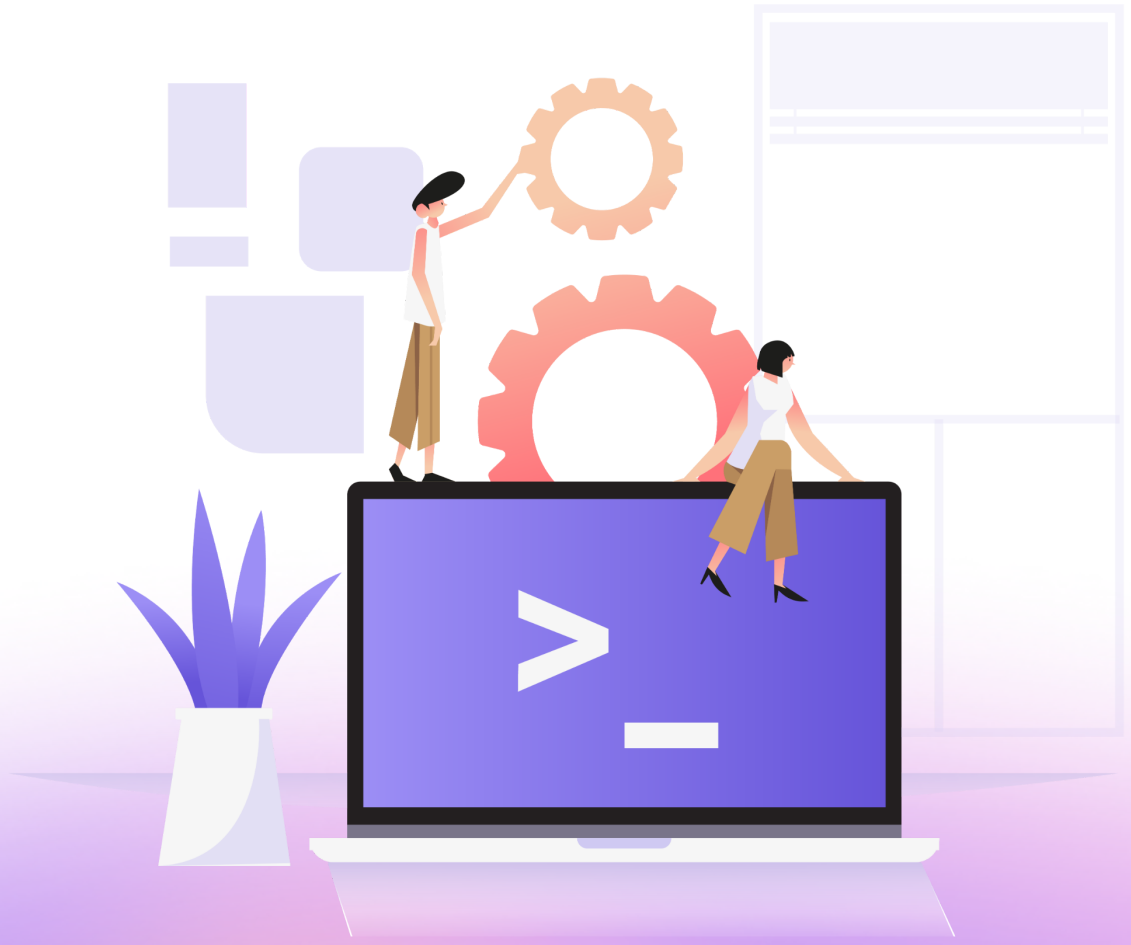
GeekBrains

Тестирование веб-приложений

REST API



На этом уроке

1. Узнаем, что такое REST API.
2. Посмотрим, какие существуют методы.
3. Научимся отправлять запросы через Postman и валидировать ответы.

Оглавление

REST API

REST (Representational State Transfer) — передача состояния представления. Это архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой системы.

Это «официальное» определение REST. Но если сказать более простыми словами, то REST:

- архитектура клиент-серверного приложения — как клиент и сервер устроены «внутри»;
- описание протокола взаимодействия — каким образом передаётся информация между клиентом и сервером.

Для клиент-серверных приложений, построенных по принципам REST и не нарушающих накладываемых REST ограничений, применяется термин RESTful.

В RESTful-сервисах клиенты взаимодействуют с объектами на сервере, которые обычно называются «ресурсы». Ресурс — любая сущность, с которой взаимодействует клиент. Если взять, например, интернет-магазин, то в нём может быть ресурс «пользователь», «категория», «товар», «заказ», «бронирование», «корзина» и так далее.

Restful-сервисы позволяют внешним системам получать доступ к текстовым представлениям ресурсов и управлять через определённый набор операций. Зачастую это стандартные действия «создать», «прочитать», «обновить», «удалить». В англоязычной нотации это часто обозначается аббревиатурой CRUD:

- Create (Создать);
- Read (Прочитать);
- Update (Обновить);
- Delete (Удалить).

В отличие от сервисов на основе SOAP, для которого есть WSDL, «официального» стандарта для RESTful API нет. Дело в том, что REST представляет собой архитектурный стиль, в то время как SOAP считается протоколом. Хотя REST — не стандарт сам по себе, большинство RESTful-реализаций используют такие стандарты, как HTTP, URL, JSON. Впрочем, REST позволяет использовать данные в текстовых машиночитаемых форматах — XML, YAML и других, просто JSON выбран как один из наиболее компактных форматов записи.

Хотя RESTful-сервисы стали популярными относительно недавно, технологии, на которых строится REST, существуют примерно столько же, сколько и интернет. А описание собственно RESTful-архитектуры опубликовано более 20 лет назад.

Термин REST ввёл Рой Филдинг (Roy Fielding), один из создателей HTTP, в 2000 году. В своей диссертации «Архитектурные стили и дизайн сетевых программных архитектур» он подвёл

теоретическую основу под способ взаимодействия клиентов и серверов в интернете, абстрагировав его и назвав «передачей представительного состояния» (Representational State Transfer).

Филдинг описал концепцию построения распределённого приложения, при которой каждый запрос (REST-запрос) клиента к серверу содержит в себе исчерпывающую информацию о желаемом ответе сервера, то есть желаемом представительном состоянии, и сервер не обязан сохранять информацию о состоянии клиента («клиентской сессии»). Но сама концепция, которая стала называться REST, развивалась параллельно с HTTP 1.1. Последний появился в 1996–1999 годах, основываясь на существующем дизайне HTTP 1.0, появившемся в 1996 году.

Таким образом, сам REST — это не что-то принципиально новое, он лишь грамотно использует давно существующие технологии.

Далее представлены ключевые свойства архитектуры, описанные Роем Филдингом.

Свойства архитектуры REST

Свойства архитектуры, которые зависят от ограничений, наложенных на REST-системы:

1. **Производительность** — взаимодействие компонентов системы может быть доминирующим фактором производительности и эффективности сети, с точки зрения пользователя.
2. **Масштабируемость** для обеспечения большого числа компонентов и взаимодействий компонентов.

Рой Филдинг описывает влияние архитектуры REST на масштабируемость следующим образом:

1. Простота унифицированного интерфейса.
2. Открытость компонентов к возможным изменениям для удовлетворения изменяющихся потребностей, даже при работающем приложении.
3. Прозрачность связей между компонентами системы для сервисных служб.
4. Переносимость компонентов системы путём перемещения программного кода вместе с данными.
5. Надёжность — это устойчивость к отказам на уровне системы, если есть отказы отдельных компонентов, соединений или данных.

Требования к REST-сервису

К REST-системам предъявляются определённые требования, и только при полном их выполнении система может называться RESTful. Большинство этих требований напрямую вытекают из особенностей HTTP.

Модель «клиент — сервер»

В основе этого требования лежит разграничение функций между клиентом и сервером. Разделение функций на клиента, предоставляющего интерфейс конечному пользователю, и сервера, хранящего и обрабатывающего данные, повышает переносимость кода клиентского приложения на другие платформы, а упрощение серверной части улучшает масштабируемость и надёжность.

Отсутствие состояния

HTTP-протокол, используемый для взаимодействия между клиентом и сервером, не подразумевает сохранения информации о соединении между запросами от клиента к серверу. Все запросы составляются так, чтобы сервер получил всю информацию для выполнения запроса. Таким образом, на клиента накладывается требование хранить некоторую идентификационную информацию, благодаря которой можно однозначно определить клиента и поддерживать сессионную работу с сервером. К такой идентификационной информации относятся, например, логины или пароли и различные виды токенов.

Кеширование

Как и браузеры, REST-клиенты выполняют кеширование ответов сервера. Ответы сервера, в свою очередь, имеют соответствующее обозначение — кешируемые или не кешируемые, чтобы исключить работу клиента с устаревшими данными. Для кешируемых данных определяется срок жизни кеша. Правильное использование кеширования способно уменьшить количество запросов между клиентом и сервером, тем самым повышая производительность всей системы.

Единообразие интерфейса

Унифицированный интерфейс — фундаментальное требование дизайна REST-сервисов. К интерфейсу API предъявляются следующие четыре условия:

Идентификация ресурсов

Все ресурсы идентифицируются в запросах, например, с использованием определённого формата URI. Ресурсы отделены от представлений, которые возвращаются клиентам. Например, сервер посылает данные из базы данных в формате HTML, XML или JSON, и ни один из них не считается реальным типом хранения данных, входящим в БД сервера.

Манипуляция ресурсами через представление

Запрашиваемый ресурс может вернуться в различных форматах, таких как JSON, YAML, HTML, XML, или SVG, PNG, JPG и т. д. Эти форматы считаются *представлениями* ресурса. Ресурс на сервере всегда один и тот же и хранится в некотором внутреннем формате сервера, понятном только ему, например, запись в базе данных. А вот для клиента надо передать данные именно в том формате, в котором их запрашивает клиент. Общий список возможных форматов, понятных и клиентам, и серверам, ограничен, а каждый формат чётко определён. Эти форматы называются MIME-типы, их полный список — на странице Media Types на сайте IANA.

Серверные приложения с REST API поддерживают несколько представлений одного и того же ресурса по одному и тому же URI. Чтобы получить требуемое представление из возможных вариантов, клиент должен указать, какое представление (формат данных) он хочет получить. Это делается через HTTP-заголовок *Accept*, который клиент передаёт серверу при каждом запросе ресурса.

Например, разные клиенты делают запросы к одному и тому же ресурсу как:

```
GET /customers/128
Accept: text/xml
```

или:

```
GET /customers/128
Accept: application/json
```

Всё зависит от того, какое внутреннее представление информации они используют: XML или JSON. Ресурс в обоих случаях будет один — информация о пользователе с данным идентификатором, а представления — разные.

С другой стороны, сервер не только отправляет различные представления, но и получает данные в разных представлениях для создания или обновления ресурсов. RESTful-приложения позволяют клиентам указывать их представление при отправке данных на сервер. Это выполняется через HTTP-заголовок *Content-Type*, который клиент передаёт серверу:

```
PATCH /customers/128
Content-type: text/xml
```

или:

```
PATCH /customers/128
Content-type: application/json
```

В первом случае клиент отправляет данные в теле запроса в формате XML, а во втором — в JSON.

«Самоописываемые» сообщения

Каждое сообщение содержит достаточно информации, чтобы понять, каким образом его обрабатывать. В качестве примера обратимся к предыдущему пункту: заголовок *Content-type* сообщает, какой именно из парсеров на сервере надо использовать для обработки этого запроса.

Гипермедиа как средство изменения состояния приложения

Клиенты изменяют состояние системы только через действия, которые динамически определяются в гипермедиа на сервере. Исключая простые точки входа в приложение, клиент не может предположить, что доступна какая-то операция над каким-то ресурсом, если не получил информацию об этом в предыдущих запросах к серверу. Нет универсального формата для предоставления ссылок между ресурсами.

Слои

Клиент обычно неспособен точно определить, взаимодействует ли он напрямую с сервером, или же с промежуточным узлом в сложной иерархической структуре. Применение промежуточных серверов позволяет повысить масштабируемость благодаря балансировке нагрузки и распределённого кеширования. Промежуточные узлы также подчиняются политике безопасности, чтобы обеспечить конфиденциальность информации.

Код по требованию (необязательное ограничение)

REST позволяет расширить функциональность клиента благодаря загрузке кода (как правило, JavaScript) с сервера. То есть не вся реализация происходит на клиенте, а сервер может предоставить часть требуемых функций.

Например, мы подключаем платёжную систему к сайту. Наш сайт — это клиент, который будет обращаться к внешней платёжной системе через API. И когда пользователь откроет платёжную форму, а сайт отправит запрос на совершение платежа, внешняя система пришлёт с ответом JavaScript-код, который проверит базовые параметры безопасности системы.

Преимущества

Приложения, не соответствующие приведённым условиям, не считаются REST-приложениями. Если же все условия соблюдены, то приложение получит следующие преимущества:

1. Надёжность благодаря отсутствию необходимости сохранять информацию о состоянии клиента, которая может быть утеряна.
2. Производительность благодаря использованию кеша.
3. Масштабируемость.
4. Ясность системы взаимодействия особенно важна для приложений обслуживания сети.

5. Простота интерфейсов.
6. Портативность компонентов.
7. Лёгкость внесения изменений.
8. Способность эволюционировать, приспосабливаясь к новым требованиям.

