ChatGPT

Учебный план «Основы веб-разработки и тестирования» (лето 2025)

Расписание: 2 занятия в неделю с июля по август 2025 года (до конца августа). Ниже приведён поурочный план для двух учеников-новичков под руководством опытного наставника. Каждое занятие содержит тему, дату, цель, ключевые понятия и материалы/слайды для проведения урока. План охватывает базовые технологии веб-разработки (HTML, CSS, JavaScript), основы клиент-серверного взаимодействия, API, базы данных, элементы backend, а также вводит инструменты разработки (VS Code, Git, DevTools и др.). Учтены потребности как будущего программиста, так и тестировщика, включая основы тестирования.

Занятие 1 (21 июля 2025) – Введение в веб-разработку. Роли разработчика и тестировщика

Цель: Познакомить учеников с основами работы веб-приложений и распределением ролей в команде (веб-разработчик vs. тестировщик). Дать общее представление о том, как устроен интернет и веб (модель «клиент–сервер»), и обсудить цели тестирования в процессе разработки.

Ключевые понятия:
- Веб-страница, веб-приложение, браузер
- Клиент и сервер, их взаимодействие (запрос-ответ по HTTP) [1]
- Frontend vs. Backend (интерфейс пользователя vs. серверная логика)
- Роли в IT: разработчик (создаёт функциональность) и тестировщик (проверяет качество)
- Жизненный цикл разработки ПО (в общих чертах), место тестирования в нём

Материалы и слайды:
- *Слайд 1:* Заголовок урока, план занятия, краткое описание целей.
- *Слайд 2:* Схема архитектуры «клиент–сервер» – на изображении показано, как несколько клиентских устройств (например, ноутбук с браузером) через сеть отправляют запросы серверу, а сервер обрабатывает запрос и возвращает ответ [1]. Обсуждение: браузер выступает как клиент, сервер хранит данные и логику.
- *Слайд 3:* Объяснение принципа работы веб: URL, HTTP-запросы и ответы, пример (поиск в Google – браузер отправляет запрос, сервер возвращает результаты) [1]. Упоминание, что сервер может обращаться к базе данных для получения информации.
- *Слайд 4:* Разделение на frontend и backend: что выполняется на стороне клиента (отображение интерфейса), что на стороне сервера (бизнес-логика, работа с данными). Примеры веб-приложения и его компонентов.
- *Слайд 5:* Роли в разработке: обязанности веб-программиста (написание кода фронтенда и/или бекенда) и обязанности тестировщика (поиск багов, проверка соответствия требованиям). Объяснение, почему даже для тестировщика важно понимать основы веб-разработки.
- *Слайд 6:* Цели тестирования: обеспечение качества продукта, поиск ошибок до релиза. Введение понятий *QA* (обеспечение качества) и *QC* (контроль качества) [2]. Кратко перечисляются основные виды тестирования (ручное vs автоматизированное, функциональное, UI-тестирование и т.д.) – подробно они будут рассмотрены в дальнейшем.

Занятие 2 (24 июля 2025) – Основы HTML. Структура веб-страницы

Цель: Научить основам языка разметки HTML – создания структуры веб-страницы. Ученики познакомятся с базовыми тегами HTML и создадут свою первую простую веб-страницу. Также будет показана настройка рабочего окружения (редактор Visual Studio Code) для разработки.

Ключевые понятия:
- HTML (HyperText Markup Language) – язык разметки для структуры веб-страниц (что это и для чего нужен)
- Структура HTML-документа: `<!DOCTYPE>`, тег `<html>`, разделы `<head>` и `<body>`
- Базовые теги: заголовки `<h1>…<h6>`, параграфы `<p>`, списки `<ul>/<ol>` и элементы списка `<li>`
- Гиперссылки `<a>` и изображения `<img>` – как вставлять ссылки и картинки [3]
- Таблицы `<table>`, строка `<tr>` и ячейки `<td>`, формы `<form>` и поля ввода (текстовое поле `<input>`, кнопка `<button>` и др.) [3]
- Атрибуты тегов (например, `href` для ссылок, `src` и `alt` для изображений)
- Инструменты: установка и использование VS Code, создание первого HTML-файла, базовые возможности редактора (подсветка синтаксиса, Emmet) [4]

Материалы и слайды:
- *Слайд 1:* Заголовок «HTML – структура веб-страницы». Коротко: что такое HTML, пример простой страницы (Hello World).
- *Слайд 2:* Минимальная структура HTML-документа (пример кода с `<!DOCTYPE html>`, `<html>`, `<head>` с мета-информацией и `<body>`). Объяснение роли каждой секции.
- *Слайд 3:* Обзор базовых тегов: примеры заголовка `<h1>` и параграфа `<p>` в коде и как они отображаются на странице. Демонстрация: ученики создают файл `index.html` и пишут пару заголовков и абзацев.
- *Слайд 4:* Пример списка и таблицы: HTML-код списка `<ul><li>...</li></ul>` и простой таблицы с строками и ячейками. Упражнение: добавить список интересов и таблицу расписания в HTML-файл.
- *Слайд 5:* Добавление изображений и ссылок: синтаксис `<img src="..." alt="...">` и `<a href="...">...</a>`. Показать на примере (вставить картинку и ссылку на любимый сайт). Упомянуть важность атрибута `alt` для доступности. [3]
- *Слайд 6:* Формы: пример формы с полем ввода текста и кнопкой отправки (элементы `<form>`, `<input type="text">`, `<button type="submit">`). Объяснить, что формы используются для отправки данных на сервер (пока без деталей реализации).
- *Слайд 7:* Настройка окружения: скриншот VS Code с открытым HTML-файлом. Рекомендации по структуре проекта (создать папку проекта, хранить файлы с понятными именами). Кратко про Emmet – как с помощью сокращений быстро писать разметку (например, ввод `!` и нажатие Tab генерирует шаблон HTML) [4].
- *Слайд 8:* Интерактив: открыть созданную страницу в браузере. Ученики видят результаты своего HTML-кода. Обсудить, как браузер интерпретирует HTML. Домашнее задание: потренироваться добавить ещё элементы на страницу, по желанию.

Занятие 3 (28 июля 2025) – Основы CSS. Оформление страниц стилями

Цель: Дать представление об каскадных таблицах стилей (CSS) и научить применять простые стили к HTML-элементам. Ученики узнают, как отделяется структура (HTML) от оформления (CSS), освоят базовые свойства CSS для изменения внешнего вида страницы.

Ключевые понятия:
- CSS (Cascading Style Sheets) – язык описания внешнего вида страниц (что это, связь с HTML)
- Подключение CSS к HTML: встроенные стили (атрибут `style`), внутренний `<style>` в `<head>`, внешние CSS-файлы и `<link rel="stylesheet">` [5]
- Базовый синтаксис CSS: правило состоит из селектора и блока деклараций `{ свойство: значение; }` [5]
- Селекторы: по тегу (например, `p { ... }`), по классу (`.classname`), по идентификатору (`#id`), вложенные селекторы
- Каскадность и приоритеты: понятие специфичности селекторов, наследование стилей
- Основные CSS-свойства: цвет текста (`color`), размер шрифта (`font-size`), фон (`background-color`), выравнивание текста (`text-align`), отступы и поля (`margin`, `padding`) и границы (`border`)
- Понятие блочных и строчных элементов, модель коробки (box model: content + padding + border + margin)

Материалы и слайды:
- *Слайд 1:* Заголовок «CSS – оформление страницы». Кратко о роли CSS (отделение дизайна от структуры).
- *Слайд 2:* Способы подключения CSS: пример встроенного стиля (атрибут `style="..."` прямо в тег – показать, но отметить, что лучше избегать), пример `<style>...</style>` в HTML и подключение внешнего файла `<link href="styles.css">`. Рекомендовать использовать внешний файл для реальных проектов [5].
- *Слайд 3:* Пример CSS-правила: селектор `h1` и свойство `color: blue;` – демонстрация изменения цвета заголовка. Показать синтаксис: селектор (выбирает элементы) и блок `{ ... }` с парами свойство:значение.
- *Слайд 4:* Селекторы и классы: объяснить, как задавать класс HTML-элементу (`class="highlight"` например) и писать селектор `.highlight { background: yellow; }` для выделения. Аналогично с id (`id="main-title"` и селектор `#main-title { ... }`). [5]
- *Слайд 5:* Принцип каскадности: что происходит, если один элемент подходит под несколько селекторов – упоминание специфичности (id > класс > тег) и порядка подключения. Пример: если к абзацу применяются стили из тега `p` и из класса `.note`, какой будет приоритет.
- *Слайд 6:* Свойства для текста: показ примеров – изменить шрифт (`font-family`), размер (`font-size`), жирность (`font-weight`), курсив (`font-style`), цвет (`color`). Демонстрация: применить несколько свойств к элементу.
- *Слайд 7:* Свойства блочной модели: показать на схеме структуру box model (content, padding, border, margin). Пример: задать для блока фиксированную ширину (`width`), границу (`border: 1px solid black;`), внутренний отступ `padding` и внешний отступ `margin` – и посмотреть как это влияет на расположение элемента.
- *Слайд 8:* Практика: ученики подключают свой файл *styles.css* к своей HTML-странице и пробуют изменить стили некоторых элементов (например, задать фон страницы, цвет и шрифт заголовков, отцентрировать заголовок, выделить список другим цветом и т.д.). Наставник

помогает и показывает инструменты разработчика (DevTools) для быстрого эксперимента со стилями прямо в браузере.

