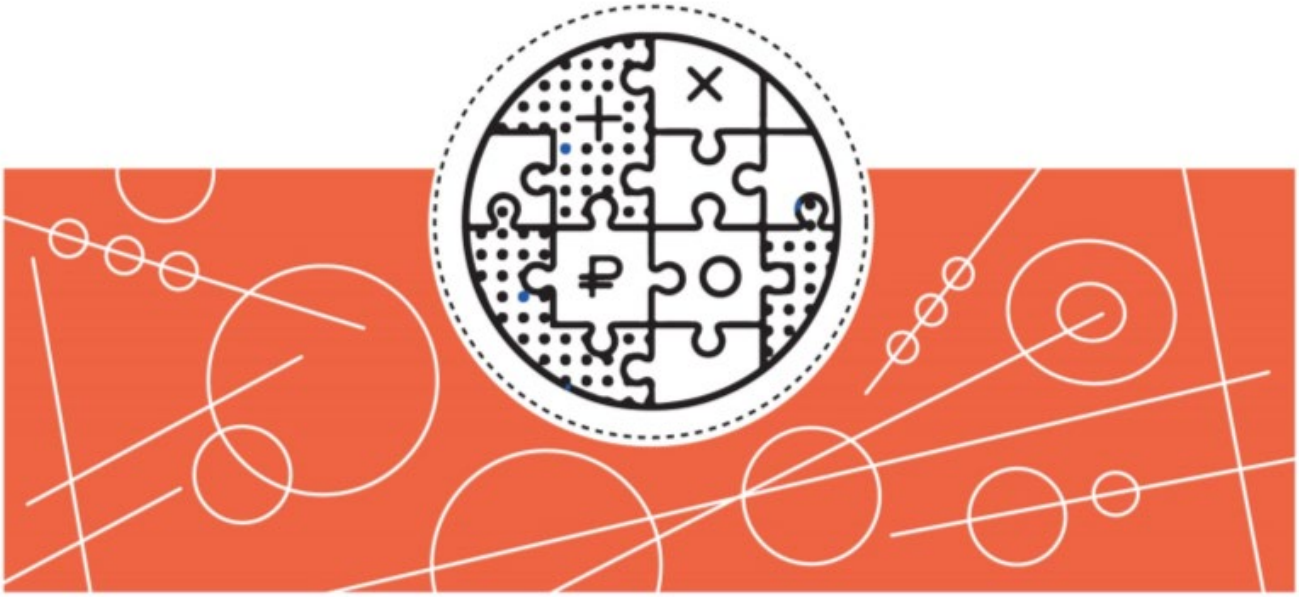
ИТМО

WEB-ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ



Санкт-Петербург
2026

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

WEB-ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

УЧЕБНОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ
В УНИВЕРСИТЕТЕ ИТМО

по направлениям подготовки
09.03.03 Мобильные и сетевые технологии,
45.03.04 Интеллектуальные системы в гуманитарной сфере
в качестве учебного пособия для реализации основных профессиональных образовательных программ высшего образования бакалавриата

Санкт-Петербург
2026

Web- программирование. Лабораторный практикум / Говоров А.И., Коряков С.А., Говорова М.М. [и др.]. – СПб: ИТМО, 2026. – 173 с.

Рецензент(ы):

Шиков Алексей Николаевич, кандидат технических наук, доцент кафедры бизнес-информатики Северо-Западного института управления РАНХиГС

Учебное пособие предназначено для студентов очной формы обучения и посвящено изучению современных средств разработки для создания клиент-серверных web-приложений в рамках освоения дисциплины «Web-программирование». В пособии представлены теоретические и практические материалы, задания к лабораторным работам, что обеспечивает комплексное освоение full-stack разработки: от низкоуровневых сокетов до современных фреймворков, с акцентом на Git-интеграцию и промышленные стандарты.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данное учебное пособие предназначено для студентов, которые изучают современные средства разработки для создания клиент-серверных web-приложений.

В пособии описаны теоретическая часть и практические задания по реализации клиент-серверных приложений с протоколами UDP и TCP, включая отправку и получение сообщений, выполнение математических операций на стороне сервера, отдачу HTML-страницы от сервера клиенту и реализацию многопользовательского чата с использованием многопоточности.

Выполнение работ позволит научиться создавать web-приложения средствами Django и Django REST Framework. Пособие содержит практические задания по реализации серверной части приложения, разработке модели базы данных, разработке API-логики и подключению регистрации/авторизации. Задания помогут студентам научиться использовать мощный инструментарий фреймворка, такой как автоматическая генерация документации API, создание пользовательских представлений и работа с сериализацией данных.

В пособии представлены полезные материалы и ссылки на дополнительные ресурсы, которые помогут студентам в изучении Django и Django REST Framework.

Студенты познакомятся с основами работы с Vue.js, научатся создавать клиентскую часть веб-приложения (включая настройку взаимодействия с серверной частью при помощи REST API), многопользовательские интерфейсы и динамические веб-приложения с использованием Vue.js.

По каждой лабораторной работе описаны этапы выполнения работы и полезные материалы, необходимые для успешного выполнения задания. В конце каждой работы приведены рекомендации по оформлению документации и загрузке работы на GitHub.

ЛАБОРАТОРНАЯ РАБОТА 1. РАБОТА С СОКЕТАМИ

Цель: овладеть практическими навыками и умениями реализации web-серверов и использования сокетов.

Оборудование: компьютерный класс.

Программное обеспечение: Python 2.7–3.6, библиотеки Python: sys, socket.

Теоретические сведения

Клиент-серверное взаимодействие

В основе работы web-приложений лежит так называемая модель взаимодействия клиент-сервер, которая позволяет разделять функционал и вычислительную нагрузку между клиентскими приложениями (заказчиками услуг) и серверными приложениями (поставщиками услуг).

При клиент-серверном взаимодействии участвуют две стороны: клиент (заказчик услуг) и сервер (поставщик услуг). Клиент и сервер физически представляют собой программы. Например, типичным клиентом является браузер. В качестве сервера можно привести следующие примеры: все HTTP-сервера (в частности, Apache), MySQL-сервер, локальный веб-сервер XAMP или готовая сборка Denwer (последних два примера – это не просто сервера, а целый набор серверов).

Клиент и сервер взаимодействуют друг с другом в сети Интернет или в любой другой компьютерной сети при помощи различных сетевых протоколов, например, IP-протоколов, протоколов HTTP, FTP и других. Протоколов на самом деле очень много, и каждый протокол позволяет оказывать ту или иную услугу. Например, при помощи протокола HTTP браузер отправляет специальное HTTP-сообщение, в котором указано, какую информацию и в каком виде он хочет получить от сервера, а сервер, получив такое сообщение, отсылает браузеру в ответ похожее по структуре сообщение (или несколько сообщений), в котором содержится нужная информация (обычно это HTML документ).

Сообщения, которые посылают клиенты, получили название HTTP-запросы. Запросы имеют специальные методы, которые говорят серверу о том, как обрабатывать сообщение, а сообщения, которые посылает сервер, получили название HTTP-ответы – они содержат кроме полезной информации еще и специальные коды состояния, которые позволяют браузеру узнать то, как сервер понял его запрос.

Стоит также заметить, что в основе взаимодействия клиент-сервер лежит принцип того, что такое взаимодействие начинает клиент, сервер лишь отвечает клиенту и сообщает о том, может ли он предоставить услугу клиенту, и если может,

то на каких условиях. Клиентское программное обеспечение и серверное программное обеспечение обычно установлено на разных машинах, но также они могут работать и на одном компьютере.

Данная концепция взаимодействия была разработана в первую очередь для того, чтобы разделить нагрузку между участниками процесса обмена информацией, а также для того, чтобы разделить программный код поставщика и заказчика. На рисунке 1 показана упрощенная схема взаимодействия “клиент-сервер”.

Из рисунка видно, что к одному серверу может обращаться сразу несколько клиентов (действительно, на одном сайте может находиться несколько посетителей). Также стоит заметить, что количество клиентов, которые могут одновременно взаимодействовать с сервером, зависит от мощности сервера и от того, что хочет получить клиент от сервера.

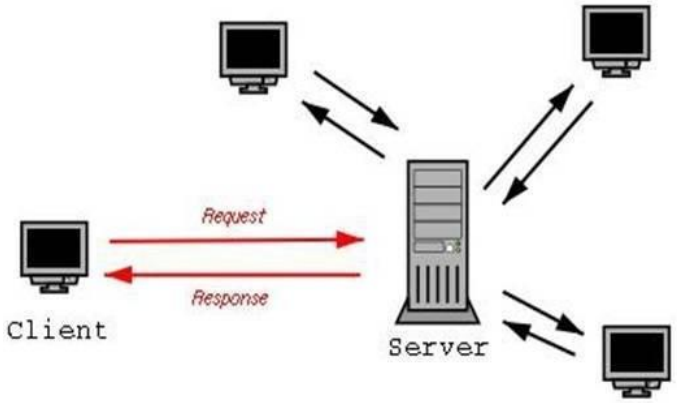


Рисунок 1 – Архитектура «клиент-сервер»: модель взаимодействия клиента и сервера

Многие сетевые протоколы построены на архитектуре клиент-сервер, поэтому в их основе обычно лежат одинаковые или схожие принципы взаимодействия, а разницу можно видеть лишь в деталях, которые обусловлены особенностями и спецификой области, для которой разрабатывался тот или иной сетевой протокол.

Модель TCP/IP

Главной задачей стека TCP/IP является объединение в сеть пакетных подсетей через шлюзы. Каждая сеть работает по своим собственным законам, однако предполагается, что шлюз может принять пакет из другой сети и доставить его по указанному адресу. Пакет из одной сети передается в другую подсеть через

последовательность шлюзов, которые обеспечивают сквозную маршрутизацию пакетов по всей сети. В данном случае под шлюзом понимается точка соединения сетей. При этом соединяться могут как локальные сети, так и глобальные сети. В качестве шлюза могут выступать как специальные устройства, например, маршрутизаторы, так и компьютеры, которые имеют программное обеспечение, выполняющее функции маршрутизации пакетов. Маршрутизация – это процедура определения пути следования пакета из одной сети в другую.

Такой механизм доставки становится возможным благодаря реализации во всех узлах сети протокола межсетевого обмена IP. Любое сообщение, которое отправляется по сети, должно быть при отправке разделено на фрагменты. Каждый из фрагментов должен быть снабжен адресами отправителя и получателя, а также номером этого пакета в последовательности пакетов, составляющих все сообщение в целом. Такая система позволяет на каждом шлюзе выбирать маршрут, основываясь на текущей информации о состоянии сети, что повышает надежность системы в целом. При этом каждый пакет может пройти от отправителя к получателю по своему собственному маршруту. Порядок получения пакетов получателем не имеет большого значения, т.к. каждый пакет несет в себе информацию о своем месте в сообщении.

Порядок распределения уровней и данных в подобной модели показан на рисунке 2.

Рисунок 2 – Модель TCP/IP и распределение информации пакета по уровням

Здесь (рисунок 2) есть пользователь, который хочет отправить другу сообщение: «Привет, Вася!». Этот пользователь открывает почтовый клиент, вводит сообщение и нажимает кнопку «Отправить». Пока пользователь вводит сообщение и пока его обрабатывает почтовый клиент, оно представлено в виде «Пользовательских данных», что является самым верхним уровнем.

С верхних уровней данные попадают на транспортный уровень, этот уровень отвечает за взаимодействие двух конечных узлов, он помогает узлам разделять трафик различных приложений при отправки и при получении, но кроме этого, транспортный уровень делит большие объемы данных, которые пользователи отправляют, на небольшие фрагменты, каждому такому фрагменту на транспортном уровне добавляется специальный заголовок, который нужен для того, чтобы принимающая сторона смогла собрать из мелких сообщений исходные данные. На транспортном уровне также еще нет представления об устройстве сети, так как транспортный уровень создает надежный виртуальный канал поверх ненадежной сети передачи данных, а нижние уровни от него изолированы.

Сообщения транспортного уровня спускаются на сетевой уровень, и к ним добавляется сетевой заголовок. Этот заголовок помогает маршрутизаторам определить путь, по которому будут следовать данные при их передаче из пункта А в пункт Б, то есть здесь появляется представление о логической топологии компьютерной сети, но нет понимания того, как получить физический доступ к ресурсам сети, эта информация сокрыта от сетевого уровня.

С сетевого уровня пакеты попадают на канальный уровень, и к ним добавляются соответствующие заголовки, после чего они становятся кадрами. Канальный уровень дает доступ компьютеру к реальным ресурсам компьютерной сети, то есть определяет интерфейсы и технологии, по которым будет осуществляться передача данных от узла к узлу. Также на канальном уровне реализована функция проверки целостности данных.

После того, как кадр будет сформирован, компьютер превратит его в последовательность нулей и единиц и отправит эту последовательность по линии связи.

Все вышеописанные операции ведутся внутри передающего компьютера, и всё это вместе называется инкапсуляцией данных.

Данные, передаваемые по сети, должны пройти процесс обработки как на передающей стороне, так и на принимающей. Когда передающая сторона готовит данные к передаче – это процесс инкапсуляции данных, и наоборот, когда принимающая сторона начинает обрабатывать входящую последовательность бит и формировать из нее сообщения – это процесс декапсуляции.

В обеих моделях передачи данных на каждом уровне, начиная с транспортного, кроме физического, идет процесс инкапсуляции данных, то есть на

каждом уровне к имеющимся данным добавляется заголовок и при необходимости к данным может быть добавлена метка конца. На транспортном уровне также происходит процесс фрагментации данных, то есть разбиение больших данных, которые отправляет пользователь, на маленькие сообщения, которые удобно передавать по сети. На физическом уровне модели передачи данных, упакованные в несколько слоев сообщения, превращаются в последовательность бит, чтобы затем отправиться в среду передачи данных.

Что содержится в заголовках, которыми упаковываются данные? Это зависит от уровня модели передачи данных и того протокола, чей заголовок добавляется к сообщению. Если говорить в общем, то заголовки содержат служебную информацию, которая помогает устройствам компьютерной сети определить: кому принадлежит данное сообщение, куда его дальше отправить, не повредилось ли сообщение, какое сообщение в полученной последовательности является первым, а какое вторым и многое другое.

Если говорить коротко, то принцип инкапсуляции и декапсуляции данных решает проблему изоляции и разграничения функционала между уровнями модели передачи данных. Это всё возможно благодаря тому, что на каждом уровне данные оборачиваются в дополнительный заголовок, когда они готовятся к передаче, а на приемной стороне эти заголовки снимаются.

Каждый компьютер (он же: узел, хост) в рамках сети Интернет имеет уникальный адрес, который называется IP-адрес (Internet Protocol Address), например: 195.34.32.116. IP- адрес состоит из четырех десятичных чисел (от 0 до 255), разделенных точкой. Но знать только IP-адрес компьютера недостаточно, т.к. в конечном счете обмениваются информацией не компьютеры сами по себе, а приложения, работающие на них. На компьютере может одновременно работать сразу несколько приложений (например, почтовый сервер, веб-сервер и пр.). Для доставки обычного бумажного письма недостаточно знать только адрес дома, необходимо еще знать номер квартиры. Также и каждое программное приложение имеет подобный номер, именуемый номером порта. Большинство серверных приложений имеют стандартные номера, например: почтовый сервис привязан к порту с номером 25, веб-сервис привязан к порту 80, FTP – к порту 21 и так далее.

Можно провести аналогию с обычным почтовым адресом:

- "адрес дома" = "IP компьютера"
- "номер квартиры" = "номер порта"

В компьютерных сетях, работающих по протоколам TCP/IP, аналогом бумажного письма в конверте является пакет, который содержит собственно передаваемые данные и адресную информацию – адрес отправителя и адрес получателя, например:

Адрес отправителя (Source address):

- IP: 82.146.49.55
- Port: 2049

Адрес получателя (Destination address):

- IP: 195.34.32.116
- Port: 53

Данные пакета:

- ...

Комбинация: "IP-адрес и номер порта" называется "сокет".

В примере выше с сокета 82.146.49.55:2049 посылается пакет на сокет 195.34.32.116:53, т.е. пакет адресован компьютеру, имеющему IP адрес 195.34.32.116, на порт 53. А порту 53 соответствует сервер распознавания имен (DNS-сервер), который примет этот пакет. Зная адрес отправителя, этот сервер сможет после обработки запроса сформировать ответный пакет, который пойдет в обратном направлении на сокет отправителя 82.146.49.55:2049, который для DNS сервера будет являться сокетом получателя.

Номера портов на клиенте не фиксируются, как у сервера, а назначаются операционной системой динамически. Фиксированные серверные порты, как правило, имеют номера до 1024 (но есть исключения), а клиентские начинаются после 1024.

Итак, IP – это адрес компьютера (узла, хоста) в сети, а порт – номер конкретного приложения, работающего на этом компьютере.

Однако человеку запоминать цифровые IP-адреса трудно – куда удобнее работать с буквенными именами. Ведь намного легче запомнить слово, чем набор цифр. Поэтому любой цифровой IP-адрес можно связать с буквенно-цифровым именем. В результате, например, вместо 82.146.49.55 можно использовать имя www.ofnet.ru. Преобразованием доменного имени в цифровой IP-адрес занимается сервис доменных имен – DNS (Domain Name System).

Как это работает? Провайдер явно (на бумажке, для ручной настройки соединения) или неявно (через автоматическую настройку соединения) предоставляет клиенту услуг (пользователю) IP- адрес сервера имен (DNS). На компьютере с этим IP адресом работает приложение (сервер имен), которое знает все доменные имена в Интернете и соответствующие им цифровые IP- адреса. DNS-сервер «слушает» 53-й порт, принимает на него запросы и выдает ответы, например:

Запрос от компьютера клиента: "Какой IP адрес соответствует имени www.ofnet.ru?"

Ответ сервера: "82.146.49.55."

Что происходит, когда в своем браузере клиент набирает доменное имя (URL) этого сайта (www.ofnet.ru) и, нажав <enter>, в ответ от веб-сервера получаете страницу этого сайта?

Например:

IP-адрес компьютера клиента: 91.76.65.216.

Браузер: Internet Explorer (IE).

DNS-сервер (стрима): 195.34.32.116 (у клиента может быть другой). Страница, которую хочет открыть клиент: www.ofnet.ru.

Если в адресной строке браузера набрать доменное имя www.ofnet.ru и нажать <enter>, далее операционная система производит примерно следующие действия:

Отправляется запрос (точнее пакет с запросом) DNS серверу на сокет 195.34.32.116:53. Как было рассмотрено выше, порт 53 соответствует DNS-серверу - приложению, занимающемуся распознаванием имен. DNS-сервер, обработав запрос, возвращает IP-адрес, который соответствует введенному имени.

Диалог примерно следующий:

- Какой IP-адрес соответствует имени www.ofnet.ru?
- 82.146.49.55.

Далее компьютер клиента устанавливает соединение с портом 80 компьютера 82.146.49.55 и посылает запрос (пакет с запросом) на получение страницы www.ofnet.ru. 80-й порт соответствует веб-серверу. В адресной строке браузера 80-й порт как правило не пишется, т.к. используется по умолчанию, но его можно и явно указать после двоеточия – http://www.ofnet.ru:80.

Приняв запрос от клиента, веб-сервер обрабатывает его и в нескольких пакетах посылает страницу на языке HTML - языке разметки текста, который понимает браузер.

Браузер, получив страницу, отображает ее. В результате клиент (пользователь) видит на экране главную страницу этого сайта.

Протокол HTTP

Веб-браузеры взаимодействуют с веб-серверами при помощи протокола передачи гипертекста (HTTP). Когда пользователь кликает на ссылку на странице, заполняет форму или запускает поиск, браузер отправляет на сервер HTTP-запрос.

Этот запрос включает:

- Путь, определяющий целевой сервер и ресурс (например, файл, определенная точка данных на сервере, запускаемый сервис, и т.д.).
- Метод, который определяет необходимое действие (например, получить файл, сохранить или обновить некоторые данные, и т.д.). Различные методы/команды и связанные с ними действия перечислены ниже:
  - GET – получить определенный ресурс (например, HTML-файл, содержащий информацию о товаре или список товаров).
  - POST – создать новый ресурс (например, новую статью на Википедии, добавить новый контакт в базу данных).
  - HEAD – получить метаданные об определенном ресурсе без получения содержания, как делает GET. Вы, например, можете использовать запрос HEAD, чтобы узнать, когда в последний раз ресурс обновлялся и только потом использовать (более «затратный») запрос GET, чтобы загрузить ресурс, который был изменен.
  - PUT – обновить существующий ресурс (или создать новый, если таковой не существует).
  - DELETE – удалить указанный ресурс.
  - TRACE, OPTIONS, CONNECT, PATCH – эти команды используются для менее популярных/продвинутых задач, поэтому мы их не будем рассматривать.

Дополнительная информация может быть зашифрована в запросе (например, данные формы). Информация может быть зашифрована как:

- Параметры URL: запросы зашифровывают данные в адресную строку, отправляемую на сервер, добавляя пары имя/значение в его конец, например, http://mysite.com?name=Fred&age=11. У вас всегда есть знак вопроса (?), отделяющий прочую часть от параметров, знак равно (=), отделяющий каждое имя от соответствующего ему значения, и амперсанд (&), разделяющий пары. Параметры по своей сути «небезопасны», так как они могут быть изменены пользователями и затем отправлены заново. В результате параметры /запросы не используются для запросов, которые обновляют данные на сервере.
- POST данные. POST запросы добавляют новые ресурсы, данные которых зашифрованы в теле запроса.
- Куки-файлы клиентской части. Куки-файлы содержат данные сессий о клиенте, включая ключевые слова, которые сервер может использовать для определения его авторизационный статус и права доступа к ресурсам.

Веб-серверы ожидают сообщений с запросами от клиентов, обрабатывают их, когда они приходят и отвечают веб-браузеру через сообщение с HTTP-ответом. Ответ содержит Код статуса HTTP-ответа, который показывает, был ли запрос успешным (например, «200 OK» означает успех, «404 Not Found» если ресурс не может быть найден, «403 Forbidden», если пользователь не имеет права просматривать ресурс, и т.д.). Тело успешного ответа на запрос GET будет содержать запрашиваемый ресурс.

После того, как HTML страница была возвращена, она обрабатывается браузером. Далее браузер может исследовать ссылки на другие ресурсы (например, HTML-страница обычно использует JavaScript и CSS файлы), и послать отдельный HTTP-запрос на загрузку этих файлов.

Как статические, так и динамические веб-сайты (обсуждаемые в следующих разделах) используют точно такой же протокол / шаблоны связи.

Пользователь может сформировать простой GET-запрос кликнув по ссылке или через поиск по сайту (например, страница механизма поиска). Например, HTTP-запрос, посланный во время выполнения запроса "client server overview" на сайте MDN, будет во многом похож на текст ниже (он не будет идентичным, потому что части сообщение зависят от браузера\настроек).

Пример запроса / ответа GET

Запрос

Каждая строка запроса содержит информацию о запросе. Первая часть называется заголовок, он содержит важную информацию о запросе, точно также как HTML-head содержит важную информацию о HTML документе (но не содержимое документа, которое расположено в body):

```
GET https://developer.mozilla.org/en-
US/search?q=client+server+overview&topic=apps&topic=html&t
opic=css&topic=js&topic=api&topic=webdev HTTP/1.1
Host: developer.mozilla.org
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/52.0.2743.116 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,imag
e/webp,*/*;q=0.8
```

```
Referer: https://developer.mozilla.org/en-US/
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie;
csrftoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT;
dwf_section_edit=False; dwf_sg_task_completion=False;
_gat=1; _ga=GA1.2.1688886003.1471911953; ffo=true
```

Первая и вторая строки содержат большую часть информации, о которой говорилось выше:

- Тип запроса (GET).
- URL целевого ресурса (/en-US/search).
- Параметры URL (q=client%2Bserver%2Boverview&topic=apps&topic=html&topic=css&topic=js&topic=api&topic=webdev).
- Целевой вебсайт (developer.mozilla.org).
- Конец первой строки так же содержит короткую строку, идентифицирующую версию протокола (HTTP/1.1).

Последняя строка содержит информацию о клиентских куки – в данном случае можно увидеть куки, включающие id для управления сессиями (Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie; ...).

Оставшиеся строки содержат информацию о используемом браузере и о его некоторых поддерживаемых возможностях. Например, здесь можно увидеть:

- Мой браузер (User-Agent) Mozilla Firefox (Mozilla/5.0).
- Он может принимать информацию, упакованную gzip (Accept-Encoding: gzip).
- Он может принимать указанные кодировки (Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7) и языков (Accept-Language: de,en;q=0.7,en-us;q=0.3).
- Строка Referer идентифицирует адрес веб страницы, содержащей ссылку на этот ресурс (т.е. источник оригинального запроса, https://developer.mozilla.org/en-US/).

HTTP-запрос также может содержать body, но в данном случае оно пусто.

Ответ

Первая часть ответа на запрос показана ниже. Заголовок содержит информацию, описанную ниже:

Первая строка содержит код ответа 200 OK, говорящий о том, что запрос выполнен успешно.

Можно видеть, что ответ имеет text/html формат (Content-Type).

Так видно, что ответ использует кодировку UTF-8 (Content-Type: text/html; charset=utf-8).

Заголовок так же содержит размер ответа (Content-Length: 41823).

В конце сообщения можно видеть содержимое body, содержащее HTML-код возвращаемого ответа.

```
HTTP/1.1 200 OK
Server: Apache
X-Backend-Server: developer1.webapp.scl3.mozilla.com
Vary: Accept,Cookie, Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:11:31 GMT
Keep-Alive: timeout=5, max=999
Connection: Keep-Alive
X-Frame-Options: DENY
Allow: GET
X-Cache-Info: caching
Content-Length: 41823

<!DOCTYPE html>
<html lang="en-US" dir="ltr" class="redesign no-js" data-
ffo-opensanslight=false data-ffo-opensans=false >
<head prefix="og: http://ogp.me/ns#">
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=Edge">
  <script>(function(d) { d.className =
d.className.replace(/\bno-js/, '');
})(document.documentElement);</script>
  ...
```

Остальная часть заголовка ответа содержит информацию об ответе (например, когда он был сгенерирован), сервере и о том, как он ожидает, что браузер обрабатывает страницу (например, строка X-Frame-Options: DENY говорит браузеру не допускать этого страницу, которая будет внедрена в <iframe> на другом сайте).

Пример запроса / ответа POST

HTTP POST создается, когда отправляется форма, содержащая информацию, которая должна быть сохранена на сервере.

Запрос

В приведенном ниже тексте показан HTTP-запрос, сделанный, когда пользователь представляет новые данные профиля на этом сайте. Формат запроса почти такой же, как пример запроса GET, показанный ранее, хотя первая строка идентифицирует этот запрос как POST.

```
POST https://developer.mozilla.org/en-
US/profiles/hamishwillee/edit HTTP/1.1
Host: developer.mozilla.org
Connection: keep-alive
Content-Length: 432
Pragma: no-cache
Cache-Control: no-cache
Origin: https://developer.mozilla.org
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/52.0.2743.116 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,imag
e/webp,*/*;q=0.8
Referer: https://developer.mozilla.org/en-
US/profiles/hamishwillee/edit
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie;
_gat=1; csrftoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT;
dwf_section_edit=False; dwf_sg_task_completion=False;
_ga=GA1.2.1688886003.1471911953; ffo=true


csrfmiddlewaretoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT&user-
username=hamishwillee&user-fullname=Hamish+Willee&user-
title=&user-organization=&user-location=Australia&user-
locale=en-US&user-timezone=Australia%2FMelbourne&user-
irc_nickname=&user-interests=&user-expertise=&user-
```

```
twitter_url=&user-stackoverflow_url=&user-
linkedin_url=&user-mozillians_url=&user-facebook_url=
```

Основное различие заключается в том, что URL-адрес не имеет параметров. Как можно видеть, информация из формы закодирована в теле запроса (например, новое полное имя пользователя устанавливается с использованием: &user-fullname=Hamish+Willee).

Ответ

Ответ от запроса показан ниже. Код состояния «302 Found» сообщает браузеру, что сообщение удалось, и что он должен выдать второй HTTP-запрос для загрузки страницы, указанной в поле «Место». В противном случае информация аналогична информации для ответа на запрос GET.

```
HTTP/1.1 302 FOUND
Server: Apache
X-Backend-Server: developer3.webapp.scl3.mozilla.com
Vary: Cookie
Vary: Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:38:13 GMT
Location: https://developer.mozilla.org/en-
US/profiles/hamishwillee
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
X-Frame-Options: DENY
X-Cache-Info: not cacheable; request wasn't a GET or HEAD
Content-Length: 0
```

Сокеты в Python 3: TCP, клиент, сервер

Язык Python имеет внутри себя множество библиотек практической направленности, как и заранее интегрированные в язык, так и распространяемые через PyPI. Одной из подобных библиотек, входящих в стандартный набор языка Python, является socket – она позволяет возвращать сокет как объект для работы с соединениями и впоследствии реализовывать клиентскую и серверную часть сокет-приложения.

Сокет – это виртуальная конструкция из IP-адреса и номера порта, позволяющая создавать прямые соединение между сервером и клиентом [1].

В рамках создания подобных подключений используются протоколы транспортного уровня модели OSI - UDP и TCP. Это специальные протоколы, разработанные для передачи данных – основная их разница в предназначении: UDP обеспечивает скорость, а TCP целостность и безопасность данных.

На основе этого разделения можно выделить два вида приложений на сокетах:

- Потоковые – приложения, использующие протокол TCP, основывающиеся на двунаправленном потоке байтов, принимающие и отправляющие данные одновременно.
- Дейтаграммные – приложения, использующие UDP-протокол и не требующие какого-то явного соединения между клиентом и сервером.

Используя подобную библиотеку, можно реализовать простейшее-клиент серверное приложение, которое будет принимать некоторое количество байт сообщения, декодировать его и выводить в консоль сервера. Пример кода для сервера, принимающего сообщение через библиотеку socket посредством протокола UDP и выводящего его на экран представлен ниже:

```
import socket

conn = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
conn.bind(("127.0.0.1", 14900))
data = conn.recv(16384)
udata = data.decode("utf-8")
print("Data: " + udata)
conn.close()
```

Объект conn получается из вызова внутреннего конструктора socket, принимающего 2 параметра – семейство сокетов (в рассматриваемом случае протокол IPv4 – AF_INET) и протокол транспортного уровня (SOCK_DGRAM – UDP протокол). После инициализации объекта происходит привязка (bind) к объекту самого сокета – комбинации адреса и порта. В объект data приходят двоичные данные через метод recv. Как аргумент указывается количество байт, которое можно единоразово принять. Объект udata декорирует байты в строку, затем строка выводится.

Код клиента для передачи данных по сокету через протокол UDP аналогичен серверному, только вместо привязки к сокету происходит присоединение к серверу (по адресу сокета) и отправка данных:

```
import socket

conn = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
conn.connect(("127.0.0.1", 14900))
conn.send(b"Hello! \n")
```

Как и в случае сервера создается объект соединения conn, но затем вызывается метод connect, где передается кортеж, содержащий характеристики

сокета. Они нужны для нахождения конечной точки отправки сообщения. Метод send у созданного объекта позволяет отправить байтовую строку на сервер.

Для того чтобы протестировать приложение, необходимо запустить два файла в разных сеансах командной строки, соответствующие клиенту и серверу, описанному выше. Пусть они будут называться client.py и server.py соответственно. На рисунке 3 изображен результат работы сервера после запуска клиента:

```
(venv) C:\Users\123\PycharmProjects\firstproj\sockets>server.py
Data: Hello!
```

Рисунок 3 – Пример исполнения клиент-серверного приложения на сокетах

Подобно приложению через UDP-протокол можно реализовать сервер и клиент через протокол TCP. Хоть по структуре они схожи, но есть несколько важных отличий. TCP-соединение требует реализацию протокола с явным подключением между двумя сторонами, сервер не может как в случае с UDP принимать любые данные. В рамках подобных ограничений код сервера может выглядеть следующим образом:

```
import socket

conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn.bind(("127.0.0.1", 14900))
conn.listen(10)

while True:
    try:
        clientsocket, address = conn.accept()
        data = clientsocket.recv(16384)
        udata = data.decode("utf-8")
        print("Data: " + udata)
    except KeyboardInterrupt:
        conn.close()
        break
```

При создании объекта conn во второй параметр передается информация о протоколе TCP – SOCK_STREAM. После этого сокет привязывается к объекту, как и в предыдущем примере. С помощью метода listen объект сокета переходит в режим приема TCP-соединений, где аргумент метода – максимальное кол-во таких соединений. Для обработки подключений к серверу необходимо войти в бесконечный цикл, который будет останавливаться вместе с сокетом только при использовании специального сочетания клавиш. Иначе, сервер будет принимать по очереди все входящие соединения и блокировать приложения в ожидании

ответа через метод accept. После получения ответа от входящего соединения все выполняемые действия аналогичны примеру с UDP-сервером.

В случае TCP-клиента разница заключается в смене протокола для подключения (на TCP) и закрытия соединения после выполнения приложения – это необходимо для того, чтобы не блокировать сервер:

```
     import socket

conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn.connect(("127.0.0.1", 14900))
conn.send(b"Hello! \n")
conn.close()
```

Аналогично UDP-приложению можно запустить серверный и клиентский файл для TCP-сокета и проверить его работоспособность. Пример выполнения такого приложения изображен на рисунке 4.

```
(venv) C:\Users\123\PycharmProjects\firstproj\sockets>server.py
Data: Hello!

Data: Hello!
```

Рисунок 4 – Выполнение приложения на сокетах через протокол TCP

В отличие от предыдущего примера, данный сервер не заканчивает свою работу после закрытия подключения, а ждет новые. Это было продемонстрировано на рисунке выше: клиент был запущен 2 раза и оба раза сервер ответил на запрос.

Библиотека также может обрабатывать TCP-запросы без включения режима блокировки основного потока. Это позволяет обрабатывать запросы от нескольких пользователей:

```
import socket
import time

conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
conn.bind(("127.0.0.1", 14900))
conn.listen(10)
conn.setblocking(False)

while True:
    try:
        clientsocket, address = conn.accept()
        conn.setblocking(True)
        data = clientsocket.recv(16384)
```

```
        udata = data.decode("utf-8")
        print("Data: " + udata)
        conn.shutdown(socket.SHUT_WR)
    except socket.error:
        print('waiting clients')
        time.sleep(1)
    except KeyboardInterrupt:
        conn.close()
        break
```

Основная разница с предыдущим примером состоит в том, что первое подключение в данном коде принимается без блокировки потока – за это отвечает метод setblocking. Если в режиме отсутствия блокировки не будут приходить данные, будет вызываться ошибка семейства socket.error, означающая, что ответ от пользователя все еще не пришел. Во время обработки такого исключения будет выведено соответствующее сообщение. Во всем остальном функционал повторяет предыдущий пример TCP-сервера.

Аналогичным решением для реализации TCP-сервера в рамках объектно-ориентированного программирования является стандартная библиотека socketserver. В такой реализации серверного приложения создается класс, наследуется от базового класса библиотеки, переопределяется функция обработчик, и производится запуск сервера через создание специального объекта - комбинации сокета и собственного класса:

```
from socketserver import *

class MyTCPHandler(StreamRequestHandler):

    def handle(self):
        self.data = self.request.recv(1024)
        print('client send: ' + self.data.decode("utf-8"))
        self.request.sendall(b'Hello from server!')


if __name__ == "__main__":
    server = TCPServer(('localhost', 14900), MyTCPHandler)
    print('starting server... for exit press Ctrl+C')
    server.serve_forever()
```

Класс MyTCPHandler, наследуемый от поставляемого библиотекой класса StreamRequestHandler, переопределяет метод handle, в котором через внутренний объекты базового класса происходит получение данных с клиента и отправка данных ему же. Для реализации такого сервера используется конструктор класса TCPServer, в который передается сокет и ссылка на переопределенный класс.

Посредством вызова метода serve_forever у созданного объекта происходит запуск сервера в бесконечном цикле.

Клиент для подобного сервера написан на библиотеке socket, но теперь не только отправляет данные, но и принимает от сервера:

```
from socket import *
import sys


tcp_socket = socket(AF_INET, SOCK_STREAM)
tcp_socket.connect(("127.0.0.1", 14900))


data = input('write to server: ')
if not data:
    tcp_socket.close()
    sys.exit(1)


data = str.encode(data)
tcp_socket.send(data)
data = tcp_socket.recv(1024)
print(data.decode("utf-8"))


tcp_socket.close()
```

Аналогично серверу при создании объекта сокета используется метод *recv*, позволяющий получить поток байт от исходящей стороны. В случае данного клиентского приложения возвращается ответ от серверного приложения, написанного с помощью socketserver.

На рисунке 5 показаны командные строки сервера и клиента при вызове соответствующих файлов.

```
(venv) C:\Users\123\PycharmProjects\firstproj\sockets>server.py
starting server... for exit press Ctrl+C
client send: sad
```

```
(venv) C:\Users\123\PycharmProjects\firstproj\sockets>client.py
write to server: sad
Hello from server!
```

Рисунок 5 – Пример выполнения сокет-приложения, написанное средствами ООП

В случае реализации TCP-сервера через socket сервер не завершает свою работу после ответа клиенту, а находится в бесконечном цикле.

