Московский государственный технический университет им. Н.Э.Баумана

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Индивидуальное домашнее задание по дисциплине БКИТ**

**“Шахматы”**

Описание

(вид документа)

писчая бумага

(вид носителя)

5

(количество листов)

|  |  |
| --- | --- |
| ИСПОЛНИТЕЛЬ: |  |
| студент группы ИУ5-34Б | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Тарасов В.Ю. | "\_\_"\_\_\_\_\_\_\_\_\_\_\_\_\_2018 г. |

Москва - 2018

# НАЗНАЧЕНИЕ ПРОГРАММНОГО ПРОДУКТА

Веб-приложение «Шахматы» для решения проблемы, отсутствия удобных и доступных аналогичных решений в области развлечений. Веб-приложение позволяет тренироваться пользователю против ботов разного уровня сложности, для оттачивания навыков своей игры. Позволяет играть по сети против настоящих людей и соперников для получения очков рейтинга. Веб-приложение «Шахматы» позволяет создавать профиль и изменять аватар, пароль.

# ВОЗМОЖНОСТИ ПРОГРАММНОГО ПРОДУКА

## Общие сведения о программном продукте

SPA Веб-приложение «Шахматы» состоит из нескольких важных компонентов

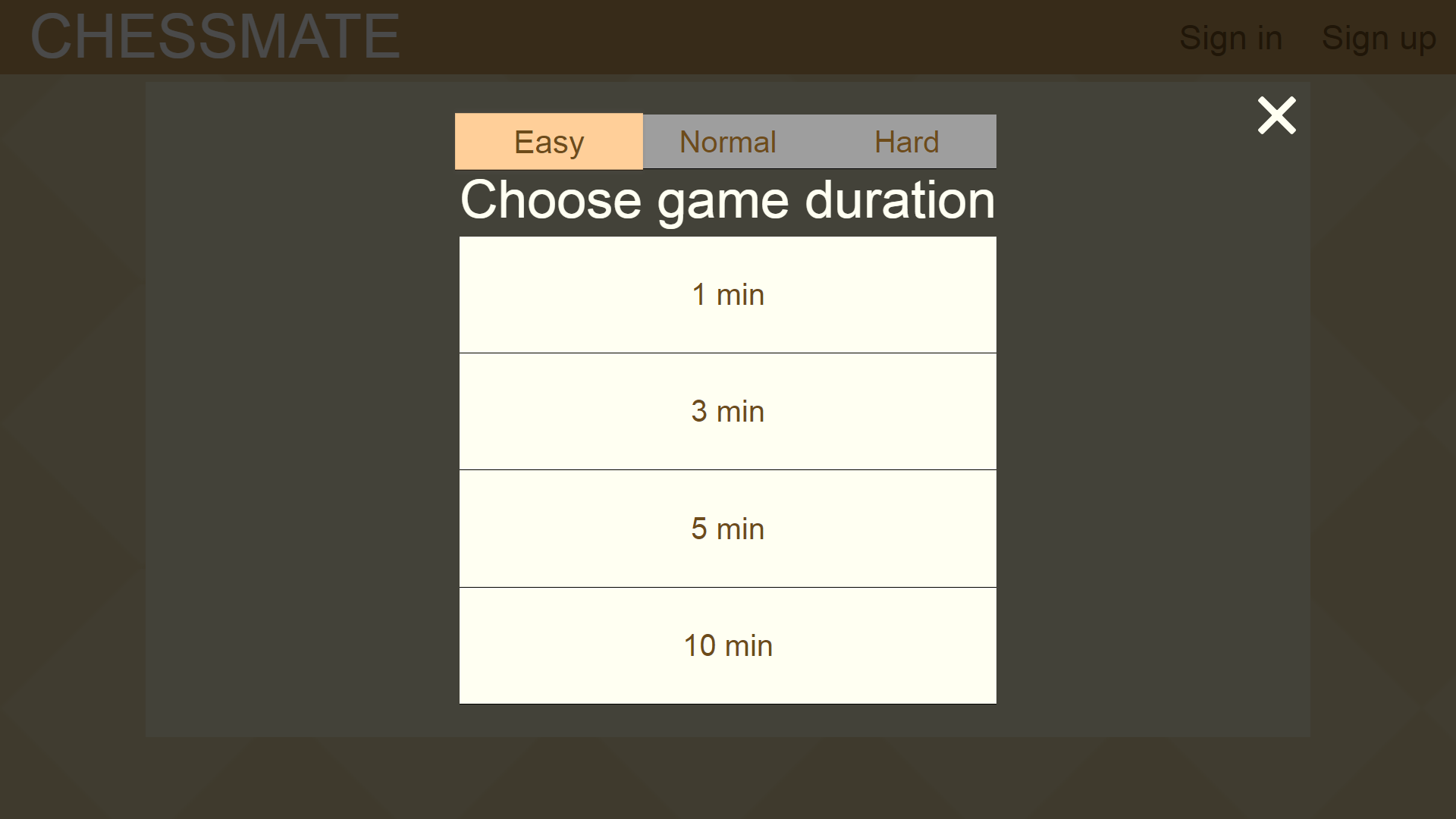
* Меню – главная страница пользователя, состоит из таких элементов как: Multiplayer, Singleplayer, About, Leaderboard. Являет начальным окном SPA приложения, которое видит пользователь при переходе на сайт.
* Шапка – находится всегда сверху приложения и позволяет быстро переходить в меню, авторизоваться и регистрироваться. Если пользователь зарегистрирован, то отображает информацию о нем и позволяет перейти в профиль или выйти из него
* Профиль – содержит информацию о пользователе: картинка, которую он выбрал, логин, email, пароль и его игровой счет. Так же профиль дает возможность сменить пароль и загрузить картинку.
* Одиночная игра – состоит из 3 режимов игры. 1 – игра против бота легкой уровни сложности, 2 – игра против бота нормального уровня сложности, 3 – игра против нейронной сети. Так же можно выбрать длительность партии: 1, 3, 5, 10 минут.
* Многопользовательская игра – состоит из 2 режимов игры. 1 – игра по сети, 2 – игра на одном телефоне один против одного. Так же можно выбрать длительность партии: 1, 3, 5, 10 минут.
* Про нас – страница, содержащая ссылки на разработчиков приложения
* Таблица лидеров – содержит таблицу игроков, отсортированных по рейтингу.
* Игра – содержит шахматную доску и компоненты описания игроков: картинка, имя, рейтинг. Кроме доски и информации о игроках еще содержит состояние игры: время оставшееся у каждого игрока, какие фигуры каждый из игроков «забрал» и кнопку «Сдаться». Управлять можно как мышкой, так и с телефона нажатием, так и голосом при нажатие на время.

Также Веб-приложение «Шахматы» можно установить, как PWA приложение на мобильный телефон, так как это современное веб-приложение. Также позволяет поделиться сайтом в социальной сети «Вконтакте».

