

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное  
учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

Лабораторная работа на тему:  
Программная реализация проекта  
«Веб-сервис: игра крестики-нолики»

Выполнил студент:  
Красников Кирилл Денисович  
Группа: С24-702  
Проверил:  
Курасов Юрий Викторович

Оценка: \_\_\_\_\_

Дата: \_\_\_\_\_

Подпись: \_\_\_\_\_

Москва, 2025



К.Д. Красников, С24-702

Тема: Программная реализация проекта «Веб-сервис: игра крестики-нолики»

Инструменты разработки: компилятор языка C (GCC) и универсальный отладчик GNU Debugger (GDB) для ОС Linux Debian 12.

## **1. Задача**

Реализовать веб-сервис, предоставляющий пользователям доступ к сетевой игре «Крестики-нолики». Клиент данной игры следует реализовать на языке JavaScript, используя возможности HTML5. Коммуникация между клиентом и сервером должна осуществляться через протокол HTTP и JSON. Сервер должен поддерживать неограниченное количество клиентских и игровых сессий.

## 2. Теоретическая часть

Для реализации веб-сервиса требуется параллельно обрабатывать несколько TCP соединений. Для этого можно использовать системный вызов `poll` ядра Linux.

Если отправлять от клиента на сервер данные в параметрах GET запроса, значительно упростится реализация сервера, так как, во-первых, нет необходимости в поддержке других методов HTTP, а во-вторых — чтение URL-параметров проще в реализации, чем чтение JSON. Для оптимизации и упрощения очистки памяти при чтении URL-параметров можно реализовать линейный аллокатор (арену).

Для отправки данных от сервера к клиенту следует использовать режим HTTP `chunked transfer encoding`, так как при его использовании не требуется знать заранее размер отправляемого сообщения. В таком случае мы можем использовать потоковую генерацию JSON. Чтобы минимизировать количество системных вызовов, можно использовать буферизацию.

С целью экономии памяти следует очищать сессии, которые были неактивны более пяти минут.

Для вывода на экран игрового поля в клиентской части можно воспользоваться Canvas API из спецификации HTML5.

### 3. Практическая часть

Было принято решение не использовать многопоточность. В структуре `http_server_t` хранятся дескриптор сокета сервера, структуры `pollfd` для всех подключенных в данный момент клиентов и их количество, а также всё, что нужно для обработки текущего соединения (дескриптор сокета текущего клиента, буфер, арена, путь, связный список пар ключ-значение URL-параметров). Память под список URL-параметров выделяется на арене.

Для упрощения отладки проекта был реализован макрос `log_message(LOG_LEVEL_INFO | LOG_LEVEL_ERROR, "module_name", "text", ...)`.

Были реализованы структура `json_writer_t` и набор функций: `init_json_writer`, `json_start_dict`, `json_stop_dict`, `json_start_array` и др., позволяющие пользователю использовать потоковую генерацию JSON. Для буферизации была создана структура `buffered_writer_t` и набор функций: `init_buffered_writer`, `buffered_writer_flush`, `buffered_writer_write` и `buffered_writer_end`. Пример использования `json_writer_t` представлен в приложении А.

Информация о клиентских и игровых сессиях хранится в структуре `session_manager_t` в виде связанных списков. Структура `session_t` содержит сессионный ключ, имя пользователя, ссылку на игровую сессию, время последней активности пользователя, флаг, указывающий, что сессию можно удалить, и ссылку на следующую в списке сессию. В структуре `game_session_t` содержатся ссылки на игроков, вид игры (архитектура приложения предусматривает возможность добавления нескольких игровых режимов), ссылку на состояние игры (`tictactoe_game_t`), флаги, описывающие состояние игровой сессии, и ссылку на следующую в списке игровую сессию. Изменения состояний структуры `game_session_t` показаны на диаграмме в приложении В.

Маршрутизация происходит в функции `main`, обрабатывают запросы функции из файла `http_handlers.c`. Для идентификации пользовательской сессии используется сессионный ключ, который передаётся как URL-параметр `id`. Для получения доступа к ресурсам,

за исключением статических файлов и станицы авторизации, требуется наличие этого параметра. На диаграмме в приложении С показано, как обрабатываются запросы до момента подключения к игровой сессии.

Пример запроса клиента во время игры выглядит следующим образом: `GET /gameRequest?id=123&cell=2`, где `id` — это сессионный ключ, а опциональный параметр `cell` — клетка, на которую делает ход клиент-отправитель. Пример ответа сервера находится в приложении D.

Клиентская часть написана на JavaScript без использования сторонних библиотек. Вывод изображения на экран осуществляется с помощью Canvas API в режиме `CanvasRenderingContext2D`. Скриншоты интерфейса находятся в приложении E.

Весь код проекта находится в приложении F.

#### **4. Содержание**

1. Задача .....	4
2. Теоретическая часть .....	5
3. Практическая часть .....	6
5. Приложения .....	9
5.A. Приложение А .....	9
5.B. Приложение В .....	10
5.C. Приложение С .....	11
5.D. Приложение D .....	13
5.E. Приложение Е .....	13
5.F. Приложение F .....	14

