ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ "РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

Курсовой проект

по дисциплине «Компьютерные технологии управления в технических системах»

на тему «Приложение с базой данных»

Выполнил: ст. гр. ТУУ–151

Николаев А.Ю., Куликова К.И., Гришина М.А.

Проверил: доц. к.т.н. Васильева М.А.

Москва – 2020 г.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 2](#_Toc62299830)

[1. ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННОЙ БАЗЫ ДАННЫХ 3](#_Toc62299831)

[1.1. ИНФОЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ 3](#_Toc62299832)

[1.1.1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc62299833)

[1.1.2. АНАЛИЗ ИНФОРМАЦИОННЫХ ЗАДАЧ И КРУГА ПОЛЬЗОВАТЕЛЕЙ СИСТЕМЫ 5](#_Toc62299834)

[1.2. ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННОЙ БД 6](#_Toc62299835)

[1.2.1. ПРЕОБРАЗОВАНИЕ ER-ДИАГРАММЫ В СХЕМУ БАЗЫ ДАННЫХ 6](#_Toc62299836)

[1.2.2. СОСТАВЛЕНИЕ РЕЛЯЦИОННЫХ ОТНОШЕНИЙ 8](#_Toc62299837)

[1.2.3. НОРМАЛИЗАЦИЯ ПОЛУЧЕННЫХ ОТНОШЕНИЙ 9](#_Toc62299838)

[1.2. ФИЗИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БД 10](#_Toc62299839)

[1.2.1. РАЗРАБОТКА СКРИПТОВ НА СОЗДАНИЕ БАЗЫ ДАННЫХ И ТАБЛИЦ 10](#_Toc62299840)

[2. РАЗРАБОТКА ПРИЛОЖЕНИЯ 15](#_Toc62299841)

[2.1. NHIBERNATE 15](#_Toc62299842)

[2.2. ORM, репозитории, сессии, сервисы. 16](#_Toc62299843)

[2.3. СЕССИИ 17](#_Toc62299844)

[2.4. ФАБРИКА СЕССИЙ (SESSION FACTORY) 18](#_Toc62299845)

[2.5. ТЕСТИРОВАНИЕ МАППИНГОВ 20](#_Toc62299846)

[2.6. ПРИМЕРЫ ЗАПРОСОВ 21](#_Toc62299847)

[2.7. ПРИНЦИПЫ REST 27](#_Toc62299848)

[СПИСОК ЛИТЕРАТУРЫ 32](#_Toc62299849)

ВВЕДЕНИЕ

Сегодня любая IT-разработка очень тесно связана с использованием баз данных. Чаще всего приходится работать с реляционными СУБД. Поэтому приходится использовать различные драйвера к базам данных для необходимых языков программирования, что приводит к постоянному написанию SQL-запросов внутри кода программы, к использованию большого количества необходимых классов, чтобы выполнить достаточно простые операции получения/изменения информации внутри базы.

Таким образом, появляется задача обеспечения работы с данными в терминах классов/объектов, а не таблиц. Необходимо преобразовать термины и данные классов в данные, пригодные для хранения в СУБД, а также обеспечить интерфейс для операций создания, получения, обновления, удаления (create, read, update, delete – CRUD) данных, и в целом избавиться от написания SQL-запросов в коде программы для взаимодействия с СУБД.

Решением проблемы является объектно-реляционное отображение (object relational mapping – ORM) – технология, связывающая объекты в рамках понятий баз данных и объекты в рамках объектно-ориентированного программирования. Применение объектно-реляционного отображения в настоящее время является общим средством в процессе разработки сложных систем, позволяющим объединить объектно-ориентированную модель представления данных с реляционной.

ORM является промежуточным слоем между базой данных и кодом программы, которая позволяет выполнять CRUD-операции в базе данных, в частности, в таблицах, с помощью использования специального API и ООП-объектов.

Целью данной работы является реализация подобного отображения, которое можно будет использовать на практике.

1. ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННОЙ БАЗЫ ДАННЫХ
   1. ИНФОЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ
      1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

БД создается для домашнего обслуживания библиотеки. Библиотека содержит книги разных авторов, изданий, тематик. БД предназначена для поиска выбранной книги на полке.

ER-диаграмма БД представлена на Рисунке 1.

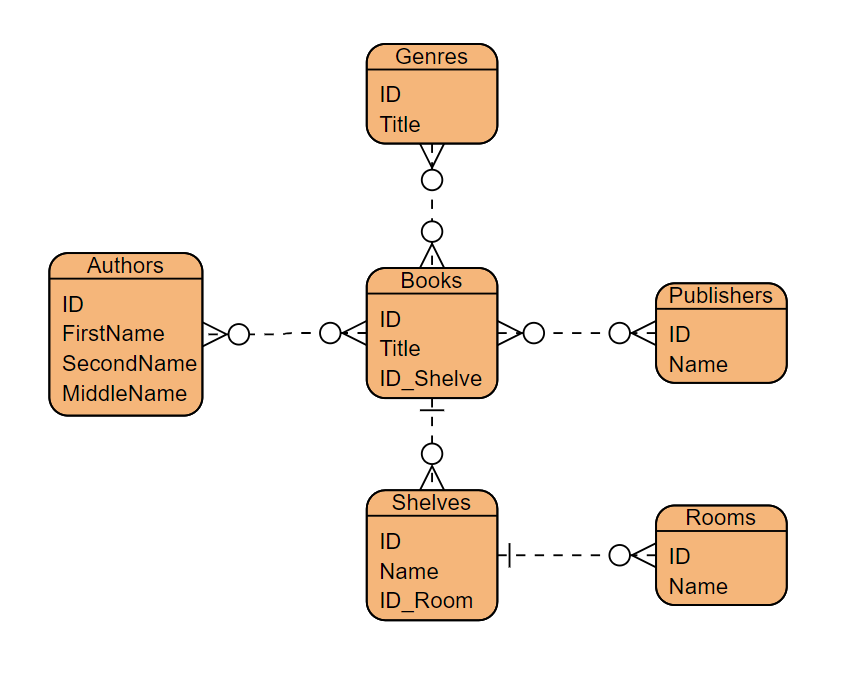


Рисунок 1 - ER-диаграмма

В полученной ER-диаграмме присутствует связь многие - ко - многим, поэтому вводится специальная вспомогательная сущность, которая является соединением первичных ключей соответствующих сущностей. Таким образом разбиваются все связи типа n:m. Затем строится уточненная ER-диаграмма (Рисунок 2).

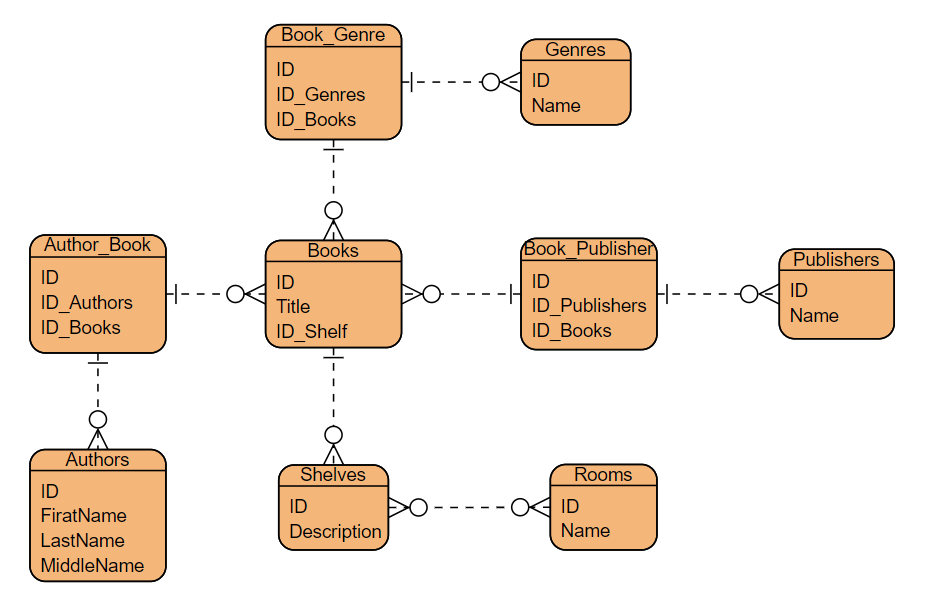


Рисунок 2 - Уточненная ER-диаграмма

* + 1. АНАЛИЗ ИНФОРМАЦИОННЫХ ЗАДАЧ И КРУГА ПОЛЬЗОВАТЕЛЕЙ СИСТЕМЫ

Система создаётся для домашнего обслуживания библиотеки.

