ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Реализация на играта Dungeon Mayhem като уеб приложение

Дипломант:
Георги Илиянов Трайков

Дипломен ръководител:
инж. Кирил Митов

СОФИЯ

2 0 2 3

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 22.11.2022 г. Утвърждавам:.....
Дата на предаване: 22.02.2023 г. /проф. д-р инж. П. Якимов/

ЗАДАНИЕ
за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Георги Илиянов Трайков от 12 Б клас

Тема: Реализация на играта Dungeon Mayhem като уеб приложение.

Изисквания:
-реализация на сървърната част с логиката на играта чрез Ruby и Ruby on Rails.
-реализация на комуникацията с браузъра чрез Hotwire, Turbo и Stimulus.
-реализация на комуникация между сървъра и клиента чрез websocket, стъпвайки върху Turbo Streams.
-реализация на функционален мултиплеър на играта.

Съдържание 3.1 Теоретична част
3.2 Практическа част
3.3 Приложение

Дипломант :.....
/ Георги Трайков /
Ръководител:.....
/ инж. Кирил Митов/

Становище

Разработената дипломна работа дава възможност на потребители да играят популярна игра с карти. Дава се възможност да четири потребителя да играя едновременно, като сървърът поддържа връзка с четиримата. Целта на работата бе и дипломнатът да се запознае с възможностите на Rails и Hotwire като реализира Progressive Web App, използващ съвременни технологии като WebSocket.

Дипломантът реализира работата, усвои и използва подходящо технологиите. Играта може да се допълва, но е подходяща рамка за бъдещо развитие.

Ръководител:.....
/ инж. Кирил Митов/

Глава 0

Увод

В областта на разработката на софтуер непрекъснато се появяват нови технологии и фреймуъркове, всеки от които има своите силни и слаби черти, които ги правят по-подходящи за дадени проекти и цели. Един фреймуърк, който набира популярност в последните години е Ruby on Rails, още познат като Rails.Езикът Ruby, на който е написан Rails, представлява динамичен, обектно-ориентиран програмен език, известен заради простия си синтаксис и гъвкавост. Еддна от причините Rails да набира популярност е леснотата и бързината, с които се строят уеб апликации.

Друга черта на фреймуърка е наличието на gem-ове – части код, библиотеки, които могат да бъдат използвани многократо, биват лесно интегрирани в проектите и улесняват разработката като добавят вече готова функционалност.

Една от причините да вляза в ТУЕС беше любовта ми към видеоигрите, още от осми клас си мечтая да създам проект, който освен като пример за пръв проект, може и да бъде използван. След известно време в това училище знаех, че едва ли ще се занимавам с разработка на игри, а по-скоро със софтуер, но все още имах тази мечта – да създам игра.

Първичната ми идея беше създаването на 3D ролева игра с взети идеи от загалвия като Skyrim и Dark Souls, но поради липса на яснота за целите на проекта се отказах.

Дипломният ми ръководител предложи готова тема, която беше хибрид между това, което исках, а именно игра и уеб приложение. Вече можеше да

реализирам играта, която играем с приятели, щом се съберем, за да можем да я играем и онлайн.

Целта на проектът е да се реализира играта с карти Dungeon Mayhem или в случай, че не получа позволение от създателите да използвам тяхната игра – spinoff с подобен или идентичен геймплей. За целта ще използвам най-новата версия на Ruby on Rails – версия 7, поддържаща Hotwire gem-а, който ще позволи презареждането на картите в хода на играта.

Очаква се проектът да има екран, в който да създадеш своя профил и да го потвърдиш чрез e-mail-а си, чак тогава ще има опция за нова игра. След това на началната страница има бутон с възможност за създаване на нова игра, който след като бъде натиснат изчаква необходимия брой играчи и стартира играта. След протичането на играта в нов екран ще има опция за преглед на предишни игри и резултатите им, както и екран, на който се виждат регистританите потребители и техният ранк. Потенциално ще бъде добавено и меню за филтри на лобито, включващо брой на играчите, минимален ранк, време на ход. Други възможни функционалности са екран за прегравне на играта, на който ще се вижда какво се е случило на всяка взядка, както и възможността за коментиране на тези взядки. Потенциално ще бъдат разработени няколко примерни игри, които ще послужат за туториал и обучение на нови играчи.

Смятам, че този проект е страхотна възможност за научаване на Rails фреймуърка и някои от принципите на уеб разработката като цяло, както и разработката на игри/*може да спомена за желанието си да разработя игра и как това ми е бил един от мотивите в туес*/. Финалният резултат ще бъде напълно функционална игра с карти, която може да се играе от набор от хора едновременно, постижение за първи самостоятелен проект в сферата на уеб development-а.

Глава 1

Проучване

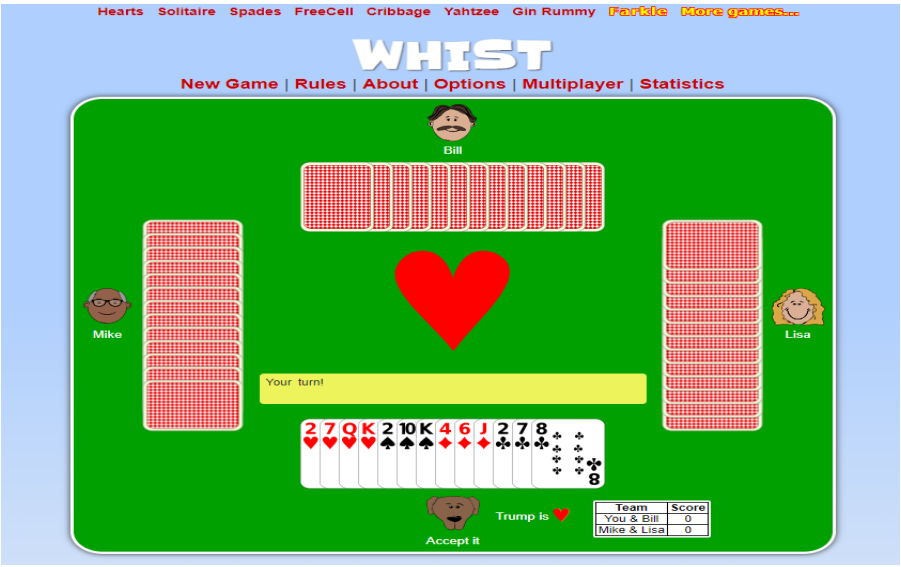
1.1 Проучване на съществуващи подобни програми

След дълго търсене на настолни картови игри за четирима или повече играчи с подобни механики като тази не успях да намеря много. Имаше много настолни игри, които се доближаваха до концепцията, но за жалост нямаха уеб приложение или някакъв друг начин да се играят освен в истинския свят. Но докато събирах вдъхновение за логиката и начинът, по който ще изглежда крайния продукт се натъкнах на няколко заглавия.

Cardgames.io е уебсайт, който предлага разнообразие от онлайн игри с карти, които могат да се играят безплатно. Той предоставя на потребителите си богат избор от класически игри с карти, като Solitaire, Hearts, Spades, и Bridge, както и по-малко известни игри, като Oh Hell!, Go Fish и Crazy Eights. Една от главните черти на уебсайтът е лекотата на използване и достъпността му, потребителите могат бързо да го достъпят и да започнат да играят без преминаването през много екрани(Фигура 1.1).

