[Модуль 8. Postman и отправка REST API запросов](https://lms.skillfactory.ru/courses/course-v1:SkillFactory+INTQAP+2022/course/" \l "block-v1:SkillFactory+INTQAP+2022+type@sequential+block@69f4124ad1354688b01d148b897ff031)

**8.2. Работа со Swagger**

Наш курс посвящён обеспечению качества, в связи с этим мы должны задать себе вопрос: можно ли покрыть API тестами и обеспечить качественную работу, если «вылавливать» вызовы в консоли? Вряд ли. Значит, будем искать другие пути.

Для API давно придуманы специальные средства документирования, которые позволяют отобразить все точки API в одном месте в стандартном формате. Мы обратимся к одному из таких средств — Swagger.

**Swagger** — способ отображения документации web-API.

Подробнее о том, как документируют API, мы ещё поговорим в разделе, посвящённом автоматизации, однако познакомиться со Swagger будет полезно уже сейчас.

Проект «Дом питомца», который  мы пытались тестировать в предыдущем модуле, не имеет документации API. Но нам повезло: Макс и Алиса, работая над прототипом социальной сети PetFriends, подошли к делу основательнее — создали документацию для API и запустили Swagger для её отображения.

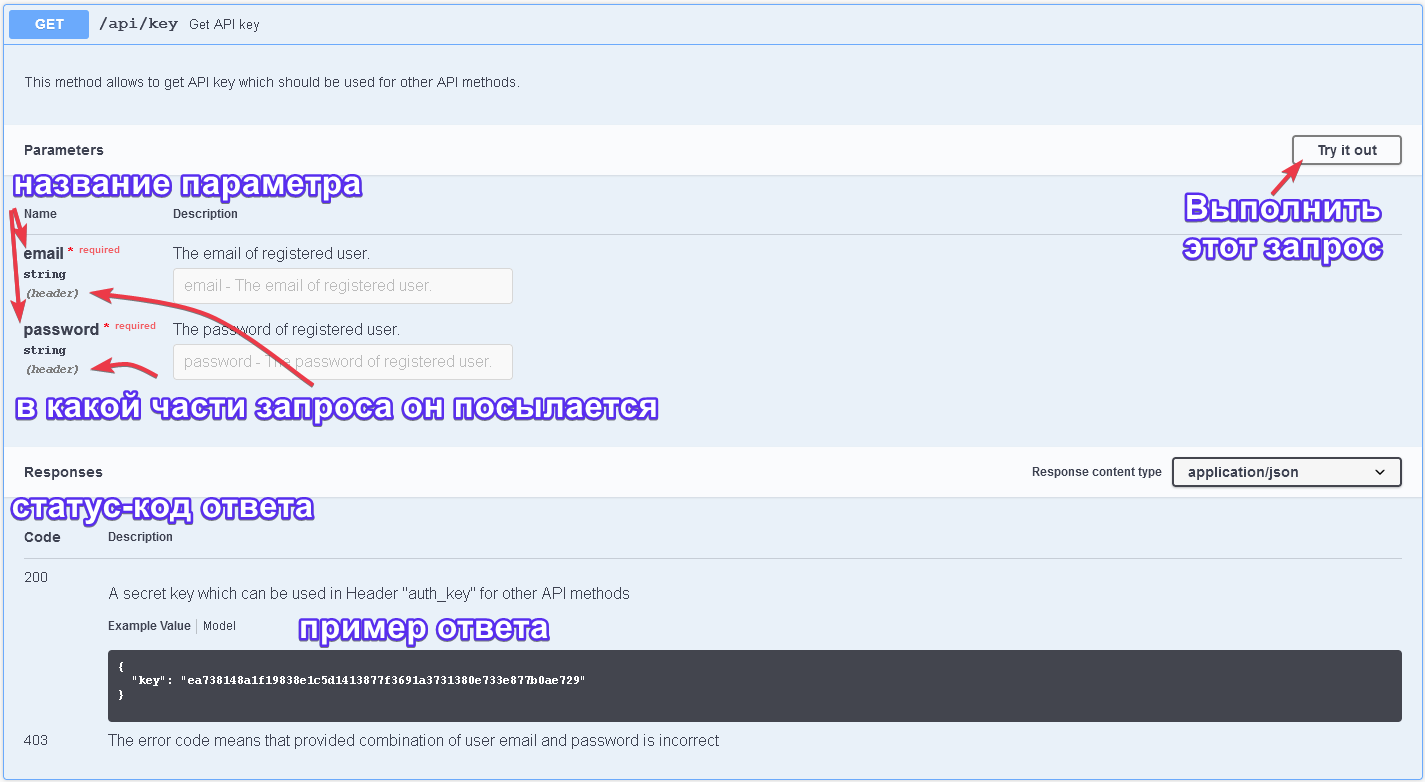
Для нас гораздо важнее информация ниже:

1. **Base URL** (базовый *URL*) — неизменная часть, указывающая на сервер в целом.
2. **Методы**, которые поддерживают конечные точки.
3. **Конечные точки** — та часть, которая добавляется к *URL*.

|  |  |
| --- | --- |
|  | 1. **Base URL** (базовый URL) — неизменная часть, указывающая на сервер в целом. 2. **Методы**, которые поддерживают конечные точки. 3. **Конечные точки** — та часть, которая добавляется к URL. |

### Получаем ключ API через Swagger

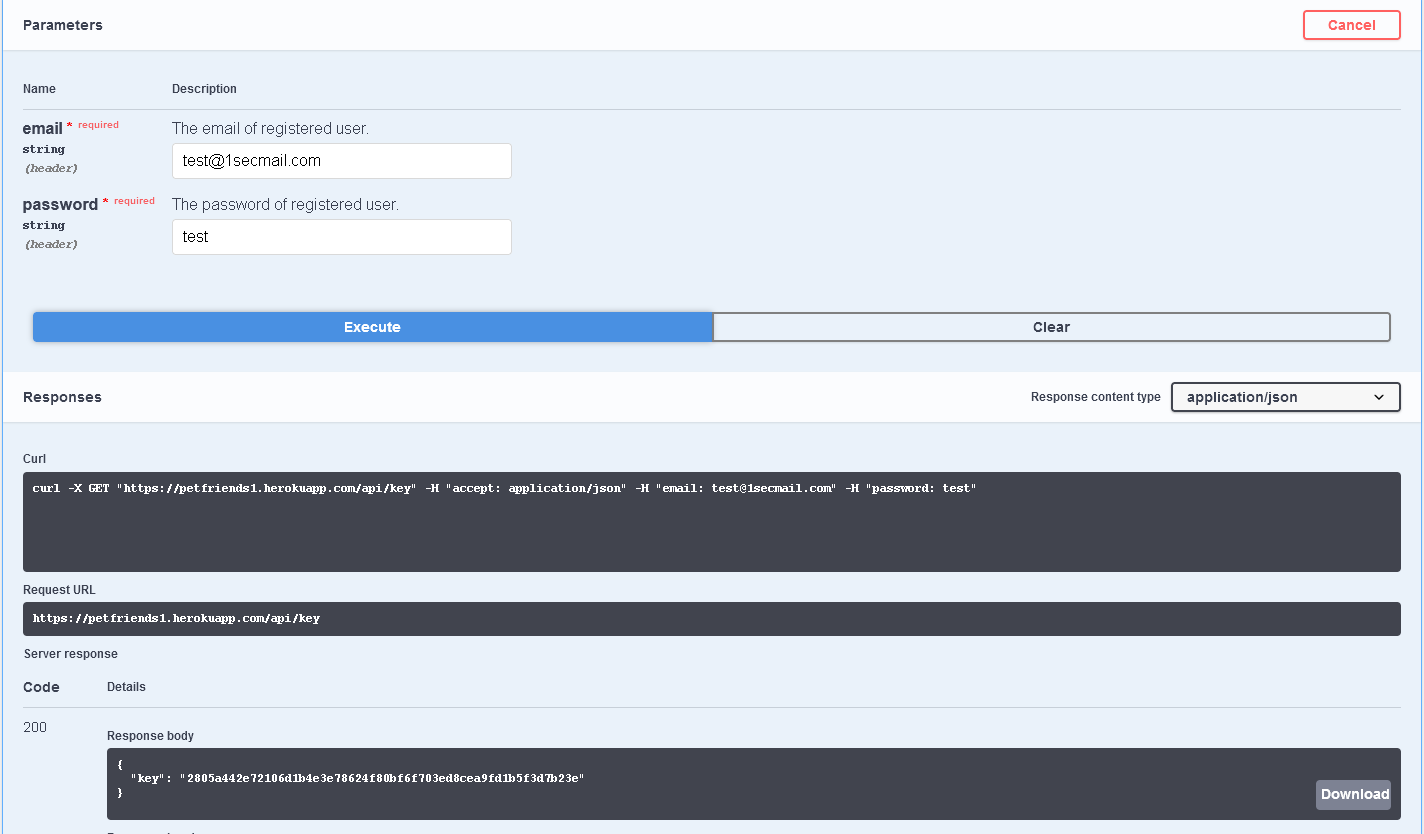
Давайте посмотрим информацию о точке **/api/key**:



Из описания точки мы узнаём, что она обрабатывает запросы **GET** и для получения секретного ключа надо отправить два **обязательных (*required*)** параметра:

* 1. ключ,
  2. пароль в заголовке (*Header*) **GET**-запроса.

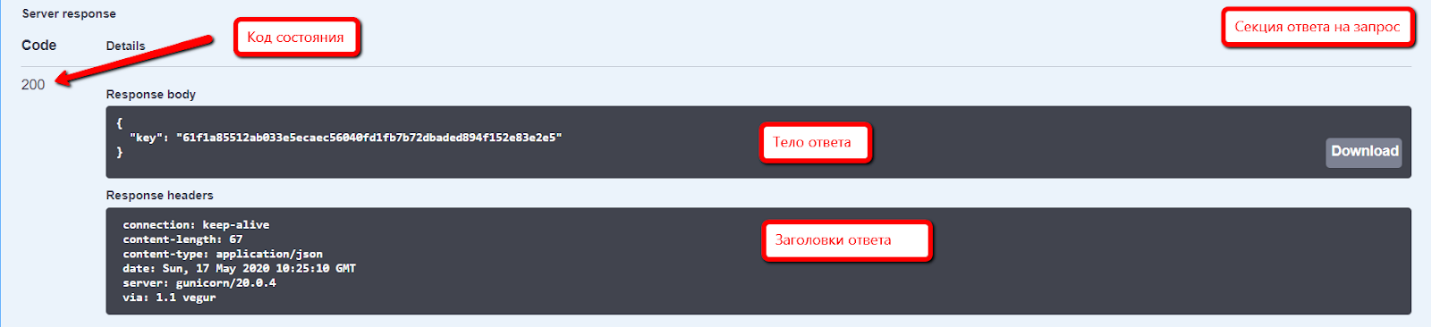
Если нажать кнопку **Try it out** и заполнить поля **email** и **password** (пароль) данными, с которыми вы зарегистрировались, вы получите автоматически сформированный запрос и ответ сервера.



В секции **Server response**, которая появится после получения ответа на сформированный запрос, вы увидите информацию об ответе — код состояния, заголовки и тело ответа.

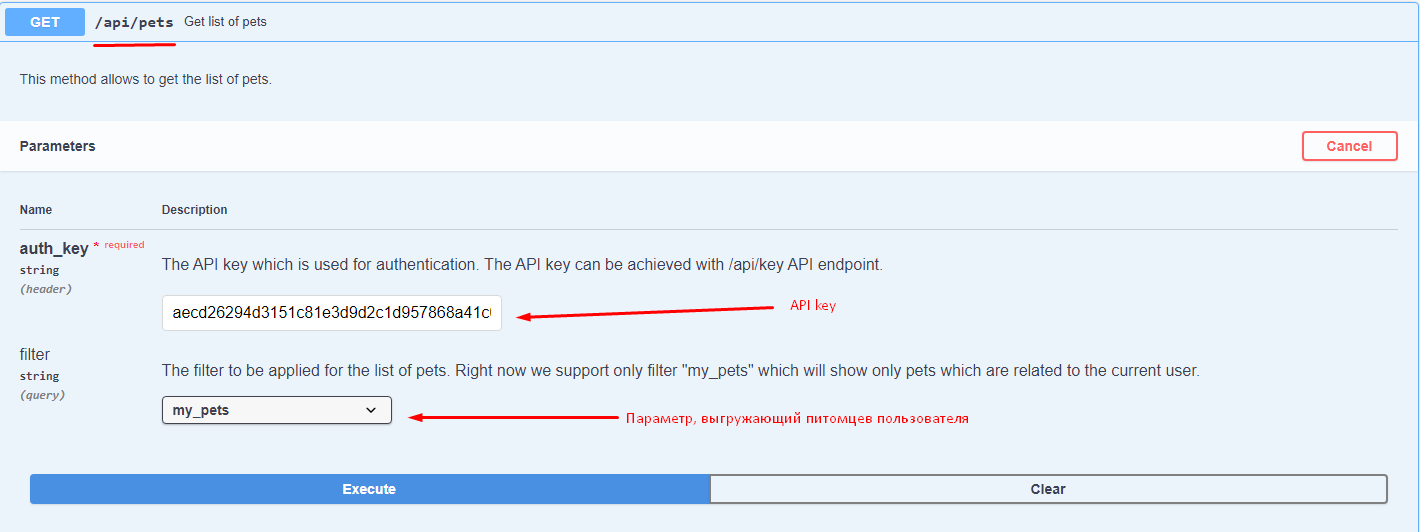
В теле ответа находится *API key*, который нужно будет использовать в *Swagger* для выполнения каждого запроса. Его можно загрузить в файл формата **json** на компьютер и потом копировать оттуда.

Можно просто копировать из ответа *Swagger* и вставлять в каждый выполняемый далее запрос.