Определим границы информационной поддержки пользователей:

1) Функциональные возможности:

1. Ведение БД (запись, чтение, модификация, удаление в архив)

2. Обеспечение логической непротиворечивости БД

3. Обеспечение защиты данных от несанкционированного или случайного доступа (определение прав доступа)

4. Реализация наиболее часто встречающихся запросов в готовом виде.

2) Готовые запросы:

1. Выдавать список книг по названию.

2. Выдавать список трудов данного автора (учитывать труды, выполненные в соавторстве).

3. Выдавать список книг по данной тематике.

4. Выдавать список книг данного издательства.

5. Выдавать местонахождение данной книги.

1.2. ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННОЙ БД

1.2.1. ПРЕОБРАЗОВАНИЕ ER-ДИАГРАММЫ В СХЕМУ БАЗЫ ДАННЫХ

База данных создаётся на основании схемы базы данных. Преобразование ER-диаграммы в схему БД выполняется путем сопоставления каждой сущности и каждой связи, имеющей атрибуты, отношения (таблицы БД). На схеме (Рисунок 3) введены обозначения:

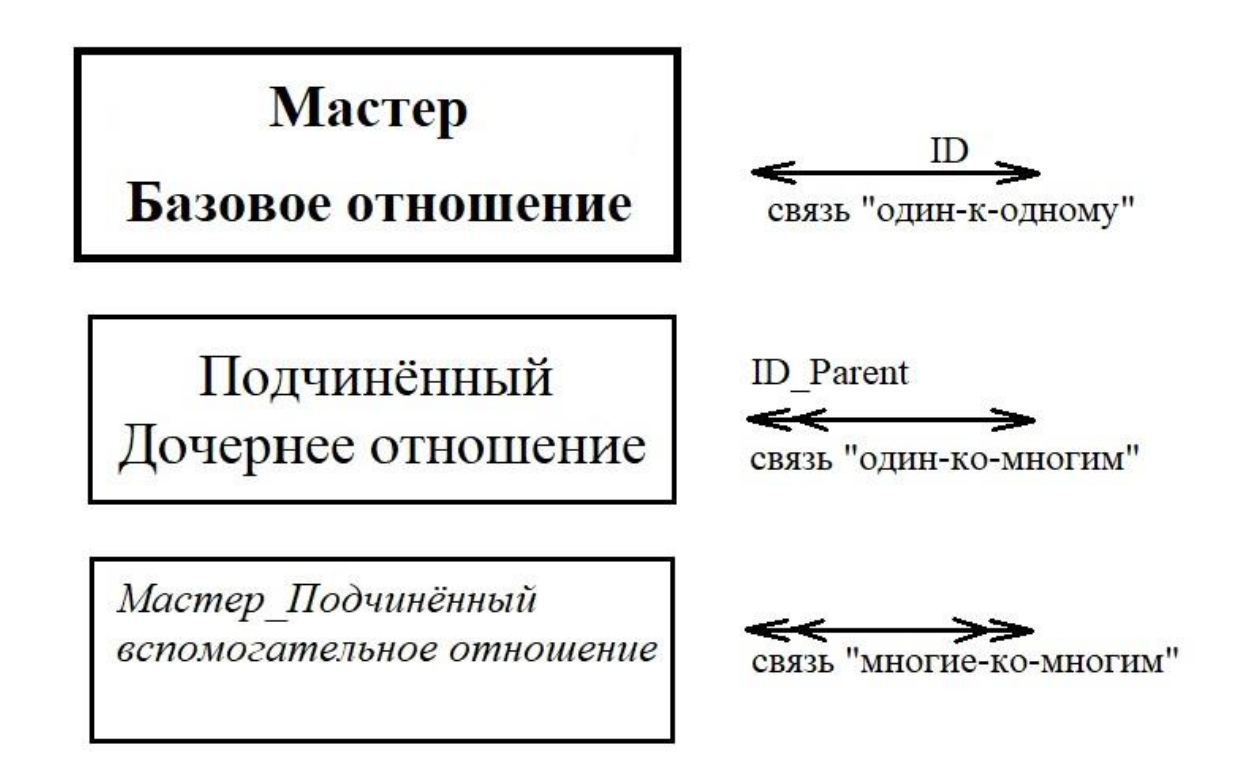


Рисунок 3 - Обозначения, используемые на схеме базы данных

Полученная схема реляционной базы данных (РБД) «Библиотека» приведена на Рисунке 4.

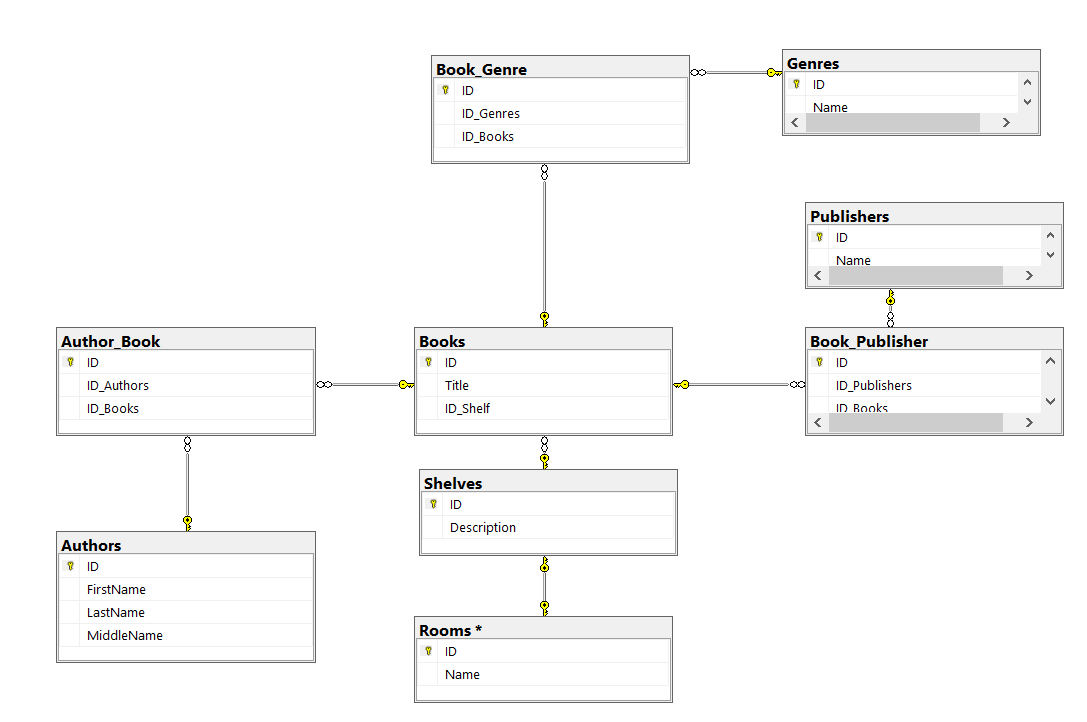


Рисунок 4 - Диаграмма базы данных

1.2.2. СОСТАВЛЕНИЕ РЕЛЯЦИОННЫХ ОТНОШЕНИЙ

Каждое реляционное отношение соответствует одной сущности (объекту ПрО) и в него вносятся все атрибуты сущности. Для каждого отношения необходимо определить первичный ключ и внешние ключи (если они есть). В том случае, если базовое отношение не имеет потенциальных ключей, вводится суррогатный первичный ключ, который не несёт с

Каждая сущность имеет связь многое-ко-многим, как показано на рисунке 1, где главной сущностью является “Books” — может принадлежать много авторов, жанров, одно издательство, находиться на одной полке в одной комнате. Но при этом много книг может быть в одной комнате. Автор может написать или участвовать во многих книгах.

