Методические указания по дисциплине
«Администрирование информационных систем»
Лабораторная работа № 2

Часть 1

Проектирование и создание базы данных (БД) информационной системы. Реализация структуры БД с использованием технологии «объектно-реляционного отображения» - Object-Relational Mapping (ORM).

***Задание*:**

1. Спроектировать БД информационной системы, описание которой было представлено на сайте в результате выполнения части 1. В БД должно быть определено не менее 5 сущностей (таблиц), при этом для ряда таблиц должны быть определены связи через внешние ключи (Foreign Keys).

2. Разработать приложение, использующее технологию ORM для представления таблиц (сущностей) спроектированной БД.

3. Реализовать основные CRUD-операции (***C**reate, **R**ead, **U**pdate, **D**elete*) для работы с БД, при этом необходимо учитывать специфику выбранного технического объекта, т.е. функции работы с информацией в БД должны предоставлять/изменять необходимые данные таким образом, как это могло бы быть на реальном объекте.

4. Используя разработанное приложение создать тестовую БД, сгенерировать таблицы и произвести популяцию БД (добавить несколько тестовых записей).

5. Написать тесты для проверки основных CRUD-операций.

6. Настроить сетевой доступ к серверу БД на виртуальной машине и проверить работу приложения на данном сервере.

Требования:

- Ознакомиться (вспомнить) основы проектирования БД: подходы, нормализация.
- Спроектировать БД, которая должна содержать не менее 3 таблиц (сущностей), при этом в структуре должны быть представлены связанные таблицы.
- Описание структуры, спроектированной БД должно быть представлено в виде ER-диаграммы.
- Разработку приложения можно вести на любом кроссплатформенном языке программирования, имеющем ORM framework. Ознакомиться с подходами, которые могут использоваться в ORM на уровне доступа к данным (DAO).
- Код приложения должен содержать комментарии в части описания моделей таблиц и бизнес-логики для работы с хранилищем данных.
- Код Python должен быть оформлен с использованием аннотаций типов (подробнее: https://habr.com/ru/company/lamoda/blog/432656/).
- Загрузить проект на портал GitHub (см. памятку для работы Git).

Методические рекомендации к выполнению лабораторной работы

1. Рекомендуемое программное обеспечение

1) DBeaver Community Edition– универсальный клиент для удаленной работы с БД:

https://dbeaver.io/files/dbeaver-ce-latest-x86_64-setup.exe

2) Клиент для передачи файлов по протоколу SSH - WinSCP (для Windows):

https://winscp.net/download/WinSCP-5.21.2-Setup.exe

3) Среда разработки Python (IDE): IDLE (устанавливается автоматически вместе со средой Python), PyCharm, VSCode.

ВАЖНО! Методические рекомендации по работе с различными ORM фреймворками изложены далее в п. 2.2 (подготовка среды разработки) и п.3 (Python **SQLAlchemy ORM** – стр. 8, Python **Django ORM** – стр. 12, Java **Spring JPA** – стр. 17)

2. Проектирование и создание БД

2.1 Проектирование БД. С основными методами проектирования БД можно ознакомиться здесь:

https://metanit.com/sql/tutorial/1.1.php

Более подробный методический материал:

https://habr.com/ru/post/535588/

Совет: если после ознакомления с теоретическим материалом с представлением БД в виде ER-диаграммы все еще возникают сложности, то на начальном этапе можно отталкиваться не от общей структуры, а от примера данных (data sample).

Допустим, в качестве тематики проекта была выбрана информационная система для сбора данных с метеостанций, тогда первая сущность, с которой будет работать система, (очевидно) будет «погода» (weather), и данные могут быть представлены в виде json-строки следующим образом:

```
{
"weather": {
     "temperature_c": 22.0,
     "temperature_f": 72.0,
     "pressure": 745,
     "type": "CLEAN"
     "city": "UFA"
}
}
```

Атрибуты: "temperature_c" – температура в град. по шкале Цельсия, "temperature_f" – температура в град. Фаренгейта, "pressure" – атмосферное давление, "type" – тип погоды («облачно», «ясно» и т.п.), "city" – населенный пункт.

Для предоставления статистических данных по изменению погоды необходимо будет хранить несколько подобных записей, но с разными временными промежутками, в таком случае, добавим два дополнительных поля с временными метками (timestamp) моментов создания и обновления записи:

```
{
"weather": {
     "temperature_c": 22.0,
     "temperature_f": 72.0,
     "pressure": 745,
     "type": "CLEAN"
     "city": "UFA"
     "updated_on": "2022-08-24 12:00:00",
     "created_on": "2022-08-24 10:00:00"
}
}
```

По представленному примеру данных видно, что поля "type" и "city" – это повторяющиеся значения для различных записей типа «weather», в таком случае, при нормализации эти значения можно вынести в отдельные таблицы («weather_type» и «city») с привязкой через внешний ключ (Foreign Key). Тогда получим 3 связанные сущности:

```
{
"weather": {
     "temperature_c": 22.0,
     "temperature_f": 72.0,
     "pressure": 745,
     "type": FK
     "city": FK
     "updated_on": "2022-08-24 12:00:00",
     "created_on": "2022-08-24 10:00:00"
}
}
{
"weather_type": {
     "id": PK,
     "type": "CLEAN"
}
}
{
"city": {
     "id": PK,
     "name": "UFA"
}
}
```

Дополнительную информацию по нормализации БД можно посмотреть здесь: https://docs.microsoft.com/ru-ru/office/troubleshoot/access/database-normalization-description

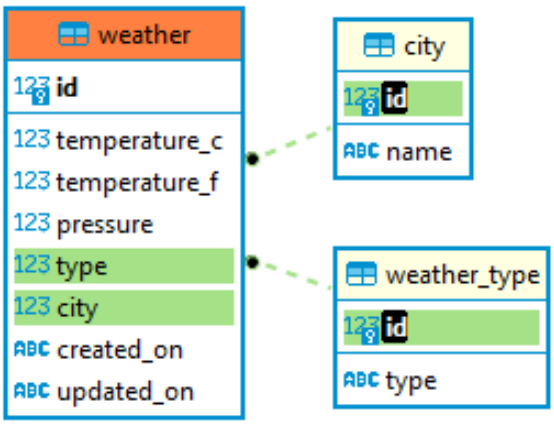


Рис. 2.1 – Диаграмма БД из рассмотренного примера

2.2 Подготовка среды и необходимых библиотек ORM. Создаем новую БД и нового пользователя для него, используя методические рекомендации из ЛР №1 (п. 5.4).

Реализация спроектированной структуры реляционной БД возможна следующими способами:

1) С помощью языка SQL.

2) С помощью технологии «объектно-реляционного отображения» - **Object-Relational Mapping (ORM)**, которая используется в современных языках программирования, поддерживающих объектно-ориентированный подход (Python, Java, C# и т.п.).

В рамках методических рекомендаций будут рассмотрены примеры работы со следующими ORM Frameworks:

- **`Python SQLAlchemy (SQLA)`**
- **`Python Django ORM`**
- **`Java Spring JPA`**

ВАЖНО! Django ORM и SQLA используют два различных подхода к реализации DAO (объектов доступа к данным). Django ORM реализует подход «**`Active Record`**» - это означает, что DAO в Django напрямую соединяется с данными, а также реализует внутри себя бизнес-логику для выполнения SQL-запросов. SQLA, в свою очередь, реализует подход «**`Data Mapper`**», где DAO является только контейнером для хранения данных, а управление соединениями к БД и SQL-запросами осуществляется с помощью отдельных функций фреймворка.

2.2.1 Подготовка среды Python. Для реализации проекта на языке программирования Python установите интерпретатор Python версии 3.9+. Не забудьте добавить путь к интерпретатору в переменные окружения ОС:

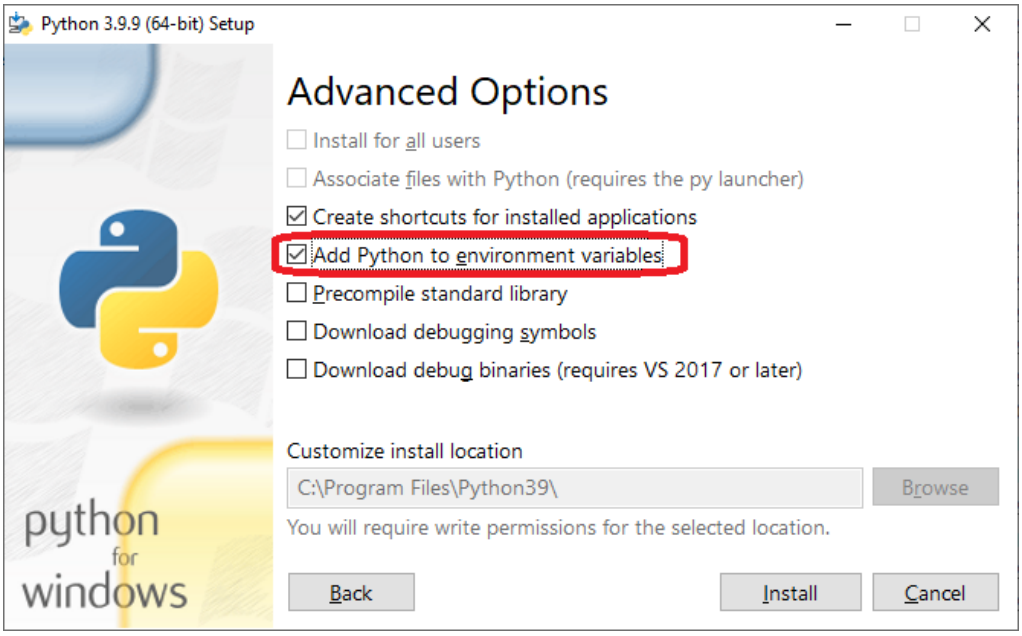
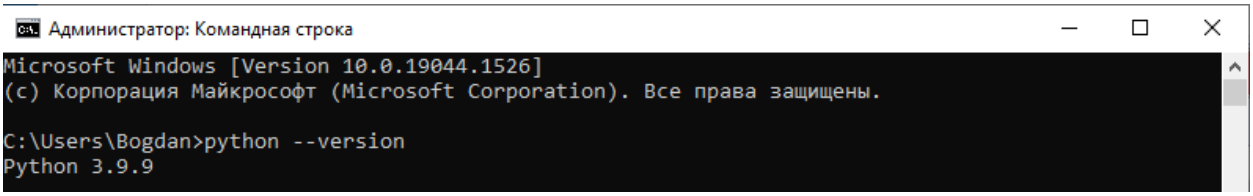


Рис. 2.2 – Установка переменных окружения ОС

Проверяем корректность установки. Запускаем командную строку Windows или Windows PowerShell и вводим:



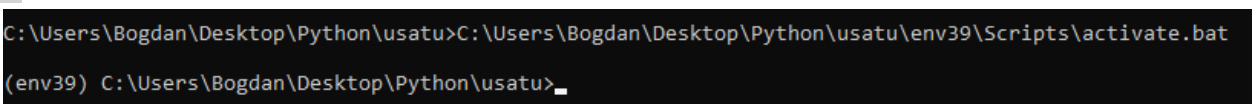
ВАЖНО! Чтобы не засорять системный интерпретатор Python пакетами и избежать конфликтов между различными версиями библиотек, необходимо создать отдельную виртуальную среду Python для реализации нового проекта. Для этого через ту же командную строку переходим в папку, где будет располагаться виртуальная среда, например:

Создаём в данной директории виртуальную среду Python в папке **env39** командой:

```
python -m venv env39
```

Теперь необходимо активировать созданную виртуальную среду:

```
C:\Users\Bogdan\Desktop\Python\usatu\env39\Scripts\activate.bat
```



Перед началом работы с проектом **в IDE не забудьте установить созданную виртуальную среду Python** в качестве основного интерпретатора **(File -> Settings):**

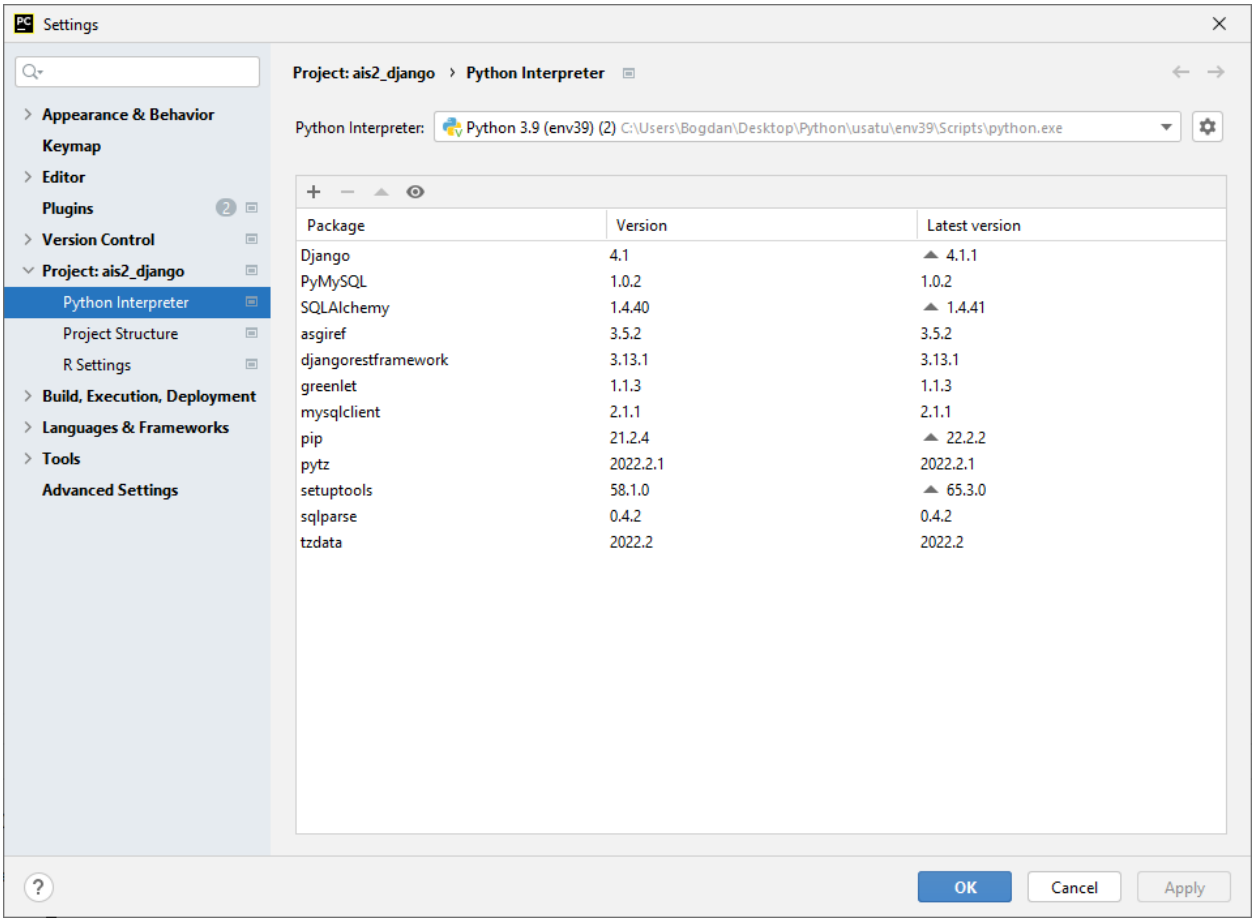


Рис. 2.3 – Установка интерпретатора Python в IDE PyCharm

2.2.2 Подготовка среды Java. Для реализации проекта используется сборщик пакетов **Maven** и среда **Liberica OpenJDK 8**. Установщик JDK можно скачать по ссылке:

https://bell-sw.com/pages/downloads/#mn

В процессе установки не забудьте установить системные переменные PATH и JAVA_HOME:

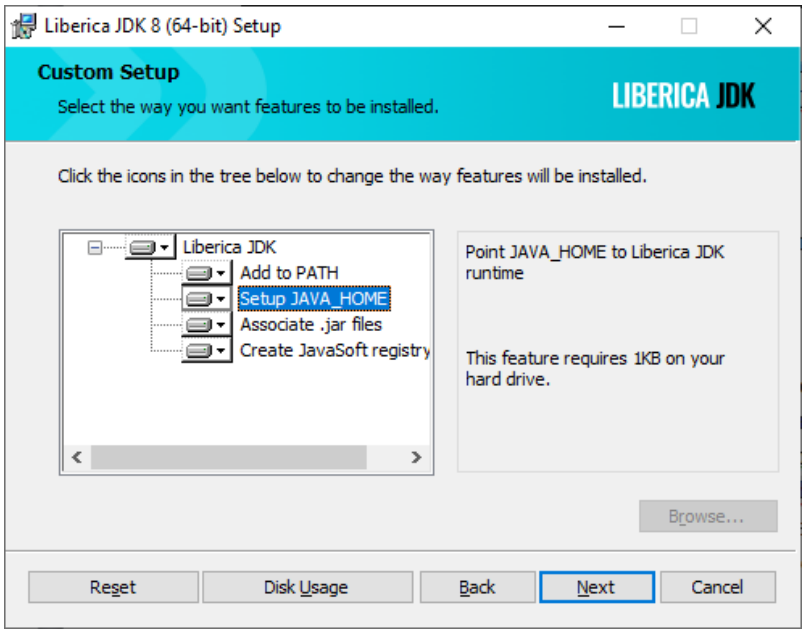


Рис. 2.4 – Установка переменных окружения ОС

Перед началом работы с проектом **в IDE не забудьте установить соответствующую среду разработки Java (SDK)** в настройка проекта **(File -> Project Structure):**