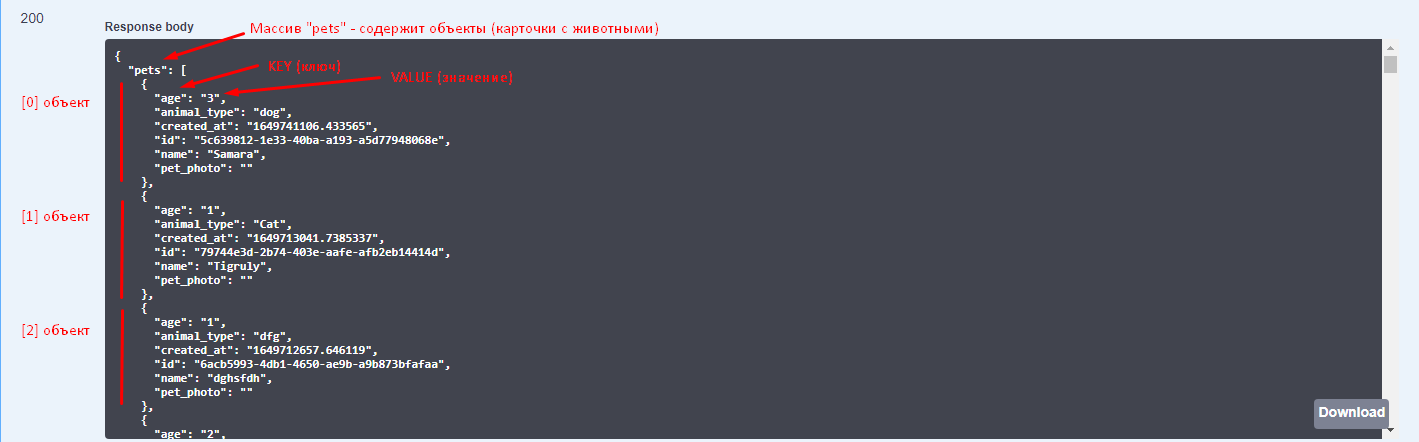


Самостоятельно создайте GET-запрос (точка /api/pets - Get list of pets) в Swagger.

Чтобы активировать возможность отправки запроса, нажмите на кнопку **Try it out**. Сначала необходимо получить API key из GET-запроса (точка /api/key), рассмотренного выше, затем следует вставить ключ в обязательный параметр auth\_key.



После нажатия на кнопку **Execute** вы получите список питомцев, отражённый на странице «Все питомцы» портала PetFriends. Если выбрать дополнительный параметр filter: my\_pets, то в ответе от сервера вы получите список ваших питомцев (предварительно необходимо добавить несколько карточек нажатием на кнопку «Добавить питомца» [на странице](https://www.google.com/url?q=https://petfriends.skillfactory.ru/my_pets&sa=D&source=docs&ust=1652380596200992&usg=AOvVaw1jOPO5l9duVinWbIjcPAGo)).



Внимательно изучите структуру ответа от сервера. Обратите внимание на ключи (параметры) объектов и их значения. Найдите различия в ответах с установленным параметром filter: my\_pets и без него (в первом случае выгружаются питомцы, созданные пользователем, а во втором — все питомцы, отображённые [на странице](https://www.google.com/url?q=https://petfriends.skillfactory.ru/all_pets&sa=D&source=docs&ust=1652380596201124&usg=AOvVaw24oFPiaWI6itTJFiPpUcVR)).

⭐ **Совет**. Для полного понимания того, как устроен JSON-файл, в формате которого приходит ответ от сервера, рекомендуем к прочтению [следующую статью](https://www.google.com/url?q=https://webref.ru/dev/json-tutorial&sa=D&source=docs&ust=1652380596201179&usg=AOvVaw0gc3wSooHRa4Gdi-zsNIm6).

### Получаем ключ API через Postman

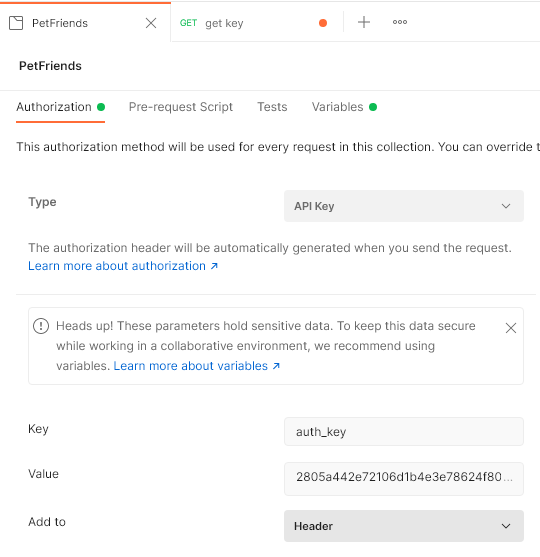
А теперь ту же процедуру — получение ключа API — проделаем в Postman.

|  |  |
| --- | --- |
| **1** | Создадим новую коллекцию PetFriend. |
| **2** | В коллекции создадим переменную для базового URL — это удобно в случае, если URL поменяется.    Переменная — **base\_url**, значение — **https://petfriends.skillfactory.ru**. |
| **3** | Создадим запрос типа **GET**. В URL укажем **{{base\_url}}/api/key**.  Сохраним запрос в коллекции PetFriend. |
| **4** | В секции **Headers** для запроса укажите заголовки **email** и **password** со значениями, которые вы использовали при регистрации на сайте. |
| **5** | Нажмите кнопку **Send**. Если всё сделано верно, вы получите ключ для работы с API. |

Почему запрос на получение ключа не работает?

Если вы внимательно просмотрели документацию Swagger, то заметили, что большинство запросов требуют наличия auth\_key в заголовке.

Потому в параметрах коллекции есть раздел авторизации, куда можно добавить параметр, значение и указать, что отправлять данные надо в заголовках.



После этого для всех запросов, которые мы создаём, авторизация будет выполнятся по умолчанию — к каждому запросу Postman будет подставлять в Header параметр auth\_key со значением ключа.

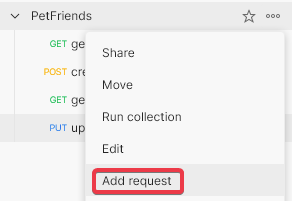
## 8.3. PUT-запросы

**Важно!** Согласно [стандарту](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT), запрос ***PUT***или создаёт новый объект или полностью заменяет указанный целевой объект данными, представленными в теле запроса.

Поэтому при использовании ***PUT*** мы не можем обновить только часть полей (например, только имя питомца): если мы не укажем данные для поля — оно примет значение **None**.

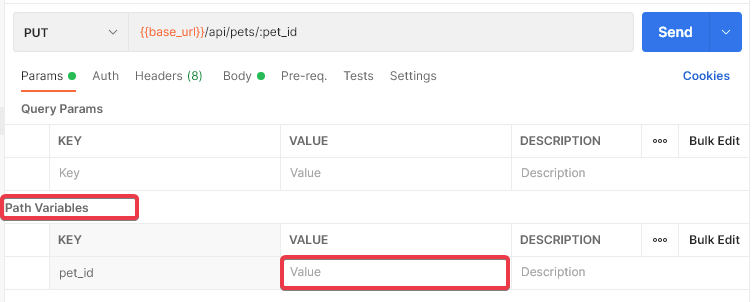
**СОЗДАНИЕ ЗАПРОСА PUT В POSTMAN**

* + 1. В контекстном меню коллекции выберите **Add request**.

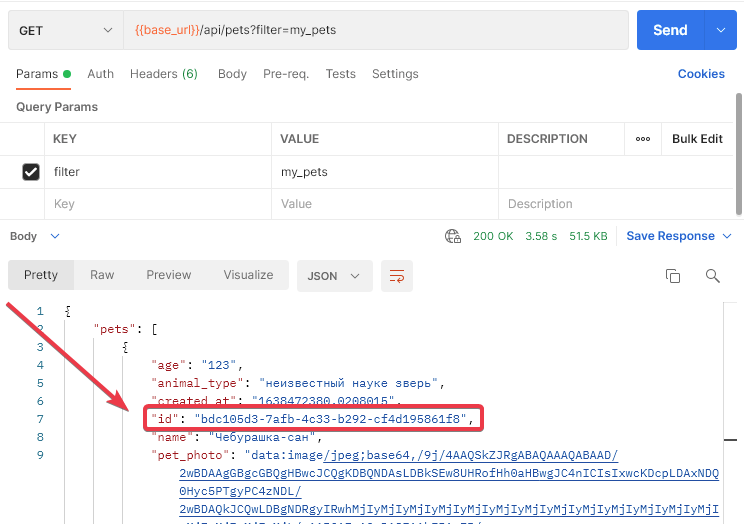


* + 1. Задайте имя новому запросу, например *update pet*.
    2. Выберите тип запроса **PUT** и в строке укажите: **{{base\_url}}/api/pets/:pet\_id**.

Двоеточие в пути указывает *Postman*, что туда надо подставить параметр, и в интерфейсе появится соответствующая опция.



Теперь надо указать *ID* созданного ранее питомца в поле **Value**. Получить *ID* питомцев можно в *Swagger*, если обратиться к */api/pets* или создать запрос в *Postman*.

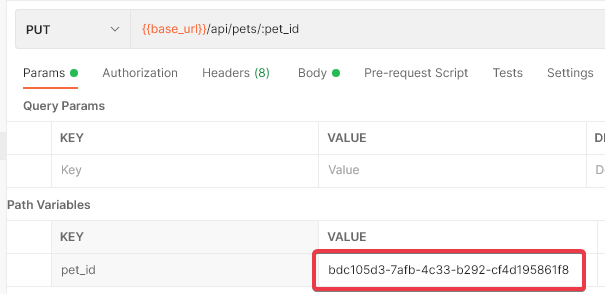


Кроме того, мы можем посмотреть состояние питомца на веб-странице.



У нас есть всё необходимое, чтобы обновить данные через **PUT**.

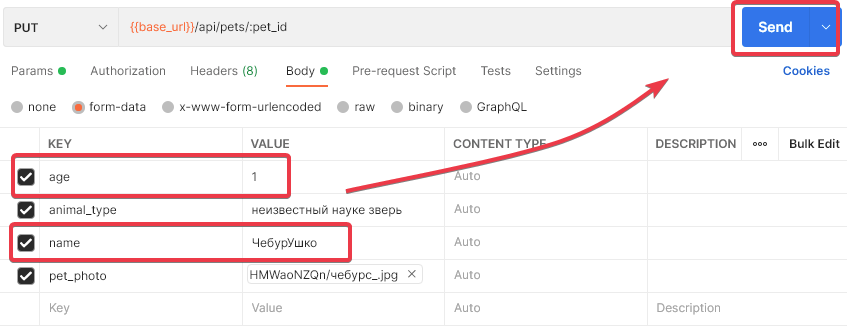
Добавляем *ID*.



**Примечание.** Надо скопировать *ID* *вашего* питомца, а не тот, что указан на скриншоте.

Аккуратно скопируйте значение — не добавляйте и не удаляйте ничего лишнего.

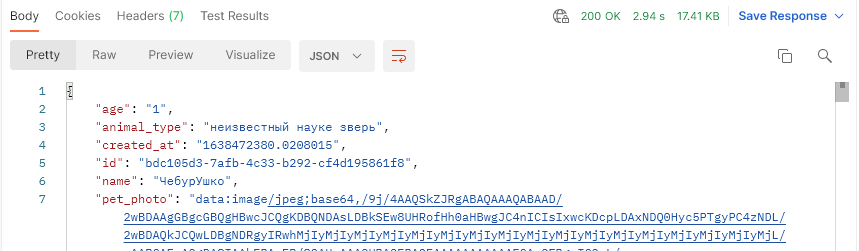
Теперь переключитесь на раздел **Body** и задайте параметры **PUT**-запроса.



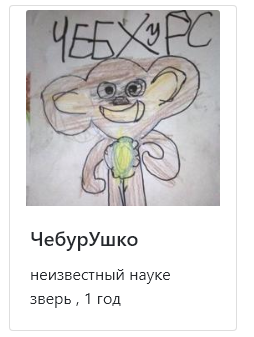
В данном случае мы меняем возраст и имя.

Нажмите кнопку **Send**.

Нам приходит ответ:

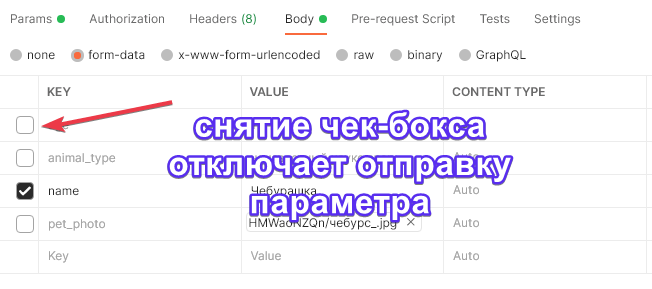


И на сайте мы можем проверить изменение данных визуально:



Как истинные тестировщики вы вряд ли решите, что этого достаточно.

Попробуем изменить только имя.



Нажимаем кнопку **Send**, получаем ответ сервера.



Результат на сайте.



Отдельно отметим, что стоит провести отправку двух абсолютно одинаковых **PUT**-запросов.

Согласно [стандарту](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT) запрос **PUT** должен быть **идемпотентным** — это значит, что два одинаковых запроса должны давать одинаковый результат, поэтому такой тест стоит выполнять для всех **PUT**-запросов.

**ТЕСТ РЕЗУЛЬТАТОВ ЗАПРОСА СРЕДСТВАМИ POSTMAN**

Чтобы написать тестовый сценарий, откройте запрос в приложении *Postman* и откройте вкладку **Tests** (Тесты). Введите следующий код *JavaScript*:

pm.test("Status code is 200", **function** () {

pm.response.to.have.status(**200**);

});

Этот код использует библиотеку *pm* для запуска метода **TEST**. Текст "Status code is 200" будет использован как название теста. Функция внутри теста представляет собой **верификационное утверждение** (или проверочное выражение; англ. *assertion*). В тестах *Postman* вы можете использовать синтаксис [BDD](https://www.chaijs.com/api/bdd/) библиотеки *Chai*, которая предоставляет широкие возможности для оптимизации читабельности тестов. В этом случае код использует конструкцию *BDD* **to.have** для проверочного выражения.

Этот тест проверяет код ответа, возвращаемый *API*. Если статус-код ответа — 200, тест пройден, в противном случае он завершится неудачей.

Нажмите **Send** и проверьте вывод результатов теста в области ответов.