Еще одним использованием библиотеки socket может быть создание http-сервера. Для этого достаточно возвращать клиенту (браузеру) информацию в формате http, чтобы данные могли корректно быть обработаны и выведены. Пример кода, отвечающий http-страницей на запрос к серверу, представлен ниже:

```
import socket


server = socket.socket()
host = '127.0.0.1'
port = 555
server.bind((host, port))

print('Starting server on', host, port)
print('The Web server URL for this would be http://%s:%d/' % (host,
port))

server.listen(5)

print('Entering infinite loop; hit CTRL-C to exit')
while True:
    client, (client_host, client_port) = server.accept()
    print('Got connection from', client_host, client_port)
    client.recv(1000)
    response_type = 'HTTP/1.0 200 OK\n'
    headers = 'Content-Type: text/html\n\n'
    body = """
        <html>
        <body>
        <h1>Hello World</h1>!
        </body>
        </html>
    """
    response = response_type + headers + body
    client.send(response.encode('utf-8'))
    client.close()
```

Таким же образом, как и в предыдущих примерах, сервер инициализирует сокет через метод bind, разрешает 5 одновременных доступных подключений посредством метода listen и запускает бесконечный цикл, который возвращает клиенту ответ, воспринимающийся браузером как html-страница. Если перейти по адресу сервера в браузере (http://127.0.0.1:555/), будет выведена html-страница, как показано на рисунке 6.

Рисунок 6 – Пример запуска http-сервера на сокетах

Лабораторная часть

Практическое задание

1. Реализовать клиентскую и серверную часть приложения. Клиент отсылает серверу сообщение «Hello, server». Сообщение должно отразиться на стороне сервера. Сервер в ответ отсылает клиенту сообщение «Hello, client». Сообщение должно отобразиться у клиента.

Указания:

– При выполнении обязательно использовать библиотеку `socket.`

– Реализовать с помощью протокола UDP.

2. Реализовать клиентскую и серверную часть приложения. Клиент запрашивает у сервера выполнение математической операции, параметры, которые вводятся с клавиатуры. Сервер обрабатывает полученные данные и возвращает результат клиенту.

Варианты:

1. Теорема Пифагора.
2. Решение квадратного уравнения.
3. Поиск площади трапеции.
4. Поиск площади параллелограмма.

Указания:

– Обязательно использовать библиотеку `socket.`

– Реализовать с помощью протокола TCP.

3. Реализовать серверную часть приложения. Клиент подключается к серверу. В ответ клиент получает http-сообщение, содержащее html-страницу, которую сервер подгружает из файла index.html.

Указания:

– Обязательно использовать библиотеку `socket`.

4. Реализовать двупользовательский или многопользовательский чат. Реализация многопользовательского часа позволяет получить максимальное количество баллов.

Указания:

– Обязательно использовать библиотеку `socket`.

– Обязательно использовать библиотеку `threading`.

Для реализации с помощью UDP thearding использовать для получения сообщений у клиента.

Для применения с TCP необходимо запускать клиентские подключения, прием и отправку сообщений всем юзерам на сервере в потоках (с помощью библиотеки `threading`). Не забудьте сохранять юзеров, чтобы потом отправлять им сообщения.

5. Необходимо написать простой web-сервер для обработки GET и POST http-запросов средствами Python и библиотеки socket, который может:
   - Принять и записать информацию о дисциплине и оценке по дисциплине.
   - Отдать информацию обо всех оценках по дисциплине в виде html-страницы.

Базовый класс для простейшей реализации web-сервера представлен ниже:

```
import socket
import sys

class MyHTTPServer:
  # Параметры сервера

  def serve_forever(self):
    # 1. Запуск сервера на сокете, обработка входящих соединений

  def serve_client(self, *):
    # 2. Обработка клиентского подключения

  def parse_request(self, *):
    # 3. функция для обработки заголовка http+запроса. Python, сокет
предоставляет возможность создать вокруг него некоторую обертку, которая
предоставляет file object интерфейс. Это дайте возможность построчно
обработать запрос. Заголовок всегда - первая строка. Первую строку нужно
разбить на 3 элемента (метод + url + версия протокола). URL необходимо
```

```
разбить на адрес и параметры (isu.ifmo.ru/pls/apex/f?p=2143 , где
isu.ifmo.ru/pls/apex/f, а p=2143 - параметр p со значением 2143)

  def parse_headers(self, *):
    # 4. Функция для обработки headers. Необходимо прочитать все
заголовки после первой строки до появления пустой строки и сохранить их в
массив.

  def handle_request(self, *):
    # 5. Функция для обработки url в соответствии с нужным методом. В
случае данной работы, нужно будет создать набор условий, который
обрабатывает GET или POST запрос. GET запрос должен возвращать данные.
POST запрос должен записывать данные на основе переданных параметров.

  def send_response(self, *):
    # 6. Функция для отправки ответа. Необходимо записать в соединение
status line вида HTTP/1.1 <status_code> <reason>. Затем, построчно
записать заголовки и пустую строку, обозначающую конец секции заголовков.

if __name__== '__main__':
  host = *
  port = *
  name = *
  serv = MyHTTPServer(host, port, name)
  try:
    serv.serve_forever()
  except KeyboardInterrupt:
    pass
```

Подробный мануал по работе доступен в [2].

Порядок выполнения и защиты работы

1. Работа выполняется индивидуально.
2. Необходимо выполнить задания 1 – 5 из пункта «Лабораторная часть».
3. По результатам работы необходимо подготовить документацию средствами MkDocs (Описание работы с MkDocs доступно в практической работе 3.3). По каждому пункту работы необходимо следующее описание полученной разработанной программы:
   a) Описание задания.
   b) Входные и выходные данные программ.
   c) Описание структуры разработанных файлов программ.

4. Полученную программу залить в форк студента репозитория группы в папку students/группа/laboratory_works/фамилия_имя/laboratiry_work_4. Инструкция о загрузке работы ниже в пункте “Загрузка работы на GitHub”. Необходимо сделать пулреквест в основной репозиторий группы. Ссылку на документацию оставить в комментариях.

Загрузка работы на GitHub

Для сдачи работы в связи необходимо загрузить работы на GitHub. Все студенческие работы хранятся в папке Students. Для сдачи работы необходимо:

1. Зарегистрироваться на https://github.com/.
2. Сделать форк (копию) репозитория с заданиями в личный аккаунт (на странице репозитория дисциплины кнопка fork справа, сверху).
3. Установить Git на компьютер.
4. Открыть папку, где хранятся личные проекты. В контекстном меню нажать «Open Git Bash here». Склонировать форкнутый репозиторий на комьютер (git clone https://github.com/Личный_аккаунт/репозиторий дисциплины).
5. В файловой системе личного компрьютера в склонированном репозитории создать в папке students/группа личную папку в формате Фамилия_Имя латиницей (например: students/k3340/Petrov_Vasya).
6. В личной папке сделать подпапку с текущей работой в формате Lr_номер (Пример: students/k3340/Petrov_Vasya/Lr_1).
7. Записать в папку отчетные материалы.
8. Сделать коммит, описать его в комментарии (например: «Был добавлен файл Презентация_Петров.pdf»). Выполнить команлы git add и git commit -m «название комита».
9. Сделать push в личный форкнутый репозиторий (git push).
10. Сделать пул-реквест в репозиторий преподавателя из личного форкнутого, описать его. Структура заголовка пулреквеста: Фамилия_Имя-Работа_Номер. (Пример: Петров_Василий_Лабораторная_работа_1).

ЛАБОРАТОРНАЯ РАБОТА 2. ИЗУЧЕНИЕ DJANGO WEB FRAEMWORK

Цель: овладеть практическими навыками и умениями реализации web-сервисов средствами Django 2.2.

Оборудование: компьютерный класс.

Программное обеспечение: Python 3.6+, Django 3, PostgreSQL *.

Практикум 2.1 Django Web framework. Установка. Реализация первого приложения

2.1.1 Установка Django Web framework

Установка возможна двумя способами:

1. Средствами командной строки в виртуальном окружении Python.

2. С помощью IDE PyCharm.

Установка Django Web framework средствами командной строки в виртуальном окружении Python

1. Используя командную строку, создать папку для проекта и перейти в нее.

2. Создать среду окружения (имя среды можно задать произвольно - venv):

`py -m venv venv`.

В результате создается каталог `venv` и также директории внутри него, содержащие копию интерпретатора Python, стандартную библиотеку и различные вспомогательные файлы.

При создании проекта в PyCharm среда окружения называется `venv`.

3. Активировать виртуальную среду командой:

`venv\Scripts\activate` (Win) или `source venv/bin/activate` (Linux).

Пример (для Windows):

```
C:\Users\123\PycharmProjects\warriors_project>py -m venv venv

C:\Users\123\PycharmProjects\warriors_project>venv\Scripts\activate

(venv) C:\Users\123\PycharmProjects\warriors_project>
```

Пример (для Linux):

```
root@owo:~/warriors_project# python3 -m venv venv
root@owo:~/warriors_project# source venv/bin/activate
(venv) root@owo:~/warriors_project#
```

В результате изменится приглашение оболочки, показывая, что используется виртуальная среда (в командной строке это отображается как название среды в скобках перед путем к проекту). Для проверки версии интерпретатора и просмотра файлов в окружении можно перейти в интерактивный режим, вызвав следующие команды:

```
(venv) C:\Users\123\PycharmProjects\warriors_project>python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40) [MSC v.1927 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['',
'C:\\Users\\123\\AppData\\Local\\Programs\\Python\\Python39\\python39.zip
', 'C:\\Users\\123\\AppData\\Local\\Programs\\Python\\Python39\\DLLs', …]
```

Выход из Интерактивного режима осуществляется через Ctrl+Z. (KeyboardInterrupt).

Примечание: если необходимо использовать какую-то определенную версию Python, то это нужно явно указывать при создании окружения, иначе будет создаваться окружение с последней установленной версией (для Windows это последняя версия, указанная в PATH).

Пример (Windows):

```
C:\Users\123\PycharmProjects\warriors_project>py -3.7 -m venv venv
C:\Users\123\PycharmProjects\warriors_project>venv\Scripts\activate

(venv2) C:\Users\123\PycharmProjects\warriors_project>python -
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information
```

Пример (Linux):

```
root@owo:~/warriors_project# python3.7 -m venv venv2
root@owo:~/warriors_project# source venv/bin/activate

(venv) root@owo:~/warriors_project# python
Python 3.7.9 (default, Aug 18 2020, 06:22:45)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

4. Установить Django.

При создании виртуального окружения автоматически должен был установится пакетный менеджер pip, позволяющий устанавливать библиотеки из индекса пакетов PyPi [3]. С помощью pip устанавливается фреймворк:

```
(venv) C:\Users\123\PycharmProjects\warriors_project>pip install django


Collecting django
  Downloading Django-4.1.2-py3-none-any.whl (8.1 MB)
     |████████████████████████████████| 8.1 MB 1.3 MB/s
Collecting tzdata; sys_platform == "win32"
  Downloading tzdata-2022.5-py2.py3-none-any.whl (336 kB)
     |████████████████████████████████| 336 kB 2.2 MB/s
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.3-py3-none-any.whl (42 kB)
Collecting asgiref<4,>=3.5.2
  Using cached asgiref-3.5.2-py3-none-any.whl (22 kB)

Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.5.2 django-4.1.2 sqlparse-0.4.3 tzdata-
2022.5
```

5. Создать проект Django.

Для того чтобы инициализировать новый проект, достаточно написать команду `django-admin startproject` с его названием:

```
> django-admin startproject warriors_app
```

6. Перейти в папку проекта.

Django автоматически создаст папку проекта с основными папками, файлами управления и настроек приложения. Посмотреть их можно, перейдя в созданную папку приложения:

```
(venv) C:\Users\123\PycharmProjects\warriors_project>cd warriors_app

(venv) C:\Users\123\PycharmProjects\warriors_project\warriors_app>dir
 Том в устройстве C не имеет метки.
 Серийный номер тома: 360F-30F0

 Содержимое папки
C:\Users\123\PycharmProjects\warriors_project\warriors_app

14.10.2022  16:38    <DIR>          .
14.10.2022  16:38    <DIR>          ..
```

```
14.10.2022  16:38               690 manage.py
14.10.2022  16:38    <DIR>          warriors_app
               1 файлов            690 байт
               3 папок  52 063 518 720 байт свободно
```

7. Создать приложение.

Приложение в Django – структурный элемент проекта, отделяющий какую-то логику сервера через файловую систему и наименование в БД. Таких приложений можно создать более одного, все зависит от архитектуры проекта. Подробнее о файловой структуре проекта будет рассказано в следующем разделе.

Для того чтобы создать новое приложение, нужно с помощью управляющего файла `manage.py` написать следующую команду:

```
(venv)
C:\Users\123\PycharmProjects\warriors_project\warriors_app>manage.py
startapp warriors

(venv) C:\Users\123\PycharmProjects\warriors_project\warriors_app>cd
warriors

(venv)
C:\Users\123\PycharmProjects\warriors_project\warriors_app\warriors>dir
Том в устройстве C не имеет метки.
 Серийный номер тома: 360F-30F0


 Содержимое папки
C:\Users\123\PycharmProjects\warriors_project\warriors_app\warriors

14.10.2022  17:22    <DIR>          .
14.10.2022  17:22    <DIR>          ..
14.10.2022  17:22                66 admin.py
14.10.2022  17:22               154 apps.py
14.10.2022  17:22    <DIR>          migrations
14.10.2022  17:22                60 models.py
14.10.2022  17:22                63 tests.py
14.10.2022  17:22                66 views.py
14.10.2022  17:22                 0 __init__.py
               6 файлов            409 байт
               3 папок  52 046 000 128 байт свободно
```

Установка Средствами PyCharm Professional Edition

Инструкции, описанные выше, можно применить и в командной строке Pycharm, создав обычный проект Python (для этого не обязательно иметь Professional Edition версию).

Ниже показан пример, как можно установить приложение, не используя командную строку:

1. Открыть папку в PyCharm.

Для этого запустить PyCharm и открыть папку проекта.

2. Создать проект можно непосредственно в PyCharm. Для этого выбрать File->New Project, выбрать тип проекта Django и папку проекта. После этого откроется окно проекта (рисунок 7).

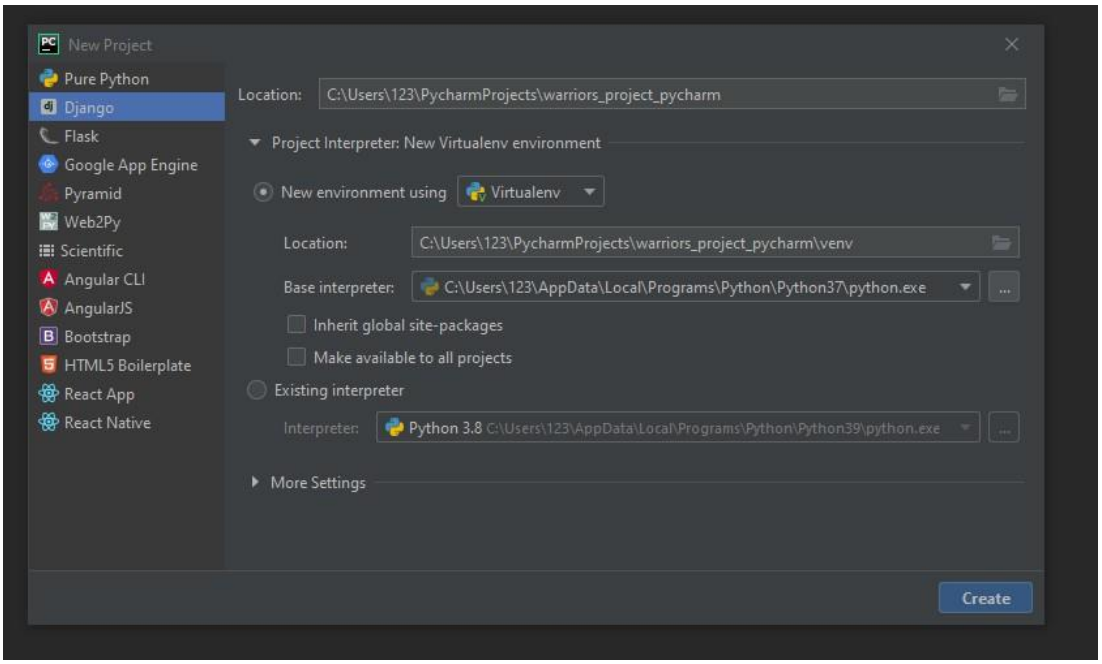


Рисунок 7 – Окно создание проекта

Виртуальное окружение создается по умолчанию, но при желании можно использовать уже существующее окружение (не создавать его в разделе Project Interpreter).

3. Папка проекта (warriors_project_pycharm) является контейнером файлов всего проекта.

4. После открытия или создания проекта можно увидеть, что папка проекта содержит дочернюю папку с тем же именем. Можно считать, что эта дочерняя папка содержит настройки проекта (рисунок 8).

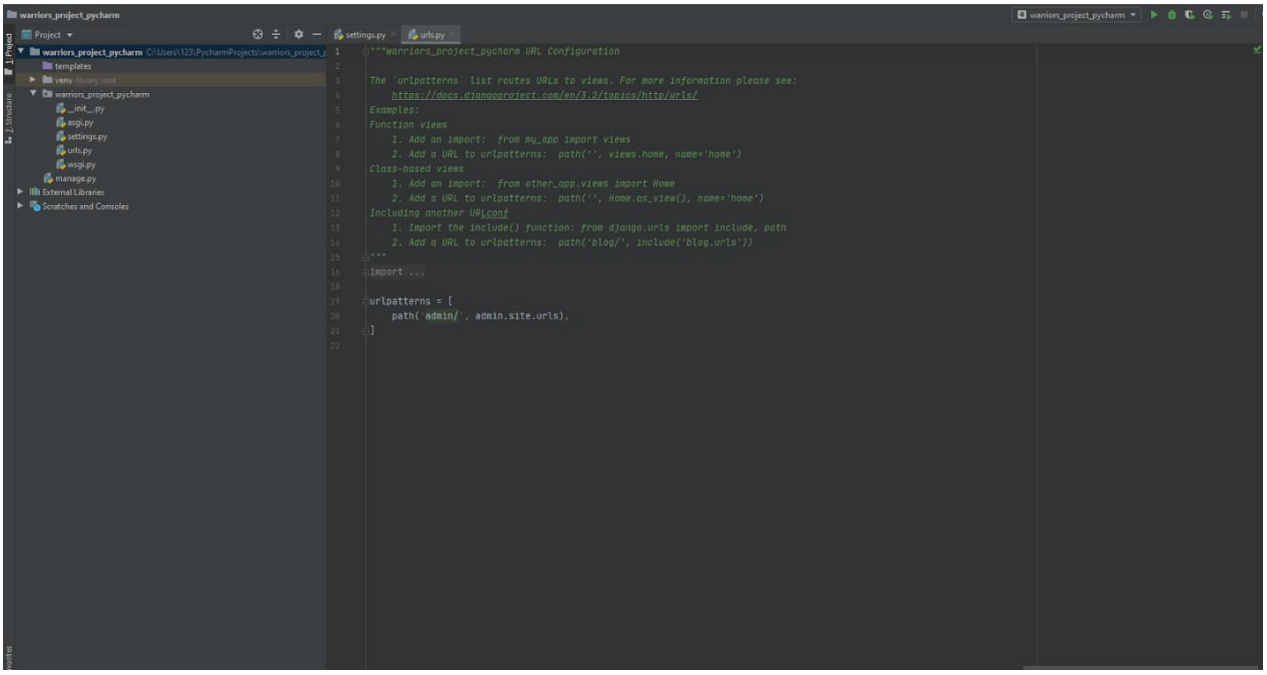


Рисунок 8 – Шаблон созданного проекта

5. Каждый проект Django состоит из приложений (application, app). Каждое такое приложение представляет собой сепарированный по логике набор файлов, которая имеет полностью самостоятельный программный код. Например, система авторизации, система оплаты и т.п. Эти приложения могут переноситься в любые другие приложения.

6. Создать приложение warriors.

Для этого в командной строке IDE нужно написать:

```
manage.py startapp warriors
```

7. Папка приложения warriors находится в папке проекта и имеет схожую структуру (рисунок 9).

Таких приложений в одном проекте Django может быть несколько.

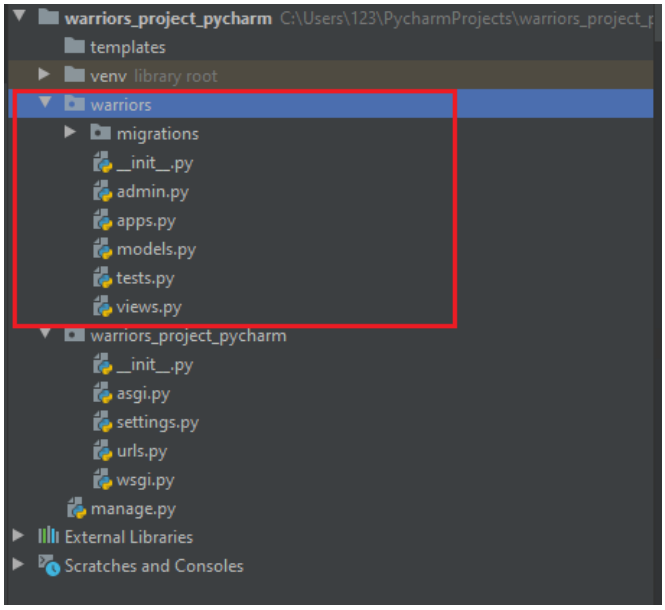


Рисунок 9 – Файловая структура созданного проекта

При создании приложения Django автоматически будет создана база данных SQLite.

Для того чтобы удостовериться, что установка фреймворка была выполнена корректно, необходимо в командной строке написать `manage.py runserver` или запустить проект через IDE. Тогда, по адресу http://127.0.0.1:8000/ должна отображаться следующая страница (рисунок 10).

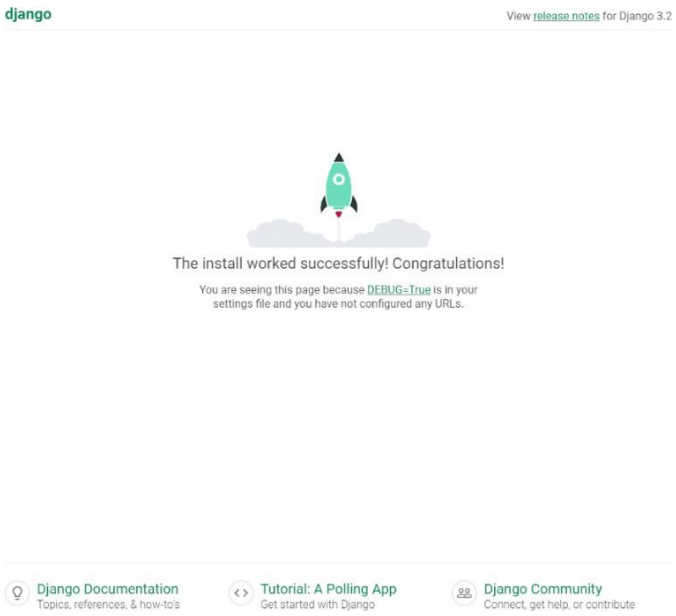


Рисунок 10 – Стартовая страница фреймворка

Практическое задание 2.1.1

Необходимо установить Django Web framework любым доступным способом.

Формат именований файлов:

- Формат имени Django-проекта: «django_project_фамилия».
- Формат имени Django-приложения: «project_first_app».

После установки фреймворка пользователю доступны файлы приложения. Далее представлено описание файловой архитектуры и содержания файлов приложения.

Описание файловой архитектуры проекта

Django имеет свою устоявшуюся архитектуру взаимодействия между элементами (файлами) приложения, основывающуюся на модели MVC. Каждый файл отвечает за определенную логику в веб-приложении и составляет каркас общей системы обмена данными и их генерации. На рисунке 10 показана стандартная файловая архитектура. Наименования у файлов, отвечающих за определенные части системы, могут быть любым, но крайне не рекомендуется называть стандартные элементы каким-то другим образом, отличным от показанного на изображении.

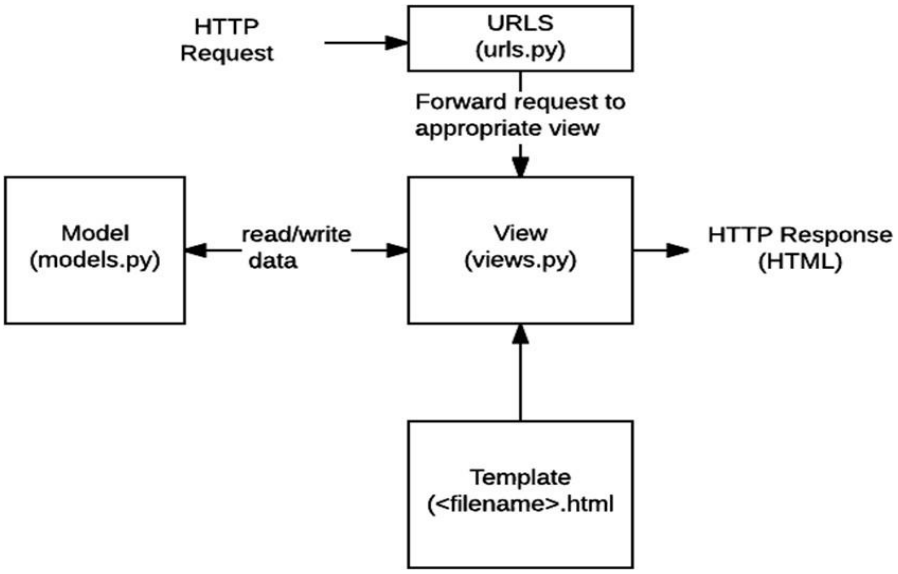


Рисунок 11 – Стандартная файловая архитектура проекта в Django

Такая архитектура может распространяться как на целые приложения (django app), так и на простые пакеты внутри конкретного приложения. Любая новая логика для какой-то группы элементов выносится отдельно по такой же схеме, как показано выше, т.е. если необходимо файлово/визуально разделить проект, то

нужно создать новое приложение (или папку внутри приложения) с таким же названием файлов (views, models, urls, template), где будет реализовываться новый функционал.

Описание файлов приложения

- URLs: URL-mapper используется для перенаправления HTTP-запросов в соответствующее представление на основе URL-адреса запроса. URL-mapper также может извлекать данные из URL-адреса в соответствии с заданным шаблоном и передавать их в соответствующую функцию в виде аргументов. [4]
- View: Представление (view) – это функция обработчика запросов, которая получает HTTP-запросы и возвращает ответы. View имеет доступ к данным через модели необходимым для удовлетворения запросов и делегирования ответа в шаблоны. [5]
- Models: Модели представляют собой объекты Python, которые определяют структуру данных приложения и предоставляют механизмы дляуправления (добавления, изменения, удаления) и выполнения запросов к базе данных. Это основной компонент для определения SQL-таблиц через Django-ORM. [6]
- Templates: Template (шаблон) – это текстовый файл, определяющий структуру или разметку страницы (например, HTML страницы), с полями для подстановки, используемыми для представления актуального содержимого. View может динамически создавать HTML-страницы, используя HTML шаблоны и заполняя их данными из модели (model). Шаблон может быть использован для определения структуры файлов любых типов, не обязательно HTML. [7]

2.1.2 Работа с моделью Django

В рамках этого задания рассматривается взаимодействие с моделями и файлом models.py.

1. Что такое модель?

Модели отображают информацию о данных, с которыми работает пользователь. Они содержат поля и поведение данных. Обычно одна модель представляет одну таблицу в базе данных. Каждая модель – это класс, унаследованный от django.db.models.Model. Атрибут модели представляет собой поле в базе данных.

2. Расположение моделей

Файл models.py находится в папке приложения Warriors. Он будет наполняться информацией для создания таблиц в базе данных (рисунок 12).

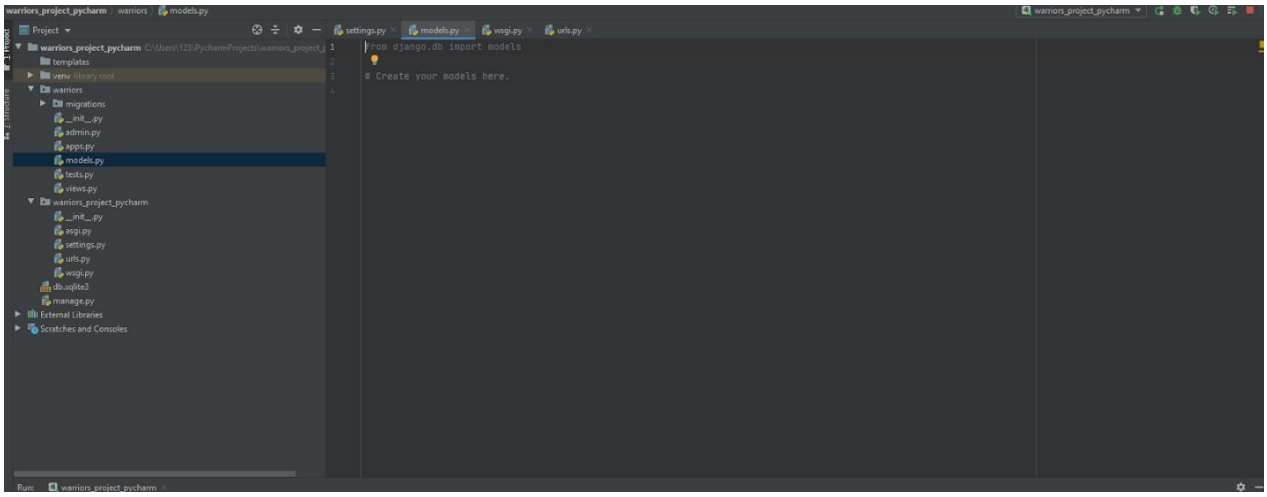


Рисунок 12 – Расположение файла models.py

3. Схема БД примера

На рисунке 13 представлена схема проектируемой БД для приложения «Воины».

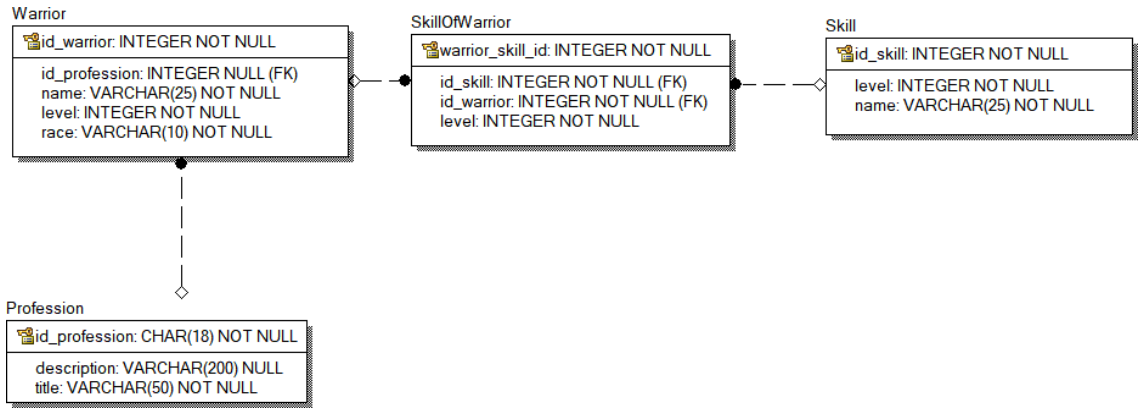


Рисунок 13 – Схема проектируемой БД

В данной схеме есть главные объекты – воины. Каждый воин может обладать профессиями и множеством применяемых умений. У каждого конкретного воина есть свой уровень умения, при этом у умения в целом можно указать максимальный уровень. Воин обладает одной расой из строго детерминированного списка, и сам имеет уровень (по умолчанию должен равняться 0).

4. Реализация таблиц

На примере таблицы «Профессия» будет продемонстрировано, как экстраполировать разработку SQL-таблиц на Django ORM.

Каждый класс модели должен обязательно наследоваться от *django.db.models.Model.* Такая реализация позволяет фреймворку обнаруживать подобные классы, чтобы впоследствии создать их в базе данных.

Модель профессии:

```
class Profession(models.Model):
    """
    Описание профессии
    """
    title = models.CharField(max_length=120, verbose_name='Название')
    description = models.TextField(verbose_name='Описание')
```

title и *description* – поля модели. Для первого поля указывается тип *Charfield*. ORM будет искать подобную реализацию типа во время преобразования класса в таблицу БД в независимости от выбранной конченой СУБД. Параметр *max_length* аналогичен SQL-значению длины поля типа Char. *verbose_name* – параметр, отвечающий за отображение поля при отображении таблицы (к примеру, в панели администратора), это исключительно визуальное свойство [8].

Реализация такой таблицы будет аналогична такому варианту на языке SQL (PostgreSQL):

```
CREATE TABLE warriors_profession(
"id" serial NOT NULL PRIMARY KEY,
"title" varchar(120) NOT NULL,
"description" text NOT NULL
);
```

Название таблицы *warriors_profession* автоматически создается с метаданных модели и может быть переопределено [9].

Поле первичного ключа id создается автоматически, но его также можно переопределить.

Django использует синтаксис SQL соответственно настройкам базы данных в файле настроек.

Модель описания умения:

```
class Skill(models.Model):
    """
    Описание умений
    """

    title = models.CharField(max_length=120, verbose_name='Наименование')

    def __str__(self):
        return self.title
```

В представленной модели используется магическая функция *__str__*. Она позволяет при преобразовании объекта в строковое значение выводить результат, указанный в *return*. Часто используется для отображения моделей в панели

администратора и при вызове метода *print()*.

Модель воина:

```
class Warrior(models.Model):
    """
    Описание война
    """
    race_types = (
        ('s', 'student'),
        ('d', 'developer'),
        ('t', 'teamlead'),
    )
    race = models.CharField(max_length=1, choices=race_types,
verbose_name='Расса')
    name = models.CharField(max_length=120, verbose_name='Имя')
    level = models.IntegerField(verbose_name='Уровень', default=0)
    skill = models.ManyToManyField('Skill', verbose_name='Умения',
        through='SkillOfWarrior', related_name='warrior_skils')
    profession = models.ForeignKey('Profession',
on_delete=models.CASCADE,
            verbose_name='Профессия', blank=True, null=True)
```

Поле *race* позволяет выбрать принадлежность воина из ограниченного списка вариантов. Для этого вместе с *CharField* используется параметр *choices,* который принимает в себя любой итерируемый двумерный объект как пару *значение_хранимое_в_БД/визуальное_описание.*

Поле *level* позволяет описать уровень у воина посредством типа *IntegerField*. С помощью параметра *default* начальное значение при создании указывается как 0.

Поля отношений:

В таблице «Воин» присутствуют два поля, указывающие на реляционные отношения. Поле *profession* имеет тип *ForeignKey*, принимающий первый обязательный аргумент ссылку на таблицу *Profession* (ссылки могут быть как строкой, так и Python-ссылкой на класс, но рекомендуется использовать первый вариант во избежание циклических импортов).

У *ForeignKey* также есть обязательный параметр *on_delete* - он позволяет определять, что будет происходить с объектом при удалении объекта, на который объявлена ссылка.

Поля blank и null отвечают за возможность не указывать значения, т.е. воин изначально может обходится без профессии.

Поле умений объявляет класс *ManyToManyField* (Многие ко многим) к таблице *Skill.* В обычном варианте этого достаточно, чтобы реализовать подобное отношение – Django автоматически сгенерирует ассоциативную сущность для связи этих двух таблиц. Но если подобной в подобной таблице хранятся какие-то

данные, то указывается параметр *through,* содержащий ссылку на таблицу, через которую будет реализована ассоциативная сущность. В рассматриваемом варианте это таблица *SkillOfWarrior.*

Модель Способность Воина:

```
class SkillOfWarrior(models.Model):
    """
    Описание умений война

    """

    skill = models.ForeignKey('Skill',
verbose_name='Умение',

      on_delete=models.CASCADE)
    warrior = models.ForeignKey('Warrior', verbose_name='Воин',
        on_delete=models.CASCADE)
    level = models.IntegerField(verbose_name='Уровень освоения умения')
```

Для корректной реализации ассоциативной сущности ее нужно делать по той же логике, что и при написании на SQL: создавать два поля-ссылки на таблицы, которые необходимо связать. Эта сущность создается, чтобы воины обладали уровнем умений.

5. Создание миграций

Система миграций в Django – это инструментарий, позволяющий программисту быстро преобразовывать объектно-ориентированный код в SQL и вести контроль преобразований таблиц в базе.

Теперь написанный код нужно преобразовать в таблицы и связи в базе данных. Это возможно осуществить с помощью миграций.

Чтобы фреймворк смог определить рабочее приложение, его нужно явно указать в переменной *INSTALLED_APPS* в файле *setting.py* (рисунок 14).

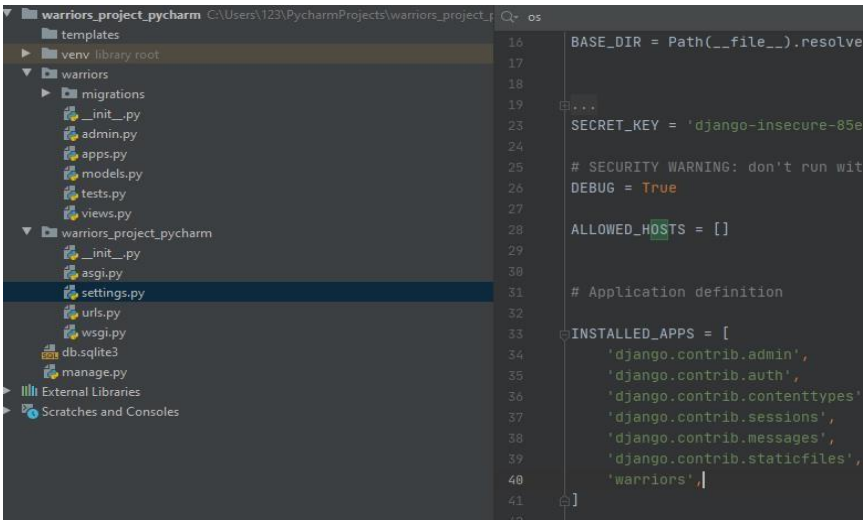


Рисунок 14 – Расположение и формат файла settings.py

Затем нужно создать файл миграций (специальный .py файл, описывающий изменения в модели данных). Для этого достаточно написать команду `manage.py makemigrations <имя_приложения>` и затем, для применения миграций к базе данных, ввести команду `manage.py migrate`.

Пример:

```
(venv) C:\Users\123\PycharmProjects\warriors_project_pycharm>manage.py
makemigrations
Migrations for 'warriors':
  warriors\migrations\0001_initial.py
    - Create model Profession
    - Create model Skill
    - Create model SkillOfWarrior
    - Create model Warrior
    - Add field warrior to skillofwarrior
```

```
(venv) C:\Users\123\PycharmProjects\warriors_project_pycharm>manage.py
migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, warriors
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying warriors.0001_initial... OK
```

Для того чтобы удостовериться в том, что таблицы были созданы, можно воспользоваться любым Database Explorer-ом и посмотреть, как выглядит используемая БД (рисунок 15).

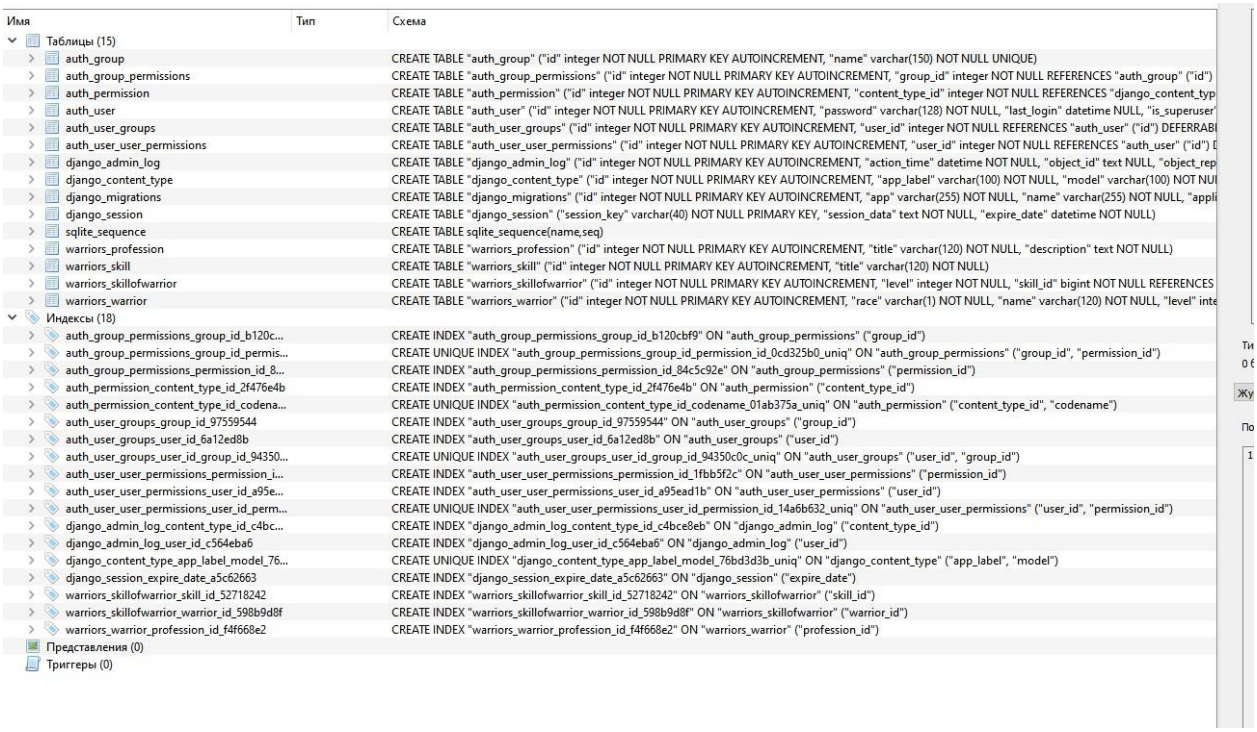


Рисунок 15 – Результат применение миграций к базе данных

Практическое задание 2.1.2.1

В проекте необходимо создать модель данных об автовладельцах в соответствии с рисунком (рисунок 16).

Указание: Таблицы и атрибуты именовать только на латинице.

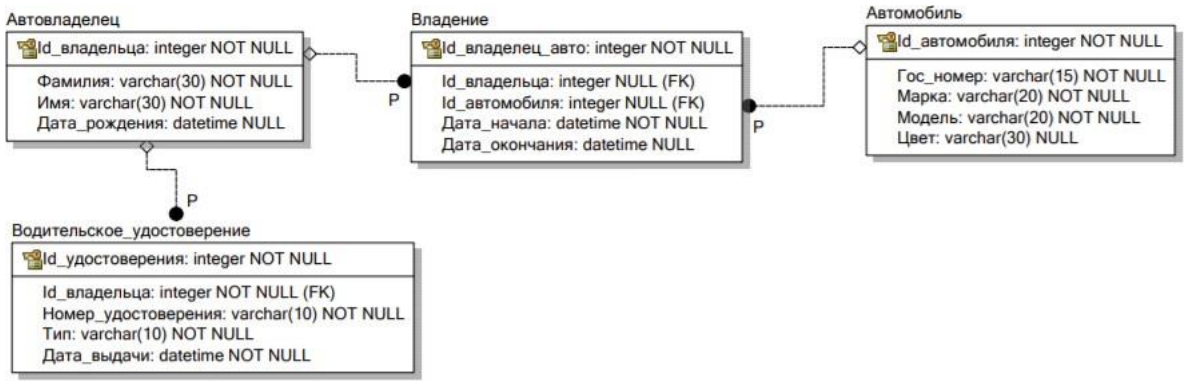


Рисунок 16 – Модель данных об автовладельцах

Примечание. Не забывать о внешних ключах, полях выбора, правильном оформлении ассоциативной сущности в модели.

Практическое задание 2.1.2.2

Создать миграции и применить их к базе данных.

Код для создания и применения миграций:

```
python manage.py makemigrations #создает миграции на основе имеющейся модели данных. Созданные миграции доступны в файле миграций, их можно изменить вручную. Просмотрите их содержание в папке migrations вашего приложения.
python manage.py migrate #применяет миграции к базе данных.
```

По умолчанию Django Web framework настроен для работы с SQLite. Во время первого выполнения миграций будет создана новая база в SQLite. При необходимости возможно скачать DB Browser for SQLite [10] и проверить работу созданной базы.

2.1.3 Создание админ-панели

Приложение Django admin может использовать модели для автоматического создания интерфейсов, предназначенных для создания, просмотра, обновления и удаления записей. Это может сэкономить много времени в процессе разработки, упрощая тестирование моделей на предмет правильности данных. Оно также может быть полезным для управления данными на стадии публикации, в зависимости от типа веб-сайта. Проект Django рекомендует это приложение только для управления внутренними данными (т.е. для использования администраторами, либо людьми внутри вашей организации), так как модельно-ориентированный подход не обязательно является наилучшим интерфейсом для всех пользователей и раскрывает много лишних подробностей о моделях.

1. Регистрация таблиц

Для взаимодействия с какой-либо таблицей через панель администратора нужно ее зарегистрировать. Для этого надо перейти в папку приложения и открыть файл admin.py. Для регистрации моделей на данном этапе можно воспользоваться функцией admin.site.register, где в аргументе указывается ссылки на классы таблиц:

```
from django.contrib import admin
from .models import Warrior, Profession, Skill, SkillOfWarrior

admin.site.register(Warrior)
admin.site.register(Profession)
admin.site.register(Skill)
admin.site.register(SkillOfWarrior)
```

2. Регистрация и вход в панель администрирования

Чтобы зайти в панель администрирования, необходимо создать какого-то пользователя: для этого через командную строку создается аккаунт администратора посредством команды *manage.py createsuperuser*.

После ввода данных о пользователе можно запустить сервер и войти в панель администрирования по адресу http://127.0.0.1:8000/admin/ (рисунок 17).

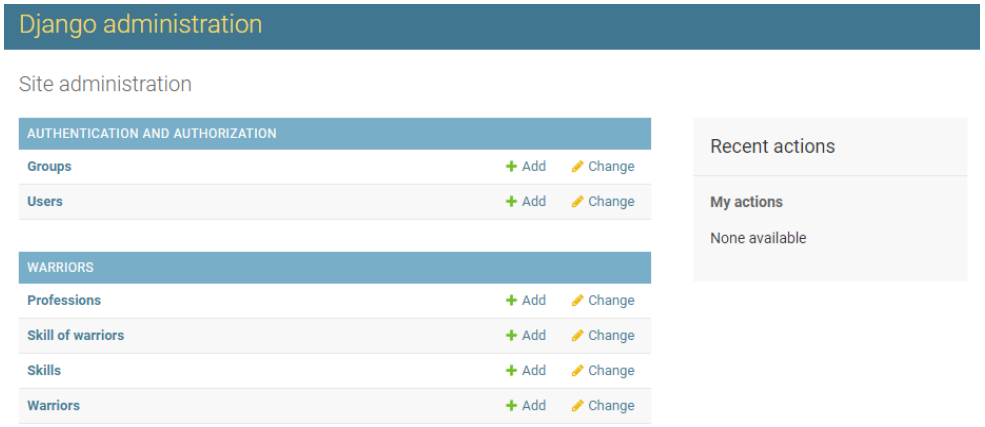


Рисунок 17 – Панель администрирования Django

Как можно заметить, здесь присутствуют все ранее зарегистрированные модели, а также модели, которые создаются автоматически при инициализации Django-проекта.

3. Ввод информации

Для этого необходимо ввести информацию о записи в соответствующие поля. Для добавления нового связанного объекта достаточно нажать «+» рядом с нужным полем (рисунок 18).

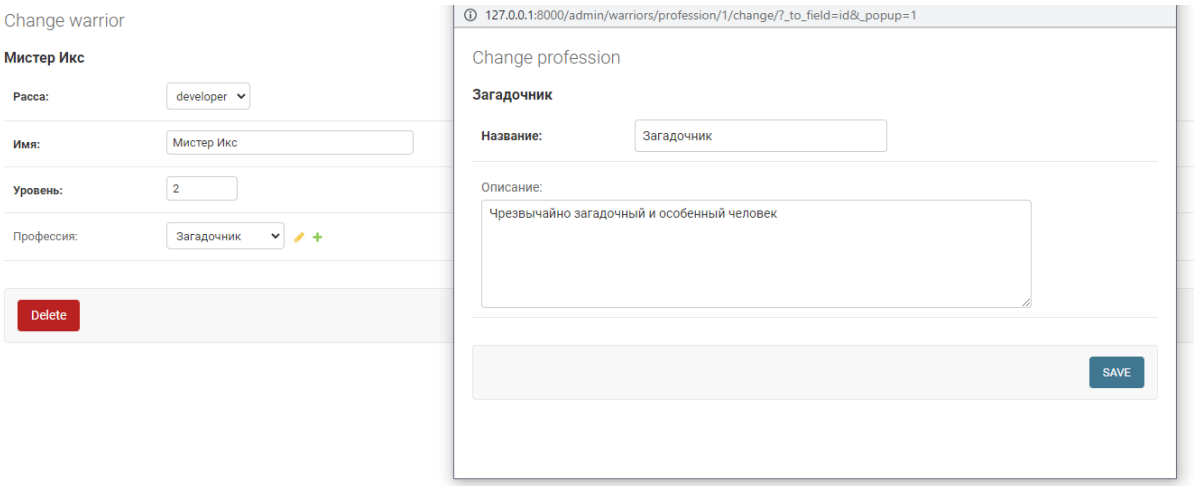


Рисунок 18 – Создание нового объекта в панели администрирования

После профессия и воин сохраняются и создаются 2 новые таблицы (рисунок 19).

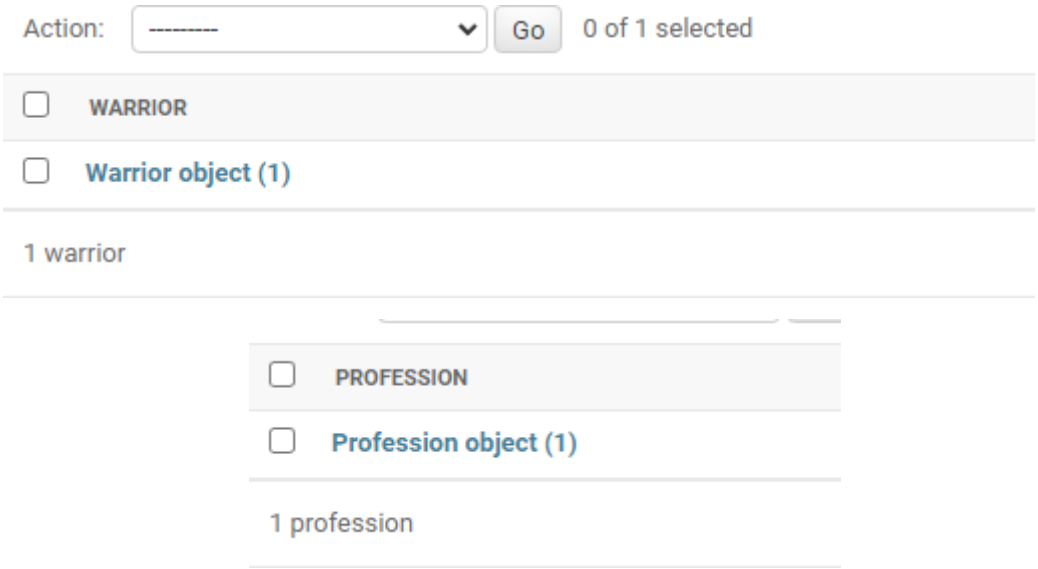


Рисунок 19 – Список созданных объектов

Для корректного отображения имен переопределяются методы *str* внутри данных моделей: Warrior:

```
def __str__(self):
    return self.name
```

Profession:

```
def __str__(self):
    return self.title
```

Теперь вместо названия класса с id отображаются описанные имена объектов.

Практическое задание 2.1.3

1. Заполнить таблицы данными средствами админ-панели.

2. Зарегистрировать владельца авто в админ-панели.

Необходимо зайти в файл admin.py в папке приложения (*_app) и зарегистрировать владельца автомобиля следующими командами:

```
    from .models import Название владельца в модели данных #то, что
указано в кавычках, - это какие-либо параметры, которые Вам нужно ввести
самостоятельно, в соответствии с Вашим проектом. В Конкретном случае, Вам
```

```
нужно указать название таблицы модели данных, которую необходимо отразить
в админ-панели.
admin.site.register(Название владельца в модели данных)
```

В ходе выполнения задания рекомендуется ознакомиться с документацией [11].

3. Зарегистрировать остальные таблицы модели данных в админ-панели.
4. Создать суперпользователя.
5. Зайти в админ-панель по url-адресу (127.0.0.1:8000/admin/) и добавить двух владельцев автомобилей, 4 автомобиля. Далее связать каждого владельца минимум с тремя автомобилями, так, чтобы не было пересечений по датам владения и продажи.

2.1.4 Создание контроллеров для обработки данных

Контроллер является центральным компонентом в архитектуре MVC. Контроллер получает ввод пользователя, обрабатывает его и посылает обратно результат обработки. В Django логика контроллеров хранится в файлах views.py.

View, или представление, – файл, где происходит взаимодействие с логикой работы приложения. Оно запрашивает информацию из ранее созданной модели, и передает ее в шаблон html-страницы.

1. Создание представлений

В папке приложений необходимо перейти в файл views.py – именно там реализуются все представления. Для начала надо создать простейшую функцию, которая будет принимать два параметра: объект *request,* содержащий внутреннюю информацию о запросе, и *id* – номер воина. Внутри функции произведен запрос к базе с помощью стандартного менеджера *objects,* предоставляющего метод *get* для получения ровно одного объекта. Любой менеджер применяется к классу модели как будто вызывается статический метод. Пример такого запроса:

```
warrior = Warrior.objects.get(name=”Konstantin Yanson”)
```

Для проверки объекта на его существование используется исключение DoesNotExist у класса модели, если такого не существует, то выводится ошибка 404.

Если запрос удался, то объект запроса, путь к шаблону и контекст (словарь, указывающий на полученный объект) передается в функцию *render*.

Результат:

```python
from django.http import Http404
from django.shortcuts import render
from warriors.models import Warrior


def get_warrior_data(request, id):
    try:
        warrior = Warrior.objects.get(id=id)
    except Warrior.DoesNotExist:
        raise Http404("owner does not exist")
    return render(request, 'html/warrior.html', {'warrior': warrior})
```

2. Реализация шаблона

Templates в Django – это html-заготовки для отображения информации, передаваемой в контекст шаблона, написанные с помощью специального синтаксиса.

Контекст шаблона – это специальный словарь, содержащий информацию, запрашиваемую из представления. Внутри шаблона можно обратиться к какому-нибудь полю контекста с помощью фигурных скобок.

Например, `<text>{{ warrior.level }}</text>` выведет уровень объекта «воин» в теге `text`. Стоит понимать, что шаблоны автоматически преобразуют любой передаваемый объект в строку, поэтому, чтобы отобразить имя в рамках рассматриваемой модели достаточно написать: `<h1>{{warrior}}</h1>`.

Для того чтобы реализовать шаблон, его необходимо поместить в специальную папку внутри приложения. Путь будет такой `/templates/html/warrior.html`. Стоит отметить, что функция render в представлениях автоматически обнаруживает папку templates, поэтому достаточно передать только расположение файла страницы внутри этой папки. Ниже представлен код, отображающий уровень и расу воина:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>{{warrior}}</h1>
<text>{{ warrior.level }}</text> <br>
<text>{{ warrior.profession }}</text><br>
<text>{{ warrior.get_race_display}}</text><br>
```

```
</body>
</html>
```

Метод *get_race_display* позволяет отобразить поле race с полным наименованием из массива choices в моделях (шаблон сам вызывает функцию, достаточно указать только ссылку на нее. Это работает с любыми функциями, вызываемыми в шаблонах).

Практическое задание 2.1.4

1. Создать в файле views.py (находится в папке приложения) представление (контроллер), который выводит из базы данных данные о владельце автомобиля.

В качестве примера воспользоваться официальной документацией [12].

Прокомментированный пример из официальной документации:

```
     from django.http import Http404
#импортирует метод обработки ситуации, когда нет    необходимых записей в бд (обработчик ошибок)
from django.shortcuts import render #импортирует метод, который "запускает" созданную html страницу и передает в нее указанные параметры
from polls.models import Poll #импортирует таблицу Poll из модели данных models, где polls - название приложения (и папки)
def detail(request, poll_id):
    try: #метод try-except - обработчик исключений
        p = Poll.objects.get(pk=poll_id)  #pk - автоматически создается в django для любой таблицы в модели (оно есть у любого объекта из бд), poll_id будет передан функции при её вызове.
#переменной p присваивается объект, полученный в результате выполнения запроса аналогичного "select * from Poll where pk=poll_id"
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist") #исключение которое будет вызвано, если блок try вернет значение False (не будут найдены записи в таблице Poll)
    return render(request, 'polls/detail.html', {'poll': p}) #данная строка рендерит хтмл страницу detail.html и передает в него объект "p", который в хтмл шаблоне будет называться "poll"
```

2. Создать страницу html-шаблона owner.html в папке templates (создать папку templates в корне проекта, если ее нигде нет, далее в контекстном меню папки создать html-файл). Страница должна содержать отображение полей, переданных из контроллера. Для обращения к этим полям используется конструкция на языке шаблонов Jinja2 как в примере ниже:

```
<body>
Имя: {{owner.first_name}}, Фамилия: {{owner.last_name}} <br>
</body>
```

2.1.5 Работа с адресацией

После создания представления и шаблона необходимо предоставить к ним доступ обычному пользователю. В Django за это отвечает архитектурный элемент URLs, занимающийся роутингом. Главный файл urls.py находится в папке с настройками проекта. В случае рассматриваемого проекта это warriors_project_pycharm. Хорошим тоном является создание urls.py-файла отдельно для каждого архитектурного MVC-блока, поэтому в папке приложения необходимо создать точно такой же файл urls.py. Чтобы приложение Django могло определить написанные адреса, необходимо подключить в основном файле urls (в папке настроек: /warriors_project_pycharm/urls.py) созданный дочерний файл. Код представлен ниже:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
   path('admin/', admin.site.urls),
   path('', include('warriors.urls'))
]
```

В новом файле адресации, созданном внутри приложения, необходимо указать описанное в задании 2.1.4 представление. Это делается посредством создания специального списка *urlpatterns,* содержащего функции *path* с аргументами в виде адреса и ссылки на представление.

Для того чтобы получить доступ к рассматриваемому представлению, недостаточно просто открыть какой-то адрес. Нужно также указывать в качестве аргумента id воина. Для этого в синтаксисе адресации присутствует специальная конструкция *<type:name>,* где type – тип данных, а name – имя переменной, которую надо передать в функцию. Ниже представлено зарегистрированное представление в файле urls.py в приложении (/warriors/urls.py):

```
from warriors import views

urlpatterns = [
   path('warrior/<int:id>/', views.get_warrior_data),
]
```

Если перейти по адресу http://127.0.0.1:8000/warrior/1/, то можно увидеть следующий результат (рисунок 20).

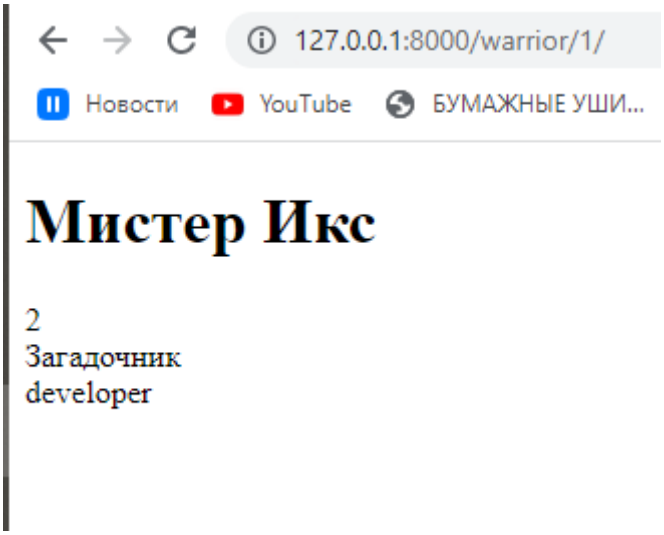


Рисунок 20 – Результат обработки шаблона

На рисунке 21 показано, что выводит браузер, если обратиться к несуществующему воину.

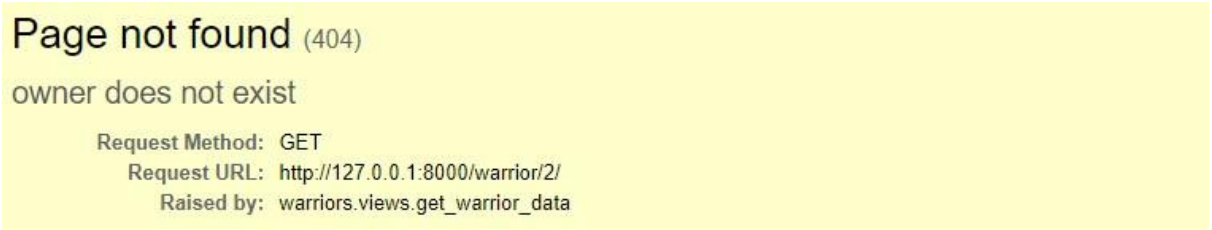


Рисунок 21 – Ошибка в случае, если id объекта не существует

Практическое задание 2.1.5

1. Создать файл адресов urls.py в папке приложения (*_app) (пока пустой).
2. Импортировать файл urls.py приложения в проект (модифицировать файл urls.py в той папке, в которой хранится файл setting.py).

Файл должен иметь подобный код:

```
     from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('example6_app.urls')), # данная строчка импортирует
в проект отдельный файл юрлов приложения (example6_app - название
приложения (название папки)), urls указывает на файл юрлов в папке
приложения (указывает на тот пустой файл, который создан в пункте 9.a.
]
```

3. Описать в файле urls.py приложения, созданном в пункте 4.1.1, такой url-адрес, который сможет обратиться к контроллеру и вывести страницу, которая должна быть отрендерена контроллером.

Необходимо обратиться к контроллеру, который создан в задаче 4, и передать параметр (идентификационный номер владельца) в адресной строке согласно примеру [13].

```
     from django.urls import path
from . import views # подключение файла контроллеров, описанного в пункте 3
urlpatterns = [
    path('articles/2003/', views.special_case_2003), # пример вызова контроллера (функции) с именем "special_case_200" из файла views
    path('articles/<int:year>/', views.year_archive), # пример вызова контроллера (функции) с именем "year_archive" из файла views и передачи в него переменной "year"
]
```

Результат: по url-адресу “127.0.0.1:8000/owner/1” возможно получить страницу с информацией о первом владельце автомобилей.

Практикум 2.2 Использование особенностей Django для разработки приложения

2.2.1 Доработка модели данных. Реализация связи «Многие ко многим»

Реализация связи «многие-ко-многим» в Django возможна несколькими способами:

1. Автоматическое создание связи, как в примере, приведенном в коде:

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

В данной реализации явно объявлено две таблицы, но в базе данных будет создана ассоциативная таблица, которая связывает две исходные таблицы.

2. Ручное создание ассоциативной таблицы, если в ней необходимо сохранить какие-то параметры (т.е. она имеет собственные атрибуты) приведено в коде:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)
    def _str_(self):

        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership') # в
данной строке through='Membership' указывает на таблицу, которая будет
использоваться, как ассоциативная сущность.

    def _str_(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

Практическое задание 2.2.1

Настроить корректно связь между автомобилем, владением и владельцем.

2.2.2 Работа с представлениями

В практическом задании 2.1.4 было создано первое представление. Далее рассмотрен пример для повторения логики работы представлений.

Пример для повторения

Код файла views.py

```
# import Http Response from django
from django.http import HttpResponse
# get datetime
import datetime

# create a function
def example_view(request):
    # fetch date and time
    now = datetime.datetime.now()
    # convert to string
    html = "Time is {}".format(now)
    # return response
    return HttpResponse(html)
```

Описание:

- Импортирован класс HttpResponse из django.http модуля вместе с библиотекой datetime в Python.
- Определена функция с именем example_view. Функция представления принимает объект HttpRequest в качестве первого параметра, который обычно называется запросом.
- Представление возвращает объект HttpResponse, который содержит сгенерированный ответ. Каждая функция представления отвечает за возврат объекта HttpResponse.

Далее настраивается адресация в example/urls.py:

```
from django.urls import path

# importing views from views..py
from .views import example_view

urlpatterns = [
    path('time/', example_view),
]
```

Затем необходимо проверить функцию, вызвав ее по ранее описанному url-адресу http://127.0.0.1:8000/time/ (рисунок 22).

Рисунок 22 – Вызов шаблона по url-адресу

Комментарий: в практикуме 2.1 была написана функция, возвращающая html-страницу. В данном примере рассмотрена функция, которой не был передан шаблон. Средствами HttpResponse возможно сразу отправить данные клиенту без рендеринга html-страницы с помощью шаблона.

Типы представлений (контроллеров) в Django

Представления в Джанго бывают двух типов (рисунок 23):

- Представления на основе функций.
- Представления на основе классов.

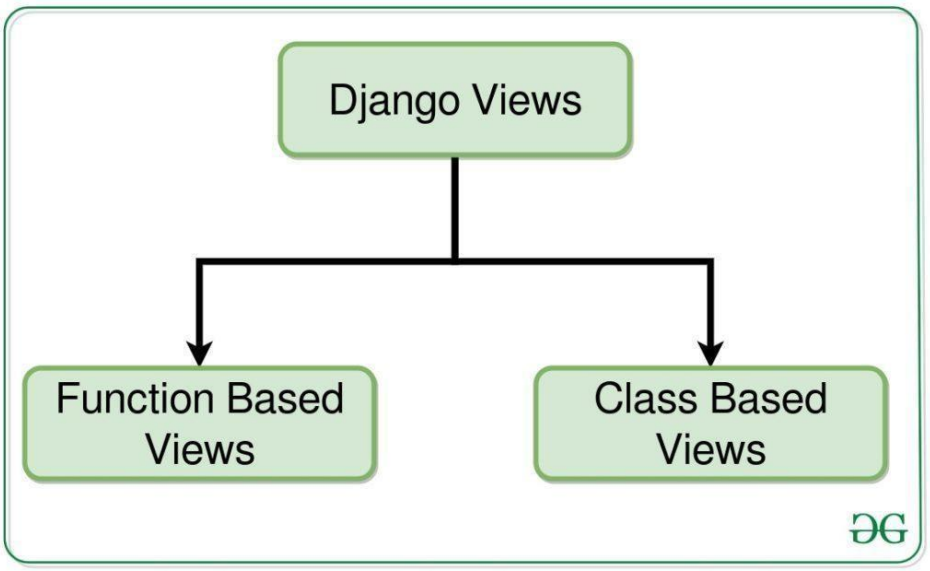


Рисунок 23 –Типы представлений в Django

Представления на основе функций

Представления на основе функций пишутся с использованием функции в python, которая получает в качестве аргумента объект HttpRequest и возвращает объект HttpResponse.

Представления, основанные на функциях, обычно реализуют 4 операции: Создать (Create), Получить (Read), Обновить (Update), Удалить (Delete) (аббревиатура: CRUD). CRUD является основой любого фреймворка, который используется для разработки.

Далее рассмотрен пример представления для вывода данных из БД:

1. Необходимо создать модель, из которой клиенту будут выводиться объекты через представления.

Код models.py:

```
# import the standard Django Model #
from built-in library
from django.db import models

# declare a new model with a name "ExampleModel" class
ExampleModel(models.Model):

    # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()

    # renames the instances of the model #
    with their title name
    def __str__(self): return
        self.title
```

После создания этой модели необходимо выполнить и применить миграции.

```
Python manage.py makemigrations
Python manage.py migrate
```

2. Теперь необходимо создать несколько экземпляров этой модели, запустив bash, используя shell:

```
Python manage.py shell
```

Для загрузки данных в таблицу в базу данных можно обратиться к ней следующим образом:

```
>>> from example.models import ExampleModel
>>> ExampleModel.objects.create (
                        title = "title1",
                        description = "description1").save()
>>> ExampleModel.objects.create (
                        title = "title2",
                        description = "description2").save()
>>> ExampleModel.objects.create (
                        title = "title2",
                        description = "description2").save()
```

3. Далее для возможности работы с моделью через админ-панель Django необходимо зарегистрировать в ней модель.

Код файла admin.py:

```
from django.contrib import admin
from .models import ExampleModel
admin.site.register(ExampleModel)
```

Средствами админ-панели, можно проверить содержание таблиц. Url: http://localhost:8000/admin/project_first_app/examplemodel/ (рисунок 24).

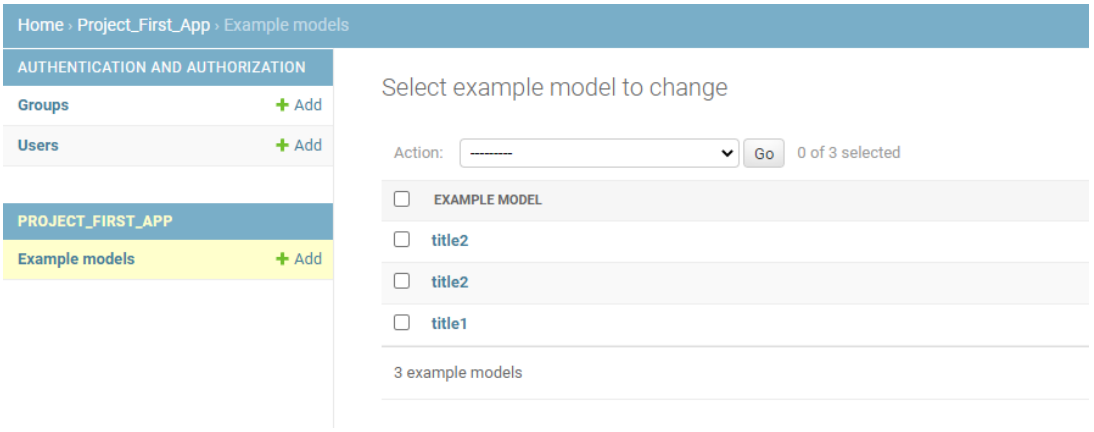


Рисунок 24 – Состав таблиц в админ-панели

4. Создать представление (контроллер).

Код в example/views.py:

```
from django.shortcuts import render

# relative import of forms
from .models import ExampleModel


def list_view(request):
    # dictionary for initial data with
    # field names as keys
    context ={}

    # add the dictionary during initialization [en]
    # добавление данных об объектах, полученных в результате выполнения
запроса exampleModel.objects.all() в словарь
    context["dataset"] = ExampleModel.objects.all()

    return render(request, "list_view.html", context)
```

5. Создать шаблон (клиентская часть).

Код создания шаблона в templates/list_view.html:

```
<div class="main">

    {% for data in dataset %}.

