Приднестровский государственный университет им. Т.Г. Шевченко
Физико-технический институт

ОСНОВЫ ВЕБ-РАЗРАБОТКИ С ПРИМЕНЕНИЕМ ИИ-АССИСТЕНТОВ

Лабораторный практикум

Разработал:
ст. преподаватель
кафедры ИТ
Бричаг Д.В.

г. Тирасполь
2025 г.

Лабораторная работа №6

Асинхронный JavaScript — API и хранилище

Цель работы:

− Научиться получать данные с сервера и сохранять состояние приложения в браузере с помощью localStorage и fetch.

− Освоить понятия JSON, Promise и async/await, а также научиться использовать ИИ для генерации асинхронного кода.

Теоретическая справка

Проблема: Синхронный ("обычный") код

Обычный JS-код синхронный и однопоточный.

- Однопоточный (Single-threaded) означает, что у JavaScript есть только один "рабочий", который может выполнять одну задачу в один момент времени.
- Синхронный (Synchronous) означает, что задачи выполняются строго друг за другом. Команда Б не начнется, пока не *полностью* завершится команда А.

Аналогия: Представьте себе кассу в фастфуде, где работает один кассир (один поток).

```
// Синхронный пример
console.log("1. Принять заказ"); // (Быстро)
// Эта "задача" очень долгая (например, 5 секунд)
someVerySlowTask(5000);
console.log("3. Отдать заказ"); // (Быстро)
```

В этом примере console.log("3. Отдать заказ") не выполнится, пока не завершится someVerySlowTask. Вся "очередь" (программа) блокируется на 5 секунд.

Почему это катастрофа для веб-сайта? Если бы запрос на сервер (скачивание данных, fetch) был синхронным, он бы "заморозил" всю страницу. Пользователь не смог бы нажать на кнопку, прокрутить страницу или даже выделить текст, пока данные не загрузятся. Весь интерфейс бы "умер".

Решение: Асинхронный код

Асинхронный код позволяет выполнять "длинные" задачи (в основном, операции ввода-вывода, I/O) без блокировки основного потока.

Аналогия: Тот же кассир. Вы делаете заказ (быстрая задача). Кассир не ждет, пока ваш бургер приготовится (долгая задача). Он отдает заказ на кухню (делегирует задачу), а сам немедленно принимает заказ у следующего человека.

Ваш бургер — это асинхронная операция. Когда он будет готов, "кухня" (среда выполнения JS) сообщит об этом, и вы его заберете.

Пример с setTimeout

```
setTimeout — это классический пример делегирования.
// Асинхронный пример
console.log("A. Принять заказ");

// "Кассир" (JS) говорит:
// "Эй, Среда Браузера (кухня), выполни вот эту функцию,
// но не сейчас, а через 1000 мс. А я пошел дальше."
setTimeout(() => {
  console.log("B. Ваш бургер готов (через 1 секунду)");
}, 1000);

// "Кассир" не ждет! Он сразу переходит к следующей команде.
console.log("C. Принять следующий заказ");
```

Результат в консоли:

```
A. Принять заказ
C. Принять следующий заказ
B. Ваш бургер готов (через 1 секунду)
```

Встроенная функция setTimeout (как и fetch, и обработчики событий) передает задачу в API браузера. Основной поток JS остается свободным и продолжает выполнять код.

Promise и async/await

Функция setTimeout — это просто. Но как получить результат от асинхронной операции? (Например, данные, которые скачались с сервера). Раньше для этого использовали "колбэки" (функции обратного вызова), что приводило к "аду колбэков" (Callback Hell).

Современный способ — Промисы (Promise).

Promise (Обещание)

Promise — это специальный объект-обертка, который "обещает", что в будущем он вернет результат (или ошибку).

Аналогия: Это как электронный номерок на той же кассе. Вы получаете его сразу (Promise в состоянии pending - "ожидание"), а не сам заказ.

У промиса есть три состояния:

- Pending (Ожидание): Заказ готовится.
- Fulfilled (Выполнено): Номерок "загорелся", заказ готов. Вы получаете данные.
- Rejected (Отклонено): На кухне закончились котлеты. Вы получаете ошибку.

Мы не знаем, когда промис выполнится, но мы можем "подписаться" на эти события с помощью методов .then() и .catch().

```
fetch('https://api.quotable.io/random') // 1. fetch возвращает Promise (номерок)
  .then(res => res.json()) // 2. .then() сработает, когда Promise выполнен (сервер ответил)
                           //    res.json() ТОЖЕ возвращает Promise (парсинг данных)
  .then(data => {          // 3. Этот .then() ждет, пока выполнится res.json()
    console.log(data.content); // 4. Наконец, мы получили данные!
  })
  .catch(error => {        // (!!!) ВАЖНО: сработает, если на ЛЮБОМ этапе (fetch или .json)
    console.error("Ошибка! Кухня закрыта!", error); // произошла ошибка (Rejected)
  });
```

async/await (Синтаксический сахар)

Код с .then().then().catch() может быть громоздким. async/await — это современный синтаксис, который позволяет писать асинхронный код так, будто он синхронный. Он не заменяет промисы, а просто делает работу с ними удобнее.

