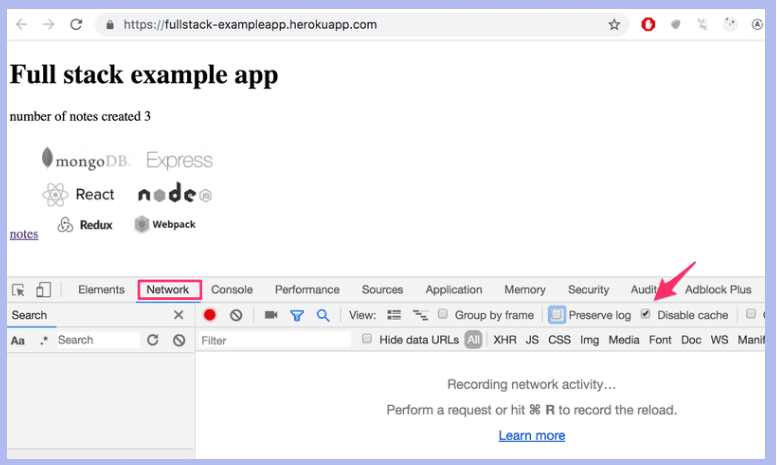
**PART 0**

**b. Fundamentals of Web apps**

Прежде чем мы начнем программировать, мы рассмотрим некоторые принципы веб-разработки, изучив пример приложения на<https://studies.cs.helsinki.fi/exampleapp>.

Откройте [пример приложения](https://studies.cs.helsinki.fi/exampleapp) в браузере.

Консоль выглядит так:



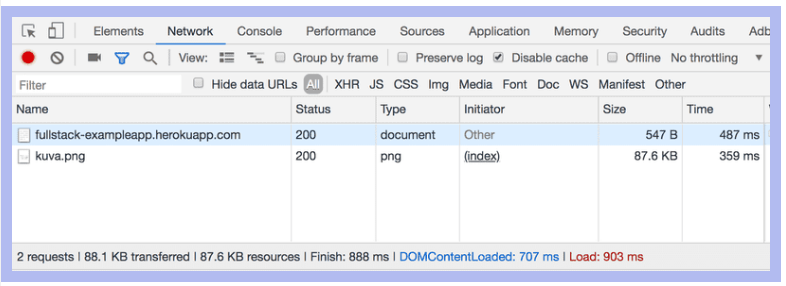
Убедитесь, что вкладка « *Network* » открыта, и установите флажок « *Disable cache* », как показано на рисунке. *Preserve log* также может быть полезным: он сохраняет log, напечатанные приложением, когда страница перезагружается.

1. **HTTP GET**

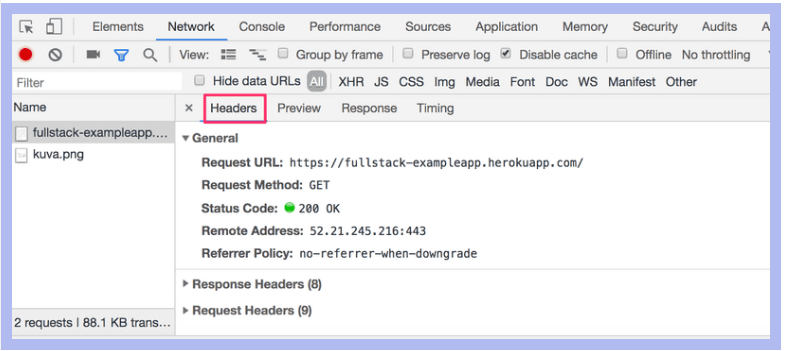
Сервер и веб-браузер взаимодействуют друг с другом по протоколу [HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP) . На вкладке « *Network* » показано, как взаимодействуют браузер и сервер.

Когда вы перезагрузите страницу, консоль покажет, что произошло два события:

* Браузер загрузил содержимое страницы *Studies.cs.helsinki.fi/exampleapp* с сервера.
* И скачал картинку *kuva.png*

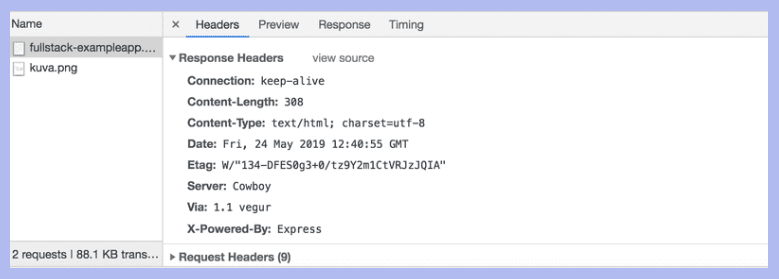


При нажатии на первое событие открывается дополнительная информация о том, что происходит:



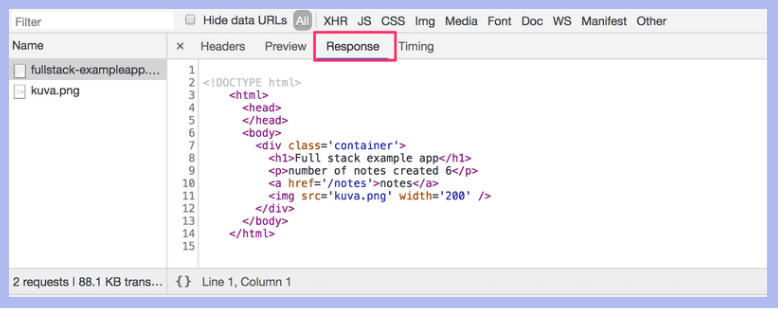
Верхняя часть *General* показывает, что браузер сделал запрос по адресу [*https://studies.cs.helsinki.fi/exampleapp*](https://studies.cs.helsinki.fi/exampleapp)  с использованием метода [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) , и что запрос был выполнен успешно, так как ответ сервера имел [код состояния](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) 200.

Запрос и ответ сервера имеют несколько [headers](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) :



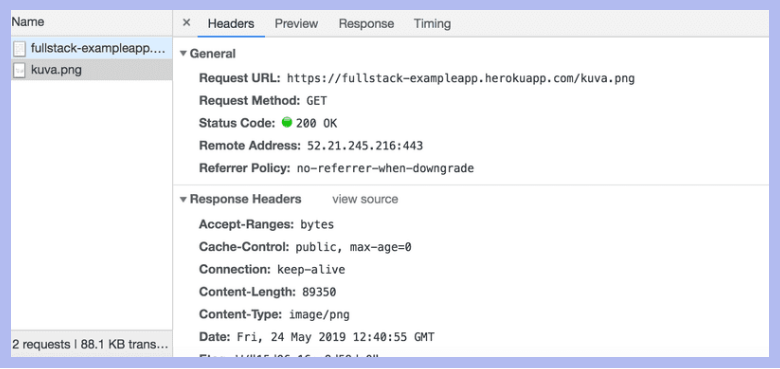
Response Headers сообщают нам, например, размер ответа в байтах и ​​точное время ответа. Важный заголовок [Content-Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type) сообщает нам, что ответ представляет собой текстовый файл в формате [utf-8](https://en.wikipedia.org/wiki/UTF-8) , содержимое которого отформатировано с помощью HTML. Таким образом, браузер узнает, что ответ является обычной [HTML](https://en.wikipedia.org/wiki/HTML) -страницей, и отображает его в браузере «как веб-страницу».

Вкладка *Response* показывает данные ответа, обычную HTML-страницу. Раздел *body* определяет структуру отображаемой на экране страницы:



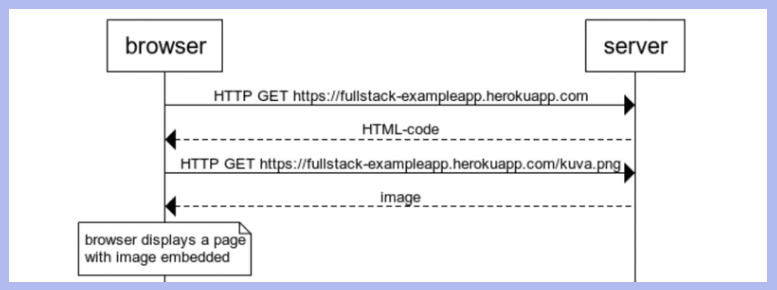
Страница содержит элемент [div](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) , который, в свою очередь, содержит заголовок, ссылку на *заметки* страницы и тег [img](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img) , а также отображает количество созданных заметок.

Из-за тега img браузер делает второй *HTTP-запрос* , чтобы получить изображение *kuva.png* с сервера. Детали запроса следующие:



Запрос был сделан на адрес <https://studies.cs.helsinki.fi/exampleapp/kuva.png> и имеет тип HTTP GET. Заголовки ответа говорят нам, что размер ответа составляет 89350 байт, а его [Content-type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type) — *image/png* , поэтому это изображение в формате png. Браузер использует эту информацию для правильного отображения изображения на экране.

Цепочка событий, вызванных открытием страницы <https://studies.cs.helsinki.fi/exampleapp> в браузере, образует следующую [диаграмму последовательности](https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/) :



Сначала браузер отправляет HTTP-запрос GET на сервер, чтобы получить HTML-код страницы. Тег *img* в HTML предлагает браузеру загрузить изображение *kuva.png* . Браузер отображает HTML-страницу и изображение на экране.

Хотя это трудно заметить, HTML-страница начинает отображаться до того, как изображение будет получено с сервера.

1. **Traditional web applications**

Домашняя страница примера приложения работает как *традиционное веб-приложение* . При входе на страницу браузер загружает с сервера HTML-документ с подробным описанием структуры и текстового содержимого страницы.

Сервер каким-то образом сформировал этот документ. Документ может быть *статическим* текстовым файлом, сохраненным в каталоге сервера. Сервер также может *динамически* формировать HTML-документы в соответствии с кодом приложения, используя, например, данные из базы данных. HTML-код примера приложения сформирован динамически, поскольку содержит информацию о количестве созданных заметок.

HTML-код главной страницы выглядит следующим образом:



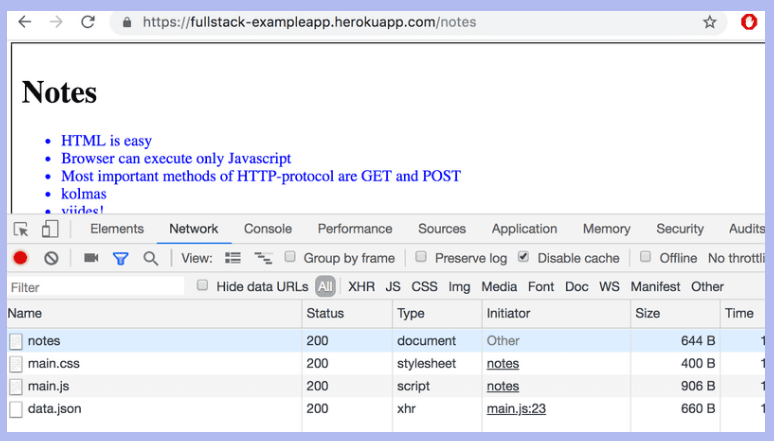
Содержимое HTML-страницы сохранено в виде строки шаблона или строки, которая позволяет вычислять, например, переменные внутри нее. Динамически изменяющаяся часть главной страницы, количество сохраненных заметок (в коде noteCount ), заменяется текущим количеством заметок (в коде notes.length ) в строке шаблона.