Няма много налична информация за технологиите, които сайтът използва към сегашен моменг, но е сигурно, че измежду тях са *HTML5, JavaScript* и *CSS*. След малко повече проучване открих, че допреди 3 години са използвали Ruby on Rails като сървър и фреймуърк. Сайтът е разработен като се адаптира спрямо устройството на потребителят, а освен това има и функции като multiplayer, чат стаи и leaderboard-ове, което добавя към цялостното преживяване на потребителят.

Като цяло, *cardgames.io* е добре проектиран уебсайт, удобен за игране на онлайн игри с карти. Простият, но достатъчен user interface прави навигацията и влизането в избрана игра лесно. Функционалността за multiplayer и leaderboard-ът добавят възможността за съревнование с други играчи, което добавя елемент на състезателност. Играта може да служи като пример за това как да бъде оформен потребителският интерфейс.



Фигура 1.1: Потребителски интерфейс на играта Whist в cardgames.io

Playingcards.io е онлайн платформа, подобна на *cardgames.io*, предоставяща на играчите си виртуално пространство, в което могат да се насладят на изобилие от игри с карти, дали с приятели, или с непознати.

За разлика от *cardgames.io,* уебсайтът не се отличава особено с потребителския си интерфейс, тъй като не е много интуитивен, но компенсира с факта, че дава възможност на потребителите си да създават стаи и да канят приятели, с които да играят в тях, чрез линк или номер на стая. Това дава възможност на играчите да се наслаждават на любимите си игри с карти, дори и да не са един до друг.

Освен това, сайтът има и чат, в който участниците на играта могат да комуникират. Тази фукционалност е сравнително полезна в случай, че трябва се дискутират стратегии по време на игра или просто с цел социализиране.

Проектът не е с отворен код, но е построен на основата на модерни технологии, като React, Node.js и Socket.io. Подобно на Hotwire, Socket.io дава възможност за имплементация на комуникация в реално време в проекти, стъпвайки на WebSocket-и за осъществяване на тази комуникация между сървър и краен потребител.

Този уебсайт може да бъде описан като такъв, в който са имплементирани точните неща, от които се нуждае за да бъде функционален и да даде възможност на потребителите си да се забавляват.

Cardzmania е третият и последен сайт, стъпващ на сходна идея, който намерих за интересен по време на прегледа на подобните съществуващи продукти. Както останалите, така и той предоставя много опции за игри с карти, които могат да бъдат играни по сходен начин като разгледаният в *playingcards.io.*

Имплементацията на играта „Война“ ми направи впечатление, тъй като лобито, в което чаках имаше изискване за минимум двама играчи, но до четирима можеха да играят играта. Спрямо броя на участниците потребителският интерфейс се променяше, картите се теглят чрез натискане на бутон, след което се взима карта от тестето и се изиграва на полето. Имплементирани са както чат, така и автоматични ходове в случай, че някой от играчите напусне играта.

Трите гореспоменати продукта показват как може да се изгради уеб приложение за игра с карти, какви функционалности могат да бъдат добавени към крайния продукт и няколко начина, по които може да се свържеш с приятели и да играеш.

1.2 Преглед на развойните средства и среди

Тъй като бях решил да направя игра с карти, а на срещата с дипломните ръководители моят дипломен предложи реализацията на такава игра като уеб приложение с Ruby и Ruby on Rails 7 се спрях на тези две технологии.

Ruby е обектно-ориентиран програмен език, базиран на концепциите за класове и обекти. Това позволява на разработчиците да употребяват създаден вече код, което прави подредбата на кода по-проста. Активната общност на езика позволява на разработчиците да намират решения ня проблемите си сравнително лесно, както и дава достъп до gem библиотеките, които лесно се интегрират в проектите и придават различни функционалности на проектите. По този начин без много писане можем да добавим функции като автентикация на потребители, валидация на дата и др. Rails е фреймуърк, базиран на Ruby. Той имплементира Model-View-Controller(MVC) девелъпмент рамката и се отличава с два главни принципа, първият от които е “Convention over Configuration”(CoC).Той улеснява работата на разработчикът като употребява дефолтни конфигурации за често срещани операции, което слага по-голям фокус на имплементирането на нужните за проекта функционалности като спестява време в начина, по който се пишат. Това обаче има и своите минуси – при създаване на проект, често в темплейта се съдържат много ненужни файлове и папки, което може да направи програмата ни по-непрактична. Другият отличителен принцип на Ruby on Rails е “Don’t Repeat Yourself”, той има за цел да намали дублирането на код и да увеличи поддръжката му. Прилага се чрез създаване на многократно използваем модулен код, абстрахиране на общи модели и използване на техники като наследяване. Тези принципи правят създаването и поддръжката на уеп апликациите лесно, което прави Rails популярен избор за фреймуърк.

Едно от изискванията на дипломният ми ръководител беше да разуча Hotwire gem-а преди да започна работа. Библиотеката позволява да се създават

реагиращи уеб приложения, действащи в реално време.Gem-ът е пуснат през 2020г. и набира популярност в седмата версия на Rails. Използва Turbo фреймуърка, което позволява на страницата да се ъпдейтва бързо без нуждата от рефреш на цялата страница. С Hotwire, разработчиците могат да създават интерактивни потребителски интерфейси, използвайки възможностите на *JavaScript* и уеб технологиите. Има и възможност за връзката с *Action Cable*, което позволява интеграцията на комукация между сървъра и client-а в реално време. Всичко това ще играе голяма роля в проекта ми, тъй като при всяка взядка трябва да се рефрешва състоянието на картите.

Други gem-ове, които намират приложение в заданието ми са *Devise*, *Simple Form* и *Foreman*. *Devise* позволява имплементацията на потребителска автентикация и авторизация както и изпращане на имейли за потвърждаване на акаунт и обновяване на парола, а *Simple Form* улеснява създаването на форми, които потребителите на играта ще попълват.

Първият ми избор на база данни беше *PostgreSQL*, но след това стигнах до заключението че конфигурацията на приложението за работа с нещо по-просто като *SQLite3* ще спести време и излини затруднение. Макар и да не е толко скалируема, тази база данни върши идеална работа за разработка поради малкия брой трафик към сървъра.

За да следя базата данни и да правя промени по тях по време на тестване използвах *DBeaver*, тъй като в практиката миналото лято ми се наложи да го изтегля и се досетих, че работата с него е сравнително лесна. Този софтуер дава възможност да се преглеждат таблици в различни бази данни, като освен това можем да правим и *SQL* query-та чрез editor-а му.

За text editor разчитам на *VS Code*, както заради простия му потребителски интерфейс, така и заради връзката му с *git*, която ми позволява да комитвам промени директно в репото на проекра си без да трябва да използвам *git bash* или *GitHub Desktop*.

Глава 2

Изисквания и структура

2.1 Функционални изисквания към разработката

Текущият проект цели реализирането на мултиплеър игра с карти като уеб приложение на програмния език *Ruby*, както и чрез фреймуърка *Rails*. Реализация чрез употребата на *Hotwire* gem-а.

2.1.1 Изисквания към потребителския интерфейс

- Възможност за регистрация и вход на потребител, използвайки *Devise* gem-а.
    - Допълнително поле за потребителско име.

- Начална страница на приложението, съдържаща:
    - Бутон „New Game“ за създаване на нова игра.
    - След присъединяване на четирима играчи, играта ни отвежда до екрана, на който се играе.

- Екран, на който се визуализира тестето и героят, с който играем, както се показват потребителските имена и точките живот на опонентите ни.
    - Използва се *Turbo* и *Action Cable* за изпращане на информация.