Занятие 4 (31 июля 2025) – Введение в JavaScript. Основы программирования

Цель: Познакомить учеников с базовыми концепциями программирования на примере языка JavaScript. Научить писать простейшие скрипты: вывод сообщений, использование переменных, простые вычисления и условные конструкции. Эта база важна как будущим разработчикам, так и тестировщикам (для понимания логики работы программ).

Ключевые понятия:
- JavaScript – язык программирования, выполняющийся в браузере на стороне клиента (кратко о предназначении: делает страницы интерактивными)
- Подключение JS к странице: тег `<script>` (в простых примерах можно писать прямо в HTML, затем лучше выносить во внешний файл)
- Переменные: что это такое, объявление (ключевое слово `let` / `const` в JS) и присваивание значений 6
- Типы данных: числа, строки, булевы (логические) значения; примеры литералов (например, `"Hello"` – строка, `42` – число)
- Основные операции: арифметические (`+` , `-` , `*` , `/`), конкатенация строк (`+` для склеивания строк)
- Встроенный вывод в консоль: функция `console.log()` (показать, как можно вывести значение переменной или сообщение)
- Условные конструкции: `if ... else` – выполнение кода по условию (логическое выражение, операторы сравнения `==` , `===` , `>` , `<` и логические операторы `&&` , `||`) 6
- Циклы: понятие повторения действий, пример цикла `for` (например, вывести числа от 1 до 5) 6
- Функции: определение функции (`function имя() { ... }`), передача параметров и возврат значения (простой пример) 6

Материалы и слайды:
- *Слайд 1:* «JavaScript – оживляем страницу». Пояснение, что JS – единственный язык, который понимают веб-браузеры для логики на странице. (На backend могут быть другие языки, о них позже).
- *Слайд 2:* Первое знакомство с кодом: пример скрипта прямо в HTML – `<script>console.log("Привет, мир!");</script>` . Показать, как открыть консоль в браузере (F12 DevTools, вкладка Console) и увидеть сообщение. Объяснить, что это вывод для разработчика.
- *Слайд 3:* Переменные и типы: объяснение концепции переменной как «коробочки» для данных. Пример: `let name = "Alice";` (строка), `let age = 30;` (число). Потом можно вывести `console.log(name, age)` . Объяснить про `let` и `const` (для постоянных). Упомянуть, что в старом коде встречается `var` , но сейчас рекомендуются `let/const` .
- *Слайд 4:* Операции: математика в JS – пример: `let x = 5, y = 2; let sum = x + y;` (результат 7). Показать вычисления, вывод через `console.log("x+y=", sum)` . Также пример конкатенации: `let msg = "Hello, " + name;` – склеивание строк.
- *Слайд 5:* Условия: пример кода `if (age >= 18) { console.log("Совершеннолетний"); } else { console.log("Несовершеннолетний"); }` . Объяснить, как работает условие, что такое блок `if` , как вычисляется логическое выражение. Показать операторы `>` `<` `===` и

логические `&&` (И), `||` (ИЛИ) – на простых условиях.
- *Слайд 6:* Циклы: пример `for (let i = 1; i <= 5; i++) { console.log(i); }` – объяснить структуру цикла (инициализация, условие, шаг). Показать, как цикл выводит ряд чисел. Упомянуть, что есть разные виды циклов (`while`, `do...while`), но на практике чаще используют `for`.
- *Слайд 7:* Функции: объяснение, зачем нужны (для многократного использования кода). Простой пример: `function greet(username) { console.log("Привет, " + username); }` – определили функцию `greet`, вызываем `greet("Мир")` и она печатает приветствие. Объяснить про параметры (здесь `username`) и возможность вернуть значение (ключевое слово `return`).
- *Слайд 8:* Практика на консоли: наставник предлагает короткие задачки – например, попросить учеников в консоли браузера посчитать 2+2, определить переменную со своим именем и вывести приветствие с этим именем, написать условие, которое выведет сообщение в зависимости от значения какой-то переменной. Ученик пробует, наставник помогает.
- *Слайд 9:* Итог: подчеркнуть, что эти конструкции – фундамент программирования. Далее будем учиться применять JS для работы непосредственно с содержимым веб-страницы. Домашнее задание: решить пару простых задач (например, написать скрипт, который выводит все чётные числа от 1 до 10).

Занятие 5 (4 августа 2025) – JavaScript в браузере: работа с DOM и отладка

Цель: Научить использовать JavaScript для взаимодействия с элементами страницы (DOM – Document Object Model). Ученики научатся через скрипт получать элементы HTML, изменять их содержимое, реагировать на действия пользователя (например, клик по кнопке). Также в этом занятии рассмотреть инструменты разработчика браузера (DevTools) для отладки HTML/CSS/JS – это важно как для разработки, так и для тестирования.

Ключевые понятия:
- DOM (Document Object Model) – представление структуры HTML-страницы в виде объекта, с которым можно работать через JS
- Поиск элементов на странице: методы `document.getElementById(...)`, `document.querySelector(...)` для получения узлов DOM
- Изменение содержания и стилей через JS: свойство `.innerText` / `.innerHTML` для текста, изменение CSS через свойство `.style` или путем добавления/удаления классов
- События и обработчики: понятие событий (например, событие `click` на кнопку, `input` на поле ввода), назначение обработчиков через `element.addEventListener("click", function) {...}`
- Пример интерактивности: показ сообщения при клике, валидация простого поля формы на лету
- Инструменты разработчика (DevTools): инспектор элементов (Elements) для просмотра и правки HTML/CSS на лету, консоль (Console) для вывода и отладки, отладчик (Debugger) для пошагового выполнения скрипта
- Использование консоли для отладки: вывод значений переменных, сообщений об ошибках, предупреждений; метод `console.table()` для отображения массивов/объектов
- Основы отладки JS: точки останова (breakpoints) в DevTools, шаговое выполнение кода, просмотр значений переменных в процессе

Материалы и слайды:
- *Слайд 1:* Заголовок «DOM – взаимодействие с страницей через JS». Краткое объяснение, что браузер представил HTML как объектную модель – «дерево» узлов.

- *Слайд 2:* Поиск и изменение элемента: пример кода – в HTML есть `<p id="demo">Hello</p>`. JS: `let p = document.getElementById("demo"); p.innerText = "Привет!";` – объяснение, как получили элемент по id и изменили текст. Упомянуть querySelector (например, `document.querySelector(".highlight")` для класса).
- *Слайд 3:* Демонстрация вживую: открыть страницу, в консоли выполнить `document.getElementById(...)` и изменить что-то. Показать, что изменения отражаются сразу.
- *Слайд 4:* События: схема «пользователь -> событие -> обработчик». Пример: HTML кнопка `<button id="btn">Нажми меня</button>`. JS:

```
let btn = document.getElementById("btn");
btn.addEventListener("click", function() {
    alert("Кнопка нажата!");
});
```

