TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

# Desenvolvemento dun sistema de soporte para partidas online do xogo de cartas de Digimon

**Estudante:** Héctor Vázquez Fuentes

**Dirección:** Dra. María Noelia Barreira Rodríguez

A Coruña, Febreiro de 2023.

*Dedicado a miña familia e amigos*

**Agradecementos**

Á directora do traballo polo seu apoio, ánimo e paciencia.
Á miña familia por distraerme cando o precisaba.
Ós meus amigos por meter a presión necesaria para que ó final rematara.

**Resumo**

Este proxecto consistirá nun sistema, dividido en tres partes, que dará soporte ás partidas *online* do xogo de cartas de *Digimon*. Por unha parte, o sistema consta dun servidor que almacena a información das cartas existentes, das partidas e das cartas que se vaian xogando na mesma. No servidor empregaremos métodos de procesado de imaxes para identificar as imaxes das cartas xogadas. Por outra parte, o sistema contará con dúas aplicacións cliente. En primeiro lugar unha aplicación móbil permitirá ós usuarios crear, unirse a partidas e enviar fotos das cartas que vaia xogando para que o servidor as identifique e rexistre. Ademais, un cliente web mostrará as partidas de maneira clara, precisando así a mínima interacción por parte do usuario. Para evitar que o usuario teña que actualizar a páxina a información das partidas actualizarase de maneira automática empregando *websockets*.

**Abstract**

This project will consist of a system, divided in three parts, that will support online matchs of Digimon Trading Card Game (DTCG). On one hand, the system consists of a server that stores the information of the existing cards, the games and the cards that are being played in the game. In the server we will use image processing methods to identify the images of the cards played. On the other hand, the system will have two client applications. Firstly, a mobile application will allow users to create, join games and send photos of the cards they are playing so that the server can identify and register them. In addition, a web client will display the games in a clear way, requiring minimal user interaction. To avoid the user having to update the page, the game information will be updated automatically using websockets.

**Palabras chave:**

- OpenCV
- Django
- Flutter
- Cartas
- Procesado de imaxes
- Digimon
- OCR
- Websockets

**Keywords:**

- OpenCV
- Django
- Flutter
- Cards
- Image processing
- Digimon
- OCR
- Websockets

# Índice Xeral


**1 Introdución** **1**

1.1 Xogo de cartas de Digimon . . . 1
1.2 Problemática online . . . 1
1.3 Solución proposta . . . 2
1.4 Estrutura da memoria . . . 4

**2 Digimon Trading Card Game** **6**

2.1 Campo de xogo . . . 6
2.2 Cartas . . . 8
2.3 Mazos . . . 10
2.4 Fases da quenda . . . 10
2.5 Memoria e quendas . . . 12
2.6 Efectos . . . 13
2.7 Cartas de seguridade . . . 14

**3 Tecnoloxías empregadas** **16**

3.1 Xerais . . . 16
3.2 Servidor . . . 16
3.2.1 Python3 . . . 17
3.2.2 Django . . . 17
3.2.3 Django Rest Framework (DRF) . . . 17
3.2.4 Tesseract-OCR . . . 18
3.2.5 OpenCV . . . 18
3.2.6 Channels . . . 18
3.2.7 BeautifulSoup . . . 18
3.3 Aplicación móbil . . . 18
3.4 Web . . . 19
3.4.1 VueJS . . . 19

3.4.2 NodeJS . . . . . 19

**4 Metodoloxía e planificación do proxecto** **20**

4.1 SCRUM . . . . . 20

4.2 Planificación . . . . . 21

4.3 Seguemento proxecto . . . . . 24

4.4 Custos . . . . . 25

**5 Deseño do sistema** **26**

5.1 Arquitectura do sistema . . . . . 26

5.2 Base de datos . . . . . 26

5.3 Servizo . . . . . 28

5.3.1 Endpoints . . . . . 33

5.3.2 Procesamento de imaxe . . . . . 35

5.4 Cliente Web . . . . . 36

5.5 Aplicación móbil . . . . . 39

**6 Desenvolvemento** **44**

6.1 Estudo do problema . . . . . 44

6.2 Detección das cartas . . . . . 45

6.3 Servidor . . . . . 48

6.3.1 Usuarios . . . . . 50

6.3.2 Clases modelo . . . . . 52

6.3.3 Partidas . . . . . 53

6.3.4 Tratamento imaxes . . . . . 54

6.3.5 Cartas . . . . . 55

6.3.6 Websockets . . . . . 57

6.4 Aplicación móbil . . . . . 59

6.4.1 Usuarios . . . . . 59

6.4.2 Partidas . . . . . 60

6.4.3 Fotos . . . . . 65

6.4.4 Cartas . . . . . 68

6.5 Cliente web . . . . . 69

6.5.1 Websockets . . . . . 72

**7 Conclusións** **74**

7.1 Traballos futuros . . . . . 75

**Relación de Acrónimos** **76**

**Glosario** 77

**Bibliografía** 79

# Índice de Figuras


1.1 Exemplo de campo de xogo . . . 2
1.2 Exemplo de *digimon* con habilidades herdadas . . . 3

2.1 Campo de xogo cas zonas importantes marcadas . . . 7
2.2 Exemplo de carta de digitama . . . 8
2.3 Exemplo de carta de digimon . . . 9
2.4 Exemplos de cartas de *tamers*. Esq.: Sen habilidade herdada. Der.: Con habilidade herdada. . . . 10
2.5 Exemplo de carta de *option* . . . 11
2.6 Barra de memoria independente . . . 13
2.7 *Digimon* con efecto activable e herdada permanente . . . 14

4.1 Estado do taboleiro de *Trello* durante o desenvolvemento . . . 21
4.2 Diagrama de Gantt . . . 22
4.3 Diagrama de Gantt coa duración real . . . 25

5.1 Arquitecura do proxecto . . . 27
5.2 Esquema base de datos . . . 29
5.3 Diagrama de clases do modelo . . . 30
5.4 Diagrama de clases dos *serializers* . . . 31
5.5 Diagrama de clases das vistas . . . 31
5.6 Esquema de peticións e respostas . . . 32
5.7 Fluxo de traballo do servizo . . . 33
5.8 Procesamento de imaxe . . . 36
5.9 Execución cliente web . . . 37
5.10 Diagrama de clases do cliente web . . . 37
5.11 Bosquexo do cliente web . . . 38
5.12 Bosquexo do cliente web despois de facer clic nun *digimon* . . . 38

5.13 Fluxo de traballo do cliente web . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.14 Acción en *Flutter* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.15 Bosquexo da pantalla de inicio de sesión . . . . . . . . . . . . . . . . . . . . . 40
5.16 Bosquexo do menú principal . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.17 Bosquexo da pantalla principal de partida . . . . . . . . . . . . . . . . . . . . 41
5.18 Bosquexo da pantalla de toma de fotos . . . . . . . . . . . . . . . . . . . . . 42
5.19 Diagrama de clases das pantallas da aplicación móbil . . . . . . . . . . . . . . 42
5.20 Diagrama de clases do modelo e do servizo . . . . . . . . . . . . . . . . . . . 43

6.1 *Digimons* con debuxos similares . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.2 Imaxe modelo da zona de nivel . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.3 Ficheiros no proxecto do servidor . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.4 Ficheiros de desenvolvemento da aplicación móbil . . . . . . . . . . . . . . . . 60
6.5 Pantalla de inicio de sesión . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.6 Pantalla principal da aplicación móbil . . . . . . . . . . . . . . . . . . . . . . . 62
6.7 Pantalla de partida con un digimon en xogo . . . . . . . . . . . . . . . . . . . 64
6.8 Toma de fotos. Esq.: Previsualización da cámara. Der.: Imaxe recortada ó tomar a foto. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.9 Imaxe da carta recortada e identificada . . . . . . . . . . . . . . . . . . . . . 68
6.10 Posibles accións sobre unha carta . . . . . . . . . . . . . . . . . . . . . . . . 70
6.11 Ficheiros do cliente web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.12 Imaxe do campo de xogo dunha partida . . . . . . . . . . . . . . . . . . . . . 72
6.13 Información dunha carta en detalle no cliente web . . . . . . . . . . . . . . . . 73

# Índice de Táboas

4.1 Historias de usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

5.1 Definición do *endpoint* de partidas . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Definición do *endpoint* para engadir xogadores . . . . . . . . . . . . . . . . . . 34
5.3 Definición do *endpoint* para engadir cartas . . . . . . . . . . . . . . . . . . . . 34
5.4 Definición do *endpoint* para ver as cartas dunha partida . . . . . . . . . . . . . 35
5.5 Definición do *endpoint* para eliminar unha carta . . . . . . . . . . . . . . . . . 35
5.6 Definición do *endpoint* de identificación de cartas . . . . . . . . . . . . . . . . 35
5.7 Definición do *endpoint* de inicio de sesión . . . . . . . . . . . . . . . . . . . . . 36

## Capítulo 1

# Introdución

NESTE capítulo faremos unha pequena introdución da franquía *Digimon*, explicaremos de maneira resumida o funcionamento dunha partida do xogo e os problemas que atopamos para realizar partidas *online*. Ademais definiremos como sería a solución proposta.

## 1.1 Xogo de cartas de Digimon

O xogo de cartas de *Digimon* (Digimon Trading Card Game (DTCG)) aparece no ano 2020. Baseado na franquía do mesmo nome, neste xogo preséntannos unhas formas de vida artificiais que reciben o nome de *Digimons (Digital Monsters)*. Estes seres conviven nun mundo paralelo ó dos humanos coñecido como Mundo Dixital. Estes *digimons* fanse máis fortes ó derrotar a outros o que os leva a pasar por un proceso de metamorfose que coñecemos como *digievolución*.

Unha partida do xogo de cartas enfronta a dous xogadores. Cada un deles contará cun mazo que cumpra as normas oficiais do xogo. Ó inicio da partida cada xogador baralla o seu mazo, coloca as 5 primeiras cartas como cartas de seguridade (vidas) e rouba as 5 seguintes á súa man. Decidirase quen empeza mediante o lanzamento dun dado ou empregando outro sistema aleatorio. Durante a partida cada xogador colocará *digimons* no campo de batalla para atacar ó opoñente ou ós *digimons* rivais. Estes *digimons* poderán *digievolucionar* para gañar poder e novas habilidades. O gañador será o que primeiro realice un ataque directo ó opoñente cando este non teña cartas na seguridade.

## 1.2 Problemática online

A propagación da COVID-19, que coincidiu coa saída do xogo, impulsou o aumento das partidas realizadas *online* e fomentou a aparición de torneos realizados neste formato.

Nas partidas presenciais o único material necesario é un mazo que cumpra as normas,

mentres que para partidas online debemos contar cun dispositivo con acceso a internet e cámara web. A realización das partidas online lévase a cabo a través dunha videochamada entre os xogadores. Ambos colocan a cámara apuntando ó terreo de xogo e mantendo as cartas na man dentro do encadramento. No transcurso da partida cada xogador vai explicando as accións realizadas e os efectos que vai activando.

Figura 1.1: Exemplo de campo de xogo

Na Figura 1.1 podemos apreciar que, aínda que algunhas cartas son identificables, os textos resultan ilexibles. Isto sucede coa maioría de cámaras web que podemos atopar de maneira asequible no mercado. Aínda que teñan unha boa calidade de imaxe, xa sexa polo posicionamento, a distancia ás cartas ou os reflexos da luz resulta moi complicado visualizar de forma correcta as xogadas do opoñente. No caso de xente que está comezando a xogar, aínda distinguindo a carta, é probable que non coñeza os seus efectos e teña que preguntarllos ó rival ou consultalos por outros medios como páxinas web específicas. Por último, o problema que máis confusión e dificultades crea nestas partidas serían as habilidades herdadas do *digimon* polas cartas que ten debaixo (Figura 1.2), xa que estas son indistinguibles polo debuxo e poden ser pasadas por alto polo rival á hora de enumerar os efectos.

## 1.3 Solución proposta

Como solución ós problemas presentados na sección anterior proponse un sistema de apoio independente das videochamadas. Este sistema estará dividido en tres partes:

- Aplicación móbil desde a que se poderá unir ás partidas e enviar fotos das cartas que imos xogando.
- Cliente web desde o cal consultar os *digimons* presentes nas partidas.

Figura 1.2: Exemplo de *digimon* con habilidades herdadas

- Servidor encargado de xestionar a información das cartas e as partidas.

A parte da aplicación móbil desenvolverase na linguaxe Flutter e permitirá iniciar a sesión na conta persoal, crear e unirse a partidas, sacar fotos das cartas, recortalas e envialas ó servidor para a súa identificación e xestionar as cartas presentes con comportamentos específicos do xogo.

De cara a parte visual levarase a cabo un cliente web empregando *VueJs*. Este cliente centrarase en presentar unha representación do campo de xogo de maneira que sexa facilmente comprensible dunha primeira ollada. Ademais aportará a capacidade de consultar en detalle os efectos dos *digimons* tendo en conta as cartas presentes debaixo que lle aporten efectos herdados. Ó tratarse dunha parte exclusivamente visual que non realizará modificacións non se precisará estar autenticado para ver partidas, soamente será preciso o identificador da partida. Isto facilitaría a retransmisión de partidas por parte de terceiros en plataformas coma Twitch ou similares.

O servidor almacenará a información das cartas, dos usuarios e xestionará as partidas en función das peticións realizadas desde a aplicación móbil. Mediante técnicas de visión artificial encargarase de identificar as cartas a partir da información presente nas fotos recibidas. Tamén se encargará de enviar un sinal ó cliente web cando se realice algún cambio na partida que se estea a ver. O servidor estará desenvolto en *Python* empregando as librerías de *Django* e *Django Rest Framework*, que nos axudarán a manexar as peticións e modificar a información almacenada, e o *OCR Tesseract* para a identificación dos textos das imaxes.

## 1.4 Estrutura da memoria

A memoria constará dos seguintes capítulos nos que se explicará, de maneira ordenada, os detalles do proxecto:

- Digimon Trading Card Game: daremos unha explicación detallada dos distintos elementos presentes nunha partida, as súas fases e a normativa básica.
- Tecnoloxías empregadas: mostraremos as tecnoloxías empregadas separadas por subsistema e explicaremos o uso que se lles deu.
- Metodoloxía e planificación do proxecto: explicaremos o método escollido para planificar o desenvolvemento do proxecto e estimar a duración e os seus custos.
- Deseño do sistema: ensinaremos a arquitectura xeral do sistema para despois presentar en detalle cada un dos subsistemas e o seu deseño.
- Desenvolvemento: presentaremos as distintas partes do sistema atendendo as funcionalidades completas e comentando porcións de código relevantes.

- Conclusións: falaremos dos obxectivos conseguidos e valoraremos posibles tarefas futuras.

Capítulo 2

# Digimon Trading Card Game

Durante este capítulo explicaremos a normativa do xogo. Comezaremos explicando os conceptos arredor dos que xira o xogo de maneira individual e, de cara o final do capítulo, incluiremos un exemplo de xogadas para ver como se combinan entre si na experiencia de xogo.

As partidas neste xogo, explicadas de maneira sinxela, desenvólvense entre dous xogadores que ó largo das quendas irán xogando os seus *digimons* ó campo, ou *digievolucionaranos* enriba doutros, co obxectivo de eliminar as 5 cartas de seguridade do rival e asestar un golpe final cando xa non lle queden.

As mecánicas de xogo máis distintivas, con respecto a outros xogos de cartas, serían:

- Fases da quenda
- Memoria e quendas
- Efectos
- Cartas de seguridade

Antes de entrar a falar destas mecánicas veremos detalles mais básicos como son o campo de xogo e como preparalo para iniciar unha partida, as cartas e os seus tipos así como os mazos e as súas normas.

## 2.1 Campo de xogo

Nesta sección apoiarémonos na Figura 2.1, sobre a que están marcadas as zonas importantes que é necesario coñecer para preparar o terreo de xogo antes de comezar unha partida:

1. A barra de memoria é compartida por ambos xogadores e utilízase para pagar o custo de xogar cartas. Supoñamos que pagamos 3 de memoria para xogar unha carta, moveríamos o contador tres ocos a dereita.

Figura 2.1: Campo de xogo cas zonas importantes marcadas

2. Zona de seguridade, onde se colocan as cartas de seguridade.

3. Zona para situar o mazo de *digitamas*.

4. A zona de cría é unha zona especial, allea ó campo de batalla, na que se poden "abrir" (colocar unha carta do mazo de *digitamas* boca arriba) os *digitama*.

5. Zona para colocar o mazo principal.

6. As cartas que se vaian eliminando do terreo de xogo ou da seguridade son colocadas na papeleira co debuxo cara arriba.

Antes de iniciar a partida cada xogador realizará as seguintes accións:

1. Barallar o mazo e colocalo no seu sitio

2. Barallar o mazo de *digitamas* e colocalo no seu lugar correspondente

3. Colocar as 5 primeiras cartas do mazo na zona de seguridade, sendo a primeira carta do mazo a última carta da seguridade

A continuación decídese quen comezará mediante algún xogo rápido aleatorio (lanzamento de dado, pedra-papel-tesoiras...). Despois ambos xogadores roubarán 5 cartas e situarán o contador de memoria no 0. O xogador coa quenda inicial non rouba carta no inicio dese primeiro quenda. Neste punto xa estaríamos preparados para iniciar a partida.

Figura 2.2: Exemplo de carta de digitama

## 2.2 Cartas

Distinguimos entre catro tipos distintos de cartas: *tamer*, *digimon*, *option* e *digitama*. As cartas de *digitama* só se poden usar no propio mazo de *digitama* mentres que as outras irían ó mazo normal. Outra distinción nas cartas sería a cor (vermello, negro, azul, verde, morado ou branco) que afectaría as evolucións e a posibilidade de utilizar cartas de *option*.

As cartas de *digitama* ou *digi-ovo* son consideradas cartas de *digimon* de nivel 2 e teñen unhas características particulares. Non teñen custo de xogo, xa que non poden ser xogadas, e non poden quedar no campo de batalla se non teñen unha carta de *digimon* de maior nivel enriba. Soamente poden estar pola súa conta na zona de cría. Só poden saír desta zona unha vez evolucionados a un *digimon* de nivel 3 ou superior. Exemplo de *digitama* sería o mostrado na Figura 2.2.