- Тук се играе играта, реализация на бизнес логиката за играта и на геймплей логиката.
- Използва се *Hotwire* gem-а за актуализиране на картите на бойното поле, както и в нашата ръка.

2.1.2 Изисквания към логиката на играта

- Създаване на модели за потребителите, картите, тестетата, игрите, игрите, в които са участвали потебителите, изтеглените и изиграните карти в базата данни.

- Групиране на различните карти спрямо това от кое тесте идват
  - Имплементиране на атрибутите на картите
    - Атака;
    - Защита;
    - Възстановяване на точки живот;
    - Повторно играене;
    - Теглене на нова карта.

- Реализирането на логиката на играта
  - Раздаване на 3 карти на всеки играч в началото на играта.
  - Ротация на ходовете и прилагане на различните атрибути на изиграните карти.
  - Изчисляване на точки живот, както и на щитове.
  - Край на играта за потребител с 0 точки живот.

- Определяне на край на играта

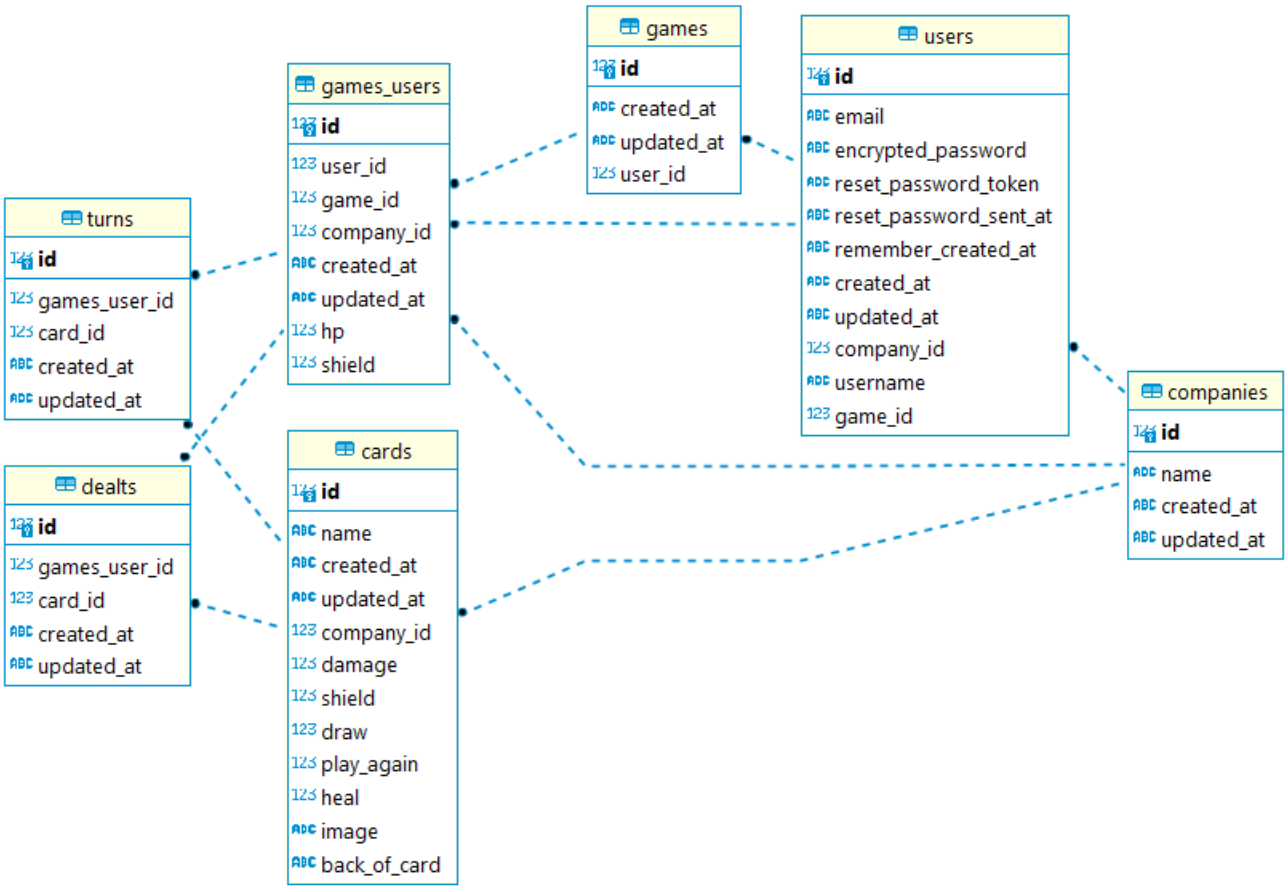
- Дава се възможност на потребителите, приключили играта да започнат или да влязат в нова.
- Класация на играчите спрямо мястото, на което са завършили.

2.1.3 Проектиране на блокова схема на базата данни и структура

За реализация на проекта се нуждаех от няколко модела, започвайки от първия – User, който сам по себе си не съдържа препратки освен препратката към текущите игра и тесте. Тук идва и Game моделът, който е нещо като черупка, тъй като информацията за играчите не се запазва в него. Единствено съдържа препратка към създателят на играта. За да се осъществи връзка между потребителите и игрите има междинен модел – GamesUser, който съдържа както и информация за потребител и игра, така и за избрано тесте. Моделът на тестетата трябваше да бъде кръстен Deck, но понеже частта от кода за връзките между картите и тестета е взета от tutorial, не смених името на този модел, който съдържа само име на тестето. Card моделът има препратака към тестетата и съдържа информация за имената и атрибутите на картите, както и пътища към челното и задното изображение на дадена карта. За да бъде реализирана самата игра трябваха още два модела – Dealt и Turn. И двата модела имат сходни полета – препратка към GamesUser и препратка към Card, но се използват за различни цели. Dealt инстанцията е еквивалентна на раздадена или изтеглена карта, а Turn – на изиграна. На фигура 2.1 е показана схемата на базата данни.

Потребителите създават акаунти, в които се пази **Login** информацията, както и текущото тесте, след което имат опция за създаване на игра. Играта служи като черупка за играчите и съдържа само id-то на създателят й, а про присъединяване към дадена игра се създават инстанции на GamesUser обектът, като по този начин се записва информацията за участието на потребител в

дадена игра. Този обект съдържа и колона, която съдържа различна стойност в зависимост от състоянието на играта за него – ако потребителят все още е в играта и ако е загубил. Щом загуби или в края на играта даден потребител има възможността да се присъедини към нова игра.



Фигура 2.1: Таблици и взаимотношенията между тях в базата данни

2.2 Подбор на езика и фреймуърка

На представянето на дипломните ръководители сегашният ми ментор представи готова тема за игра с карти, реализирана като уеб приложение, използвайки *Rails* фреймуърка. Идеята, че мога да постигна желаното, макар и с нещо, което не съм учил ми хареса и заради това се спрях на тази технология.

Дипломният ми ръководител беше възложил тази тема като една от готовите за да могат дипломантите му да се запознаят с нов език и фреймуърк, както и за да могат да се научат на работа с технологиите в калибъра на *Rails* и като цяло Web Development-а.

Едно от главните предимства в употребата на *Ruby* и *Ruby on Rails* е простотата на езика и улесняването на програмистите, използващи технологията. Принципите, внедрени във фреймуърка позволяват по-бърза разработка с по-постоянен код. Освен това *MVC* архитектурата разделя различните части на приложението нагледно, което позволява с лекота да се добавят нови функционалности.