Объяснить, что сделали: нашли кнопку, привязали к ней обработчик на событие `click`, который вызывает функцию (здесь – просто `alert`). Показать, как это работает – при клике всплывает alert.
- *Слайд 5:* Пример динамического изменения: сделать поле ввода и подсказку. HTML: `<input id="nameInput"> <span id="greet"></span>`. JS: добавить обработчик на событие `input` для поля, который берет значение и вставляет в `<span id="greet">` приветствие, например: «Привет, {имя}». Это демонстрирует работу с `.value` поля и изменением текста другого элемента.
- *Слайд 6:* Инструменты разработчика: скриншот браузерных DevTools (например, Chrome) – выделена вкладка Elements и Console. Пояснить, что **Elements** позволяет смотреть HTML-дерево и CSS-стили, пробовать править их прямо в браузере для эксперимента (эти изменения не сохраняются в файле, но помогают дебагу). **Console** – для вывода сообщений, и можно вводить команды JS прямо на работающей странице.
- *Слайд 7:* Отладка JS: показать вкладку **Sources** или **Debugger** – как ставить breakpoint (точку останова) на определенной строке JS-кода. Объяснить, как выполнение кода останавливается и можно по шагам проходить (Step over, Step into) и видеть значения переменных. Это помогает находить логические ошибки.
- *Слайд 8:* Пример отладки: написать скрипт с небольшой ошибкой (например, опечатка в имени переменной или неверное условие). Запустить, увидеть ошибку в консоли (красное сообщение с указанием строки). Показать, как перейти к строке, исправить ошибку. Подчеркнуть: ошибки синтаксиса браузер показывает в консоли с сообщением. Тестировщику важно уметь понимать сообщения об ошибках, а разработчику – их исправлять.
- *Слайд 9:* Практическая часть: ученики получают готовый простенький HTML-файл с формой (например, поле ввода и кнопка). Задача – написать скрипт в консоли или файле, который при нажатии кнопки берет текст из поля и выводит `alert` с этим текстом. Наставник помогает, если что-то не выходит, используя DevTools для диагностики.
- *Слайд 10:* Резюме: благодаря JS страница может реагировать на действия пользователя. Отладочные инструменты позволяют и разработчикам, и тестировщикам исследовать работу приложения: **разработчик** – чтобы исправлять баги, **тестировщик** – чтобы отслеживать ошибки и получать дополнительную информацию о найденных дефектах. (Например, консоль показывает сообщения об ошибках скрипта, что полезно при описании бага разработчикам.) [7]

Занятие 6 (7 августа 2025) – Введение в серверную часть и языки программирования (Python)

Цель: Дать ученикам общее представление о backend-разработке – что происходит на сервере. Познакомить с базовыми принципами работы серверных программ: обработка запросов, бизнес-логика, взаимодействие с базой данных. Для расширения кругозора будет показан другой язык программирования (например, Python) и написан небольшой скрипт, чтобы сравнить с JavaScript и понять общие принципы.

Ключевые понятия:
- Backend – серверная часть приложения, которая выполняется на удалённом сервере (или «на облаке») и отвечает за логику, недоступную напрямую пользователю
- Обзор серверных языков: Python, Java, PHP, Node.js (JavaScript на сервере) – разные технологии для backend, в курсе обзорно упоминаются
- Выполнение кода на сервере: приложение постоянно работает и ожидает запросов от клиентов; когда запрос приходит – код на сервере обрабатывает его, при необходимости обращается к базе данных, и формирует ответ (например, HTML-страницу или данные в формате JSON) [8]
- Веб-сервер и серверное приложение: веб-сервер (например, Nginx, Apache) может передавать запросы приложению (написанному на Flask, Django, Express, etc.) для обработки
- HTTP-запросы методов POST/GET: различия в назначении (GET – получить данные, POST – отправить данные на сервер для обработки)
- (Символическое) знакомство с языком Python: синтаксис отличается от JS (отступы вместо фигурных скобок, ключевые слова `def`, типизация динамическая). Python часто используется в автоматизации тестирования и скриптах.
- Пример простого кода на Python: вывод текста (`print("Hello, world")`), работа с переменными и операциями (показать аналогично тому, что делали на JS). Продемонстрировать простейшую программу на Python в терминале.
- Принцип работы небольшого серверного скрипта: (по возможности) показать минимальный пример кода сервера. Например, с помощью Python/Flask – несколько строк, которые поднимают сервер и при обращении на определённый URL возвращают строку “Hello from server”. Либо аналог на Node.js (Express) – чтобы понять, как код «слушает» запросы.
- Без углубления: объяснить, что детали написания серверов выходят за рамки базового курса, но важно понимать концепцию: фронтенд (браузер) обращается к бэкенду (серверному приложению), которое работает постоянно и умеет обрабатывать запросы.

Материалы и слайды:
- *Слайд 1:* Заголовок «Что такое Backend?». Определения: backend – всё, что скрыто “за кулисами” веб-приложения на сервере. Аналогия: если сайт представить как ресторан – frontend это меню и обслуживание, backend – кухня и повара, которые готовят по запросу.
- *Слайд 2:* Схема взаимодействия фронтенда с бэкендом: браузер (клиент) -> запрос -> интернет -> сервер -> обработка -> ответ -> браузер. Показать, где в этой схеме могла бы участвовать база данных (сервер может обратиться к ней во время обработки) [8].
- *Слайд 3:* Языки backend: перечисление с примерами – **PHP** (исторически для веб-сайтов, WordPress), **Node.js (JS)** – позволяет использовать JavaScript на сервере, **Python** – используется в веб-фреймворках (Django, Flask) и в автоматизации, **Java/C#** – для крупных корпоративных систем, и др. Упомянуть, что несмотря на разнообразие языков, принципы серверной логики схожи.
- *Слайд 4:* Мини-урок Python: показать простой код на Python и сравнить с JS. Например:

```
name = "Alice"
age = 21
if age >= 18:
    print(f"Привет, {name}! Ты совершеннолетний.")
else:
    print(f"Привет, {name}! Тебе ещё нет 18.")
```

Здесь показать синтаксис: нет точки с запятой, блоки кода определяются отступами, строковые шаблоны через `f"..."`. Выполнить этот код (можно скопировать в interpreter) и показать вывод.
- *Слайд 5:* Отметить общие элементы: переменные, условия, циклы – есть и там и там (привести пример цикла `for i in range(5):` в Python, это аналогичный результат как JS-цикл). Идея: языки отличаются синтаксисом, но логика схожа. Это важно понять начинающим – освоив один язык, легче понять второй.
- *Слайд 6:* **Демонстрация серверного кода:** объяснить, что на практике backend-разработчик не пишет низкоуровневый код обработки сетевых соединений – используют фреймворки. Показать упрощённо: - Пример с Flask (Python):