Строка кода pm.response.to.have.status(200); дословно читается так: «Отклик (после выполнения запроса) должен иметь статус (200)». Таким образом, данный код подчиняется булевой логике: истина-ложь. Если данное утверждение истинно и запрос действительно будет иметь статус 200, то тест будет пройден. Если утверждение ложно и в результате выполнения запроса придёт код 403, например, то тест будет провален.

Попробуйте изменить код состояния в тесте и отправить запрос ещё раз, чтобы увидеть, как выглядят результаты успешного и провалившегося тестов.

Структура наших тестовых верификаций может быть различна — в зависимости от того, какую логику вы выстраиваете и как хотите выводить результаты.

Код в следующем примере — альтернатива того же теста, что был приведён выше, но с использованием атрибута **expect**:

pm.test("Status code is 200", () => {

pm.expect(pm.response.code).to.eql(**200**);

});

Дословно данный код означает: «Ожидается, что код ответа равен 200». Как видите, появилось новое ключевое выражение to.eql, которое является в данном случае оператором равно.

## 8.4. DELETE-запросы

Последний не рассмотренный нами до сих пор REST-метод — это метод **DELETE**. Напомним, что этот метод, как понятно из названия, используется только для удаления каких-либо сущностей. DELETE-запрос может иметь тело запроса и тело ответа, и этот запрос не кэшируется.

Рассмотрим пример такого запроса для [Pet Friends](http://petfriends.skillfactory.ru/). Опишем шаги в виде **тест-кейса**:

**Предусловия:**

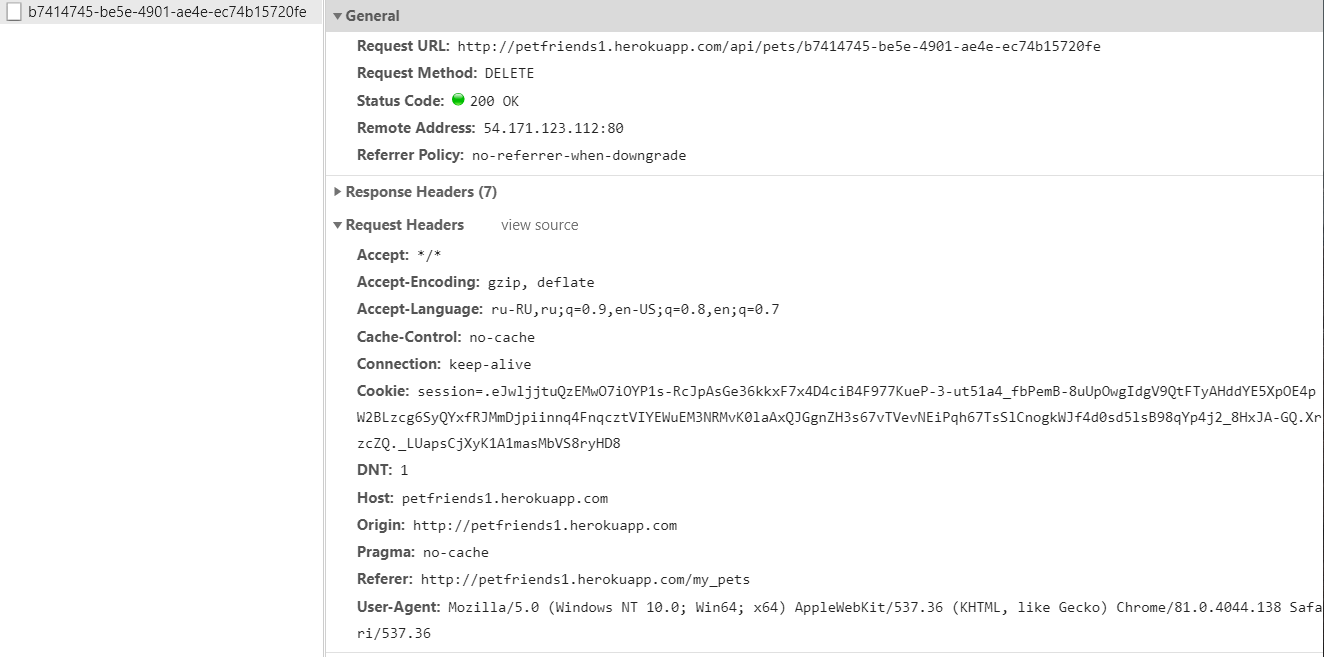
* 1. Зарегистрироваться на сайте.
  2. Добавить одного питомца.

**Шаги воспроизведения:**

* 1. Открыть раздел «Мои питомцы» (http://petfriends.skillfactory.ru/my\_pets).
  2. Открыть вкладку Network (Сеть) в консоли браузера.
  3. Удалить одного питомца, нажав на крестик.

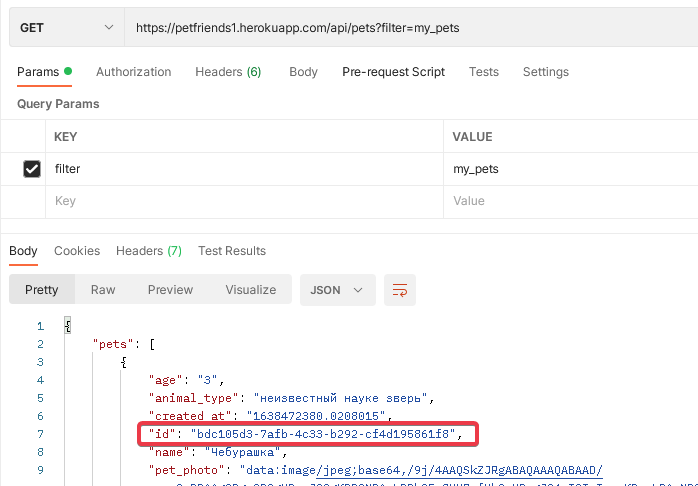
**Ожидаемый результат:**

* 1. Питомец исчез из списка.
  2. Во вкладке Network инструментов разработчика отображается запрос DELETE:

Рис. 1. Приложение [Pet Friends](http://petfriends.skillfactory.ru/) во вкладке Network инструментов разработчика

Мы можем видеть, что в URL запроса передаётся id питомца, и запрос DELETE имеет тот же набор специфических заголовков, что и рассмотренные ранее запросы POST и PUT. Возникает закономерный вопрос: **как нам в получить id питомца для удаления?**

Заглянув в документацию (Swagger), мы увидим, что все данные о наших питомцах, включая их id, можно получить из запроса GET: https://petfriends.skillfactory.ru/api/pets?filter=my\_pets.

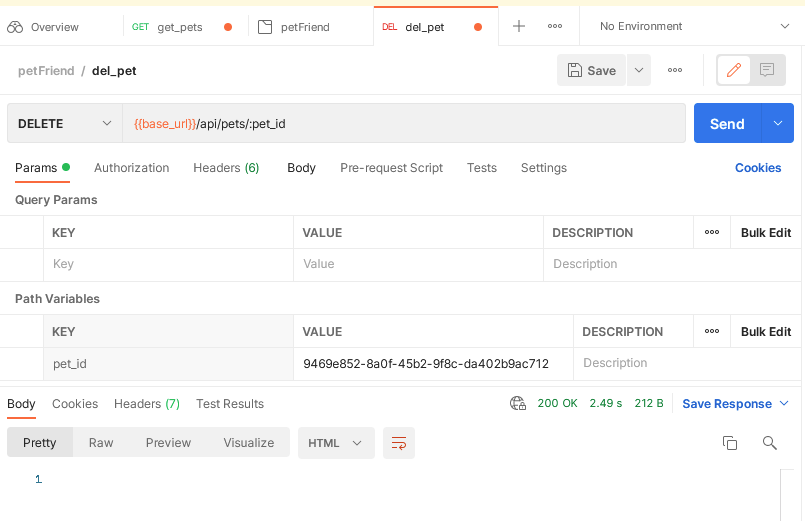
Рис. 2. Получение id в запросе к точке /api/pets

Очевидно, что можно создать таблицу заранее заготовленных id, как мы делали в предыдущем уроке. Однако каждая запись в этой таблице будет использована в тестах однократно, потому что удалить объект можно только один раз.

В реальном тестировании у вас, скорее всего, будет либо заранее созданный набор тестовых данных в виде объектов, и вы каждый раз будете их пересоздавать, либо сценарий, в котором вы сначала создаёте объект, а потом удаляете его. Второй случай будет рассмотрен нами позднее, когда мы будем говорить о сценариях. Сейчас же наша задача заключается в том, чтобы рассмотреть использование DELETE-метода без подобных усложнений.

Теперь перейдём к **созданию запроса в Postman:**

* 1. Тип запроса — DELETE.
  2. Строка запроса — {{base\_url}}/api/pets/:pet\_id.
  3. Значение Path Variables pet\_id  — id вашего существующего питомца.

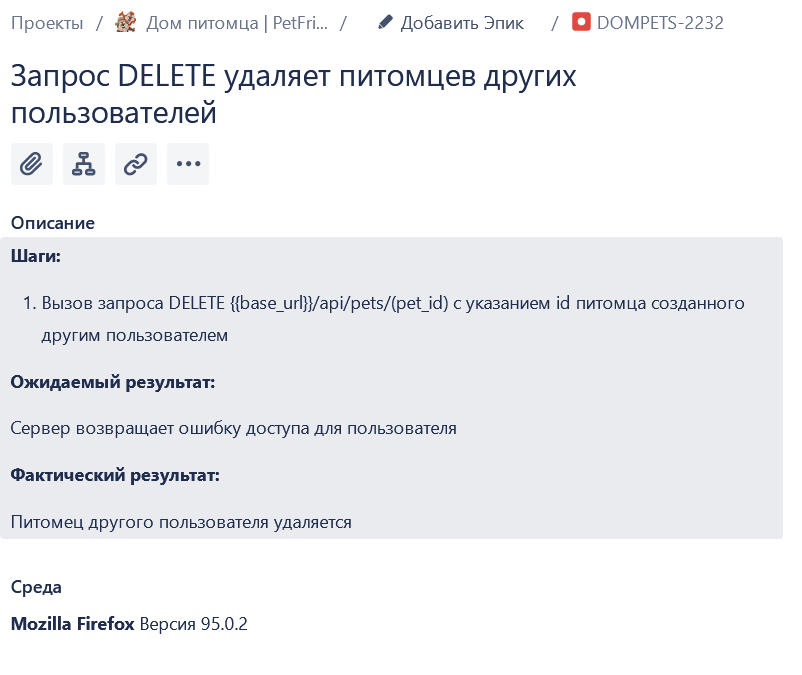
Рис. 3. Запрос DELETE и его выполнение в Postman

Выполнение запроса должно приводить к удалению питомца.

## ****Описание ошибок****

В случае обнаружения ошибки вам следует создать отчёт в баг-трекинговой системе. Так как вызовы API являются низкоуровневыми, то, как правило, ошибки будут иметь приоритет от серьёзного до блокирующего.

В шагах  необходимо указать тип запроса, ожидаемый и фактический результат.

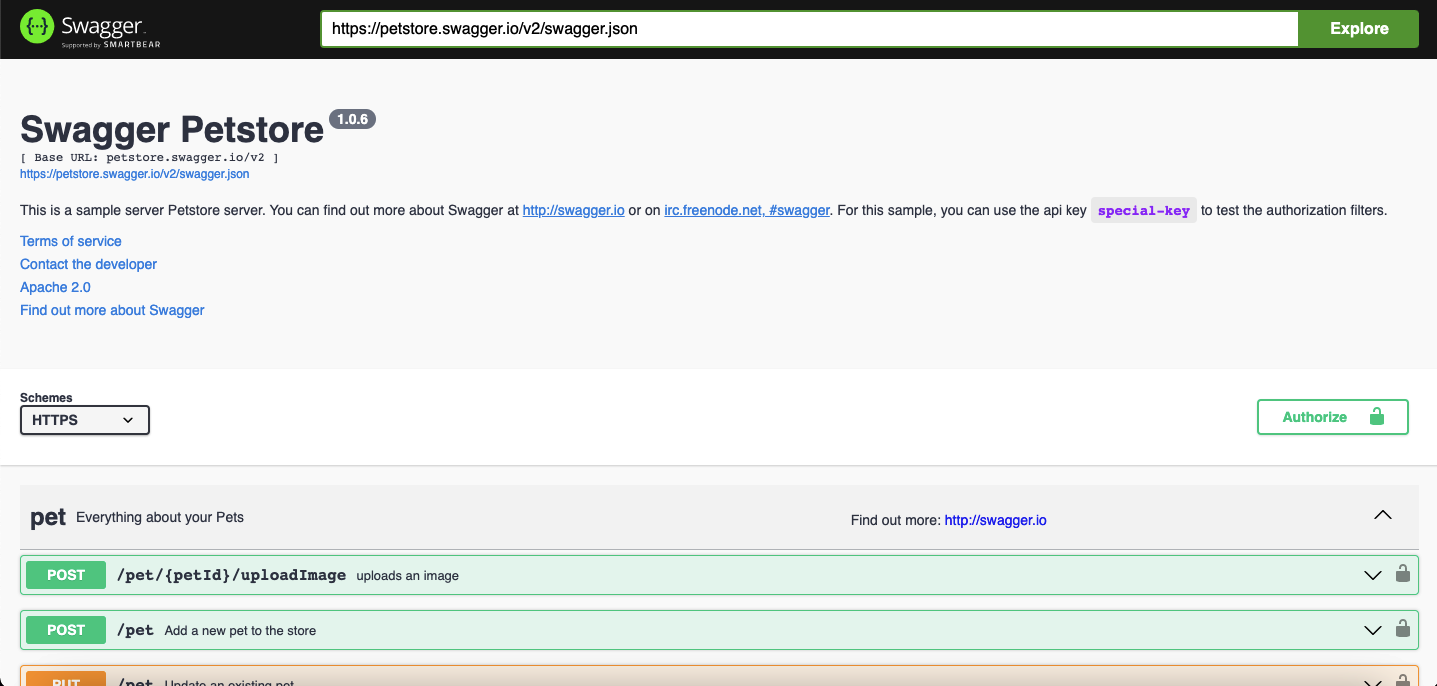
Рис. 4. Ошибка API, оформленная в баг-трекинговой системе

## 8.5.1. Введение в сценарное тестирование API

В этом юните мы обобщим ранее полученную информацию, научимся запускать запросы последовательно в составе коллекции, а также познакомимся с основами написания скриптов, которыми будут сопровождаться наши запросы.