O tipo de carta máis frecuente no mazo serían as de *digimons* (Figura 2.3). Na esquina superior esquerda da carta podemos ver primeiro o custo de xogo (*play cost*) e debaixo os custos e condicións da evolución. No exemplo da Figura 2.3, o *digimon* podería evolucionar dun nivel 3 vermello ou negro pagando 2 de memoria. Na esquina superior dereita temos o *DP (Digi-*

Figura 2.3: Exemplo de carta de digimon

*mon Power).* No caso dunha batalla gañaría o que tivera maior *DP*. Existen efectos que poden reducir este valor e, en caso de que fora 0, o *digimon* sería eliminado e enviado á papeleira. Na imaxe do *digimon*, na carta e cun fondo translúcido, irían os efectos do propio *digimon*. Neste caso trátase dun efecto que se activaría ó ser xogado directo ó campo (*On Play*). Xusto debaixo atopamos na mesma liña o nivel, o nome, o código (EX3-007), o bloque[1] ó que pertence (02) e na liña negra máis pequena que o perfila teríamos as características (forma/atributo/tipo). O recadro inferior mostra a habilidade que proporcionaría ós que evolucionarán sobre esta carta.

As cartas de *tamers* só poden ser xogadas, non evolucionadas. Hai determinadas cartas de *digimon* que poderían evolucionar enriba deles (condición especificada na carta) e por iso temos os dous exemplos na Figura 2.4. Nun dos casos se proporciona unha habilidade herdada mentres que no outro emprégase o recadro para o efecto de seguridade (efecto que se activa cando é revelado na zona de seguridade). Os *tamers* non poden atacar nin bloquear.

O último tipo de carta a tratar son as *options.* Para usar estas cartas é necesario cumprir

[1] Nun futuro poderíase limitar as cartas a usar segundo o bloque

Figura 2.4: Exemplos de cartas de *tamers*. Esq.: Sen habilidade herdada. Der.: Con habilidade herdada.

as condicións de cor, é dicir, no campo de batalla ou na zona de cría debe haber polo menos unha carta que teña esa cor (en caso de *options* de dúas cores deben estar ambos presentes). Na Figura 2.5 mostramos un exemplo de *option*.

## 2.3 Mazos

Existen dous tipos de mazo, o mazo normal e o mazo de *digitamas*, cada un coas súas normas e limitacións.

O mazo normal debe conter 50 cartas (*digimons, options e tamers*) e o número de copias da mesma carta (mesmo código) non pode ser superior a catro.

O mazo de *digitamas* só pode conter este tipo de cartas e o número de cartas nel estará entre cero e cinco. Neste mazo aplica a mesma norma de que o número de cartas co mesmo código non supere as catro.

## 2.4 Fases da quenda

As quendas dunha partida desenvólvense en catro fases secuenciais. As fases teñen un inicio e fin claros e dentro de cada unha pódense realizar diferentes accións. En determinados casos unha quenda podería acabar sen pasar por todas as fases. As fases serían as seguintes:

1. Fase de enderezado: colócanse en posición vertical as cartas suspendidas [2].

2. Fase de roubo: róubase unha carta do mazo. Se non quedan cartas no mazo antes de roubar, o xogador perde a partida. Na quenda inicial non se roubaría durante esta fase.

[2] Unha carta suspendida estará colocada en posición horizontal e poderá ser obxectivo de ataques

Figura 2.5: Exemplo de carta de *option*

3. Fase de cría: nesta fase podemos realizar unha acción das seguintes ou non facer nada:
    - Abrir un *digitama*, poñer a primeira carta do mazo de *digitamas* boca arriba na zona de cría.
    - Mover un *digimon* da zona de cría á zona de batalla. Neste caso é necesario que teña DP.
4. Fase principal: pódense realizar calquera das seguintes accións tantas veces como se queira e sen necesidade de seguir unha orde específica:
    - Xogar un *digimon* ou *tamer*: colocaríamos a carta enderezada no campo desde a man e pagaríamos o seu custo.
    - *Digievolucionar*: colocaríamos unha carta desde a nosa man enriba de outra presente no campo que cumpra as condicións de *digievolución* e pagaríamos o custo de *digievolución*. Se unha carta ten múltiples requisitos soamente será necesario cumprir un deles. Despois de *digievolucionar* roubamos unha carta do mazo.
    - Usar cartas de *option*: para xogar estas cartas é necesario ter, polo menos, un *tamer ou digimon* da mesma cor. Situaríamos a carta no campo, pagaríamos o custo e activaríamos o seu efecto principal (*[Main]*). Salvo que se especifique o contrario a carta despois de usada vai á papeleira.
    - Atacar: para declarar un ataque debemos ter un *digimon* enderezado na zona de batalla, suspendémolo e dicimos o obxectivo (pode ser o opoñente ou algún dos seus *digimons* suspendidos). No caso de batalla entre *digimons*, compárase o DP. O que teña menos é derrotado e enviado a papeleira. Se o ataque fora ó opoñente e tivera cartas na seguridade revelaríamos a primeira e activaríamos, se o tivera, o seu efecto de seguridade. Se a carta revelada é un *digimon* compárase o DP. Se o atacante resulta perdedor é eliminado. Despois desta comprobación a carta da seguridade irá á papeleira.

Se en calquera das fases o contador de memoria pasa ó lado rival iniciaríase o fin da quenda e, despois de resolver os efectos pendentes, pasaría a ser a quenda do opoñente.

## 2.5 Memoria e quendas

Para representar a memoria utilizamos a barra de memoria. Esta barra pode vir xa debuxada no tapete como na Figura 1.1 ou ser independente como na Figura 2.6. Ningún xogador pode ter máis de 10 de memoria.

Durante a quenda pagaríamos memoria para xogar cartas ou *digievolucionar*. Se xogar unha carta custa tres moveríamos o contador de memoria tres ocos á dereita.

Figura 2.6: Barra de memoria independente

Cando o contador pasa ó lado rival e todos os efectos pendentes están resoltos iníciase a quenda rival. Se o contador queda no 0 a quenda continúa. É posible que, entre que o contador pase o lado rival e a resolución completa dos efectos, o contador volva ó teu lado ou caia no 0. En ambos casos non se considera a quenda finalizada e podes seguir realizando accións.

Debido a que ningún xogador pode ter máis de 10 de memoria, se queremos xogar algunha carta de custo 10 ou maior debemos ter memoria suficiente, é dicir, se temos o contador en 2 podemos xogar un custo 12 pero non un custo 13 xa que o contador caería en 11 de memoria para o rival.

Existe a posibilidade de que o xogador coa quenda activa decida pasarlla ó rival aínda que o contador siga no seu lado. Nestes casos o rival comezaría a súa quenda con 3 de memoria.

## 2.6 Efectos

Podemos distinguir entre efectos que se activan ó realizar determinadas accións ou efectos que son permanentes en campo. Na Figura 2.7 vemos ambos tipos, a habilidade herdada marcada como *[Your turn]* afectaría ó campo durante toda a quenda do xogador mentres que o efecto propio da carta activaríase no momento de *digievolucionar* nesa carta (*[When digivolving]*).

Pode ocorrer que múltiples habilidades se activen ó mesmo tempo. Nestes casos o xogador activo decide cal resolver primeiro mentres que o resto quedan pendentes. En caso de que despois de resolver un efecto se active unha habilidade nova, esta tería preferencia sobre as que estiveran pendentes de resolver. Noutras palabras, emprega unha cola Last In First Out (LIFO).

As condicións de activación máis frecuentes serían:

- *When digivolving*: actívase ó *digievolucionar.*
- *When attacking*: actívase ó declarar o ataque. Débense resolver antes de que o ataque se leve a cabo.
- *On play*: actívase ó xogar a carta.

Figura 2.7: *Digimon* con efecto activable e herdada permanente

- *On deletion*: actívase cando o *digimon* é eliminado.

As habilidades poden levar unha restrición de que soamente se poden activar unha vez por quenda (*[Once per turn]*).

## 2.7 Cartas de seguridade

Denominamos cartas de seguridade ás cartas colocadas boca abaixo na zona de seguridade e que podemos considerar como as proteccións do xogador. Soamente se revelan cando un xogador ataca directamente a un opoñente.

Cando o xogador activo declara un ataque á seguridade, revélase a primeira carta da seguridade rival e, en función da carta, podería ocorrer unha das seguintes situacións:

- A carta ten efecto específico de seguridade que se activa e resolve.

- A carta non ten efecto de seguridade pero trátase dunha carta de *digimon*. Neste caso comparamos o *DP* e se o do atacante fora menor este sería eliminado. A carta da seguridade vai á papeleira.

- A carta non ten efecto de seguridade nin se trata dun *digimon*, polo que vai directa á papeleira.

No xogo existen efectos polos que podemos, nun mesmo ataque, sacar varias cartas da seguridade rival (*[Security attack +1]*). Cando se dá esta situación as cartas revélanse dunha en unha e compróbase o anterior de cada vez. En caso de que o *digimon* atacante sexa eliminado antes de comprobar todas as que lle tocaba non se continúa.

Capítulo 3

# Tecnoloxías empregadas

DURANTE o seguinte capítulo presentaremos as tecnoloxías empregadas neste proxecto. Falaremos das ferramentas xerais e, despois, das especificas a cada parte do sistema ó que pertencen explicando as funcionalidades principais que nos aportaron.

## 3.1 Xerais

As ferramentas presentadas aquí non son específicas de ningún sistema pero nos aportaron soporte durante a planificación e o desenvolvemento.

- Trello: trátase dunha ferramenta para a xestión de proxectos mediante taboleiros e tarxetas. Empregámolo para estruturar as tarefas do desenvolvemento e realizarlles un seguimento.
- Git: trátase dun *software* que empregamos para levar control das modificacións que se ían realizando no desenvolvemento do proxecto.
- *Android Studio*: trátase dun IDE optimizado para o desenvolvemento de aplicacións en *Android*. Empregámolo para realizar o desenvolvemento da aplicación móbil.
- *PyCharm*: trátase dun Integrated Development Environment (IDE) optimizado para o desenvolvemento en *Python*. Utilizouse no desenvolvemento do servidor.
- *Visual Studio Code*: trátase dun IDE con complementos que ofrecen soporte para múltiples linguaxes. Empregámolo para desenvolver o cliente web.

## 3.2 Servidor

No servidor utilizamos Python3 [1] como linguaxe principal xunto con varios frameworks e librarías que nos facilitarán múltiples tarefas.

### 3.2.1 Python3

Trátase dun linguaxe de programación multiplataforma na que escribimos o servidor. Permítenos usar múltiples paradigmas e, neste caso, decidimos empregar o de orientación a obxectos, tanto porque é a máis coherente nun servidor deste estilo como para manter a congruencia cos frameworks empregados.

Desde o principio do desenvolvemento decidiuse manter a versión 3.8 por compatibilidades e estabilidade á hora de utilizar librarías como *OpenCV*.

### 3.2.2 Django

Django [2] é un framework web orientado a desenvolvementos rápidos e limpos. Entre as ferramentas que nos proporciona temos o seguinte:

- Object Relational Mapping (ORM) permítenos empregar as nosas clases modelo para realizar as consultas á base de datos. Danos ademais unha capa de abstracción que nos permite cambiar de base de datos sen necesidade de modificar as consultas.
- Esquema de asociación entre Uniform Resources Locator (URL) e vistas.
- Interface de administración na que podemos rexistrar os modelos para poder realizar operacións sobre eles sen necesidade de crear vistas específicas.
- Comando de xestión por consola para iniciar o servidor de desenvolvemento, lanzar tests ou programar as nosas propias instrucións.
- Aplicacións que se poden engadir na configuración para obter máis características.

Este framework cumpre o patrón Modelo Vista Controlador (MVC) separando a capa de acceso a datos (clases modelo) da de visualización (vistas) e do controlador (asociación das URL a vistas).

### 3.2.3 Django Rest Framework (DRF)

Este framework consiste nunha aplicación de Django que nos permite orientar o servidor para cumprir o patrón REpresentational State Transfer (REST). A principal ferramenta que utilizamos é a parte de serialización que substitúe, por así dicilo, a parte de visualización que en *Django* recae nas vistas. Así podemos servir os recursos en formatos como JavaScript Object Notation (JSON) sen necesidade de presentar unha resposta en HyperText Markup Language (HTML).

Outra característica importante de DRF son a variedade de clases que nos ofrece, das que podemos herdar nos *serializers*, para prototipar rapidamente as operacións Create Read Update Delete (CRUD) sobre os nosos recursos [3].

### 3.2.4 Tesseract-OCR

Trátase dun sistema de recoñecemento óptico de caracteres ou Optical Character Recognition (OCR) [4] de código aberto. Ten soporte para multitude de linguaxes, entre elas *Python3* mediante a libraría *pytesseract*.

En combinación con expresións regulares empregámolo no proxecto para o recoñecemento dos códigos de identificación das cartas.

### 3.2.5 OpenCV

Trátase dunha libraría de código aberto con funcionalidades de visión artificial e aprendizaxe máquina. Conta con máis de 2500 algoritmos optimizados para o tratamento de imaxes [5]. Temos unha interface para *Python3* no paquete *openCV-contrib-python*.

O uso que lle daremos será para manipular a imaxe que lle pasaremos ó Optical Character Recognition (OCR) e obter os mellores resultados na identificación.

### 3.2.6 Channels

O framework *Channels* permite aumentar as capacidades de *Django* para manexar protocolos distintos de Hypertext Transfer Protocol (HTTP). De este xeito podemos xestionar conexións tanto asíncronas como síncronas [6].

Ó engadirse a Django modificará o comando de lanzamento do servidor para usar un propio que habilita o control de websockets. Usarémolos para enviar comandos de actualización á páxina web.

### 3.2.7 BeautifulSoup

Trátase dunha librería de *Python* que se emprega para extraer información de ficheiros HTML e XML. Durante o desenvolvemento empregouse para a extracción de información das cartas de páxinas externas.

## 3.3 Aplicación móbil

Á hora de deseñar a aplicación é importante chegar o maior número de usuarios polo que, en vez de usar o linguaxe nativo do sistema Android ou iOS, utilizamos *Flutter*.

Trátase dun framework de código aberto que a partir dun código base nos permite compilalo para múltiples plataformas (android, ios, web...) [7]. A linguaxe a empregar é Dart. Conta con multitude de paquetes para facilitar tarefas como, por exemplo, o acceso a ficheiros independentemente da plataforma [8].

Os paquetes importantes empregados foron:

- *Camera*: permítenos controlar a cámara en Android, iOS e navegador.
- *Bloc*: paquete que nos axuda a separar a lóxica de funcionamento da presentación.
- *Path*: proporciónanos acceso a ficheiros locais independentemente do sistema operativo.

## 3.4 Web

Para levar a cabo o cliente web usamos a linguaxe Javascript co apoio do framework de *VueJS* e o entorno de ejecución de *NodeJS*.

### 3.4.1 VueJS

*VueJS* é un framework de desarrollo de interfaces web centrado en compoñentes independentes e directivas para modificar o HTML en función do estado. Proporciona ademais múltiples enganches durante o ciclo de vida dos compoñentes por se fora necesario engadir ou modificar algún estado neses puntos [9].

Empregamos xunto con *VueJS* o framework *Vuetify* que nos axuda a manter un estilo similar en todos os puntos da aplicación. Ofrece ademais múltiples compoñentes visuais que se empregan habitualmente [10].

### 3.4.2 NodeJS

Trátase dun entorno de ejecución asícnrono para a linguaxe *Javascript*. Salvo que realicen operacións síncronas intencionadas pódese contar con que non se bloqueará o proceso, xa que case todas as operacións de E/S son realizadas indirectamente.

Para a xestión de paquetes neste entorno empregamos Node Package Manager (NPM).

Capítulo 4

# Metodoloxía e planificación do proxecto

Ó largo deste capítulo mostraremos como se planificou este proxecto. Explicaremos a metodoloxía escollida e como a adaptamos ás nosas necesidades.

## 4.1 SCRUM

A metodoloxía escollida para o desenvolvemento deste proxecto foi *SCRUM*. Trátase dunha metodoloxía áxil moi empregada actualmente. As súas características principais son as seguintes:

- Desenvolvemento en iteracións de curta duración (entre unha e catro semanas), coñecidas como sprints.
- Os requisitos defínense en historias de usuario. Cada unha delas describe unha funcionalidade do noso sistema.
- Múltiples entregas dun sistema funcional, é dicir, cada fin de sprint produce un resultado que se pode valorar.
- Reunións diarias de quince minutos do equipo de desenvolvemento para comentar avances e erros que se van atopando.
- Alta colaboración do cliente no desenvolvemento ó irlle entregando os resultados visibles ó final de cada sprint.

*SCRUM* non se considera unha metodoloxía estrita xa que permite coller os elementos que máis favorezan ó equipo. Con isto en mente, quedámonos cos sprints, a definición de

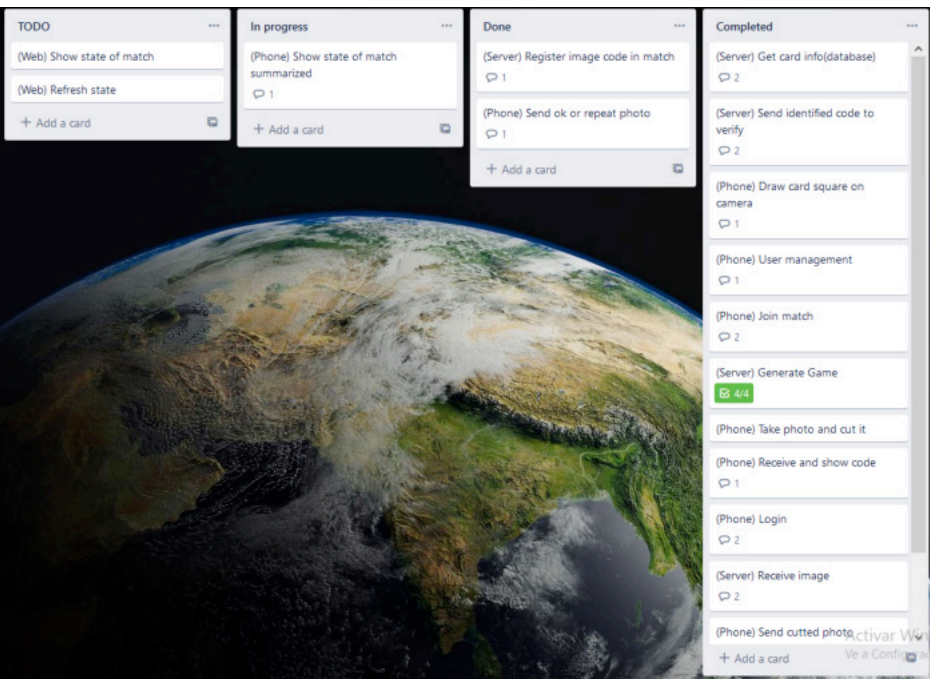


Figura 4.1: Estado do taboleiro de *Trello* durante o desenvolvemento

requisitos en historias de usuario, as entregas continuas (xa que neste proxecto somos á vez equipo de desenvolvemento e cliente) e as reunións de final de sprint coa directora do traballo.

Como apoio durante o desenvolvemento empregouse a ferramenta Trello para levar os rexistros das historias de usuario e os sprints. Na Figura 4.1 podemos ver o taboleiro de desenvolvemento do proxecto.

