**КТ № 13.2. Express**

**Время на выполнение практической работы:** 6 часов.

**Теоретические сведения к выполнению практической работы:**

Рассмотрим создание сервера с помощью фреймворка Express. Казалось бы, зачем нам нужен дополнительный фреймворк, если мы можем воспользоваться готовым модулем http, который есть в Node.js API. Однако Express сам использует модуль http, но вместе с тем предоставляет ряд готовых абстракций, которые упрощают создание сервера и серверной логики, в частности, обработка отправленных форм, работа с куками, CORS и т.д.

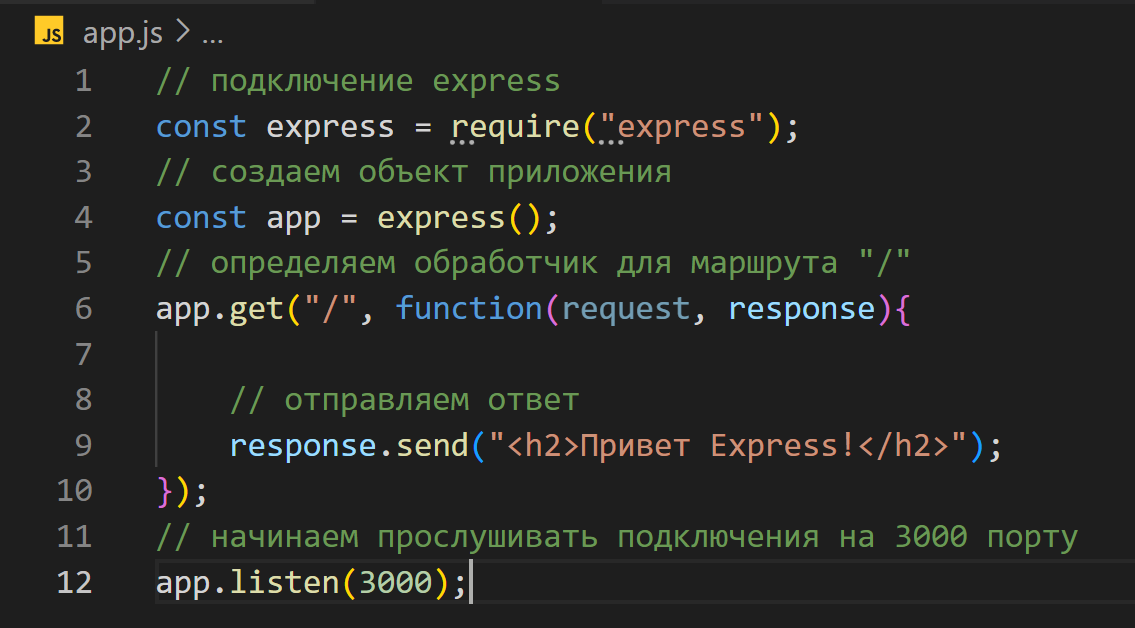
Создадим для проекта новый каталог, который назовем, к примеру, expressapp. Для хранения информации обо всех зависимостях проекта определим в этом каталоге новый файл package.json:



Далее перейдем к этому каталогу в командной строке/терминале и для добавления всех нужных пакетов выполним команду:



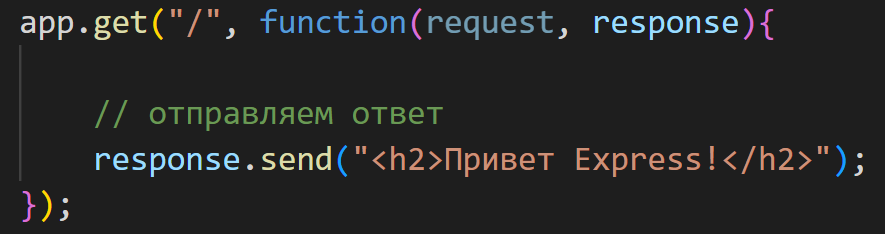
Создадим в каталоге проекта новый файл app.js:



Для использования Express в начале надо создать объект, который будет представлять приложение:



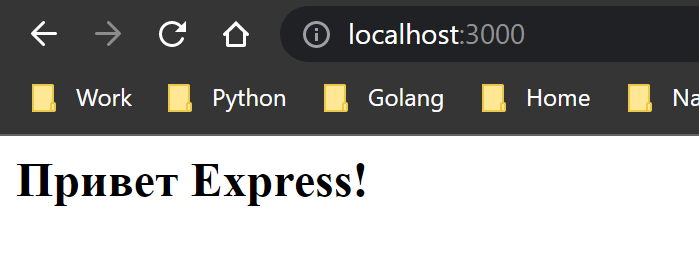
Для обработки запросов в Express определено ряд встроенных функций, и одной из таких является функция app.get(). Она обрабатывает GET-запросы протокола HTTP и позволяет связать маршруты с определенными обработчиками. Для этого первым параметром передается маршрут, а вторым - обработчик, который будет вызываться, если запрос к серверу соответствует данному маршруту:



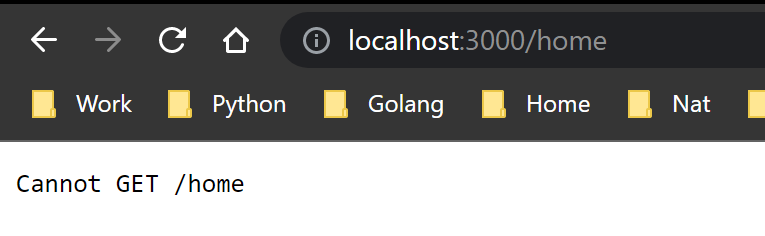
Маршрут "/" представляет корневой маршрут.

Для запуска сервера вызывается метод app.listen(), в который передается номер порта.

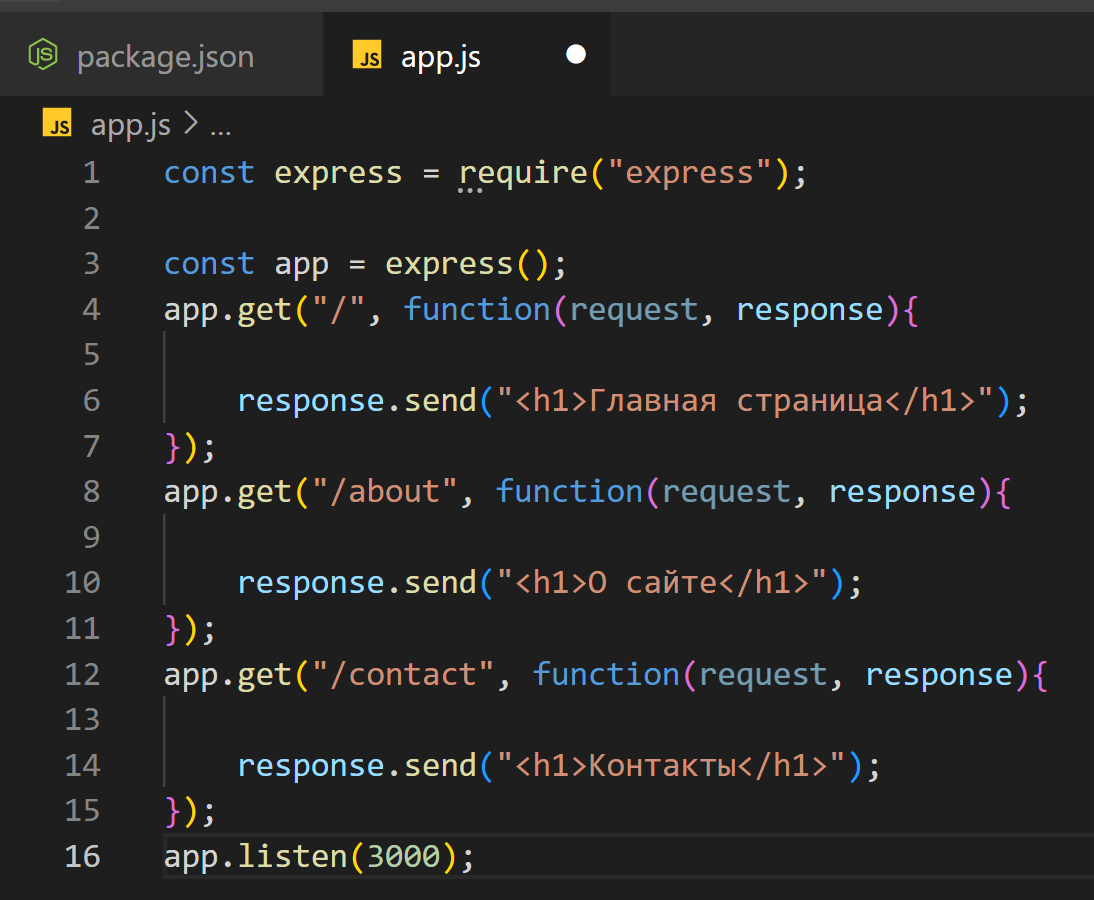
Запустим проект и обратимся в браузере по адресу <http://localhost:3000/>:



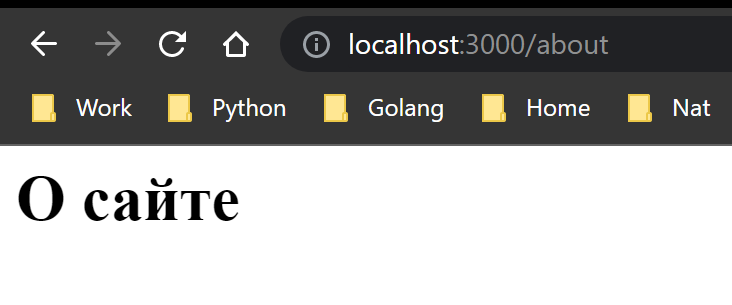
И что важно, Express опирается на систему маршрутов, поэтому все другие запросы, которые не соответствуют корневому маршруту "/", не будут обрабатываться:

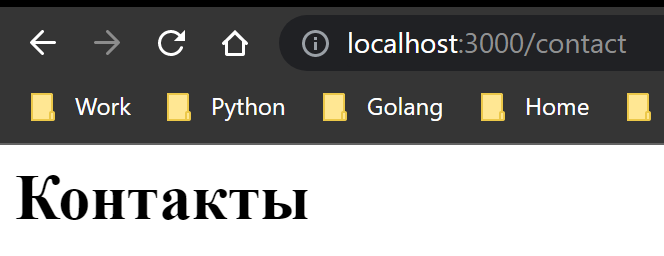


Теперь изменим файл app.js:



Теперь в приложении определено три маршрута, которые будут обрабатываться сервером:

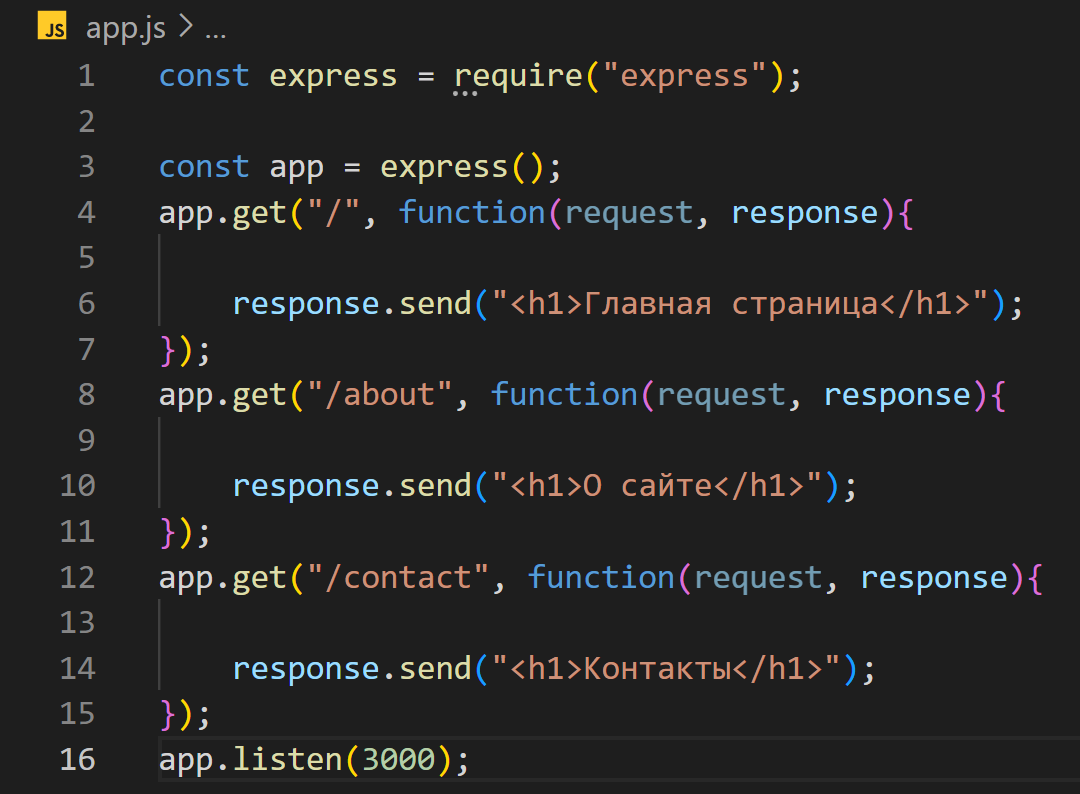




**Конвейер обработки запроса и middleware**

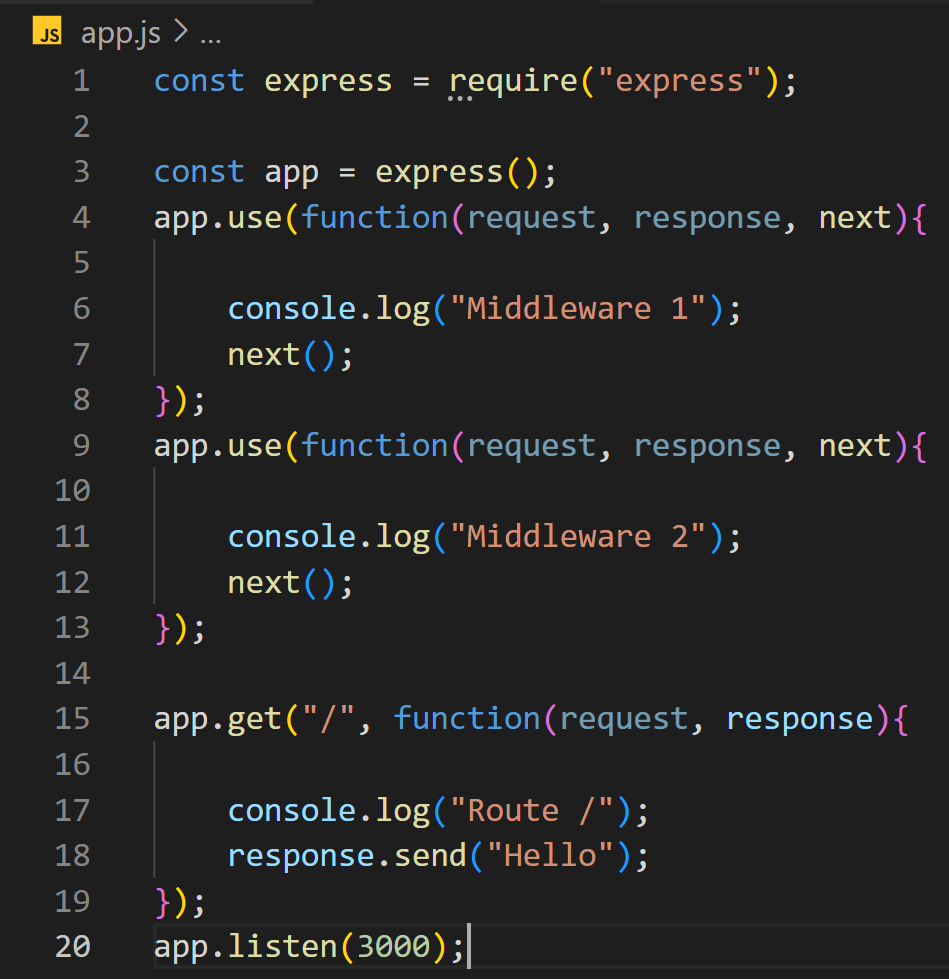
Когда фреймворк Express получает запрос, этот запрос передается в конвейер обработки. Конвейер состоит из набора компонентов или middleware, которые получают данные запроса и решают, как его обрабатывать.

Наш файл приложения выглядит следующим образом:



Здесь конвейер обработки состоял из вызовов app.get(), которые сравнивали запрошенный адрес с маршрутом, и если между адресом и маршрутом было соответствие, то данный запрос обрабатывался методом app.get().

При необходимости мы можем встроить в конвейер обработки запроса на любом этапе любую функцию middleware. Для этого применяется метод app.use().Изменим файл app.js следующим образом:

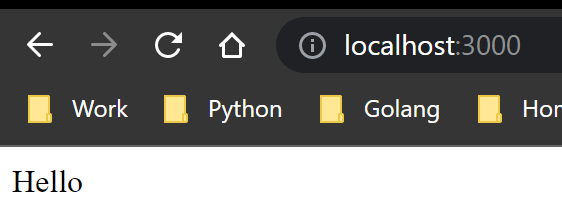


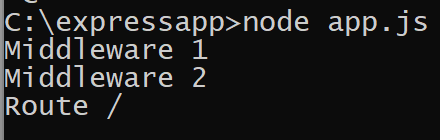
Функция, которая передается в app.use(), принимает три параметра:

* request: данные запроса
* response: объект для управления ответом
* next: следующая в конвейере обработки функция

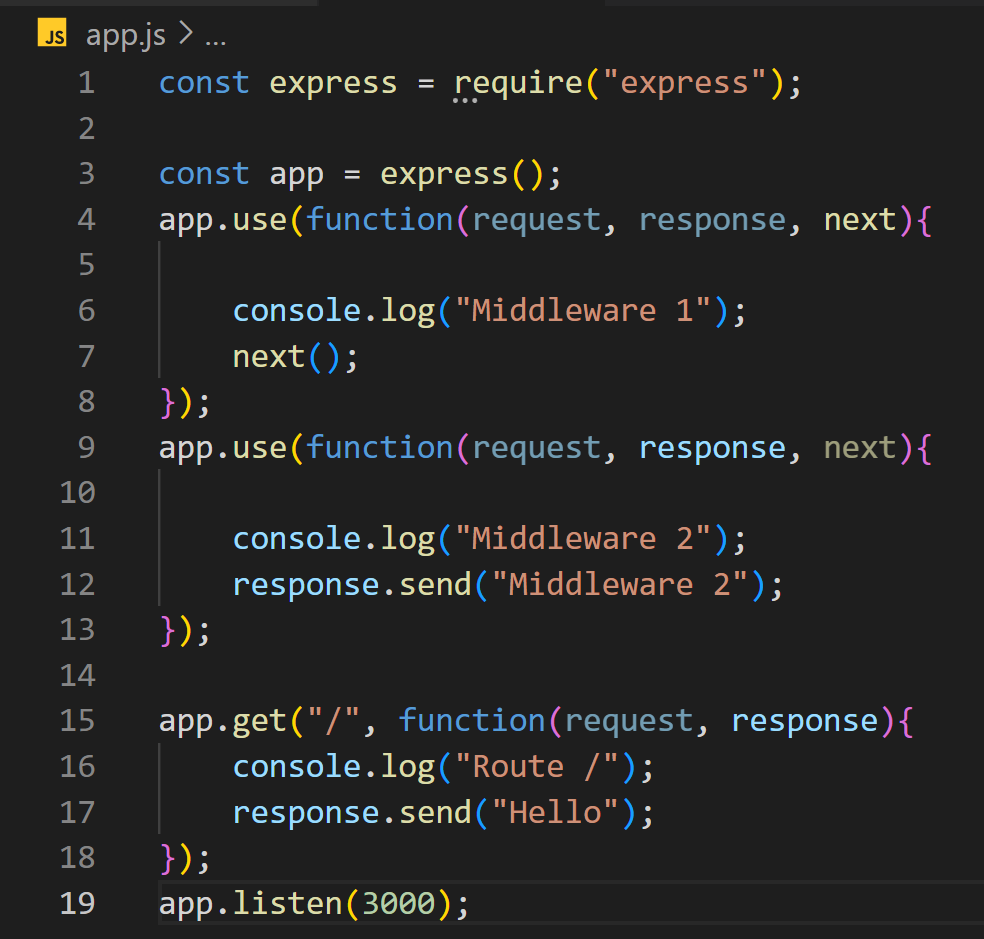
Каждая из функций middleware просто выводит на консоль сообщение и в конце вызывает следующую функцию с помощью вызова next().

При запуске приложения после обращения по адресу "http://localhost:3000/" последовательно отработают все три middleware:

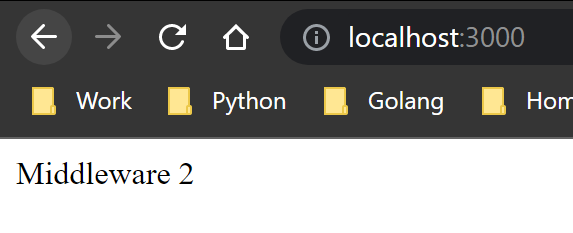


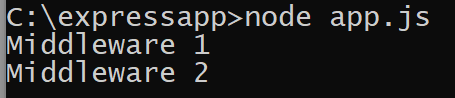


Однако необязательно вызывать все последующие middleware, мы можем на каком-то этапе остановить обработку:



Теперь обработка завершается на Middleware 2, так как в этом методе происходит отправка ответа с помощью response.send(), а вызова следующей функции через next():

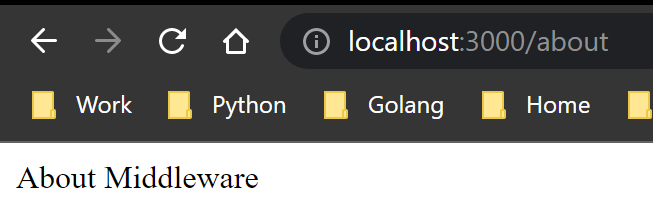


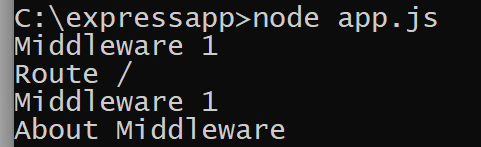


Функции middleware также могут сопоставляться с определенными маршрутами. Например:



В данном случае вторая функция middleware явно сопоставляется с маршрутом "/about", поэтому она будет обрабатывать только запрос "http://localhost:3000/about". Первая функция middleware по прежнему обрабатывает все запросы:





**Отправка ответа**

Для отправки ответа в express у объекта response можно использовать ряд функций. Самый распространенный способ отправки ответа представляет функция send(). В качестве параметра эта функция может принимать объект Buffer, строку, в том числе с html-кодом, объект javascript или массив.