Отвлечёмся на время от уже привычного нам ресурса PetFriends и представим, что нам пришла задача протестировать создание нового пользователя на ресурсе <https://petstore.swagger.io/>

При переходе по ссылке мы видим лишь документацию в Swagger по тестированию API. Таким образом, мы понимаем, что работать нам придётся «вслепую», без UI-интерфейса.



Ниже представлен один из возможных сценариев тестирования блока с созданием нового пользователя. Данный сценарий поможет нам разобраться с основными методами запросов к указанному выше блоку ресурса, а также даст понимание основ составления скриптов для сопровождения наших запросов.

**НАСТРОЙКА НОВОЙ КОЛЛЕКЦИИ**

Довольно часто возникает ситуация, при которой нам необходимо выполнить определённую последовательность запросов последовательно, один за другим. У этих запросов могут быть общие сущности (например, имена, *URL*, номера телефонов и тому подобное), которые удобнее всего записывать в переменные, доступные любому запросу внутри коллекции.

Наш текущий сценарий предполагает наличие подобных переменных в запросах к серверу. Поэтому прежде всего необходимо определиться с количеством и наименованием переменных, которые мы будем использовать в рамках сценария.

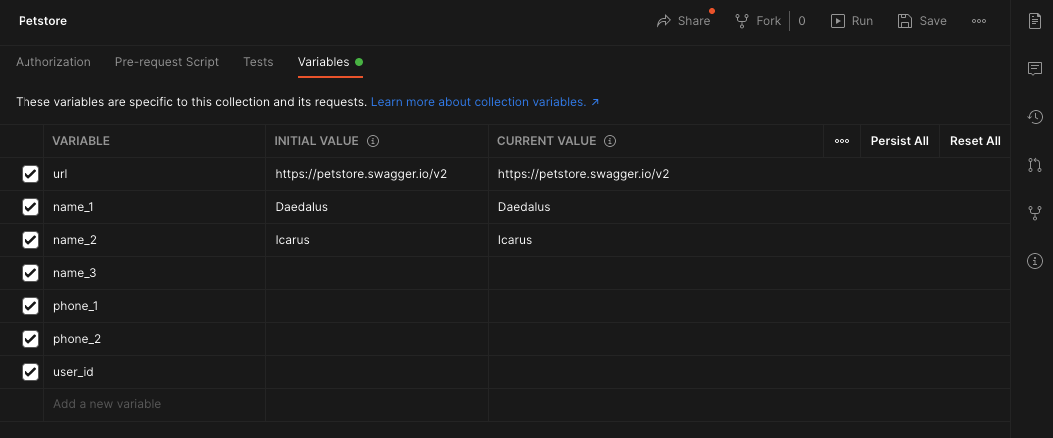
Перейдём на вкладку *Variables*, предварительно нажав на название коллекции, и введём в колонку VARIABLE следующий список переменных:

* + url,
  + name\_1,
  + name\_2,
  + name\_3,
  + phone\_1,
  + phone\_2,
  + user\_id.

Для начала такого списка нам хватит, позже, при создании дополнительных запросов, всегда существует возможность вернуться, добавить нужные переменные либо изменить уже имеющиеся.

* + Для переменной url следует указать INITIAL VALUE и CURRENT VALUE — *https://petstore.swagger.io/v2*.
  + Для переменных name\_1 и name\_2 можно подставить любые значения.
  + Остальные значения переменных необходимо оставить пустыми.

Должно получиться так, как на скриншоте ниже.





**ЗАДАНИЕ 8.5.1.1**

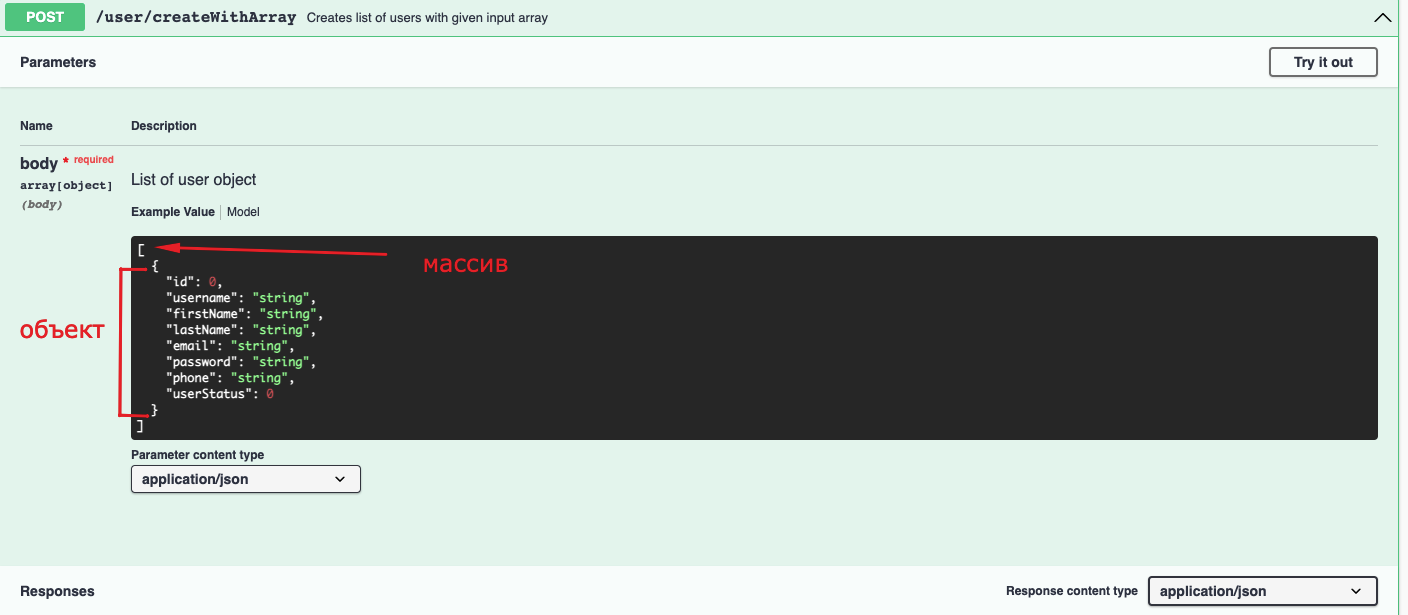
Создание группы пользователей.

Для создания массива с пользователями, согласно документации *Swagger* к ресурсу *Petstore*, используется эндпоинт /user/createWithArray с методом запроса POST.

⭐ Перед каждым эндпоинтом не забываем указывать базовый *URL* (обычно разработчики документации указывают базовый *URL* ресурса в самом верху страницы под названием документации).

В нашем случае базовый *URL* — *petstore.swagger.io/v2*. Ранее мы его уже записывали в переменную, поэтому адрес нашего запроса будет выглядеть следующим образом: {{url}}/user/createWithArray

Согласно документации, тело нашего запроса обязательно должно содержать строку в формате *JSON* в виде массива с вложенными объектами.



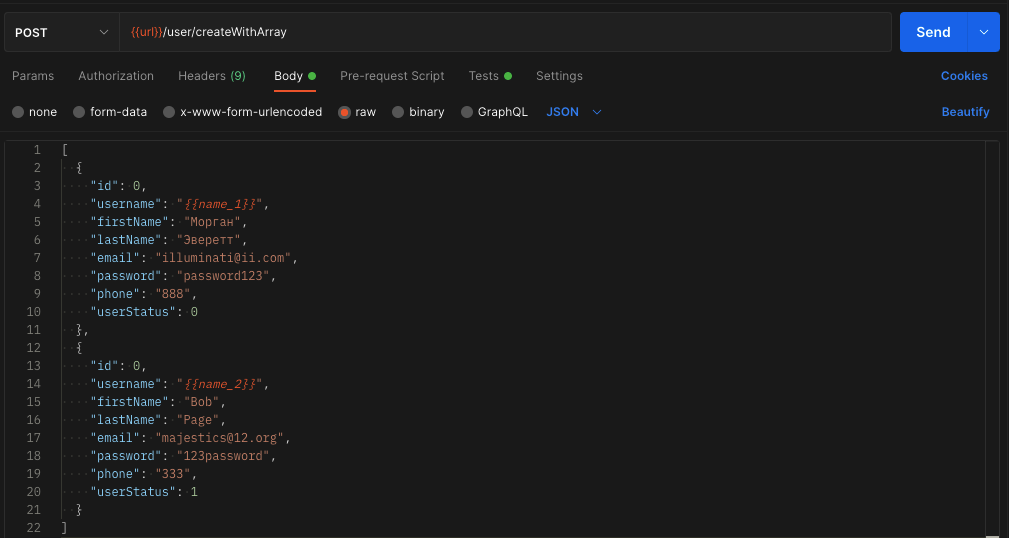
Если не заполнять значение ключа id, то его значение подставится автоматически. Оставим его нулевым, как предложено по умолчанию.

Для ключа username в кавычках и двойных фигурных скобках укажем имя нашей переменной с первым именем: "username": "{{name\_1}}" . Остальные поля можно заполнить произвольными значениями.

⭐ Важно помнить, что параметр, указанный в кавычках, является строковым (**string**), а тот, который указан цифровым (например, **id** или **userStatus**), принимает числа.

В рамках нашего сценария создадим внутри массива два объекта с новыми пользователями. Объекты разделяются между собой внутри массива запятыми. Например, [ {объект}, {объект}] .

Таким образом, у нас должен получиться подобный запрос:

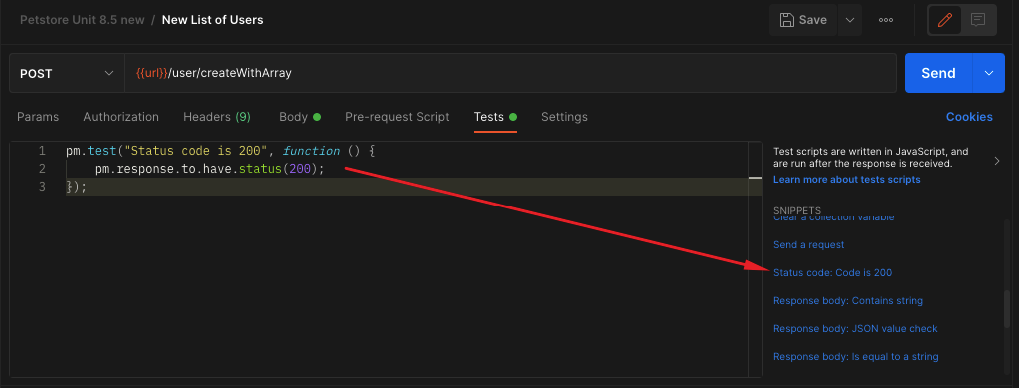


**НАПИСАНИЕ СКРИПТОВ**

Теперь разберём самую важную часть данного юнита — написание скриптов для сопровождения запроса тестами.

Скрипты для тестов в *Postman* пишутся на языке *JavaScript*. Разработчиками *Postman* заранее были написаны особые инструкции для самых распространённых тестов к запросам. В дальнейшем мы рассмотрим их.

Перейдём во вкладку *Tests* нашего первого запроса и выберем самый популярный тест из предложенных сниппетов (*SNIPPETS*) справа от поля для ввода кода — *Status code: Code is 200*. Как только мы нажмём на указанный сниппет, в поле для ввода кода автоматически появится заранее составленный разработчиками код:



Данный тест будет пройден, если после выполнения нашего запроса будет получен **статус-код: 200**.

После того как мы полностью составили запрос и сопроводили его необходимыми скриптами, следует нажать на кнопку *Save* (слева от названия запроса).

**Важно!** Скрипты из вкладки *Tests* выполняются после выполнения запроса. Если есть необходимость написать и выполнить скрипт перед отправкой запроса, то следует писать код скрипта во вкладке *Pre-request Script*

**ЗАДАНИЕ 8.5.1.2**

Запрос карточки первого созданного пользователя, запись значения в переменную, метод **to.include**.

Согласно документации, нужный нам эндпоинт для получения данных о созданном пользователе выглядит следующим образом: /user/{username}

Если мы внимательно посмотрим на документацию, то увидим, что с данным эндпоинтом есть целых три запроса:



Однако, несмотря на кажущееся сходство, это абсолютно разные по своим функциям запросы. Эти функции определяет метод запроса, в данном случае:

* + GET,
  + PUT,
  + DELETE.

Простое объяснение указанных методов будет следующее:

* + GET — получает данные,
  + PUT — изменяет данные,
  + DELETE — удаляет данные.

В нашем случае нужен вышеуказанный эндпоинт с методом GET. Вместо {username} подставим переменную с первым именем — name\_1 . Перейдём во вкладку *Test* и напишем два теста для нашего запроса. Первый скрипт сохранит телефон из ответа после выполнения запроса в переменную phone\_1.

Сперва напишем тестовую обёртку-функцию для будущих скриптов:

pm.test("Сохраняем номер телефона", **function** () {

});

Внутрь обёртки уже запишем необходимый нам код:

pm.test("Сохраненяем номер телефона", **function** () {

**var** jsonData = JSON.parse(responseBody)

pm.collectionVariables.set("phone\_1", jsonData.phone)

});

Данный код следует разбирать частями. Первая строка:

**var** jsonData = JSON.parse(responseBody)

Означает дословно следующее: «Объявляется переменная jsonData, которой присваивается значение в формате JSON(тело ответа)». Звучит заковыристо. Однако на деле это обозначает простую вещь: мы создаём новый объект, который будет содержать в себе тело ответа в формате *JSON*.

Следующая строка:

pm.collectionVariables.set("phone\_1", jsonData.phone)

Данная строка означает: «В переменную коллекции записать («имя переменной — phone\_1, что записать — ключ phone из объекта jsonData)».