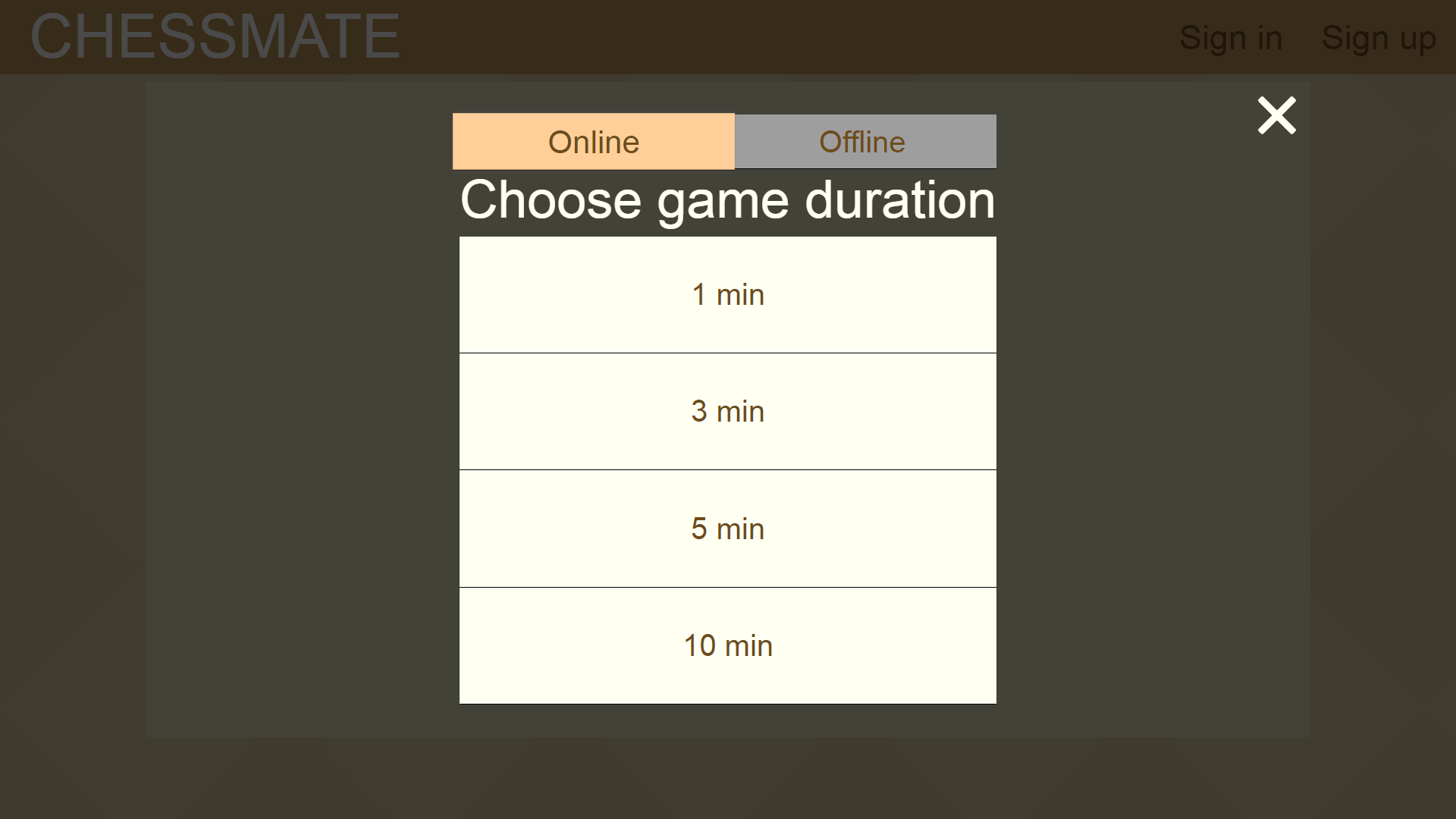
## Принтскрины страниц веб-приложения

### Меню

### Настройки однопользовательской игры



### Настройки многопользовательской игры.