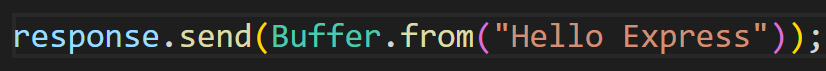
Отправка объекта:



Отправка массива:



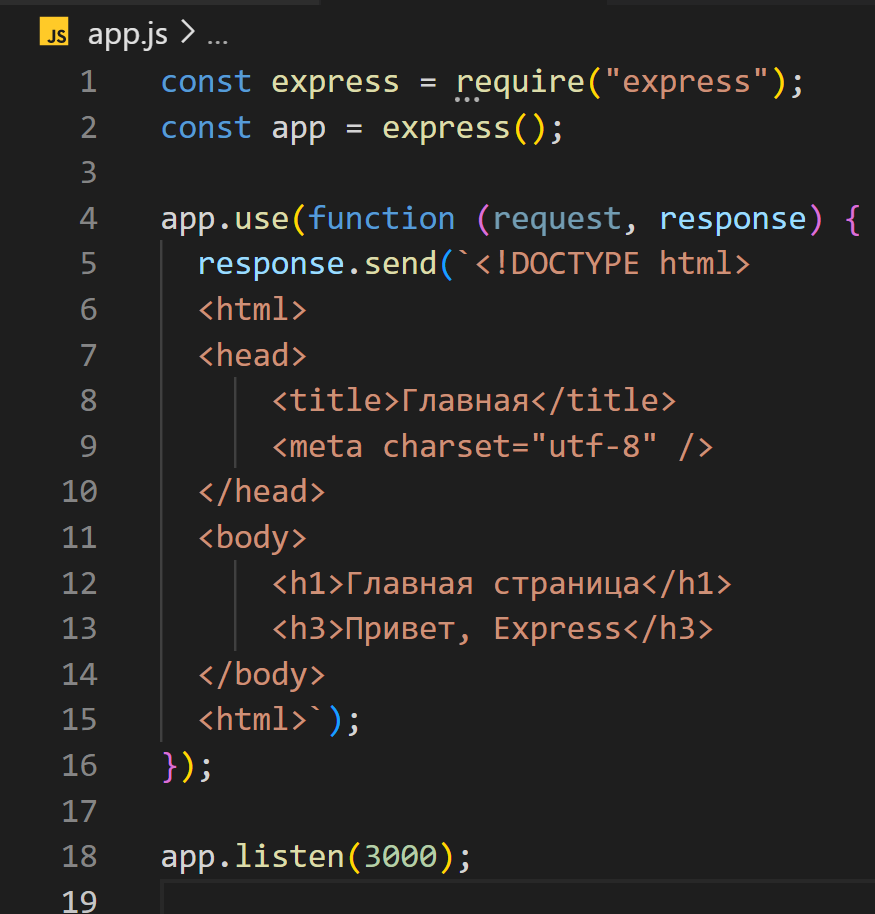
Отправка объекта:

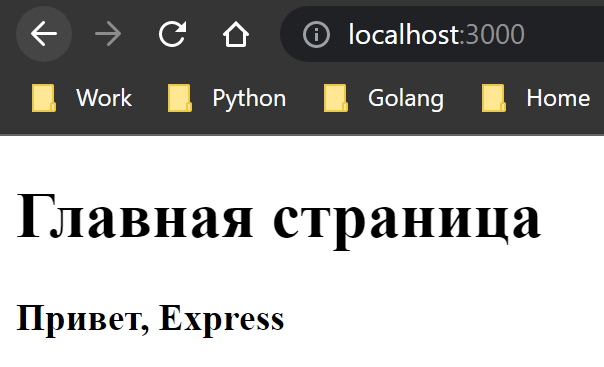


Объект Buffer формально представляет некоторые бинарные данные. Так, в случае выше при выполнении кода по умолчанию браузер загрузит файл, в котором будет строка "Hello Express".

**sendFile**

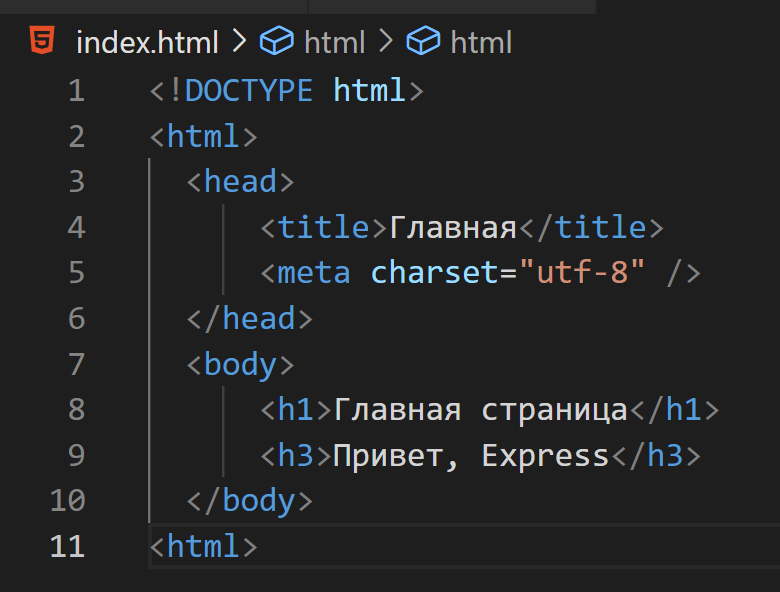
Метод send удобен для отправки строк, некоторого кода html небольшой длины, однако есть отправляемый код html довольно большой, то соответственно код приложения тоже становится громоздким. Например, мы можем написать так:





Однако гораздо лучше определять код html в отдельных файлах и затем эти файлы отправлять с помощью функции sendFile().

Например, определим в папке проекта новый файл index.html:



Отправим этот файл с помощью функции sendFile:



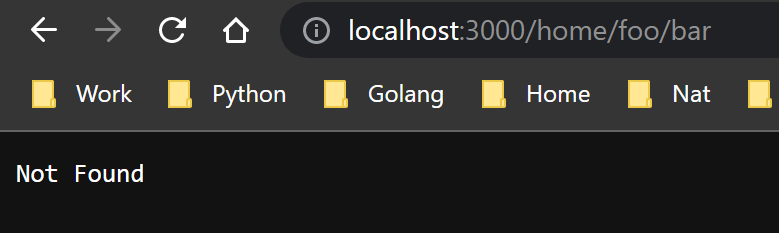
В итоге мы получим тот же самый результат.

Следует учитывать, что в функцию sendFile необходимо передавать абсолютный путь к файлу, именно для этого с помощью \_\_dirname получаем абсолютный путь к текущему проекту и затем добавляем к нему путь к файлу в рамках текущего проекта.

**Отправка статусных кодов**

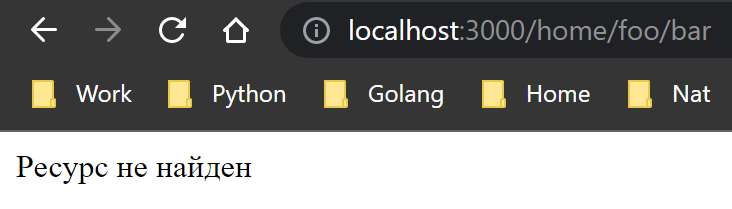
Функция sendStatus() отправляет пользователю определенный статусный код с некоторым сообщением по умолчанию. Например, отправим статусный код 404, который говорит, что ресурс не найден:





При отправке статусного кода 404 также отправляется сообщение "Not Found". Но, возможно, мы захотим отправлять какие-то свои более информативные сообщения. В этом случае можно использовать комбинацию функции status(), которая также отправляет статусный код, и функции send():

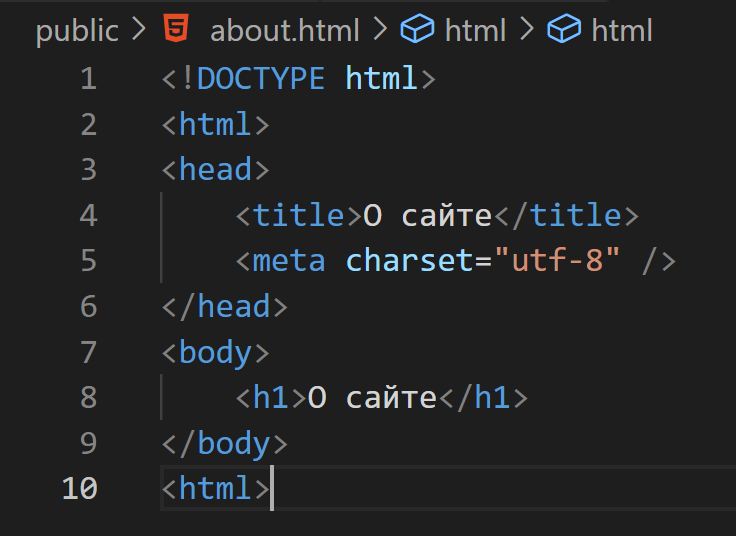




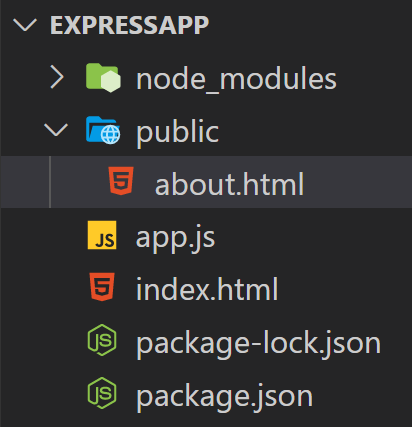
**Статические файлы**

Рассмотрим, как в Node.js + Express отправлять пользователю статические файлы html. Для работы со статическими файлами в Express определен специальный компонент express.static(), который указывает на каталог с файлами.