    {{ data.title }}<br/>
    {{ data.description }}<br/>
    <hr/>

    {% endfor %}

</div>
```

Функция render (пункт 5) выполняет «сборку» страницы, используя словарь context.

6. Создать файл для работы с адресацией.

Файл, в котором хранятся адреса, по которым вызывается функционал приложения, urls.py:

```
from django.urls import path
from .views import list_view

urlpatterns = [
   path('example_list/', list_view),

]
```

Результат можно посмотреть в браузере по следующему url-адресу http://localhost:8000/example_list/ (рисунок 25).

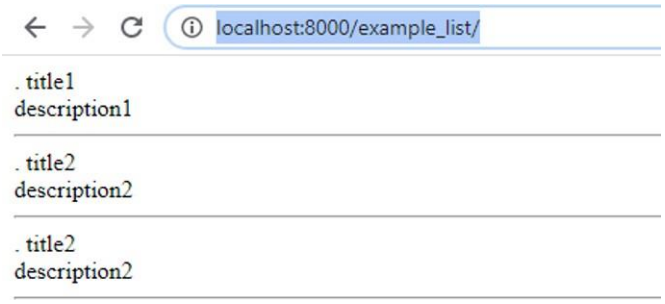


Рисунок 25 – Результат вывода списка объектов через функцию-представление

Подобные представления на основе функций могут быть использованы для просмотра, создания, добавления и удаления данных.

Представления на основе классов

Представления на основе классов предоставляют альтернативный способ реализации представлений в виде объектов Python вместо функций. Они не заменяют функциональные представления, но имеют определенные различия и преимущества по сравнению с функциональными представлениями:

- Организация кода, связанного с конкретными методами HTTP (GET, POST и т.д.), может решаться отдельными методами вместо условного ветвления.
- Объектно-ориентированные методы, такие как mixins (множественное наследование), могут использоваться для разделения кода на повторно используемые компоненты.

Представления на основе классов проще и эффективнее в управлении, чем представления на основе функций. Представление на основе функций с тоннами строк кода может быть преобразовано в представления на основе классов только с несколькими строками.

Получение списка объектов

Далее рассмотрен пример получение списка данных об объектах, использующий методы класса ListView:

1. Представление.

Код example/views.py:

```
from django.views.generic.list import ListView
from .models import ExampleModel

class ExampleList(ListView):

    # specify the model for list view
    model = ExampleModel
    template_name = 'cvb_list_view.html'
```

ListView_относится к generic классам. В их основе лежит набор идиом и шаблонов, базирующийся на практическом опыте создания представлений, который дает пользователю абстрактный каркас для быстрого создания собственных представлений без необходимости писать лишний, повторяющийся код [14].

2. Для проверки работы примера необходимо настроить адресацию.

Код urls.py:

```
from django.urls import path

# importing views from views..py
from .views import ExampleList
urlpatterns = [
    path('cvb_example/', ExampleList.as_view()),
]
```

3. Далее необходимо настроить шаблоны в templates/cvb_list_view.html:

```
<ul>
    <!-- Iterate over object_list -->
    {% for object in object_list %}
    <!-- Display Objects -->
    <li>{{ object.title }}</li>
    <li>{{ object.description }}</li>

    <hr/>
    <!-- If objet_list is empty  -->
    {% empty %}
    <li>No objects yet.</li>
    {% endfor %}
</ul>
```

Результат можно посмотреть в браузере по следующему url- адресу http://localhost:8000/cvb_example/ (рисунок 26).

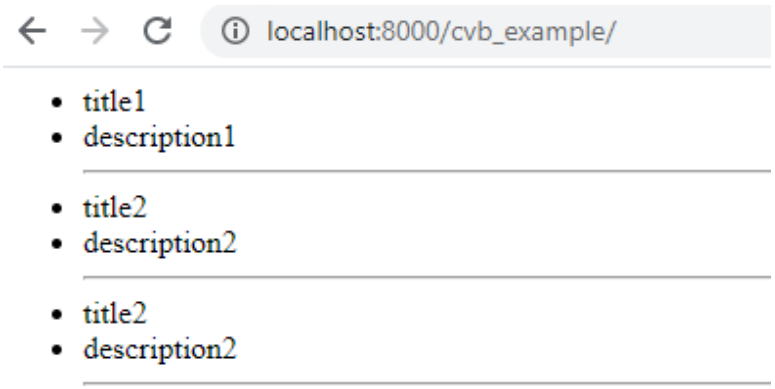


Рисунок 26 – Вывод списка объектов с помощью представления на основе классов

Получение одного объекта

1. Создать модель для этого и следующих примеров:

```
# models.py

class Publisher(models.Model):

  first_name = models.CharField(max_length=30)
  last_name = models.CharField(max_length=30)
  birthdate = models.DateField()

  def __str__(self):
     return "{} {}".format(self.first_name, self.last_name)
```

2. Реализовать получение одного объекта из БД по первичному ключу с использованием DetailView (относится к generic views):

```
# views.py
from django.views.generic.detail import DetailView

class PublisherRetrieveView(DetailView):
  model = Publisher
```

3. Generic view начинает искать шаблон по определенной маске. Чтобы он мог его найти, необходимо предварительное создание папки с названием приложения в директории templates, где и должен быть создан следующий шаблон:

```
<!-- publisher_detail.html -->

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Publisher detail</title>
</head>

<body>
  <h1>{{ object }}</h1>
  <p><strong>Birthday date:</strong> {{ object.birthdate }}</p>
  <a href="/publisher/list/">Get back</a>
</body>
</html>
```

4. Дополнить urls.py:

```
# urls.py
from django.urls import path
from .views import *

urlpatterns = [

  path('publisher/<int:pk>/', PublisherRetrieveView.as_view()),
]
```

5. Результат можно посмотреть в браузере по следующему URL-адресу: http://127.0.0.1:8000/publisher/1/ (рисунок 27).

Рисунок 27 – Получение конкретного объекта через представление

Переопределение стандартных методов в generic view

1. Создать модель данных:

```
# models.py
class Book(models.Model):
  name = models.CharField(max_length=100)
  desc = models.CharField(max_length=200)
  publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)

  def __str__(self):
      return "{}, {}".format(self.name, self.publisher)
```

2. Создать представление, возвращающего список книг по id автора:

```
# views.py
class BookListView(ListView):
  model = Book
  queryset = model.objects.all()

  def get_queryset(self):
      publisher = self.request.GET.get('publisher')

      if publisher:

        try:
          publisher = int(publisher)
          queryset = self.queryset.filter(publisher=publisher)

        except ValueError:

          queryset = self.model.objects.none()

        return queryset

      return self.queryset
```

Функция get_queryset() переопределяет получение queryset, обращаясь к GET-параметрам из запроса. В случае, если параметр publisher определен и является числом, происходит фильтрация и функция возвращает отфильтрованные данные.

3. Создать шаблон:

```
<!-- book_list.html -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Book list</title>
</head>

<body>
  <span><h1>Books</h1><a href="/book/create/">Add new one</a></span>
  <ul>
      {% for book in object_list %}
      <li><strong>Name:</strong> {{ book.name }}, actions: <a
href="/book/{{ book.id }}/update/">Edit</a> | <a href="/book/{{ book.id
}}/delete/">Delete</a> | <a href="/book/{{ book.id }}/">Detail</a>
      <p><strong>Description:</strong> {{ book.desc }}</p>
      <p><strong>Publisher:</strong> <a href="/publisher/{{
book.publisher.id }}">{{ book.publisher }}</a></p>
      </li>
      {% empty %}
      <li>No books yet.</li>
      {% endfor %}
  </ul>
</body>

</html>
```

4. Добавить URL-адрес:

```
# urls.py
from django.urls import path

from .views import *

urlpatterns = [
  path('book/list/', BookListView.as_view()),
]
```

5. Результат можно посмотреть в браузере по следующему URL-адресу: http://127.0.0.1:8000/book/list/?publisher=1.

Результат без параметров представлен на рисунке 28.

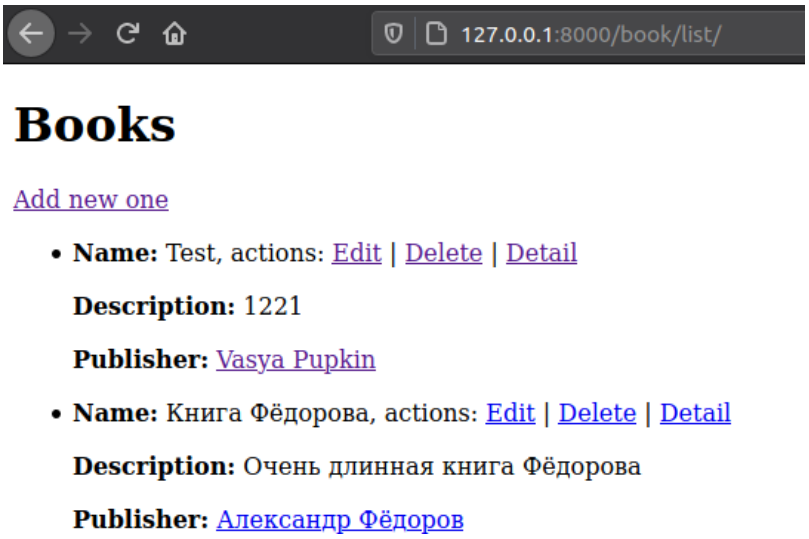


Рисунок 28 – Вывод списка книг без параметров в URL

Результат с параметрами представлен на рисунке 29.

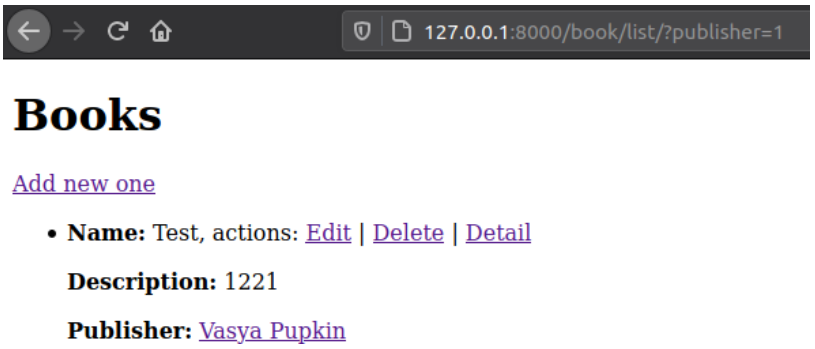


Рисунок 29 – Вывод списка книг с параметром id издателя

Практическое задание 2.2.2

1. Реализовать вывод всех владельцев авто функционально. Добавить данные минимум о трех владельцах. Должны быть реализованы контроллер (views) и шаблоны (temlates).
2. Реализовать вывод всех автомобилей, вывод автомобиля по id, обновления на основе классов. Добавить данные минимум о трех автомобилях. Должны быть реализованы контроллер (views) и шаблоны (temlates).

2.2.3 Работа с формами и представлениями

Формы на основе функций

Далее рассматривается процесс создания клиентской формы для разработанных в пункте 2.2.2 моделей.

1. Код forms.py:

```
from django import forms
from .models import ExampleModel


# creating a form
class ExampleForm(forms.ModelForm):

    # create meta class
    class Meta:
        # specify model to be used
        model = ExampleModel

        # specify fields to be used
        fields = [
            "title",
            "description",
        ]
```

2. Код представления (контроллера), основанного на функциях, для создания объектов в базе данных.

Код views.py:

```
from django.shortcuts import render

# relative import of forms
from .models import exampleModel
from .forms import ExampleForm # импортируем только-что созданную форму

def create_view(request):
    # dictionary for initial data
    with # field names as keys
    context ={}

    # add the dictionary during initialization
    form = ExampleForm(request.POST or None) # создание экземпляра
формы, передача в него данных из формы (из полей в браузере)
    if form.is_valid(): # проверка формы на корректность (валидация)
        form.save()
    context['form'] = form
    return render(request, "create_view.html", context)
```

3. Создание шаблона в templates/create_view.html.

```
<form method="POST" enctype="multipart/form-data">

    <!-- Security token -->
    {% csrf_token %}

    <!-- Using the formset -->
    {{ form.as_p }}

    <input type="submit" value="Submit">
</form>
```

4. Адресация.

Код urls.py:

```
from django.urls import path
# importing views from views..py
from .views import create_view


urlpatterns = [
    path('example_create', create_view),
]
```

5. Страница, получаемая по url-адресу http://localhost:8000/example_create/ (рисунок 30).

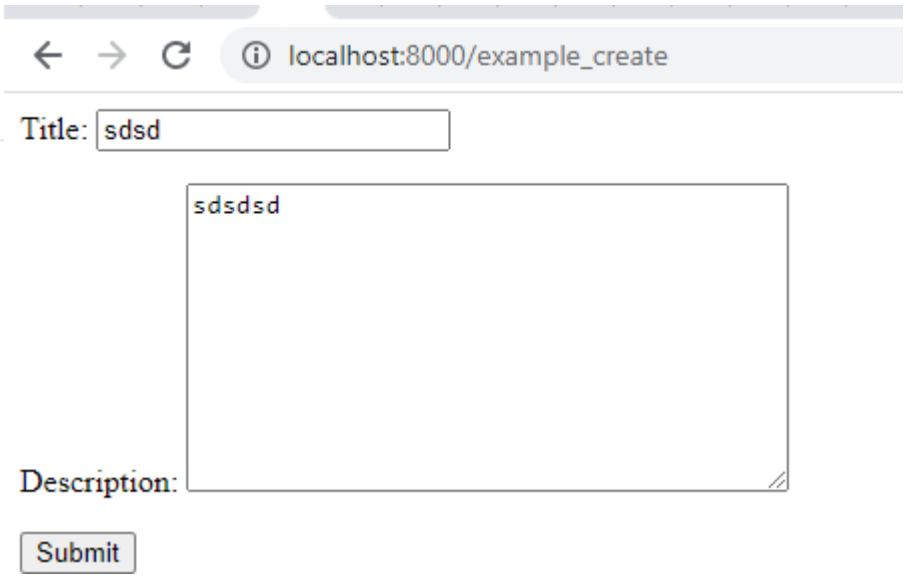


Рисунок 30 – Формы представления

Необходимо проверить работоспособность формы, при создании объекта (рисунок 31).

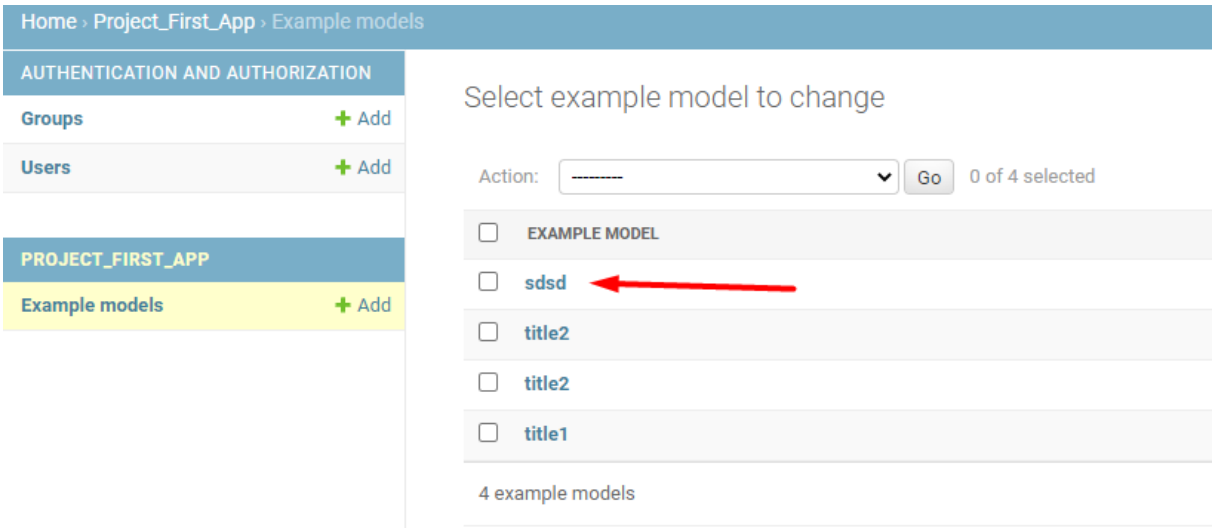


Рисунок 31 – Созданный через форму объект

Формы на основе классов

Обновление объекта по первичному ключу

Далее рассмотрен пример реализации представления, позволяющего обновлять данные с использованием UpdateView:

```
# views.py
from django.views.generic.edit import UpdateView

class PublisherUpdateView(UpdateView):
  model = Publisher
  fields = ['first_name', 'last_name', 'birthdate']
  success_url = '/publisher/list/'
```

В переменной fields указываются те поля модели, которые подлежат изменению, в переменной model – сама модель, а в последней переменной – тот URL, на который нужно перенаправить пользователя после успешного внесения изменений.

1. Создание шаблона:

```
<!-- publisher_form.html -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Create publisher</title>
</head>

<body>
  <form method="post">{% csrf_token %}
      {{ form.as_p }}
      <input type="submit" value="Save">
  </form>
</body>

</html>
```

2. Создание URL-адреса в urls.py:

```
# urls.py
from django.urls import path
from .views import *

urlpatterns = [
  path('publisher/<int:pk>/update/', PublisherUpdateView.as_view()),
]
```

URL-параметр <int:pk> указывает на то, какую запись следует отобрать для редактирования из БД.

Результат можно посмотреть в браузере по следующему URL-адресу: http://127.0.0.1:8000/1/update/ (рисунок 32).

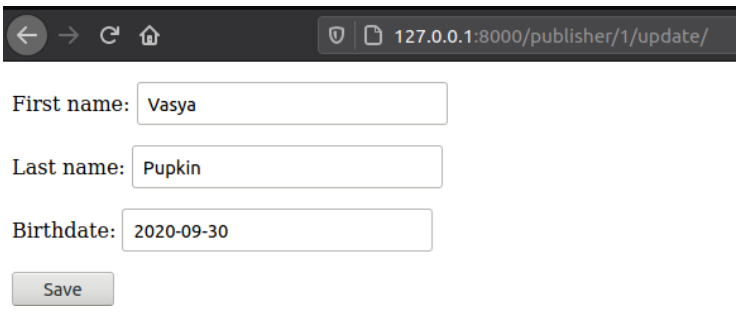


Рисунок 32 – Форма, получившаяся в результате использования представления на основе классов

Создание объекта

Пример 1 (модель: ExampleModel)

Представления на основе классов автоматически настраивают взаимодействие с формами. Необходимо указать, для какой модели создать представление Create для полей. Затем CreateView на основе классов автоматически попытается найти шаблон в app_name/modelname_form.html. В этом примере шаблон указан явно с помощью `template_name`.

1. Код views.py:

```
class ExampleCreate(CreateView):

    # specify the model for create view
    model = ExampleModel
    template_name = 'cvb_create_view.html'

    # specify the fields to be displayed

    fields = ['title', 'description']
```

2. Адресация в urls.py:

```
from django.urls import path

#   importing   views   from
views..py from .views import
exampleCreate urlpatterns =
[
    path('cvb_example_create',
ExampleCreate.as_view(success_url="/cvb_example/")), # success_url
позволяет указать адрес, на который необходимо перейти, если функция
выполнена корректно
```

3. Шаблон cvb_create_view.html:

```
<form method="POST" enctype="multipart/form-data">

    <!-- Security token -->
    {% csrf_token %}

    <!-- Using the formset -->
    {{ form.as_p }}

    <input type="submit" value="Submit">
</form>
```

4. Страница, получаемая при запуске http://localhost:8000/cvb_example_create (рисунок 33).

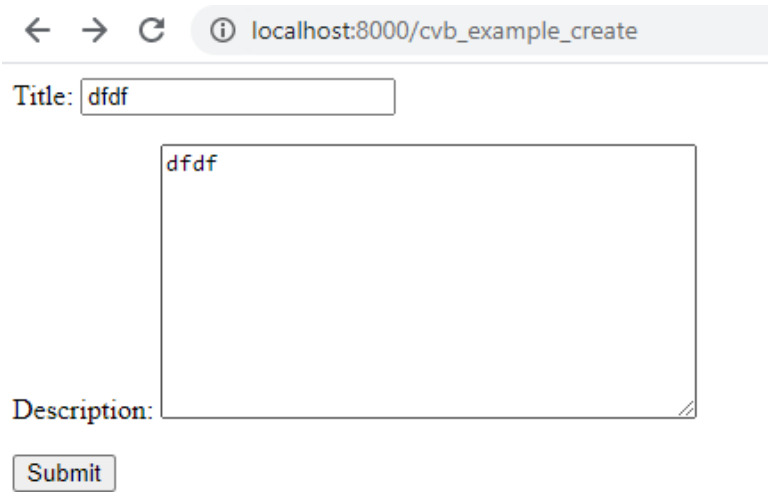


Рисунок 33 – Форма создания объекта

Пример 2 (модель: Publisher)

Создание объекта с использованием CreateView:

```
# views.py
from django.views.generic.edit import CreateView

class PublisherUpdateView(CreateView):
  model = Publisher
  fields = ['first_name', 'last_name', 'birthdate']
  success_url = '/publisher/list/'
```

В реализации создании объекта всё аналогично UpdateView, за исключением заполнения файла urls.py:

```
# urls.py
from django.urls import path
from .views import *

urlpatterns = [
  path('publisher/create/', PublisherCreateView.as_view()),
]
```

Результат можно посмотреть в браузере по следующему URL- адресу: http://127.0.0.1:8000/publisher/create/ (рисунок 34).

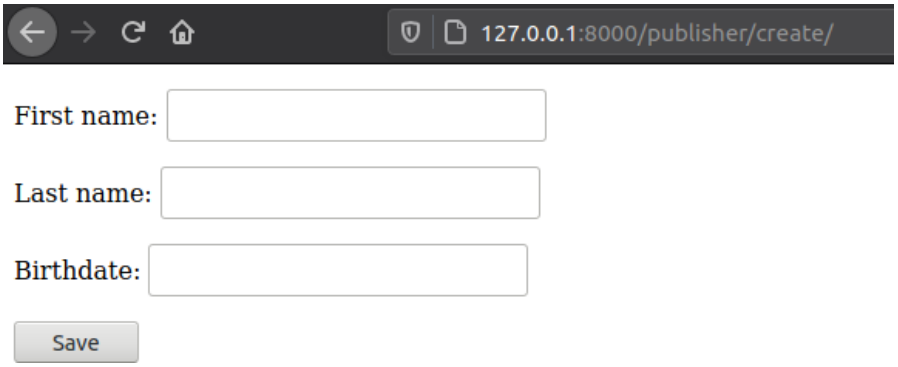


Рисунок 34 – Форма для создания объекта пользователя

Удаление объекта

1. Удаление объекта с помощью DeleteView:

```
# views.py
from django.views.generic.edit import DeleteView

class PublisherDeleteView(DeleteView):
  model = Publisher
  success_url = '/publisher/list/'
```

Создание шаблона:

```
<!-- publisher_confirm_delete.html -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Delete publisher</title>
</head>

<body>
  <form method="post">{% csrf_token %}
    <p>Are you sure you want to delete publisher "{{ object }}"?</p>
    <input type="submit" value="Confirm">
  </form>
  <br>
  <br>
  <a href="/publisher/list/">Get back</a>
</body>
</html>
```

2. Добавление необходимого адреса для представления с использованием URL-параметра <int:pk>:

```
# urls.py
from django.urls import path
from .views import *

urlpatterns = [
  path('publisher/<int:pk>/delete/', PublisherDeleteView.as_view()),
]
```

Результат можно посмотреть по следующему URL- адресу: http://127.0.0.1:8000/publisher/1/delete (рисунок 35).

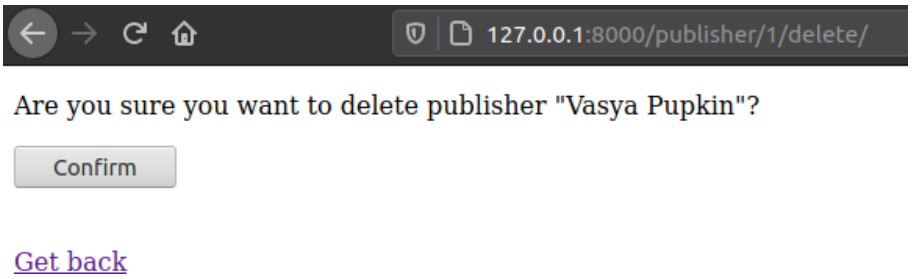


Рисунок 35 – Страница удаления объекта

Практическое задание 2.2.3

1. Реализовать форму ввода всех владельцев функционально. Добавить данные минимум о еще трех владельцах. Должны быть реализованы форма (Form), контроллер (views) и шаблоны (temlates).
2. Реализовать форму ввода, обновления и удаления всех автомобилей на основе классов. Добавить данные минимум о еще трех автомобилях. Должны быть реализованы форма (Form), контроллер (views) и шаблоны (temlates).

Практикум 2.3 Расширение пользовательской модели

В Django пользователь по умолчанию имеет ограниченной набор атрибутов. Почти в любом проекте возникает необходимость добавить новых атрибутов пользователю. Существует 4 способа сделать это [15]:

1. Простое расширение модели (proxy).

Эта стратегия без создания новых таблиц в базе данных. Используется, чтобы изменить поведение существующей модели (например, упорядочение по умолчанию, добавление новых методов и т.д.), не затрагивая существующую схему базы данных.

Можно использовать эту стратегию, когда не нужно хранить дополнительную информацию в базе данных, а просто необходимо добавить

дополнительные методы или изменит диспетчер запросов модели.

2. Использование связи один-к-одному с пользовательской моделью (user profiles).

Это стратегия с использованием дополнительной обычный модели Django со своей таблицей в базе данных, которая связана пользователем стандартной модели через связь OneToOneField.

Можно использовать эту стратегию, чтобы хранить дополнительную информацию, которая не связана с процессом аутентификации (например, дата рождения). Обычно это называется пользовательский профиль.

3. Расширение AbstractBaseUser.

Это стратегия использования совершенно новой модели пользователя, которая отнаследована от AbstractBaseUser. Требует особой осторожности и изменения настроек в settings.py. В идеале должно быть сделано в начале проекта, так как будет существенно влиять на схему базы данных.

Можно использовать эту стратегию, когда сайт имеет специфические требования в отношении процесса аутентификации. Например, в некоторых случаях имеет смысл использовать адрес электронной почты в качестве идентификации маркера вместо имени пользователя.

4. Расширение AbstractUser.

Это стратегия использования новой модели пользователя, которая отнаследована от AbstractUser. Требует особой осторожности и изменения настроек в settings.py. В идеале должно быть сделано в начале проекта, так как будет существенно влиять на схему базы данных.

Эту стратегию можно использовать, когда сам процесс аутентификации Django полностью удовлетворяет и не нужно его менять. Хотя, обычно менее трудозатратным является добавление некоторой дополнительной информацию непосредственно в модели пользователя, без необходимости создавать дополнительный класс (как в варианте 2)

К лучшим практикам следует отнести расширение средствами AbstractUser.

Расширение модели пользователя средствами AbstractUser

Следующий пример показывает, как добавить новые атрибуты для стандартного пользователя Django.

Необходимо создать таблицу модели для хранения новых полей пользователя.

В models.py:

```
from django.db import models
from django.contrib.auth.models import AbstractUser



class User(AbstractUser):
    name = models.CharField(max_length=100, blank=True, null=True)
```

Вторым этапом необходимо объявить созданную модель в settings.py:

```
AUTH_USER_MODEL = 'your_app.User'
```

Для использования своей модели пользователя в других таблицах в файле models.py, необходимо сослаться на эту модель с помощью `settings.AUTH_USER_MODEL` или `get_user_model ()` модуля авторизации Django.

```
from django.conf import settings
   from django.contrib.auth import get_user_model
   class Book(models.Model):
        author = models.ForeignKey(settings.AUTH_USER_MODEL)
# OR

from django.contrib.auth import get_user_model
   User = get_user_model()
   class Book(models.Model):
        author = models.ForeignKey(User)
```

Практическое задание 2.3

1. Сделать «Владельца автомобиля» пользователем и расширить модель пользователя его атрибутами так, чтобы о нем хранилась следующая информация:
   - номер паспорта;
   - домашний адрес;
   - национальность.
2. Отобразить новые поля пользователя в Django Admin. Отредактировать код из предыдущих работ, так, чтобы выводилась информация о пользователях.
3. Реализовать интерфейс создания пользователя с новыми атрибутами.

Лабораторная часть

Практическое задание

Реализовать сайт с использованием фреймворка Django 3 и СУБД PostgreSQL в соответствии с вариантом задания лабораторной работы.

Варианты индивидуальных заданий

1. Список отелей.

Необходимо учитывать название отеля, владельца отеля, адрес, описание, типы номеров, стоимость, вместимость, удобства.

Необходимо реализовать следующий функционал:

- Регистрация новых пользователей.
- Просмотр и резервирование номеров. Пользователь должен иметь возможность редактирования и удаления своих резервирований. Написание отзывов к номерам. При добавлении комментариев должны сохраняться период проживания, текст комментария, рейтинг (1–10), информация о комментаторе.
- Администратор должен иметь возможность заселить пользователя в отель и выселить из отеля средствами Django-admin.
- В клиентской части должна формироваться таблица, отображающая постояльцев отеля за последний месяц.

2. Доска домашних заданий.

О домашнем задании должна храниться следующая информация: предмет, преподаватель, дата выдачи, период выполнения, текст задания, информация о штрафах.

Необходимо реализовать следующий функционал:

- Регистрация новых пользователей.
- Просмотр домашних заданий по всем дисциплинам (сроки выполнения, описание задания).
- Сдача домашних заданий в текстовом виде.
- Администратор (учитель) должен иметь возможность поставить оценку за задание средствами Django-admin.
- В клиентской части должна формироваться таблица, отображающая оценки всех учеников класса.

3. Табло отображения информации об авиаперелетах.

Хранится информация о номере рейса, авиакомпании, отлете, прилете, типе (прилет, отлет), номере гейта.

Необходимо реализовать следующий функционал:

- Регистрация новых пользователей.
- Просмотр и резервирование мест на рейсах. Пользователь должен иметь

возможность редактирования и удаления своих резервирований.

- Администратор должен иметь возможность зарегистрировать на рейс пассажира и вписать в систему номер его билета средствами Django-admin.
- В клиентской части должна формироваться таблица, отображающая всех пассажиров рейса.

4. Написание отзывов к рейсам. При добавлении комментариев должны сохраняться дата рейса, текст комментария, рейтинг (1–10), информация о комментаторе.
5. Список туров туристической фирмы.

Хранится информация о названии тура, турагенстве, описании тура, периоде проведения тура, условиях оплаты.

Необходимо реализовать следующий функционал:

- Регистрация новых пользователей.
- Просмотр и резервирование туров. Пользователь должен иметь возможность редактирования и удаления своих резервирований.
- Написание отзывов к турам. При добавлении комментариев должны сохраняться даты тура, текст комментария, рейтинг (1–10), информация о комментаторе.
- Администратор должен иметь возможность подтвердить резервирование тура средствами Django-admin.
- В клиентской части должна формироваться таблица, отображающая все проданные туры по странам.

6. Список научных конференций.

Интерфейс описывает названия конференций, список тематик, место проведения, период проведения, описание конференций, описание место проведения, условия участия.

Необходимо реализовать следующий функционал:

- Регистрация новых пользователей.
- Просмотр конференций и регистрацию авторов для выступлений. Пользователь должен иметь возможность редактирования и удаления своих регистраций.
- Написание отзывов к конференциям. При добавлении комментариев должны сохраняться даты конференции, текст комментария, рейтинг (1–10), информация о комментаторе.
- Администратор должен иметь возможность указания результатов выступления (рекомендован к публикации или нет) средствами Django-

admin.

- В клиентской части должна формироваться таблица, отображающая всех участников по конференциям.

7. Табло победителей автогонок.

Табло должно отображать информацию об участниках автогонок: ФИО участника, название команды, описание автомобиля, описание участника, опыт и класс участника.

Необходимо реализовать следующий функционал:

a. Регистрация новых пользователей.

b. Просмотр автогонок и регистрацию гонщиков. Пользователь должен иметь возможность редактирования и удаления своих регистраций.

c. Написание отзывов и комментариев к автогонкам. Предварительно комментатор должен зарегистрироваться. При добавлении комментариев должны сохраняться даты заезда, текст комментария, тип комментария (вопрос о сотрудничестве, вопрос о гонках, иное), рейтинг (1–10), информация о комментаторе.

d. Администратор должен иметь возможность указания времени заезда и результата средствами Django-admin.

e. В клиентской части должна формироваться таблица всех заездов и результатов конкретной гонки.

8. Индивидуальный вариант (по согласованию с преподавателем).

Порядок выполнения и защиты работы

a. Вариант выбирается в соответствии с порядковым номером в журнале.

b. Работа выполняется индивидуально.

c. По результатам работы необходимо подготовить документацию средствами MkDocs. Описание работы с MkDocs доступно в практикуме 3.3.

d. Полученную программу разместить в основном репозитории группы согласно инструкции по загрузке работы на github в Лабораторной работе

i. Ссылку на документацию оставить в комментариях.

ЛАБОРАТОРНАЯ РАБОТА 3. РЕАЛИЗАЦИЯ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ СРЕДСТВАМИ DJANGO REST FRAMEWORK

Цель: овладеть практическими навыками и умениями реализации web-серверов и использования сокетов.

Оборудование: компьютерный класс.

Программное обеспечение: Python 3.6+, Django 3, PostgreSQL *, Django Rest framework 3+.

Практикум 3.1 Django Web framework. Запросы и их выполнение

3.1.1 Создание объектов

Реализация новых объектов в Django может осуществляться не только через панель администратора – в большинстве случаев создаются новые записи в БД через внутренние методы ORM. По логике реализации подобные запросы соответствуют SQL-запросам, но с учетом синтаксиса языка и фреймворка.

Запуск интерактивного режима

В дальнейшем все задания в этой работе будут реализовываться через интерактивный интерпретатор, запущенный посредством управляющего файла manage.py. Если используется IDE Pycharm Professional Edition и используемый проект реализуется проект через Шаблон Django, то интерпретатор (во вкладке python) автоматически запустится через управляющий файл фреймворка (рисунок 36).

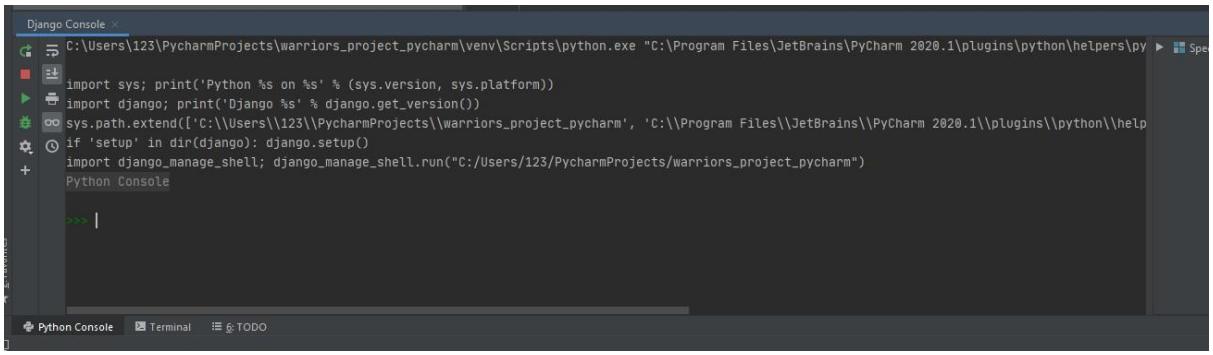


Рисунок 36 – Запуск интерактивного режима

Если взаимодействие с проектом реализуется через командную строку, то в папке разработки для открытия интерпретатора, поддерживающего Django нужно написать **manage.py shell** (для Windows) или **python3 manage.py shell** (для Linux):

```
(venv) C:\Users\123\PycharmProjects\warriors_project_pycharm>manage.py
shell
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

После открытия интерпретатора нужно импортировать классы моделей из файла models.py текущего приложения (рисунок 37).

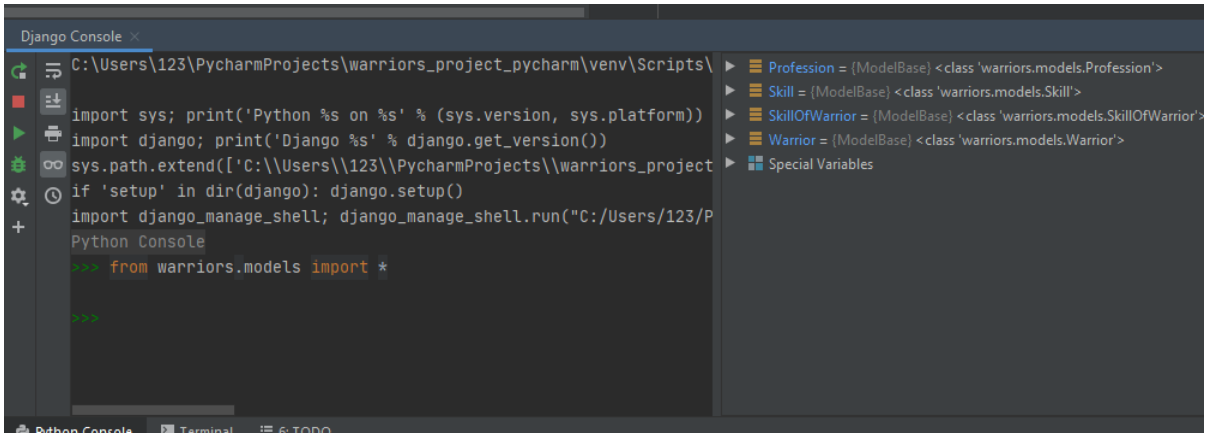


Рисунок 37 – Импорт классов моделей в интерактивном режиме

Далее необходимо создать объекты.

Создание объектов

Объекты в Django можно создавать двумя способами:

1. Реализацией записи таблицы в базе как класс (классом можно это считать условно, т.к. models.Model от которого наследуются все реализуемые модели использует иную логику поведения объекта на уровне метакласса). Такой метод создания, удобен, когда нужно хранить объект в памяти некоторое время динамически преобразуя его, не отправляя дополнительных запросов к базе. Когда нужно сохранить запись, у объекта вызывается метод *.save()*.

Пример:

```
short_warrior=Warrior(race="s", name="Николай Леонтьев", level=20)
short_warrior
<Warrior: Николай Леонтьев>
short_warrior.save()
```

Этот же метод используется для изменения уже существующего объекта, подробней об этом в следующем пункте раздела.

2. Реализация через менеджер *objects*. Любую запись можно сразу же создать в БД, если вызвать его класс через стандартный менеджер с методом *.create()* и передачи в последний полей как аргументы. Результат создания объекта можно также получить как объект класса для последующий внесений изменений через *.save()*.

Пример:

```
smart_warrior = Warrior.objects.create(race="s", name="Дмитрий Мартынов",
level=66)
smart_warrior
<Warrior: Дмитрий Мартынов>
```

Изменение и добавление many-to-many

Каждый созданный объект, сохраненный в памяти, можно впоследствии изменить. Для этого достаточно обратиться к столбцу таблицы как к полю класса и вызвать *.save()* для применений этих изменений в БД. Переведем воина на должность тимлида:

```
smart_warrior.race="t"
smart_warrior.save()
smart_warrior.race
't'
```

Подобным можно заниматься и при реализации отношений между объектами. В описываемом случае, необходимо создать новую many-to-many связь между коротким воином и новой способностью через ассоциативную сущность SkillOfWarrior:

```
new_skill = Skill.objects.create(title="Взлом компьютера")
skill_related = SkillOfWarrior.objects.create(skill=new_skill,
warrior=short_warrior, level=15)
skill_related.skill
<Skill: Взлом компьютера>
skill_related.warrior
<Warrior: Николай Леонтьев>
```

Как можно заметить, для добавления внешних таблиц использовались уже существующие объекты. Они точно также могут присваиваться к полям модели во время изменения объекта, как и другие значение, по примерам выше.

Чтобы добавить новые объекты для связи многие-ко-многим, также можно к полю, реализующему данное отношение, применить метод *.add()*, в аргументы которого передать все объекты, которые надо связать с исходным объектом. Кроме того, часто может встретиться ситуация, что есть некоторый итерируемый объект с записями, которые надо привязать. Тогда в метод *.add()* надо передать

подобный объект, указав перед ним оператор распаковки “*”.

Пример:

```
short_warrior.skill.add(java_skill)
short_warrior.skill
<django.db.models.fields.related_descriptors.create_forward_many_to_many_
manager.<locals>.ManyRelatedManager object at 0x0000022FBE31F358>
short_warrior.skill.all()
<QuerySet [<Skill: Взлом компьютера>, <Skill: Java programming>]>
```

Практическое задание 3.1.1

Используя модель практикума 2.1 (рисунок 38), написать запрос на создание 6 – 7 новых автовладельцев и 5 – 6 автомобилей, каждому автовладельцу назначить удостоверение и от 1 до 3 автомобилей.

Задание можно выполнить либо в интерактивном режиме интерпретатора, либо в отдельном python-файле. Результатом должны стать запросы и отображение созданных объектов.

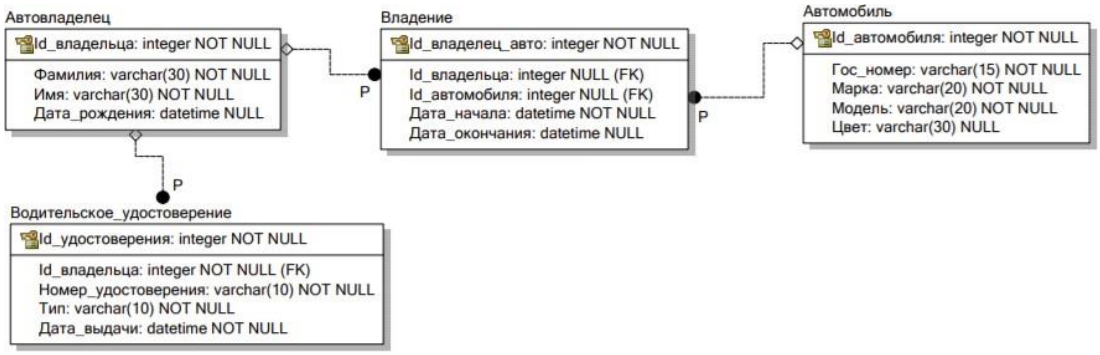


Рисунок 38 – Модель базы данных

3.1.2 Создание простых запросов

Любой DQL-запрос посредством ORM в Django может осуществить с помощью вызова менеджера и функций, реализующих фильтрацию или агрегацию. Получаемые объекты зависят от используемого метода, но чаще всего разработчик взаимодействует с итерируемыми объектами типа Queryset, предоставляемыми посредством функции *filter(),* или объектами класса модели типа app_name.models.YourModel, содержащимися в Queryset и получаемыми с помощью функции .get().

