GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

CIP FP BATOI

C/ Societat Unió Musical, 8 – 030802 Alcoi
Tel: 966527660 – Fax: 966527661
e-mail: 03012165.secret@gva.es
www.cipfpbatoi.es

Unió Europea
Fons Social Europeu
L'FSE inverteix en el teu futur

Desarrollo de una aplicación fantasy en Kotlin

Proyecto final de ciclo formativo superior "Desarrollo de aplicaciones multiplataforma"

Adrián Ruiz Mira

Tutor del proyecto: Sergio Galisteo Castro

Centro: CIP FP Batoi

Contenido

Introducción

Como muchos alumnos, nos enfrentamos al proyecto final sin tener claro qué queremos hacer, y yo, con lo indeciso que soy, no iba a ser una excepción. Este proyecto no fué mi idea inicial, ni se me había planteado en un primer momento. Decidí esperar a ver qué me encontraba en la empresa donde iba a realizar mis prácticas, pero al tratarse del desarrollo de una aplicación, por confidencialidad, no podía realizar un proyecto que estuviera vinculado a esta. Allí he estado desarrollando Android en Kotlin, y yo con este lenguaje no había desarrollado nunca, por lo que decidí enfocar mi proyecto en acelerar mi aprendizaje en este. Desarrollando tanto en las prácticas como en el proyecto, todo iba a ir de la mano.

En cuanto a la idea, una aplicación fantasy de Fórmula 1, quise incentivar mi motivación con uno de mis hobbies. Para aquellos que no sepáis qué es un fantasy, es una aplicación de entretenimiento donde eliges a tus jugadores de una competición y, dependiendo de su rendimiento en la vida real, obtienes más o menos puntos. En este caso, un usuario elegirá sus pilotos al crear o unirse a una liga y competirá con los demás usuarios de esta.

Además, la aplicación tendrá otros dos apartados: uno donde se podrá acceder para ver la información del próximo gran premio y otro donde se listarán todos los grandes premios del campeonato para poder ver su correspondiente información y resultados, si están disponibles.

Descripción general

El proyecto se ha desarrollado utilizando una arquitectura basada en microservicios, con una API REST implementada en Spring Boot y una aplicación móvil desarrollada en Kotlin para Android. La infraestructura está desplegada en AWS.

Herramientas y tecnologías utilizadas

AWS (Amazon Web Services)

- **EC2**: Para la creación y gestión de la máquina virtual.

Servidor MySQL

- **MySQL**: Servidor de gestión de bases de datos relacional utilizado para almacenar los datos de la aplicación.

Spring Boot

- **Spring Boot**: Framework principal para la creación de la API.
- **JPA (Java Persistence API)**: Para la gestión de la persistencia de datos.
- **Hibernate**: Implementación de JPA para ORM.

Android Studio y Kotlin

- **Android Studio**: IDE utilizado para el desarrollo de la aplicación móvil.
- **Kotlin**: Lenguaje de programación utilizado para el desarrollo de la aplicación Android.

API Externa

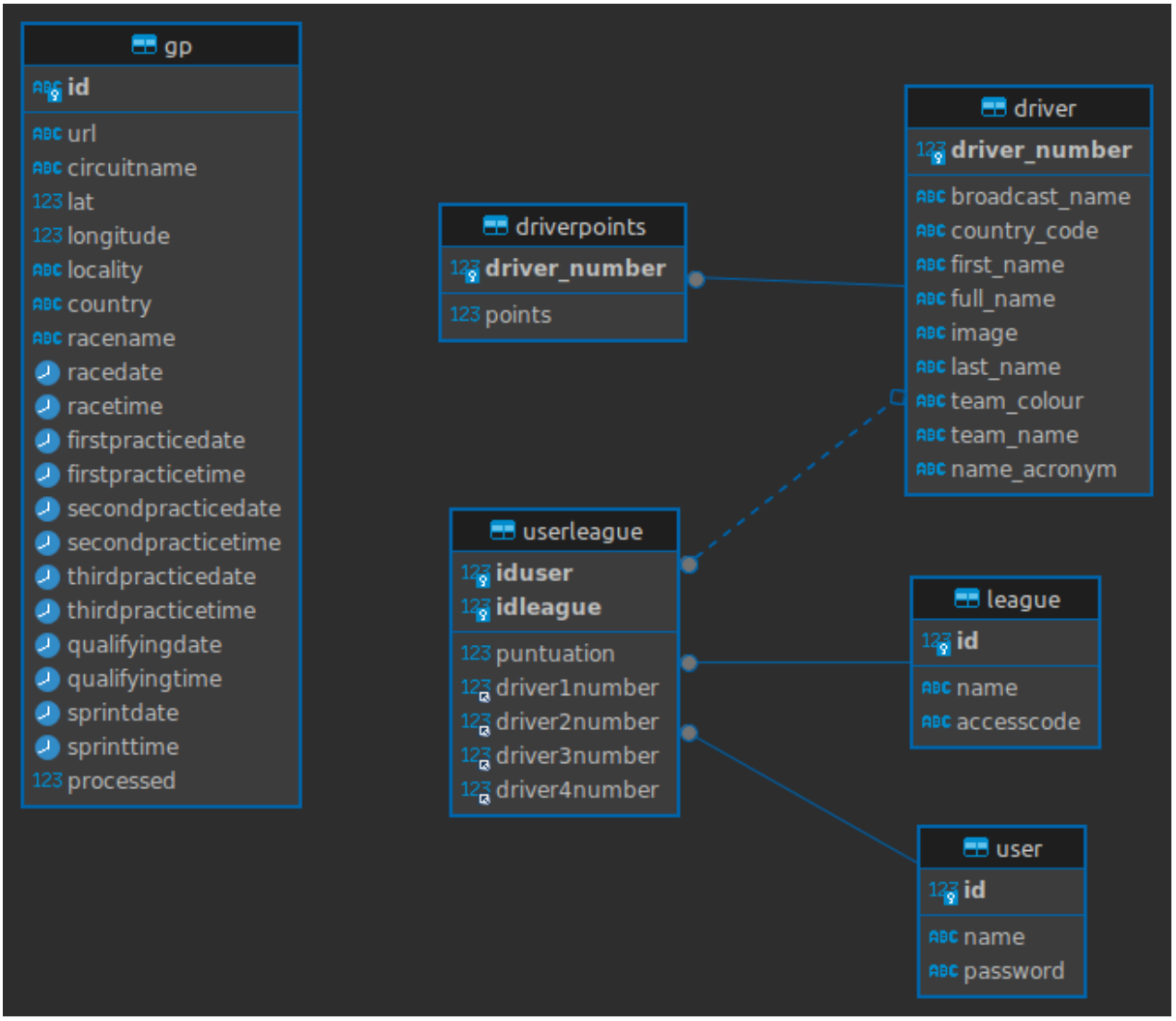
- **OpenF1 API**: API externa utilizada para obtener datos de la competición de Fórmula 1.

Otras Herramientas y Librerías

- **Retrofit**: Para realizar llamadas a la API desde la aplicación Android.
- **Moshi**: Para la conversión de JSON a objetos Kotlin.
- **Dagger Hilt**: Para la inyección de dependencias en la aplicación Android.

Base de datos

Esquema entidad - relación



Entidades

- **gp**: almacena los datos de los grandes premios.
- **driver**: almacena los datos de los pilotos.
- **league**: almacena los datos de las ligas creadas.
- **user**: almacena los datos de los usuarios registrados.
- **userleague**: almacena la información de los usuarios registrados en ligas.
- **driverpoints**: almacena los puntos obtenidos por cada piloto.

Triggers

- **generate_access_code**

  Al insertar una nueva liga, el código de acceso se generará a través de un randomizador de números y letras.

```
BEFORE INSERT ON `league` FOR EACH ROW BEGIN
    SET NEW.accesscode= SUBSTRING(MD5(RAND()) FROM 1 FOR 8);
END
```

- **update_userleague_points**

  Al finalizar la carrera de un gran premio, la puntuación obtenida por cada piloto en relación a su posición final, se calculará y se insertará desde el api a la tabla driverpoints, cuando eso suceda, la nueva puntuación de cada piloto cuyo número (Pk) corresponda con el número de piloto de una de las columnas driverxnumber (Fk) de la tabla userleague, se sumará a la columna puntuation de esta.

```
AFTER UPDATE ON `driverpoints` FOR EACH ROW BEGIN
    UPDATE userleague SET
        puntuation = ABS(NEW.points - OLD.points) + puntuation
    WHERE
        driver1number = OLD.driver_number OR
        driver2number = OLD.driver_number OR
        driver3number = OLD.driver_number OR
        driver4number = OLD.driver_number;
END
```

Procedures

- **getnextgp()**

  Este procedimiento devolverá el gp cuya fecha de carrera sea más próxima o igual a la fecha actual del sistema.

```
PROCEDURE `fx3`.`getnextgp`()
BEGIN
    SELECT *
    FROM gp
    WHERE racedate >= CURRENT_DATE()
    ORDER BY racedate ASC
    LIMIT 1;
END
```

Desarrollo del Modelo de Datos con JPA y Hibernate

Introducción a Hibernate

Hibernate es un framework de mapeo objeto-relacional (ORM) para Java que facilita la persistencia de datos en bases de datos relacionales. Utiliza Java Persistence API (JPA) para definir cómo las clases de Java se relacionan con las tablas de la base de datos. Hibernate gestiona automáticamente las operaciones de CRUD (crear, leer, actualizar y eliminar), permitiendo poner el foco en la lógica de negocio.

Configuración de Hibernate

La configuración de Hibernate se realizó mediante archivos XML, definiendo las propiedades de conexión a la base de datos y las entidades del modelo.

