МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет Прикладной информатики
Образовательная программа Мобильные и сетевые технологии

ОТЧЕТ
по лабораторной работе №2

по теме:

ВЗАИМОДЕЙСТВИЕ С ВНЕШНИМ API

Вариант: Платформа для фитнес-тренировок и здоровья

Студент: В.С. Корчагин

Преподаватель:
Должность Д.И. Добряков

Санкт-Петербург 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В первой лабораторной работе была разработана адаптивная вёрстка многостраничного веб-приложения «Grizzly Gym» для варианта «Платформа для фитнес-тренировок и здоровья». Интерфейс включал страницы входа и регистрации, личный кабинет, поиск тренировок с фильтрацией, детальные страницы тренировок и тренировочных планов, а также блог о здоровье и питании.

Цель лабораторной работы №2 — привязать разработанный интерфейс ко внешнему API, обеспечив реальный обмен данными между клиентской частью и сервером. В качестве API используется доработанный учебный проект, выполненный в предыдущем семестре, который развёрнут в виде набора сервисов и выступает в роли мокового (тестового) backend для фронтенда Grizzly Gym.

В рамках работы необходимо:

- реализовать авторизацию пользователей через внешний API;
- получать и отображать данные о тренировках, тренировочных планах, постах блога и комментариях;
- сохранять и читать прогресс пользователя;
- выполнять запросы к API средствами fetch через единый модуль-обёртку;
- обработать типовые ошибки сети и сервера на стороне клиента.

В настоящем отчёте описаны структура используемого API, организация взаимодействия с ним на стороне фронтенда и ключевые результаты интеграции. В лабораторной работе намеренно не использовались какие-либо графические диаграммы; архитектура описывается текстово.

1 Постановка задачи и исходные данные

1.1 Задачи лабораторной работы №2

Основные задачи, поставленные во второй лабораторной работе:

- использовать результаты ЛР1 в качестве основы клиентской части;
- подключить внешний API (моковый, локально развёрнутый) для:
    - регистрации и аутентификации пользователей;
    - работы со списком и деталями тренировок;
    - работы с тренировочными планами пользователя;
    - отображения записей блога и комментариев к ним;
    - хранения показателей прогресса пользователя;
- реализовать все запросы к API средствами fetch (через модуль `api.ts`);
- обеспечить корректную обработку ошибок HTTP и информирование пользователя.

Отдельно уточняется, что серверная часть не является предметом данной лабораторной работы и рассматривается как готовый внешний сервис, с которым взаимодействует фронтенд.

1.2 Используемый внешний API

В качестве внешнего API используется учебный backend, написанный и доработанный ранее. Он включает несколько независимых HTTP-сервисов (по сути, отдельные микросервисы), доступных по разным портам:

1. AUTH_API_URL — сервис аутентификации и управления пользователями (порт 4000);
2. WORKOUT_API_URL — сервис тренировок и тренировочных планов (порт 4001);
3. PROGRESS_API_URL — сервис хранения прогресса и тренировочных планов пользователя (порт 4002);
4. ORDER_API_URL — сервис заказов/подписок (порт 4003, в текущей лабораторной напрямую не используется фронтендом);
5. BLOG_API_URL — сервис записей блога и комментариев (порт 4004).

Адреса этих сервисов на стороне клиентского кода задаются в модуле `src/api.ts` через константы: при наличии глобальных переменных в `window` используются значения, переопределённые при сборке/развёртывании; в противном случае используются значения по умолчанию вида `http://localhost:400X`.

Ниже приведён фрагмент объявления этих констант в коде:

Листинг 1 — «Определение базовых URL внешних сервисов в модуле api.ts»

```
const AUTH_API_URL = (window as any).AUTH_API_URL || "http://localhost:4000";
const WORKOUT_API_URL = (window as any).WORKOUT_API_URL || "http://localhost:4001";
const BLOG_API_URL = (window as any).BLOG_API_URL || "http://localhost:4004";
const PROGRESS_API_URL = (window as any).PROGRESS_API_URL || "http://localhost:4002";
const ORDER_API_URL = (window as any).ORDER_API_URL || "http://localhost:4003";
```

Таким образом, с точки зрения фронтенда API представляет собой набор типичных REST-эндпоинтов, работающих с JSON-данными.

