1. Memoria

El 23 de octubre de 2023, el Centro ITICBCN, una institución educativa especializada en Informática y Comunicaciones, nos contactó para encargarnos del desarrollo de una aplicación integral. Esta aplicación tiene como objetivo mejorar y agilizar la gestión de su bolsa de trabajo para estudiantes. El Centro ITICBCN ha expresado la necesidad de optimizar sus procesos actuales, que en su mayoría se realizan manualmente, con el fin de brindar a sus estudiantes un acceso más eficiente a las oportunidades de empleo proporcionadas por las empresas colaboradoras.

2. Objeto del proyecto

2.1 Usuarios y contexto de uso

En la situación actual, la gestión de ofertas de trabajo y la búsqueda de oportunidades de prácticas laborales para los alumnos suelen ser tareas que requieren tiempo y recursos considerables, ya que son procesos que se realizan de forma manual. Desde la recopilación y distribución de información, la validación de los perfiles de los estudiantes, el envío de currículos a empresas, entre otras tareas administrativas. Estos procesos, además de ser laboriosos, a menudo pueden tener errores humanos.

Por dicho motivo, se realizará una aplicación que permita automatizar una serie de funcionalidades con el objetivo de mejorar la eficacia del trabajo a realizar.

Se realizará una aplicación para móviles y una página web en la que empresas puedan añadir ofertas de prácticas y los alumnos puedan postular en ellas. Se trabajarán dos perfiles: alumnos y administradores, el primero con el objetivo de que puedan observar las ofertas y escoger empresas con las que se sientan cómodos y más afines a su perfil y el segundo perfil con el objetivo de que administre el tema de ofertas de empresa.

Principalmente, tanto alumnos como administradores tienen que tener un login, una vez dentro de la aplicación, los alumnos tendrán un apartado para insertar sus datos

personales y su CV. Los alumnos, tendrán un buscador donde prácticas donde podrán apuntarse y se enviará un correo automáticamente a la empresa en cuestión, donde habrá la información del alumno y su CV.

Los administradores, gestionan tanto las ofertas como las empresas, pudiendo modificar, insertar, como eliminar. También tendrán que validar los CV de los alumnos y encargarse de que la información es correcta y podrán validarlos masivamente mediante un documento CSV.

2.2 Funcionalidad

La plataforma proporciona a los alumnos la capacidad de buscar empleo, cargar sus CV y postularse a ofertas una vez que sus CV sean validados. A los administradores se les da control sobre las ofertas de empleo, la gestión de usuarios y la validación de CV, entre otras funciones. Ambos tipos de usuarios deben autenticarse a través de un inicio de sesión para acceder a estas funcionalidades.

Funcionalidades para Alumnos:

- Registro: Los alumnos deberán registrarse en la plataforma introduciendo su nombre, apellidos, correo electrónico y una contraseña segura.

- Login: Tendrán que iniciar sesión en la plataforma con su correo electrónico y contraseña para acceder a la plataforma digital.

- Subir CV: Los estudiantes podrán cargar su currículum vitae en formato digital para su revisión y validación.

- Buscar Ofertas de Empleo: Podrán encontrar diferentes ofertas de empleo utilizando filtros como ubicación, industria, y tipo de empleo.

- Visualizar Detalles de Ofertas: Al encontrar una oferta que les interese podrán ver detalles de estas, que incluyen información sobre la empresa, requisitos, beneficios y cómo postular.

- Aplicar a Ofertas: Una vez que el CV del alumno sea validado, podrán postularse a las ofertas de empleo que les interesen.

- Notificaciones: Los alumnos recibirán notificaciones por correo electrónico, sobre el estado de sus aplicaciones.

Funcionalidades para Administradores:

- Inicio de Sesión: Los administradores también deberán hacer un login con credenciales específicas para acceder a las funcionalidades de administrador.

- Gestión de Ofertas de Empleo: Podrán agregar nuevas ofertas, incluyendo información detallada sobre la oferta, requisitos y fecha.

- Editar y Eliminar Ofertas: Tendrán la capacidad de editar o eliminar ofertas de empleo existentes en caso de cambios en los detalles de la oferta o cuando expiren.

- Validación de CV: Los administradores revisarán y validarán los curriculum de los alumnos para asegurarse de que cumplan con los requisitos antes de permitirles postularse a ofertas.

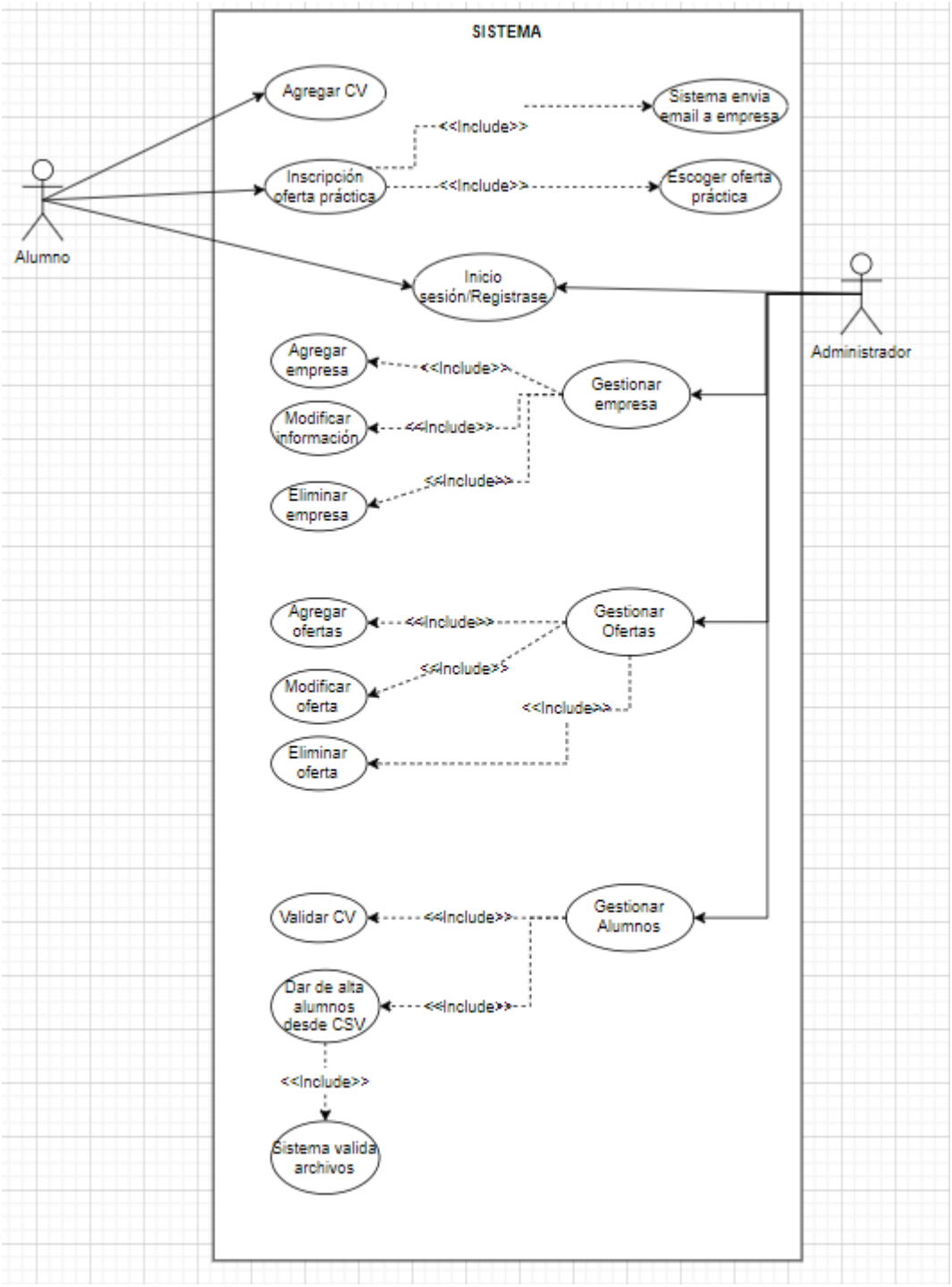
- Registro Masivo: Los administradores podrán realizar un registro masivo de alumnos en la plataforma a través de la carga de un archivo CSV que contenga información de los estudiantes.

- Gestión de Usuarios: Los administradores podrán crear y gestionar cuentas de usuario para alumnos, también tendrán habilitado restablecer contraseñas en caso de que la olviden.

- Historial de Actividades: Se mantendrá un registro de las actividades realizadas por los administradores para garantizar la transparencia y la seguridad de la plataforma.