1.2.3. НОРМАЛИЗАЦИЯ ПОЛУЧЕННЫХ ОТНОШЕНИЙ

1НФ. Наша база данных соответсвует 1НФ. Т.к все данные в полях атомарны, т.е одно поле – одно значение.

2НФ. Все отношения находятся во 2НФ, так как для каждого из них введен уникальный ключ и все записи от него зависят.

3НФ. Отношения в БД находится в 3НФ. Т.к третья нормальная форма исключает зависимость неключевых полей от других неключевых полей.

* 1. ФИЗИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БД
     1. РАЗРАБОТКА СКРИПТОВ НА СОЗДАНИЕ БАЗЫ ДАННЫХ И ТАБЛИЦ

БД «Библиотека» проектируется в среде Microsoft SQL Server Management Studio.

Создание базы данных «Библиотека»:

create database DBLibraryHome

go

Создание таблицы «Authors» :

use DBLibraryHome

go

CREATE TABLE [dbo].[Authors]

(

[ID] int primary key

, [FirstName] nvarchar(255) not null

, [LastName] nvarchar(255) not null

, [MiddleName] nvarchar(255)

);

Создание таблицы «Genres»:

use DBLibraryHome

go

CREATE TABLE [dbo].[Genres]

(

[ID] int primary key

, [Name] nvarchar(255) not null

);

Создание таблицы «Publishers» :

use DBLibraryHome

CREATE TABLE [dbo].[Publishers]

(

[ID] int primary key

, Name nvarchar(255) not null

);

Создание таблицы «Shelves»:

CREATE TABLE [dbo].[Shelves]

(

[ID] int primary key

, Description nvarchar(255) not null

, FOREIGN KEY ([ID\_Room])

REFERENCES Rooms ([ID])

);

Создание таблицы «Rooms»:

CREATE TABLE [dbo].[Rooms]

(

[ID] int primary key

, Name nvarchar(255) not null

);

Создание таблицы «Books»:

CREATE TABLE [dbo].[Books]

(

[ID] int primary key

, Title nvarchar(255) not null

, [ID\_Shelve] int

, foreign key ([ID\_Shelve])

references Shelves([ID])

);

Создание таблицы «Author\_Book»:

use DBLibraryHome

CREATE TABLE [dbo].[Author\_Book]

(

[ID] int primary key

, [ID\_Authors] int not null

, [ID\_Books] int not null

, FOREIGN KEY ([ID\_Authors])

REFERENCES Authors ([ID])

, FOREIGN KEY ([ID\_Books])

REFERENCES Books ([ID])

);

Создание таблицы «Book\_Genre»:

use DBLibraryHome

CREATE TABLE [dbo].[Book\_Genre]

(

[ID] int primary key

, [ID\_Genres] int not null

, [ID\_Books] int not null

, FOREIGN KEY ([ID\_Genres])

REFERENCES Genres ([ID])

, FOREIGN KEY ([ID\_Books])

REFERENCES Books ([ID])

);

Создание таблицы «Book\_Publisher»:

use DBLibraryHome

CREATE TABLE [dbo].[Book\_Publisher]

(

[ID] int primary key

, [ID\_Publishers] int not null

, [ID\_Books] int not null

, FOREIGN KEY ([ID\_Publishers])

REFERENCES Publishers ([ID])

, FOREIGN KEY ([ID\_Books])

REFERENCES Books ([ID])

);

* + 1. РАЗРАБОТКА СКРИПТОВ НА ДОБАВЛЕНИЕ ДАННЫХ В ТАБЛИЦЫ

Заполнение таблицы «Authors»:

insert into Authors

values (1, 'Александр', 'Пушкин', 'Сергеевич')

, (2, 'Лев', 'Толстой', 'Николаевич')

, (3, 'Фёдор', 'Достоевский', 'Михайлович')

, (4, 'Денис', 'Фонвизин', 'Иванович')

, (5, 'Василь', 'Быков', 'Владимирович');

Заполнение таблицы «Books»:

insert into Books

values (1, 'Капитанская дочка', 1)

, (2, 'Война и Мир', 1)

, (3, 'Преступление и наказание', 1)

, (4, 'Недоросль', 2)

, (5, 'Сотников', 3);

Заполнение таблицы «Genres»:

insert into Genres

values (1, 'Роман')

, (2, 'Повесть')

, (3, 'Комедия');

Заполнение таблицы «Publishers»:

insert into Publishers

values (1, 'Эксмо')

, (2, 'Юрайт')

, (3, 'Рипол классик');

Заполнение таблицы «Rooms»:

insert into Rooms

values (1, 'Зал')

, (2, 'Спальня')

, (3, 'Коридор');

Заполнение таблицы «Shelves»:

insert into Shelves

values (1, 2, 'Верхняя полка')

, (2, 1, 'Нижняя полка')

, (3, 3, 'Средняя полка');

Заполнение таблицы «Author\_Book»:

insert into AuthorBook

values (1, 1, 1)

, (2, 2, 2)

, (3, 3, 3)

, (4, 4, 4)

, (5, 5, 5);

Заполнение таблицы «Book\_Publisher»:

insert into Book\_Publisher

values (1, 1, 1)

, (2, 2, 2)

, (3, 3, 3)

, (4, 1, 4)

, (5, 3, 5);

Заполнение таблицы «Book\_Genre»:

insert into Book\_Genre

values (1, 1, 1)

, (2, 2, 2)

, (3, 3, 3)

, (4, 2, 4)

, (5, 3, 5);

1. РАЗРАБОТКА ПРИЛОЖЕНИЯ
   1. NHIBERNATE

NHibernate — ORM-решение для платформы Microsoft .NET, портированное с Java. Это бесплатная библиотека с открытым кодом, распространяется под лицензией GNU Lesser General Public License.

NHibernate позволяет отображать объекты бизнес-логики на реляционную базу данных. По заданному XML-описанию сущностей и связей NHibernate автоматически создает SQL-запросы для загрузки и сохранения объектов.

* 1. ORM, репозитории, сессии, сервисы.

ORM – это аббревиатура для Object Relational Mapping (Объектно-реляционное отображение). Техника программирования, предназначенная для преобразования между несовместимыми типами данных в объектно-ориентированных языках программирования. Суть определения заключается в том, что создается абстракция - “виртуальная объектная база”, запросы к которой, преобразуются в SQL команды.

М. Фаулер даёт следующую формулировку архитектурному шаблону проектирования отображение данных (Data Mapper): слой отображателей (Mapper), который осуществляет передачу данных между объектами и БД, сохраняя последние независимыми друг от друга и от самого отображателя.

Типы отображения:

* Отображение на основе XML.
* Отображение на основе атрибутов.
* Текучее отображение (fluent mapping).
* Отображение на основе соглашений, иногда называемое автоотображением (auto-mapping).
  1. СЕССИИ

NHibernate сессию можно представить как абстрактный или виртуальный канал для БД. Раньше вам надо было создавать ADOConnection, открывать Connection, передавать Connection к объекту Command, создавать DataReader из объекта Connection и так далее.