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "Hello from server!"
app.run()
```

Объяснить: этот код создает веб-приложение, которое при обращении к корневому URL "/" возвращает строку. - (Или пример Express.js на Node: несколько строк, создающих сервер). Наставник запускает такой мини-сервер локально и демонстрирует через браузер: при заходе на `http://localhost:5000/` появляется заданное сообщение от сервера.
- *Слайд 7:* Разбор демонстрации: по шагам объяснить, что произошло – браузер отправил HTTP-запрос на локальный сервер, наш код (функция `hello()`) выполнился и вернул строку, сервер обернул её в ответ, браузер отобразил. Это и есть основа backend-принципов. Ученики видят, что можно самим написать простой сервер.
- *Слайд 8:* Вопросы-ответы: возможно, у учеников возникнут вопросы (например, насколько сложно сделать полноценный сайт). Объяснить, что обычно backend включает больше (маршрутизация разных URL, обработка данных форм, авторизация пользователей, и пр.), но эти детали вне рамок текущего плана. Наша цель – понимать, что происходит после того, как пользователь нажал кнопку «Отправить» на сайте: запрос ушёл на сервер, там выполнился код (например, проверил данные, сохранил в БД) и вернул результат.
- *Слайд 9:* Подведение итога: теперь мы знаем, что веб-разработка состоит из двух частей – фронтенд (HTML/CSS/JS в браузере) и бекенд (код на сервере + база данных). В следующих занятиях рассмотрим, как фронтенд и бэкенд общаются друг с другом (API), а также познакомимся с базами данных.

Занятие 7 (11 августа 2025) – Взаимодействие клиент–сервер: основы API и форматы данных

Цель: Рассмотреть подробно, как фронтенд и бекенд общаются посредством запросов. Ввести понятие API (Application Programming Interface) применительно к вебу – как определённого интерфейса для обмена данными между клиентом и сервером. Показать на практике формат JSON, основы HTTP-протокола (методы запросов, коды ответов). Также познакомить с инструментом Postman (или его аналогами) для тестирования API – навык, полезный для тестировщиков.

Ключевые понятия:
- API (Application Programming Interface) – **интерфейс прикладного программирования**, набор правил и способов, по которым программы (клиент и сервер) взаимодействуют друг с другом [9]
- Веб-API (REST API): API веб-сервиса, обычно реализованный через протокол HTTP, который принимает запросы (обычно URL + метод) и возвращает данные (часто в формате JSON)
- Форматы данных: **JSON** (JavaScript Object Notation) – текстовый формат обмена данными (ключ: значение, похож на объекты JS), пример: `{"name": "Alice", "age": 25}`; упоминуть **XML** как другой формат (менее популярный в новых API)
- HTTP методы: GET (получить данные), POST (создать/отправить новые данные), PUT/PATCH (изменить), DELETE (удалить) – представить как глаголы для операций с ресурсами
- URL и эндпойнты: как формируется адрес запроса (например, `https://api.weather.com/city?name=London`), понятие пути и параметров запроса
- HTTP-коды ответа: 200 OK (успех), 404 Not Found (не найдено), 500 Internal Server Error (ошибка на сервере), и т.д. – тестировщик должен понимать их значение при проверке ответов API
- Пример работы API: запрос погоды – клиент отправляет GET-запрос на URL с названием города, сервер (weather API) обрабатывает (обращается к своей базе данных погоды) [8] и возвращает JSON с прогнозом (температура, условие) [10]
- Postman (или аналогичный инструмент) для тестирования API: позволяет вручную посылать HTTP-запросы и получать ответы, менять методы, заголовки, тело запроса. Это важный инструмент для тестировщиков при проверке backend без UI.
- Безопасность и ключи API (в общих чертах): некоторые сервисы требуют ключ API для доступа; упомянуть, что не все API открытые.

Материалы и слайды:
- *Слайд 1:* Заголовок «Что такое API?». Определение простыми словами: API – «контракт» между клиентом и сервером, описывающий, **как** клиент может запросить данные или выполнить действие, и **какой** ответ он получит [9]. Привести бытовую аналогию: API похож на меню в ресторане – вы заказываете по меню (правилам), кухня (сервер) знает, что и как приготовить, и выдаёт вам готовое блюдо (ответ).
- *Слайд 2:* HTTP-запросы и ответы: на диаграмме показать цикл: Клиент -> HTTP-запрос -> Сервер -> обработка -> HTTP-ответ -> Клиент. Расписать структуру HTTP-запроса: строка запроса (метод + URL), заголовки (headers), тело (body – для методов типа POST), и HTTP-ответ: код, заголовки, тело.
- *Слайд 3:* Методы HTTP: таблица или список с основными методами и их назначением (GET – запросить/читать ресурс, POST – создать новый ресурс или передать данные, PUT – полное обновление, PATCH – частичное обновление, DELETE – удаление). Пример для каждого: GET `/users` – получить список пользователей, POST `/users` – добавить нового пользователя, etc.
- *Слайд 4:* Формат JSON: показать пример JSON-объекта и объяснить синтаксис (ключи в кавычках, двоеточие, значения — число, строка в кавычках, вложенные объекты или массивы). Сравнить JSON с JS-объектом — почти то же самое, за исключением того, что строки в JSON обязательно в двойных кавычках. Возможно, взять реальный пример – фрагмент ответа от публичного API

(например, погода: `{"city":"Москва","temp":17,"condition":"ясно"}`). Упомянуть, что JSON читается легко и используется повсеместно в веб-API.
- *Слайд 5:* Демонстрация API вызова: наставник в реальном времени делает запрос к публичному API. Например, через браузер или Postman получить данные – возможно, OpenWeatherMap (если есть демо-API) или httpbin. Для простоты, можно использовать браузер: открыть `https://api.agify.io?name=Michael` (сервис, который по имени даёт предполагаемый возраст) – браузер покажет JSON `{"name":"Michael","age":XX,"count":YY}`. Показать этот JSON, объяснить, что браузер просто отобразил «сырые данные».
- *Слайд 6:* Знакомство с Postman: скриншот окна Postman с примером GET-запроса. Объяснить, что тестировщик может вручную сформировать любой запрос: выбрать метод, ввести URL, добавить параметры, нажать Send и увидеть ответ (JSON или др.), а также код ответа и время. Это удобно для проверки API отдельно от фронтенда. [11] (в программе курсов тестирования упоминается умение работать с Postman и API).
- *Слайд 7:* Практика: ученики пробуют отправить запрос через Postman (либо через онлайн-аналог, например reqbin). Задача – обратиться к тестовому API, например: `https://jsonplaceholder.typicode.com/todos/1` (это фейковый API, который вернёт JSON задачи). Они должны получить ответ, увидеть код 200. Наставник помогает интерпретировать результат.
- *Слайд 8:* HTTP-коды: вывести список важных кодов: 200 (OK, всё хорошо), 201 (Created – успешно создан ресурс, чаще ответ на POST), 400 (Bad Request – ошибка клиента, неправильный запрос), 401/403 (Unauthorized/Forbidden – нет доступа, например, нужен API-ключ или авторизация), 404 (Not Found – ресурс не найден), 500 (Internal Server Error – сбой на сервере). Объяснить, что при тестировании API проверяются не только данные, но и корректность кодов (например, если запрос неверный, должен вернуться 400, а не 200).
- *Слайд 9:* Безопасность: буквально пару слов о том, что некоторые API требуют ключ (API key) или токен доступа – их нужно указывать, иначе сервер вернёт 401 Unauthorized. Тестировщик должен знать, как использовать такие ключи (на уровне вводной информации).
- *Слайд 10:* Закрепление материала: задать вопросы ученикам – что такое API в двух словах? чем отличается GET от POST? почему формат JSON популярен? Также связать с предыдущими темами: фронтенд (JS) может обращаться к API через встроенные функции (например, `fetch` в JS для AJAX-запросов) – хотя мы подробно это не разбирали, важно знать, что браузерный JS тоже может делать HTTP-запросы (например, динамически подгружать данные без перезагрузки страницы).
- *Слайд 11:* В контексте командной работы: подчеркнуть, что тестировщики часто проверяют API напрямую, особенно в случаях, когда UI ещё не готов или для полного охвата (например, проверить граничные случаи, ошибки). А разработчики должны документировать свой API (спецификации, Swagger и т.п.), чтобы тестерам и другим разработчикам было понятно, как им пользоваться.

Занятие 8 (14 августа 2025) – Основы баз данных. Работа с данными на сервере

Цель: Дать базовое представление о базах данных (БД) – зачем они нужны веб-приложениям и как с ними взаимодействовать. Рассмотреть основные концепции реляционных баз данных: таблицы, строки, столбцы, SQL-запросы (на примере простых SELECT/INSERT). Также обсудить роль БД в приложении и упомянуть другие виды хранилищ (NoSQL) для общего развития.

Ключевые понятия:
- База данных – организованное хранилище данных, к которому может обращаться приложение. В веб-разработке чаще всего используется для хранения информации о пользователях, товарах, сообщениях и т.д.
- Реляционная база данных (SQL-БД): хранит данные в таблицах (таблица похожа на Excel-лист:

строки – записи, столбцы – поля)