### Таблица лидеров



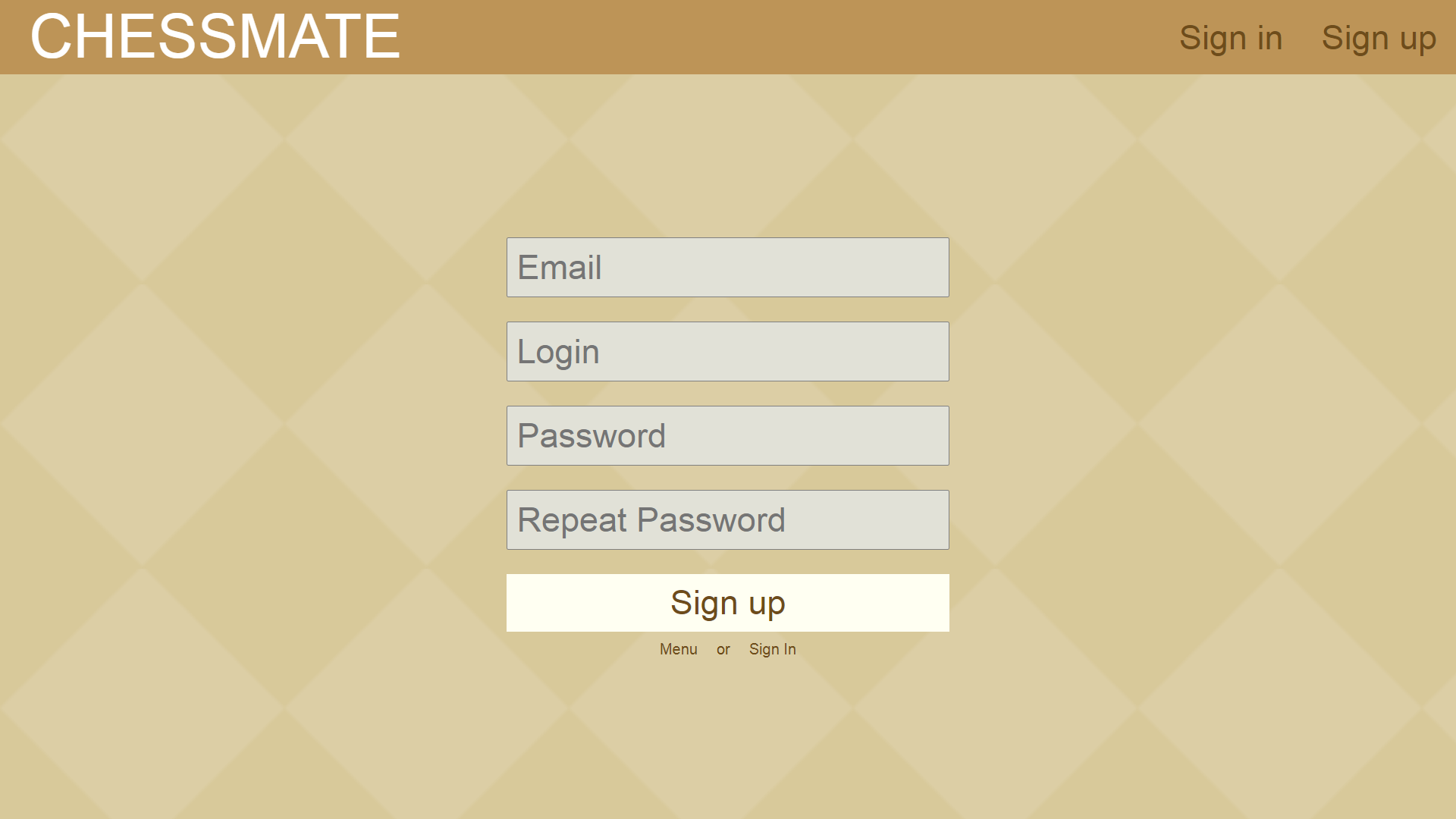
### Страница о нас



### Авторизация

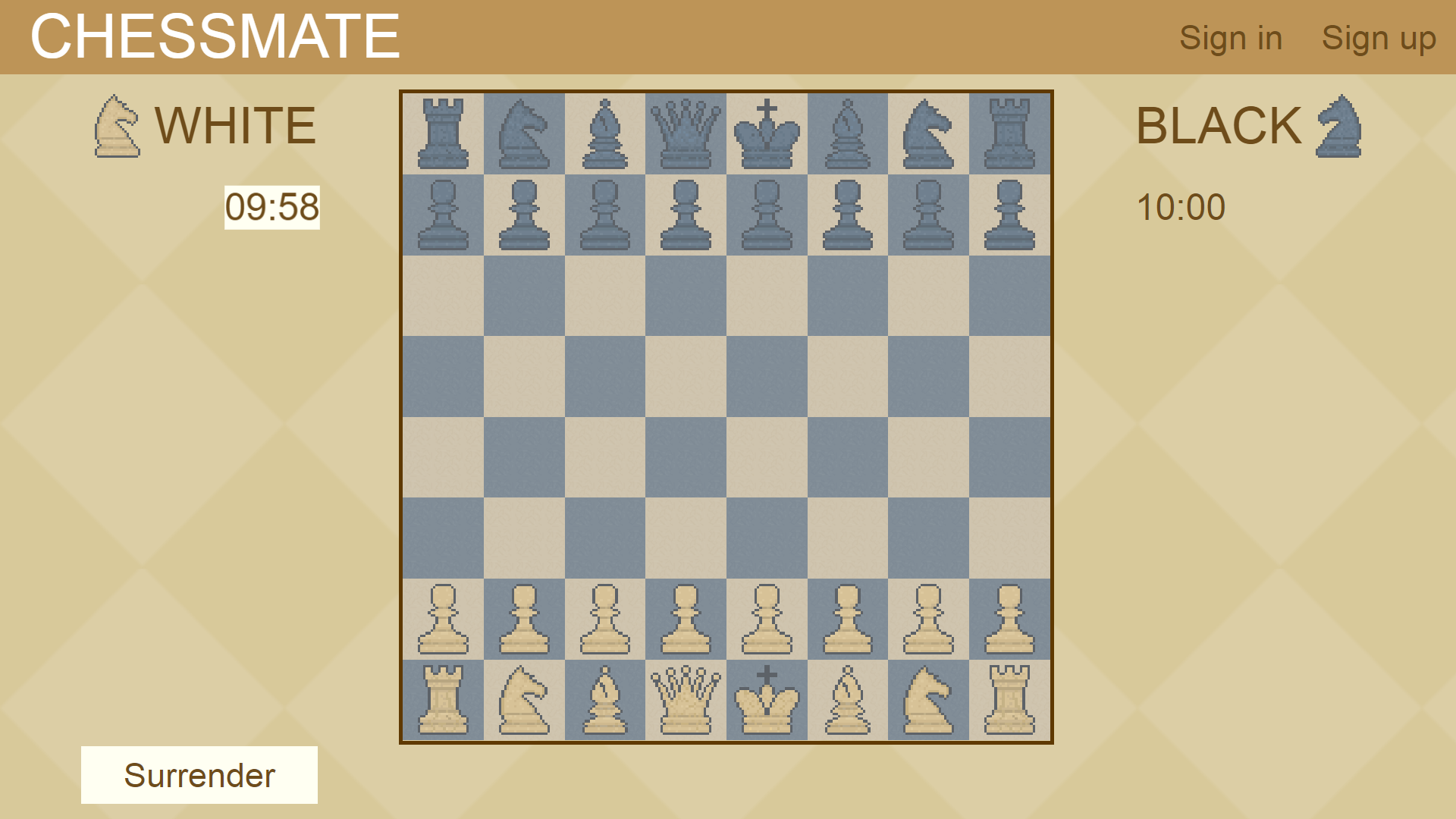


