[Модуль 7. Тестирование API с помощью Postman](https://lms.skillfactory.ru/courses/course-v1:SkillFactory+INTQAP+2022/course/" \l "block-v1:SkillFactory+INTQAP+2022+type@sequential+block@4b916ceb7c7e4ea48ea8805b45bf3655)

## 7.1. Что такое API. Виды API

**Программный интерфейс** означает совокупность средств, методов и правил взаимодействия с приложением или сервисом. Происходит это понятие от перевода значения английской аббревиатуры ***API*** — application programming interface — программный интерфейс приложений.

Все действия, которые пользователь совершает на сайте (нажатие кнопки, ввод текста и другие), сопровождаются отправлением запросов на сервер.

Пользователь нажал на кнопку → запрос отправился на сервер → сервер обработал запрос и выдал ответ → браузер обработал ответ и выдал пользователю результат.

Так работает трехуровневая архитектура клиент — сервер — приложение. Обратите внимание: **все данные хранятся на сервере**.

В каких случаях проводится тестирование *API*:

* + когда еще нет интерфейса приложения, только его «внутренняя часть» (*back*);
  + когда нужно локализовать ошибку — понять, на чьей она стороне, чтобы передать нужному разработчику. Это ускорит процесс исправления бага; когда нужно проверить негативные тесты;
  + когда нужно ускорить тестирование — промежуточные шаги быстрее выполнить через *API*, чем через интерфейс.

Основной объект, к которому происходит обращение по *HTTP*-протоколу, — это **ресурс**.

Ресурсом может быть что угодно: файл, логический объект или нечто абстрактное. Ресурс, к которому производится обращение по *HTTP*-протоколу, идентифицируется по пути к нему, передаваемому в стартовой строке запроса. Операция, которую сервер выполняет по запросу клиента, идентифицируется методом.

**Структура запроса HTTP-протокола**

Запрос состоит из:

* 1. Стартовой строки.
  2. Заголовков запроса.
  3. Тела сообщения, передаваемого вместе с запросом.

**Стартовая строка запроса** включает метод, путь запроса и версию протокола, например:

GET 130.193.37.179/app/pets HTTP/1.1

В приведённой строке:

* + GET — **метод**;
  + 130.193.37.179 — **базовый URL** ресурса (в указанном выше примере представлен в виде *IP*-адреса, но чаще встречается в виде привычного адреса, например, *skillfactory.ru/*);
  + /app/pets — **URN**;
  + HTTP/1.1 — **тип и версия протокола**, с использованием которого клиент отправляет запрос и ожидает ответ от сервера.

**Метод** — последовательность символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. *HTTP*-запрос обязательно должен содержать какой-то метод.

Всего их девять:

* 1. GET,
  2. POST,
  3. PUT,
  4. OPTIONS,
  5. HEAD,
  6. PATCH,
  7. DELETE,
  8. TRACE,
  9. CONNECT.

В данном модуле мы рассмотрим самые распространённые — GET и POST. С большей частью остальных методов мы познакомимся в следующем модуле.

Чтобы в будущем не путаться в том, какое действие осуществляет тот или иной метод, обязательно сохраните [в закладки памятку](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods).

Чуть позже мы рассмотрим на практике, как методы управляют запросами.

**Базовый URL** (*Uniform Resource Locator* — унифицированный указатель ресурса) — простыми словами, это адрес ресурса в сети, определяет местонахождение и способ обращения к нему. Типичный пример *URL*: <https://skillfactory.ru/>.

**URN** (*Uniform Resource Name*) — имя ресурса в сети, определяет только название ресурса, но не говорит, как к нему подключиться.

Вместе (но необязательно!) *URL* и *URN* представляют собой **URI** (*Uniform Resource Identifier* — унифицированный (единообразный) идентификатор ресурса. Это символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и так далее.

*URI* является либо *URL*, либо *URN*, либо одновременно обоими. Для того, чтобы понять это непростое определение, взгляните на картинку ниже:

**Версия протокола** — определяет, в соответствии с какой версией стандарта *HTTP* составлен запрос. Указывается как два числа, разделённых точкой (например, 1.1).

**Заголовки запроса** характеризуют тело сообщения, параметры передачи и содержат прочие сведения. Заголовки представлены парой поле-значение, разделённых двоеточием.

Например: *User-Agent*: *Mozilla*/5.0 (*Windows NT* 10.0; *Win*64; x64) *AppleWebKit*/537.36 (*KHTML*, *like Gecko*) *Chrome*/107.0.0.0 *Safari*/537.36

Заголовок *User-Agent* является обязательным и должен присутствовать в каждом запросе, так как указывает программное обеспечение клиента и его характеристики.

Например, во вкладке *Network* инструментов разработчика браузера (*DevTools*) можно увидеть данный заголовок в любом запросе:

**Тело сообщения, передаваемое в запросе,** — объект, с которым ассоциирован данный запрос. Если стартовая строка и заголовки всегда присутствуют в запросе, то тело сообщения может отсутствовать. Например, оно, как правило, отсутствует в GET-запросах.

**Структура ответа HTTP-протокола**

Ответ *HTTP*-протокола также состоит из стартовой строки, заголовков и тела сообщения, передаваемого вместе с ответом.