Archivo de configuración hibernate.cfg.xml

El archivo `hibernate.cfg.xml` se utiliza para configurar las propiedades de conexión a la base de datos y especificar las clases mapeadas.

```
   -//Hibernate/Hibernate Configuration DTD 3.0//EN (doctype with catalog)
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!DOCTYPE hibernate-configuration PUBLIC
 3         "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 4         "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
 5 <hibernate-configuration>
 6     <session-factory>
 7         <property name="hibernate.connection.driver_class">org.mariadb.jdbc.Driver</property>
 8         <property name="hibernate.connection.password">xxxxxx</property>
 9         <property name="hibernate.connection.url">jdbc:mariadb://192.168.56.101:3306/fx3</property>
10         <property name="hibernate.connection.username">xxxxxx</property>
11         <property name="hibernate.dialect">org.hibernate.dialect.MariaDBDialect</property>
12         <property name="hibernate.show_sql">false</property>
13         <property name="hibernate.format_sql">false</property>
14 <!--         Si se hace con anotaciones... -->
15 <!--         <mapping class="modelo.Driver"></mapping>-->
16 <!--         <mapping class="modelo.Gp"></mapping>-->
17 <!--         <mapping class="modelo.League"></mapping>-->
18 <!--         <mapping class="modelo.User"></mapping>-->
19 <!--         <mapping class="modelo.Userleague"></mapping>-->
20 <!--         <mapping class="modelo.Userleague"></mapping>-->
21 <!--         <mapping class="modelo.Driverpoints"></mapping>-->
22     </session-factory>
23 </hibernate-configuration>
```

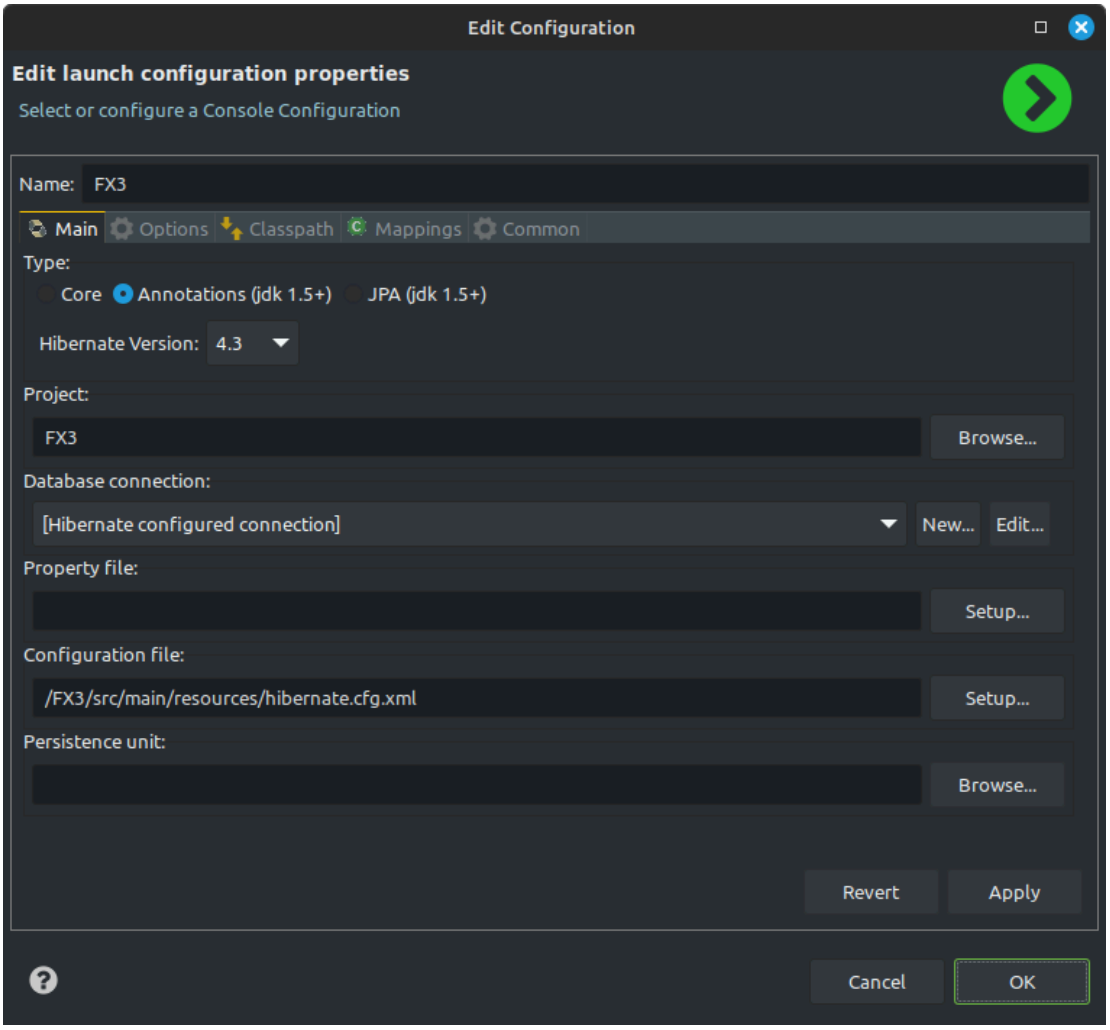
Archivo de ingeniería inversa hibernate.reveng.xml

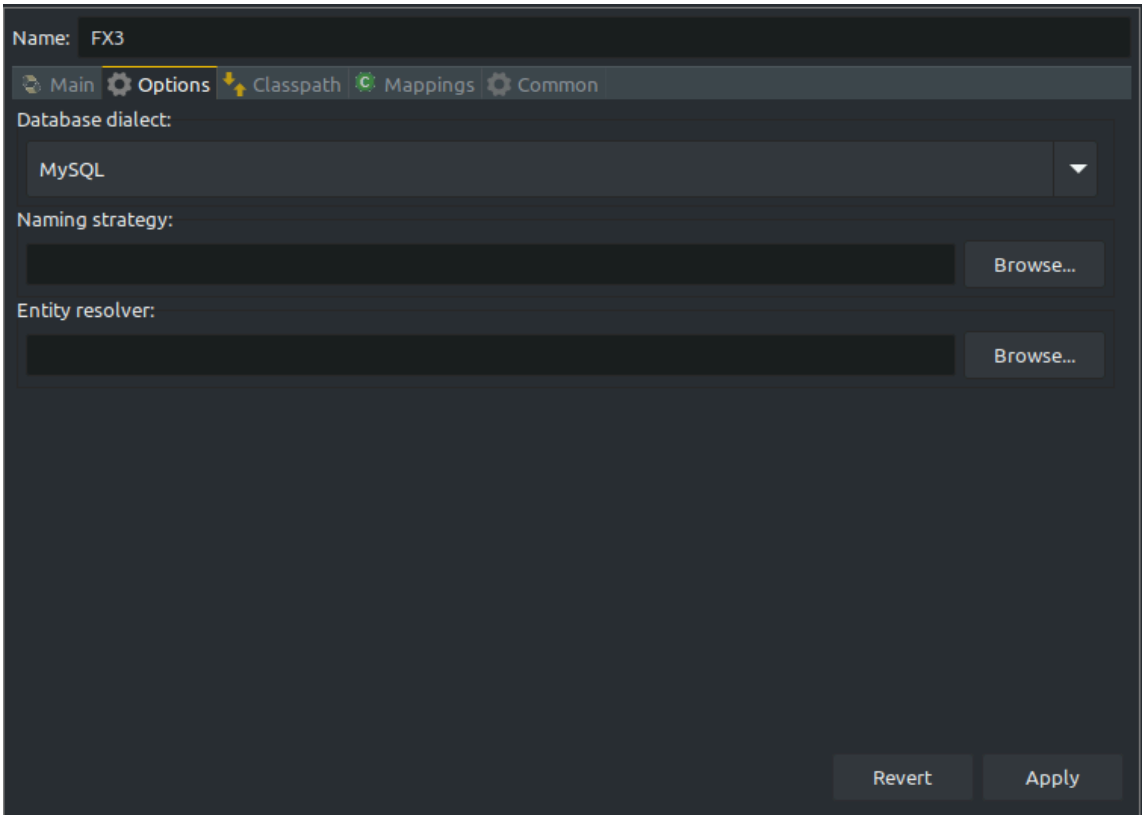
El archivo `hibernate.reveng.xml` se utiliza para definir la ingeniería inversa, qué es el proceso de generar clases Java a partir de las tablas de la base de datos.

```
   -//Hibernate/Hibernate Reverse Engineering DTD 3.0//EN (doctype with catalog)
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse
3
4  <hibernate-reverse-engineering>
5     <table-filter match-catalog="fx3" match-name="gp"/>
6     <table-filter match-catalog="fx3" match-name="driver"/>
7     <table-filter match-catalog="fx3" match-name="driverpoints"/>
8     <table-filter match-catalog="fx3" match-name="user"/>
9     <table-filter match-catalog="fx3" match-name="userleague"/>
10    <table-filter match-catalog="fx3" match-name="league"/>
11 </hibernate-reverse-engineering>
```

Creación de Hibernate configuration

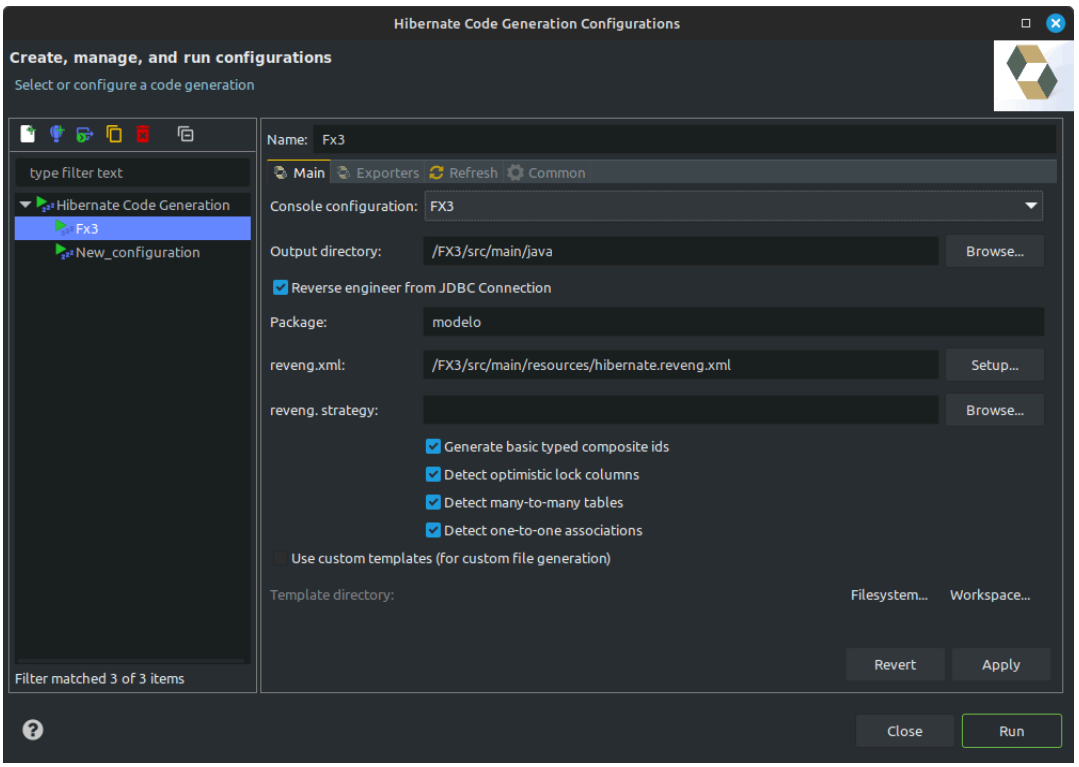
Aquí indicamos el archivo `hibernate.cfg.xml` mostrado anteriormente, y realizamos las configuraciones pertinentes para la creación del modelo y que se pueda llevar a cabo el mapeo.