Макар и прост, фреймуъркът е сравнително мощен и в последните години набира популярност, особено в startup общността. През 2021 година нова библиотека на име *Turbo* се интегрира в *Rails*, тя дава още повече възможности за разработка на софтуер. Позволява на разработчиците да създават уеб приложения, които работят в истинско време с почти никакъв *JavaScript,* възможна е реализацията и без никакъв *JavaScript,* но това представлява повече писане. Благодарение на тази библиотека Rails става още по-достъпен фреймуърк за начинаещи.

Длъжен съм да спомена обаче, че фреймуърка може да не е най-добрият избор за high-performance приложения. Макар и *CoC* принципът да помага, в същото време дава по-малко контрол на разработчиците спрямо структурата на приложението, което го прави не толкова гъвкаво колкото други фреймуъркове. Също така, може да се каже, че за програмистите, нови към уеб програмирането и незапознати с някои от основните му принципи, *Rails* може да бъде малко по-труден за разбиране и научаване.

С всичките си недостатъци фактът, че фреймуъркът е много подходящ за създаването на функциониращи проекти за кратко време, е неуспорим. Простотата на езика и наложените принципи ввъ фреймуърка правят работата с него приятна щом принципите бъдат разбрани.

2.3 Подбор на развойната среда

За работа и тестване на проектът си не намерих особен проблем с изборът на среди. От години насам използвам *Visual Studio Code* за писане и тестване на код, общността на editor-ът, интегрираният терминал и производителността и стабилността на продукта го правят добър избор за разработката на multiplayer игра с карти.

Както споменах по-рано трябваше да използвам нещо, с което да наглеждам, а ако е нужно да променям базата данни. *Visual Studio Code* предоставя възможността за прегледа на таблиците в базата данни, не може да се правят промени по нея, нито е интегрирана функционалност за изпълняване на SQL скриптове. Точно поради таза причина се спрях на *DBeaver*, който освен да визуализира таблици в базата данни, позволява да се правят и промени по тях, както и да се създават ER диаграми.

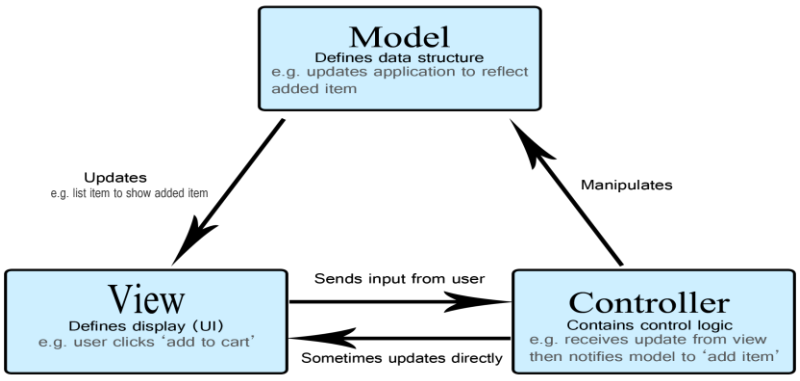
2.4 Структура на кода

Има много различни начини, по които може да се разработи multiplayer уеб приложение за игра с карти, но смятам, че *Model-View-Controller* архитектурата е най-подходяща за този проект.

Както съм споменал по-горе, Rails поддържа точно тази архитектура, което би направило работата ми с други невъзможна заради заданието ми.

В основата си кодът се разделя на 3 основни части:

- Model – компонентът, съдържащ обектите и бизнес логиката. В случая – потребителите, картите, игрите и останалите модели, които съм описал се съдържат на този слой.
- View – тук се съдържа потребителският интерфейс на приложението. От този слой зависи по каквъ начин ще бъде представена информацията до потребителят. Тук се съдържат изгледите за картите, екраните, на които се играя игрите и всичко останало, което играчите виждат.
- Controller – компонентът, който отговаря на заявките от браузъра и менежира взаимодействието между изгледа и модела.



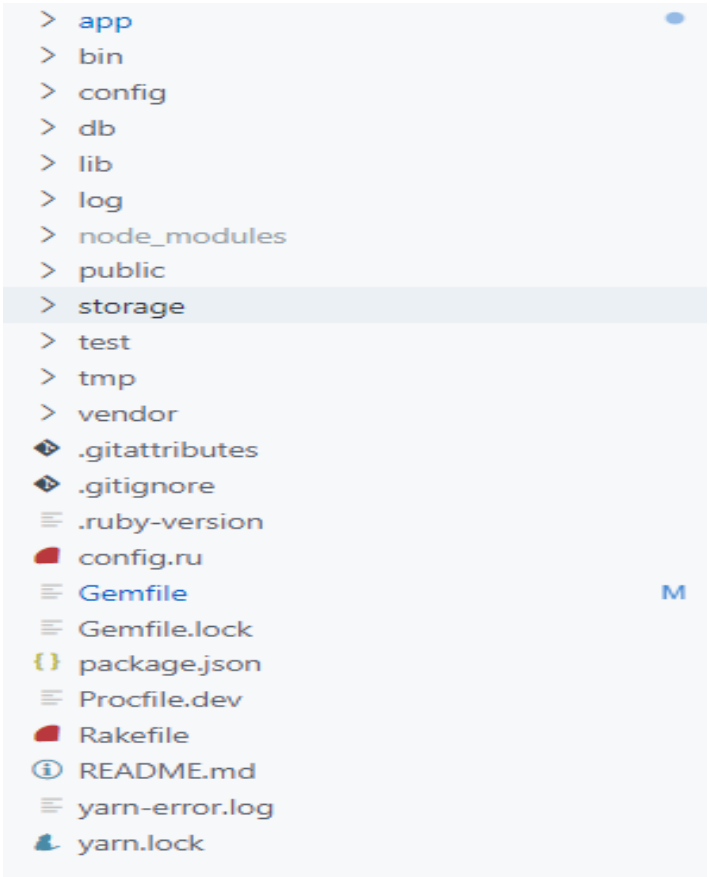
Фигура 2.2: Диаграма, изобразяваща MVC архитектурата

Глава 3

Реализация

3.1 Структура на проекта

Създаването на нов Rails проект с команда, в случая ***“rails new card-game --css=sass --javascript=esbuild --database=sqlite3“*** се създава нов проект с предефинирана структура(Фигура 3.1).



Фигура 3.1: Шаблон за нов Rails проект.

В основата на приложението седят три папки – *app*, *config* и *db*. В първата папка се съдържат най-важните неща за приложението - моделите, контролерите и изгледите, както и папка, съдържаща изображенията, custom CSS-а и *JavaScript* файловете.

Бизнес логиката и моделите са енкапсулирани в ***.rb*** файлове, които съдържат *Ruby* код и седят в *models* папката. Изгледите са представени в ***html.erb*** файлове, които съдържат *html* и вграден *Ruby* код(Embedded Ruby Code).

В *db* се съдържа всичко, свързано с базата данни – това включва миграциите, дефинициите на таблиците и други неща, които са свързани с базата данни.

В *config* папката се намират файловете за конфигурацията на приложението, които настройват какво и него, така и базата данни, различните пътища, които могат да се достъпят в браузъра.

Освен тези папки има и много други, но те не са от толкова голямо значение, а някои от тях дори не са били отваряни по време на разработка. Други два важни файла са *Gemfile*, съдържащ gem-овете, с които се разботи в проекта, както и *Procfile.dev*(Фигура 3.2), който се използва при стартиране на приложението чрез команди.