В традиционных веб-приложениях браузер «тупой». Он только извлекает данные HTML с сервера, и вся логика приложения находится на сервере.

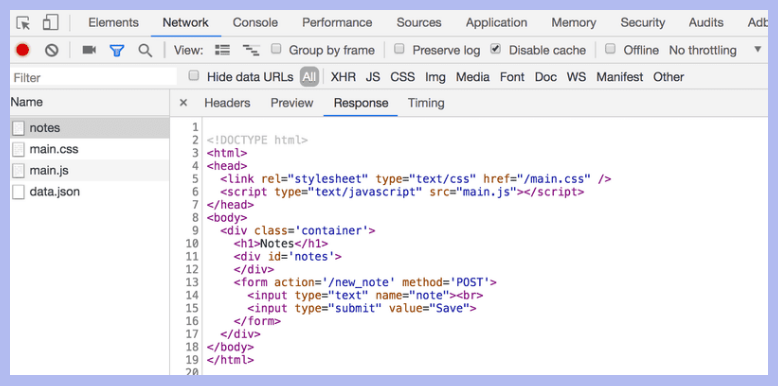
В примере используется библиотека [Express](https://expressjs.com/) с [Node.js](https://nodejs.org/en/). Этот курс будет использовать Node.js и Express для создания веб-серверов.

1. **Running application logic in the browser**

Очистите консоль, щелкнув символ 🚫 или набрав в консоли clear(). Теперь при переходе на страницу [notes](https://studies.cs.helsinki.fi/exampleapp/notes) браузер делает 4 HTTP-запроса:



Все запросы имеют *разные* типы. Тип первого запроса — *document* . Это HTML-код страницы, и выглядит он следующим образом:



Когда мы сравниваем страницу, отображаемую в браузере, и HTML-код, возвращенный сервером, мы замечаем, что код не содержит списка заметок. Раздел [head](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/head) в HTML содержит тег [script](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) , который заставляет браузер получать файл JavaScript с именем *main.js.*

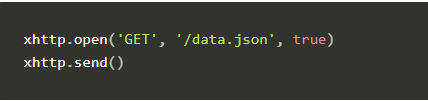
Код JavaScript выглядит следующим образом:



Некоторые могут задаться вопросом, почему вместо современной выборки используется xhttp-object. Это связано с тем, что пока вообще не хочется вдаваться в промисы, а код играет второстепенную роль в этой части. Мы вернемся к современным способам выполнения запросов к серверу во второй части.

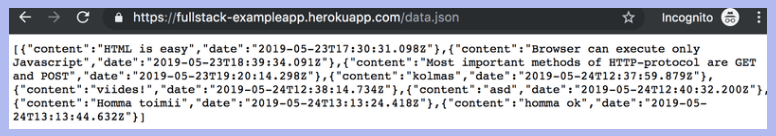
Сразу после получения тега *script* браузер начинает выполнять код.

Последние две строки указывают браузеру выполнить HTTP-запрос GET к адресу сервера */data.json* :



Это самый нижний запрос, отображаемый на вкладке «Network».

Мы можем попробовать перейти по адресу <https://studies.cs.helsinki.fi/exampleapp/data.json> прямо из браузера:

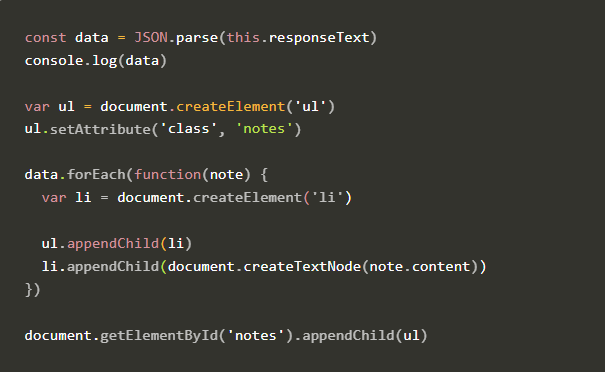


Там мы находим заметки в [формате JSON](https://en.wikipedia.org/wiki/JSON) «необработанные данные – raw data». По умолчанию браузеры на основе Chromium не очень хорошо отображают данные JSON. Плагины могут использоваться для обработки форматирования. Установите, например, [JSONVue](https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc) в Chrome и перезагрузите страницу. Данные теперь красиво отформатированы:

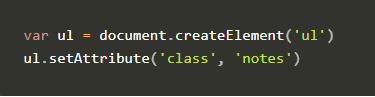


Таким образом, код JavaScript на странице заметок выше загружает JSON-данные, содержащие заметки, и формирует маркированный список из содержимого заметок:

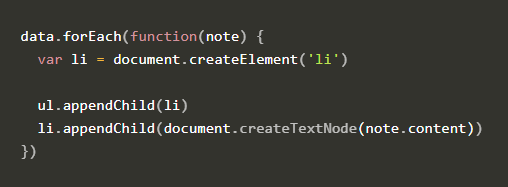
Это делается с помощью следующего кода:



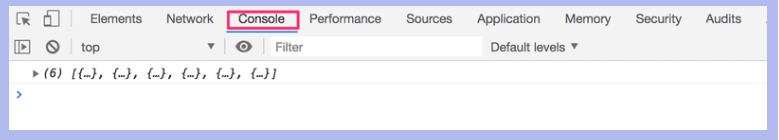
Сначала код создает неупорядоченный список с тегом [ul](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ul)



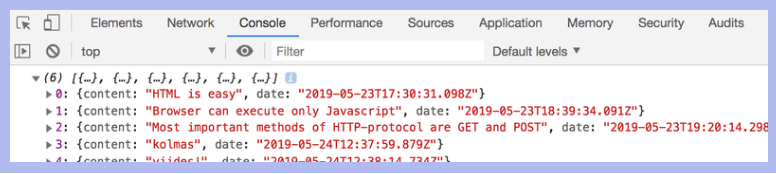
Только поле content каждой заметки становится содержимым li-тега. date, найденные в данных, здесь ни для чего не используются.



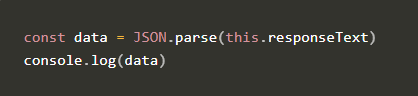
Теперь откройте вкладку « *console* » в консоли разработчика:



Нажав на маленький треугольник в начале строки, вы можете развернуть текст в консоли.



Этот вывод на консоль вызван командой console.log в коде:

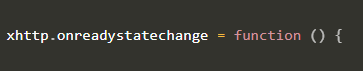


Итак, получив данные с сервера, код выводит их в консоль.

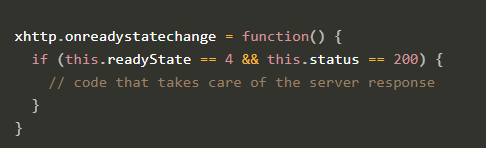
## **Event handlers and Callback functions**



Запрос к серверу отправляется в последней строке, но код для обработки ответа можно найти выше. В чем дело?



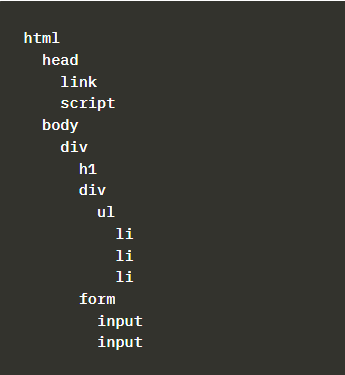
В этой строке определяется *event handler -обработчик* события *onreadystatechange* для объекта xhttp , выполняющего запрос. Когда состояние объекта изменяется, браузер вызывает функцию обработчика событий. Код функции проверяет, что [readyState](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState) равно 4 (что изображает ситуацию *Операция завершена* ) и что код состояния HTTP ответа равен 200.



Механизм вызова обработчиков событий очень распространен в JavaScript. Функции обработчика событий называются функциями [callback-обратного вызова](https://developer.mozilla.org/en-US/docs/Glossary/Callback_function) . Код приложения не сам вызывает функции, а среда выполнения - браузер вызывает функцию в соответствующее время, когда произошло *событие - event*.

## **Document Object Model or DOM**

Мы можем думать о HTML-страницах как о неявных древовидных структурах.



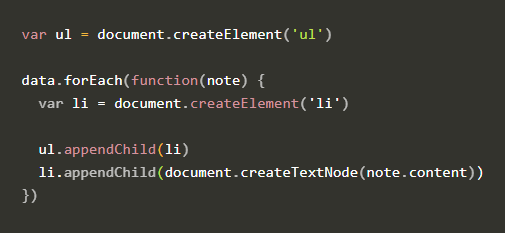
Такую же древовидную структуру можно увидеть на вкладке консоли *Elements*.



Объектная модель документа, или [DOM](https://en.wikipedia.org/wiki/Document_Object_Model) , представляет собой интерфейс прикладного программирования ( *API -* Application Programming Interface ), который позволяет программно модифицировать *деревья элементов,* соответствующие веб-страницам.

Код JavaScript, представленный в предыдущей главе, использовал DOM-API для добавления списка заметок на страницу.

Следующий код создает новый узел для переменной ul и добавляет к нему несколько дочерних узлов:

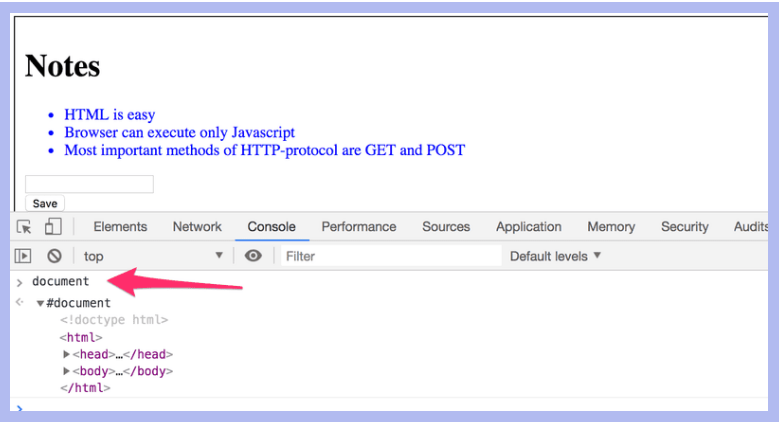


Наконец, древовидная ветвь переменной ul подключается к соответствующему месту в дереве HTML всей страницы:



## **Manipulating the document-object from console**

Самый верхний узел дерева DOM документа HTML называется объектом document. Мы можем выполнять различные операции на веб-странице с помощью DOM-API. Вы можете получить доступ к объекту document , введя document на вкладке Консоль:

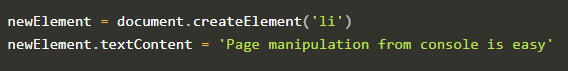


Давайте добавим новую заметку на страницу из консоли.