С NHibernate мы просим sessionFactory объект Session, и все. NHibernate поддерживает все «реальные» сессии с БД (соединение, пулы и т.д.). Мы используем преимущества соединения с БД без знания всех тонкостей, лежащих в его основе.

Через объект сессии мы можем добавить новые данные в БД, изменить или удалить существующие данные, а также считывать данные из БД. Все эти операции могут быть выполнены в стиле ООП, без знания SQL и других специфических тонкостей БД. Объект сессии позволяет нам взаимодействовать с данными, хранящимися в БД, и не зависеть от выбранной СУБД (SQL Server, MySQL, Oracle и т.д.). NHibernate полностью абстрагирует эти детали от нас.

* 1. ФАБРИКА СЕССИЙ (SESSION FACTORY)

NHibernate использует объект фабрика, чтобы создать сущности сессии. Один объект фабрики сессии может создавать множество объектов сессии. Создание новой сессии очень дешёвая операция, в отличие от трудозатратной операции – создание фабрики сессий. В зависимости от сложности системы, на создание объекта фабрики сессий тратится значительное количество времени. По этой причине мы должны создавать фабрику сессий единожды в течении жизненного цикла приложения.

Фабрика сессий специфична к БД. Если наше приложение нуждается только в одной БД, нам необходима только одна фабрика. Если наше приложение взаимодействует с несколькими разными БД, тогда нам необходим объект фабрики сессий для каждой БД. В более развитых сценариях может случится так, что понадобится более, чем один объект фабрики сессий, даже, если мы имеем доступ к единственной БД. Причиной для этого может служить то, что наша БД имеет несколько разных схем, и мы хотим иметь для каждой схемы свой объект фабрики сессий. Другой причиной, по которой может существовать более одного объекта фабрики сессий для единственной БД, является существование нескольких предметных областей (часто называемый ограниченный контекст ), и мы хотим иметь доступ к БД из каждой подобласти через сессии, созданные фабрикой сессий ограниченного контекста.

Фабрика сессий потокобезопасна. Код, запущенный в разных потоках, может использовать тот же самый объект фабрики сессий для создания новых объектов сессии. Это контрастирует с тем фактом, что объект сессии может быть использован только в единственном потоке. Другими словами, объект сессии не потокобезопасен.

Важные интерфейсы Hibernate:

* SessionFactory (org.hibernate.SessionFactory) – неизменяемый потокобезопасный объект с компилированным маппингом для одной базы данных. Необходимо инициализировать SessionFactory всего один раз. Экземпляр SessionFactory используется для получения объектов Session, которые используются для операций с базами данных.
* Session (org.hibernate.Session) – однопоточный короткоживущий объект, который предоставляет связь между объектами приложения и базой данных. Он оборачивает JDBC java.sql.Connection и работает как фабрика для org.hibernate.Transaction. Разработчик должен открывать сессию по необходимости и закрывать ее сразу после использования. Экземпляр Session является интерфейсом между кодом в java приложении и hibernate framework и предоставляет методы для операций CRUD.
* Transaction (org.hibernate.Transaction) – однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC или JTA транзакций. org.hibernate.Session может занимать несколько org.hibernate.Transaction в определенных случаях.
  1. ТЕСТИРОВАНИЕ МАППИНГОВ

Для тестирования отображений FluentNHibernate предоставляет Persistence specification testing framework. Необходимо создать новую сущность PersistenceSpecification с универсальным типом сущности, которую нужно протестировать. Для каждого свойства вызывается метод CheckProperty, для ссылок вызывается CheckReference, для коллекций в связи многие-ко-многим – CheckList. Далее назначаются тестовые значения и параметр для сравнения. После вызывается VerifyTheMappings, который выполняет основные операции CRUD.

В тесте не важно, установлено ли какое-то конкретное или случайное значение. Важно то, что свойство может быть записано в базу данных и получено из нее.

* 1. ПРИМЕРЫ ЗАПРОСОВ
* Метод GET

/api/author - список всех авторов из БД . Результат запроса представлен на Рисунке 5.

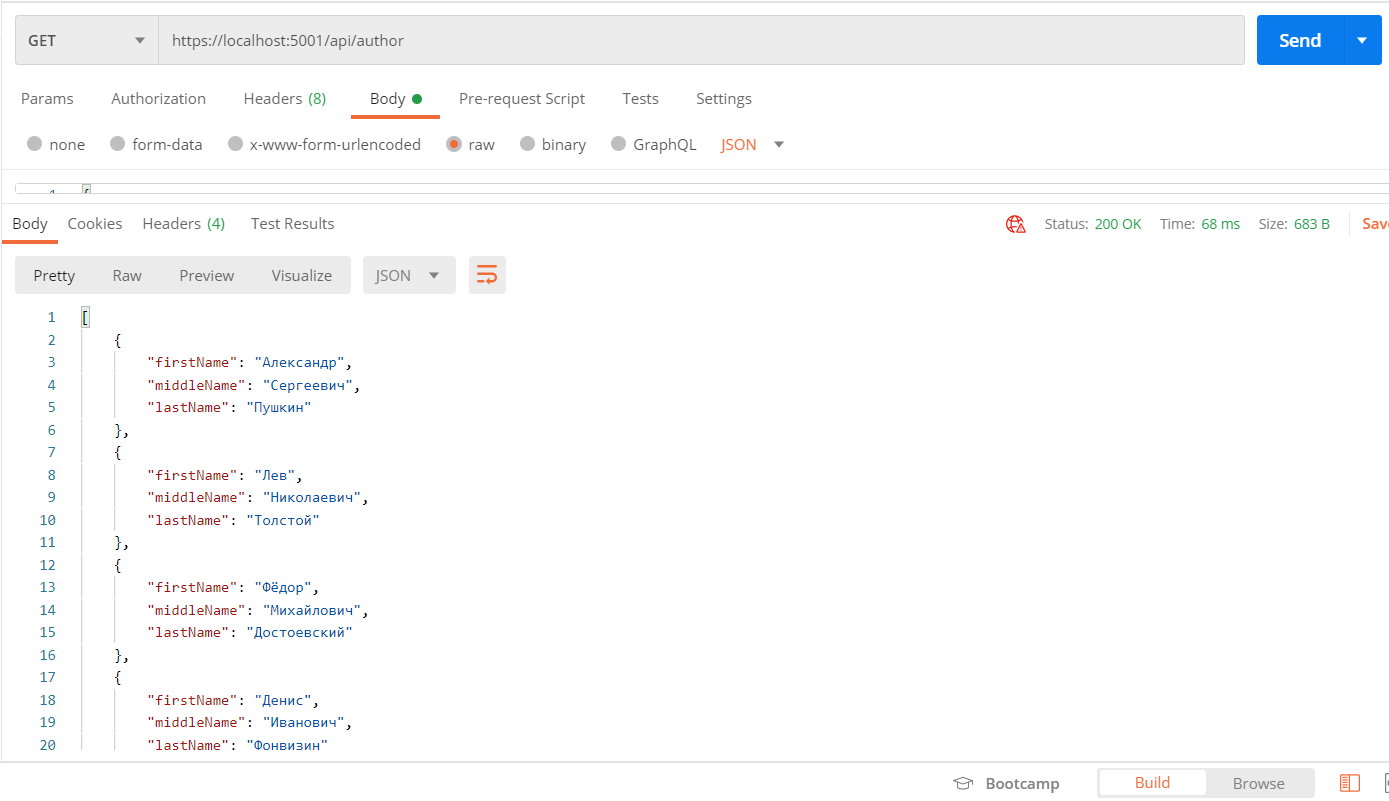


Рисунок 5 – Получение всех авторов из БД

/api/author/{id} - автор определенного id. Результат запроса представлен на Рисунке 6.