2 Проектирование взаимодействия с внешним API

2.1 Основные сущности и конечные точки

Фронтенд взаимодействует со следующими сущностями:

– пользователь (регистрация, вход по e-mail и паролю);
– тренировка (список тренировок, детальная информация по тренировке);
– тренировочный план (список планов, детализация одного плана);
– связка «план — пользователь» (добавление плана в «мои планы»);
– запись блога (список записей, отдельная запись);
– комментарии к записи блога;
– прогресс пользователя (вес, шаги, вода и т.п.).

Для каждой сущности предусмотрен набор HTTP-методов. Ключевые REST-эндпоинты в обобщённом виде:

1. POST `/auth/login` — аутентификация пользователя, выдача токена;
2. POST `/users` — регистрация нового пользователя;
3. GET `/users/email/{email}` — получение данных пользователя по адресу электронной почты;
4. GET `/workouts` — список всех тренировок;
5. GET `/workouts/{id}` — детальная информация по тренировке;
6. GET `/training-plans` — список тренировочных планов;
7. GET `/training-plans/{id}` — детальная информация по тренировочному плану;
8. GET `/training-plan-workouts` — связи «план — тренировка»;
9. GET `/blog-posts` — список записей блога;
10. GET `/blog-posts/{id}` — отдельная запись блога (включая комментарии);
11. POST `/blog-comments` — добавление комментария к записи блога;
12. GET `/user-progress/` — получение записей прогресса текущего пользователя;
13. POST `/user-progress/` — создание/обновление записи прогресса;

14. GET `/user-training-plans/` — получение списка тренировочных планов, добавленных пользователем;
15. POST `/user-training-plans` — добавление нового плана в «мои планы».

Часть эндпоинтов доступна только при наличии токена (защищённые ресурсы), часть — общедоступна (список тренировок, блога и т.п.).

2.2 Требования к клиентской части

Со стороны фронтенда были сформулированы следующие требования:

- все запросы выполнять через единый вспомогательный метод, который:
  - добавляет заголовки `Content-Type: application/json`;
  - при необходимости подставляет заголовок `Authorization: Bearer <token>`;
  - проверяет флаг `response.ok` и при ошибках выбрасывает исключение с текстом ответа;
- типизировать ответы и payload-ы через интерфейсы TypeScript:
  - `Workout`, `TrainingPlan`, `TrainingPlanWorkout`;
  - `BlogPost`, `BlogComment`;
  - `UserProgress`, `UserTrainingPlan`;
  - `AuthResponse`, `User`;
- хранить токен аутентификации и идентификаторы пользователя в `localStorage`;
- реализовать простую защиту маршрутов: доступ к `dashboard.html` и ряду API эндпоинтов — только при наличии токена, в противном случае перенаправление на `login.html`;
- обрабатывать сетевые ошибки и ошибки авторизации на уровне интерфейса (отображение сообщений об ошибках в блоках `alert` на формах входа/регистрации, прогресса и комментариев).

Ниже приведён фрагмент описания сущности тренировки в виде TypeScript-интерфейса:

Листинг 2 — «Интерфейс Workout для описания структуры данных тренировки»

```
export interface Workout {
  id: number;
  title: string;
  level?: string;
  workout_type?: string;
  duration_min?: number;
  description?: string;
  instructions?: string;
  video_url?: string;
}
```

3 Реализация взаимодействия с API на стороне клиента

3.1 Модуль api.ts: обёртка над fetch и типы данных

Модуль `src/api.ts` отвечает за инкапсуляцию всей работы с внешним API. В нём реализованы:

- константы с базовыми URL сервисов;
- интерфейсы TypeScript для описания структур данных, возвращаемых API;
- вспомогательная функция `apiFetch<T>(url, options, requiresAuth)`, работающая поверх стандартного `fetch`;
- функции-обёртки для конкретных REST-эндпоинтов.

Ключевая функция `apiFetch` реализует шаблон работы с HTTP-запросами: добавление заголовков, подстановка токена, проверка кода ответа и разбор JSON. Упрощённый фрагмент реализации:

Листинг 3 — «Упрощённый фрагмент функции apiFetch в модуле api.ts»

```
async function apiFetch<T>(
  url: string,
  options: RequestInit = {},
  requiresAuth = false,
): Promise<T> {
  const headers: HeadersInit = {
    "Content-Type": "application/json",
    ...(options.headers || {}),
  };

  if (requiresAuth) {
    const token = getAuthToken();
    if (token) {
      (headers as Record<string, string>).Authorization = `Bearer ${token}`;
    }
  }

  const response = await fetch(url, { ...options, headers });
  if (!response.ok) {
    const message = await response.text();
    throw new Error(message || "Request failed");
```

```
  }

  const contentType = response.headers.get("content-type") || "";
  if (contentType.includes("application/json")) {
    return response.json() as Promise<T>;
  }

  return response.text() as unknown as T;
}
```

