GeekBrains

Backend-разработка на Go. Модуль 1

HTTP и проектирование RESTful API

На этом уроке

1. Познакомимся с протоколом HTTP
2. Рассмотрим основные принципы построения RESTful API
3. Познакомимся со стандартом OpenAPI и опишем с его помощью API веб-сервера в соответствии с бизнес-задачей

HTTP и HTTPS

На прошлом уроке мы упоминали протокол HTTP в составе TCP/IP стека, теперь пришло время рассмотреть этот сетевой протокол подробнее.

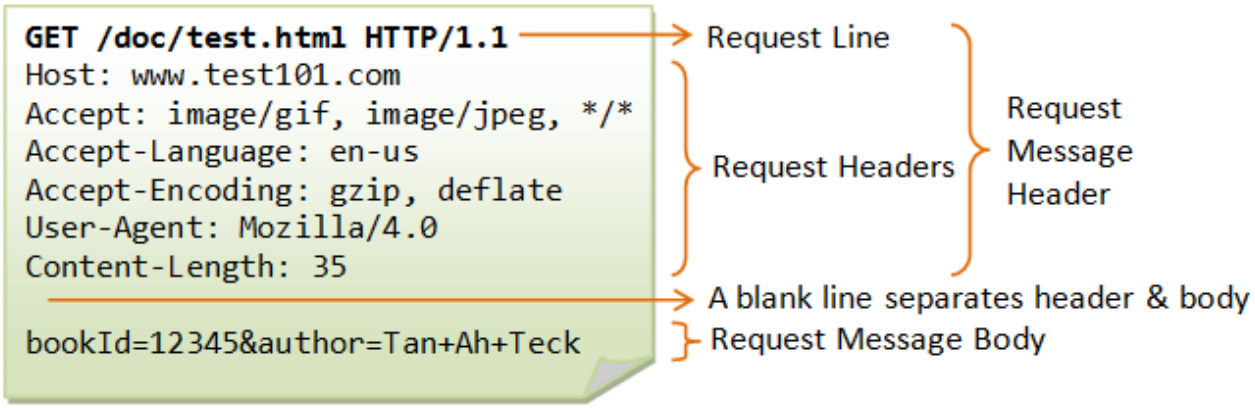
HyperText Transfer Protocol — клиент-серверный протокол, изначально созданный для доставки гипертекстовых документов по сети, однако сейчас он широко используется для передачи абсолютно любых данных в Интернете, будь то картинка, видео или документ.

HTTPS — это расширение протокола HTTP с поддержкой шифрования для повышения безопасности. Данные по HTTPS передаются поверх протокола TLS. Протокол TLS — криптографический и служит для защищенной передачи данных в Интернете.

Так как в основе HTTP лежит клиент-серверная модель, то весь обмен данными происходит исключительно по принципу “запрос-ответ”. При этом HTTP-запрос и HTTP-ответ - это две отдельных сущности, имеющие разные структуры.

HTTP-запрос состоит из:

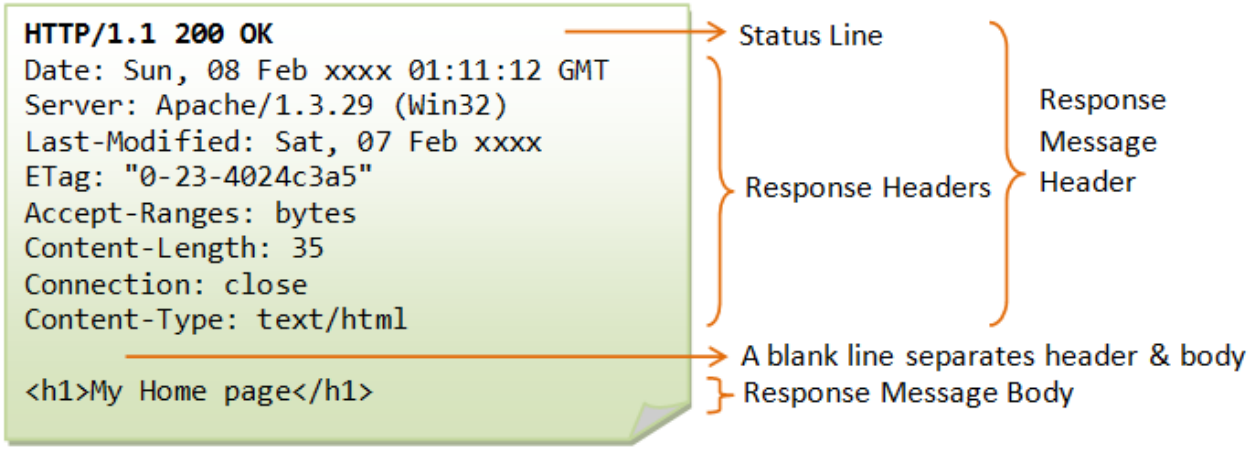
- **HTTP-метода**, который определяет ***действие*** совершаемое запросом (метод GET указывает на то, что мы запрашиваем у сервера какой-то ресурс).
- **URI (Uniform Resource Identifier*)***, который указывает на ***ресурс,*** над которым совершается ***действие*** (например, **/doc/test.html** указывает на конкретную html-страницу на сервере)
- Также, вместе с запросом, в зависимости от используемого HTTP-метода, принято отправлять два типа данных:
    - **Query-параметры** (или query string - строка запроса), которые представляют из себя набор текстовых данных формата “ключ1=значение1&ключ2=значение2”, которые клиент передает в адресной строке, для уточнения/фильтрации ресурса в запросах GET/DELETE.
    - **Тело запроса**, в котором передается непосредственно сам ресурс: html-страница, документ, картинка и тд., тип передаваемых данных может быть указан в соответствующем **HTTP-заголовке** (см. ниже).
- **HTTP-заголовки запроса** содержат различные дополнительные данные для сервера в формате “ключ: значение”, в том числе: информация о браузере клиента, используемом языке, операционной системе, сжатия передаваемых в теле запроса данных, длины данных и т.д.