Procfile.dev

```
1   web: unset PORT && bin/rails server
2   js: yarn build --watch
3   css: yarn build:css --watch
4   css: ruby bin/rails tailwindcss:watch
```

Фигура 3.2: Променението съдържание на Procfile.dev файлът, за да може сървърът и watch-овете да бъдат пуснати на моята система.

3.2 Основен сценарий

В основата си играта не се различава много от други игри с карти, играе се от четирима играчи, всеки от които избира тесте с различни карти. Първи на ход е най-младият играч, като всеки ход играчът тегли карта и изиграва някоя от наличните си, те имат атрибути, като главните са атака, защита и възстановяване на точки живот. Щом даден потребител достигне нула точки живот губи играта, докато другите продължават ходовете си, докато не остане само един играч, който се счита за победител.

За да реализирам това имплементирах възможността за създаване на потребителски акаунт с помощта на *Devise* библиотеката, след като бъде създаден акаунт, пред потребителят се открива възможността да създаде или да се присъедини в игра чрез линк. В играта е реализирана визуализация на ръката на играча с помощта на Turn, Dealt, Card моделите и подзаявки. Имплементирана е възможността да се играят карти, както и логиката на играта. За да може в една игра да играят четирима потребители се осъществява връзка между тях с помощта на *Hotwired* библиотеката чрез стъпване на *WebSocket*, по-специфично *Turbo Streams* и *Turbo Frames*. При загуба потребителят може да се присъедини към нова игра.

3.3 Създаване на потребителски модел

Туториалът за употребата на *Turbo,* който следвах ми помогна до голяма степен със запознаването с технологиите, особено с *Hotwire*.

Той се състои от 12 глави, всяка, от които обхваща различна тема – от създаване на нов проект, до добавяне на *Turbo frame*-ове, дори и писане на собствен *CSS*. Всичко това е добре описано, включени са и схеми за по-лесно възприемане.

Преди да се стигне до работата по сегашното ми репо, бях направил няколко пробни проекта, като някой от тях не бяха създадени с нужните папки, а в някои случай не успявах и да пусна сървъра.

Създаването на Rails проект е сравнително лесно за *Mac* и *Ubuntu* потребители, но на *Windows* процесът е по-сложен.

Като за начало трябваше да изтегля езикът, на който е построен фреймуърка – *Ruby*, следваше инсталацията на *Yarn* и *Node.js*. После дойде и изборът на база данни, туториалът, който следвах използваше *PostgreSQL*, която наистина е подходяща база за проект с такава тежест, но тъй като настройването ѝ беше по-сложно реших да използвам *SQLite3*, която върши идеална работа за разработка, тъй като няма много потребители и игри, които се играят.

Повечето туториали за *Ruby on Rails* в интернет са писани за потребители на *Mac* и *Ubuntu*, както споменах по-горе, така че командите, които писах в command prompt-а си бяха малко по-различни от тези, които бяха показани.

След като оправих връзката между базата данни и проекта си и успях да пусна сървъра започнах да проучвам *Devise* gem-а, който служи за създаване на потребители и автентикация. Като почти всеки друг gem, трябваше да го добавя в *Gemfile*-а на проекта си(Фигура 3.3).

Оттам се използва ***"bundle install"***, по този начин се инсталират gem-овете, а след това се извиква и rails generate ***"devise:install"***, което завършва инсталацията на Devise.

```
4   ruby "3.1.3"
5
6   gem "rails", "~> 7.0.4", ">= 7.0.4.1"
7
8   gem "simple_form", "~> 5.1.0"
9
10  gem "devise"
11
12  gem "sprockets-rails"
13
14  gem "sqlite3", "~> 1.4"
```

Фигура 3.3: Откъс от Gemfile файлът, съдържащ имената на gem-овете и в някои случаи пояснение за версията им.

Създаването на потребителският модел се извършва с друга команда – ***"rails generate devise User"***, моделът вече е изграден и не е нужно да добавяме полета и проверки. Той съдържа много полета, но най-значимите от тях или поне най-значимите за моя проект са полетата за *email*, *encrypted_password*, благодарение на тях с дадените изгледи от библиотеката сравнително лесно се имплементират **Login** и **Sign up** страници.

Но за цел имам създаването на акаунт, в който има и *Username*, който ще се визуализира по време на игра. Първото нещо, което направих е да създам миграция, с която добавих *Username* полето в потребителския модел. Но за да реализирам приемането на потребителско име като поле при създаване на нов user, трябваше да генерирам *Devise* контролерите, чрез които може да се модифицира поведението на библиотеката.

Следваше да нанеса промени по изгледите, за да може при създаването на акаунт да се въведе и потебителско име. Това стана като добавих полета за потребителско име в *new.html.erb* и *edit.html.erb* изгледите във *views/devise/registrations* папката.

Потребителите вече можеше да се логват и с потребителско име, но освен quote editor-а, който бях построил нямаше нищо друго, което можеше да видят.

Quote-овете при мен бяха преименувани като карти, добавих и колона за изображение, за да могат после да се визуализират в ръката на играча.

Компанията също беше добавена като допълнителен атрибут за създаване на потребител, по този начин можеше да ъпдейтвам това поле, когато започне игра и бъде избрано тесте.

3.4 Създаване на модел за картите

Моделът на картата се състои от колона за името на картата, колона с препратка към тестето, от което принадлежи, колони за атрибутите, които по подразбиране имат стойност, равна на нула.

Друго нещо, което отличава картите една от друга са снимките, целта ми беше да създам игра, подобна на Dungeon Mayhem, там всяка от картите е илюстрирана по различен начин. За да спестя време и да обърна повече време на кода преснимах изображенията и ги оформих, като стандартът беше 147x147 пиксела. След това поставих изображенията на картите, заедно с гърбовете на всяко от тестетата в папка *"cards"*, която се намира в *app/assets/images* директорията.

По този начин когато визуализирам картите мога да достъпя изображението чрез пътището към него, което е записано в *image* атрибута на картата(Фигура 3.4).

image	back_of_card
cards/sneak_attack.png	cards/rogue_back.png
cards/battle_roar.png	cards/barbarian_back.png
cards/winged_serpent.png	cards/rogue_back.png
cards/brutal_punch.png	cards/barbarian_back.png
cards/burning_hands.png	cards/wizard_back.png
cards/divine_smite.png	cards/paladin_back.png
cards/the_goon_squad.png	cards/rogue_back.png
cards/flex.png	cards/barbarian_back.png
cards/vampiric_touch.png	cards/wizard_back.png
cards/fighting_words.png	cards/paladin_back.png
cards/shield.png	cards/wizard_back.png

Фигура 3.4: Изображения, свързани с някои от записите на карти в базата данни

За да мога да боравя с картите по време на разработа и да тествам игрите ги дефинирах като записи в *test/fixtures/cards.yml* файлът. Там описах всяка ена от картите в тестетата с пътища към снимките, стойности на атрибутите, имена и принадлежност към тесте. След това пълних базата данни с тези записи чрез ***"rails db:seed"*** командата, която извикваше seeds.rb файлът в db

папката. В този файл се съдържа командата ***"ruby bin/rails db:fixtures:load"***, с която се зареждат пренаписаните записи в базата.

3.5 Създаване и влизане в игри

Първият Game модел, който изградих се състоеше от колона, в която да се пазят идентификационните номера на играчите, броят им, както и Boolean колона за четирите тестета.

В “New” action-а на Game контролера написах логика за обработката на заявките при създаване на играта, която приема ID-то на създателя и изборът му за тесте, но в последствие разбрах, че тази методология върши работа само при създаване на играта, тъй като в *SQLite3* масиви няма и не можеше ID-тата на всички играчи да се съдържат в една колона.

