REST-API

Rest - архитектурный стиль взаимодействия распределенных систем. Будь то взаимодействие Frontend и backend приложения, взаимодействие двух и более микросервисов на бэке или совершенно разных систем, которые имеют некую интеграцию.

Давайте рассмотрим на классическом примере Frontend и Backend приложений, одно из которых будет в качестве клиента, а второе поставщиком и обработчиком данных. Данные в терминологии REST называются **ресурсы**.

Что нужно для их успешного взаимодействия.

- Адрес, куда мы будем доставлять или откуда будем забирать данные. Представим, что вы зашли в библиотеку, в которой сидит библиотекарь. В данном случае вы являетесь клиентским приложением, а библиотекарь - серверным, имеющем доступ к ресурсам. Чтобы узнать, какие книги интересующего вас автора есть в библиотеке, вам нужно знать где находится сам библиотекарь и назвать ему автора - то есть вы готовы сформировать запрос по конкретному адресу.

  Этот адрес должен быть уникальным и гарантировать, что именно по нему мы получим те ресурсы, которые нам нужны, в случае с RESTApi - уникальный адрес ресурса называется **URI**. Важно не путать его с **URL**, который в свою очередь указывает на местоположение ресурса в сети, его также называют **endpoint**.
  Например у нас есть адрес, по которому мы получаем список всех книг:

```
https://librarian.com/books
```

Это явный пример URL.
И тот же адрес, но с фильтрацией книг по автору

```
https://librarian.com/books?author=Gogol
```

Это явный пример URI.

- Нам нужен способ получения ресурсов. В случае с RestApi это протокол HTTP с набором его основных методов GET, POST, PUT, DELETE, PATCH. Вернемся к библиотекарю. Чтобы получить информацию о списке книг конкретного автора, мы подойдем к библиотекарю и назовем ему интересующего автора (сделаем запрос по URI). Этот процесс крайне схож с работой HTTP запроса с методом GET. Метод GET - это ваша просьба “Расскажи” , с помощью которой клиентское приложение запрашивает ресурсы у серверного.
  Если вам нужно украсть все книжки гоголя, вы по тому же URI произведете другой запрос, “принеси, я заберу и не верну”. Это аналогично работе метода DELETE.
  То есть вашу речь можно сравнить с работой протокола HTTP, а ваши просьбы сделать что-то с методами. К методам мы вернемся чуть позже.
- Нам нужна обратная связь от серверного приложения, позволяющая понять состояние выполнения запроса. Для этого в RestApi используют HTTP статус-коды. Каждый статус-код имеет свое определенное числовое значение и смысл и позволяет клиентскому приложению понять, что произошло с запросом. Например, при обращении к несуществующему автору библиотекарь скажет нам, что такого автора нет. Это аналогично HTTP response со статус кодом 404_NOT_FOUND. На основе статус кода, клиентское приложение, выполнит определенные действия для обработки этого статус-кода. О статус кодах мы также поговорим, когда рассмотрим проектирование REST API.
- Как передаются данные.
  - В запросе к серверу мы используем:

        - GET-параметры для формирования URI
        - Тело запроса, чаще всего в виде JSON для отправки данных на сервер. Нам важен формат данных, который сможет понять любая из систем. Один из таких форматов JSON. Процесс перевода данных из формата, понятного языку серверного приложения в формат, понятный всем клиентам называется сериализацией данных. Обратный процесс - десериализацией.
        - заголовки и куки для авторизации
    - В ответе сервер нам может отдать:
        - ресурсы, формат которых определяется в заголовке content-type, как правило это application/json
        - status code ответа
        - Заголовки и куки, которые устанавливает сервер

Посмотрим на пример такого запроса

Важно помнить, что сервер не хранит информацию о предыдущих запросах. Это такой библиотекарь с деменцией, который забывает, что вы у него просили минуту назад. Именно поэтому Rest называют StateLess, то есть не имеющий состояния.s

Важный момент, характерный для REST API. Сколько бы слоев у нас не было со стороны backend приложения, то есть различных интеграций, обращений во внешние системы и прочее, клиент всегда обращается именно к внешнему слою и стучит именно в него, при этом, ему абсолютно все равно, что происходит под капотом, лишь бы вернулись необходимые ресурсы.