3. Funcionamiento

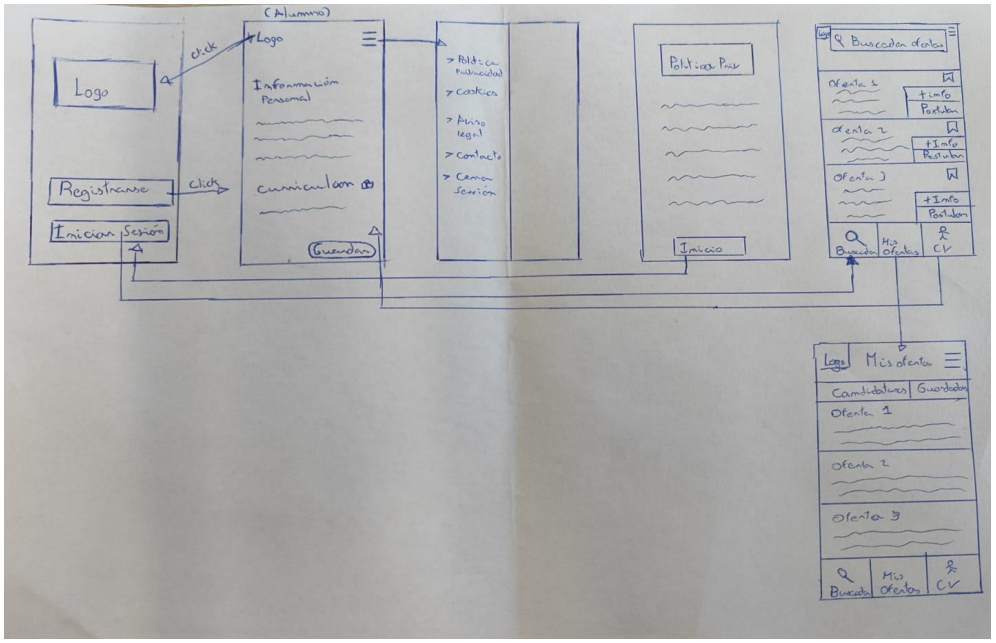
3.1 Diagrama de casos de uso

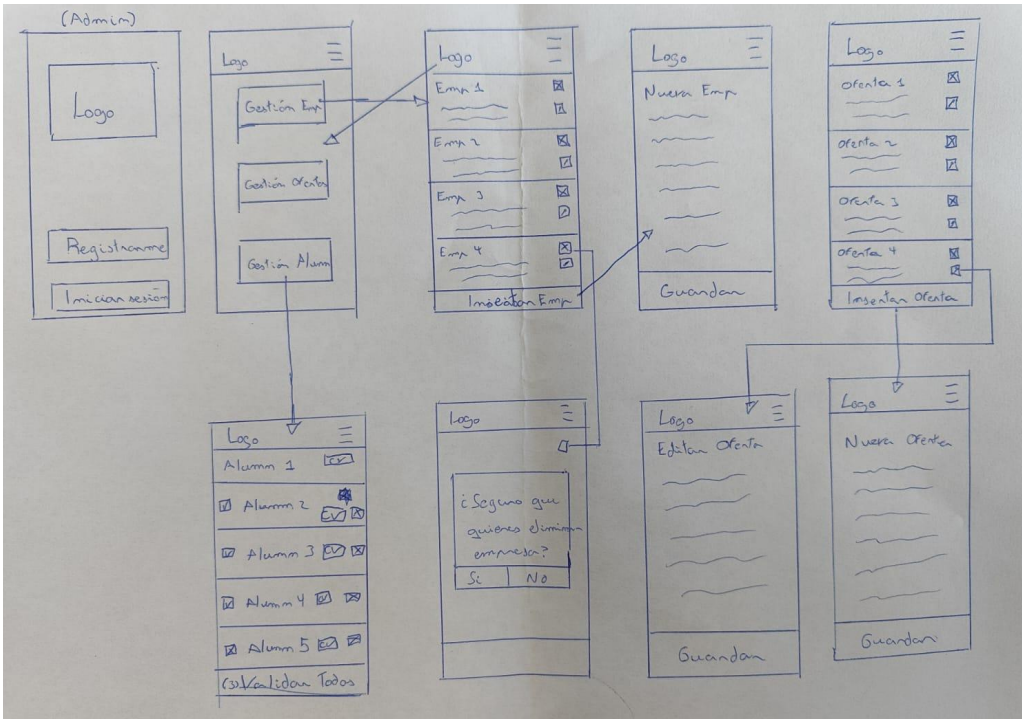


Enlace Diagrama:

https://drive.google.com/file/d/1Q3wmaqDjd9YFx1DBmwBL7l-vS2_AsUEm/view?usp=sharing

3.2 Wireframe de baja fidelidad





4. Api-Rest

Requisitos previos para el inicio del API REST:

- Ir a *Spring Initializr* y agregar las dependencias de: WEB, H2 y JPA.
- Descargar el archivo .ZIP y descomprimirlo.
- Abrir el proyecto con un IDE a elección, en nuestro caso, Eclipse.

Con esos pasos ya hechos, comenzamos con el desarrollo de las funcionalidades del API REST.

Comenzamos creando la clase objeto, en este caso es **Empresa**, ya que en ella recaen las funcionalidades que vamos a ejecutar. Funciona como una entidad específica, que contiene una serie de atributos y métodos, como los getters y setters correspondientes y la implementación de un método equals que nos permite comparar cada campo de la clase para verificar si son iguales en ambas estancias.

Utilizaremos métodos JPA como *@Entity*, para marcar la clase Empresa como una entidad JPA y pueda ser mapeable a una tabla en una base de datos.
@Table para especificar el nombre de la tabla en la base de datos a la que se mapeara la entidad.
@Id para marcar un campo como clave primaria de la entidad.
@GeneratedValue que lo utilizaremos junto al método mencionado anteriormente para especificar cómo se generará el valor de la clave primaria.

A continuación, configuramos el archivo **application.properties** con:

- *spring.datasource.url=jdbc:h2:mem:testdb* para establecer la URL de la base de datos y utilizarla como prueba y desarrollo del proyecto, hay que tener en cuenta que cada vez que iniciamos la aplicación, la base de datos se elimina y se comienza de 0.
- *spring.datasource.driverClassName=org.h2.Driver* especificamos el nombre del controlador, en nuestro caso H2.
- *spring.datasource.username=* definimos el nombre de usuario para acceder a la base de datos.
- *spring.datasource.password=* establecer la contraseña de nuestro usuario de la base de datos.
- *spring.h2.console.enabled=true* Habilita la consola web de H2 para acceder y administrar la base de datos.

- *spring.jpa.show-sql=true* permite que Spring Boot imprima las sentencias SQL generada por JPA.
- *spring.jpa.hibernate.ddl-auto=update* para controlar la creación y actualización de la estructura de la base de datos.

El siguiente paso fue configurar una interfaz que extiende *JpaRepository* para que interactúe con la entidad **Empresa**. Se utilizaron dos parámetros: *Empresa* y *Long*.
Utilizamos la anotación @*Repository* para marcar la interfaz como un componente de Spring. Gracias a *JpaRepository* que proporciona una amplia variedad de métodos para realizar operaciones de acceso a datos en la entidad **Empresa** pudimos desarrollar las siguientes funcionalidades creando un controlador y un servidor que manejan las solicitudes HTTP entrantes:

Comenzamos creando la clase Servidor, la cual está anotada con @*Component* lo que automáticamente la convierte en un elemento de Spring y de esta manera nos permite utilizar sus dependencias para realizar los métodos CRUD de la entidad **Empresa**. También utilizamos la anotación @*Autowired* para integrar el repositorio que creamos de **Empresa** y de esta manera poder interactuar con él.

Después creamos la clase Controlador, con la anotación @*RestController* para manejar las solicitudes HTTP y la anotación @*RequestMapping* con una URL asignada para establecer un punto común para todas las rutas URL de los métodos CRUD creados.

Utilizamos @*Autowired* de la instancia del Servidor para realizar las operaciones de los métodos establecidos en la entidad **Empresa**. Además cada método utiliza un endpoint REST, que básicamente es un punto establecido en la API al que se puede acceder a través de una solicitud HTTP.

En los métodos creados también existen las anotaciones @*RequestBody* y @*PathVariable*. La primera anotación la utilizamos para obtener información de una solicitud realizada en formato JSON, el cual usamos para las pruebas a través de POSTMAN. Y la segunda anotación, sirve para capturar los valores que necesitamos de la dirección URL asignada.

Por último, manejamos las excepciones a través de bloques **try-catch** y poder devolver respuestas personalizadas.

A continuación, los métodos CRUD creados, con sus endpoints respectivos:

US: Como usuario puedo agregar una empresa en la API.

@PostMapping("/empresas")

Los usuarios pueden enviar una solicitud POST con los datos de la empresa en formato JSON y la empresa se guardará en la base de datos.

US: Como usuario puedo listar todas las empresas en la API.

@GetMapping("/empresas")

Los usuarios pueden enviar una solicitud GET para obtener una lista de todas las empresas que se encuentran en la base de datos. Utilizamos *List*<> para representar un conjunto de datos, en este caso, un conjunto de empresas.

US: Como usuario puedo consultar una empresa en la API.

@GetMapping("/empresas/{id}")

Los usuarios pueden enviar una solicitud GET para obtener información sobre una empresa en específico a través de una ID en la URL.

US: Como usuario, puedo modificar una empresa en la API.

@PutMapping("/empresas/{id}")

Los usuarios deben proporcionar el ID de la empresa que desean modificar y enviar una solicitud PUT con los datos actualizados en formato JSON. El controlador llamará al servicio para actualizar la empresa correspondiente en la base de datos.

US: Como usuario, puedo eliminar una empresa de la API.

@DeleteMapping("/empresas/{id}")

Los usuarios deben proporcionar el ID de la empresa que desean eliminar y enviar una solicitud DELETE. La empresa correspondiente se eliminará de la base de datos.

Para la siguiente parte del proyecto, nos informarnos sobre cómo funcionan las relaciones 1-N mediante anotaciones JPA

Inicialmente creamos la entidad **Oferta** para representarla en el sistema y asociar cada oferta creada a la empresa que nos interese. Utilizamos las anotaciones JPA *@Entity* y @Table para indicar que **Oferta** es una entidad y se encuentra mapeada a una tabla asignada. Esta entidad continúa el mismo proceso que la entidad **Empresa**, donde el esqueleto de la clase se encuentra formado por los atributos, un constructor y los getters y setters correspondientes.