След обмисляне на възможностите за създаване и присъединяване към игра на ум ми дойде следната идея – да разчиста максимално User модела, за да може да съдържа само ID-то на сегашната игра и *Login* информацията, а Game моделът да съдържа само информация за създателят на играта, като колона, в която се съдържа номерът му.

Това обаче значеше, че трябва да има междинен модел, който осъществява връзката между потрбителите и игрите, самият играч, а не потребител. Създадох модел Player, който имаше препратки към *user_id* и *game_id*. Целта ми беше при създаване на акаунт за всеки потребител да бъде създадена и Player инстанция с номерът му.

Но след няколко дни писане нямах особен напредък, тъй като при създаването на потребител не можах правилно да създам и играч. След като обсъдих проблемът с дипломният си ръководител, той предложи нещо сходно – вместо да се създава инстанция на този модел за всеки акаунт, да се създава инстанция при присъединяване в игра, по този начин ще можеше да пази

история за игрите, в които потребителят е участвал. Тогава обсъдихме и цялостното оформление на базата данни.

Преработих Game моделът и премахнах колоната, която трябваше да съдържа идентификационните номера на потребителите и броят им, вместо тези двете колони оставих само една колона в таблицата – *user_id*, която представлява препратка към създателят на играта(Фигура 3.5).

app > controllers > games_controller.rb > Ruby > GamesController

```ruby
class GamesController < ApplicationController
    before_action :set_game, only: [:show, :edit, :destroy, :setup_game, :play]

    def new
        @game = Game.new
    end

    def create
        @game = Game.new(user_id: current_user.id)


        if latest_game.present?
          redirect_to game_path(latest_game.game_id), alert: "You are already in another game."
          return
        end

        if @game.save
            redirect_to @game, notice: "Game was successfully created."
          else
            render :new, status: :unprocessable_entity
        end
    end
```

Фигура 3.5: Откъс от контролерът на Game моделът, във функцията Create, отговаряща на POST заявки се записва номерът на потребителя, създал играта.

Моделът GamesUser, който съм споменал във втора глава върши същата работа, каквато трябваше да върши Player моделът, а именно осъществяване на връзка между Game и User моделите за да може играчите да се присъединяват в игри.

Отново генерирах модел в базата данни, който имаше препратки към потребителят, играта и тестето. Използвах командата ***“rails g migration create_games_users user:references game:references company:references”***, което добави нов файл към миграциите ми в *db/migrate* папката(Фигура 3.6).

db > migrate > 20230217104503_create_games_users.rb > ...

```ruby
class CreateGamesUsers < ActiveRecord::Migration[7.0]
  def change
    create_table :games_users do |t|
      t.references :user, null: false, foreign_key: true
      t.references :game, null: false, foreign_key: true
      t.references :company, null: false, foreign_key: true

      t.timestamps
    end
  end
end
```

Фигура 3.6: Файл, съдържащ миграция за създаване на нова таблица в базата данни

След създаването на каквато и да е миграция, трябва по някакъв начинн да се направят промените освен по миграционните файлове, така и по самата база данни. За целта се използвах ***"rails db:migrate"*** командата, която е отговорна за нанасянето на промени по базата данни спрямо миграциите в приложението.

След създаване на игра, потребителят има възможността да се присъедини към нея, като натисне бутон, който го изпраща до *new.html.erb*. На този изглед се рендерира *Simple Form* за избор на едно от наличните тестета в играта(Фигура 3.7).

Във формата се зарежда нова инстанция на GamesUser модела, създадена от "New" action-а на контролера, като тя се променя след изпращане. Формата съдържа колекция от радио бутони, които показват наличните опции за тестета с помощта на подзаявка, която връща колекция от играчи, свързани с тази игра. След това с друга подзаявка се връща релация с наличните тестета и те се представят като избори на потребителя.

Щом потребителят се присъедини в игра заедно с още трима играчи се изобразява екранът, на който играта се играе.

app > views > games_users > _form.html.erb

```erb
<%= simple_form_for @games_user,
                    html: {
                      class: "game form",
                      data: { turbo: false }
                    } do |f| %>
  <% if @games_user.errors.any? %>
    <div class="error-message">
      <%= @games_user.errors.full_messages.to_sentence.capitalize %>
    </div>
  <% end %>


    <%= f.hidden_field :game_id, value: @games_user.game_id || params[:game_id] %>


    <% game_users = GamesUser.where(game_id: @games_user.game_id || params[:game_id]) %>
    <% taken_company_ids = game_users.pluck(:company_id).uniq %>
    <% available_companies = Company.where.not(id: taken_company_ids).where.not(name: 'Default') %>
    <%= f.collection_radio_buttons :company_id, available_companies, :id, :name %>

  <%= f.submit "Join", class: "btn btn--secondary" %>
<% end %>
```

Фигура 3.7: Форма за присъединяване към игра, потребителят трябва да избере едно от наличните тестета.

3.6 Реализация на бизнес логиката

За да реализирам логиката, спомената в началото на главата, създох още 2 модела – един за раздадени карти и един за изиграни карти. И двата модела са сходни по това, че съдържат идентификаторът на картата, за която се отнасят, както и препратка към GamesUser-а, за когото се отнасят(Фигура 3.8).

След това в моделът *games_user.rb* дефинирах функция за настройването на началото на една игра, която се вика след създаване на нова инстанция на GamesUser модела, по-конкретно след присъединяване на играч в игра. Тя проверява дали има общо четири потребителя, свързани с дадената игра. В случай, че това условие е изпълнено се създават по 3 инстанции на Dealt за всеки един от играчите, като те се взимат от картите, които не са били

раздадени по прозволен начин. По този начин се реализира тегленето на три карти в началото на играта.

```
create_table "dealts", force: :cascade do |t|
  t.integer "games_user_id", null: false
  t.integer "card_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index ["card_id"], name: "index_dealts_on_card_id"
  t.index ["games_user_id"], name: "index_dealts_on_games_user_id"
end

create_table "turns", force: :cascade do |t|
  t.integer "games_user_id", null: false
  t.integer "card_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.integer "damaged_games_user_id"
  t.integer "healed_games_user_id"
  t.integer "shielded_games_user_id"
  t.index ["card_id"], name: "index_turns_on_card_id"
  t.index ["games_user_id"], name: "index_turns_on_games_user_id"
end
```

Фигура 3.8: Откъс от *schema.rb* файлът, съдържащ информация за полетата на таблиците в базата данни

За реализацията на ротация на играчите и определяне на сегашен ход съм трябваше да дефинирам друга функция, която връща *Boolean* отговор в същия модел(Фигура 3.9).

```
def is_current_turn?
    players = GamesUser.where(game_id: self.game_id, completed_at: nil).order(:created_at).to_a
    my_position = players.index(self)
    previous_player_id = players[(my_position - 1) % players.count].id

    last_turn = Turn.joins(:games_user).where(games_users: { game_id: self.game_id }).last
    last_dealt = Dealt.joins(:games_user).where(games_users: { game_id: self.game_id }).last

    if last_turn.nil?
        #v nachaloto na igrata nqma playove, rezultatut se opredelq sprqmo koi e vlqzul purvi
        if last_dealt.games_user_id == previous_player_id
            return players[0].id == self.id
        else
            return false
        end
    elsif last_turn.games_user_id == previous_player_id && last_dealt.games_user_id == previous_player_id
        #ako posledniq hod e na igracha predi men znachi sega e moi red
        return true
    else
        return false
    end

    #dovurshi
end
```

Фигура 3.9: Функция за определяне на реда в игра

В тази функция играчите в играта биват представени като масив, сортират се само тези играчи, които все още не са приключили играта и се подреждат по времето, в което са се присъединили. След това се намират последния играч и последния ход и се правят няколко проверки за определяне на реда.