### Регистрация.

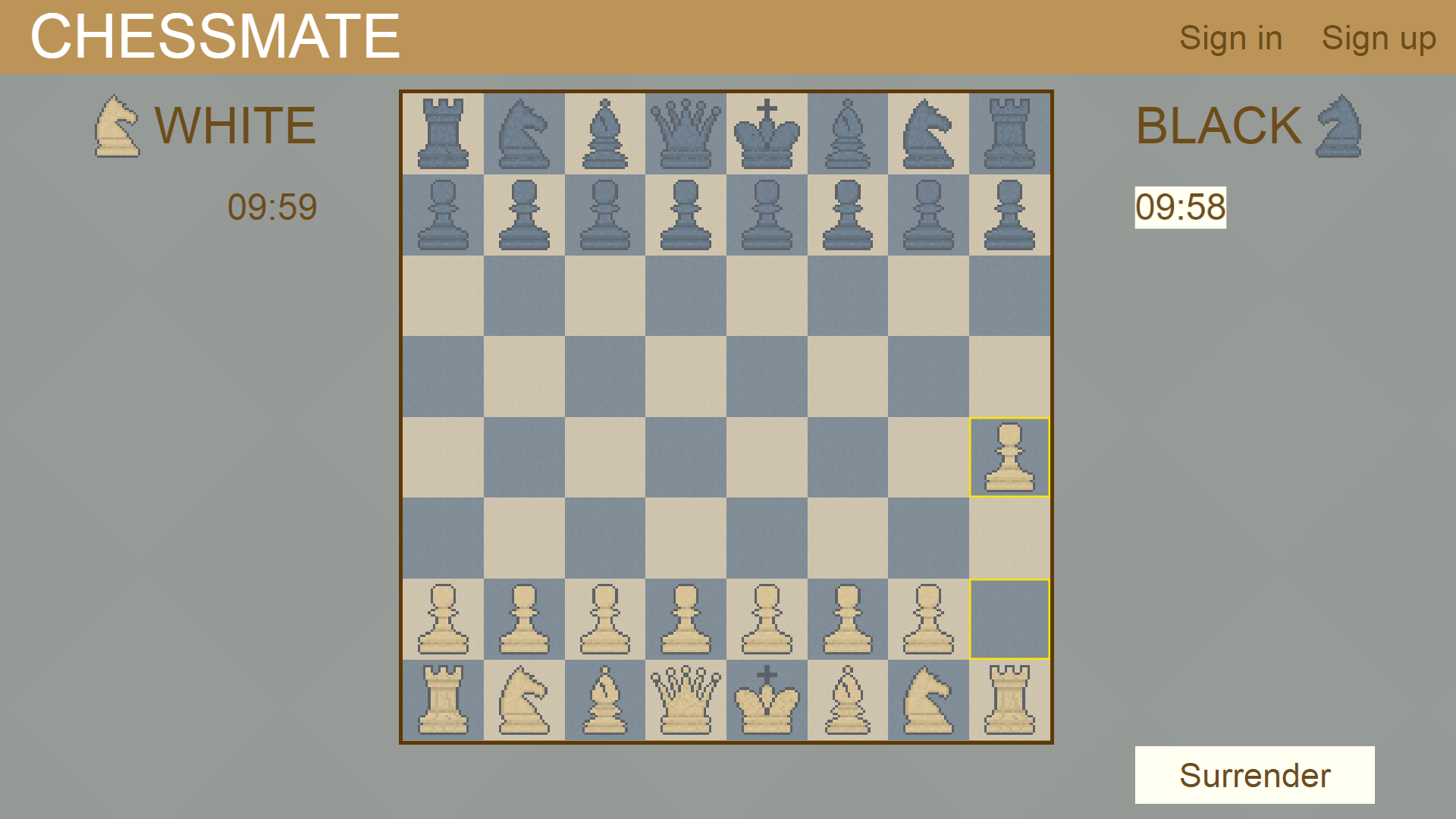


### Страница игры.

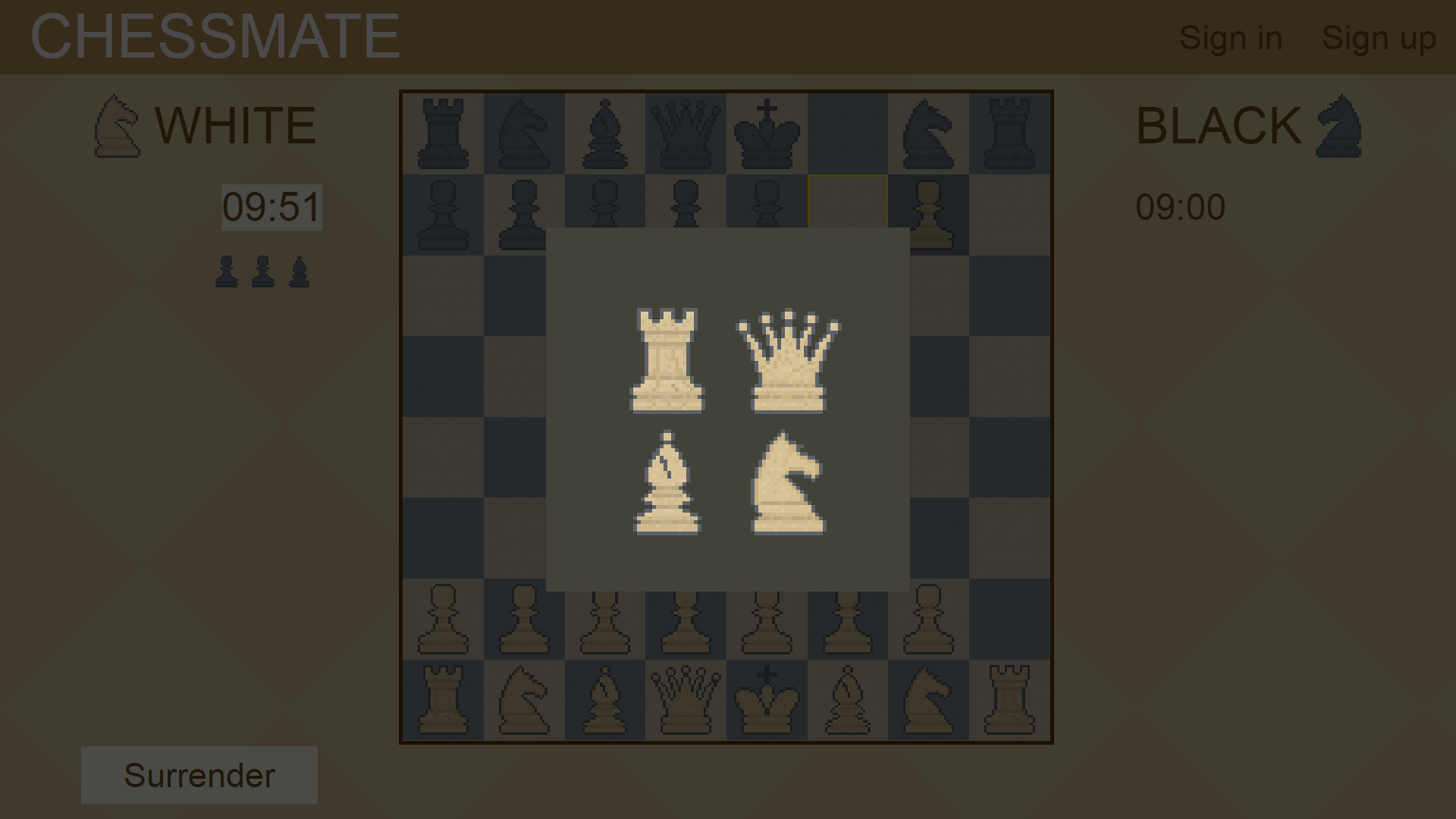
Ход белых.



Ход черных.