Сначала мы получим список заметок со страницы. Список находится в первом ul-элементе страницы:

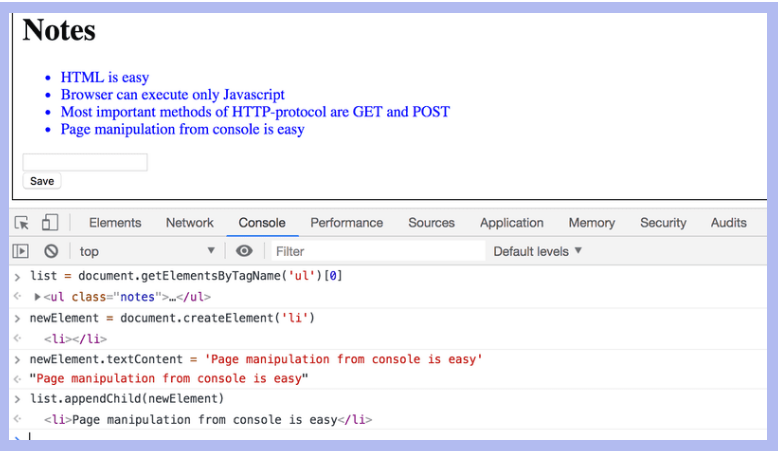


Затем создайте новый элемент li и добавьте к нему текстовое содержимое:



И добавьте в список новый элемент li:





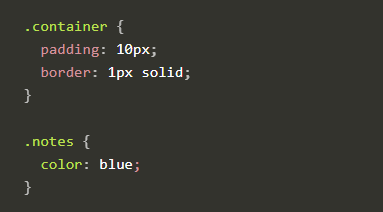
Несмотря на то, что страница обновляется в вашем браузере, изменения не являются постоянными. При перезагрузке страницы новая заметка исчезнет, ​​так как изменения не были отправлены на сервер. Код JavaScript, который извлекает браузер, всегда будет создавать список заметок на основе JSON-данных с адреса <https://studies.cs.helsinki.fi/exampleapp/data.json> .

## **CSS**

Элемент *head* HTML-кода страницы Notes содержит тег [link](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) , который определяет, что браузер должен получить таблицу стилей [CSS с адреса](https://developer.mozilla.org/en-US/docs/Web/CSS)[main.css](https://studies.cs.helsinki.fi/exampleapp/main.css) .

Каскадные таблицы стилей или CSS — это язык таблиц стилей, используемый для определения внешнего вида веб-страниц.

Полученный CSS-файл выглядит следующим образом:

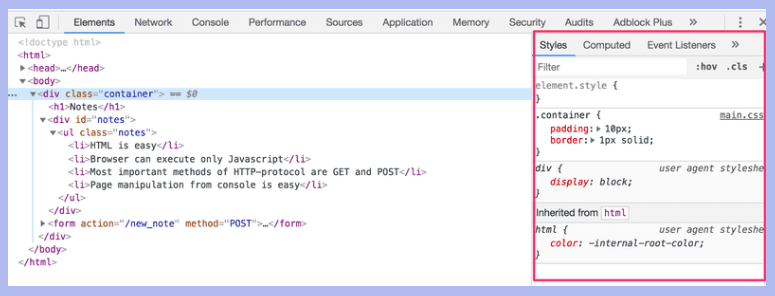


Файл определяет два [селектора классов](https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors) . Они используются для выбора определенных частей страницы и для определения правил стилизации для их стилизации.

Определение селектора класса всегда начинается с точки и содержит имя класса.

Классы — это [атрибуты](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class) , которые можно добавлять к элементам HTML.

Атрибуты CSS можно просмотреть на вкладке *элементов* консоли:



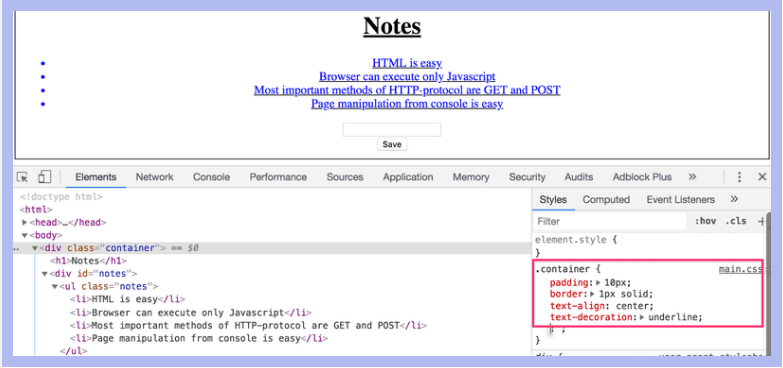
Самый внешний элемент *div имеет* класс container . Элемент *ul* , содержащий список заметок, имеет класс *notes* .

Правило CSS определяет, что элементы с классом *container будут обведены*[рамкой - border](https://developer.mozilla.org/en-US/docs/Web/CSS/border) шириной в один пиксель . Он также устанавливает 10-пиксельный [отступ - padding](https://developer.mozilla.org/en-US/docs/Web/CSS/padding). Это добавляет пустое пространство между содержимым элемента и границей.

Второе правило CSS устанавливает синий цвет текста заметок.

Элементы HTML также могут иметь другие атрибуты помимо классов. Элемент *div* , содержащий заметки, имеет атрибут [id](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/id) . Код JavaScript использует идентификатор id для поиска элемента.

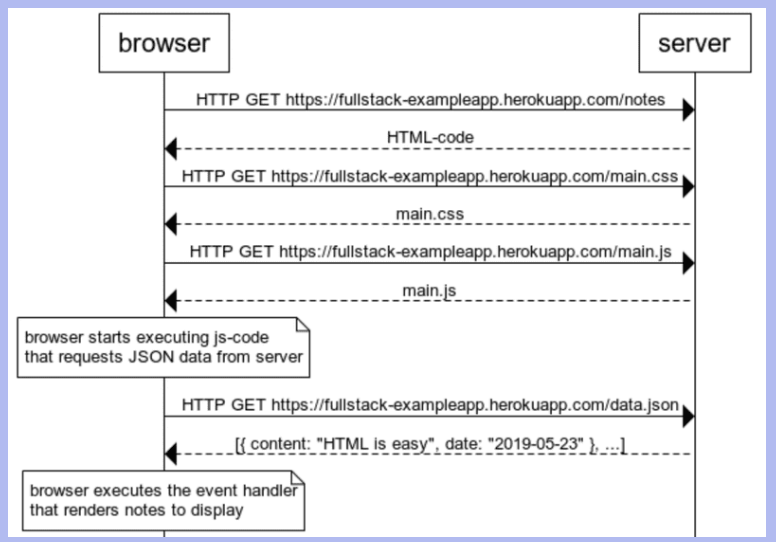
Вкладку « *Elements* » консоли можно использовать для изменения стилей элементов.



Изменения, сделанные на консоли, не будут постоянными. Если вы хотите внести долгосрочные изменения, их необходимо сохранить в таблице стилей CSS на сервере.

## **Loading a page containing JavaScript - review**

Давайте рассмотрим, что происходит, когда в браузере открывается страница [https://studies.cs.helsinki.fi/exampleapp/notes .](https://studies.cs.helsinki.fi/exampleapp/notes)



* Браузер получает HTML-код, определяющий содержимое и структуру страницы, с сервера с помощью HTTP-запроса GET.
* Ссылки в коде HTML заставляют браузер также получать таблицу стилей CSS *main.css* ...
* ...и файл кода JavaScript *main.js*
* Браузер выполняет код JavaScript. Код делает HTTP-запрос GET по адресу <https://studies.cs.helsinki.fi/exampleapp/data.json> , который возвращает заметки в виде данных JSON.
* Когда данные извлечены, браузер выполняет *обработчик событий* , который отображает заметки на странице с помощью DOM-API.

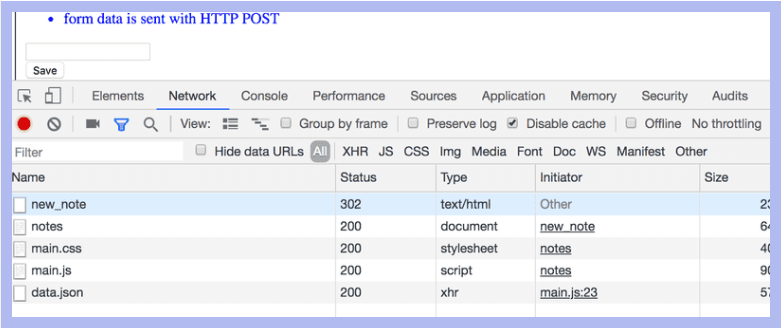
## **Forms and HTTP POST**

Далее давайте рассмотрим, как делается добавление новой заметки.

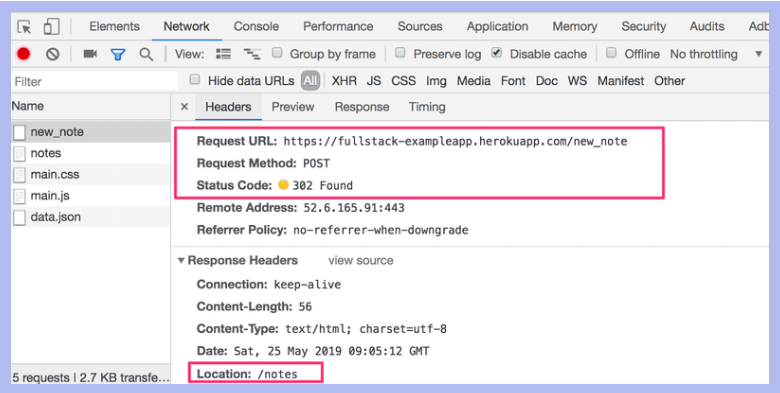
Страница Notes содержит элемент [form](https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Your_first_HTML_form).