## 4.2 Planificación

Nesta sección mostraremos as historias de usuario recollidas e, en cada subsección correspondente, veremos como foron repartidas.

Xa que queremos ter un produto funcional ó final de cada sprint desenvolvéronse as distintas partes en paralelo, priorizando as partes que conteñen máis lóxica como son o servidor e a aplicación móbil.

As historias de usuario recollidas e separadas por subsistema podémolas ver na Táboa 4.1. A continuación irémolas repartindo nos *sprints*

O *sprint 0* trátase de maneira especial xa que é previo o inicio en si do proxecto. Durante este *sprint* as historias que se levarán a cabo serán as seguintes:

- Estudo do problema
- Análise das cartas

Táboa 4.1: Historias de usuario

| Servidor | Aplicación móbil | Cliente web |
|---|---|---|
| Xestionar usuarios | Iniciar sesión | Barra de busqueda partida |
| Xestionar partidas | Unirse a partidas | Mostrar partida |
| Recibir imaxes | Configurar cámara | Actualizar estado partida |
| Identificar códigos de cartas | Sacar e recortar fotos | |
| Verificar código | Enviar fotos | |
| Rexistrar carta en partida | Confirmar/modificar código | |
| Obter información cartas(BD) | Mostrar estado partida | |
| *Websockets* | Xestionar *digimons* | |

- Primeiras probas de identificación
- Configuración dos entornos de traballo
- Recollida de requisitos

Unha vez recollidos os requisitos e asignados ós seus correspondentes *sprints* debuxamos o diagrama de Gantt (Figura 4.2). Con isto podemos apreciar de maneira visual a duración do proxecto e de cada unha das tarefas.

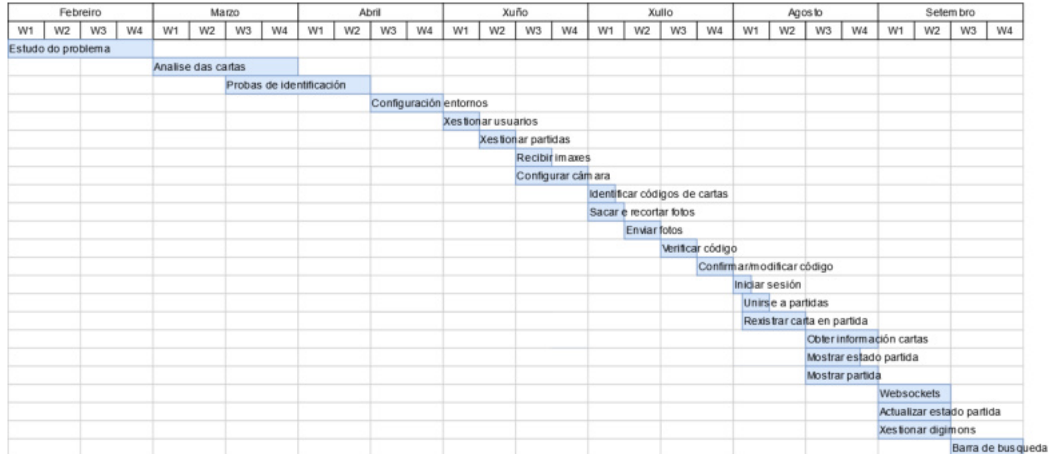


Figura 4.2: Diagrama de Gantt

O primeiro sprint centrarase nas tarefas iniciais de xestión na parte do servidor, en concreto:

- Xestionar usuarios: crear a URL de inicio de sesión e a configuración da autenticación e autorización.
- Xestionar partidas: configurar un *endpoint* para a creación e listado das partidas.

Durante o segundo sprint empezaremos a desenvolver a aplicación móbil e configuramos a parte do servidor para recibir e almacenar as imaxes. Isto implica as tarefas:

- Recibir imaxes: crear un *endpoint* que reciba imaxes e xestione o seu almacenamento no servidor.
- Configurar cámara: configuración do *plugin* de cámara na aplicación móbil e creación dunha ventá que a empregue.

Coa finalización do terceiro sprint obteremos unha aplicación móbil capaz de enviar fotos recortadas das cartas ó servidor e que este nos devolva o código identificado. As tarefas que se realizarán serán as seguintes:

- Identificar códigos de cartas: engadir o código obtido no *sprint 0* para identificar as imaxes que se envíen ó *endpoint* de recepción de imaxes do servidor.
- Sacar e recortar fotos: engadir funcionalidade para previsualizar o centrado da carta, á hora de obter unha foto, e o seu almacenaxe temporal no dispositivo móbil.
- Enviar fotos: configurar o servizo na aplicación móbil para que realice peticións ó servidor coas imaxes a identificar.

No cuarto sprint realizaremos as tarefas que engaden validación ó comportamento e resultados do sistema, é dicir, as seguintes historias de usuario:

- Verificar código: empregar a expresión regular apropiada para validar o código identificado no servidor. Ou, en caso de que non sexa válido, obter a opción máis probable que si o fora.
- Confirmar/modificar código: engadir, na aplicación móbil, funcionalidade para que o código da carta recibido do servidor poida ser modificado ou aceptado como correcto.

Durante o sprint número cinco engadiremos, tanto no móbil como no servidor, as funcionalidades que lles faltaban de cara ás partidas:

- Iniciar sesión: crear a pantalla de inicio de sesión na aplicación móbil e engadir a petición necesaria ó servidor.
- Unirse a partidas: engadir a opción, na pantalla principal, para que o usuario poida unirse a partidas (aplicación móbil). Isto implica crear a petición necesaria ó servidor

- Rexistrar carta en partida: engadir funcionalidade, cun *endpoint* no servidor, para que poida recibir o código da carta e rexistralo como carta xogada na partida correspondente.

O sexto sprint centrarase en engadir información visual á partida no móbil e no cliente web. Para poder ver información visual relevante neste punto será necesario ter a información das cartas xa almacenada na base de datos. As tarefas que se levarán a cabo serán as seguintes:

- Obter información cartas (BD): crear dun *script* que nos permita obter os datos das cartas, entre eles unha URL para mostralas.
- Mostrar estado partida (Móbil): engadir á pantalla de partida unha zona na que visualizar as cartas xogadas polo usuario en dita partida.
- Mostrar partida (Web): crear a principal pantalla do cliente web co estado da partida.

O sprint número sete dedicarémolo á parte de automatización das actualizacións. Ademais engadiremos comportamentos do xogo para que se puideran realizar dende a aplicación móbil. As tarefas das que consta este *sprint* son:

- Comunicación mediante *Websockets*: engadir soporte para o uso de *sockets* mediante a librería *Channels*, xestionar o *socket*, así como configurar a mensaxe de renovación e o sinal que a lanzará.
- Xestionar *digimons*: engadir comportamentos específicos do xogo na aplicación móbil.
- Actualizar estado partida: configurar a creación do *socket* co servidor e o comportamento á hora de recibir as mensaxes.

O último sprint dedicarémolo a realizar a tarefa que nos queda pendente, isto é, desenvolver unha barra de búsqueda de partida, a cal permitirá introducir o identificador dunha partida para proceder a súa visualización.

## 4.3 Seguemento proxecto

Importante destacar que o *sprint 0* comeza en febreiro do ano 2022 cando se planea a idea inicial do proxecto e se comeza o seu estudo e acotación. No diagrama de Gantt da Figura 4.3 podemos ver cal foi a duración real do proxecto.

No *sprint* 0 levounos máis tempo do estimado a análise das cartas e realización das probas de identificación debido a cantidade de imaxes que foi necesario obter e recortar, por iso ambas tarefas se prolongaron ata xullo. Durante o mes de setembro, por circunstancias persoais, produciuse un atraso no desenvolvemento de tres semanas. Estes atrasos xunto con algúns

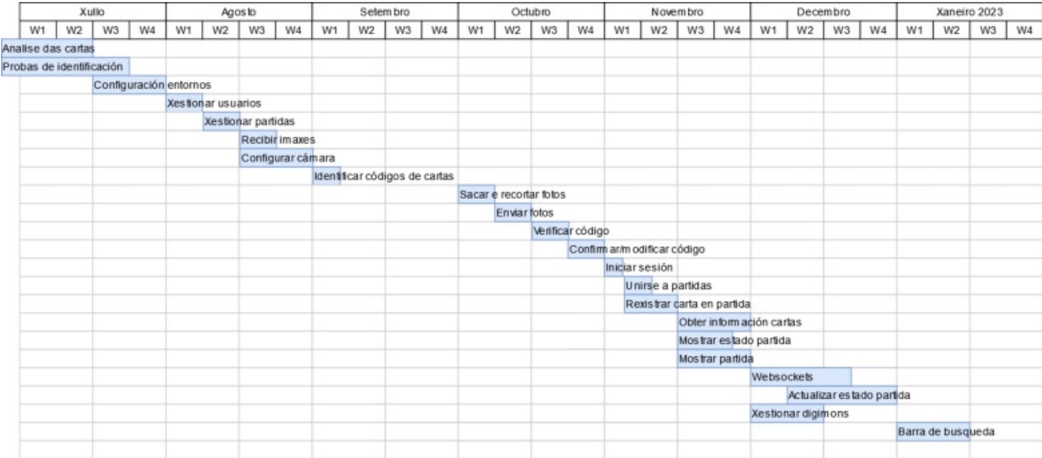


Figura 4.3: Diagrama de Gantt coa duración real

problemas á hora de engadir o soporte para *websockets* no servidor implicou que o proxecto atrasara a súa finalización ata principios de xaneiro de 2023.

## 4.4 Custos

Para este proxecto estimouse unha duración do desenvolvemento de 16 semanas. Durante estas semanas dedicaríase un único traballador, traballando durante 4 horas diarias, 6 días á semana. Obtendo un total de 384h.

Estimando o custo do traballador en 10 euros por hora traballada obtemos que o desenvolvemento do proxecto eleva o seu custo a 3840 euros.

Ó final o proxecto alargouse máis aló do estimado o que implicou un aumento seu custo. O desenvolvemento durou 22 semanas o que implicou un aumento de duración de 144h. O sobrecusto disto foron 1440 euros, dando un custo real do proxecto de 5280 euros.

## Capítulo 5

# Deseño do sistema

NESTE capítulo explicaremos o deseño que segue este sistema. Presentaremos a arquitectura seguida polo sistema completo para, a continuación, mostrar en detalle cada unha das súas pezas.

## 5.1 Arquitectura do sistema

Como arquitectura para o sistema escóllese empregar o estándar cliente-servidor. Esta arquitectura é a que mellor se adapta ás nosas necesidades separando claramente os nosos clientes (aplicación móbil e cliente web) do servidor. As vantaxes que nos proporciona este sistema son as seguintes:

- Centralización dos recursos: a integridade dos datos e o acceso a eles contrólase no servidor. Isto evita que un cliente poida realizar modificacións non autorizadas. Ademais facilita a modificación ou actualización dos datos.
- Escalabilidade: podemos aumentar os elementos por separado.
- Seguridade: múltiples tecnoloxías cun amplo percorrido neste paradigma melloran a seguridade nas transaccións.
- Mantemento sinxelo: a modificación dun servidor non implica, necesariamente, que os clientes se vexan afectados.

Na Figura 5.1 podemos observar a arquitectura completa. Durante as seguintes seccións presentaremos cada unha das súas partes.

## 5.2 Base de datos

O esquema deseñado para a base de datos podémolo observar na Figura 5.2. As táboas mostradas son as que creamos para a nosa aplicación sen incluír as propias de *Django*, a

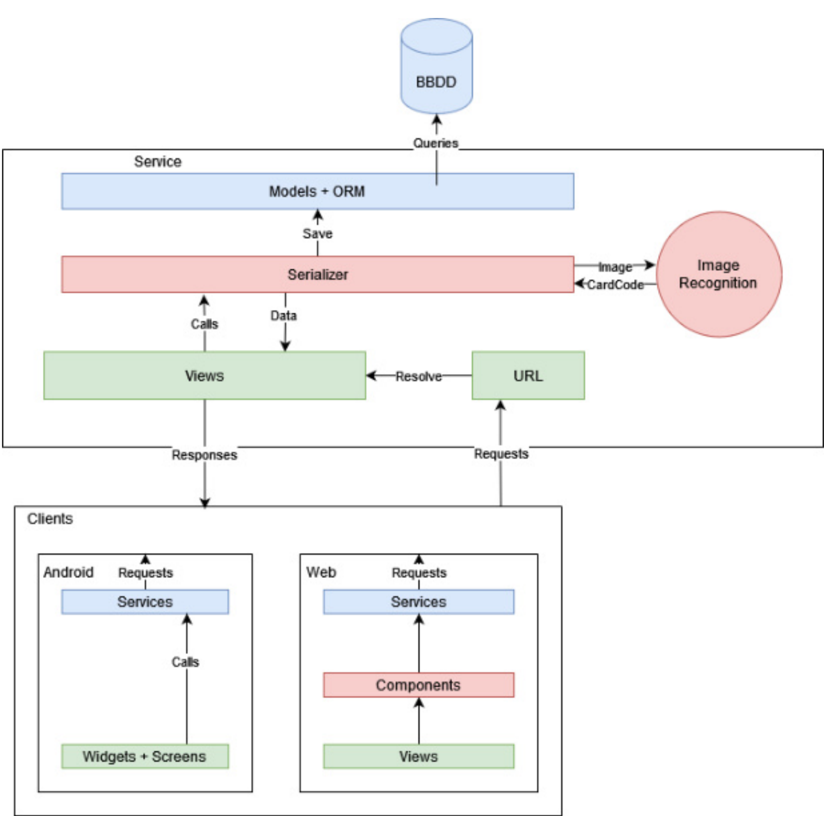


Figura 5.1: Arquitecura do proxecto

excepción da táboa *auth_user*. Os propósitos de cada táboa serían os seguintes:

- *dtcg_card*: almacena a información das cartas do xogo.
- *dtcg_game*: almacena a información das partidas.
- *dtcg_game_players*: almacena as relacións entre os usuarios e as partidas nas que participan.
- *dtcg_played_cards*: garda información das cartas xogadas, a partida na que se xogaron e o usuario que a xogou.
- *auth_user*: almacena a información dos usuarios. Esta táboa é creada por Django. No noso caso non se modificou pero poderiámola estender se quixeramos almacenar máis información dos usuarios.

Xa que todos os accesos a base de datos se realizan a través do ORM de *Django* non estamos atados a ningún sistema de xestión de bases de datos concreto. Durante o desenvolvemento empregamos SQLite pola facilidade para recrear a base de datos. Nun futuro poderíamos empregar outro con mellor rendemento como, por exemplo, PostgreSQL

## 5.3 Servizo

O servizo REST desenvolto sigue o patrón MVC á hora de separar cada capa. A continuación vemos a relación entre os ficheiros xerados de base nun proxecto de *Django* e este patrón:

- Vista: encargada de representar o estado actual do sistema. Relaciónase cos ficheiros *urls.py e view.py*.
- Controlador: actúa de intermediario entre a acción realizada polo usuario contra a vista e o resto do sistema. Tamén se encargaría de transformar datos para adaptalos ó modelo ou á vista. Relacionado co ficheiro *serializers.py*.
- Modelo: encargado de gardar a información do sistema aplicando as restricións que correspondan. Relaciónase co ficheiro *models.py*.

Nas Figuras 5.3, 5.4 e 5.5 podemos observar o diagrama de clases do proxecto, separados por ficheiros para que fora lexíbel. No diagrama do modelo podemos ver que os tipos non son tipos básicos (*char,integer...*) senón encapsulacións dos mesmos proporcionadas por *Django*. En caso de empregar outras librerías deberiamos modificar estes polos seus tipos

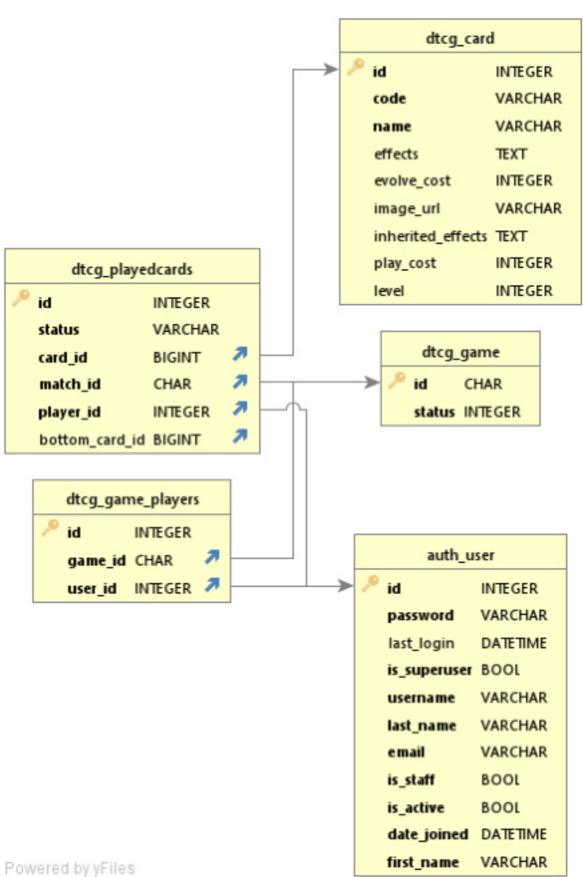


Figura 5.2: Esquema base de datos

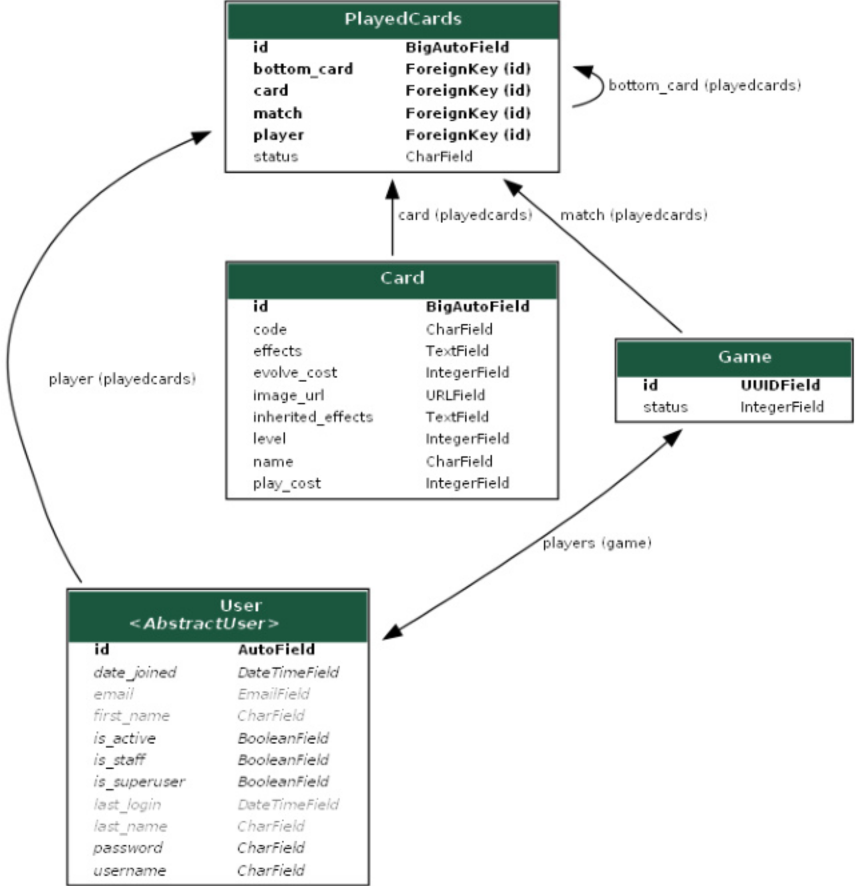


Figura 5.3: Diagrama de clases do modelo

básicos correspondentes e engadir funcións para realizar consultas á base de datos. Estas funcionalidades veñen proporcionadas pola clase *Model* de *Django* da que herdan todos os nosos modelos.