Методы

Так как RESTful-сервис основан на HTTP, он использует те же стандартные методы протокола: OPTIONS, HEAD, GET, POST, PUT, PATCH, DELETE, TRACE, CONNECT. В большинстве существующих сервисов реализуются методы GET, POST, PUT, PATCH, DELETE. Другие же HTTP-методы практически не встречаются, но их использование разрешается в REST-сервисах.

HTTP-метод	**Ресурс коллекции https://api.example.com/collection/**	**Ресурс экземпляра https://api.example.com/collection/{id}**
GET	Получение списка URI-элементов, входящих в эту коллекцию.	Получение описания конкретного ресурса в теле ответа.
POST	Создание экземпляра объекта в коллекции на основе данных, переданных в теле запроса. URI созданного экземпляра автоматически назначается при создании и передаётся клиенту в заголовке ответа Location.	Создание экземпляра коллекции в экземпляре этой коллекции. Может использоваться в системе с произвольным уровнем вложенности, но на практике встречается редко.
PUT	Полная замена данных во всех экземплярах коллекции данными, переданными в теле запроса, или создание нового ресурса **коллекции**, если его ещё нет.	Замена всех данных в экземпляре коллекции данными, полученными в теле запроса, или создание нового **экземпляра**, если его ещё нет.
PATCH	Обновление данных в экземплярах коллекции данными, полученными в теле запроса. Создаёт ресурс коллекции, если его ещё нет.	Обновление данных в экземпляре ресурса. Создаёт экземпляр, если его ещё нет.
DELETE	Удаление всех экземпляров в коллекции.	Удаление всех данных указанного экземпляра.

Существуют общепринятые практики по использованию тех или иных методов для работы с ресурсами.

POST /collection — создание нового элемента. Если сделать несколько одинаковых запросов с одним набором данных, то, в зависимости от логики приложения, мы получаем или N одинаковых экземпляров коллекции, или ошибку на втором и последующем запросах, что элемент с такими данными уже существует.

PUT /collection/{id} — полная перезапись данных в элементе. Например, если в запросе нет некоторых полей, которые уже есть в этом экземпляре, при обновлении они удалятся.

PATCH /collection/{id} — обновление поля в указанном экземпляре. Перепишутся только те поля, которые указаны в теле запроса. Остальные данные в экземпляре коллекции останутся без изменений.

Следующие варианты, как правило, не используются:

POST /collection/{id}, PATCH /collection, PUT /collection

Но конкретный список ресурсов и методов зависит от конкретного API.

Ответы

Обязательный компонент HTTP-ответа — код состояния HTTP.

Вспомним некоторые, наиболее распространённые, коды при работе с REST API.

200 OK — ответ успешно получен

Как правило, возвращается после успешного GET-запроса, сообщая, что ресурс по этому адресу есть (GET /animals/cat/01).

201 Created — элемент создан

Возвращается после успешного POST/PUT-запроса (POST /animals/cats) и сообщает, что новый элемент в указанном ресурсе создан. Адрес нового ресурса возвращается в заголовке Location.

204 No Content — нет содержимого

Присылается сервером, если в результате POST- или DELETE-запроса по той или иной причине не обработан ресурс, но URI корректен, а запрос корректен.

400 Bad Request — задан неверный запрос

Как правило, возвращается, если запрос создан неверно — не хватает каких-то заголовков, неверная структура внутри тела запроса.

401 Unauthorized — пользователь не авторизован

Если ресурс требует авторизации, а в запросе нет требуемой авторизационной информации (логин-пароль, токен, сертификат), или она неверна, то в ответе придёт 401.

403 Forbidden — доступ запрещён

Если какому-то пользователю запрещён доступ к запрашиваемому ресурсу, то вернётся код 403. Он означает, что аутентификация прошла успешно, но у пользователя нет соответствующих прав.

404 Not Found — не найдено, ресурс по этому URI отсутствует

Есть несколько вариантов возврата кода 404. Во-первых, этот код возвращается, когда задаётся действительно неверный адрес, и по нему в принципе нет и не должно быть ресурса. Во-вторых, если по этому адресу когда-то был ресурс, но удалён запросом DELETE, то система также будет возвращать 404. Например, после отправки DELETE-запроса можно перепроверить, что ресурс действительно удалён.

405 Method Not Allowed — метод не разрешён

Этот код ответа приходит в том случае, если на URI ресурса отправляется запрос с методом, который там применяться не может. Например, система не разрешает удалять коллекции, тогда запрос вида DELETE /animals/cats вернёт 405. А если запрос в этом контексте корректен и успешно выполнен (DELETE /animals/cats/42), то код будет 200.

409 Conflict — конфликт

Такой код ответа отправляется в ответ на POST/PUT/PATCH-запросы, если невозможно создать или обновить указанную запись. Например, если какое-то поле должно быть уникальным.

Коды серии 3xx и 5xx, как правило, зависят от состояния и настройки сервера, и напрямую к REST API редко имеют отношение.

Ниже представлена таблица, кратко характеризующая возможные статусы:

HTTP-метод	**Операция**	**Коллекция (/customers)**	**Элемент (/customers/{id})**
POST	Создать	201 (Created), заголовок 'Location' со ссылкой /customers/{id} на новый ресурс.	404 (Not Found) — когда ресурс отсутствует. 409 (Conflict) — если ресурс уже существует.
GET	Прочитать	200 (OK), список элементов коллекции.	200 (OK) — содержимое одного элемента. 404 (Not Found) — если элемент с этим ID не существует.

PUT	Обновить/ Заменить	405 (Method Not Allowed).	200 (OK) или 204 (No Content). 404 (Not Found) — если элемент с этим ID не существует.
PATCH	Обновить/ Изменить	405 (Method Not Allowed).	200 (OK) или 204 (No Content). 404 (Not Found) — если элемент с этим ID не существует.
DELETE	Удалить	405 (Method Not Allowed).	200 (OK) — в случае успешного удаления. 404 (Not Found) — если элемент с этим ID не существует.

JSON

JSON (JavaScript Object Notation) — простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. JSON — текстовый формат, полностью независимый от языка реализации, но он использует соглашения, знакомые программистам C-подобных языков, таких как C, C++, C#, Java, JavaScript, Perl, Python и многих других. Эти свойства делают JSON очень удобным языком для обмена данными.

JSON базируется на двух структурах данных:

1. Коллекция пар «ключ — значение». В разных языках эта концепция реализуется как объект, запись, структура, словарь, хеш, именованный список или ассоциативный массив.
2. Упорядоченный список значений. В большинстве языков реализуется как массив, вектор, список или последовательность.

Объект — неупорядоченный набор пар «ключ — значение». Объект начинается с открывающей фигурной скобки «{» и заканчивается закрывающей фигурной скобкой «}». Каждое имя сопровождается двоеточием «:», пары «ключ — значение» разделяются запятой.

Массив — упорядоченная коллекция значений. Массив начинается с открывающей квадратной скобки «[» и заканчивается закрывающей квадратной скобкой «]». Значения разделяются запятой.

Значение представляет собой **строку** в двойных кавычках, **число**, ***true***, ***false***, ***null***, объект или массив. Эти структуры могут быть вложенными.

Строка — коллекция нуля или больше символов Unicode, заключённая в двойные кавычки. Строковый символ представляется как односимвольная строка.

Пример записи JSON:

```
{
    "firstname": "Sally",
    "lastname": "Brown",
    "totalprice": 111,
    "depositpaid": true,
    "bookingdates": {
        "checkin": "2013-02-23",
        "checkout": "2014-10-23"
    },
    "additionalneeds":  ["Breakfast", "Transfer"]
}
```

Авторизация в REST API

В отличие от обычных веб-приложений, работающих в браузере, RESTful API обычно не сохраняет информацию о состоянии. Это означает, что технологии сессий и cookie могут быть неприменимы. Следовательно, раз состояние аутентификации пользователя не сохранится в сессиях или cookie, каждый запрос должен приходить вместе с определённым видом информации об аутентификации.

Общепринятая практика — для аутентификации пользователя с каждым запросом отправляется секретный токен доступа. Так как токен доступа может использоваться для уникальной идентификации и аутентификации пользователя, запросы к API всегда отсылаются через HTTPS, чтобы предотвратить атаки «человек посередине».

Есть различные способы отправки авторизационной информации:

1. HTTP Basic Auth: логин и пароль кодируются в base64 и отправляются в заголовке запроса.

   Authorization — в незашифрованном виде. В таком варианте пароль хранится на клиенте, поэтому способ авторизации применяется весьма в ограниченном перечне вариантов, например, когда в качестве клиента выступает другое серверное приложение в рамках той же сети.

```
Authorization: Basic YWRtaW46c2VjcmV0
```

2. Bearer Auth: токен доступа выдаётся пользователю API сервером авторизации и отправляется

   API-серверу через HTTP Bearer Token:

```
Authorization: Bearer FFFF70it7tzNsHddEiq0BZ0i-OU8S3xV
```

Bearer-аутентификация — это схема HTTP-аутентификации, где используются токены безопасности, называемые bearer-токенами. Название «Bearer-Аутентификация» можно понимать, как «предоставить доступ предъявителю этого токена». Bearer-токен представляет собой зашифрованную строку, обычно генерируемую сервером в ответ на запрос входа в систему. Клиент должен отправить этот токен в заголовке Authorization при выполнении запросов к защищённым ресурсам:

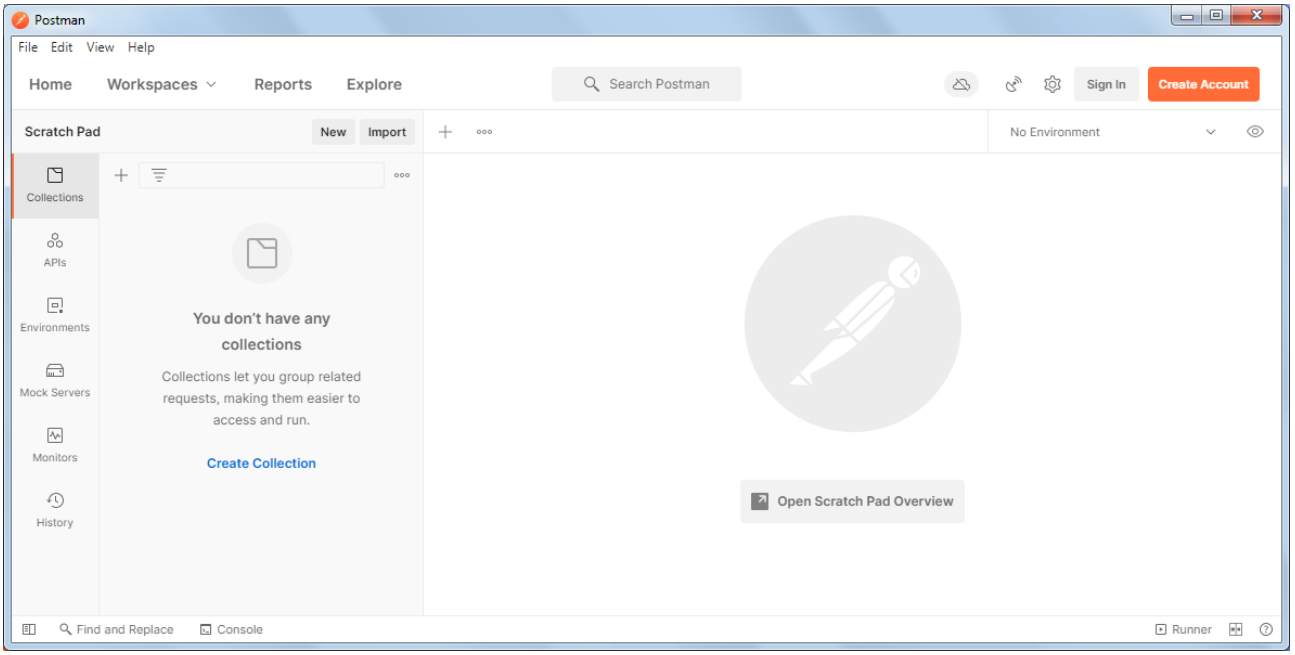
```
Authorization: Bearer <токен>
```

Схема аутентификации Bearer изначально появилась как часть OAuth 2.0, но сейчас зачастую используется сама по себе. Аналогично обычной аутентификации, Bearer-аутентификация применяется только по HTTPS (SSL).

POSTMAN

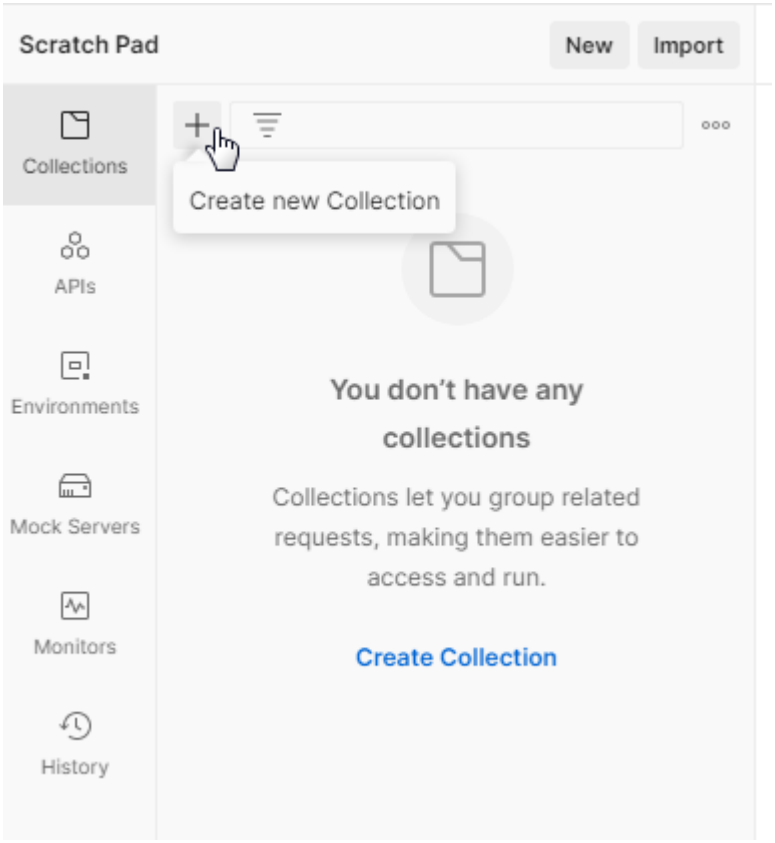
Скачиваем инструмент с этого сайта и устанавливаем.