Когда кнопка в форме нажата, браузер отправит пользовательский ввод на сервер. Откроем вкладку *Network* и посмотрим, как выглядит отправка формы:



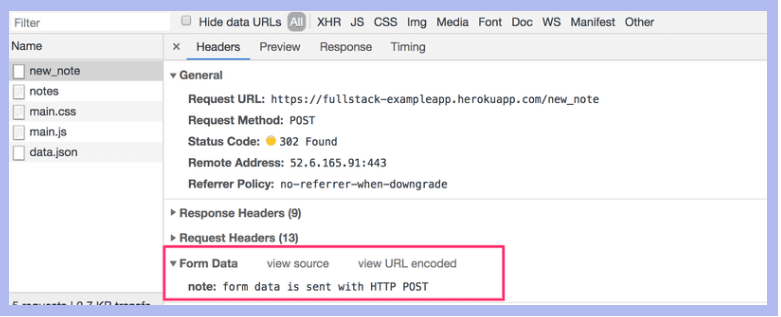
Удивительно, но отправка формы вызывает не менее *пяти* HTTP-запросов. Первый — это событие отправки формы. Давайте увеличим его:



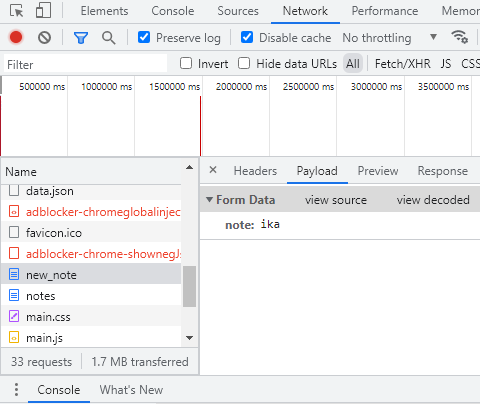
Это [HTTP-запрос POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) на адрес сервера *new\_note* . Сервер отвечает кодом состояния HTTP 302. Это [перенаправление URL](https://en.wikipedia.org/wiki/URL_redirection) -адреса , с помощью которого сервер запрашивает у браузера новый HTTP-запрос GET на адрес, указанный в заголовке *Location* —  к адресу notes.

Итак, браузер перезагружает страницу Notes. Повторная загрузка вызывает еще три HTTP-запроса: получение таблицы стилей (main.css), кода JavaScript (main.js) и необработанных данных (data.json).

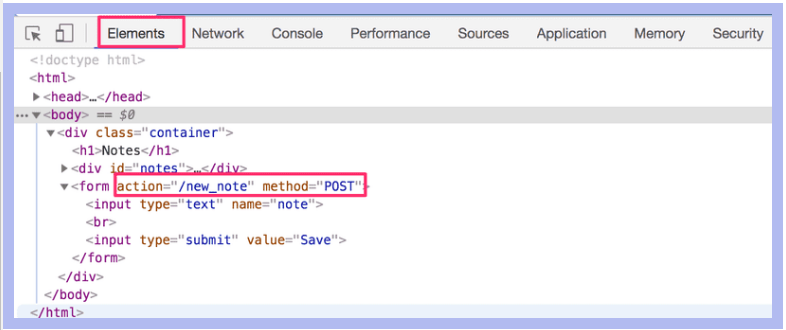
На вкладке сети также отображаются данные, отправленные с помощью формы:



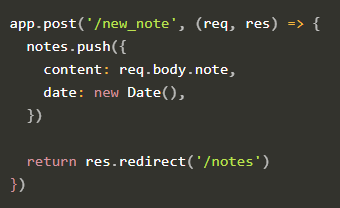
 в более новых версиях Chrome раскрывающийся список «form data» находится на новой вкладке «Payload», расположенной справа от вкладки «Headers».



Тег Form имеет атрибуты *action* и *method* , которые определяют, что отправка формы выполняется как запрос HTTP POST на адрес *new\_note* .



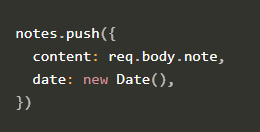
Код на сервере, отвечающий за POST-запрос, довольно прост (NB: этот код находится на сервере, а не в коде JavaScript, полученном браузером):



Данные отправляются в виде [тела-body](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) POST-запроса.

Сервер может получить доступ к данным, обратившись к полю req.body объекта запроса req .

Сервер создает новый объект заметки и добавляет его в массив с именем notes.



Объекты Note имеют два поля: *content* , содержащее фактическое содержание заметки, и *date* , содержащее дату и время создания заметки.

Сервер не сохраняет новые заметки в базу данных, поэтому новые заметки исчезают при перезапуске сервера.

## **AJAX**

Страница Notes приложения соответствует стилю веб-разработки начала 90-х годов и использует «Ajax». Таким образом, он находится на гребне волны веб-технологий начала 2000-х годов.

[AJAX](https://en.wikipedia.org/wiki/Ajax_(programming)) (асинхронный JavaScript и XML) — это термин, введенный в феврале 2005 года в связи с достижениями в технологии браузеров для описания нового революционного подхода, позволяющего загружать содержимое на веб-страницы с помощью JavaScript, включенного в HTML, без необходимости повторного рендеринга страници.

До эпохи AJAX все веб-страницы работали как [традиционные веб-приложения](https://fullstackopen.com/en/part0/fundamentals_of_web_apps#traditional-web-applications) , которые мы видели ранее в этой главе. Все данные, показанные на странице, были получены с помощью HTML-кода, сгенерированного сервером.

Страница Notes использует AJAX для получения данных заметок. Отправка формы по-прежнему использует традиционный механизм отправки веб-форм.

URL-адреса приложений отражают старые беззаботные времена. Данные JSON извлекаются из URL -адреса <https://studies.cs.helsinki.fi/exampleapp/data.json> , а новые заметки отправляются по URL-адресу <https://studies.cs.helsinki.fi/exampleapp/new_note> .  
В настоящее время такие URL-адреса не считаются приемлемыми, поскольку они не соответствуют общепризнанным соглашениям [RESTful](https://en.wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services) API, которые мы рассмотрим подробнее в [части 3](https://fullstackopen.com/en/part3) .

Вещь, называемая AJAX, сейчас настолько обыденна, что считается само собой разумеющейся. Термин канул в лету, а новое поколение о нем даже не слышало.

## **Single page app**

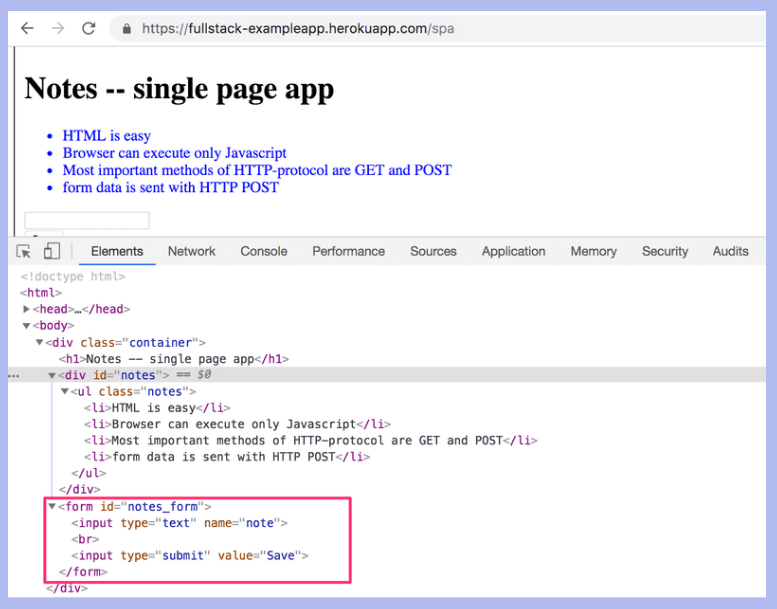
В нашем примере домашняя страница работает как традиционная веб-страница: вся логика находится на сервере, а браузер только отображает HTML в соответствии с инструкциями.

Страница «Notes» возлагает часть ответственности за создание HTML-кода для существующих заметок на браузер. Браузер решает эту задачу, выполняя код JavaScript, полученный с сервера. Код получает заметки с сервера в виде JSON-данных и добавляет элементы HTML для отображения заметок на странице с помощью [DOM-API](https://fullstackopen.com/en/part0/fundamentals_of_web_apps#document-object-model-or-dom).

В последние годы появился стиль [одностраничных приложений](https://en.wikipedia.org/wiki/Single-page_application) (SPA – Single-page application) для создания веб-приложений. Веб-сайты в стиле SPA не извлекают все свои страницы отдельно с сервера, как это делает наше примерное приложение, а вместо этого содержат только одну HTML-страницу, полученную с сервера, содержимое которой обрабатывается с помощью JavaScript, который выполняется в браузере.

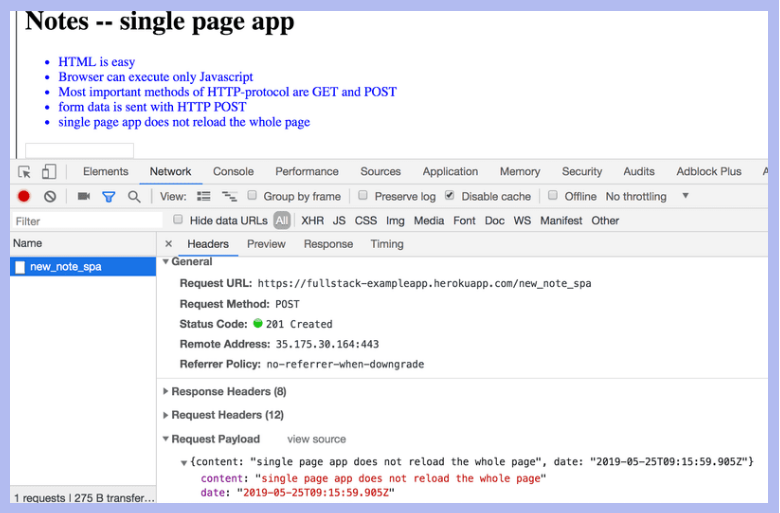
Страница Notes нашего приложения имеет некоторое сходство с приложениями в стиле SPA, но это еще не совсем так. Несмотря на то, что логика отображения заметок выполняется в браузере, на странице по-прежнему используется традиционный способ добавления новых заметок. Данные отправляются на сервер с отправкой формы, и сервер указывает браузеру перезагрузить страницу Notes с помощью *перенаправления* .

Одностраничную версию нашего примера приложения можно найти по адресу <https://studies.cs.helsinki.fi/exampleapp/spa> . На первый взгляд приложение выглядит точно так же, как и предыдущее. Код HTML почти идентичен, но файл JavaScript отличается ( *spa.js* ), и есть небольшое изменение в том, как определяется тег form:



form не имеет атрибутов *action* или *method* , определяющих, как и куда отправлять входные данные.