Кроме того, REST API поддерживает кэширование, которое играет важную роль для повышения производительности, снижения нагрузки на сервер и сокращения задержек передачи данных. Кэширование позволяет сохранять ранее полученные данные и использовать их повторно, вместо повторной отправки запроса на сервер. Сервер может добавить заголовки HTTP-ответа, такие как Cache-Control, Expires и ETag, чтобы управлять кэшированием на стороне клиента. Использование этих заголовков позволяет указывать клиентам, как долго данные могут быть закешированы и как проверять актуальность закэшированных данных. Клиентские приложения могут

использовать локальное хранилище, такое как LocalStorage или IndexedDB, для сохранения закэшированных данных на стороне клиента. Когда клиент делает запрос, приложение сначала проверяет, есть ли данные в локальном хранилище, и, если они присутствуют, использует их вместо отправки запроса на сервер.

Резюмируя.

Фронтенду нужны данные для отображения - ресурсы. Он делает HTTP-запрос на бэкенд по определенному URI. Бэкенд готовит и сериализует данные и возвращает клиенту HTTP-response с самими ресурсами и статус-кодом, на основе которого фронтенд отображает данные или, например, выводит сообщение об ошибке.

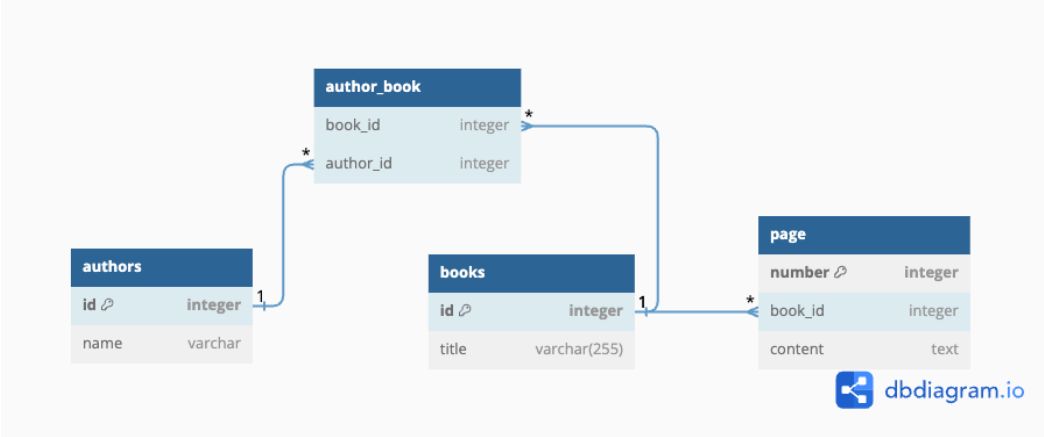
Давайте перейдем к **проектированию.** Сначала рассмотрим процессную составляющую проектирования API, а затем техническую.

- Проектирование всегда начинается после составленного бизнес-флоу и дизайн проекта. Заранее нам нужно знать, как клиентская и серверная сторона будут обрабатывать ресурсы. То есть в нашем дизайне есть какой-то экран, на котором выводится список книг, соответсвенно, на бэке нам нужен REST метод, который эти книги нам отдаст, а клиент отобразит в браузере.
- Затем происходит процесс написания спецификации API методов, то есть подробно описывается каждый EndPoint и входные/выходные параметры каждого метода.

  Кто этим занимается?
  Если в команде есть системный аналитик, то это непосредствтенно его задача. Аналитик проектирует методы, фиксирует их в документации проекта и согласует вместе с представителями разработки фронта и бэка, чтобы не упустить важные моменты, сделать метод наиболее гибким, расширяемым и легко реализуемым с обеих сторон.

> Если в команде нет системного анализа, то эта задача ложится либо на тимлидов разработки, либо непосредственно на разработчиков. Ключевой момент - обе стороны разработки (клиентская и серверная) должны быть удовлетворены первоначальными требованиями.

Теперь давайте подробно про принципы проектирования.



Структура URL для доступа к ресурсам.

Вернемся к нашим книгам. Представим, что в нашей реляционной БД есть следующая структура. (даю комментарии по стурктуре)
У нас есть отдельно взятые таблицы, таблицы, имеющие связи one to many, many to many, то есть наиболее частые кейсы, которые встречаются при повседневной разработке.