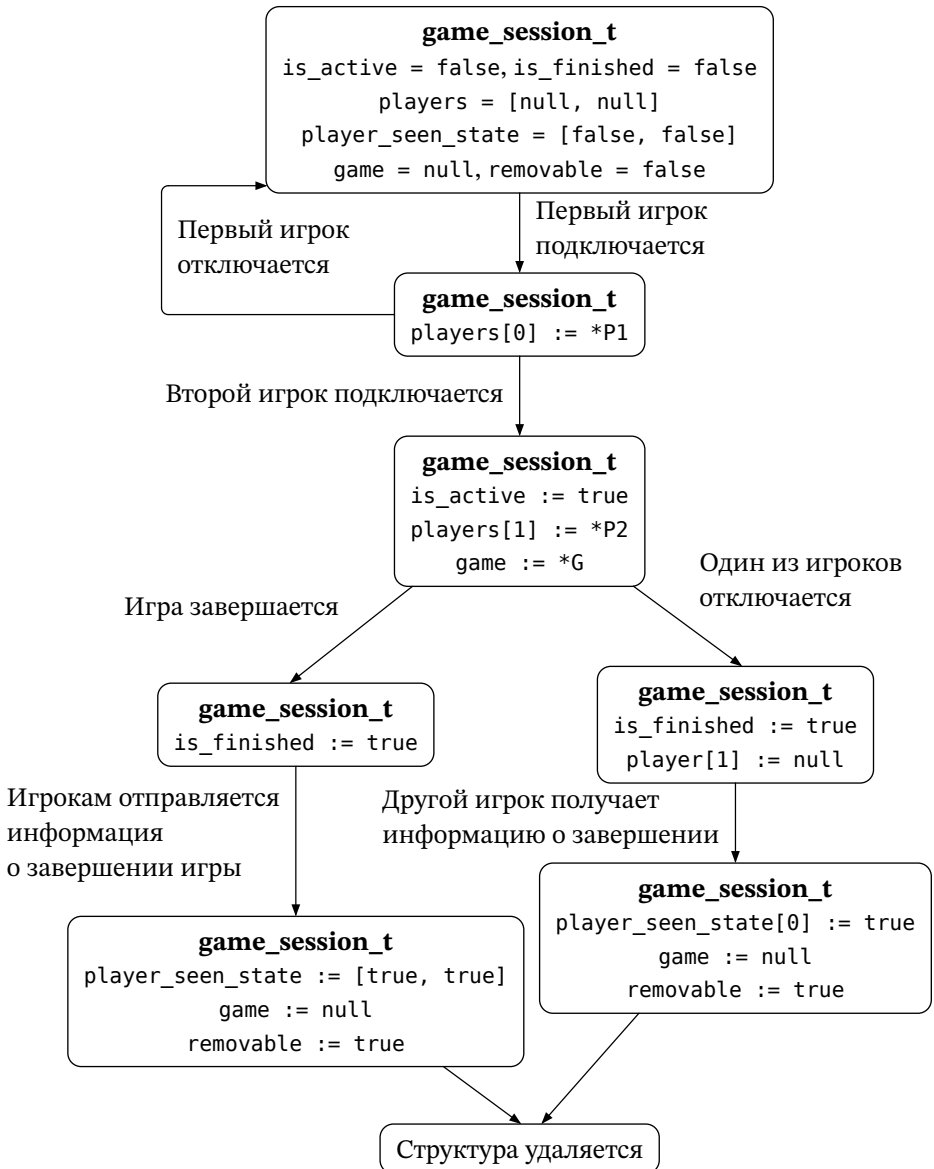


## 5. Приложения

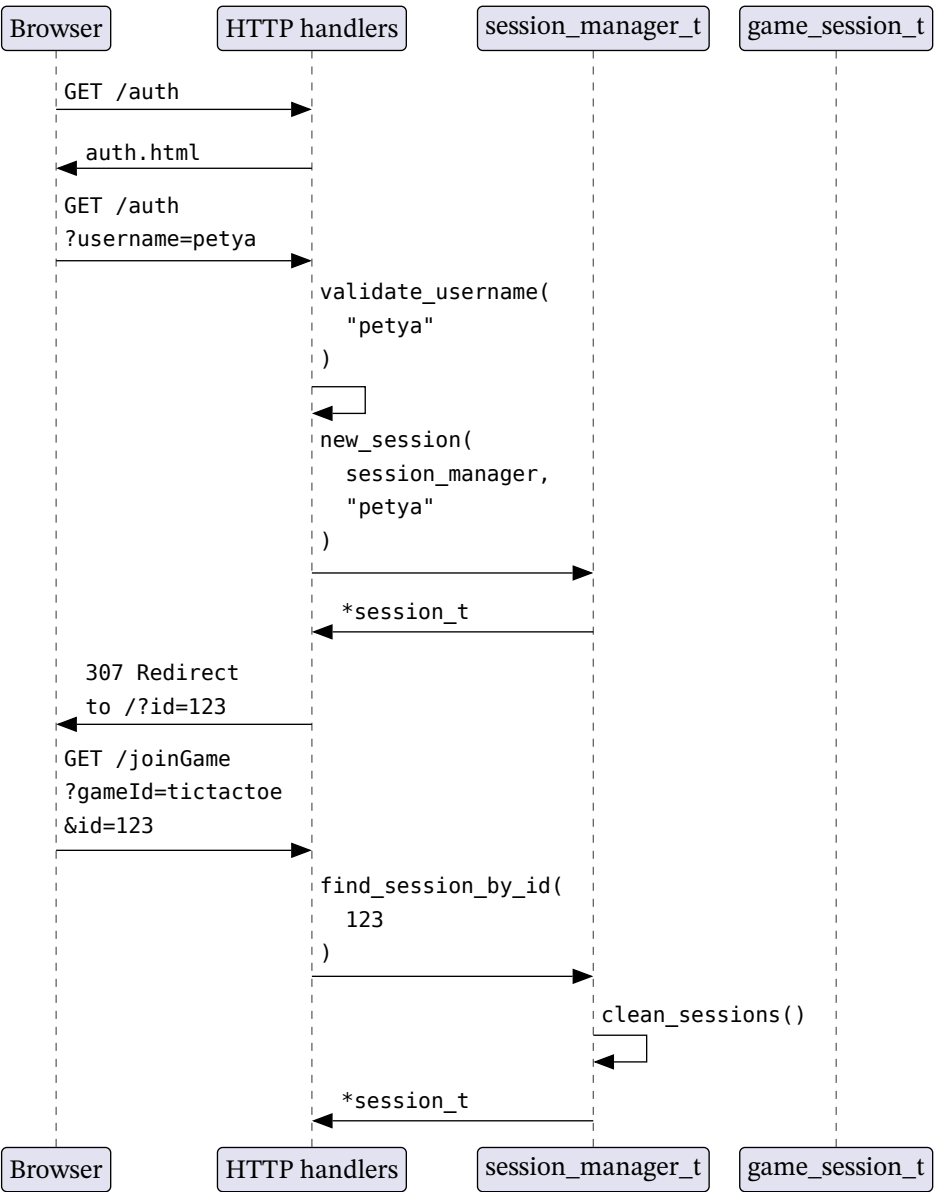
### Приложение А

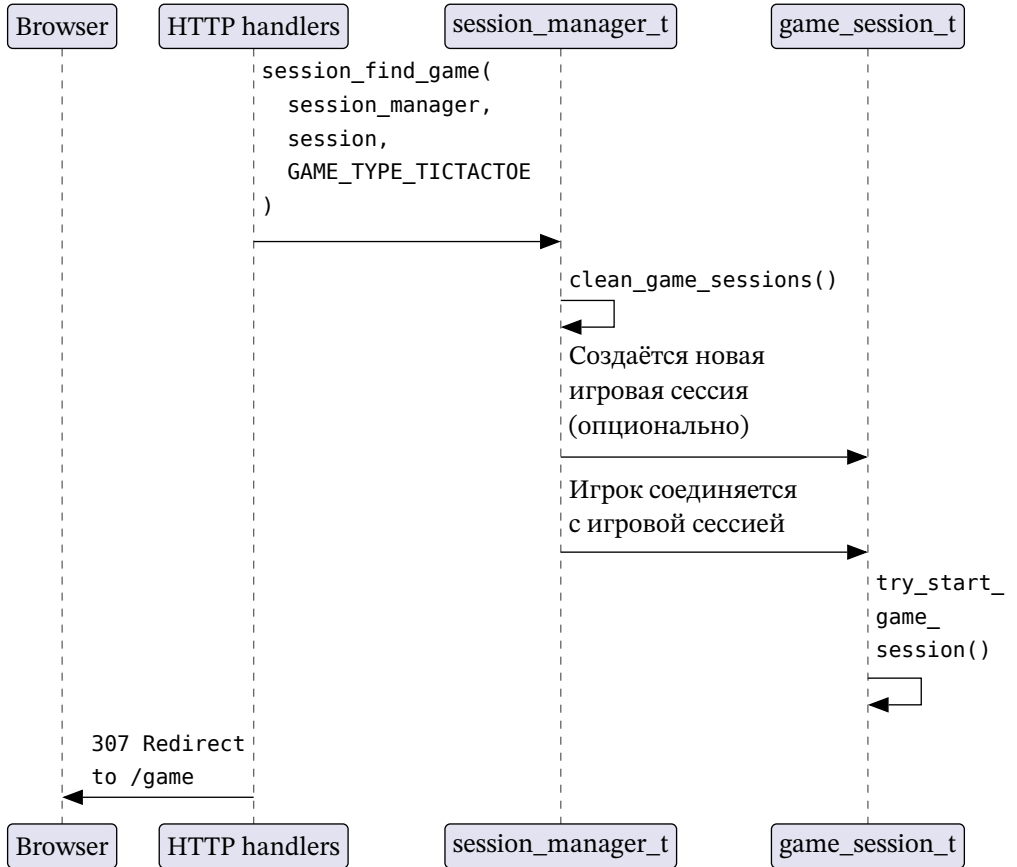
```
void handle_session_list_request(
    session_manager_t *session_manager,
    json_writer_t *json_writer) {
    clean_sessions(session_manager);
    json_start(json_writer);
    json_start_array(json_writer);
    session_t *session = session_manager->sessions;
    while(session != NULL) {
        if (!session->is_active) {
            continue;
        }
        json_start_dict(json_writer);
        json_write_key(json_writer, "id");
        json_write_number(json_writer, session->id);
        json_write_key(json_writer, "name");
        json_write_string(json_writer, session->username);
        json_write_key(json_writer, "player_index");
        json_write_number(json_writer, session->player_index);
        json_write_key(json_writer, "game_session");
        if (session->game_session == NULL) {
            json_write_null(json_writer);
        } else {
            json_start_dict(json_writer);
            json_write_key(json_writer, "is_active");
            json_write_bool(
                json_writer, session->game_session->is_active
            );
            json_write_key(json_writer, "game_type");
            json_write_number(
                json_writer, session->game_session->game_type
            );
            json_stop_dict(json_writer);
        }
        json_stop_dict(json_writer);
        session = session->next_session;
    }
    json_stop_array(json_writer);
    json_end(json_writer);
}
```

## Приложение В



Приложение С





## Приложение D

```
{
  "status": "game_active",
  "your_index": 0.000000,
  "player_names": ["kirill", "fedya"],
  "state": {
    "active_player_index": 1.000000,
    "winner_index": -1.000000,
    "cells": ["X", " ", " ", " ", " ", " ", " ", " ", " ", " "]
  }
}
```

## Приложение E

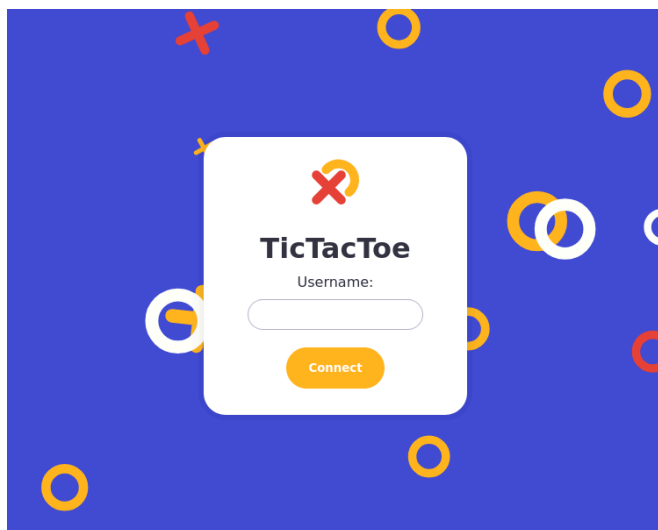


Рис. 1. Страница авторизации

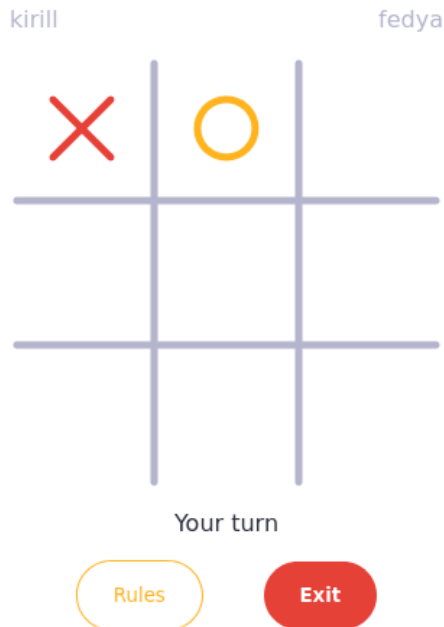


Рис. 2. Страница с игрой

## Приложение F

```
// src/arena.c
#include <stdlib.h>
#include "arena.h"
#include "log.h"

void init_arena(arena_t *arena, int size) {
    arena->size = size;
    arena->position = 0;
    arena->buffer = (char *)malloc(size);
    if (arena->buffer == NULL) {
        log_message(LOG_LEVEL_ERROR, "arena", "Failed allocating %d
bytes for arena", size);
        exit(EXIT_FAILURE);
    }
}
```

```
log_message(LOG_LEVEL_INFO, "arena", "Allocated %d bytes for
arena", size);
}
void deinit_arena(arena_t *arena) {
    free(arena->buffer);
}

char *alloc_arena(arena_t *arena, int size) {
    if (arena->position + size > arena->size) {
        log_message(
            LOG_LEVEL_ERROR,
            "arena",
            "Failed allocating %d bytes on %d bytes arena",
            size,
            arena->size
        );
        return NULL;
    }
    char *res = arena->buffer + arena->position;
    arena->position += size + (8 - size % 8);
    return res;
}

void free_arena(arena_t *arena) {
    arena->position = 0;
}
// src/arena.h
#ifndef ARENA_H
#define ARENA_H

typedef struct {
    int size;
    int position;
    char *buffer;
} arena_t;

void init_arena(arena_t *arena, int size);

void deinit_arena(arena_t *arena);

char *alloc_arena(arena_t *arena, int size);

void free_arena(arena_t *arena);
```

```

#endif
// src/buffered_writer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "buffered_writer.h"
#include "http.h"
#include "log.h"

void init_buffered_writer(buffered_writer_t *buffered_writer,
http_server_t *http_server) {
    buffered_writer->position = 0;
    buffered_writer->http_server = http_server;
    buffered_writer->buffer =
malloc(BUFFERED_WRITER_BUFFER_SIZE);
    if (buffered_writer->buffer == NULL) {
        log_message(
            LOG_LEVEL_ERROR,
            "buffered_writer",
            "Failed allocating %d bytes for buffered_writer",
            BUFFERED_WRITER_BUFFER_SIZE
        );
        exit(EXIT_FAILURE);
    }
    log_message(
        LOG_LEVEL_INFO,
        "buffered_writer",
        "Allocated %d bytes for buffered_writer",
        BUFFERED_WRITER_BUFFER_SIZE
    );
}

void buffered_writer_flush(buffered_writer_t *buffered_writer)
{
    send_http_content(buffered_writer->http_server,
buffered_writer->buffer, buffered_writer->position);
    buffered_writer->position = 0;
}

void buffered_writer_write(buffered_writer_t *buffered_writer,
char *buffer, int size) {
    while (size > 0) {