No diagrama da Figura 5.4 temos os *serializers* empregados. Os que herdan do *ModelSerializer* só requiren que se lles proporcione a clase modelo para xerar os seus campos de maneira automática, pero poderíanse limitar os campos que queremos expoñer. Por outro lado, os que herdan da clase *Serializer* non teñen un reflexo directo no modelo, polo que se definen os seus campos e se sobrescribe o método *create* da clase pai para xestionar o comportamento ó gardar. As chamadas ó procesamento de imaxes realízanse dende o *create* do *CardImageSerializer*.

A Figura 5.5 mostra as vistas do servizo. Por un lado as que herdan de *ListAPIView* xeran automaticamente o comportamento de listado como resposta a unha petición *GET*. Só precisan definir o *queryset* (listado a mostrar) e a clase *serializer* que empregarán para mostrar os resultados. As que herdan de *APIView* permítennos definir os comportamentos para cada tipo

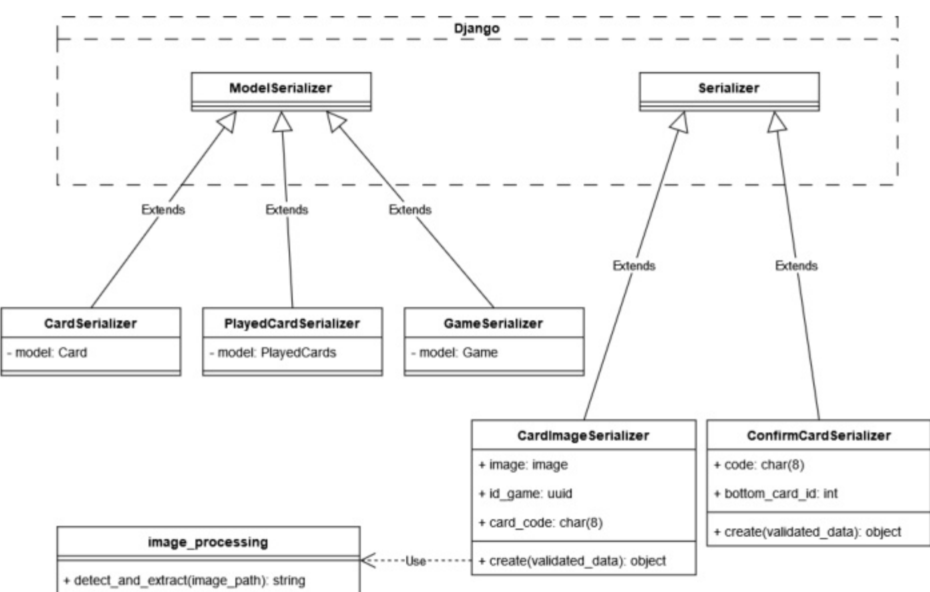


Figura 5.4: Diagrama de clases dos *serializers*

de petición.

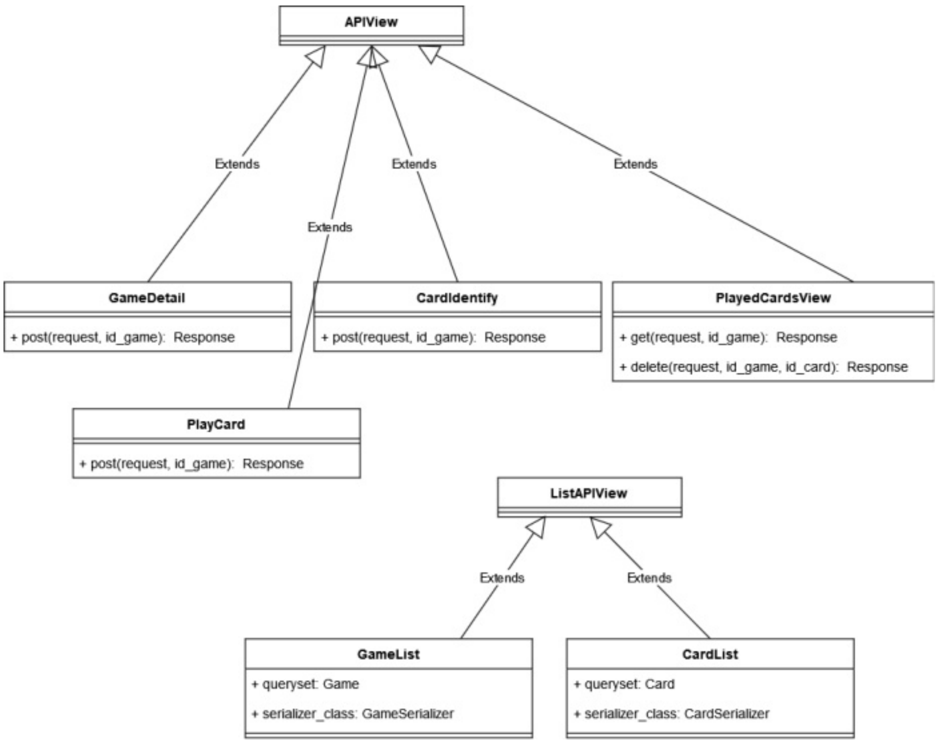


Figura 5.5: Diagrama de clases das vistas

O servizo emprega a tecnoloxía de websockets para enviar mensaxes de actualización ó

cliente web. Isto modifica o esquema de peticións e respostas, quedando coma se mostra na Figura 5.6. A capa *channels* encárgase de manexar a petición e dirixila pola canle adecuada. Desta maneira as peticións HTTP son dirixidas ás vistas estándar mentres que as de websockets son tratadas polo consumidor asíncrono adecuado.

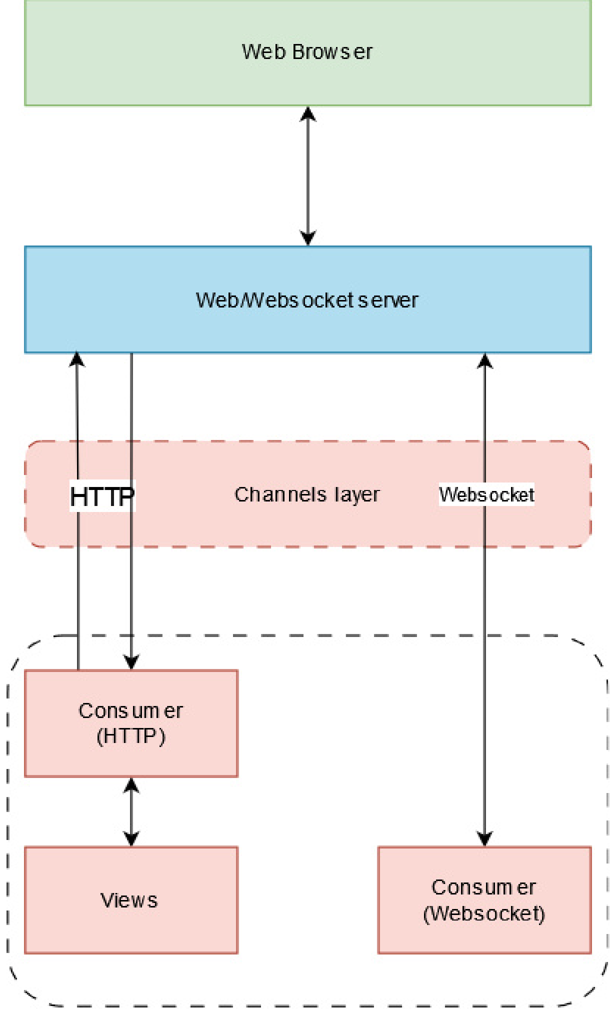


Figura 5.6: Esquema de peticións e respostas

Estes *sockets* son agrupados segundo a partida. Cando se realiza algunha modificación nas cartas xogadas lánzase un sinal coa información desa carta, obtemos o identificador da partida e envíase a petición de renovación de datos a todos os *sockets* que pertenzan a ese grupo. Na Figura 5.7 podemos ver o comportamento dende o punto de vista so servizo. Cando o usuario se conecta a unha partida no cliente web realízanse os seguintes pasos:

- O cliente web solicita os datos da partida mediante unha petición HTTP.
- O servizo executa a vista, obtén os datos, dálles formato e envía a resposta.

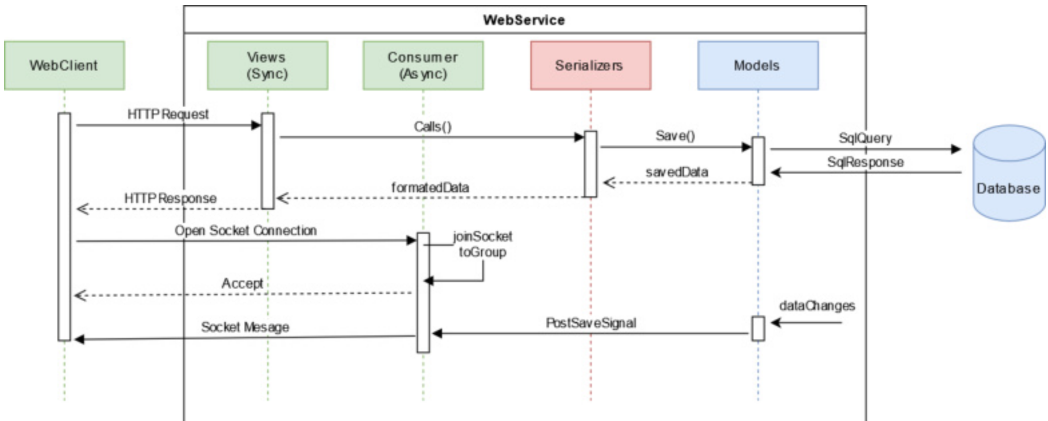


Figura 5.7: Fluxo de traballo do servizo

- O cliente web recibe a resposta e inicia unha conexión por *socket*.
- O servizo acepta a conexión por *socket* e engádeo ó grupo da partida.
- Cando a información das cartas xogadas na partida cambie, o modelo emitirá un sinal que será procesado no *consumer* e enviará unha mensaxe de renovación ós *sockets* apropiados. A mensaxe de renovación de datos e a única que enviamos a través do *socket*.

Do observado na Figura 5.7 cabe explicar, a maiores do xa visto, que:

- As chamadas ó procesado de imaxes realízanse no *serializer*.
- O sinal enviado polo modelo soamente ocorre se o cambio é realizado a través do propio modelo, é dicir, se modificáramos a base de datos directamente dito sinal non se emitiría.

Nun futuro poderíase aumentar a información que se envía a través do *socket* en busca de melloras de rendemento.

### 5.3.1 Endpoints

Neste apartado imos ver as distintas peticións que acepta o servidor e como se asocian coas vistas da Figura 5.5.

Na Táboa 5.1 podemos observar como crear unha partida ou obter un listado delas. Para obter o listado empregaríamos unha petición *GET* e, para creala, un *POST*. Este *endpoint* asociaríase a vista *GameList*.

A petición para engadir xogadores vémola definida na Táboa 5.2. Mediante o *POST* envíase unha petición baleira, o servizo encárgase de identificar o usuario que a realiza analizando

Táboa 5.1: Definición do *endpoint* de partidas

| | |
|---|---|
| Url | /games |
| Descrición | Listar e crear partidas |
| Tipo petición | GET/POST |
| Parámetros | {} |
| Resposta | Devolve a información das partidas ou da partida creada |

Táboa 5.2: Definición do *endpoint* para engadir xogadores

| | |
|---|---|
| Url | /games/<uuid:id_game>/addPlayer |
| Descrición | Engade ó xogador que realiza a petición á partida |
| Tipo petición | POST |
| Parámetros | {} |
| Resposta | Devolve a información da partida ou un erro se estivera completa |

as cabeceiras, e engádeo á partida. Esta petición devólvenos un código de erro se a partida estivera completa. Asóciase á vista *GameDetail*.

Para engadir cartas a unha partida empregaríamos a petición mostrada na Táboa 5.3. Esta petición acepta o código identificativo da carta a engadir e o identificador da carta sobre a que se esta xogando, se a houbese. Devolveríanos o código e o nome da carta engadida. Asóciase á vista *PlayCard*.

Táboa 5.3: Definición do *endpoint* para engadir cartas

| | |
|---|---|
| Url | /games/<uuid:id_game>/addCard |
| Descrición | Engade unha carta á partida |
| Tipo petición | POST |
| Parámetros | {code, bottom_card_id} |
| Resposta | {code, card_name} |

Na Táboa 5.4 podemos ver detalles da petición para obter as cartas dunha partida. Asóciase á vista *PlayedCardsView*.

A petición para eliminar cartas da partida vémola definida na Táboa 5.5. Ó igual que a anterior asóciase á vista *PlayedCardsView*. Esta petición eliminará unha carta que xa fora

Táboa 5.4: Definición do *endpoint* para ver as cartas dunha partida

| | |
|---|---|
| Url | /games/<uuid:id_game>/cards |
| Descripción | Obtén as cartas da partida |
| Tipo petición | GET |
| Parámetros | {} |
| Resposta | Devolve a información das cartas xogadas na partida |

xogada, o que modificará o estado das cartas que tivera relacionadas.

Táboa 5.5: Definición do *endpoint* para eliminar unha carta

| | |
|---|---|
| Url | /games/<uuid:id_game>/cards/<int:id_card> |
| Descripción | Elimina unha carta da partida |
| Tipo petición | DELETE |
| Parámetros | {} |
| Resposta | Devolve un código de ok ou de erro se non existe a carta |

Na Táboa 5.6 podemos ver a definición da petición de identificación. Para esta petición é necesario realizar un *POST* dunha imaxe, o código da partida sería opcional.

Táboa 5.6: Definición do *endpoint* de identificación de cartas

| | |
|---|---|
| Url | /identify |
| Descripción | Identifica a carta na imaxe enviada |
| Tipo petición | POST |
| Parámetros | {image, id_game} |
| Resposta | Devolve o código identificado |

A petición de inicio de sesión da Táboa 5.7 é xestionada por *Django*.

### 5.3.2 Procesamento de imaxe

Neste apartado veremos como se realiza o procesado dunha imaxe para obter o seu código identificativo (Figura 5.8). Unha vez que chega unha imaxe comprobamos se pertence a unha partida ou non. En caso de facelo gárdase no directorio específico desa partida. En caso

Táboa 5.7: Definición do *endpoint* de inicio de sesión

| Url | /login |
|---|---|
| Descripción | Inicio de sesión |
| Tipo petición | POST |
| Parámetros | {user, password} |
| Resposta | Devolve un ok ou un código de erro se o usuario non é correcto |

contrario iría a unha carpeta de imaxes independente. A continuación detectamos a rexión da imaxe na que se atopa o código da carta. Unha vez temos a rexión procesámola co OCR para obter o texto. Para mellorar os resultados obtidos usamos unha expresión regular sobre o texto recibido para que coincida cos patróns dos códigos.

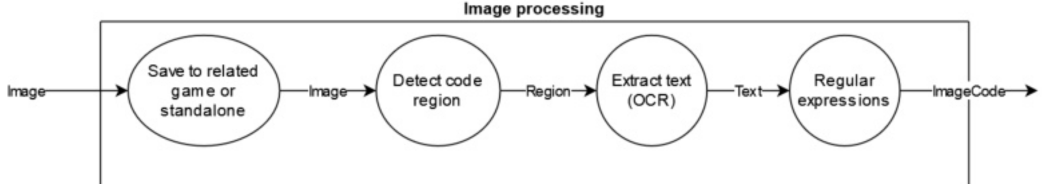


Figura 5.8: Procesamento de imaxe

## 5.4 Cliente Web

O esquema de funcionamento vémolo na Figura 5.9. O usuario só precisa acceder a *url* da partida. Iniciase a petición dos datos da partida e ábrese o *socket* co servidor. A partir deste punto non se require máis acción por parte do usuario na web xa que as actualizacións de información realizaranse automaticamente cando o servidor o comunique polo *socket*.

Neste cliente o usuario só pode realizar dúas accións, ou empregar a barra de búsqueda para cambiar de partida ou facer clic nun dos *digimons* para ver a información de maneira máis detallada.

No diagrama de clases da Figura 5.10 podemos observar como se dividiu a parte visual en compoñentes máis simples. Un campo de partida contaría con dous xogadores cada un cos seus *digimons*. Cada *digimon* está composto por unha lista de cartas ordenada e ten o nome da carta superior.