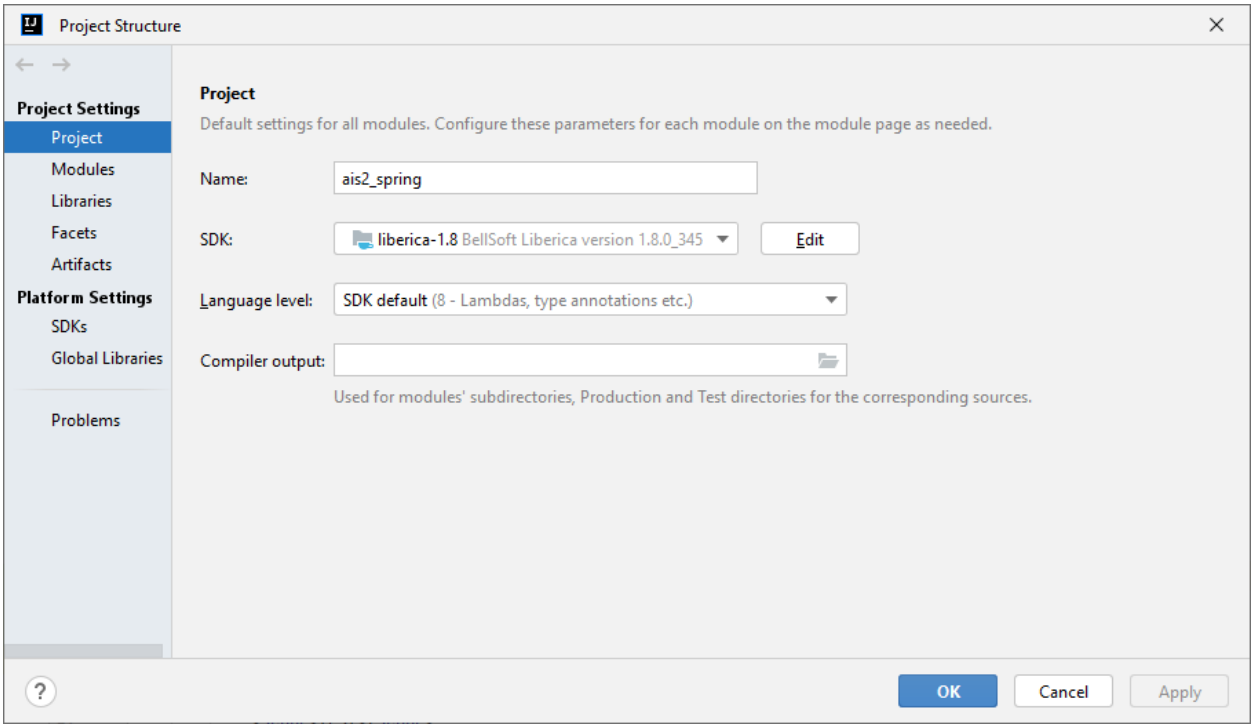


Рис. 2.5 – Установка Java SDK в IDE IDEA

3. Работа с ORM Framework

3.1 Пример ORM-приложения для работы с БД (Python SQLAlchemy ORM framework). После активации среды устанавливаем необходимые пакеты в виртуальную среду, используя пакетный менеджер **pip**:

```
pip install sqlalchemy
```

В методических рекомендациях представлен код Python приложения, использующего ORM (**`ais2_fastapi.zip`**). Рассмотрим структуру данного приложения (рис. 3.1):

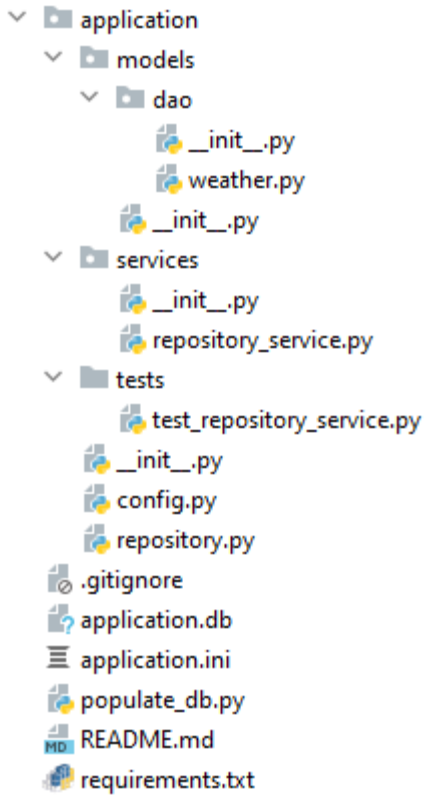


Рис. 3.1 – Структура Python-приложения для работы с БД

`__init__.py` – файл инициализации пакета python. Данный файл может быть использован для инициализации каких-либо общих для пакета переменных, также в данный файл можно сделать импорт необходимых функций или объектов из внутренних модулей (см. файл **application/models/dao/__init__.py**), тогда, все импортированные в **__init__.py** объекты и модули можно будет подключить в любом внешнем модуле общим импортом через import <имя_пакета>.

`settings.py` – файл инициализации конфигурации приложения (парсинг *.ini файла, подключение к БД).

`application.ini` – основной файл с параметрами конфигурации приложения. Стандартные библиотеки Python содержат средства для парсинга **`*.ini`** файлов конфигурации (подробнее см.: https://docs.python.org/3/library/configparser.html). В качестве современной альтернативы могут использоваться файлы конфигурации формата ***.yml** (подробнее см.: https://pypi.org/project/PyYAML).

`repository.py` – содержит функции инициализации подключения к источнику данных (драйвер (engine) подключения к БД, а также фабрику для создания сессий подключения к БД).

`application.models.dao` – пакет, содержащий классы, которые представляют собой объектно-реляционное отображение структуры БД. Таким образом, экземпляры данных классов (***Data Access Object*** или сокр. **DAO**) будут содержать данные соответствующих таблиц БД.

`services.repository_service.py` – программный слой работы с хранилищем данных. Содержит функции (бизнес-логику) для работы с БД.

`tests.test_repository_service` – модуль с реализацией тестов основных CRUD-функций приложения (по соглашению наименование тестовых модулей должно содержать префикс **test_***, а остальная часть имени обозначать тестируемых модуль или класс).

`application.db` – файл локальной тестовой БД SQLite. Генерируется автоматически при запуске приложения.

`requirements.txt` – файл со списком зависимостей (необходимых библиотек) python-приложения. В дальнейшем, например, при развертывании или переносе проекта на другой ПК все необходимые библиотеки можно будет поставить одной командой:

```
pip install -r /path/to/requirements.txt
```

ВАЖНО! Код, представленный в примере, подробно комментирован. Более детально с принципами работы с SQLAlchemy можно ознакомиться здесь:

https://docs.sqlalchemy.org/en/14/core/type_basics.html

ВАЖНО! При разработке CRUD-функций (особенно если используется ORM framework, реализующий подход data mapper) для обеспечения потокобезопасной работы с данными необходимо использовать транзакционный подход при выполнении операций с БД. Т.е., если, например, нам поступает список с несколькими записями о погоде, то нужно предусмотреть операцию, которая обновляет сразу несколько строк в БД в рамках одной транзакции и не использовать операцию, которая для каждой записи будет создавать отдельную сессию.

Так делать НЕ НАДО:

```
class ThingOne():
    def execute_update(self):
        session = Session()
        try:
            session.query(MyDaoOne.update({"x": 5}))
            session.commit()
        except:
            session.rollback()
            raise

class ThingTwo():
    def execute_update(self):
        session = Session()
        try:
            session.query(MyDaoTwo.update({"q": 18}))
            session.commit()
        except:
            session.rollback()
            raise
```

```
def run_my_program():
    ThingOne().execute_update()
    ThingTwo().execute_update()
```

Корректный вариант:

```
class ThingOne():
    def execute_update (self, session):
        session.query(MyDaoOne.update({"x": 5}))

class ThingTwo():
    def execute_update (self, session):
        session.query(MyDaoTwo.update({"q": 18}))

def run_my_program():
    session = Session()
    try:
       # выполняем несколько операций в рамках одной транзакции
        ThingOne().execute_update(session)
        ThingTwo().execute_update(session)
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()
```

3.1.1 Тестирование CRUD-функций (SQLite). Для тестирования функций приложения используется встроенный в Python модуль для тестирования - `unittest`.

Установите в конфигурационном файле `application.ini` подключение к БД SQLite:

```
database_url = sqlite:///application.db
```

Находясь в среде разработки, запустите команду в терминале:

```
python -m unittest
application/tests/test_repository_service.py
```

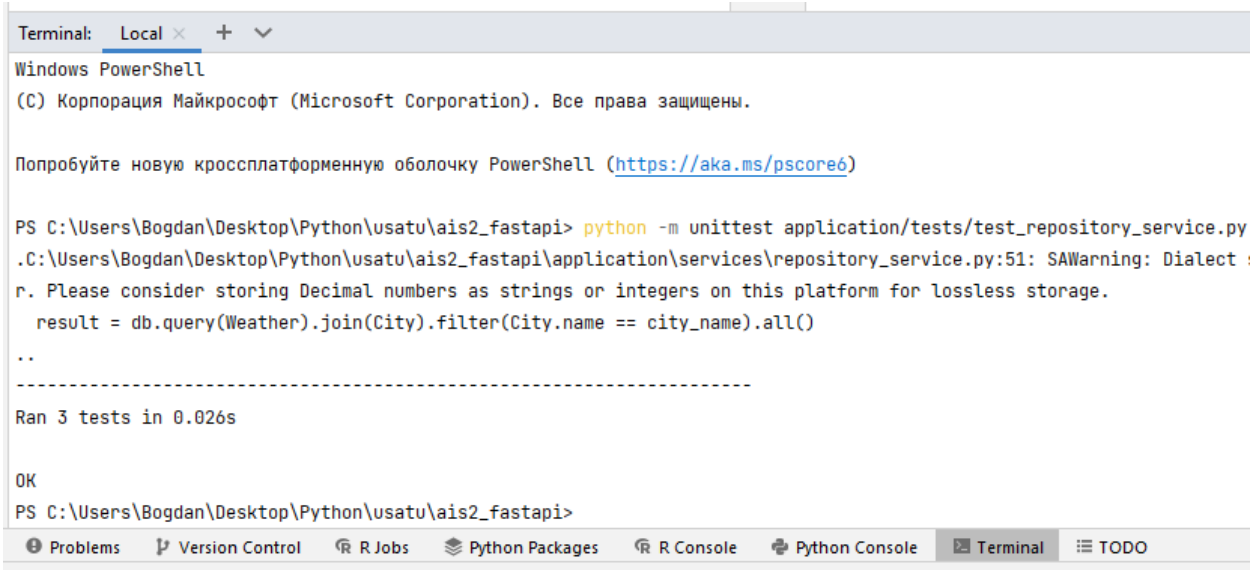


Рис. 3.2 – Тестирование приложения с помощью unittest

Результатом запуска должен быть вывод: «**ОК**»

3.1.2 Тестирование CRUD-функций (MariaDB). Для тестирования функци приложения используется встроенный в Python модуль для тестирования - unittest.

Установите в конфигурационном файле application.ini подключение к БД MatiaDB:

```
database_url = mysql+pymysql://usr:qwerty@192.168.56.104/weatherdb?charset=utf8
```

Для работы с MySQL и MariaDB для SQLA необходима установка дополнительного модуля с соответствующими библиотеками:

```
pip install pymysql
```

Подробнее о настройках подключения к различным БД в SQLA можно посмотреть здесь: https://docs.sqlalchemy.org/en/14/core/engines.html

Запустите ВМ с установленным сервером БД. Создайте БД на сервере и выполните настройку сетевого доступа согласно п.4 методических рекомендаций.

Добавьте в БД необходимые тестовые данные.

Находясь в среде разработки, запустите команду в терминале:

```
python -m unittest application/tests/test_repository_service.py
```

Результатом запуска должен быть вывод: «**ОК**»

3.2 Пример ORM-приложения для работы с БД (Python Django ORM framework). После активации среды устанавливаем необходимые пакеты в виртуальную среду, используя пакетный менеджер **pip**:

```
pip install django djangorestframework
```

В методических рекомендациях представлен код Python приложения, использующего ORM (**`ais2_django.zip`**).

Для создания нового проекта Django перейдите в директорию проекта и введите команду:

```
django-admin startproject application
```

Структура базового проекта Django будет выглядеть следующим образом:

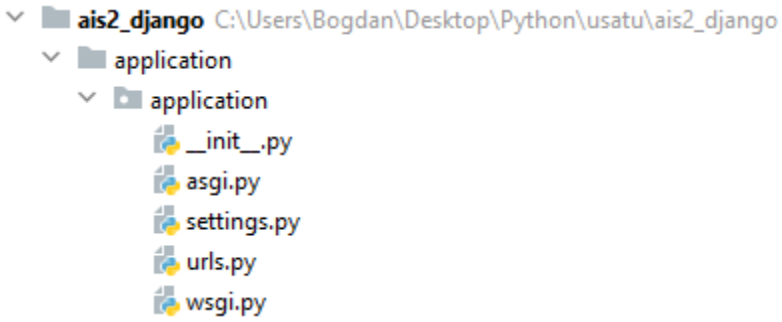


Рис. 3.3 – Структура базового проекта Django

Перейдите в корень проекта: `cd application`

Создайте новое приложение командой:

```
python manage.py startapp weatherapp
```

Для проверки корректности установки запустите приложение командой:

```
python manage.py runserver
```

И перейдите на стартовую страницу Django (по умолчанию http://127.0.0.1:8000/). В браузере должна появиться следующая страница (рис. 3.4):

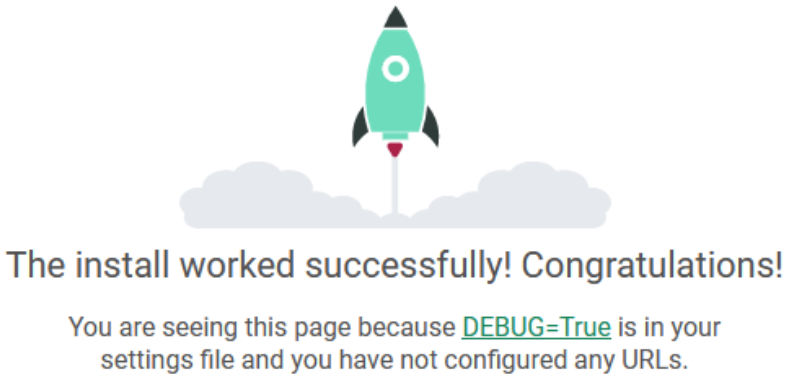


Рис. 3.4 – Стартовая веб-страница Django-приложения по умолчанию

Рассмотрим структуру данного приложения (рис. 3.4):

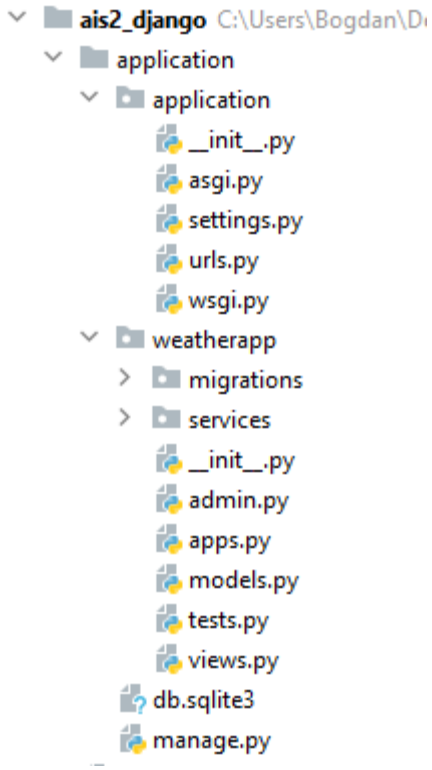


Рис. 3.5 – Структура Django-приложения

`__init__.py` – файл инициализации пакета python. Данный файл может быть использован для инициализации каких-либо общих для пакета переменных, также в данный файл можно сделать импорт необходимых функций или объектов из внутренних модулей (см. файл **application/models/dao/__init__.py**), тогда, все импортированные в **__init__.py** объекты и модули можно будет подключить в любом внешнем модуле общим импортом через import <имя_пакета>.

`manage.py` – управляющий модуль приложения.

`urls.py` – модуль регистрации маршрутов веб-приложения (будет рассмотрен в следующих лабораторных работах).

`wsgi.py` – модуль запуска приложения Django в контейнере веб-сервера (будет рассмотрен в следующих лабораторных работах).

`asgi.py` – модуль запуска асинхронного приложения Django в контейнере веб-сервера.

`settings.py` – модуль конфигурации приложения (регистрация приложений, подключение к БД, авторизация и т.д. Подробнее см.: https://docs.djangoproject.com/en/4.1/ref/settings/).

Описание структуры пакета приложения `wetherapp`:

`models.py` – модуль, содержащий классы, которые представляют собой объектно-реляционное отображение структуры БД. Таким образом, экземпляры данных классов (***Data Access Object*** или сокр. **DAO**) будут содержать данные соответствующих таблиц БД. По умолчанию Django генерирует только одиночный модуль в корне приложения, но хорошей практикой считается выделение моделей в отдельный пакет,

это позволяет группировать модели различного назначения, что особенно важно в приложении с большим количеством моделей.

admin.py – модуль регистрации моделей DAO для дальнейшей работы с ними через администраторскую веб-панель Django.

apps.py – модуль, содержащий конфигурационный класс созданного приложения. Данный класс должен быть зарегистрирован в **setting.py** следующим образом:

```
INSTALLED_APPS = [
    'weatherapp.apps.WeatherappConfig',
…
]
```

services.repository_service.py – программный слой работы с хранилищем данных. Содержит функции (бизнес-логику) для работы с БД.

wsgi.py – модуль-контроллер «отображений» веб-приложения Django (будет рассмотрен в следующих лабораторных работах).

tests.py – модуль с реализацией тестов (test cases) приложения (по соглашению наименование тестовых методов внутри класса **TestCase** должно содержать префикс **test_***, а остальная часть имени обозначать тестируемую функцию).

db.sqlite3 – файл локальной БД SQLite (используется Django по умолчанию). Генерируется автоматически при запуске приложения.

requirements.txt – файл со списком зависимостей (необходимых библиотек) python-приложения. В дальнейшем, например, при развертывании или переносе проекта на другой ПК все необходимые библиотеки можно будет поставить одной командой:

```
pip install -r /path/to/requirements.txt
```

ВАЖНО! Код, представленный в примере, подробно комментирован. Более детально с принципами работы с Django ORM можно ознакомиться здесь:

https://developer.mozilla.org/ru/docs/Learn/Server-side/Django/Models

https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models

3.2.1 Популяция БД с помощью моделей Django. Сгенерируйте базовую структуру таблиц приложения Django с помощью команд:

```
python manage.py makemigrations
python manage.py migrate
```

Вывод должен быть (примерно) следующим:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, weatherapp
Running migrations:
  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 weatherapp.0001_initial... OK
```

Далее необходимо создать суперпользователя администраторской панели Django:

`python manage.py createsuperuser`

1. Введите (и запомните учетные данные).