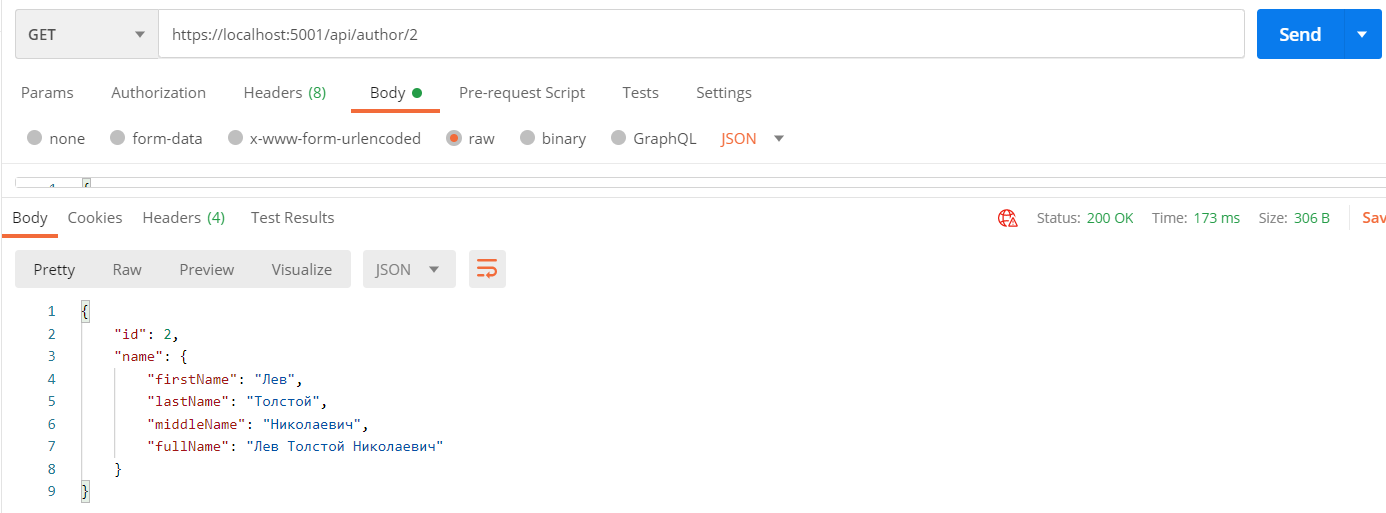


Рисунок 6 - Получение автора с определенным id

/api/book - список всех книг библиотеки. Результат запроса представлен на Рисунках 7-8.

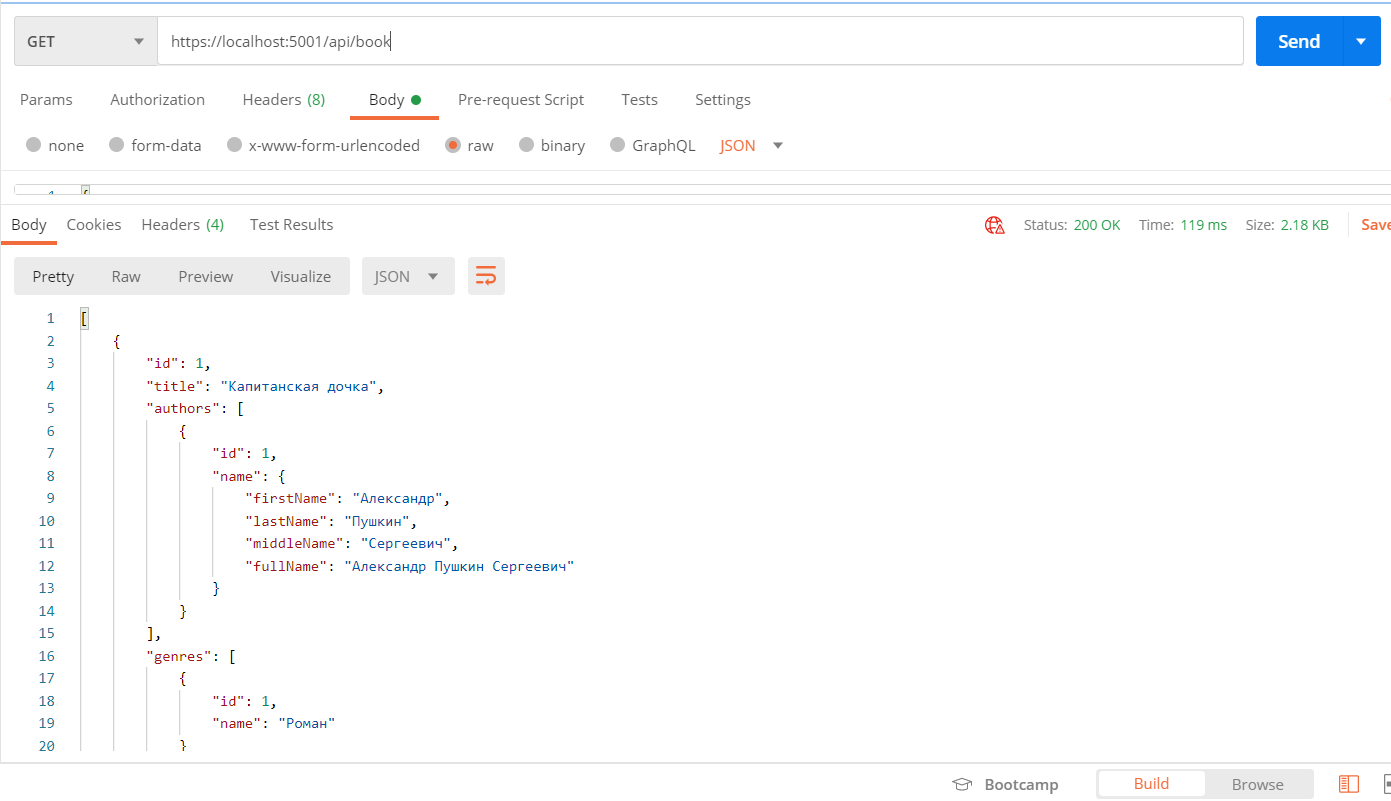


Рисунок 7 - Получение всех книг библиотеки



Рисунок 8 - Получение всех книг библиотеки

/api/book/{id} - книга определенного id. Результат запроса представлен на Рисунке 9.