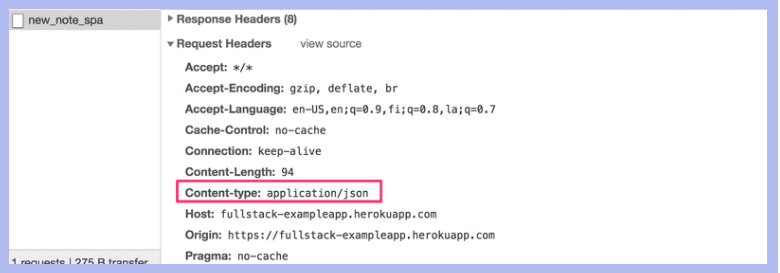
Откройте вкладку «*Network»* и очистите ее. Теперь, когда вы создаете новую заметку, вы заметите, что браузер отправляет на сервер только один запрос.



Запрос POST на адрес *new\_note\_spa* содержит новую заметку в виде JSON-данных, содержащих как содержимое заметки ( *content* ), так и метку времени ( *date* ):



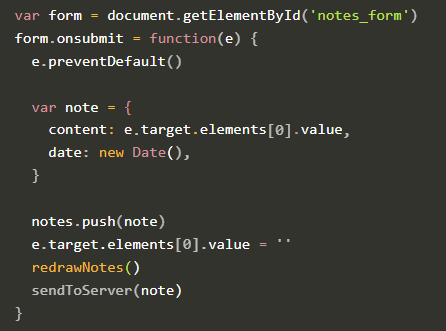
Заголовок *Content-Type* запроса сообщает серверу, что включенные данные представлены в формате JSON.



Без этого заголовка сервер не знал бы, как правильно анализировать данные.

Сервер отвечает кодом состояния [201 created](https://httpstatuses.com/201) . На этот раз сервер не запрашивает перенаправление, браузер остается на той же странице и больше не отправляет HTTP-запросов.

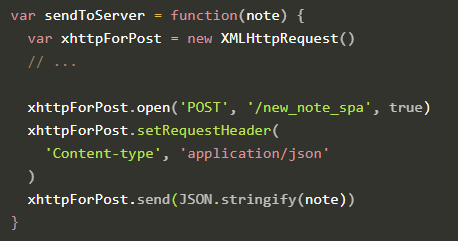
SPA-версия приложения не отправляет данные формы традиционным способом, а вместо этого использует код JavaScript, полученный с сервера. Мы немного рассмотрим этот код, хотя понимание всех его деталей пока не важно.



Команда document.getElementById('notes\_form') дает указание коду извлечь элемент формы со страницы и зарегистрировать *обработчик событий* для обработки события отправки формы. Обработчик событий немедленно вызывает метод e.preventDefault() , чтобы предотвратить обработку отправки формы по умолчанию. Метод по умолчанию отправляет данные на сервер и вызывает новый запрос GET, чего мы не хотим.

Затем обработчик событий создает новую заметку, добавляет ее в список заметок с помощью команды notes.push(note) , перерисовывает список заметок на странице и отправляет новую заметку на сервер.

Код для отправки заметки на сервер выглядит следующим образом:



Код определяет, что данные должны быть отправлены с запросом HTTP POST, а тип данных должен быть JSON. Тип данных определяется заголовком *Content-type .*Затем данные отправляются в виде JSON-строки.

## **JavaScript-libraries**

Пример приложения создан с помощью так называемого [ванильного JavaScript](https://www.freecodecamp.org/news/is-vanilla-javascript-worth-learning-absolutely-c2c67140ac34/) , с использованием только DOM-API и JavaScript для управления структурой страниц.

Вместо использования только JavaScript и DOM-API для управления страницами часто используются различные библиотеки, содержащие инструменты, с которыми проще работать по сравнению с DOM-API. Одной из таких библиотек является очень популярная библиотека [jQuery](https://jquery.com/) .

jQuery был разработан еще тогда, когда веб-приложения в основном следовали традиционному стилю создания сервером HTML-страниц, функциональность которых была расширена на стороне браузера с использованием JavaScript, написанного с помощью jQuery. Одной из причин успеха jQuery была его так называемая кроссбраузерность. Библиотека работала независимо от браузера или компании, которая ее создала, поэтому не было необходимости в решениях для конкретного браузера. В настоящее время использование jQuery не так оправдано, учитывая развитие JavaScript, и самые популярные браузеры обычно хорошо поддерживают базовые функции.

Появление одностраничных приложений принесло несколько более «современных» способов веб-разработки, чем jQuery. Фаворитом первой волны разработчиков был [BackboneJS](http://backbonejs.org/) . После своего [запуска](https://github.com/angular/angular.js/blob/master/CHANGELOG.md#100-temporal-domination-2012-06-13) в 2012 году [AngularJS](https://angularjs.org/) от Google быстро стал практически стандартом де-факто современной веб-разработки.

Однако популярность Angular резко упала в октябре 2014 года после того, как [команда Angular объявила, что поддержка версии 1 прекратится](https://web.archive.org/web/20151208002550/https:/jaxenter.com/angular-2-0-announcement-backfires-112127.html) , а Angular 2 не будет обратно совместим с первой версией. Angular 2 и более новые версии не получили слишком теплого приема.

В настоящее время наиболее популярным инструментом для реализации логики веб-приложений на стороне браузера является библиотека Facebook [React .](https://reactjs.org/)В ходе этого курса мы познакомимся с React и библиотекой [Redux](https://github.com/reactjs/redux) , которые часто используются вместе.

Статус React кажется сильным, но мир JavaScript постоянно меняется. Например, в последнее время некоторый интерес вызывает новичок — [VueJS .](https://vuejs.org/)

## **Full stack web development**

Что означает название курса *Full stack web development* ? Полный стек — это модное слово, о котором все говорят, но никто толком не знает, что оно означает. Или, по крайней мере, не существует согласованного определения этого термина.

Практически все веб-приложения имеют (как минимум) два «слоя»: браузер, будучи ближе к конечному пользователю, является верхним слоем, а сервер — нижним. Часто под сервером находится уровень базы данных. Поэтому мы можем думать об *архитектуре* веб-приложения как о некоем *наборе* слоев.

Часто мы также говорим о [фронтенде и бэкенде](https://en.wikipedia.org/wiki/Front_and_back_ends) . Браузер — это внешний интерфейс, а JavaScript, работающий в браузере, — это внешний код. Сервер, с другой стороны, является бэкэндом.

В контексте этого курса полнофункциональная веб-разработка означает, что мы фокусируемся на всех частях приложения: внешнем интерфейсе, серверной части и базе данных. Иногда программное обеспечение на сервере и его операционная система рассматриваются как части стека, но мы не будем вдаваться в подробности.

Мы будем кодировать серверную часть с помощью JavaScript, используя среду выполнения [Node.js.](https://nodejs.org/en/)Использование одного и того же языка программирования на нескольких уровнях стека открывает для веб-разработки с полным стеком совершенно новое измерение. Однако для веб-разработки с полным стеком не требуется использовать один и тот же язык программирования (JavaScript) для всех слоев стека.

Раньше разработчики чаще специализировались на одном уровне стека, например, на бэкенде. Технологии на бэкэнде и фронтенде сильно отличались. С тенденцией полного стека для разработчиков стало обычным делом владеть всеми уровнями приложения и базы данных. Часто разработчики полного стека также должны обладать достаточными навыками настройки и администрирования для работы своего приложения, например, в облаке.

## **JavaScript fatigue**

усталость от Javascript - Веб-разработка с полным стеком сложна во многих отношениях. Все происходит во многих местах одновременно, и отладка немного сложнее, чем с обычными настольными приложениями. JavaScript не всегда работает так, как вы ожидаете (по сравнению со многими другими языками), а асинхронный способ работы его среды выполнения вызывает всевозможные проблемы. Общение в Интернете требует знания протокола HTTP. Нужно также заниматься базами данных и администрированием и настройкой серверов. Также было бы неплохо знать достаточно CSS, чтобы сделать приложения хоть сколько-нибудь презентабельными.

Мир JavaScript быстро развивается, что приносит с собой ряд проблем. Инструменты, библиотеки и сам язык находятся в постоянном развитии. Некоторые начинают уставать от постоянных изменений и придумали для этого термин: JavaScript fatigue . См. [Как справиться с усталостью JavaScript на auth0](https://auth0.com/blog/how-to-manage-javascript-fatigue/" \t "_blank) или [усталостью JavaScript на Medium](https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4) .

Во время этого курса вы сами будете страдать от усталости от JavaScript. К счастью для нас, есть несколько способов сгладить кривую обучения, и мы можем начать с кодирования вместо настройки. Мы не можем полностью избежать настройки, но мы можем весело продвигаться вперед в течение следующих нескольких недель, избегая при этом худшего из конфигурационных адов.

Ознакомьтесь с основами HTML, прочитав этот учебник от Mozilla: [Учебник по HTML](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics) .

Ознакомьтесь с основами CSS, прочитав этот учебник от Mozilla: [Учебник по CSS](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics) .

Узнайте об основах HTML-форм, прочитав учебник Mozilla [Ваша первая форма](https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Your_first_HTML_form) .

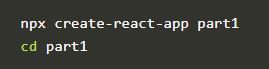
**PART 1**

## **Introduction to React**

Теперь мы начнем знакомиться с, пожалуй, самой важной темой этого курса, а именно с библиотекой [React](https://reactjs.org/) . Давайте начнем с создания простого приложения React, а также познакомимся с основными концепциями React.

Самый простой способ начать работу — это использовать инструмент под названием [create-react-app](https://github.com/facebook/create-react-app) . Можно (но не обязательно) установить *create-react-app* на свой компьютер, если инструмент *npm* , который был установлен вместе с Node, имеет номер версии не ниже *5.3* .

Давайте создадим приложение с именем *part1* и перейдем в его каталог.