```
PS C:\Users\Bogdan\Desktop\Python\usatu\ais2_django\application> python manage.py createsuperuser
Username (leave blank to use 'bogdan'): admin
Email address: admin@test.mail
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

2. Для перехода на панель администратора запустите приложение: **`python manage.py runserver`** – и введите в браузере: http://127.0.0.1:8000/admin/

3. Авторизируйтесь, используя учетные данные, введённые в п.1.

4. Используйте панель администратора для заполнения БД тестовыми данными (рис. 3.6):

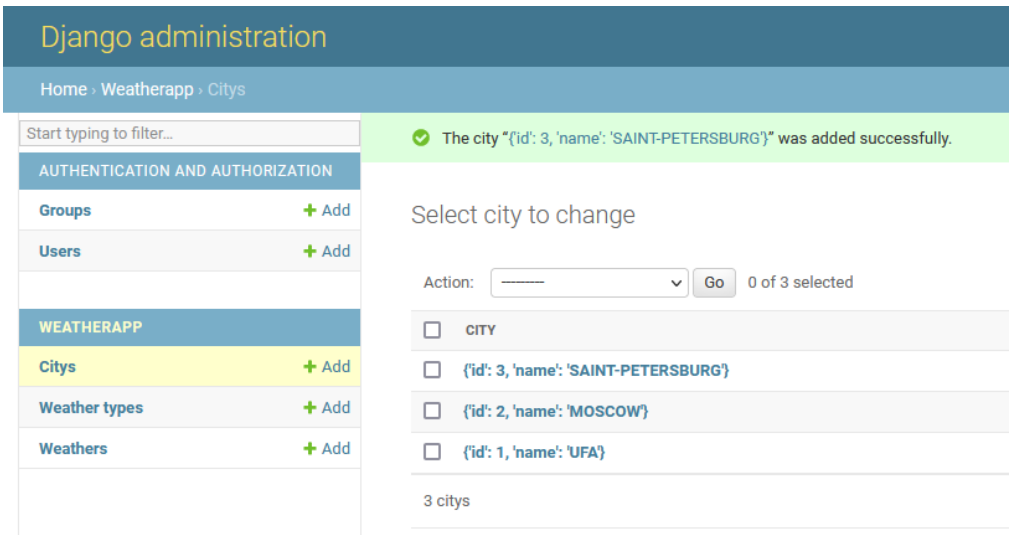


Рис. 3.6 – Панель администратора Django

По умолчанию Django в качестве БД использует SQLite. Для подключения к **MariaDB** установите дополнительный модуль:

```
pip install django mysqlclient
```

Отредактируйте модуль конфигурации **`settings.py`** следующим образом:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'weatherdb',
        'USER': 'usr',
        'PASSWORD': 'qwerty',
        'HOST': '192.168.56.104',
        'PORT': '3306',
    }
}
```

Повторно выполните миграцию, создайте суперпользователя и заполните БД необходимыми данными (**см. п. 3.2.1**).

Подробнее о подключении к различным БД в Django см.: https://docs.djangoproject.com/en/4.1/ref/settings/#std-setting-DATABASE-ENGINE

3.2.2 Тестирование CRUD-функций в Django. Для тестирования Django использует собственную реализацию класса TestCase основанную на стандартном Python модуле для тестирования - **unittest**.

ВАЖНО! При тестировании моделей DAO Django автоматически создаёт тестовую БД с префиксом **`test_*`**, используя подключение **DATABASES** в `settings.py`. Тестовая БД будет автоматически удалена после прогона тестов (подробнее см.: https://docs.djangoproject.com/en/4.1/topics/testing/overview/).

Для запуска тестов верните в setting.py настройки подключения к SQLite и введите команду в терминале:

```
python manage.py test weatherapp.tests
```

```
PS C:\Users\Bogdan\Desktop\Python\usatu\ais2_django\application> python manage.py test weatherapp.tests
Found 2 test(s).
Creating test database for alias 'default'...
Got an error creating the test database: (1044, "Access denied for user 'usr'@'%' to database 'test_weatherdb'")
PS C:\Users\Bogdan\Desktop\Python\usatu\ais2_django\application> python manage.py test weatherapp.tests
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.{'temperature_c': Decimal('26.42'), 'temperature_f': Decimal('79.56'), 'type': <WeatherType: {'id': 1, 'type': 'CLEAN'}>,
_on': datetime.datetime(2022, 9, 11, 21, 18, 59, 136644, tzinfo=datetime.timezone.utc)}
.
----------------------------------------------------------------------
Ran 2 tests in 0.005s

OK
Destroying test database for alias 'default'...
```

Рис. 3.7 – Тестирование приложения с помощью Django

Результатом запуска должен быть вывод: «**ОК**»

3.3 Пример ORM-приложения для работы с БД (Java Spring JPA). Spring – один из ключевых фреймворков для промышленной разработки на ЯП Java (Enterprise Framework). Spring содержит инструменты для реализации различных функциональных слоёв приложения: Web, обмен сообщениями (Messaging Services), работа с SQL и NoSQL хранилищами данных (Persistence Layer). Данный фреймворк построен на концепциях ООП: **IOC** (**Inversion of Control** – инверсия управления, т.е. разработчик лишь определённым образом реализует код архитектурных слоёв приложения, а фреймворк берёт на себя функции связывания и исполнения этого кода); **Dependency Injection** – внедрение зависимостей как основной инструмент связывания программных слоёв Spring-приложения. Подробнее о Spring можно почитать здесь: https://habr.com/ru/post/490586/

В методических рекомендациях представлен код (с комментариями) Java приложения, использующего Spring JPA (**`ais2_spring.zip`**).

После установки JDK, инициализируйте базовый проект **Spring Boot** по ссылке: https://start.spring.io/

В качестве основных зависимостей (dependency) проекта выберите **Spring Web** и **Spring JPA** , а в качестве сборщика зависимостей **Maven** (Maven Project):

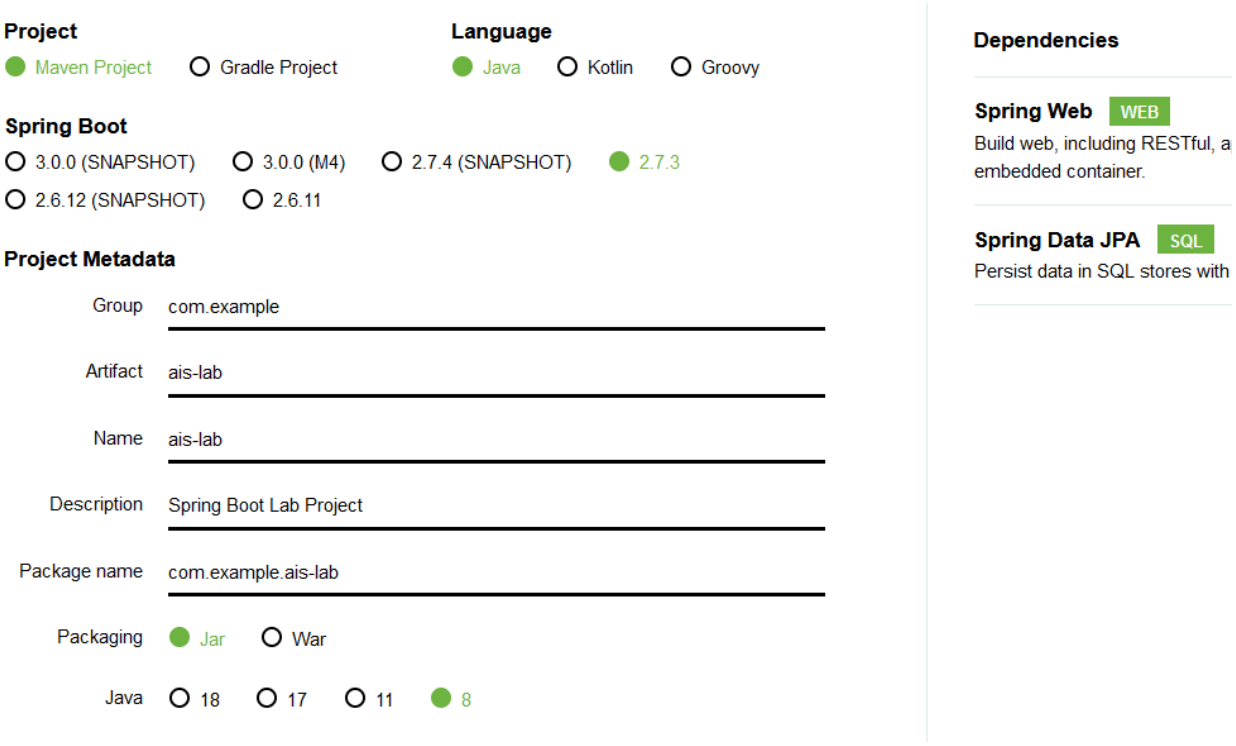


Рис. 3.8 – Инициализация проекта Spring Boot

Распакуйте архив, откройте папку с проектом в вашей среде разработки и установите SDK (см. п. 2.2.2). Базовый проект Spring будет содержать только файл зависимостей проекта **`pom.xml`**, скрипты для управления сборщиком зависимостей **`mvn`**, класс с функцией **`main`** (точка запуска приложения) и пустой файл конфигурации приложения **`application.properties`**.

Чтобы установить все указанные в `pom.xml` зависимости запустите команды **clean** и **install** в панели Maven:

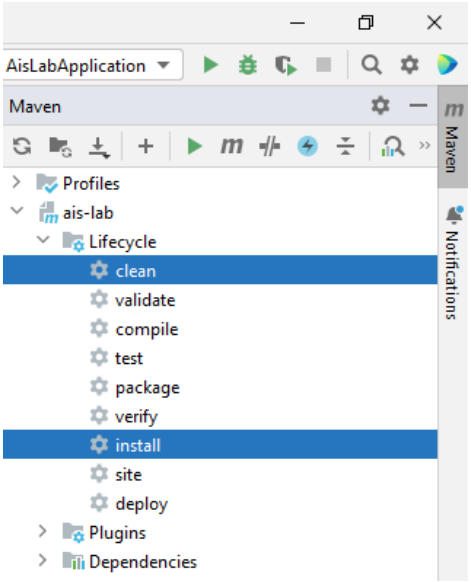


Рис. 3.9 – Панель Maven в IDEA IDE

Или введите команду в терминале IDE:

```
mvn clean install
```

Рассмотрим структуру данного приложения из примера (рис. 3.10):

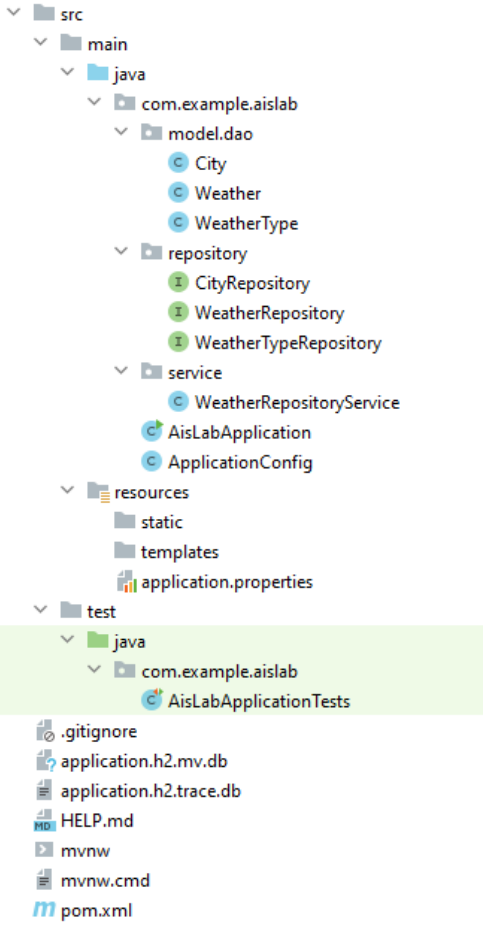


Рис. 3.10 – Структура Spring-приложения

`AisLabApplication.java` – main-класс приложения. Аннотация `@SpringBootApplication` инициализирует приложение как SpringBoot-приложение

`ApplicationConfig.java` – класс конфигурации приложения. Может использоваться для инициализации необходимых бинов (beans) приложения, а также для инициализации дополнительных параметров файла конфигурации application.properties. Для создания класса-конфигурации в Spring используется соответствующая аннотация `@Configuration`.

`model.dao` – пакет, содержащий классы, которые представляют собой объектно-реляционное отображение структуры БД. Таким образом, экземпляры данных классов (***Data Access Object*** или сокр. **DAO**) будут содержать данные соответствующих таблиц БД. Данные классы определяются аннотацией `@Entity`.

`repository` – пакет с реализацией интерфейсов для работы с соответствующими DAO. Данный интерфейс наследуют обобщенные Spring-интерфейсы `JpaRepository` и `CrudRepository`, в которых уже реализованы основные операции для работы с DAO: `findById`, `findAll`, `deleteById`, `getOne` и т.д.

`service` – пакет содержит классы, где реализована бизнес-логика приложения. Все подобные классы в Spring определяются аннотацией `@Service`. Класс **`WeatherRepositoryService.java`**, соответственно, реализует программный слой работы с хранилищем данных. Содержит функции (бизнес-логику) для работы с БД.

`test` – пакет с реализацией тестов Spring приложения (использует JUnit 5 в качестве промежуточного ПО для тестирования).

`application.properties` – файл конфигурации приложения Spring по умолчанию (также может использоваться файл application.yml). Параметры конфигурации с префиксом **spring.*** используются для установки параметров Spring-приложения (доступные параметры Spring можно посмотреть здесь: https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html).

Настройки подключения к различным БД устанавливаются с помощью параметров с префиксом **spring.datasource.***

Пользовательские параметры в файле конфигурации необходимо инициализировать в коде приложения с помощью аннотаций `@Value` или `@ConfigurationProperties`.

`application.h2.mv.db` – файл встраиваемой БД H2. Данная БД реализована на Java и имеет тесную интеграцию с Spring.

ВАЖНО! Код, представленный в примере, подробно комментирован. Более детально с SpringBoot Framework можно ознакомиться здесь:

https://docs.spring.io/spring-boot/docs/current/reference/html/

3.3.1 Популяция H2 БД. Для работы с БД H2 добавьте соответствующую зависимость в **pom.xml** и обновите pom-конфигурацию:

```
    <dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

Все последние версии библиотек можно найти в репозитории Maven по адресу:

https://mvnrepository.com/

Если в **application.properties** установлен параметр:

```
spring.jpa.hibernate.ddl-auto = update
```

Структура таблиц будет сгенерирована автоматически в соответствии с моделями DAO, также автоматически будет создан файл БД H2, указанный в параметре:

```
spring.datasource.url = jdbc:h2:file:./application.h2
```

`jdbc:h2:file` – БД будет создана в виде файла.

`jdbc:h2:mem` – БД будет создана только в оперативной памяти.
`./имя_бд.h2` – файл БД будет создан в корне проекта.

Параметры конфигурации:

```
spring.h2.console.enabled = true
spring.h2.console.path = /h2
```

активируют встроенную панель управления БД H2. Для запуска панели, запустите приложение и перейдите по адресу http://127.0.0.1:8080/h2 . Укажите соответствующие настройки подключения (из application.properties) и нажмите «Connect». С помощью данной панели заполните необходимые для запуска тестов табличные данные.