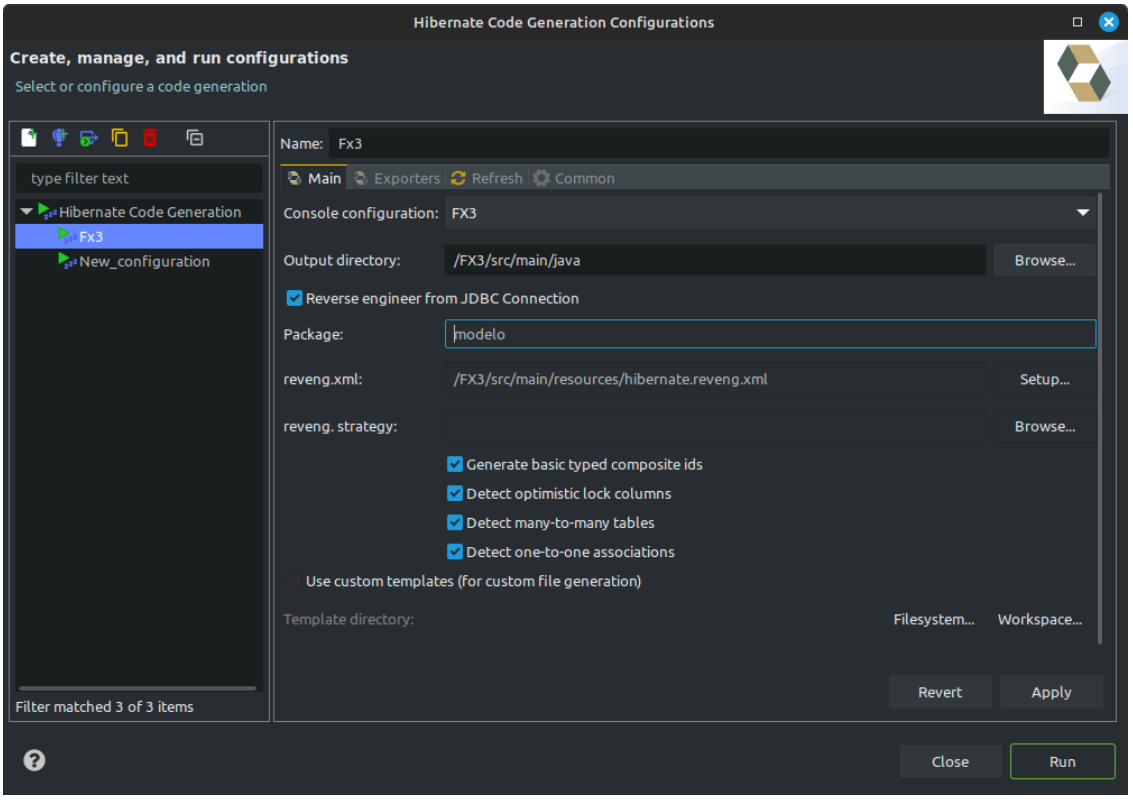




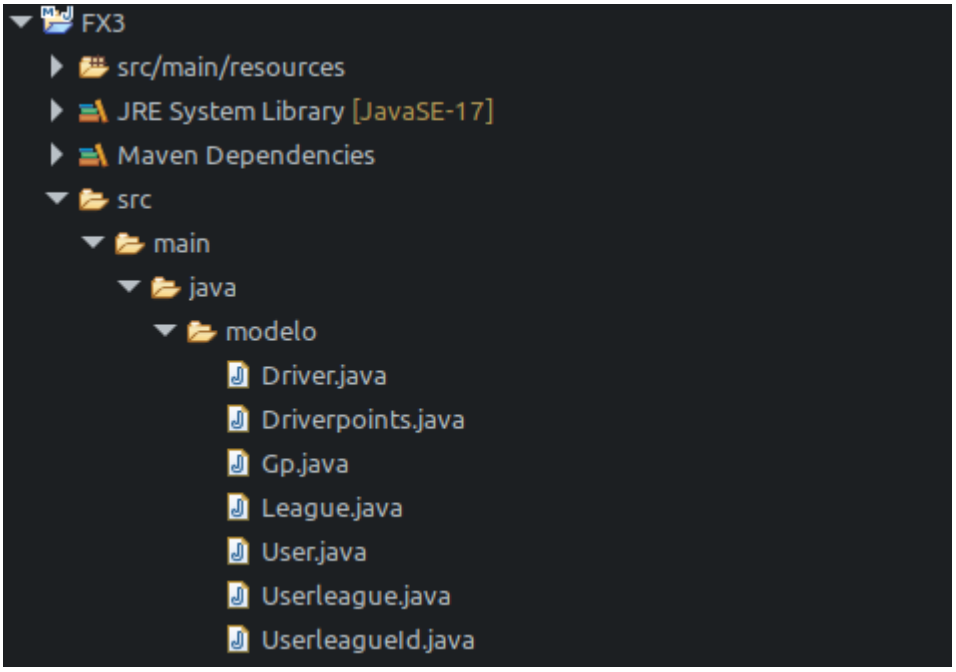
Hibernate code generation configurations

Llegados a este punto, solo queda realizar la ingeniería inversa para crear nuestro modelo con las entidades de la base de datos, para ello requerimos de esta herramienta hibernate. Aquí se debe indicar el fichero `hibernate.reveng.xml` mostrado anteriormente.





Una vez hecho esto, se clica en Run y se generará el modelo:



Uso de Lombok para Simplificar el Código

Para reducir la cantidad de código se ha utilizado la librería Lombok. Lombok permite generar automáticamente getters, setters, constructores y los métodos toString, equals y hasCode gracias a la anotación @Data.

Ejemplo de Clase Generada con Lombok

```java
package dam.fx3.modelo.entities;
// Generated 3 may 2024 1:08:40 by Hibernate Tools 4.3.6.Final



import java.io.Serial;

/**
 * User generated by hbm2java
 */
@Data
@Entity
@Table(name = "user", catalog = "fx3")
public class User implements java.io.Serializable {

    @Serial
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "password", nullable = false, length = 64)
    private String password;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "user")
    private List<Userleague> userleagues = new ArrayList<Userleague>();

}

```

Desarrollo de la API con Spring Boot

Introducción a Spring framework

Java Spring Framework (Spring Framework) es un popular entorno de trabajo empresarial de código abierto que sirve para crear aplicaciones autónomas de producción que se ejecutan en una máquina virtual Java (JVM).

Java Spring Boot (Spring Boot) es una herramienta que acelera y simplifica el desarrollo de microservicios y aplicaciones web con Spring Framework gracias a tres funciones principales:

- **Configuración automática:** las aplicaciones se inicializan con dependencias predefinidas que no es necesario configurar manualmente.
- **Un enfoque de configuración obstinado:** Spring Boot elige qué paquetes instalar y qué valores predeterminados utilizar, en lugar de que sea el usuario quien deba tomar esas decisiones y configurarlo todo manualmente.
- **La capacidad de crear aplicaciones autónomas:** permite crear aplicaciones autónomas que se ejecutan por sí solas, sin depender de un servidor web externo, integrando un servidor web como Tomcat o Netty en la app durante el proceso de inicialización.

Configuración del proyecto

Para configurar el proyecto Spring Boot, se utilizan las siguientes dependencias en el archivo `pom.xml`:

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
   </dependency>
   <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <scope>runtime</scope>
   </dependency>
   <dependency>
      <groupId>org.mariadb.jdbc</groupId>
      <artifactId>mariadb-java-client</artifactId>
      <scope>runtime</scope>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
```

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.1.Final</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Configuración del archivo application.properties

El archivo `application.properties` contiene las configuraciones específicas de la aplicación, tales como las propiedades de conexión a la base de datos y los niveles de logging.

```
spring.application.name=APiFx3
spring.datasource.url=jdbc:mysql://localhost:3306/fx3
spring.datasource.username=xxxxx
spring.datasource.password=xxxxx
server.port=8085
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
logging.level.dam.fx3.schedules.RaceChecker=INFO
```

Detalle sobre la configuración de logging

La configuración de logging es muy útil para el monitoreo y depuración de la aplicación. En el archivo application.properties, se ha definido un nivel de logging específico para la clase RaceChecker y de esta forma garantizar que se registren mensajes.

Repositories

Los repositorios son interfaces que extienden JpaRepository.

JpaRepository extiende de PagingAndSortingRepository el cuál extiende de CrudRepository, por lo que con el repositorio Jpa obtenemos tanto los métodos CRUD cómo también para paginación y ordenación.

Ejemplo de repository:

```
package dam.fx3.api.repository;
import dam.fx3.modelo.entities.Driverpoints;
import org.springframework.data.jpa.repository.JpaRepository;
public interface DriverPointsRepository extends JpaRepository<Driverpoints, Integer> {
}
```

Además, es posible definir consultas más complejas y llamar a procedimientos directamente desde el repositorio:

En este ejemplo, el método findByAccesscode utiliza la anotación @Query para definir una consulta SQL nativa que busca una liga por su código de acceso.