HTTP-ответ состоит из:

- **HTTP-кода**, целое число из трех десятичных цифр. Клиент узнаёт по коду ответа о результате обработки его запроса и определяет, какие действия ему предпринимать дальше.

- **Тело ответа**, в котором принято передавать запрашиваемый ресурс, который так же, как и передаваемые в запросе данные, может быть документом, картинкой и т.д. В случае, если произошла ошибка, в теле ответа также принято указывать информацию об ошибке.
- **HTTP-заголовки ответа** по аналогии с заголовками запроса, содержат в себе дополнительную информацию для клиента о сервере, а также различные подробности о данных, передаваемых в теле ответа.
  Так как данные, отправляемые клиенту в ответ на запрос представляют из себя лишь набор байт, серверу необходимо явно указывать, в каком именно формате клиенту следует просматривать полученную информацию (например, как картинку, html- или json-документ). Для этого используется заголовок **Content-Type**.



Теперь, давайте подробнее остановимся на составляющих HTTP-запросов и ответов.

HTTP-методы

Как уже было сказано, HTTP-методы указывают на основную операцию над ресурсом, то есть дают команду для сервера. Обычно это короткое слово, написанное заглавными буквами. Основные методы:

1. GET — используется для запроса содержимого указанного ресурса;
2. PUT — применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существует ресурс, то сервер создает его и возвращает статус 201 (Created). Если же ресурс был изменён, сервер возвращает 200 (Ok) или 204 (No Content).
3. POST — применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные (в примере с блогами — текст комментария) включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы на сервер.
4. DELETE — удаляет указанный ресурс.
5. PATCH — аналогично PUT, но применяется только к фрагменту ресурса.
6. HEAD — работает аналогичным GET образом, за исключением того, что в возвращаемом ответе обязательно отсутствует тело. Обычно используется для получения различных метаданных, проверки наличия ресурса на сервере или чтобы узнать, не изменился ли ресурс с последнего обращения. HEAD-запросы обычно посылает браузер, чтобы, например, проверить актуальность своего кэша.

7. OPTIONS — Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответ серверу следует включить заголовок Allow со списком поддерживаемых методов. Также в заголовке ответа может включаться информация о поддерживаемых расширениях.

Кстати, в Go в стандартной библиотеке net/http для удобства работы с методами существуют строковые константы, см. код на гитхабе.

HTTP-коды

При отправке любого запроса на сервер (в том числе когда вы просто открываете страницу сайта в браузере) в любом случае возвращается ответ на ваш запрос в виде кода. Думаю, каждый сталкивался с известной ошибкой 404.

Код состояния HTTP — часть первой строки ответа сервера при запросах по протоколу HTTP. Он представляет собой целое число из трёх десятичных цифр. Первая цифра указывает на класс состояния. За кодом ответа обычно следует отделенная пробелом соответствующая статус-коду

фраза на английском языке, которая разъясняет человеку причину именно такого ответа. И снова в библиотеке net/http мы найдем константы для HTTP-кодов и текстов, см. гитхаб. Примеры:

- 201 Created;
- 401 Unauthorized;
- 507 Insufficient Storage.

Клиент может не знать все коды состояния, но он обязан отреагировать в соответствии с классом кода. Выделено пять классов кодов состояния:

1. **1xx: Information.**

В этот класс выделены коды, информирующие о процессе передачи. Это обычно предварительный ответ, который состоит только из Status Line и опциональных заголовков и завершается пустой строкой. Обязательных заголовков для этого класса кодов состояния нет. Из-за того, что стандарт протокола HTTP/1.0 не определял информационных кодов состояния, серверы **не обязаны** посылать 1xx-ответы HTTP/1.0-клиентам, за исключением экспериментальных условий.

2. **2xx: Success.**

Этот класс кодов состояния указывает, что запрос клиента был успешно получен, понят, и принят. Примеры ответов: 200 OK, 201 Created.