Фильтрация и получение объектов

Для выполнения какого-либо запроса необходимо через класс модели обратиться к менеджеру объекта. Manager – это интерфейс, через который для моделей Django предоставляются операции запросов к базе данных. У любой модели всегда есть как минимум один менеджер – *objects.* Но при желании можно реализовать и другие менеджеры [17].

Самым простым способом, как можно получить все объекты, является метод *all()*. Таким образом можно получить все объекты воинов:

```
Warrior.objects.all()
<QuerySet [<Warrior: Евгений Смирнов>, <Warrior: Николай Леонтьев>,
<Warrior: Дмитрий Мартынов>]>
```

Если нужно получить список объектов по какому-то определенному условию (аналог WHERE в SQL), можно воспользоваться методом *filter()*, позволяющим находить объекты по значениям в полях. Ниже представлен запрос на получение всех воинов, чей уровень равен 20:

```
Warrior.objects.filter(level=20)
<QuerySet [<Warrior: Николай Леонтьев>]>
```

Хоть нашелся только один объект, но возвращаемый объект Queryset по формату отображения намекает, что объект итерируемый и может хранить в себе более одного значения. Можно обратиться к 0-му элементу контейнера, чтобы показать, какого кого типа объекты хранит кварисет:

```
Warrior.objects.filter(level=20)[0]
<Warrior: Николай Леонтьев>
Warrior.objects.filter(level=20)[0].name
'Николай Леонтьев'
```

В примере отображено, что внутри такой коллекции хранится обычный объект модели.

Можно получить сразу же конкретную запись как объект класса, если воспользоваться функцией *.get()*. Синтаксис у нее аналогичен *.filter()*:

```
Warrior.objects.get(level=20)
<Warrior: Николай Леонтьев>
```

Django выдаст ошибку, если более чем один элемент соответствует запросу get(). В этом случае вызывается исключение MultipleObjectsReturned.

Помимо *.filter(),* Django имеет функцию фильтрации *.exclude().* Она производит операцию, обратную фильтру, и исключает все данные по запросу. Таким образом можно найти всех воинов, у которых уровень не равен 20:

```
Warrior.objects.exclude(level=20)
<QuerySet [<Warrior: Евгений Смирнов>, <Warrior: Дмитрий Мартынов>]>
```

Queryset и цепочки фильтров

Как было сказано выше, Queryset – это итерируемый объект, хранящий в себе записи из БД в виде реализации объектов класса модели. К любому Queryset'у можно применять все методы, описанные в предыдущем пункте. Данные методы будут применены исключительно уже к существующей выборке внутри фильтруемого объекта. Таким образом можно применять цепочки фильтров.

Ниже приведен пример на создание нескольких новых объектов с применением подобной цепочки:

```
powerful_warrior=Warrior.objects.create(name="Дмитрий Урбан", level="11",
race="d")
cute_warrior=Warrior.objects.create(name="Никита Михайловский",
level="15", race="s")
Warrior.objects.exclude(level=20).filter(race="s")
<QuerySet [<Warrior: Никита Михайловский>]>
```

Так были получены все воинов расы «Студент», чей уровень не равен 20. Очевидно, что подобные цепочки к Queryset-ам можно делать до бесконечности длинные – все зависит лишь от фантазии разработчика и целесообразности подобных действий.

Еще несколько фактов о подобном типе данных:

- При применении нового метода фильтрации создается новый уникальный Queryset.
- QuerySets ленивые – процесс создания QuerySet не связан с какими-либо действиями с базой данных, никаких запросов выполняться не будет без явной попытки получить объекты оттуда (например, проитерироваться).
- Аналог LIMIT в SQL для QuerySet – это обычное ограничение итерируемого объекта (например: *object[:5]).*

Функции, применяемые к полям, и поиск через отношения

Как и в SQL, в Django ORM есть большой набор различных критериев для фильтрации. Полный их список можно посмотреть в документации [17].

Чтобы применить какой-то дополнительный элемент фильтрации к полю, после его ввода необходимо дополнить значением “*__function_name*”.

Какие основные элементы поиска могут понадобиться?

- *contains/icontains* – поиск по вхождению в строку с учетом/без учета регистра соответственно. Пример: Все объекты с именем «Дмитрий»:

```
Warrior.objects.filter(name__contains="Дмитрий")
<QuerySet [<Warrior: Дмитрий Мартынов>, <Warrior: Дмитрий Урбан>]>
```

- *in* – вхождение поля в какой-либо итерируемый объект как отдельный элемент этого объекта. Пример: Все тимлиды и студенты:

```
Warrior.objects.filter(race__in=["s","t"])
<QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>,
<Warrior: Никита Михайловский>]>
```

- *gt/gte/lt/lte* – [Больше]/[Больше или равно]/[Меньше]/[Меньше или равно] для определенного значения. Пример: Все воины с уровнем больше или равным 20:

```
Warrior.objects.filter(level__gte=20)
<QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>]>
```

- *regex* – Применяет регулярное выражение к запросу на поиск: Пример: Имя воина, содержащее буквы только с A до C:

```
Warrior.objects.filter(name__regex=r'([A-C])\w+ \w+')
<QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>,
<Warrior: Дмитрий Урбан>, <Warrior: Никита Михайловский>]>
```

Такой синтаксис (нижнее подчеркивание после названия поля) можно применить не только к специализированным функциям для поиска - он также позволяет обращаться к полям связанных объектов, проводить по ним фильтрацию и применять функции к полям.

Ниже приведен пример создания нескольких объектов умений, с привязкой последних к объектам воинов:

```
gaming=Skill.objects.create(title="Умение играть во все подряд")
powerful_warrior.skill.add(gaming)
cute_warrior.skill.add(gaming)
```

Теперь необходимо получить всех воинов, обладающих таким умением и имеющих 15-ый уровень. Для этого объекты фильтруются по названию умения через обращение к полю skill:

```
Warrior.objects.filter(level=15, skill_title="Умение играть во все
подряд")
<QuerySet [<Warrior: Никита Михайловский>]>
```

Подобный «проброс полей» можно производить и дальше – если бы были какие-то дополнительные таблицы с внешними ключами в модели Skill, то можно было бы обратиться к полям и этой модели.

Такая логика фильтрации похожа на применение WHERE к какому-то SQL-запросу с JOIN из нескольких таблиц.

Не всегда нужно фильтровать по полям таблицы, на которую ссылается модель. Иногда нужно написать запрос, обращаясь к объекту, который, наоборот, ссылается на рассматриваемую таблицу. Специально для таких ситуаций в Django есть специальный аргумент для полей внешних ключей – *related name.* Он позволяет делать запрос к исходной таблице через имя, написанном в аргументе. Такое имя можно указать для таблицы SkillOfWarrior:

```
class SkillOfWarrior(models.Model):
    """
    Описание умений война
    """

    skill = models.ForeignKey('Skill', verbose_name='Умение',
    on_delete=models.CASCADE)
    warrior = models.ForeignKey('Warrior',
    verbose_name='Воин', related_name="warrior_skill",
    on_delete=models.CASCADE)
    level = models.IntegerField(verbose_name='Уровень освоения умения',
    blank=True, null=True)
```

Чтобы выполнить следующий запрос необходимо добавить уровень созданным объектам SkillOfWarrior, которые были созданы ранее при добавлении способностей двум новым воинам:

```
cute_skill=SkillOfWarrior.objects.get(skill=gaming, warrior=cute_warrior)
cute_skill.level=12
cute_skill.save()
powerful_skill=SkillOfWarrior.objects.get(skill=gaming,
warrior=powerful_warrior)
powerful_skill.level=22
powerful_skill.save()
```

Теперь при попытке получить воина с геймерским умением такого умения больше 13 получается следующий результат:

```
Warrior.objects.filter(warrior_skill_level_gt=13,
skill_title_icontains="играть")
<QuerySet [<Warrior: Дмитрий Урбан>]>
```

Практическое задание 3.1.2

По созданным в практикуме 3.1.1 данным написать следующие запросы на фильтрацию:

- Где это необходимо, добавить related_name к полям модели.
- Вывести все машины марки «Toyota» (или любой другой марки, которая у вас есть).
- Найти всех водителей с именем «Олег» (или любым другим именем на ваше усмотрение)
- Взяв любого случайного владельца, получить его id и по этому id получить экземпляр удостоверения в виде объекта модели (можно в 2 запроса).
- Вывести всех владельцев красных машин (или любого другого цвета).
- Найти всех владельцев, чей год владения машиной начинается с 2010 (или любой другой год, который присутствует в БД).

Модель БД приведена на рисунке 36.

3.1.3 Агрегация и аннотация запросов

Кроме основного инструментария для выполнения запросов, Django предоставляет различные возможности для нахождения информации по всему Queryset-у. С помощью функций *.annotate()* и *.aggregate()* можно выполнить большинство запросов аналогичных запросов со встроенными функциям в SQL (MAX, COUNT и т.д), а метод *.values()* позволит реализовывать аналоги запросов с GROUP BY.

Агрегация

Функция *.aggregate()* занимается агрегированием объектов базы данных. Ее можно применять как к менеджеру, так и к Queryset-у, но возвращает она словарь вида: {«имя функции/заданное имя»: «рассчитанное значение»}. Такой метод должен принимать в себя как минимум одну из стандартных функций, находящихся в репертуаре Django для агрегации. Это можно показать на примере подсчета среднего уровня воина:

```
from django.db.models import Avg
Warrior.objects.aggregate(Avg("level"))
{'level__avg': 22.8}
```

В агрегат можно передавать несколько таких функций, а самостоятельно

указывать имена полей. Например, так находится минимальный и максимальный уровень воина:

```
from django.db.models import Min, Max
Warrior.objects.aggregate(weakest_powerlevel=Min("level"),
highest_powerlevel=Max("level"))
{'weakest_powerlevel': 2, 'highest_powerlevel': 66}
```

По аналогии с запросами, функции агрегирования можно вызывать и для полей объектов, находящихся в отношении с моделью посредством цепочки имен любой глубины. Таким образом можно найти среднее значение уровня умений у воинов:

```
Warrior.objects.aggregate(Avg("warrior_skill__level"))
{'warrior_skill__level__avg': 16.333333333333332}
```

Стоит отметить, что агрегаты, как и подобные методы менеджеров, могут взаимодействовать с Queryset, т.е. всегда можно сначала отфильтровать выборку, а только потом проагрегировать полученный результат.

Аннотация

Иногда нужно создавать агрегаты не для самой модели, а для таблиц, находящихся в отношении с исходной моделью. В таком случае используется функция *.annotate()* к полю внешней таблицы (или самой этой таблицы в случае Count) с указанием метода агрегации, применяемого на функцию. При таком вызове создается Queryset с объектами моделей и новым полем, реализуемым по аналогии с агрегацией. Сам метод *.annotate()* вычисляет агрегационную функцию для каждого набора записей относящихся к одному экземпляру исходной модели, группируя таблицы. Такой метод можно применять и полям типа *ManyToMany* и к *related_filed.*

Для следующего примера необходимо получить список всех умений и привязать к новым воинам:

```
skills=Skill.objects.all()
cute_warrior.skill.add(*skills)
powerful_warrior.skill.add(*skills)
```

Теперь, необходимо выполнить запрос, который подсчитает конечное количество умений у каждого из воинов:

```
counter_skills=Warrior.objects.annotate(Count("skill"))
for warrior in counter_skills:
...     print(warrior.name, warrior.skill__count)
...
Николай Леонтьев 2
Дмитрий Мартынов 0
Дмитрий Урбан 5
Никита Михайловский 5
Евгений Смирнов 0
```

Как можно заметить, взаимодействие с объектами Queryset происходит как с обычными объектами моделей, у которых есть новое поле, сгенерированное посредством названия столбца и примененного метода.

Теперь можно получить минимальный уровень каждого умения у воинов:

```
min_level_skills=\
Warrior.objects.annotate(min_lvl=Min("warrior_skill_level"))
for warrior in min_level_skills:
...     print(warrior.name, warrior.min_lvl)
...
Николай Леонтьев 15
Дмитрий Мартынов None
Дмитрий Урбан 22
Никита Михайловский 12
Евгений Смирнов None
```

Группировка посредством *.values()* и упорядочивание

При выполнении запросов рано или поздно нужно воспользоваться группировкой посредством GROUP BY. В Django метод для группировки именуется как *.values()*. Этот метод обязательно вызывать вместе с *.annotate()* в позиции перед последним; values() принимает в качестве параметров названия полей, а возвращает словарь со значением поля и вычисленной агрегационной функцией. Таким образом, группировку можно производить не только по полям основной модели, но и по полям связанных таблиц, а аннотирование, наоборот, становится валидным для полей основной таблицы.

Так, можно составить запрос с группировкой, где подсчитывается количество воинов разных рас:

```
Warrior.objects.values("race").annotate(Count("id"))
<QuerySet [{'race': 'd', 'id__count': 2}, {'race': 's', 'id__count': 2},
{'race': 't', 'id__count': 1}]>
```

Помимо группировки, Django также предоставляет метод сортировки по полям. Называется такой метод *.order_by()*. Он позволяет упорядочить Queryset как по полю основной модели, так и по полям связанных таблиц. Может быть

вызвана через менеджер или через созданный Queryset и возвращает все тот же, но уже упорядоченный Queryset .

Сортировка воинов по их уровню:

```
 Warrior.objects.order_by("level")
<QuerySet [<Warrior: Евгений Смирнов>, <Warrior: Дмитрий Урбан>,
<Warrior: Никита Михайловский>, <Warrior: Николай Леонтьев>, <Warrior:
Дмитрий Мартынов>]>
```

Практическое задание 3.1.3:

1. Реализовать следующие запросы:
   - Вывести даты выдачи самого старшего водительского удостоверения.
   - Указать самую позднюю дату владения машиной, имеющую какую-то из существующих моделей в вашей базе.
   - Вывести количество машин для каждого водителя.
   - Подсчитать количество машин каждой марки.
2. Отсортировать всех автовладельцев по дате выдачи удостоверения.

   Примечание: чтобы не выводить несколько раз одни и те же таблицы, воспользоваться методом *.distinct()*.

Модель БД приведена на рисунке 38.

Практикум 3.2 Django Rest framework

3.2.1. Restful и аналоги

REST – это акроним от Representational state transfer (передача состояния представления).

Сервер может считаться RESTful, если он соответствует принципам REST.

Принципы REST

1. Независимость от состояния (Statelessness)

Сервер не должен запоминать состояние пользователя между запросами – в каждом запросе передается информация, идентифицирующая пользователя (например, token, полученный через OAuth-авторизацию) и все параметры, необходимые для выполнения операции.

Независимость от состояния означает, что данные, возвращаемые определенным вызовом API, не должны зависеть от вызовов, сделанных ранее.

2. Многоуровневая архитектура (Layered System)

Многоуровневая архитектура означает, что клиенту всё равно, является ли отвечающий сервер конечным в цепочке или нет. Это даёт отличную возможность для кэширования и распределения нагрузки. Кроме того, многоуровневая архитектура говорит и о том, что внутри сервера всё делится на компоненты, которые не обязаны быть связаны напрямую друг с другом. Например, есть метод, который пишет какие-то данные в файл, и есть метод, который записывает что-то в базу данных. Они могут быть не связаны друг с другом, но при этом вызываются из третьей компоненты, которая и содержит в себе логику того, как они должны взаимодействовать.

3. Единый унифицированный программный интерфейс

Например, для получения списка фильмов используется URL вида/videos.com/movies, а для получения информации о конкретном фильме URL –/videos.com/movies/1.

4. Кэшируемая архитектура (Cacheable)

Ответ сервера может быть кэширован на определенный период времени и использоваться повторно без новых запросов к серверу. Это касается, в основном, GET-запросов, потому что POST-запросы кэшировать было бы довольно странно.

5. Удобное представление данных

В качестве представления данных объекта передаются данные в формате JSON или XML. Сейчас почти везде используется JSON, который де-факто уже стал стандартом для обмена данными между клиентом и сервером. XML, хотя и даёт возможность структурировать информацию лучше, чем JSON, но при этом является слишком “раздутым” на выходе. Обмен данными в формате XML является слишком медленным по сравнению с JSON.

HTTP-методы и их обозначения:

- GET — для получения (чтение)
- POST — для создания
- PUT — для изменения
- PATCH — для частичного изменения
- DELETE — для удаления

Статус коды указывают на результат HTTP запроса.

Статус-коды HTTP:

- 1XX — информационный
- 2XX — успешное выполнение
- 3XX — перенаправление

- 4XX — ошибка клиента
- 5XX — ошибка сервера

Аналоги REST

До REST использовался протокол SOAP (Simple Object Access Protocol – простой протокол доступа к объектам). По сути, SOAP является расширением XML-RPC (RPC – Remote Procedure Call, Удалённый вызов процедур). Обычно его используют поверх HTTP-протокола. Главный минус SOAP в том, что XML увеличивает объём передаваемых данных, а значит, снижает скорость их обработки.

Структура SOAP-сообщения выглядит так [18]:

- **Envelope** — корневой элемент, который определяет сообщение и пространство имен, используемое в документе.
- **Header** — содержит атрибуты сообщения, например: информация о безопасности или о сетевой маршрутизации.
- **Body** — содержит сообщение, которым обмениваются приложения.
- **Fault** — необязательный элемент, который предоставляет информацию об ошибках, которые произошли при обработке сообщений.

Пример SOAP-запроса:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetails xmlns="http://warehouse.example.com/ws">
       <productID>12345</productID>
     </getProductDetails>
   </soap:Body>
</soap:Envelope>
```

Пример ответа:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productID>12345</productID>
         <productName>Стакан граненый</productName>
         <description>Стакан граненый. 250 мл.</description>
         <price>9.95</price>
         <currency>
             <code>840</code>
             <alpha3>USD</alpha3>
             <sign>$</sign>
             <name>US dollar</name>
             <accuracy>2</accuracy>
         </currency>
         <inStock>true</inStock>
       </getProductDetailsResult>
     </getProductDetailsResponse>
   </soap:Body>
</soap:Envelope>
```

У REST тоже есть свои болевые точки. Как правило, когда об этом говорят, упоминают обычно то, что приходится делать довольно большое количество запросов, чтобы добиться нужного эффекта. Например, API отдаёт список постов из социальной сети с лайками и комментариями, в которых нужно ещё вложить ссылки на аватарки пользователей, кроме самой информации о комментариях, но у разных клиентов разная скорость загрузки, поэтому в мобильном приложении, к примеру, такие ответы будут загружаться дольше, чем на десктопе, из чего следует потребность появления второго эндпоинта, который будет возвращать посты без лайков и комментариев, а те, в свою очередь, будут подгружаться по необходимости. Таким образом, клиенту придётся делать два запроса вместо одного. Для решения подобных проблем в Facebook был придуман синтаксис описания запросов GraphQL, который нужен для того, чтобы клиент сам говорил серверу о том, какие данные ему нужны, а сервер, в свою очередь, отдавал ему именно то, что нужно и в том формате, в котором нужно. Кроме того, GraphQL является типизированным, что позволяет исключить ошибки в непонимании типов данных, которые приходят с сервера.

В общих чертах, клиент-серверное взаимодействие в GraphQL строится следующим образом (рисунок 39).

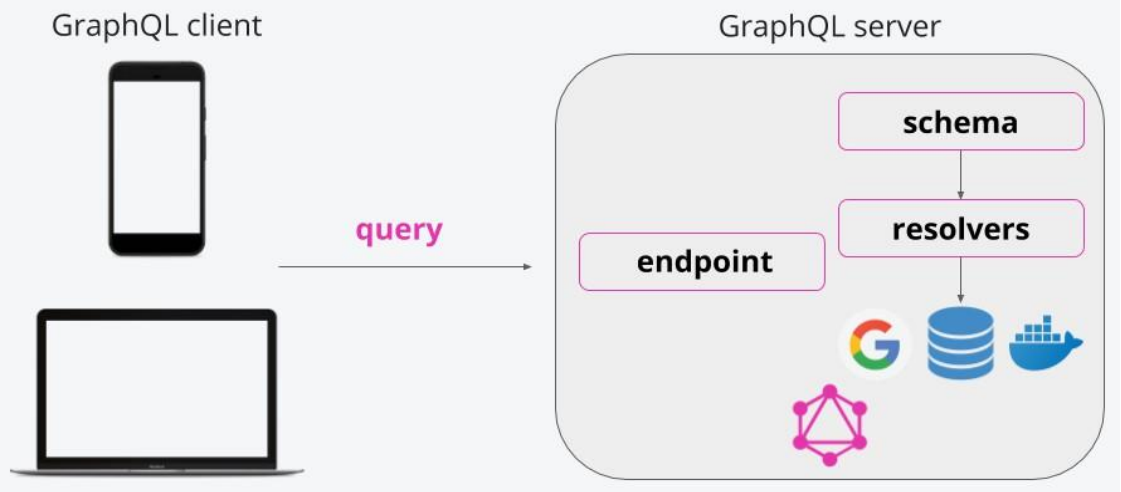


Рисунок 39 – Клиент-серверное взаимодействие в GraphQL

Сам GraphQL сервер состоит из трёх компонентов: Запрос, Распознаватель, Схема. Схема отвечает за структуру данных, над которыми происходят операции, она является типизированной. Распознаватели отвечают за то, откуда нужно получать те или иные данные, а запрос то, в каком виде эти данные хочет клиент.

Пример Запроса представлен на рисунке 40.

```
query {
  post(id: "123foo"){
    title
    body
    author{
      name
      avatarUrl
      profileUrl
    }
  }
}
```

Рисунок 40 – Пример запроса в GraphQL

Пример Распознавателя представлен на рисунке 41.

```
Query: {
  post(root, args) {
    return Posts.find({ id: args.id });
  }
},
Post: {
  author(post) {
    return Users.find({ id: post.authorId})
  }
}
```

Рисунок 41 – Распознаватель GraphQL

Пример Схемы представлен на рисунке 42.

```
export const typeDef = `
  extend type Query {
    author(id: Int!): Author
  }

  type Author {
    id: Int!
    firstName: String
    lastName: String
    books: [Book]
  }
`;
```

```
export const resolvers = {
  Query: {
    author: () => { ... },
  },
  Author: {
    books: () => { ... },
  }
};


// Modularizing your GraphQL
// schema code
```

Рисунок 42 – Схема GraphQL

3.2.2 Описание установки библиотеки Django REST Framework в фреймворк Django

В рамках выполнения лабораторной работы 2 были изучены основы работы с Django web framework.

Django – это высокоуровневый веб-фреймворк для языка Python, позволяющий быстро разрабатывать динамические веб-приложения. Изначально Django был разработан для написания новостных сайтов, поэтому основным требованием было решение двух проблем:

- регулярное интенсивное наполнение сайта материалом по требованию

редакции;

- соответствие высоким стандартам качества сайта.

В результате появился удобный фреймворк, который существенно ускоряет процесс разработки и дает возможность разделить процессы программирования, верстки и дизайна. По сути, Django можно назвать каркасом веб-приложений.

Для чего нужен Django REST Framework?

Как понятно из названия, Django REST Framework – это лёгкий фреймворк для Django, поддерживающий идеологию REST (Representational State Transfer) – репрезентативную передачу состояния. Использование этого фреймворка позволяет легко стандартизировать запросы к базе данных и одновременно создавать RESTful WEB API сайта.

Особенности:

- Отображение ресурсов с использованием новой функции Django – Class Based Views.
- Поддержка ModelResources и проверка входных данных.
- Наличие подключаемых парсеров, отображений, авторизации и прав доступа – все легко настраивается.
- Указание типа материала с использованием Access в заголовках HTTP-запросов.
- Поддержка форм с возможностью проверки.

Ниже описан процесс создания простейшего приложения для демонстрации скорости разработки с использованием Django REST Framework:

1. Необходимо установить Django и Django Rest Framework.

```
pip install django
```

2. Далее, необходимо установить Django Rest Framework (DRF) для создания API.

```
pip install djangorestframework
```

3. Теперь все готово для того, чтобы создать первый DRF API.

Для этого нужно создать проект Django для работы. Для этого необходимо запустить следующую команду:

```
django-admin startproject warriors_project
```

4. Затем необходимо запустить миграцию базы данных. Таким образом будет создана база данных по умолчанию (SQLite), со всеми требуемыми таблицами:

```
python manage.py migrate
```

5. Теперь можно запустить сервер, чтобы убедиться, что все в порядке:

```
python manage.py runserver
```

6. Имеется проект Django, в который возможно интегрировать Django rest framework.

Необходимо открыть файл settings.py и добавить в установленные приложения INSTALLED_APPS = […] строку rest_framework.

```
INSTALLED_APPS = [
    'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'rest_framework'
]
```

7. Далее необходимо создать проект:

```
python manage.py startapp warriors_app
```

8. Потом необходимо включить созданное приложение в список установленных приложений:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'warriors_app'
]
```

3.2.3 Описание модели данных для реализации примеров

В рамках приложения «Воины» будут рассматриваться модели воинов, имеющие личные профессии и множество уникальных навыков. доступных всем воинам:

```
class Warrior(models.Model):
    """
    Описание война
    """
    race_types = (
        ('s', 'student'),
        ('d', 'developer'),
        ('t', 'teamlead'),
    )
    race = models.CharField(max_length=1, choices=race_types,
verbose_name='Расса')
    name = models.CharField(max_length=120, verbose_name='Имя')
    level = models.IntegerField(verbose_name='Уровень', default=0)
    skill = models.ManyToManyField('Skill', verbose_name='Умения',
through='SkillOfWarrior',
                                   related_name='warrior_skils')
    profession = models.ForeignKey('Profession',
    on_delete=models.CASCADE,
 verbose_name='Профессия',
                                   blank=True, null=True)
 class
    Profession(models.Model):
    """
 Описание профессии
    """
    title = models.CharField(max_length=120, verbose_name='Название')
    description = models.TextField(verbose_name='Описание')
 class
    Skill(models.Model):
    """
    Описание умений
    """
    title = models.CharField(max_length=120, verbose_name='Наименование')
    def __str__(self):
        return self.title
class SkillOfWarrior(models.Model):
    """
    Описание умений воина
    """
    skill = models.ForeignKey('Skill', verbose_name='Умение',
    on_delete=models.CASCADE)
    warrior = models.ForeignKey('Warrior', verbose_name='Воин',
on_delete=models.CASCADE)
    level = models.IntegerField(verbose_name='Уровень освоения умения')
```

3.2.4 Представления на основе Apiview

Далее рассмотрены примеры работы с Apiview.

В первую очередь, необходимо реализовать метод для просмотра всех воинов. Для этого в файле warriors_app/views.py необходимо реализовать следующий код:

```
class WarriorAPIView(APIView):
    def get(self, request):
        warriors = Warrior.objects.all()
        serializer = WarriorSerializer(warriors, many=True)
        return Response({"Warriors": serializer.data})
```

Далее необходимо настроить сериализацию данных из Django к формату JSON. Для этого необходимо создать файл serializers.py и настроить в нем работу сериализаторов. Используемые в файле контроллеров сериализаторы необходимо импортировать в файл views.py:

```
from rest_framework import serializers
from .models import *


class WarriorSerializer(serializers.ModelSerializer):

    class Meta:
        model = Warrior
        fields = "__all__"
```

Далее необходимо создать URL-адрес, с которого пользователь сможет получить доступ к этому методу.

Необходимо создать файл warriors_app/urls.py и вставить следующий код:

```
from django.urls import path
from .views import *


app_name = "warriors_app"


urlpatterns = [
    path('/warriors/', WarriorAPIView.as_view()),
```

Необходимо импортировать url-файл приложения в проект. В папке проекта изменить файл urls.py:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('war/', include('warriors_app.urls')),
]
```

В итоге по объявленному адресу будет доступна следующая страница (рисунок 43).

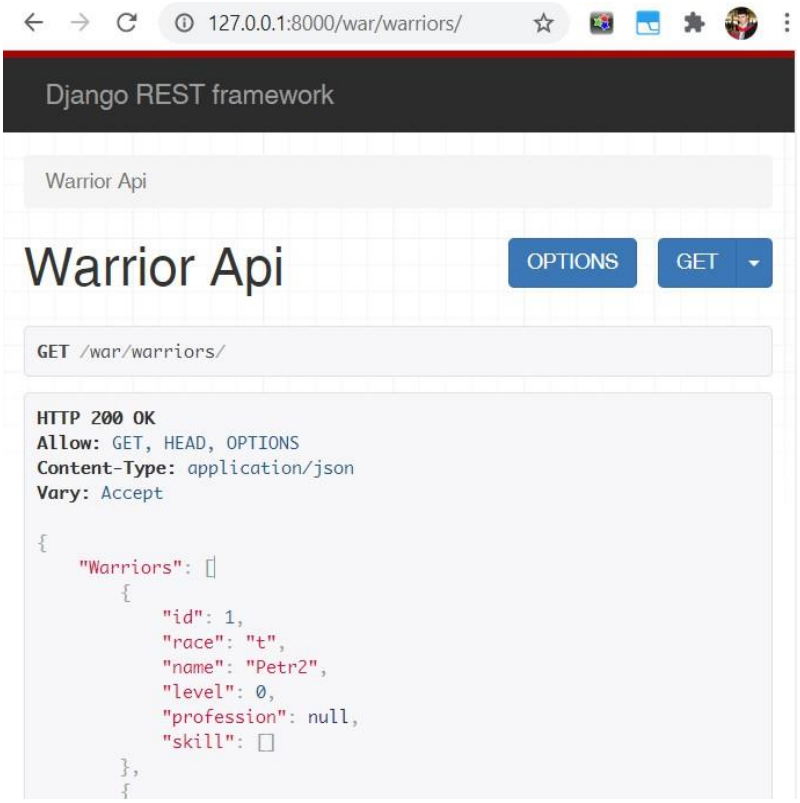


Рисунок 43 – Вывод списка воинов через APIView-представление

На данный момент реализован API, через который можно просматривать информацию обо всех войнах в базе данных.

В примере выше был рассмотрен метод GET, позволяющий получить данные.

Далее рассмотрен метод POST, позволяющий отправить данные на сервер.

Код файла warriors_app/views.py:

```
class ProfessionCreateView(APIView):

   def post(self, request):
       profession = request.data.get("profession")
       serializer = ProfessionCreateSerializer(data=profession)

       if serializer.is_valid(raise_exception=True):
           profession_saved = serializer.save()

       return Response({"Success": "Profession '{}' created
succesfully.".format(profession_saved.title)})
```

Код файла urls.py:

```
path('profession/create/', ProfessionCreateView.as_view()),
```

Работоспособность можно проверить в браузере по указанному выше адресу

(рисунок 44).

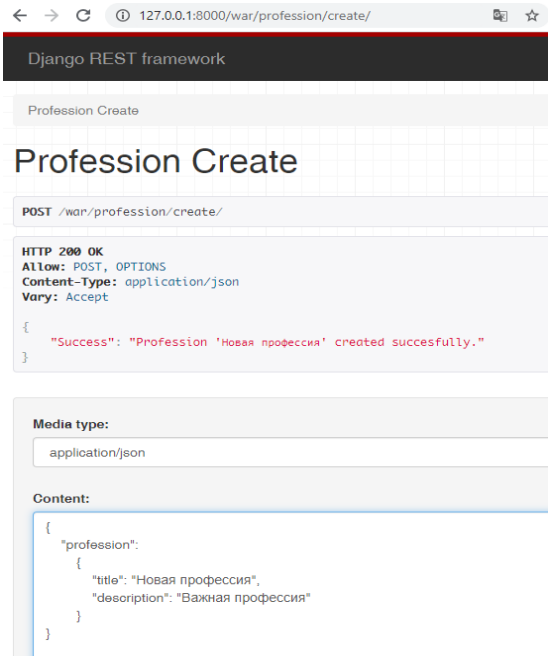


Рисунок 44 – Представление на создание объекта

Практическое задание 3.2.4

Реализовать ендпоинты для добавления и просмотра скилов методом, описанным в 3.2.4.

3.2.5 Сериализация

Сериализация – это процесс перевода какой-либо структуры данных в последовательность битов.

В зависимости от применяемого набора правил, на выходе получается некая текстовая строка, которая будет оформлена по каким-то правилам. Это набор правил называется форматом.

Обычно данные представляются в формате JSON (JavaScript Object Notation).

JSON – это удобный формат, потому что в большинстве ЯП имеется библиотека для парсинга JSON, а соответственно и его перевода в структуры данных этого ЯП.

Соотношение типов данных из python с типами из JSON представлены в таблице 1.

Таблица 1 – Соотношение типов данных из python с типами JSON

Python	**JSON**
dict	Object

Python	JSON
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

Процесс сериализации данных необходим для того, чтобы представить данные в виде, который необходим конечному клиенту. Например, клиент приходит в магазин купить стул. Его не слишком волнуют габариты стула, он знает только примерно, что хочет высокий стул с мягкими подлокотниками тёмного цвета и не дороже 5000 рублей. Можно сказать, что клиент выполняет с помощью продавца запрос на выборку стульев по указанным параметрам. Продавец получает информацию от клиента и подготавливает список подходящих стульев. Продавец, опираясь на формат описания стульев, понятный обычным клиентам, а не продавцам стульев, подготавливает коммерческое предложение с характеристиками стульев. В данном примере перевод данных из формата продавца в формат клиента называется сериализацией.

Сериализаторы в Django REST Framework

Самый простой и низкоуровневый класс для сериализации данных в DRF выглядит следующим образом:

```
class ProfessionCreateSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=120)
    description = serializers.CharField()


    def create(self, validated_data):
       profession = Profession(**validated_data)
       profession.save()
      return Profession(**validated_data)
```

В таком сериализаторе обязательно нужно указать те же поля, что и в описываемой модели со всеми ограничениями. Далее можно задать различные

методы, например, при попытке создания записи в БД через этот сериализатор данные сначала будут провалидированы (здесь уместна аналогия с формами).

Представление, которое позволит работать с этим сериализатором:

```
class ProfessionCreateView(APIView):
    def post(self, request):
      profession = request.data.get("profession")

      serializer = ProfessionCreateSerializer(data=profession)

      if serializer.is_valid(raise_exception=True):
           profession_saved = serializer.save()

      return Response({"Success": "Profession '{}' created
succesfully.".format(profession_saved.title)})
```

При получении данных из POST-запроса, представление перенаправит их в сериализатор, который должен будет вернуть их в правильном виде и при этом, можно будет узнать, прошли ли они валидацию. В том случае, если валидация пройдена успешно, данные будут записаны в БД.

Более простой, с точки зрения количества кода, сериализатор, который наследуется от класса ModelSerializer:

```
class ProfessionSerializer(serializers.ModelSerializer):
     class Meta:
          model = Profession
          fields = ["title", "description"]
```

Для его корректной работы нужно указать модель, к которой ему следует обратиться за данными, и поля, которые нужно вывести. Если есть необходимость вывести все поля, можно использовать ключевое слово “__all__”:

```
class WarriorSerializer(serializers.ModelSerializer):
    class Meta:
      model = Warrior
      fields = "__all__"
```

Удобства на этом не заканчиваются, потому что сериализаторы позволяют возвращать данные и по связным таблицам, пример для ManyToManyField:

```
class SkillRelatedSerializer(serializers.ModelSerializer):
    warrior_skils = WarriorSerializer(many=True)

    class Meta:
      model = Skill
      fields = ["title", "warrior_skils"]
```

Причём такой сериализатор может равно работать в обе стороны:

```
class WarriorRelatedSerializer(serializers.ModelSerializer):
    skill = serializers.SlugRelatedField(read_only=True, many=True,
slug_field='title')
    # skill = serializers.StringRelatedField(read_only=True, many=True)

    class Meta:
      model = Warrior
      fields = "__all__"
```

Необходимо отметить отличие. В этом сериализаторе используется SlugRelatedField, который позволяет обратиться к определённому полю из связной таблицы. Кроме этого, можно использовать и StringRelatedField: он будет ориентироваться на то, что возвращает функция __str__в модели.

Получение информации по связным таблицам будет работать и через указание глубины дерева данных, которое возвращает сериализатор:

```
class WarriorDepthSerializer(serializers.ModelSerializer):

    class Meta:
      model = Warrior
      fields = "__all__"

      # добавляем глубину
      depth = 1
```

Такой подход плох тем, что нет возможности указать необходимые поля в связных таблицах и глубина не всегда может быть точно определена.

Чтобы решить эти проблемы, следует обратиться к наследуемым сериализаторам:

```
class WarriorNestedSerializer(serializers.ModelSerializer):
    # делаем наследование
    profession = ProfessionSerializer()
    skill = SkillSerializer(many=True)


    # уточняем поле
    race = serializers.CharField(source="get_race_display",
read_only=True)

    class Meta:
      model = Warrior
      fields = "__all__"
```

Здесь сразу несколько интересных примеров: идёт обращение к сериализатору скиллов и сериализатору профессии, причём по скиллам будут получены все связные записи. Кроме того, внутри сериализатора есть возможность переопределения значения поля, что и было сделано с полем race, в котором в качестве аргумента source был передан метод get_race_display, необходимый для того, чтобы вытащить значение из choices, а не возвращать только ключ.

3.2.6 Generic классы

Общие представления Django построены на основе базовых, и были созданы для стандартных ситуаций, например, получить объект и отрендерить его страницу. Они позволяют легко решить стандартные задачи. Generic классы скрывают от разработчика часть бизнес-логики представления, позволяя писать меньше кода.

Можно сказать, что Generic API View позволяют быстро создавать Rest API.