```
public interface LeagueRepository extends JpaRepository<League, Integer> {

   @Query(value = "select * from league l where l.accesscode = ?", nativeQuery = true)
   Optional<League> findByAccesscode(String accesscode);
}
```

En este ejemplo, el método getNextGp llama a un procedimiento almacenado definido en la base de datos, utilizando la anotación @Procedure.

```
public interface GpRepository extends JpaRepository<Gp, Integer> {
   @Procedure(name = "getnextgp")
   Optional<Gp> getNextGp();
}
```

Repositorios para la API Externa

Para obtener los resultados de las carreras y comprobar si una carrera ha finalizado, se realiza la integración con una API externa, para ello he creado un repository adicional.

```
package dam.fx3.api.repository;
import java.util.Map;

public interface ExternApiInterface {
   Map<Integer, Integer> getPositionMap(String date, int sessionType) throws Exception;
   boolean isRaceFinished(String date) throws Exception;
}
```

```
@Repository
public class ExternApiRepository implements ExternApiInterface{
   private static final String POSITION_URL = "https://api.openf1.org/v1/position?date=";
         private static final String CHEQUERED_URL =
"https://api.openf1.org/v1/race_control?flag=CHEQUERED&date=";

   public ExternApiRepository() {
   }
```

makeApiCall: realiza una llamada HTTP GET a una URL dada y devuelve la respuesta como una cadena. Este método se utiliza para interactuar con la API externa.

```
   private String makeApiCall(String apiUrl) throws Exception {
       HttpClient httpClient = HttpClient.newHttpClient();
       HttpRequest request = HttpRequest.newBuilder()
               .uri(URI.create(apiUrl))
               .GET()
               .build();
       HttpResponse<String> response =
              httpClient.send(request,HttpResponse.BodyHandlers.ofString());
       return response.body();
   }
```

getPositionToQualifying: obtiene las posiciones de clasificación para una fecha específica. Realiza dos llamadas a la API externa: una para obtener la clave de la sesión de clasificación y otra para obtener las posiciones basadas en esa clave .

```
   public Map<Integer, Integer> getPositionToQualifying(String date)throws Exception {
       String jsonResponse =
makeApiCall("https://api.openf1.org/v1/sessions?session_type=Qualifying&date_start="+date+"&da
te_end="+date);
       List<Map<String, Object>> dataList = parseJsonToList(jsonResponse);
       int session_key = 0;
       for (Map<String, Object> entry : dataList) {
            session_key = (int) entry.get("session_key");
            break;
       }
       jsonResponse =
        makeApiCall(
       "https://api.openf1.org/v1/position?session_key="+session_key
       );
       return filterAndRetrieveLatestPositions(parseJsonToList(jsonResponse));
   }
```

getPositionMap: obtiene un mapa de posiciones para una fecha y tipo de sesión específicos. Llama a la API externa para obtener los datos y luego los filtra y procesa.

```
   public Map<Integer, Integer> getPositionMap(String date, int sessionType)throws
Exception {
           String jsonResponse = makeApiCall(POSITION_URL + date);
           List<Map<String, Object>> dataList = parseJsonToList(jsonResponse, sessionType);
           return filterAndRetrieveLatestPositions(dataList);
   }
```

isRaceFinished: verifica si una carrera ha terminado en una fecha específica. Realiza una llamada a la API externa para obtener el estado de la carrera y comprueba si se ha mostrado la bandera a cuadros.

```
   public boolean isRaceFinished(String date) throws Exception {
       String apiUrl =  CHEQUERED_URL + date;
       String jsonResponse = makeApiCall(apiUrl);
       List<Map<String, Object>> dataList = parseJsonToList(jsonResponse);

       if (dataList.isEmpty()) {
           return false;
       }

       for (Map<String, Object> entry : dataList) {
           String flag = (String) entry.get("flag");
           if ("CHEQUERED".equals(flag)) {
               return true;
           }
       }

       return false;
   }
```

parseJsonToList(2 argumentos): convierte una cadena JSON en una lista de mapas y filtra la lista según el tipo de sesión.

```
private List<Map<String, Object>> parseJsonToList(String jsonResponse, int sessionType) throws
Exception {
```

```
        ObjectMapper mapper = new ObjectMapper();
        List<Map<String, Object>> dataList = mapper.readValue(jsonResponse, new
TypeReference<List<Map<String, Object>>>(){});


        if (sessionType == 0) {
            return dataList;
        } else {
            List<Map<String, Object>> filteredList = new ArrayList<>();
            Map<String, Object> firstMap = dataList.get(0);
            int firstSessionKey = (int) firstMap.get("session_key");
            for (Map<String, Object> entry : dataList) {
                int currentSessionKey = (int) entry.get("session_key");
                if (sessionType == 1 && currentSessionKey == firstSessionKey) {
                    filteredList.add(entry);
                } else if (sessionType == 2 && currentSessionKey != firstSessionKey) {
                    filteredList.add(entry);
                }
            }
            return filteredList;
        }
    }
```

parseJsonToList(1 argumento): convierte una cadena JSON en una lista de mapas sin filtrar los datos.

```
    private List<Map<String, Object>> parseJsonToList(String jsonResponse) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        return mapper.readValue(jsonResponse, new TypeReference<List<Map<String,
Object>>>(){});
    }
```

filterAndRetrieveLatestPositions: Filtra y ordena las posiciones de los pilotos.

```
    private Map<Integer, Integer> filterAndRetrieveLatestPositions(List<Map<String, Object>>
dataList) {
        Map<Integer, Integer> positionMap = new HashMap<>();
        for (Map<String, Object> entry : dataList) {
            int driverNumber = (int) entry.get("driver_number");
            int position = (int) entry.get("position");
            positionMap.put(driverNumber, position);
        }
        positionMap = positionMap.entrySet()
                .stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,
                        (oldValue, newValue) -> oldValue, LinkedHashMap::new));

        return positionMap;
    }
}
```

Data Transfer Objects (DTOs)

Los DTOs (Data Transfer Objects) se utilizan para transportar datos entre las diferentes capas de la aplicación. Son objetos que contienen únicamente los datos necesarios para la transferencia.

Ejemplo de DTO

```
import lombok.Data;

import java.io.Serial;
import java.io.Serializable;

@Data
public class DriverDTO implements Serializable {

   @Serial
   private static final long serialVersionUID = 1L;

   private int driverNumber;
   private String countryCode;
   private String image;
   private String lastName;
   private String teamColour;
   private String teamName;
   private String nameAcronym;
}
```

Services

Los services contienen la lógica de negocio, se encuentran entre los controllers y repositories y actúan de intermediarios entre estos.

Ejemplo de service

```
package dam.fx3.service;

import dam.fx3.api.repository.LeagueRepository;
import dam.fx3.modelo.dto.LeagueDTO;
import dam.fx3.modelo.entities.League;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
@Transactional
public class LeagueService {

   @Autowired
   private LeagueRepository leagueRepository;

   public LeagueService(){
   }

   public LeagueDTO findByAccesscode(String accesscode) {
       return leagueRepository
```

```
                                .findByAccesscode(accesscode)
                                .map(this::mapLeagueToDTO)
                                .orElse(null);
    }

    public List<LeagueDTO> findAll(){
        return leagueRepository
                                .findAll()
                                .stream()
                                .map(this::mapLeagueToDTO)
                                .toList();
    }

    public LeagueDTO findById(int id) {
        return leagueRepository
                                .findById(id)
                                .map(this::mapLeagueToDTO)
                                .orElse(null);
    }


    public List<LeagueDTO> listAllLeagues() {
        return leagueRepository
                                .findAll()
                                .stream()
                                .map(this::mapLeagueToDTO)
                                .toList();
    }

    public LeagueDTO save(LeagueDTO leagueDTO) {
       League league = leagueRepository.save(mapLeagueDTOToEntity(leagueDTO));

       return mapLeagueToDTO(league);
    }
```

Mapeamos las entities que proporciona el repository a DTO

```
    private League mapLeagueDTOToEntity(LeagueDTO leagueDTO) {
        League league = new League();
        league.setName(leagueDTO.getName());
        return league;
    }
```

Mapeamos los DTO que proporciona controller a entity

```
    public LeagueDTO mapLeagueToDTO(dam.fx3.modelo.entities.League league) {
        LeagueDTO leagueDTO = new LeagueDTO();
        leagueDTO.setId(league.getId());
        leagueDTO.setName(league.getName());
        leagueDTO.setAccesscode(league.getAccesscode());
        return leagueDTO;
    }
}
```

Controllers

Los controllers gestionan las solicitudes HTTP entrantes.

Ejemplo de controller

```
package dam.fx3.controller;

import dam.fx3.modelo.dto.DriverDTO;
import dam.fx3.service.DriverService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/driver")
public class DriverController {
    @Autowired
    DriverService driverService;

    @GetMapping("")
    public List<DriverDTO> list() {
        return driverService.listAllDrivers();
    }

    @GetMapping("/{number}")
    public ResponseEntity<DriverDTO> get(@PathVariable int number) {
        try {
            DriverDTO driver = driverService.findByNumber(number);
            return ResponseEntity.ok(driver);
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }

    @GetMapping("/user/{idUser}/league/{idLeague}")
 public ResponseEntity<List<DriverDTO>> getByUserId(@PathVariable int idUser, @PathVariable
int idLeague) {
        try {
          List<DriverDTO> driver = driverService.findByUserIdLeagueId(idUser, idLeague);
            return ResponseEntity.ok(driver);
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Schedule

La anotación Schedule permite ejecutar tareas de manera programada. En este caso, se utiliza en la clase RaceChecker para verificar periódicamente si un Gran Premio ha finalizado y procesar los resultados de la carrera.

```
package dam.fx3.schedules;

import dam.fx3.modelo.entities.Gp;
import dam.fx3.service.DriverPointsService;
import dam.fx3.service.GpService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Map;

@EnableScheduling
@Component
public class RaceChecker{
   private static final Logger logger = LoggerFactory.getLogger(RaceChecker.class);
   @Autowired
   private GpService gpService;
   @Autowired
   private DriverPointsService driverPointsService;

   public RaceChecker() {

   }


  @Scheduled(fixedRate = 3600000) // Ejecutar cada hora (3600000 milisegundos)
   public void checkRace() {
       try {
           if (actualGPIsFinished()) {
             Map<Integer, Integer> raceResults = gpService.findNextGp().getRaceResults();
             if (!raceResults.isEmpty()){
                 Gp gp = gpService.findById(gpService.findNextGp().getId());
                 if(!gp.getProcessed()){
                     driverPointsService.insertDriverPoints(raceResults);
                     logger.info("Puntos listos: {}", raceResults);
                     gp.setProcessed(true);
                     gpService.save(gp);
                 } else {
                     logger.info("Gp ya procesado");
                 }
             }
             else{
                 logger.info("Puntos no disponibles");
             }
           } else{
               logger.error("El próximo GP no ha terminado");
           }
       } catch (Exception e) {
           logger.error("Error en la ejecución de RaceChecker", e);
           throw new RuntimeException(e);
```

```
        }
    }

    private boolean actualGPIsFinished() {
        try {

            return gpService.actualGPIsFinished();
        } catch (Exception e) {

            throw new RuntimeException(e);
        }
    }
}
```

Desarrollo de la App con Kotlin

En este apartado voy a explicar cómo he desarrollado la aplicación, su arquitectura, y los principios de diseño utilizados. La aplicación se ha diseñado siguiendo el paradigma de programación modular, la arquitectura Clean Architecture y el patrón MVVM.