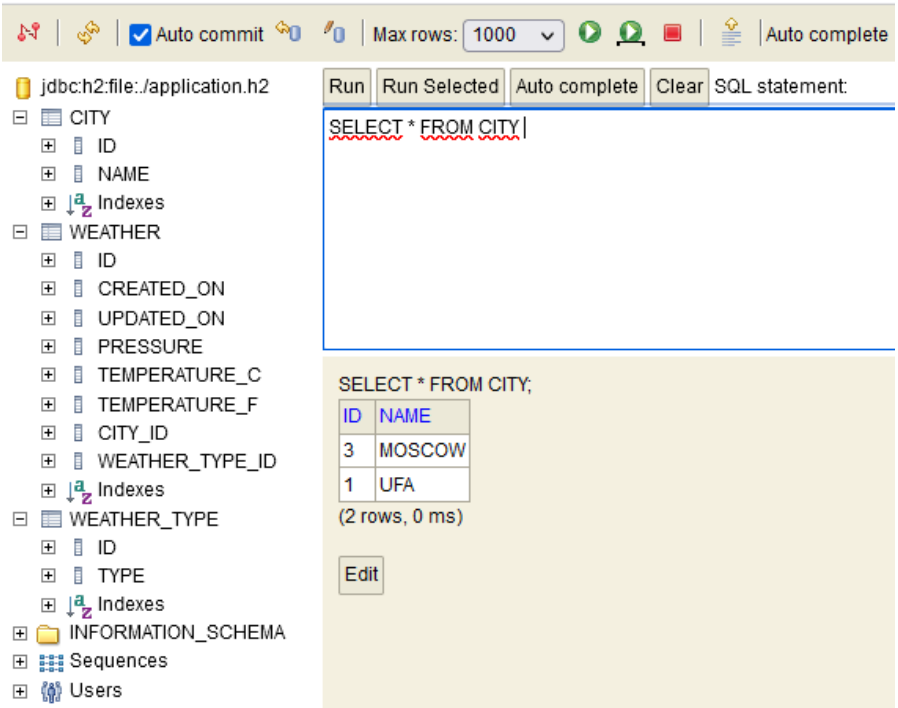


Рис. 3.11 – Панель администратора H2 DB

3.3.2 Тестирование CRUD-функций. Для запуска тестов из пакета test, запустите команду «test» из панели Maven или введите команду в терминале IDE:

```
mvn test
```

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.600 s
[INFO] Finished at: 2022-09-18T21:42:43+05:00
[INFO] ------------------------------------------------------------------------
```

Рис. 3.12 – Тестирование приложения с помощью Django

Результатом выполнения должен быть вывод: «**BUILD SUCCESS**»

3.3.3 Тестирование CRUD-функций (MariaDB). Для работы с MariaDB добавьте зависимость в pom.xml и обновите конфигурацию Maven:

```
    <dependency>
   <groupId>org.mariadb.jdbc</groupId>
   <artifactId>mariadb-java-client</artifactId>
   <version>3.0.7</version>
</dependency>
```

Установите в конфигурационном файле `application.properties` подключение к БД MariaDB:

```
spring.datasource.url = jdbc:mariadb://192.168.56.104:3306/weatherdb_spring
spring.datasource.username = usr
spring.datasource.password = qwerty
spring.datasource.driver-class-name = org.mariadb.jdbc.Driver
spring.jpa.hibernate.ddl-auto = update
```

Запустите ВМ с установленным сервером БД. Создайте БД на сервере и выполните настройку сетевого доступа согласно п.4 методических рекомендаций.

Добавьте в БД необходимые тестовые данные.

Находясь в среде разработки запустите команду «test» из панели Maven, или запустите команду в терминале:

```
mvn test
```

Результатом запуска должен быть вывод: «**BUILD SUCCESS**»

4. Организация удаленного подключения к серверу БД

4.1 Настройка сервера БД. Настройка производится в 2 этапа:

1) Изменение сетевых настроек сервера БД. Редактируем файл конфигурации **`/etc/mysql/mariadb.conf.d/50-server.cnf`**

В файле параметры port и bind (или bind-address) должны быть приведены к следующему виду:

`port = 3306` # основной порт для MySQL

`bind-address = 0.0.0.0` # сервер БД будет принимать подключения на всех сетевых интерфейсах

Перезапускаем сервер БД:

```
systemctl restart mysqld
```

2) Создаем БД для приложения:

```
CREATE DATABASE weatherdb DEFAULT CHARACTER SET utf8;
```

3) Создаем пользователя с разрешенным удаленным доступом к заданной базе из определенной локальной сети:

```
GRANT ALL ON weatherdb.* TO usr@'%' IDENTIFIED BY 'qwerty';
```

Конструкция `usr@'%'` означает, что пользователю usr будет разрешён доступ к данной БД из всех сетей, к которым подключен сервер. Для настройки доступа к БД только с определенных IP-адресов, например, из пула 192.168.55.X, выражение будет выглядеть следующим образом: `usr@'192.168.55.%'`

ВАЖНО! Настройка удаленного доступа к БД без привязки к конкретным разрешённым IP-адресам позволяет удобно работать с данными, но СУЩЕСТВЕННО снижает безопасность сервера, даже если сервер работает в локальной сети и использует для доступа в интернет сетевой шлюз или роутер. Такими образом, при развёртывании приложения на продуктивном сервере доступ к БД должен быть максимально ограничен, и если приложение и БД физически находятся на одном сервере, то доступ ограничивается локальным интерфейсом, т.е. разрешён только с адреса 127.0.0.1.

4.1 Работа с сервером БД. В дальнейшем для работы с сервером БД можно использовать графический клиент **DBeaver**. Для этого запустите приложение и выберите в выпадающем меню создание нового соединения к соответствующему серверу БД и в появившемся окне введите адрес и учетные данные для подключения:

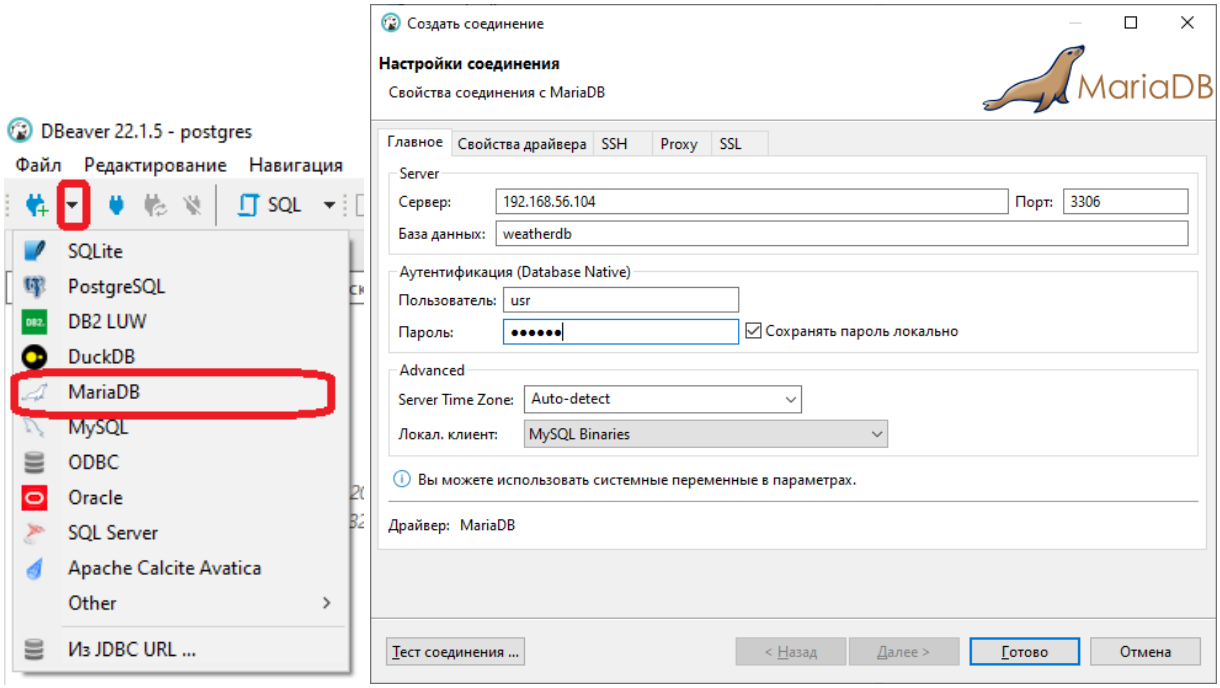


Рис. 4.1 – Создание нового соединения в DBeaver

Подключение к БД появится в списке подключений в левом окне. Щелкните на подключение, приложение автоматически предложит скачать соответсвующий драйвер (если он не установлен в системе):

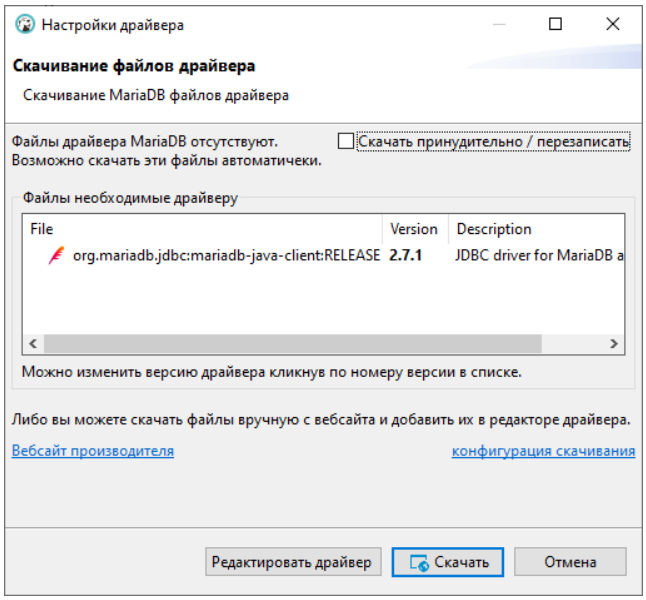


Рис. 4.2 – Установка драйвера подключения к MariaDB для DBeaver

Для просмотра и редактирования таблиц БД откройте необходимую таблицу, щелкнув на ней правой кнопкой мыши и выбрав опцию «ViewData»:

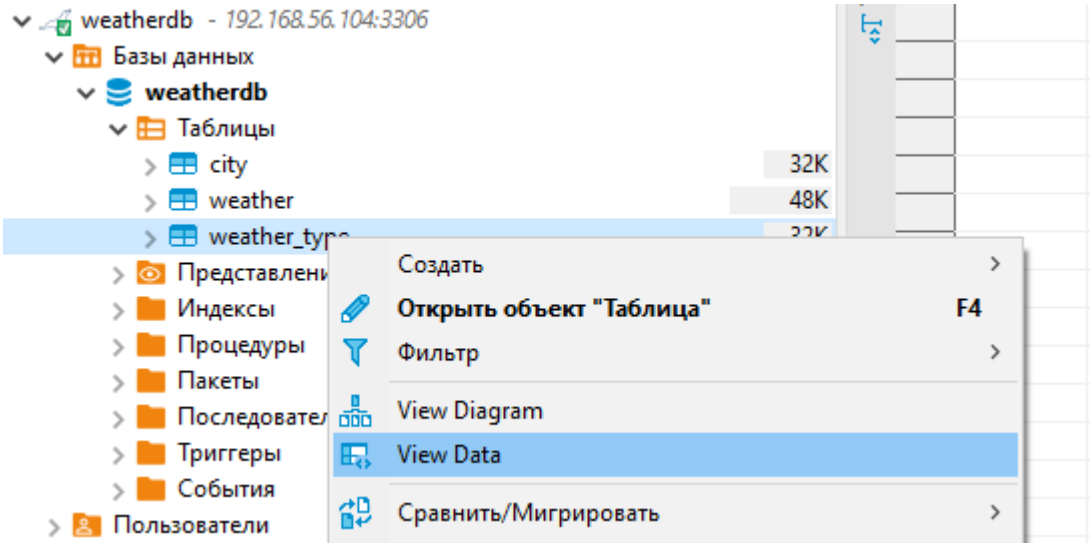


Рис. 4.3 –Вывод и редактирование таблиц в DBeaver

Для выполнения SQL-запросов воспользуйтесь редактором SQL, вызвав его из панели инструментов:

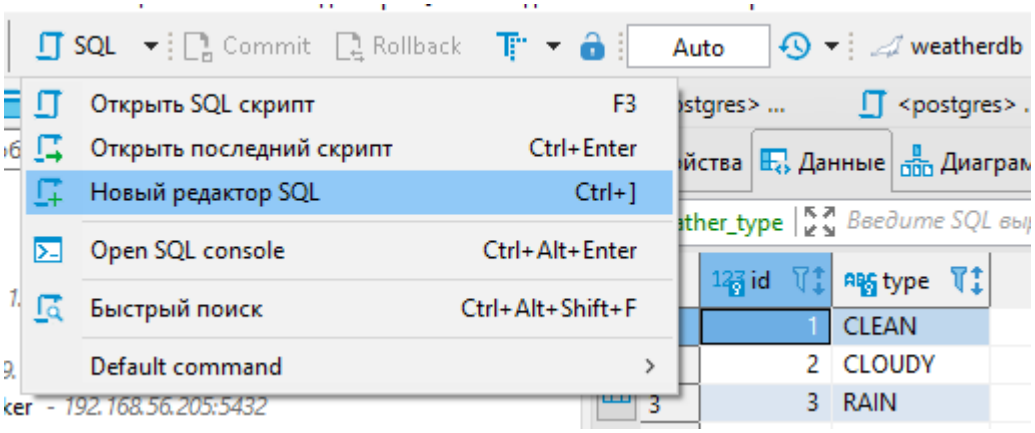


Рис. 4.4 – Вызов редактора SQL в DBeaver

Часть 2

Разработка RESTful веб-сервиса

Цель работы: разработать RESTful веб-приложение на основе созданной ранее ORM модели базы данных.

Задание:

1) Ознакомиться с протоколом HTTP (вспомнить материал с курса «Архитектура вычислительных систем и компьютерные сети»). Особенности реализации данного протокола, коды возвращаемых ошибок.

2) Ознакомиться с понятием REST API (что это, методы, понятие URI, клиент-серверное взаимодействие).

3) Спроектировать REST API для серверной части вашей информационной системы.

4) Реализовать RESTful веб-приложение и оформить страницу с документацией к вашему REST API на основе спецификации OpenAPI.

5) Протестировать RESTful приложение и продемонстрировать результат.

Результатом выполнения задания являются RESTful веб-приложение и отчет, содержащий следующую информацию:

1) Конспект необходимого теоретического материала для выполнения лабораторной работы.

2) Пошаговое описание практической части.

3) Отчет по лабораторной работе, содержащий ход выполнения работы с описанием и скриншотами выполнения (основные моменты), таблицу с описанием REST API приложения.

4) Код приложения, сохраненный в системе контроля версий.

Для успешной защиты лабораторной работы студенты должны предоставить проект (демонстрацию проекта) и отчет к нему.

Требования к оформлению отчета:

Способ выполнения текста должен быть единым для всей работы. **Шрифт** – **Times New Roman**, кегль 14, **межстрочный интервал** – 1,5, **размеры полей**: левое – 30 мм; правое – 10 мм, верхнее – 20 мм; нижнее – 20 мм. Сокращения слов в тексте допускаются только общепринятые.

Абзацный отступ (1,25) должен быть одинаковым во всей работе. **Нумерация страниц** основного текста должна быть сквозной. Номер страницы на титульном листе не указывается. Сам номер располагается внизу по центру страницы или справа.

Методические рекомендации к выполнению лабораторной работы

1. Понятие REST API

REST (Representational State Transfer – дословно «передача состояния представления») – это модель взаимодействия клиент-серверного приложения в сети по протоколу HTTP.

API (Application Programming Interface – программный интерфейс приложения) – описание классов, процедур, функций и методов взаимодействия между приложениями. Проще говоря, это «язык общения» между приложениями.

Данная модель взаимодействия позволяет осуществлять вызов удаленных процедур (методов) web-приложения, используя структурированные (унифицированные) адреса, которые обозначаются аббревиатурой **URI (Uniform Resource Identifier)**.

Web-приложение, использующее для предоставления своих ресурсов REST-модель взаимодействия, называется **RESTful веб-сервис**. Такое приложение использует специальные методы протокола HTTP и соответствующие структурированные веб-адреса.

В табл. 1 представлены основные методы HTTP для взаимодействия с RESTful веб-сервисом и примеры URI для предоставления ресурсов приложения.

Табл. 1 – Основные методы HTTP для взаимодействия с RESTful веб-сервисом

Метод HTTP	Действие	Пример URI
GET	Получить информацию о ресурсе	example.com/api/orders (получить список заказов)
GET	Получить информацию о ресурсе	example.com/api/orders/123 (получить заказ №123)
POST	Создать новый ресурс	example.com/api/orders (создать новый заказ из данных переданных с запросом)
PUT	Обновить ресурс	example.com/api/orders/123 (обновить заказ №123 данными переданными с запросом)
DELETE	Удалить ресурс	example.com/api/orders/123 (удалить заказ #123)

Модель REST API описывается документом на основе спецификации **OpenAPI**. В общем случае данная спецификация не зависит от языка программирования, и может быть использована вне протокола HTTP. Также спецификация позволяет использовать различные форматы данных, передаваемых в теле запроса: JSON, XML, YAML. Для автоматического генерирования OpenAPI-документа в современных фреймворках используется **Swagger.** Swagger предоставляет инструментарий для описания и работы с REST API. Swagger предлагает два основных подхода к генерированию документации:

- Автогенерация на основе кода.

- Самостоятельная разметка.

В рамках первого подхода достаточно добавить необходимые зависимости (библиотеки) в проект, и Swagger сам сгенерирует схему документа на основе вашего кода. Также необходимо добавить в код поясняющие аннотации и комментарии для API-методов, которые автоматически будут отображены на странице Swagger.

Второй подход предполагает знание синтаксиса Swagger. В данном случае схема документа реализуется вручную в формате YAML или JSON (можно использовать специальный редактор **Swagger Editor**). Данный подход позволяет сделать документацию более качественной и предусмотреть ряд нестандартных решений в рамках конкретного проекта.

2. Проектирование и разработка RESTful веб-сервиса

2.1 Проектирование REST API. При проектировании RESTful веб-сервиса в первую очередь нужно определить ресурсы, которые будут доступны и запросы, с помощью которых можно будет работать с данными ресурсами. **Основные ресурсы и методы работы с ними (в рамках базы данных) были разработаны в процессе выполнения лабораторной работы № 2. Таким образом, на данном этапе необходимо определить URI и HTTP-методы, которые будут вызывать разработанные ранее CRUD операции для БД** (и, если необходимо, добавляем дополнительные операции).

В общем случае структуру URI или URL можно представить следующим образом:

```
http://{hostname}/api/v1.0/{servicename}/{resource}
```

hostname – доменное имя или адрес сервера;

servicename – название веб-сервиса (должно быть осмысленным);

api/v1.0 – необязательная часть URI, указывающая версию вашего API или формат передаваемых данных (необходимо, если сервис поддерживает различные форматы данных и различные версии API, например, для совместимости со старыми версиями клиентского приложения);

resource – ресурс, который запрашивается для получения данных или изменения.

Страница Swagger с описанием REST API может быть представлена по следующим адресам:

```
http://{hostname}/api
http://{hostname}/docs
http://{hostname}/swagger
http://{hostname}/swagger-ui
```

Страница Swagger с описанием REST API приложения из примеров с кодом:

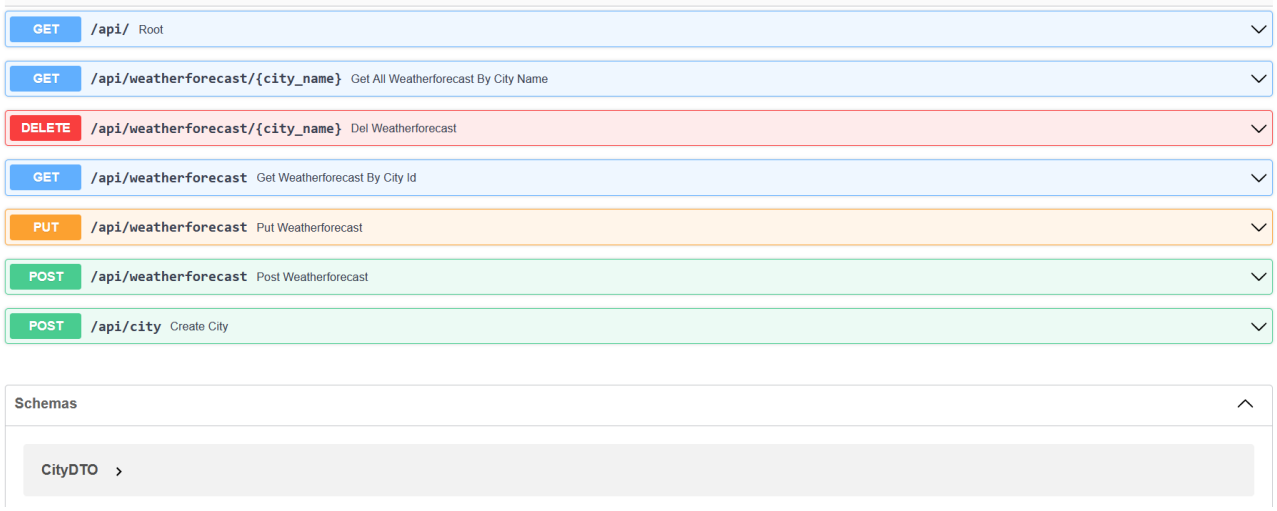


Рис. 2 – Пример страницы Swagger

2.2 Разработка RESTful веб-сервиса. Классическая архитектура RESTful веб-сервиса представлена на рисунке 2:

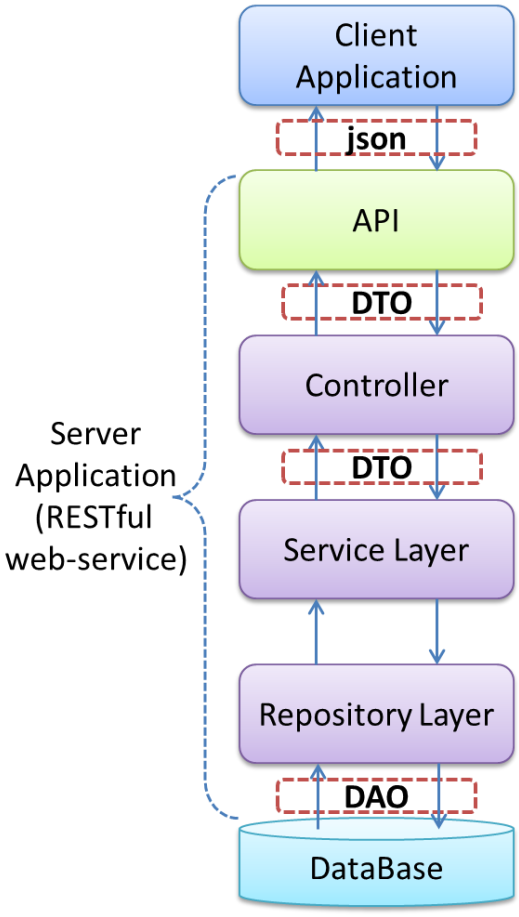


Рис. 2 – Классическая архитектура RESTful веб-сервиса

Как видно по рисунку, некоторые программные слои RESTful приложения уже были реализованы в рамках лабораторной работы №2 (Repository Layer, DAO), рассмотрим остальные элементы:

API – данный уровень реализует обработку соответствующих HTTP-методов, а также осуществляет преобразование поступающих от клиента данных (например, запросов в формате json) в специальные объекты для передачи данных (**Data Transfer Object**, сокр. **DTO**).

DTO – объекты-контейнеры, которые в своих атрибутах содержат значения или данные, необходимые для выполнения бизнес-логики в на других уровнях приложения. Также DTO могут выполнять валидацию (проверку) данных (например, может проверяться корректности типа значения или обязательное наличие соответствующего значения в данных). Обычно DTO в коде представлены в виде моделей в соответствующей папке проекта **`models.dto`**.

Controller – на данном уровне происходит маршрутизация данных в соответствующие методы для дальнейшей обработки, а также формирование ответа для клиента. В разных фреймворках и языках программирования реализация данного уровня приложения может содержаться в файлах с различным названием, например, в **Python Django** реализация обычно находится в файлах и пакетах с ключевым словом views, в **Python Flask** и **FastAPI** это routes, в Java Spring файлы должны содержать ключевое слово Controller, а классы должны помечаться аннотацией @Controller.

ServiceLayer – промежуточный слой (не обязательный, но в больших проектах требуемый), реализующий дополнительную бизнес-логику в приложении.

2.2.1 Реализация RESTful веб-сервиса (Python FastAPI Framework). Устанавливаем необходимые зависимости:

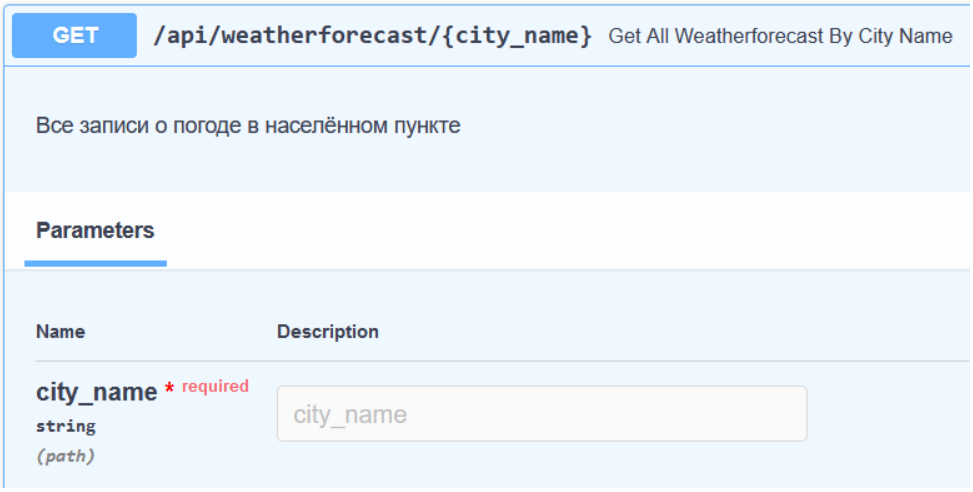
```
pip3 install fastapi
```

Пример кода веб-сервиса с комментариями представлен в файле **`ais3_fastapi.zip`**

Фреймворк FastAPI по умолчанию предоставляет инструменты Swagger. По умолчанию страница Swagger будет представлена по адресу **/docs**, тем не менее, необходимо добавить описание к вашим методам, которые обрабатывают соответствующие HTTP-методы (GET, POST и т.д.). Это можно сделать, добавив обычные комментарии в коде, например, метод:

```
@router.get('/weatherforecast/{city_name}', response_model=List[WeatherDTO])
async def get_all_weatherforecast_by_city_name(city_name: str):
    """ Все записи о погоде в населённом пункте """
    return service.get_all_weather_in_city(city_name)
```

Отображение на странице Swagger:

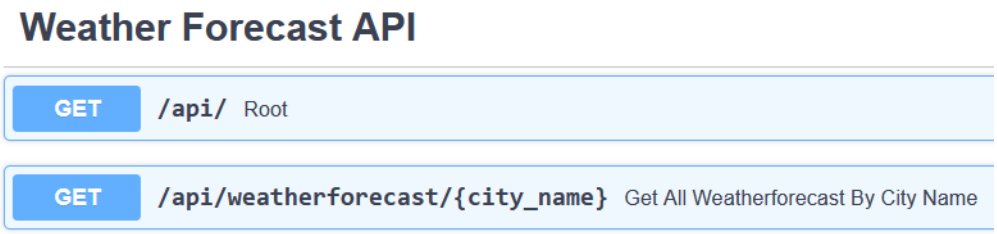


Модель входных данных Swagger определит по аргументам соответствующих методов, т.о. для всех функций роутера должны быть указаны входные типы данных (например, `str`, если принимается простой строковый аргумент или соответствующий класс модели данных – `WeatherDTO`).

Модель выходных данных Swagger определяет по параметру `response_model` в декораторе метода, если метод возвращает только код HTTP, то в декораторе указывается параметр `status_code`.

Чтобы определить заголовок для страницы Swagger заполните параметр tags в соответствующем объекте router (см. **`routes.py`** в примере):

```
router = APIRouter(prefix='/api', tags=['Weather Forecast API'])
```



Более подробно с возможностями FastAPI можно ознакомиться здесь: https://fastapi.tiangolo.com/ru/features/

2.2.2 Реализация RESTful веб-сервиса (Python Django REST Framework, сокр. DRF). Устанавливаем необходимые зависимости:

```
pip3 install djangorestframework
```

Пример кода веб-сервиса с комментариями представлен в файле **`ais3_django.zip`**

Для реализации страницы Swagger в DRF необходима установка дополнительного пакета:

```
pip install -U drf-yasg
```

Пример подключаем приложение в settings.py:

```
INSTALLED_APPS = [
    'rest_framework',
    'drf_yasg',
    …]
```

Также убедитесь, что в INSTALLED_APPS подключен модуль django.contrib.staticfiles

Добавляем в локальном модуле urls.py приложения (например, weatherapp/urls.py) метаданные Swagger и пути к странице:

```
      schema_view = get_schema_view(
   openapi.Info(
      title=" Weather Forecast API",
      default_version='v1',
      description="Weather Forecast API",
      terms_of_service="https://example.com",
      contact=openapi.Contact(email="contact@mail.local"),
      license=openapi.License(name="BSD License"),
   ),
   public=True,
   permission_classes=[permissions.AllowAny],
)


      urlpatterns = [
    …
    re_path(r'^swagger(?P<format>\.json|\.yaml)$',
schema_view.without_ui(cache_timeout=0), name='schema-json'),
    re_path(r'^swagger/$', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui')
]
```

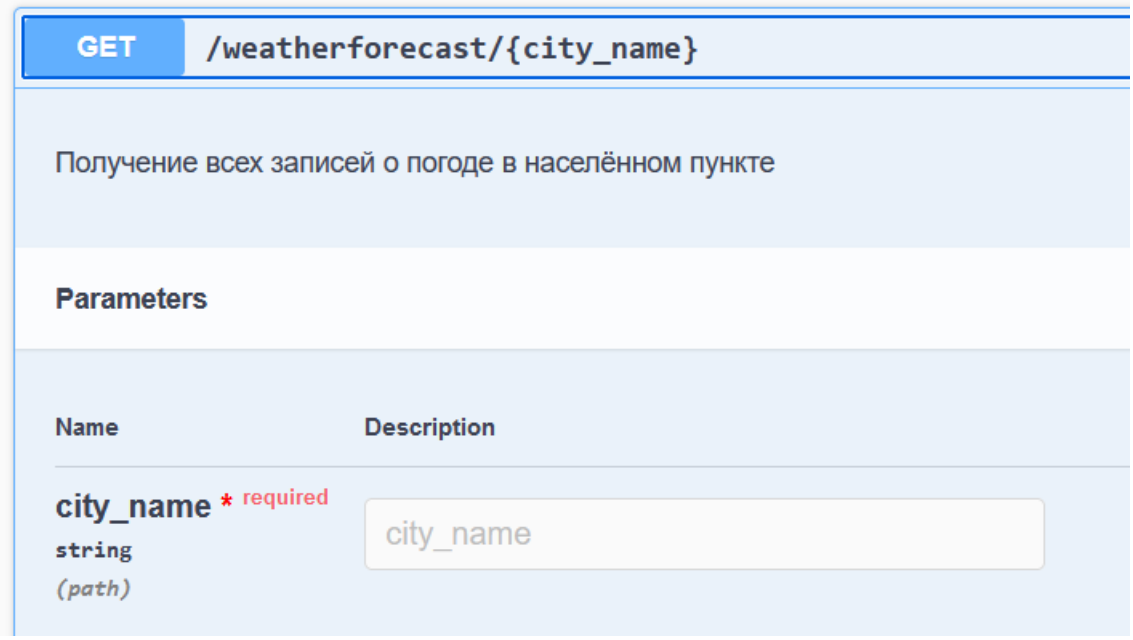
Таким образом, Swagger будет доступен по адресу:
<корневой_url_вашего_приложения**>/swagger**

Чтобы добавить описание для ваших методов (API endpoints) в Swagger необходимо добавить комментарии для соответствующих методов в коде, например:

```
      class GetDelAllWeather(GenericAPIView):
    serializer_class = WeatherSerializer     # определяем сериализатор (необходимо для
генерирования страницы Swagger)
    renderer_classes = [JSONRenderer]        # определяем тип входных данных

    def get(self, request: Request, city_name: str) -> Response:
        """ Получение всех записей о погоде в населённом пункте """
        response = service.get_all_weather_in_city(city_name)
        return Response(data=response.data)
```

Отображение на странице Swagger:



Ознакомиться подробнее с информацией по работе с Swagger в Django можно здесь: https://github.com/axnsan12/drf-yasg

2.2.3 Реализация RESTful веб-сервиса (Java Spring). Добавьте в pom.xml необходимые зависимости:

```
    <!-- Зависимость для валидации атрибутов классов (@NotNull, @NotEmpty) с помощью аннотации @Valid -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-validation</artifactId>
   <version>2.7.3</version>
</dependency>
<!-- Подключение зависимости для работы со Swagger -->
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.6.9</version>
</dependency>
```

Пример кода веб-сервиса с комментариями представлен в файле **ais3_spring.zip**

После подключения зависимостей страница Swagger по умолчанию будет доступна по адресу:

<корневой_url_вашего_приложения**>/swagger-ui/index.html**

С помощью параметра в `application.properties` можно настроить переадресацию на страницу Swagger с желаемого адреса:

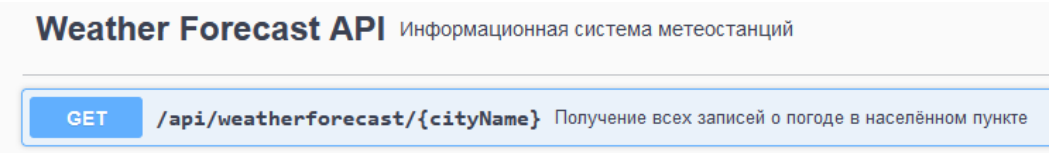
springdoc.swagger-ui.path = /docs

Чтобы добавить описание для ваших методов (API endpoints) в Swagger необходимо добавить специальные spring-аннотации (@Tag и @Operation) для соответствующих методов в коде, например:

```
@Tag(name = "Weather Forecast API", description = "Информационная система
метеостанций")
@RestController
@RequestMapping(value = "/api")
public class WeatherController {
    @Autowired
    private WeatherService service;      // подключаем слой с бизнес-логикой