Стартовая строка ответа состоит из:

* + версии протокола;
  + кода состояния;
  + его расшифровки.

Например, стартовая строка ответа может быть:

HTTP/1.1 200 OK

Согласно стандарту *RFC*7231, **код состояния** — трехзначный цифровой код, представляющий результат выполнения запроса. Коды состояния могут попадать в одну из пяти категорий определенного формата:

* + **1xx** — информационные;
  + **2xx** — коды, относящиеся к категории успешных, означающие успешное выполнение запроса;
  + **3xx** — коды, относящиеся к категории перенаправления, означающие, что клиент должен сделать повторный запрос по другому URI, для успешного выполнения операции;
  + **4xx** — коды, обозначающие, что была ошибка на стороне клиента;
  + **5xx** — коды, обозначающие, что была ошибка на стороне сервера.

**Что такое REST**

***REST*** — аббревиатура от ***RE****presentational****S****tate****T****ransfer* — архитектурный стиль проектирования распределенных систем, таких как *web*-службы.

А что же такое архитектурный стиль? **Архитектурный стиль** — это набор согласованных ограничений и принципов проектирования, позволяющий разработчику добиться определённых свойств системы.

Другими словами, *REST* — это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

Системы, поддерживающие *REST*, называются *RESTful*-системами.

*REST*-архитектура базируется на системе свойств и ограничений, у неё довольно много особенностей. Отметим наиболее важные:

* 1. Архитектура приложения должна быть типа клиент-сервер. *HTTP-*протокол также работает только в клиент-серверных системах.
  2. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и, если необходимо, идентификации клиента. Состояние сессии сохраняется на стороне клиента. Это также согласуется с концепцией *HTTP*-протокола.
  3. Наличие кэша. Клиенты и промежуточные узлы могут кэшировать ответы от сервера (напоминаем, **кэширование** — это процесс сохранения данных локально, который позволяет быстрее получить к ним доступ при будущих запросах).
  4. Слои системы. В *REST-*системе клиент не может точно определить, взаимодействует ли он напрямую с сервером или между ними есть промежуточный узел такой, как прокси-сервер или туннель. Добавление промежуточного узла не должно приводить к обновлению кода приложения на клиентской или на серверной частях.
  5. Код по требованию. *REST* может позволить расширить функциональность клиента за счёт загрузки кода с сервера, например *Java Script*.
  6. Унифицированный интерфейс. Данное ограничение является фундаментальным требованием *REST-*сервисов. Унифицированный интерфейс имеет 4 признака:

Идентификация ресурса. Каждая единица информации однозначно определяется *URL* — это значит, что *URL* по сути является первичным ключом для единицы данных. Этот идентификатор не меняется в зависимости от изменения состояния ресурса.

Управление ресурсом через представления. Представление в *REST* используется для выполнения действий над ресурсами. Представление являет собой текущее состояние ресурса, передаваемое в теле запроса или ответа. Формат представления может быть *XML*, *HTML*, *JSON*, *YAML* или даже обыкновенный текстовый формат.

Запрос и ответ должны хранить в себе всю необходимую информацию для их обработки. Не должно быть дополнительных сообщений для обработки одного запроса. Другими словами, отсутствие состояния, сохраняемого между запросами к ресурсам.

Состояние ресурса передается через содержимое запроса или ответа, параметры строки запроса, заголовки запросов и ответов. В случае необходимости ссылки на ресурс могут содержаться в теле ответа (или заголовках) для формирования *URI* или извлечения запрошенных объектов.

Перечисленные выше признаки выглядят сложно из-за того, что они изложены в виде сухой теории. Однако рекомендуем зафиксировать информацию в удобном для вас формате и вернуться к ней позже, после прохождения этого и следующего модуля. С учётом полученной практики понимание изложенных выше особенностей архитектуры *REST* будет более полным.

Прежде чем мы обобщим полученную информацию, познакомимся ещё с одним определением, которое мы используем при изучении запросов в архитектуре *REST* — *API*.

**API** (*Application programming interface*) — программный интерфейс приложения для взаимодействия с другими приложениями. Ещё немного простых слов для большего понимания термина!

Если программу (модуль, библиотеку) рассматривать как чёрный ящик, то *API* — это набор «ручек», которые доступны пользователю данного ящика, он может их вертеть и дёргать. Программные компоненты взаимодействуют друг с другом посредством *API*.

И вот теперь, когда появилось представление о *REST* и об *API*, пора соединить оба термина в один.

**REST API** — архитектурный стиль разработки *API* веб-приложений или компонентов распределённого приложения с использованием протокола *HTTP*.

Это программный интерфейс управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как *URL*.

Для реализации *REST API* используются в основном **4 метода протокола *HTTP****:*

* 1. *GET* — для получения информации о ресурсе.
  2. *POST* — для добавления нового ресурса.
  3. *PUT* — для обновления существующего ресурса.
  4. *DELETE* — для удаления существующего ресурса.

Таким образом, реализуются 4 действия концепции ***CRUD***:

* + ***C****reate* — метод *POST*
  + ***R****ead* — метод *GET*
  + ***U****pdate* — метод *PUT*
  + ***D****elete* — метод *DELETE*

**Кое-что о SOAP**

Стоит заметить, что *REST API* — это не единственный программный интерфейс приложения. Несмотря на то, что в нашем курсе мы будем работать только с *REST API,*необходимо упомянуть про протокол *SOAP*, о котором любят спрашивать на собеседованиях.