@JoinColumn(name = "empresa_id"): Especifica el nombre de la columna en la tabla de ofertas que se utilizará como clave externa para la relación.
También, agregamos la anotación *@ManyToOne* para crear la relación con la entidad **Empresa**, lo que significa que varias ofertas pueden pertenecer a una empresa. En la clase Empresa, se utiliza la anotación @OneToMany para representar la relación con la clase Oferta. Esta anotación específica que una empresa puede tener una colección de ofertas.

@OneToMany(mappedBy = "empresa", cascade = CascadeType.ALL, orphanRemoval = true)
mappedBy = "empresa": Indica que la relación está mapeada por el atributo empresa en la clase Oferta.

cascade = CascadeType.ALL: Especifica que las operaciones de cascada, como eliminar, se aplicarán a las ofertas asociadas cuando se realice una operación en la empresa.

orphanRemoval = true: Indica que las ofertas se eliminarán si se eliminan de la colección en Empresa.

A continuación, los métodos CRUD creados, con sus endpoints respectivos:

US. Como usuario, puedo agregar una oferta de una empresa a la API.
@PostMapping("/empresas/{empresaId}/ofertas")
Los usuarios pueden agregar una oferta enviando una solicitud POST con los datos de la oferta en formato JSON. Las ofertas agregadas se almacenarán en la base de datos. Además deberán tener presente agregar la ID de la empresa a la que está asociada la oferta.

US. Como usuario, puedo listar todas las ofertas de una empresa en la API.
@GetMapping("/ofertas")
Los usuarios pueden listar todas las ofertas existentes a través de una solicitud GET. De la misma manera que empresas, el conjunto de ofertas también se creó utilizando *List*<> para manejar la colección creada.

US. Como usuario, puedo consultar una oferta específica en la API.
@GetMapping("/ofertas/{ofertaId}")
Los usuarios pueden visualizar una oferta en particular a través de una solicitud GET y de la ID de la oferta que les interesa.

US. Como usuario, puedo consultar ofertas de una empresa específica en la API.

@GetMapping("/empresas/{empresaId}/ofertas")

Los usuarios pueden consultar las ofertas de una empresa específica enviando una solicitud GET al endpoint que incluye el ID de la empresa. Esto devolverá una lista de ofertas asociadas a esa empresa en formato JSON.

US. Como usuario, puedo actualizar una oferta de una empresa dada en la API.

@PutMapping("/empresas/{empresaId}/ofertas/{ofertaId}")

Los usuarios deben proporcionar el ID de la oferta que desean actualizar y el ID de la empresa a la que está asignada y enviar una solicitud PUT con los datos actualizados en formato JSON. El controlador llamará al servicio para actualizar la oferta correspondiente en la base de datos.

US. Como usuario, puedo eliminar una oferta de una empresa dada en la API.

@DeleteMapping("/empresas/{empresaId}/ofertas/{ofertaId}")

Los usuarios deben proporcionar el ID de la oferta que desean eliminar y el ID de la empresa a la que está asignada dicha oferta y enviar una solicitud DELETE. La oferta correspondiente se eliminará de la base de datos.

US. Como usuario, si borro una empresa, automáticamente se eliminarán todas las ofertas asociadas a esa empresa.

Esta opción implica que hay una relación entre las empresas y sus ofertas, y cuando se elimina una empresa, las ofertas vinculadas también se eliminan para mantener la coherencia de los datos. Como existe esta relación entre las entidades, aprovechamos el método creado anteriormente para eliminar una Empresa y automáticamente se eliminan las ofertas asociadas.

A continuación, la empresa pidió que se agregaran más consultas básicas aparte del CRUD. Para cumplir dicho objetivo utilizamos los repositorios creados previamente agregando una serie de métodos.

US. Como usuario puedo buscar una empresa a partir de su nombre.

@GetMapping("/empresas/nombre/{nom}")

En el repositorio de **Empresa**, agregamos Empresa findByNom(String nom). Lo que buscamos es implementar un método que por el nombre de la empresa, recupere la información de esta y se pueda visualizar. Spring Data JPA contiene métodos automáticos para consultas, en este caso, utilizando el findBy se genera automáticamente la consulta necesaria para buscar una empresa por su nombre. Cuando en el service utilizamos empresaRepository.findByNom(nom) traduce esto

a una consulta query, como la siguiente, *SELECT e FROM Empresa e WHERE e.nom = :nom*. El usuario a través de una solicitud GET y el nombre de la empresa recupera la información de esta.

US. Como usuario puedo buscar una oferta a partir del tipo de estudio.

@GetMapping("/ofertas/pais/{pais}")

En el repositorio de **Oferta**, agregamos List<Oferta> findByEstudios(String estudios). De la misma manera que actúa findBy anteriormente, actúa en los métodos implementados. El usuario a través de una solicitud GET y el nombre del tipo de estudio que le interese buscar, recupera información de las ofertas asignadas a ese estudio en particular.

US. Como usuario puedo buscar una oferta a partir de la ciudad.

@GetMapping("/ofertas/ciudad/{ciutat}")

En el repositorio de **Oferta**, agregamos List<Oferta> findAllByEmpresaCiutat(String ciutat). El usuario a través de una solicitud GET y la ciudad que le interese, podrá visualizar las ofertas coincidentes con dicha ciudad.

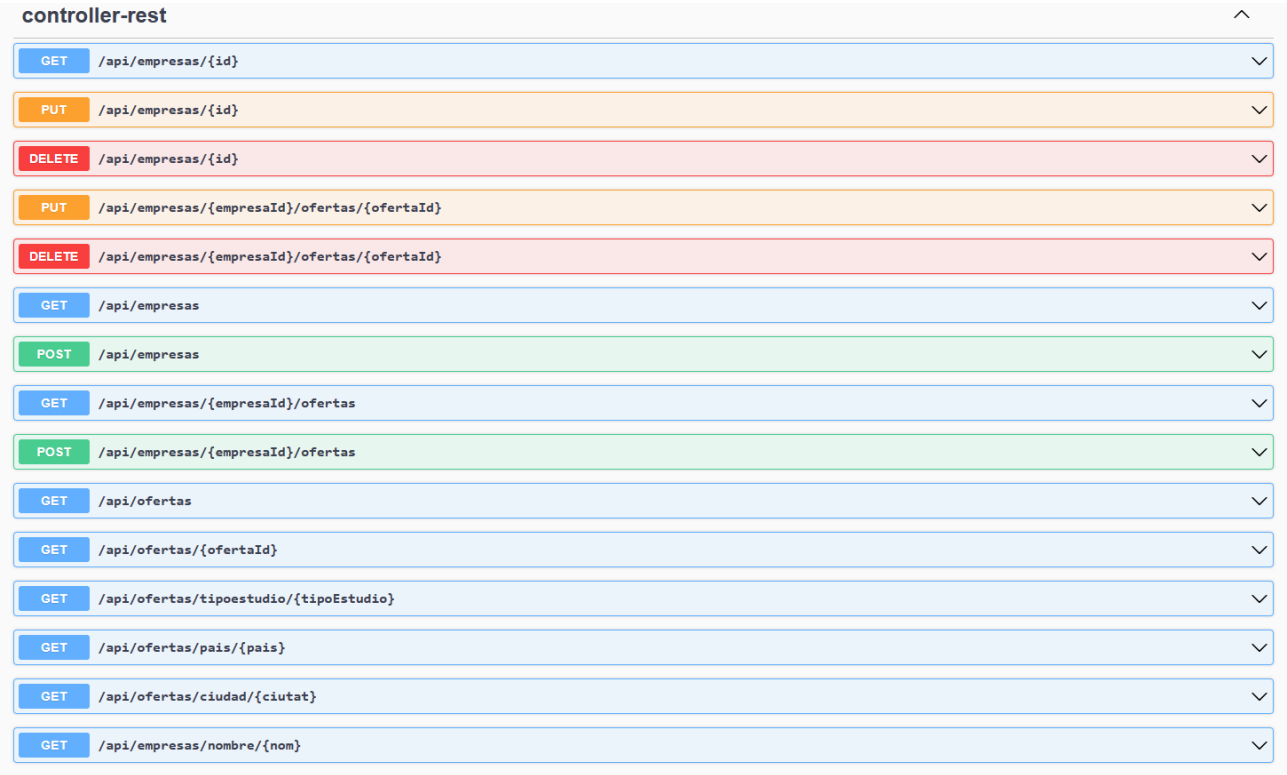
US. Como usuario puedo buscar una oferta a partir del país.

@GetMapping("/ofertas/pais/{pais}")

En el repositorio de **Oferta**, agregamos List<Oferta> findAllByEmpresaPais(String pais). El usuario a través de una solicitud GET y el país que le interese, podrá visualizar las ofertas coincidentes con dicho país.