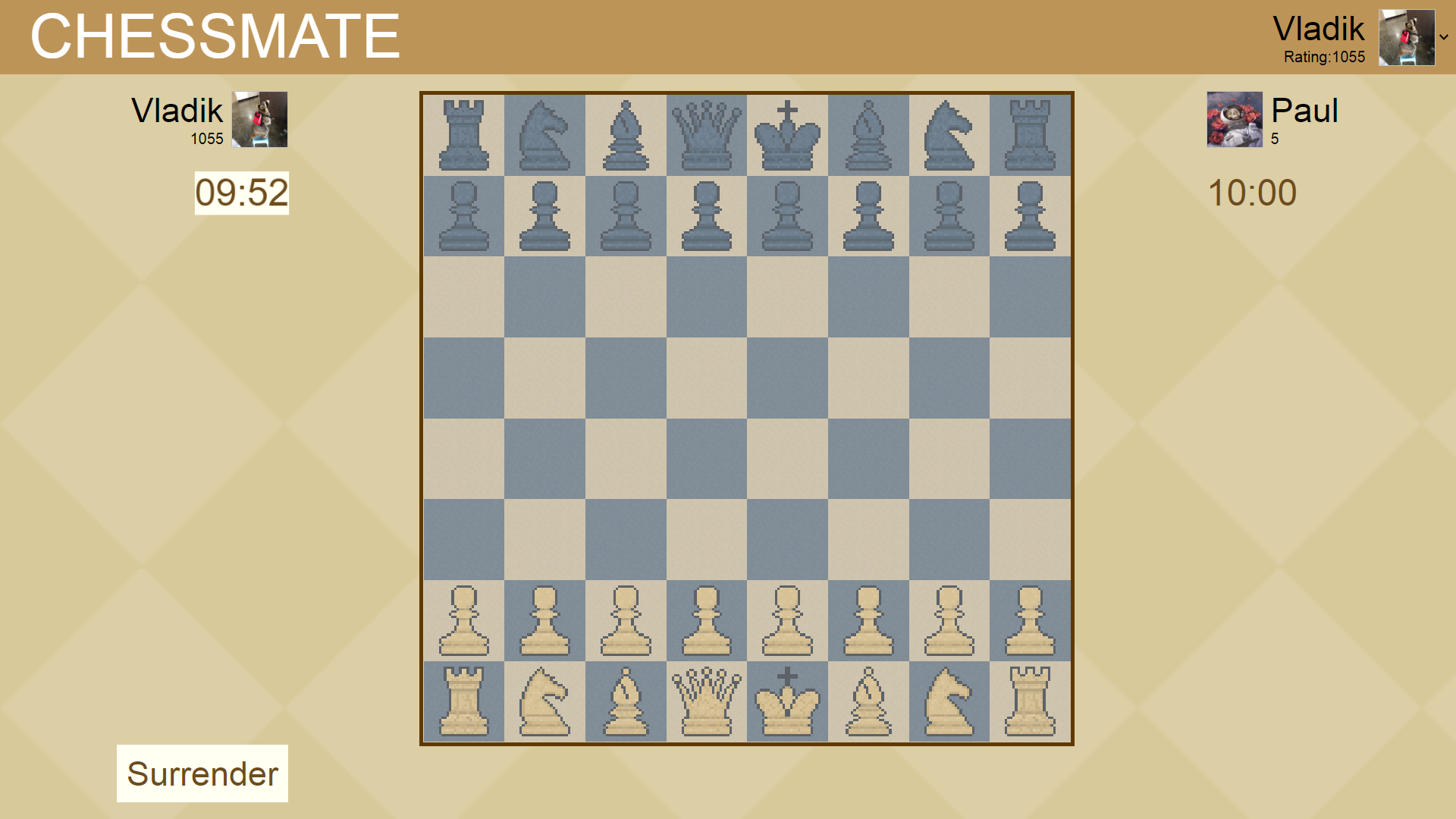
Продвижение.