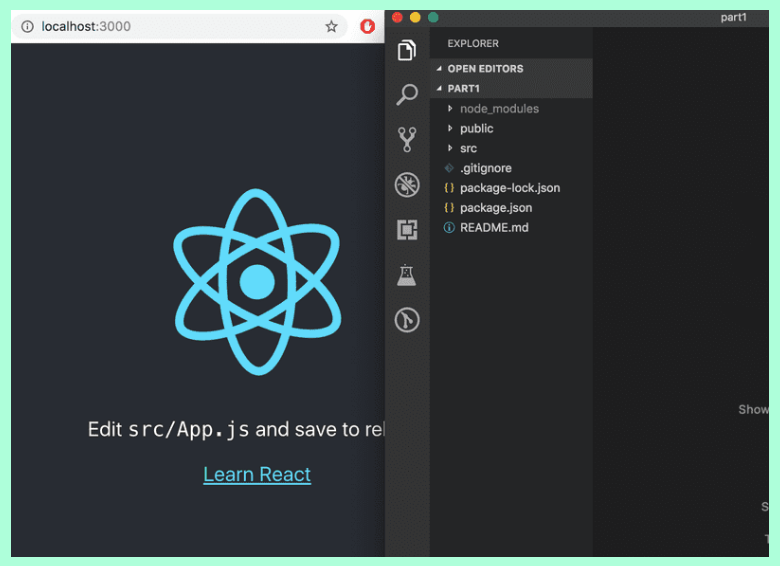


Приложение запускается следующим образом

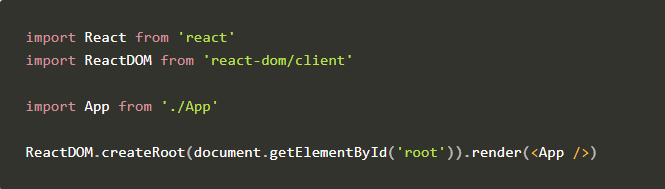


По умолчанию приложение работает на порту 3000 локального хоста с адресом [http://localhost:3000.](http://localhost:3000/)

Браузер по умолчанию должен запуститься автоматически. **Немедленно** откройте консоль браузера . Также откройте текстовый редактор, чтобы вы могли одновременно просматривать код и веб-страницу на экране:



Код приложения находится в папке *src .*Давайте упростим код по умолчанию, чтобы содержимое файла *index.js* выглядело так:

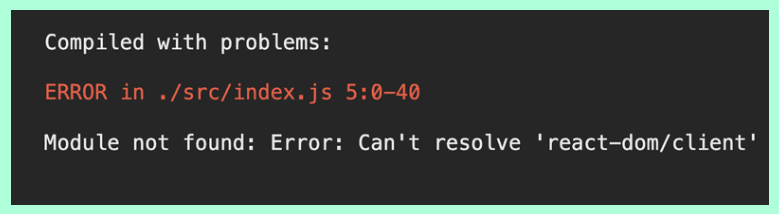


и файл *App.js* выглядит так



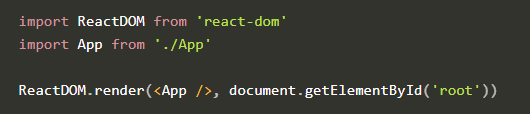
Файлы *App.css* , *App.test.js* , *index.css* , *logo.svg* , *setupTests.js* и *reportWebVitals.js* можно удалить, так как они сейчас не нужны в нашем приложении.

Если вы столкнулись со следующей ошибкой:



Затем по какой-то причине вы используете версию React старше текущей версии 18.

Исправление состоит в том, чтобы изменить *index.js* следующим образом:



Скорее всего, вам придется сделать то же самое для других ваших проектов.

Подробнее [см. здесь](https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps/#a-note-on-react-version) различиях версий.

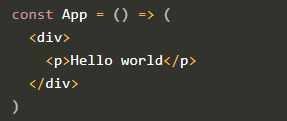
### Component

Файл *App.js* теперь определяет [компонент React](https://reactjs.org/docs/components-and-props.html) с именем *App* . Команда в последней строке файла *index.js* отображает свое содержимое в элемент *div* , определенный в файле *public/index.html* , имеющий значение *id* 'root'.



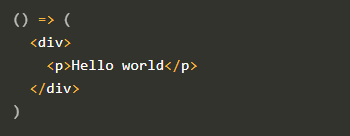
По умолчанию файл *public/index.html* не содержит HTML-разметки, видимой нам в браузере. Вы можете попробовать добавить HTML в файл. Однако при использовании React весь контент, который необходимо отобразить, обычно определяется как компоненты React.

Давайте подробнее рассмотрим код, определяющий компонент:



Как вы, наверное, догадались, компонент будет отображаться как *div* -тег, который обертывает *p* -тег, содержащий текст *Hello world* .

Технически компонент определяется как функция JavaScript. Ниже приведена функция (которая не получает никаких параметров):



Затем функция присваивается постоянной переменной *App* :



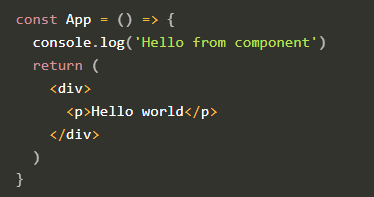
Есть несколько способов определить функции в JavaScript. Здесь мы будем использовать [стрелочные функции](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) , которые описаны в более новой версии JavaScript, известной как [ECMAScript 6](http://es6-features.org/#Constants) , также называемой ES6.

Поскольку функция состоит только из одного выражения, мы использовали сокращение, которое представляет этот фрагмент кода:



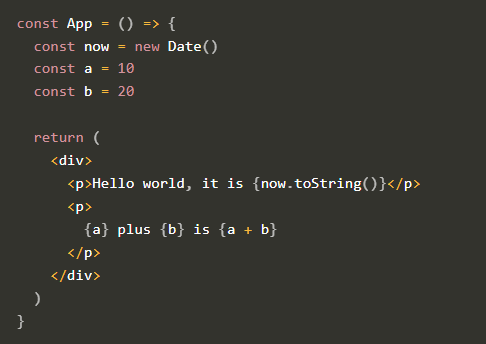
Другими словами, функция возвращает значение выражения.

Функция, определяющая компонент, может содержать любой код JavaScript. Измените свой компонент следующим образом и посмотрите, что произойдет в консоли:



Также возможно визуализировать динамическое содержимое внутри компонента.

Измените компонент следующим образом:



Любой код JavaScript в фигурных скобках оценивается, и результат этой оценки встраивается в определенное место в HTML, созданном компонентом.

### JSX

Похоже, что компоненты React возвращают HTML-разметку. Однако, это не так. компоненти React в основном написана с использованием [JSX](https://reactjs.org/docs/introducing-jsx.html) . Хотя JSX выглядит как HTML, на самом деле мы имеем дело со способом написания JavaScript. Под капотом JSX, возвращаемый компонентами React, компилируется в JavaScript.

После компиляции наше приложение выглядит так:



На практике JSX очень похож на HTML с тем отличием, что с помощью JSX вы можете легко встраивать динамический контент, написав соответствующий код JavaScript в фигурных скобках. Идея JSX очень похожа на многие языки шаблонов, такие как Thymeleaf, используемый вместе с Java Spring, которые используются на серверах.

JSX похож на [XML](https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction) , что означает, что каждый тег должен быть закрыт. Например, новая строка — это пустой элемент, который в HTML можно записать следующим образом:



но при написании JSX тег нужно закрыть:



### Multiple components

Давайте изменим файл *App.js следующим образом (NB: в этих примерах* не используется export внизу , сейчас и в будущем. Он по-прежнему необходим для работы кода):



Мы определили новый компонент *Hello* и использовали его внутри компонента *App* . Естественно, компонент можно использовать несколько раз:



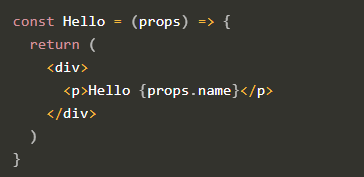
Написание компонентов с помощью React легко, а путем объединения компонентов даже более сложное приложение можно поддерживать достаточно легко. Действительно, основной философией React является создание приложений из множества специализированных повторно используемых компонентов.

Другим важным соглашением является идея *корневого компонента* с именем *App* наверху дерева компонентов приложения. Тем не менее, как мы узнаем в [части 6](https://fullstackopen.com/en/part6) , бывают ситуации, когда компонент *App* не является корневым, а заключен в соответствующий служебный компонент.

### props: passing data to components

Передавать данные компонентам можно с помощью так называемых [props](https://reactjs.org/docs/components-and-props.html).

Модифицируем компонент *Hello* следующим образом:

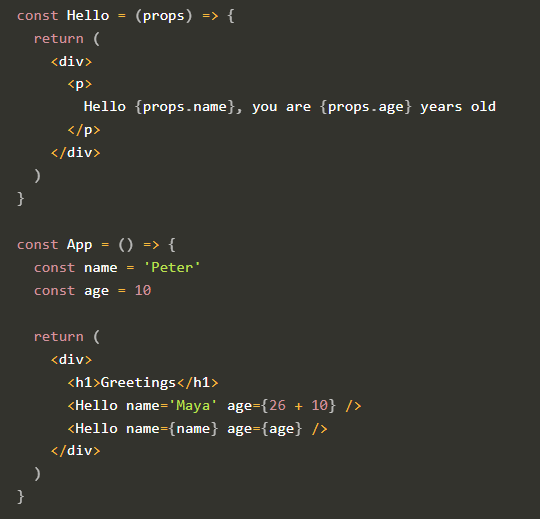


Теперь функция, определяющая компонент, имеет параметр *props* . В качестве аргумента параметр получает объект, в котором есть поля, соответствующие всем «props», которые определяет пользователь компонента.

props определяется следующим образом:



Может быть произвольное количество props, а их значения могут быть «hard-coded» строками или результатами выражений JavaScript. Если значение свойства достигается с помощью JavaScript, оно должно быть заключено в фигурные скобки.

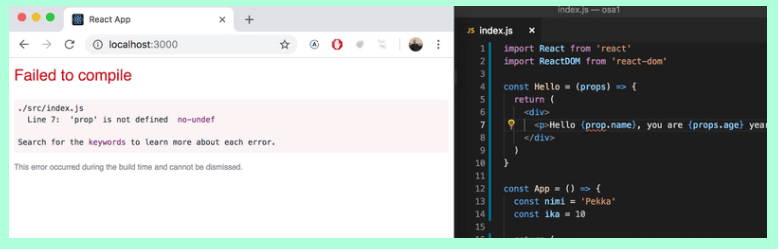


props, отправляемые компонентом *App* , — это значения переменных, результат вычисления выражения суммы и обычная строка.

### Some notes

React настроен так, чтобы генерировать довольно четкие сообщения об ошибках. Несмотря на это, вы должны, по крайней мере в начале, продвигаться вперед **очень маленькими шагами** и следить за тем, чтобы каждое изменение работало так, как вы хотите.

**Консоль всегда должна быть открыта** . Если браузер сообщает об ошибках, не рекомендуется продолжать писать код в надежде на чудо. Вместо этого вы должны попытаться понять причину ошибки и, например, вернуться к предыдущему рабочему состоянию:

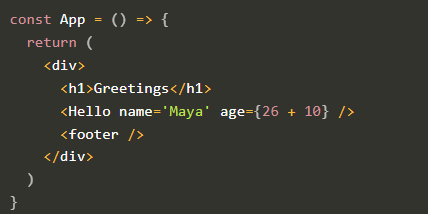


Полезно помнить, что в React возможно и полезно писать команды console.log() (которые выводят на консоль) внутри вашего кода.