Para poder visualizar el desarrollo de la API utilizamos en una primera instancia POSTMAN que después por recomendación cambiamos a SWAGGER, lo que nos ayudó para describir los servicios desarrollados gracias a su interfaz interactiva y mucho más entendible que POSTMAN. Para poder hacer esto, tuvimos que agregar al archivo *pom*:

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.0.3</version>
</dependency>
```



5. Capas y test

Pasos previos para capas y tests de persistencia y servicio:

- Hicimos el cambio de IDE Eclipse a IntelliJ para poder manejar de manera más efectiva los errores que se puedan generar, ya que IntelliJ nos ofrece un análisis del código más completo, señalando métodos y/o variables que no se estén utilizando o que tengan un error en particular. Además, IntelliJ nos facilita la creación y ejecución de pruebas unitarias, pudiendo crear una prueba directamente desde el código fuente.

- Hemos decidido inicializar la base de datos para tener información con la que poder realizar las pruebas, mediante dos archivos SQL, uno que inserta información sobre empresas y otro sobre ofertas, ubicados en la carpeta resources. Para asegurarnos de que estos archivos se ejecuten al iniciar la aplicación, hemos agregado los siguientes datos al archivo *properties:*
    - *spring.sql.init.schema-locations=classpath:empresas.sql*
    - *spring.sql.init.data-locations=classpath:ofertas.sql*

  Esta configuración indica a Spring que busque y ejecute los archivos SQL al iniciar la aplicación

Comenzamos con los test en la capa de persistencia. Estos nos permiten evaluar el funcionamiento de las operaciones que interactúan en la base de datos y suelen estar dirigidos a los repositorios. De esta manera, pueden ayudar a identificar problemas de mapeo de entidades, entre otros. Los casos de prueba que vamos a realizar están enfocados en las consultas que hemos creado, ya sean tipo CRUD o las que agregamos adicionalmente a estas.

Nuestras pruebas unitarias son realizadas utilizando JUnit 5 y las anotaciones de Spring Boot correspondientes, como *@DataJpaTest* y *@TestEntityManager*. La primera se utiliza para configurar las pruebas JPA y la segunda nos permite manipular entidades en el desarrollo de las pruebas.

Integraremos mediante *@Autowired* los repositorios que hemos creado anteriormente y nuestra entidad creada para el entorno de pruebas, de esta manera haremos consultas sobre los métodos CRUD y el resto de métodos.

Además, creamos dos métodos de ayuda, tanto de **Empresa** como de **Oferta** que nos facilitan el poder insertar datos de prueba. En dichos métodos podemos encontrar *entityManager.persist* que realiza un seguimiento de la entidad que especifiquemos y sincronizarla con la base de datos y e*ntityManager.flush()* para sincronizar los cambios con la base de datos.