```



```
    int available_space = BUFFERED_WRITER_BUFFER_SIZE -
buffered_writer->position;
    if (available_space == 0) {
        buffered_writer_flush(buffered_writer);
        available_space = BUFFERED_WRITER_BUFFER_SIZE;
    }
    int segment_size = size < available_space ? size :
available_space;
    memcpy(buffered_writer->buffer + buffered_writer->position,
buffer, segment_size);
    buffered_writer->position += segment_size;
    buffer += segment_size;
    size -= segment_size;
}
}

void buffered_writer_end(buffered_writer_t *buffered_writer) {
    buffered_writer_flush(buffered_writer);
    close_http_connection(buffered_writer->http_server);
}
// src/buffered_writer.h
#ifndef BUFFERED_WRITER_H
#define BUFFERED_WRITER_H

#include "http.h"

#define BUFFERED_WRITER_BUFFER_SIZE 4096

typedef struct {
    http_server_t *http_server;
    char *buffer;
    int position;
} buffered_writer_t;

void init_buffered_writer(buffered_writer_t *buffered_writer,
http_server_t *http_server);

void buffered_writer_flush(buffered_writer_t *buffered_writer);

void buffered_writer_write(buffered_writer_t *buffered_writer,
char *buffer, int size);

void buffered_writer_end(buffered_writer_t *buffered_writer);
```

```

#endif
// src/http.c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "http.h"
#include "arena.h"
#include "log.h"

void init_http_server(http_server_t *http_server, int port) {
    init_arena(&http_server->arena, HTTP_ARENA_SIZE);
    http_server->connected_client_count = 0;
    http_server->pfds = calloc(HTTP_SERVER_MAX_CONNECTED + 1,
sizeof(struct pollfd));
    if (http_server->pfds == NULL) {
        log_message(LOG_LEVEL_ERROR, "http", "Failed allocating
pfds");
        exit(EXIT_FAILURE);
    }
    http_server->buffer = (char
*)malloc(HTTP_REQUEST_BUFFER_SIZE);
    if (http_server->buffer == NULL) {
        log_message(LOG_LEVEL_ERROR, "http", "Failed allocating
request buffer");
        exit(EXIT_FAILURE);
    }
    int server_fd;
    struct sockaddr_in server_addr;
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "socket: %s",
strerror(errno));
        exit(EXIT_FAILURE);
    }
    int opt = 1;

```

```
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt)) < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "setsockopt: %s",
strerror(errno));
        exit(EXIT_FAILURE);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);
    if (bind(server_fd, (struct sockaddr*)&server_addr,
sizeof(server_addr)) < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "bind: %s",
strerror(errno));
        close(server_fd);
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, HTTP_SERVER_BACKLOG) < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "listen: %s",
strerror(errno));
        close(server_fd);
        exit(EXIT_FAILURE);
    }
    socklen_t len = sizeof(server_addr);
    if (getsockname(server_fd, (struct sockaddr *)&server_addr,
&len) < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "getsockname: %s",
strerror(errno));
        close(server_fd);
        exit(EXIT_FAILURE);
    }
    port = ntohs(server_addr.sin_port);
    http_server->server_fd = server_fd;
    http_server->pfd[0].fd = server_fd;
    http_server->pfd[0].events = POLLIN;
    log_message(LOG_LEVEL_INFO, "http", "Server started on port
%d", port);
}

void send_http_head(http_server_t *http_server, char *status) {
    char buffer[HTTP_SERVER_RESPONSE_LINE_SIZE];
    snprintf(buffer, HTTP_SERVER_RESPONSE_LINE_SIZE, "HTTP/1.1
%s\r\n", status);
    write(http_server->client_fd, buffer, strlen(buffer));
}
```

```

}

void send_http_header(http_server_t *http_server, char *key,
char *value) {
    char buffer[HTTP_SERVER_RESPONSE_LINE_SIZE];
    snprintf(buffer, HTTP_SERVER_RESPONSE_LINE_SIZE, "%s:
%s\r\n", key, value);
    write(http_server->client_fd, buffer, strlen(buffer));
}

void send_default_http_headers(http_server_t *http_server) {
    send_http_header(http_server, "Transfer-Encoding",
"chunked");
}

void send_http_end_headers(http_server_t *http_server) {
    write(http_server->client_fd, "\r\n", 2);
}

void send_http_content(http_server_t *http_server, char
*buffer, int size) {
    char size_buffer[64];
    snprintf(size_buffer, sizeof(size_buffer), "%X\r\n", size);
    write(http_server->client_fd, size_buffer,
strlen(size_buffer));
    write(http_server->client_fd, buffer, size);
    write(http_server->client_fd, "\r\n", 2);
}

void close_http_connection(http_server_t *http_server) {
    for (int i = 0; i < http_server->connected_client_count; i++)
    {
        if (http_server->pfds[i + 1].fd == http_server->client_fd)
        {
            log_message(LOG_LEVEL_INFO, "http", "Removing FD %d",
http_server->client_fd);
            http_server->connected_client_count--;
            http_server->pfds[i + 1].fd = http_server-
>pfds[http_server->connected_client_count + 1].fd;
            break;
        }
    }
    write(http_server->client_fd, "0\r\n\r\n", 5);
}

```

```
    close(http_server->client_fd);
}

void send_simple_http_error(http_server_t *http_server, char
*status) {
    char buffer[HTTP_SERVER_RESPONSE_LINE_SIZE];
    snprintf(buffer, HTTP_SERVER_RESPONSE_LINE_SIZE, "<h1>%s</
h1>", status);
    send_http_head(http_server, status);
    send_default_http_headers(http_server);
    send_http_header(http_server, "Content-Type", "text/html");
    send_http_end_headers(http_server);
    send_http_content(http_server, buffer, strlen(buffer));
    close_http_connection(http_server);
}

void send_http_redirect(http_server_t *http_server, char *path)
{
    send_http_head(http_server, HTTP_STATUS_TEMPORARY_REDIRECT);
    send_http_header(http_server, "Location", path);
    send_http_end_headers(http_server);
    close_http_connection(http_server);
}

char read_url_hex(char *buffer) {
    char out = 0;
    for (int i = 1; i < 3; i++) {
        out *= 16;
        switch (buffer[i]) {
            case '0'...'9':
                out += buffer[i] - '0';
                break;
            case 'a'...'f':
                out += buffer[i] - 'a' + 10;
                break;
        }
    }
    return out;
}

char *read_url_segment(http_server_t *http_server, int *index)
{
    char *delimiter = "\r\n?&= ";
```

```

int end;
int length = 0;
for (int i = *index; i < HTTP_REQUEST_BUFFER_SIZE; i++) {
    bool end_found = false;
    for (int j = 0; delimiter[j] != '\0'; j++) {
        if (http_server->buffer[i] == delimiter[j]) {
            end_found = true;
            break;
        }
    }
    if (end_found || (http_server->buffer[i] == '\0')) {
        end = i;
        break;
    }
    if (http_server->buffer[i] == '%') {
        i += 2;
    }
    length++;
}
char *result = alloc_arena(&http_server->arena, length + 1);
if (result == NULL) {
    return NULL;
}
result[length] = '\0';
for (int i = *index, j = 0; j < length; i++, j++) {
    if (http_server->buffer[i] == '+') {
        result[j] = ' ';
    } else if (http_server->buffer[i] == '%') {
        result[j] = read_url_hex(&http_server->buffer[i]);
        i += 2;
    } else {
        result[j] = http_server->buffer[i];
    }
}
// log_message(LOG_LEVEL_INFO, "http", "Got url segment
'%s'", result);
*index = end;
return result;
}

bool parse_query_entry(http_server_t *http_server, int *index)
{
    char *key = read_url_segment(http_server, index);

```

```
    if (key == NULL) {
        return false;
    }
    if (http_server->buffer[*index] != '=') {
        return false;
    }
    (*index)++;
    char *value = read_url_segment(http_server, index);
    if (value == NULL) {
        return false;
    }
    parameter_list_entry_t **tail = &http_server->parameter_list;
    while (*tail != NULL) {
        tail = (parameter_list_entry_t **)&((*tail).next);
    }
    parameter_list_entry_t *new_entry = (parameter_list_entry_t
*)alloc_arena(
    &http_server->arena,
    sizeof(parameter_list_entry_t)
);
    if (new_entry == NULL) {
        return false;
    }
    new_entry->key = key;
    new_entry->value = value;
    new_entry->next = NULL;
    *tail = new_entry;
    log_message(LOG_LEVEL_INFO, "http", "Got parameter '%s' =
'%s'", key, value);
    return true;
}

void parse_request(http_server_t *http_server) {
    free_arena(&http_server->arena);
    http_server->parameter_list = NULL;
    int i = 0;
    char *prefix = "GET ";
    while(prefix[i]) {
        if (http_server->buffer[i] != prefix[i]) {
            log_message(LOG_LEVEL_ERROR, "http", "Request doesn't
start with 'GET '");
            send_simple_http_error(http_server,
HTTP_STATUS_NOT_IMPLEMENTED);
        }
    }
}
```

```

        return;
    }
    i++;
}
http_server->path = read_url_segment(http_server, &i);
log_message(LOG_LEVEL_INFO, "http", "Got path = '%s'",
http_server->path);
if (http_server->buffer[i] == '?') {
    i++;
    if (!parse_query_entry(http_server, &i)) {
        log_message(LOG_LEVEL_ERROR, "http", "Error while reading
request url");
        send_simple_http_error(http_server,
HTTP_STATUS_BAD_REQUEST);
        return;
    }
    while (http_server->buffer[i] == '&') {
        i++;
        if (!parse_query_entry(http_server, &i)) {
            log_message(LOG_LEVEL_ERROR, "http", "Error while
reading request url");
            send_simple_http_error(http_server,
HTTP_STATUS_BAD_REQUEST);
            return;
        }
    }
}
http_server->is_ok = true;
}

void accept_http_request(http_server_t *http_server) {
    http_server->is_ok = false;
    int ready = poll(http_server->pfds, http_server-
>connected_client_count + 1, -1);
    if (ready < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "poll: %s",
strerror(errno));
        exit(EXIT_FAILURE);
    }
    int count = http_server->connected_client_count;
    if (
        (http_server->connected_client_count <
HTTP_SERVER_MAX_CONNECTED)