Для начала необходимо дать возможность версионирования наших API. Для чего? Мы сделали эндпоинт на скорою руку, отдали в прод для интеграции. Спустя время, появилась потребность рефакторинга этого метода, и мы, вместо того, чтобы править старый метод, создаем его новую версию, доступную по новому эндпоинту.
Все наши методы будут начинаться с префикса их версии /v1 или /v2

```
https://librarian.com/api/v1/books
```

```
https://librarian.com/api/v2/books
```

Начнем с endpoint для книг.

Первое правило именования URL для доступа к ресурсам -
1 эндпоинт - 1 сущность, которая отражена в пути во множественном числе.

То есть все операции с книгами мы будем осуществлять по URL:

```
https://librarian.com/api/v1/books
```

Видим, в ентпоинте отражено название ресурса во множественном числе, который мы будем обрабатывать.
Какие действия мы можем совершить с книгами отдельно, не привязываясь к другим ресурсам?

1. Получить список всех книг, для этого мы пойдем в наш endpoint с методом GET.
2. Получить детальную информацию по 1 книге
3. Изменить название 1 книги
4. удалить 1 книгу или удалить список книг (что не очень рекомендуется делать в таком случае)

Давайте спроектируем каждый из этих методов.

При проектировании учитываем 3 основные составляющие.

- Входные GET-параметры, заголовки и тело запроса
- Способ авторизации
- content-type

- Тело ответа, при всех возможных статус кодах

$
г

```
https://librarian.com/api/v1/books
```

Входные параметры

Parameter	type	In	Description
Authorization	Bearer-Token	Headers	Токен для авторизации

Ответ:

STATUS_CODE	Content-Type	Пример ответа

200 (OK)	application/json	`[`
		`{`
		`"id": 1,`
		`"title": "Example Book2"`
		`},`
		`{`
		`"id": 2,`
		`"title": "Example Book2"`
		`}`
		`]`

403 (Forbidden)	application/json	`{`
		`"detail": "unauthorized"`
		`}`

- Статус код: 200 (OK) - говорит нам о том, что сервер успешно обработал запрос и вернул необходимые нам ресурсы
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
[
   {
      "id": 1,
```

```
    "title": "Example Book 1"
  },
  {
    "id": 2,
    "title": "Example Book 2"
  }
]
```

- Статус код: 403 (Forbidden) - говорит нам об ошибке авторизации, то есть сервер нас не распознал и вернул 403 статус-код, чуть позже рассмотрим отедельно наиболее частые статус-коды
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
  "detail": "unauthorized"
}
```

То есть сейчас мы написали описание к методу, зафиксировали его входные параметры, URL и возможные варианты ответа, к которым должен быть готов клиент.

Давайте по такому же пути спроектируем остальные эндпоинты, а к этому методу еще вернемся.

Начнем с метода GET для получения одной книги

```
https://librarian.com/api/v1/books/{id}
```

Входные параметры

Parameter	type	In	Description
Authorization	Bearer-Token	Headers	Токен для авторизации

id	int	path	Идентификатор книги

Мы знаем, чтобы грамотно спроектировать REST APi, нам необходим идентификатор каждого ресурса, поэтому, чтобы получить информацию о конкретной книге, мы в наш URI добавили параметр id.
Это еще одно правило проектирования эндпоинтов:
Сначала название ресурса во множественном числе, следом его идентификатор
/books/{id} (для методов, обрабатывающих конкретную сущность)

Ответ:
Мы сделали запрос на

```
https://librarian.com/api/v1/books/2
```

- **Статус код: 200 (OK)** - говорит нам о том, что сервер успешно обработал запрос и вернул необходимые нам ресурсы
- Тип содержимого (Content-Type): application/json
- Пример ответа:
-

STATUS_CODE	Content-Type	Пример ответа
200 (OK)	application/json	`{ "id": 2, "title": "Example Book 2" }`
403 (Forbidden)	application/json	`{ "detail": "unauthorized" }`
404 (Not Found)	application/json	`{ "detail": "not found" }`

```
{
   "id": 2,
```

```
    "title": "Example Book 2"
}
```

- **Статус код: 403 (Forbidden)**
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
    "detail": "unauthorized"
}
```

- **Статус код: 404 (Not Found)** - говорит нам о том, что ресурс с переданным идентификатором не найден.
  Еще раз самые частые кейсы для GET:
  - 200 - OK
  - 403 - ошибка авторизации
  - 404 - ресурс не найден
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
    "detail": "not found"
}
```

Спроектируем метод POST для создания книги

```
https://librarian.com/api/v1/books
```

Входные параметры

Parameter	type	In	Description
Authorization	Bearer-Token	Headers	Токен для авторизации

Parameter	type	In	Description
title	string	body	Название книги
Content-Type		Headers	application/json

Заголовок `Content-Type` в запросе или ответе добавляется, чтобы сообщить серверу или клиенту о типе тела запроса или ответа.