***SOAP*** — это протокол обмена структурированными сообщениями в распределенной среде. [Стандарт](https://www.w3.org/TR/soap12/) определяет формат сообщений, базирующийся на *XML*. *SOAP* работает поверх *HTTP*-протокола, хотя могут использоваться и другие протоколы прикладного уровня, такие как *FTP*, *SMTP*.

**Основное отличие SOAP от REST**

В *SOAP* вся логика доступа к ресурсу содержится в сообщении формата *XML*, передаваемом в теле запроса и ответа *HTTP*-протокола, который *SOAP* использует как транспорт. Поэтому сообщения, передаваемые по *SOAP-*протоколу, занимают гораздо больший объём.

*SOAP* поставляется со строгими правилами и расширенными функциями безопасности, поэтому широко используется для веб-сервисов уровня предприятия, которые требуют высокой степени безопасности и сложных транзакций.

Более высокая сложность*SOAP* требует большей пропускной способности и ресурсов, что может привести к снижению времени загрузки страницы. *REST* был создан для решения проблем *SOAP*. Поэтому у него более гибкая архитектура. Он состоит только из простых рекомендаций и позволяет разработчикам реализовывать рекомендации по-своему.

*REST* допускает различные форматы сообщений:

* + *HTML*,
  + *JSON*,
  + *XML,*
  + простой текст.

В то время как SOAP допускает только XML.

**Эндпоинты** (*endpoints* или конечные точки) указывают, как получить доступ к ресурсу, а метод указывает разрешенные взаимодействия (такие как *GET*, *POST* или *DELETE*) с ресурсом.

Эндпоинт состоит из метода запроса, базового URL и URN. Пример, эндпоинта:

*GET* <http://51.250.89.234/app/pets>

Подробнее о том, как работать с таким эндпоинтом, мы разберёмся позже.

**Зачем тестировать API**

Многие из вас могут задать вопрос: «Зачем нужно тестирование через *API,* если с системой мы также можем взаимодействовать и через *web*-интерфейс*?»* Отвечаем.

1

Тестирование *API* даёт возможность протестировать работу кода приложения даже тогда, когда *UI* ещё не написан *frontend*-разработчиком.

2

Когда мы тестируем *API*, то понимаем, что тестируем код, написанный *backend*-разработчиком. Соответственно, разделяем области тестирования на:

* + тестирование *UI* (*frontend*);
  + тестирование архитектурного кода приложения (*backend*).

3

Тесты на *API* можно запускать каждый раз, когда новая сборка приложения разворачивается на тестовом сервере перед ручным тестированием. В этом случае мы можем быть уверены, что если тесты по *API* были выполнены успешно, то ошибки, возникающие при ручном тестировании, с большей степенью вероятности будут в клиентской части (*UI* — *frontend*).

4

Тестируя *API*, мы сможем проверить намного больше вариаций параметров, нежели при тестировании через *UI*.

Например, мы можем создать файл с заранее написанными тестовыми данными для различных полей и прогнать его за несколько секунд. Тогда как ручное тестирование этих полей каждый раз отнимает большое количество времени. Если по текстовым полям стоит ограничение на бэке в 255 символов, а таких полей штук 30, то проверка каждого поля через *UI* заняла бы порядка 25-30 минут.

Но представим, что эти поля нужно проверить не только на максимальную длину, а ещё на ввод определённых символов, цифр, *SSL*-инъекций и тому подобное. Такого рода проверки отняли бы у инженера целый день.

Однако заранее написанный файл с проверками ускорит время последующих циклов тестирования, сократив количество потраченного времени до нескольких секунд.

## 7.2. POSTMAN

## ****Рассмотрим секцию ответа на выполненный запрос****

Секция состоит из 4-х закладок:

* + *Body*,
  + *Cookies*,
  + *Headers,*
  + *Test Result*.

Также блок содержит информацию о результате выполнения запроса: код ответа, время выполнения запроса и размер самого ответа.

При наведении курсора мыши на каждое из этих значений появится информация о коде ответа, детализация времени выполнения запроса и детализация размеров запроса и ответа соответственно. В данный момент нас интересуют только вкладки *Body* и *Headers*.

Во **вкладке Body** этого окна будет выведено содержимое тела ответа. Ответ на данный запрос возвращается в *HTML-*формате. Установите *Auto*, если хотите, чтобы *Postman* автоматически подбирал форматирование информации переданной в теле ответа.

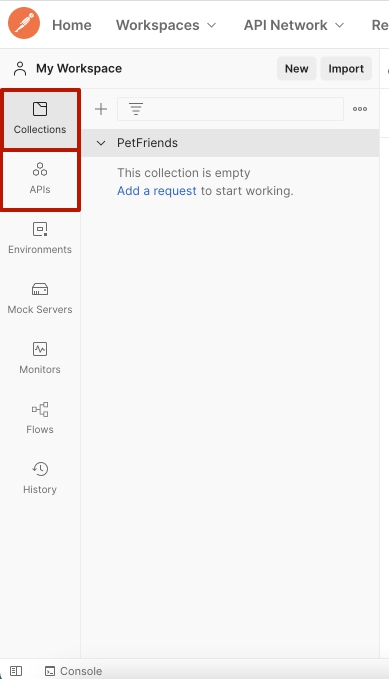
Откройте приложение «Дом питомца» в браузере с панелью инструментов разработчика. В списке запросов найдем запрос с *URI* /*app*, кликнем по нему, чтобы отобразилась панель параметров запроса, выберем вкладку *Response* (ответ). Вы можете видеть, что отображаемый *HTML-*файл совпадает с тем, что возвращен во вкладке *Body* секции ответа *Postman*.