Първо се проверява дали последния ред е празен, това се случва само в началото, когато нито един от играчите не е играл карта. Вътре се проверява дали последната раздадена карта е на предишния играч и в този случай функцията връща *true*, ако това не е така връща обратното.

Втората част на *if* statement-а проверява дали последният ход и последната раздадена карта са на предишния играч, в случай, че това е така се връща *true*. Всички други случаи показват, че не е наш ред.

Логиката за ходовете беше налице, но изиграните карти все още не правеха нищо, а дори и да правеха – трябваше да се дефинира край на играта за играчите. За целта трябваше функция в *turn.rb*, която да прилага атрибутите на картата след като бъде изиграна(Фигура 3.10).

Функцията се извиква след запазване на инстанция на ход в базата данни и търси картата по препратката към нея в Turn модела.

Когато картата има атрибут за щета, се визуализират радио бутони, които показват текущите участници в играта за да може бъде избран противникът, на когото ще бъде нанесена щета. По този начин се запълва колоната *damaged_games_user_id* в Turn модела, която седи без стойност ако картата е нямала атрибут за щета. След като бъде проверен моделът се изчислява животът на противника, като се взима предвид и щитът му. Точките живот и щитовете имат долна и горна граница – нула и десет точки, които се понижават или повишават ако границите бъдат прекрачени.

Следващата проверка е за върнати точки живот, играч не може да лекува друг, освен себе си, заради това при изиграване на карта, която има *heal*

атрибут по-голям от нулата *healed_games_user_id* колоната се попълва с номерът на потребителят, изиграл картата.

Същата логика се имплементира и при пресмятането на щитове, само че е използвана трета колона в Turn модела – *shielded_games_user_id*. И при връщане на точки живкот, и при добавяне на щит се проверяват горната и долна граница за да не се работи с некоректни стойности.

Единственото нещо, което оставаше да се имплементира е функция за следене на края на играта, след което да се позволи на играчът да се присъедини към нова игра.

Към GamesUser модела добавих нова колона – *completed_at*, която да променя стойността си ако играчът е загубил играта. Във функцията има проверка дали точките живот са равни на нула, ако това условие е изпълнено колоната се запълва със времето, в което това е станало(Фигура 3.11). За целта се използва ***"Time.zone.now"*** методът, който връща сегашното време спрямо конфигурацията в *config.time_zone* настройките на приложението.

По този начин, функцията работи както трябва, докато не остане само един играч. Щом това стане, той остава в играта без изход, заради това трябваше да се направи още една проверка, която проверява останалите играчи и ако има само един останал се запълва и неговата *completed_at* колона. Функцията се извиква в тази, която нанася атрубутите на картите, но само когато картата има атрибут за щета, защото само тогава може един играч да е загубил играта.

```ruby
def apply_card_stats
  card = Card.find_by(id: card_id)

  if card.present?
    if damaged_games_user_id.present? && card.damage > 0
      damaged_user = GamesUser.find_by(id: damaged_games_user_id)
      damage_after_shield = [card.damage - damaged_user.shield, 0].max
      damaged_user.update(hp: [damaged_user.hp - damage_after_shield, 0].max)
      damaged_user.update(shield: [damaged_user.shield - card.damage, 0].max)

      damaged_user.check_end_game

    end

    if healed_games_user_id.present? && card.heal > 0
      #ne sum izpolzval .max i .min tuk zashtoto purviq put ne proraboti, az sum time efficent
      healed_user = GamesUser.find_by(id: healed_games_user_id)
      health = healed_user.hp + card.heal
      if health > 10
        health = 10
      elsif health < 0
        health = 0
      end
      healed_user.update(hp: health)
    end

    if shielded_games_user_id.present? && card.shield > 0
      shielded_user = GamesUser.find_by(id: shielded_games_user_id)
      shield = shielded_user.shield + card.shield
      if shield > 10
        shield = 10
      elsif shield < 0
        shield = 0
      end
      shielded_user.update(shield: shield)
    end
  end
end
```

Фигура 3.10: Функция за нанасяне на атрибутите на изиграна карта.

```ruby
def check_end_game
    if hp <= 0
        update(completed_at: Time.zone.now)
    end

    remaining_players = GamesUser.where(game_id: self.game_id, completed_at: nil)
    if remaining_players.count == 1
        remaining_players.last.update(completed_at: Time.zone.now)
    end
end
```

Фигура 3.11: Функция за проверка края на играта.

За да може един играч да не се присъединява в друга игра, докато е във все още течаща, и за да може да се присъединява в игра след края на една създадох нов *helper метод* и промени по *games_user_controller.rb* файлът.

Дефинирах метода в *application_controller.rb* файлът, там се намира сегашната игра на потребителя чрез подзаявка и резултатът се записва в *latest_game* метода(Фигура 3.12).

```ruby
def latest_game
  @latest_game ||= GamesUser.where(user_id: current_user.id, completed_at: nil).last
end

helper_method :latest_game
```

Фигура 3.12: Дефиниция на *helper* метод за сегашната игра на потребителя

Проверката в контролера на GamesUser се състои в това, че ако сегашната игра е налична, потребителят се пренасочва към екранът, на който тя се играе, придружено от съобщение, че вече е в игра(Фигура 3.13).

```ruby
def create
  @games_user = GamesUser.new(games_user_params.merge(user_id: current_user.id))
  #@games_user.hp = 10
  #put default values in model not here

  if latest_game.present?
    redirect_to game_path(latest_game.game_id), alert: "You are already in another game."
    return
  end

  if @games_user.save
    #current_user.update(game_id: @games_user.game_id)
    current_user.update(company_id: @games_user.company_id)
    respond_to do |format|
      format.html { redirect_to game_path(@games_user.game), notice: "Successfully joined game." }
      format.turbo_stream{ flash.now[:notice] = "Successfully joined game." }
    end
  else
    render :new
  end
end
```

Фигура 3.13: Create action-а в контролерът на GamesUser модела.

3.7 Реализация на промени в реално време

За да не трябва потребителите да презареждат непрестанно страницата, докато не дойте техният ред, добавих broadcast функционалност с Turbo. След като бъде запазена инстанция на Turn модела се преминава през проверка дали е ред на следващия играч, която в случай, че премине, се излъчва бутонът за теглене на карта. Ако проверката не премине, се зарежда *_empty.html.erb* файлът, който не съдържа нищо. По този начин потребителят тегли само тогава, когато трябва.

Глава 4

Потребителско ръководство

4.1 Системни изисквания

По време на разработка съм използвал *Windows 11*, който има вграден *WSL*(Windows Subsystem for Linux). Необходима е машина с минимална версия на ОС *Windows 10* с настроен *WSL* или *Windows 11*.

За да се стартира правилно приложението ще е нужно да бъдат изтеглени и програмният език *Ruby* чрез следния линк: https://www.ruby-lang.org/en/downloads/, като след това ще трябва да се изтегли и *Rails*, който в същността си е gem. Това става след извикване на ***"gem install rails"*** командата.