```



```
&& ((http_server->pfds[0].revents & POLLIN) != 0)
) {
    int client_fd;
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    client_fd = accept(http_server->server_fd, (struct
sockaddr*)&client_addr, &client_len);
    if (client_fd < 0) {
        log_message(LOG_LEVEL_ERROR, "http", "accept: %s",
strerror(errno));
        return;
    }
    char *host = inet_ntoa(client_addr.sin_addr);
    int port = ntohs(client_addr.sin_port);
    log_message(LOG_LEVEL_INFO, "http", "Connected client %s:
%d", host, port);
    http_server->pfds[http_server->connected_client_count +
1].fd = client_fd;
    http_server->pfds[http_server->connected_client_count +
1].events = POLLIN;
    http_server->connected_client_count++;
    log_message(LOG_LEVEL_INFO, "http", "Connected clients
count %d", http_server->connected_client_count);
}
for (int i = 0; i < count; i++) {
    if ((http_server->pfds[i + 1].revents & POLLIN) != 0) {
        int client_fd = http_server->pfds[i + 1].fd;
        int n = read(client_fd, http_server->buffer,
HTTP_REQUEST_BUFFER_SIZE);
        if (n > 0) {
            http_server->buffer[n] = '\0';
        }
        http_server->client_fd = client_fd;
        parse_request(http_server);
        return;
    }
}
}

char *get_http_parameter(http_server_t *http_server, char *key)
{
    parameter_list_entry_t *parameter = http_server-
>parameter_list;
```

```

while (parameter != NULL) {
    if (strcmp(parameter->key, key) == 0) {
        return parameter->value;
    }
    parameter = parameter->next;
}
return NULL;
}
// src/http.h
#ifndef HTTP_H
#define HTTP_H

#include <stdbool.h>
#include <poll.h>
#include "arena.h"

#define HTTP_SERVER_BACKLOG 16
#define HTTP_SERVER_MAX_CONNECTED 16
#define HTTP_ARENA_SIZE 4096
#define HTTP_REQUEST_BUFFER_SIZE 4096
#define HTTP_SERVER_RESPONSE_LINE_SIZE 512
#define HTTP_STATUS_OK "200 OK"
#define HTTP_STATUS_TEMPORARY_REDIRECT "307 Temporary Redirect"
#define HTTP_STATUS_BAD_REQUEST "400 Bad Request"
#define HTTP_STATUS_NOT_FOUND "404 Not Found"
#define HTTP_STATUS_INTERNAL_SERVER_ERROR "500 Internal Server Error"
#define HTTP_STATUS_NOT_IMPLEMENTED "501 Not Implemented"

typedef struct {
    char *key;
    char *value;
    void *next;
} parameter_list_entry_t;

typedef parameter_list_entry_t *parameter_list_t;

typedef struct {
    int server_fd;
    int client_fd;
    bool is_ok;
    char *buffer;
    arena_t arena;

```

```
    char *path;
    parameter_list_t parameter_list;
    int connected_client_count;
    struct pollfd *pfd;
} http_server_t;

void init_http_server(http_server_t *http_server, int port);

void send_http_head(http_server_t *http_server, char *status);

void send_http_header(http_server_t *http_server, char *key,
char *value);

void send_default_http_headers(http_server_t *http_server);

void send_http_end_headers(http_server_t *http_server);

void send_http_content(http_server_t *http_server, char
*buffer, int size);

void send_simple_http_error(http_server_t *http_server, char
*status);

void send_http_redirect(http_server_t *http_server, char
*path);

void close_http_connection(http_server_t *http_server);

void accept_http_request(http_server_t *http_server);

char *get_http_parameter(http_server_t *http_server, char
*key);

#endif
// src/http_handlers.c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include "http_handlers.h"
#include "http.h"
#include "log.h"
#include "json_writer.h"
```

```

#include "sessions.h"
#include "tictactoe.h"

void init_static_handler(static_handler_t *static_handler) {
    DIR *d = opendir(STATIC_FILES_DIRECTORY);
    if (!d) {
        log_message(LOG_LEVEL_ERROR, "http_handlers", "Can't open
%s directory", STATIC_FILES_DIRECTORY);
        exit(EXIT_FAILURE);
    }
    struct dirent *dir;
    FILE *file;
    static_file_entry_t *entry;
    char path[256];
    while ((dir = readdir(d)) != NULL) {
        if (dir->d_type != DT_REG) {
            continue;
        }
        entry = (static_file_entry_t
*)malloc(sizeof(static_file_entry_t));
        if (entry == NULL) {
            log_message(LOG_LEVEL_ERROR, "http_handlers", "Failed
allocating static_file_entry_t");
            exit(EXIT_FAILURE);
        }
        entry->name = strdup(dir->d_name);
        entry->next = NULL;
        if (entry->name == NULL) {
            log_message(LOG_LEVEL_ERROR, "http_handlers", "Failed
allocating static_file_entry_t name");
            exit(EXIT_FAILURE);
        }
        snprintf(path, sizeof(path), "%s/%s",
STATIC_FILES_DIRECTORY, entry->name);
        file = fopen(path, "rb");
        if (file == NULL) {
            log_message(LOG_LEVEL_ERROR, "http_handlers", "Can't open
file %s", path);
            exit(EXIT_FAILURE);
        }
        fseek(file, 0, SEEK_END);
        entry->size = ftell(file);
        fseek(file, 0, SEEK_SET);
    }
}

```

```
entry->data = (char *)malloc(entry->size);
if (entry->data == NULL) {
    log_message(LOG_LEVEL_ERROR, "http_handlers", "Can't
allocate %d bytes for %s", entry->size, entry->name);
    exit(EXIT_FAILURE);
}
fread(entry->data, 1, entry->size, file);
fclose(file);
*static_handler = entry;
static_handler = (static_file_entry_t *)&entry->next;
log_message(LOG_LEVEL_INFO, "http_handlers", "Loaded static
file %s", dir->d_name);
}
closedir(d);
}

bool starts_with(char *string, char *prefix) {
    if (string == NULL || prefix == NULL) {
        return false;
    }
    int len_string = strlen(string);
    int len_prefix = strlen(prefix);
    if (len_prefix > len_string) {
        return false;
    }
    return strncmp(string, prefix, len_prefix) == 0;
}

bool ends_with(char *string, char *suffix) {
    if (string == NULL || suffix == NULL) {
        return false;
    }
    int len_string = strlen(string);
    int len_suffix = strlen(suffix);
    if (len_suffix > len_string) {
        return false;
    }
    return strncmp(string + (len_string - len_suffix), suffix,
len_suffix) == 0;
}

bool match_static_path(http_server_t *http_server) {
    return starts_with(http_server->path, STATIC_PATH_PREFIX);
}
```

```

}

char *get_mime(char *path) {
    if (ends_with(path, ".html")) {
        return "text/html";
    }
    if (ends_with(path, ".js")) {
        return "text/javascript";
    }
    if (ends_with(path, ".css")) {
        return "text/css";
    }
    if (ends_with(path, ".svg")) {
        return "image/svg+xml";
    }
    return "text/plain";
}

void send_static_file(static_handler_t *static_handler,
http_server_t *http_server, char *name) {
    static_file_entry_t *entry = *static_handler;
    while (true) {
        if (entry == NULL) {
            send_simple_http_error(http_server,
HTTP_STATUS_NOT_FOUND);
            return;
        }
        if (strcmp(entry->name, name) == 0) {
            break;
        }
        entry = entry->next;
    }
    send_http_head(http_server, HTTP_STATUS_OK);
    send_default_http_headers(http_server);
    send_http_header(http_server, "Content-Type",
get_mime(name));
    send_http_end_headers(http_server);
    send_http_content(http_server, entry->data, entry->size);
    close_http_connection(http_server);
}

void handle_static_request(static_handler_t *static_handler,
http_server_t *http_server) {

```

```
    char *name = http_server->path + strlen(STATIC_PATH_PREFIX);
    send_static_file(static_handler, http_server, name);
}
```

```
void handle_session_list_request(session_manager_t
*session_manager, json_writer_t *json_writer) {
    clean_sessions(session_manager);
    json_start(json_writer);
    json_start_array(json_writer);
    session_t *session = session_manager->sessions;
    while(session != NULL) {
        if (!session->is_active) {
            continue;
        }
        json_start_dict(json_writer);
        json_write_key(json_writer, "id");
        json_write_number(json_writer, session->id);
        json_write_key(json_writer, "name");
        json_write_string(json_writer, session->username);
        json_write_key(json_writer, "player_index");
        json_write_number(json_writer, session->player_index);
        json_write_key(json_writer, "game_session");
        if (session->game_session == NULL) {
            json_write_null(json_writer);
        } else {
            json_start_dict(json_writer);
            json_write_key(json_writer, "is_active");
            json_write_bool(json_writer, session->game_session-
>is_active);
            json_write_key(json_writer, "game_type");
            json_write_number(json_writer, session->game_session-
>game_type);
            json_stop_dict(json_writer);
        }
        json_stop_dict(json_writer);
        session = session->next_session;
    }
    json_stop_array(json_writer);
    json_end(json_writer);
}
```

```
void handle_auth_request(
    http_server_t *http_server,
```

```

    session_manager_t *session_manager,
    static_handler_t *static_handler
) {
    char *username = get_http_parameter(http_server, "username");
    if (username == NULL) {
        send_static_file(static_handler, http_server, "auth.html");
        return;
    }
    if (!validate_username(username)) {
        send_http_redirect(http_server, "/auth?
msg=Invalid%20username");
        return;
    }
    session_t *session = new_session(session_manager, username);
    if (session == NULL) {
        send_simple_http_error(http_server,
HTTP_STATUS_INTERNAL_SERVER_ERROR);
        return;
    }
    char path[256];
    snprintf(path, sizeof(path), "/?id=%d", session->id);
    send_http_redirect(http_server, path);
    log_message(LOG_LEVEL_INFO, "http_handlers", "Created new
session #%d '%s'", session->id, session->username);
}

session_t *get_session_from_parameter(http_server_t
*http_server, session_manager_t *session_manager) {
    char *session_id_str = get_http_parameter(http_server, "id");
    if (session_id_str == NULL) {
        return NULL;
    }
    int session_id = atoi(session_id_str);
    return find_session_by_id(session_manager, session_id);
}

void handle_join_game(http_server_t *http_server,
session_manager_t *session_manager, session_t *session) {
    char *game_id_str = get_http_parameter(http_server,
"gameId");
    if (game_id_str == NULL) {
        send_simple_http_error(http_server,
HTTP_STATUS_BAD_REQUEST);
    }
}