После запуска появится основное окно:

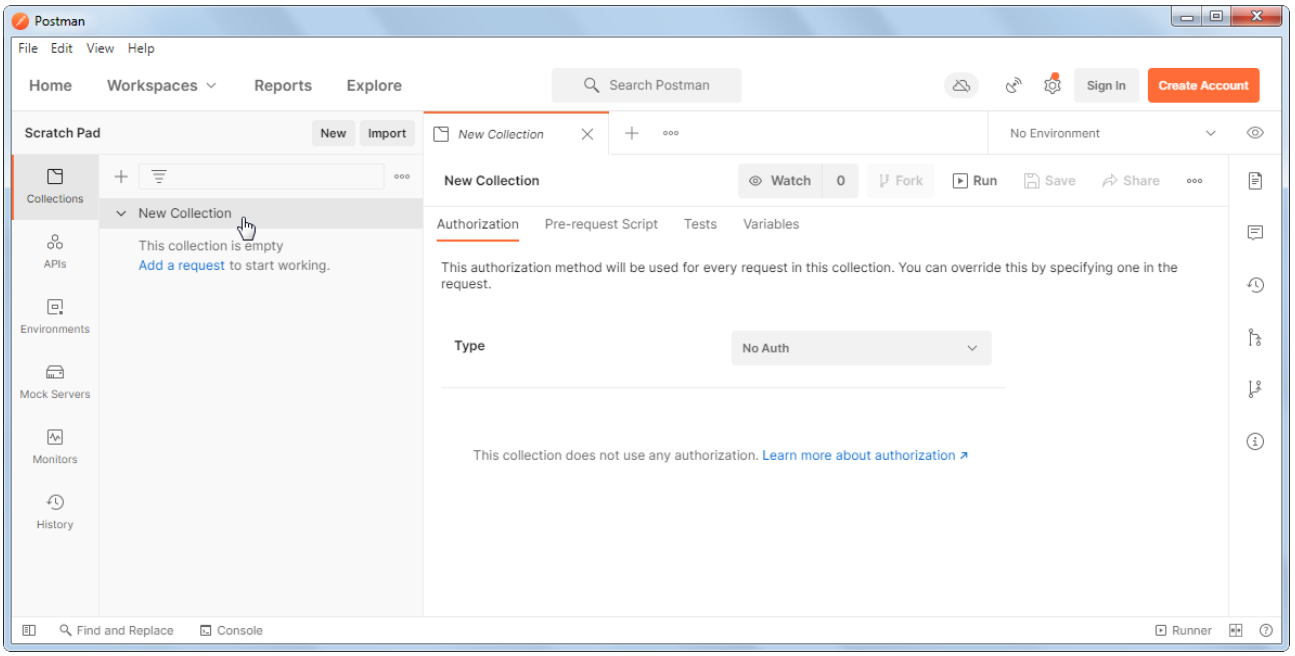


По умолчанию открыта вкладка Collections, которая содержит список коллекций — специальных контейнеров для запросов. Коллекции — это сущность программы Postman, служащая для группировки запросов.

Сейчас надо создать первую коллекцию. Для этого нажимаем на значок «+» вверху списка или кликаем на ссылку Create Collection.



В списке появится коллекция New Collection. Кликаем на её заголовок — откроется окно редактирования свойств:

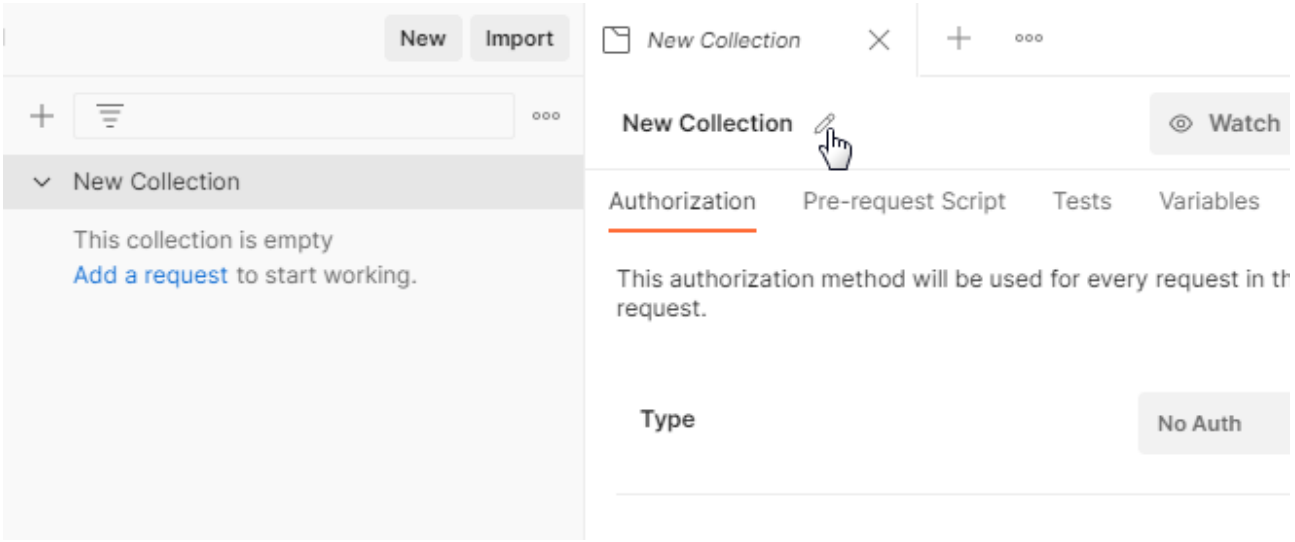


В качестве примера поработаем с Restful Booker. Это простой пример, созданный для изучения тестирования REST API.

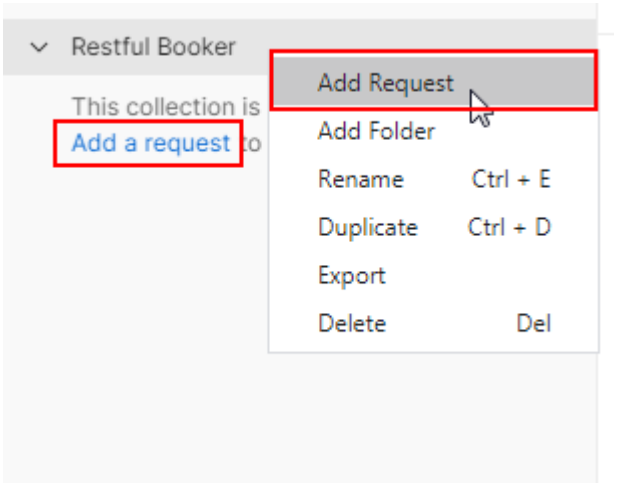
Документация по этому API — на этой странице.

Так как у RESTful API нет такого документа, как WSDL, все параметры, свойства, ограничения приходится изучать самостоятельно. Для этого создаётся обширная документация для каждого API, запроса и ответа.

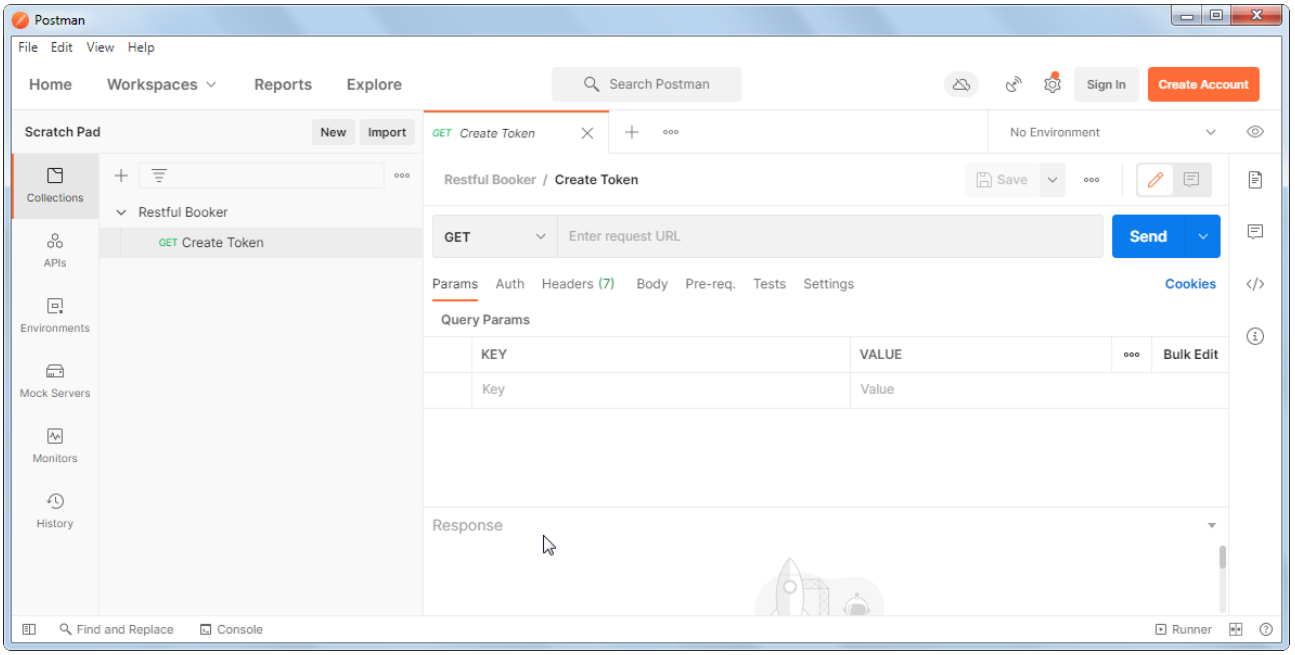
Чтобы стало понятно, с какой же системой мы работаем, сразу переименуем коллекцию в Restful Booker:



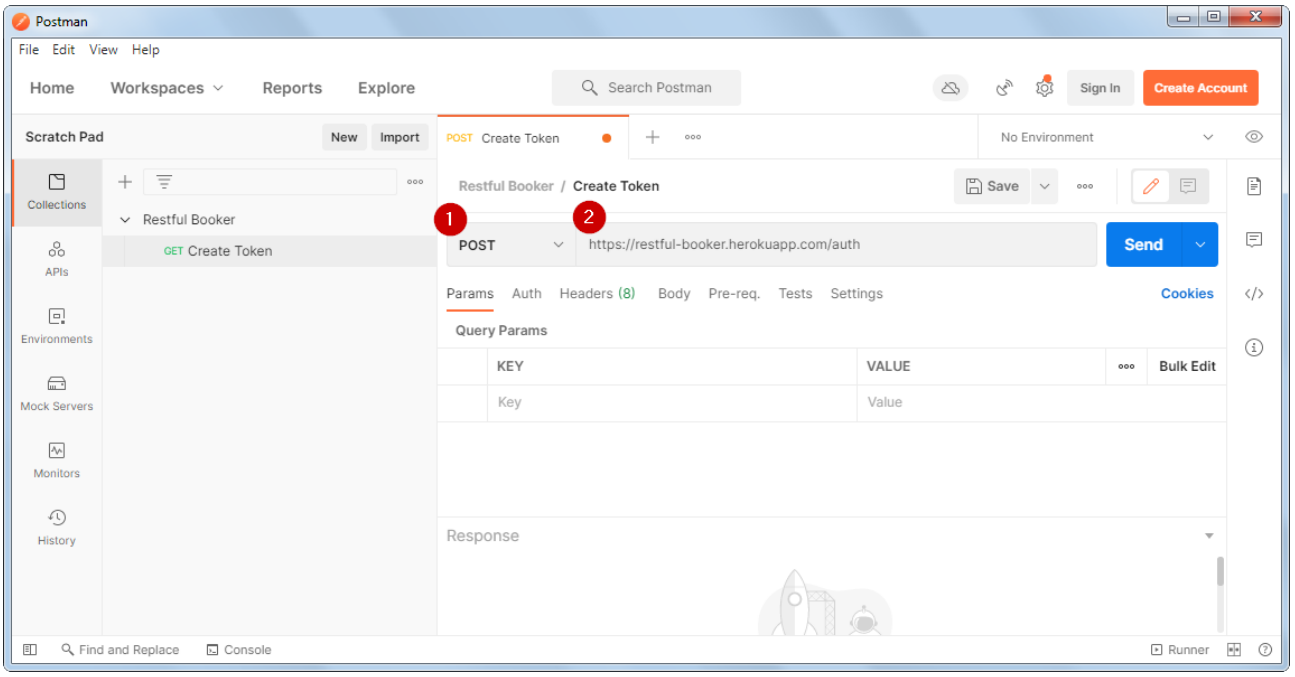
Итак, для начала разберёмся с авторизацией и получим токен доступа. Создадим новый запрос типа GET. Чтобы создать новый запрос, нажимаем правой кнопкой мыши по коллекции и выбираем **Add Request** или кликаем на ссылку **Add a request**:



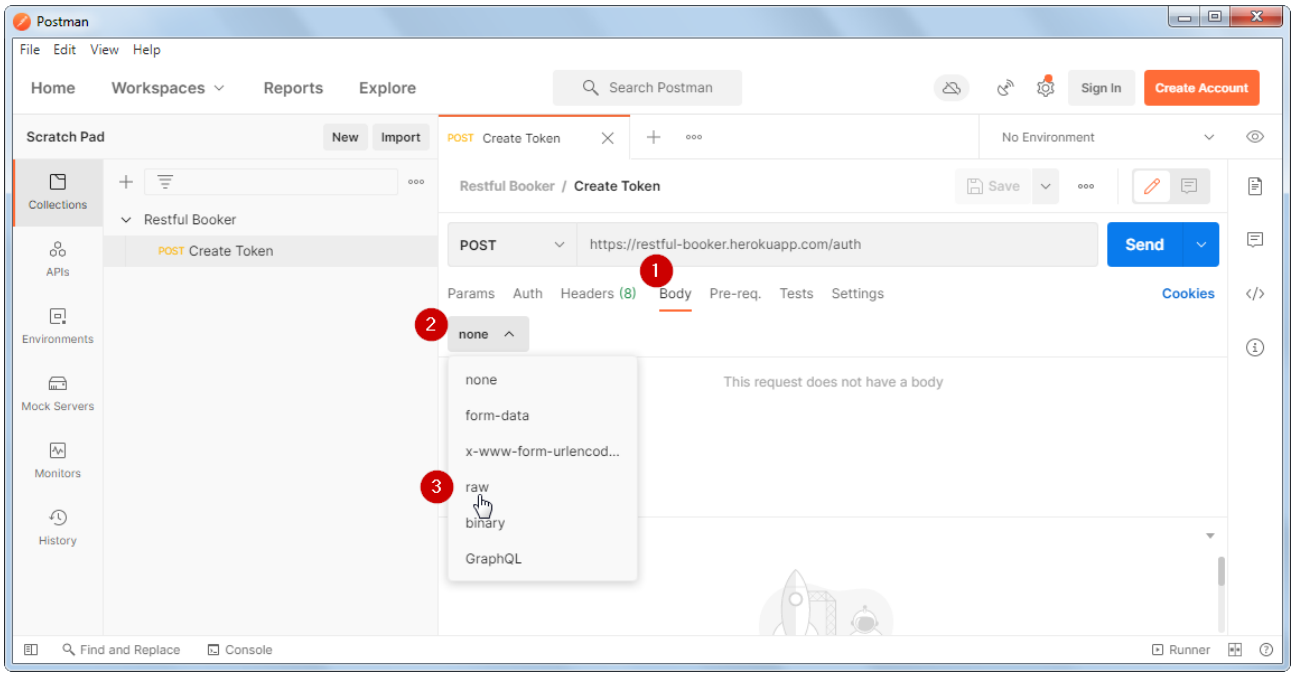
В списке появляется новый GET-запрос New Request, а в правой части окна открывается диалог редактирования запроса. Задаём имя запроса, в нашем случае это Create Token:



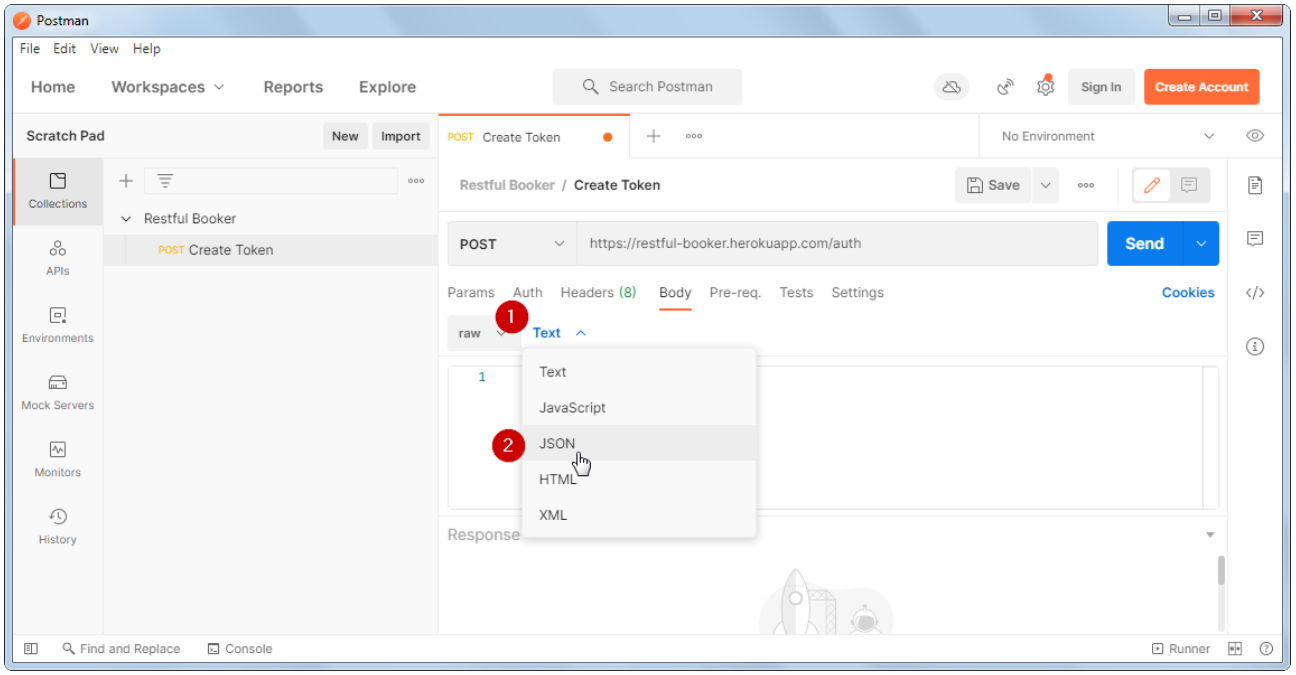
Выбираем метод POST и вводим URL запроса, в нашем случае — https://restful-booker.herokuapp.com/auth.



Далее зададим тело запроса. В форме редактирования запроса открываем вкладку Body, задаём тип — raw:

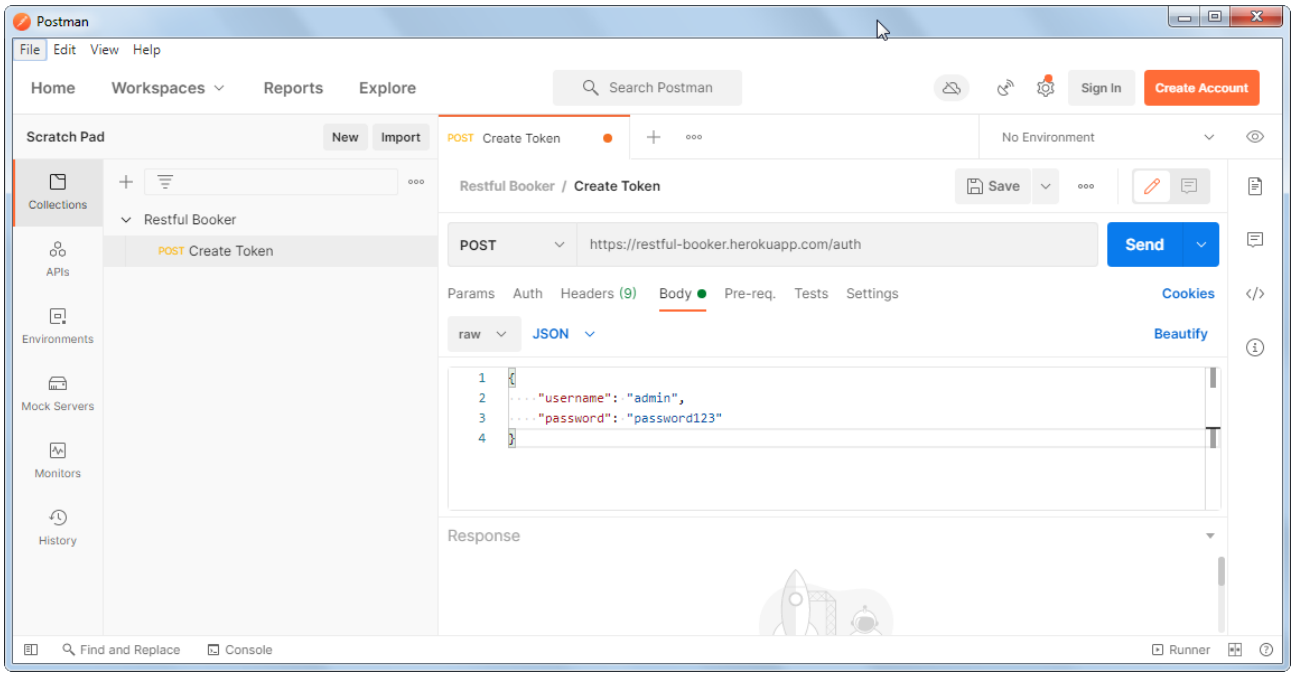


Далее выбираем формат JSON:

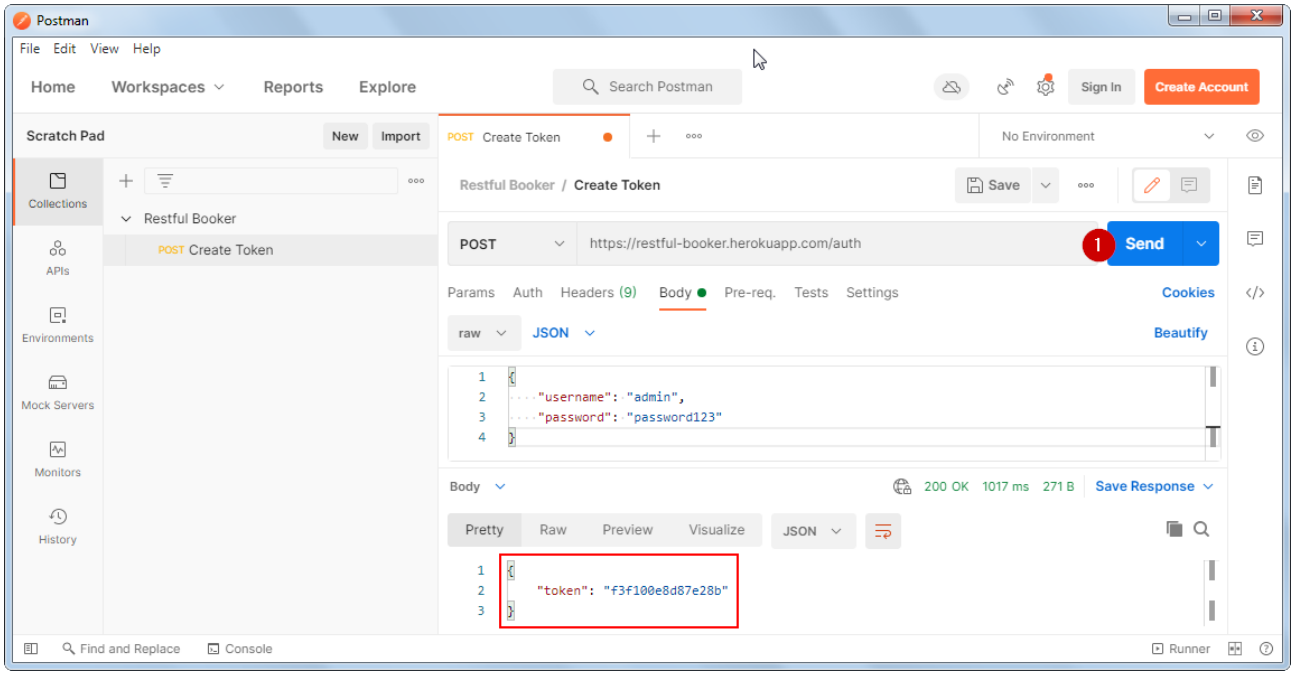


И указываем следующее тело запроса (как в документации):

```
{
    "username": "admin",
    "password": "password123"
}
```



И отправляем запрос. В ответе получим токен:

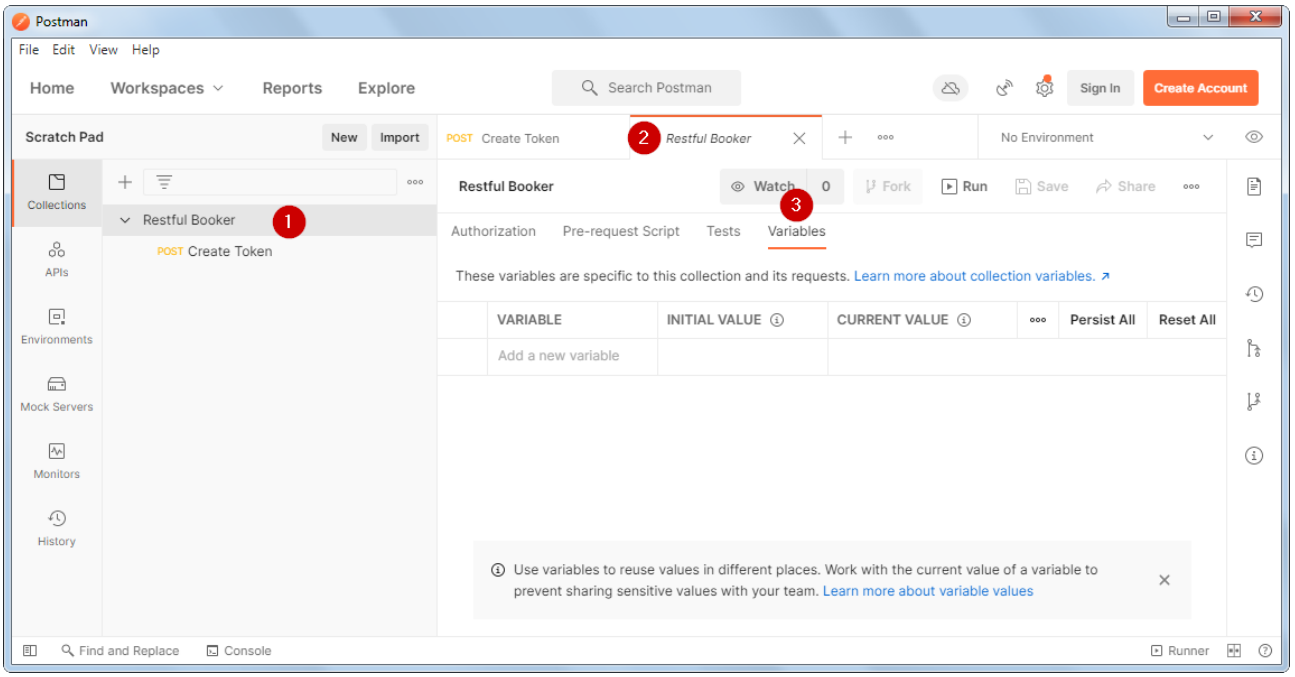



```
{
    "token": "f3f100e8d87e28b"
}
```

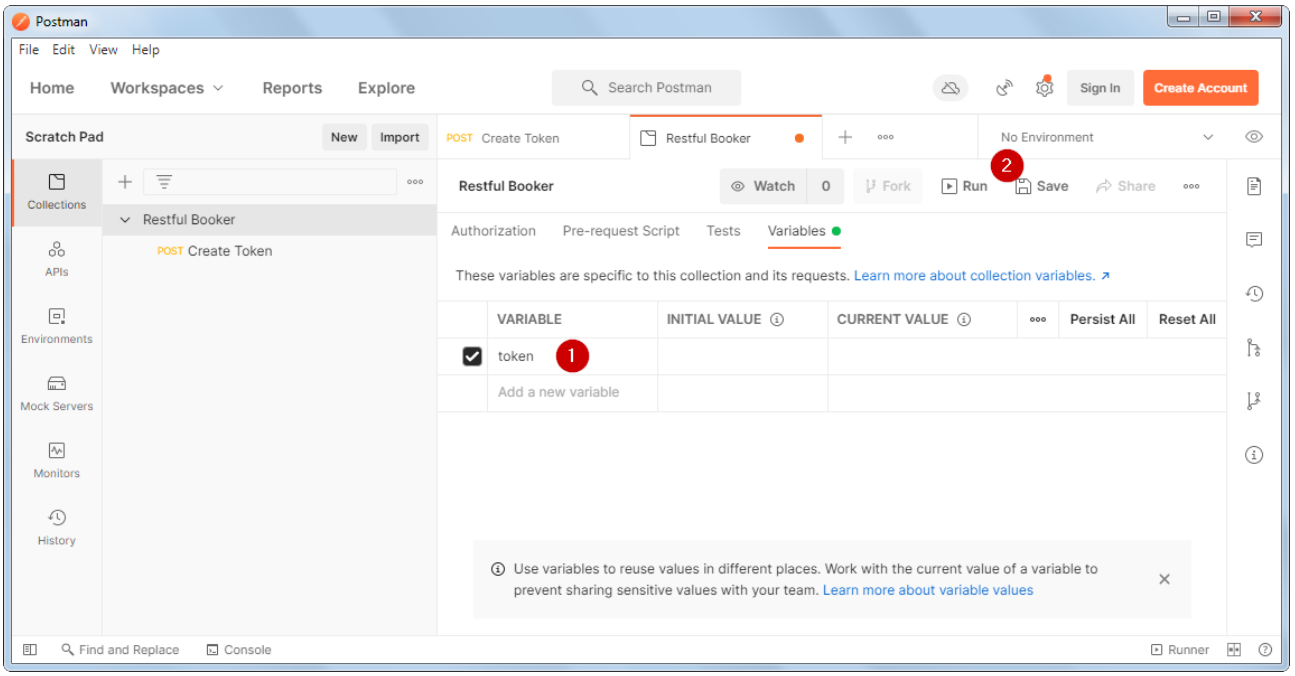
При каждом запросе мы будем получать новый токен.

Если у вас не пришёл токен, или сервер вернул ошибку — внимательно посмотрите, правильно ли составлен запрос: URL, HTTP-метод, тело запроса.

Теперь надо сохранить токен для дальнейшей работы: он используется для авторизации нескольких типов запросов. Для хранения токена создадим переменную в Collection и запишем в неё значение токена. Выбираем в списке слева коллекцию. Откроется окно редактирования коллекции.



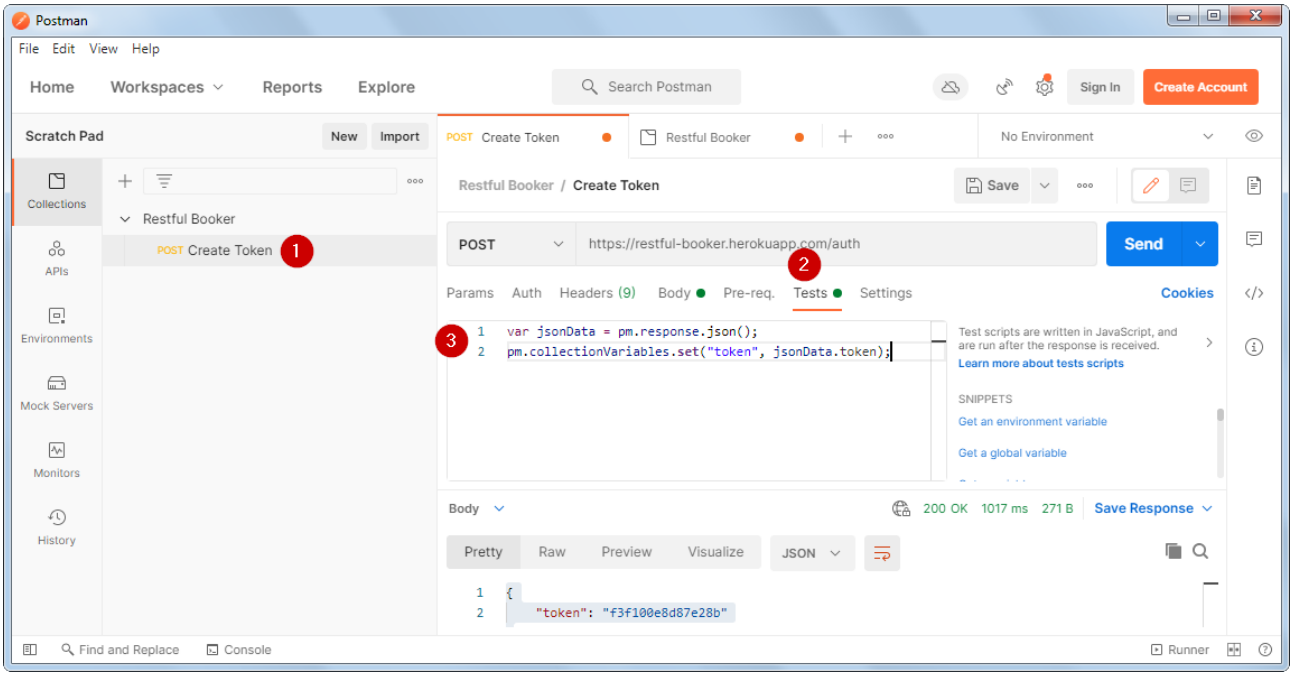
Далее создаём новую переменную, например, token, и нажимаем Save:



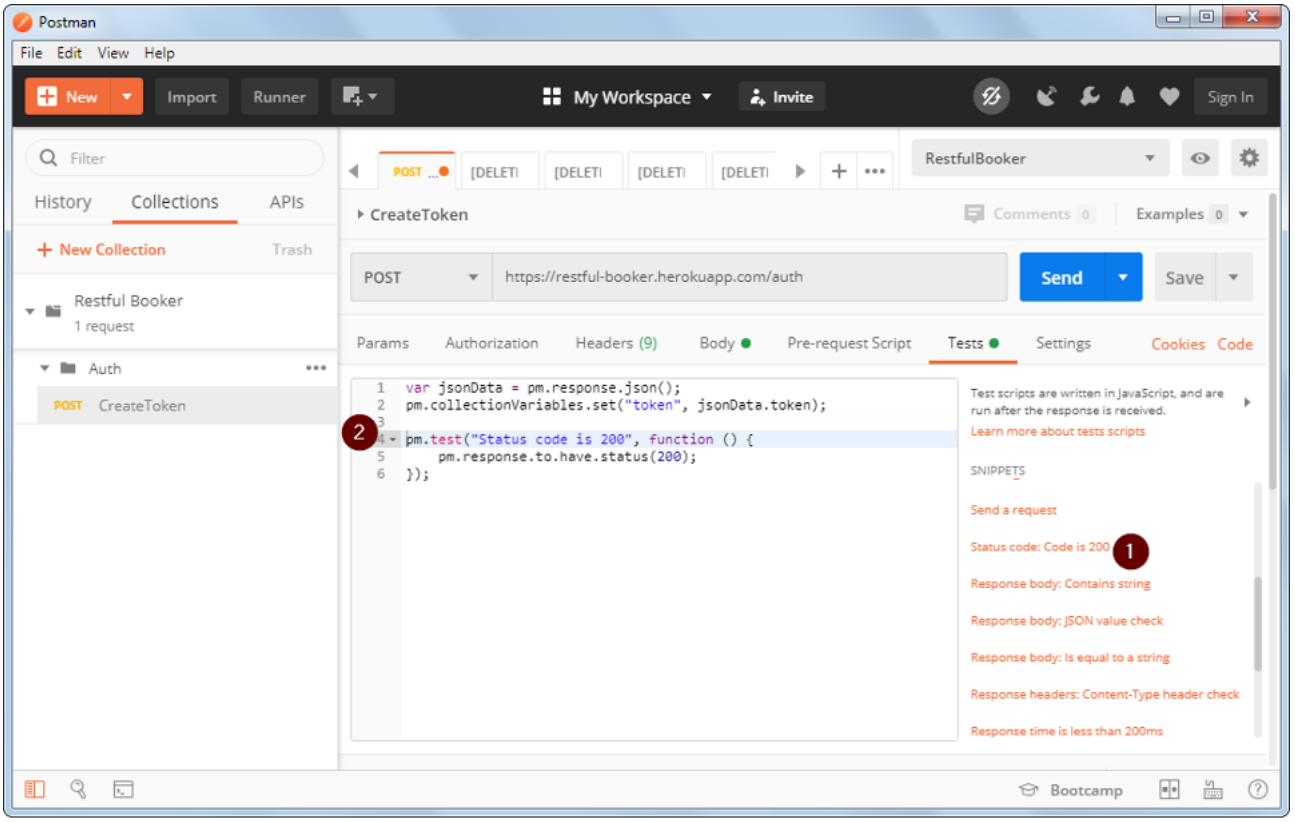
Чтобы сохранить токен в переменную, надо написать скрипт, который будет получать значение из ответа и сохранять его в указанную переменную. За программирование и, в частности, действия после получения ответа появляется вкладка Tests. Снова выбираем запрос Create Token в списке коллекций или запросов, идём во вкладку Tests и вставляем следующий код:

```
var jsonData = pm.response.json();
pm.collectionVariables.set("token", jsonData.token);
```

Первая строка сохраняет json-ответ в переменную jsonData, а вторая — значение, находящееся в json-строке jsonData в ключе token, в переменную token.