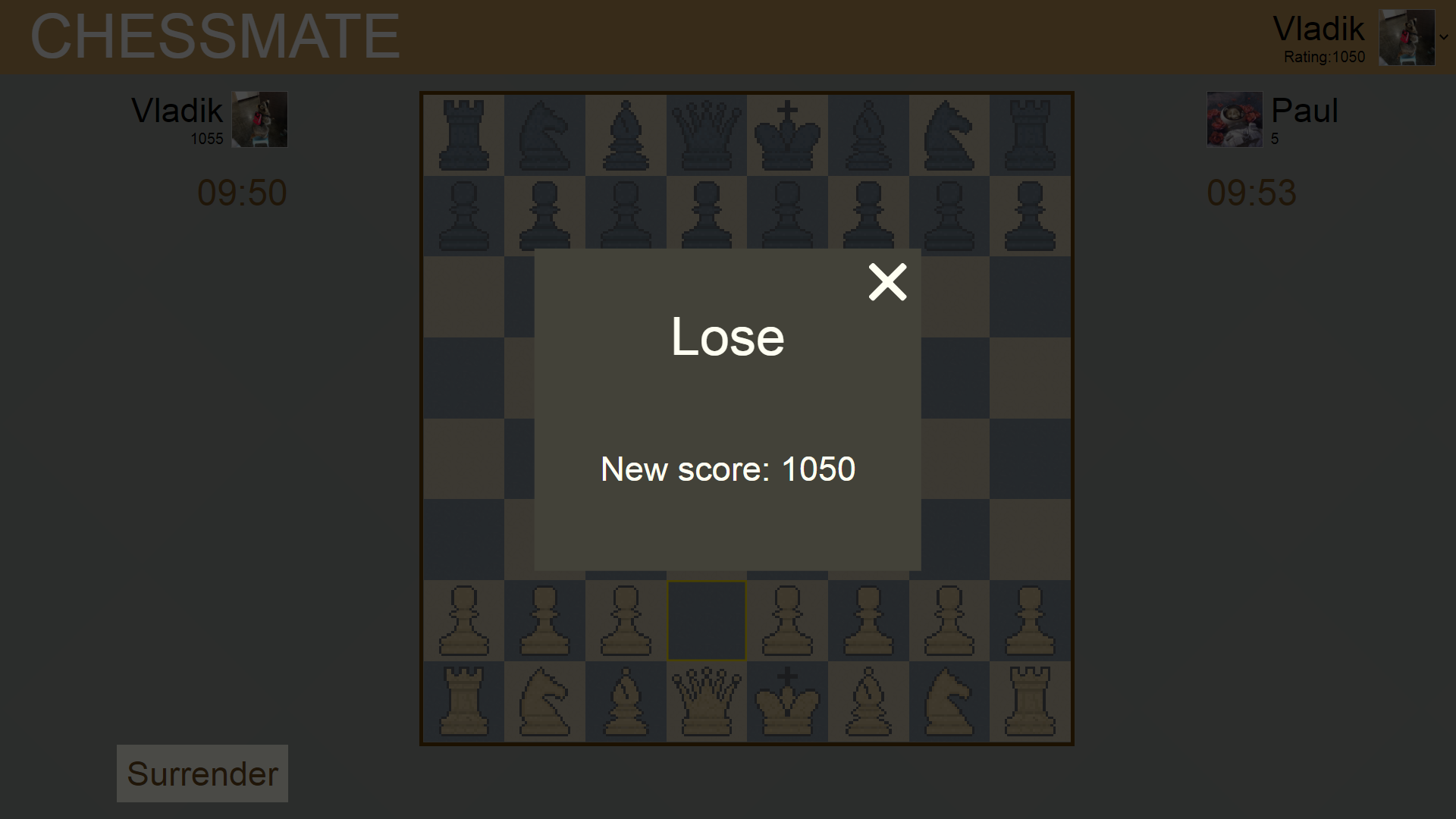
Победа



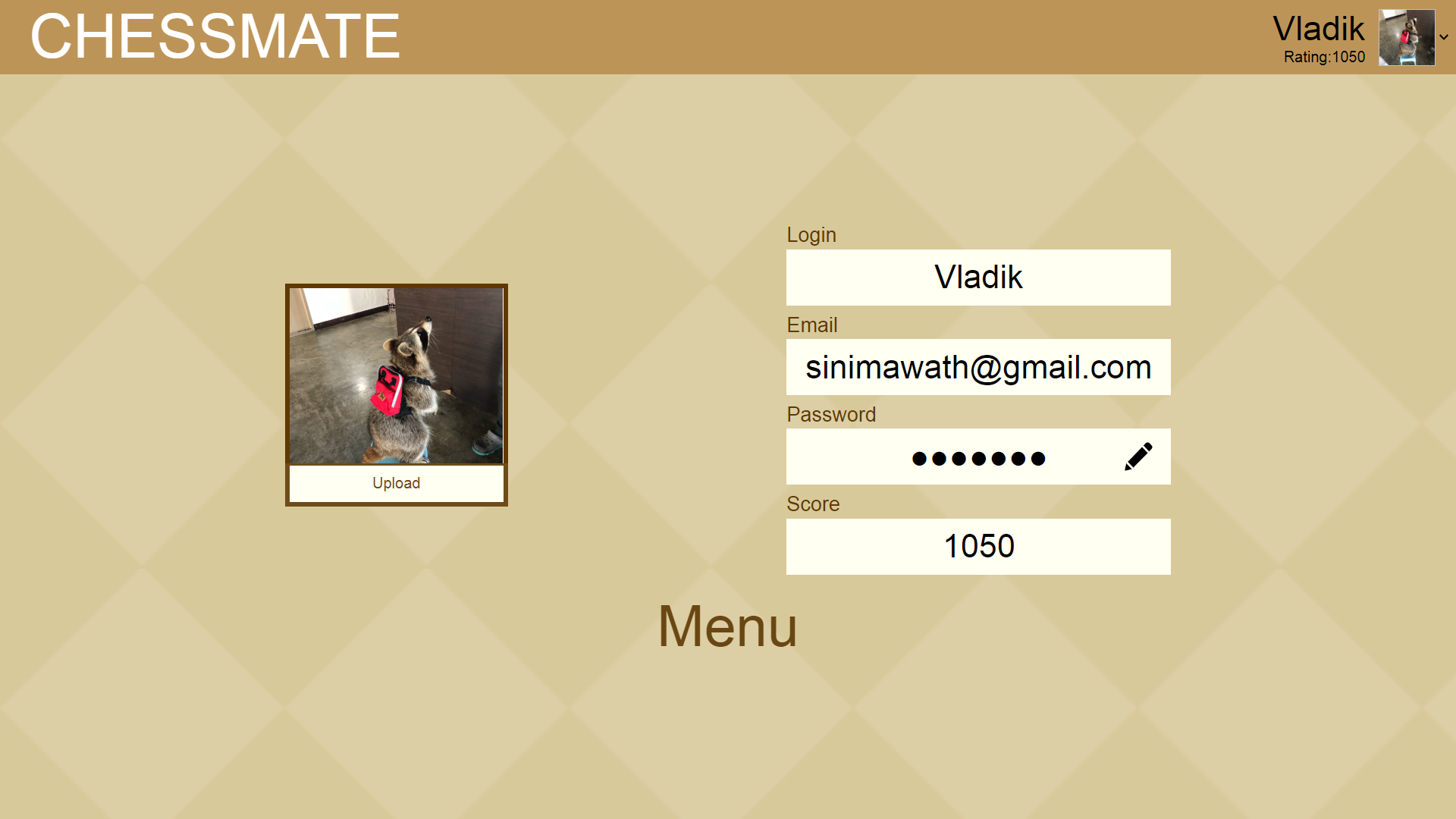
Многопользовательская игра



Поражения



### Профиль



# Описание архитектуры веб-приложения

## Клиентская сторона

### Задачи клиентской стороны

Клиентская сторона – это часть продукта, с которой взаимодействует конечный пользователь продукта.

Основные задачи клиентской стороны это:

* Обеспечивать бесперебойное и ошибочное исполнение приложения, при нормальных условиях
* Предоставления возможности, пользователю пользоваться продуктом. В не зависимости, есть интернет-соединение или нет
* Не должна компрометировать пользовательские данные
* Должна предоставлять понятный и простой интерфейс
* При появлении каких-либо ошибок, должна оповещать пользователя о них
* Даже при отсутствие интернета, клиентская сторона должно предоставлять рабочий функционал, даже если он урезан из-за отсутствия интернета.

### Single Page Application

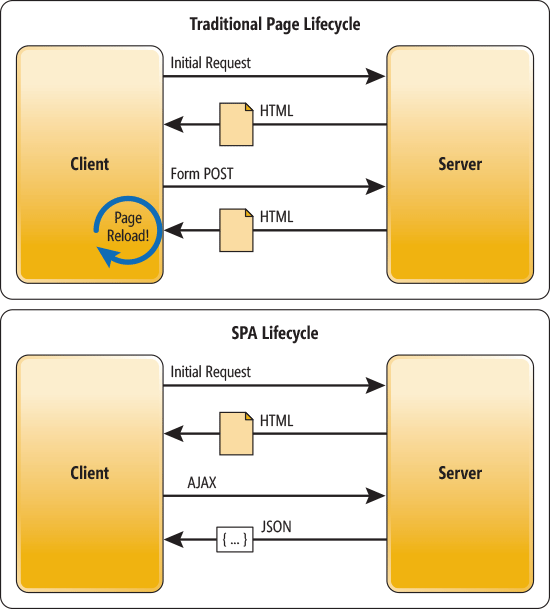
**Single Page Application** - это веб-приложение или веб-сайт, использующий единственный HTML-документ как оболочку для всех веб-страниц и организующий взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript, обычно посредством AJAX.

Веб-приложения «Шахматы» - являются полностью SPA приложением, весь рендеринг HTML кода происходит на клиенте, а данные запрашиваются от API сервера.

Плюсы SPA:

* Уменьшение трафика. Для работы приложения достаточно, одного запроса за статикой
* Работа приложения без доступа к интернету
* Простое создание User-Friendly интерфейса
* Клиентское кэширование с помощью ServiceWorker

Схема работы SPA и обычной загрузки страницы



### Клиентская навигация

SPA приложения должны брать всю навигацию по сайту на себя, так как сервер отдает лишь данные, а не страницы. Для это был реализован класс Router с History Api, для имитации реальных переходов.

Листинг

**export** **default** **class** Router {

constructor (root) {

**this**.root = root;

**this**.routes = **new** Map();

**this**.currentRoute = null;

**this**.isCurrentNotFound = false;

window.addEventListener('popstate', () => {

**const** pathname = Router.\_normalizePath(location.pathname);

**this**.change(pathname, false);

});

}

/\*\*

     \* Переходит на начальную страницу с путем '/'

     \* @param delPrev удаляет из истории Путь из которого сделан переход

     \*/

toStartPage (delPrev = false) {

**if** (delPrev) {

window.history.replaceState(null, null, "/");

}

**this**.change('/', !delPrev);

}

/\*\*

     \* Добавляет маршрут для роутера.

     \* @param path путь при переходе на который будет вызвана view

     \* @param root элемент куда будет рисоваться view, по-умолчание это this.root

     \* @param view компонент, который отрисуется

     \* @param data router data

     \*/

add (path, root = **this**.root, view, data) {

**this**.routes.set(path, {

root,

view,

data

});

}

/\*\*

     \* Устанавливает View компонент, который будет отрисовываться, если не найден запрашиваемый маршрут

     \* @param root элемент куда будет рисоваться view, по-умолчание это this.root

     \* @param view компонент, который отрисуется

     \*/

setNotFoundView (root = **this**.root, view) {

**this**.notFoundView = view;

**this**.notFoundViewRoot = root;

}

/\*\*

     \* Переход на маршрут с путем path

     \* @param path путь

     \* @param addToHistory добавлять Path в History Api или нет.