```



```
        return;
    }
    game_type_t game_type = str_to_game_type(game_id_str);
    if (game_type == GAME_TYPE_EMPTY) {
        send_simple_http_error(http_server,
HTTP_STATUS_BAD_REQUEST);
        return;
    }
    session_find_game(session_manager, session, game_type);
    char path[256];
    snprintf(path, sizeof(path), "/game?id=%d&gameType=%s",
session->id, game_type_to_str(game_type));
    send_http_redirect(http_server, path);
}

void handle_game_request(http_server_t *http_server, session_t
*session, json_writer_t *json_writer) {
    update_session(session);
    char *status;
    if (session->game_session == NULL) {
        log_message(LOG_LEVEL_INFO, "http_handlers", "Trying to
handle_game_request without game_session");
        status = "no_game_session";
    } else if (!session->game_session->is_active) {
        status = "game_not_started";
    } else if (session->game_session->is_finished) {
        status = "game_finished";
    } else {
        status = "game_active";
    }
    json_start(json_writer);
    json_start_dict(json_writer);
    json_write_key(json_writer, "status");
    json_write_string(json_writer, status);
    json_write_key(json_writer, "your_index");
    json_write_number(json_writer, session->player_index);
    json_write_key(json_writer, "player_names");
    json_start_array(json_writer);
    if (session->game_session != NULL) {
        for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
            session_t *isession = session->game_session->players[i];
            if (isession == NULL) {
                json_write_null(json_writer);
            }
        }
    }
}
```

```

    } else {
        json_write_string(json_writer, isession->username);
    }
}
}
json_stop_array(json_writer);
if (session->game_session != NULL && session->game_session-
>game != NULL) {
    json_write_key(json_writer, "state");
    switch (session->game_session->game_type) {
        case GAME_TYPE_EMPTY:
            json_write_null(json_writer);
            break;
        case GAME_TYPE_TICTACTOE:
            tictactoe_handle_request(session->game_session,
session, http_server);
            tictactoe_write_json(session->game_session,
json_writer);
            break;
    }
    if (session->game_session->is_finished) {
        end_game_session(session->game_session);
    }
}
json_stop_dict(json_writer);
json_end(json_writer);
if (session->game_session != NULL) {
    session->game_session->player_seen_state[session-
>player_index] = true;
}
}

void handle_disconnect_from_game_session(http_server_t
*http_server, session_t *session) {
    disconnect_from_game_session(session);
    char path[256];
    snprintf(path, sizeof(path), "/?id=%d", session->id);
    send_http_redirect(http_server, path);
}
// src/http_handlers.h
#ifndef HTTP_HANDLERS_H
#define HTTP_HANDLERS_H

```

```
#include <stdbool.h>
#include "http.h"
#include "json_writer.h"
#include "sessions.h"

#define STATIC_PATH_PREFIX "/static/"
#define STATIC_FILES_DIRECTORY "./static/"

typedef struct {
    char *name;
    int size;
    char *data;
    void *next;
} static_file_entry_t;

typedef static_file_entry_t *static_handler_t;

void init_static_handler(static_handler_t *static_handler);

bool starts_with(char *string, char *prefix);

bool ends_with(char *string, char *suffix);

bool match_static_path(http_server_t *http_server);

char *get_mime(char *path);

void send_static_file(static_handler_t *static_handler,
http_server_t *http_server, char *name);

void handle_static_request(static_handler_t *static_handler,
http_server_t *http_server);

void handle_session_list_request(session_manager_t
*session_manager, json_writer_t *json_writer);

void handle_auth_request(
    http_server_t *http_server,
    session_manager_t *session_manager,
    static_handler_t *static_handler
);

session_t *get_session_from_parameter(http_server_t
```

```

*http_server, session_manager_t *session_manager);

void handle_join_game(http_server_t *http_server,
session_manager_t *session_manager, session_t *session);

void handle_game_request(http_server_t *http_server, session_t
*session, json_writer_t *json_writer);

void handle_disconnect_from_game_session(http_server_t
*http_server, session_t *session);

#endif
// src/json_writer.c
#include <stdio.h>
#include <string.h>
#include "json_writer.h"
#include "buffered_writer.h"

void init_json_writer(
    json_writer_t *json_writer, http_server_t *http_server,
    buffered_writer_t *buffered_writer
) {
    json_writer->http_server = http_server;
    json_writer->buffered_writer = buffered_writer;
}

void json_start_dict(json_writer_t *json_writer) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",
1);
    }
    json_writer->is_first = true;
    buffered_writer_write(json_writer->buffered_writer, "{", 1);
}

void json_stop_dict(json_writer_t *json_writer) {
    buffered_writer_write(json_writer->buffered_writer, "}", 1);
    json_writer->is_first = false;
}

void json_start_array(json_writer_t *json_writer) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",

```

```
1);
}
json_writer->is_first = true;
buffered_writer_write(json_writer->buffered_writer, "[", 1);
}

void json_stop_array(json_writer_t *json_writer) {
    json_writer->is_first = false;
    buffered_writer_write(json_writer->buffered_writer, "]", 1);
}

void json_write_number(json_writer_t *json_writer, double
number) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",
1);
    }
    json_writer->is_first = false;
    char buffer[256];
    snprintf(buffer, sizeof(buffer), "%f", number);
    buffered_writer_write(json_writer->buffered_writer, buffer,
strlen(buffer));
}

void json_write_bool(json_writer_t *json_writer, bool value) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",
1);
    }
    json_writer->is_first = false;
    char *buffer = value ? "true" : "false";
    buffered_writer_write(json_writer->buffered_writer, buffer,
strlen(buffer));
}

bool is_printable(char c) {
    switch (c) {
        case '\0':
        case '\r':
        case '\n':
        case '"':
        case '\t':
            return false;
    }
```

```

    }
    return true;
}

char *escape(char c) {
    switch (c) {
        case '\r':
            return "\\r";
        case '\n':
            return "\\n";
        case '"':
            return "\\\"";
        case '\t':
            return "\\t";
    }
    return NULL;
}

void json_write_string(json_writer_t *json_writer, char
*string) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",
1);
    }
    buffered_writer_write(json_writer->buffered_writer, "\"", 1);
    json_writer->is_first = false;
    int start = 0, end = 0;
    while (string[start] != '\\0') {
        while (is_printable(string[end])) {
            end++;
        }
        buffered_writer_write(json_writer->buffered_writer, string
+ start, end - start);
        char *escaped = escape(string[end]);
        if (escaped == NULL) {
            break;
        }
        buffered_writer_write(json_writer->buffered_writer,
escaped, 2);
        start = ++end;
    }
    buffered_writer_write(json_writer->buffered_writer, "\"", 1);
}

```

```
void json_write_key(json_writer_t *json_writer, char *key) {
    json_write_string(json_writer, key);
    json_writer->is_first = true;
    buffered_writer_write(json_writer->buffered_writer, ":", 1);
}

void json_write_null(json_writer_t *json_writer) {
    if (!json_writer->is_first) {
        buffered_writer_write(json_writer->buffered_writer, ",",
1);
    }
    json_writer->is_first = false;
    char *buffer = "null";
    buffered_writer_write(json_writer->buffered_writer, buffer,
strlen(buffer));
}

void json_start(json_writer_t *json_writer) {
    json_writer->is_first = true;
    send_http_head(json_writer->http_server, HTTP_STATUS_OK);
    send_default_http_headers(json_writer->http_server);
    send_http_header(json_writer->http_server, "Content-Type",
"application/json");
    send_http_end_headers(json_writer->http_server);
}

void json_end(json_writer_t *json_writer) {
    json_writer->is_first = true;
    buffered_writer_end(json_writer->buffered_writer);
}
// src/json_writer.h
#ifndef JSON_WRITER_H
#define JSON_WRITER_H

#include <stdbool.h>
#include "http.h"
#include "buffered_writer.h"

typedef struct {
    http_server_t *http_server;
    buffered_writer_t *buffered_writer;
    bool is_first;
}
```

```

} json_writer_t;

void init_json_writer(
    json_writer_t *json_writer, http_server_t *http_server,
    buffered_writer_t *buffered_writer
);

void json_start_dict(json_writer_t *json_writer);

void json_stop_dict(json_writer_t *json_writer);

void json_start_array(json_writer_t *json_writer);

void json_stop_array(json_writer_t *json_writer);

void json_write_number(json_writer_t *json_writer, double
number);

void json_write_bool(json_writer_t *json_writer, bool value);

void json_write_string(json_writer_t *json_writer, char
*string);

void json_write_key(json_writer_t *json_writer, char *key);

void json_write_null(json_writer_t *json_writer);

void json_start(json_writer_t *json_writer);

void json_end(json_writer_t *json_writer);

#endif
// src/log.c
#include <stdio.h>
#include <time.h>
#include "log.h"

void log_header(log_level_t level, char *module) {
    time_t rawtime;
    struct tm * timeinfo;
    char timestamp_buffer[80];
    time(&rawtime);
    timeinfo = localtime(&rawtime);