Podemos observar na Figura 5.11 como se vería o cliente web durante a visualización dunha partida en curso. Podemos ver que o compoñente *NavBar* estaría colocado na parte superior e nos permitiría ir cambiando de partida. O compoñente *Field* abarcaría o resto,

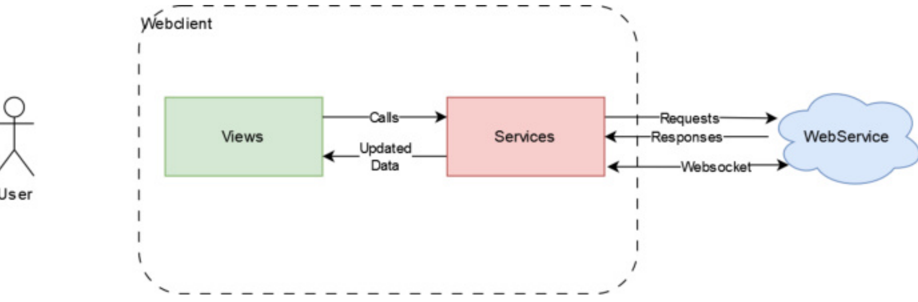


Figura 5.9: Execución cliente web

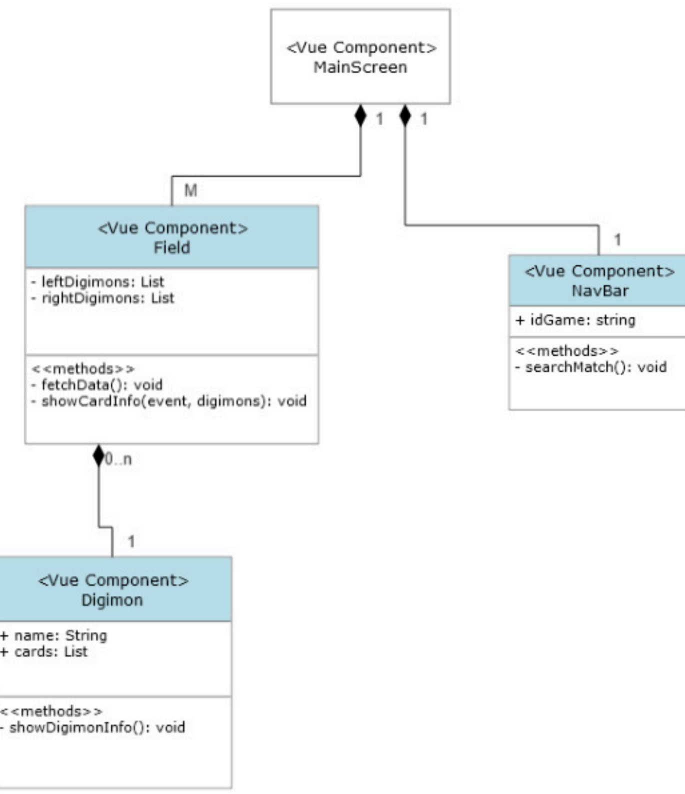


Figura 5.10: Diagrama de clases do cliente web

dividido en tantos compoñentes *Digimon* como fora necesario. Facer clic en calquera dos *digimons* nos permitiría consultar os detalles máis claramente como se ve na Figura 5.12.

Outro punto importante do cliente sería a apertura do *socket* co servidor e como se xestiona a automatización das actualizacións. Dende o punto de vista do cliente quedaría como na Figura 5.13. Despois de que o usuario entre nunha partida realízase a primeira petición de datos e a apertura do *socket*. A continuación, cada vez que se reciba un mensaxe de renovación polo *socket*, volveríase realizar a petición de datos e actualizaríase a vista coa nova

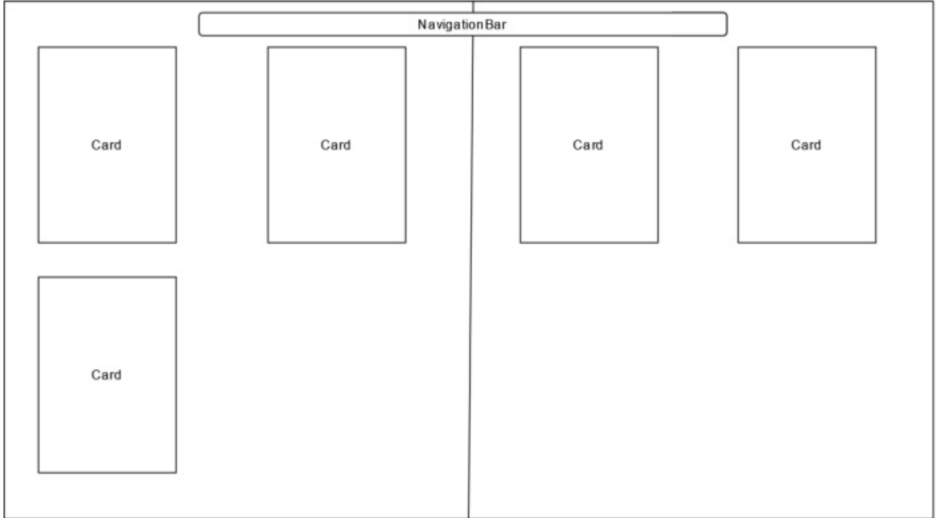


Figura 5.11: Bosquexo do cliente web

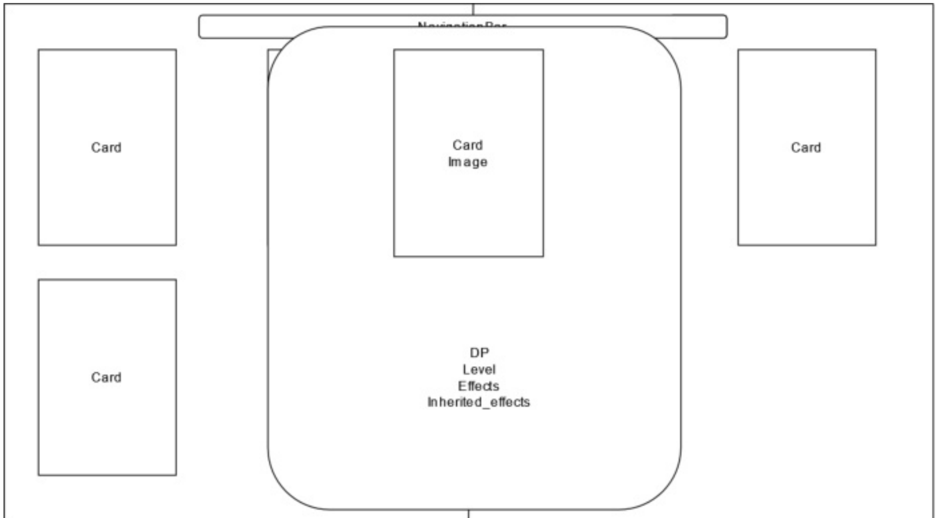


Figura 5.12: Bosquexo do cliente web despois de facer clic nun *digimon*

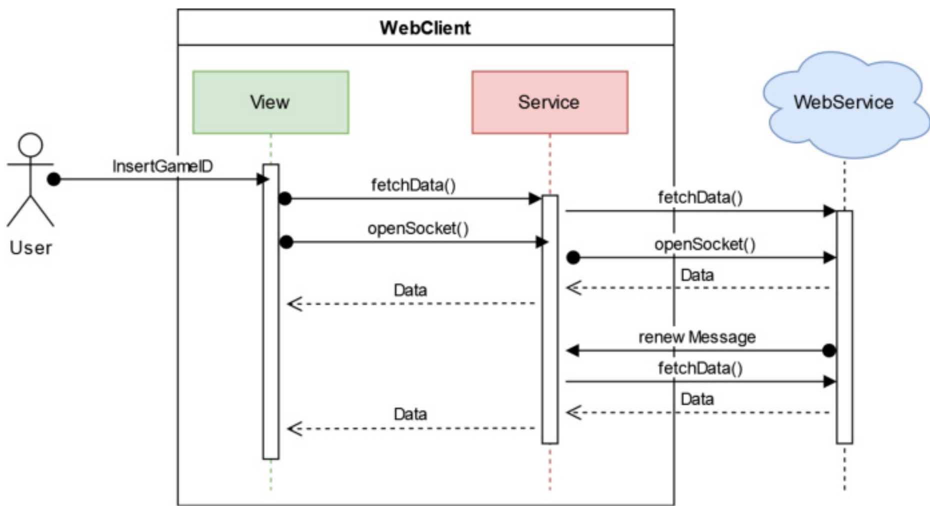


Figura 5.13: Fluxo de traballo do cliente web

información.

## 5.5 Aplicación móbil

A aplicación móbil estructúrase en clases da linguaxe Dart que extenden de clases *widget*. Dentro destas construímos a parte visual e a lóxica de control dos botóns, campos de texto e demais elementos, derivando a lóxica máis complexa cara servizos e modelos externos. Na Figura 5.14 podemos ver a execución dunha acción na aplicación.

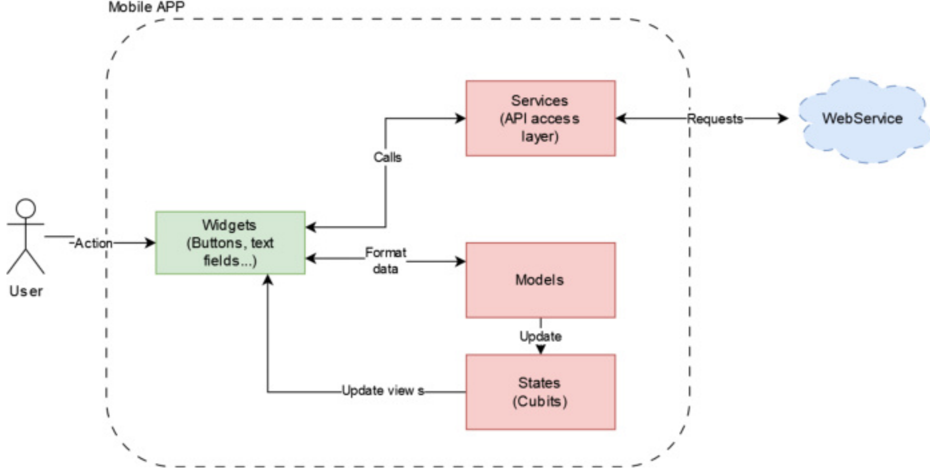


Figura 5.14: Acción en *Flutter*

Durante a execución da aplicación o usuario pasará por varias pantallas diferentes. Unha vez se inicia a aplicación tería que iniciar sesión (Figura 5.15). A continuación no menú prin-

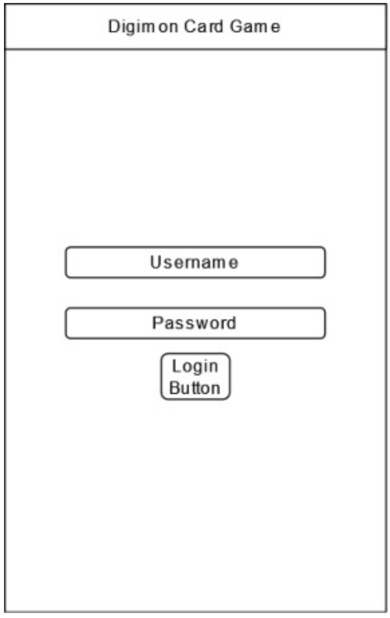


Figura 5.15: Bosquexo da pantalla de inicio de sesión

cipal (Figura 5.16) xa poderemos unirnos a unha partida ou creala. Ademais temos a opción de identificar cartas.

En caso de crear unha partida engádese automaticamente ó creador como xogador e pasaríamos directamente a pantalla de partida (Figura 5.17). Para unirnos a unha partida ingresaríamos o código nun campo de texto e iríamos a mesma pantalla. Desde a pantalla de partida poderíamos xogar novos *digimons* pulsando o botón flotante, isto levaríanos a pantalla de toma de fotos (Figura 5.18). No caso de querer *digievolucionar* as nosas cartas en xogo poderíamos facer clic no *digimon* e seleccionar a opción.

Despois de tomar a foto iríamos a unha pantalla similar a da Figura 5.18 que nos mostrará a foto xa recortada ó tamaño do recadro. Dende aquí poderíamos enviar a foto ou descartala e sacar outra.

Para levar a cabo esta parte do proxecto seguimos o diagrama de clases visto na Figura 5.19. No diagrama podemos apreciar dous tipos de pantallas, as que dependen do estado e a que non. As clases que dependen do estado herdan de *StatefulWidget* e delegan a súa creación e actualización ás clases *state*.

Para completar o diagrama de clases temos a Figura 5.20 na que mostramos as clases modelo e a clase servizo que se emprega para realizar peticións a *API*.

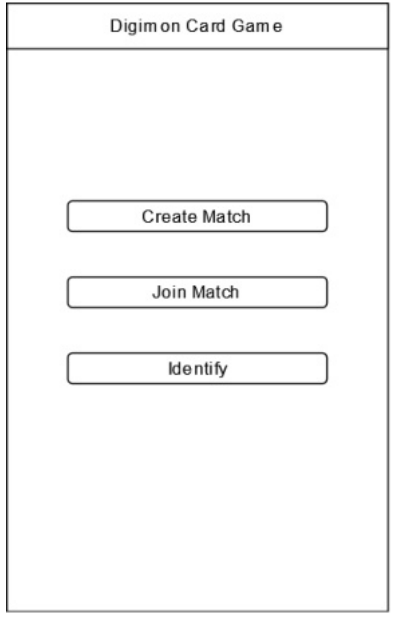


Figura 5.16: Bosquexo do menú principal

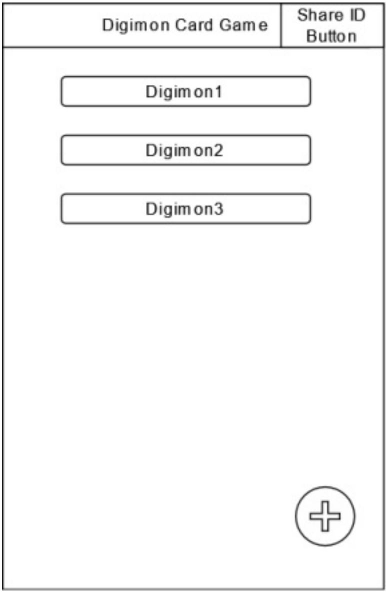


Figura 5.17: Bosquexo da pantalla principal de partida

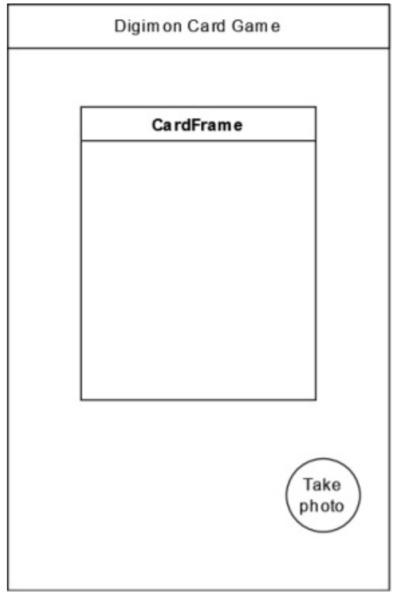


Figura 5.18: Bosquexo da pantalla de toma de fotos

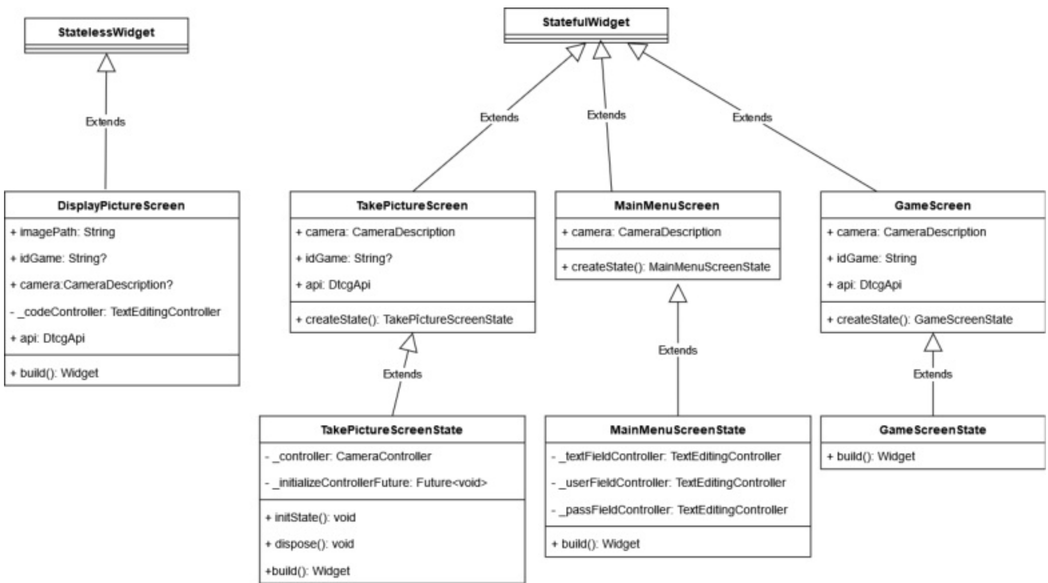


Figura 5.19: Diagrama de clases das pantallas da aplicación móbil

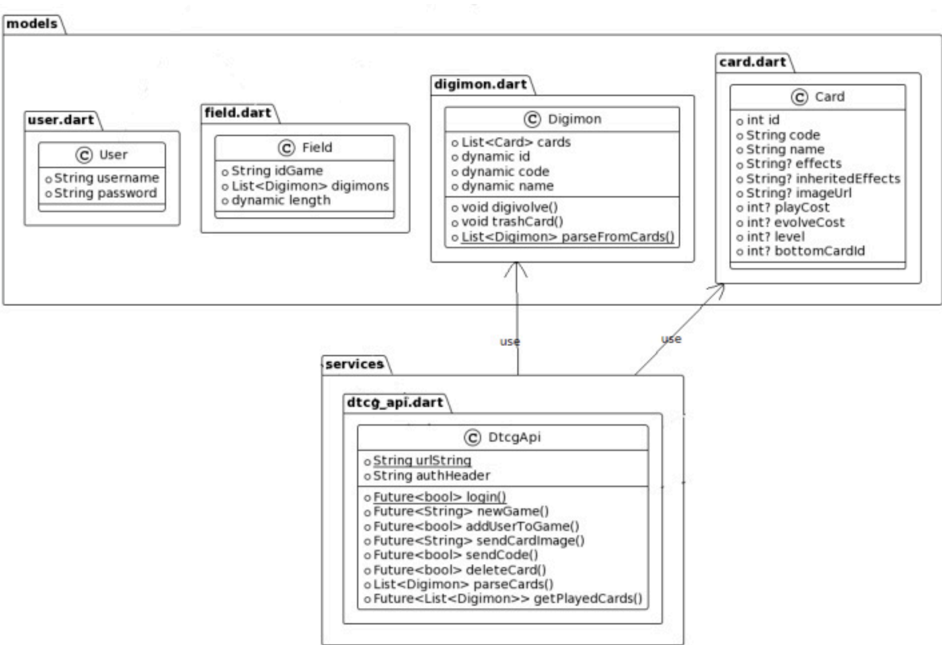


Figura 5.20: Diagrama de clases do modelo e do servizo

Capítulo 6

# Desenvolvemento

NESTE capítulo analizaremos como se realizou o desenvolvemento do proxecto. Comezaremos vendo como se estudou o problema e se fixeron as primeiras probas de procesamento de imaxe, para despois, tratar o desenvolvemento de cada un dos subsistemas.

## 6.1 Estudo do problema

O problema inicial considerado para o proxecto non só tiña en conta o xogo de cartas de *Digimon* senón que incluía outros como *Magic e Dragon Ball*. Isto limitouse bastante rápido xa que as cartas de xogo non se estruturan da mesma maneira.