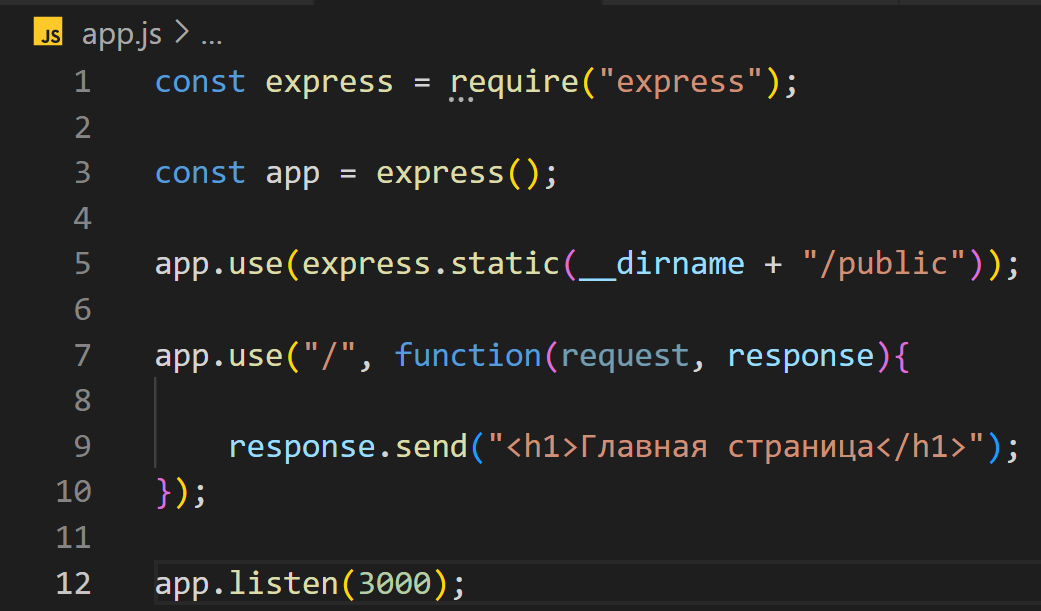
Создадим для статических файлов в проекте каталог public, в который добавим новую html-страницу, которую назовем about.html:



В итоге проект будет выглядеть следующим образом:



Теперь изменим файл app.js:



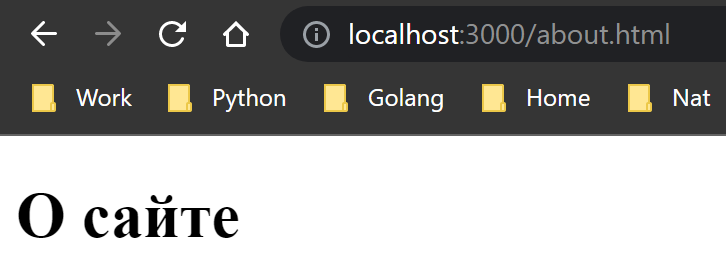
Чтобы встроить компонент express.static в процесс обработки запроса, вызывается функция app.use(). Эта функция позволяет добавлять различные компоненты, которые еще называются middleware, в конвейер обработки запроса:



Причем данный вызов помещается до всех остальных вызовов функции app.get().

В саму же функцию express.static() передается путь к папке со статическими файлами. Специальное выражение \_\_dirname позволяет получить полный путь к папке.

Запустим приложение на выполнение и обратимся в браузере по пути "http://localhost:3000/about.html":



Следует отметить, что для обращения к файлу мы указываем только имя файла без названия каталога public. Также не следует путать подобное обращение к файлам с действием функции sendFile: в данном случае мы напрямую обращаемся к статическим файлам, а функция sendFile фактически берет содержимое из файла и отсылает его пользователю.

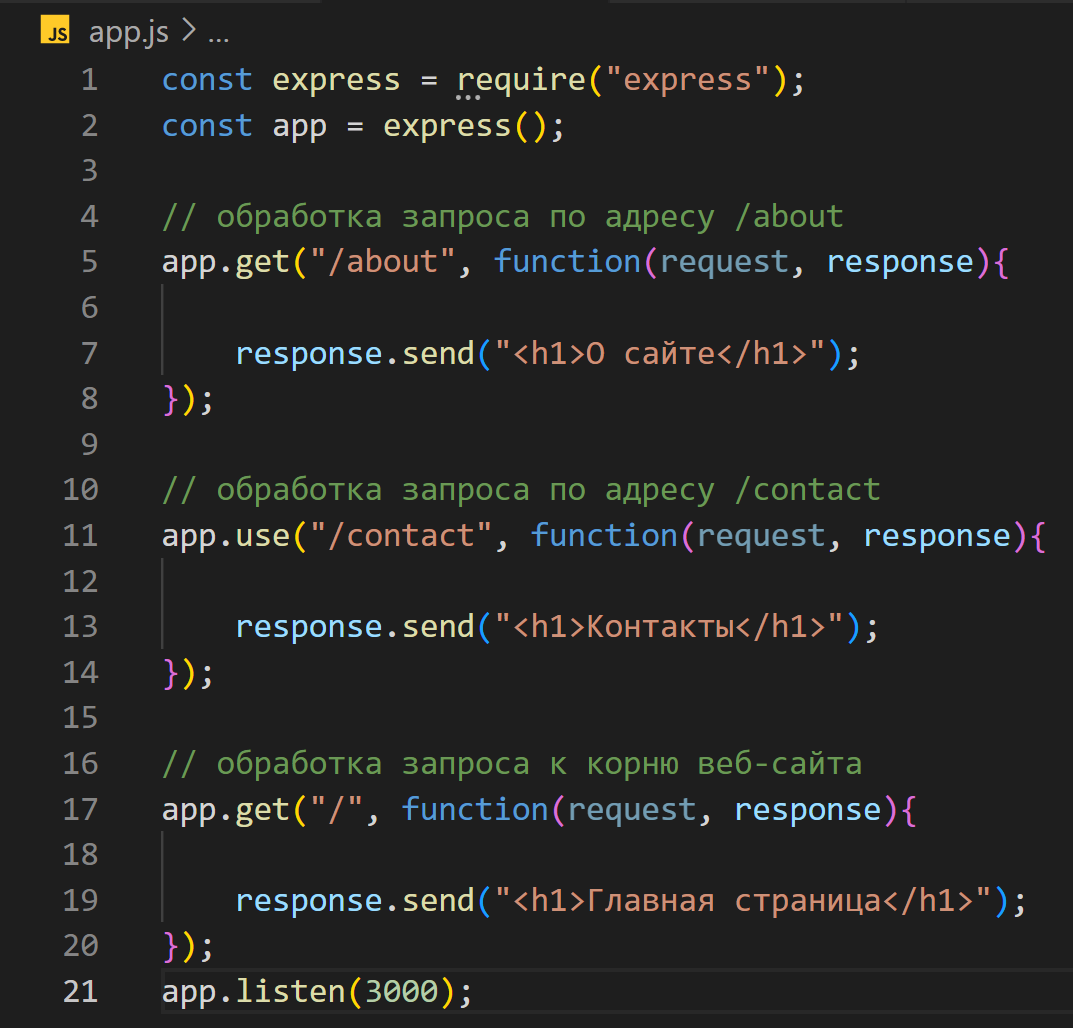
**Маршрутизация**

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик.

Для обработки данных по определенному маршруту можно использовать ряд функций, в частности:

* use
* get
* post
* put
* delete

В качестве первого параметра эти функции могут принимать шаблон адреса, запрос по которому будет обрабатываться. Второй параметр функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу. Например:

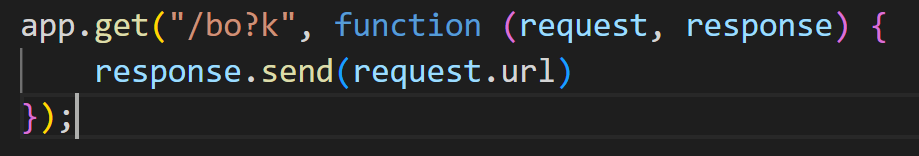


Когда приходит запрос Express сопоставляет запрошенный адрес с каждым из маршрутов. Затем выбирается первый совпавший маршрут. При совпадении маршрута вызывается его функция обработчика.

**Символы подстановок**

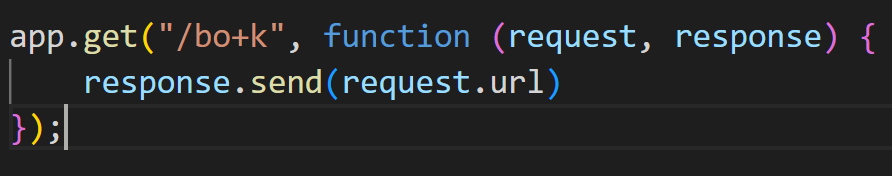
Используемые шаблоны адресов могут содержать регулярные выражения или специальные символы подстановок. В частности, мы можем использовать такие символы, как ?, +, \* и ().

К примеру, символ ? указывает, что предыдущий символ может встречаться 1 раз или отсутствовать. И мы можем определить следующую функцию:



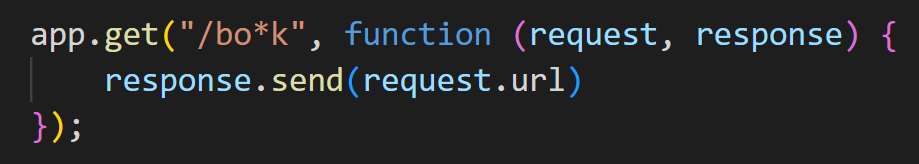
Такой маршрут будет соответствовать строке запроса "/bk" или "/bok".

Символ + указывает, что предыдущий символ может встречаться 1 и более раз:



Такой маршрут будет соответствовать запросам "/bok", "/book", "/boook" и так далее.

Символ звездочка \* указывает, что на месте данного символа может находиться любое количество символов:



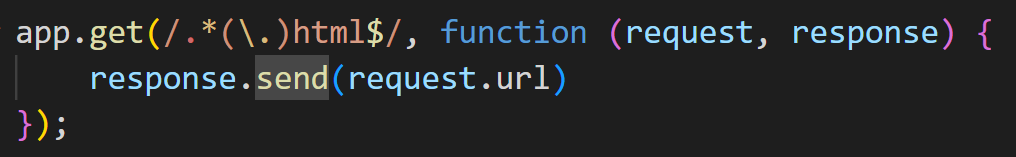
Такой маршрут будет соответствовать запросам "/bork", "/bonk", "/bor.dak", "/bor/ok" и так далее.

Скобки () позволяют оформить группу символов, которые могут встречаться в запросе:



Выражение (.html)? указывает, что подстрока ".html" может встречаться или отсутствовать в строке запроса. И такой маршрут будет соответствовать запросам "/book" и "/book.html".

Также вместо определения маршрутов мы можем указывать регулярные выражения. Например, необходимо перехватить запрос ко всем файлам html или всем путям, которые в конце имеют ".html":



**Переадресация**

Для переадресации применяется метод redirect():

*redirect([status,] path)*

В качестве параметра path передается путь, на который будет перенаправляться пользователь. Дополнительный параметр status задает статусный код переадресации. Если этот параметр не задан, тогда по умолчанию отправляется статусный код 302, который предполагает временную переадресацию.

С помощью данного метода можно выполнять переадресацию как по относительным путям, так и по абсолютным, в том числе на другие домены.

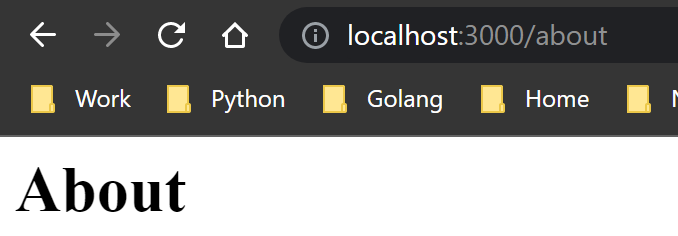
Переадресация по абсолютному пути:



В данном случае при обращении по пути "/index" будет идти переадресация на сайт https://nodejs.org/en.