Второй скрипт проверит, содержится ли значение переменной name\_1 в теле ответа. Напишем функцию-обёртку для нашего кода и поместим внутри неё скрипт. Всё вместе будет выглядеть так:

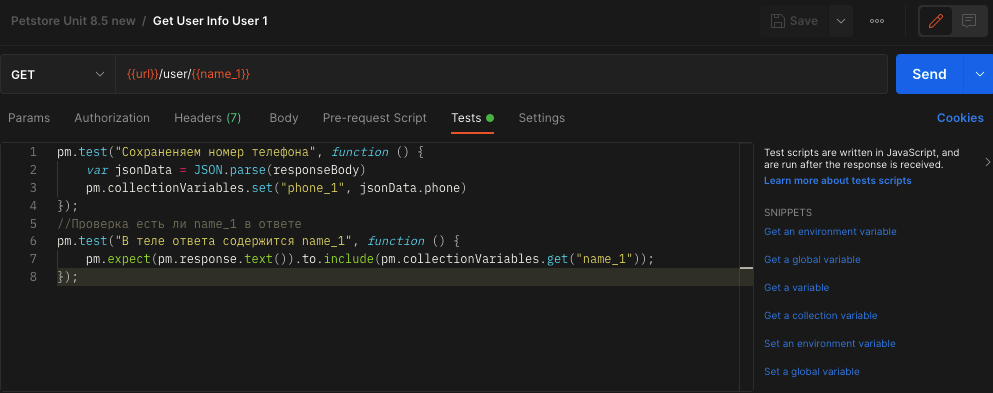
pm.test("В теле ответа содержится name\_1", **function** () {

pm.expect(pm.response.text()).to.include(pm.collectionVariables.get("name\_1"));

});

Разберём значение скрипта, который мы написали: «ожидается(что текст ответа.содержит(значение переменной коллекции("name\_1");»

Готовый запрос должен выглядеть следующим образом:





**ЗАДАНИЕ 8.5.1.3**

Запрос карточки второго созданного пользователя, запись значения в переменную, метод **to.not.include**.

Эндпоинт третьего запроса тот же, что и у предыдущего запроса. Нам нужно лишь верно указать имя. Укажем имя второго созданного нами пользователя — name\_2.

Во вкладку *Tests* поместим также два теста. Первый, как и в предыдущем запросе, запишет в переменную номер телефона из ответа. Однако поскольку мы хотим не перезаписать значение, которое мы установили в переменную phone\_1, а записать ещё одно, то имя переменной укажем другое — phone\_2. Таким образом мы всегда будем иметь записанное значение телефонов и первого, и второго пользователей.

Готовый скрипт выглядит так:

pm.test("Сохраненяем номер телефона", **function** () {

**var** jsonData = JSON.parse(responseBody)

pm.collectionVariables.set("phone\_2", jsonData.phone)

});

Тут для нас уже нет ничего необычного. А вот следующий скрипт будет немного отличаться от похожего в предыдущем запросе. Наш скрипт должен будет осуществить проверку, что имя первого пользователя **НЕ записалось** в объект со вторым пользователем. Для этого мы познакомимся с оператором to.not.

pm.test("В теле ответа не содержится name\_1", **function** () {

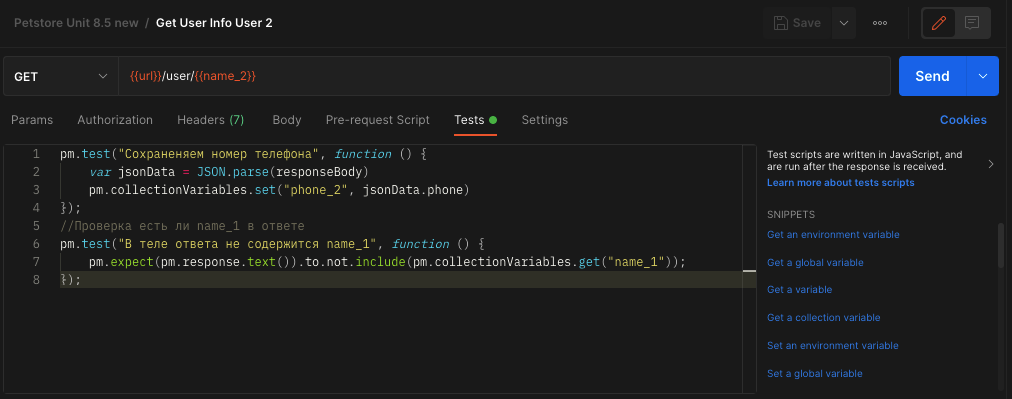
pm.expect(pm.response.text()).to.not.include(pm.collectionVariables.get("name\_1"));

});

Разберём подробнее строку нашего скрипта: «ожидается(что текст ответа.НЕ.содержит(значение переменной коллекции("name\_1");».

Таким образом, мы заменили to на to.not, и теперь мы уже хотим «не найти» указанное в скобках выражение в тексте ответа после выполнения запроса.

В корректно работающей системе провал данного теста исчезающе мал, однако если вдруг мы увидим статус данного теста *FAILED* после выполнения запроса, то это будет повод для заведения бага.

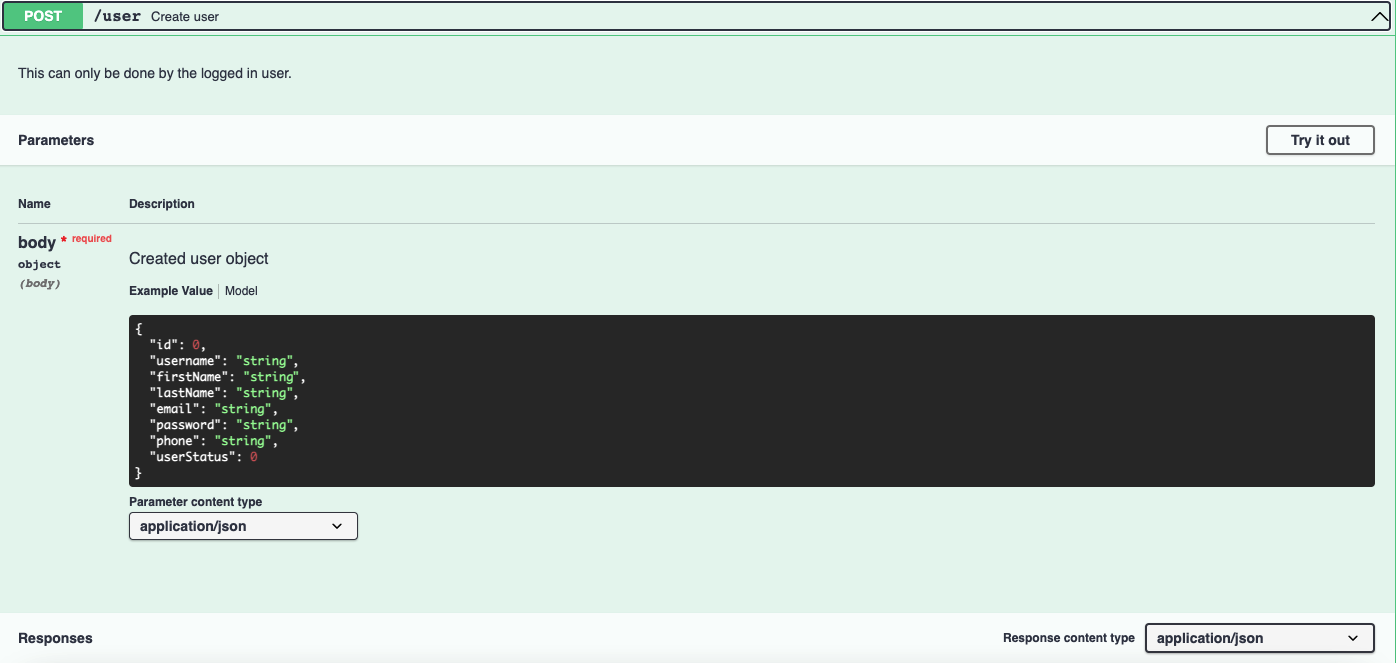


**ЗАДАНИЕ 8.5.1.4**

Создание третьего пользователя, *pre-request Script*, запись *id* в переменную.

Создадим нового пользователя. Пришло время протестировать другой эндпоинт для создания пользователя. Конечно, мы можем создавать единичных пользователей, используя уже знакомый нам эндпоинт /user/createWithArray, однако так мы не протестируем другие возможности системы, а наши тесты всё-таки должны стараться охватить как можно больший функционал.

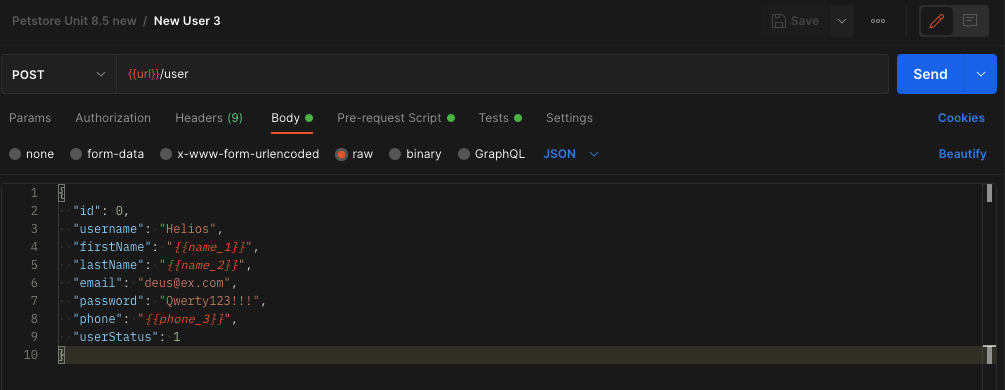
Поэтому вновь обратимся к документации и поищем способ для создания единичного пользователя:



Наш эндпоинт /user с методом POST. Тело запроса также состоит из строки в формате *JSON*. Отмечаем также, что значение, которое мы поместим в тело запроса, не содержит массива (квадратные скобки), как при создании группы пользователей, а только объект в фигурных скобках.

Значение ключа id также, как и в первом POST-запросе, ставим нулевым. В данном случае после успешного выполнения запроса система вернёт нам значение id в ответе от сервера. Но к этому мы вернёмся позже.

Во вкладке *Body* дадим произвольные значения для ключей username, email, password и userStatus объекта нашего нового пользователя. Ключам firstName и lastName дадим значения в виде переменных name\_1 и name\_2 соответственно. Как это делается, мы уже разбирали при первом запросе. Ключу phone дадим значение в виде переменной phone\_3.



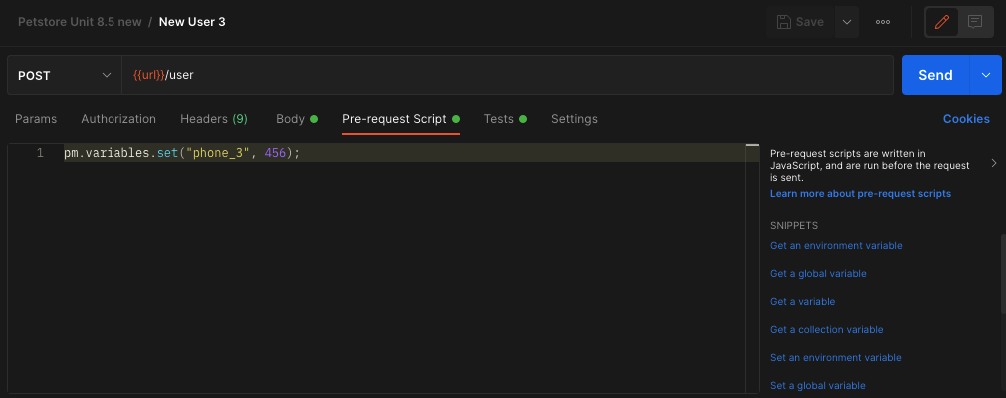
И вот тут вы меня спросите: «А откуда взялась переменная phone\_3? Ведь мы не создавали её в рамках переменных коллекции!»

Если мы вернёмся к переменным коллекции, то вы окажетесь абсолютно правы. Действительно, ранее мы не создавали такую переменную и взяться ей не откуда. Но не будем торопиться дописывать её имя в список.

Перейдём во вкладку *Pre-request Script*. Функция данного раздела такая же, как и у раздела *Tests*: принимать и обрабатывать скрипты. Однако если скрипты из вкладки *Tests* выполняются **ПОСЛЕ отправки запроса** и получения ответа от сервера, то код из вкладки *Pre-request Script* выполняется **ДО отправки запроса**.

Воспользуемся данной замечательной возможностью и напишем в теле вкладки *Pre-request Script* следующий код:

pm.variables.set("phone\_3", **456**);



В отличие от предыдущих скриптов в данном случае мы использовали pm.variables.set конструкцию, а в скобках указали имя переменной, которую ранее нигде не объявляли. Такая запись является корректной.

Разница с pm.collectionVariables.set заключается в том, что используя последнее выражение, мы обращаемся к ранее объявленной переменной в рамках коллекции, и после выполнения запросов её значение будет сохранено. Сохранённое значение можно будет использовать при следующих прогонах либо внутри новых запросов в рамках коллекции.

Использование же pm.variables.set предполагает, что значение переменной будет храниться только во время выполнения запросов. Нам это подходит.

Таким образом, перед тем как отправить текущий запрос на сервер, система выполнит указанный выше скрипт из вкладки *Pre-request Script*, тем самым присвоит временной переменной phone\_3 значение 456. Конечно, вы можете использовать любые другие названия переменных и цифровых значений.