Unha vez limitado ó xogo de *Digimon* xogáronse partidas na modalidade presencial e na *online* para ver as diferenzas e as posibles maneiras de solucionalas. Considérase unha primeira idea que empregase a mesma cámara web da chamada como o elemento que enviaría as imaxes. Aquí tiñamos dúas vías posibles e os motivos do descarte:

- Detectar as cartas directamente do terreo de xogo, recortalas e envialas a identificar. Descártase esta opción xa que se volve case imposible a detección das cartas con respecto ó fondo (non é necesariamente uniforme).
- Presentar a carta a xogar acercándoa á cámara. Aínda que obtiña mellores resultados que a anterior, atopamos problemas coas cámaras que non contan con autoenfocado. Ademais, as condicións de iluminación e inclinación da carta xogaban un factor moi decisivo de cara ó recoñecemento.

Un problema compartido por ambos métodos era a calidade de imaxe obtida polas cámaras. Coa pandemia do COVID-19 os prezos das cámaras web, tanto de boa como de mala calidade, aumentou notablemente. Isto ocasionou que moitas das partidas que se realizan *online* empreguen cámaras de baixa calidade que imposibilitaban obter bos resultados.

Figura 6.1: *Digimons* con debuxos similares

Como solución final preséntase o sistema en tres partes que vemos no proxecto. Solucionamos o problema de calidade das cámaras web ó empregar a cámara do móbil que, na maioría dos casos, ten mellores resolucións. Ademais permite controlar de mellor maneira a iluminación, os posibles reflexos e a inclinación das cartas.

## 6.2 Detección das cartas

Unha vez xa tiñamos definido o método para obter fotos das cartas de boa calidade realizamos a análise das cartas para a súa identificación. Descartamos empregar o nome das cartas xa que algúns se repiten con frecuencia ó largo das coleccións (*Agumon, Greymon...*). Os debuxos das cartas quedan descartados xa que para unha mesma carta poden existir varios distintos e cartas distintas poderían ter debuxos similares. Na Figura 6.1 podemos ver un exemplo de debuxos similares. A análise das cartas permítenos distinguir que traen un código identificativo que se sitúa, na maioría dos casos, na mesma posición. Este código será o que buscaremos para identificar a carta. Na Figura 6.1 podémolo ver, a dereita do nome, marcado en vermello.

As primeiras probas que realizamos implicaban recortar a imaxe da carta manualmente e procesala no OCR. Os textos recoñecidos variaban moito e soían mesturar o código co nome ou co recadro de diamante que ten debaixo.

Para evitar esta problemática buscamos delimitar a rexión a procesar para que só conteña o código. Isto lévanos a empregar a búsqueda de patróns na imaxe (template matching). Para detectar a zona do código empregamos unha imaxe modelo da parte que indica o nivel (Figura 6.2). Partindo desa zona detectamos a rexión que nos interesa.

A imaxe modelo que vemos na Figura 6.2 é a que empregamos de maneira máis habitual.

Figura 6.2: Imaxe modelo da zona de nivel

Pero atopamos tres casos nos que esta non nos da bos resultados. Os casos son os seguintes:

- As cartas de cor negra teñen o nivel en negro sobre branco.
- Cartas de *option* e *tamer*, non teñen a zona de nivel.
- Unha serie de cartas da colección *BT1.5* teñen a zona de nivel dourado sobre negro.

Para solucionar as do primeiro caso empregouse outra imaxe modelo que ten letras negras sobre fondo branco. E tanto para o segundo como para o terceiro e para os casos que escapen dos anteriores facemos detección posicional (tendo a carta recortada buscamos directamente a zona do código). No seguinte código vemos como se detectaría a zona nos casos que se empregan imaxes modelo:

```
def detect_code_region(file_path, template):

    im = cv2.imread(str(file_path), cv2.IMREAD_GRAYSCALE)
    rows = im.shape[0] * template.shape[0] // 1295
   # Height of the original image where template was cropped
    cols = im.shape[1] * template.shape[1] // 951
   # Width of the original image where template was cropped
    scaled_template = cv2.resize(template, (rows, cols))
```

```
    result = cv2.matchTemplate(im, scaled_template,
                               cv2.TM_CCOEFF_NORMED)

    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(result)

    unit_size = im.shape[0] * 1 / 9
    roi = im[maxLoc[1] - round(unit_size*0.4):maxLoc[1] +
             round(unit_size*0.2), im.shape[1] * 3 // 4:]

    return maxVal, roi
```

No código lemos a imaxe a identificar en branco e negro. Adaptamos a imaxe modelo para que as súas dimensións coincidan coas da rexión que estamos a buscar na foto. Buscamos as coincidencias empregando o método *TM_COEFF_NORMED* de OpenCV [11]. Calculamos o número de filas aproximado que equivalen a 1cm na carta. Sacamos a rexión que mellor resultado obtivo e a continuación obtemos, a partir da distancia estimada á rexión, a zona do código.

O parámetro *maxVal* poderíamolo empregar para determinar se nos fiamos da rexión (marca a confianza no resultado), pero neste caso resulta máis útil se procesamos a zona no OCR. Se non obtemos un código válido, utilizaremos outras maneiras de detectar a zona. A extracción de texto e a detección de se o código é válido recaería na seguinte función:

```
def extract_text(img):
    text = pytesseract.image_to_string(img)
    res = re.search(r'[BES]?[TX]\d{1,2}-\d{2,3}', text)
    if res:
        text = res[0]
        if text.startswith('X'):
            text = 'E' + text
        if text.startswith('T'):
            if len(text.split('-')[1]) == 2:
                text = 'S' + text
            else:
                text = 'B' + text
        return text
    else:
        res = re.search(r'P?-\d{3}', text)
        if res:
```

```
            return res[0]
```

En caso de que a primeira detección e extracción falle, iniciaríase a seguinte empregando a imaxe modelo alternativa (negro sobre branco). Como último recurso empregamos a detección posicional xa que, se a carta está ben recortada, a rexión do código estará na mesma posición para todas as cartas, excepto para algunhas cartas de nivel seis. Esta detección posicional realízase así:

```
def detect_code_region_positional(file_path):

    im = cv2.imread(str(file_path), cv2.IMREAD_GRAYSCALE)

    unit_size = im.shape[0] * 1 / 9
    roi = im[round(unit_size*6.5):
        round(unit_size*7.2), im.shape[1] * 3 // 4:]

    return roi
```

Antes de seguir avanzando no proxecto realizáronse probas sobre 500 fotos de cartas recortadas manualmente que nos levaron a obter os seguintes resultados:

- Texto detectado e código correcto: 93,6%
- Texto detectado e incorrecto: 2,4%
- Non detectou nada: 4%

Consideramos que estes resultados foron o suficientemente bos como para pasar ás seguintes fases do desenvolvemento.

## 6.3 Servidor

Empezamos o desenvolvemento preparando o servidor xa que as librerías que iamos empregar xa as tiñamos preparadas do estudo previo. Comezamos creando un entorno virtual con *Python 3.8* . Dentro deste entorno teremos as librerías do proxecto illadas. As librerías instaladas neste entorno, mediante Pip, serían as mencionadas na sección 3.2. Na seguinte lista temos unha pequena explicación do contido de cada ficheiro e directorio do servidor creado como un proxecto de *Django* (Figura 6.3):

- *management*: aquí podemos engadir comandos propios para executar desde o entorno de *Django* mediante o *manage.py*.

- *migrations*: contén ficheiros de *python* coas modificacións que se foron realizando na base de datos según modificábamos os modelos. Permítenos reconstruír as táboas da base de datos.
- *resources*: almacenamos aquí as imaxes plantilla que empregamos para o procesado de imaxes.
- *admin.py*: aquí rexistramos os modelos que queiramos poder manexar desde a interface web de administración que nos facilita *Django*.
- *apps.py*: neste ficheiro asegurámonos de que o entorno de *Django* teña constancia do ficheiro de sinais.
- *consumers.py*: aquí xestionamos o *socket*.
- *image_proccesing*: contén as funcións de procesado de imaxe.
- *models.py*: aquí temos os modelos da aplicación.
- *serializers.py*: este ficheiro contén as clases que se encargan de convertir os *bytes* da petición en obxectos python (normalmente estas clases se empregan para gardar os cambios que chegan na base de datos). É neste punto onde lle damos pé ó procesado de imaxe modificando o comportamento que tería o *serializer* ó gardar.
- *signals.py*: encárgase de xestionar o que facer cando se envía un sinal. No noso caso xestiona o sinal *post_save* para que ó recibilo envíe un mensaxe de renovación ós *sockets* da partida.
- *tests.py*: contén as probas da aplicación.
- *urls.py*: aquí podemos ver as direccións dos recursos e a súa vista asociada. O que está no directorio *dtcg_support_server* é o primeiro nivel de resolución, dentro delegamos parte das resolucións ó que está en *dtcg*.
- *utils.py*: ficheiro con funcionalidades útiles, entre elas, a mellora de identificación dos códigos mediante expresións regulares.
- *views.py*: contén as vistas da aplicación.
- *asgi.py*: encárgase de levantar a aplicación como asíncrona e contén as rutas que se xestionarán desta maneira.
- *settings.py*: ficheiro de configuración.
- *wsgi.py*: xestiona a aplicación síncrona.

- *media*: almacena as imaxes recibidas.
- *db.sqlite3*: ficheiro ca base de datos.
- *manage.py*: arquivo do entorno *Django* que podemos executar para iniciar o servidor ou lanzar comandos de xestión.
- *requirements.txt*: contén os requisitos de librerías da aplicación.

A continuación comentaremos o desenvolvemento presentando cada unha das súas pezas completas.

### 6.3.1 Usuarios

Como decidimos empregar os usuarios base de *Django* non precisamos crearlle un modelo. Xa que non queremos permitir a creación de usuarios por parte de calquera non se habilita ningún punto para rexistrarse. Os usuarios rexistraríanse por parte do administrador do sistema empregando a interface de administración de *Django*. Para isto é preciso engadir no ficheiro *admins.py* as seguintes liñas:

```
from django.contrib.auth.models import User
admin.register(User)
```

Xa que a autenticación non é o punto principal do proxecto empregamos autenticación básica o que nos permite xerar unha vista de inicio de sesión protexida por dous intermediarios que nos proporciona DRF quedando a clase así:

```
class LoginCheckView(APIView):

    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]

    def post(self, request):
        return Response(status=status.HTTP_200_OK)
```

Cada petición deberá enviar a cabeceira *Authorization* co usuario e contrasinal en *base64*. A clase de autenticación comprobará se a cabeceira enviada é correcta e a de permisos se lle fora posible acceder a esa vista. En caso de que algún dos datos non fora correcto devolverá un código de erro *401 Unauthorized*. Este tipo de autenticación non é adecuada para produción polo que chegado ese momento sería adecuado cambiala por unha máis fiable, por exemplo, a autenticación por token.

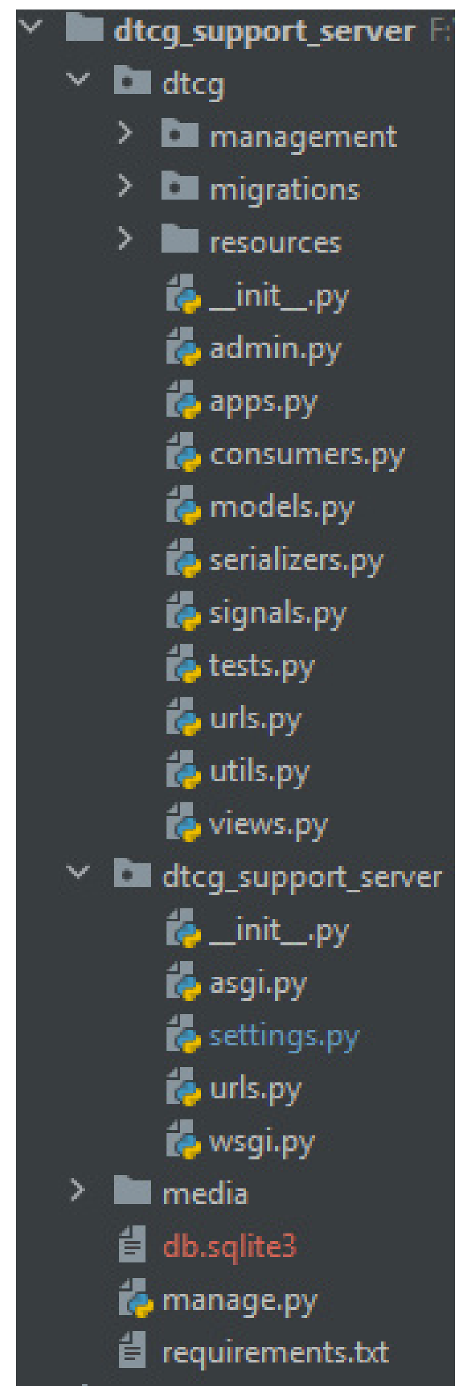


Figura 6.3: Ficheiros no proxecto do servidor

### 6.3.2 Clases modelo

A parte inicial do desenvolvemento consistiu na creación das clases modelo a empregar no servidor. A partir destas clases xeraremos as táboas apropiadas na base de datos. *Django* permítenos crear un ficheiro que almacena os cambios realizados nestas clases para, máis tarde, realizar as modificacións pertinentes na base de datos. As clases modelo que empregamos no proxecto son *Card*, que conterá a información dunha carta do xogo, *Game*, que representa unha partida e *PlayedCards*, que gardará a información das cartas xogadas nunha partida:

```
class Card(models.Model):
    code = models.CharField(max_length=8)
    name = models.CharField(max_length=30)
    play_cost = models.IntegerField(null=True)
    evolve_cost = models.IntegerField(null=True)
    level = models.IntegerField(null=True)
    dp = models.IntegerField(max_length=5)
    effects = models.TextField(null=True)
    inherited_effects = models.TextField(null=True)
    image_url = models.URLField(null=True)


class Game(models.Model):
    id = models.UUIDField(primary_key=True,
                default=uuid.uuid4, editable=False)
    players = models.ManyToManyField(User)
    played_cards = models.ManyToManyField(Card,
                            through='PlayedCards',
                          through_fields=('match', 'card'))

    class GameStatus(models.IntegerChoices):
        CLOSED = 0, 'Closed'
        OPEN = 1, 'Open'
   status = models.IntegerField(choices=GameStatus.choices,
                            default=GameStatus.OPEN)


class PlayedCards(models.Model):
   match = models.ForeignKey(Game, on_delete=models.CASCADE)
```

```
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    player = models.ForeignKey(User, on_delete=models.CASCADE)
    bottom_card = models.ForeignKey('PlayedCards', null=True,
                                    on_delete=models.CASCADE)

    class PlayedCardStatus(models.TextChoices):
        PLAYED = 'PL', 'Played'
        DELETED = 'DE', 'Deleted'
        EVOLVED = 'EV', 'Evolved'
        INHERITED = 'IN', 'Inherited'
    status = models.CharField(max_length=2,
                              choices=PlayedCardStatus.choices,
                              default=PlayedCardStatus.PLAYED)
```

Estas clases herdan da clase *Model* de *Django* que nos aporta a funcionalidade para construír as consultas a base de datos. Cada un dos campos determina o tipo aceptado e as restricións que lle aplican. As definicións dos estados (*PlayedCardStatus e GameStatus*) permítenos limitar os valores que se poden gardar nese campo. As explicacións dos estados das cartas serían:

- *INHERITED*: ten algunha carta posicionada enriba.
- *EVOLVED*: non ten cartas enriba pero si debaixo.
- *PLAYED*: non ten cartas nin enriba nin debaixo.
- *DELETED*: eliminada do campo.

### 6.3.3 Partidas

Para engadir o soporte para as partidas creamos os *serializers* e as vistas correspondentes. Os *serializers* son os encargados de converter a información do corpo da petición nos obxectos que necesitemos. Ademais permítennos aplicar lóxica adicional á hora de crealos. *Django* danos unha clase da que herdar que nos automatiza as tarefas CRUD dos obxectos do modelo. Para as partidas empregamos o seguinte:

```
class GameSerializer(serializers.ModelSerializer):

    class Meta:
        model = Game
        fields = ['id', 'players', 'played_cards', 'status']
```

Nesta clase é necesario indicarlles ó modelo a partir do que se xerará. No parámetro *fields* podemos limitar os campos que devolveríamos na petición.

A xestión das peticións de rede realízase a nivel de vista. No seguinte código podemos ver como definir o comportamento de listado (método *GET*) e creación dunha partida (método *POST*):

```
class GameList(APIView):
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request):
        games = Game.objects.all()
        serializer = GameSerializer(games, many=True)
        return Response(serializer.data,
                        status=status.HTTP_200_OK)

    def post(self, request):
        serializer = GameSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
```

A herdanza da clase *APIView* permítenos definir as peticións ás que responderá. Ademais podemos definir a este nivel o sistema de autenticación e autorización que se empregará na vista. No interior de cada método escollemos como sería o corpo da petición e as respostas posibles, mediante os *serializer* e os códigos de estado HTTP.

Como último paso asociaríamos a vista á URL que nos conveña.

### 6.3.4 Tratamento imaxes

Para realizar o tratamento de imaxes no servidor engadimos un *serializer* apropiado. Ó non realizar operacións estándar sobre o modelo crearemos unha clase que herdará do *Serializer* xeral de *Django*. De esta maneira escribiremos a funcionalidade encargada de gardar a imaxe e detectar o código :

```
class CardImageSerializer(serializers.Serializer):
    image = serializers.ImageField()
```

```
    id_game = serializers.UUIDField(required=False)
    card_code = serializers.CharField(max_length=8,
                                      required=False)

    def create(self, validated_data):
        img = validated_data.get('image')
        p = Path(MEDIA_ROOT + '/standalone')
        if validated_data.get('id_game'):
            p = Path(MEDIA_ROOT + '/' +
                str(validated_data.get('id_game')))
        p.mkdir(parents=True, exist_ok=True)
        p = p.joinpath(img.name)
        with p.open(mode='wb') as f:
            for chunk in img.chunks():
                f.write(chunk)
        card_code = detect_and_extract(p)
        validated_data['card_code'] = card_code
        return validated_data
```

Neste *serializer* definimos os campos que aceptaremos en vez de obtelos do modelo. O campo da imaxe (*ImageField*) require a instalación da librería *Pillow*. Empregamos a librería *pathlib* para crear as direccións de gardado das imaxes independente do sistema operativo.

Como podemos ver no código, se a imaxe pertence a unha partida gardariámola na carpeta correspondente a esa partida. En caso contrario iría a unha carpeta xeral [1]. A continuación pasariámoslle a dirección de almacenamento á función encargada de obter o código (*detect_and_extract*). Esta función encargaríase de ler a imaxe modelo e chamar ás funcións de detección e extracción do texto.