Programación Modular

La programación modular divide la aplicación en módulos independientes pero interconectados- En el desarrollo de aplicaciones Android con Kotlin, esto se logra creando múltiples módulos Gradle.

Ventajas de la Programación Modular:

- **Mantenibilidad:** Cada módulo es más pequeño y manejable, facilitando el mantenimiento y la actualización del código.
- **Reusabilidad:** Los módulos pueden ser reutilizados en diferentes partes de la aplicación o incluso en diferentes proyectos.
- **Escalabilidad:** Facilita el trabajo en equipos grandes, permitiendo que los desarrolladores trabajen en diferentes módulos de manera independiente.
- **Rendimiento:** Mejora los tiempos de compilación, ya que solo los módulos modificados necesitan ser compilados.

Lista de módulos

```
include ':app'
include ':knowledge'
include ':grandprix'
include ':core'
include ':core:styles'
include ':core:data'
include ':domain'
include ':core:presentation'
include ':fantasy'
```

- **app**: módulo donde se ejecuta el inicio de la aplicación, en este podemos encontrar la pantalla splash, el login y el menú principal.
- **core**: módulo que contiene diferentes módulos, es el encargado de proveer de necesidades de varios módulos, está dividido en módulo **data** (acceso a constantes, datos del inicio de sesión), módulo **presentation** (clases abstractas de activity, fragment, viewmodel) y módulo **styles** (layouts que se reutilicen en varios módulos, strings.xml)
- **domain**: todo lo relacionado con el modelo, las llamadas a la api, repositorios, casos de uso.
- **fantasy**: módulo donde se gestionan las ligas de los usuarios.
- **grandprix**: módulo donde se gestiona lo relacionado con el próximo gran premio.
- **knowledge**: módulo donde se gestiona la información de todos los gran premios de la temporada.

Clean Architecture

La arquitectura limpia, propuesta por Robert C. Martin, busca separar las responsabilidades en diferentes capas y facilitar el mantenimiento y la escalabilidad de la aplicación. En un proyecto Android típico con Kotlin, esta arquitectura se implementa a través de cuatro capas:

- **Entidad:** Modelos de datos que representan los objetos de negocio.
- **Caso de Uso (UseCase):** Contiene la lógica de negocio específica de la aplicación.
- **Repository:** Interfaz que define las operaciones de datos. Los detalles de implementación se delegan a la capa de datos.
- **Controller:** Intermediario entre la lógica de negocio y la interfaz de usuario.

Mi proyecto en concreto, se compone de lo siguiente:

- **ApiService**: Define las llamadas a la API externa. Este componente maneja la comunicación con servicios externos.
- **Repository**: Define las interfaces para las operaciones de datos. Proporciona una abstracción sobre los datos.
- **RepositoryImpl**: Implementación de repository. Contiene la lógica de acceso a datos, interactuando con ApiService y otras fuentes de datos.
- **UseCase**: Contiene la lógica de negocio específica de la aplicación. Los casos de uso orquestan las operaciones de datos a través de los repositorios.
- **ViewModel**: Maneja la lógica de presentación y la comunicación entre la Vista (Fragment) y el UseCase.
- **Fragment**: La interfaz de usuario que observa los datos del ViewModel y presenta la información al usuario.

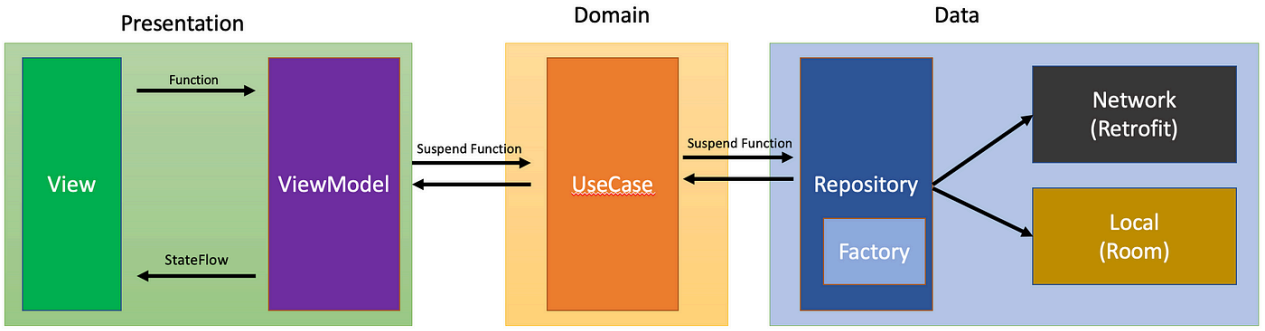
Patrón MVVM (Modelo – Vista – Vista-Modelo)

El patrón MVVM es una arquitectura que facilita la separación de la lógica de presentación de la lógica de negocio. En Android, se implementa usualmente con ViewModels y LiveData.

Componentes de MVVM:

- **Modelo (Model):** Representa los datos y la lógica de negocio.
- **Vista (View):** Representa la interfaz de usuario y muestra los datos del ViewModel.
- **Vista-Modelo (ViewModel):** Interactúa con el modelo para obtener los datos y se comunica con la vista para actualizar la interfaz de usuario.

La combinación de aplicar clean architecture y patrón mvvm podemos verla en la siguiente imágen, donde se observan perfectamente cada capa y el nivel de abstracción:



Imágen extraída de: MVVM Clean Architecture Pattern in Android with Use Cases

Inyección de dependencias con Hilt

Para gestionar las dependencias en el proyecto, he utilizado Hilt, una biblioteca de inyección de dependencias para Android que está basada en Dagger. Hilt simplifica la configuración y el uso de Dagger en proyectos Android.

Con esta herramienta, conseguimos simplificar el código y gestionar las dependencias de manera más eficiente.

Para configurar Hilt, se añaden las dependencias necesarias en el build.gradle y anotar la clase Application con `@HiltAndroidApp`.

Para poder llevarlo a cabo, necesitamos crear un módulo, para indicarle a Hilt qué necesita una clase para ser inicializada.

Ejemplo de módulo con Hilt

```
package dam.adri.domain.apiModule
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import dam.adri.domain.api.DriverApiService
import retrofit2.Retrofit
import javax.inject.Singleton


@Module
@InstallIn(SingletonComponent::class)
object DriverApiModule {

    @Singleton
    @Provides
    fun provideDriverApiService(retrofit: Retrofit): DriverApiService {
        return retrofit.create(DriverApiService::class.java)
    }
}
```

Indico la anotación `@Singleton` para que solo se cree una instancia de la clase en toda la aplicación.

El argumento que le hemos pasado, necesita ser inyectado también, por lo que se debe crear otro módulo para este, y así sucesivamente.

Hilt sabe que tiene que realizar la inyección gracias a la anotación `@Provides` y a la clase que devuelve la función, de esta forma sabe que cuando inyectamos DriverApiService, lo que necesita para inicializarlo.

Uso de Retrofit y Moshi

Para poder hacer llamadas a la API mostrada anteriormente y poder serializar el JSON a los objetos, he utilizado las librerías Retrofit y Moshi.

Retrofit

Con Retrofit conseguimos comunicarnos con la API mediante solicitudes HTTP.

Moshi

Moshi se ocupa de la conversión de JSON a objetos y viceversa.

Módulo Hilt Retrofit

```
@Module
@InstallIn(SingletonComponent::class)
object RetrofitModule {

    @Singleton
    @Provides
    fun provideRetrofit(moshi: Moshi): Retrofit {
        return Retrofit.Builder()
            .baseUrl(ConstantInfo.BASE_URL)
            .addConverterFactory(MoshiConverterFactory.create(moshi))
            .build()
    }
}
```

Módulo Hilt Moshi

```
@Module
@InstallIn(SingletonComponent::class)
object MoshiModule {

    @Singleton
    @Provides
    fun provideMoshi(): Moshi {
        return Moshi.Builder()
            .add(KotlinJsonAdapterFactory())
            .build()
    }
}
```

Uso de moshi en DTOs

Para usar Moshi, debemos indicar los nombres de los parámetros del JSON, por cada atributo del data class:

```
import com.squareup.moshi.Json
import com.squareup.moshi.JsonClass
import dam.adri.domain.modelo.entities.Driver


@JsonClass(generateAdapter = true)
data class DriverDto(
   @Json(name = "driverNumber") val driverNumber: Int,
   @Json(name = "countryCode") val countryCode: String?,
   @Json(name = "image") val image: String?,
   @Json(name = "lastName") val lastName: String?,
   @Json(name = "teamColour") val teamColour: String?,
   @Json(name = "teamName") val teamName: String?,
   @Json(name = "nameAcronym") val nameAcronym: String?
)
```

De esta forma, cada nombre metido en el parámetro name, los cuáles són los identificadores JSON, serán el valor proporcionado a cada atributo del data class.

Ejemplos de código

Una vez explicado como se ha trabajado en el proyecto, procedo a mostrar ejemplos específicos y cómo conectan unos con otros.