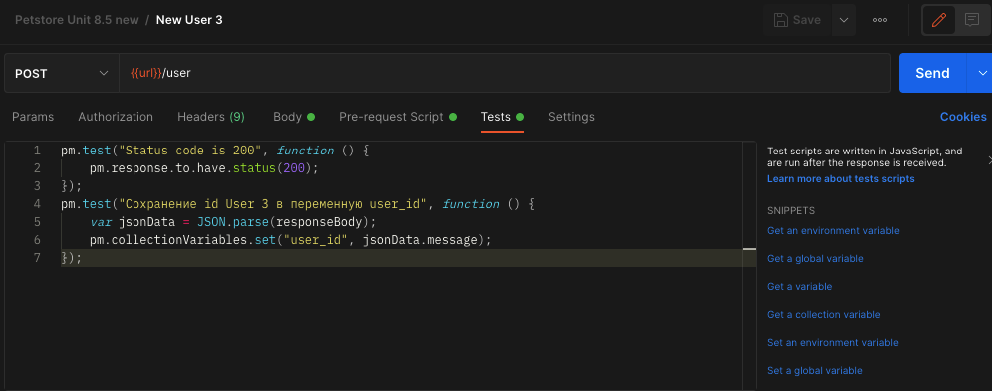
Далее необходимо перейти во вкладку *Tests* и добавить два скрипта. Уже знакомый нам тест на проверку статус-кода:

pm.test("Status code is 200", **function** () {

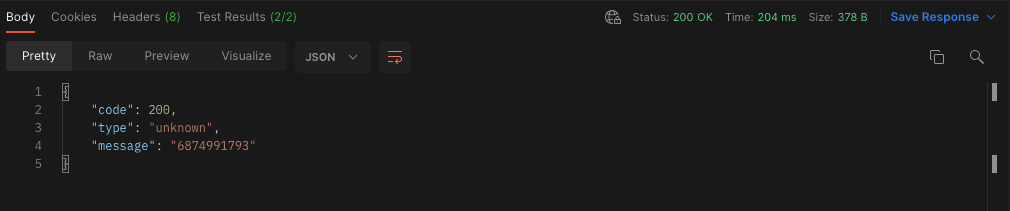
pm.response.to.have.status(**200**);

});

А второй скрипт сохранит в переменную коллекции user\_id значение ключа message из ответа от сервера.



Почему message хранит в себе id? Так продуман сервис разработчиками *backend*. Выполнение подобного запроса даёт ответ от сервера имеющий следующую структуру:



Ключ message содержит в себе строку с цифрами 6874991793 (значение может быть другим). При отправке следующего запроса на получение данных созданного пользователя мы увидим, что эти же цифры будут содержаться в значении ключа id. Однако к этому мы вернёмся позже, когда будем получать данные о созданном пользователе при выполнении следующего запроса.

⭐ Сейчас же нам важно понять, что ключ **message** в теле ответа после выполнения запроса содержит в себе **id** нового созданного пользователя.

**ЗАДАНИЕ 8.5.1.5**

Запрос карточки третьего созданного пользователя, запись значения username в переменную, метод to.include.

Последним запросом мы получим данные о созданном нами пользователе.

Знакомый нам эндпоинт /user/{username} с методом GET вернёт нам данные пользователя с именем, которое мы подставим вместо {username}.

Переходим во вкладку *Test* и пишем следующий скрипт:

pm.test("Сохранение имени User 3 в переменную name\_3", function () {

var jsonData = JSON.parse(responseBody);

pm.collectionVariables.set("name\_3", jsonData.username);

});

pm.test("Id User 3 содержится в ответе от сервера", function () {

pm.expect(pm.response.text()).to.include(pm.collectionVariables.get("user\_id"));

});

Следующие тесты помогут нам при выполнении заданий для самостоятельной работы:

* 1. Скрипт первого теста помещает значение имени нового пользователя в переменную **name\_3**.
  2. А скрипт второго теста проверит, содержится ли в тексте ответа **id**, который мы записали в переменную **user\_id** при создании пользователя в предыдущем запросе.

И вот теперь самое время для того, чтобы пояснить: для чего нам нужна эта проверка и зачем мы записывали id в переменную в предыдущем запросе. Дело в том, что система, которую мы тестируем, устроена таким образом, что запрос на получение данных о пользователе идёт через username в эндпоите, а не через id, что было бы корректнее. Ведь если в системе будет создано несколько объектов с одинаковыми именами, в базе данных эти сущности будут храниться в разных областях под разными уникальными id.

Таким образом, данная проверка будет пройдена, если пользователь имеет уникальный username, и может быть (но не всегда!) провалена, если username не уникален и в БД содержатся несколько записей с одинаковыми именами.

Это хороший повод потренироваться в заведении баг-репортов и завести в *Jira* задачу на устранение дефекта!

**Важно!** Вы всегда можете удалить лишних или задублированных пользователей из БД с помощью запроса с методом **DELETE**. В документации данный запрос описан и регламентирован!

## 8.5.2. Сценарное тестирование REST API

Очень часто в реальных проектах приходится тестировать API-запросы не по одному, а вызывая их последовательность, в которой каждый следующий запрос связан с предыдущим. Например, в интернет-магазинах товары объединяются в группы. Если есть запрос на получение товаров определённой группы, то, прежде чем выполнить его, нужно знать id этой конкретной группы. Для того чтобы его получить, нужно выполнить запрос на получение списка групп, в котором и найти нужный id. То есть необходимо выполнить как минимум два запроса.

Нередки случаи, когда нужно выполнить и большее количество запросов. Очень часто последовательность запросов имитирует определённый пользовательский сценарий. Если обратиться к сайту приюта животных, то таким сценарием может быть:

* 1. Открыть приложение.
  2. Открыть карточку питомца.
  3. Нажать на кнопку «Приютить».

Каждое из действий — это один или несколько API-запросов.

В этой части модуля мы реализуем различные сценарии для приложения [Pet Friends](http://petfriends.skillfactory.ru/) в Postman и рассмотрим, как правильно запускать их.

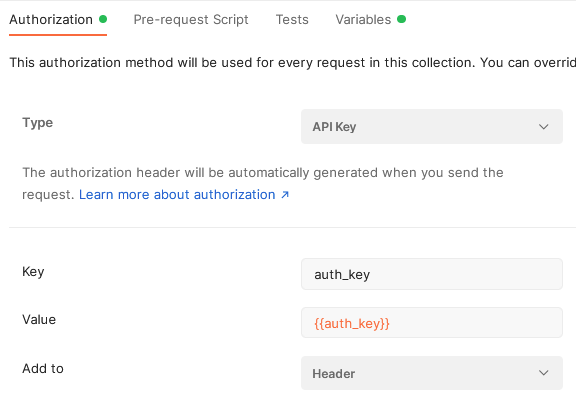
## ****Общая подготовка коллекции для теста****

Для обучения и создания первого сценарного теста вы можете сделать копию вашей коллекции petFriends или вносить изменения в уже созданную коллекцию на свой страх и риск.

В рамках коллекции мы реализуем один простой сценарий:

* 1. Пользователь получает ключ API.
  2. Пользователь добавляет питомца.
  3. Пользователь проверяет, что питомец добавился.
  4. Пользователь удаляет питомца.
  5. Пользователь проверяет, что питомец удалён.

Для начала проверим общие настройки коллекции (клик на названии коллекции):

Рис. 1. Настройки авторизации

Настройки авторизации — через API Key в заголовках. Значение самого ключа будет подгружено из переменных, созданных в ходе теста.

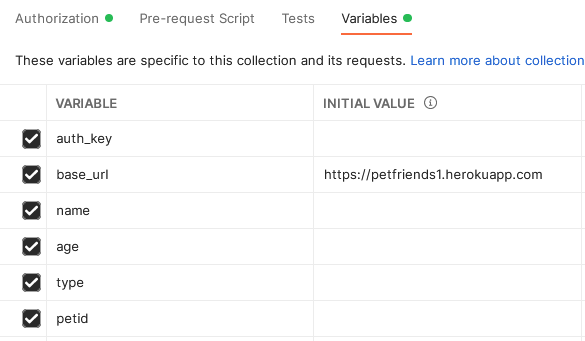
### **Задание 8.5.2.1**

Проверьте настройки авторизации вашей коллекции и приведите их в соответствие с указанными на скриншоте.

**Перечень переменных для коллекции:**

* + auth\_key;
  + base\_url;
  + name;
  + age;
  + type;
  + petid.

Все переменные, кроме base\_url, не имеют значений — они получают значения в ходе выполнения тестов.

Рис. 2. Переменные для тестового сценария

**Изменение тестов для работы с переменными**

**1. Получение ключа и его запись в переменные Postman**

Первый тест должен не только получить значение *APIKEY*, но и установить его для других тестов. Потому в первом тесте, в разделе *Tests*, мы допишем:

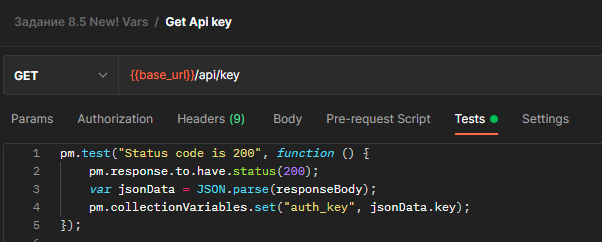
**var** jsonData = JSON.parse(responseBody);

pm.collectionVariables.set("auth\_key", jsonData.key);

**Этот код работает так:**

* + 1. responseBody — полученное от сервера тело ответа преобразуется в словарь методом JSON.parse, а затем значение словаря записывается в переменную jsonData.
    2. collectionVariables — означает, что обращение идёт к переменной коллекции.
    3. set — функция, которая присваивает значение.
    4. А в скобках находятся два параметра: первый — auth\_key — переменная, в которую будет помещено значение, а второй — jsonData.key — значение ключа key, полученное в ответе от сервера.

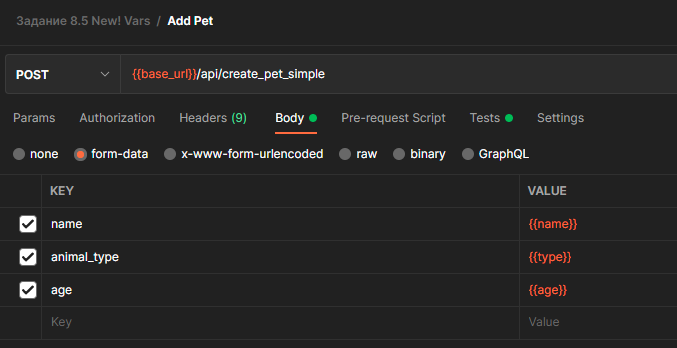
Таким образом, вторая строка дословно читается так: «В переменную коллекции auth\_key записать значение ключа key, полученное в ответе от сервера».



Затем из переменной jsonData читается значение key и записывается в переменную Postman auth\_key.

**2. Добавление питомца с переменными**

Второй запрос нашего сценария добавляет питомца. Как хорошие тестировщики, мы должны провести этот тест с данными различной длины имён, разных возрастов, типов животных. *Postman* может загрузить и использовать данные из файла. В параметрах самого запроса мы просто должны указать переменные:

Рис. 4. Переменные в теле запроса на создание питомца

К подготовке и использованию файла мы приступим позже, а пока продолжим работать с тестами. Если запрос на добавление успешно выполнился, то у нас есть *ID* питомца в ответе сервера. Добавим код, чтобы использовать этот *ID* в дальнейших тестах:

pm.test("Status code is 200", **function** () {

pm.response.to.have.status(**200**);

**var** jsonData = JSON.parse(responseBody);

pm.collectionVariables.set("petid", jsonData.id);

});

**3. Проверка того, что питомец добавлен**

Уже знакомая нам точка /api/pets?filter=my\_pets (метод запроса *GET*) возвращает всех наших питомцев, а самых новых возвращает первыми, мы можем добавить простую проверку, что нулевой ([0]) элемент списка питомцев имеет *ID* и этот *ID*равен тому же *ID*, что вернул сервер после добавления питомца. Проверка выглядит так:

pm.test("first pets with new-created ID", **function** () {

**var** jsonData = JSON.parse(responseBody);

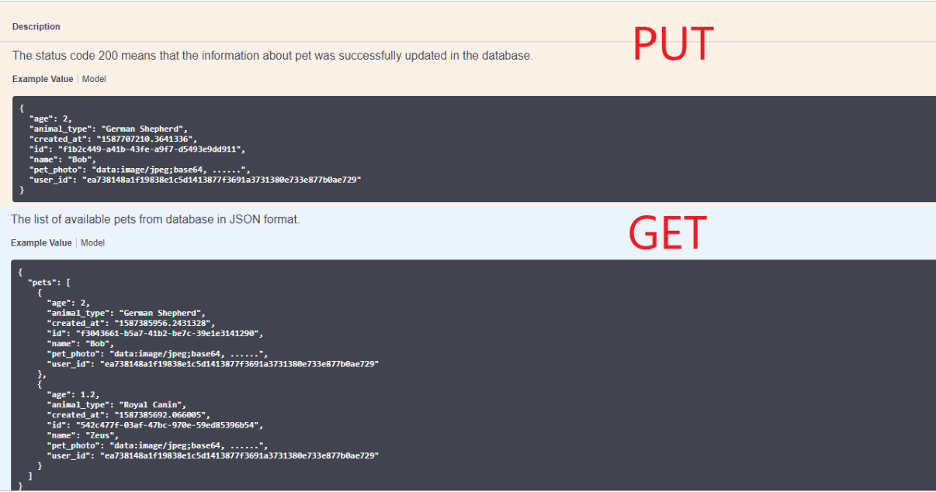
pm.expect(pm.collectionVariables.get("petid")).to.eql(jsonData.pets[**0**].id);

});

Дословно последняя строка скрипта читается так: «Ожидается, что переменная коллекции petid равна значению *ID* нулевого элемента списка pets из ответа от сервера».

**Обратите внимание:** предыдущий запрос устанавливал значение (pm.collectionVariables.**set**), а этот его получает (pm.collectionVariables.**get**).

Это объясняется тем, как построен ответ от сервера после выполнения запроса. Рассмотрим два примера.



Первым на скриншоте указан запрос **PUT**. Обратите внимание, из чего состоит ответ от сервера: **один объект**. Потому что в данном случае запрос изменял одну сущность (карточку питомца) и сервер возвращает нам только эту карточку.

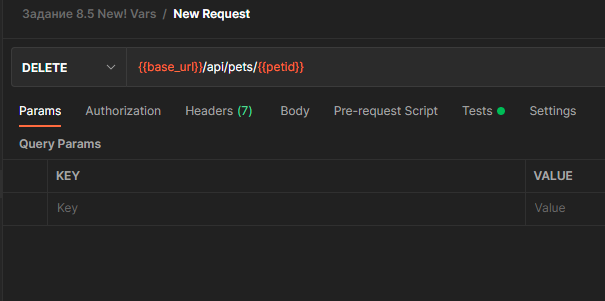
Второй запрос — **GET**. Он направлен на то, чтобы вернуть список всех питомцев. Обратим внимание на его структуру: вложенный массив **pets** с двумя объектами внутри.