3. **3xx: Redirect.**

Класс кодов состояния показывает, какие действия должен выполнить клиент, чтобы запрос завершился успешно. Служит для переадресации на мобильную версию, на HTTPS, когда клиент подключается по HTTP, или на другое доменное имя, если сайт переехал или доступен по нескольким доменным именам.

4. **4xx: Client Error.**

Класс кодов 4xx предназначен для определения ошибок, которые произошли на стороне клиента. Например, для обозначения ошибок, связанных с валидацией данных, полученных от клиента можно использовать 400 Bad Request. А при обращении клиента к несуществующему ресурсу допустимо возвращать 404 Not Found.

5. **5xx: Server Error.**

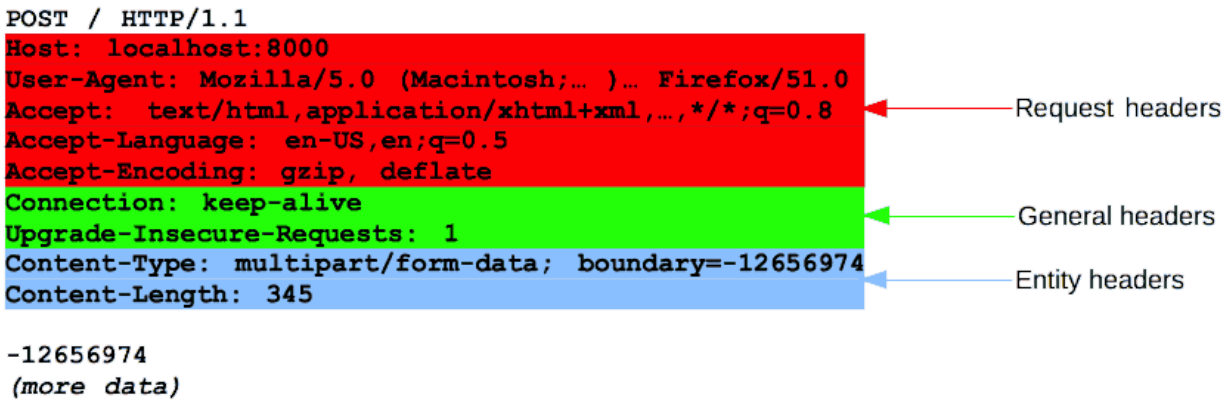
Коды ответов, начинающиеся с 5, указывают на случаи, когда ошибка произошла на стороне сервера. Серверу следует включить в ответ поясняющие сообщение, также в нем не помешает указать характер неполадок: временная или постоянная.
Например, код ответа 500 Internal Server Error принято использовать для обозначения ошибок, которые возникают независимо от передаваемых пользователем данных, например, если база или удаленный ресурс, к которым обращается сервер в процессе обработки запроса, не доступны.

HTTP-заголовки

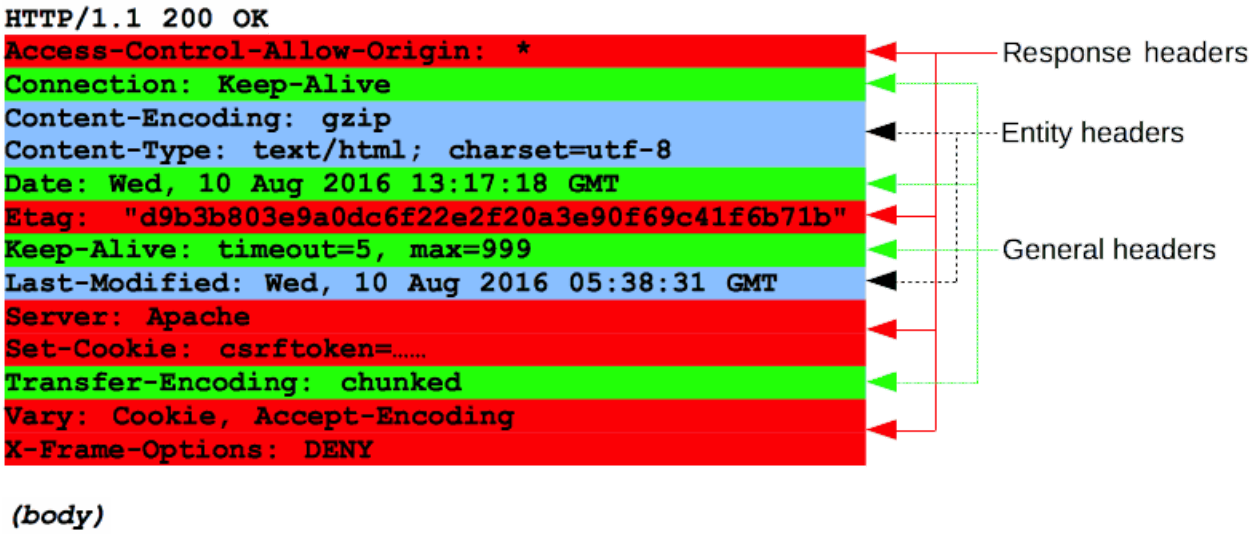
Все заголовки разделяются на четыре основных группы:

- General Headers (общие заголовки) — должны включаться в любое сообщение клиента и сервера. Пример: в случае заголовков запроса, часто передают **User-Agent**, который указывает программное обеспечение клиента и его характеристики.
- Request Headers (заголовки запроса) — используются только в запросах клиента. Пример: **Authorization**, в котором передаются данные для авторизации.
- Response Headers (заголовки ответа) — только для ответов от сервера. Пример: **Age**, который хранит количество секунд с момента модификации ресурса.
- Entity Headers (заголовки сущности) — сопровождают каждый объект сообщения, используются при передаче множественного содержимого (multipart/*). Такие заголовки характеризуют каждый конкретный блок информации: например, ответ сервера содержит как JSON-информацию, так и файл, разбитый на части. Пример: **Allow**, в котором содержится список методов, применимых к запрашиваемому ресурсу.

Пример групп заголовков в запросе:



Пример групп заголовков в ответе:



JSON

Теперь, когда мы разобрались как структурно устроены запрос и ответ, пришло время поговорить о формате, в котором передаются данные. Как было уже упомянуто, через HTTP можно обмениваться данными любого типа, будь то картинка или документ.

Но как быть, если мы хотим передать данные, которые представляют из себя не файл, а информацию об объекте (например, товар в интернет-магазине)? Конечно, мы можем скомпоновать html-страницу со всей необходимой информацией и форматированием на стороне сервера и возвращать ее клиенту “as is”.

Но в случае, если мы говорим о приложении с клиентской частью (фронтендом), нам необходимо передавать данные, сгруппированные по полям, чтобы фронтенд мог сам собрать итоговую страницу, подставляя пришедшие от сервера данные в нужные места на сайте, таким образом, избавляя сервер от необходимости рендерить страницу и передавать ее по сети.

Как раз для такого представления данных и был разработан формат **JSON (JavaScript Object Notation)**, который, несмотря на свое название, используется далеко не только в JavaScript’e - на самом деле, одним из основных преимуществ JSON, наряду с человеко-читаемостью, является поддержка формата всем, что так или иначе взаимодействует с глобальной сетью.

Давайте рассмотрим то, как мы можем представить, Go-структуру в качестве JSON-документа, пригодного для передачи по сети:

```
// Go-структура
type Person struct{
   Name string
   Age int
   Email string
}
```

```
// Инициализированная значениями
p := Person{
   Name: "Ivan",
   Age: 69,
   Email: "ivan@mail.ru",
}

// Эквивалентный JSON-документ
{
   "Name":"Ivan",
   "Age":69,
   "Email":"ivan@mail.ru",
}
```

Преобразование, или ***сериализация***, некоторой структуры данных в JSON, или в любой другой формат, из которого можно получить посредством обратной операции ***десериализации*** исходную структуру, очень часто применяется для передачи данных по сети, так как позволяет нам абстрагироваться от внутреннего устройства получателей, которое нам зачастую неизвестно.

Процессов сериализации и десереализации данных мы подробнее коснемся в следующем уроке, когда перейдем к рассмотрению различных инструменты разработки веб-сервера в Go.

REST

Разобравшись как и в каком виде мы будем обмениваться данными между клиентом и сервером, углубимся в реализацию веб-приложений с архитектурной точки зрения.

Наиболее популярным стилем проектирования архитектуры интерфейсов веб-приложений является так называемый **REST** (**Re**presentational **S**tate **T**ransfer), описанный в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. API, спроектированные согласно принципам REST, принято называть **RESTful API**.

Отличительной особенностью сервисов REST является то, что они позволяют наилучшим образом использовать протокол HTTP, концентрируясь целиком на ресурсах.

Ресурс однозначно идентифицируются с помощью строго определенного формата URI, а взаимодействие с ним посредством HTTP-методов (GET, DELETE и т. д.) осуществляется наиболее “естественным” образом:

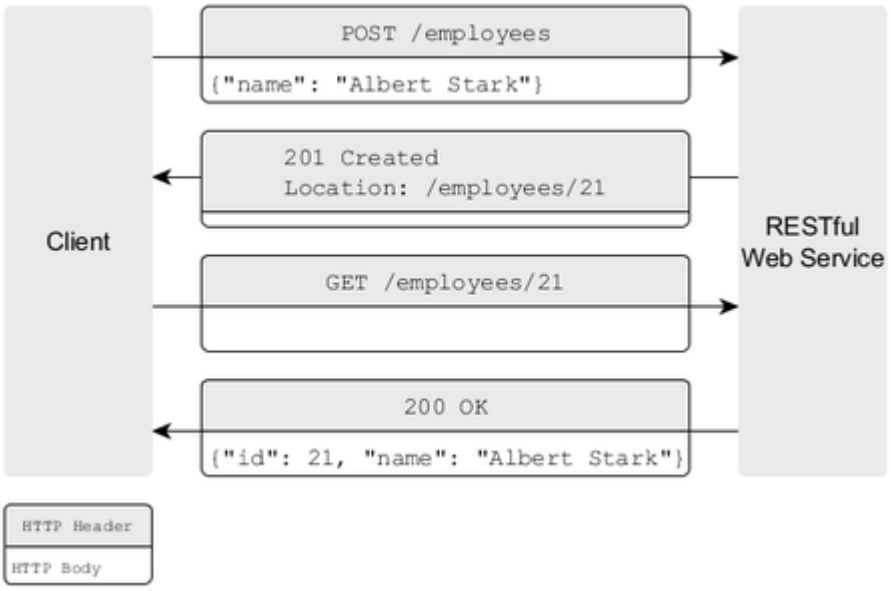
/{entity}/{id}?{query-params}

- **entity** — название сущности, например, модели или таблицы/представления в БД.
  Примеры: user, item

- **id (необязательный параметр)** — идентификатор объекта. Может представлять из себя как целочисленное значение, так и уникальную строку (например, хэш)
- **query-params (необязательный параметр**) — дополнительные параметры выборки для списочных запросов (фильтрация, сортировка, паджинация и пр.). Форматируются по правилам HTTP GET параметров

Комбинацию URI и HTTP-метода принято называть ***HTTP-эндпоинтом (endpoint)***

Примеры эндпоинтов



GET /{entity}/{id} — получить объект по его ID

В случае успеха сервер возвращает **200 OK** с полями объекта в определенном формате (часто им выступает **JSON**) в теле ответа. В случае, если объект с такими id не существует, сервер возвращает **404 Not Found**

GET /{entity}?{param1=...&param2=...} — вывод списка из объектов, отфильтрованных согласно параметрам

В случае успеха сервер возвращает **200 OK** с массивом объектов в формате JSON в теле ответа (т.е. ответ начинается с [и заканчивается]).
Если серверу не удалось найти ни одного объекта, удовлетворяющего параметрам фильтрации, принято возвращать пустой массив и статус **200 OK**, а не **404 Not Found**.

На первый взгляд это может показаться неочевидным, но достаточно вспомнить что обычно происходит в интернет-магазинах при задании фильтров, которым не удовлетворяет ни один товар - клиентская часть обычно отображает ту же самую страницу списка товаров, только без товаров. Ошибка 404 же, зачастую, означает, что клиентская часть должна перенаправить пользователя на отдельную страницу-заглушкой, где и будет отображен соответствующий статус код.

Также, вместе с массивом объектов, сервер может возвращать и различные метаданные, которые пригодятся клиентской части при выполнении последующих запросов: примененные фильтры, информацию о пагинации (номер текущей страницы, сколько всего страниц, количество объектов на странице и тд.)

POST /{entity} — создает новый объект типа {entity}

В случае работы с протоколом JSON в теле запроса принято перечислять поля объекта в виде JSON-документа без дополнительного заворачивания, т.е. **{"field1":"value","field2":10}**

В случае успеха сервер должен возвращать **201 Created** с идентификатором только что созданного объекта. ID может возвращаться в теле ответа вместе с остальными полями, аналогично телу ответа запроса GET /{entity}/{id}. Либо же тело ответа может отсутствовать, при этом в ответе имеется дополнительный заголовок **Location: /{entity}/{new_id}**, указывающим на месторасположение созданного объекта, вместе с его идентификатором.

Единственный неидемпотентный некешируемый метод, т.е. повтор двух одинаковых POST запросов создаст два одинаковых объекта.

PUT /{entity}/{id} — изменяет объект целиком

В запросе должны содержаться все поля изменяемого объекта в формате JSON, так как отсутствующие поля будут перезаписаны пустыми значениями. В случае успеха сервер должен возвращать **204 No Data** с пустым телом, т.к. у клиента есть все необходимые данные для дальнейшей работы с ресурсом.

Идемпотентный запрос, т.е. повторный PUT с таким же телом не приводит к каким-либо изменениям в БД.

PATCH /{entity}/{id} — изменяет часть полей объекта

Работает аналогичным методу PUT образом, однако в случае с PATCH отсутствующие в запросе поля объекта НЕ будут перезаписаны пустыми значениями. Использовать PATCH как основной метод для обновления данных в вашей API может быть полезно для обеспечения “защиты от дурака”, чтобы клиент не мог случайно испортить данные в базе, передав в PUT только те поля, которые он хочет изменить.

Идемпотентный запрос, т.е. повторный PUT с таким же телом не приводит к каким-либо изменениям в БД.

DELETE /{entity}/{id} — удаляет объект, если он существует.

В случае успеха возвращает **204 No Data** с пустым телом, т.к. возвращать уже нечего.