Пример тела запроса

```
{
  "title": "New Book"
}
```

Ответ:

STATUS_CODE	Content-Type	Пример ответа
201 (Created)	application/json	`{ "id": 3, "title": "New Book" }`
403 (Forbidden)	application/json	`{ "detail": "unauthorized" }`
400 (Bad request)	application/json	`{ "detail": "Validation Error" }`

- **Статус код: 201 (Created)** - данный статус код мы используем, когда оповещаем о создании ресурса
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "id": 3,
   "title": "New Book"
}
```

- **Статус код: 403 (Forbidden)**
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "detail": "unauthorized"
}
```

- Статус код: 400 (Bad request) - говорит клиенту о некорректно сформированном запросе, например, мы вместо title в теле запроса отправили следующее

```
{
   "some_field": "some_value"
}
```

Серверу не удастся создать новую книгу, когда нет названия, поэтому данный запрос принято обрабатывать как некорректный.

- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "detail": "unauthorized"
}
```

Спроектируем метод PATCH для частичного обновления информации о книге. Есть еще метод PUT для обновления ресурса. Разница в них в том, что PUT обновляет весь ресурс целиком, а PATCH частично. На практике чаще используют patch, поэтому остановимся на нем.

```
https://librarian.com/api/v1/books/{id}
```

Входные параметры

Parameter	type	In	Description
Authorization	Bearer-Token	Headers	Токен для авторизации
id	int	path	Идентификатор книги
title	string	body	Название книги
Content-Type		Headers	application/json

Пример тела запроса

```
{
  "title": "New title"
}
```

Ответ:

STATUS_CODE	Content-Type	Пример ответа
200 (OK)	application/json	`{ "id": 2, "title": "New title" }`
403 (Forbidden)	application/json	`{ "detail": "unauthorized" }`
400 (Bad request)	application/json	`{ "detail": "bad request" }`
404 (Not Found)	application/json	`{ "detail": "Book with id 2 Not found" }`

- **Статус код: 200 (OK)**
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "id": 2,
   "title": "New title"
}
```

- **Статус код: 403 (Forbidden)**
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "detail": "unauthorized"
}
```

- Статус код: 400 (Bad request)

```
{
   "some_field": "some_value"
}
```

Серверу не удастся создать новую книгу, когда нет названия, поэтому данный запрос принято обрабатывать как некорректный.

- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "detail": "bad request"
}
```

- **Статус код: 404 (Not Found)** - говорит нам о том, что ресурс с переданным идентификатором не найден и обновлять нечего.
- Тип содержимого (Content-Type): application/json
- Пример ответа:

```
{
   "detail": "Book with id 2 Not bound"
}
```

И спроектируем **DELETE** метод для удаления книги.

Входные параметры

Parameter	type	In	Description
Authorization	Bearer-Token	Headers	Токен для авторизации
id	int	path	Идентификатор книги

- **Статус код: 204 (NO_CONTENT)**

  Как правило, мы не возвращаем ничего при методе DELETE
- **Статус код: 403 (Forbidden)**
- Тип содержимого (Content-Type): application/json
- Пример ответа:
-

STATUS_CODE	Content-Type	Пример ответа
204 (NO_CONTENT)	application/json	ничего не возвращаем
403 (Forbidden)	application/json	`{ "detail": "unauthorized" }`
400 (Bad request)	application/json	`{ "detail": "bad request" }`
404 (Not Found)	application/json	`{ "detail": "Book with id 2 Not found" }`

```
{
   "detail": "unauthorized"
}
```

Давайте повторим:

- Метод **GET** мы используем для получения конкретного ресурса по URI. Частью URI может мыть параметр в пути, например идентификатор ресурса и GET-параметры. Кроме того, нам нужно передавать параметр авторизации на сервер, он может быть JWT токеном в заголовке Authorization или идентификатором сессии в cookies.
  Если метод GET отработал успешно, сервер нам возвращает HTTP RESPONSE со статус кодом 200 и тело ответа.
- Метод **POST** мы используем для создания ресурса на сервере, передавая ему тело запроса с указанным заголовком content-type, если создается какая-то сущность, то сервер возвращает нам ответ 201, если post был в качестве отправки данных на сервер без создания сущности, сервер вернет нам 200. Если наш запрос не удовлетворяет требованиям, то сервер вернет нам ответ 400 и 403 в случае если мы не прошли авторизацию.
- Метод **PATCH** мы используем для частичного обновления ресурса, указывая в пути его идентификатор, при успешном обновлении сервер вернет нам 200 и обновленный ресурс. Если ресурс не найдет, мы получим ответ со статус кодом 404.
- Метод **DELETE** мы используем для удаления ресурса по его идентификатору, в случае успешного удаления получаем 204.
- Вставка STATUSCODES

Еще пару слов про методы.
Методы могут быть безопасными и идемпонентными.

Безопасные методы - это методы, которые не изменяют состояние сервера

или ресурсов при его вызове. Они обычно используются для получения данных.

. При повторном вызове идемпотентный метод не дает отличий в результате.

Метод	Безопасный	Идемпотентный
GET	да	да
POST	нет	нет
PUT	нет	да
PATCH	нет	да
DELETE	нет	да

Наиболее часто используемые статус коды:

code	Значение
200 OK	Успешный ответ на запрос. Возвращается, когда запрос успешно обработан, и результаты запроса находятся в теле ответа.
201 Created	Означает, что новый ресурс успешно создан. Возвращается после успешного создания нового ресурса, и его идентификатор или ссылка на него могут быть возвращены в теле ответа.
204 No Content	Запрос успешно обработан, но в ответе нет содержимого. Возвращается, например, после успешного удаления ресурса.
400 Bad Request	Сервер не может обработать запрос из-за ошибки в самом запросе, например, неправильного формата данных или отсутствующих обязательных параметров.
401 Unauthorized	Требуется аутентификация пользователя для доступа к запрашиваемому ресурсу. Ответ отправляется, если пользователь не предоставил или предоставил недействительные учетные данные.
403 Forbidden	Запрос не может быть выполнен, поскольку клиенту отказано в доступе к запрашиваемому ресурсу. Ответ отправляется, если у пользователя отсутствуют достаточные полномочия или разрешения для доступа к ресурсу.
404 Not Found	Запрашиваемый ресурс не найден на сервере. Возвращается, когда сервер не может найти ресурс, указанный в URI.
500 Internal Server Error	Обозначает внутреннюю ошибку сервера. Он указывает на то, что сервер не смог корректно обработать запрос из-за ошибки,

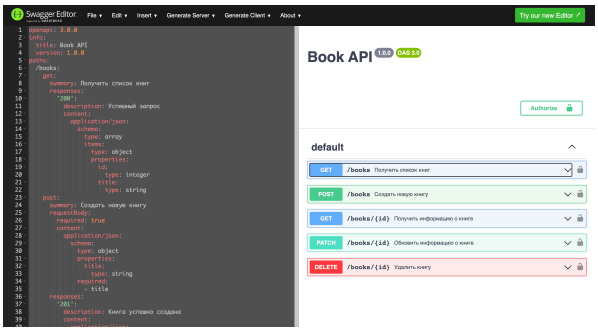
code	Значение
	произошедшей на его стороне.

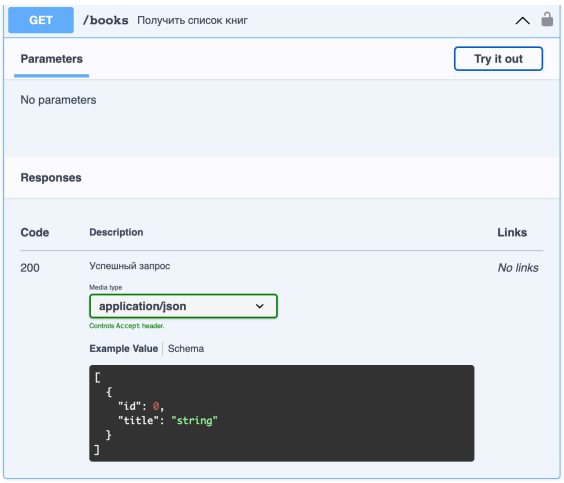
Мы уже разобрались с основами проектирования, спроектировали первые методы и зафиксировали документацию в текстовом формате. Этот формат подходит для ведения полной документации всего проекта, но не является единственным.