Таким образом:

* + в первом случае мы обращаемся сразу к ключам ответа, который мы поместили в переменную **jsonData**, — **jsonData.имя\_ключа**;
  + во втором случае — с указанием вложенного массива **jsonData.pets[0].имя\_ключа**.

**4. Удаление питомца**

В запросе на удаление мы вновь используем переменную коллекции *Postman* petid вместо вручную заданного *ID* питомца. Поскольку в *POST*-запросе была создана новая карточка питомца, а во вкладке *Tests* мы записали *ID* новой карточки в переменную коллекции petid.

Рис. 5. Запрос на удаление с переменной

**Скринкаст: Удаление питомца**

**5. Проверка того, что питомец удалён**

Знакомая нам точка /api/pets?filter=my\_pets возвращает всех наших питомцев, а самых новых возвращает первыми. Таким образом, копируем запрос 3 и вносим изменения в скрипт во вкладке *Tests*, проверяя, что нулевой ([0]) элемент списка питомцев имеет *ID* **НЕ** равный тому *ID*, что вернул сервер после добавления питомца. Потому что мы уже удалили последнюю созданную карточку в запросе 4 и все ключи, с ней связанные, соответственно, текущий нулевой элемент списка — это та карточка, которая была создана до удалённой.

Попробуйте самостоятельно составить тест, который проверит неравенство полученного ID и того, что записан в переменных Postman. Нажмите на «Показать ответ», чтобы проверить себя.

Посмотреть ответ

Тест может выглядеть так:

pm.test("first pets without new-created ID", **function** () {

**var** jsonData = JSON.parse(responseBody);

pm.expect(pm.collectionVariables.get("petid")).to.not.equal(jsonData.pets[**0**].id);

});

Дословно последняя строка нашего скрипта читается следующим образом: «Ожидается, что переменная коллекции petid не равна значению ID нулевого элемента списка pets из ответа от сервера».

## ****Скринкаст: Проверка того, что питомец удалён****

## ****Файлы для теста****

## ****Файл с данными****

Для запуска тестов нам необходим файл с данными. Мы можем создать простой текстовый файл в «Блокноте» — главное, чтобы в файле были заголовки, например:

name, age, type

Tom, 2, кот

Garfield,5, кот

Jerry, 1, мышь

Chip, 5, бурундук

**Совет:** файл с данными мы можем подготовить так, чтоб проверить разные граничные значения. Можно ли дать имя или указать тип питомца длиной в один символ? А если использовать более 256 символов? Можно ли указать отрицательный возраст? А возраст в 99999999 лет?

### **Задание 8.5.2.7**

Создайте в стандартном приложении Windows Блокнот файл и сохраните его с расширением \*.CSV с данными, приведёнными для примера, или с любыми собственными данными.

## ****Файлы подстраховки****

Для подстраховки — [файл CSV](https://drive.google.com/file/d/1C6KG0WSQYBJw-1D1t8SZNW623_dyGhkU/view?usp=sharing) с примером тестовых данных.

## ****Запуск тестов****

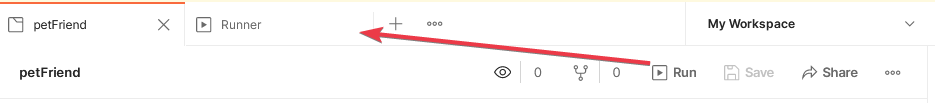
**Перед запуском тестов:**

* 1. Убедитесь, что все открытые вкладки с тестами сохранены (нет красных «точек» на табах):

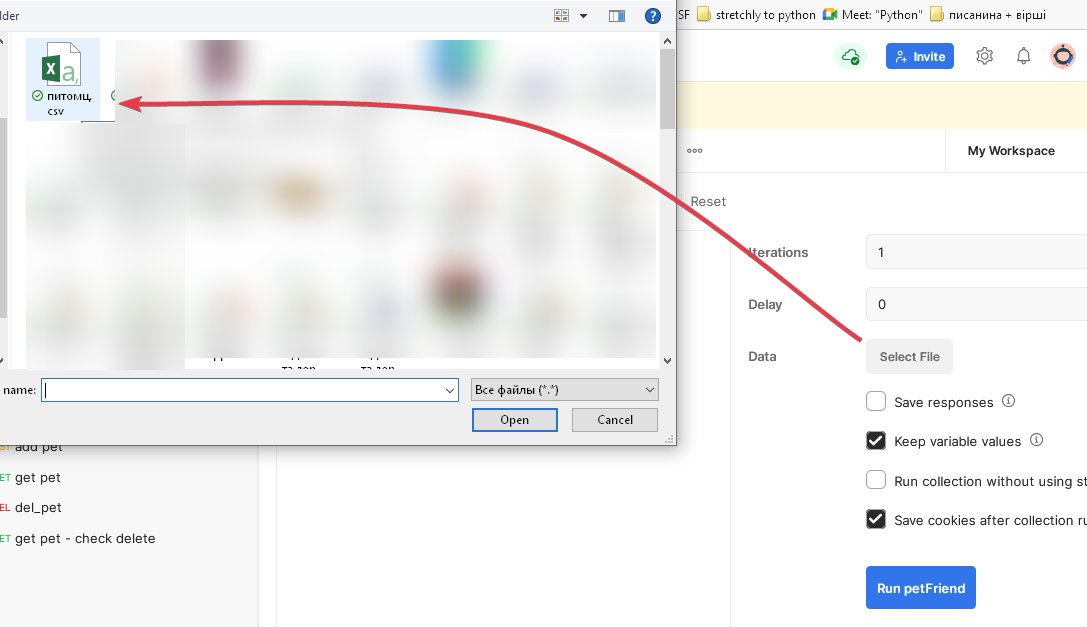
Рис. 6. Пример эндпоинта, изменения в котором не сохранены

* 1. Тщательно проверьте, что внесли все описанные изменения и добавили все указанные в этом юните тесты.

В папке коллекции есть кнопка Run, которая открывает вкладку Runner:

Рис. 7. Кнопка Run, вкладка Runner

Во вкладке Runner мы можем нажать на кнопку Select File и выбрать файл с подготовленными данными и нажать Open (Открыть).

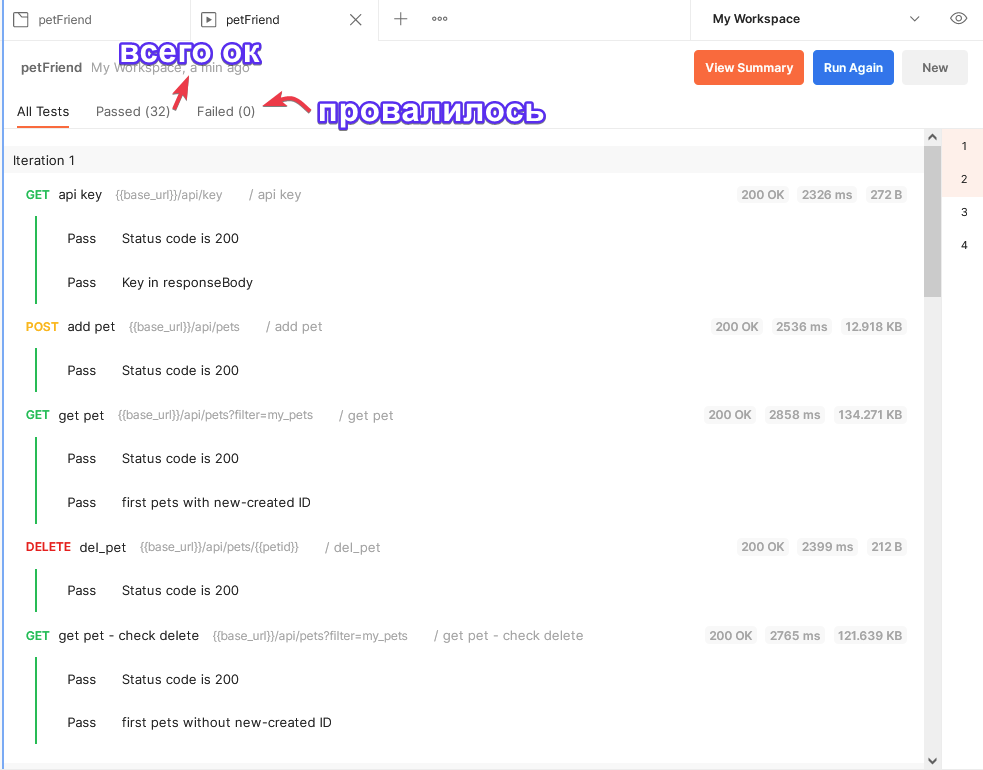
Рис. 8. Выбор файла с данными для тестов

**ВАЖНО:** каждая строчка в загруженном файле приводит к вызову всех точек один раз. Например, четыре строки на пяти тестовых точках дают 20 вызовов. Некоторые точки имеют один тест, другие — два. В итоге выполняется 32 теста.

Один цикл вызовов для одной строки от начала и до конца называется **итерация**.

После этого мы можем нажать на синюю кнопку Run petFriend.

Запросы будут выполняться по очереди, и окно Runner будет заполняться успешно (или неуспешно) пройденными тестами.

Рис. 9. Результаты выполнения тестов на вкладке Runner

## ****Интерпретация результатов****

После того как тест завершён, результаты и ваша реакция на них могут быть следующими:

* 1. Все тесты прошли успешно.
     + - Немного порадовались и пошли искать ошибки дальше.
  2. Тесты при запросе к одному из эндпоинтов провалились, к другим тесты прошли нормально.
     + - Ошибка в статус-коде или сообщении от сервера для данного эндпоинта. Сам запрос на сервере, скорее всего, отработал нормально, раз связанные эндпоинты не поломались.
       - Ваша ошибка в составлении сценария — последующие вызовы не связаны с движением данных на сервере.
  3. Провалились тесты к одной конечной точке и ко всем последующим.
     + - Почти наверняка обнаружен блокирующий баг, при котором дальнейшее прохождение сценария невозможно.
  4. Из серии запусков провален только один запрос или несколько связанных, но только в одной из итераций. В остальных итерациях все тесты успешны.
     + - Вы хорошо подготовили значения для файла запуска, и в одном из запусков сервер неправильно обработал указанное значение.
       - Случайный сбой при отправке данных. При повторной отправке всё ок.
  5. Все тесты провалились.
     + - Убедитесь, что ваш сервер точно работает и у вас есть интернет. Проверьте, что происходит при обращении к сайту в браузере. Сообщите об ошибке вашей команде.

## ****Скринкаст: Создание CSV-файла и работа с запросами коллекции****

## ****Подведём итоги****

Итак, в этом юните мы научились тому, как подготовить и провести тестовый сценарий, который проверяет взаимодействие нескольких точек API, а также научились интерпретировать результаты тестового пробега.

Помните, что в учебных целях мы реализовали лишь один сценарий. Для рабочего проекта таких сценариев должно быть намного, намного больше — например, добавление питомца и изменение его данных запросом PUT.

 НазадВперёд

© Все права защищены

* [Help center](https://connect.yandex.ru/forms/5c5a843f6b6a50004cde4563/)
* [Политика конфиденциальности](https://skillfactory.ru/terms_of_use)
* [Пользовательское соглашение](https://skillfactory.ru/user_agreement)

Built on

## ****Файлы для теста****

## ****Файл с данными****

Для запуска тестов нам необходим файл с данными. Мы можем создать простой текстовый файл в «Блокноте» — главное, чтобы в файле были заголовки, например:

name, age, type

Tom, 2, кот

Garfield,5, кот

Jerry, 1, мышь

Chip, 5, бурундук

**Совет:** файл с данными мы можем подготовить так, чтоб проверить разные граничные значения. Можно ли дать имя или указать тип питомца длиной в один символ? А если использовать более 256 символов? Можно ли указать отрицательный возраст? А возраст в 99999999 лет?

### **Задание 8.5.2.7**

Создайте в стандартном приложении Windows Блокнот файл и сохраните его с расширением \*.CSV с данными, приведёнными для примера, или с любыми собственными данными.

## ****Файлы подстраховки****

Для подстраховки — [файл CSV](https://drive.google.com/file/d/1C6KG0WSQYBJw-1D1t8SZNW623_dyGhkU/view?usp=sharing) с примером тестовых данных.

## ****Запуск тестов****

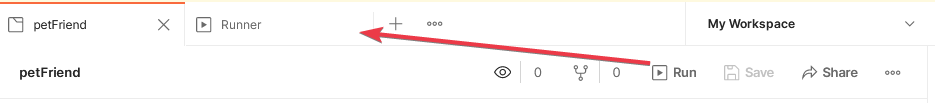
**Перед запуском тестов:**

* 1. Убедитесь, что все открытые вкладки с тестами сохранены (нет красных «точек» на табах):

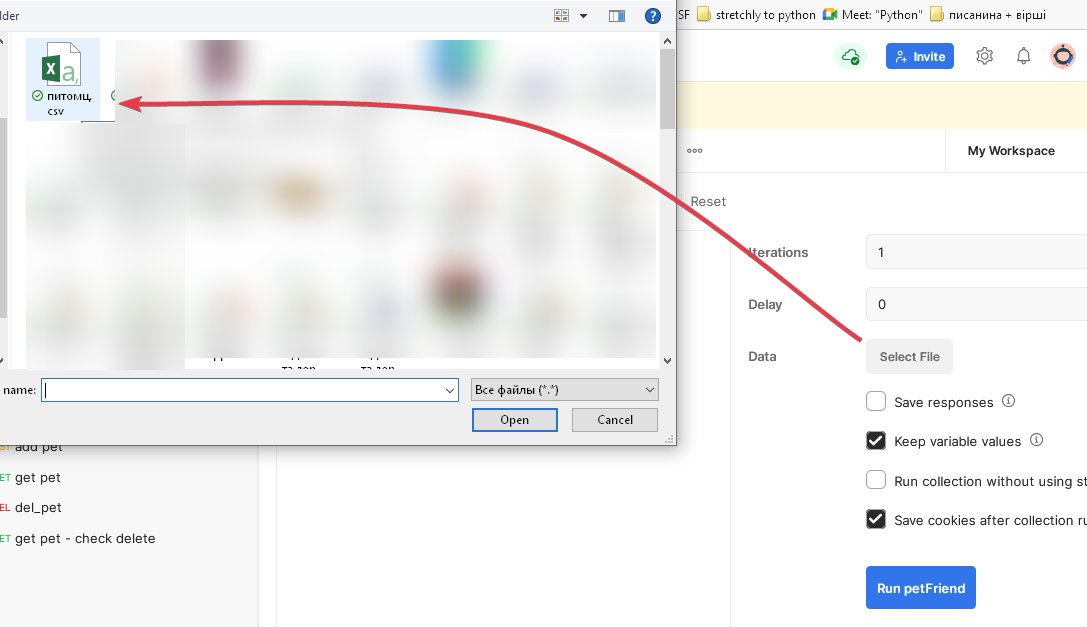
Рис. 6. Пример эндпоинта, изменения в котором не сохранены

* 1. Тщательно проверьте, что внесли все описанные изменения и добавили все указанные в этом юните тесты.