Это views.py из предыдущей части инструкции:

```
class WarriorAPIView(APIView):
    def get(self, request):
        warriors = Warrior.objects.all()
        serializer = WarriorSerializer(warriors, many=True)
        return Response({"Warriors": serializer.data})

class ProfessionCreateView(APIView):

    def post(self, request):
        print("REQUEST DATA", request.data)
        profession = request.data.get("profession")
        print("PROF DATA", profession)

        serializer = ProfessionCreateSerializer(data=profession)

        if serializer.is_valid(raise_exception=True):
            profession_saved = serializer.save()

        return Response({"Success": "Profession '{}' created
succesfully.".format(profession_saved.title)})
```

Это длинный сценарий для создания CRUD-системы. С помощью Generic API View можно написать более короткий сценарий с той же функциональностью. Далее переписаны классы WarriorListAPIView и ProfessionCreateAPIView в файле views.py приложения:

```
class WarriorListAPIView(generics.ListAPIView):
    serializer_class = WarriorSerializer
    queryset = Warrior.objects.all()


class ProfessionCreateAPIView(generics.CreateAPIView):
    serializer_class = ProfessionCreateSerializer
    queryset = Profession.objects.all()
```

Теперь для описания каждого класса используется всего 3 строчки кода.

ListAPIView позволяет реализовать ендпоинты для просмотра коллекции экземпляров модели.

CreateAPIView используется для ендпоинтов на создание объектов.

Далее необходимо отредактировать настройки роутинга и URL-адреса в файле urls.py приложения:

```
urlpatterns = [
    ...
    path('warriors/list/', WarriorListAPIView.as_view()),
    path('rofession/generic_create/
', ProfessionCreateAPIView.as_view()),
    ...
]
```

Затем необходимо проверить работоспособность написанного кода.

Интерфейс просмотра списка воинов представлен на рисунке 45.

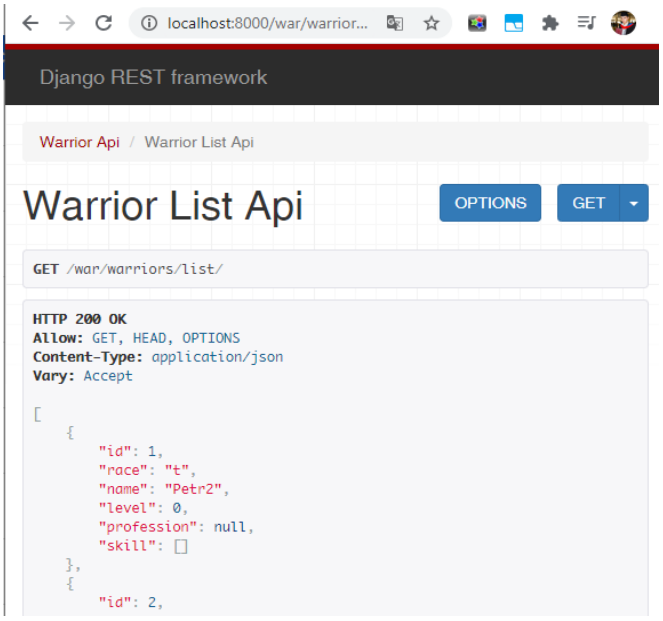


Рисунок 45 – Вывод списка воинов через представление, наследуемое от ListAPIView

Интерфейс для создания профессий представлен на рисунке 46.

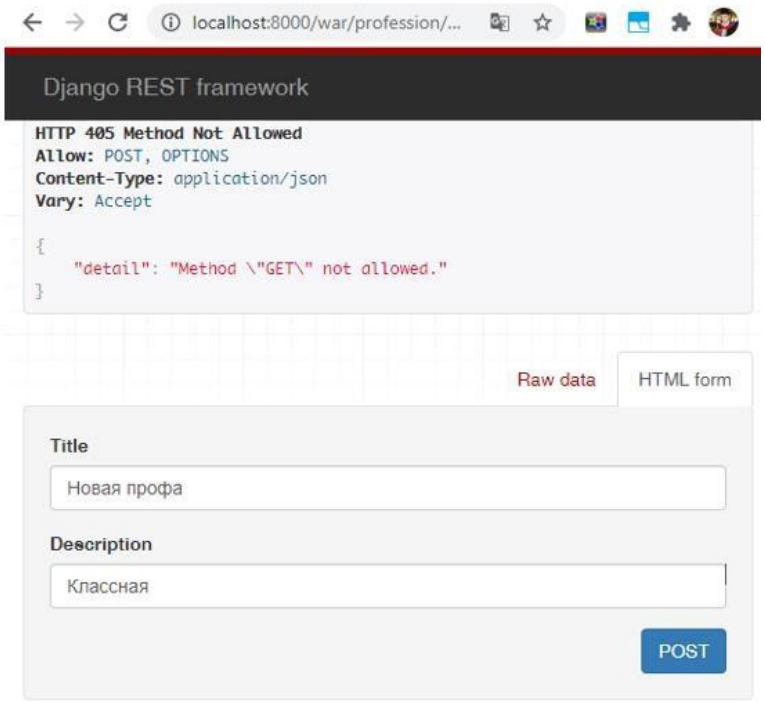


Рисунок 46 – Создание объекта профессий через CreateAPIView

Существуют и другие generic-методы:

- CreateAPIView

  Используется для создания объектов.

  Предоставляет post обработчик метода.

  Расширяется: GenericAPIView , CreateModelMixin

- ListAPIView

Используется для ендпоинтов, доступных только для чтения, для представления коллекции экземпляров модели .

Предоставляет get обработчик метода.

Расширяется: GenericAPIView , ListModelMixin

- GetAPIView

Используется для ендпоинтов, доступных только для чтения, для представления одного экземпляра модели .

Предоставляет get обработчик метода.

Расширяется: GenericAPIView , RetrieveModelMixin

- DestroyAPIView

Используется для ендпоинтов только для удаления для одного экземпляра

модели.

Предоставляет delete обработчик етода.

Расширяется: GenericAPIView, DestroyModelMixin

- UpdateAPIView

Используется для ендпоинтов только для обновления для одного экземпляра модели.

Предоставляет put и patch обработчики методов.

Расширяется: GenericAPIView , UpdateModelMixin

- ListCreateAPIView

Используется для ендпоинтов чтения-записи для представления коллекции экземпляров модели.

Предоставляет get и post обработчики методов.

Расширяется: GenericAPIView , ListModelMixin , CreateModelMixin

- RetrieveUpdateAPIView

Используется для чтения или обновления одного экземпляра модели.

Обеспечивает get, put и patch обработчики методов.

Расширяется: GenericAPIView , RetrieveModelMixin , UpdateModelMixin

- RetrieveDestroyAPIView

Используется для чтения или удаления конечных точек для представления одного экземпляра модели.

Предоставляет get и delete обработчики методов.

Расширяется: GenericAPIView, RetrieveModelMixin , DestroyModelMixin

- RetrieveUpdateDestroyAPIView

Используется для чтения-записи-удаления для представления одного экземпляра модели .

Обеспечивает get, put, patch и delete обработчики методов.

Расширяется: GenericAPIView , RetrieveModelMixin , UpdateModelMixin, DestroyModelMixin

Переопределение методов

Для каждого класса можно переопределить его методы.

Пример:

```
class WarriorCreateAPIView(generics.CreateAPIView):
    serializer_class = WarriorCreateSerializer
    queryset = Warrior.objects.all()
    # permission_classes = [permissions.AllowAny]


    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

В примере сверху переопределен метод создания одного объекта. При сохранении сериализованных данных в базу данных, для нового объекта в поле owner проставляется ссылка на текущего пользователя.

Практическое задание 3.2.5–3.2.6:

Реализовать ендпоинты:

- Вывод полной информации обо всех воинах и их профессиях (в одном запросе).
- Вывод полной информации обо всех воинах и их скилах (в одном запросе).
- Вывод полной информации обо воине (по id), его профессиях и скилах.
- Удаление воина по id.
- Редактирование информации о воине.

Модель БД приведена на рисунке 12.

Практикум 3.3 Документирование

3.3.1 Типы документации к коду

Документирование – процесс создания какой-либо справочной информации по установленной форме и правилам.

Документирование подразделяется на несколько видов. Самыми популярными видами документирования являются:

1. Документация кода или аннотация кода. Основное внимание уделяется краткому описанию функции непосредственно в коде. Кроме самой сути функции, могут указываться входящие и исходящие параметры.
2. Создается несколькими способами: при помощи специальных линтеров (jsdoc, phpdoc), вручную и при помощи специального плагина в текстовом редакторе, например, ‘heavenshell/vim-jsdoc’ для vim;
3. markdown или md файлы. Чаще всего они встречаются в виде README.md файлах. Конкретных правил по созданию и описанию таких файлов не существует. Можно лишь следовать основой сути – это краткое описание проекта для введения в курс о его предметной области с краткой

инструкцией, как его запустить и, в отдельных случаях, как пользоваться.

4. Создание отдельного форматированного документа. Как правило подобными задачами занимается отдельный отдел и этим видом документации чаще всего занимаются фирмы специализирующиеся на аутсорсе. Данный вид не будет описан дальше, а представлен для ознакомления.
5. Использование специального сервиса Swagger. Он необходим для создания автоматической документации на основе уже существующего кода.

3.3.2 Аннотирование кода

Форма аннотирования кода, как правило, принимается совместно с другими разработчиками. Например, выносится на обсуждение, насколько детализированно должна быть описана та или иная функция. Кроме того, можно указывать, каким образом функция должна быть вызвана. Аннотация делается для обеспечения быстрого использования *незнакомой* функции без потери времени на ее чтения и на погружение в контекст исполнения функции:

```
# Class create and send message
# === USAGE ===================================
#
# log_sender = LogSender.new(data, account)
# log_sender.genetate_log!
# log_sender.send_log
#
# =============================================

class LogSender
  def initialize(data, user)
    @data = data
    @user = user
    @message = ''
  end

  def generate_log!
    @message = @data.map { |row| "[#{Time.now}] User: #{row.user} action
#{row.action}"}.join("\n")
  end

  def send_log
    Mailer.deliver(to: @user.email, message: @message)
  end
end
```

Аннотирование класса на Ruby с описанием выполняемой задачи и цепи

вызовов методов.

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 *
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */

function ListNode(val) {
  this.val = val
  this.next = null
}
```

Стоить напомнить, что подобное описание можно выполнить вручную или же при помощи специальных плагинов в текстовом редакторе.

3.3.3 Документирование кода

Markdown

Правилом хорошего тона является создание текстового файла в корне проекта с названием README. Файла содержит в себе информацию о программном обеспечении с применением языка облегченной разметки Markdown.

Информация о программном обеспечении может быть разнообразная, и не совсем ясно, какую суть должен отображать подобный текстовый файл. Как и в случае аннотирования кода, правила оформления README могут отображать разную информацию, например, инструкцию по развертыванию проекта на локальной машине и описание к основным командам проекта (создание сборки под конкретную операционную систему, если проект располагает таким функционалом, как запуск тестов, запуск установки зависимостей для запуска программы).

Пример README файла представлен ниже:

```
## Set up the environment
Install node version manager and required version of node js
```
$ chmod u+x ./bin/nvm.sh && ./bin/nvm.sh
$ nvm install
```
> For windows use [nvm for windows](https://github.com/coreybutler/nvm-windows)
```

```
Install Yarn package manager
```
$ brew install yarn --without-node
```
> Use `--without-node` flag  because nvm version of Node is used
Or download it directly from the
[website](https://yarnpkg.com/en/docs/install)
## Install npm packages
```
$ yarn install --frozen-lockfile
```
## Available scripts
### Start the server
```
$ yarn start
```
### Run ESLint with `--fix` option
```
$ yarn lint
```
### Run tests
```
$ yarn test
```
### Authentication
To manage pages you need to authorize (available on `/auth`).
To generate password use `yarn generatePassword [password]` command.
```

Отображение README файла в репозитории проекта можно посмотреть по ссылке [19].

Так как Markdown является языком стилизации текста, у него есть правила оформления определенных участков кода.

Markdown можно использовать в контексте какого-либо крупного модуля, а не только в масштабе всего проекта.

Описание синтаксиса Markdown доступно в [20], [21].

Средства создания документации к коду

Средства создания документации к коду следует разделить на 2 группы:

1. Средства документирования ендпоинтов.

   Примеры: Swagger [22], Redoc [23].

2. Средства составления полной документации.

   Примеры: MkDoks [24], readthedocs [25].

Swagger

Swagger – библиотека для генерации документации API. Она содержит описание url-адресов API с примерами описания передаваемых данных. Добавляется в проект как модуль, который необходимо сконфигурировать.

Демоверсию можно посмотреть в материале [26].

Установка Swagger

Далее рассмотрена инструкция по внедрению Swagger средствами drf_yasg в Django. Есть и другие способы внедрить автодокументирование Swagger в DRF [27].

Необходимо установить drf_yasg в виртуальное окружение:

```
pip install drf_yasg
```

В файле settings.py необходимо добавить 'drf_yasg' в INSTALLED_APPS:

```
# settings.py
# ...
INSTALLED_APPS = [
    ...,
    'drf_yasg'
]
# ...
```

В файле urls.py необходимо:

1. Инициализировать зависимости:

```
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
Создаем view
schema_view = get_schema_view(
   openapi.Info(
      title="API",
      default_version='v2',
      description="Description",
      terms_of_service="https://www.google.com/policies/terms/",
      contact=openapi.Contact(email="hardbeat34@gmail.com"),
      license=openapi.License(name="BSD License"),
   ),
   public=True,
   permission_classes=(permissions.AllowAny,),
)
```

2. Настроить роутинг:

```
urlpatterns = [
    ...,
    path('doc/swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('doc/redoc', schema_view.with_ui('redoc', cache_timeout=0),
name='schema-redoc')
]
```

Теперь Swagger доступен по url-адресу `doc/swagger/`.

Кроме того, drf_yasg дает возможность использовать redoc. Он доступен по url-адресу `doc/redoc`.

С примером инструкции в виде md-файла по внедрению Swagger средствами drf_yasg в Django проект можно ознакомиться в репозитории [28].

MkDocs

MkDocs – инструмент для создания статической проектной документации. Базируется на одном единственном файле YAML, где описаны все необходимые настройки. Суть в том, что инструмент позволяет сгенерировать сайт с документацией на основе структуры из *.md файлов [24].

Далее представлен краткий мануал по созданию документации средствами MkDoks для проекта “Warriors”.

Установка MkDocs

Чтобы использовать MkDocs, понадобится pip. Во время установки MkDocs необходимо выбрать тему [29]. В данном примере используется тема Material.

Код установки:

```
pip install --upgrade pip
pip install MkDocs
pip install MkDocs-material
```

Настройка MkDocs

Далее необходимо создать проект документации:

```
MkDocs new warriors # warrior - название документации
cd warrior
```

В папке warriors появится папка docs и файл MkDocs.yml.

Чтобы запустить режим отладки документации, необходимо использовать `MkDocs serve`. Далее перейти на страницу http://127.0.0.1:8000/ (рисунок 47).

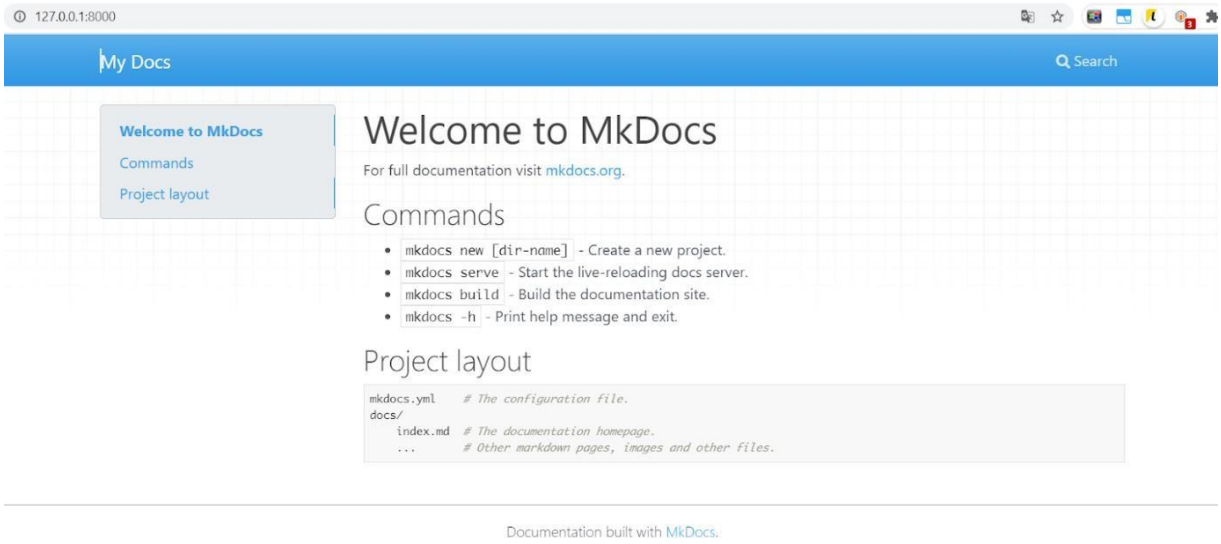


Рисунок 47 – Реализация MkDocs

Для настройки структуры документации требуется настроить файл MkDocs.yml. Пример описания структуры представлен ниже:

```
site_name: NAME
nav:
 - Home: index.md
 - Page2: page2.md
 - Section1:
   - Subpage1: subpage1.md
   - Subpage2: subpage2.md
theme:
 name: THEME_DOWNLOADED
```

Далее представлен пример описания MkDocs.yaml, описывающий малую часть описания сервиса “Воины” (см. практикум 3.1):

```
site_name: Описание сервиса "Войны"
nav:
 - Home: index.md
 - Professions:
     - "mining engineering": professions/mining_engineering.md
     - "picking herbs": professions/picking_herbs.md
 - Api:
     - "api/warriors[GET]": api/warriors.md

theme:
 name: material
```

Следует обратить внимание, что все папки и каталоги, упомянутые в MkDocs.yaml, должны быть созданы разработчиком вручную в docs-каталоге. Структура представлена на рисунке 48.

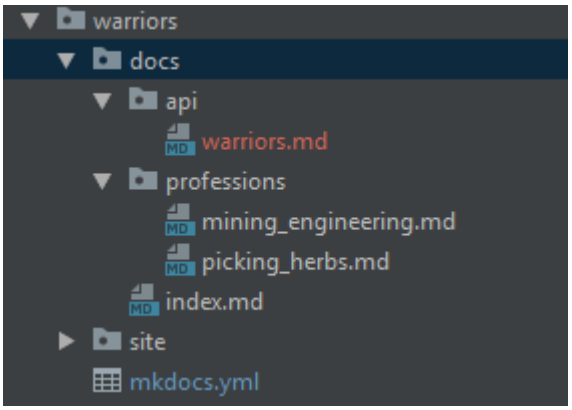


Рисунок 48 – Ручное описание ендпоинтов

Ручное описание едпоинтов часто требуется для описание сложных ендпоинтов с возможностью передачи в них различных параметров, от которых будет кардинально меняться результат и его структура.

Код примера, описывающего получение воинов, представлен ниже:

```
# Показать всех воинов

Выводит информацию обо всех войнах

**URL** : `/api/warriors/`

**Method** : `GET`

**Auth required** : YES

**Permissions required** : None

**Data constraints** : `{}`

## Success Responses

**Code** : `200 OK`

**Content** : `{[]}`

```json
{
   "Warriors": [
```
```

```
        {
            "id": 1,
            "race": "t",
            "name": "Petr2",
            "level": 0,
            "profession": null,
            "skill": []
        },
        {
            "id": 2,
            "race": "t",
            "name": "Petr",
            "level": 0,
            "profession": null,
            "skill": []
        }
    ]
}
```

Кроме того, возможно установить в MkDocs плагины, которые позволяют автоматически генерировать документацию к API [30].

Развертывание на GitHub

Далее необходимо разместить документацию на страницах GitHub. Необходимо выполнить команду `MkDocs gh-deploy`. Будет создана новая ветка в репозитории проекта, с которым ведется работа.

Пример получившегося описания доступен в репозитории [31]

Теперь документация размещена в общем доступе.

Если есть необходимость в информации о том, как дополнительно настроить документацию или другие параметры MkDocs, то можно ознакомиться с документацией [24].

Практическое задание 3.3.3

1. Составить README файл для описания проекта.
2. Внедрить Swagger в проект.
3. Инициализировать MkDocs с тремя страницами, описывающими общее описание проекта в репозитории, трех ендпоинтов, существующих в проекте, регистрации, авторизации и изменения учетных данных пользователя.

Лабораторная часть

Практическое задание

Реализовать серверную часть приложения средствами django и djangorestframework в соответствии с заданием:

1. Выполнить Практикумы 3.1 и 3.2.
2. Выбрать вариант из Приложения 1 или согласовать с преподавателем авторский вариант.
3. Реализовать модель базы данных средствами DjangoORM (необходимо предварительно согласовать модель базы данных с преподавателем).
4. Реализовать логику работу API средствами Django REST Framework, используя методы сериализации.
5. Подключить регистрацию / авторизацию по токенам / вывод информации о текущем пользователе средствами Djoser.
6. Выполнить Практикум 3.3 по оформлению документации.
7. Реализовать документацию, описывающую работу всех используемых endpoint-ов из пункта 3 и 4 средствами Read the Docs или MkDocs.

Порядок выполнения и защиты работы

1. Работа выполняется индивидуально.
2. По результатам работы необходимо подготовить документацию средствами MkDocs (Описание работы с MkDocs доступно практикуме в 3.3).
3. Полученную программу разместить в основном репозитории группы согласно инструкции по загрузке работы на github в Лабораторной работе 1. Ссылку на документацию оставить в комментариях.

ЛАБОРАТОРНАЯ РАБОТА 4. РЕАЛИЗАЦИЯ КЛИЕНТСКОЙ ЧАСТИ СРЕДСТВАМИ VUE.JS

Цель: Получить навыки реализации клиентский web-приложений средствами JavaScript и Vue.js

Оборудование: компьютерный класс.

Программное обеспечение: Node.js, Vue.js **3+**.

Практикум 4.1 Введение во Vue.js

4.1.1 Подготовка к работе

Вся работа с Vue и зависимостями будет происходить через NPM (Node Package Manager). Для начала необходимо установить node.js и npm. С инструкциями для установки можно ознакомиться на официальном сайте [32].

После успешной установки node и npm (ориентироваться следует на последние LTS-версии, который на момент написания соответствуют 16.18.0 и 8.19.2, соответственно) можно переходить к установке vue.

Для установки vue требуется выполнить следующую команду в терминале (важное уточнение: в unix-системах нужно вызывать эту команду с правами суперпользователя):

```
$ npm install -g @vue/cli
```

После выполнения этой команды на компьютере будет установлен vue и vue-cli (инструмент, необходимый для того, чтобы облегчить работу с инициализацией проекта на vue).

Теперь, когда vue и vue-cli глобально установлены, можно приступать к инициализации проекта:

```
$ npm init vue@latest
```

Далее необходимо следовать шагам, которые предлагает CLI-утилита:

```
Vue.js - The Progressive JavaScript Framework

? Project name: › vue-project
```

Дальше CLI-утилита спрашивает про различные опции, которые помогают определить базовую архитектуру проекта:

```
Vue.js - The Progressive JavaScript Framework
```

```
✓ Project name: … vue-project
✓ Add TypeScript? … No / Yes
✓ Add JSX Support? … No / Yes
✓ Add Vue Router for Single Page Application development? … No / Yes
✓ Add Pinia for state management? … No / Yes
✓ Add Vitest for Unit Testing? … No / Yes
✓ Add an End-to-End Testing Solution? › No
✓ Add ESLint for code quality? … No / Yes
✓ Add Prettier for code formatting? … No / Yes

Scaffolding project in /Users/kantegory/ITMO/edu/vue-project...

Done. Now run:

  cd vue-project
  npm install
  npm run dev
```

Для запуска проекта необходимо перейти в директорию проекта:

```
$ cd vue-project
```

Установить зависимости:

```
$ npm install
```

Далее необходимо запустить проект следующей командой:

```
$ npm run dev
```

В случае удачного запуска в консоли отобразится адрес, на котором запустился проект (рисунок 49).

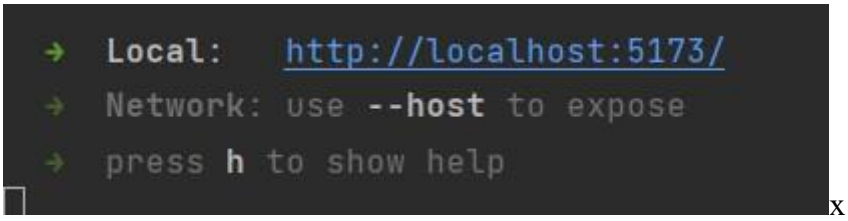


Рисунок 49 – Адрес запуска проекта

После перехода по указанному в консоли url-адресу отобразится стартовая страница vue.js (рисунок 50).

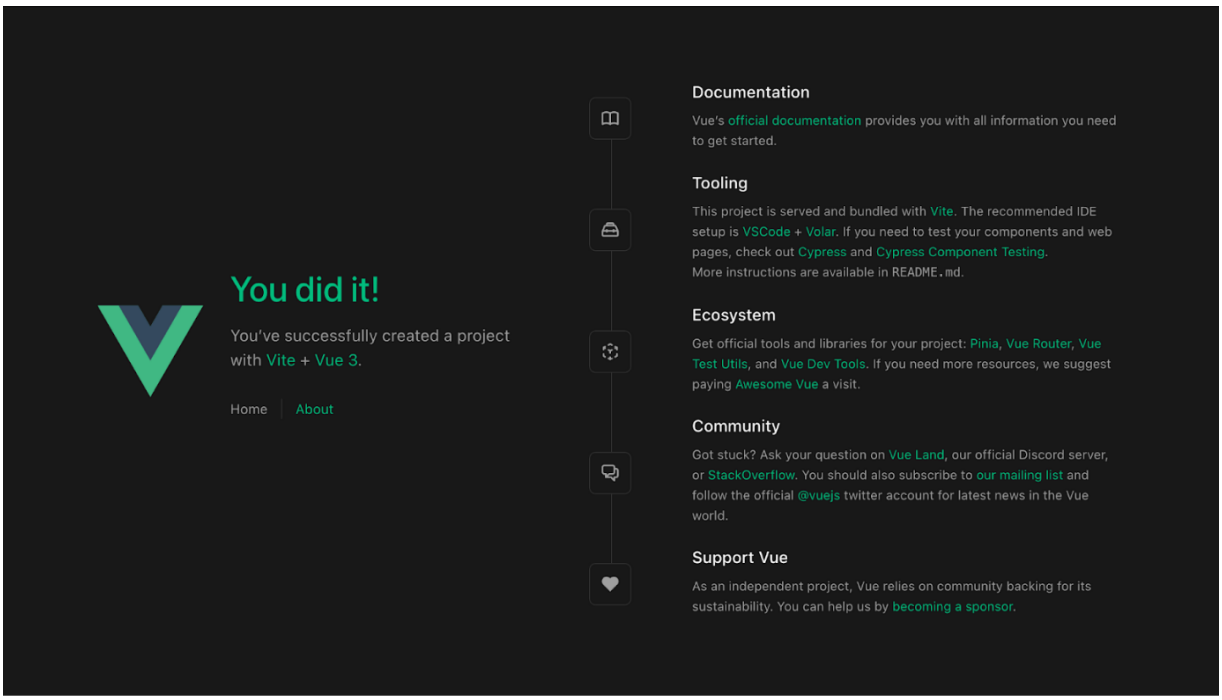


Рисунок 50 – Стартовая страница vue.js

4.1.2 Vue.js devtools

Для удобства работы рекомендуется установить Vue.js devtools [33].

Данное расширение упростит выполнение данной практики, т.к. позволяет просматривать состав компонентов и данные в них (рисунок 51).

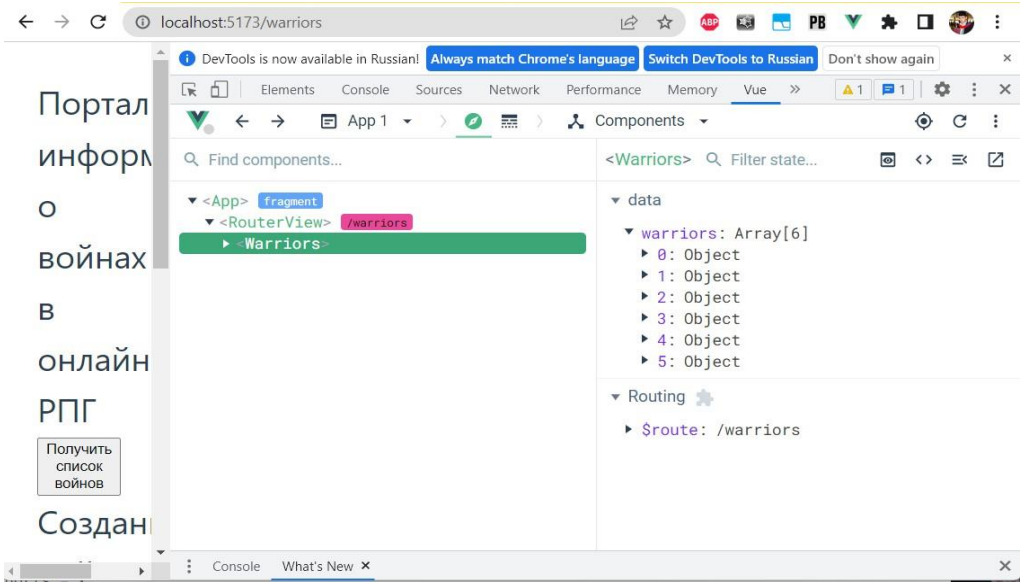


Рисунок 51 – Отображение компонента через расширение

4.1.3 Компоненты, роутинг

Создание первого компонента

Необходимо создать отдельный компонент src/components/Hello.vue, чтобы разобраться с тем, как работает компонентный подход:

```
<template>
 <div>
   <h1>Привет, это твой первый компонент</h1>
 </div>
</template>

<script>
export default {
 name: "Hello"
}
</script>

<style scoped>



</style>
```

Весь код компоненты во Vue логически можно разделить на три части: template, script, style. Template отвечает за вёрстку, html-код. Script – за все скрипты, которые используются для добавления какой-либо логики (запросы к API, вывод информации и так далее). Style отвечает за все стили компонентов.

Создание роутера

Далее рассмотрен простой пример работы с роутингом, достаточного для выполнения лабораторной работы. Почитать как работает роутинг можно в документации [34].

В файле конфигурации роутера (src/router/index.js) есть массив со всеми путями, которые используются в приложении:

```
import Hello from "@/components/Hello.vue";
import {createRouter, createWebHistory} from "vue-router";

const routes = [  // массив с роутами
   // отдельный роут:
   {
       path: '/hi', // конкретный url-адрес
       component: Hello // Ссылка на компонент
   },
]

const router = createRouter({
   history: createWebHistory(), routes
})

export default router // экспортируем сконфигурированный роутер
```

Файл src/main.js должен выглядеть так:

```
import { createApp } from 'vue'
import App from './App.vue'
import './assets/main.css'
import router from "./router";
createApp(App).use(router).mount('#app')
```

Роутер необходимо встроить в файл App.vue (файл src/App.vue) (остальное содержимое файла рекомендуется удалить):

```
<template>
 <div class="app">
   <router-view/>
 </div>
</template>
```

Запуск первого компонента

Для проверки работоспособности созданного компонента требуется перейти по ссылке http://localhost:5173/hi (порт может отличаться, его можно посмотреть при запуске сервера) (рисунок 52).

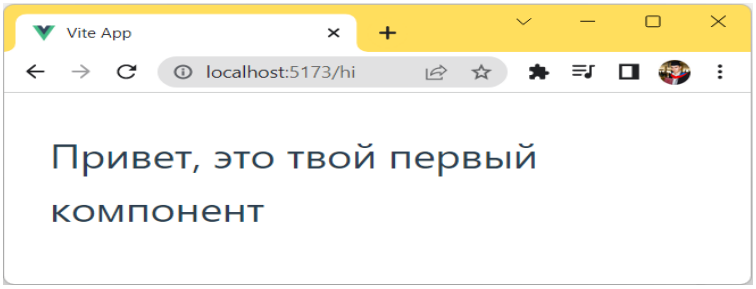


Рисунок 52 – Запуск компонента Vue.js

После создания первого компонента рекомендуется пройти 15 заданий, которые помогут понять, как осуществляется работа с данными внутри компонентов и передача данных между компонентами [35].

Разделение компонентов на view и components

Что такое представления Vue в Vue.js? Представление в Vue.js – это компонент, который ссылается на фактическую страницу, с которой работает пользователь. Vue.js – это технология, основанная на компонентах. Это означает, что все построено как компонент, а разница между представлением и компонентом заключается скорее в структуре папок в проекте.

Нет никакой разницы между представлением или компонентом в проекте Vue.js. Это скорее структура папок, которая означает, что компонент представления Vue является страницей, а компонент относится к чему-то, что можно использовать повторно и можно использовать в представлении.

Vue.js обычно использует маршрутизатор Vue, который является официальной библиотекой для навигации по страницам в приложениях Vue.js. Маршрутизатор Vue используется для создания одностраничного приложения (SPA). Маршрутизатор Vue помогает связать URL-адрес/историю с компонентами Vue.js. Это означает, что определенные пути отображают определенный компонент представления Vue.js, связанный с ним.

Что такое компонент в Vue.js? Компонент – это и представление Vue, и компонент по техническому использованию. Компонент определяет что-то, что можно использовать повторно и может быть сохранено в src/components. Например, компонент может быть заголовком, нижним колонтитулом, рекламой, таблицей, текстовыми полями или кнопками. Затем можно получить доступ к одному или нескольким компонентам внутри представления, например, к верхнему и нижнему колонтитулу.

Можно добавить любой компонент в router.js и получить к нему доступ по URL-адресу. Если необходимо получить доступ к компоненту через URL-адрес, обычной практикой является его определение как представления и перемещение в src/views, а не размещение в src/components.

Структура папок компонентов Vue.js

На рисунке 53 показана стандартная структура папок при создании проекта Vue.js.

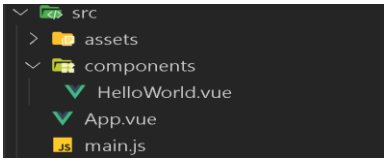


Рисунок 53 – Структура папок во Vue приложении

Компоненты находятся в папке src/components. Это может быть рабочая структура папок для обработки компонентов в небольших проектах. Данная структура хорошо работает в небольших проектах.

Когда проекты становятся больше, нужно сортировать структуру папок компонентов, чтобы упростить навигацию по проекту. Чтобы упростить взаимодействие с приложением, ниже представлено несколько советов, которые помогут структурировать компоненты в более крупных проектах.

На рисунке 54 представлен проект большего размера, на котором без разделения на компоненты и представления было бы сложно ориентироваться:

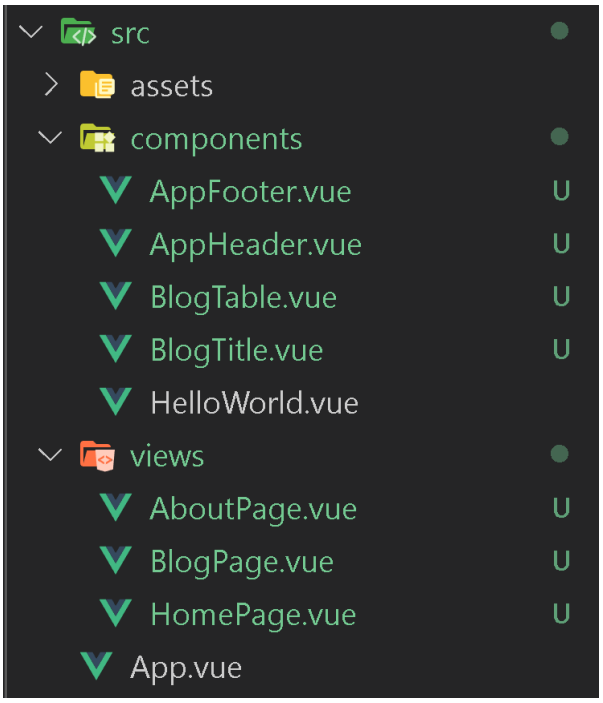


Рисунок 54 – Пример неудачной структуры приложения без разбиения по компонентам

Компоненты разбиты по разным папкам для представлений и компонентов, чтобы упростить работу с кодовой базой:

- Компонент представлений Vue относится к страницам, на которые может переходить пользователь. Представления Vue также могут использовать компоненты из папки компонентов, такие как AppHeader.vue и AppFooter.vue.
- Папка компонентов относится к компонентам, которые можно повторно использовать в проекте, а также можно использовать для создания компонентов представлений Vue.

Теперь, когда известно, что компонент представления может содержать другие компоненты, можно приступить к еще большему разделению проекта.

Создание компонентов и представления для получения и отображения данных

Создание представления

Весь код компоненты или представления во Vue логически можно разделить на три части: template, script, style. Template отвечает за вёрстку, html-код. Script – за все скрипты, которые используются для добавления какой-либо логики (запросы к API, вывод информации и так далее). Style – за все стили компоненты.