- async: Ставится перед функцией. Это слово "включает" магию. Оно заставляет функцию всегда возвращать Promise.
- await: Можно использовать только внутри async функции. Это слово говорит: "Поставь выполнение этой функции на паузу и дождись, пока Promise справа выполнится. Затем 'разверни' его и положи результат в переменную".

```
// Мы "оборачиваем" наш код в async-функцию
async function getQuote() {
  try {
    // 1. "Пауза", ждем ответа от сервера.
    //    Результат (объект Response) кладется в 'res'.
    const res = await fetch('https://api.quotable.io/random');

    // 2. "Пауза", ждем, пока данные распарсятся из JSON.
    //    Результат (объект с цитатой) кладется в 'data'.
    const data = await res.json();

    // 3. Только теперь код идет дальше.
    console.log(data.content);

  } catch (error) {
    // 4. Если ЛЮБОЙ из 'await' выше был отклонен (rejected),
    //    выполнение "прыгнет" сюда.
    console.error("Что-то пошло не так:", error);
  }
}

// Не забываем вызвать нашу async-функцию
getQuote();
```

Этот код намного чище, и в нем легко обрабатывать ошибки с помощью обычного try...catch.

JSON (JavaScript Object Notation)

JSON — это не JavaScript. Это текстовый формат для обмена данными.

– Проблема: Сервер (написанный на Python, Java, PHP) и ваш браузер (JavaScript) должны общаться на одном языке. Они не могут просто обменяться «объектами» из своих языков.

– Решение: Они обмениваются текстом (строками), который отформатирован по строгим правилам JSON.

Этот формат очень похож на синтаксис объектов в JS, но:

- Все ключи (свойства) должны быть в двойных кавычках.
- Нельзя использовать функции, undefined, комментарии.

```
// JavaScript ОБЪЕКТ (в памяти)
const obj = { name: "Alex", isStudent: true };

// --- Передаем на сервер ---

// 1. Превращаем объект в ТЕКСТОВУЮ СТРОКУ (для отправки по сети)
const str = JSON.stringify(obj);
// str теперь равен: '{"name":"Alex","isStudent":true}'  <-- ЭТО СТРОКА

// --- Получаем с сервера ---

// 2. Получили с сервера строку '{"name":"Alex","isStudent":true}'
//    Нам нужен объект, а не строка.
const back = JSON.parse(str);
// back теперь снова ОБЪЕКТ: { name: "Alex", isStudent: true }
```

res.json() из примера с fetch — это просто удобный метод, который берет ответ сервера (в виде строки JSON) и сам делает за вас JSON.parse().

localStorage (Локальное хранилище)

localStorage — это маленькая "база данных" (скорее, словарь "ключ-значение") прямо в вашем браузере.

Проблема: Переменные JavaScript (let user = "Alex") "умирают" при перезагрузке или закрытии страницы.

Решение: localStorage сохраняет данные на диске. Они переживают перезагрузку и даже закрытие браузера.

Это отлично подходит для хранения нечувствительных данных:

Настройки пользователя (например, theme: 'dark')

Содержимое корзины (если пользователь не вошел в аккаунт)

Имя пользователя, чтобы "вспомнить" его при следующем визите.

```
// Сохраняем "темную тему" под ключом 'theme'
localStorage.setItem('theme', 'dark');

// ...пользователь перезагружает страницу...

// Пытаемся прочитать данные из-под ключа 'theme'
const savedTheme = localStorage.getItem('theme'); // 'dark'

if (savedTheme) {
  console.log(`Добро пожаловать обратно! Ваша тема: ${savedTheme}`);
}
```

Важный нюанс: localStorage может хранить только строки. Если вы хотите сохранить объект, вы должны сначала превратить его в JSON-строку!

```
const userSettings = { theme: 'dark', fontSize: 16 };

// НЕПРАВИЛЬНО:
// localStorage.setItem('settings', userSettings);
// Сохранит '[object Object]'

// ПРАВИЛЬНО (сериализация):
localStorage.setItem('settings', JSON.stringify(userSettings));

// Как получить обратно (десериализация):
const settingsStr = localStorage.getItem('settings');
const userSettingsObj = JSON.parse(settingsStr);
// userSettingsObj теперь { theme: 'dark', fontSize: 16 }
```

Практическая часть

Сформируем простое техническое задание. Необходимо:

Используя верстку и скрипты из ЛР №5 (семантическая страница):

- добавить асинхронные методы JavaScript;
- расширить возможности скриптов при помощи ИИ;
- протестировать их работу через консоль.