     \* @param data

     \* @private

     \*/

change (path, addToHistory = true, data = {}) {

**if** (**this**.currentRoute === path) {

**return**;

}

**let** currentData = **this**.routes.get(**this**.currentRoute);

**if** (currentData) {

currentData.view.close();

currentData.view.hide(currentData.root);

}

**if** (**this**.isCurrentNotFound) {

**this**.notFoundView.close();

**this**.notFoundView.hide(**this**.notFoundViewRoot);

}

**if** (addToHistory){

window.history.pushState(null, null, path);

}

**if** (**this**.routes.has(path)) {

**let** route = **this**.routes.get(path);

route.view.render(route.root, {...route.data, ...data});

**this**.currentRoute = path;

} **else** {

**this**.notFoundView.render(**this**.notFoundViewRoot);

**this**.currentRoute = null;

**this**.isCurrentNotFound = true;

}

}

/\*\*

     \* Удаляет суффикс '/', если path != '/'

     \* @param path

     \* @returns {string}

     \* @private

     \*/

**static** \_normalizePath (path) {

**return** path.**charAt**(path.**length** - 1) === '/' && path !== '/' ? path.**slice**(0, path.**length** - 1) : path;

}

/\*\*

     \* запускает роутер

     \*/

start () {

**this**.root.addEventListener('click', (ev) => {

**if** (ev.target.tagName === 'A' && ev.target.hostname === location.hostname) {

ev.preventDefault();

**this**.change(Router.\_normalizePath(ev.target.pathname));

}

});

**this**.change(Router.\_normalizePath(window.location.pathname), false);

}

}

### Компоненты

Приложение разделено на следующие составляющие:

* Lib – клиентские и независимые библиотеки, которые ни как не зависят от проекта. Написаны так, что их можно постоянно переиспользовать
* Css – общие стили всего проекта
* Dist – то что доставляется пользователю в бразуер. Этот компонент собирается специальными сборщиками проекта.
* Models – модели MVC компонентов
* Views – отображения MVC компонентов. Каждая View содержит в себе стили, шаблон и скрипт отображения
* Controllers – контроллеры MVC компонентов. Они связывают View и Model
* Components – отдельные, пере используемые, независимые компоненты отображения.

### Шаблоны проектирования

Для реализации взаимодействия между элементами использовался шаблон EventBus, который состоит из:

* Observable – это объект или функция, которая выдает последовательности данных во времени
* Observer – это объект или функция, которая знает, как обрабатывать последовательности данных
* Subscriber – это объект или функция, которая связывает

Листинг EventBus

**export** **default** **class** EventBus {

/\*\*

     \* @param listOfEvents Array[string] массив ивентов, доступных в этом eventBus

     \*/

constructor (listOfEvents) {

**this**.events = **new** Map();

listOfEvents.forEach((eventName) =>

**this**.events.set(eventName, [])

);

}

/\*\*

     \* Подписка на событие eventBus

     \* @param eventName string имя события

     \* @param callback Function функция, которая выполнится при событии

     \*/

subscribeToEvent (eventName, callback) {

**if** (!**this**.events.has(eventName)) {

**throw** **new** Error(`EventBus: Unknown event ${eventName}`);

}

**this**.events.get(eventName).push(callback);

}

/\*\*

     \* Стриггерить все коллбеки ивенты.

     \* @param eventName string имя события

     \* @param args

     \*/

triggerEvent (eventName, ...args) {

**if** (!**this**.events.has(eventName)) {

**throw** **new** Error(`EventBus: Unknown event ${eventName}`);

}

**let** eventListeners = **this**.events.get(eventName);

eventListeners.forEach((callback) =>

callback(...args)

);

}

}

Для реализации бизнес- логики приложения использовался популярный шаблон проектирования MVC, который состоит из:

* Модель(Model) – под Моделью, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.
* Отображение(View) – в обязанности Отображения входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.
* Контроллер(Controller) – является связующим звеном между Моделью и Отображением. Для это содержит в себе EventBus с определенными событиями

Листинг View класса

**export** **default** **class** View {

constructor (template, eventBus, globalEventBus) {

**this**.el = document.createElement('div');

**this**.el.classList.add('wrapper');

**this**.template = template;

**this**.\_eventBus = eventBus;

**this**.\_globalEventBus = globalEventBus;

**this**.\_prevRoot = null;

**this**.isViewClosed = false;

}

/\*\*

     \* Компилирует шаблон в root элемент. Сохраняет последний root в this.\_prevRoot

     \* @param root если root === null, то рендрится в this.\_prevRoot

     \* @param data

     \* @returns {View}

     \*/

render (root, data = {}) {

**this**.isViewClosed = false;

**if** (root === undefined || root === null) {

root = **this**.\_prevRoot;

} **else** {

**this**.\_prevRoot = root;

}

**this**.el.innerHTML = **this**.template(data);

root.innerHTML = '';

root.appendChild(**this**.el);

**return** **this**;

}

close () {

**this**.isViewClosed = true;

**try** {

**this**.\_eventBus.triggerEvent(VIEW.CLOSE);

} **catch** (e) {

console.**log**('no such event: VIEW.CLOSE');

}

}

hide (root) {

root.innerHTML = '';

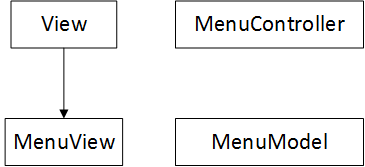
**return** **this**;

}

}

### Диаграмма классов MVC

Из-за их большого количества (больше 50), я приведу лишь малую часть.



Приложение разрабатывалось по паттерну программирования MVC. M – модель, V – отображение, C – контроллер. На примере, приведен MVC компонент Menu.

* View – базовый класс для все отображений. Он рисует, закрывает, обновляет отображения в DOM элемент который ему передали
* …Model – модели компонентов реализуют бизнес-логику
* …View – отображают данные, полученные из модели, удобным для пользователя способов
* …Controller – является контейнером для отображений и моделей, связывает их и создает

Общение между View и Model происходит по EventBus. View вызывает события, Model – подписывается на события. Controller – создает события.

### Web-worker

Так как Javascript однопоточный язык программирования, с асинхронной моделью взаимодействия реализованной через EventLoop, то производить сложные вычисления без замедления всего потока выполнения невозможно.

В проекте используются боты использующие рекурсивные деревья выбора(Easy, Normal) и сверточная нейронная сеть (Hard) – эти алгоритмы дают оптимальное решение от 1 до 12 секунд, в зависимости от платформы запуска: телефон, компьютер, планшет и т.д.. И каждая задержка выбора, вызывает задержку отрисовки DOM бразуера. Для решения это проблемы были выбраны встроенные в браузеры WebWorker-ы, которое позволяют пользоваться настоящей многопоточностью, но не средствами языка, а при помощи браузерного движка.