ApiService

Define las llamadas a la API externa. Este componente maneja la comunicación con servicios externos utilizando Retrofit.

```
import dam.adri.domain.modelo.dto.UserLeagueDto
import retrofit2.http.*
import retrofit2.Response

interface UserLeagueApiService {

   @GET("/userleague/user/{userId}")
   suspend fun getUserLeaguesByUserId(@Path("userId") userId: Int):
Response<List<UserLeagueDto>>

   @GET("/userleague/league/{leagueId}")
   suspend fun getUserLeaguesByLeagueId(@Path("leagueId") leagueId: Int):
Response<List<UserLeagueDto>>

   @GET("/userleague/{userId}/{leagueId}")
   suspend fun getUserLeagueByUserIdAndLeagueId(
       @Path("userId") userId: Int,
       @Path("leagueId") leagueId: Int
   ): Response<UserLeagueDto>

   @POST("/userleague")
   suspend fun addUserLeague(@Body userLeagueDto: UserLeagueDto): Response<Unit>

   @DELETE("/userleague/{userId}/{leagueId}")
   suspend fun deleteUserLeague(
       @Path("userId") userId: Int,
       @Path("leagueId") leagueId: Int
   ): Response<Unit>
}
```

Repository

Define las interfaces para las operaciones de datos. Proporciona una abstracción sobre los datos, facilitando la interacción con diferentes fuentes de datos.

```
import dam.adri.domain.modelo.entities.UserLeague

interface UserLeagueRepository {

    suspend fun getUserLeaguesByUserId(userId: Int): List<UserLeague>

    suspend fun getUserLeaguesByLeagueId(leagueId: Int): List<UserLeague>

    suspend fun getUserLeagueByUserIdAndLeagueId(userId: Int, leagueId: Int): UserLeague?

    suspend fun addUserLeague(userLeague: UserLeague)

    suspend fun deleteUserLeague(userId: Int, leagueId: Int)
}
```

RepositoryImpl

Implementa Repository, contiene la lógica de acceso a datos, llamando a su correspondiente apiService. Implementa un sistema de caché para mejorar el rendimiento.

```
import dam.adri.domain.api.UserLeagueApiService
import dam.adri.domain.modelo.dto.UserLeagueDto
import dam.adri.domain.modelo.dto.toUserLeagueDto
import dam.adri.domain.modelo.entities.UserLeague
import dam.adri.domain.modelo.entities.toUserLeague
import dam.adri.domain.repository.UserLeagueRepository
import retrofit2.HttpException
import javax.inject.Inject

class UserLeagueRepositoryImpl @Inject constructor(
    private val userLeagueApiService: UserLeagueApiService
) : UserLeagueRepository {

    private var cachedUserLeaguesByUserId: Map<Int, List<UserLeague>> = emptyMap()
    private var cachedUserLeaguesByLeagueId: Map<Int, List<UserLeague>> = emptyMap()
    private var cachedUserLeagueByUserIdAndLeagueId: Map<Pair<Int, Int>, UserLeague?> = emptyMap()

    override suspend fun getUserLeaguesByUserId(userId: Int): List<UserLeague> {
        return cachedUserLeaguesByUserId[userId] ?: run {
            val response = userLeagueApiService.getUserLeaguesByUserId(userId)
            if (response.isSuccessful) {
                val userLeagues = response.body()?.map { it.toUserLeague() } ?: emptyList()
                cachedUserLeaguesByUserId = cachedUserLeaguesByUserId + (userId to userLeagues)
                userLeagues
            } else {
                throw Exception("Error fetching user leagues: ${response.errorBody()?.string()}")
            }
        }
    }
```

```
    override suspend fun getUserLeaguesByLeagueId(leagueId: Int): List<UserLeague> {
        return cachedUserLeaguesByLeagueId[leagueId] ?: run {
            val response = userLeagueApiService.getUserLeaguesByLeagueId(leagueId)
            if (response.isSuccessful) {
                val userLeagues = response.body()?.map { it.toUserLeague() } ?: emptyList()
                cachedUserLeaguesByLeagueId = cachedUserLeaguesByLeagueId + (leagueId to userLeagues)
                userLeagues
            } else {
                throw Exception("Error fetching user leagues: ${response.errorBody()?.string()}")
            }
        }
    }

    override suspend fun getUserLeagueByUserIdAndLeagueId(userId: Int, leagueId: Int): UserLeague? {
        return cachedUserLeagueByUserIdAndLeagueId[userId to leagueId] ?: run {
            val response = userLeagueApiService.getUserLeagueByUserIdAndLeagueId(userId, leagueId)
            if (response.isSuccessful) {
                val userLeague = response.body()?.toUserLeague()
                cachedUserLeagueByUserIdAndLeagueId = cachedUserLeagueByUserIdAndLeagueId + ((userId
to leagueId) to userLeague)
                userLeague
            } else {
                throw Exception("Error fetching user league: ${response.errorBody()?.string()}")
            }
        }
    }

    override suspend fun addUserLeague(userLeague: UserLeague) {
        val userDto = userLeague.toUserLeagueDto()
        val response = userLeagueApiService.addUserLeague(userDto)
        if (response.isSuccessful) {

            cachedUserLeaguesByUserId = cachedUserLeaguesByUserId.filterNot { it.key ==
userLeague.user }
            cachedUserLeaguesByLeagueId = cachedUserLeaguesByLeagueId.filterNot { it.key ==
userLeague.league }
            cachedUserLeagueByUserIdAndLeagueId = cachedUserLeagueByUserIdAndLeagueId.filterNot {
it.key == Pair(userLeague.user, userLeague.league) }
        } else {
            throw Exception("Error adding user league:\n ${response.errorBody()?.string()}")
        }
    }

    override suspend fun deleteUserLeague(userId: Int, leagueId: Int) {
        val response = userLeagueApiService.deleteUserLeague(userId, leagueId)
        if (response.isSuccessful) {

            cachedUserLeaguesByUserId = cachedUserLeaguesByUserId.filterNot { it.key == userId }
            cachedUserLeaguesByLeagueId = cachedUserLeaguesByLeagueId.filterNot { it.key == leagueId
}
            cachedUserLeagueByUserIdAndLeagueId = cachedUserLeagueByUserIdAndLeagueId.filterNot {
it.key == Pair(userId, leagueId) }
        } else {
            throw Exception("Error deleting user league: ${response.errorBody()?.string()}")
        }
    }
}
```

Sistema de caché

El uso de las variables `cachedUserLeaguesByUserId`, `cachedUserLeaguesByLeagueId` y `cachedUserLeagueByUserIdAndLeagueId` se destina a guardar la información una vez se hace una llamada a la API, de esta forma, y como previamente hemos dicho, solo se instancia una vez cada clase en toda la aplicación, tendremos los datos a disposición de forma rápida una vez se haya cargado en memoria, sin necesidad de hacer otra llamada a la API.

Esto se consigue gracias a los eficientes mapeadores que nos proporciona Kotlin.collections, ya que, cada variable almacena la información con un id como clave única, basada en los parámetros de la solicitud, es decir, si buscamos un userleague por idUser y idLeague, esta información se guardará y podremos volver a acceder a ella gracias a las claves idUser y idLeague.

Podremos acceder a ella de la siguiente manera:

```
cachedUserLeagueByUserIdAndLeagueId[userId to leagueId].
```

Antes de realizar una solicitud a la API, se comprobará si los datos ya están en memoria.

Este sistema lo implanté tras notar deficiencias en cuanto a la velocidad de la aplicación a la hora de cargar los datos.

Use Case

Es el intermediador entre la capa de presentación y la capa de datos, el encargado de llamar a Repository y proveer a ViewModel.

```
import dam.adri.domain.modelo.entities.Driver
import dam.adri.domain.repository.DriverRepository
import dam.adri.domain.repository.UserLeagueRepository
import javax.inject.Inject

class GetDriversByUserLeagueUseCase @Inject constructor(
   private val userLeagueRepository: UserLeagueRepository,
   private val driverRepository: DriverRepository
) {
   suspend fun getDriversByUserLeague(
       userId: Int,
       leagueId: Int
   ): List<Driver> {
       return try {
           val userLeague = userLeagueRepository.getUserLeagueByUserIdAndLeagueId(userId, leagueId)
           val driversNumbers = userLeague?.let {
               listOfNotNull(
                   it.driver1Number,
                   it.driver2Number,
                   it.driver3Number,
                   it.driver4Number
               )
           } ?: emptyList()

           driversNumbers.map { driverNumber ->
               driverRepository.findByNumber(driverNumber)
           }
       } catch (e: Throwable) {
           emptyList()
```

```
        }
    }
}
```

ViewModel

Maneja la lógica de presentación y la comunicación entre la vista (Fragment) y el caso de uso (UseCase). Mantiene los datos necesarios para la interfaz de usuario y gestiona su ciclo de vida.