Давайте теперь познакомимся со Swagger и OpenApi.

Swagger и OpenAPI - это инструменты для описания и документирования веб-сервисов и REST API. Swagger - отдельный эндпоинт на вашем проекте, который предоставляет информацию о всех доступных методах.

Swagger может быть автогенерируемым средствами библиотек или написан вручную. Посмотрим на оба примера. Перейдем на сайт https://editor.swagger.io/ и задокументируем наши методы вручную.
Пропишем все методы в YAML формате, согласно документации. Не будем подробно останавливаться на формате описания методов, это вы можете успешно загуглить.
Справа мы видим список наших методов. Нажав на каждый из них мы можем посмотреть список входных параметров и параметров ответа, а также попробовать отправить запроса, нажав на Try it out

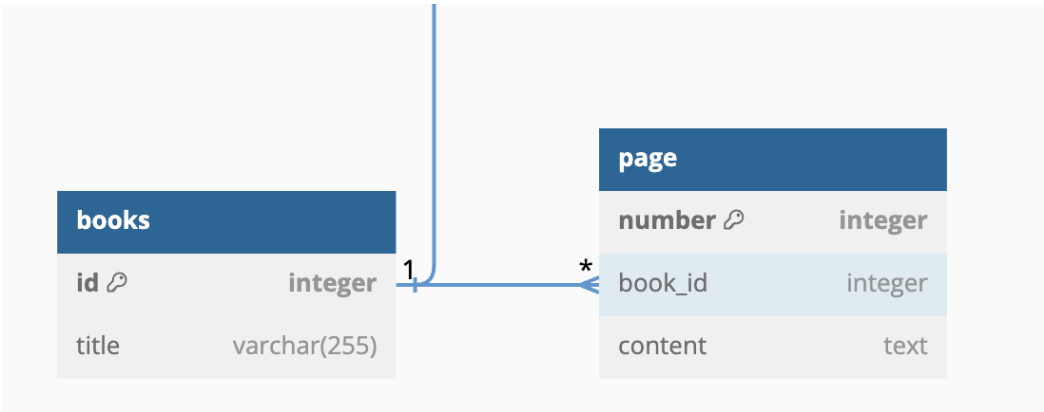




И вот пример на FastApi:

Мы создаем endpoint с указанием метода и в response model указываем возвращаемые параметры, сваргер с этим методом будет доступен на /docs.

Теперь давайте вернемся к проектированию наших методов и спроектируем методы для связанных ресурсов.



В схеме нашей БД есть таблица page и таблица books, связанные отношением one to many. То есть у одной книги может быть много страниц. Спроектируем методы для страниц. Я буду все проецировать в swagger для удобства.

Прежде всего именование URL.

Порядок такой:
api_version/parent/{parent_id}/children

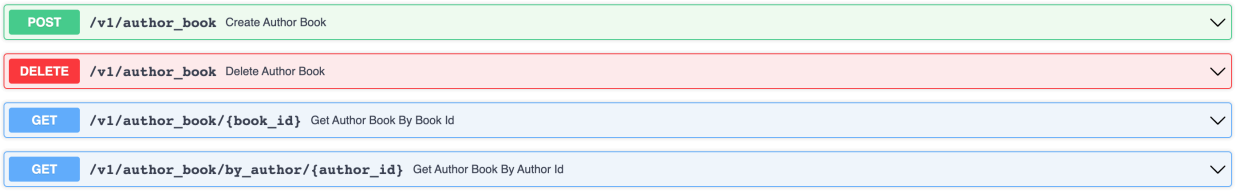
То есть сначала идет версия апи, затем название родительской сущности (в нашем случае книги) во множественном числе, идентификатор книги, название дочерней сущности /page

Для получения списка страниц книги мы используем ендпоинт /books/{id}/pages, а дальше все методы работают по аналогии с книгой.