Идемпотентный запрос, т.е. повторный DELETE с таким же адресом не приводит к ошибке 404.

Аутентификация

В отличие от веб-приложений, RESTful API обычно не сохраняют информацию о состоянии.

В связи с этим возникает вопрос, допустимо ли использовать сессии и куки при использовании REST API. Сам формат HTTP работает с понятием сессий и куков. Например, почитать про HTTP Cookies можно здесь.

Тем не менее, при использовании REST часто стараются избегать аутентификации пользователя в сессиях или куках. Одна из распространенных практик гласит, что каждый запрос должен приходить вместе с определенным видом параметров аутентификации.

Так, для аутентификации пользователя с каждым запросом можно отправлять секретный токен доступа. Так как токен доступа может использоваться для уникальной идентификации и аутентификации пользователя, запросы к API всегда должны отсылаться через протокол **HTTPS**, чтобы предотвратить атаки типа «человек посередине» (англ. "man-in-the-middle", **MitM**).

Откуда клиент будет получать этот токен доступа? Его может генерировать как ваш собственный сервис авторизаций (пример), так и third-party решение, использующее, например, SAML SSO (пример) или протокол OAuth (пример).

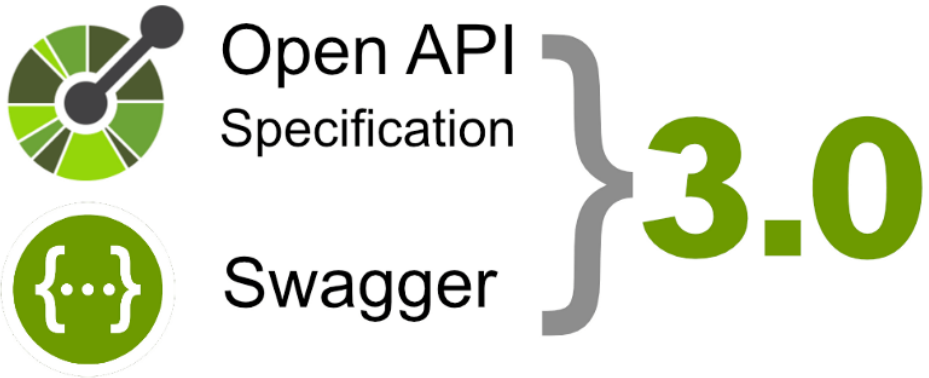
OpenAPI

Предположим, что мы спроектировали наше RESTful API, определились, какие эндпоинты доступны для клиентов, какие заголовки им следует отправлять (аутентификация, user-agent и тд.) вместе с запросами и какие ресурсы, статусы и заголовки им будут возвращаться.

Отлично, теперь неплохо было бы изложить все эти соглашения по взаимодействию клиентов с нашим сервером в виде документации. Конечно, мы можем написать документ в Word, PDF, может даже сделать презентацию в PowerPoint с кучей анимаций и слайдов.

Но, у произвольного текстового описания есть существенный минус: оно может неоднозначно интерпретироваться людьми из разных подразделений и команд, поэтому потребовалась единая система понятий и терминов. По мере роста количества разработчиков стала необходимой и формализация описаний взаимодействий.

Также неплохо было бы использовать документацию и для прикладных нужд, например для упрощения написания автоматизированных тестов или даже для генерации кода сервера и клиента, взаимодействующих посредством RESTful API, описываемого в этой документации.



Для этих нужд была разработана спецификация **OpenAPI (Swagger до версии 3.0)** - формализованный стандарт и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API.

Например, эндпоинты создания и получения ресурса по ID, которые в вольной форме описали выше, в OpenAPI спецификации будут выглядеть следующим образом (для демонстрации возьмем объект User):

```
openapi: 3.0.1 <- версия OpenAPI спецификации
info:
 title: Пример спецификации
 version: 1.0.0 <- версия описываемого API
Paths:
 /user: <- URI сущности
   post: <- используемый HTTP-метод
     summary: Создает новый объект типа User
     operationId: CreateUser
     requestBody: <- ожидаемое сервером содержимое тела запроса
       description: Структура объекта User, которую следует передать в теле запроса
       Content: <- непосредственно сами данные (в данном случае, объект User)
         application/json: <- формат данных, здесь мы используем JSON
           Schema: <- ссылка на схему с полями сущности User в разделе Components
             $ref: '#/components/schemas/User'
       required: true <- необходимо ли передавать тело в данном эндпоинте
     Responses: <- возможные статусы ответа с описанием ошибок
       405:
         description: Invalid input
         content: {} <- тело ответа
     x-codegen-request-body-name: body
 /user/{userId}: <- URI с шаблоном ID сущности User
   get:
     summary: Получить объект User по ID
     operationId: GetUser
     Parameters: <- список параметров запроса
     - name: userId <- имя параметра
       in: path <- указание на то, что параметр следует искать в URI
       description: ID искомого User
       required: true <- необходимо ли передавать данный параметр
       Schema: <- тип параметра в данном случае описан здесь, но можно также, как в
случае с телом, указать референс на Component
         type: integer
         format: int64
     responses:
       200: <- мы также можем описать ответ при успешном выполнении запроса
         description: successful operation
         Content: <- описание тела ответа по аналогии с телом запроса
           application/json:
             schema:
```

```
              $ref: '#/components/schemas/User'
      400:
        description: Передан невалидный ID
        content: {}
      404:
        description: User не найден
        content: {}
Components: <- здесь мы описываем структуры сущностей, на которые ссылаемся в описании
эндпоинтов
 schemas:
   User:
     type: object <- тип object означает, что мы описываем структуру
     properties:
       Id: <- не забываем, что согласно REST, у каждой сущности (ресурса) обязательно
должен быть ID
         type: integer
         format: int64
       username:
         type: string
       email:
         type: string
       password:
         type: string
```

