores.

En resumen, mediante el uso de un `SharedFlow` con `replay = 1`, he implementado un mecanismo para obtener notificaciones en tiempo real de las citas existentes y reaccionar a los cambios posteriores. Esto permite que mi programa sea reactivo y proporcione una experiencia de obtención de notificaciones en tiempo real para los usuarios.