```



```
    strftime(timestamp_buffer, sizeof(timestamp_buffer), "%Y-%m-%d %H:%M:%S", timeinfo);
    char *level_str;
    switch (level) {
        case LOG_LEVEL_INFO:
            level_str = "\033[34mINFO";
            break;
        case LOG_LEVEL_ERROR:
            level_str = "\033[31mERROR";
            break;
    }
    fprintf(stderr, "\033[90m[%s] %s\033[90m %s: \033[0m",
timestamp_buffer, level_str, module);
}
// src/log.h
#ifndef LOG_H
#define LOG_H

#include <stdio.h>

typedef enum {
    LOG_LEVEL_INFO,
    LOG_LEVEL_ERROR,
} log_level_t;

void log_header(log_level_t level, char *module);

#define log_message(level, module, fmt, ...) if (1) { \
    log_header(level, module); fprintf(stderr, fmt "\n", \
    ##__VA_ARGS__); \
}

#endif
// src/main.c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <stdbool.h>
#include "log.h"
#include "http.h"
#include "buffered_writer.h"
```

```

#include "json_writer.h"
#include "http_handlers.h"
#include "sessions.h"

#define DEFAULT_HTTP_SERVER_PORT 8080

void print_usage(char *name) {
    printf(
        "Usage: %s [port]\n"
        "\n"
        "Options:\n"
        "  -h, --help\tPrint this message\n",
        name
    );
}

int main(int argc, char **argv) {
    signal(SIGPIPE, SIG_IGN);
    srand(time(NULL));
    http_server_t http_server;
    buffered_writer_t buffered_writer;
    json_writer_t json_writer;
    static_handler_t static_handler;
    session_manager_t session_manager;
    int port;
    if (argc == 1) {
        port = DEFAULT_HTTP_SERVER_PORT;
    } else if (argc == 2) {
        if (strcmp(argv[1], "--help") == 0 || strcmp(argv[1], "-h")
== 0) {
            print_usage(argv[0]);
            exit(EXIT_SUCCESS);
        }
        port = atoi(argv[1]);
    } else {
        print_usage(argv[0]);
        exit(EXIT_FAILURE);
    }
    log_message(LOG_LEVEL_INFO, "main", "Starting...");
    init_http_server(&http_server, port);
    init_buffered_writer(&buffered_writer, &http_server);
    init_json_writer(&json_writer, &http_server,
&buffered_writer);

```

```
init_static_handler(&static_handler);
init_session_manager(&session_manager);
while (true) {
    accept_http_request(&http_server);
    if (!http_server.is_ok) {
        continue;
    }
    if (match_static_path(&http_server)) {
        handle_static_request(&static_handler, &http_server);
        continue;
    }
    if (strcmp(http_server.path, "/sessions") == 0) {
        handle_session_list_request(&session_manager,
&json_writer);
        continue;
    }
    if (strcmp(http_server.path, "/auth") == 0) {
        handle_auth_request(&http_server, &session_manager,
&static_handler);
        continue;
    }
    session_t *session =
get_session_from_parameter(&http_server, &session_manager);
    if (session == NULL) {
        send_http_redirect(&http_server, "/auth");
        continue;
    }
    if (strcmp(http_server.path, "/") == 0) {
        send_static_file(&static_handler, &http_server,
"index.html");
        continue;
    }
    if (strcmp(http_server.path, "/joinGame") == 0) {
        handle_join_game(&http_server, &session_manager,
session);
        continue;
    }
    if (strcmp(http_server.path, "/game") == 0) {
        send_static_file(&static_handler, &http_server,
"game.html");
        continue;
    }
    if (strcmp(http_server.path, "/gameRequest") == 0) {
```

```

        handle_game_request(&http_server, session, &json_writer);
        continue;
    }
    if (strcmp(http_server.path, "/disconnect") == 0) {
        handle_disconnect_from_game_session(&http_server,
session);
        continue;
    }
    send_simple_http_error(&http_server,
HTTP_STATUS_NOT_FOUND);
}
return 0;
}
// src/sessions.c
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "sessions.h"
#include "log.h"
#include "tictactoe.h"

char *game_type_to_str(game_type_t game_type) {
    switch (game_type) {
        case GAME_TYPE_EMPTY:
            return NULL;
        case GAME_TYPE_TICTACTOE:
            return "tictactoe";
    }
    return NULL;
}

game_type_t str_to_game_type(char *string) {
    if (strcmp(string, "tictactoe") == 0) {
        return GAME_TYPE_TICTACTOE;
    }
    return GAME_TYPE_EMPTY;
}

void reset_session(session_t *session) {
    disconnect_from_game_session(session);
    session->is_active = false;
}

```

```
void init_session_manager(session_manager_t *session_manager) {
    session_manager->game_sessions = NULL;
    session_manager->sessions = NULL;
}

bool validate_username(char *username) {
    if (username == NULL) {
        log_message(LOG_LEVEL_ERROR, "sessions", "Expected username
got NULL");
        return false;
    }
    if (strlen(username) == 0) {
        log_message(LOG_LEVEL_ERROR, "sessions", "Empty username");
        return false;
    }
    if (strlen(username) > USERNAME_MAX_LENGTH) {
        log_message(LOG_LEVEL_ERROR, "sessions", "Username is too
long");
        return false;
    }
    char *c = username;
    while (*c != '\0') {
        switch (*c) {
            case 'A'...'Z':
            case 'a'...'z':
            case '0'...'9':
            case '_':
                c++;
                continue;
            default:
                log_message(LOG_LEVEL_ERROR, "sessions", "Invalid
character in username");
                return false;
        }
    }
    return true;
}

void update_session(session_t *session) {
    session->last_active = time(NULL);
}
```

```

session_t *new_session(session_manager_t *session_manager, char
*username) {
    if (!validate_username(username)) {
        return NULL;
    }
    session_t *session = malloc(sizeof(session_t));
    if (session == NULL) {
        log_message(LOG_LEVEL_ERROR, "sessions", "Can't allocate
new session");
        exit(1);
    }
    session->next_session = session_manager->sessions;
    session_manager->sessions = session;
    session->is_active = true;
    session->id = rand();
    session->game_session = NULL;
    update_session(session);
    strcpy(session->username, username);
    log_message(LOG_LEVEL_INFO, "sessions", "New session #%d
'%s'", session->id, session->username);
    return session;
}

session_t *find_session_by_id(session_manager_t
*session_manager, int id) {
    clean_sessions(session_manager);
    session_t *session = session_manager->sessions;
    while (session != NULL) {
        if (!session->is_active) {
            session = session->next_session;
            continue;
        }
        if (session->id == id) {
            return session;
        }
        session = session->next_session;
    }
    return NULL;
}

void clean_game_sessions(session_manager_t *session_manager) {
    game_session_t **last_ref = &session_manager->game_sessions;
    game_session_t *session = session_manager->game_sessions;

```

```
while (session != NULL) {
    if (!session->removable) {
        *last_ref = session;
        last_ref = (game_session_t **>(&session-
>next_game_session);
        session = session->next_game_session;
    } else {
        game_session_t *next_session = session-
>next_game_session;
        free(session);
        session = next_session;
    }
}
*last_ref = NULL;
}

void session_find_game(session_manager_t *session_manager,
session_t *session, game_type_t game_type) {
    clean_game_sessions(session_manager);
    update_session(session);
    disconnect_from_game_session(session);
    game_session_t *game_session = session_manager-
>game_sessions;
    while (game_session != NULL) {
        if (
            game_session->is_active ||
            (game_session->game_type != game_type && game_session-
>game_type != GAME_TYPE_EMPTY)
        ) {
            game_session = game_session->next_game_session;
            continue;
        }
        for (int j = 0; j < MAX_PLAYERS_PER_GAME; j++) {
            if (game_session->players[j] == NULL) {
                session->game_session = game_session;
                session->player_index = j;
                game_session->players[j] = session;
                game_session->game_type = game_type;
                try_start_game_session(game_session);
                return;
            }
        }
        game_session = game_session->next_game_session;
    }
}
```

```

    }
    game_session = malloc(sizeof(game_session_t));
    if (game_session == NULL) {
        log_message(LOG_LEVEL_ERROR, "sessions", "Can't allocate
new game session");
        exit(1);
    }
    session->game_session = game_session;
    session->player_index = 0;
    game_session->is_active = false;
    game_session->is_finished = false;
    for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
        game_session->player_seen_state[i] = false;
        game_session->players[i] = NULL;
    }
    game_session->players[0] = session;
    game_session->game_type = game_type;
    game_session->game = NULL;
    game_session->removable = false;
    game_session->next_game_session = session_manager-
>game_sessions;
    session_manager->game_sessions = game_session;
}

void clean_sessions(session_manager_t *session_manager) {
    session_t *session = session_manager->sessions;
    time_t now = time(NULL);
    while (session != NULL) {
        if (!session->is_active) {
            session = session->next_session;
            continue;
        }
        if (now - session->last_active > MAX_SESSION_INACTIVE_TIME)
    {
        log_message(LOG_LEVEL_INFO, "sessions", "Cleaning session
#%d", session->id);
        reset_session(session);
    }
    session = session->next_session;
}
session_t **last_ref = &session_manager->sessions;
session = session_manager->sessions;
while (session != NULL) {

```



```
    if (session->is_active) {
        *last_ref = session;
        last_ref = (session_t **>(&session->next_session));
        session = session->next_session;
    } else {
        session_t *next_session = session->next_session;
        free(session);
        session = next_session;
    }
}
*last_ref = NULL;
}

void disconnect_from_game_session(session_t *session) {
    if (session->game_session == NULL) {
        return;
    }
    session->game_session->players[session->player_index] = NULL;
    switch (session->game_session->game_type) {
        case GAME_TYPE_EMPTY:
            break;
        case GAME_TYPE_TICTACTOE:
            tictactoe_handle_disconnect(session->game_session,
session);
            break;
    }
    end_game_session(session->game_session);
    session->game_session = NULL;
}

void end_game_session(game_session_t *game_session) {
    if (!game_session->is_active) {
        return;
    }
    if (!game_session->is_finished) {
        for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
            game_session->player_seen_state[i] = false;
        }
    }
    game_session->is_finished = true;
    for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
        if (game_session->players[i] == NULL) {
            continue;
        }
    }
}
```

```

    }
    if (!game_session->player_seen_state[i]) {
        return;
    }
}
for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
    session_t *session = game_session->players[i];
    if (session == NULL) {
        continue;
    }
    session->game_session = NULL;
}
if (game_session->game != NULL) {
    switch (game_session->game_type) {
        case GAME_TYPE_EMPTY:
            break;
        case GAME_TYPE_TICTACTOE:
            tictactoe_deinit(game_session);
            break;
    }
}
game_session->removable = true;
}

void try_start_game_session(game_session_t *game_session) {
    for (int i = 0; i < MAX_PLAYERS_PER_GAME; i++) {
        if (game_session->players[i] == NULL) {
            return;
        }
    }
    game_session->is_active = true;
    switch (game_session->game_type) {
        case GAME_TYPE_EMPTY:
            log_message(LOG_LEVEL_ERROR, "sessions", "Attempting to
start GAME_TYPE_EMPTY");
            break;
        case GAME_TYPE_TICTACTOE:
            tictactoe_init(game_session);
            break;
    }
}
}
// src/sessions.h
#endif SESSIONS_H