**Вкладка Headers** (заголовки) отображает заголовки, переданные с ответом. Заголовки запроса/ответа *HTTP-*протокола — это пара типа «параметр-значение». Пиктограммы информации, отображаемые рядом с каждым из заголовков дают информацию о заголовке при наведении курсора мыши. Перейдите на вкладку браузера, где открыто приложение «Дом питомца» и инструменты разработчика с вкладкой сеть. Найдите запрос *pets*, нажмите по нему и сравните переданные заголовки ответа с теми, что переданы в *Postman*. Вы заметите, что они идентичны.

**Левая панель Postman**

Панель имеет 7 вкладок:

* + *Collections*,
  + *API's*,
  + *Environments*,
  + *Mock Services*,
  + *Monitors*,
  + *Flows*,
  + *History*.



Рассмотрим две из них, которые будут важны для нас на данном этапе изучения *Postman*.

**Вкладка history** содержит историю выполнения запросов. Выполните ещё раз введённый запрос, и он появится в истории выполненных запросов.

Перейдите на **вкладку Collections** левой панели. В ней отображается имя коллекции, введённой при конфигурации запроса. Разверните имя коллекции. Видите запрос, который вы настраивали и выполняли?

## 7.3. Разбор GET-запросов с помощью DevTools

Первый запрос отправляется по URL, введённому в строке браузера, браузер получает ответ с **кодом 302**. Этот код означает, что ресурс найден и браузер должен сделать второй запрос по URI/URL, указанному в заголовке Location ответа.

Откроем информацию о первом запросе, перейдём к разделу заголовков ответа. Найдём заголовок Location. Его содержимое — это URI /app. Именно по такому URI браузер должен сформировать следующий запрос. Этот запрос идёт следующим в списке запросов. Откроем его содержимое. Можно видеть, что код ответа на этот запрос 200. Значит, запрос выполнен успешно.

Нажмём на вкладку Response, чтобы увидеть тело ответа, содержащее загруженный контент в HTML-формате. Если мы взглянем ещё раз на содержимое HTML, то увидим под тегом link обращение к файлу каскадных стилей — app.css, файлу фавикона — favicon.ico, а внизу HTML-файла, под тегом script, ссылку на файл js-скрипта — app.js. Браузер выполнит GET-запросы, чтобы получить все эти три файла. Вы можете видеть эти три запроса во вкладке network.

## ****Какую информацию мы можем получить по REST API-запросам****

Вернитесь к *DevTools* приложения «Дом питомца». Если вы закрыли инструменты разработчика, то откройте их, перейдите во вкладку *Network* и перезагрузите страницу приложения.

Обратим внимание на четыре запроса, следующие за запросом, загружающим шрифты:

* + ?page=1&page\_size=6
  + species/
  + breed/
  + gender/

Рассмотрим первый из четырех запросов — ?page=1&page\_size=6

Это *HTTP GET-з*апрос. Однако формат *URL* этого запроса может слегка смутить, так как в нем после знака ? передаются два параметра:

* 1. page = 1
  2. page\_size=6

Является ли это *REST API GET-*запросом? Да, несомненно, является. *REST API-*запросы могут также передавать параметры в строке запроса после ?. Общий шаблон *URL* для *REST API-*запросов может быть записан так:

/URI?option\_1=a&option\_2=b&…..&option\_k=z

Где option\_1=a, option\_2=b ... option\_k=z — это возможные параметры запроса и присвоенные им значения.

**Выделим второй запрос** — /species.

Справа откроется окно параметров запроса. Во вкладке *Headers* мы видим, что это *HTTP GET-*запрос. Обратите внимание на *URL* запроса. Его формат подходит под определение *REST API GET-*запроса. Перейдем во вкладку *Response*: видим ответ в формате *JSON*. Если перейти во вкладку *Preview*, увидим отформатированный ответ.

Нетрудно догадаться, что данный запрос — это *REST API GET-*запрос, выводящий информацию о видах животных. Изучив внимательно ответ, можно понять, что каждый вид животного имеет идентификатор, наименование и некий код.