Переадресация по относительным путям также не очень сложна, но здесь важно учитывать, как именно определяется адрес для редиректа. Рассмотрим редирект относительно текущего пути, с которого производится редирект. Например:





В данном случае будет идти перенаправление с ресурса "/home" на ресурс "/about", то есть, условно говоря, с http://localhost:3000/home на http://localhost:3000/about.

Теперь рассмотрим похожий пример:

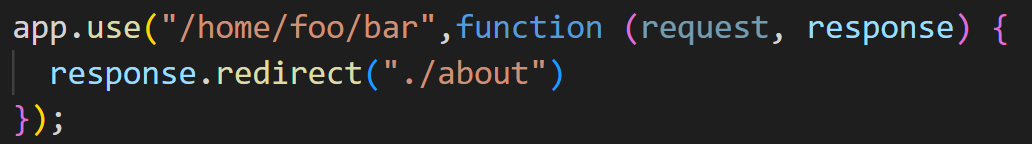


Здесь с ресурса "/home/bar" также идет переадресация на ресурс "about", однако в реальности теперь это будет не http://localhost:3000/about, а http://localhost:3000/home/about. То есть мы как бы поднимаемся на один уровень вверх - с "home/bar" на "home/" и затем к нему добавляется "about".

Если нам необходимо выполнить переадресацию не относительно текущего ресурса, а относительно корневого каталога приложения, то в начале адреса ставится слеш:

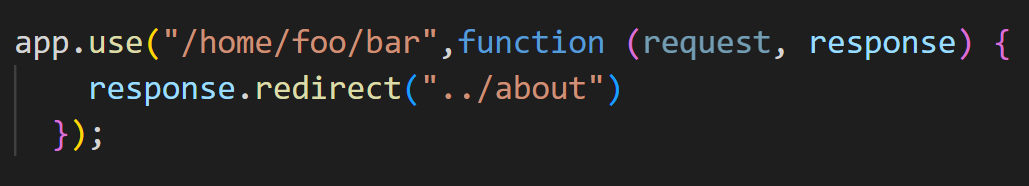


Еще несколько примеров. Переадресация относительно текущего адреса на адрес на том же уровне:



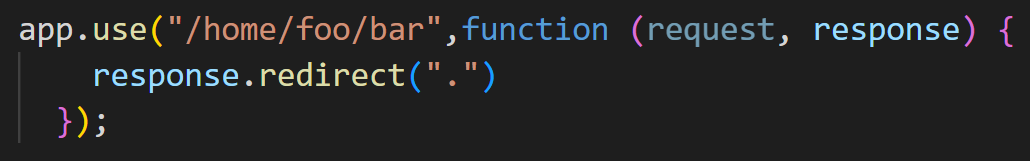
Здесь идет переадресация с http://localhost:3000/home/foo/bar на http://localhost:3000/home/foo/about

Переадресация на адрес, который располагается уровнем выше:



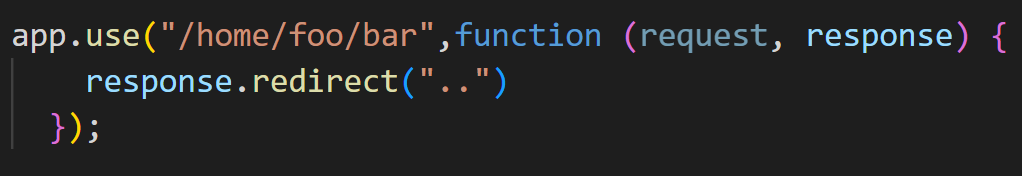
Здесь идет переадресация с http://localhost:3000/home/foo/bar на http://localhost:3000/home/about

Переадресация на уровень выше:



Здесь идет переадресация с http://localhost:3000/home/foo/bar на http://localhost:3000/home/foo

Переадресация на два уровня выше:



Здесь идет переадресация с http://localhost:3000/home/foo/bar на http://localhost:3000/home

По умолчанию при редиректе передается статусный код 302, который указывает, что ресурс временно доступен по новому адресу. Но мы можем указать статусный код 301, чтобы сделать переадресацию постоянной:

*response.redirect(301, "/about");*

**Передача данных приложению. Параметры строки запроса**

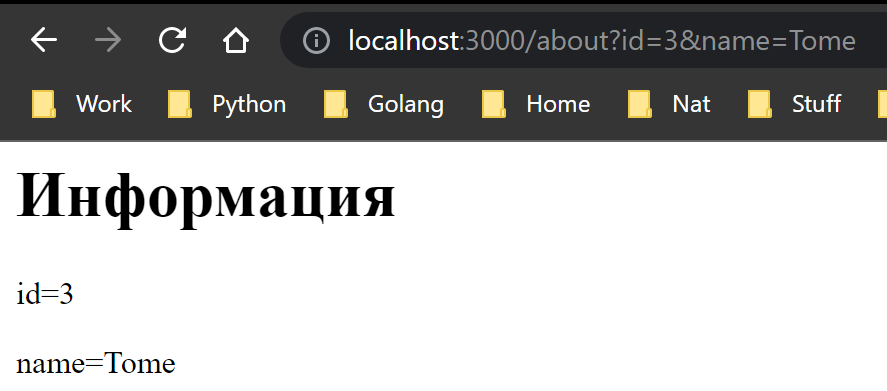
Одним из способов передачи данных в приложение представляет использование параметров строки запроса. Строка запроса (query) - фактически это часть запрошенного адреса, которая идет после знака вопроса. Например, в запросе http://localhost:3000/about?id=3&name=Tome часть id=3&name=Tome представляет строку запроса.

Строку запроса образуют параметры. После названия каждого параметра после знака равно (=) идет его значение. Друг от друга параметры отделяются знаком амперсанда. Например, в адресе выше использовалось два параметра: параметр "id" имеет значение "3" и параметр "name" имеет значение "Tom".

В express мы можем получить параметра строки запроса через свойство query объекта request, который передается в функцию обработки запроса. Например:

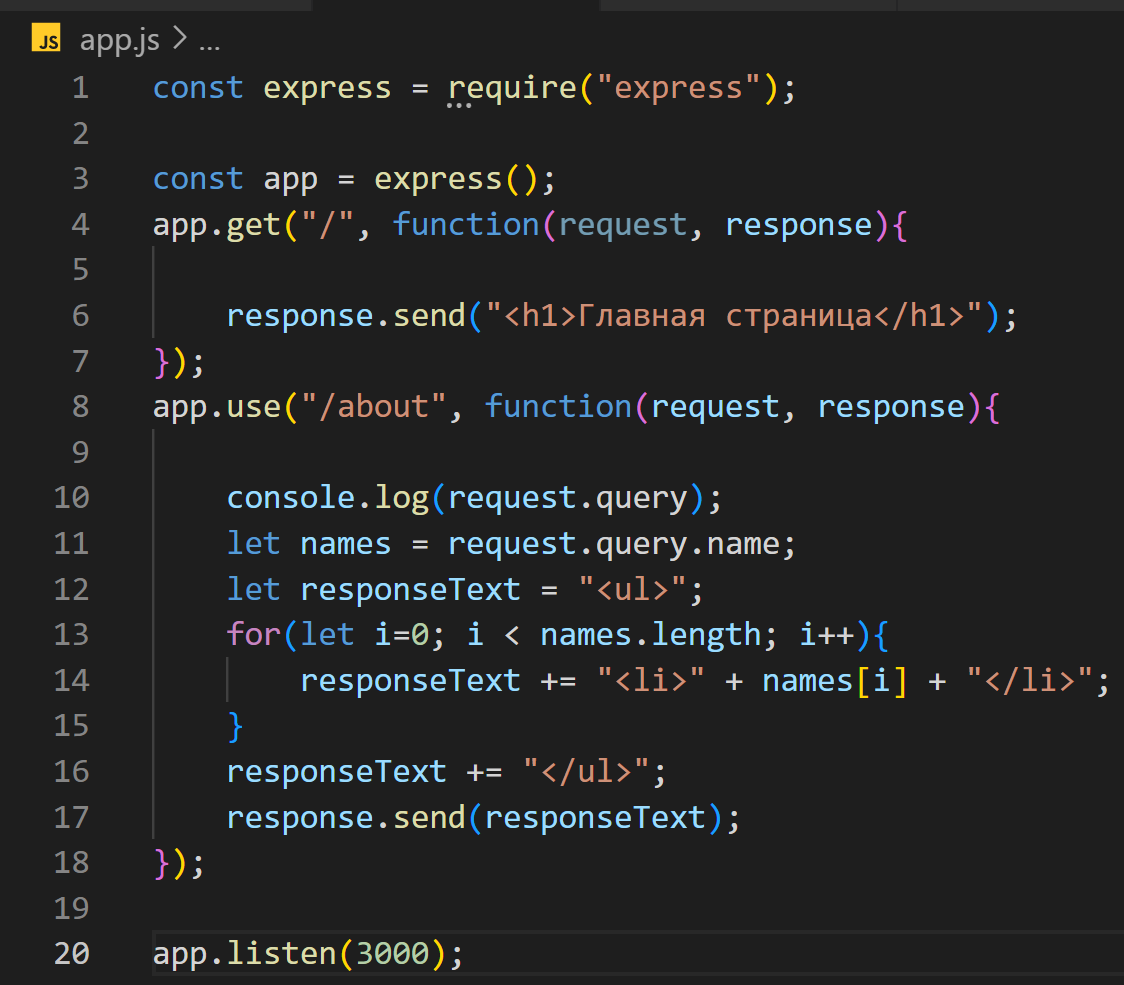


С помощью выражения request.query мы можем получить все параметры строки запрос в виде объекта javascript, а с помощью выражения request.query.название\_параметра мы можем обратиться к каждому отдельному параметру.

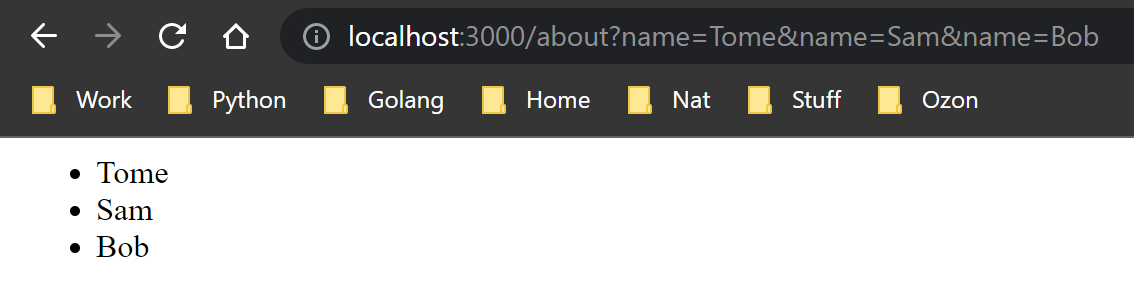


**Передача массивов**

Подобным образом мы можем передавать массивы данных:



В данном случае в приложение будет передаваться параметр name, который представляет массив.