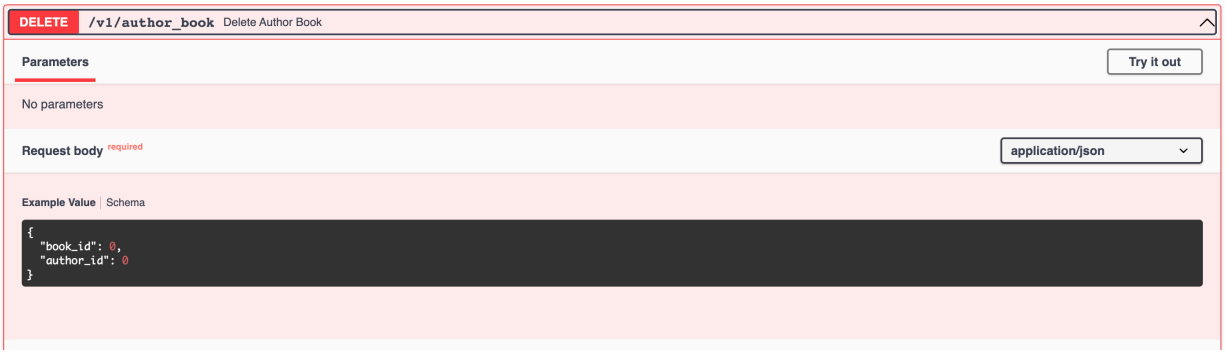
Основная суть в том, что идентификатор родительской сущности мы передаем в параметрах пути.

Что касается связи many to many.
У нас есть табличка связи author_book, которая говорит нам о том, что у одного автора может быть много книг и у одной книги может быть несколько авторов.

Я предпочитаю формировать URI для этого случая в виде названия таблички связи. То есть у нас будет.



Для удаления автора у книги или книги у автора используем delete запрос с указанием в теле идентификатора автора и книги.



Кроме того, у нас появляется 2 метода.
Первый метод для получения всех книг автора и метод для получения всех авторов книги.

Расширение функциональности RESTAPi путем ввода дополнительных параметров.
Давайте вернемся к эндпоинту для получения списка книг и подумаем, что можно улучшить.

1. Пагинация.
   Нам важно не загрузить клиент огромным количеством данных, поэтому для получения ресурсов небольшими частями используют пагинацию.

   a. Пагинация через limit и offset
      В GET-параметры мы добавляем 2 параметра:
      limit: определяет, сколько элементов мы хотим получить

      offset: сколько мы хотим пропустить (пропускаем мы те, которые были на предыдущей странице)

      A backend обязан нам сообщить общее количество элементов для отрисовки на странице

   b. Второй способ через page и size

      i. Параметр page - номер страницы

      ii. Параметр size - количество элементов на странице

      iii. backend рассчитывает для нас количество страниц с указанным size

2. Фильтрация.

   a. Если фильтров много, то мы можем передавать их в теле запроса и запрос необходимо сделать POST, чтобы не выйти за максимальную длину урла в браузере

   b. Если фильтров немного, то указываем их в виде GET параметров

      Например, нам нужно получить только те книги, у которых названия “молчание небес” и “война и мир” для этого, мы вводим фильтр titles[], пара квадратных скобок говорит бэку, что необходимо парсить параметр в массив.

      Наш URI будет выглядеть следующим образом

      ```
      https://librarian.com/api/v1/books/?titles[]='Война и мир'&titles[]='Молч
      ```

      Если нам нужно использовать **range фильтр**, то есть диапозон чего-то мы можем написать следующий параметр

```
https://librarian.com/api/v1/books/?price[lte]=500&price[gte]=200
```

Если мы хотим отсортировать наш массив, то вводим GET-параметр

```
https://librarian.com/api/v1/books/?ordering=title
```

или

```
https://librarian.com/api/v1/books/?sort_by=-title,author
```

Примерно таким образом мы можем расширить функциональность наших апи введением дополнительных параметров.

Давайте еще поговорим о тестировании всего этого добра.

Тестировать ресты можно:

- на бэке писать интеграционные тесты например на Pytest
- вручную через сваггер
- автотесты через Postman

Проверять нужно максимальное количество тест кейсов, покрывая схему ответа, все возможные статус коды и входные параметры.