В папке коллекции есть кнопка Run, которая открывает вкладку Runner:

Рис. 7. Кнопка Run, вкладка Runner

Во вкладке Runner мы можем нажать на кнопку Select File и выбрать файл с подготовленными данными и нажать Open (Открыть).

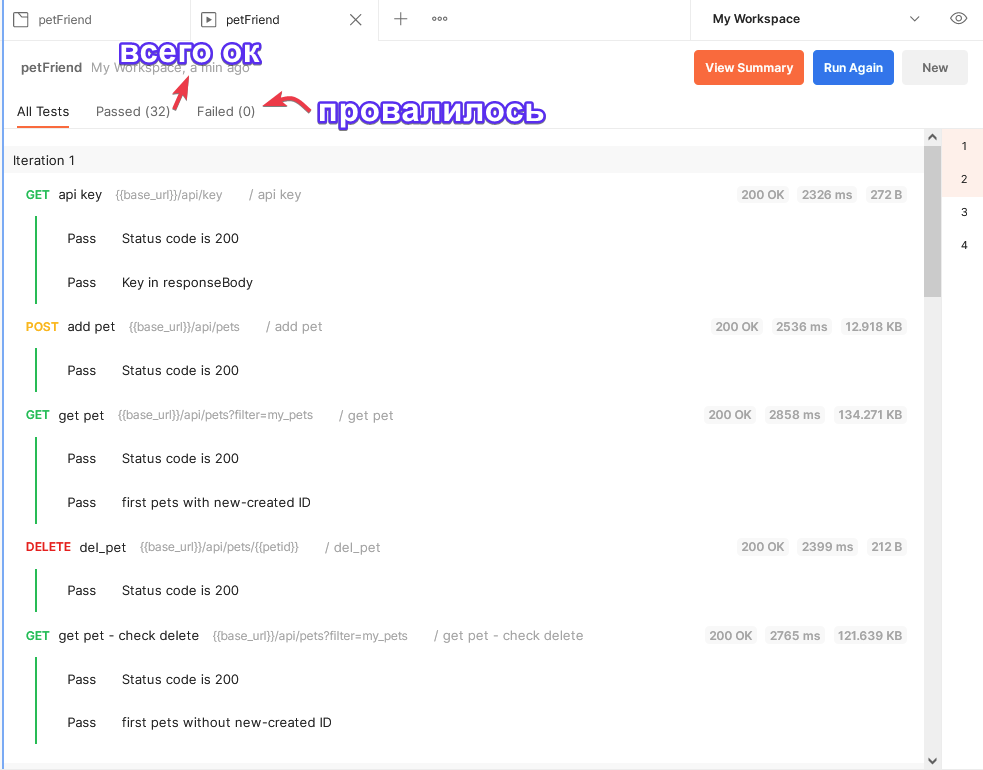
Рис. 8. Выбор файла с данными для тестов

**ВАЖНО:** каждая строчка в загруженном файле приводит к вызову всех точек один раз. Например, четыре строки на пяти тестовых точках дают 20 вызовов. Некоторые точки имеют один тест, другие — два. В итоге выполняется 32 теста.

Один цикл вызовов для одной строки от начала и до конца называется **итерация**.

После этого мы можем нажать на синюю кнопку Run petFriend.

Запросы будут выполняться по очереди, и окно Runner будет заполняться успешно (или неуспешно) пройденными тестами.

Рис. 9. Результаты выполнения тестов на вкладке Runner

## ****Интерпретация результатов****

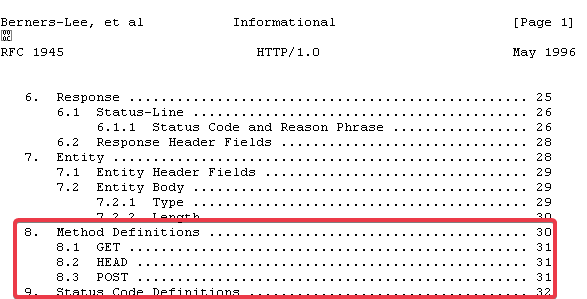
После того как тест завершён, результаты и ваша реакция на них могут быть следующими:

* 1. Все тесты прошли успешно.
     + - Немного порадовались и пошли искать ошибки дальше.
  2. Тесты при запросе к одному из эндпоинтов провалились, к другим тесты прошли нормально.
     + - Ошибка в статус-коде или сообщении от сервера для данного эндпоинта. Сам запрос на сервере, скорее всего, отработал нормально, раз связанные эндпоинты не поломались.
       - Ваша ошибка в составлении сценария — последующие вызовы не связаны с движением данных на сервере.
  3. Провалились тесты к одной конечной точке и ко всем последующим.
     + - Почти наверняка обнаружен блокирующий баг, при котором дальнейшее прохождение сценария невозможно.
  4. Из серии запусков провален только один запрос или несколько связанных, но только в одной из итераций. В остальных итерациях все тесты успешны.
     + - Вы хорошо подготовили значения для файла запуска, и в одном из запусков сервер неправильно обработал указанное значение.
       - Случайный сбой при отправке данных. При повторной отправке всё ок.
  5. Все тесты провалились.
     + - Убедитесь, что ваш сервер точно работает и у вас есть интернет. Проверьте, что происходит при обращении к сайту в браузере. Сообщите об ошибке вашей команде.

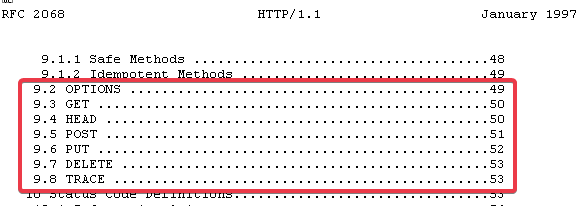
## 8.6 OPTIONS, HEAD, CONNECT и прочие HTTP-запросы

Интернет — явление, которое пребывает в постоянной динамике. Эта страница, которую вы сейчас читаете в браузере, и сотни других, которые открыты в соседних вкладках, — это результат работы сотен  тысяч запросов, распознавания и выполнения десятков тысяч скриптов и согласования и слаженной работы тысяч протоколов.

Первая версия протокола HTTP (1.0), опубликованная в 1996 году, содержала всего три метода запросов: GET, HEAD и POST.

Рис. 1. Документация по первой [версии протокола HTTP/1.0](https://www.rfc-editor.org/rfc/rfc1945.txt)

В 1997 году была опубликована [версия HTTP/1.1](https://www.rfc-editor.org/rfc/rfc2068.txt), содержащая семь типов запросов + отдельный документ на метод PATCH.

Рис. 2. Документация обновлённого протокола HTTP 1.1

Сейчас в интернете также распространён [протокол HTTP/2](https://datatracker.ietf.org/doc/html/rfc7540). Уже пять лет активно разрабатывается [протокол HTTP/3](https://quicwg.org/base-drafts/draft-ietf-quic-http.html).

Если вам придётся вплотную тестировать API, то необходимо будет мониторить и изучать все эти изменения, но даже сейчас в интернете на каждом отдельно взятом сервере могут не использоваться все методы или типы запросов.

Чаще всего используются GET и POST, чуть реже — PUT и DELETE. Эти запросы мы уже рассмотрели в этом модуле, но наше знакомство с тестированием API будет неполным, если мы хоть немного не затронем другие методы запросов.

## ****HEAD****

Метод HEAD аналогичен GET, просто сервер не отправляет тело ответа. При этом сервер должен отправить все те заголовки, которые бы он отправил при формировании GET-запроса для того же ресурса, что и в HEAD-запросе. Этот метод может использоваться для получения метаданных о ресурсе, например о возможных его представлениях. Вся эта информация передаётся в заголовках ответа.

Вы можете использовать HEAD-запросы для тестирования гипертекстовых ссылок на корректность, доступность и дату модификации.

[Общая документация по методу](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/HEAD)

Такой тип запроса редко тестируют или документируют: проще сразу послать (использовать в API) запрос GET и получить те же заголовки и содержимое ответа и протестировать всё разом.

## ****CONNECT****

Метод CONNECT является наиболее распространённой формой HTTP-туннелирования. Основная специфика использования метода состоит в том, что между клиентом и сервером часто существует множество промежуточных точек. Вызывая метод CONNECT, клиент просит прокси-сервер HTTP перенаправить TCP-соединение в желаемый пункт назначения. Затем сервер устанавливает соединение от имени клиента. Запрос создаёт прямой туннель с желаемым сервером.

Да, это всё очень сложные материи. Демонстрация такого типа соединений выходит далеко за рамки этого курса, поэтому больше деталей и примеров не будет.

Если вам любопытно, то подробнее можно почитать в [официальном стандарте](https://tools.ietf.org/html/rfc7231#section-4.3.6), в [переводе от Mozilla](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/CONNECT) или на [Wikipedia](https://en.wikipedia.org/wiki/HTTP_tunnel).

В практической работе вам, скорее всего, никогда не придётся тестировать CONNECT, но знать о нём полезно для понимания работы интернета.

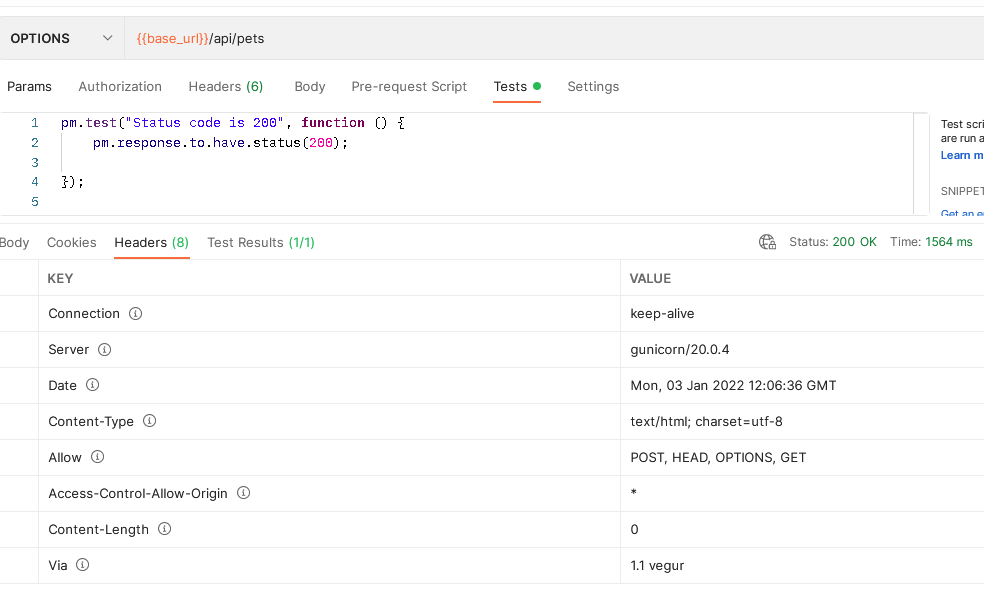
## ****OPTIONS****

Метод OPTIONS используется для получения информации о вариантах взаимодействия с ресурсом, сервером или промежуточным узлом, находящимся между клиентом и сервером (прокси-сервер, шлюз или туннель). Вызов HTTP-запроса с методом OPTIONS позволит определить параметры и требования, связанные с ресурсом или сервером.

Если запрос выполнен успешно, то код состояния ответа должен быть 200, а в заголовках ответа будут возвращены параметры, характеризующие данный ресурс и методы взаимодействия с ним. Например, в заголовке Allow будут возвращены допустимые методы взаимодействия с этим ресурсом, а в заголовке Content-Type — формат представления этого ресурса. Запросы OPTIONS не кэшируются.

Запустим Postman и выполним HTTP-запрос с методом OPTIONS для ресурса /api/pets приложения Pet Friends.

* 1. Создайте новый запрос, выберите метод OPTIONS и в поле URL запроса введите адрес ресурса:  {{base\_url}}/api/pets или <http://petfriends.skillfactory.ru/api/pets>.
  2. Нажмите на кнопку Send.
  3. Перейдите на вкладку Headers в разделе ответа.

Рис. 3. Вкладка Headers в разделе ответа содержит ключ Allow — перечень допустимых методов при обращении к эндпоинту /api/pets

## ****PATCH****

Метод PATCH, согласно [документации](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PATCH), должен работать, как команда UPDATE — т. е. обновлять часть данных. Но ведь для этого используется метод PUT — возразит нам внимательный студент и будет частично прав. Как мы обсуждали, PUT целиком заменяет указанный целевой объект данными, представленными в теле запроса. [PATCH](https://datatracker.ietf.org/doc/html/rfc5789) же может заменять объект частично (например, только имя питомца или только фото).

Изначально это выглядит как преимущество: нам не надо высылать все данные — достаточно послать несколько полей, и данные обновятся. Но стандарты также выдвигают требования к целостности данных, и поэтому сервер должен убедиться, что клиент завершил «пропатчивать» данные. Кроме того, в это время эти же данные может начать «патчить» другой пользователь, и возникнет «конфликт версий».

Конечно, все возникающие проблемы можно решить, однако при использовании PUT решения проще. По этой причине, а ещё по историческим причинам, появившись позже PUT, метод PATCH не очень широко используется.

В Web API PetFriends метод PATCH не реализован.