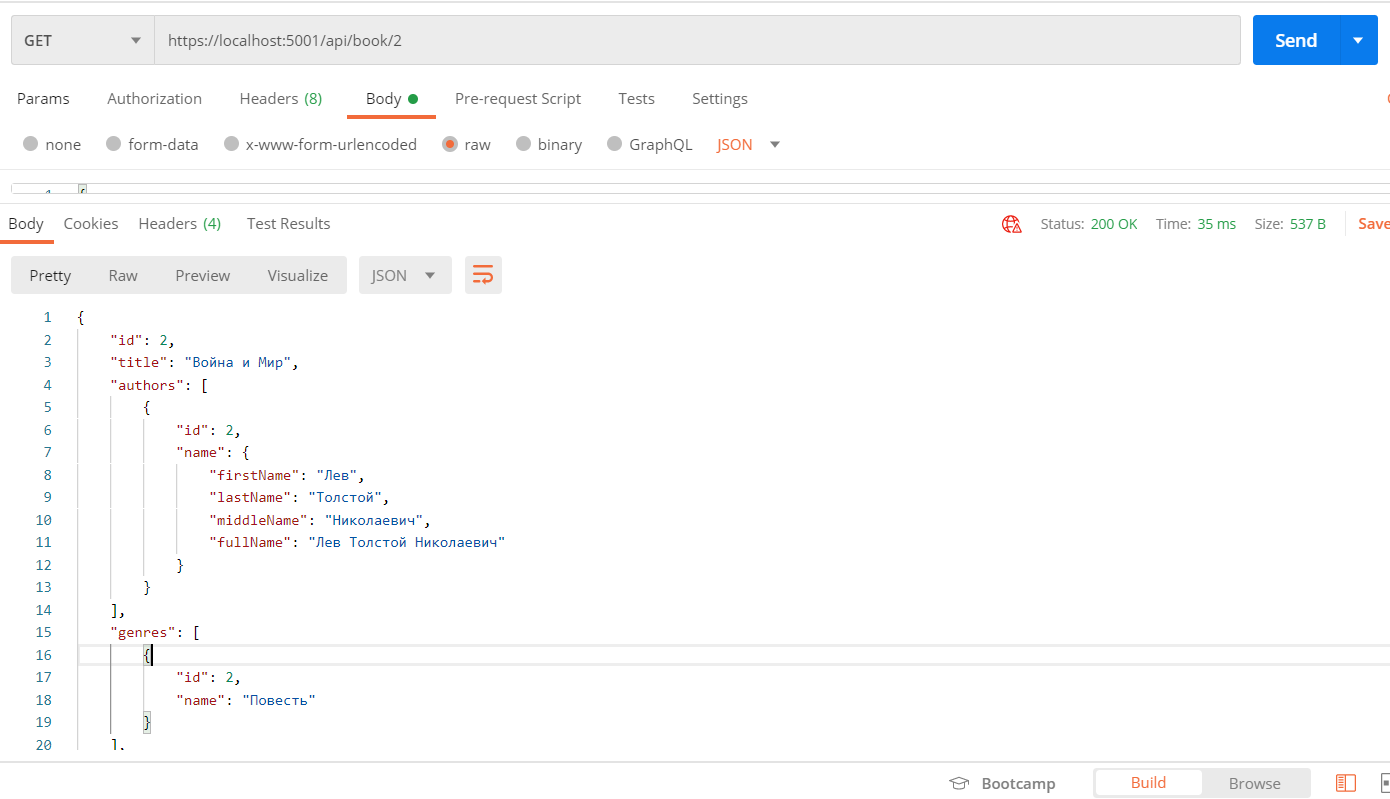


Рисунок 9 - Получение книги по id

/api/book/title/{str} – поиск книги по названию. Результат запроса представлен на Рисунке 10.

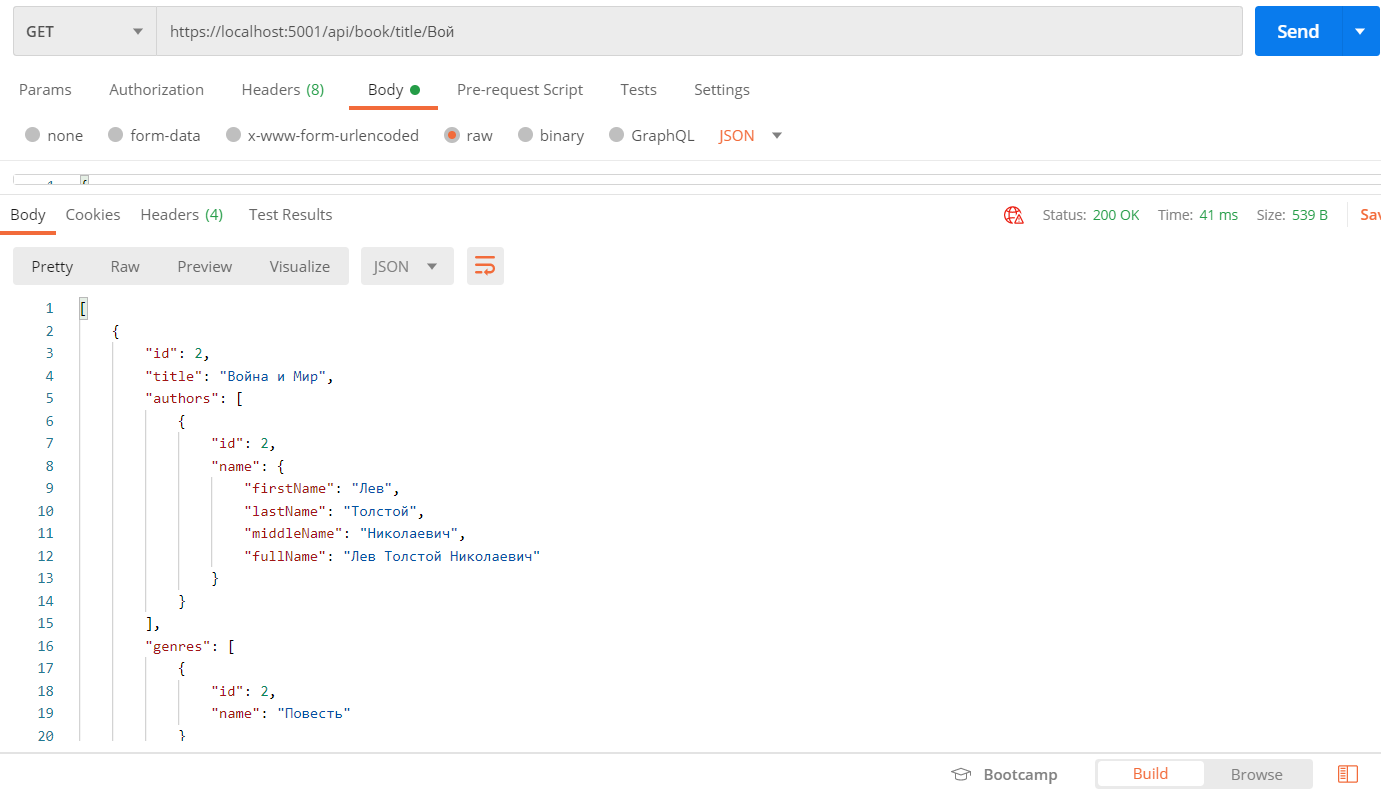


Рисунок 10 - Получение книги по названию

/api/author/for-book/{id} - список авторов книги по её id. Результат запроса представлен на Рисунке 11.

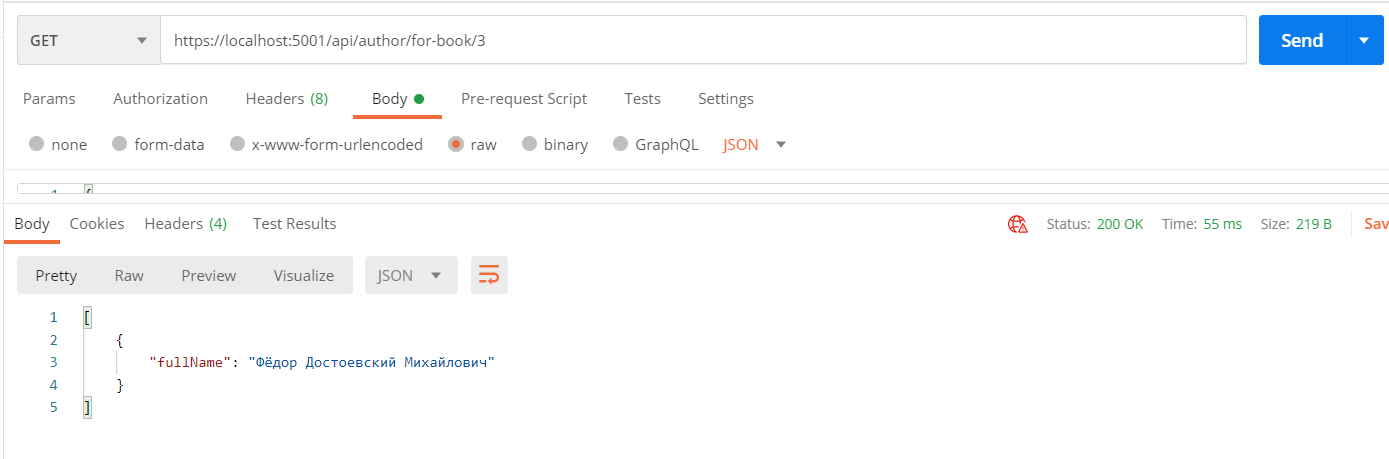


Рисунок 11 - Получение автора по id

* Метод POST:

/api/author - добавление нового автора в БД. Результат запроса представлен на Рисунке 12.