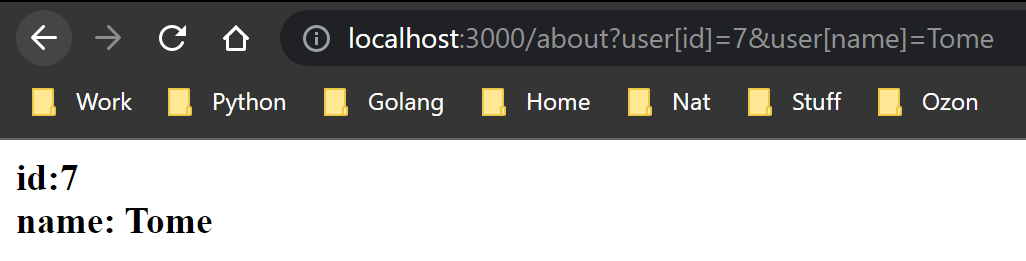


**Передача сложных объектов**

Также можно передавать более сложные объекты, которые состоят из множества свойств:



В данном случае мы получаем объект user, который содержит два свойства id и name, например, user: { id: '7', name: 'Tom' } .

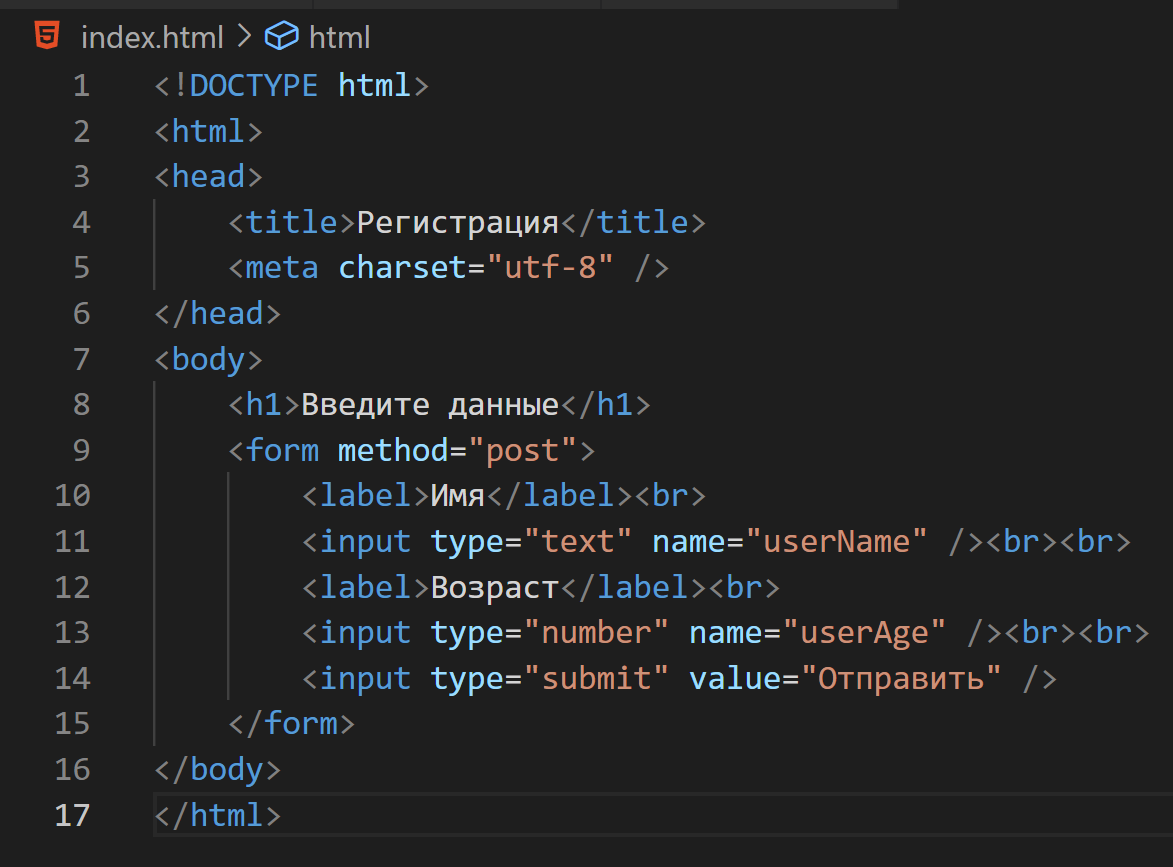


При передаче в строке запроса свойства объекта помещаются в квадратные скобки: user[id].

**POST-запросы и отправка форм**

При оправке каких-то сложных данных обычно используются формы. Рассмотрим, как получать отправленные данные в Express.

Вначале определим в папке проекта новый файл index.html с формой ввода данных:

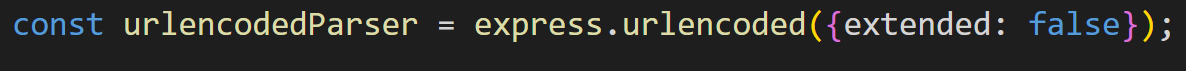


Здесь определены два поля для ввода имени и возраста пользователя.

В файле app.js определим следующий код:



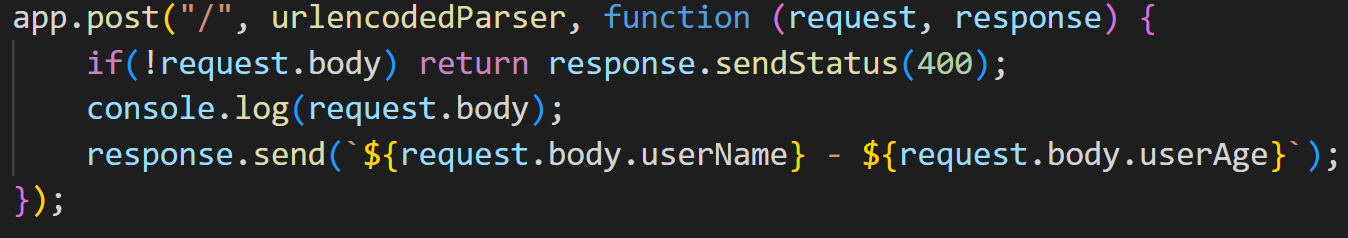
Прежде всего для получения отправленных данных необходимо создать парсер:



Поскольку данные отправляются с помощью формы, то для создания парсера применяется функция urlencoded(). В эту функцию передается объект, устанавливающий параметры парсинга. Значение extended: false указывает, что объект - результат парсинга будет представлять набор пар ключ-значение, а каждое значение может быть представлено в виде строки или массива.

При переходе по адресу "/" будет срабатывать метод app.get, который отправит пользователю файл index.html.

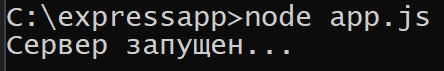
Так как данные отправляются с помощью метода POST, то для обработки определяем функцию app.post("/",...). Первый параметр функции - адрес, на который идет отправка - "/". В коде формы в файле index.html мы не указали адрес для отправки, поэтому по умолчанию данные будут отправляться по тому же адресу, по которому доступна веб-страница index.html - "/". Таким образом, с одним адресом "/" связаны две функции, только одна обрабатывает запросы get, а другая - запросы post. Второй параметр - выше созданный парсер. Третий параметр - обработчик:



Для получения самих отправленных данных используем выражения типа request.body.userName, где request.body инкапсулирует данные формы, а userName - ключ данных, который соответствует значению атрибута name поля ввода на html-странице:

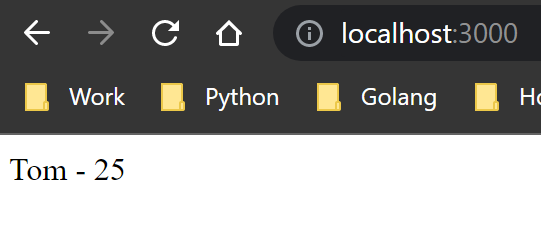


Запустим приложение и обратимся к странице index.html:





После отправки введенные данные будут получены в функции app.post("/",...), которая выведет результат обработки:



**Параметры маршрута**

Параметры маршрута представляют именованные сегменты URL-адреса. Не стоит их путать с параметрами строки запроса. Например:

*localhost:3000/about/user?id=3&name=Tome*

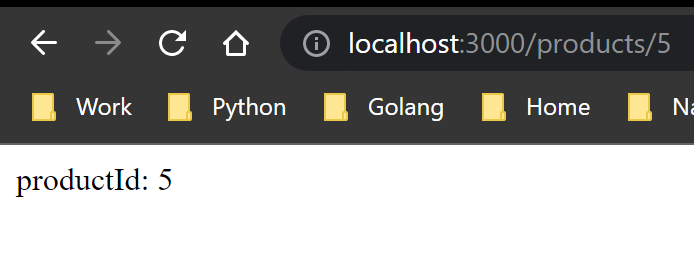
Здесь параметры строки запроса - это то, что идет после вопросительного знака - id=3&name=Tome. Остальная часть, которая идет до вопросительного знака может содержать параметры маршрута.

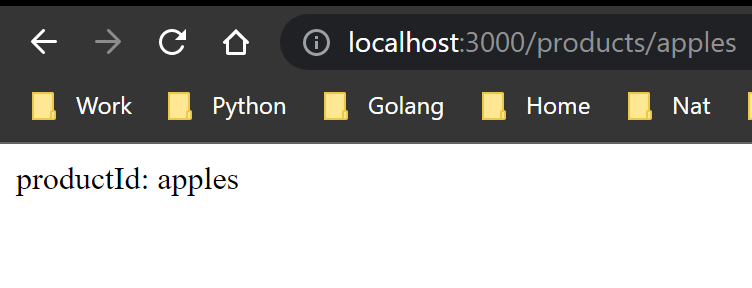
Название параметра должно включать символы из диапазона [A-Za-z0-9\_]. В определении маршрута параметры предваряются знаком двоеточия:



В данном случае параметр называется "productId". Через коллекцию request.params можно получить все параметры и, в частности, значение параметра productId.