```

```
#define SESSIONS_H

#include <time.h>
#include <stdbool.h>

#define USERNAME_MAX_LENGTH 256
#define MAX_PLAYERS_PER_GAME 2
#define MAX_SESSION_INACTIVE_TIME 300

typedef enum {
    GAME_TYPE_EMPTY,
    GAME_TYPE_TICTACTOE,
} game_type_t;

typedef struct {
    bool is_active;
    bool is_finished;
    void *players[MAX_PLAYERS_PER_GAME];
    bool player_seen_state[MAX_PLAYERS_PER_GAME];
    game_type_t game_type;
    void *game;
    bool removable;
    void *next_game_session;
} game_session_t;

typedef struct {
    bool is_active;
    int id;
    char username[USERNAME_MAX_LENGTH + 1];
    game_session_t *game_session;
    int player_index;
    time_t last_active;
    void *next_session;
} session_t;

typedef struct {
    game_session_t *game_sessions;
    session_t *sessions;
} session_manager_t;

char *game_type_to_str(game_type_t game_type);

game_type_t str_to_game_type(char *string);
```

```

void init_session_manager(session_manager_t *session_manager);

bool validate_username(char *username);

void update_session(session_t *session);

session_t *new_session(session_manager_t *session_manager, char
*username);

session_t *find_session_by_id(session_manager_t
*session_manager, int id);

void session_find_game(session_manager_t *session_manager,
session_t *session, game_type_t game_type);

void clean_sessions(session_manager_t *session_manager);

void disconnect_from_game_session(session_t *session);

void end_game_session(game_session_t *game_session);

void try_start_game_session(game_session_t *game_session);

#endif
// src/tictactoe.c
#include <stdio.h>
#include <stdlib.h>
#include "tictactoe.h"
#include "log.h"
#include "http.h"

void tictactoe_init(game_session_t *game_session) {
    log_message(LOG_LEVEL_INFO, "tictactoe", "Creating new
game");
    tictactoe_game_t *game = malloc(sizeof(tictactoe_game_t));
    if (game == NULL) {
        log_message(LOG_LEVEL_ERROR, "tictactoe", "Failed
allocating tictactoe_game_t");
        return;
    }
    game->active_player_index = 0;
    game->winner = -1;
}

```

```
    for (int i = 0; i < 9; i++) {
        game->cells[i] = TICTACTOE_CELL_EMPTY;
    }
    game_session->game = game;
}

void tictactoe_deinit(game_session_t *game_session) {
    log_message(LOG_LEVEL_INFO, "tictactoe", "Deinitializing
game");
    if (game_session->game == NULL) {
        return;
    }
    free(game_session->game);
}

void tictactoe_write_json(game_session_t *game_session,
json_writer_t *json_writer) {
    tictactoe_game_t *game = game_session->game;
    json_start_dict(json_writer);
    json_write_key(json_writer, "active_player_index");
    json_write_number(json_writer, game->active_player_index);
    json_write_key(json_writer, "winner_index");
    json_write_number(json_writer, game->winner);
    json_write_key(json_writer, "cells");
    json_start_array(json_writer);
    for (int i = 0; i < 9; i++) {
        char *value;
        switch (game->cells[i]) {
            case TICTACTOE_CELL_EMPTY:
                value = " ";
                break;
            case TICTACTOE_CELL_X:
                value = "X";
                break;
            case TICTACTOE_CELL_O:
                value = "O";
                break;
        }
        json_write_string(json_writer, value);
    }
    json_stop_array(json_writer);
    json_stop_dict(json_writer);
}
```

```

tictactoe_cell_t tictactoe_check_line(tictactoe_game_t *game,
int start, int step) {
    tictactoe_cell_t first_cell = game->cells[start];
    if (first_cell == TICTACTOE_CELL_EMPTY) {
        return TICTACTOE_CELL_EMPTY;
    }
    for (int i = 1; i < 3; i++) {
        tictactoe_cell_t cell = game->cells[start + i * step];
        if (cell != first_cell) {
            return TICTACTOE_CELL_EMPTY;
        }
    }
    return first_cell;
}

tictactoe_cell_t tictactoe_check_all_lines(tictactoe_game_t
*game) {
    tictactoe_cell_t result = tictactoe_check_line(game, 0, 4);
    if (result != TICTACTOE_CELL_EMPTY) {
        return result;
    }
    result = tictactoe_check_line(game, 2, 2);
    if (result != TICTACTOE_CELL_EMPTY) {
        return result;
    }
    for (int i = 0; i < 3; i++) {
        result = tictactoe_check_line(game, i * 3, 1);
        if (result != TICTACTOE_CELL_EMPTY) {
            return result;
        }
        result = tictactoe_check_line(game, i, 3);
        if (result != TICTACTOE_CELL_EMPTY) {
            return result;
        }
    }
    return TICTACTOE_CELL_EMPTY;
}

void tictactoe_try_finish(game_session_t *game_session) {
    tictactoe_game_t *game = game_session->game;
    tictactoe_cell_t winner_cell =
tictactoe_check_all_lines(game);

```

```
    if (winner_cell == TICTACTOE_CELL_X) {
        game->winner = 0;
    } else if (winner_cell == TICTACTOE_CELL_0) {
        game->winner = 1;
    }
    if (winner_cell != TICTACTOE_CELL_EMPTY) {
        end_game_session(game_session);
        return;
    }
    for (int i = 0; i < 9; i++) {
        if (game->cells[i] == TICTACTOE_CELL_EMPTY) {
            return;
        }
    }
    end_game_session(game_session);
}

void tictactoe_handle_request(game_session_t *game_session,
session_t *session, http_server_t *http_server) {
    char *cell_str = get_http_parameter(http_server, "cell");
    if (cell_str == NULL) {
        return;
    }
    int cell = atoi(cell_str);
    if (cell < 0 || cell >= 9) {
        log_message(LOG_LEVEL_INFO, "tictactoe", "Invalid move
cell=%d", cell);
    }
    tictactoe_game_t *game = game_session->game;
    if (game->active_player_index != session->player_index) {
        return;
    }
    if (game->cells[cell] != TICTACTOE_CELL_EMPTY) {
        return;
    }
    game->cells[cell] = session->player_index == 1 ?
TICTACTOE_CELL_0 : TICTACTOE_CELL_X;
    game->active_player_index = (game->active_player_index + 1) %
2;
    tictactoe_try_finish(game_session);
}

void tictactoe_handle_disconnect(game_session_t *game_session,
```

```

session_t *session) {
    if (game_session == NULL || game_session->game == NULL ||
    session == NULL) {
        return;
    }
    tictactoe_game_t *game = game_session->game;
    game->winner = 1 - session->player_index;
}
// src/tictactoe.h
#ifndef TICTACTOE_H
#define TICTACTOE_H

#include "sessions.h"
#include "json_writer.h"

typedef enum {
    TICTACTOE_CELL_EMPTY,
    TICTACTOE_CELL_X,
    TICTACTOE_CELL_O,
} tictactoe_cell_t;

typedef struct {
    int active_player_index;
    int winner;
    tictactoe_cell_t cells[9];
} tictactoe_game_t;

void tictactoe_init(game_session_t *game_session);

void tictactoe_deinit(game_session_t *game_session);

void tictactoe_write_json(game_session_t *game_session,
    json_writer_t *json_writer);

void tictactoe_handle_request(game_session_t *game_session,
    session_t *session, http_server_t *http_server);

void tictactoe_handle_disconnect(game_session_t *game_session,
    session_t *session);

#endif

```



```
// static/auth-background.js
const canvas = document.getElementById("background-canvas");
const ctx = canvas.getContext("2d");

const CONFIG = {
  particleCount: 50,
  minSize: 8,
  maxSize: 32,
  minSpeed: 0.01,
  maxSpeed: 0.15,
  colors: [
    "#E94138", "#FFB520", "#FFFFFFB"
  ],
  lineWidth: 4,
  background: "#424CD4",
};

function rand(min, max) {
  return Math.random() * (max - min) + min;
}
function randInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
class Particle {
  constructor(width, height) {
    this.reset(width, height, true);
  }
  reset(width, height, init = false) {
    this.type = Math.random() < 0.5 ? "cross" : "circle";
    this.radius = rand(CONFIG.minSize, CONFIG.maxSize);
    this.x = rand(0, width);
    this.y = rand(0, height);
    const speed = rand(CONFIG.minSpeed, CONFIG.maxSpeed);
    const angle = rand(0, Math.PI * 2);
    this.vx = Math.cos(angle) * speed;
    this.vy = Math.sin(angle) * speed;
    this.color = CONFIG.colors[randInt(0, CONFIG.colors.length
- 1)];
    this.rotation = rand(0, Math.PI * 2);
    this.rotationSpeed = rand(-0.003, 0.003);
  }
  update(width, height) {
    this.x += this.vx;
```

```

    this.y += this.vy;
    this.rotation += this.rotationSpeed;
    if (this.x < -2 * this.radius) {
        this.x = width + 2 * this.radius;
    }
    if (this.x > width + 2 * this.radius) {
        this.x = -2 * this.radius;
    }
    if (this.y < -2 * this.radius) {
        this.y = height + 2 * this.radius;
    }
    if (this.y > height + 2 * this.radius) {
        this.y = -2 * this.radius;
    }
}
draw(ctx) {
    ctx.save();
    ctx.lineWidth = Math.max(1, CONFIG.lineWidth *
(this.radius / 8));
    if (this.type === "circle") {
        ctx.beginPath();
        ctx.strokeStyle = this.color;
        ctx.arc(this.x, this.y, this.radius, 0, Math.PI * 2);
        ctx.stroke();
    } else {
        ctx.translate(this.x, this.y);
        ctx.rotate(this.rotation);
        ctx.lineCap = "round";
        ctx.strokeStyle = this.color;
        const r = this.radius;
        ctx.beginPath();
        ctx.moveTo(-r, 0);
        ctx.lineTo(r, 0);
        ctx.moveTo(0, -r);
        ctx.lineTo(0, r);
        ctx.stroke();
    }
    ctx.restore();
}
}
class ParticleSystem {
    constructor(canvas, ctx, count) {
        this.canvas = canvas;