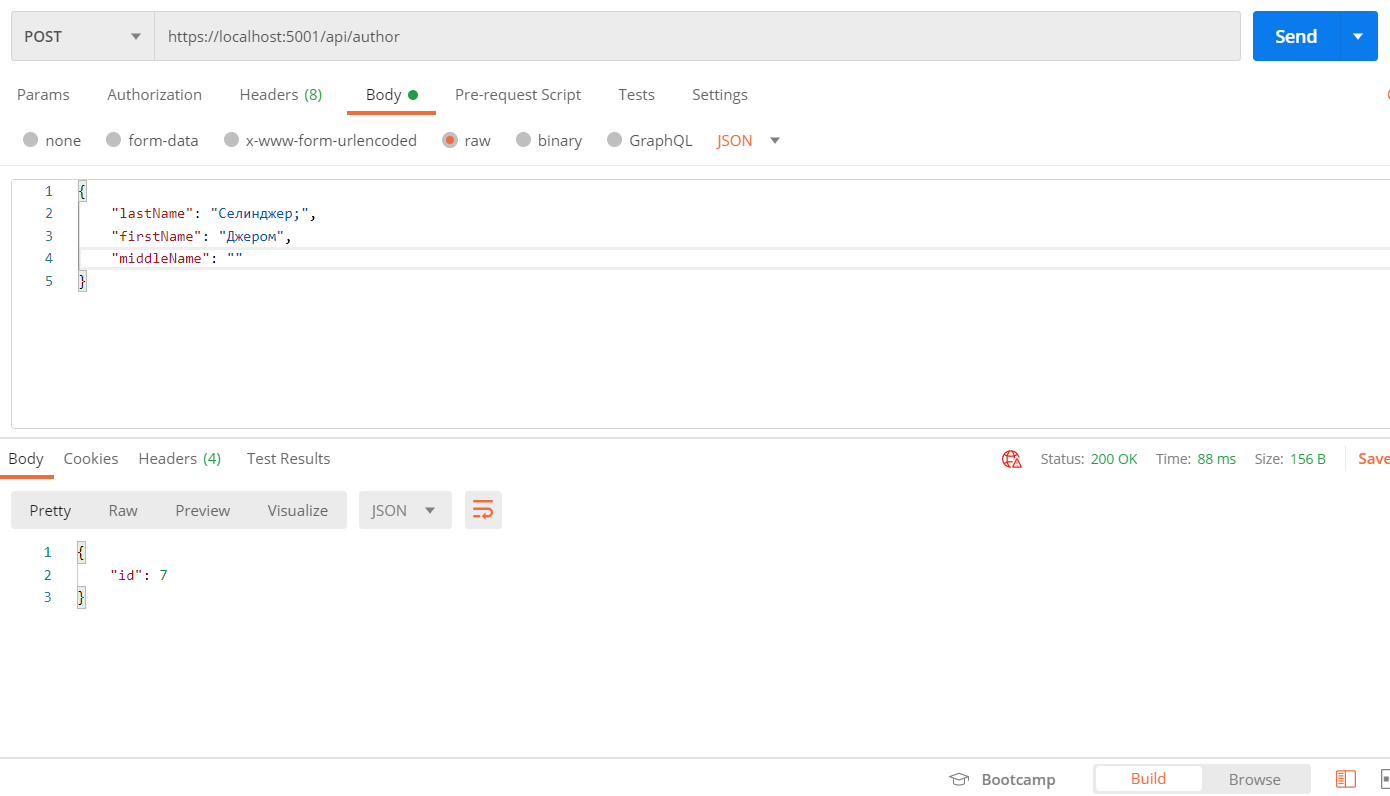


Рисунок 12 - Добавление нового автора в БД

* Метод DELETE

/api/author/{id} - удаление автора по id. Результат запроса представлен на Рисунке 13.

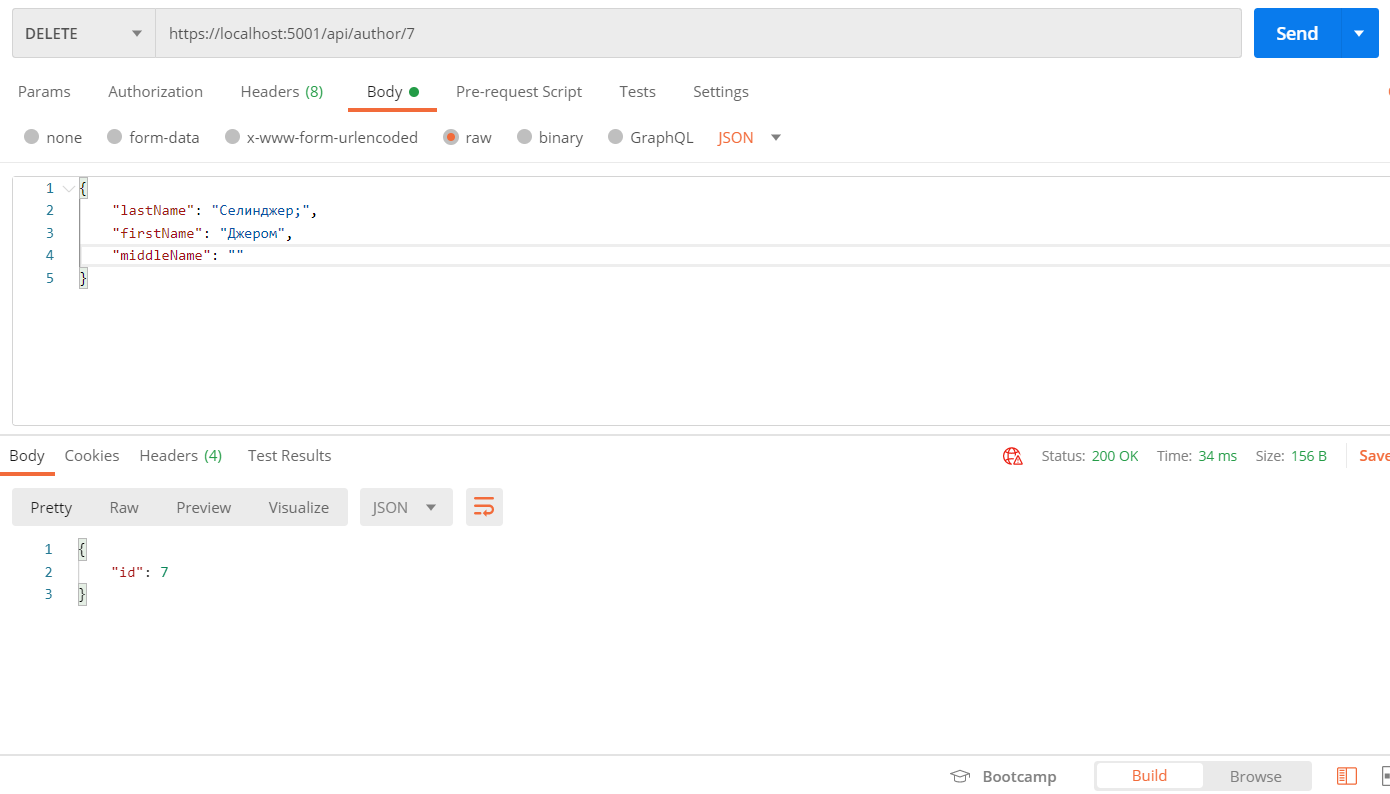


Рисунок 13 - Удаление автора по id

* 1. ПРИНЦИПЫ REST

REST, или Representational State Transfer (англ. «передача состояния представления») — набор принципов, которых рекомендовано придерживаться при создании API. Если API сделан по этим принципам, его называют REST API.