Таким образом, мы получаем описание для будущего API, которое можно использовать как:

- Человеко-читаемую документацию, которая при этом одинаково полезна как разработчикам клиентской части, так и серверной, позволяя тем самым лучше организовать командную работу
- Источник данных для Swagger Codegen который позволяет генерировать из спецификации кодовую основу для наших серверов и клиентов - нам остается лишь написать реализацию логики
- Источник данных для Swagger UI и Swagger Editor для интерактивной визуализации API

Именно на примере написания OpenAPI-спецификации мы и научимся проектировать API веб-приложений, отталкиваясь от некоторой бизнес-задачи.

Проектируем интернет-магазин

Итак, предположим, что к нам поступила задача - спроектировать API для интернет-магазина, в котором администраторы смогут логиниться через специальную форму и добавлять/удалять товары из каталога, а пользователи смогут фильтровать товары по цене и категории, а также создавать заказы на несколько товаров.

Обычно, реализацию бизнес-задачи можно расписать в несколько этапов:

Декомпозиция

Мы собрали список требований к итоговому продукту/версии продукта, и теперь формализуем их с технической точки зрения и разбиваем на более конкретные задачи, на выходе получая то, с чем сможет работать техническая команда:

1. Добавить объект User и реализовать простую систему аутентификации и авторизации для юзеров:
    a. Добавить эндпоинт **POST:/user/login**, который принимает логин+пароль юзера и возвращает токен авторизации
    b. Добавить авторизацию во все методы, которые требуют особого доступа (редактирование каталога товаров, добавление новых товаров на сайт)
2. Добавить объект Item (товар) с методами:
    a. Добавление нового товара **POST:/item** (требует авторизации)
    b. Редактирование товара по ID **PUT:/item/{id}** (требует авторизации)
    c. Удаление товара по ID **DELETE:/item/{id}** (требует авторизации)
    d. Получение товара по ID **GET:/item/{id}**
    e. Получение списка товаров с фильтрацией по цене **GET:/item?price_min=1&price_max=15**
3. Добавить объект Order (заказ) с методами:
    a. Добавление нового заказа **POST:/order**
    b. Редактирование заказа по ID **PUT:/order/{id}** (требует авторизации)
    c. Получение заказа по ID **GET:/order/{id}** (требует авторизации)
    d. Получение списка заказов с сортировкой по дате **GET:/order** (требует авторизации)

Проектирование API

Теперь, пришло время проектировать и расписывать конкретные HTTP-эндпоинты, которые будут описывать модели взаимодействия с сущностями, которые мы определили на прошлом шаге.

Для этого создадим новый yaml-файл под спецификацию локально или же воспользуемся Swagger Editor

Начнем опять же с сущности User, добавив описание ее структуры в раздел **Components** OpenAPI-спецификации

```yaml
openapi: 3.0.1
info:
 title: Shop
 version: 1.0.0
components:
 schemas:
   User:
     type: object
     properties:
       id:
         type: integer
```

```
        format: int64
      email:o
        type: string
      password:
        type: string
```

Затем, добавим эндпоинты для логина и логаута, эндпоинт для логина будет возвращать непосредственно токен, с которым администратор будет ходить в методы с ограниченным доступом:

```
paths:
 /user/login:
   post:
     operationId: loginUser
     parameters:
     - name: username
       in: query
       description: The user name for login
       required: true
       schema:
         type: string
     - name: password
       in: query
       description: The password for login in clear text
       required: true
       schema:
         type: string
     responses:
       200:
         description: successful operation
         headers:
           X-Expires-After:
             description: date in UTC when token expires
             schema:
               type: string
               format: date-time
         content:
           application/json:
             schema:
               type: string
       400:
         description: Invalid username/password supplied
         content: {}
```

```
/user/logout:
  post:
    summary: Logs out current logged in user session
    operationId: logoutUser
    responses:
      200:
        description: successful logout
        content: {}
```

Нам нужно также добавить **api_key** (токен) в раздел **securityDefinitions**, чтобы потом добавлять его в эндпоинты, требующие ограничения доступа:

```
securityDefinitions:
 api_key:
   type: "apiKey"
   name: "api_key"
   in: "header"
```