Можно воспользоваться готовым сниппетом (примером кода) и добавить проверку, что код ответа — 200:



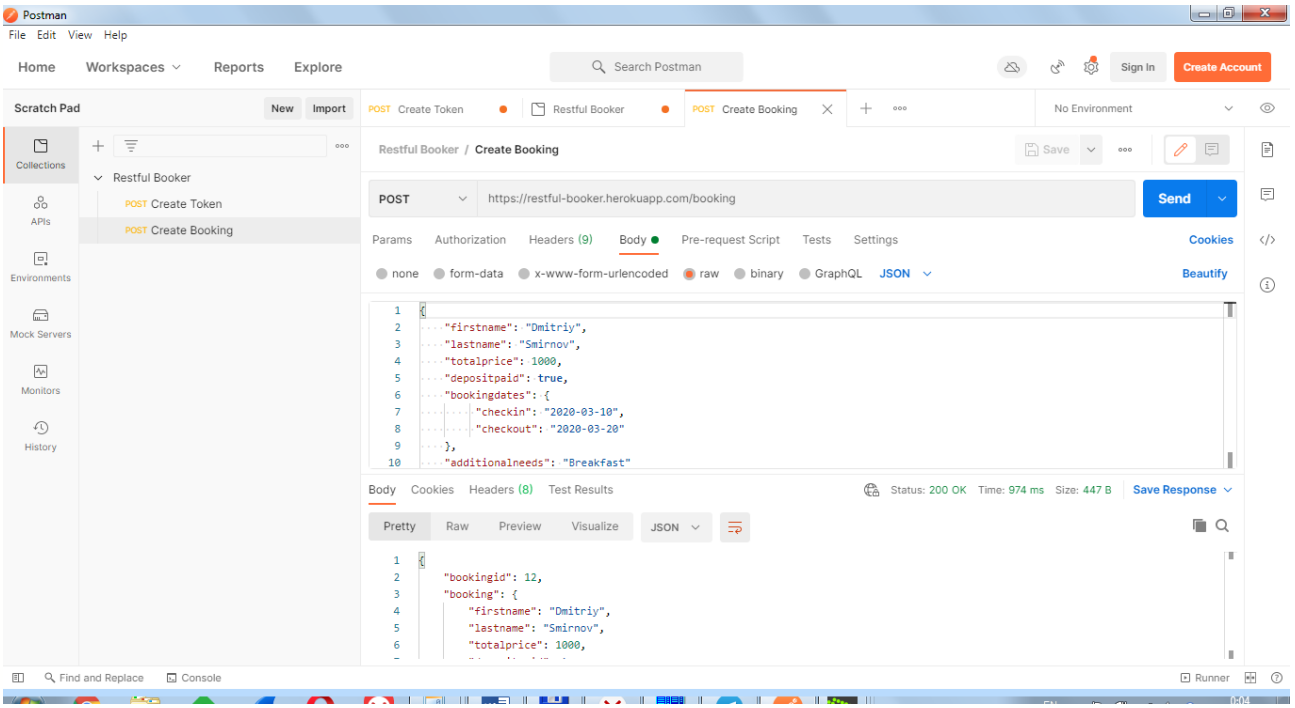
Теперь создадим новую папку. Добавим в неё запросы, напрямую связанные с бронированием: создание, просмотр, редактирование, удаление.

Добавляем новый POST-запрос. В качестве URL задаём https://restful-booker.herokuapp.com/booking, во вкладке Body выбираем Raw, формат — JSON, и в самом Body задаём JSON такой структуры:

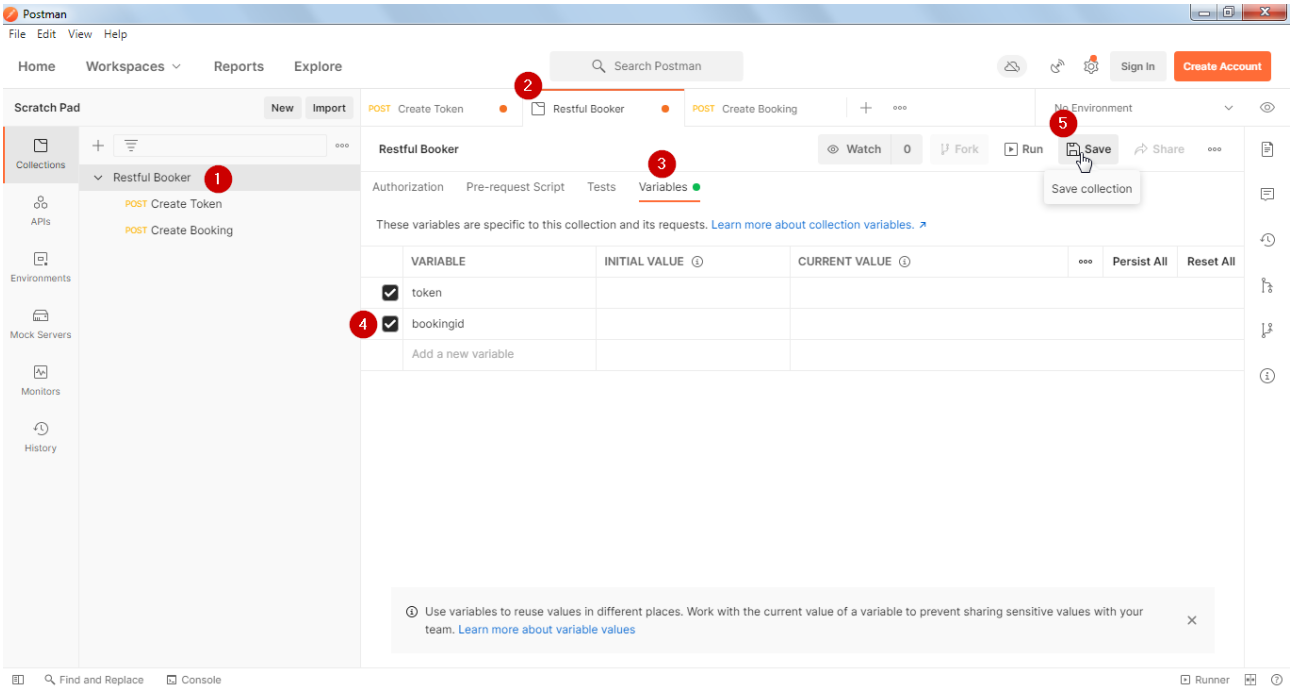
```
{
    "firstname": "Dmitriy",
    "lastname": "Smirnov",
    "totalprice": 1000,
    "depositpaid": true,
    "bookingdates": {
         "checkin": "2020-03-10",
         "checkout": "2020-03-20"
    },
    "additionalneeds": "Breakfast"
}
```

Отправляем запрос и получаем ответ в формате:

```
{
    "bookingid": 24,
    "booking": {
        "firstname": "Dmitriy",
        "lastname": "Smirnov",
        "totalprice": 1000,
        "depositpaid": true,
        "bookingdates": {
            "checkin": "2020-03-10",
            "checkout": "2020-03-20"
        },
        "additionalneeds": "Breakfast"
    }
}
```



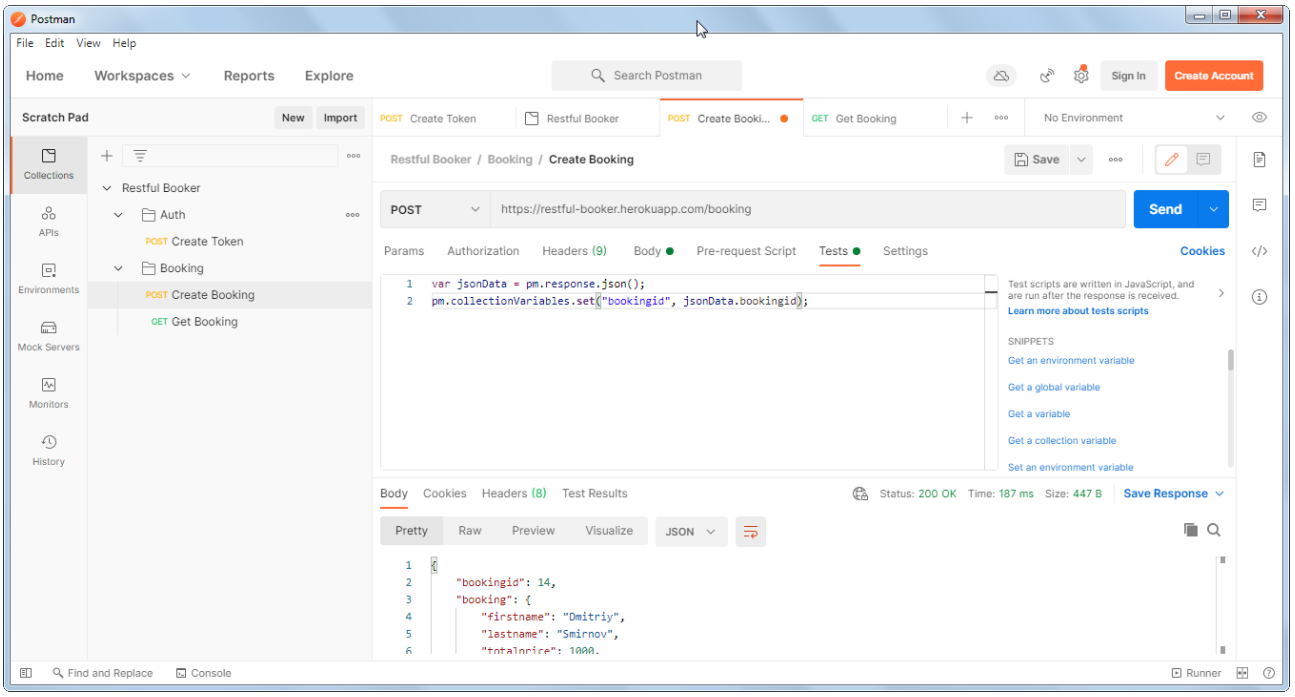
Для дальнейшей работы с данным бронированием надо сохранить значение bookingid. Создаём переменную bookingId аналогично token:



Идём во вкладку Tests запроса, сохраняем значение переменной:

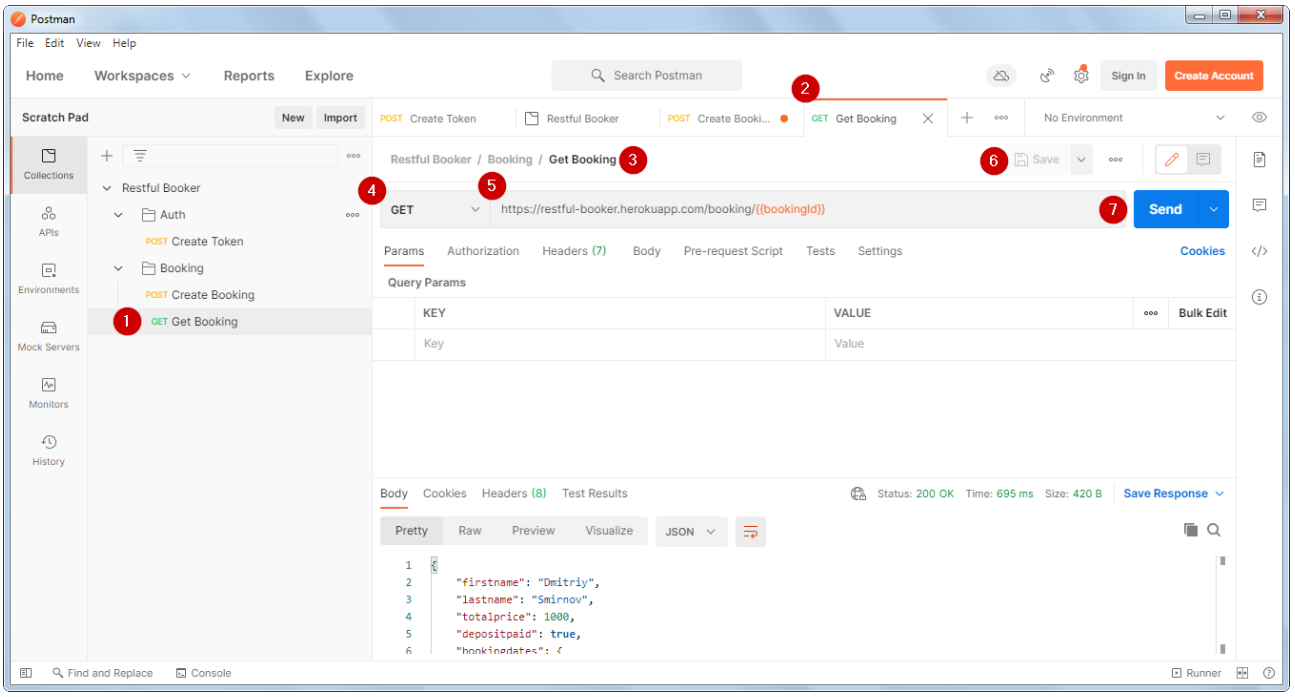
```
var jsonData = pm.response.json();
pm.collectionVariables.set("bookingId", jsonData.bookingid);
```

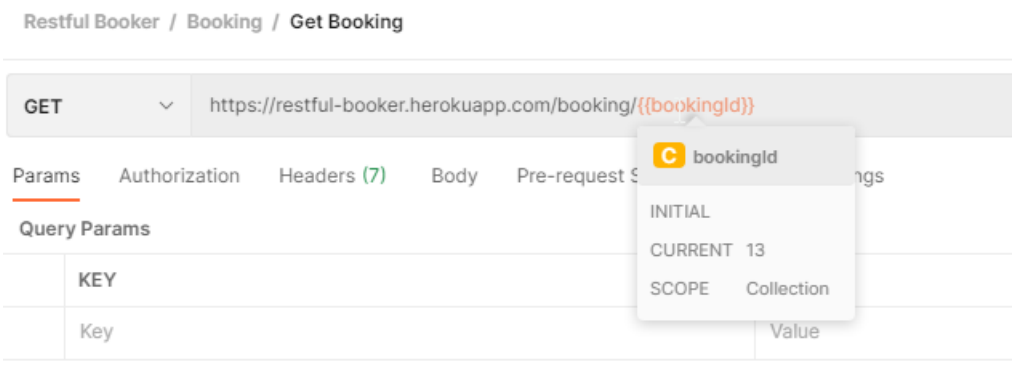
где bookingId — название нашей переменной, а bookingid — имя ключа в JSON-ответе.