- Пример таблицы: **Users** (поля: id, name, email, age; каждая строка – один пользователь). Показать небольшую таблицу с 2-3 записями в презентации.
- SQL (Structured Query Language) – язык запросов для реляционных БД. Основные команды: SELECT (выбрать данные), INSERT (вставить новую запись), UPDATE (обновить), DELETE (удалить)
- Пример простого SQL-запроса: `SELECT name, age FROM Users WHERE age >= 18;` – выберет имена и возраст всех пользователей 18+. Объяснить синтаксис (что такое SELECT, FROM, WHERE).
- Другой пример: `INSERT INTO Users (name, email, age) VALUES ('Alice','alice@example.com',30);` – добавление новой строки. Упомянуть, что после такого запроса в таблице появится новая запись.
- Связь с backend: серверное приложение выполняет SQL-запросы, когда нужно получить или изменить данные. Например, при логине пользователя сервер сделает SELECT к таблице пользователей, чтобы найти запись с нужным логином и паролем.
- Понятие *ORM* (Object-Relational Mapping) можно упомянуть поверхностно: разработчики часто используют библиотеки, которые позволяют работать с БД не пиша сырые SQL, а вызывая методы на объектах. Но принципиально, под капотом всё равно выполняются SQL-запросы.
- NoSQL базы данных: упомянуть, что кроме реляционных есть и другие (например, документы JSON хранящиеся как в MongoDB, или ключ-значение как Redis). Но для начала фокус – на реляционных, так как они широко распространены.
- Роль тестировщика: умение сделать базовые запросы к БД для проверки данных (например, убедиться, что данные корректно записались). В курсе тестирования обычно учат основам SQL [12].

Материалы и слайды:
- *Слайд 1:* Заголовок «Что такое база данных?». Определение: специализированная программа для хранения и извлечения данных. Объяснить, что данные можно хранить в файлах, но с ростом объёма и требований удобнее использовать БД, которая эффективно ищет, хранит, защищает данные.
- *Слайд 2:* Реляционная модель: показать схему таблицы. Например, таблица *Students*: колонки (ID, Name, BirthYear, Grade). И привести 2-3 строк в табличке. Пояснить, что каждая строка – отдельный объект (запись), а колонки – атрибуты. Реляционная означает, что таблицы могут иметь связи (например, таблица Orders может ссылаться на таблицу Customers через CustomerID), но глубоко пока не идём.
- *Слайд 3:* Пример SQL – SELECT: написать образец на слайде и разобрать. Например, `SELECT Name, Grade FROM Students WHERE Grade >= 90;`. Пояснить: выберем имена и оценки всех студентов с оценкой >= 90. Подчеркнуть ключевые слова: SELECT (что выбрать), FROM (откуда – таблица), WHERE (условие фильтрации).
- *Слайд 4:* Пример SQL – INSERT/UPDATE: `INSERT INTO Students (Name, BirthYear, Grade) VALUES ('Иван', 2005, 85);` – добавление нового студента. `UPDATE Students SET Grade = 95 WHERE Name = 'Иван';` – обновление его оценки. `DELETE FROM Students WHERE Name = 'Иван';` – удаление записи. Эти примеры для демонстрации основных операций CRUD (Create, Read, Update, Delete).
- *Слайд 5:* На диаграмме показать взаимодействие: Сервер <-> База данных. Повторить путь обработки запроса: браузер -> запрос -> сервер, сервер при необходимости делает SQL-запрос к БД [8], получает данные, формирует ответ -> отправляет клиенту. Заметить, что база данных обычно развернута на отдельном сервере или на том же, и общается с приложением через локальную сеть (не путать с прямым доступом из браузера – браузер не лезет напрямую в БД, всегда через сервер).
- *Слайд 6:* Транзакции (упоминание): чтобы дать понимание надёжности – в БД есть механизм транзакций, который гарантирует целостность данных (например, либо все связанные изменения применяются, либо ни одного – чтобы не было неконсистентных состояний). В веб-

приложениях это важно, например, при переводе денег между счетами. Но детали можно не разбирать, просто обозначить, что такое есть.
- *Слайд 7:* Инструменты для работы с БД: показать интерфейс какой-нибудь простой СУБД или утилиты (например, Adminer, TablePlus, phpMyAdmin, или даже SQLite Browser) – где можно увидеть таблицы и выполнять запросы. Тестировщики могут использовать такие инструменты, чтобы смотреть содержимое БД напрямую при необходимости.
- *Слайд 8:* Практика (теоретическая, без реальной БД): Наставник даёт ученикам несколько ситуаций и просит сформулировать на бумаге простейшие SQL-запросы. Например: *«Найти всех пользователей старше 18 лет»* -> ученик пишет пример SELECT с WHERE age > 18. *«Добавить нового пользователя с именем ...»* -> пишет INSERT. Проверить и обсудить ответы. Цель – закрепить понимание синтаксиса.
- *Слайд 9:* Обзор NoSQL: коротко упомянуть, что не все БД используют таблицы. Например, **MongoDB** хранит данные как JSON-документы (пример JSON документа для пользователя), **Redis** хранит пары ключ-значение (например, для кэша), **ElasticSearch** для поиска по тексту и т.д. Но основная мысль: независимо от типа БД, задача – хранение данных, и backend-разработчик взаимодействует с БД через запросы или библиотеки.
- *Слайд 10:* С точки зрения тестирования: очень полезно знать основы SQL. В реальной работе **тестировщик** может проверять, правильно ли сохраняются данные после действий в приложении. Например, после регистрации пользователя – зайти в БД и убедиться, что появилась новая запись с правильными полями. Многие курсы для тестировщиков включают основы SQL [11], поэтому мы заложили базу.
- *Слайд 11:* Вывод: теперь у нас есть полная картина: **клиент** (HTML/CSS/JS) <-> **сервер** (принимает запросы, выполняет логику на языке программирования) <-> **база данных** (хранит информацию). Все эти компоненты взаимодействуют. В следующих занятиях мы перейдём к качественной стороне – как тестировать всё это и как работать в команде с помощью современных инструментов.

Занятие 9 (18 августа 2025) – Основы тестирования web-приложений. Ручное тестирование

Цель: Ввести учащихся в методологию тестирования. Рассмотреть процесс тестирования веб-приложений вручную: от анализа требований и составления тест-кейсов до поиска багов и написания отчетов о дефектах. Изучить базовые принципы тестирования (например, принцип неполноты тестирования, пестицидного парадокса и др.), жизненный цикл багов. Также обсудить различные типы тестирования (функциональное, UX, кросс-браузерное) применительно к веб.

Ключевые понятия:
- Что такое тестирование ПО и зачем оно нужно: проверка соответствия продукта требованиям и ожиданиям, повышение качества продукта [2]
- Роль тестировщика в команде: на этапе разработки – превентивно участвует (например, в уточнении требований), на этапе тестирования – проводит проверки, после – контролирует исправление дефектов
- Требования и спецификация: основа для тестирования – тестировщик разрабатывает сценарии, опираясь на требования к функционалу
- Тест-дизайн: процесс создания тест-кейсов (шагов проверки) на основе требований и здравого смысла. Понятие тест-кейса: описание шагов, ожидаемого результата
- Пример тест-кейса: **Сценарий** – проверка формы логина. **Шаги:** открыть страницу логина, ввести корректный email, ввести неверный пароль, нажать "Войти". **Ожидание:** должно показать сообщение об ошибке "неверный пароль" и не пустить в систему.
- Виды тестирования веб-приложений: - **Функциональное тестирование** – проверка, что каждая

функция (форма, кнопка, ссылка) работает согласно требованиям. - **UI/UX-тестирование** – проверка удобства использования, соответствия дизайну, корректности отображения элементов, отзывчивости интерфейса. - **Кросс-браузерное тестирование** – проверка сайта в разных браузерах (Chrome, Firefox, Safari, Edge) и на разных устройствах (ПК, планшет, мобильный) – нет ли отличий или проблем. - **Тестирование производительности** (упомянуть, но не углубляться) – измерение скорости загрузки страниц, нагрузки. - **Безопасностное тестирование** (упоминание) – проверка базовых уязвимостей (веб-формы: XSS, SQL-инъекции – хотя это скорее задача спецов, но важно знать о существовании).
- Принципы тестирования (согласно ISTQB, 7 принципов) – перечислить: например, *тестирование показывает наличие дефектов, но не их отсутствие* (нельзя доказать, что багов нет, только что не найдены) [2]; *исчерпывающее тестирование невозможно* (невозможно перебрать все комбинации, нужно разумно выбирать сценарии); *раннее тестирование* (начинать на самых ранних стадиях, чтобы дешевле исправить); *скопление дефектов* (обычно большинство багов сосредоточено в нескольких модулях); *парадокс пестицида* (если постоянно повторять одни и те же тесты, со временем они теряют эффективность – нужно периодически пересматривать и обновлять тесты); *тестирование зависит от контекста*; *заблуждение об отсутствии ошибок* (продукт без багов, но ненужный пользователю, всё равно неуспешен).
- Жизненный цикл дефекта (bug life cycle): шаги от обнаружения баги до её исправления и закрытия. Статусы баг-репорта: new, open, in progress, fixed, re-test, closed, reopen (если повторно воспроизвелась) и т.д.