## ****Исследуем формируемые HTTP GET-запросы****

## 7.4. Postman GET-запросы

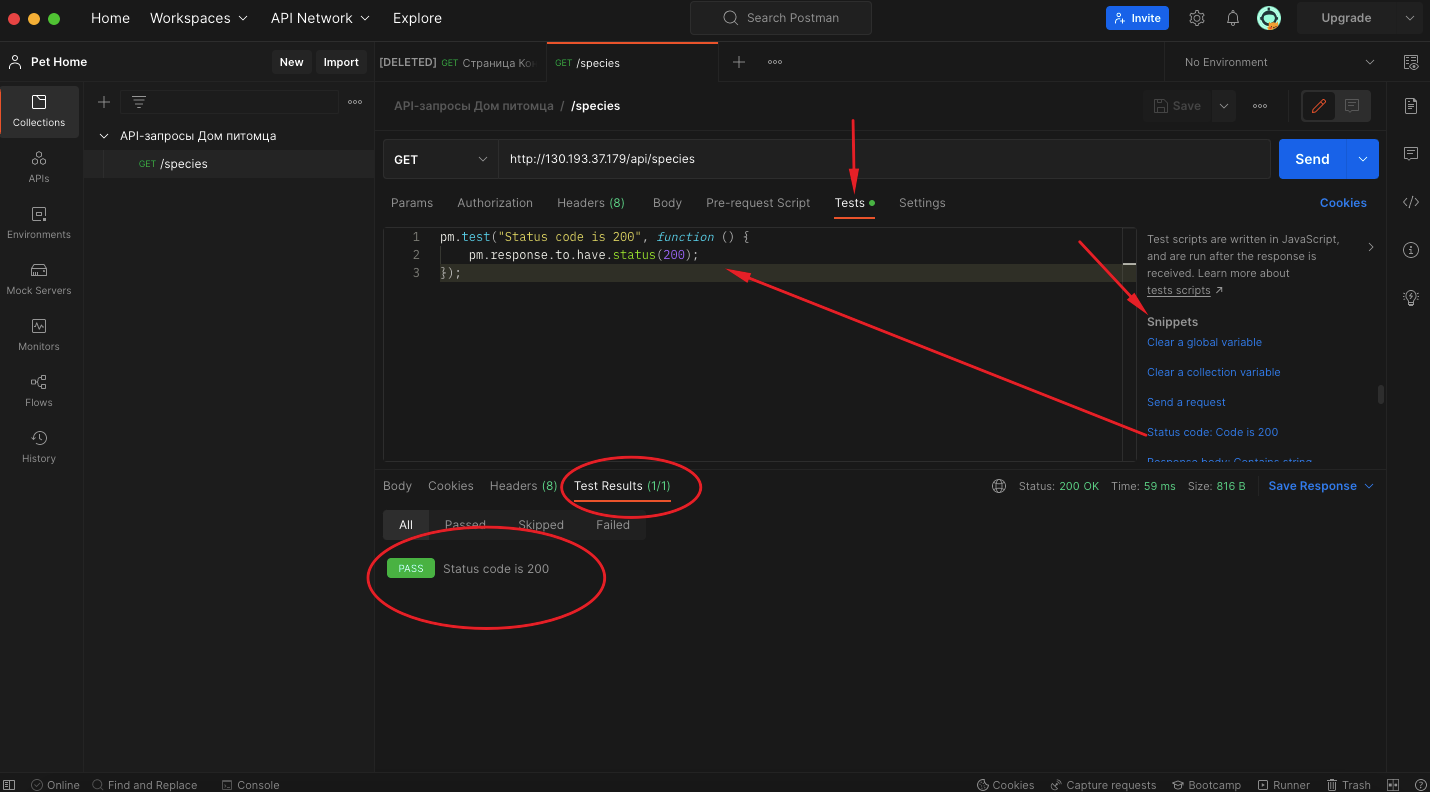
**Как превратить запросы в тесты**

Теперь пора освоить то, как работают тесты в приложении *Postman*. Перейдите во вкладку *Tests* в секции конфигурации запроса /species. Здесь в этой панели, умея программировать на *JavaScript,* возможно написать код, который будет проверять код состояния данного запроса. Например, можно сделать проверку на то, что код состояния на данный запрос будет 200.

Так как не все умеют программировать на *JavaScript*, то *Postman* имеет набор заранее написанных скриптов на этом языке, называемых *snippets*, делающих различные проверки. Найдите скрипт с названием *Status code*: *code is* 200 и кликните по нему. Код скрипта будет скопирован в содержимое вкладки *Tests*. Данный скрипт проверяет, что код ответа равен 200. Сохраните запрос и запустите его снова.

**Обратите внимание**, что рядом с наименованием вкладки *Test results* секции ответа, появилась информация об успешных результатах запуска этого теста. Откройте эту вкладку. В ней вы увидите отчет об успешном запуске этого теста.

На изображении ниже показан составленный тест и результат его выполнения.



## ****Запросы, в строке URL которых передаются параметры****

**Как улучшить тесты**

Сейчас в нашей коллекции 3 теста. Давайте улучшим наши тесты еще раз. Вы уже обратили внимание, что в *URL* наших тестов есть постоянная часть — это *IP-*адрес сервера. Объявим ее как переменную baseURL и внесем соответствующие коррективы в *URL* запросов.

Для этого:

* 1. Выделим коллекцию.
  2. Откроем контекстное меню и выберем *Edit* для редактирования свойств коллекции.
  3. В свойствах коллекции перейдем на вкладку *Variables*.
  4. Добавим переменную *baseURL*в колонку *Variable* и значение *IP-*сервера приложения «Дом питомца» в колонку *initial value*и нажать на поле*current value*, вставив в него тот же *IP*-адрес.