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import dagger.hilt.android.lifecycle.HiltViewModel
import dam.adri.domain.modelo.entities.UserLeague
import dam.adri.domain.useCase.userLeague.GetUserLeagueByLeagueIdUseCase
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class QualificationLeagueViewModel @Inject constructor(
    private val getUserLeagueByLeagueIdUseCase: GetUserLeagueByLeagueIdUseCase
) : ViewModel() {

    private val _clasificacion = MutableLiveData<List<UserLeague>>()
    val clasificacion: LiveData<List<UserLeague>> get() = _clasificacion

    private val _error = MutableLiveData<String>()
    val error: LiveData<String> get() = _error

    fun loadQualification(leagueId: Int) {
        viewModelScope.launch(Dispatchers.IO) {
            try {
                val usuarios = getUserLeagueByLeagueIdUseCase.getUserLeagueByLeagueId(leagueId)
                _clasificacion.postValue(usuarios)
            } catch (throwable: Throwable) {
                _error.postValue(throwable.message ?: "unknown_error")
            }
        }
    }
}
```

Fragment

Representa la interfaz de usuario que observa los datos del ViewModel y presenta la información al usuario. Se encarga de la parte visual y de interactuar con el usuario.

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.fragment.app.Fragment
import androidx.fragment.app.commit
import androidx.fragment.app.viewModels
import dagger.hilt.android.AndroidEntryPoint
import dam.adri.core.data.utils.viewBinding
import dam.adri.domain.modelo.entities.UserLeague
import dam.adri.fantasy.R
import dam.adri.fantasy.databinding.QualificationBinding
import dam.adri.fantasy.presentation.leagueQualification.QualificationLeagueViewModel
import dam.adri.fantasy.presentation.leagueQualification.adapter.QualificationLeagueAdapter
import dam.adri.fantasy.presentation.userLeagueDescription.UserLeagueDescriptionFragment

@AndroidEntryPoint
class QualificationLeagueFragment : Fragment() {

    private val viewModel: QualificationLeagueViewModel by viewModels()
    private val binding by viewBinding(QualificationBinding::bind)

    private lateinit var adapter: QualificationLeagueAdapter

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.qualification, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupAdapter()
        observeViewModel()


        val leagueId = arguments?.getInt(ARG_LEAGUE_ID) ?: 0
        viewModel.loadQualification(leagueId)
    }

    private fun setupAdapter() {
        adapter = QualificationLeagueAdapter(requireContext(), emptyList()) { userleague ->
            openUserLeagueDescriptionFragment(userleague)
        }
        binding.listViewClasificacion.adapter = adapter
    }

    private fun openUserLeagueDescriptionFragment(userleague:UserLeague) {
        val fragment = UserLeagueDescriptionFragment.newInstance(userleague)
        parentFragmentManager.commit {
            replace(dam.adri.core.styles.R.id.fragment_container, fragment)
            addToBackStack(null)
```

```
        }
    }

    private fun observeViewModel() {
        viewModel.clasificacion.observe(viewLifecycleOwner) { usuarios ->
            adapter.updateUsuarios(usuarios)
        }

        viewModel.error.observe(viewLifecycleOwner) { errorMessage ->
            Toast.makeText(requireContext(), errorMessage, Toast.LENGTH_SHORT).show()
        }
    }

    companion object {
        private const val ARG_LEAGUE_ID = "leagueId"

        fun newInstance(leagueId: Int): QualificationLeagueFragment {
            val fragment = QualificationLeagueFragment()
            val args = Bundle().apply {
                putInt(ARG_LEAGUE_ID, leagueId)
            }
            fragment.arguments = args
            return fragment
        }
    }
}
```

Comunicación entre Fragment y ViewModel

Para el flujo de datos entre Fragment y ViewModel, para ello utilizamos el data class LiveData, una clase observable.

Ventajas de usar LiveData:

- **Evitar fuga de memoria**: al estar asociados a un objeto con ciclo de vida como són las activity y fragment, una vez estos objetos són destruidos, los observables se liberan de memoria.
- **Datos siempre actualizados**: aunque se produzca una actualización de los datos cuando un objeto con ciclo de vida se encuentre en segundo plano, los valores se almacenan en el LiveData. Cuando este objeto vuelva a estar activo, lo notificará y se mostrarán los datos actualizados.
- **Maneja cambios de configuración**: Los datos persisten a través de cambios de configuración, como la rotación de la pantalla.

Implementación de LiveData en ViewModel

Cuando creamos un LiveData en el viewModel para pasar datos al fragment, primero debemos crear un MutableLiveData privado, que contendrá los datos obtenidos por el caso de uso. Posteriormente crearemos un LiveData público que tendrá como valor el resultado del MutableLiveData privado. Esto se hace así para evitar modificaciones directas.

```
private val _clasificacion = MutableLiveData<List<UserLeague>>()
val clasificacion: LiveData<List<UserLeague>> get() = _clasificacion
```

Una vez llegamos al punto donde queremos enviar los datos al fragment, haremos uso del método .postValue(<valor>)

```
_clasificacion.postValue(usuarios)
```

Observación de LiveData en Fragment

Para obtener los datos modificados desde el fragment, haremos uso del método .observe:

```
viewModel.clasificacion.observe(viewLifecycleOwner) { usuarios ->
            adapter.updateUsuarios(usuarios)
        }
```

En este caso, cuando el viewModel actualice la lista de Userleague, el fragment lo observará y actualizará la lista de Userleague del adapter.

De esta forma actualizamos la interfaz de usuario de manera reactiva y eficiente, respetando el ciclo de vida de los componentes, además de las ventajas descritas con anterioridad.

Activity

Cada módulo tiene una Activity principal, sirve como contenedor para los Fragments.

```
import android.content.Context
import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.commit
import dagger.hilt.android.AndroidEntryPoint
import dam.adri.core.data.utils.viewBinding
import dam.adri.core.styles.databinding.ActivityFrameBinding

@AndroidEntryPoint
class UserLeaguesActivity : AppCompatActivity() {

    private val binding by viewBinding(ActivityFrameBinding::inflate)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(binding.root)

        if (savedInstanceState == null) {
            supportFragmentManager.commit {
                replace(dam.adri.core.styles.R.id.fragment_container, UserLeaguesFragment())
            }
        }
    }

    companion object {
        fun buildIntent(context: Context) = Intent(context, UserLeaguesActivity::class.java)
    }
}
```

Vistas XML

A continuación, se muestran los ejemplos de las vistas XML utilizadas en los Fragments y Activities para construir la interfaz de usuario de la aplicación.

activity_frame.xml

Define el diseño para cada activity, contiene un framelayout utilizado para cargar diferentes fragments

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical">

   <FrameLayout
       android:id="@+id/fragment_container"
       android:layout_width="match_parent"
       android:layout_height="match_parent" />

</LinearLayout>
```

qualification.xml

Define el diseño para QualificationLeagueFragment, contiene un textView para el título y un listView para mostrar una lista de clasificación de usuarios.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="16dp">


   <TextView
       android:id="@+id/textViewEncabezado"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="@string/clasificaci_n_de_usuarios"
       android:textSize="18sp"
       android:textStyle="bold"
       android:layout_marginBottom="16dp" />

   <ListView
       android:id="@+id/listViewClasificacion"
       android:layout_width="match_parent"
       android:layout_height="wrap_content" />

</LinearLayout>
```

qualification_item.xml

Define el diseño de cada elemento en la lista de clasificación de usuarios.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="16dp"
    android:background="@color/white">

    <TextView
        android:id="@+id/textViewNombreUsuario"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Nombre del Usuario"
        android:textSize="16sp"
        android:textStyle="bold"
        android:layout_gravity="center_vertical"
        android:textColor="@color/black"/>

    <TextView
        android:id="@+id/textViewNumeroClasificacion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="#1"
        android:textSize="16sp"
        android:layout_marginEnd="8dp"
        android:textColor="@color/redF1"/>

    <TextView
        android:id="@+id/textViewPuntosUsuario"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="1000 pts"
        android:textColor="@color/blueF1"
        android:textSize="16sp"/>

</LinearLayout>
```

Adapter

El adapter se encarga de la conexión con el ListView de una lista, crea un qualification_item, por cada objeto de la lista.

```
import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.BaseAdapter
import android.widget.TextView
import dam.adri.fantasy.R
import dam.adri.domain.modelo.entities.UserLeague

class QualificationLeagueAdapter(
   private val context: Context,
   private var userLeagues: List<UserLeague>,
   private val onItemClickListener: (UserLeague) -> Unit
) : BaseAdapter() {

   override fun getCount(): Int {
       return userLeagues.size
   }

   override fun getItem(position: Int): UserLeague {
       return userLeagues[position]
   }

   override fun getItemId(position: Int): Long {
       return position.toLong()
   }

   override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
       val view: View
       val viewHolder: ViewHolder

       if (convertView == null) {
           view = LayoutInflater.from(context).inflate(R.layout.qualification_item, parent, false)
           viewHolder = ViewHolder(view)
           view.tag = viewHolder
       } else {
           view = convertView
           viewHolder = convertView.tag as ViewHolder
       }

       val userLeague = getItem(position)
       viewHolder.textViewNombreUsuario.text = userLeague.userName
       viewHolder.textViewNumeroClasificacion.text = "#${position + 1}"
       viewHolder.textViewPuntosUsuario.text = "${userLeague.puntuation} pts"
       view.setOnClickListener {
           onItemClickListener(userLeague)
       }

       return view
   }

   fun updateUsuarios(newUsuarios: List<UserLeague>) {
       userLeagues = newUsuarios
```

```
        notifyDataSetChanged()
    }

    private class ViewHolder(view: View) {
        val textViewNombreUsuario: TextView = view.findViewById(R.id.textViewNombreUsuario)
        val textViewNumeroClasificacion: TextView =
view.findViewById(R.id.textViewNumeroClasificacion)
        val textViewPuntosUsuario: TextView = view.findViewById(R.id.textViewPuntosUsuario)
    }
}
```

Después en el fragment, lo inicializamos de la siguiente forma:

```
 private fun setupAdapter() {
        adapter = QualificationLeagueAdapter(requireContext(), emptyList()) { userleague ->
            openUserLeagueDescriptionFragment(userleague)
        }
        binding.listViewClasificacion.adapter = adapter
    }
```

Cuando el viewModel tiene lista de userleagues, actualiza la lista del adapter:

```
viewModel.clasificacion.observe(viewLifecycleOwner) { usuarios ->

    adapter.updateUsuarios(usuarios)

}
```

MainActivity

Es el primer punto en el que se encuentra la aplicación una vez es lanzada, gestiona la pantalla de bienvenida (splash), inicializa repositorios y pasa a LoginActivity.

```
import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.core.splashscreen.SplashScreen.Companion.installSplashScreen
import androidx.lifecycle.lifecycleScope
import dagger.hilt.android.AndroidEntryPoint
import dam.adri.core.data.utils.viewBinding
import dam.adri.domain.initializer.RepositoryInitializer
import dam.adri.fx3application.databinding.ActivityMainBinding
import dam.adri.fx3application.presentation.login.LoginActivity
import kotlinx.coroutines.launch
import javax.inject.Inject

@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var repositoryInitializer: RepositoryInitializer

    private val binding by viewBinding(ActivityMainBinding::inflate)
```

```
    override fun onCreate(savedInstanceState: Bundle?) {

        val splashScreen = installSplashScreen()

        super.onCreate(savedInstanceState)
        setContentView(binding.root)

        initializeRepositories()
    }

    private fun initializeRepositories() {
        lifecycleScope.launch {
            try {
                repositoryInitializer.initializeRepositories()
                navigateToLogin()
            } catch (e: Exception) {
                e.printStackTrace()
                Toast.makeText(this@MainActivity,
                    getString(dam.adri.core.styles.R.string.error_initializing_repositories),
Toast.LENGTH_LONG).show()
            }
        }
    }

    private fun navigateToLogin() {
        val intent = Intent(this, LoginActivity::class.java)
        startActivity(intent)
        finish()
    }
}
```

Gestión de sesión de usuario

Para gestionar la sesión del usuario, se ha implementado un sistema que permite verificar si un usuario está logueado, iniciar sesión, cerrar sesión y obtener el ID del usuario logueado.

Implementación de la gestión de sesión

SessionUseCase

Requiere de UserRepository y UserManager para gestionar la sesión del usuario.