Cada prueba va marcada con la anotación *@Test* para marcar dicho método como un entorno de prueba, y utilizamos *AssertEquals* para poder comparar la salida que buscamos con el resultado final, *AssertTrue* para verificar si la condición que hemos insertado es verdadera y/o *AssertNotNull* para asegurarnos de que el valor que proporcionamos no sea nulo.

Las pruebas realizadas son las siguientes:

@Test public void testCreateEmpresa(). Verifica que se pueda crear una nueva empresa y que los datos coincidan con los valores insertados. Para dichos métodos usamos las demos que hemos creado anteriormente. Además usamos *AssertTrue* para verificar la salida y AssertEquals para la comparación de resultados que esperamos.

@Test public void testAllEmpresas(). Insertamos la demo creada anteriormente y hacemos un *findAll* en el repositorio de la empresa, utilizando finalmente un *AssertEquals* para poder comparar el número de empresas que esperamos con el que realmente hay.

@Test public void testEmpresaFindById() y @Test public void testFindByNom(). Probamos la búsqueda de empresas a través de su ID o del nombre que proporcionamos y las verificamos a través de un *AsserTrue* para verificar que la empresa exista y un *AssertEquals* que compare los resultados que esperamos con los que realmente surgen.

@Test public void testEmpresaUpdate() y @Test public void testEmpresaDeleteById(). Finalmente para terminar con las empresas, probamos actualizar la demo de empresa que creamos y comparamos con un *AssertEquals* los resultados que hemos modificados con los que realmente tiene la empresa y para el siguiente intentamos eliminar una empresa por su ID utilizando un *AssertTrue* y un *isEmpty()* para comprobar que la empresa está vacía, no existe o en su defecto, un mensaje informándonos esta no eliminación.

Con los métodos de oferta sucede exactamente lo mismo. Para los métodos CRUD tenemos:
@Test public void testCreateOferta(), public void testOfertaFindById(), public void testOfertaFindByIdEmpresa(), public void testOfertaUpdate() y public void testOfertaDeleteById(). Siguen la misma lógica que los desarrollados para la empresa.

Para las consultas personalizadas hicimos los siguientes:
@Test public void testOfertadByTipoEstudio(), public void testGetOfertasByCiutat() y **public void testGetOfertasByPais().** Siguen la misma lógica que los desarrollados para la empresa.