Также имейте в виду, что **имена компонентов React должны быть написаны с заглавной буквы** . Если вы попытаетесь определить компонент следующим образом:



и использовать его так

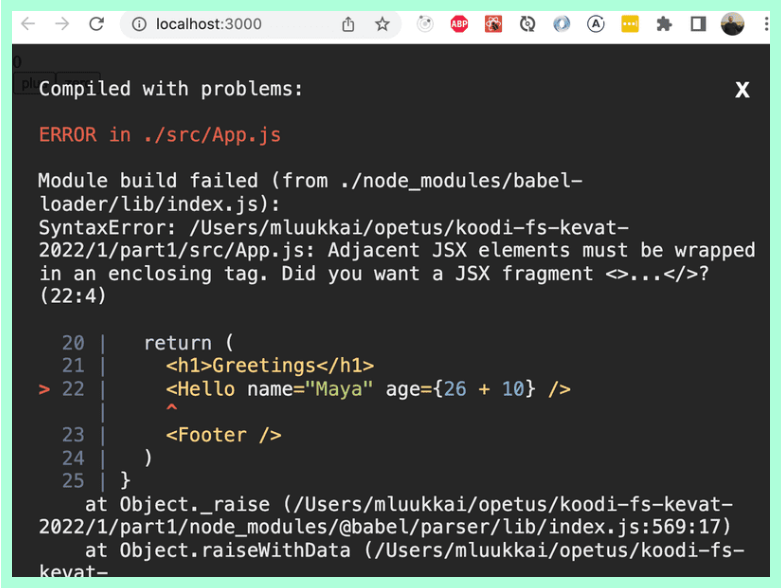


страница не будет отображать содержимое, определенное в компоненте footer, и вместо этого React создаст только пустой элемент [footer](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/footer) , то есть встроенный элемент HTML вместо пользовательского элемента React с тем же именем. Если вы измените первую букву имени компонента на заглавную, то React создаст элемент *div* , определенный в компоненте footer, который отображается на странице.

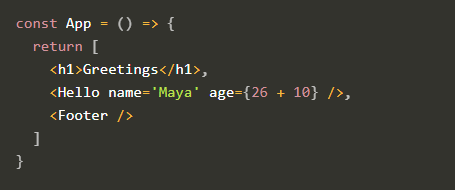
Обратите внимание, что содержимое компонента React (обычно) должно содержать **один корневой элемент** . Если мы, например, попытаемся определить компонент *App* без самого внешнего элемента *div* :



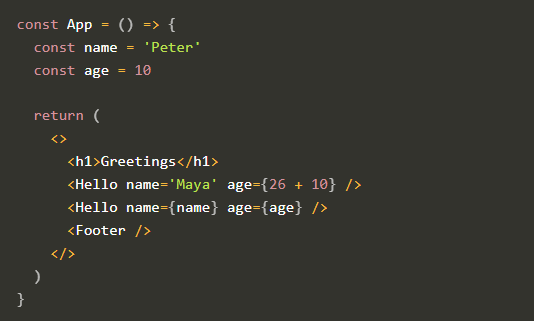
результатом является сообщение об ошибке.



Использование корневого элемента — не единственный рабочий вариант. Массив *компонентов* также является допустимым решением:



Поскольку корневой элемент оговорен, у нас есть «лишние» div-элементы в DOM-дереве. Этого можно избежать, используя [фрагменты](https://reactjs.org/docs/fragments.html#short-syntax) , т.е. оборачивая элементы, которые должны быть возвращены компонентом, пустым элементом:



Теперь он успешно компилируется, и DOM, сгенерированный React, больше не содержит лишнего элемента div.

## **JavaScript**

Во время курса у нас есть цель и необходимость изучить достаточное количество JavaScript в дополнение к веб-разработке.

JavaScript быстро развивался за последние несколько лет, и в этом курсе мы используем функции из более новых версий. Официальное название стандарта JavaScript — [ECMAScript](https://en.wikipedia.org/wiki/ECMAScript) . На данный момент последней версией является версия, выпущенная в июне 2021 года под названием [ECMAScript®2021](https://www.ecma-international.org/ecma-262/) , также известная как ES12.

Браузеры еще не поддерживают все новейшие функции JavaScript. Из-за этого большая часть кода, выполняемого в браузерах, была *перенесена* из более новой версии JavaScript в более старую, более совместимую версию.

На сегодняшний день наиболее популярным способом транспиляции является использование [Babel](https://babeljs.io/) . Транспиляция автоматически настраивается в приложениях React, созданных с помощью create-react-app. Мы более подробно рассмотрим настройку транспиляции в [части 7](https://fullstackopen.com/en/part7) этого курса.

[Node.js](https://nodejs.org/en/) — это среда выполнения JavaScript, основанная на Google [Chrome V8](https://developers.google.com/v8/) JavaScript движке, которая работает практически где угодно — от серверов до мобильных телефонов. Давайте попрактикуемся в написании JavaScript с использованием Node. Предполагается, что на вашем компьютере установлена ​​версия Node.js не ниже *16.13.2* . Последние версии Node уже понимают последние версии JavaScript, поэтому код не нужно транспилировать.

Код записывается в файлы, оканчивающиеся на *.js* , которые запускаются путем ввода команды node name\_of\_file.js.

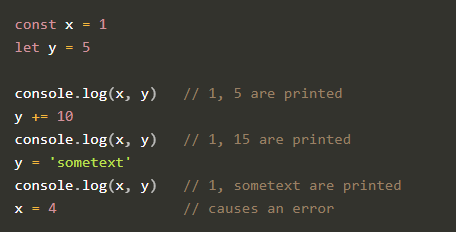
Также можно написать код JavaScript в консоли Node.js, которую можно открыть, введя node в командной строке, а также в консоли инструментов разработчика браузера. [Новейшие версии Chrome довольно хорошо обрабатывают новые функции JavaScript](http://kangax.github.io/compat-table/es2016plus/" \t "_blank) без транспиляции кода. В качестве альтернативы вы можете использовать такой инструмент, как [JS Bin](https://jsbin.com/?js,console) .

JavaScript чем-то напоминает Java как по названию, так и по синтаксису. Но когда дело доходит до основного механизма языка, они не могут быть более разными. Исходя из опыта работы с Java, поведение JavaScript может показаться немного чуждым, особенно если не прилагать усилий для поиска его возможностей.

В определенных кругах также были популярны попытки «имитировать» функции Java и шаблоны проектирования в JavaScript. Мы не рекомендуем этого делать, так как языки и соответствующие экосистемы в конечном итоге очень разные.

### Variables

В JavaScript есть несколько способов определения переменных:



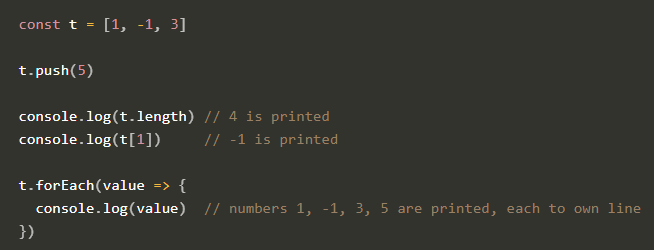
[const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) на самом деле определяет не переменную, а *константу* , значение которой больше нельзя изменить. С другой стороны, [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) определяет нормальную переменную.

В приведенном выше примере мы также видим, что тип данных, присвоенных переменной, может измениться во время выполнения. В начале y хранит целое число, а в конце — строку.

Также возможно определить переменные в JavaScript, используя ключевое слово [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) . var долгое время был единственным способом определения переменных. const и let были недавно добавлены в версию ES6. В определенных ситуациях var работает иначе, чем определения переменных в большинстве языков — см. [Переменные JavaScript — следует ли использовать let, var или const? на Medium](https://medium.com/craft-academy/javascript-variables-should-you-use-let-var-or-const-394f7645c88f" \t "_blank) или [Keyword: var vs. let на JS Tips](http://www.jstips.co/en/javascript/keyword-var-vs-let/) для получения дополнительной информации. Во время этого курса использование var не рекомендуется, и вы должны придерживаться использования const и let! Вы можете найти больше по этой теме на YouTube — например  [var, let и const — особенности ES6 JavaScript](https://youtu.be/sjyJBL5fkp8)

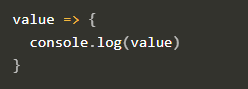
### Arrays

[Массив](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) и пара примеров его использования:



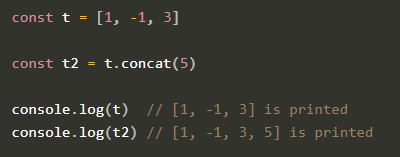
Примечательным в этом примере является тот факт, что содержимое массива может быть изменено, несмотря на то, что он определен как константа . Поскольку массив является объектом, переменная всегда указывает на один и тот же объект. Однако содержимое массива изменяется по мере добавления к нему новых элементов.

Одним из способов перебора элементов массива является использование forEach , как показано в примере. forEach получает в качестве параметра *функцию* , определенную с использованием синтаксиса стрелки.



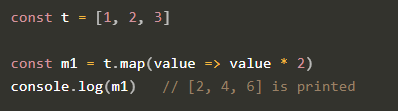
forEach вызывает функцию *для каждого элемента массива* , всегда передавая отдельный элемент в качестве аргумента. Функция в качестве аргумента forEach может принимать и [другие аргументы](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach).

В предыдущем примере новый элемент был добавлен в массив с помощью метода [push](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push) . При использовании React часто используются приемы из функционального программирования. Одной из характеристик парадигмы функционального программирования является использование [неизменяемых - immutable](https://en.wikipedia.org/wiki/Immutable_object) структур данных. В коде React предпочтительнее использовать метод [concat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) , который не добавляет элемент в массив, а создает новый массив, в который включается как содержимое старого массива, так и новый элемент.



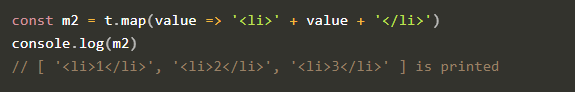
Вызов метода t.concat(5) не добавляет новый элемент в старый массив, а возвращает новый массив, который содержит не только элементы старого массива, но и новый элемент.

Для массивов определено множество полезных методов. Давайте рассмотрим краткий пример использования метода [map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map).



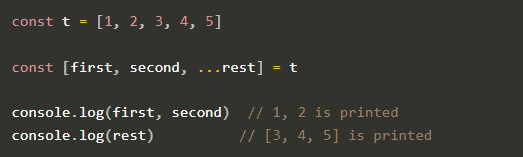
На основе старого массива map создает *новый массив* , для которого функция, заданная в качестве параметра, используется для создания элементов. В этом примере исходное значение умножается на два.

Map также может преобразовать массив во что-то совершенно другое:



Здесь массив, заполненный целочисленными значениями, преобразуется в массив, содержащий строки HTML, с использованием метода map. Во [второй](https://fullstackopen.com/en/part2) части этого курса мы увидим, что map довольно часто используется в React.