**Как улучшить тесты**

Сейчас в нашей коллекции 3 теста. Давайте улучшим наши тесты еще раз. Вы уже обратили внимание, что в *URL* наших тестов есть постоянная часть — это *IP-*адрес сервера. Объявим ее как переменную baseURL и внесем соответствующие коррективы в *URL* запросов.

Для этого:

* 1. Выделим коллекцию.
  2. Откроем контекстное меню и выберем *Edit* для редактирования свойств коллекции.
  3. В свойствах коллекции перейдем на вкладку *Variables*.
  4. Добавим переменную *baseURL*в колонку *Variable* и значение *IP-*сервера приложения «Дом питомца» в колонку *initial value*и нажать на поле*current value*, вставив в него тот же *IP*-адрес.

Зачем нужны такие манипуляции, наверняка спросите вы. Представьте ситуацию: в ходе разработки вы переехали на новое тестовое окружение, у которого другой URL. Очевидно, что проще поменять значение URL в одной переменной, чем в сотне API-тестов. Или, что встречается гораздо чаще, на проекте может быть несколько окружений. Создав набор тестов на одном окружении, очень просто создать копию коллекции для другого окружения.

## ****Прогон запросов внутри коллекции****

Выберите коллекцию, кликнув на неё. Справа кликните на значок запуска. Откроется диалог запроса группового запуска тестов коллекции.

Кликните на кнопку Run. Откроется диалог на выбор тестов коллекции, которые нужно будет выполнить. В этом диалоге можно будет выбрать тесты коллекции, установить порядок их запуска, количество итераций и задержку между итерациями.

Всегда старайтесь перед прогоном отмечать галочкой пункт Save response для сохранения тел запросов при прогоне.

## 7.5. REST API POST-запросы

Запустите браузер, зарегистрируйтесь на сайте «[Pet Friends](http://petfriends.skillfactory.ru/login)», выйдите из учётной записи. Затем откройте панель разработчика, откройте вкладку «Сеть», введите в адресной строке адрес страницы авторизации приложения «[Pet Friends](http://petfriends.skillfactory.ru/login)» и нажмите Enter. Браузер откроет соответствующую страницу приложения, на которой вы увидите два поля для ввода текста, а именно «Электронная почта» и «Пароль».

Заполните эти поля своими учетным данными и проверьте консоль разработчика. На данном этапе во вкладке Network присутствует лишь знакомый нам уже GET-запрос. Это говорит о том, что до нажатия кнопки «Войти» мы не посылаем никаких данных на сервер.

Разберём кратко, что произошло на самом деле.

Первый запрос login отправляется по URL, введённом в строке браузера. Браузер получает ответ с кодом 302. Этот код означает, что ресурс найден. И клиент, то есть браузер, должен сделать второй запрос по URI/URL, указанному в заголовке Location ответа. Именно по такому URI должен быть следующий запрос браузера. Этот запрос идёт следующим в списке запросов. Тут всё как в рассматриваемом ранее (юнит 3 данного модуля) GET-запросе.

Однако на этот раз стоит обратить внимание на то, что последующий GET-запрос связан с предыдущим POST-запросом не только заголовком Location, но и заголовком Cookie.

**Cookie** — это небольшой фрагмент данных, который сервер отправляет браузеру пользователя. Браузер может сохранить этот фрагмент у себя. И отправлять на сервер с каждым последующим запросом. Это, в частности, позволяет узнать, с одного ли браузера пришли несколько запросов (например, для аутентификации пользователя).

Кроме того, если вы попробуете ввести некорректные данные для входа в систему, то получите код ответа отличный от 302, а в списке заголовков ответа не будет ни Cookie, ни Location. Вернёмся к самому POST-запросу.

## ****Для чего нужны POST-запросы****

**POST-запросы** предназначены для передачи определенной информации в своем теле. В рассмотренном примере с авторизацией такой информацией будет значение полей формы «Электронная почта» и «Пароль». Выберите в консоли разработчика вашего браузера POST-запрос (он первый в списке), и обратите внимание на секцию Form Data во вкладке Payload — это те самые данные, которые вы отправили на сервер POST-запросом.

Вы можете нажать на кнопку «view source», чтобы увидеть данные в том виде, в котором ваш браузер отправляет их серверу. Этот формат может быть различным, но этот вопрос мы подробнее рассмотрим позже.

Также стоит обратить внимание на Response Headers во вкладке Headers, а точнее на заголовок Set-Cookie. Этот заголовок содержит в себе уникальную строку session, которая будет использоваться во всех последующих запросах. Ответ на POST-запрос (в отличие от GET) не всегда имеет тело, но мы всегда можем понять результат его работы.

Согласно [спецификации RFC-7231](https://tools.ietf.org/html/rfc7231#section-4.3.3) POST в основном используется для следующих функций:

* 1. Предоставление блока данных, такого как поля, введенные в форму HTML, для процесса обработки данных.
  2. Публикация сообщения, например, на доске объявлений, в группе новостей, списке рассылки, блоге и так далее.
  3. Создание нового ресурса.
  4. Добавление данных к существующим.

Кроме передачи данных о заполнении форм, частым использованием POST-запросов является загрузка файлов на сервер, в этом случае содержимое может быть передано в теле запроса, а соответствующие заголовки могут говорить о его содержимом.