На основе `apiFetch` реализуется набор функций, каждая из которых инкапсулирует работу с конкретным эндпоинтом. Например, функция логина и получения списка тренировок выглядят следующим образом:

Листинг 4 — «Примеры функций-обёрток login и getWorkouts»

```
async function login(email: string, password: string): Promise<AuthResponse>
{
  return apiFetch<AuthResponse>(`${AUTH_API_URL}/auth/login`, {
    method: "POST",
    body: JSON.stringify({ email, password }),
  });
}

async function getWorkouts(): Promise<Workout[]> {
  return apiFetch<Workout[]>(`${WORKOUT_API_URL}/workouts`);
}
```

Во второй лабораторной работе фронтенд ни разу не обращается к API напрямую через `fetch`; все вызовы выполняются через функции объекта `window.GrizzlyApi`.

3.2 Модуль auth.ts: хранение токена и контроль доступа

Файл `src/auth.ts` реализует минимальную логику аутентификации на стороне клиента. Основные функции:

- `setAuthData({ token, userId, userEmail, userName })` — сохраняет токен и данные пользователя в `localStorage`;
- `getAuthData()` — считывает токен и данные из `localStorage`;

– `requireAuth()` — проверяет наличие токена и при его отсутствии перенаправляет пользователя на страницу логина;

– `logout()` — очищает данные и перенаправляет на главную страницу.

Упрощённый фрагмент кода:

Листинг 5 — «Фрагмент модуля auth.ts для работы с данными авторизации»

```
interface AuthData {
  token: string | null;
  userId: string | null;
  userEmail: string | null;
  userName: string | null;
}

function setAuthData({ token, userId, userEmail, userName }:
Partial<AuthData>): void {
  if (token) localStorage.setItem("authToken", token);
  if (userId) localStorage.setItem("userId", userId);
  if (userEmail) localStorage.setItem("userEmail", userEmail);
  if (userName) localStorage.setItem("userName", userName);
}

function getAuthData(): AuthData {
  return {
    token: localStorage.getItem("authToken"),
    userId: localStorage.getItem("userId"),
    userEmail: localStorage.getItem("userEmail"),
    userName: localStorage.getItem("userName"),
  };
}
```

Эти функции экспортируются через глобальный объект `window.GrizzlyAuth` и используются, например, для защиты страницы `dashboard.html` вызовом `requireAuth()` в момент инициализации.

3.3 Инициализация страниц и работа с API в main.ts

Основная логика инициализации страниц и подключения данных API реализована в `src/main.ts`. После загрузки документа выполняются:

– `setupNavbar()` — анализирует наличие токена, скрывает/отображает ссылки Login, Register, Dashboard, Logout, показывает бейдж с именем пользователя;
– `routeInit()` — определяет значение `data-page` у `body` и вызывает соответствующую функцию инициализации (`initLanding`, `initWorkouts`, `initDashboard`, `initBlog`, `initBlogPost`, `initTrainingPlans`, `initTrainingPlanDetail`, `initWorkoutDetail`, `initLogin`, `initRegister`).

Фрагмент инициализации страницы логина:

Листинг 6 — «Фрагмент функции initLogin с использованием внешнего API»

```
function initLogin(): void {
  const form = document.getElementById("loginForm") as HTMLFormElement |
null;
  const alertBox = document.getElementById("loginAlert");

  form?.addEventListener("submit", async (e) => {
    e.preventDefault();
    alertBox?.classList.add("d-none");

    const email = getFormValue(form, "email");
    const password = getFormValue(form, "password");

    try {
      const res = await api.login(email, password);
      const token =
        res?.token ||
        (res as any)?.access_token ||
        (res as any)?.authToken;

      if (!token) throw new Error("No token returned");

      const userData = await api.getCurrentUserByEmail(email);

      window.GrizzlyAuth.setAuthData({
        token,
        userId: String(userData?.id ?? ""),
        userEmail: email,
```

```
        userName: userData?.name,
      });

      window.location.href = "dashboard.html";
    } catch (err) {
      const message =
        err instanceof Error ? err.message : "Unable to login. Please try
again.";
      if (alertBox) {
        alertBox.textContent = message;
        alertBox.classList.remove("d-none");
      }
    }
  });
}
```

Этот пример демонстрирует полный путь запроса: чтение данных формы, вызов `api.login`, последующее обращение к `getCurrentUserByEmail`, сохранение токена в `GrizzlyAuth` и реакцию на возможные ошибки.