Creamos unha vista que responda ó método *POST*, empregue este *serializer* e controle os estados das respostas. Devolverá un código de estado HTTP 201 (*CREATED*) ou un 400 (*BAD_REQUEST*) se a petición non fora válida. Por último asignámoslle a dirección */identify*.

### 6.3.5 Cartas

Para poder realizar unha partida é necesario ter a información das cartas. Istos datos podémolos atopar na páxina de DigimonCard [12]. Co obxectivo de obter esta información desenvolvemos un pequeno *script* que recibe o nome da colección e o número de cartas da

[1] Durante o desenvolvemento asegurámonos de que as carpetas están creadas ou as creamos. De cara a produción estas accións realizaríanse no despregamento

mesma. Este *script* realizaría peticións empregando o parámetro *search* da páxina de *DigimonCard*, no que enviaríamos o código da carta a consultar como se mostra a continuación:

```
https://digimoncard.io/card/?search=BT5-086
```

O resultado é a paxina coa información desa carta. Recompilamos a información necesaria empregando a librería *BeautifulSoup*. A información obtida é a necesaria para construír a clase modelo (nome, nivel, custo de xogo, efectos, efectos herdados e custo de evolución).

Xa que esta tarefa realízase por cada colección de maneira independente decidiuse engadir como comando de *Django*, é dicir, para podelo lanzar desde o mesmo entorno e con acceso ó noso modelo e base de datos. Para engadilo creámolo como unha clase que herda de *django.core.management.base.BaseCommand* e gardámola na carpeta *management/commands*.

A outra funcionalidade relacionada coas cartas sería xogalas nunha partida. Para realizar esta funcionalidade empregamos o seguinte *serializer*:

```
class ConfirmCardSerializer(serializers.Serializer):
    code = serializers.CharField()
    name = serializers.CharField(required=False)
    bottom_card_id = serializers.IntegerField(required=False)
    # success = serializers.BooleanField()

    def create(self, validated_data):
        card = Card.objects.get(code=validated_data.
                                        get('code'))
        status = PlayedCards.PlayedCardStatus.PLAYED
        if validated_data.get('bottom_card_id', None)
                                        is not None:
            status = PlayedCards.PlayedCardStatus.EVOLVED
            bottom_card = PlayedCards.objects.get(
                        id=validated_data.get(
                                'bottom_card_id')
                        )
            bottom_card.status = PlayedCards.PlayedCardStatus.
                                        INHERITED
            bottom_card.save()
        played_card = PlayedCards.objects.create(**{
                    'card_id': card.id,
                    'player_id': validated_data.
                                        get('id_user'),
```

```
                        'match_id': validated_data.
                                        get('id_game'),
                       'bottom_card_id': validated_data.get
                                    ('bottom_card_id'),
                        'status': status
                    })
        return {
            'code': validated_data.get('code'),
            'name': card.name
        }
```

O código da carta é o único parámetro necesario. Se se envía co *bottom_card_id* implicaría que se trata dunha *digievolución* polo que modificaríamos o estado da carta sobre a que evoluciona. Despois creamos a carta xogada empregando o modelo.

Por último crearíamos a vista que empregue este *serializer* no método *post* e asignariamoslle a URL */games/<uuid:id_game>/addCard*.

### 6.3.6 Websockets

Para empregar os *sockets* no servidor precisamos a librería *Channels*. Estes *sockets* serán necesarios para o envío de mensaxes desde o servidor ó cliente web. Estas mensaxes informarán o cliente web de que debe renovar a información da partida.

Comezamos creando un consumidor (*consumer*) que acepte a conexión e manexe o *socket*:

```
class SocketView(AsyncWebsocketConsumer):

    async def websocket_connect(self, message):
        self.room_name = self.scope["url_route"]["kwargs"]
                                             ["id_game"]
        self.room_group_name = "group_%s" % self.room_name
        await self.channel_layer.group_add(
                                    self.room_group_name,
                                    self.channel_name)
        await self.accept()

    async def event_alarm(self, event):
       await self.send(text_data=json.dumps(event['data']))
```

Neste código aceptaríamos as conexións por *socket*. Cada *socket* que se abra asociárase ó grupo correspondente (un grupo de *sockets* por partida). E na función *event_alarm* xestionamos o

envío da mensaxe.

Para o envío da información creouse un controlador de sinais (no ficheiro *signals.py*) que envía a mensaxe ó grupo de *sockets* que corresponda.

```
@receiver(post_save, sender=PlayedCards)
def send_update_order(sender, instance, **kwargs):
    group_name = "group_%s" % instance.match_id
    channel_layer = channels.layers.get_channel_layer()

    async_to_sync(channel_layer.group_send)(
        group_name,
        {
            'type': 'event_alarm',
            'data': {
                'message': 'renew'
            }
        }
    )
```

Como podemos ver no código o *socket* recibe sinais despois de que se garde unha nova carta nunha partida. A función *async_to_sync* permítenos chamar de maneira síncrona a métodos asíncronos. Escóllese o grupo ó que enviar a mensaxe, definimos a mensaxe e enviámola. O tipo da mensaxe define quen a xestionará no consumidor, é dicir, no noso caso o tipo sería *event_alarm*. No consumidor é manexado pola función do mesmo nome.

Para xestionar a dirección a que responderán os *sockets* engadimos este código no ficheiro *asgi.py*:

```
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter,
                            URLRouter
from channels.security.websocket import
                        AllowedHostsOriginValidator
from django.urls import re_path
from dtcg.consumers import SocketView

application = ProtocolTypeRouter(
     {
        "http": get_asgi_application(),
        "websocket": AllowedHostsOriginValidator(
```

```
            URLRouter([
                re_path('^ws/(?P<id_game>[A-Za-z0-9_-]+)',
                                            SocketView.as_asgi()),
            ])
        ),
    }
)
```

Nesta peza de código indicámoslle que para a resposta das peticións HTTP as manexe a aplicación de maneira estándar e definimos a ruta sobre a que se poden abrir os *sockets*.

## 6.4 Aplicación móbil

O desenvolvemento da aplicación móbil realizámolo empregando *Android Studio*. Un proxecto de *Flutter* xera ficheiros específicos de *android* e *IOS* que definen tratamentos específicos para cada tipo de sistema. Estes ficheiros non foi necesario realizar nada sobre eles polo que os omitiremos. Os ficheiros *dart* nos que realizamos o desenvolvemento serían os da Figura 6.4:

- *models*: define os modelos.
- *screens*: describe as pantallas principais.
- *services*: especifica os servizos que se conectarán ós recursos remotos.
- *states*: define os ficheiros de estado con clases *Cubit*.
- *widgets*: define pequenos compoñentes reutilizables.
- *generated_plugin_registrant.dart*: rexistra determinados *plugins* como a cámara.
- *main.dart*: punto de entrada da aplicación móbil.

A continuación veremos as distintas funcionalidades que se engadiron.

### 6.4.1 Usuarios

O tratamento dos usuarios na aplicación móbil limítase ó inicio de sesión e a autenticación das peticións ó servidor.

O inicio de sesión realízase na pantalla principal e só aparece en caso de non introducir un usuario e contrasinal correctos, que se gardarán no *Cubit* de autenticación. Na Figura 6.5 vemos a pantalla de inicio de sesión. Desde aquí chamaríamos ó servizo enviándolle a información do usuario e contrasinal.

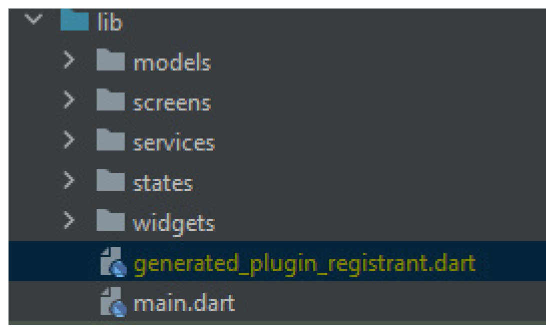


Figura 6.4: Ficheiros de desenvolvemento da aplicación móbil

```
Future<bool> login(String username, String password) async {
    final url = Uri.parse('$urlString/login');
    final response = await http.post(url, headers: {
                            'Authorization': 'Basic
                            ${base64.encode(utf8.encode(
                                '$username:$password'
                            ))}'
    });
    return response.statusCode == 200 ? true : false;
}
```

En caso de que o inicio falle mostraría un mensaxe de erro para informar ó usuario:

```
ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
    content: Text('Wrong username or password'),
    backgroundColor: Colors.redAccent,
));
```

Unha vez que o inicio de sesión foi realizado móstranse as opcións da aplicación como vemos na Figura 6.6.

A opción de identificar cartas levaríanos directamente á pantalla de toma de fotos para identificalas directamente sen necesidade de crear unha partida.

### 6.4.2 Partidas

Desde a pantalla principal podemos crear unha partida ou unirnos a unha xa creada. Unha vez creamos unha partida automaticamente engádesenos a ela como participante. Para unirnos a unha partida sería preciso ter o identificador da partida en cuestión. Para cada unha destas tarefas creamos unha petición ó servizo remoto:

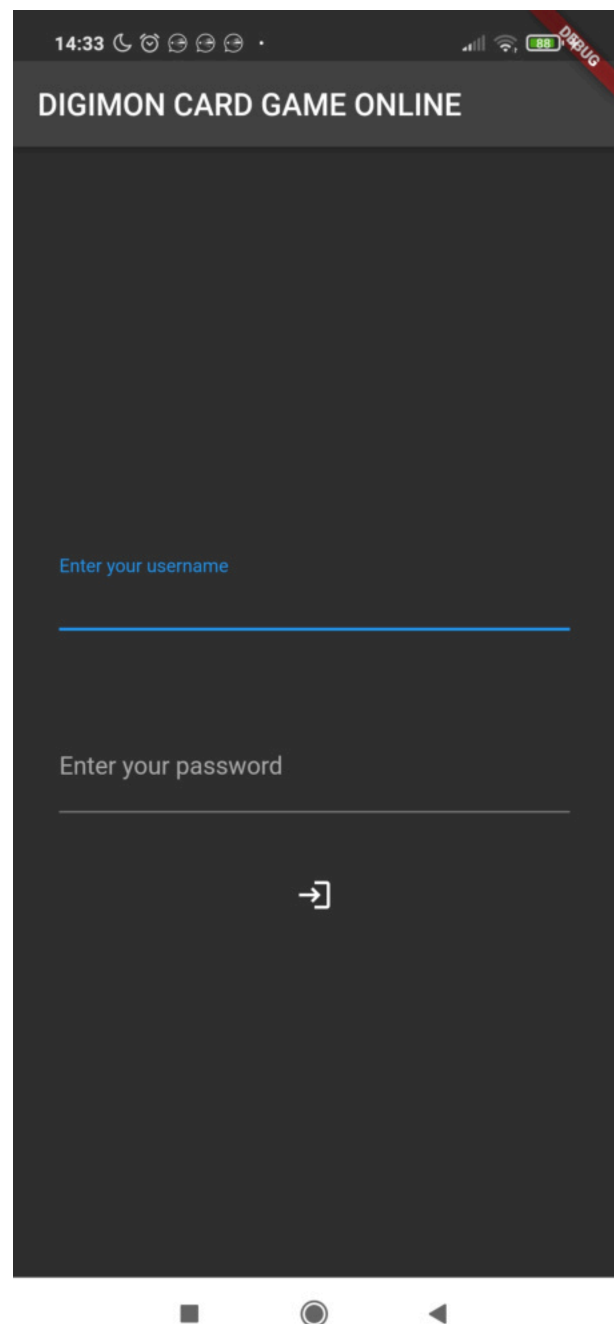


Figura 6.5: Pantalla de inicio de sesión

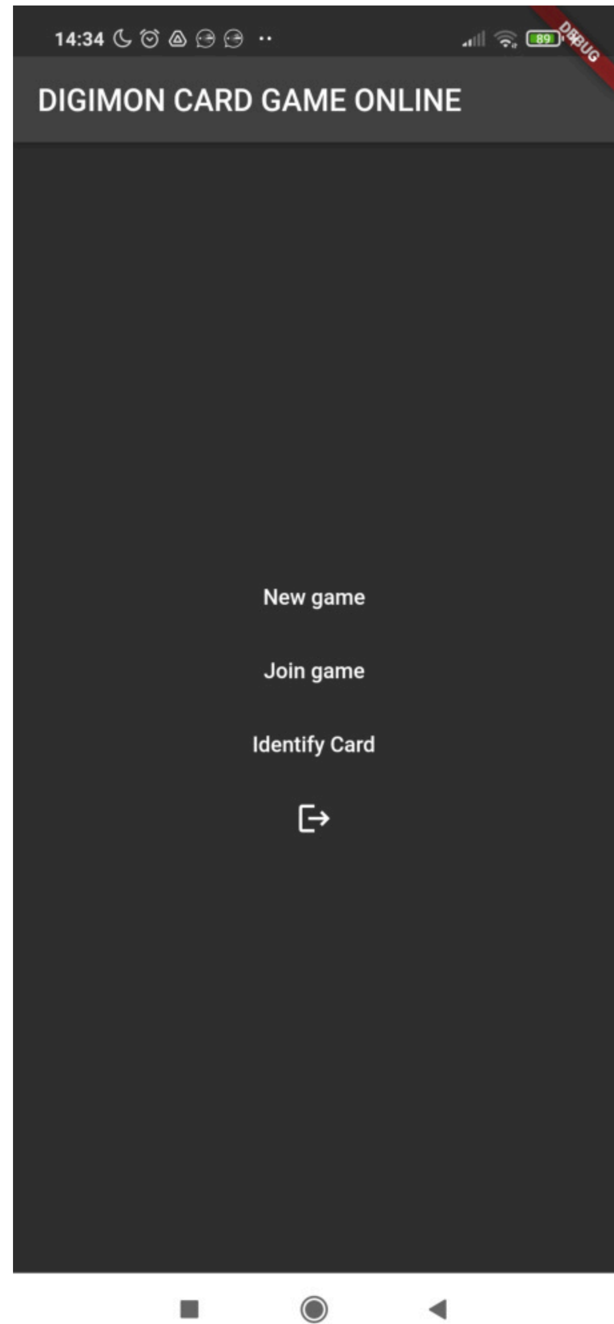


Figura 6.6: Pantalla principal da aplicación móbil

```
Future<String> newGame() async {
    final url = Uri.parse('$urlString/games');
    final response = await http.post(url);
    if (response.statusCode == 201) {
        final data = jsonDecode(response.body);
        final id = data['id'];
        return id;
    } else {
        throw Exception('Failed to create game');
    }
}
```

A petición para unirse á partida quedaría como segue:

```
Future<bool> addUserToGame(String id) async {
   final url = Uri.parse('$urlString/games/$id/addPlayer');
    final response = await http.post(url, headers: {
        'Authorization': authHeader
    });
    return response.statusCode == 200 ? true : false;
}
```

Unha vez entramos nunha partida podemos copiar o identificador para compartilo ou engadir *digimons* a esa partida como vemos na Figura 6.7.

Na esquina superior dereita vemos o botón para copiar o identificador e abaixo á dereita vemos o botón para xogar unha carta. Este botón definiríamolo así:

```
IconButton(
    onPressed: () => Clipboard.setData(ClipboardData(
        text: widget.idGame
    )).then((_) {
        ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Id of the
                                game copied to clipboard'))
        );
    }),
    tooltip: 'Copy id game',
    icon: const Icon(Icons.copy)
)
```

O botón flotante lévanos á pantalla de toma de fotos para que se envíen a identificar.

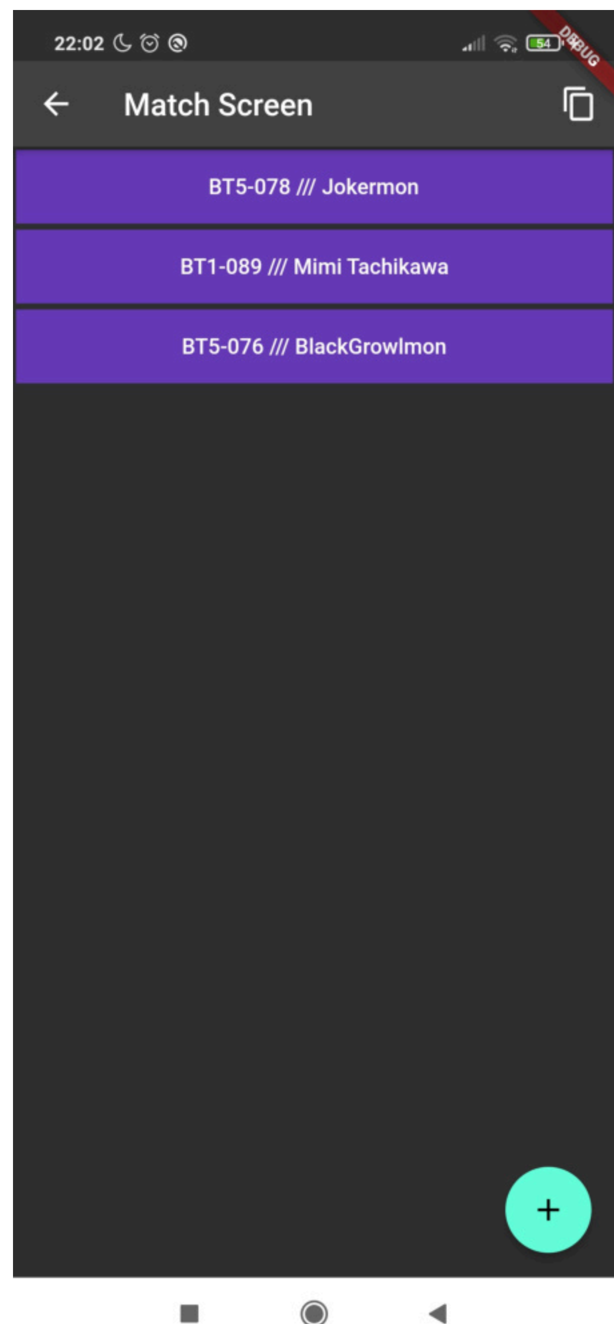


Figura 6.7: Pantalla de partida con un digimon en xogo

### 6.4.3 Fotos

Para xestionar o comportamento da cámara instalamos o complemento *camera* [13] de *Flutter*. Empregar a cámara en Android require ter unha version mínima do *SDK* que sería a 21 (Android 5.0 ou superior). A iniciación da cámara require as seguintes liñas que engadimos no ficheiro *main.dart*:

```
WidgetsFlutterBinding.ensureInitialized();
final cameras = await availableCameras();
final firstCamera = cameras.first;
```

A primeira liña asegúrase de que os complementos estean iniciados e así poder acceder, nas seguintes, ás cámaras dispoñibles para seleccionar a primeira, que sería a exterior. No noso caso realizamos estas chamadas ó arrancar a aplicación para non perder tempo máis adiante.