Теперь рассмотрим, как метод POST применяется в REST-архитектуре веб-приложений. Известно, что метод POST может покрывать все действия, связанные с созданием удалением или изменением объектов в приложении. Однако классическим паттерном REST является более четкое разделение действий по методам. POST чаще всего используется для создания новых сущностей в системе, будь то добавление нового животного в каталог, или, как в нашем примере с авторизацией, создание новой активной сессии пользователя.

Запрос с таким методом не является безопасным или идемпотентным. В результате выполнения идентичных POST-запросов предоставляются сильно похожие, но не идентичные данные.

Например, рассмотрим некое приложение интернет-магазина. Запрос POST /api/item, в форме которого перечислены атрибуты товара, создаст новый товар, например, с кодом 3050. Если мы повторно выполним такой же запрос, в котором будут указаны такие же атрибуты, то будет создан такой же товар, но, например, с кодом 3051.

**Идемпотентными** запросами называются те запросы, повторное выполнение которых возвращает тот же результат, который был получен ранее. Например, GET-запрос — типичный идемпотентный запрос.

Тела ответа в данном случае не предусмотрено, как и при авторизации. Отличие этого случая от предыдущего примера лишь в первоначальном запросе и в Location, обратите внимание, что сейчас он указывает на главную страницу, это делается, чтобы, не запрашивая данные, подтвердить, что ресурс найден, и можно продолжать работу.

Рассмотрим более сложный случай — создание нового объекта, содержащего в себе не только текст или состояние, но и картинку, добавим нового питомца. Для этого, находясь в системе как авторизованный пользователь, мы перейдем [на страницу](http://petfriends.skillfactory.ru/my_pets). Здесь мы нажмем «Добавить питомца», заполним все поля и загрузим картинку, после чего нажмем «Добавить». Во вкладке network панели разработчика вы увидите следующую информацию:

Обратите внимание, что формат *URL* и метод соответствуют требованиям *REST API* для создания ресурса. Также стоит обратить внимание на следующий заголовки запроса:

* + *Content-Length* — имеет внушительную величину, что может говорить нам о том, что файл действительно был передан серверу. Данное поле формируется клиентом (в данном случае браузером) автоматически и отображает объём переданной информации.
  + *Content-Type* — говорит нам о том, что тело запроса является составным, и эти части разделены специальной строкой, которая указана в атрибутах этого заголовка. Части — это параметры формы и двоичное содержимое файла.

Для лучшего понимания давайте посмотрим на тело этого запроса:

Здесь вы можете увидеть Content-Disposition по каждой части тела запроса. Заголовок и Content-Disposition имеют значения, которые определяют тип передаваемого контента в каждой части тела запроса.

## 7.6. POSTMAN POST-запросы

Вернёмся в Postman. Для начала мы укажем URL для создаваемого запроса, сменим метод на POST (внимание на выпадающий список в верхней строке рядом с адресом сайта) и попробуем отправить запрос, как есть:

Мы получили ответ с кодом 400 и сообщением Bad Request. Это означает, что запрос сформировали не правильно, однако к серверу подключились, значит, мы на верном пути.

Думается, уже стало очевидно, что мы не добавили тело для нашего запроса, а в случае авторизации оно является обязательным. Найдите в секции конфигурации запроса **вкладку Body**, в ней включите **режим form-data**, перейдите на неё и добавьте туда ключ/значение для email и pass из нашего запроса. А затем нажмите на кнопку Send, чтобы отправить запрос.

Как видите, запрос прошел успешно. В секции ответа на наш запрос мы по умолчанию видим HTML-код веб-страницы, полученной после успешной авторизации и код состояния 200. Если в этой секции вы перейдете во вкладку Cookies, то увидите уникальную строку для текущей сессии, аналогичную рассматриваемой в предыдущем уроке.

Стоит сказать, что нам могло потребоваться повторить все заголовки нашего запроса. Настоятельно рекомендуем проверять и дополнять их при реальном тестировании запросов. В данном же случае достаточно было использовать лишь те заголовки, которые Postman создал автоматически.

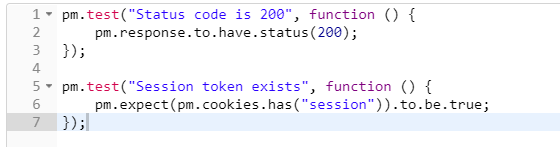
Обратите внимание: Postman, в отличие от веб-браузера, как бы совместил в себе сразу 2 запроса — POST /login и GET /all\_pets, отсюда и HTML-страница в теле ответа, и 200 код вместо 302. Так произошло потому, что в нашем POST-запросе явно инициализируется последующий запрос. В заголовках его ответа есть Location, а его первоначальный код ответа 302.

**Ремарка-совет**:  несмотря на то, что мы видим сообщение FOUND и понимаем, что ресурс был найден, нужно помнить о кодах, начинающихся с тройки (3хх) и означающих перенаправление. Убедиться в том, что действительно сделали все правильно, можно, открыв консоль Postman, нажав (Ctrl + Alt + C).

Добавьте тест на код состояния 200 к запросу, который был рассмотрен выше. В предыдущем модуле мы уже рассматривали, как это правильно выполняется.