Далее получим информацию о только что созданном бронировании. Создадим новый GET-запрос, и в качестве URL укажем https://restful-booker.herokuapp.com/booking/{{bookingId}}.

В этом случае {{bookingId}} — это название нашей переменной. Чтобы Postman знал, что мы хотим подставить в строку значение переменной, её имя заключается в двойные фигурные скобки. После выполнения предыдущего запроса CreateBooking у нас в переменной сохранится идентификатор бронирования. Это значение подставляется при отправке запроса, и мы получим успешный результат:

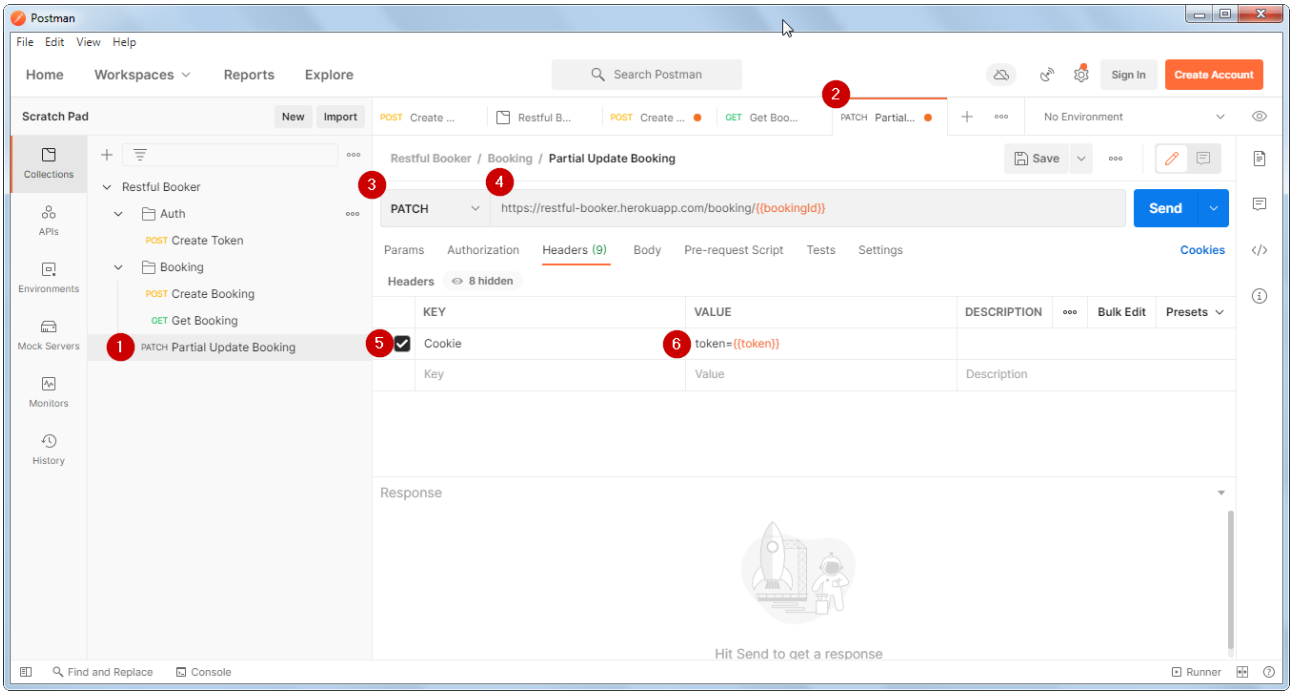




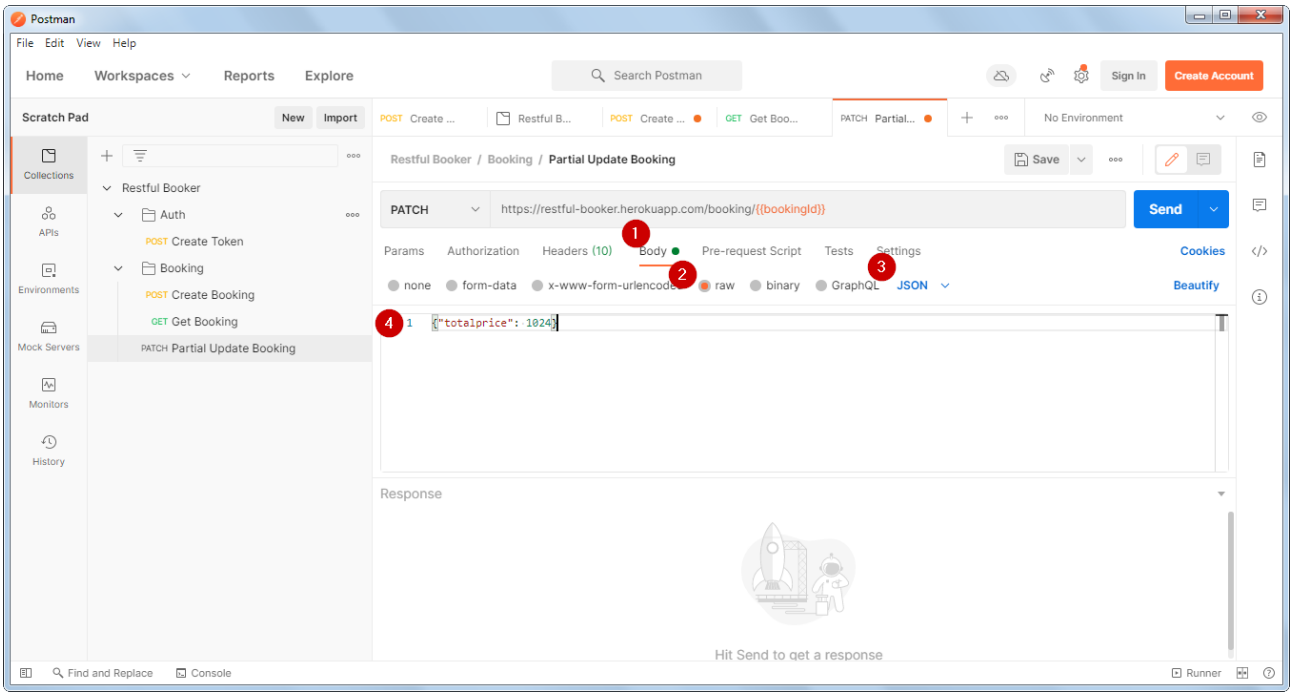
А сейчас добавим ещё один запрос, который требует аутентификации.

Например, PartialUpdateBooking. Это PATCH-запрос, который позволяет отредактировать только одно поле. Создадим требуемый PATCH-запрос в папке Booking. В качестве URL воспользуемся https://restful-booker.herokuapp.com/booking/{{bookingId}} — редактировать тот же экземпляр.

Далее надо задать токен. Согласно документации, он передаётся в HTTP-заголовке Cookie со значением token={{token}}, где {{token}} — наш токен, полученный в результате запроса GetToken. Открываем вкладку Headers, создаём новый заголовок с названием Cookie и требуемой строкой в качестве значения:

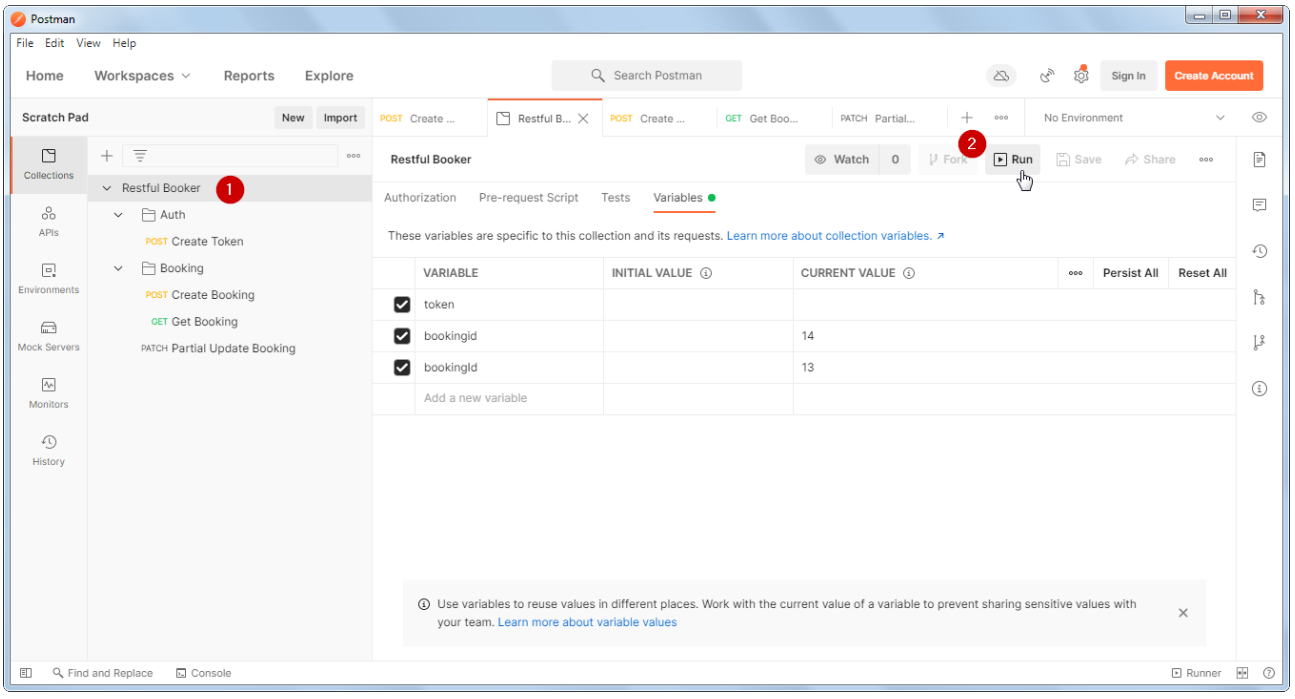


Далее зададим содержимое запроса, которое часто называется payload — «полезная нагрузка»: открываем вкладку Body, устанавливаем формат raw, тип — JSON, и в качестве самого Body укажем простой JSON, например, {"totalprice": 1024}. В этом случае изменится только одно поле, которое хранит общую стоимость бронирования:

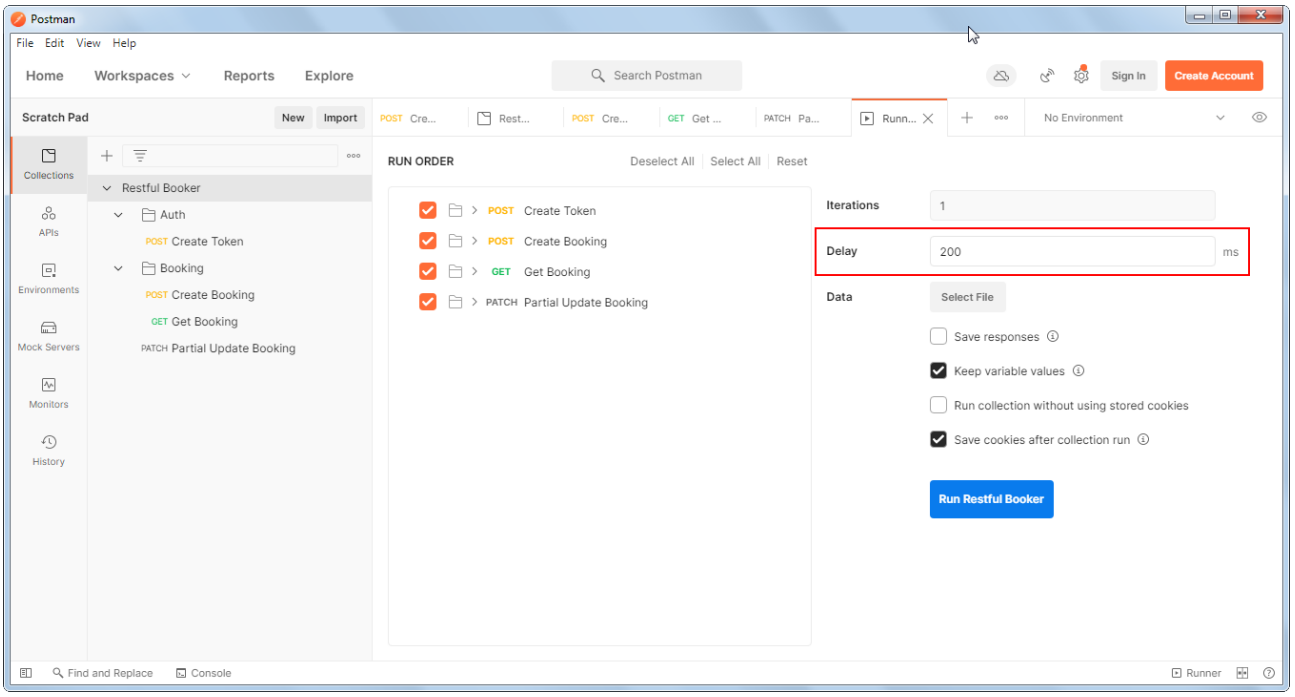


Важно! *Если запрос возвращается с кодом ответа 403 Unauthorized, перезапросите токен. Скорее всего, у него истёк срок действия.*