Para empregar a cámara e necesario iniciar o seu controlador, isto permítenos previsualizar o que vería a cámara. Xa que habería que esperar a súa iniciación é necesario que a vista proporcione información ó usuario. Para isto empregamos o compoñente *FutureBuilder*.

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Take a picture')),
      body: FutureBuilder<void>(
        future: _initializeControllerFuture,
        builder: (context, snapshot) {
          if (snapshot.connectionState ==
                            ConnectionState.done) {
            return CameraPreview(_controller,
              child:
                Center(
                    child: Container(
                        width: MediaQuery.of(context).size.
                                            width / 4 * 3,
                       height: MediaQuery.of(context).size.
                                       width /4 * 3 * 1.428571,
                  decoration: BoxDecoration(
                    border: Border.all(
                      width: 2,
                      color: Colors.blue,
                    ),
```

```
                ),
              ),
            ),
          );
        } else {
          return const Center(
              child: CircularProgressIndicator()
          );
        }
      },
    ),
  );
}
```

O *Scaffold* proporciónanos o esqueleto para a vista. O corpo da vista constrúese unha vez o controlador da cámara está iniciado, mentres isto non ocorre mostra un indicador de carga para informar. Dentro da previsualización da cámara móstrase o recadro de dimensións similares a unha carta para que o usuario a centre ben (Figura 6.8).

Unha vez tomada a foto podémola ver xa recortada (Figura 6.8) e mandala ó servidor para a súa identificación (Figura 6.9).

Para enviar estas imaxes realizamos unha chamada a *API* enviando a seguinte petición:

```
Future<String> sendCardImage(File imageFile) async {
    final url = Uri.parse('$urlString/identify');
    final stream = http.ByteStream(imageFile.openRead());
    stream.cast();

    final length = await imageFile.length();
    final request = http.MultipartRequest("POST", url);
    final multipartFile = http.MultipartFile('image',
       stream, length, filename: basename(imageFile.path));

    request.files.add(multipartFile);
    final response = await request.send();
    if (response.statusCode == 201) {
      final http.Response res = await
               http.Response.fromStream(response);
      final body = jsonDecode(res.body);
      if (body['card_code'] != null) {
```

Figura 6.8: Toma de fotos. Esq.: Previsualización da cámara. Der.: Imaxe recortada ó tomar a foto.

```
        return body['card_code'];
      }
    }
    return '';
}
```

Unha vez identificada a carta poderíamos modificar o código por se fora erróneo ou mandarllo directamente ó servidor para que o engada á partida. Para isto realizamos a seguinte petición:

```
Future<bool> sendCode(String code, String? idGame,
                    {int? bottomCardId}) async {
    final url = Uri.parse(
```

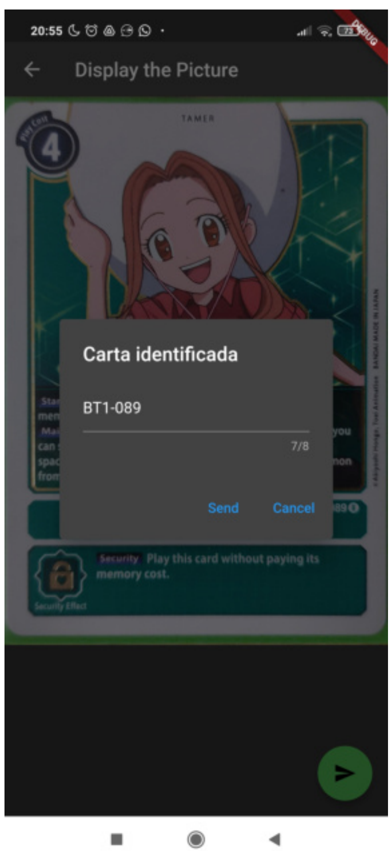


Figura 6.9: Imaxe da carta recortada e identificada

```
              '$urlString/games/$idGame/addCard');
    var body = {'code': code.toUpperCase()};
    if (bottomCardId != null) {
      body['bottom_card_id'] = bottomCardId.toString();
    }
    final response = await http.post(url, headers: {
      'Authorization': authHeader
    }, body: body);
    return response.statusCode == 201 ? true : false;
  }
```

### 6.4.4 Cartas

Dentro da pantalla de partida vemos un listado das cartas presentes no noso campo. Dende aí poderíamos iniciar os comportamentos específicos como serían *digievolucionar, de-digievolucionar* e eliminar a carta. Para isto facemos clic na carta sobre a que queremos actuar

e nos mostrará un *dialog* coas opcións posibles (Figura 6.10).

No recadro do *dialog* vemos que podemos *digievolucionar* mediante texto ou enviando unha foto para identificar. En ambos casos realizariamos as peticións vistas na sección anterior coa única diferenza de que enviaríamos o identificador sobre o que se actúa. Tamén podemos eliminar o *digimon* por completo ou facer *de-digivolve* que implicaría eliminar só a carta superior (*Trash card*).

## 6.5 Cliente web

O desenvolvemento do cliente web realizouse en ficheiros *Vue* que conteñen a definición HTML do compoñente e a lóxica. A estrutura de ficheiros que se empregou podémola ver na Figura 6.11:

- *assets*: inclúe ficheiros *CSS* e recursos como iconos.
- *components*: inclúe os ficheiros *Vue* coa definición dos compoñentes.
- *router*: determina as direccións URL e que compoñente ou vista responderá.
- *store*: almacenaría información dos estados do cliente web. Non se empregou no proxecto.
- *views*: almacena os compoñentes *Vue* coa definición de compoñentes que se empregarán como páxinas completas.
- *App.vue*: ficheiro co compoñente de maior nivel de *Vue*.
- *main.js*: ficheiro de entrada que monta o compoñente de maior nivel e rexistra outros compoñentes e directivas a empregar.

O seguinte código, que podemos ver no *main.js*, encárgase de iniciar a libraría *Vuetify* e engadir o enrutador (*router*) a empregar. Ademais crea o compoñente raíz da aplicación e asígnallo ó elemento do DOM:

```
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";
import "./assets/main.css";
// Vuetify
import "vuetify/styles";
import { createVuetify } from "vuetify";
import * as components from "vuetify/components";
```

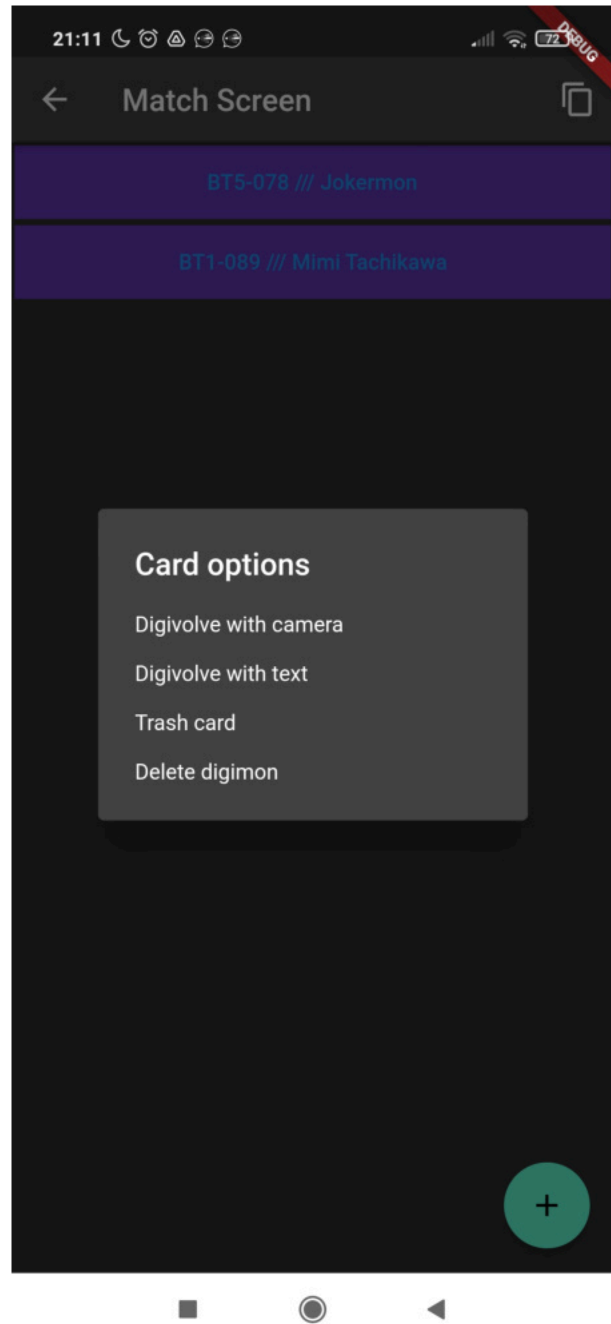


Figura 6.10: Posibles accións sobre unha carta

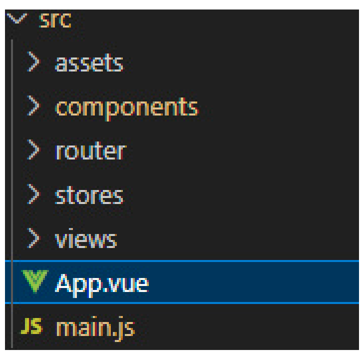


Figura 6.11: Ficheiros do cliente web

```
import * as directives from "vuetify/directives";

const vuetify = createVuetify({
  components,
  directives,
});

const app = createApp(App);
app.use(router);
app.use(vuetify);

app.mount("#app");
```

Para mostrar a información das cartas dunha maneira sinxela creamos unha vista co compoñente *MainScreen*. Este cargará a barra de navegación superior e un compoñente *Field* que conterá a lista das cartas xogadas por cada xogador e delegará esta información en compoñentes *Digimon* para mostrala.

Cando se crea o compoñente *Field* cárgase a información da partida desde o servidor:

```
async fetchData() {
    let id_game = this.$route.params.game_id;
    axios.get(`${API_URL}/games/${id_game}/cards`)
        .then((response) =>
        this.formatResponse(response.data));
}
```

Cando se recibe a resposta adaptamos o formato ó que necesitamos. Cada unha das cartas mostraríase da seguinte maneira:

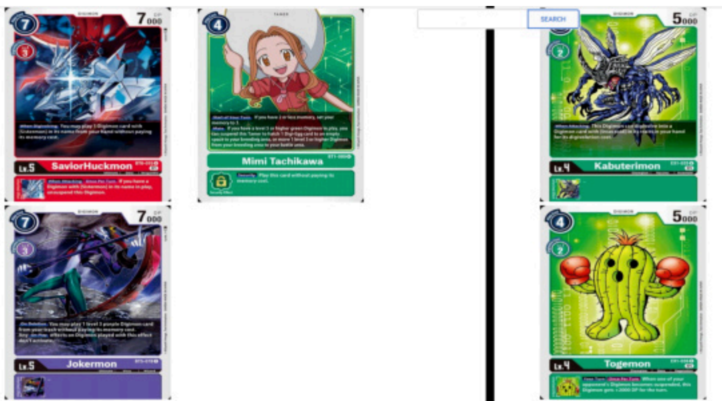


Figura 6.12: Imaxe do campo de xogo dunha partida

```
<img :src="digimon['card'].image_url"
    v-if="digimon.status !== 'IN'"
    @click.left="showDigimonInfo()"
/>
```

Cada compoñente *Digimon* contén a lista das cartas que o compoñen e leva o nome da carta superior. Na Figura 6.12 podemos ver que só mostramos a carta superior de cada un. Cando se fai clic nunha das cartas abriríase o *dialog* que mostraría en detalle a carta e a información das cartas que teña debaixo (Figura 6.13).

### 6.5.1 Websockets

Para poder actualizar as cartas en xogo abrimos unha conexión por *socket*, no compoñente *Field*, ó servidor e cando recibamos unha mensaxe pedimos a información actualizada ó servidor:

```
this.connection = new WebSocket(
    `ws://192.168.1.135:8000/ws/${id_game}`);
this.connection.addEventListener('message',
                                   (event) => {
    this.fetchData();
});
```

Figura 6.13: Información dunha carta en detalle no cliente web

Capítulo 7

# Conclusións

Neste derradeiro capítulo da memoria presentaremos as conclusións obtidas durante o desenvolvemento do proxecto. Ademais falaremos das posibles tarefas futuras.

Neste proxecto intentouse mellorar a experiencia das partidas *online* do DTCG. Estas partidas presentan maiores dificultades que as presenciais debido á imposibilidade de distinguir as cartas a través dunha cámara web.

Como solución a este problema desenvolveuse un sistema en tres partes:

- Un servidor implementado en *Python* que emprega os frameworks de *Django* e DRF para a xestión das peticións e respostas e a comunicación coa base de datos. O servidor centraliza a creación de partidas e o desenvolvemento do xogo. Recibe as cartas xogadas por cada xogador e realiza as notificacións correspondentes ós espectadores da partida. Parte deste sistema emprega *OpenCV* para procesar as imaxes das cartas recibidas e detectar a rexión de interese que conteña o código da carta. Esta rexión é procesada polo OCR *Tesseract* para obter o código da carta e validado por unhas expresións regulares. Esta identificación ten un porcentaxe de acerto do 93,6%, este resultado considérase aceptable xa que o usuario pode corrixir o código manualmente en caso de erro.

- Unha aplicación móbil desenvolta en *Flutter* que nos permite crear partidas, unirnos a partidas xa creadas e enviar fotos das cartas xogadas ó servidor.

- Un cliente web desenvolto en *VueJs* no que poder conectarnos a unha partida, visualizar o estado xeral do campo de xogo e consultar en detalle as cartas en xogo. Este sistema, unha vez conectado a unha partida, non precisa máis interacción xa que conta con actualizacións automáticas da información grazas o emprego de *websockets*.

Chegados a este punto o obxectivo de crear un sistema que de soporte ás partidas *online* do DTCG considerámolo cumprido. Aínda así durante o desenvolvemento e as probas démonos conta de que o sistema ralentiza o ritmo das partidas. Este problema non é culpa do procesado

de imaxes, como cabería esperar, senón da toma de fotos. O tempo que o usuario tarda en encadrar e obter unha foto da carta é, normalmente, superior ó tempo que tarda o servidor en identificala. Con práctica este tempo redúcese de maneira considerable. Non obstante, sempre se podería engadir a opción de enviar o código identificativo da carta directamente para optimizar este tempo.

Como conclusión, este sistema resultará moi útil para xente que estea xogando as súas primeiras partidas de maneira *online* e aínda non esté o tanto das cartas máis empregadas. Ademais aporta a posibilidade de mellorar as retransmisións das partidas mediante outras plataformas como *Twitch*.

## 7.1 Traballos futuros

De cara ó futuro preséntanse as seguintes tarefas que se poderían realizar para mellorar o sistema:

- Automatizar a obtención da foto da carta na aplicación móbil. Se conseguimos que se detecten automaticamente os bordes dunha carta coa suficiente exactitude aforraríamos un tempo valiosísimo durante a partida. O feito de non poder asegurar un fondo uniforme por debaixo da carta e unha das dificultades desta tarefa.

- Melloras visuais no cliente web, engadir animacións que resalten a última carta xogada axudaría a detectala nunha primeira ollada.

- Optimizar a actualización das partidas no cliente web aumentando a información que se envía a través do *websocket*. Isto implicaría xerar mensaxes máis específicas desde o servidor nas que se enviaría toda a información da última xogada.

# Relación de Acrónimos

**CRUD** Create Read Update Delete. 17, 53

**DOM** Document Object Modal. 69

**DRF** Django Rest Framework. i, 17, 50, 74

**DTCG** Digimon Trading Card Game. 1, 4, 74

**HTML** HyperText Markup Language. 17–19, 69

**HTTP** Hypertext Transfer Protocol. 18, 32, 54, 55, 59

**IDE** Integrated Development Environment. 16

**JSON** JavaScript Object Notation. 17

**LIFO** Last In First Out. 13

**MVC** Modelo Vista Controlador. 17, 28

**NPM** Node Package Manager. 19

**OCR** Optical Character Recognition. 18, 36, 45, 47, 74

**ORM** Object Relational Mapping. 17, 28

**REST** REpresentational State Transfer. 17, 28

**URL** Uniform Resources Locator. 17, 23, 24, 54, 57, 69

**XML** eXtensible Markup Language. 18

# Glosario

**Android** Sistema operativo para dispositivos móbiles que emprega o núcleo de Linux. 18, 19

**Dart** Linguaxe de programación multiplataforma desenvolto por Google. 18, 39

**framework** Conxunto de ferramentas que serven de estrutura base para a elaboración de proxectos máis concretos. 16–19

**Git** Software empregado para levar un control das versións. 16

**iOS** Sistema operativo para dispositivos móbiles creado por Apple para o seu hardware propio. 18, 19

**Javascript** Linguaxe de programación interpretado utilizado nun principio para facer páxinas web interactivas. 19

**Pip** Sistema de xestión de paquetes para Python. 48

**PostgreSQL** Sistema de xestión de bases de datos relacional coñecido pola súa robustez e rendemento. 28

**serialización** Proceso de converter un obxecto a *bytes* para a súa transmisión. 17

**sprint** Na metodoloxía SCRUM Período de tempo no que se levan a cabo unhas tarefas predefinidas. 20–24

**SQLite** Sistema de xestión de bases de datos relacional. Contido nunha pequena libraría escrita en C. 28

**template matching** Técnica empregada en procesamento de imaxes para detectar partes pequenas dunha imaxe partindo dunha imaxe plantilla. 45

**Trello** Ferramenta de xestión de proxectos que nos proporciona taboleiros virtuais nos que plantexar ideas mediante tarxetas. 16, 21

**websocket** Tecnoloxía que posibilita a creación dunha canle de comunicación bidirecional sobre un *socket*. 18, 31, 32

# Bibliografía


[1] [En liña]. Dispoñible en: https://www.python.org/downloads/

[2] [En liña]. Dispoñible en: https://www.djangoproject.com/

[3] [En liña]. Dispoñible en: https://www.django-rest-framework.org/

[4] [En liña]. Dispoñible en: https://tesseract-ocr.github.io/tessdoc/

[5] [En liña]. Dispoñible en: https://opencv.org/about/

[6] [En liña]. Dispoñible en: https://channels.readthedocs.io/en/stable/

[7] [En liña]. Dispoñible en: https://flutter.dev/

[8] [En liña]. Dispoñible en: https://pub.dev/

[9] [En liña]. Dispoñible en: https://vuejs.org/

[10] [En liña]. Dispoñible en: https://next.vuetifyjs.com

[11] [En liña]. Dispoñible en: https://docs.opencv.org/4.x/df/dfb/group__imgproc__object.html

[12] [En liña]. Dispoñible en: https://digimoncard.io/

[13] [En liña]. Dispoñible en: https://pub.dev/packages/camera