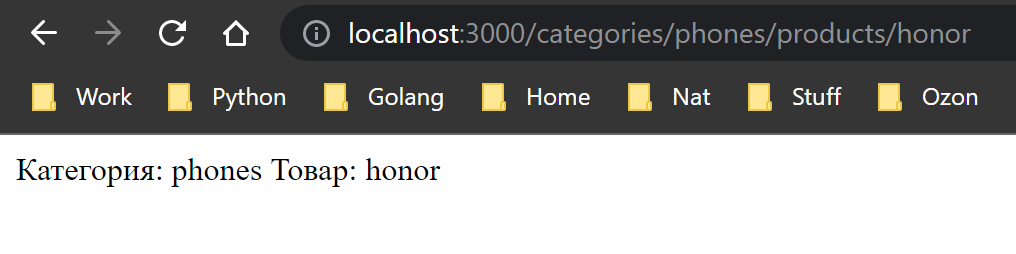
Если нам потребуется передать для этого параметра значение, то оно указывается в качестве последнего сегмента в строке запроса:



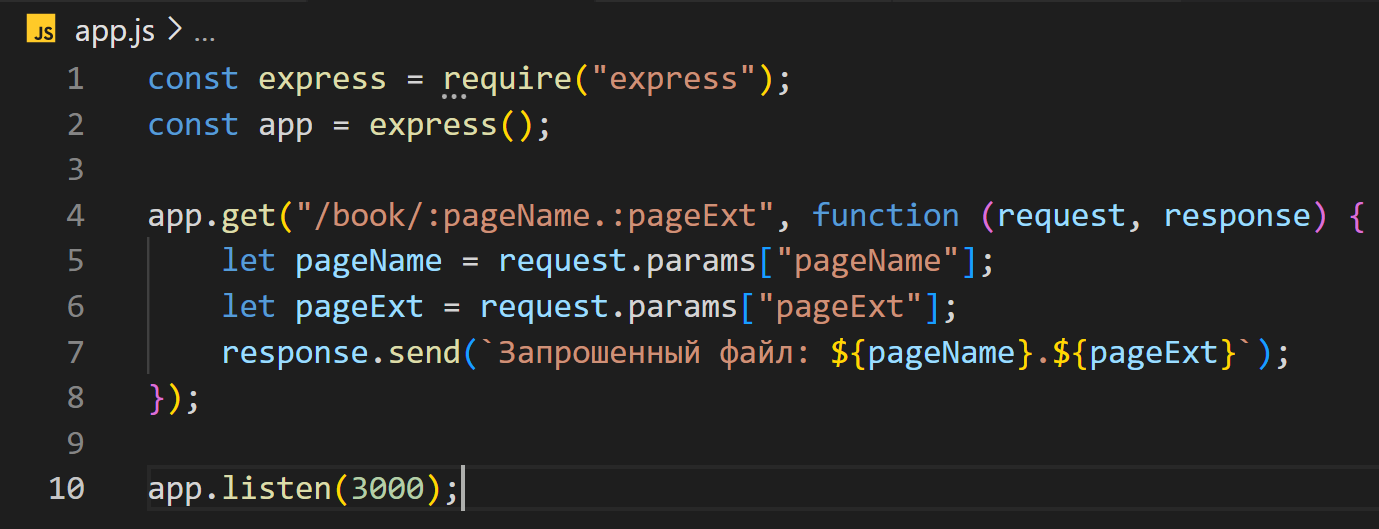


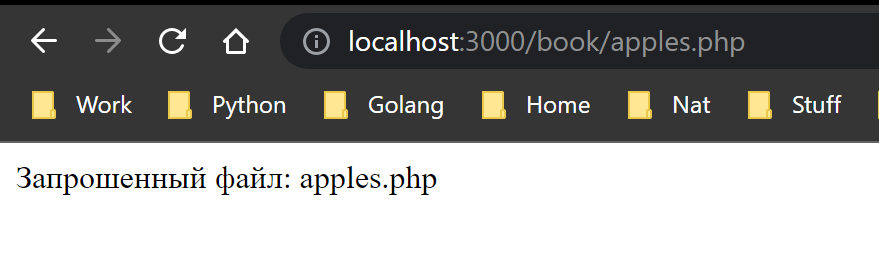
Мы можем использовать более сложные комбинации параметров:





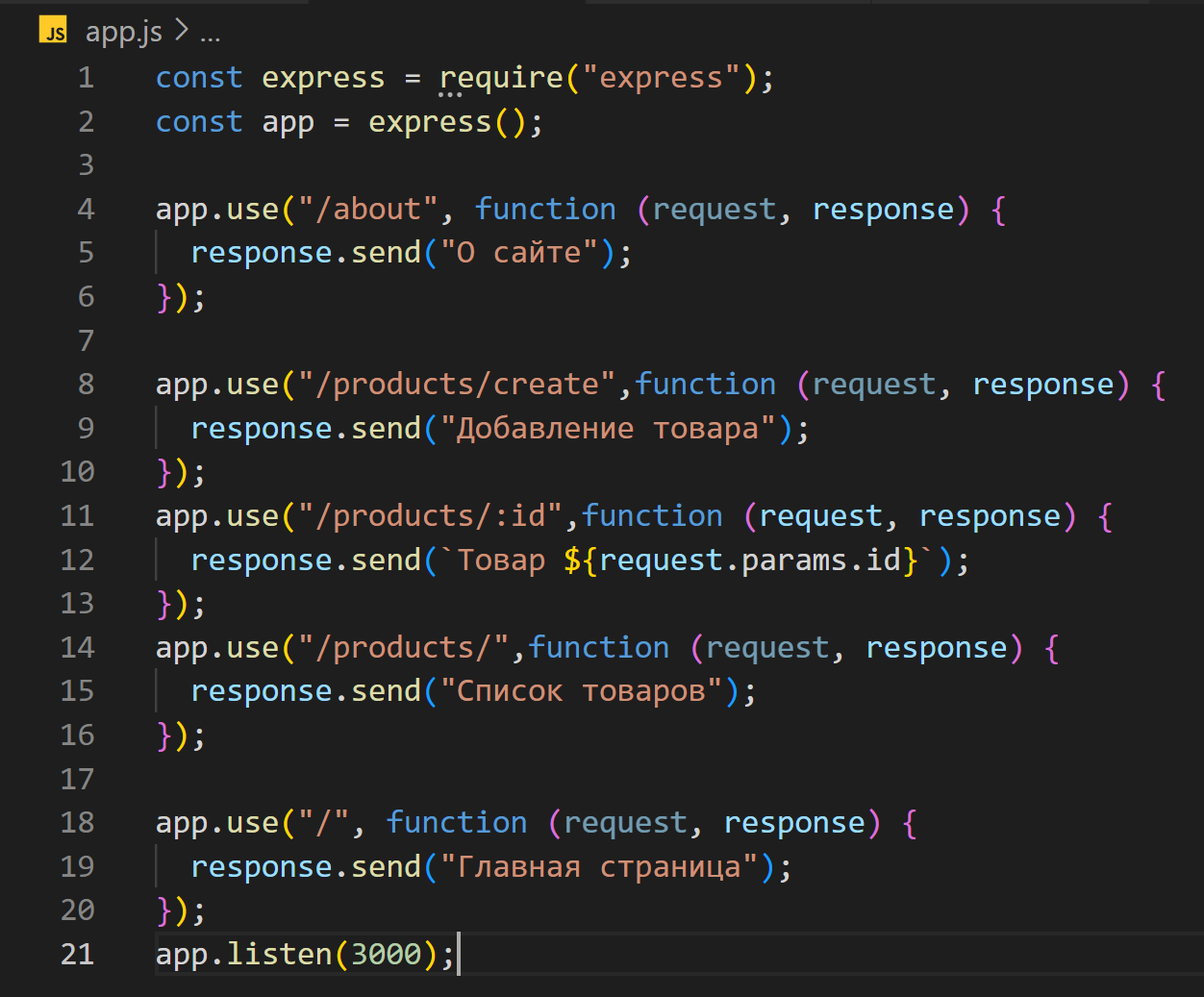
Или к примеру:





**Router**

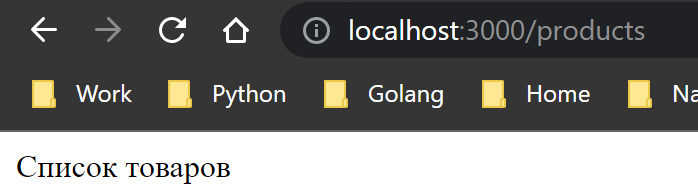
Router позволяет определить дочерние подмаршруты со своими обработчиками относительно некоторого главного маршрута. Например, определим следующее приложение:

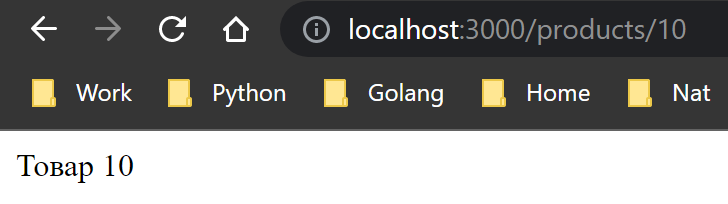


Здесь у нас есть пять маршрутов, которые обрабатываются различными обработчиками. Но три из этих маршрутов начинаются с "/products" и условно относятся к некоторому функционалу по работе с товарами (просмотр списка товаров, просмотр одного товара по id и добавление товара). Объект Router позволяет связать подобный функционал в одно целое и упростить управление им. Например, перепишем предыдущий пример с использованием объекта Router:



Здесь определен объект productRouter, который обрабатывает все запросы по маршруту "/products". Это главный маршрут. Однако в рамках этого маршрута может быть подмаршрут "/" со своим обработчиком, а также подмаршруты "/:id" и "/create", которые также имеют свои обработчики.

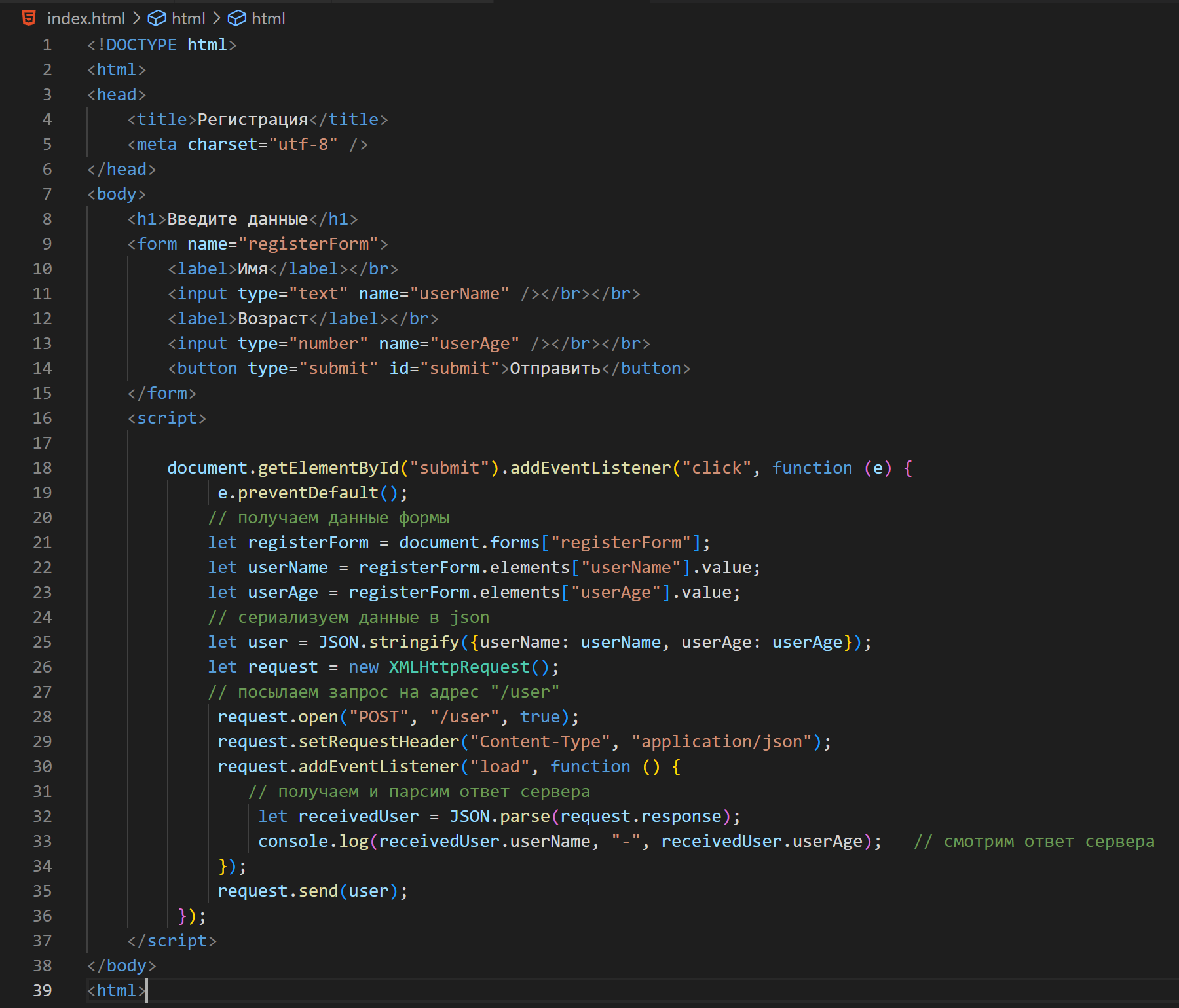




**JSON и AJAX**

JSON представляет один из самых популярных форматов хранения и передачи данных, и Express имеет все возможности для работы с JSON.