Ниже представлен код представления с комментариями (файл src/views/Warriors.vue):

```
<template>
 <div class="app">
   <h1>Портал информации о воинах в онлайн РПГ</h1>
   <!-- Кнопка вызывает функцию получения списка данных (функция
fetchWarriors объявлена в блоке "methods") -->
   <button class="btn btn-light m-10" v-on:click="fetchWarriors">Получить список
воинов</button>
   <div class="warriors-layout">
     <warrior-form />
     <!-- Встраивание компонента вызывающего список объектов. v-bind –
     директива служит для так называемой data binding -- привязки данных
     (данные объявляются в блоке данных data() (см. код ниже)). -->
     <warrior-list v-bind:warriors="warriors" />
   </div>
 </div>
</template>

<script>
import WarriorForm from "@/components/WarriorForm.vue";
import WarriorList from "@/components/WarriorList.vue";
import axios from "axios";

export default {
 components: {
   WarriorForm,
   WarriorList
 },
 // data - это функция, которая возвращает объект с данными
 data() { return
   {
     warriors: [], // Массив данных (передается в компонент WarriorList,
     получает данные средствами функции fetchWarriors
   }
 },
 // methods. Это объект, который содержит список Javascript функций, которые
 должны выполняться в зависимости от того, какие действия
 производит пользователь.
```

```
  methods: {
    // асинхронная функция для получения данных
    async fetchWarriors()
      { try {
        // Выполнение GET-запроса Backend-серверу. Запрос вернет JSON.  const
                                response                  =
                                await
 axios.get('http://localhost:8890/warriors/list/')

       // Массив данных warriors из блока(функции) data() получает значением
результат только-что выполненного запроса
       this.warriors = response.data.results
     } catch (e) {
       alert('Ошибка')
     }
   },
 },
 // Vue вызывает хук mount(), когда компонент добавляется в DOM.  В данном примере
это  позволяет  вызвать  fetchWarriors  для  получения  списка  воинов  до  отрисовки
страницы в браузере, благодаря этому страница загружается с уже полученными ранее
данными.
 mounted() {
   this.fetchWarriors();
 }
}
</script>
```

Создание компонента получения данных

Ниже представлен код компоненты получения данных с комментариями (файл src/components/WarriorList.vue):

```
<template>
 <div class="warrior" v-for="warrior in warriors"> <!-- v-for - директива для отображения
списка элементов на основе массива. -->
    <div><strong>Имя:</strong> {{ warrior.name }}</div>
    <div><strong>Раса:</strong> {{ warrior.race }}</div>
 </div>
</template>
<script>
export default {
 props: { // «Props» -- это специальное ключевое слово, обозначающее свойства . Его
можно зарегистрировать в компоненте для передачи данных от родительского компонента к
одному из его дочерних компонентов.
    warriors: { type:
      Array, required:
      true
    }
 }
}
</script>
<style scoped>
</style>
```

Создания формы в компоненте

Ниже представлен код компоненты получения данных с комментариями (src/components/WarriorForm.vue):

```
<template>
 <form @submit.prevent> <!-- @submit.prevent позволяет остановить
перезагрузку страницы после нажатия кнопки -->
   <h1>Создание воина</h1>
   <input
       v-model="warrior.name"
       class="input"
       type="text"
       placeholder="Имя"> <!-- Вызывает форму, при нажатии кнопки
"Создать", "warrior.name" получит значение из формы -->
   <input
       v-model="warrior.race"
       class="input"
       type="text"
       placeholder="Расса">
   <button class="btn" v-on:click="createWarrior">Создать воина</button>
<!-- Нажатие кнопки "Создать" вызывает метод "createWarrior" и записывает
данные в объект "warrior" -->
 </form>
</template>

<script>
import axios from "axios";

export default {
 name: "WarriorForm",
 data () {
   return {
     warrior: {
       name: '',
       race: ''
     }
   }
 },
 methods: {
   createWarrior() {
     axios.post('http://62.109.28.95:8890/warrior/create1', {
       race: this.warrior.race,
       name: this.warrior.name,
     })
  }
 }
```

```
</script>

<style scoped>

</style>
```

Запуск разработанного представления и компонент

Перед запуском проекта необходимо обновить роутер (src/router/index.js). Добавить ссылку на новое представление:

```
{
   path: '/warriors',
   component: Warriors
}
```

Для выполнения практических работ развернут общий Backend-сервер. Дополнительно бэкенд можно развернуть локально [36].

Запуск версии для разработки осуществляется командой npm run dev.

Листинги Frontend-кода приведены без стилей. Для получения более полного представления о демонстрационном проекте рекомендуется ознакомиться с содержимым репозитория [36]. CSS присутствует как в компонентах и представлениях, так и в src/assets, где размещены общие стили проекта.

4.1.4 Работа с локальным состоянием во Vue.js

В рассмотренном ранее примере данные из формы (src/components/WarriorForm.vue) попадали в свойство data компонента благодаря v-model.

Директива v-model используется для двунаправленного связывания данных с элементами форм input, textarea и select. Это означает что данные, вводимые пользователем в текстовое поле, автоматически окажутся в data, в данном случае – в warrior.name, что позволит обращаться к имени воина в методах компонента с помощью this.warrior.name (рисунок 55).

Рисунок 55 –Указание директивы

Однако данные текстовых полей - далеко не единственное изменяемое состояние, которое обычно требуется при разработке веб-приложения. Далее будет рассмотрен пример изменения состояния компонента вручную без v-model.

Ниже представлен обновленный код src/components/WarriorForm.vue (без стилей) с комментариями. В функцию data было добавлено свойство warrior.damage. Управление им происходит при помощи методов increaseDamage и decreaseDamage. После отправки данных на сервер с помощью axios.post введенные данные о воине сбрасываются:

```
<template>
 <form @submit.prevent class="warrior-form">
   <h1>Создание воина</h1>
   <input       v-model="warrior.name"       class="input"       type="text"
placeholder="Имя">
   <input       v-model="warrior.race"       class="input"       type="text"
placeholder="Раса">
   <div class="damage">
     <button class="btn" v-on:click="decreaseDamage">-</button>
     <span>Текущее значение урона: <b>{{ warrior.damage }}</b></span>
     <button class="btn" v-on:click="increaseDamage">+</button>
   </div>
   <button    class="btn    btn-light"    v-on:click="createWarrior">Создать
воина</button>
 </form>
</template>

<script>
import axios from "axios";

export default {
 name: "WarriorForm",
 data() {
   return {
     warrior: {
       name: '',
       race: '',
       // Величина наносимого урона
       damage: 0
     }
   }
 },
 methods: {
   async createWarrior() {
     await axios.post('http://localhost:8890/warrior/create1', {
       race: this.warrior.race,
       name: this.warrior.name,
```

```
        damage: this.warrior.damage,
      })

      // После создания нового воина введенные характеристики обнуляются
      this.warrior = {
        race: '',
        name: '',
        damage: 0
      };
    },

    // Метод, отвечающий за увеличение урона
    increaseDamage() {
      // С помощью this происходит обращение к свойству из data
      this.warrior.damage++;
    },

    // Метод, отвечающий за уменьшение урона
    decreaseDamage() {
      if (this.warrior.damage > 0) {
        this.warrior.damage--;
      }
    }
  },
}
</script>
```

4.1.5 Создание детальной страницы воина

Ниже представлен код детальной страницы воина (src/views/WarriorDetail.vue) с комментариями. URL детальной страницы будет иметь вид http://127.0.0.1:5173/warrior/:id.

В хуке mounted с помощью this.$route.params.id осуществляется доступ к id текущего воина из path-параметра URL. Далее в методе fetchWarrior происходит запрос данных с сервера в соответствии с этим id, и полученная информация о воине подставляется на страницу:

```
<template>
  <div>
    <h1>Warrior details</h1>
    <ul>
      <li>Имя – {{ warrior.name }}</li>
      <li>Раса – {{ warrior.race }}</li>
      <li>Урон – {{ warrior.damage }}</li>
    </ul>
  </div>
</template>
```

```
<script>
import axios from "axios";

export default {
 data() {
   return {
     warrior: {
       name: '',
       race: '',
       damage: 0,
       id: 0
     }
   }
 },
 methods: {
   async fetchWarrior(warriorId) {
     try {
       const response = await
axios.get(`http://localhost:8890/warrior/detail/${warriorId}`)
       // Получение данных с сервера и запись в состояние компонента
       this.warrior = response.data;
     } catch (e) {
       alert('Ошибка')
     }
   }

 },
 mounted() {
   // При загрузке страницы происходит получение id воина из path params
   const warriorId = this.$route.params.id;
   // Далее будет выполнено получение данных о воине с конкретным id
   this.fetchWarrior(warriorId);
 }
}
</script>
```

После создания нового представления необходимо добавить ссылку на него в роутер:

```
{
    path: '/warrior/:id',
    component: WarriorDetail
},
```

После этого требуется связать страницу с информацией обо всех воинах с детальными страницами. Для этого необходимо добавить ссылки в компонент списка воинов.

Ниже представлен обновленный код компонента

src/components/WarriorList.vue. Связь с детальными страницами осуществляется при помощи компонента <router-link>, в атрибуте to которого указывается ссылка в формате warrior/:id:

```
<template>
 <div>
   <h1>Список воинов</h1>
   <!-- Если массив воинов пуст, будет показано соответствующее сообщение -
->
   <div v-if="warriors.length === 0">
     Информации о воинах пока нет
   </div>
   <div class="warrior" v-for="warrior in warriors">
     <!-- Компонент router-link ведет на страницу детального представления
-->
     <router-link :to="`warrior/${warrior.id}`">
       <div>Имя: {{ warrior.name }}</div>
       <div>Раса: {{ warrior.race }}</div>
     </router-link>
   </div>
 </div>
</template>

<script>
export default {
 props: {
   warriors: {
     type: Array,
     required: true
   }
 }
}
</script>
```

После выполнения описанных действий интерфейс веб-приложения должен выглядеть, как показано на рисунке 56.

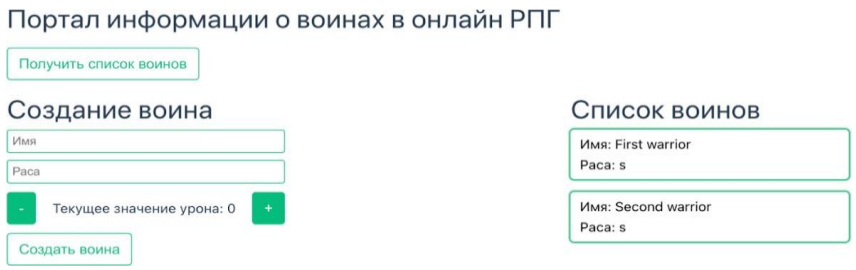


Рисунок 56 – Результат создания страницы со списком воинов и интерфейсом для их создания

При клике на любого воина из списка будет осуществлен переход на его

детальную страницу (рисунок 57).

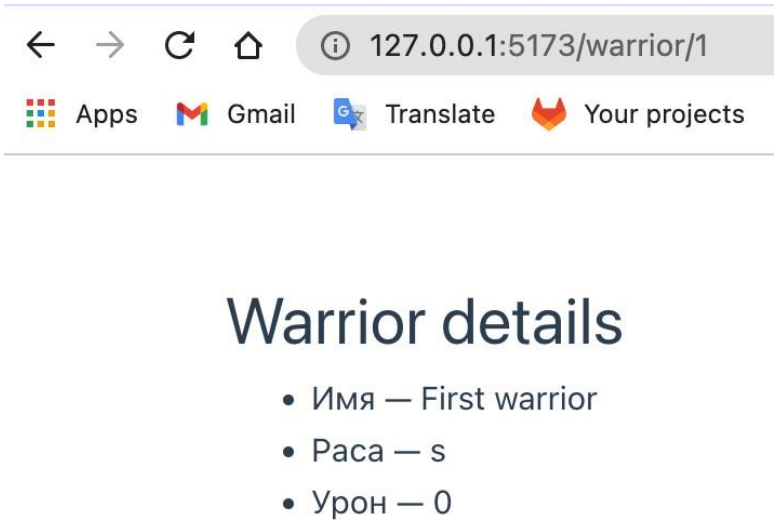


Рисунок 57 – Детальное отображение конкретного воина

Стейт-менеджмент во Vue

В ранее рассмотренных примерах использовалось локальное состояние компонента. Такое решение подходит для небольших проектов, однако совершенно не работоспособно для крупных сервисов, где необходим обмен данными между компонентами.

Для решения данной задачи во Vue (и аналогичных фреймворках) используется стейт-менеджер. Можно сказать, что стейт-менеджер – это набор правил по управлению глобальным хранилищем, а также специальное API позволяющее распространять данные по приложению.

Во Vue в качестве стейт-менеджера обычно используют Vuex или Pinia. Библиотека Pinia является довольно молодой, но в то же время довольно интуитивной в использовании и быстроразвивающейся, поэтому в следующем примере будет рассмотрена именно она.

Первым делом требуется выполнить установку pinia в проект. Это делается командой `npm install pinia`.

Далее необходимо сконфигурировать глобальное хранилище и подключить его в App.vue. Ниже представлено обновленное содержимое файла src/main.js:

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'
import App from './App.vue'
import router from "@/router/router";

import './assets/main.css'

// Создаем экземпляр pinia
const pinia = createPinia()

createApp(App)
 .use(router)
 .use(pinia) // Подключаем его в App
 .mount('#app')
```

Далее необходимо создать файл store.js в src, как показано на рисунке 58.

Рисунок 58 – Положение хранилища данных store.js в проекте

Ниже приведено содержимое файла src/store.js с комментариями. При помощи функции defineStore из pinia создается модуль глобального хранилища:

```
import axios from "axios";

export const useWarriorStore = defineStore('warriors', {
 // Состояние проекта
 state: () => {
   return {
     // Массив с воинами
     warriors: []
   }
 },
 // Геттеры — функции для получения данных
```

```
 getters: {
   allWarriors: (state) => {
     return state.warriors;
   },
 },
 // Экшены — функции для операций над данными
 actions: {
   // Могут быть как синхронными (например, добавление сущности в массив
воинов)
   addWarrior(payload) {
     this.warriors.push(payload);
   },
   // Так и асинхронными
   async fetchWarriors() {
     try {
       // Загрузка воинов с сервера
       const response = await
axios.get('http://localhost:8890/warriors/list/')
       // С последующим помещением их в состояние проекта
       this.warriors = response.data.results
     } catch (e) {
       console.log('Cannot fetch warriors')
     }
   },
 },
})
```

В модуле присутствует состояние – state, геттеры (getters) – функции для получения данных из состояния в компонентах и экшены (actions) – функции для изменения состояния. Actions могут быть как синхронными (например, добавление новой сущности в массив состояния), так и асинхронными (например, включающими в себя получения данных с сервера).

Использование pinia позволяет избавиться от необходимости получать данные на каждой странице. Ниже приведен обновленный код App.vue, позволяющий загрузить список информации о воинах для всех страниц приложения:

```
  <div class="app">
    <router-view />
  </div>
</template>


<script>
import Navigation from "@/components/Navigation.vue";
// Функция из pinia для подключения модуля глобального состояния
import { mapStores } from 'pinia'
// Созданный модуль глобального состояния
import { useWarriorStore } from "./store";


export default {
  components: {
    Navigation
  },
  computed: {
    // Подключение useWarriorStore в компонент
    ...mapStores(useWarriorStore)
  },
  async mounted() {
    // Вызван action получения всех воинов
    await this.warriorsStore.fetchWarriors();
  }
}
</script>
```

Данные pinia также видны во Vue Devtools (рисунок 59).

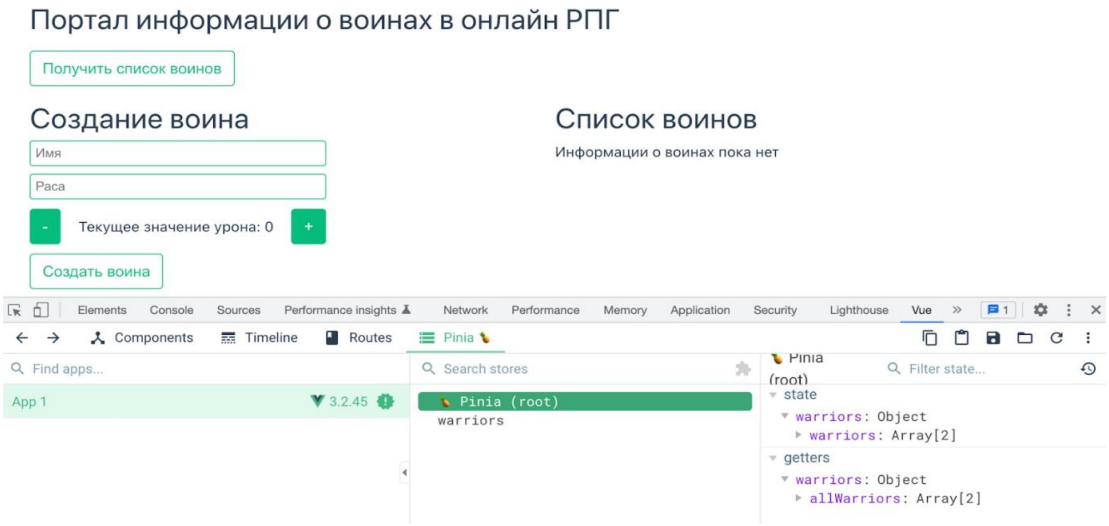


Рисунок 59 – Данные pinia в DevTools

Ниже представлены обновления файлов src/views/Warriors.vue и

src/components/WarriorForm.vue с учетом использования pinia.

Переход на глобальное состояние позволяет избавиться от получения данных на каждой странице. Более того, поддерживать данные в актуальном состоянии также становится проще. После вызова this.warriorsStore.addWarrior(result.data) в WarriorForm.vue информация о созданном воине распространяется по всему приложению.

Содержимое файла src/components/WarriorForm.vue с комментариями:

```
<template>
 <form @submit.prevent class="warrior-form">
   <h1>Создание воина</h1>
   <input       v-model="warrior.name"       class="input"       type="text"
placeholder="Имя">
   <input       v-model="warrior.race"       class="input"       type="text"
placeholder="Раса">
   <div class="damage">
     <button class="btn" v-on:click="decreaseDamage">-</button>
     <span>Текущее значение урона: <b>{{ warrior.damage }}</b></span>
     <button class="btn" v-on:click="increaseDamage">+</button>
   </div>
   <button    class="btn    btn-light"    v-on:click="createWarrior">Создать
воина</button>
 </form>
</template>
```

```
<script>
import axios from "axios";
// Функция из pinia для подключения модуля глобального состояния
import { mapStores } from 'pinia'
import { useWarriorStore } from "../store";

export default {
 name: "WarriorForm",
 data() {
   return {
     warrior: {
       name: '',
       race: '',
       damage: 0
     }
   }
 },
 computed: {
```

```
    // Подключение useWarriorStore в компонент
    ...mapStores(useWarriorStore)
  },
  methods: {
    async createWarrior() {
      const                  result                  =
                             await
 axios.post('http://localhost:8890/warrior/create1', {
        race: this.warrior.race,
        name: this.warrior.name,
        damage:
        this.warrior.damage,
      })

      // Добавление нового воина в
      warriorsStore
      this.warriorsStore.addWarrior(result.data
      ); this. warrior = {
        race: '',
        name:
        '',
        damag
        e: 0
      };
    },

    increaseDamage() {=
      this.warrior.damage++;
    },

    decreaseDamage() {
      if (this.warrior.damage >
        0) { this.warrior.damage-
        -;

     }
   }
 },
}
</script>
```

Содержимое файла src/views/Warriors.vue с комментариями:

```
<template>¶
 <div·class="app">¶
   <h1>Портал·информации·о·воинах·в·онлайн·РПГ</h1>¶
   <button·class="btn·btn-light·m-10"·v-on:click="fetchWarriors">Получить·
список·воинов</button>¶
   <div·class="warriors-layout">¶
     <warrior-form·/>¶
     <!--·Получение·данных·сразу·из·warriorsStore·-->¶
     <warrior-list·v-bind:warriors="warriorsStore.allWarriors"·/>¶
   </div>¶
 </div>¶
</template>¶
<script>
import WarriorForm from "@/components/WarriorForm.vue";
import WarriorList from "@/components/WarriorList.vue";
// Функция из pinia для подключения модуля глобального состояния
import { mapStores } from 'pinia'
import { useWarriorStore } from "../store";


export default {
 components: {
   WarriorForm,
   WarriorList
 },
 computed: {
   // Подключение useWarriorStore в компонент
   ...mapStores(useWarriorStore)
 },
}
</script>
```

4.1.6 Рекомендации по самостоятельному обучению

Для более продуктивной работы с vue.js рекомендуется изучить:

1. Библиотеки стилей:
   a) Библиотека стилей Vuetify.js предоставляет готовые компоненты для быстрой и удобной разработки пользовательского интерфейса вприложении. Она включает в себя множество готовых элементов, таких как кнопки, таблицы, формы, меню, диалоговые окна, пагинация и многое другое. Vuetify.js также поддерживает адаптивную верстку и позволяет легко настроить цвета, шрифты и другие параметры дизайна [37].
   b) Bootstrap 5 предлагает большой набор стилей и компонентов, которые могут использоваться вместе с Vue 3. Он не зависит от какого-либо конкретного фреймворка, что делает его очень гибким.

c) Quasar – это фреймворк для разработки приложений на Vue 3 с поддержкой большого количества платформ, включая веб, настольные приложения, мобильные приложения и другие. Он предоставляет множество готовых компонентов и утилит, которые упрощают создание кросплатформенных приложений.

d) Tailwind – это набор инструментов для создания пользовательских интерфейсов. Он предоставляет множество классов CSS для быстрого создания стилей и макетов. Он не является фреймворком в традиционном понимании, но может использоваться вместе с Vue 3 и другими фреймворками.

2. Статья о том, как использовать кнопку в качестве ссылки в Vue, помогает реализовать навигацию между страницами в приложении. Эта статья объясняет, как создать кнопку, которая при нажатии перенаправляет пользователя на другую страницу. Также рассматривается использование плагина Vue Router для реализации более сложной навигации в приложении [38].
3. Пример создания селектора для формы из JSON позволяет упростить создание форм в приложении. JSON-структура используется для хранения и передачи данных в формате, который легко парсится и обрабатывается. Этот пример демонстрирует, как можно использовать JSON для создания выпадающего списка в форме [39].
4. Пример создания селектора для формы с выбором типа номера может помочь решить задачу выбора из множества опций в приложении. В этом примере показывается, как создать селектор, который содержит несколько вариантов выбора и позволяет выбрать нужный тип номера [39].
5. Изучение стейт-менеджеров полезно для работы с большими и сложными приложениями, когда необходимо эффективно управлять состоянием приложения.

    a) State Management Talks – это набор докладов и статей, посвященных этой теме, который может помочь углубить знания в этой области [40].

    b) Pinia – это библиотека для управления состоянием приложения, созданная специально для Vue.js. Она предоставляет простой и интуитивный интерфейс для работы с состоянием, а также поддерживает множество функций, таких как подписка на изменения состояния и управление асинхронными операциями [41].

Практикум 4.2. Настройка CORS (Cross-origin resource sharing)

4.2.1 Введение

Многие начинающие фронтенд-разработчики, сталкиваются с ошибкой, показанной на рисунке ниже (рисунок 60):

```
Access to XMLHttpRequest at 'XXXX' from origin 'YYYY' has been blocked by
CORS policy: No 'Access-Control-Allow-Origin' header is present on the
requested resource..
```

Рисунок 60 – Ошибка CORS-policy

Далее рассказано, что это за ошибка, почему она возникает и как настроить Django REST framework, чтобы этой ошибки не было.

4.2.2 Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) – механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, который использует сайт в данный момент [42]. Говорят, что агент пользователя делает запрос с другого источника (cross-origin HTTP request), если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.

Пример cross-origin запроса: HTML страница, обслуживаемая сервером с http://domain-a.com, запрашивает <img> src по адресу http://domain-b.com/image.jpg. Сегодня многие страницы загружают ресурсы вроде CSS-стилей, изображений и скриптов с разных доменов, соответствующих разным сетям доставки контента (Content delivery networks, CDNs).

В целях безопасности браузеры ограничивают cross-origin запросы, инициируемые скриптами. Например, XMLHttpRequest [43] и Fetch API [44]следуют политике одного источника same-origin policy [45]. Это значит, что web- приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS- заголовки (рисунок 61).

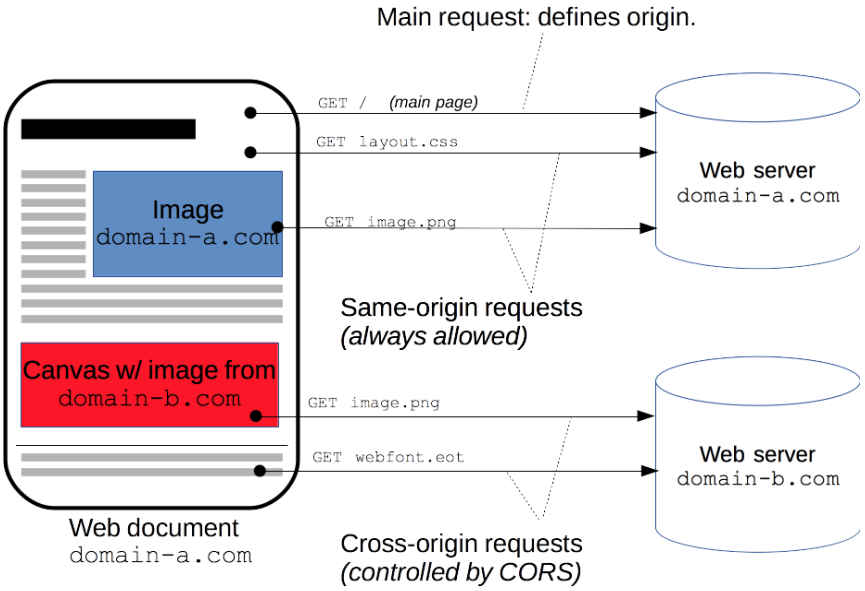


Рисунок 61 – Схема взаимодействия клиента и сервера при разных типах запроса

Механизм CORS поддерживает кросс-доменные запросы и передачу данных между браузером и web-серверами по защищенному соединению. Современные браузеры используют CORS в API-контейнерах, таких как XMLHttpRequest или Fetch, чтобы снизить риски, присущие запросам с других источников.

Обзор функциональности

Стандарт Cross-Origin Resource Sharing работает с помощью добавления новых HTTP-заголовков, которые позволяют серверам описывать набор источников, которым разрешено читать информацию, запрашиваемую web-браузером. В частности, для методов HTTP-запросов, которые могут привести к побочным эффектам над данными сервера (в частности, для HTTP-методов, отличных от GET, или для POST запросов, использующих определенные MIME-типы), спецификация требует, чтобы браузеры «предпроверяли» запрос, запрашивая поддерживающие методы с сервера с помощью метода HTTP-запроса OPTIONS и затем, поверх "подтверждения" с сервера, отсылали фактический запрос с фактическим методом HTTP-запроса. Сервера также могут оповещать клиентов должны ли "полномочия" (включая Cookies и HTTP Authentication данные) быть отправлены с запросом.

4.2.3 Обрабатываемые запросы

Стандарт CORS различает “простые” и “сложные” запросы. Простым считается запрос, работающий со следующими методами:

- HEAD
- GET

- POST
- и заголовками:
- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type, но только со значениями:
  - application/x-www-form-urlencoded
  - multipart/form-data
  - text/plain

Если запрос удовлетворяет этим критериям, можно отсылать Ajax-запрос к другому домену из любого современного браузера. При этом браузер добавит заголовок Origin с адресом страницы, откуда инициирован запрос. Подделать заголовок скриптом не удастся.

Сервер, получив на обработку подобный запрос, должен прочесть Origin и решить, как его обрабатывать. Заголовок ответа Access-Control-Allow-Origin регулирует, с какого домена разрешено запрашивать данные. Это может быть как веб-адрес, так и знак астерикса (звездочки), если разрешено всем.

Пример CORS-запроса:

```
POST /foo/bar HTTP/1.1
Origin: http://foreign.com
Host: test.com
```

и ответа с разрешением на получение данных:

```
200 OK HTTP/1.1
Access-Control-Allow-Origin: http://foreign.com
Content-Type: text/html; charset=utf-8

<h1>Welldone</h1>
```

Необходимо обратить внимание, что рассматривается ситуация, в которой ведется взаимодействие с чужими API. С вероятностью почти 100% они работают по протоколу JSON, то есть принимают и отдают заголовок Content-Type: application/json. Такой запрос автоматически перестает быть простым и переходит в разряд «сложных», где схема взаимодействия иная.

Сложные запросы проходят в два этапа. Сначала браузер делает запрос по тому же URL, но методом OPTIONS. Сервер должен ответить: какими другими методами и дополнительными заголовками (помимо стандартных) можно

обращаться к этому урлу. И только получив разрешение, браузер сделает запрос на основной URL.

При этом браузер все запомнит: если разрешили только методы GET и POST, то PUT и DELETE не сработают. Аналогично с заголовками: если помимо стандартных разрешено использовать только Authorization, то нужно передать его и ничего другого.

Пример сложного запроса:

```
OPTIONS /cors HTTP/1.1
Origin: http://api.bob.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.alice.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

Клиент хотел отправить Ajax-запрос методом PUT на URL http://api.alice.com/cors с сайта http://api.bob.com. Поскольку это сложный запрос, браузер запросил разрешение: “хочу сделать PUT на этот URL с особым заголовком X-Custom-Header”. Сервер ответил:

```
200 OK HTTP/1.1
Access-Control-Allow-Origin: http://api.bob.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Content-Type: text/html; charset=utf-8
```

Таким образом, разрешено использовать методы GET, POST, PUT с заголовком X-Custom-Header. Это подходит под критерии первоначального запроса. Браузер делает второй запрос по реальному URL-адресу.

Первая стадия, когда делается запрос OPTION, официально называется preflight request. Необходимо отметить, что такое взаимодействие весьма прозрачно отражается в браузере. Например, в консоли разработчика в Хроме видны оба запроса со всеми заголовками.

4.2.4 Настройка CORS в Django REST framework

Необходимо установить django-cors-headers с помощью pip:

```
pip install django-cors-headers
```

Необходимо добавить его в свой settings.руфайл проекта:

```
INSTALLED_APPS = (
    ##...
    'corsheaders'
)
```

Затем необходимо добавить corsheaders.middleware.CorsMiddleware в “MIDDLEWARE_CLASSES” в settings.py (необходимо добавить перед всеми остальными объектами):

```
MIDDLEWARE_CLASSES = (
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.BrokenLinkEmailsMiddleware',
    'django.middleware.common.CommonMiddleware',
    #...
)
```

Затем необходимо включить CORS для всех доменов, добавив следующий параметр:

```
CORS_ORIGIN_ALLOW_ALL = True
```

Или включить CORS только для указанных доменов:

```
CORS_ORIGIN_ALLOW_ALL = False

CORS_ORIGIN_WHITELIST = (
    'http//:localhost:8000',
)
```

Дополнительные параметры конфигурации доступны по ссылке [46].

Лабораторная часть

Практическое задание

Реализовать клиентскую часть приложения средствами vue.js.

Этапы работы:

1. Выполнить Практикум 4.1 Введение во Vue.js.
2. Настроить для серверной части, реализованной в лабораторной работе №3 CORS (Cross-origin resource sharing) в соответствии с Практикумом 4.2).
3. Утвердить с преподавателем список интерфейсов для взаимодействия с серверной частью в соответствии с индивидуальной предметной областью.
4. Реализовать интерфейсы авторизации, регистрации и изменения учётных данных и настроить взаимодействие с серверной частью. Серверная часть авторизации и регистрации была реализована в предыдущей работе средствами библиотеки Djoser.

5. Реализовать клиентские интерфейсы и настроить взаимодействие с серверной частью (Практикум 4.1).
6. Подключить Bootstrap у или аналогичную библиотеку стилей для Vue.js [47].
7. Реализовать документацию, описывающую работу разработанных интерфейсов средствами MkDocs.

В отчете должны быть представлены:

- Описание роутов в приложении.
- Описание компонентов и представлений.
- Скриншоты интерфейсов, разработанных в приложении.
- Дополнительные материалы на усмотрение студента.

Порядок выполнения и защиты работы

1. Работа выполняется индивидуально.
2. По результатам работы необходимо подготовить документацию средствами MkDocs (Описание работы с MkDocs доступно в практической работе 3.3).
3. Полученную программу разместить в основном репозитории группы согласно инструкции по загрузке работы на github в Лабораторной работе 1. Ссылку на документацию оставить в комментариях.

ПРИЛОЖЕНИЕ 1. ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

Задание 1

Вариант 1

Создать веб-приложение, предназначенное для администратора отеля.

Система должна обеспечивать хранение данных об имеющихся в гостинице номерах, о проживающих в отеле клиентах, горничных, убирающихся в номерах.

В отеле имеются номера следующих типов: одноместный эконом, одноместный стандарт, одноместный люкс, двухместный эконом, двухместный стандарт, двухместный люкс, трехместный эконом, трехместный стандарт, трехместный люкс, отличающиеся стоимостью проживания в сутки. В каждом номере установлен телефон.

О каждом госте должна храниться следующая информация: номер паспорта, где и когда выдан, фамилия, имя, отчество, город, из которого он прибыл, дата бронирования, даты заселения в отель и выселения из него, гостиничный номер для проживания.

Дежурному администратору отеля должна быть известна следующая информация об уборке номеров: фамилия, имя, отчество, где (номер) и когда (дата) он убирает.

Работа с системой предполагает получение следующей информации:

- о клиентах, проживавших в заданном номере в заданный период времени;
- о количестве неубранных номеров,
- о том, кто из горничных убирал номер указанного клиента в заданную дату,
- сколько в гостинице свободных номеров,
- список клиентов с указанием места жительства, которые проживали в те же дни, что и заданный клиент, в определенный период времени.

Администратор отеля должен иметь возможность выполнить следующие операции:

- принять на работу или уволить сотрудника отеля;
- изменить расписание работы сотрудника;
- поселить или выселить клиента;
- принять заказ на бронирование номера.

Вариант 2

Создать веб-приложение, предназначенное для сотрудников библиотеки.

Приложение должно обеспечивать хранение данных об имеющихся в библиотеке книгах, о читателях, читальных залах и абонементе.

Для каждой книги в БД должны храниться следующие сведения: название книги, автор(ы), издательство, год издания, раздел, число экземпляров книги в каждом зале библиотеки, шифр книги, дата выдачи книги читателю и дата возврата. Книги могут перерегистрироваться в разных залах.

Сведения о читателях должны включать номер читательского билета, ФИО читателя, номер паспорта, где и когда он выдан, дату рождения, домашний адрес, номер телефона, образование, наличие ученой степени.

Читатели регистрируются в определённом зале, могут переписаться в другой зал и выписываться из библиотеки.

Библиотека имеет несколько читальных залов, каждый из которых имеет номером, название, вместимость, то есть количеством читателей, которые могут одновременно работать в зале.

Необходимо хранить данные о поступлении новых книг и списании старых или негодных.

Шифр книги может меняться в результате переклассификации, номер читательского билета – в результате перерегистрации.

Библиотекарю может потребоваться следующая информация:

- Какие книги закреплены за заданным читателем и на какой срок?
- Кто из читателей взял книги более года тому назад?
- За кем из читателей закреплены книги, количество экземпляров которых в библиотеке не превышает 1?
- Сколько в библиотеке читателей младше 18 лет?
- Сколько читателей в процентном отношении имеют общее, общее среднее, высшее образование, ученую степень?

Библиотекарь имеет возможность выполнять следующие операции:

- Записать в библиотеку нового читателя.
- Исключить из списка читателей людей, записавшихся в библиотеку более года назад и не прошедших перерегистрацию.
- Списать старую, потерянную или негодную книгу.
- Принять на баланс книгу в фонд библиотеки.

Вариант 3

Создать веб-приложение, предназначенное для завуча школы.

Оно должно обеспечивать хранение сведений о каждом учителе, классном руководстве, о предметах, которые ведет преподаватель в заданный период, номере закрепленного за преподавателем кабинета, о расписании занятий. Есть учителя, которые не имеют собственного кабинета. Об учениках должны храниться следующие данные: фамилия, имя, отчество, в каком классе учится, какую оценку имеет в текущей четверти по каждому предмету, контакты родителей.

Завуч должен иметь возможность добавить сведения о новом учителе или ученике, внести в базу данных системы оценки за четверть учеников каждого класса по каждому предмету, удалить данные об уволившемся учителе и отчисленном из школы ученике, внести изменения в данные об учителях и учениках, в том числе изменить оценку ученика по заданному предмету. В функции завуча входит составление расписания.

Завучу может потребоваться следующая информация:

- Какой предмет будет в заданном классе в заданный день недели на заданном уроке?
- Сколько учителей преподает каждую из дисциплин в школе?
- Список учителей, преподающих те же предметы, что и учитель, ведущий информатику в заданном классе.
- Сколько мальчиков и девочек в каждом классе?
- Сколько кабинетов в школе для базовых и профильных дисциплин?

Вариант 4

Создать веб-приложение, предназначенное для организаторов ежегодных выставок собак.

Выставки могут быть моно- и полипородные. Приложение должно обеспечивать хранение данных о собаках-участниках выставок и экспертах. Участие может быть индивидуальным или от клуба. У выставки могут быть спонсоры, которые могут спонсировать разные выставки.

Для каждой собаки в БД должны храниться данные, о том, к какому клубу она относится, кличка, порода, возраст, классность, сведения о родословной (номер документа, клички родителей), дата последней прививки, фамилия, имя, отчество и паспортные данные хозяина.

Перед соревнованиями собаки должны пройти обязательный медосмотр.

Т.к. участие является платным, то хозяин обязан после регистрации до прохождения медосмотра должен оплатить счет и предоставить его организаторам. Собака допускается до соревнований, если она успешно прошла медосмотр.

Сведения об эксперте должны включать фамилию и имя, номер ринга, который он обслуживает, клуб, название клуба, в котором он состоит. Каждый ринг могут обслуживать несколько экспертов. Каждая порода собак выступает на своем ринге, но на одном и том же ринге в разное время могут выступать разные породы.

Каждая собака должна выполнить 3 упражнения, за каждое из которых она получает баллы от каждого эксперта. Итогом выставки является определение медалистов по каждой породе по итоговому рейтингу.

Организатор выставки должен иметь возможность добавить в базу нового участника или нового эксперта, снять эксперта с судейства, заменив его другим, отстранить собаку от участия в выставке.

Организатору выставки могут потребоваться следующие сведения;

- На каком ринге выступает заданный хозяин со своей собакой?
- Какими породами представлен заданный клуб?
- Сколько собак были отстранены от участия в выставке?
- Какие эксперты обслуживают породу?
- Количество участников по каждой породе?