Данные передаются между worker-ами и главным потоком через систему сообщений — обе стороны передают свои сообщения используя метод postMessage() и отвечают на сообщения при помощи обработчика событий onmessage. Данные при этом копируются, а не делятся.

## Серверная сторона

### Задачи серверной стороны

Серверная сторона – это часть проекта, которая отвечает за хранение, генерацию, обработки данных, используемые клиентской стороной.

Основные задача сервера это:

* Хранение сессий пользователей
* Хранение данных о пользователях
* Бесперебойная работа
* Реализация API для клиентов

### Компоненты

* Controllers – принимают и отвечают на запросы к API
* Auth – компонент авторизации, ходит в базу данных авторизации
* Middleware – функции посредники, изменяют, отвечают, дополняют запросы до их поступления в Controller
* Game – регистрация, создание игровых комнат, распределение игроков по комнатам, получение команд от клиентов.
* Chess – игровой движок шахмат

### Программный интерфейс приложения

* /avatar
  + POST /avatar – загрузка аватара в хранилище
* /session
  + DELETE /session – удаление сессии
  + GET /session – получить сессию игрока
  + POST /session – авторизация
* /user/ – guid – уникальный id пользователя
  + GET /user/{guid} – получить пользователя
  + POST /user – создание пользователя
  + PUT /user/{guid} – обновление данных пользователя
* /user/count
  + GET /user/count – получить количество пользователей
* /user/score
  + GET /user/score – получить счет пользователей
* /game
  + PUT /game – создать новую комнату или найти и вернуть ее id

### Микросервисная архитектура

Микросервисная архитектура - вариант [сервис-ориентированной архитектуры](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0) программного обеспечения, ориентированный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов.

В проекте есть 3 микросервиса

1. Core – главное api приложения. У этого микросервиса есть база данных информации о пользователях. Данные хранят в NoSql БД MongoDB
2. Game – игра, комнаты
3. Auth – авторизация. Хранит сессии пользователей в отдельной MongoDB.

Формат общения между микросервисами gRPC.  
GRPC — это высокопроизводительный фреймворк разработанный компанией Google для вызов удаленных процедур (RPC), работает поверх HTTP/2.

### Nginx

Nginx – асинхронный высокопроизводительный веб-сервер.

Nginx очень хорошо настраиваемый веб-сервер, который выполняет множество важной работы.

Во-первых, nginx принимает запросы по https соединению, но для сервера бизнес-логики распаковывает его в обычный http.

Во-вторых, nginx отдает клиентскую статику: скрипты, картинки, файлы разметки документа, стили и т.д.

В-третьих, nginx автоматически сжимает исходящие http запросы с помощью gzip алгоритма, для уменьшения сетевого трафика.

В-четвертых, nginx является входным узлом, так что внешняя система ничего не знает о микросервисах и обращается ко всему API серверу как к монолиту. Nginx в свою очередь проксирует запросы на нужный микросервис.

В-пятых, nginx управляет политикой кэширования, так что ее можно легко настраивать.

### HTTP/2

Общение между клиентской стороной и серверной происходит по протоколу HTTP2, для обеспечения надежности, скорости.

Рассмотрим HTTP2:

HTTP/2 - вторая крупная версия сетевого протокола HTTP, используемая для доступа к World Wide Web.

Основные нововведения:

* **Мультиплексирование -** позволяет браузеру выполнять множество запросов в рамках одного TCP-соединения
* **Приоритеты - Каждому запросу можно назначить приоритет.**Существует два подхода к назначению приоритетов: на основе веса и на основе зависимостей.
* **Сжатие HTTP-заголовков методом HPACK** - В HTTP/2 заголовки передаются в сжатом виде. Благодаря этому уменьшается количество информации, которой обмениваются между собой сервер и браузер. Вместо алгоритмов gzip/deflate используется HPACK.
* **Обязательное шифрование -** разработчики браузеров приняли решение внедрить новый протокол только для TLS(HTTPS)-соединений.

# ОСНОВНЫЕ ХАРАКТЕРИСТИКИ ПРОГРАММНОГО ПРОДУКТА

Состав набора исходных файлов статики, которые отдаются браузеру, при

загрузке приложения.

|  |  |  |
| --- | --- | --- |
| **Название** | **Размер и тип** | **Описание** |
| Index.html | 10КБ | Основная разметка шаблона SPA |
| Application.js | 1МБ | Содержит все приложение после сборки. |
| Style.css | 30КБ | Содержит все каскандные стили приложения после сборки |
| Sw.js | 40КБ | Содержит код servie worker-а после сборки |

# УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММНОГО ПРОДУКТ

1. Наличие браузера Chrome версии 60 и выше
2. Наличие свободного места на диске 10 мб
3. Наличие интернета при первой загрузке приложения

Так как это веб-приложение, то на любом устройстве, которое удовлетворяет этим условиям, можно запустить приложение.

# ОБЩИЕ ХАРАКТЕРИСТИКИ ПРОГРАММНОГО ПРОДУКТА

Занимаемое место на жестком диске исходными текстами: 200мб

Занимаемое место на жестком диске собрки проекта: 5мб

Общее количество классов: 50

Основной язык программирования: Javascript, стандарт ECMA6

Язык разметки тексты: HTML5

Язык стилизации текста: CSS( с препроцессором LESS )

Платформа сборки проекта: nodejs v.5, npm v.8

Система сборки проекта: webpack v.4

Дополнительные npm-пакеты:

"@babel/core": "^7.1.2",

"@babel/preset-env": "^7.1.5",

"autoprefixer": "^9.3.1",

"babel-loader": "^8.0.4",

"babel-plugin-transform-remove-console": "^6.9.4",

"babel-polyfill": "^6.26.0",

"babel-preset-es2015": "^6.24.1",

"babel-preset-es2016": "^6.24.1",

"body-parser": "^1.18.3",

"cookie-parser": "^1.4.3",

"css-loader": "^1.0.0",

"debug": "^3.1.0",

"eslint": "^5.6.1",

"eslint-config-standard": "^12.0.0",

"eslint-plugin-import": "^2.14.0",

"eslint-plugin-node": "^7.0.1",

"eslint-plugin-promise": "^4.0.1",

"eslint-plugin-standard": "^4.0.0",

"express": "^4.16.3",

"express-http-proxy": "^1.4.0",

"fest": "^0.12.1",

"fest-webpack-loader": "^1.1.1",

"less": "^3.8.1",

"less-loader": "^4.1.0",

"mini-css-extract-plugin": "^0.4.4",

"postcss-loader": "^3.0.0",

"style-loader": "^0.23.0",

"url-loader": "^1.1.2",

"uuid": "^3.3.2",

"webpack": "^4.18.0",

"webpack-cli": "^3.1.0",

"webpack-dev-server": "^3.1.10"

"@tensorflow/tfjs": "^0.14.1",

"normalize.css": "^8.0.1",

"optimize-css-assets-webpack-plugin": "^5.0.1",

"serviceworker-webpack-plugin": "^1.0.1",

"uglifyjs-webpack-plugin": "latest"