Пусть в папке проекта имеется файл index.html со следующим кодом:



Здесь определена форма с двумя полями для ввода имени и возраста пользователя. Но теперь с помощью обработчика нажатия перехватывается отправка этой формы. Мы получаем значения ее полей и сериализум в объект json, который затем отправляется на сервер с помощью ajax на адрес "/user".

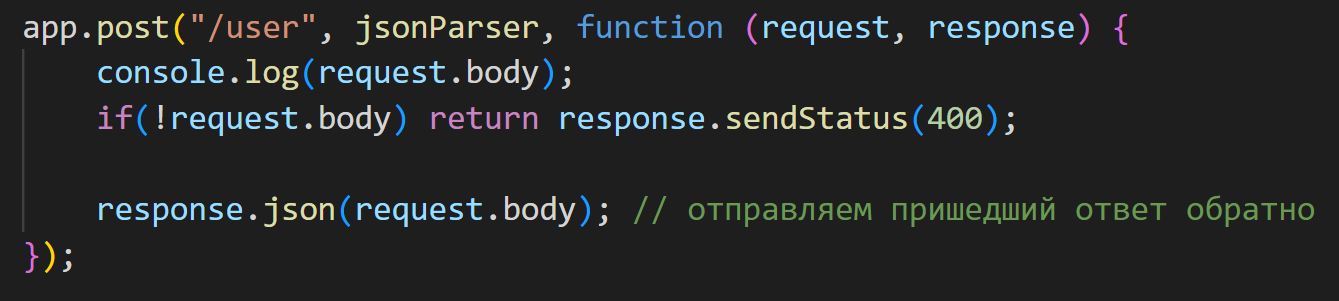
Далее определим в главном файле приложения - в app.js - код, который бы принимал отправленные данные:



Прежде всего для получения данных в формате json необходимо создать парсер с помощью функции json:



И поскольку с клиентом мы взаимодействуем через формат json, то данные клиенту отправляются с помощью метода response.json():

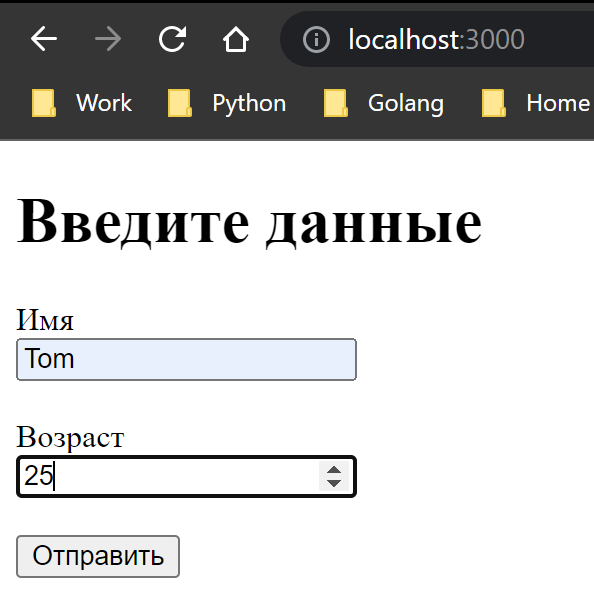


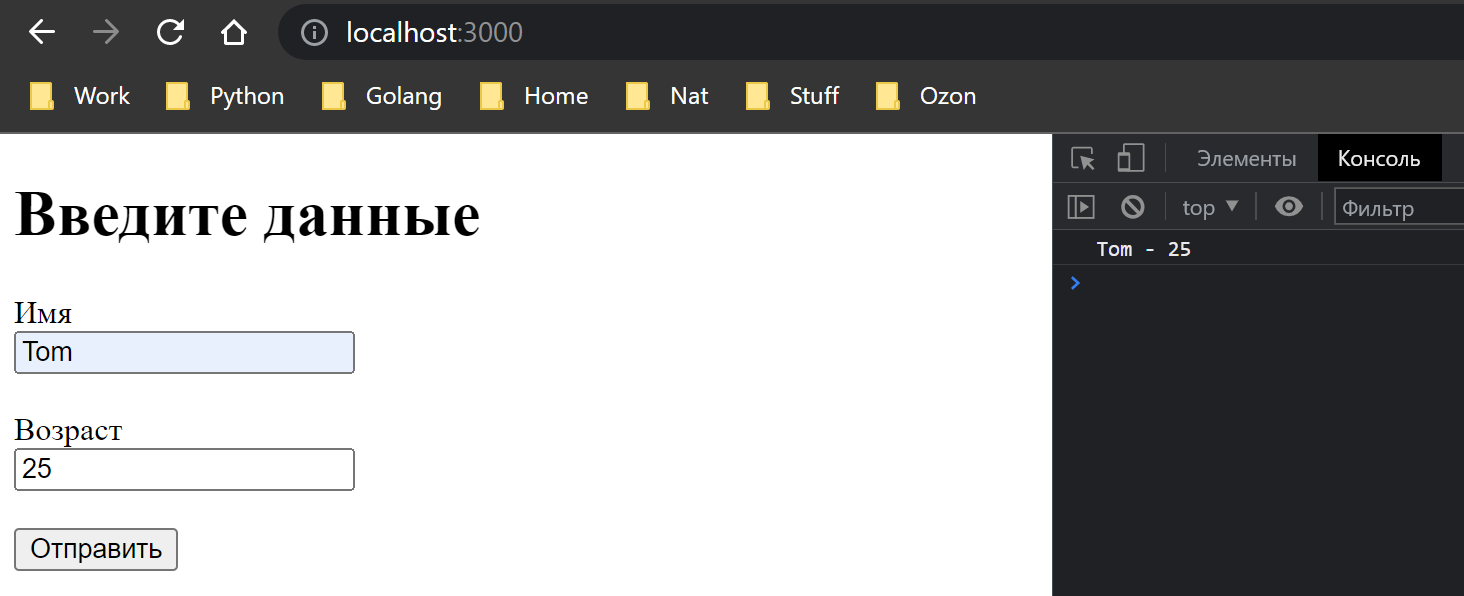
Следует отметить, что в принципе мы можем отправить объект и с помощью стандартного метода response.send(request.body). В реальности метод response.json() устанавливает для заголовка "Content-Type" значение "application/json", серилизует данные в json с помощью функции JSON.stringify() и затем отправляет данные с помощью response.send().

Для получения данных, как и в случае с формами, используются выражения типа request.body.userName, где request.body инкапсулирует данные формы, а userName - ключ данных.

При обращении к корню веб-приложения пользователю будет отправляться содержимое файла index.html с формой ввода данных.

Запустим приложение и обратимся к корню веб-сайта. Введем какие-нибудь данные и после отправки в консоли браузера отобразится ответ сервера:





**Этапы выполнения:**

1. ***Создайте новый проект, подключите пакет express. Сделайте скриншот файла package.json и консоли после установки пакета.***
2. ***Сделайте первую элементарную страницу при помощи express – для этого создайте объект приложения и пропишите get-запрос. В обработчик get-запроса вместо «Привет Express» напишите свое имя. Сделайте скриншот получившейся страницы.***
3. ***Теперь обработайте 4 маршрута: корневой, news, statistics и audios. Для каждой страницы напишите что-то уникальное, например новость для news, статистику для statistics и название трека для audios.***
4. ***Встройте в конвейер обработки запроса на любом этапе функцию middleware. Продемонстрируйте результат вывода в консоли.***
5. ***Остановите обработку на каком-то этапе при помощи next() и продемонстрируйте изменения.***
6. ***Сопоставьте функцию middleware с маршрутом /news. Продемонстрируйте код и получившуюся страницу.***
7. ***Отправьте ответ при помощи функции send(). Код разметки страницы определите в отдельном HTML-файле. На страницу выведите приветствие со своим именем. Продемонстрируйте скриншоты index.html, app.js и получившейся страницы.***
8. ***Отправьте пользователю любой статусный код кроме 404 при помощи функции sendStatus(). Продемонстрируйте код и страницу.***
9. ***Отправьте осмысленное сообщение при отправке статусного кода. Продемонстрируйте код и страницу.***
10. ***Отправьте пользователю статический файл html, продемонстрируйте код и результат.***
11. ***Обработайте данные по определенным маршрутам при помощи функций use, get, post, put, delete. Продемонстрируйте код и результат.***
12. ***В шаблонах адресов воспользуйтесь символами ?, +, \* и (). Продемонстрируйте код и результат использования символов.***
13. ***Реализуйте переадресацию на какой-нибудь сайт с технической документацией любой технологии на ваш выбор (кроме node.js). Продемонстрируйте код и результат.***
14. ***Реализуйту педирект относительно текущего пути, с которого производится редирект. Продемонстрируйте код и результат.***
15. ***Реализуйте переадресацию на адрес, который располагается уровнем выше. Продемонстрируйте код и результат.***
16. ***Реализуйте переадресацию на уровень выше. Продемонстрируйте код и результат.***
17. ***Реализуйте переадресацию на два уровня выше. Продемонстрируйте код и результат.***
18. ***Передайте ЛЮБЫЕ данные в приложение (кроме представленных в примере выше), используя параметры строки запроса. Продемонстрируйте код и результат.***
19. ***Таким же образом передайте массив и продемонстрируйте результат.***
20. ***Передайте сложный объект, состоящий из множества свойств, при помощи параметров строки запроса. Продемонстрируйте код и результат.***
21. ***Создайте форму для отправки данных, получите отправленные данные. Используйте СВОЕ имя и возраст, продемонстрируйте на скриншоте результат.***
22. ***Через коллекцию request.params получите все параметры маршрута, которые представляют именованные сегменты URL-адреса. Продемонстрируйте код и результат.***
23. ***Воспользуйтесь объектом Router для обработки маршрутов, добавьте подмаршруты в проект. Продемонстрируйте код и результат.***
24. ***Воспользуйтесь JSON для хранения и передачи данных в вашем проекте. Продемонстрируйте код, страницу и результат в консоли браузера – свое имя и возраст.***