Аналогичным образом реализованы функции инициализации других страниц:

- `initLanding()` — запрашивает данные тренировок и записей блога, выводит превью на главной странице;
- `initWorkouts()` — загружает список тренировок, вешает обработчики на фильтры и рендерит карточки;
- `initDashboard()` — загружает прогресс и планы пользователя, выводит данные в личном кабинете;
- `initBlog()` и `initBlogPost()` — отображают список записей блога, содержимое поста и комментарии.

3.4 Обработка ошибок и сценарии без доступного API

Важная часть интеграции — корректная работа интерфейса при ошибках:

- во всех местах, где выполняются сетевые запросы, используются блоки `try/catch` и обработка отклонённых промисов (`.catch(...)`);

- при невозможности загрузить данные:
    - на страницах главной и списка тренировок используются заранее подготовленные тестовые данные (fallback-массивы тренировок и постов блога);
    - на страницах, где нет смысла в фолбэках (детальная тренировка, детальный план, пост блога), выводятся текстовые сообщения «Unable to load…» / «Could not load…»;
- при ошибках авторизации:
    - пользователь видит сообщение в `alert`-блоке (например, при неверном логине/пароле);
    - при отсутствии токена доступ к защищённым страницам блокируется логикой `requireAuth()`.

Благодаря этому поведение приложения остаётся предсказуемым, даже если внешний API временно недоступен или возвращает ошибочные ответы.

4 Тестирование и результаты

В рамках лабораторной работы было выполнено функциональное тестирование основных пользовательских сценариев, связанных с внешним API.

Проверенные сценарии:

- регистрация нового пользователя:
  - корректное заполнение формы `register.html`;
  - успешная отправка данных и автоматический вход под созданной учётной записью;
- вход в существующую учётную запись:
  - успешный вход при корректных данных;
  - вывод сообщения об ошибке при неверном пароле или e-mail;
- работа с защищённой страницей `dashboard.html`:
  - при прямом переходе без токена — перенаправление на `login.html`;
  - при наличии токена — доступ к странице и отображение имени пользователя;
- загрузка списка тренировок и фильтрация:
  - отображение карточек тренировок при доступном API;
  - корректная фильтрация по уровню, типу и продолжительности;
  - использование запасного списка тренировок при недоступности API;
- просмотр детальной информации по тренировке и тренировочному плану:
  - правильная подстановка данных по `id` из строки запроса;
  - отображение текстовых полей и (при наличии) ссылки на видео;
- работа с блогом:
  - загрузка списка записей блога;
  - открытие отдельной записи;

- добавление комментария авторизованным пользователем и обновление списка комментариев;

- работа с прогрессом:
  - отображение последней записи прогресса и истории обновлений;
  - добавление новой записи прогресса и обновление отображения на дашборде.

По результатам тестирования все основные сценарии, связанные с обменом данными между фронтендом и внешним API, функционируют корректно. Ошибки авторизации и сетевые проблемы корректно обрабатываются, пользователю выводятся понятные сообщения.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы №2 выполнена интеграция клиентского интерфейса платформы «Grizzly Gym», разработанного в ЛР1, с внешним API, основанным на учебном backend-проекте предыдущего семестра.

Было реализовано:

– взаимодействие с несколькими сервисами API (аутентификация, тренировки, тренировочные планы, блог, прогресс пользователя);
– единая обёртка над fetch с обработкой заголовков, токена и ошибок ответа;
– хранение токена и пользовательских данных в `localStorage` и простая защита страниц от неавторизованного доступа;
– загрузка данных на все основные страницы приложения (главная, список тренировок, детальные страницы, блог, личный кабинет);
– обработка типичных ошибок (ошибки логина, недоступность API) с информированием пользователя.

Клиентская часть теперь не только отображает заранее свёрстанный интерфейс, но и полноценно обменивается данными с внешним API, что готовит проект к дальнейшему развитию: подключению реальных сервисов авторизации, платёжных систем и расширению бизнес-логики. Задачи лабораторной работы №2 по привязке фронтенда к внешнему API средствами fetch считаются выполненными.