```
import dam.adri.core.presentation.userManager.UserManager
import dam.adri.domain.modelo.entities.User
import dam.adri.domain.repository.UserRepository
import javax.inject.Inject
import javax.inject.Singleton


class SessionUseCase @Inject constructor(
   private val userRepository: UserRepository,
   private val userManager: UserManager
) {
   fun getUserId(): Int? = userManager.getUserId()

   fun isUserLoggedIn(): Boolean = userManager.isUserLoggedIn()

   fun logout() {
       userManager.logout()
   }

   fun logIn(id: Int) {
       userManager.logIn(id)
   }

}
```

UserManager

Esta clase utiliza SharedPreferences para almacenar la sesión de usuario.

```
import dam.adri.core.data.ConstantInfo.KEY_USER_ID
import dam.adri.core.data.ConstantInfo.KEY_USER_LOGGED_IN
import android.content.SharedPreferences

class UserManager(private val sharedPreferences: SharedPreferences) {

    fun isUserLoggedIn(): Boolean {
        return sharedPreferences.getBoolean(KEY_USER_LOGGED_IN, false)
    }

    fun logIn(userId: Int) {
        sharedPreferences.edit().apply {
            putBoolean(KEY_USER_LOGGED_IN, true)
            putInt(KEY_USER_ID, userId)
            apply()
        }
    }

    fun logout() {
        sharedPreferences.edit().apply {
            putBoolean(KEY_USER_LOGGED_IN, false)
            remove(KEY_USER_ID)
            apply()
        }
    }

    fun getUserId(): Int? {
        return if (isUserLoggedIn()) {
            sharedPreferences.getInt(KEY_USER_ID, -1)
        } else null
    }
}
```

SessionModule

Módulo de Hilt que proporciona las dependencias necesarias para la gestión de sesión.

```
import android.content.Context
import android.content.SharedPreferences
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import dam.adri.core.presentation.userManager.UserManager
import javax.inject.Singleton
import dam.adri.core.data.ConstantInfo.SHARED_PREFERENCES
import dam.adri.domain.repository.UserRepository

@Module
@InstallIn(SingletonComponent::class)
object SessionModule {

    @Singleton
    @Provides
    fun provideUserManager(sharedPreferences: SharedPreferences): UserManager {
        return UserManager(sharedPreferences)
    }

    @Singleton
    @Provides
    fun provideSessionUseCase(
        userRepository: UserRepository,
        userManager: UserManager
    ): SessionUseCase {
        return SessionUseCase(userRepository, userManager)
    }

    @Singleton
    @Provides
    fun provideSharedPreferences(@ApplicationContext context: Context): SharedPreferences {
        return context.getSharedPreferences(SHARED_PREFERENCES, Context.MODE_PRIVATE)
    }
}
```

Tras esto, desde el viewModel podemos llamar a sessionUseCase y obtener el usuario logueado con el método .getUserId() de sessionUseCase, por ejemplo, a la hora de listar las ligas donde está registrado:

```
import android.util.Log
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import dagger.hilt.android.lifecycle.HiltViewModel
import dam.adri.core.presentation.sessionModule.SessionUseCase
import dam.adri.domain.modelo.entities.League
import dam.adri.domain.useCase.league.GetLeagueByAccessCodeUseCase
import dam.adri.domain.useCase.league.GetLeaguesByUserUseCase
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class UserLeaguesViewModel @Inject constructor(
    private val getLeaguesByUserUseCase: GetLeaguesByUserUseCase,
    private val getLeagueByAccessCodeUseCase: GetLeagueByAccessCodeUseCase,
    private val sessionUseCase: SessionUseCase
) : ViewModel() {

    private val _ligas = MutableLiveData<List<League>>()
    val ligas: LiveData<List<League>> get() = _ligas

    private val _error = MutableLiveData<String>()
    val error: LiveData<String> get() = _error

    private val _ligaSearched = MutableLiveData<League>()
    val ligaSearched: LiveData<League> get() = _ligaSearched

    init {
        cargarLigasUsuario()
    }

    fun searchLeague(codigoAcceso: String) {
        viewModelScope.launch(Dispatchers.IO) {
            try {
                val liga = getLeagueByAccessCodeUseCase.getLeagueByAccessCode(codigoAcceso)
                _ligaSearched.postValue(liga)
            } catch (throwable: Throwable) {
                _error.postValue(throwable.message ?: "unknown_error")
            }
        }
    }

    private fun cargarLigasUsuario() {
        viewModelScope.launch(Dispatchers.IO) {
            try {
                val ligas = sessionUseCase.getUserId()
                    ?.let { getLeaguesByUserUseCase.getLeaguesByUser(it) }
                Log.d("UserLeaguesViewModel", "Fetched leagues: ${ligas?.size ?: 0}")
                _ligas.postValue(ligas ?: emptyList())
            } catch (throwable: Throwable) {
                _error.postValue(throwable.message ?: "unknown_error")
```

```
            }
        }
    }
}
```

Clase aplicación

La clase Fx3App, es la clase aplicación, encargada de inicializar la inyección de dependencias con la anotación `@HiltAndroidApp`, y MultiDexApplication para soportar múltiples archivos DEX en caso de que el número de métodos de la aplicación exceda el límite de un solo archivo DEX.

```
@HiltAndroidApp
class Fx3App : MultiDexApplication() {
}
```

AndroidManifest.xml

Define la configuración de la aplicación, desde aquí se gestionan los permisos, las activity existentes en el proyecto y la activity que tiene que ser lanzada al iniciar la aplicación, cabe destacar que cada módulo tiene su manifest, pero solo en el módulo app configuraremos parámetros de la aplicación, en los demás sólo indicaremos las activity existentes en su propio módulo.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">


    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <application
        android:name=".Fx3App"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AppCompat.DayNight.NoActionBar"
        tools:targetApi="31"
        android:networkSecurityConfig="@xml/network_security_config">
        <activity
            android:name=".presentation.splash.MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

```
        <activity
            android:name=".presentation.mainMenu.MainMenuActivity"
            android:exported="true">
        </activity>
        <activity android:name=".presentation.login.LoginActivity"/>
        <activity android:name="dam.adri.fantasy.presentation.userLeagues.UserLeaguesActivity"/>
        <activity
android:name="dam.adri.grandprix.presentation.grandPrixDescription.NextGrandPrixActivity"/>
        <activity android:name="dam.adri.knowledge.presentation.KnowledgeActivity"/>
    </application>
</manifest>
```

Vídeo aplicación en funcionamiento

Link de aplicación en funcionamiento

Propuestas de mejora

Una aplicación siempre se puede mejorar, pero esta es la lista de las mejoras que me gustaría hacer en un futuro:

- **Mejorar el aspecto de la aplicación**
    - **Diseño de interfaz de usuario**: hacer un diseño más atractivo y moderno, añadiendo tipografías, una navegación más dinámica.
    - **Temas:** posibilidad de cambiar a temas claros y oscuros.
- **Añadir jugabilidad**
    - **Puntuación en otras sesiones**: actualmente, solo puntúa la carrera, sería interesante añadir puntuaciones en otras sesiones como clasificación, sprint, etc.
    - **Predicción de resultados**: añadir una funcionalidad donde el usuario pueda predecir el resultado de una carrera, y obtener puntos dependiendo de cuantas posiciones hayas acertado.
- **Mejorar la UX**
    - **Notificaciones**: añadir notificaciones para mantener a los usuarios informados cuando estén listos los resultados, las puntuaciones, etc.
    - **Multilingüe**: esta implementación está a medias pero no he conseguido finalizarlo, la idea es que la aplicación pueda estar tanto en español como en inglés.
- **Mejoras de código**:
    - Añadir tests.
    - Refactorizar código.

Conclusión

Han sido unos meses duros, compaginar las prácticas y el desarrollo del proyecto no ha sido nada fácil. Puedo decir que realizar un proyecto con las mismas tecnologías con las que trabajas en la empresa, tiene sus pros y sus contras. Por un lado, la velocidad de aprendizaje es mayor, pues inviertes más tiempo en practicar y enseñarte, pero acabar 8 horas de trabajo y ponerte a hacer lo mismo otras 4 horas en el proyecto es pesado, pero vale la pena.

Me queda mucho por aprender de Kotlin. Al principio pensaba que sería pan comido, pues en las prácticas no he tenido prácticamente problemas. Me di cuenta que no es lo mismo unirte a un proyecto que empezarlo de cero, debido a todas las automatizaciones que hay por detrás. Estas automatizaciones facilitan la programación y he experimentado lo que me gustaría que otros desarrolladores sintieran al unirse a mi proyecto.

Una buena elección fue enfocar este proyecto como una inversión en mí, sin pensar únicamente en el resultado final. Estoy satisfecho, pero sé que el resultado no refleja completamente el trabajo que hay por detrás. Decidí aprender a programar y utilizar técnicas que permitan que mi proyecto siga creciendo en el futuro, que otros programadores puedan entenderlo sin dificultad y que sea más fácil avanzar. Esto es lo que más valoro de esta experiencia.

No voy a engañaros, al principio me daba pereza el "proyecto final". Lo peor es comenzar, pero después, cuando ves que las cosas van saliendo tras muchos intentos, cada vez tienes más ganas de mejorar. A día de hoy, estoy seguro de que seguiré mejorando este proyecto.

WebGrafía

Kotlin

Android developers

SpringBoot

JPA Repository

Dependency-injection

MVVM-CleanArchitecture

LiveData