Раньше клиент и сервер были тесно связаны. Например, сервер собирал веб-страницу и отдавал клиенту. Таким образом, серверу нужно было знать, какие технологии работают на стороне клиента, чтобы тот смог открыть вернувшуюся веб-страницу.

Принципы REST позволили отделить клиента от сервера. Благодаря этому:

* стало проще переносить веб-приложение на другие платформы;
* появилась возможность делать открытые API;
* разрабатывать и тестировать серверное ПО стало проще и быстрее.

Поэтому принципы REST важно понимать для программирования сервера.

1. Клиент-сервер

Сервер и клиент отвечают за разные вещи. Ответственность клиента — пользовательский интерфейс, а ответственность сервера — данные. Если API возвращает HTML-страницу, его нельзя назвать REST API: ведь при этом сервер берёт на себя ответственность за интерфейс.

В REST API сервер возвращает данные в каком-то формате — обычно в JSON. Именно этот принцип делает возможным точечную отрисовку страницы и существование единого API для браузера и мобильного приложения.

2. Отсутствие состояния (англ. "Stateless")

Запрос клиента к серверу должен содержать всю информацию, необходимую для обработки этого запроса. В проекте Mesto вы отправляли на сервер токен, чтобы получить карточки. Этот токен сообщал системе, что вы — это вы: запрос содержит информацию о том, ктó запрашивает данные и какие.

Раньше серверы программировали иначе. При аутентификации сервер создавал сессию, то есть запоминал, что пользователь совершил вход. Сейчас такой подход применяется редко.

При отправке токена создавать сессию не нужно. Так что сервер не хранит информацию о состоянии пользователя. Поэтому принцип и называется «отсутствие состояния».

3. Единый интерфейс (англ. "Uniform Interface")

Интерфейс обращения к серверу не зависит от клиента. Он одинаковый для всех.

Таким образом запрос к карточке может быть сформирован из браузера, мобильного приложения и с умного чайника — для всех интерфейс един.

4. Многоуровневость

Первый принцип гласит, что в коммуникации участвуют двое: клиент и сервер. Но мы всё равно можем строить более сложные системы, не нарушая этого принципа. Благодаря многоуровневости.

API сервиса Яндекс.Такси может использовать API Яндекс.Навигатора. Вы как клиент взаимодействуете только с API Яндекс.Такси, а он, в свою очередь, является клиентом навигатора.

5. Кешируемость

Данные ответа могут быть закешированы. Это значит, что мы можем сохранить данные на клиенте и при идентичном запросе взять их из памяти клиента — кеша.

6. Код по запросу (англ. "Code on demand")

Этот принцип необязательный. Он гласит, что функциональность клиента может быть расширена кодом, приходящим с сервера. Сейчас такое можно встретить повсеместно: мы получаем с сервера JS-файлы и исполняем их в браузере. Но принципы формулировались в 2000 году — тогда с сервера код возвращали редко. Потому и выделили это в отдельный пункт.

В REST применяются 5 основных методов:

* GET получает ресурсы;
* POST создаёт ресурс;
* PUT заменяет существующий ресурс целиком;
* PATCH частично изменяет существующий ресурс;
* DELETE удаляет ресурс.

Реже применяют ещё 2 метода:

* HEAD позволяет получить только заголовки ответа. HEAD похож на GET, но у его ответа нет тела;
* OPTIONS — узнать, какие HTTP-методы поддерживает сервер.

Сам по себе метод не определяет логику работы сервера. Можно закодить API так, чтобы POST удалял ресурс, а DELETE — добавлял. Но такой подход противоречил бы заложенной в REST идее, что методы должны применяться по назначению.

Ответ от сервера всегда содержит статус. Правильно выставленный статус делает API более понятным. В этом уроке разберём основные статусы ответов сервера.

Статусы делят на 5 категорий. Запросы каждой из категорий начинаются с разных цифр:

1xx: Informational. Информационный ответ. Такие ответы приходят только техническим системам, вам их отправлять не придётся.

2xx: Success. Запрос прошёл успешно.

3xx Redirection. Запрос не завершён, и клиенту нужно предпринять какие-то действия, чтобы запрос успешно прошёл.

4xx: Client Error. Ошибка на стороне клиента: запрос сформирован неверно или у клиента нет нужных прав доступа.

5xx: Server Error. Ошибка на стороне сервера: что-то сломалось или сервер перегружен.

Самые частые статусы

* 200 OK.

Запрос прошёл успешно. Ответ с таким статусом должен содержать тело. Чаще всего этот статус используют при ответе на GET-запрос ресурса.

* 201 Created.

Ресурс был создан на сервере. Подходящий ответ при создании нового поста в блоге, например.

* 202 Accepted.

Сервер начал работу по удовлетворению запроса, но ещё не закончил. Этот статус применяют для ответа на запросы, обработка которых занимает много времени, например, при обработке большого объёма данных.

* 301 Moved Permanently.

API переделали, а ресурс переместили в другое место. Новый URL указывают в заголовке Location ответа сервера.

* 302 Found.

Запрос должен быть перенаправлен на другой URL. В заголовке Location сервер должен отправить новый URL. При получении запроса браузер автоматически отправит запрос на новый URL.

* 400 Bad Request.

Ошибка на стороне клиента. Например, запрос был неправильно сформирован. Это общий статус: его отправляют в том случае, когда ни один другой 4xx статус не подходит.

* 401 Unauthorized.

Запрос требует авторизации, но соответствующие авторизационные заголовки отсутствуют или сформированы неправильно.

* 403 Forbidden.

Запрос сформирован правильно, но у клиента недостаточно прав, чтобы запрос прошёл успешно. Например, клиент пытается удалить чужой пост.

* 404 Not Found.

Ресурс не найден. Например, пользователя с запрошенным id не существует.

* 405 Method Not Allowed.

Ресурс запроса не поддерживает HTTP-метод, которым сделан запрос.

* 500 Internal Server Error.

Общий статус для ошибок на стороне сервера. Это не ошибка клиента.

* 501 Not Implemented.

Ресурс есть на сервере, но способ обращения к нему пока не реализован.

СПИСОК ЛИТЕРАТУРЫ

J. Dentler, NHibernate 3.0 Cookbook, Birmingham: Packt Publishing Ltd., 2010.

J. Nilsson, «The Cost of GUIDs as Primary Keys,» PEARSON, 8 March 2002. [В Интернете]. Available: http://mng.bz/4q49. [Дата обращения: 22 Июнь 2016].

А. Кондуфоров, «Немного о проектировании: паттерны из мира ORM,» [В Интернете]. Available: http://merle-amber.blogspot.ru/2009/02/orm.html. [Дата обращения: .2016 12 7].

Д. Палермо, «Конфигурационная инфраструктура приложения в NHibernate,» в ASP.NET MVC 4 в Действии, Москва, Manning, 2012.

М. Фаулер, Шаблоны корпоративных приложений, Москва: Вильямс, 2010.

В. Вернон, Реализация методов предметно-ориентированного проектирования, Москва: Вильямс, 2016.

Э. Троелсен и Ф. Джепикс, Язык программирования C# 6.0 и платформа .NET 4.6, Москва: Издательсткй дом "Вильямс", 2016.