Подготовка проекта

– Создайте папку lab6.

– Скопируйте в неё вёрстку из ЛР №5 (страницу с интерактивностями).

– Убедитесь, что подключён файл script.js или иной, что подключен к странице, как в примере.

```
<script src="script.js"></script>
```

Повторяем: переключатель темы

В прошлых работах вы уже делали тёмную и светлую темы. А также переключение посредством JS. Однако после перезагрузки страницы состояние сбрасывалось. Мы привыкли к тому, что сайты запоминают наши настройки на сайте.

Добавим сохранение выбора темы, ведь для этого даже нет необходимости записывать куки. Для пользовательских настроек удобно применять localStorage

Добавьте к стилям страницы стили для тёмной темы и переключатель. Также попросите нейросеть реализовать логику переключателя и сохранение состояния. Пример промпта для ИИ:

```
Модифицируй этот скрипт переключения темы, чтобы выбранная тема сохранялась в localStor-
age и автоматически применялась при перезагрузке страницы.
```

Ожидаемый результат:

- Пользователь выбирает тему (dark/light).
- После перезагрузки страница открывается с выбранной темой.

Получаем данные с API (fetch + async/await)

Теперь научимся получать данные с сервера. Для этого добавим на страницу блок отзывов клиентов. А текст будем получать динамически из сгенерированных случайных цитат стороннего сервиса, который нам их вежливо передаст средствами API.

Создайте запрос для ИИ, например:

```
Создай блок отзывов для лендинга, содержащий несколько отзывов, которые можно листать в карусели.
```

Следующим шагом, запросите доработку, чтобы текст отзывов формировался случайной цитатой:

```
Доработай JS-код, чтобы использовался fetch и async/await для получения текста каждого отзыва с https://api.quotable.io/random
```

Добавьте полученный код в script.js и проверьте, что при загрузке страницы получены разные тексты у пользовательских отзывов.

Обработка состояния «Загрузка...»

Хорошим тоном при разработке веб страниц является явно показывать пользователю процесс загрузки данных. Чтобы страница выглядела информативнее, и не сбивала с толку пользователя в случае, если ответ от удалённого сервера задерживается добавим обработку загрузки.

Попросите текстовую модель:

```
Измени код, чтобы перед получением цитаты показывалось слово "Загрузка...", а после
получения — заменялось текстом цитаты.
```

Также часто слово «Загрузка» заменяют различными дизайнерскими «статус-барами». Не забудьте проверить полученный результат:

- При загрузке страницы видно “Загрузка...”.
- После ответа API появляется текст-цитата.

Подгружаем картинки с API

Аналогичным образом реализуйте динамическое получение изображений для галереи. Только вместо сервиса цитат используйте API сервисов Unsplash или Picsum.photos.

Пример промта:

```
Сделай JS-функцию, которая получает случайные изображения с Unsplash API (или
Picsum.photos)
и отображает их в div с id="images".
```

Проверка работы скриптов

- Откройте сайт в браузере.
- Проверьте:
  - Переключатель темы сохраняется при перезагрузке.
  - Цитаты обновляются по кнопке.
  - При загрузке виден статус "Загрузка...".
  - Галерея получает изображения из API.
- Убедитесь, что в консоли нет ошибок. В случае если есть, уточните у нейросети причину и как исправить.

Работа с git и GitHub

1. Инициализируйте репозиторий:

```
git init
git add .
git commit -m "Lab6: fetch, async/await, localStorage"
```

2. Создайте новый репозиторий на GitHub.

3. Свяжите локальный проект с GitHub:

```
git remote add origin https://github.com/<ваш_логин<lab5_repo>.git
git branch -M main
git push -u origin main
```

Отчёт (в README.md)

В README.md добавить:

- Заголовок: *Лабораторная работа №6. Асинхронный JavaScript: API и хранилище*
  - Скриншоты работы интерактивных элементов.
  - Ответы на вопросы:
    - Что делает fetch?
    - Зачем нужны async/await?
    - Как работает localStorage?
    - Где помог ИИ, а где пришлось разбираться вручную?

Результаты работы

В итоге у вас должно быть:

1. Страница из ЛР5 с добавленным блоком отзывов и галереей.
2. Рабочий fetch с API и обработкой загрузки.

3. Код сохранения темы в localStorage.
4. Галерея с изображениями из API.
5. Репозиторий на GitHub и отчёт в README.md.

Критерии оценки

- localStorage сохраняет выбор темы (2 балла).
- fetch получает цитату из API (2 балла).
- Реализована обработка состояния "Загрузка..." (2 балла).
- Реализована галерея с динамическим получением изображений (2 балла).
- Код выложен на GitHub и оформлен отчёт (2 балла).