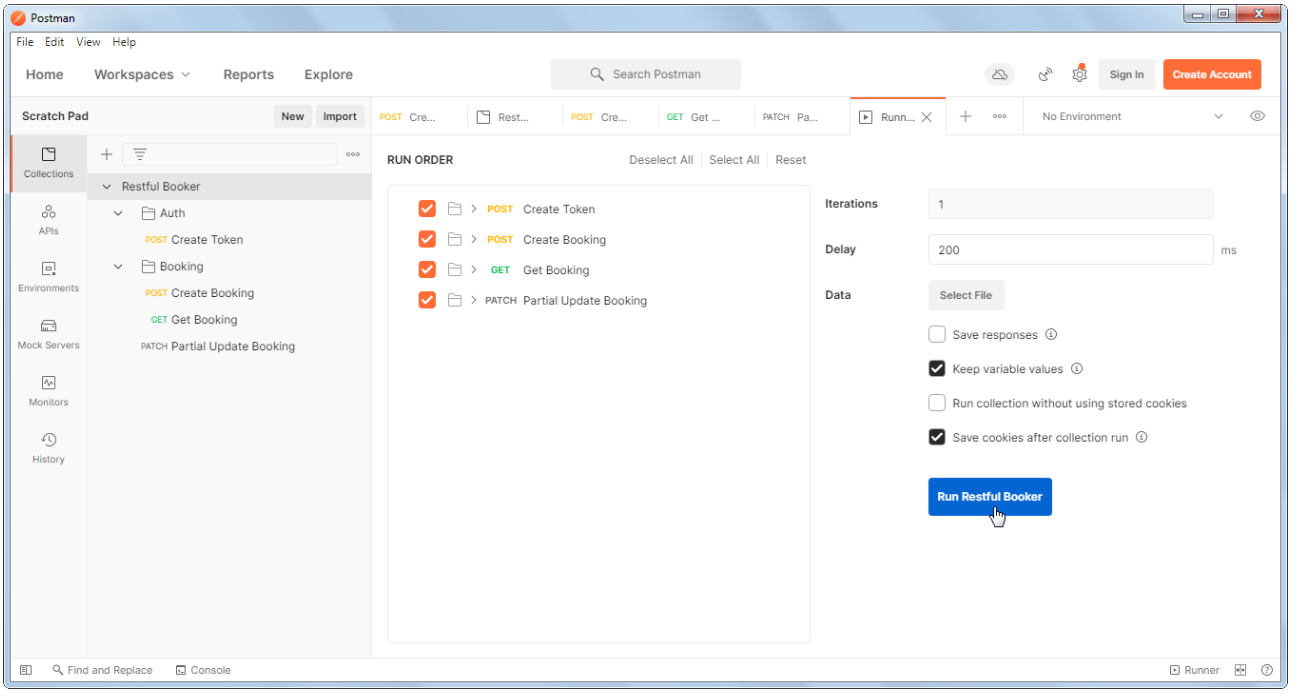
А теперь запустим последовательно все созданные тесты, используя Runner. Выбираем коллекцию и кликаем на кнопку Run:



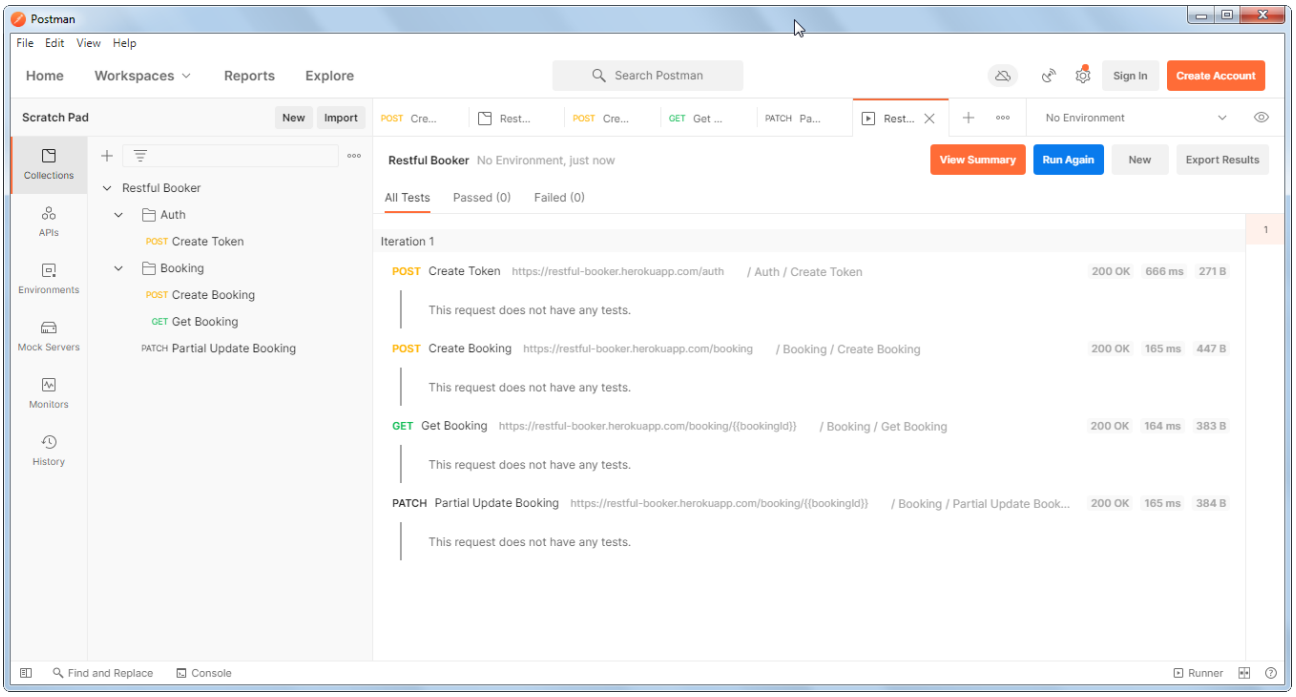
Важно указывать небольшую паузу между запросами. Она используется, чтобы сервер успел обработать запросы, если их окажется много:



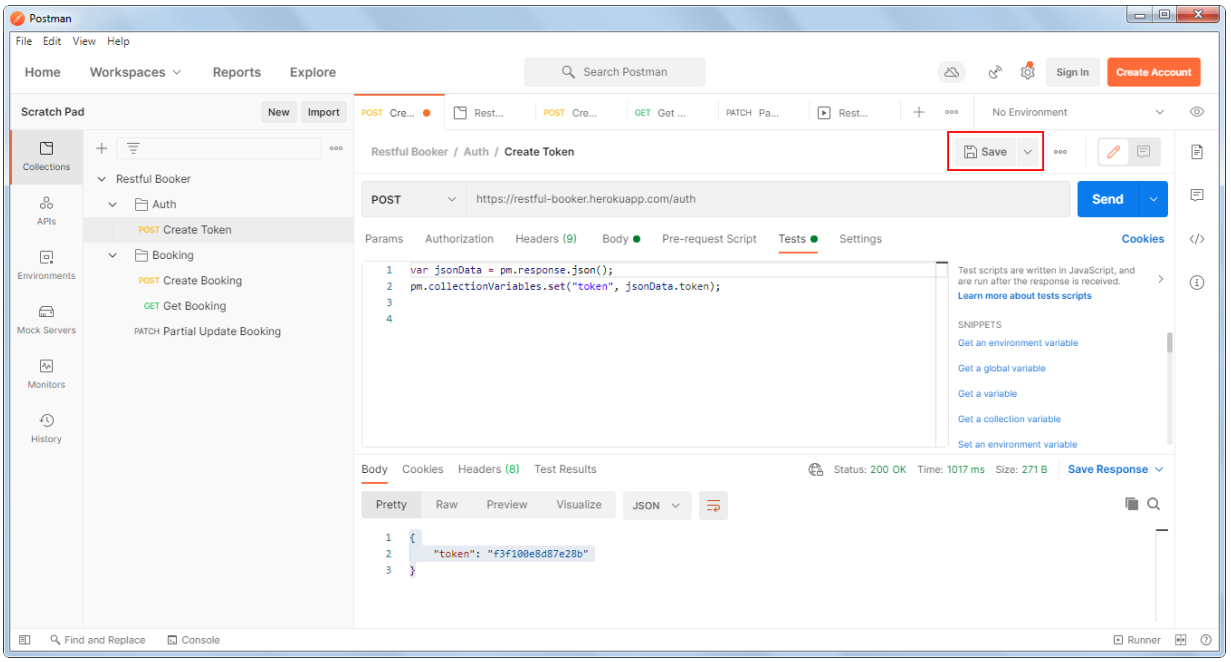
Можно поменять порядок тестов или исключить некоторые из них. Нажимаем кнопку Run:



Всё хорошо, запросы выполнились успешно:

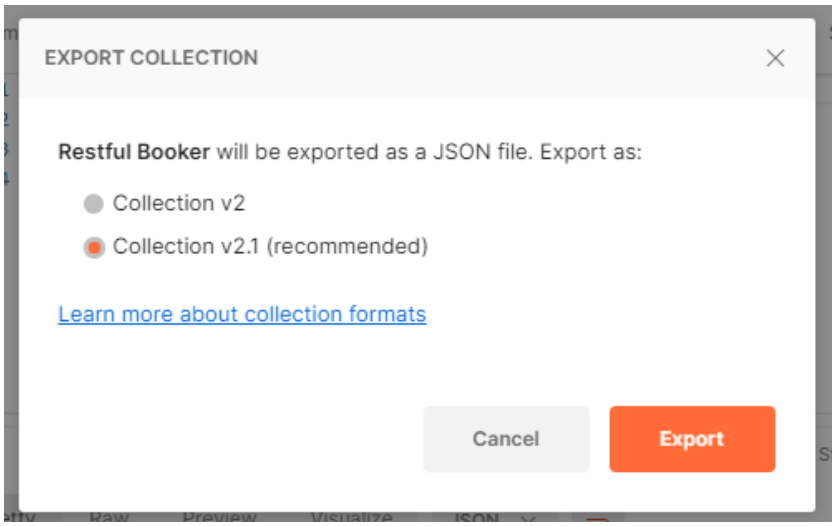


Важно! *Если по каким-то странным причинам запросы не запускаются, пройдитесь по всем запросам в окне редактирования и нажмите кнопку Save:*



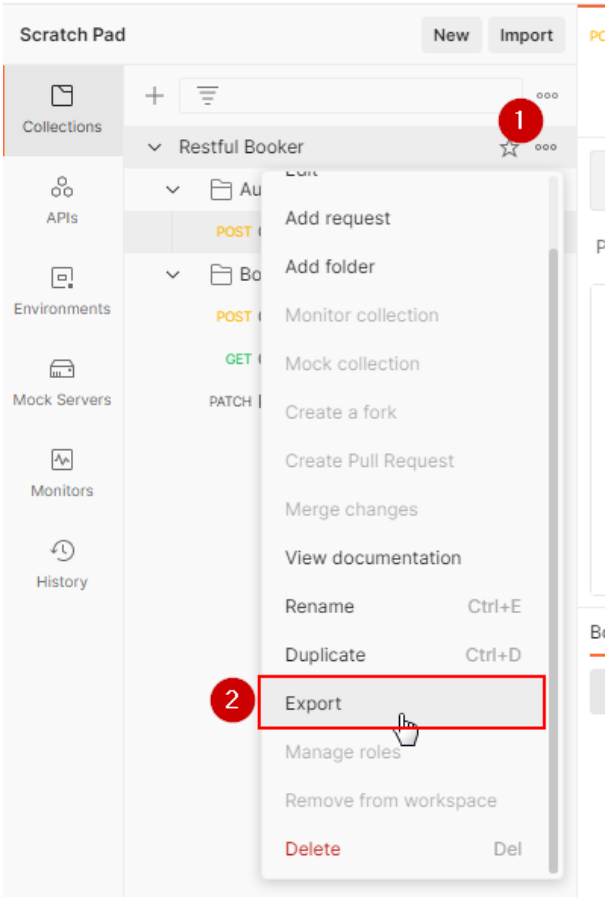
Сохраним коллекцию в файл для переноса на другой компьютер. За сохранение в Postman отвечает функция экспорта.

Для экспорта созданной коллекции выбираем коллекцию, нажимаем на кнопку с многоточием и выбираем Export:



В открывшемся окне снова нажимаем Export и сохраняем файл.

Практическое задание



Задание 1.

Напишите тесты для API **Restful Booker** для следующих функций:

- Auth — CreateToken;
- Booking — GetBooking;
- Booking — CreateBooking;
- Booking — UpdateBooking или Booking — PartialUpdateBooking;
- Booking — DeleteBooking;
- Ping — HealthCheck.

У каждого теста должны быть проверки хотя бы на статус ответа — 200, 201, 404 и т. д.

Важно, чтобы токен и bookingid хранились в переменных Collection.

Запускать тесты надо через Runner.

Формат сдачи: экспортированный файл коллекции Postman и сделать скриншот run'а коллекции. При необходимости:

- экспортированный Environment, если у вас там есть переменные;
- скриншот результатов запуска Runner.

Задание 2. Тест для самопроверки

https://coreapp.ai/app/player/lesson/614a22df8478af554b6f4dc7 (сдавать не нужно)

Глоссарий

REST (Representational State Transfer) — передача состояния представления. Это архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

Используемые источники

1. Статья JSON.
2. Статья «Введение в JSON».
3. Статья «Запись данных в формате JSON».
4. Статья «Работа с JSON».
5. Коды ответа HTTP.
6. Standard ECMA-404.