```

```
    this.ctx = ctx;
    this.width = window.innerWidth;
    this.height = window.innerHeight;
    this.particles = [];
    this.count = count;
    this._createParticles();
  }
  _createParticles() {
    this.particles.length = 0;
    for (let i = 0; i < this.count; i++) {
      this.particles.push(new Particle(this.width,
this.height));
    }
  }
  resize(width, height) {
    this.width = width;
    this.height = height;
    for (const p of this.particles) {
      p.x = Math.min(Math.max(p.x, -p.radius), width +
p.radius);
      p.y = Math.min(Math.max(p.y, -p.radius), height +
p.radius);
    }
  }
  updateAndDraw() {
    this.ctx.fillStyle = CONFIG.background;
    this.ctx.fillRect(0, 0, this.width, this.height);
    for (const p of this.particles) {
      p.update(this.width, this.height);
      p.draw(this.ctx);
    }
  }
}
const system = new ParticleSystem(canvas, ctx,
CONFIG.particleCount);
function animate() {
  system.updateAndDraw();
  requestAnimationFrame(animate);
}
function setCanvasSizeToDisplaySize() {
  const w = window.innerWidth;
  const h = window.innerHeight;
  if (canvas.width !== w || canvas.height !== h) {
```

```

        canvas.width = w;
        canvas.height = h;
        system.resize(w, h);
    }
}
window.addEventListener("resize", setCanvasSizeToDisplaySize);
setCanvasSizeToDisplaySize();
animate();
// static/game.js
const COLORS = {
    cross: "#E94138",
    circle: "#FFB520",
    outline: "#B8B8D1",
};
const ANIMATION_STEP = 0.003;
const canv = document.getElementById("canv");
const ctx = canv.getContext("2d");
const statusDiv = document.getElementById("status");
const modalExit = document.getElementById("modal-exit");
const exitButton = document.getElementById("exit-button");
const splashElem = document.getElementById("splash");
const playersElem = document.getElementById("players");
const spinner = document.getElementById("spinner");
const params = new URLSearchParams(window.location.search);
const id = params.get("id");
const gameType = params.get("gameType");
let state = { active_player_index: 0, cells:
Array.from({ length: 9 }).map((_) => " ") };

function showThisGuide() {
    showModalGuide("/static/" + gameType + "-guide.html");
}

async function disconnectFollowLink(url) {
    if (url == undefined) {
        url = "/"
    }
    modalBg.classList.remove("hidden");
    modalExit.classList.remove("hidden");
    setTimeout(
        () => {
            modalBg.classList.remove("modal-hidden");
            modalExit.classList.remove("modal-hidden");

```

```
    },
    20
  );
  exitButton.onclick = () => {
    fetch("/disconnect?id=" + id).then(() =>
      window.open(url + "?id=" + id, "_self")
    )
  };
}

function easingFunction(x) {
  if (x > 1) {
    return 1;
  }
  if (x < 0) {
    return 0;
  }
  return (1 - Math.cos(Math.PI * x)) / 2;
}

class Sprite {
  constructor(type) {
    this.type = type;
    this.progress = 0;
  }

  update(deltaTime) {
    this.progress += ANIMATION_STEP * deltaTime;
  }
}

class Drawer {
  constructor(canv, ctx, statusDiv, splashElem, playersElem,
    spinner) {
    this.canv = canv;
    this.ctx = ctx;
    this.statusDiv = statusDiv;
    this.splashElem = splashElem;
    this.playersElem = playersElem;
    this.spinner = spinner;
    this.playerIndex = -1;
    this.cells = Array.from({ length: 9 }).map((_) => " ");
    this.sprites = Array.from({ length: 9 }).map((_) => null);
  }
}
```

```

    this.lastTime = null;
    this.finished = false;
}

splash(text) {
    this.splashElem.innerText = text;
    this.splashElem.classList.remove("hidden");
    const keyframes = [
        { transform: 'scale(0)' },
        { transform: 'scale(1)' },
    ];
    const options = {
        duration: 300,
        easing: "cubic-bezier(0.68, -0.55, 0.265, 1.55)",
        iterations: 1,
    };
    this.splashElem.animate(keyframes, options);
}

update(response) {
    if (this.finished) {
        return;
    }
    this.playersElem.innerHTML = response.player_names.map((x)
=> {
        if (x == null) {
            return "<div class='unknown'>???

```

```

        this.splash("You win!");
    } else if (state.winner_index == 1 - this.playerIndex) {
        this.splash("You lost!");
    } else {
        this.splash("Draw!");
    }
}

const cells = state.cells;
for (let i = 0; i < 9; i++) {
    if (cells[i] != this.cells[i]) {
        this.sprites[i] = new Sprite(cells[i]);
        this.cells[i] = cells[i];
    }
}
}

drawGrid(n) {
    const cellWidth = this.canv.width / n;
    this.ctx.strokeStyle = COLORS.outline;
    this.ctx.lineCap = "round";
    this.ctx.lineWidth = 5.0;
    for (let i = 1; i < n; i++) {
        this.ctx.beginPath();
        this.ctx.moveTo(i * cellWidth, 5);
        this.ctx.lineTo(i * cellWidth, n * cellWidth - 5);
        this.ctx.stroke();
        this.ctx.beginPath();
        this.ctx.moveTo(5, i * cellWidth);
        this.ctx.lineTo(n * cellWidth - 5, i * cellWidth);
        this.ctx.stroke();
    }
}

drawSprite(sprite, x, y, size) {
    if (sprite == null) {
        return;
    }
    size *= 0.2;
    this.ctx.save();
    this.ctx.translate(x, y);
    this.ctx.lineCap = "round";
    this.ctx.lineWidth = 5.0;
    if (sprite.type == "X") {

```

```

const p1 = easingFunction(sprite.progress / 0.8);
if (p1 > 0) {
  this.ctx.strokeStyle = COLORS.cross;
  this.ctx.beginPath();
  this.ctx.moveTo(-size, -size);
  this.ctx.lineTo(-size + 2 * p1 * size, -size + 2 * p1 *
size);
  this.ctx.stroke();
}
const p2 = easingFunction((sprite.progress - 0.2) / 0.8);
if (p2 > 0) {
  this.ctx.beginPath();
  this.ctx.moveTo(size, -size);
  this.ctx.lineTo(size - 2 * p2 * size, -size + 2 * p2 *
size);
  this.ctx.stroke();
}
} else if (sprite.type == "0") {
  this.ctx.strokeStyle = COLORS.circle;
  this.ctx.rotate(-1);
  this.ctx.beginPath();
  ctx.arc(0, 0, size, 0, easingFunction(sprite.progress) *
Math.PI * 2);
  this.ctx.stroke();
}
this.ctx.restore();
}

draw(time) {
  let deltaTime = null;
  if (this.lastTime == null) {
    deltaTime = 0;
  } else {
    deltaTime = time - this.lastTime;
  }
  this.lastTime = time;
  this.ctx.clearRect(0, 0, this.canv.width,
this.canv.height);
  this.drawGrid(3);
  const cellWidth = this.canv.width / 3;
  for (let i = 0; i < 9; i++) {
    const x = (i % 3);
    const y = Math.floor(i / 3);

```



```
        this.drawSprite(this.sprites[i], (x + 0.5) * cellWidth,
(y + 0.5) * cellWidth, cellWidth);
        if (this.sprites[i] != null) {
            this.sprites[i].update(deltaTime);
        }
    }
}
}

const drawer = new Drawer(canv, ctx, statusDiv, splashElem,
playersElem, spinner);

function draw(time) {
    drawer.draw(time);
    requestAnimationFrame(draw);
}

draw(0);

async function update(cell) {
    console.log(cell);
    let path = "/gameRequest?id=" + id;
    if (cell != null) {
        path += "&cell=" + cell
    }
    const response = await fetch(path);
    if (!response.ok) {
        return;
    }
    const result = await response.json();
    console.log(result);
    if (result.state) {
        state = result.state;
    }
    // statusDiv.innerText = result.status;
    drawer.update(result);
}

update();
setInterval(() => update(), 500);

canv.addEventListener("click", (e) => {
    const x = Math.floor(3 * (e.clientX -
```

```

    canv.getBoundingClientRect().left) / canv.width);
    const y = Math.floor(3 * (e.clientY -
    canv.getBoundingClientRect().top) / canv.height);
    const cell = x + 3 * y;
    update(cell);
  });
  // static/pages-general.js
  const modalBg = document.getElementById("modal-bg");
  const modalGuide = document.getElementById("modal-guide");
  const iframe = document.getElementById("iframe");
  function showModalGuide(url) {
    modalBg.classList.remove("hidden");
    modalGuide.classList.remove("hidden");
    setTimeout(
      () => {
        modalBg.classList.remove("modal-hidden");
        modalGuide.classList.remove("modal-hidden");
      },
      20
    );
    iframe.src = url;
  }
  function hideModal() {
    modalBg.classList.add("modal-hidden");
    Array.from(document.getElementsByClassName("modal")).map((x)
    => x.classList.add("modal-hidden"));
    setTimeout(
      () => {
        modalBg.classList.add("hidden");
        modalGuide.classList.add("hidden");
      },
      400
    );
  }
  function followLink(link) {
    window.open(link + window.location.search, "_self");
  }
  function joinGame(gameId) {
    const params = new URLSearchParams(window.location.search);
    params.set("gameId", gameId);
    const newUrl = "/joinGame?" + params.toString();
    window.open(newUrl, "_self");
  }

```