Finalmente, también desarrollamos un test utilizando Mockito a la capa de servicio. Una vez finalizada la capa de Servicio, crearemos los test de la clase servicio, creando una carpeta en test llamada com.example.service, siguiendo el mismo package que en la clase. En estos test, usaremos testing unitario que nos permitirá testear los métodos implementados singularmente para ver cómo funcionan. Grácias a IntelliJ, haciendo click derecho, generate test en la clase servicio, nos crea los tests vacíos con los métodos que teníamos hechos.

Iniciamos dicha clase con la anotación *@InjectMocks* en el servicio de la API-Rest que hemos creado para inyectar mocks, objetos simulados que replican el comportamiento de objetos reales en el entorno de prueba.

La anotación *@Mock* marcará los repositorios que creamos y que se utilizan para más tarde usarlos como mocks en la clase de prueba.

De la misma manera que los test hechos en la capa de persistencia, estos también utilizan la anotación *@Test*. Al inicio de la clase tenemos un *@BeforeEach* y

MockitoAnnotations.openMocks(this) para poder inicializar los mocks antes de cada test creado. De esta forma nos evitamos repetir dicha inicialización en cada test que queramos ejecutar.

En general, en cada Test vamos a encontrar *when* y *thenReturn* para poder controlar el comportamiento de los mocks. Por ejemplo, en nuestro primer test llamado **@Test void createEmpresa()** configuramos lo siguiente *when(empresaRepository.save(any(Empresa.class))).thenReturn(empresaEntrada)*, de esta manera, controlamos que cuando se guarda una empresa se retorne la entidad que se ha guardado. Dicha configuración la podemos adaptar según la prueba que estemos realizando, por ejemplo, para la búsqueda por ID configuramos que cuando la empresa encuentre el ID esta retorne la empresa existente a ese ID, y así hasta configurarlo para cada entorno de prueba que ejecutemos.

De la misma manera, es muy propio de los Tests hechos con mockito utilizar *verify* para verificar si los métodos de los mocks tienen los argumentos esperados, como por ejemplo, en el método **@Test void UpdateOferta()** configuramos los siguientes:

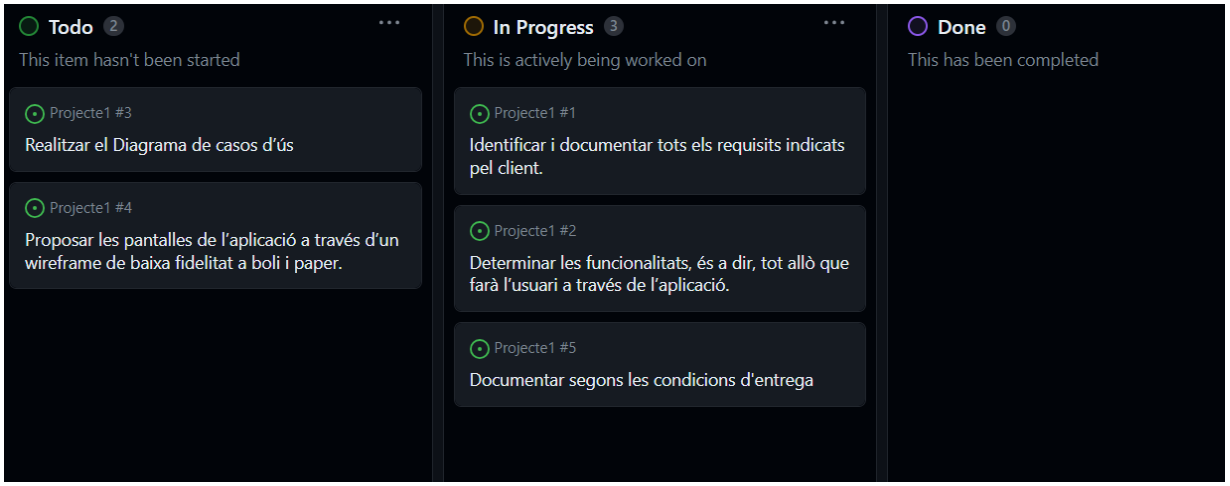
- *verify(ofertaRepository, times(1)).findById(ofertaIdExistente)*. Los utilizamos para verificar que el método *findById* se ha llamado una vez correctamente con el argumento *ofertaIdExistente*. De esta forma nos aseguramos que se haya ejecutado correctamente.
- *verify(ofertaRepository, times(1)).save(ofertaExistente)*. De la misma manera que antes, nos aseguramos que el método *save* se ha llamado una vez con el argumento especificado.
- *verify(ofertaRepository, never()).deleteById(anyLong())*. Verificamos que dicho método no se haya llamado con ningún argumento de tipo Long. De esta manera nos aseguramos que a la hora de actualizar una empresa no se esté intentando eliminar durante dicha operación.