Отдельные элементы массива легко присвоить переменным с помощью [деструктурирующего присваивания](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment).



Благодаря присваиванию переменные first и second получат в качестве своих значений первые два целых числа массива. Остальные целые числа «собираются» в собственный массив, который затем присваивается переменной rest .

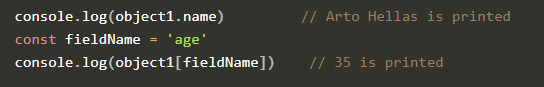
### Objects

Есть несколько разных способов определения объектов в JavaScript. Одним из очень распространенных методов является использование [литералов объекта - object literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Object_literals) , что происходит путем перечисления его свойств в фигурных скобках:

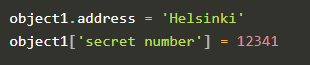


Значения свойств могут быть любого типа, например, целые числа, строки, массивы, объекты...

На свойства объекта ссылаются с помощью нотации «точка .» или с помощью квадратных скобок:



Вы также можете добавлять свойства к объекту, используя запись через точку или скобки:



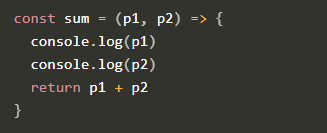
Последнее из дополнений должно быть сделано с помощью квадратных скобок, потому что при использовании записи через точку *секретный номер* не является допустимым именем свойства из-за символа пробела.

Естественно, объекты в JavaScript также могут иметь методы. Однако в ходе этого курса нам не нужно определять какие-либо объекты с собственными методами. Вот почему они лишь кратко обсуждаются в ходе курса.

Объекты также могут быть определены с помощью так называемых функций-конструкторов, что приводит к механизму, напоминающему многие другие языки программирования, например, классы Java. Несмотря на это сходство, в JavaScript нет классов в том же смысле, что и в объектно-ориентированных языках программирования. Однако, начиная с версии ES6, был добавлен *синтаксис класса , который в некоторых случаях помогает структурировать объектно-ориентированные классы.*

### Functions

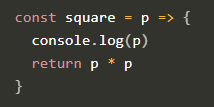
Мы уже познакомились с определением стрелочных функций. Полный процесс определения стрелочной функции выглядит следующим образом:



и функция вызывается, как и следовало ожидать:



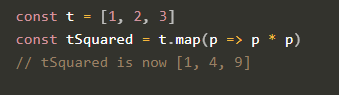
Если есть только один параметр, мы можем исключить круглые скобки из определения:



Если функция содержит только одно выражение, фигурные скобки не нужны. В этом случае функция возвращает только результат своего единственного выражения. Теперь, если мы удалим консольную печать, мы сможем еще больше сократить определение функции:

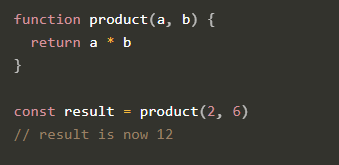


Эта форма особенно удобна при работе с массивами, например, при использовании метода map:

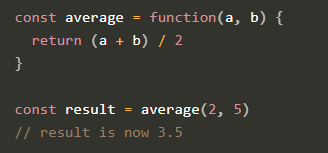


Функция стрелки была добавлена ​​в JavaScript всего пару лет назад, в версии [ES6](http://es6-features.org/) . До этого единственным способом определения функций было использование ключевого слова function .

Есть два способа сослаться на функцию; один дает имя в [объявлении функции - function declaration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function).



Другой способ определить функцию — использовать [функциональное выражение - function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function). В этом случае нет необходимости давать функции имя, и определение может находиться среди остального кода:



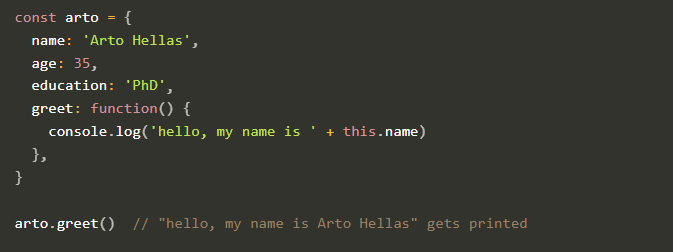
В ходе этого курса мы определим все функции, используя синтаксис стрелки.

### Object methods and "this"

Из-за того, что в этом курсе мы используем версию React, содержащую React Hooks, нам не нужно определять объекты с помощью методов. **Содержание этой главы не имеет отношения к курсу** , но, безусловно, полезно во многих отношениях. В частности, при использовании старых версий React необходимо понимать темы этой главы.

Стрелочные функции и функции, определенные с помощью ключевого слова function, существенно различаются, когда речь заходит о том, как они ведут себя по отношению к ключевому слову [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) , которое относится к самому объекту.

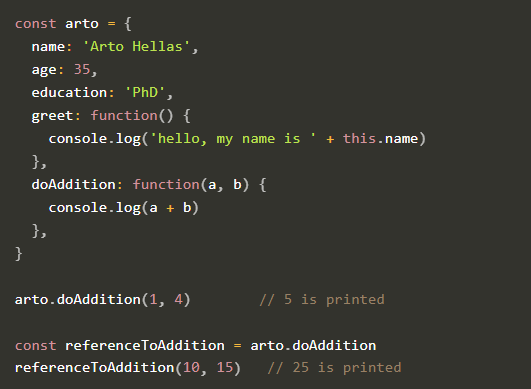
Мы можем назначать методы объекту, определяя свойства, которые являются функциями:



Методы можно назначать объектам даже после создания объекта:

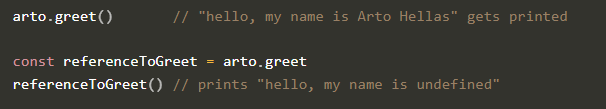


Немного изменим объект:



Теперь у объекта есть метод doAddition , который вычисляет сумму чисел, переданных ему в качестве параметров. Метод вызывается обычным способом, используя объект arto.doAddition(1, 4) или сохраняя *ссылку на метод* в переменной и вызывая метод через переменную: referenceToAddition(10, 15) .

Если мы попытаемся сделать то же самое с методом greet , мы столкнемся с проблемой:



При вызове метода через ссылку метод теряет информацию о том, что было оригинал this. В отличие от других языков, в JavaScript значение [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) определяется на основе *того, как вызывается метод* . При вызове метода через ссылку значение this становится так называемым [глобальным объектом](https://developer.mozilla.org/en-US/docs/Glossary/Global_object) , и конечный результат часто отличается от того, что изначально планировал разработчик программного обеспечения.

Утеря this при написании кода JavaScript приводит к нескольким потенциальным проблемам. Часто возникают ситуации, когда React или Node (или, точнее, движок JavaScript веб-браузера) должен вызвать какой-либо метод в объекте, определенном разработчиком. Однако в этом курсе мы избегаем этих проблем, используя JavaScript «без this».

Еше одна ситуация, приводящая к «исчезновению» this , возникает, когда мы устанавливаем тайм-аут для вызова функции приветствия для объекта arto с помощью функции [setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout).



Как уже упоминалось, значение this в JavaScript определяется на основе того, как вызывается метод. Когда setTimeout вызывает метод, именно движок JavaScript фактически вызывает метод, и в этот момент this ссылается на глобальный объект.

Есть несколько механизмов, с помощью которых можно сохранить оригинал this. Один из них использует метод [bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind):



Вызов arto.greet.bind(arto) создает новую функцию, в которой this обязательно указывает на Arto, независимо от того, где и как вызывается метод.

Используя [стрелочные функции](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) , можно решить некоторые проблемы, связанные с этим . Однако их не следует использовать в качестве методов для объектов, потому что тогда это вообще не работает. Позже мы вернемся к поведению this применительно к стрелочным функциям.

Если вы хотите лучше понять, как это работает в JavaScript, в Интернете полно материалов на эту тему, например, серия скринкастов [Understand JavaScript's this Keyword in Depth](https://egghead.io/courses/understand-javascript-s-this-keyword-in-depth) от [egghead.io](https://egghead.io/) настоятельно рекомендуется!

### Classes

Как упоминалось ранее, в JavaScript нет такого механизма классов, как в объектно-ориентированных языках программирования. Однако есть возможности, позволяющие «симулировать» объектно-ориентированные [классы](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) .

Давайте кратко рассмотрим *синтаксис классов* , который был введен в JavaScript с ES6, что существенно упрощает определение классов (или классоподобных вещей) в JavaScript.

В следующем примере мы определяем «класс» с именем Person и два объекта Person:



Что касается синтаксиса, классы и созданные из них объекты очень напоминают классы и объекты Java. Их поведение также очень похоже на объекты Java. По своей сути они по-прежнему являются объектами, основанными на [прототипном наследовании](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance) JavaScript . Тип обоих объектов на самом деле Object , поскольку JavaScript по существу определяет только типы [Boolean, Null, Undefined, Number, String, Symbol, BigInt и Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures).

Введение синтаксиса класса было спорным дополнением. Проверьте [Not Awesome: классы ES6](https://github.com/petsel/not-awesome-es6-classes) или [«класс» в ES6 — новая «плохая» часть? на Medium](https://medium.com/@rajaraodv/is-class-in-es6-the-new-bad-part-6c4e6fe1ee65) для более подробной информации.

Синтаксис класса ES6 часто используется в «старом» React, а также в Node.js, поэтому его понимание полезно даже в этом курсе. Однако, поскольку мы используем новую функцию React [Hooks](https://reactjs.org/docs/hooks-intro.html) на протяжении всего этого курса, у нас нет конкретного применения синтаксиса классов JavaScript.

### JavaScript materials

В Интернете есть как хорошие, так и плохие руководства по JavaScript. Большинство ссылок на этой странице, относящихся к функциям JavaScript, относятся [к Руководству Mozilla по JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript) .

Настоятельно рекомендуется немедленно прочитать [Повторное введение в JavaScript (учебник по JS)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript) на веб-сайте Mozilla.

Если вы хотите глубже изучить JavaScript, в Интернете есть отличная бесплатная серия книг под названием [You-Dont-Know-JS](https://github.com/getify/You-Dont-Know-JS) .

Еще один отличный ресурс для изучения JavaScript — [javascript.info](https://javascript.info/) .

Бесплатная и увлекательная книга Eloquent JavaScript [https://eloquentjavascript.net](https://eloquentjavascript.net/) . Быстро перенесет вас от основ к интересным вещам, смесь теоретических проектов и упражнений, охватывает общую теорию программирования, а также язык JavaScript.

На сайте [egghead.io](https://egghead.io/) есть множество качественных скринкастов по JavaScript, React и другим интересным темам. К сожалению, часть материалов находится за платным доступом.