Однако в этот раз мы пойдём немного дальше и добавим не 1, а 2 теста. Вторым тестом для этого запроса будет проверка наличия того самого session в разделе cookies. Иными словами, в тесте мы будем ожидать, что наличие параметра session в cookies истинно (true).

Запись такая:



Как видите, кода здесь очень мало. Можете переиспользовать эту конструкцию для проверки соответствующих параметров в своих тестах.

Рассмотрим строку 6 подробнее. Переводя с английского, мы буквально получим следующее:

«Postman ожидает, что утверждение (Postman в разделе cookies имеет параметр session) будет правдой».

Более подробно рассмотреть принципы и примеры построения тестов по шаблонам можно, посетив [раздел справки по написанию скриптов.](https://learning.postman.com/docs/postman/scripts/test-scripts/)

### **Задание 7.6.2**

Далее, следуя плану предыдущего занятия, мы должны рассмотреть регистрацию нового пользователя и повторить её при помощи инструмента Postman.

Эта задача является идентичной той, с помощью которой мы проверяли авторизацию. Поэтому проделайте это задание сами. На всякий случай напоминаем ход выполнения задания:

1. Получить эндпоинт по созданию нового пользователя из DevTools.
2. Из вкладки Payload взять обязательные для выполнения запроса ключи.
3. Сформировать запрос в Postman.

Помните, что для каждого нового запуска теста поля name и email обязаны быть уникальными!

Обратимся к наиболее интересному примеру этого урока — добавление нового питомца при помощи POST-запроса и инструмента Postman.

Перейдём в раздел создания карточек питомцев — <https://petfriends.skillfactory.ru/my_pets> — и создадим через UI новую карточку (DevTools должны быть при этом открыты). Взглянем на содержимое вкладки network панели разработчика браузера:

На этот раз обратим внимание на URL, код состояния (он должен быть 200 без всяких перенаправлений), тело запроса (один из параметров содержит в себе файл) и заголовки запроса.

Добавим новый POST-запрос в нашу коллекцию, назовем его POST /api/pets (по аналогии с тем, что нам показывает консоль разработчика в веб-браузере). Рассмотрим основные заголовки для запроса подробнее. Для этого перейдем на соответствующую вкладку в Postman и сравним уже имеющиеся заголовки с тем, что у нас должно быть.

* + **Accept: \*/\***— это заголовок, определяющий *MIME* тип содержимого, в данном случае его значение говорит о поддержке любого типа контента. *Postman* также предлагает нам это значение.
  + **Accept-Encoding: gzip, deflate**  — сообщает, какую кодировку содержимого (как правило, алгоритм сжатия) может понять клиент. Тут *Postman* предлагает *gzip*, *deflate*, *br*  — это более широкий список кодировок и алгоритмов, чем нам нужен.
  + **Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7** — поддерживаемые языки. Такой заголовок в списке стандартных отсутствует, поэтому копируем его.
  + **Cache-Control: no-cache** — указание на отсутствие кэширования запроса. Также добавляем, как есть.
  + **Connection: keep-alive** — постоянное использование открытого *TCP* соединения, вместо создания нового — копируем.
  + **Content-Length и Content-Type Postman** —генерирует автоматически, исходя из передаваемого контента.
  + **Cookie** — в данном случае это самый важный заголовок. У нас есть 2 способа для корректного заполнения этого заголовка. Воспользуемся наиболее простым и понятным. Если вспомнить предыдущий пример с авторизацией, то можно заметить, что запрос *GET* /all\_pets, на который было перенаправление, также требовал *Cookie*, но мы ничего для этого не предпринимали. Идея в том, что *Postman* понимает заголовок *Set-Cookie* в ответе запроса и устанавливает его значение на последующие запросы. Это значит, что нам не нужно искать способы записи этой уникальной строки в переменную.

В предыдущем примере мы добавляли тест на проверку наличия нужного нам *session* (который хранится в  *Cookie*), но есть способ отследить и даже отредактировать *Cookie* текущей сессии в любой момент времени. Для этого просто нажмите кнопку *Cookies*.

Несмотря на то, что загрузка файла является более сложной операцией, Postman позволяет сделать это достаточно просто. Выберите тип File, нажмите на кнопку в поле VALUE и выберите картинку.

Отправим запрос и убедимся, что он прошел успешно, тесты не провалились, и новый питомец  был добавлен в список питомцев пользователя:

Нажимаем на Console внизу на панели инструментов Postman. Теперь мы можем сравнить сгенерированный итоговый запрос с тем, что мы видели в браузере в прошлом уроке:

## ****Подведём итоги****

В этом уроке мы:

* + рассмотрели, как можно вызывать POST-запросы, используя инструмент Postman;
  + написали собственный простой тест на проверку наличия Cookie;
  + рассмотрели основные заголовки на нашего POST-запроса и сравнили их с теми, что предлагает Postman изначально;
  + научились загружать файлы, отправляя POST-запрос;
  + посмотрели, как можно отслеживать Cookie текущей сессии.

**Для закрепления первичных умений рекомендуем выполнить задание**

Создайте ещё одного питомца на сайте [PetFriends](http://petfriends.skillfactory.ru/" \t "_blank) самостоятельно. Добавьте тест на проверку в Postman, что запрос на создание питомца был выполнен успешно.