6. KANBAN

6.1 Primer Sprint

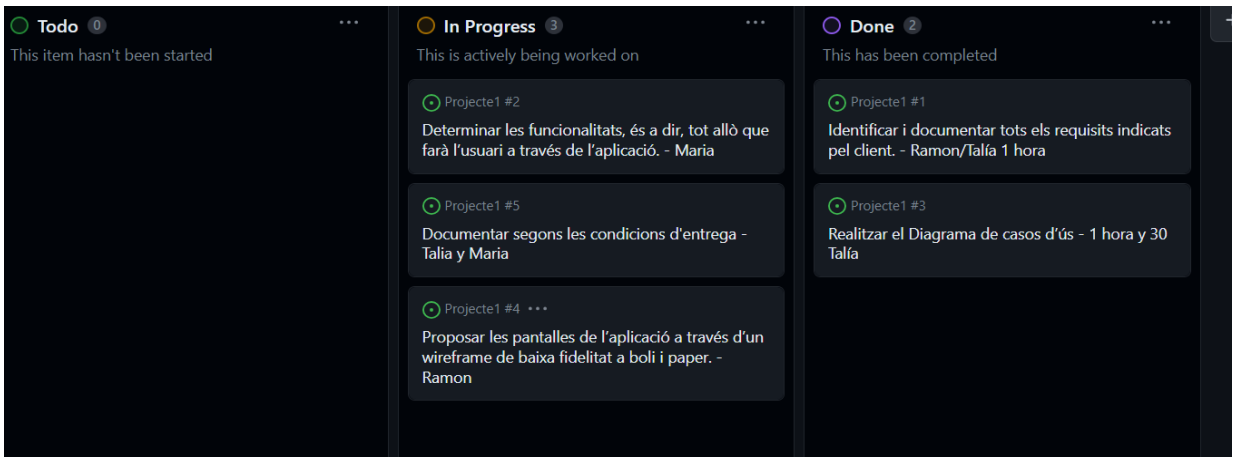
Primer día / Backlog organizativo:

- Comenzamos a definir las tareas que serán el esqueleto organizativo del proyecto.



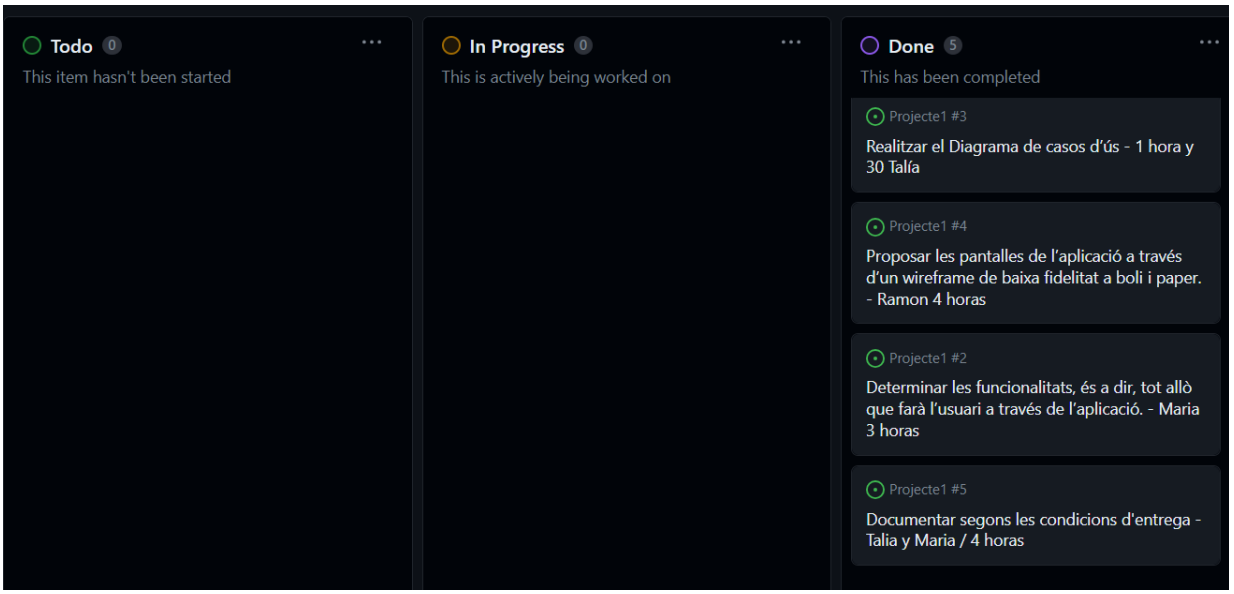
Segundo día:

- Se dan como hechas las dos primeras asignadas y se cronometra el tiempo. Documentar según las condiciones de entrega se queda en progreso porque es un trabajo que se va haciendo junto al desarrollo de la misma memoria y determinar las funcionalidades no se terminó porque se busco la forma de documentar y se buscaron ejemplos de cómo enfocar estas funcionalidades.



Tercer día:

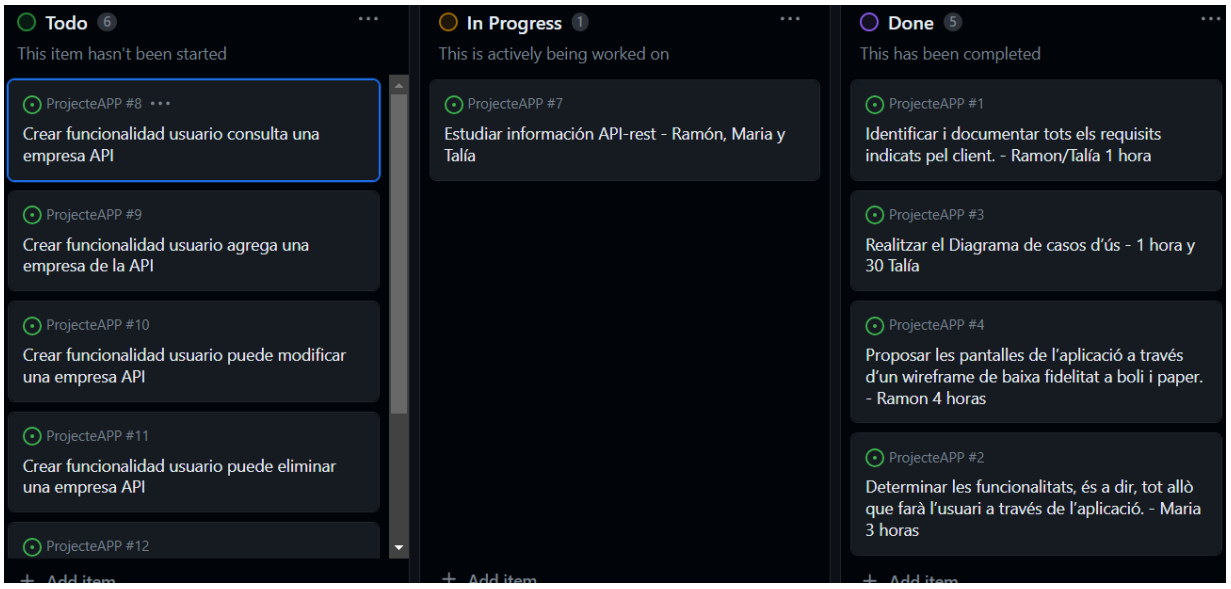
- Se finalizan las tareas y se revisa que todo esté bien documentado.



6.2 Segundo Sprint

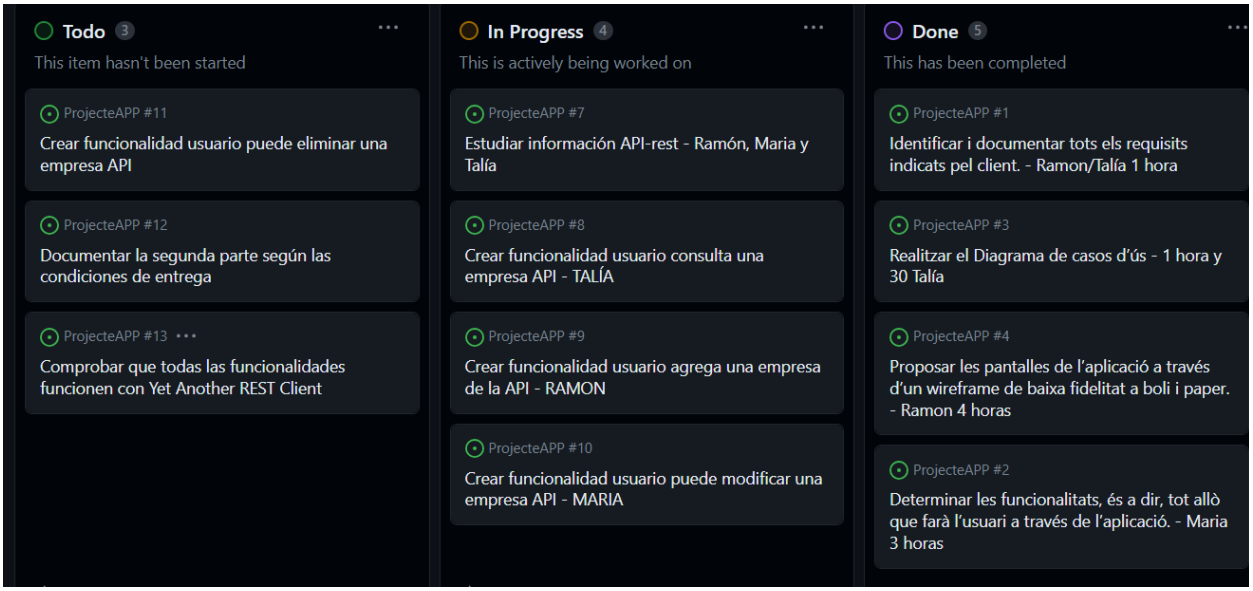
Primer día / backlog organizativo:

- Comenzamos a definir las nuevas tareas a realizar y asignamos las que se ejecutarán el día de hoy.



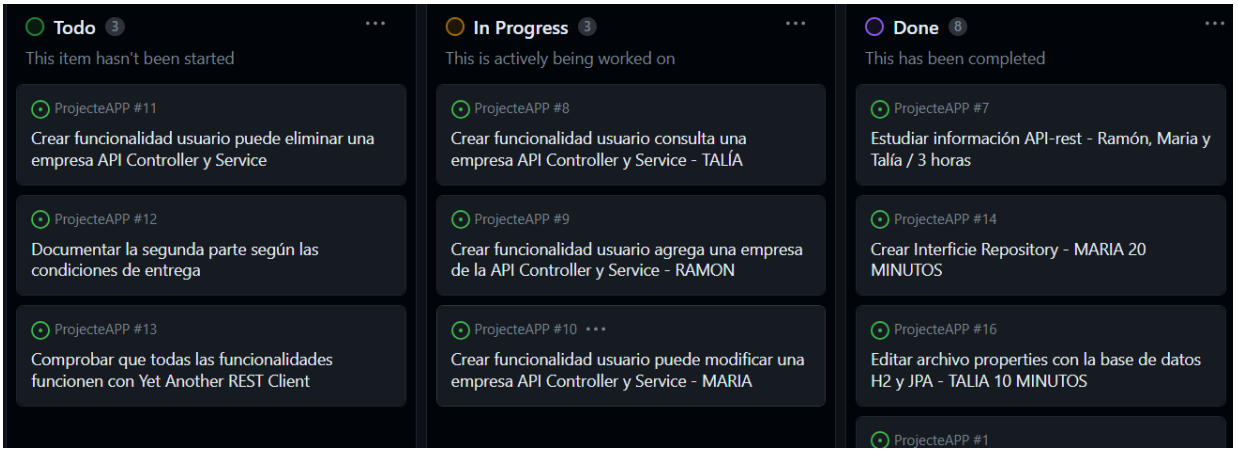
Segundo día:

- Seguimos recopilando información y comenzamos a programar las funcionalidades descritas.



Tercer día:

- Continuamos con las antiguas funcionalidades y agregamos detalles en KANBAN que acabamos el mismo día como crear la clase REPOSITORY y editar el archivo Properties. Lo del día anterior no se acabó porque agregamos el SERVICE al proyecto, que era lo que nos faltaba.

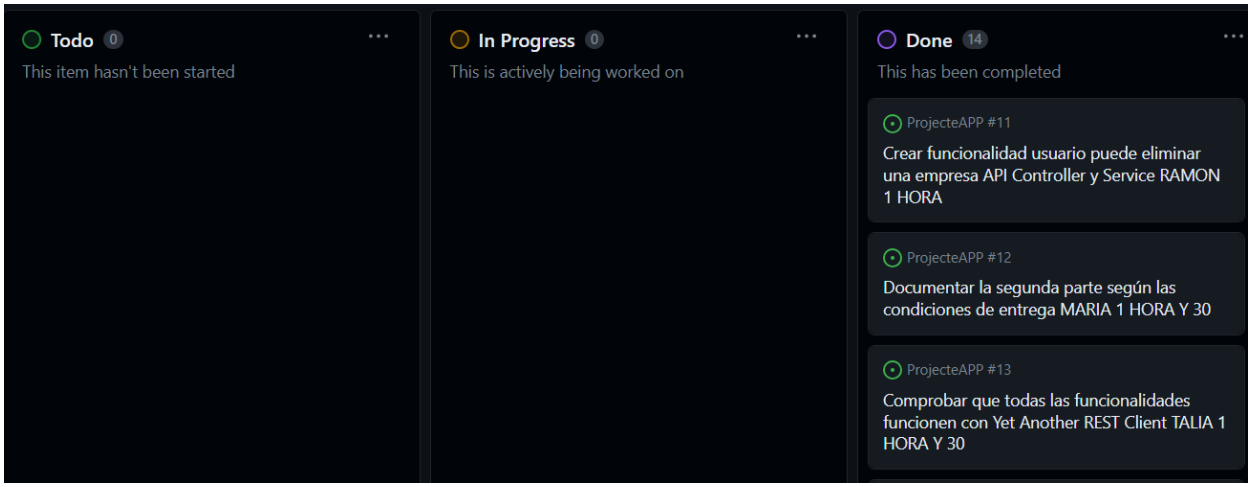


Cuarto día:

- Se finalizan las tareas asignadas y se asignan las últimas, teniendo en cuenta las pautas de documentación, además se comienzan a hacer las pruebas con Yet Another REST Client.

Quinto día:

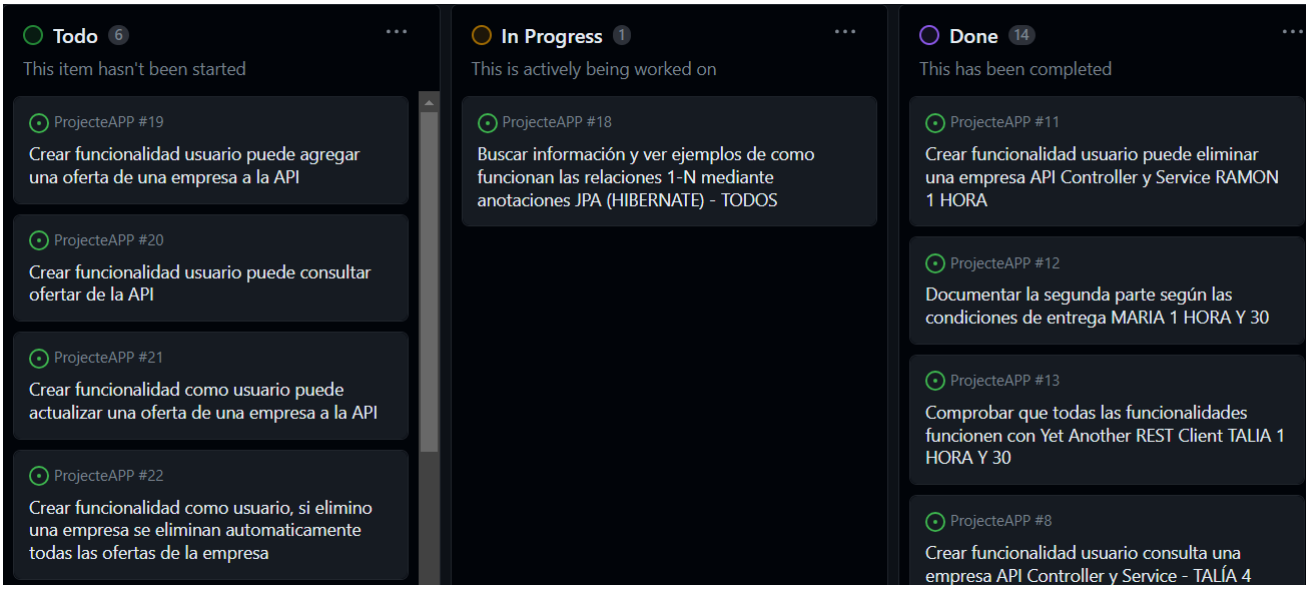
- Se termina lo pendiente y finalmente se realizaron las pruebas finales con POSTMAN, las cuales tuvieron una larga duración por un problema en la ejecución con la jerarquía de los packages del proyecto.



6.3 Tercer Sprint

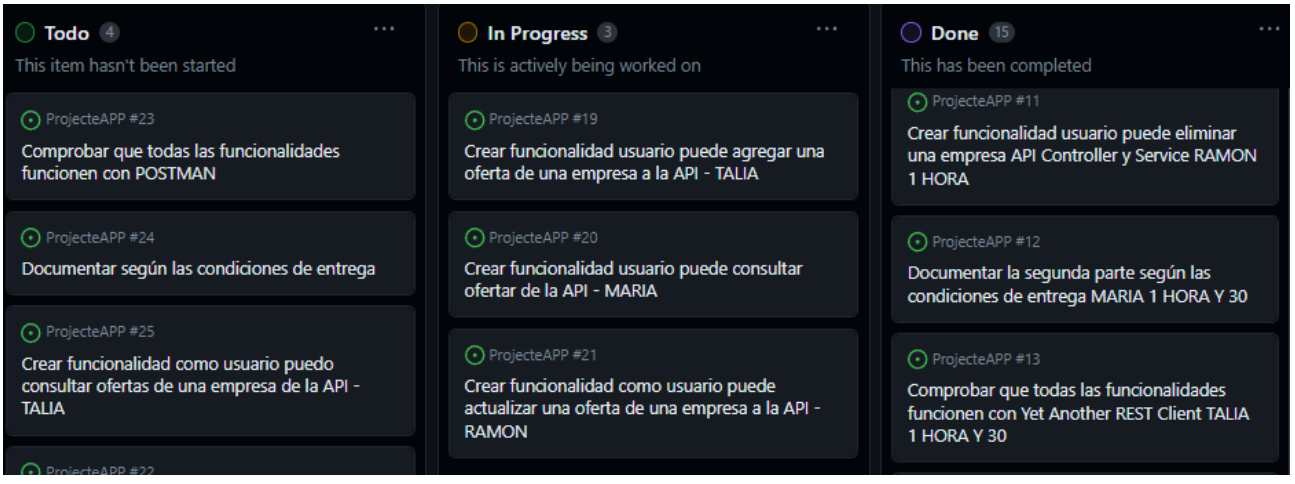
Primer y segundo día / Backlog organizativo:

- Comenzamos a asignar las tareas que vamos a desarrollar a lo largo de la semana y comenzamos con la principal para desarrollar el resto, que se basa en el autoaprendizaje, en este proceso empleamos los dos primeros días.



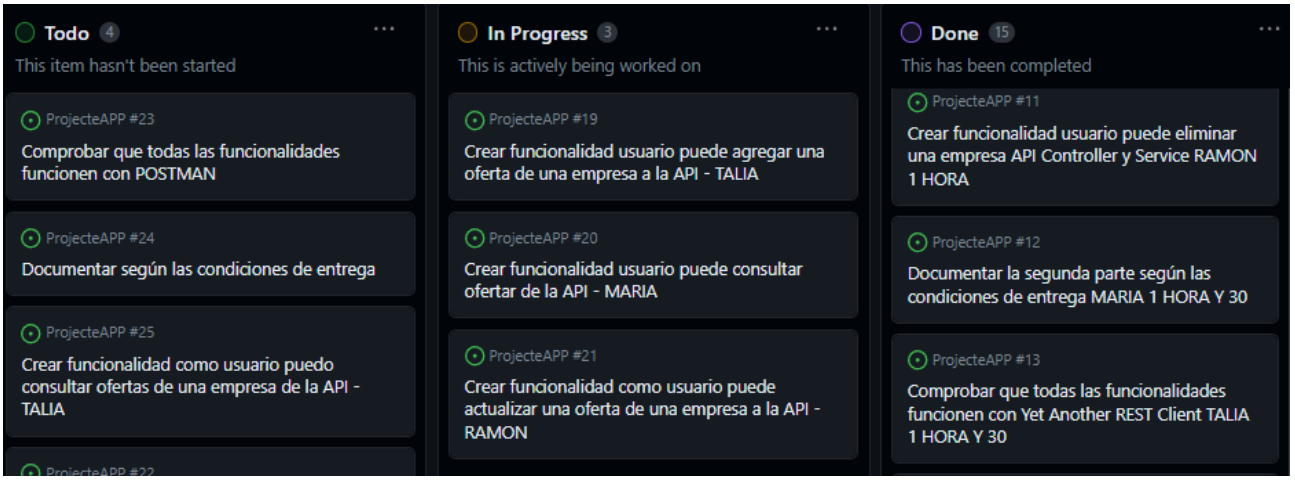
Tercer dia:

- Nos dedicamos a desarrollar el funcionamiento del programa, centrándonos en la manera en que los usuarios podrían tanto agregar como consultar ofertas a través de la API.



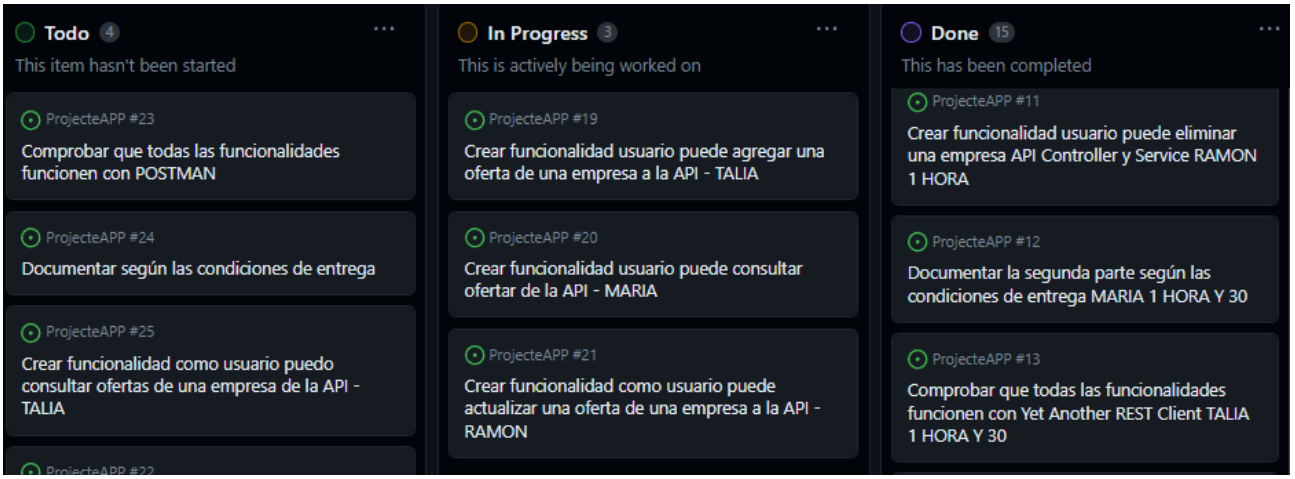
Cuarto dia:

- Continuamos con la implementación de funciones, siguiendo una variedad de ejemplos y tutoriales que nos sirvieron de guía para iniciar el proceso de creación y definición de los distintos métodos.



Quinto dia:

- Concluimos la implementación de las funciones para añadir y consultar ofertas, asegurándonos de su correcto funcionamiento mediante la herramienta Postman.

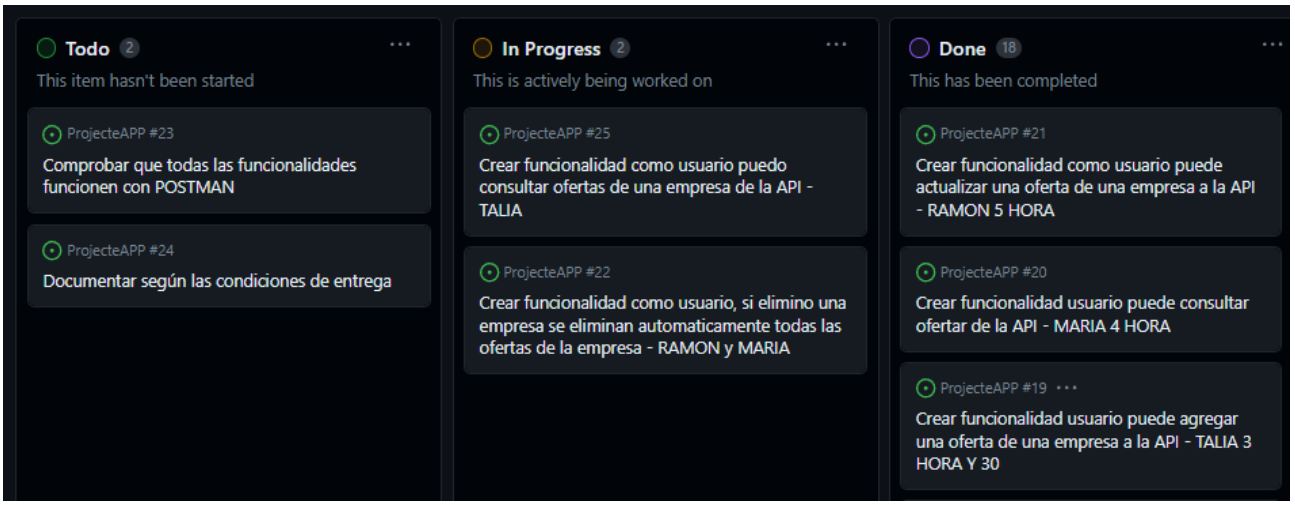


Sexto dia:

- Hemos completado la programación de la función que permite a los usuarios actualizar distintas ofertas, nos adentramos en la fase inicial de la implementación de nuevas funciones, como la búsqueda de ofertas específicas de empresas y la capacidad de los usuarios para eliminarlas.

Septimo dia:

- Persistimos en el desarrollo del código iniciado el día anterior, para mejorar y perfeccionar su funcionalidad.

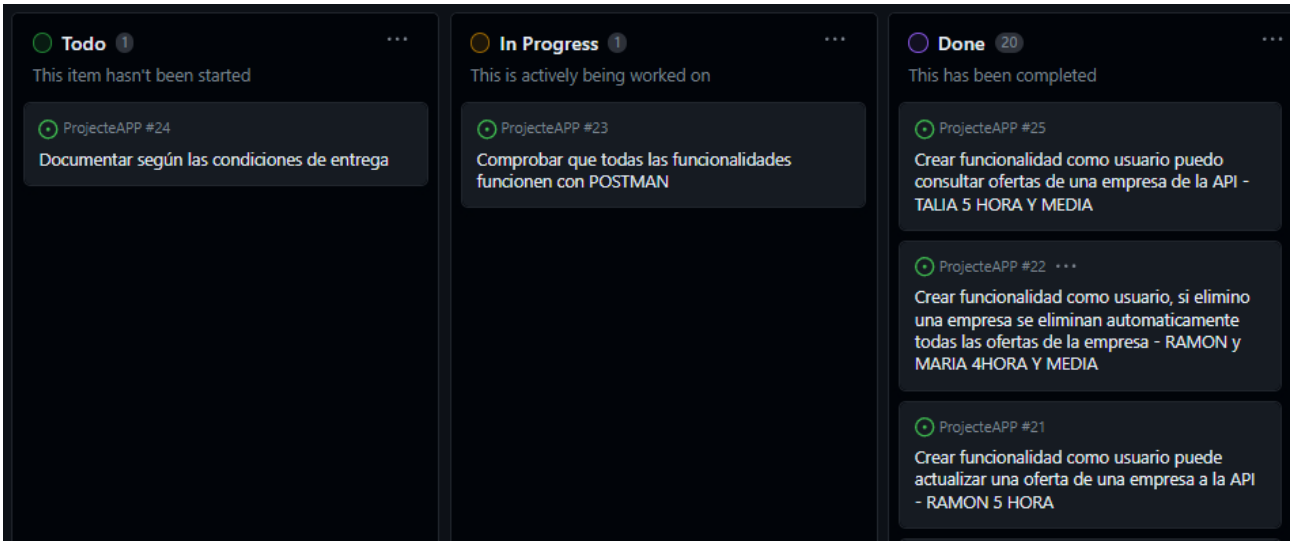


Octavo dia :

- Hemos finalizado el desarrollo del código iniciado previamente y hemos dado inicio a las pruebas para verificar y asegurar su correcto funcionamiento.

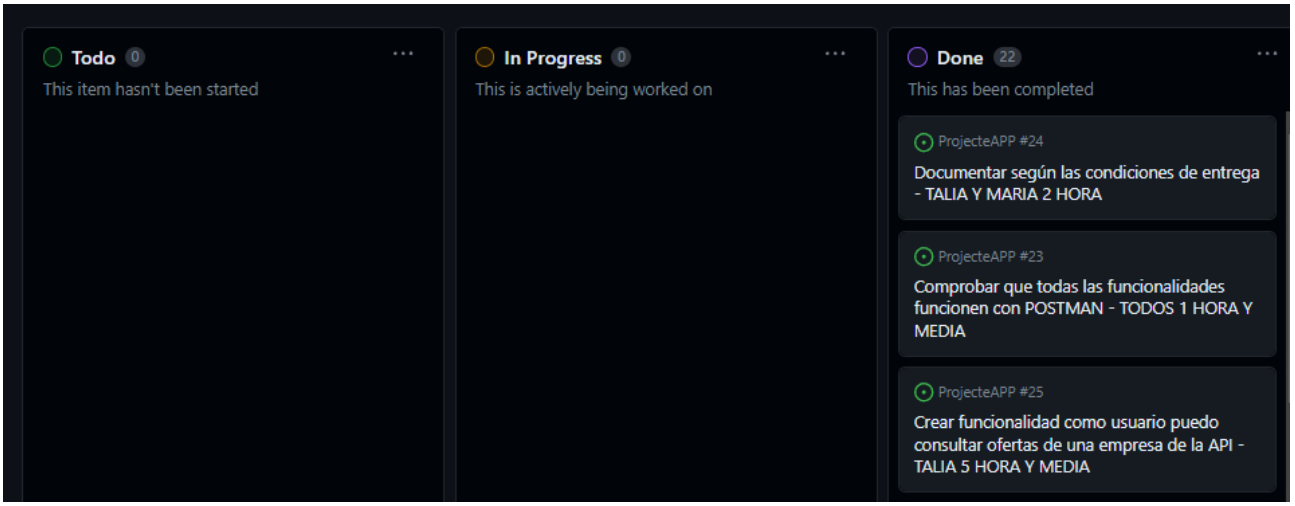
Noveno dia:

- Hemos concluido las pruebas con Postman y verificado que cada una de las funciones operan correctamente.



Decimo dia:

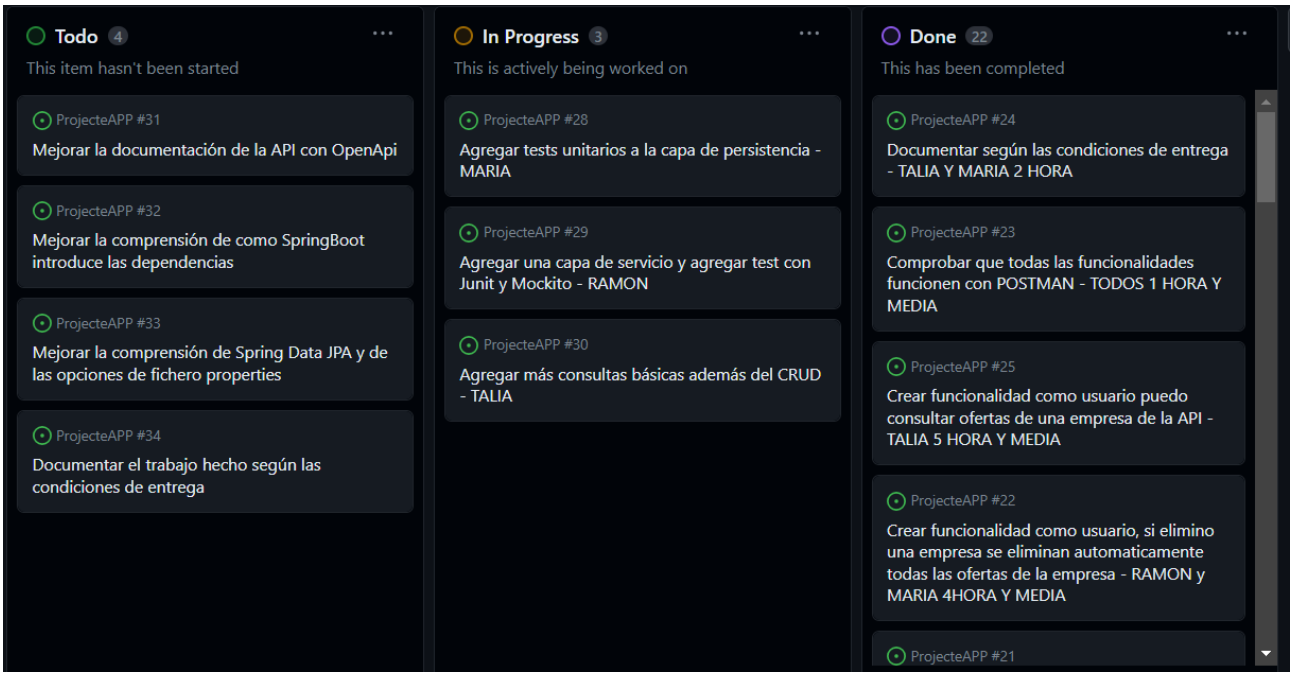
- En el último día, repasamos todo el proyecto y nos aseguramos de tener una documentación completa. Así nos aseguramos de que todo esté en orden y cumpla las condiciones de entrega.



6.4 Cuarto Sprint

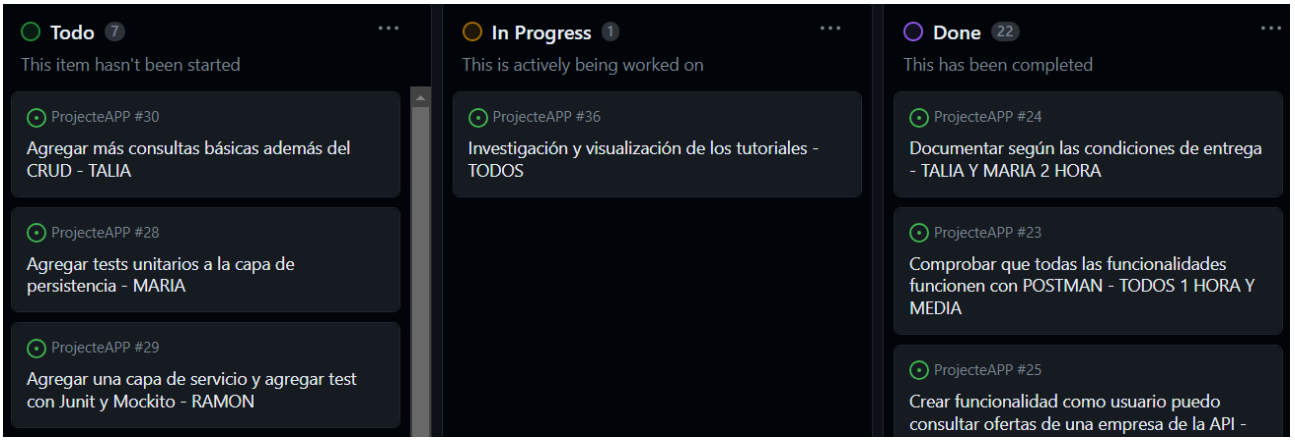
Primer día:

- Se crearon las tareas pendientes y se asignaron a cada miembro.



Segundo, tercero y cuarto día:

- Continuamos con las mismas funciones ya que estamos llevando a cabo cursos de OpenWebinars y algunos tuvimos complicaciones a la hora de acceder ya que el usuario tenía que ser aprobado por el profesor correspondiente.



Quinto día :

- Nos enfocamos en planificar detalladamente como funciona nuestro programas y de las diferentes partes el cual sería formado.

Sexto y séptimo día :

- Una vez terminada la planificación , iniciamos la fase de desarrollo siguiendo los tutoriales paso a paso.

Octavo y noveno día :

- Nos enfocamos en completar los tests y las consultas básicas. Revisamos todo para asegurarnos de que funcionara correctamente

Décimo y undécimo día:

- Nos centramos en profundizar en la comprensión de cómo Spring Boot introduce dependencias, mejorar nuestro conocimiento de Spring Data JPA y explorar las opciones de los archivos de propiedades. También dedicamos tiempo a mejorar la documentación de la API mediante OpenAPI.

Duodécimo día:

- En el último día, documentamos nuestro trabajo, asegurándonos de cumplir con todas las condiciones de entrega y finalizando la documentación de la API según lo requerido.