За да може да функционира правилно *JavaScript*-а в приложението трябва да се изтегли и *Node.js* чрез този линк: https://nodejs.org/en/download/. Освен *Node.js*, за да сработи тестването, машината се нуждае от Yarn, посетете следния линк за инсталация: https://yarnpkg.com/getting-started/install.

Следва да се инсталира *Redis* върху машината, като преди това трябва да бъдат последвани инструкциите за инсталация на *Redis* от официалния уебсайт чрез този линк: https://redis.io/docs/getting-started/installation/install-redis-on-windows/.

4.2 Инсталация

За да се тества проектът, трябва да се посети хранилището на проекта чрез следния GitHub линк: https://github.com/horhe579/Diploma2. След това клонирайте репото чрез *Github Desktop* или ***"git clone"*** командата.

Когато репото е клонирано и машината Ви отговаря на системните изисквания, трябва да се изтеглят нужните gem-ове и да се създаде базата данни, която да се попълни с нужната информация.

Инсталацията на библиотеките трябва да се случи в нов терминал чрез командата ***"bundle install"***, като е хубаво преди това да се провери дали *Ruby* и *Bundler* са били инсталирани правилно с командите ***"ruby -v"*** и ***"bundle -v"***. *Foreman* е библиотека, която не се инсталира като се запише в *Gemfile*, заради това трябва да се извика ***"gem install foreman"***.

За настройване на базата данни трябва в терминала, който използвате да отворите проекта и да извикате командите ***"rails db:create"*** и ***"rails db:migrate"***, по този начин в базата данни ще бъдат създадени нужните таблици.

За да бъдат заредени картите в базата данни се извиква ***"rails db:seed"*** командата. Сега остава само да бъдат пуснати сървърите и watch-овете и проектът е готов за тестване.

Първият сървър, който трябва да бъде пуснат е *Rails* сървърът, в нов терминал трябва да бъде изпълнена ***"ruby bin/rails server"*** командата, след това може да се посети http://localhost:3000/ линкът и да се тества(Фигура 4.1), но преди това трябва да се пусне *Redis* сървъра за комуникацията чрез *websocket*, както и watch-овете за да могат да бъдат заредени промените по *JavaScript* и *CSS* файловете.

Фигура 4.1: Console log-ът, който се показва при пускане на *Rails* сървъра

За *Redis* сървъра трябва нов терминал, в случая *WSL* терминал, тъй като *Redis* няма поддръжка само на *Windows.* След като е отворен терминалът трябва да бъде извикана командата ***"sudo service redis-server start"*** и да се въведе *Ubuntu* паролата на потребителят.

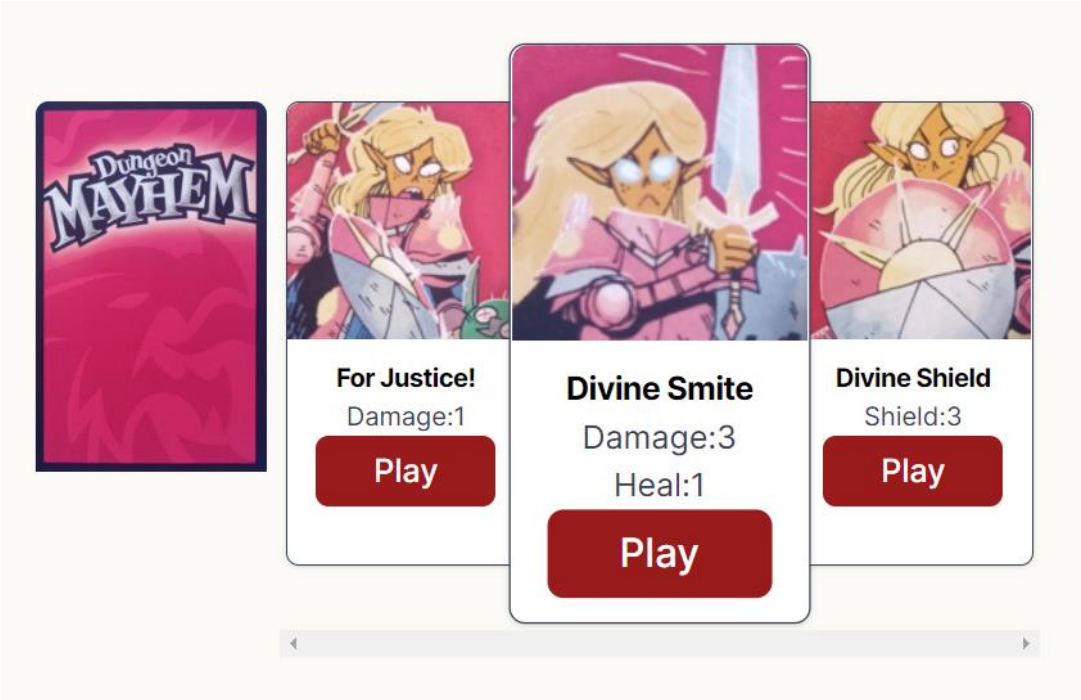
Командите за пускане на watch-овете се намират в *Procfile.dev* файлът, като за всяка от тях трябва да се използва нов терминал. Щом всичко е изпълнено се очаква посещаването на гореспомантия адрес да рендерира **Login** страницата.

4.3 Употреба

За да изпробвате играта, трябва първо да създадете акаунт, това е първото нещо, което се визуализира, щом отворите уеб приложението(Фигура 4.2).

Щом влезете в акаунта си или направите нов, ще имате опцията да създадете нова игра. Когато се присъединят всички потребители, ще се визуализира екрана, на който играте се играе и ще видите картите в ръката си(Фигура 4.3).

Фигура 4.2: **Login** изгледът, чрез който се влиза във вече създаден акаунт



Фигура 4.3: Визуализация на картите в ръката на играча.

Когато влезете в играта първият на ред е първият, влязъл в играта. Щом е ред на сегашният играч ще се появи опцията да се изтегли карта, след което и да я изиграе.

По този начин играта продължава, докато не загубят трима играчи и остане последен, който се счита за победител. След това потребителите имат опцията да се присъединяват към други игри.

Заключение

В обобщение, целта да се разработи multiplayer игра за карти с поставените от дипломния ми ръководител е изпълнена. Потребителите имат възможност да създадат нов акаунт и да игра, чийто линк да изпратят на приятели и да играят заедно с тях, като изберат тесте, всчко с различни карти.

Като бъдеща реализация могат да се имплементират или подобрят следните функционалности:

- Екран с възможност за обучение на нови играчи.
- Добавяне на още тестета и атрибути на картите.
- Имплементиране на нива на потребителя.
- Екран за Leaderboard, сортиращ играчите по опит.
- Екран, на който се виждат предишните игри и резултатите им.
- Имплементиране на чат в играта.

Източници

1. Rails Guides
   https://guides.rubyonrails.org/

2. Hotrails Turbo Tutorial
   https://www.hotrails.dev/turbo-rails

3. Документацията на Turbo Rails
   https://github.com/hotwired/turbo-rails

4. Документацията на Devise
   https://github.com/heartcombo/devise

5. Уебсайтът на Tailwind CSS
   https://tailwindcss.com/docs/guides/ruby-on-rails

6. Уебсайтът на Redis
   https://redis.io/docs/getting-started/installation/install-redis-on-windows/

7. Ruby on Rails YouTube курс
   https://www.youtube.com/watch?v=fmyvWz5TUWg&t=3567s&ab_channel=freeCodeCamp.org

8. Документацията на Simple Form
   https://github.com/heartcombo/simple_form

Съдържание