- Инструменты для ведения баг-репортов: например, Jira, Trello, YouTrack – тестировщик фиксирует найденные баги в системе отслеживания, прикладывает шаги воспроизведения, скриншоты. Разработчик по ним исправляет.
- Взаимодействие тестировщика и разработчика: важно уметь правильно описать проблему (четко, с шагами, ожидаемым vs фактическим результатом) и приоритетом. Разработчик воспроизводит, исправляет, тестировщик проверяет фикс.

Материалы и слайды:
- *Слайд 1:* Заголовок «Введение в тестирование». Определение: тестирование – процесс оценки качества программного продукта на протяжении жизненного цикла, с целью нахождения ошибок и убедиться, что продукт удовлетворяет требованиям [2]. Возможно, показать диаграмму V-model (где на одну сторону разработческие этапы, на другой – соответствующие уровни тестирования), но может быть избыточно для вводного уровня.
- *Слайд 2:* Роль и задачи тестировщика: список – анализ требований, написание тест-планов и тест-кейсов, проведение тестирования (мануального или с помощью инструментов), заведение баг-репортов, регресс-тестирование после исправлений, участие в улучшении процессов качества.
- *Слайд 3:* Пример тест-кейса (графически оформить): Шаги vs Ожидаемый результат. Взять простой пример с веб-формой или навигацией. Проговорить, что хороший тест-кейс должен быть понятен и воспроизводим любым человеком.
- *Слайд 4:* Типы тестирования веб-приложения: перечисление с кратким описанием каждого. Можно добавить и *Smoke testing* (поверхностная проверка основных функций, что приложение в целом работоспособно) и *Regression testing* (регрессия – повторная проверка всего важного после внесения изменений, чтобы убедиться что ничего не сломалось старое).
- *Слайд 5:* Принципы тестирования: вывести 7 принципов кратко箇条 и обсудить каждый. Например: 1. **Тестирование демонстрирует наличие дефектов, а не их отсутствие** [2] – после тестирования мы можем найти определённые баги и исправить, но нельзя быть уверенным, что багов больше нет. 2. **Исчерпывающее тестирование невозможно** – объяснить, что комбинаций данных бесконечно много, поэтому нужно выбирать наиболее важные наборы. ... и так далее.
- *Слайд 6:* Инструменты тестировщика: показать интерфейс Jira (скриншот формы создания бага).

Разъяснить обязательные поля баг-репорта: заголовок (summary) – кратко о проблеме, описание – шаги воспроизведения, фактический и ожидаемый результат, приоритет/важность, окружение (например, браузер, версия ОС). Показать пример заполненного бага.
- *Слайд 7:* Жизненный цикл бага: блок-схема или последовательность статусов. Например: Новый -> Назначен разработчику -> Исправляется -> Выставлен статус "Исправлено" -> Тестировщик проверяет на новой сборке -> либо закрывает (если ок), либо переоткрывает (если проблема осталась). Объяснить понятия *retest* (перепроверка) и *regression test* (проверка, что исправление ничего не сломало рядом).
- *Слайд 8:* Практическая работа: предложить ученикам сыграть роли – один «разработчик», другой «тестировщик». Наставник заранее вставил небольшой баг в тестовый учебный сайт (например, кнопка не делает ничего или опечатка на странице). Тестировщик должен найти эту проблему, составить баг-репорт (можно на бумаге или шаблоне), разработчик – «исправить» (на словах, или действительно наставник исправит код), потом тестировщик – проверить опять. Это игровое упражнение поможет понять процесс.
- *Слайд 9:* Особенности веб-тестирования: обратить внимание на вещи, специфичные для веба – кросс-браузерность (например, в Safari может что-то выглядеть иначе, нужно проверять), адаптивность (верстка на мобильных устройствах), состояние сессии (что будет, если открыть одну страницу в двух вкладках, будет ли конфликт?), ссылки (нет ли битых ссылок 404), безопасность (не введёт ли XSS, например, вставка `<script>` в поле – стоит ли приложение от такого защищено?). Эти вещи обычно входят в чек-листы веб-тестирования [11].
- *Слайд 10:* Вопросы для самопроверки/обсуждения: *Почему нельзя протестировать программу полностью?* (ожидаемый ответ: слишком много случаев, нужно применять принципы и анализ риска). *Какие, на ваш взгляд, самые критичные вещи в веб-приложении надо протестировать в первую очередь?* (авторизация, основные функции). *Как бы вы протестировали поле ввода пароля?* (ответ: граничные значения, пустой ввод, спецсимволы, очень длинная строка, SQL-инъекция – проверить, не попадают ли непредусмотренные данные).
- *Слайд 11:* Резюме: тестирование – широкая область, но зная основы веб-технологий (HTML/JS/сервер) вы уже лучше понимаете, где могут скрываться дефекты. Хороший тестировщик мыслит с точки зрения пользователя и предполагет, что может пойти не так. На следующем занятии мы поговорим об инструментах совместной разработки, которые используют и разработчики, и тестировщики – системах контроля версий и о том, как работать над проектом сообща.

Занятие 10 (21 августа 2025) – Системы контроля версий и совместная разработка (Git, GitHub)

Цель: Познакомить учеников с системой контроля версий Git – фундаментальным инструментом разработки. Показать, как с её помощью несколько человек могут работать над одним проектом, отслеживая изменения. Научить основным командам Git (инициализация репозитория, коммит изменений, просмотр истории). Рассмотреть практику совместной работы через платформу GitHub: форк, pull request, код-ревью. Для тестировщиков – объяснить, как брать свежий код, как отслеживать версии сборок.

Ключевые понятия:
- Контроль версий – система, которая хранит историю изменений файлов проекта, позволяет возвращаться к прошлым версиям и сливать изменения от разных участников. Зачем нужен: чтобы не терять работу, иметь возможность отменить изменения, работать параллельно в команде без конфликтов.
- **Git** – наиболее популярная распределённая система контроля версий. Репозиторий Git – это хранилище проекта с полным журналом изменений.
- Основные операции Git: - `git init` – создание нового репозитория в текущей папке

- `git clone` – клонирование (скачивание) существующего удалённого репозитория на свой компьютер (например, с GitHub)
- `git status` – показать состояние (какие файлы изменены, какие не отслеживаются)
- `git add` – начать отслеживание нового файла или отметить изменения для коммита
- `git commit` – зафиксировать изменения с сообщением (снимок состояния с описанием)
- `git log` – журнал коммитов (история)
- `git diff` – показать отличия между текущими изменениями и последним коммитом (или между двумя коммитами)
- Ветвление (branches): концепция параллельных линий разработки. По умолчанию есть ветка `master` или `main` (основная). Можно создать новую ветку для фичи (`git branch feature1` + переключение `git checkout feature1` или командой `git switch`). Ветка позволяет экспериментировать, не затрагивая основную, а потом сливать (merge) изменения [13].
- Слияние (merge) и разрешение конфликтов: если два человека изменили одну и ту же строку в файле по-разному, при попытке слить Git отметит конфликт – нужно вручную выбрать верный вариант. Объяснить, что это нормально и как решается (открыть файл, решить, сделать commit).
- Удалённый репозиторий: например, **GitHub** – платформа для хранения Git-репозиториев в интернете и совместной работы. Понятия origin (удалённый репозиторий по умолчанию), push (отправить свои коммиты на сервер), pull (скачать изменения с сервера) [13].
- Fork и Pull Request (PR): на GitHub тестировщик или сторонний разработчик может форкнуть (скопировать) репозиторий, внести предложения (например, тестировщик может поправить опечатку в тексте, или разработчик – новую фичу) и через PR предложить интегрировать изменения обратно [14]. Наставник может упомянуть, что даже тестировщики могут участвовать – например, поправить README или автотесты.
- Code Review: практика, когда каждый PR просматривается другим разработчиком прежде чем объединиться – для контроля качества кода. В контексте курса – показать, что работа в команде подразумевает коммуникацию, обсуждение кода.
- DevOps знакомство (упоминание из плана курса): интеграция с CI/CD – возможно упомянуть, что после слияния кода могут автоматически запускаться тесты, сборка деплоиться на тестовый сервер (но деталей не давать, просто сказать, что Git часто связан с автоматизированными процессами развертывания).

Материалы и слайды:
- *Слайд 1:* Заголовок «Знакомство с Git – хранение истории проекта». Небольшая история: Git создан Линусом Торвальдсом для управления разработкой ядра Linux, сейчас де-факто стандарт в индустрии.
- *Слайд 2:* Проблема без контроля версий: показать комичную ситуацию – два разработчика работают над одним файлом одновременно, перезаписывают работу друг друга; или разработчик хранит папки *project_v1*, *project_final*, *project_final2* – путаница. Констатировать: нужен инструмент, чтобы избежать такого хаоса.
- *Слайд 3:* Основные концепции Git: репозиторий как *история изменений*. Представить коммиты как цепочку снимков. Показать на диаграмме несколько кружков (коммитов) с хешами и сообщениями.
- *Слайд 4:* Мини-демо (теоретически): создать простой репозиторий. Привести команды:

```
git init
git add index.html
git commit -m "Add index page"
```

Далее: изменить файл (например, добавить заголовок), опять `git add` и `git commit -m "Add heading"`. Показать `git log` – видно 2 коммита. Объяснить, как

сообщения коммитов помогают понять, что делалось.
- *Слайд 5:* Ветвление: нарисовать две ветки исходящие от одного коммита. Объяснить кейс: разработчик А делает новую фичу (ветка feature-A), разработчик В – другую (feature-B). Они оба ответвились от main. Каждый коммитит в свою ветку. Потом, когда готовы, сливают ветки обратно в main. Git умно совмещает истории. [13] Показать, что это позволяет параллелизм.
- *Слайд 6:* GitHub: скриншот страницы репозитория на GitHub (например, с файлами проекта). Пояснить, что GitHub – удаленное хранилище + веб-интерфейс. Он отображает файлы, историю коммитов, статистику, позволяет заводить issues (задачи/баги) – полезно для отслеживания.
- *Слайд 7:* Push/Pull: диаграмма – разработчик сделал коммит локально, затем `git push` – коммит ушел в облако (GitHub). Другой разработчик хочет получить последние изменения – делает `git pull` и синхронизируется [14]. Объяснить, что pull под капотом делает fetch + merge (или rebase) – то есть скачивает и мержит.
- *Слайд 8:* Fork & PR: показать процесс: тестировщик форкает проект (получает копию в свой аккаунт GitHub), делает изменения (например, корректирует опечатку в документации), коммитит и на GitHub нажимает "New Pull Request". Maintainer (владелец проекта) видит PR, делает review, оставляет комментарии или принимает. Когда принимают – его коммиты попадают в основной проект. Это распространённый workflow в open-source и внутри команд при участии нескольких человек. [14]
- *Слайд 9:* Конфликты: картинка или пример кода, где двое изменили одну строку. Показать, как выглядит конфликт в Git (в файле пометки `<<<<<< HEAD` и `>>>>>>> branch` и разделителем `======`). Объяснить, что нужно вручную выбрать, какой вариант оставить или объединить оба. Это часть ежедневной работы, не страшно.
- *Слайд 10:* Практика: если есть возможность, наставник заранее создаёт на GitHub приватный репозиторий с простым проектом (например, тот самый учебный сайт). Каждый ученик клонирует (через VS Code или Git GUI, либо командой). Затем сделать небольшое изменение: ученик А правит текст заголовка, ученик В добавляет пункт в список. Каждый делает commit и push. Посмотреть на GitHub историю – оба коммита видны. Если был конфликт (например, оба правили один файл рядом) – решить его вместе.
- *Слайд 11:* Интеграция с тестированием: рассказать, как **тестировщик** может использовать Git: - Получать актуальную версию приложения для теста (клон репозитория, запуск на своей машине – если это возможно). - Просматривать историю изменений между сборками (например, знать, что изменилось – на что обратить внимание, какие баги могли потенциально появиться). - В некоторых случаях тестировщики сами верстают или пишут автотесты – тогда они тоже вносят изменения в репозиторий, поэтому должны знать Git.
- *Слайд 12:* DevOps (упоминание): связать с тем, что после того как код в репозитории, существуют инструменты CI (Continuous Integration) – например, GitHub Actions, Jenkins – которые на каждый push выполняют скрипты: запускают тесты, собирают приложение, раскатывают на сервер. Это уже следующая стадия – деплой. Мы глубоко не идём, но студенты должны понимать, что Git – это центральное звено, вокруг которого автоматизировано многое.
- *Слайд 13:* Заключение: освоение Git – обязательный навык разработчика, и очень полезный навык для тестировщика. Рекомендация: практиковаться с Git при любых изменениях кода, читать дополнительные материалы. Домашнее задание: пройти интерактивный туториал по Git (например, learngitbranching.js.org – если владеют английским, или русские аналоги) для закрепления.

Занятие 11 (25 августа 2025) – Мини-проект (часть 1): Планирование и разработка функционала

Цель: Начать командный мини-проект, применяя знания, полученные за курс. В этой части ученики совместно с наставником планируют простое веб-приложение, распределяют роли и

задачи. Будет разработан основной функционал (в основном фронтенд, возможно с имитацией бекенда или использованием простого API). Параллельно будет вестись контроль версий через GitHub. Тестировщик участник подготовит тестовые сценарии для функционала.

Ключевые понятия и задачи:
- Выбор проекта: **простое веб-приложение**, соответствующее уровню учеников. Например: *«Список дел»* (to-do list) – страница, где пользователь может добавлять задачи и отмечать выполненными; или *«Гостевая книга»* – форма для отправки сообщений и список сообщений; либо *«Конвертер валют»* – ввод числа, выбор валют и вывод результата через внешний API. Проект должен быть достаточно прост, чтобы выполнить за 2 занятия, и охватывать HTML/CSS/JS, а также позволить тестировщику что-то проверять.
- Определение требований: что должен делать проект. Например, для списка дел – функции: добавить новую задачу, увидеть список задач, пометить задачу выполненной (и, скажем, убирать или перечёркивать её). Желательно оформить эти требования письменно как список user stories или bullet points.
- Разделение ролей: учащийся-разработчик берёт на себя написание кода (HTML, CSS, JS). Учащийся-тестировщик – контроль качества: помогает составлять критерии при планировании (что считать успешным добавлением задачи), пишет тест-кейсы на каждую функцию, потом будет тестировать. (Конечно, в обучающем проекте обе роли работают рядом и обучаются взаимно, но мы стимулируем специализацию).
- Планирование реализации: набросать простой макет интерфейса (можно ручкой на бумаге или на доске – где будет поле ввода, кнопка, как будут отображаться задачи). Обсудить, какие HTML-элементы нужны, какие события JS. Тестировщик тут же думает о граничных случаях (а что если добавить пустую задачу? а очень длинный текст? и т.д.). Это совместный **brainstorm**, моделирующий реальный процесс.
- Создание репозитория для проекта: наставник помогает создать репозиторий на GitHub (если не сделали ранее). Настройка базовой структуры: возможно, использовать шаблон или с нуля создать файлы index.html, style.css, script.js. Разработчик форкает/клонит, тестировщик тоже (он может не код писать, но может написать README, например).
- Реализация: разработчик начинает с структуры HTML (разметка страницы) – создает форму и список. Коммитит это. Затем стили – простое оформление (CSS). Затем JS – функциональность добавления элемента в список (массив задач, отрисовка). Шагами, с промежуточными коммитами. Тестировщик может выполнять роль ассистента: параллельно открывать страницу, проверять, что элементы на месте, предлагать улучшения (“кнопка добавления неактивна, пока текст не введён” – возможно, пожелание).
- Интерактивность проекта: при добавлении задачи JS берёт текст из поля, добавляет в DOM новый элемент (например, `<li>`). Можно сделать, что при клике по задаче она отмечается выполненной (например, зачеркивается через CSS класс). Это демонстрирует DOM-манипуляции.
- (Если хватает времени и желания, можно подключить простейший backend/API: например, использовать localStorage браузера, чтобы сохранять задачи между перезагрузками, или хитрее – подключить бесплатный REST API like jsonbin for storing tasks. Но можно обойтись без этого, сфокусировавшись на фронте).
- Тестирование по ходу разработки: после реализации каждой функции тестировщик выполняет её проверку согласно своим сценариям. Например, после того как функция добавления готова, тестировщик пробует: добавить обычную задачу – работает? добавить пустую задачу – что происходит (если ничего, то баг или фича? Решают – либо запрещаем кнопку если пусто, либо выводим alert, либо это баг). Таким образом, сразу выявляются улучшения.
- Документирование: тестировщик готовит список тест-кейсов на проект (можно в табличке или текстом). Разработчик может готовить небольшое описание для README (о проекте, как запускать). Это хорошая практика.
- Git сотрудничество: разработчик пушит изменения. Тестировщик может пробовать их получать

(`git pull`). Если тестировщик вносит изменения (например, правки текста в HTML или README) – делает коммит и пуш (или PR). Они таким образом практикуют совместную работу через Git.

Материалы и слайды:
- *Слайд 1:* Объявление мини-проекта: название и краткое описание задачи (например, «To-Do List App – приложение для управления списком дел»). Цель проекта – применить всё выученное: создать рабочую страничку с интерактивностью и протестировать её.
- *Слайд 2:* Требования к функционалу (списком). Например: - Пользователь должен иметь возможность ввести текст задачи и добавить её в список нажатием кнопки "Добавить". - Новая задача появляется в списке с чекбоксом или кнопкой отметки выполнения. - При отметке задачи выполненной – задача визуально помечается (например, перечёркнута) или перемещается в другую секцию. - (Опционально) Кнопка "Удалить выполненные" для очистки списка от завершённых задач. - Интерфейс должен быть понятным и аккуратным (минимально оформить стили).
- *Слайд 3:* Эскиз интерфейса: нарисовать блок-схему страницы – где поле ввода и кнопка, где список. Можно отметить элементы с ID, которые будут в коде, для ясности.
- *Слайд 4:* Распределение задач между учениками: - Ученик A (разработчик): верстка HTML, стили, JS-логика. - Ученик B (тестировщик): написание тестовых сценариев, проверка каждого шага разработки, фиксирование найденных багов, помощь в улучшении UX. - Работа под присмотром наставника, который помогает обоим и направляет.
- *Слайд 5:* План работ на занятие 11: 1. Создать проект (репозиторий, базовые файлы). 2. Сделать основу HTML (форма + контейнер для списка). 3. Добавить стили для базового расположения элементов. 4. Реализовать добавление задачи (JS). 5. Реализовать отметку выполнения задачи (JS, через событие на элементах списка). 6. (Опционально) Реализовать удаление выполненных или сохранение в localStorage. 7. Тестирование каждой функции по мере готовности. 8. Исправление багов/недочётов, найденных тестировщиком.
- *Слайд 6:* Снимки кода/скриншоты по ходу (если делались заранее или live-coding): например, показать код функции добавления:

```
function addTask(text) {
    const li = document.createElement('li');
    li.innerText = text;
    // ... добавить чекбокс или кнопку "выполнено"
    list.appendChild(li);
}
```

Объяснять, что происходит, как использовали DOM API.
- *Слайд 7:* Скриншот текущего состояния приложения в браузере (например, после добавления пары задач). Обсудить, всё ли соответствует требованиям, удобно ли выглядит. Тестировщик может предложить улучшения (например, автоматически очищать поле ввода после добавления задачи – если этого не сделали). Тогда разработчик вносит правку.
- *Слайд 8:* Интеграция GitHub: показать страничку репозитория проекта – что вот, мы ведём историю, вот коммиты «setup project», «add addTask feature», и т.д. Отметить, что благодаря Git можно отслеживать, кто какие изменения внёс.
- *Слайд 9:* Обсуждение и совместное решение проблем: если что-то не получается – например, код не работает как ожидалось – показать, как совместно отлаживать (тестировщик, хоть и не писал код, может посмотреть консоль на ошибки). Такая парная работа – полезный опыт.
- *Слайд 10:* Завершение первой части проекта: сделать пуш финальных изменений сегодняшнего дня на GitHub. Проверить, что все заявленные функции реализованы. Если что-то не успели – определить, что осталось доделать на следующий урок (например, стили дополировать или

дополнительную кнопку). Тестировщик к следующему разу допишет недостающие тест-кейсы. Разработчик может дома немного улучшить код (по согласованию с наставником).

Занятие 12 (28 августа 2025) – Мини-проект (часть 2): Тестирование, отладка и презентация результата

Цель: Завершить мини-проект: провести полноценное тестирование созданного приложения, исправить обнаруженные дефекты, при необходимости улучшить функциональность. Отработать навыки совместной командной работы: тестировщик формально проходит по тест-кейсам, разработчик оперативно вносит исправления через Git. В заключение – презентовать получившийся проект (коротко) и подвести итоги всего учебного курса, повторив ключевые знания.

Ключевые понятия и задачи:
- Проведение тестирования проекта: тестировщик использует ранее подготовленные тест-кейсы. Если что-то не документировано, составляет чек-лист основных функций (например: *добавление задачи – работает ли с обычным текстом? с пустой строкой? с очень длинной строкой?*; *отметка выполнения – задача визуально меняется? повторное нажатие снимает отметку?*; *удаление выполненных – действительно удаляет?*; *UI – текст кнопок понятен? верстка не ломается при разных размерах экрана?*; *кросс-браузерно – проверить хотя бы в двух браузерах*).
- Логирование результатов: при нахождении несоответствий тестировщик документирует баги (в простейшем виде – записывает). В учебном формате можно использовать GitHub Issues: завести issue с описанием проблемы, чтобы разработчик увидел. Это симуляция баг-трекера.
- Исправление дефектов: разработчик получает список найденных багов и пытается их исправить. Например, тестировщик нашёл, что можно добавить пустую задачу – баг; разработчик решает: добавить проверку в код (не позволять или показывать ошибку). Делает изменение, коммит с сообщением типа "Fix: disable adding empty tasks". Тестировщик проверяет снова – баг ушёл. Отмечает issue как закрытый.
- Рефакторинг и улучшение кода: если время позволяет, показать, как разработчик может немного улучшить структуру кода после того, как функциональность готова (например, вынести повторяющиеся части в функцию, переименовать переменные для ясности). Объяснить тестировщику, что легкий рефакторинг полезен и не должен менять поведение, поэтому не страшно – но на каждый рефакторинг желательно запускать регресс-тест (тестировщик пробегается по основным сценариям, чтобы убедиться, что ничего не сломалось).
- Финальное оформление: привести проект в презентабельный вид – например, добавить заголовок странице, краткую инструкцию по использованию в интерфейсе, проверить, что нигде нет сырых незаконченных элементов. Тестировщик проверяет UX – удобно ли все, понятны ли надписи кнопок, не нужно ли всплывающих подсказок. Маленькие UX-улучшения ещё можно внести.
- Документация: заполнить `README.md` на GitHub – написать, что за проект, какие функции, как запустить (например, "просто открыть index.html в браузере"). Это учит хорошему тону в разработке. Тестировщик/разработчик вместе могут это сделать.
- Презентация проекта: Each student (или оба вместе) рассказывают наставнику (и, возможно, приглашённым) о том, что они сделали. Например, разработчик показывает приложение в действии: добавляет задачи, отмечает, удаляет – демонстрирует функционал. Тестировщик рассказывает, как они проверяли, какие проблемы нашли и исправили. Это развивает навык коммуникации результатов.
- Подведение итогов всего курса: наставник возвращается к списку тем, коротко повторяет: *«За это лето вы прошли путь от основ HTML до совместной разработки проекта. Теперь вы знакомы с …»* – перечислить HTML, CSS, JS, основы API, DB, тестирование, Git. Упомянуть, что освоено лишь

введение, но заложен фундамент для дальнейшего обучения. Дать рекомендации: куда двигаться дальше (например, для будущего разработчика – углубиться в JavaScript, изучить фреймворки; для тестировщика – изучить автоматизацию тестирования, например Selenium или Postman тест-скрипты, продолжить с SQL и теория тест-дизайна глубже, возможно получить сертификат ISTQB Foundation).
- Обратная связь: спросить учеников, что было самым интересным, что самым сложным. Ответить на оставшиеся вопросы. Поздравить с окончанием учебного плана.

Материалы и слайды:
- *Слайд 1:* План занятия: 1) Тестирование проекта, 2) Исправление ошибок, 3) Завершение и презентация, 4) Итоги курса.
- *Слайд 2:* Таблица/список тест-кейсов (подготовленный тестировщиком). Например:
TC1: Добавление задачи – **Шаги:** ввести "Купить молоко", нажать "Добавить". **Ожидаемо:** задача "Купить молоко" появляется в списке.
TC2: Добавление пустой – **Шаги:** ничего не вводя, нажать "Добавить". **Ожидаемо:** задача не добавляется, возможно, кнопка неактивна или появляется сообщение об ошибке.
... и так далее.
Тестировщик отмечает галочкой прошедшие, крестиком провалившиеся тесты. Эти результаты показываются.
- *Слайд 3:* Список найденных багов (если были): например: - BUG1: Добавляется пустая задача (Expected: не должно добавляться). - BUG2: Если задача очень длинная, вылезает за границы блока (верстка ломается). - BUG3: Кнопка "Удалить выполненные" не отключается, когда нет выполненных задач (не критично, но улучшение UX).
Эти пункты – для исправления.
- *Слайд 4:* Исправленные моменты: для каждого баг-репорта – отметить "исправлено" или "решено". Можно демонстрировать сразу: например, пустые задачи – теперь кнопка disabled, пока нет текста (показать, как кнопка становится активной только при вводе текста). Длинный текст – теперь в CSS добавлено свойство `word-wrap: break-word;`, текст переносится нормально (показать до/после).
- *Слайд 5:* Финальный вид приложения – скриншот или живой показ готовой страницы. С парой примеров задач, продемонстрировать все функции работают.
- *Слайд 6:* Презентация от учеников: дать им слово описать проект. Можно оформить как небольшую структуру: **Цель проекта**, **Функциональность**, **Стек технологий** (HTML, CSS, JS), **Кто что делал**, **С какими трудностями столкнулись** (и как решили). Это учит их структурировать рассказ о проделанной работе.
- *Слайд 7:* Сертификат/похвала (не обязателен слайд, но морально): "Молодцы! Теперь вы знакомы с ..." – перечислить ключевые навыки, которые получили за курс, и поздравление с успешным окончанием учебного плана.
- *Слайд 8:* Рекомендации: список ресурсов для продолжения обучения: - Онлайн-курсы (например, HTML Academy, freeCodeCamp для фронтенда; Stepik для тестирования). - Книги: "Изучаем HTML/CSS" или "JavaScript для детей" – для закрепления, "Testing Dot Com" или стандарт ISTQB syllabus – для тестирования теории. - Практиковаться на пет-проектах, может быть совместно продолжить улучшать этот проект (например, прикрутить backend в будущем, или автоматизировать тесты).
- *Слайд 9:* Обратная связь: попросить учеников поделиться впечатлениями о курсе, что было наиболее полезно. Наставник даёт напутственное слово, подбадривает на дальнейшее обучение.

Итого: Учебный план выполнен, ученики получили целостное представление о веб-разработке и тестировании, выполнили практический проект, приобрели базовые навыки работы в команде и с инструментами разработки [15] [13]. Поздравляем с окончанием курса и желаем успехов в дальнейшем пути в IT!

---

1 Что такое клиент-серверная архитектура простыми словами — testengineer.ru
https://testengineer.ru/client-servernaya-arhitektura/

2 3 5 6 13 14 15 Курсы IT для начинающих ≡ Подготовительные курсы IT | PASV
https://pasv.us/ru/course/essential

4 7 Бесплатный курс «Основы веб-разработки» от Хекслета: начните программировать с нуля!
https://ru.hexlet.io/programs/web-development-free

8 9 10 API для чайников: что это, как работает и зачем нужно | Университет СИНЕРГИЯ
https://synergy.ru/akademiya/programming/chto_takoe_api_prostyimi_slovami_i_kak_s_nim_rabotat

11 12 20 бесплатных курсов обучения на тестировщика (QA-инженера) в 2025 году
https://kata.academy/blog/qa/20-besplatnyh-kursov-obucheniya-na-testirovshchika-qa-inzhenera