Вариант 5

Создать веб-приложение, предназначенное для информационного обслуживания редакторов, менеджеров и других категорий сотрудников типографии.

Приложение должно обеспечить хранение данных о сотрудниках, издаваемых книгах, авторах, финансовом состоянии компании и предоставлять возможность получать разнообразные отчёты.

В соответствии с предметной областью система строится с учётом следующих особенностей:

- каждая книга издаётся в рамках контракта;
- книга может быть написана несколькими авторами;
- контракт подписывается одним менеджером и всеми авторами книги;
- каждый автор может написать несколько книг (по разным контрактам);
- порядок, в котором авторы указаны на обложке, влияет на размер гонорара;
- если сотрудник является редактором, то он может работать одновременно над несколькими книгами;

- у каждой книги может быть несколько редакторов, один из них – ответственный редактор;
- каждый заказ оформляется на одного заказчика;
- в заказе на покупку может быть перечислено несколько книг.

Сотрудникам типографии могут понадобиться следующие сведения:

- список всех изданных книг заданного автора с указанием количество изданных экземпляров;
- список ответственных редакторов для всех изданий;
- количество редакторов каждой книги, изданной в прошедшем году;
- количество контрактов за каждый месяц за истекший год;
- список всех менеджеров, которые имеют максимальное количество заказов на покупку книг за заданный период.

Вариант 6

Создать веб-приложение, предназначенное для отслеживания распределения по почтовым отделениям газет, печатающихся в типографиях города.

Приложение должно обеспечивать хранение, просмотр и изменение сведений о газетах, почтовых отделениях, получающих газеты и о типографиях, выпускающих газеты. Сведения о газетах включают в себя: название газеты, индекс издания, фамилию, имя и отчество редактора, цену экземпляра газеты. Цены могут меняться. Возможно появление новых газет и изменение индекса существующего издания. Для типографий хранятся их названия и адреса.

В типографии разными тиражами печатаются газеты нескольких наименований. Типография может быть закрыта, тогда необходимо скорректировать работу других типографий с учетом потребностей почтовых отделений в газетах.

Почтовое отделение имеет номер и адрес. На каждое почтовое отделение поступают в определенных количествах газеты разных наименований, причем часть экземпляров одной и той же газеты может быть напечатана в одной типографии, а часть – в другой.

Пользователям системы может потребоваться следующая информация:

- Сколько и в каких типографиях печатались газеты заданного наименования за истекший месяц?
- Фамилия редактора газеты, которая печаталась в указанной типографии самым большим тиражом в заданный период?

- Какие газеты и в каком количестве типография отправляет в каждое почтовое отделение в заданную дату?
- Какие газеты и куда (номер почты) поступают в количестве меньшем, чем заданное?
- Куда поступает данная газета, печатающаяся по данному адресу.

Вариант 7

Создать веб-приложение, предназначенное для администрации птицефабрики. Приложение должно обеспечить работу с информацией о работниках фабрики и об имеющихся на ней курицах.

О каждой курице должна храниться следующая информация: вес, возраст, порода, количество ежемесячно получаемых от курицы яиц, а также информация о местонахождении курицы.

Сведения о породе куриц включают: название породы, среднее количество яиц в месяц (производительность) и средний вес, номер рекомендованной и содержание диеты. Диеты носят сезонный характер.

Птицефабрика имеет несколько цехов. В каждой клетке может находиться несколько куриц. Код клетки, где находится курица, характеризуется номером цеха, номером ряда в цехе и номером клетки в ряду. Курицы могут пересаживаться из клетки в клетку. Необходимо знать количество ежедневно получаемых яиц по цеху.

Директор птицефабрики может принять или уволить работника. О сотрудниках птицефабрики в БД должна храниться следующая информация: паспортные данные, зарплата, договор о трудоустройстве, данные об увольнении, закрепленные за работником клетки в каждую смену.

Не должно быть клеток, не обслуживаемых ни одним работником. Количество куриц в клетке может изменяться как в большую, так и в меньшую сторону, в отдельные моменты времени часть клеток может пустовать.

Директору могут потребоваться следующие сведения:

- Средняя производительность по каждой породе по цехам за истекший месяц.
- В каком цехе наибольшее количество куриц определенной породы на текущий момент?
- Среднее количество яиц, которое получает в день каждый работник от обслуживаемых им клеток?
- Сколько куриц каждой породы в каждом цехе на заданную дату?

- Какова для каждой породы разница между показателями породы и средними показателями по птицефабрике?

Вариант 8

Создать веб-приложение, предназначенное для отдела маркетинга рекламного агентства.

Одной из задач, решаемых отделом маркетинга рекламного агентства, является учет работы с клиентами. Для этого необходимо организовать оперативный учет поступивших и выполненных заказов клиентов (рекламодателей).

Рекламное агентство заключает договоры с заказчиками на исполнение определенного вида рекламных услуг. Для оформления заявки рекламодатель должен указать контактное лицо, телефон и электронный адрес для связи. Рекламодатель оформляет заказ на рекламу, пользуясь прайс–листом, в котором указаны цены по наименованию рекламных услуг, предоставляемых агентством. Здесь же оговариваются исполнители изготовления рекламы (сотрудники агентства), стоимость и объем (количество) работ. Для выполнения работ необходимо знать единицы измерения и материалы. Заказчик должен иметь контактные данные исполнителя для согласования макета рекламы.

Согласно заказу, выписывается счета для оплаты Заказчику.

После оплаты счета агентство обязуется предоставить рекламные продукты. Заказ считается выполненным, если все счета по заказу.

Перечень возможных типовых запросов:

- список выполненных работ, фиксирующих дату оплаты заявки, заказчиков, код услуги, фамилию исполнителя;
- список счетов, выставленных рекламодателям за любой промежуток времени, фиксирующий заказчика, услугу, состояние заказа (оплачено или нет);
- общая стоимость работ, выполненных всеми исполнителями, за последний квартал;
- список заявок, заключенных каждым отдельным заказчиком за любой промежуток времени;
- список сотрудников с указанием количества заявок, которые выполнял каждый сотрудник в заданный период.

Вариант 9

Создать веб-приложение, предназначенное для диспетчера автобусного парка частной транспортной фирмы.

Фирма обслуживает несколько коммерческих автобусных маршрутов. Приложение должно обеспечить хранение данных о водителях, маршрутах и характеристиках автобусов.

О каждом водителе известны паспортные данные, класс, стаж работы и оклад, причем оклад зависит от класса и стажа работы.

Маршрут автобуса характеризуется номером маршрута, названием начального и конечного пункта движения (адрес), временем начала и конца движения, интервалом движения и протяженностью в минутах (время движения от кольца до кольца). Характеристиками автобуса являются: номер государственной регистрации автобуса, производитель, год выпуска, модель, вместимость, количество сидячих и стоячих мест.

Каждый водитель закреплен за определенным автобусом и работает на определенном маршруте, но в случае поломки своего автобуса или болезни другого водителя может пересесть на другую машину.

В базе данных должен храниться график работы водителей.

Необходимо предусмотреть возможность корректировки БД в случаях поступления на работу нового водителя, списания старого автобуса, добавления нового маршрута или изменения старого и т.п.

Диспетчеру автопарка могут потребоваться следующие сведения:

- Список водителей, работающих на определенном маршруте с указанием графика их работы, в заданную дату?
- Сколько автобусов каждой модели имеется в автопарке на текущий момент?
- Какова общая протяженность маршрутов, обслуживаемых автопарком?
- Какие автобусы не вышли на линию в заданный день и по какой причине (неисправность, отсутствие водителя)?
- Сколько водителей каждого класса работает в автопарке?

Вариант 10

Создать веб-приложение, предназначенное для администратора частной медицинской клиники.

Прием пациентов ведут несколько врачей различных специализаций. На каждого пациента клиники заводится медицинская карта, в которой отражается вся информация по личным данным клиента и истории его заболеваний (диагнозы).

При очередном посещении специалиста в карте отражается дата и время приема, диагноз, текущее состояние больного, рекомендации по лечению. Так как прием ведется только на коммерческой основе, после очередного посещения клиент должен оплатить медицинские услуги. Каждый прием оплачивается отдельно. Расчет стоимости посещения определяется врачом согласно прейскуранту по клинике.

Для ведения внутренней отчетности необходима следующая информация о враче: фамилия, имя, отчество, специальность, образование, пол, дата рождения и дата начала и окончания работы в клинике, данные по трудовому договору. Для каждого врача составляется график работы с указанием рабочих и выходных дней.

Прием пациентов врачи могут вести в разных кабинетах. Каждый кабинет имеет определенный режим работы, ответственного и внутренний телефон.

Перечень типовых запросов:

- Вывести по алфавиту список всех пациентов заданного врача с датами и стоимостью приемов.
- Вывести телефоны всех пациентах, которые посещали отоларингологов и год рождения которых больше, чем заданный.
- Вывести список врачей, в графике которых среди рабочих дней имеется заданный.
- Количество приемов пациентов по датам.
- Вычислить суммарную стоимость лечения пациентов по дням и по врачам.

Вариант 11

Для биржи труда создается автоматизированная информационная система, которая должна хранить информацию о соискателях и вакансиях, а также формирование резюме соискателя, обеспечить вывод данных о работодателях, состоянии вакансии, проходящих курсах и т.д. В отчетах фигурируют данные о соискателях, закрытых вакансиях, проводимых курсах, работодателях.

Для каждого соискателя составляется резюме, осуществляется подбор вакансии по профессии и образованию, начисляется пособие. Размер пособия зависит от размера его последней заработной платы. При начислении пособия сохраняется информация о датах начала и окончания выплаты пособия.

Соискателям предлагается пройти курсы по повышению квалификации или переквалификации. После похождения курсов соискатели получают разряд по освоенной профессии.

Для работодателей хранятся их контактные данные (название организации, адрес, контактное лицо, телефон, электронный адрес). По вакансии хранятся дата

подачи вакансии, состояние вакансии. Работодатель при размещении вакансии должен указать в заявке перечень требуемых профессий, образование соискателя, требуемый стаж, разряд, заработную плату. Возможна дополнительная информация.

Перечень возможных типовых запросов:

1. Выбор профессий соискателей, не представленных в таблице Вакансии.
2. Получить все возможные варианты вакансий для соискателей.
2. Посчитать количество дней с момента предложения вакансии для незакрытых вакансий.
3. Подсчитать количество выплачиваемых пособий на текущий момент.
4. Подсчитать количество вакансий, в которых требуется высшее образование и заработная плата от 5000 до 60000.

Вариант 12

Создать автоматизированную информационную систему, предназначенную для учебной части колледжа.

Система должна обеспечить хранение сведений о каждом преподавателе, о дисциплинах, которые он преподает, номере закрепленного за ним кабинета, о расписании занятий. Существуют преподаватели, которые не имеют собственного кабинета.

О студентах должны храниться следующие сведения: фамилия, имя, отчество, в какой группе учится, какую оценку имеет в текущем семестре по каждой дисциплине.

Замдекана должен иметь возможность добавить сведения о новом преподавателе или студенте, внести в базу данных семестровые оценки студентов каждой группы по каждой дисциплине, удалить данные об уволившемся преподавателе и отчисленном из колледжа студенте, внести изменения в данные о преподавателях и студентах, в том числе поменять оценку студента по той или иной дисциплине.

В задачу диспетчера учебной части входит составление расписания занятий.

Зам. декана могут потребоваться следующие сведения:

1. Какой предмет будет в заданной группе в заданный день недели на заданном уроке?
2. Вычислить средний балл в каждой группе по каждой дисциплине за заданный семестр.
3. Найти дисциплины с самой низкой успеваемостью за заданный семестр.

4. Расписание на заданный день недели для указанной группы?

5. Сколько студентов обучается на каждом курсе в заданный семестр?

Вариант 13

Создать автоматизированную информационную систему, предназначенную для хранения информации о торгах на товарно-сырьевой бирже.

На торги могут быть представлены разные товары одной и той же фирмы и одни и те же товары разных фирм.

Каждый товар имеет свой уникальный код, произведен определенной фирмой в определенное время. Товар имеет гарантийный срок хранения, единицу измерения. Товар считается просроченным, если дата его отгрузки более поздняя, чем дата производства этого товара в сумме с гарантийным сроком хранения.

Товары поставляются партиями. Партия характеризуется: номером, количеством единиц в партии, ценой поставляемого товара, условиями поставки (предоплата или нет). Партии товаров выставляют брокеры. В одну партию товаров включаются разнообразные товары от разных производителей.

Считается, что партии товаров, выставленные на продажу, покупает сама биржа, и она же расплачивается с брокером и производителями товара. Если условием поставки указана предоплата, то биржа перечисляет деньги в день заключения договора, а если нет – то в день отгрузки.

Брокеры работают за фиксированный процент прибыли – 10% от суммы заключенных сделок. Ежемесячно брокеры перечисляют конторе, в которой они работают, фиксированную сумму денег, а все остальные заработанные ими деньги составляют их чистый доход (зарплату).

Перечень возможных типовых запросов:

1. Подсчитать, сколько единиц товара каждого вида выставлено на продажу от начала торгов до заданной даты.
2. Ннайти фирму-производителя товаров, которая за заданный период времени выручила максимальную сумму денег.
3. Найти товары, которые никогда не выставляли на продажу брокеры заданной конторы.
4. Найти все факты выставления на продажу товаров с просроченной годностью (номер партии, код товара, наименование товара, данные о брокере).
5. Найти зарплату всех брокеров заданной конторы.

Вариант 14

Создать автоматизированную информационную систему, предназначенную для администрации аэропорта некоторой компании-авиаперевозчика.

Рейсы обслуживаются бортами, принадлежащими разным авиаперевозчикам. О каждом самолете необходима следующая минимальная информация: номер самолета, тип, число мест, скорость полета, компания-авиаперевозчик. Один тип самолета может летать на разных маршрутах.

О каждом рейсе необходима следующая информация: номер рейса, расстояние до пункта назначения, пункт вылета, пункт назначения; дата и время вылета, дата и время прилета, транзитные посадки (если есть), пункты посадки, дата и время транзитных посадок и дат и время их вылета, количество проданных билетов. Каждый рейс обслуживается определенным экипажем, в состав которого входят командир корабля, второй пилот, стюардессы или стюарды. Необходимо предусмотреть наличие информации о допуске члена экипажа к рейсу.

Администрация компании-владельца аэропорта должна иметь возможность принять работника на работу или уволить. При этом необходима следующая информация: ФИО, возраст, образование, стаж работы, паспортные данные. Эта же информация необходима для сотрудников сторонних компаний.

Перечень возможных типовых запросов:

1. Выбрать марку самолета, которая чаще всего летает по маршруту.
2. Выбрать маршрут/маршруты, по которым летают рейсы, заполненные менее чем на XX %.
3. Определить наличие свободных мест на заданный рейс.
4. Определить количество самолетов, находящихся в ремонте.
5. Определить количество работников компания-авиаперевозчика.

Вариант 15

Создать автоматизированную информационную систему, предназначенную для администратора альпинистского клуба.

Альпинистский клуб организует восхождения в разных точках мира. Система должна обеспечить сохранение информации о хронике восхождений.

Для каждого восхождения формируется группа. В состав группы могут входить альпинисты из других клубов. Поэтому нужно иметь информацию о каждом клубе (название, страна, город, контактное лицо, e-mail, телефон). Необходимо иметь описание маршрута и продолжительность восхождения. Необходимо обеспечить сохранение даты/времени начала и завершения каждого

восхождения (планируемого и фактического), имен и адресов участвовавших в нем альпинистов, названия и высоты горы, страны и района, где эта гора расположена. После завершения восхождения фиксируется информация об успешности восхождения для каждого участника и группы в целом. При возникновении нештатных ситуаций необходимо указать для каждого участника, что случилось (травма, пропал без вести, летальный исход и т.д.) и в пояснении о группе дать подробности.

Администратор должен иметь возможность:

- добавления сведений о новом альпинисте, новой вершине;
- изменения сведений об альпинистах и вершинах;
- формирования новых групп и внесения всей информации после завершения восхождения группой.

Перечень возможных типовых запросов:

1. Показать список альпинистов, осуществлявших восхождение в указанный интервал дат.
2. Показать список восхождений (групп), которые осуществлялись в указанный пользователем период времени.
3. Предоставить информацию о том, сколько альпинистов побывали на каждой горе.
4. Предоставить данные о вершинах, если на них не было восхождений.
5. Показать информацию о количестве восхождений каждого альпиниста на каждую гору.

Вариант 16

Создать автоматизированную информационную систему, предназначенную для управления музейными фондами.

Музейные предметы хранятся в музейных фондах. Фонды могут располагаться по различным адресам.

Существуют различные фонды: живопись, графика, икона, скульптура, декоративно-прикладное искусство (ДПИ), нумизматика, археология, рукописи и редкая книга и т.п. Для удобства работы в ряде фондов предусмотрены вспомогательные картотеки комплектов – сервизов и гарнитуров в ДПИ, альбомов в графике, иконостасов в древнерусском искусстве и т.п. Необходимо реализовать ведение карточек музейных предметов – инвентарный номер, название, дата создания, точно определена дата создания или приблизительно, авторах работы (только первый автор – ФИО, дата рождения, страна), выставки, в которых

участвовал музейный предмет. Необходимо вести учет движения (прием на хранение, передача на выставку, возвращение с выставки, списание и т.п.) музейных предметов вне (знать информацию об организации, которой на время передается предмет – название, адрес, телефон, ФИО контактного лица, адрес, где проводится выставка, название выставки, дата начала работы, дата окончания работы) и внутри музея (из фонда в фонд), осуществлять оформление актов движения. Акты подписывает руководитель музея и хранитель фонда, отвечающий за предметы в музейном фонде. Предметы могут передаваться как в составе целого комплекта, так и по отдельности.

Перечень возможных типовых запросов:

1. Для каждого фонда указать количество выставок, в которых участвовали предметы из этого фонда.
2. Для каждого комплекта указать количество единиц в комплекте.
3. Для заданного предмета вывести список других предметов, которые участвовали в тех же выставках, что и заданный.
4. Вывести информацию по количеству списанных предметов по каждому музейному фонду в заданный период времени.
5. Найти процентное соотношение объема музейных фондов.

Вариант 17

Создать автоматизированную информационную систему, предназначенную для учета горюче-смазочных материалов (ГСМ).

Предприятие имеет несколько автобаз. На каждой автобазе, о которой известны код, название и адрес, ведется учет горюче-смазочных средств (ГСМ), заправляемых в автомобили, выполняющие рейсы по соответствующим путевым листам.

В путевом листе отражается информация о рейсе: пункты погрузки и разгрузки, пробег общий и с грузом, наименования грузоотправителя и грузополучателя, время в наряде (в днях и/или часах).

При заправке автомобилей в гараже формируется раздаточная ведомость, в которой указаны: номер ведомости, дата. В одной ведомости могут быть оформлены данные на нескольких водителей. В каждой позиции ведомости записывается: марка автомобиля, государственный регистрационный номер автомобиля, номер путевого листа, фамилия, инициалы водителя, количество заправленного ГСМ, в литрах и килограммах. ГСМ – это бензин, дизтопливо, дизмасло, автол, солидол, нигрол и т.п. Для каждого автомобиля на одну поездку может быть выделено несколько видов ГСМ. Ведомость подписывает сотрудник, имеющий должность заправщика. Указываются его ФИО. В заголовке ведомости

указывается автобаза, которой принадлежит автомобиль и гараж. У каждой автобазы может быть несколько гаражей, расположенных по различным адресам. Ведомости формируются отдельно для каждого гаража заправщиком гаража.

Перечень возможных типовых запросов:

1. Для каждой автобазы указать количество закрепленных за ней автомобилей.
2. Вывести список водителей, заправлявшихся в те же дни, что и заданный водитель.
3. Для каждого рейса вывести общий объем топлива в литрах и килограммах в заданный промежуток времени.
4. Вывести общий объем отпущенного топлива на предприятии по каждому виду в заданный промежуток времени.
5. Для заданного водителя вывести информацию о всех его заправках с указанием номеров путевых листом и общим объемом отпущенных ГСМ по каждому путевому листу по каждому виду.

Вариант 18

Создать автоматизированную информационную систему, предназначенную для управления договорами страхования с физическими лицами и юридическими организациями.

Страховая организация заключает договоры страхования.

Для организации оформляется коллективный договор, в котором перечислены страхуемые сотрудники: ФИО, возраст, категория риска (первая, вторая, высшая и т.п.). О предприятии хранится следующая информация: код, полное наименование, краткое наименование, адрес, банковские реквизиты (номер банка), специализация предприятия (медицинское учреждение, автотранспортное предприятие, учебное заведение и т.п.). В заключаемом коллективном договоре указывается дата заключения, срок договора (начало и конец действия договора), сумма выплат по каждой категории сотрудников, выплаты по страховым случаям. Выплаты зависят от категории сотрудника. Необходимо также хранить информацию о страховом агенте, заключившем договор (ФИО, паспортные данные, контактные данные). Каждый агент может заключить много договоров, в каждом договоре может быть оформлено несколько сотрудников.

С физическим лицом заключается индивидуальный договор. Каждый конкретный договор может быть заключен только одним агентом.

При возникновении страхового случая необходима информация о его дате, причине, решении о выплате страховой суммы и размере выплаты.

Директор компании должен иметь возможность принять и уволить на работу страхового агента. Поэтому должна сохраняться информация о заключенных с ними трудовых договорах.

Перечень возможных типовых запросов:

1. Для заданной организации вывести список других организаций, застрахованных теми же агентами, что и заданная, для действующих договоров.
2. Для каждого агента вывести количество заключенных им договоров каждого типа за определенный период времени.
3. Для заданной персоны вывести список застрахованных сотрудников в одном коллективном договоре для действующих коллективных договоров.
4. Вывести общую сумму выплат по каждому типу договоров при возникших страховых случаях за заданный период времени.
5. Для каждого юридического лица вывести реквизиты договора и общую сумму выплат по всем категориям сотрудников.

Вариант 19

Создать автоматизированную информационную систему, предназначенную для учета животных, птиц, рептилий (далее по тексту – животных) в зоопарке.

Каждому новому питомцу зоопарка присваивается уникальный номер, имя. Необходимо также хранить дату рождения, пол. О птицах дополнительно необходимо хранить сведения о месте зимовки (если такое существует – код, название страны, дата улета, дата прилета). Для рептилий необходимо хранить сведения об их нормальной температуре, сроках зимней спячки.

Каждому питомцу назначается рацион кормления, который характеризуется номером, названием, типом (детский, диетический, усиленный и т.п.). Каждый тип рациона может содержать несколько рационов, отличающихся по содержанию. Рацион может со временем меняться.

Необходимо знать зону обитания животного (название, местоположение (материк, страна), характеристика). Каждое животное относится к одной зоне обитания.

Животное может быть собственностью зоопарка или взято в аренду. Тогда необходима информация о зоопарке-владельце, сроках и стоимости аренды. Зоопарк также может предоставлять животных в аренду другим зоопаркам. Если животное стало собственность зоопарка в результате покупки, то нужно знать дату поступления в зоопарк и организацию-продавца.

Территория зоопарка разделена на отделы (грызуны, хищники, птицы и т.д.). Каждое животное размещается в отделе в определенном вольере. В некоторых вольерах могут размещаться одновременно несколько животных. Такие вольеры называются «коммунальными квартирами». Животных могут пересаживать из вольера в вольер в одном отделе. Несколько вольеров могут размещаться в одном здании («летнем» или «зимнем»). Каждое здание закреплено за одним отделом. Вольеры могут быть изолированными. Вольеры могут иметь дополнительные параметры (наличие бассейна, дополнительное оборудование, внутреннее помещение и т.д.).

Необходимо хранить информацию о том, к какому смотрителю на текущий момент прикреплен питомец. За каждым животным закреплены несколько смотрителей, а каждый смотритель одновременно может обслуживать нескольких животных.

В зоопарке есть ветеринары, которые закреплены за животными. Каждый сотрудник имеет табельный номер, ФИО, дату рождения. Каждый ветеринар может обслуживать несколько животных, и каждое животное может обслуживаться несколькими ветеринарами.

Необходимо знать номер телефона и электронную почту (при наличии) сотрудников.

Перечень возможных типовых запросов:

1. Для каждого отдела зоопарка вывести общее количество животных в отделе.
2. Вывести список всех животных, размещающихся в «коммунальных квартирах».
3. Для заданного животного вывести список животных, размещенных в том же здании, что и это животное.
4. Вывести список пустых вольеров.
5. Для каждого из зоопарков, предоставивших животных в аренду, вывести общее количество животных в аренде и общую стоимость.

Вариант 20

Создать автоматизированную информационную систему, предназначенную для предприятия по благоустройству парков.

Предприятие оказывает такие виды услуг, как: формирование ландшафтов, насаждение парков, озеленение улиц и скверов. Фирма имеет название, юридический адрес, код по ЕГРЮЛ.

Для обслуживаемого объекта необходимо знать название, юридический адрес, код по ЕГРЮЛ.

У объекта может быть несколько декораторов. О декораторах парка необходимо хранить информацию о ФИО, телефоне, адресе, образовании, названии законченного учебного заведения, категории (высшая, первая, без категории).

Каждый обслуживаемый объект делится на зоны.

Каждому высаживаемому растению присваивается уникальный номер в пределах зоны. Необходимо хранить дату высадки растения и возраст растения. Растение может быть высажено в многолетнем возрасте. Каждое растение относится к какому-либо одному виду и жизненной форме (дерево, кустарник, лиана и т.д.), характеризуется временем возможной высадки и в грунт, временем цветения, особыми характеристиками.

Режим полива каждого растения зависит от возраста растения и его вида. Каждый полив характеризуется днем (каждый день, один раз в неделю и т.п.), временем полива, нормой воды в литрах в зависимости от сезона. Насаждения поливаются максимум один раз в день.

Необходимо иметь информацию о сотрудниках, обслуживающих объект, которые ухаживают за насаждениями (ФИО, телефон, адрес). Каждый служитель закрепляется за насаждением по рабочему графику (дата). На каждую дату закреплен за насаждением только один сотрудник.

Перечень возможных типовых запросов:

1. Вывести информацию о количестве обслуживаемых и необслуживаемых объектов на текущую дату.
2. Для каждого сотрудника вывести количество объектов, которые он обслуживает на текущую дату.
3. Для заданного сотрудника вывести список сотрудников, работающих на тех же объектах, что и заданный.
4. Найти самый популярный по количеству высаженных единиц вид растения на обслуживаемых объектах за заданный период.
5. Для каждого сотрудника вывести количество обслуживаемых растений на каждом объекте в заданный период времени.

СПИСОК ИСТОЧНИКОВ

1. Что такое сокет и зачем он нужен // КОД : журнал Яндекс Практикума : сайт. – URL: https://thecode.media/socket/ (дата обращения: 15.02.2026).
2. Величко И. Пишем свой веб-сервер на Python: протокол HTTP // Серверная разработка с Иваном Величко : сайт. – URL: https://iximiuz.com/ru/posts/writing-python-web-server-part-3/ (дата обращения: 15.02.2026).
3. Индекс пакетов Python (PyPI) : сайт. – URL: https://pypi.org/ (дата обращения: 01.02.2026).
4. Диспетчер URL // DJANGO.FUN : сайт. – URL: https://django.fun/ru/docs/django/3.2/topics/http/urls/ (дата обращения: 01.02.2026).
5. Представления на основе классов // DJANGO.FUN : сайт. – URL: https://django.fun/ru/docs/django/3.2/topics/class-based-views/ (дата обращения: 22.02.2026).
6. Модели // DJANGO.FUN : сайт. – URL: https://django.fun/ru/docs/django/3.2/topics/db/models/ (дата обращения: 22.02.2026).
7. Шаблоны // DJANGO.FUN : сайт. – URL: https://django.fun/ru/docs/django/3.2/ref/templates/ (дата обращения: 22.02.2026).
8. Documentation. Model field reference // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.2/ref/models/fields/ (дата обращения: 25.02.2026).
9. Documentation. Model Meta options // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.2/ref/models/options/#table-names (дата обращения: 25.02.2026).
10. DB Browser for SQLite : офиц. сайт. – URL: https://sqlitebrowser.org/ (дата обращения: 27.02.2025).
11. The Django admin site // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.0/ref/contrib/admin/ (дата обращения: 20.02.2026).
12. Documentation. Writing views // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.0/topics/http/views/#the-http404-exception (дата обращения: 26.02.2026).

13. Documentation. URL dispatcher // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.0/topics/http/urls/#example (дата обращения: 26.02.2026).
14. Documentation. Built-in class-based generic views // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.0/topics/http/urls/#example (дата обращения: 26.02.2026).
15. Клименко Д. Стратегии расширения Django User Model // Хабр : сайт. 2006 – 2023. – URL: https://habr.com/ru/post/313764/#AbstractUser (дата обращения: 26.02.2026).
16. Documentation. Managers // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.2/topics/db/managers/ (дата обращения: 26.02.2026).
17. Documentation. QuerySet API reference // Django : сайт. 2005 – 2023. – URL: https://docs.djangoproject.com/en/3.2/ref/models/querysets/#field-lookups (дата обращения: 26.02.2026).
18. SOAP // Википедия : свободная энциклопедия : сайт. – URL: https://ru.wikipedia.org/wiki/SOAP (дата обращения: 16.02.2026).
19. CodeX Docs // Github : сайт. – URL: https://github.com/codex-team/codex.docs/blob/main/README.md (дата обращения: 16.03.2026).
20. Gist markdown examples // Github : сайт. – URL: https://gist.github.com/ww9/44f08d44327a40d2ab309a349bebec57 (дата обращения: 16.03.2026).
21. Basic Syntax // Markdown Guide : сайт. – URL: https://www.markdownguide.org/basic-syntax/ (дата обращения: 16.03.2025).
22. API Development for Everyone // Swagger Logo : сайт. – URL: https://swagger.io/ (дата обращения: 10.01.2024).
23. Make API docs your superpower // Redocly : сайт. – URL: https://redocly.com/ (дата обращения: 10.01.2025).
24. MkDocs. Project documentation with Markdown : офиц. сайт. URL: https://www.mkdocs.org/ (дата обращения: 10.01.2025).
25. Read the Docs : сайт. – URL: https://readthedocs.org/ (дата обращения: 10.01.2026).
26. Swagger Petstore // Swagger : сайт. – URL: https://petstore.swagger.io/ (дата обращения: 10.01.2025).

27. Django REST Swagger // Django REST Swagger : сайт. – URL: https://django-rest-swagger.readthedocs.io/en/latest/ (дата обращения: 01.01.2026).
28. Имплементация Swagger в Django // Github : сайт. – URL: https://github.com/TonikX/ITMO_ICT_WebDevelopment_Examples/tree/documentation-example-add-swagger/example_0611 (дата обращения: 10.01.2026).
29. MkDocs Themes // Github : сайт. – URL: https://github.com/mkdocs/mkdocs/wiki/MkDocs-Themes (дата обращения: 10.01.2026).
30. MkDocs Plagins // Github : сайт. – URL: https://github.com/MkDocs/MkDocs/wiki/MkDocs-Plugins#api-documentation-building (дата обращения: 15.01.2026).
31. Описание сервиса «Войны» // Github : сайт. – URL: https://tonikx.github.io/ITMO_ICT_WebDevelopment_Examples/professions/picking_herbs/ (дата обращения: 15.01.2026).
32. Node.js : офиц. сайт. – URL: https://nodejs.org/en (дата обращения: 05.01.2026).
33. Vue.js devtools // Интренет-магазин Chrome : сайт. – URL: https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=RU (дата обращения: 15.01.2026).
34. Routing // Vue.js : сайт. – URL: https://vuejs.org/guide/scaling-up/routing.html (дата обращения: 17.01.2026).
35. Getting Started // Vue.js : сайт. – URL: https://vuejs.org/tutorial/#step-1 (дата обращения: 17.01.2026).
36. ITMO_ICT_WebDevelopment_Examples // Github : сайт. – URL: https://github.com/TonikX/ITMO_ICT_WebDevelopment_Examples (дата обращения: 20.01.2026).
37. Vue Component Framework // VUETIFY : сайт. – URL: https://vuetifyjs.com/en/ (дата обращения: 19.01.2026).
38. How to Use a Button as a Link in Vue // </CB> Coding Beauty : сайт. – URL: https://codingbeautydev.com/blog/vue-button-link/ (дата обращения: 19.01.2024).
39. Пример создания формы // Vue SFC Playground : сайт. – URL: https://sfc.vuejs.org/#eNp9UkFOwzAQ/MrKl4LUJPcoIBUewAFumENotjRVYlvrTVop6t/Z2HFUgUCKIo93djzj9aR2zuXjgKpUld9T6/hRG7w4SwwNHuqhY5i0AWhqru/u4xqAkAcyCQF47HDP2JSw2W22adc6bq3xJbynHYAJGC8svBeDmy2MdTdg6ILr2nfDejvbG9bTX6wj4a3as/AS7SMuApaffFWxJhXA2LuuZhQEUMUgMGa9bbB70Col0yoQhBJTCeVgSQgLbE2KqxWUwchazANcFcT4tJDzOQBcF7NVEXejlSIeHVxqrpp2fHxdr1kUkjNpr4q5Oidb06itavt5jFlfu/zkrZERh3nppSBGRSeerJ

W8gRlrdWR2viwKf9jPD+Pkc0tfhaxyGgy3Pebo++yT7NkjibBWy0SChpimESkjNA0S0n+aP6i/dNPE1PUbO4beKQ== (дата обращения: 20.01.2025).

40. State Management Talks // AllTalks : сайт. – URL: https://alltalks.dev/talk/state-management-talks (дата обращения: 20.01.2025).
41. Pinia. The intuitive store for Vue.js // Vue Mastery : сайт. – URL: https://pinia.vuejs.org/ (дата обращения: 20.01.2026).
42. Агент пользователя // Mdn Web Docs : сайт. – URL: https://developer.mozilla.org/ru/docs/Glossary/User (дата обращения: 19.01.2024).
43. XMLHttpRequest // Mdn Web Docs : сайт. – URL: https://developer.mozilla.org/ru/docs/Web/API/XMLHttpRequest (дата обращения: 19.01.2026).
44. Fetch API // Mdn Web Docs : сайт. – URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (дата обращения: 19.01.2026).
45. Same-origin policy // Mdn Web Docs : сайт. – URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (дата обращения: 19.01.2026).
46. Django-cors-headers // Github : сайт. – URL: https://github.com/adamchainz/django-cors-headers#configuration (дата обращения: 19.02.2026).
47. BootstrapVue : офиц. сайт. – URL: https://bootstrap-vue.org/ (дата обращения: 19.01.2026).

СПИСОК РЕКОМЕНДОВАННЫХ ИСТОЧНИКОВ

1. Сокеты в Python 3: TCP, клиент, сервер. // URL: https://andreymal.org/socket3/ (дата обращения: 20.02.2025).
2. Socket Programming HOWTO // Python : сайт. – URL https://docs.python.org/3.6/howto/sockets.html (дата обращения: 20.02.2026).
3. socket — Low-level networking interface // Python : сайт. – URL: https://docs.python.org/3.6/library/socket.html (дата обращения: 20.02.2026).
4. Сокеты в Python для начинающих // Github : сайт. – URL: https://habr.com/ru/post/149077/ (дата обращения: 20.02.2026).
5. Сокеты в Python 3: TCP, клиента сервер // URL: https://habr.com/ru/post/149077/ (дата обращения: 20.02.2026).
6. Sockets Tutorial with Python 3 part 1 - sending and receiving data // URL: https://www.youtube.com/watch?v=Lbfe3-v7yE0 (дата обращения: 20.02.2026).
7. Python socket // ZetCode : сайт. – URL: http://zetcode.com/python/socket/ (дата обращения: 20.02.2025).
8. threading – Tread-based parallelism // Python : сайт. – URL: https://docs.python.org/3/library/threading.html (дата обращения: 20.02.2026).
9. Введение в потоки в Python // Еще один блог веб-разработчика : сайт. – URL: https://webdevblog.ru/vvedenie-v-potoki-v-python/ (дата обращения: 20.02.2025).
10. Django. Documentation. Models // Django : сайт. – URL: https://docs.djangoproject.com/en/3.0/topics/db/models/ (дата обращения: 20.02.2026).
11. Типы данных в Django ORM // METANIT.COM : сайт. – URL: https://metanit.com/python/django/5.2.php (дата обращения: 20.02.2026).
12. Django. Documentqtion. URL dispatcher // Django : сайт. – URL dispatcher: (URL: https://docs.djangoproject.com/en/3.2/topics/http/urls (дата обращения: 20.02.2026).
13. Обучающее видео о создание модели базы данных // URL: https://www.youtube.com/watch?v=LZyk9p0tKXc (дата обращения: 20.02.2026).
14. Django Rest framework. Class-based Views // Django Rest framework: сайт. – URL: https://www.django-rest-framework.org/api-guide/views/ (дата обращения: 20.02.2026).

15. DJANGO API VIEWS, GENERICS, FILTER // URL: https://www.youtube.com/watch?v=AHnBL9x6-rs (дата обращения: 20.02.2026).
16. JSON. Сериализация данных. Пишем свой сериализатор. Разбираем Django REST Framework Serializers // URL: https://youtu.be/sxdPf3z6Uw8 (дата обращения: 20.02.2026).
17. DRF + Djoser. Часть 1. Регистрация, авторизация по токенам, получение и изменение данных пользователя // URL: https://www.youtube.com/watch?v=NT-cI6rJl5Q (дата обращения: 20.02.2026).
18. Настройка авторизации средствами Vue.js и Django REST framework // URL: https://youtu.be/dnRIbgZU-Gk (дата обращения: 20.02.2026).

Говоров Антон Игоревич
Коряков Сергей Алексеевич
Шнайдер Полина Анатольевна
Говорова Марина Михайловна

WEB-ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А