Теперь опишем структуру товара (Item), так же, как описали User:

```
   Item:
     type: object
     properties:
       id:
         type: integer
         format: int64
       name:
         type: string
       description:
         type: string
       price:
         type: integer // чтобы не потерять точность, денежные поля мы будем передавать
         format: int64 // в виде целочисленного значения минимальной размерности валюты
(например, центы или копейки)
       image_link:
         type: string
```

Далее, опишем методы взаимодействия с объектом Item, часть из которых будет требовать передачи api_key, который мы также объявили раннее. Начнем с операции добавления нового товара:

```
/items:
  post:
    summary: "Add a new item to the store"
    operationId: "CreateItem"
    consumes:
    - "application/json"
    produces:
    - "application/json"
    parameters:
    - in: "body"
      name: "body"
      description: "Item object that needs to be added to the store"
      required: true
      schema:
        $ref: "#/definitions/Item"
    responses:
      "405":
        description: "Invalid input"
    security:
    - api_key: []
```

Обратите внимание, что прежде чем добавить новый товар, нам необходимо будет сначала загрузить для него картинку на наш сервер, а после передать ссылку на нее в поле “image_link” объекта Item.

Поэтому, нам понадобится отдельный POST-эндпоинт для загрузки файлов, в который также целесообразно будет добавить необходимость авторизовываться через передачу api_key:

```
/items/upload_image:
  post:
    summary: "uploads an image"
    operationId: "uploadFile"
    consumes:
    - "multipart/form-data"
    produces:
    - "application/json"
    parameters:
    - name: "additionalMetadata"
      in: "formData"
      description: "Additional data to pass to server"
      required: true
      type: "string"
    - name: "file"
```

```
      in: "formData"
      description: "file to upload"
      required: true
      type: "file"
    responses:
    security:
    - api_key: []
```

Опишем эндпоинты, работающие с конкретным товаром по itemID, а именно GET, PUT и DELETE. Не забываем добавить api_key для PUT и DELETE:

```
/items/{itemId}:
  get:
    summary: Find item by ID
    operationId: GetItem
    parameters:
    - name: itemId
      in: path
      description: ID of item to return
      required: true
      schema:
        type: integer
        format: int64
    responses:
      200:
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Item'
      400:
        description: Invalid ID supplied
        content: {}
  put:
    summary: Updates a item in the store with form data
    operationId: UpdateItem
    parameters:
    - name: itemId
      in: path
      description: ID of item that needs to be updated
      required: true
      schema:
```

```
        type: integer
        format: int64
    requestBody:
      content:
        application/json:
          schema:
            properties:
              name:
                type: string
                description: Updated name of the item
              status:
                type: string
                description: Updated status of the item
    responses:
      405:
        description: Invalid input
        content: {}
    security:
    - api_key: []
  delete:
    summary: Deletes a item
    operationId: DeleteItem
    parameters:
    - name: itemId
      in: path
      description: Item id to delete
      required: true
      schema:
        type: integer
        format: int64
    responses:
      400:
        description: Invalid ID supplied
        content: {}
      404:
        description: Item not found
        content: {}
    security:
    - api_key: []
```

И, наконец, опишем эндпоинт для получения списка товаров. Для этого нам нужно будет так же описать query-параметр, согласно которым, сервер будет фильтровать возвращаемый список товаров. В данном случае, авторизацию добавлять не нужно.

Тело ответа будет представлять из себя, в свою очередь, массив из товаров:

```
/items:
  get:
    summary: Lists Items with filters
    operationId: ListItems
    parameters:
    - name: price_min
      in: query
      description: Lower price limit
      required: false
      schema:
       type: integer
       format: int64
    - name: price_max
      in: query
      description: Upper price limit
      required: false
      schema:
       type: integer
       format: int64
    responses:
      200:
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Item'
      400:
        description: Invalid price range
        content: {}
```

Практическое задание

1. Добавить параметры для фильтрации товаров по диапазону цены
2. Добавить в спецификацию объект Order (заказ), подумать, какие поля у нее должны быть и какие эндпоинты потребуется описать
3. *Спроектировать REST API для упрощенной версии Twitter

Дополнительные материалы

1. Инициатива OpenAPI: https://www.openapis.org/
2. Документация по Swagger: https://swagger.io/docs/specification/about/
3. Swagger Editor: https://editor.swagger.io
4. Простым языком об HTTP: https://habr.com/ru/post/215117/
5. Перевод статьи REST API Best Practices: https://habr.com/ru/post/351890
6. Пагинация, сортировка в REST API: https://www.moesif.com/blog/technical/api-design/REST-API-Design-Filtering-Sorting-and-Pagination/
7. HTTP методы в RFC: https://tools.ietf.org/html/rfc2616#section-9
8. JSON-schema - формат описания JSON-ответов: https://json-schema.org/