    @Operation(summary = "Получение всех записей о погоде в населённом пункте")
    @GetMapping(value = "/weatherforecast/{cityName}")
    public List<WeatherDto> getAllWeatherForecastByCityName(@PathVariable("cityName")
String cityName) {
        return service.getWeatherInfoInCity(cityName);
    }
```

Отображение на странице Swagger:



3. Тестирование REST API

Для тестирования вашего RESTful сервиса можно воспользоваться готовыми инструментами для работы с REST API, например: **SoapUI** (https://www.soapui.org/) или **Insomnia** (https://insomnia.rest/).

Часть 3-4

Нагрузочное тестирование веб-приложения

Разработка графического интерфейса клиентского приложения.

Цель работы: провести нагрузочное тестирование разработанного RESTful веб-приложения и спроектировать графический интерфейс (GUI) клиентского приложения.

Задание:

1) Ознакомиться с теоретическим материалом по нагрузочному тестированию: понятие нагрузочного тестирования, виды тестов, популярный инструментарий.

2) Провести нагрузочное тестирование вашего веб-приложения, используя 3 подхода: нагрузочный, объемный и стрессовый. Процедура выполнения тестов и фиксирования параметров описана в **п. 1.4 Фиксирование результатов тестирования для отчета.**

3) Ознакомиться с теоретическим материалом по проектированию графического интерфейса (Приложение1).

4) Разработать графический интерфейс (GUI) клиентского приложения в соответствии с основными принципами проектирования интерфейса. Интерфейс должен быть достаточен для реализации функциональных возможностей

5) Для интерфейса подготовьте текстовое описание полей и графических элементов в следующем виде:

Таблица 1 – Текстовое описание полей GUI

Название поля	Тип	Условия видимости	Условия доступности	Описание
				Формат, допустимые значения, макс. и мин. длина, поведение

6) Отобразить в отчете пошаговое выполнение разработки интерфейса, а также пример использования каждого из 6 основных принципов проектирования интерфейса.

Результатом выполнения задания являются программа для тестирования с набором тестовых данных и отчет, содержащий следующую информацию:

Условие проведения нагрузочных тестов:

- технические характеристики тестового стенда (виртуальной машины или ЭВМ, на которой было запущено тестируемое приложение);
- описание тестовых сценариев (т.е. какие REST-методы вызывались);
- максимальное количество эмулируемых пользователей и скорость их увеличения;

- максимальное количество записей в БД при проведении тестирования.

Результаты нагрузочного тестирования (скриншоты Locust) с анализом результатов по каждому из проведенных тестов: при каком количестве пользователей

Пошаговое описание практической части.

Скриншот главного окна клиентского приложения с описанием полей ввода (Таблица 1) и активных элементов интерфейса (поля вывода, кнопки, вкладки и т.п.).

Для успешной защиты лабораторной работы студенты должны предоставить проект (демонстрацию проекта) и отчет к нему.

Требования к оформлению отчета:

Способ выполнения текста должен быть единым для всей работы. **Шрифт** – **Times New Roman**, кегль 14, **межстрочный интервал** – 1,5, **размеры полей**: левое –30 мм; правое – 10 мм, верхнее – 20 мм; нижнее – 20 мм. Сокращения слов в тексте допускаются только общепринятые.

Абзацный отступ (1,25) должен быть одинаковым во всей работе. **Нумерация страниц** основного текста должна быть сквозной. Номер страницы на титульном листе не указывается. Сам номер располагается внизу по центру страницы или справа.

Методические рекомендации к выполнению лабораторной работы (Часть 3)

1. Нагрузочное тестирование

1.1 Тестирование производительности или нагрузочное тестирование – процесс тестирования с целью определения производительности программного продукта.

Виды тестов:

- нагрузочное тестирование (Performance and Load testing) – вид тестирования производительности, проводимый с целью оценки поведения компонента или системы при возрастающей нагрузке, например количестве параллельных пользователей и/или операций, а также определения какую нагрузку может выдержать компонент или система;
- объемное тестирование (Volume testing) – позволяет получить оценку производительности при увеличении объемов данных в базе данных приложения;
- тестирование стабильности и надежности (Stability / Reliability testing) – позволяет проверять работоспособность приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
- стрессовое тестирование (Stress testing) – вид тестирования производительности, оценивающий систему или компонент на граничных значениях рабочих нагрузок или за их пределами, или же в состоянии ограниченных ресурсов, таких как память или доступ к серверу.

Для тестирования производительности существует множество утилит и фреймворков, среди наиболее популярных можно отметить: Locust, Apache JMeter, Taurus, Gatling, Яндекс.Танк. Каждый из данных инструментов успешно решает задачу нагрузочного тестирования, тем не менее, существует особенности в настройке, например: необходимость создания сценариев тестирования (необходимо знание определенного языка программирования), наличие или отсутствие графического интерфейса, возможности распределенного тестирования.

1.2 Подготовка базы данных. При нагрузочном тестировании производятся РЕАЛЬНЫЕ запросы к вашей БД, таким образом, часть необходимых записей в БД может быть удалена или, при генерировании большого количества POST-запросов, БД может быть «засорена» большим количеством тестовых данных.

ВАЖНО! Для предотвращения потери рабочей версии БД создайте ее РЕЗЕРВНУЮ КОПИЮ (DUMP)! Это можно сделать с помощью клиента для работы с БД. Например, в DBeaver (см. ЛР №1) процедура будет выглядеть следующим образом (рис. 1-3):

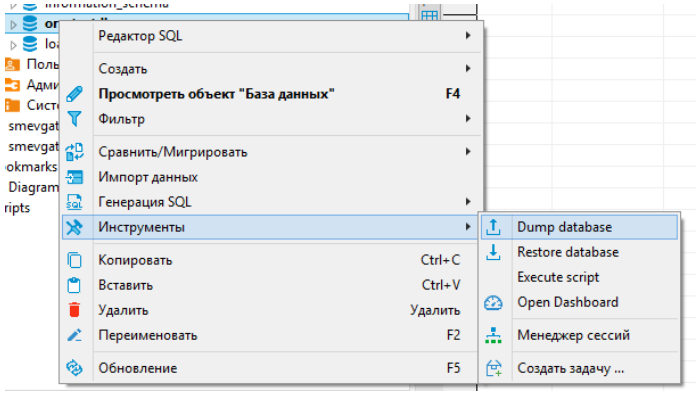


Рис. 1 – Вызов процедуры создания дампа нажатием правой клавиши мыши на сохраняемой БД

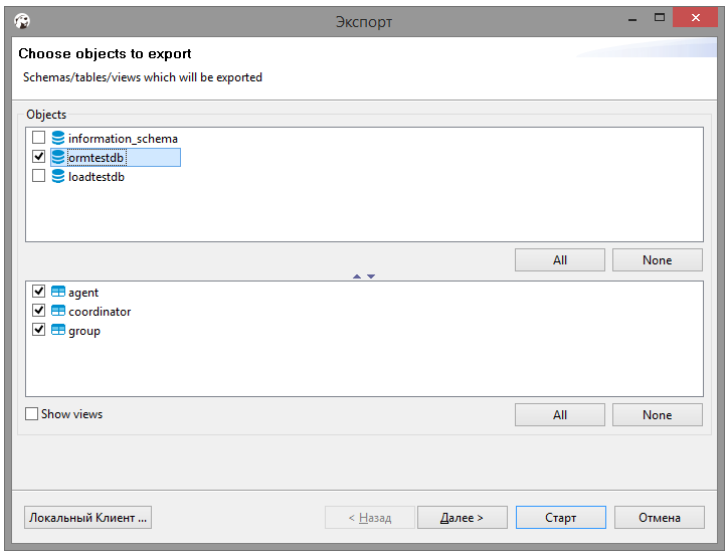


Рис. 2 – Окно настроек экспорта (выбор сохраняемых таблиц)

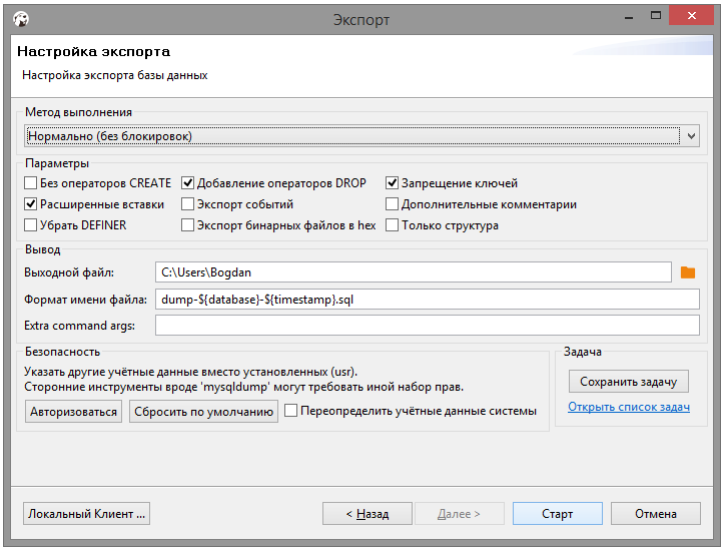


Рис. 3 – Окно дополнительных настроек экспорта

После определения параметров экспорта нажимаем кнопку «**Старт**».

Для восстановления копии вашей БД из сохраненного дампа необходимо создать новую «пустую» БД и дать права текущему пользователю БД с помощью следующих команд в консоли MySQL / MariaDB:

```
CREATE DATABASE <имя_новой_БД> DEFAULT CHARACTER SET utf8;
GRANT ALL ON <имя_новой_БД>.* TO <ваш_пользователь>@'%' IDENTIFIED BY 'ваш_пароль';
FLUSH PRIVILEGES;
```

Далее задаем имя новой БД и запускаем процедуру восстановления (Restore) для созданной БД в клиенте DBeaver (рис. 4):

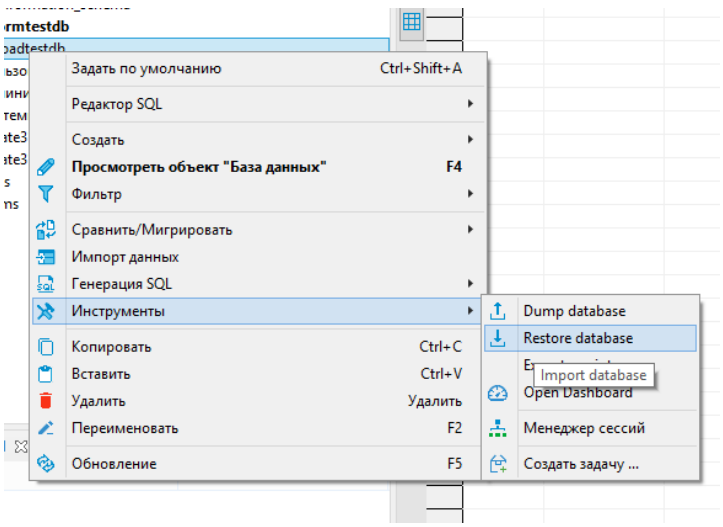


Рис. 4 – Вызов процедуры восстановления

В меню импорта выбираем сохраненный дамп и запускаем процедуру восстановления кнопкой «Старт» (рис. 5):

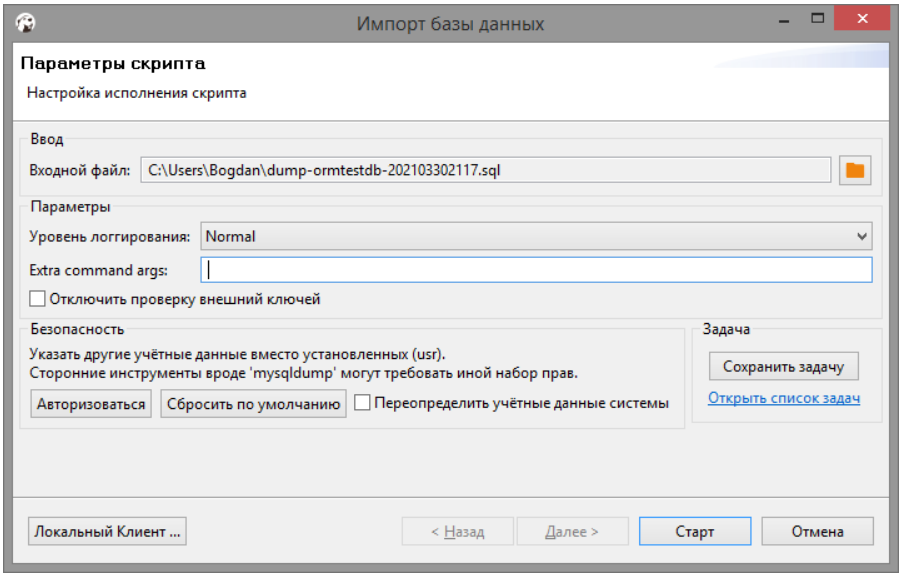


Рис. 5 – Запуск процедуры восстановления

1.3 Подготовка тестового стенда. Тестовый стенд – сервер, на котором развёртывается тестируемое приложение. В рамках лабораторной работы качестве тестового стенда может быть использована виртуальная машина с установленной ранее БД и CMS. Помимо непосредственно приложения на стенде также потребуется установка утилит для предоставления системной информации.

```
apt install atop htop
```

htop – утилита предоставляет удобный вывод информации о загрузке памяти (Mem) и процессора (CPU).

atop – утилита предоставляет полную информацию о состоянии системы, в том числе нагрузку на жесткий диск и сетевые интерфейсы.

1.3.1 Развёртывание Python приложения. Как правило, популярные дистрибутивы Linux по умолчанию имеют предустановленную среду Python. Для проверки наличия интерпретатора Python в системе введите команду «**python3**» в терминале, вывод должен быть следующим:

```
root@aislab22:/home/ais# python3
Python 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Если интерпретатор не найден, устанавливаем с помощью системного менеджера пакетов:

```
apt install python3 python3-pip python3-dev python3-venv
```

Добавьте в проекте в список необходимых пакетов (**requirements.txt**) следующие пакеты:

uvicorn – Веб-сервер для запуска асинхронных (**ASGI**) Python приложений. Добавляем для FastAPI приложения.

gunicorn – Веб-сервер для запуска синхронных (**WSGI**) Python приложений. Добавляем для Django приложения.

Основные зависимости в файле **requirements.txt:**

Для FastAPI:

```
SQLAlchemy
pymysql
fastapi
uvicorn
```

Для Django:

```
django
djangorestframework
mysqlclient
drf-yasg
gunicorn
```

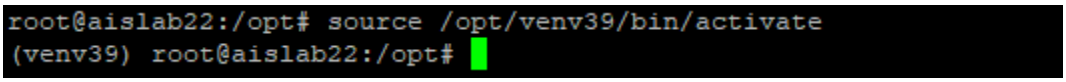
Копируем на тестовый стенд необходимые для запуска файлы вашего Python приложения и requirements.txt в отдельную папку. Папку с приложением переносим в системную директорию, которую обычно используют для сторонних приложений: **/opt**

Развёртываем виртуальную среду Python в директории **/opt/venv39** с помощью команды:

```
python3 -m venv /opt/venv39
```

Для активации созданной виртуальной среды выполняем команду:

```
source /opt/venv39/bin/activate
```



(Для деактивации среды достаточно ввести в терминале команду: **deactivate**)

Устанавливаем необходимые зависимости в активированную виртуальную среду, например:

```
pip install -r /opt/lab-app-python-fastapi/requirements.txt
```

или

```
pip install -r /opt/lab-app-python-django/requirements.txt
```

ВАЖНО! Если при установке mysqlclient для Django возникает ошибка, установите дополнительные системные зависимости командой:

```
apt install python3-dev default-libmysqlclient-dev build-essential
```

и повторно запустите установку пакетов python.

ВАЖНО! Перед запуском приложения не забудьте убедиться, что в конфигурации выставлены настройки для подключения к серверу **MySQL / MariaDB**, а не к локальной БД.

Веб-серверы приложений Python (gunicorn и uvicorn) для распараллеливания нагрузки используют несколько рабочих процессов (workers), которые определяются параметром **`--workers или -w .`** Количество workers обычно задается по формуле **2*CORE + 1**, где CORE – это количество ядер CPU, которое планируется задействовать для работы веб-приложения.

ВАЖНО! При запуске приложения в терминале Linux терминал будет заблокирован, и весь вывод приложения будет также выводиться в консоль, чтобы запустить приложение в фоновом режиме и перенаправить вывод в текстовый файл (например, для анализа возможных ошибок) используйте следующую конструкцию:

```
команда_запуска_приложения >> output.log &
```

Здесь >> **output.log** означает перенаправление вывода в файл **output.log**, символ **&** - запуск в фоновом режиме.

Чтобы остановить запущенный в фоне процесс используйте команду:

```
pkill -f имя_процесса
```

Например:

```
pkill -f uvicorn
```

Переходим в директорию приложения, активируем виртуальную среду и запускаем, используя следующие команды:

для FastAPI:

uvicorn в качестве первого аргумента необходимо передать имя **py**-файла, который запускает приложение, а также имя объекта приложения. Таким образом, команда запуска для приложения из примера будет следующая:

```
uvicorn main:app --host 0.0.0.0 --port 8000 --workers 3 --
log-level debug >> debug.log &
```

```
(venv39) root@aislab22:/opt/lab-app-python-fastapi# uvicorn main:app --host 0.0.0.0
--port 8000 --workers 2 --log-level debug >> debug.log &
[1] 866
```

для Django:

ВАЖНО! Перед запуском приложения Django не забудьте установить в модуле конфигурации **settings.py** разрешение для доступа со всех внешних адресов и отключить режим отладки:

```
DEBUG = True
ALLOWED_HOSTS = ['*']
```

Запускаем командой:

```
gunicorn -b 0.0.0.0:8000 -w 3 application.wsgi >>
debug.log &
```

1.3.2 Развёртывание Java приложения. Скачиваем пакет JDK8 для Linux, например, здесь: https://bell-sw.com/pages/downloads/

Выбираем для скачивания JDK8 x86 64 for Linux в виде **.DEB** пакета (для Debian Linux). Должен загрузиться файл, примерно, со следующим именем: bellsoft-jdk8u352+8-linux-amd64.deb. Копируем загруженный deb-пакет на виртуальную машину в папку /opt. Устанавливаем с помощью apt:

```
apt install /opt/bellsoft-jdk8u352+8-linux-amd64.deb
```

Проверяем корректность установки Java-среды:

```
root@aislab22:/opt# java -version
openjdk version "1.8.0_352"
OpenJDK Runtime Environment (build 1.8.0_352-b08)
OpenJDK 64-Bit Server VM (build 25.352-b08, mixed mode)
```

Далее собираем проект. Проверьте, что в **pom.xml** установлен плагин для сборки всех зависимостей проекта в единый jar-файл:

```
<build>
   <plugins>
      <plugin>
         <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Собираем проект с помощью команды **`mvn package`** или запускаем «**`package`**» через панель Maven в IDE. В результате будет создан *.jar файл вашего приложения в папке **`target`** в директории проекта, например: **`target/ais-lab-0.0.1-SNAPSHOT.jar`**

Копируем **jar-файл** и файл конфигурации **application.properties** на виртуальную машину в отдельную папку. Папку с приложением переносим в системную директорию, которую обычно используют для сторонних приложений: **`/opt`**

ВАЖНО! Перед запуском приложения не забудьте убедиться, что в **application.properties** выставлены настройки для подключения к серверу **MySQL / MariaDB**, а не к локальной БД.

ВАЖНО! При запуске приложения в терминале Linux терминал будет заблокирован, и весь вывод приложения будет также выводиться в консоль, чтобы запустить приложение в фоновом режиме и перенаправить вывод в текстовый файл (например, для анализа возможных ошибок) используйте следующую конструкцию:

```
команда_запуска_приложения >> output.log &
```

Здесь >> **output.log** означает перенаправление вывода в файл **output.log**, символ & - запуск в фоновом режиме.

Чтобы остановить запущенный в фоне процесс используйте команду:

```
pkill -f имя_процесса
```

Для Java-приложений в качестве имени процесса обычно можно использовать имя jar-файла, например:

```
pkill -f ais-lab
```

Переходим в директорию приложения (где находится jar) и запускаем:

```
java -jar ais-lab-0.0.1-SNAPSHOT.jar >> output.log &
```

1.4 Нагрузочное тестирование с помощью Locust. Locust – это открытый Python framework для тестирования производительности. Инструмент имеет графический интерфейс (веб-интерфейс) и возможность распределенного тестирования. Сценарии для тестирования также реализуются на языке Python.

Для установки Locust необходимо установить соответствующий пакет в среде Python:

```
pip3 install locust
```

После установки пакета в папке с исполняемыми файлами вашей среды Python появится исполняемый файл приложения Locust:

```
C:\Users\Bogdan\Desktop\Python\usatu\env39\Scripts\locust.exe
```

Данный исполняемый файл будет использоваться для запуска процесса тестирования. Запуск можно осуществлять как из терминала в вашей среде разработки, так и из терминала ОС. Для запуска locust из терминала ОС убедитесь, что данный путь к папке Scripts внесен в системную переменную PATH (см. «Свойства компьютера» → «Дополнительные параметры «системы → «Переменные среды»). Для проверки запустите системную консоль **cmd** и наберите команду «locust», если все настроено верно, то вывод должен быть следующим (рис. 6):

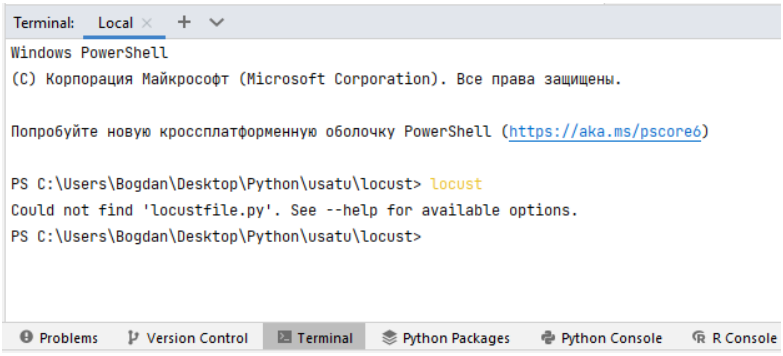


Рис. 6 – Вывод команды «locust» без параметров

Сценарий для тестирования с помощью Locust реализуется в отдельном Python файле. Пример такого файла представлен в `locust_test.py`.

В основе сценария Locust находится класс, объект которого имитирует пользователя тестируемого приложения. Для эмуляции пользователя (клиента) веб-приложения создается класс, наследующий «суперкласс» HttpUser:

```
class RESTServerUser(HttpUser):
    ...
```

«Поведение» клиента веб-приложения реализуется с помощью специальных методов (задач) и атрибутов данного класса, рассмотрим основные из них.

Задаем время (например, диапазон от 1 до 5 с.) ожидания пользователя перед выполнением новой задачи:

```
wait_time = between(1.0, 5.0)
```

Задаем действие клиента при первом подключении к веб-приложению (обычно это страница логина или стартовая страница приложения):

```
def on_start(self):
        self.client.get("/docs")
```

Клиент запрашивает (операция GET) несколько записей из БД:

```
@tag("get_all_task")
@task(3)
def get_all_task(self):
    """ Тест GET-запроса (получение нескольких записей о погоде) """
    city_id = random.randint(0, 3)       # генерируем случайный id в
диапазоне [0, 3]
    city_name = CITY_NAMES[city_id]      # получаем случайное значение
населенного пункта из списка CITY_NAMES
    with self.client.get(f'/api/weatherforecast/{city_name}',
                         catch_response=True,
                         name='/api/weatherforecast/{city_name}') as
response:
        # Если получаем код HTTP-код 200, то оцениваем запрос как
"успешный"
        if response.status_code == 200:
            response.success()
        # Иначе обозначаем как "отказ"
        else:
            response.failure(f'Status code is {response.status_code}')
```

В данном примере Python-кода добавлены «декораторы» (инструкции со знаком «@»). В Python декораторы позволяют выполнять дополнительные действия при выполнения основной функции, в данном примере используются следующие встроенные декораторы Locust:

`@tag("get_task")` – присваиваем действию клиента (get_task) название, чтобы данное действие можно было запускать отдельно.

`@task(3)` – присваиваем «вес» для метода get_task. Вес будет определять частоту выполнения данного действия относительно других (т.е. метод get_task будет выполняться в 3 раза чаще, чем метод, у которого установлено @task(1)).

При описании действий пользователя также необходимо учитывать возвращаемый вашим веб-приложением результат для определения результата выполнения действия (успешно или отказ). В рассматриваемом примере распознаются

HTTP-коды: если получили код 200 – «успешно»; остальные квалифицируем как «отказ».

При реализации сценария действий клиента выберете самые «тяжёлые» методы, т.е. те, которые генерируют наиболее сложные запросы к вашей БД – это позволит эмулировать нагрузку системы для «худшего случая».

Запуск возможен с помощью одной из следующих команд:

1) запуск в консольном режиме (параметр `--headless`)

```
locust -f locust_test.py --host=http://192.168.56.104:8000 --tags get_task get_id_task --run-time 10s -u 100 -r 10 --headless
```

2) запуск с веб-интерфейсом, доступным по адресу **http://localhost:8089**

```
locust -f locust_test.py --host=http://192.168.56.104:8000 --tags get_task get_id_task
```

`--host` – адрес тестируемого веб-приложения;

`--tags` – список выполняемых действий (методов, обозначенных декоратором **@tag**);

`--run-time` – длительность нагрузки (10s – 10 секунд, 10m – 10 минут и т.д.);

`-u` – максимальное количество пользователей;

`-r` – количество пользователей, которое будет добавляться каждую секунду до достижения максимального значения;

ВАЖНО!

- **Перед запуском тестов прочитайте п. 1.5 методических рекомендаций (см. ниже).**
- **Пробный запуск теста Locust сделайте в консольном режиме (--headless) для вывода возможных ошибок в вашем сценарии.**

Стартовая страница веб-интерфейса Locust выглядит следующим образом рис. 7:

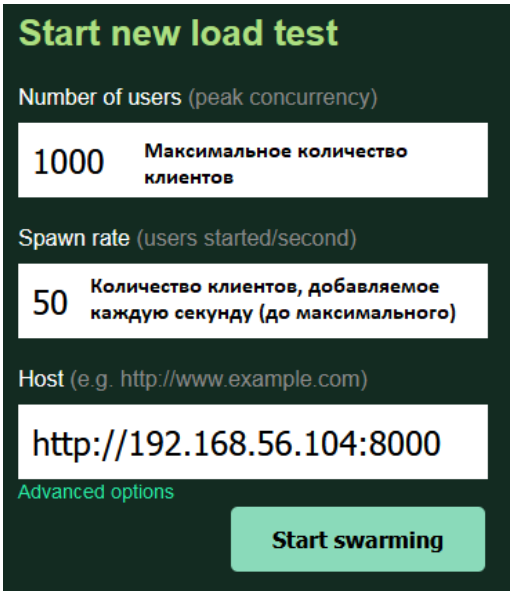


Рис. 7 – Стартовая страница Locust

Чтобы запустить тестирование нажмите «**Start swarming**», появится форма с таблицей адресов запросов и детализацией по количеству запросов (Requests), среднему (Average), максимальному (Max) и минимальному (Min) времени выполнения запросов, объему передаваемых данных (Average size) и количеству запросов в секунду (Current RPS).

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
PUT	/api/weatherforecast	1863	5	25	92	190	39	2	253	0	63.1	0.4
GET	/api/weatherforecast/{city_name}	1801	3	60	130	240	76	2	291	248	62.2	0.2
GET	/api/weatherforecast?city_id={ID}	6270	11	58	130	240	75	14	292	122	202	0.7
GET	/docs	1000	0	110	160	200	112	65	202	931	0	0
	Aggregated	**10934**	**19**	**58**	**130**	**230**	**72**	**2**	**292**	**196**	**327.3**	**1.3**

Рис. 8 – Процесс тестирования в главном окне Locust

Значения будут меняться динамически в процессе выполнения теста. Тест будет выполняться, пока не будет нажата кнопка «Stop» в правом верхнем углу.

1.5 Фиксирование результатов тестирования для отчета. Для выполнения необходимых этапов тестирования алгоритм действий следующий:

1) Для первого этапа тестирования выберите только те действия пользователя, которые реализую GET и PUT запросы. Также на первом этапе необходимо подобрать такое количество эмулируемых клиентов, чтобы нагрузка на CPU в системном мониторе (см. пп. 3) **не превышала 60%.**

2) Запустите тест и после 3-5 мин. выполнения сделайте скриншот основной таблицы (как на рис. 8).

3) В процессе выполнения теста зайдите в консоль вашей виртуальной машины, где установлен сервер БД и выполните команду `htop` (запуститься монитор ресурсов).

Чтобы отфильтровать в мониторе ресурсов необходимые процессы нажмите клавишу « \ » на клавиатуре и введите «python» (или «java» для Java-приложения), монитор отобразит все процессы python в системе (рис.9)

Делаем скриншот вывода htop (как на рис. 9).

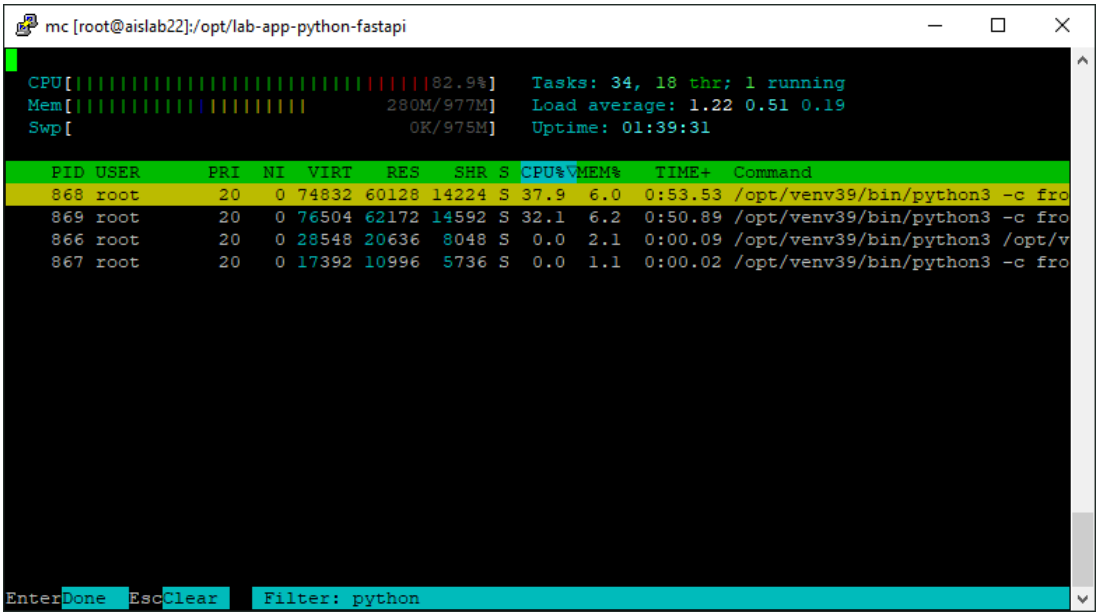


Рис. 9 – Вывод команды htop (процессы python)

4) Аналогично п.3 выводим процессы БД и делаем скриншот (рис. 10).

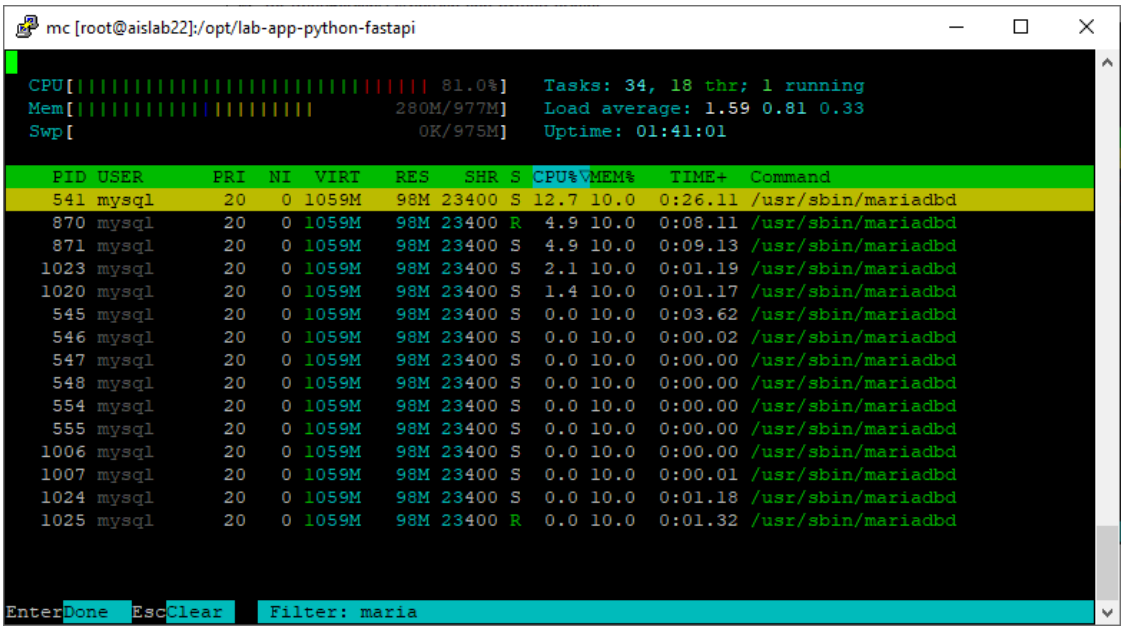


Рис. 10 – Вывод команды htop (процессы БД MariaDB)

5) Во время выполнения теста запускаем общий монитор ресурсов устройств ввода-вывода командой **`atop`** (рис. 11). Делаем скриншот и фиксируем максимальное значение в поле **DSK** (нагрузка на жесткий диск).

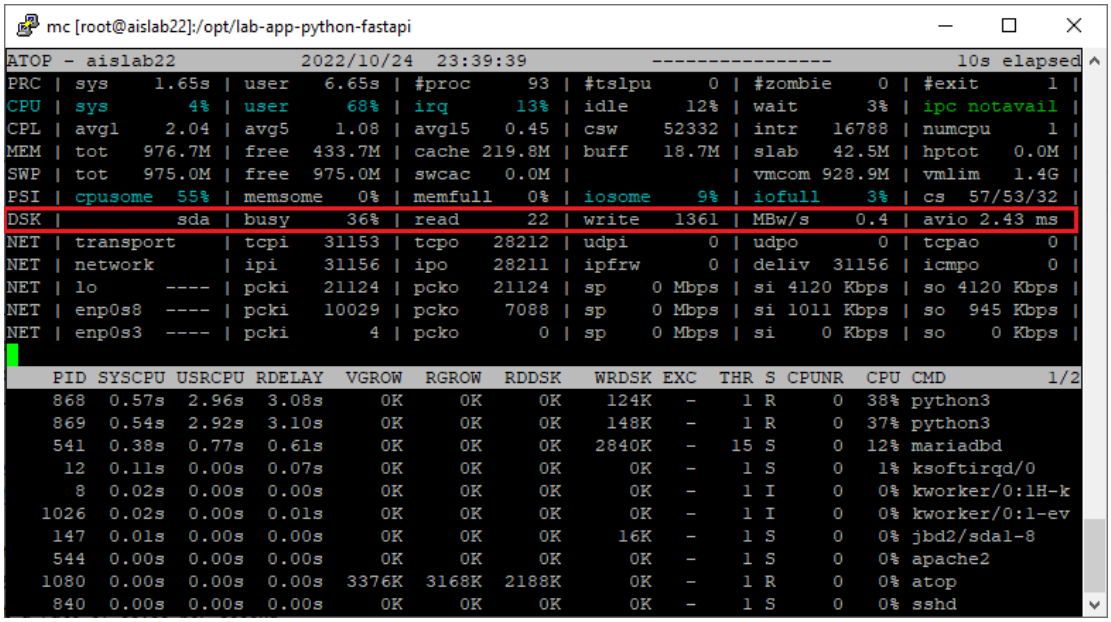


Рис. 11 – Вывод команды atop

4) Запустите новый тест. Включите в тестирование хотя бы один (наиболее часто выполняемый на вашем сервере) POST-запрос. и зафиксируйте результаты как указано в **пп. 3-5**. Как изменились значения и почему? **Сделать вывод.**

5) Подберите максимальное значение эмулируемых пользователей таким образом, чтобы общая доля отказов (FAILURES) оставалась в пределах не более **1%** , а нагрузка на CPU была в пределах 90-100% в течение 5 минут. Зафиксируйте результаты как указано в **пп. 3-5**. Указать в отчете, что является наиболее нагруженным элементом сервера (процессор, жесткий диск или оперативная память)? Какие возможны варианты решения данной проблемы?

6) После завершения тестирования согласно пп.5 перейдите на вкладку «Download Data» в Locust и выберите «**Download Report**». **Сохраните отчет, он должен быть предоставлен вместе с отчетом по лабораторной работе.**

Дополнительная литература:

https://habr.com/ru/company/infopulse/blog/430502/

https://docs.locust.io/en/stable/quickstart.html

Методические рекомендации по разработке веб-клиента (ReactJS) (Часть 4)

Пример с кодом веб-клиента, реализованного на основе *ReactJS*, находится в архиве **`ais-reactjs-client.zip`**. Пример кода подробно комментирован. В примере реализованы следующие компоненты:

`App.js` – корневой компонент приложения, рендерит вложенные компоненты `CityWeatherForm и CityWeatherMonitor.`

`CityWeatherForm` – компонент реализует форму для обновления данных на сервере (**PUT**-запрос);

`CityWeatherMonitor` – компонент реализует периодическую отправку **GET**-запроса на сервер для динамического обновления таблицы с погодой.

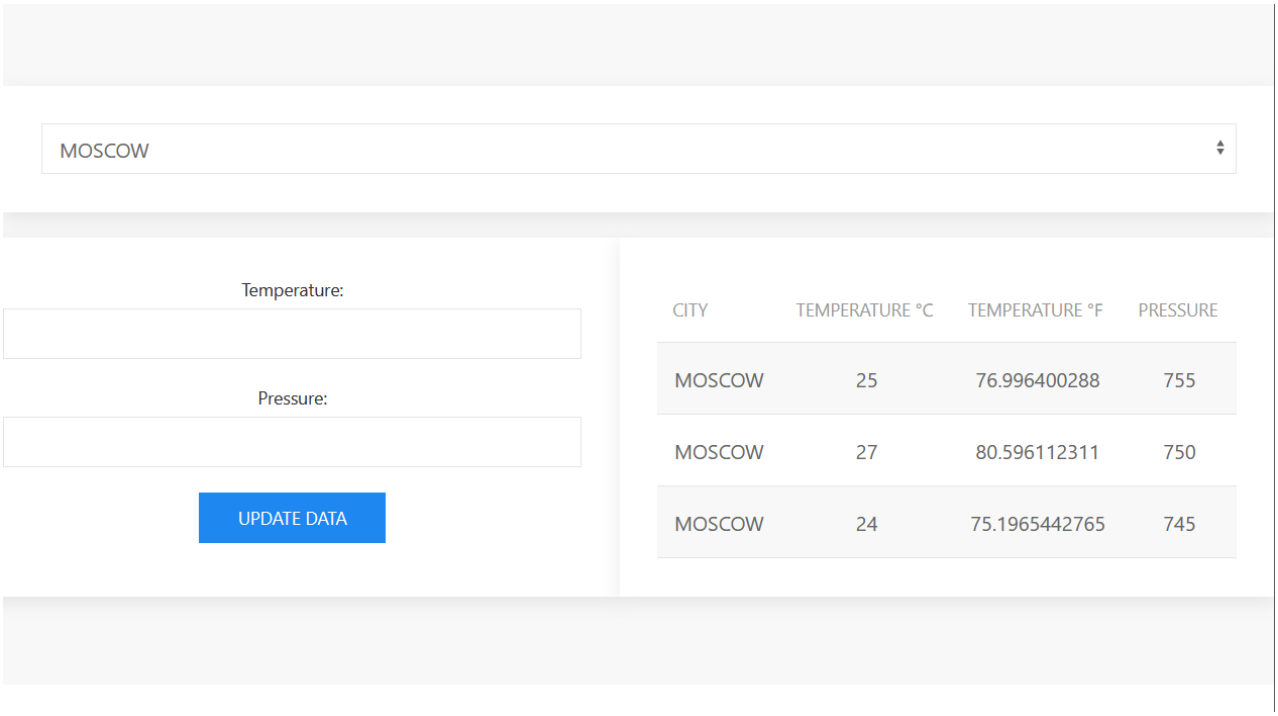


Рис. 1 – Интерфейс приложения *ReactJS* из примера

Для запуска примера и развёртывания своего базового проекта необходима установка среды **Node.js**:

https://nodejs.org/en/download/

Проект приложения создан с помощью менеджера пакетов **`npm`** и утилиты для создания базовой структуры React приложения: **`create-react-app`**.

Для того чтобы глобально установить create-react-app, воспользуйтесь следующей командой:

```
npm i -g create-react-app
```

Для создания шаблона своего приложения, выполните следующую команду:

```
create-react-app my-react-app
```

Более подробную информацию по созданию базового проекта React и tutorial по созданию базовых компонентов можно посмотреть здесь:

https://habr.com/ru/company/ruvds/blog/428077/

В данном примере реализовано простое React приложение с малым количеством компонентов. При усложнении проекта и, как следствие, увеличении количества компонентов, потребуется создание более гибкой архитектуры приложения. Одним из подходов, позволяющих реализовать более продвинутую архитектуру в React приложении, является архитектурный подход **Flux** и его наиболее популярная реализация **Redux**.

Дополнительную информацию по данным архитектурным подходам можно посмотреть здесь:

- https://madfinn.medium.com/flux-%D0%BF%D1%80%D0%BE%D1%82%D0%B8%D0%B2-redux-%D1%81%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5-883c0932931a

- https://medium.com/@marina.kovalyova/flux-the-react-js-application-architecture-773f515d068d

ВАЖНО! Для корректной работы React приложения с серверной частью необходима настройка разрешений кросс-браузерных запросов CORS (Cross-Origin Resource Sharing) на стороне серверного приложения.

Настройка CORS в Django. Устанавливаем дополнительное middleware для Django:

```
pip install django-cors-headers
```

После установки пакета вносим изменения в файл **settings.py**. Добавляем в **INSTALLED_APPS**:

```
INSTALLED_APPS = [
    ...
    'corsheaders',
]
```

Добавляем параметр в **MIDDLEWARE**:

```
MIDDLEWARE = [
```

```
    ...,
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...,
]
```

Создаём дополнительные переменные конфигурации в **settings.py**:

```
    # Запрещает / разрешает CORS запросы для всех адресов
CORS_ORIGIN_ALLOW_ALL = False

# Список определяет конкретные адреса, для которых разрешён CORS
CORS_ORIGIN_WHITELIST = [
     'http://localhost:3000',
]
```

Настройка CORS в FastAPI. После создания экземпляра приложения (app = FastAPI()) добавляем следующий код:

```
    app = FastAPI()          # создаём экземпляр приложения FastAPI

    # Настройка CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],     # здесь указываем разрешённые адреса, символ "*" разрешает
запросы для всех адресов
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Подробнее см.: https://fastapi.tiangolo.com/tutorial/cors/

Для настройки CORS в приложении Spring используются соответствующие аннотации для всего контроллера:

```
    @CrossOrigin(origins = "http://localhost:4200", maxAge = 3600,
allowCredentials = "true")
    @RestController
```

или для отдельных методов контроллера:

```
    @CrossOrigin(origins = "http://localhost:4200", allowedHeaders =
"Requestor-Type", exposedHeaders = "X-Get-Header")
    @GetMapping
    public ResponseEntity<T> getData()
```

ВАЖНО! При отладке вашего React приложения также используйте встроенную веб-консоль браузера для вывод ошибок и сообщений console.log():

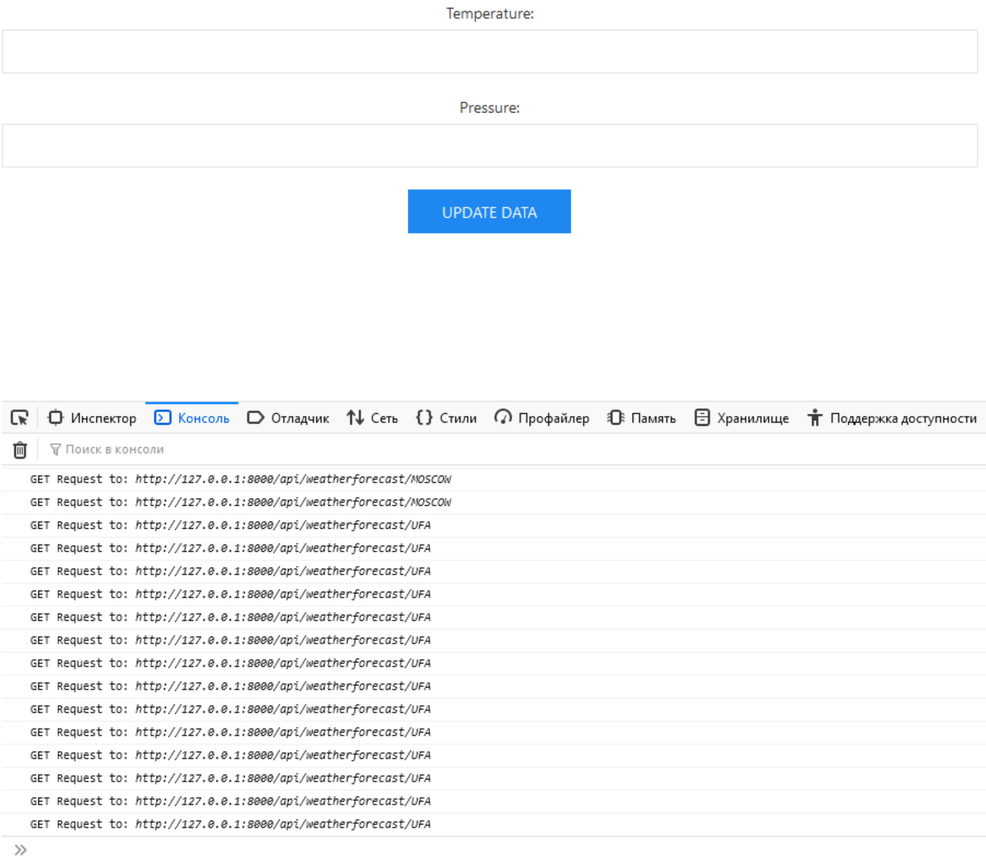


Рис. 2 – Вывод веб-консоли браузера Firefox

Приложение 1

Проектирование интерфейса пользователя и бизнес-логики клиентского приложения.

Структура типового интерактивного приложения в рамках клиент-серверного взаимодействия:

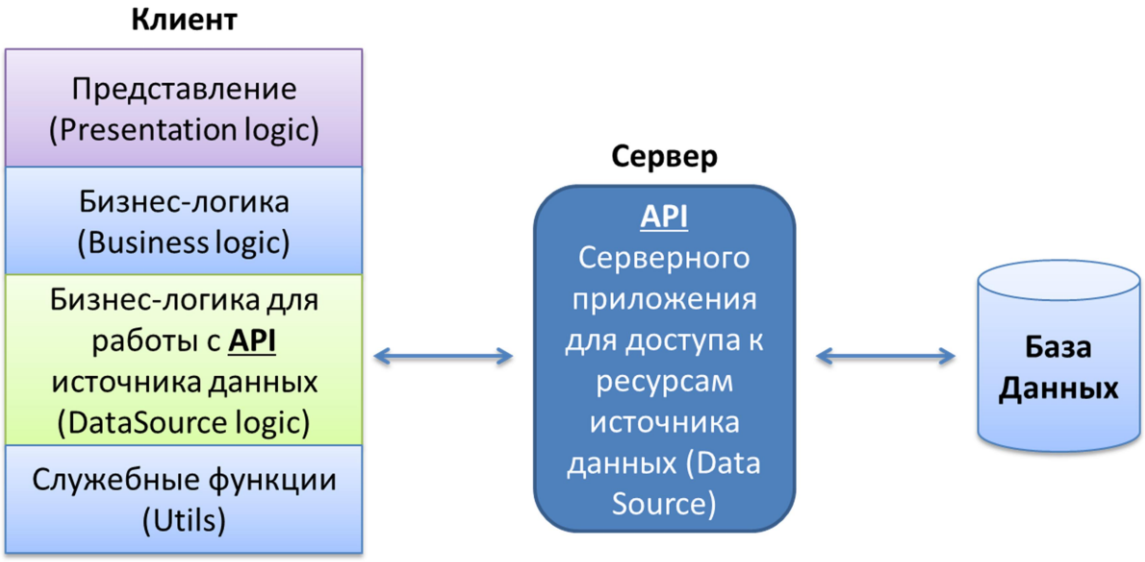


Рисунок 1 – Структура типового интерактивного приложения в рамках клиент-серверного взаимодействия

Презентационная логика (*Presentation Logic*) как часть приложения определяется тем, что пользователь видит на своем экране, когда работает приложение:

- интерфейсные экранные формы, которые пользователь видит или заполняет в ходе работы приложения (*Graphical User Interface, GUI*);
- результаты решения некоторых промежуточных задач;
- справочная информация.

Основными задачами презентационной логики:

- формирование экранных изображений;
- чтение и запись в экранные формы информации;
- управление экраном;
- обработка движений мыши и нажатий клавиш клавиатуры.

Бизнес-логика, или логика приложений (*Business processing Logic*) — часть кода приложения, определяющая алгоритмы решения конкретных задач приложения.

Логика обработки данных (*Data manipulation Logic*) — часть кода приложения, связанная с обработкой данных внутри приложения.

Логика предоставления данных (*DataSource Logic*) — программная часть клиентского приложения, отвечающая за обмен данными с серверным приложением,

т.е. подключение, формирование запросов, получение ответа и т.д. Данный слой содержит функции работы с API серверного приложения. Если интерактивное приложение работает напрямую с базой данных, то этот слой также может содержать функции для работы с СУБД (*Repository Service*).

Служебные функции (*Utils*) – дополнительный необязательный слой, обычно содержащий ряд вспомогательных функций и методов, например, математические функции, преобразователи величин, функции форматирования строк.

Проектирование пользовательского интерфейса приложения

Графический интерфейс пользователя (**GUI**) — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений. В GUI пользователь имеет произвольный доступ (с помощью устройств ввода — клавиатуры, мыши, джойстика и т. п.) ко всем видимым экранным объектам (элементам интерфейса) и осуществляет непосредственное манипулирование ими.

Графический интерфейс пользователя является частью пользовательского интерфейса и определяет взаимодействие с пользователем на уровне визуализированной информации. Можно выделить следующие виды графического интерфейса пользователя:

- простой: типовые экранные формы и стандартные элементы интерфейса, обеспечиваемые самой подсистемой GUI;
- истинно-графический, двухмерный: нестандартные элементы интерфейса и оригинальные метафоры, реализованные собственными средствами приложения или сторонней библиотекой;

Проектирование графического интерфейса пользователя представляет собой междисциплинарную деятельность. Оно требует усилий многофункциональной бригады — один человек, как правило, не обладает знаниями, необходимыми для реализации многоаспектного подхода к проектированию GUI-интерфейса. Надлежащее проектирование GUI интерфейса требует объединения навыков художника-графика, специалиста по анализу требований, системного проектировщика, программиста, эксперта по технологии, специалиста в области социальной психологии, а также, возможно, некоторых других специалистов, в зависимости от характера системы.

В современном мире миллиарды вычислительных устройств. Еще больше программ для них. И у каждой свой интерфейс, являющийся «рычагами» взаимодействия между пользователем и машинным кодом. Не удивительно, что чем лучше интерфейс, тем эффективнее взаимодействие.

Однако далеко не все разработчики и даже дизайнеры, задумываются о создании удобного и понятного графического интерфейса пользователя.

Какие элементы интерфейса (ЭИ) создавать?

1. Разработка интерфейса обычно начинается с определения задачи или набора задач, для которых продукт предназначен.

2. Простое должно оставаться простым. Не стоит усложнять интерфейсы. Нужно постоянно думать о том, как сделать интерфейс проще и понятнее.

3. Пользователи не задумываются над тем, как устроена программа. Все, что они видят — это интерфейс. Поэтому, с точки зрения потребителя именно интерфейс является конечным продуктом.

4. Интерфейс должен быть ориентированным на человека, т. е. отвечать нуждам человека и учитывать его слабости. Нужно постоянно думать о том, с какими трудностями может столкнуться пользователь.

5. Необходимо думать о поведении и привычках пользователей. Не менять хорошо известные всем ЭИ на неожиданные, а новые делать интуитивно понятными.

Разрабатывать интерфейс необходимо исходя из принципа наименьшего возможного количества действий со стороны пользователя.

Какой должен быть дизайн элементов интерфейса? В дизайне ЭИ нужно учитывать все: начиная от цвета, формы, пропорций, заканчивая когнитивной психологией. Однако, несколько принципов все же стоит отметить:

1. Цвет. Цвета делятся на теплые (желтый, оранжевый, красный), холодные (синий, зеленый), нейтральные (серый). Обычно для ЭИ используют теплые цвета. Это как раз связано с психологией восприятия.

2. Стоит отметить, что мнение о цвете — очень субъективно и может меняться даже от настроения пользователя.

3. Форма. В большинстве случаев — прямоугольник со скругленными углами. Или круг. Опять же, форма, как и цвет достаточно субъективна. Основные ЭИ (часто используемые) должны быть выделены (например, размером или цветом).

4. Иконки в программе должны быть очевидными. Или подписанными.

Как правильно расположить элементы интерфейса на экране?

1. Есть утверждение, что визуальная привлекательность основана на пропорциях. Помните известное число 1.62? Это так называемый принцип Золотого сечения. Суть в том, что весь отрезок относится к большей его части так, как большая часть относится к меньшей. Например, общая ширина сайта 900px, делим 900 на 1.62, получаем ~555px, это ширина блока с контентом. Теперь от 900 отнимаем 555 и получаем 345px. Это ширина меньшей части.

2. Перед расположением, ЭИ следует упорядочить (сгруппировать) по значимости. Т. е. определить, какие наиболее важны, а какие — менее.

3. Обычно (но не обязательно), элементы размещаются в следующей градации: слева направо, сверху вниз. Слева вверху самые значимые элементы, справа внизу — менее. Это связано с порядком чтения текста. В случае с сенсорными экранами, самые важные элементы, располагаются в области действия больших пальцев рук.

4. Необходимо учитывать привычки пользователя. Например, если в Windows кнопка закрыть находится в правом верхнем углу, то программе аналогичную кнопку необходимо расположить там же. Т.е. интерфейс должен иметь как можно больше аналогий, с известными пользователю вещами.

5. Размещать ЭИ стоит поближе там, где большую часть времени находится курсор пользователя, что бы ему не пришлось перемещать курсор, например, от одного конца экрана к другому.

6. Элемент интерфейса можно считать видимым, если он либо в данный момент доступен для органов восприятия человека, либо он был настолько недавно воспринят, что еще не успел выйти из кратковременной памяти. Для нормальной работы интерфейса, должны быть видимы только необходимые вещи — те, что идентифицируют части работающих систем, и те, что отображают способ, которым пользователь может взаимодействовать с устройством.

Отступы между ЭИ лучше делать равными или кратными друг другу. *Как элементы интерфейса должны себя вести?*

1. Например, при удалении файла, появляется окно с подтверждением: «Да» или «Нет». Со временемпользователь перестает читать предупреждение и по привычке нажимает «Да». Поэтому диалоговое окно, которое было призвано обеспечить безопасность, абсолютно не выполняет своей роли. Следовательно, необходимо дать пользователю возможность отменять, сделанные им действия.

2. Если пользователю дают информацию, которую он должен куда-то ввести или как-то обработать, то информация должна оставаться на экране до того момента, пока человек ее не обработает. Иначе он может просто забыть.

3. Нужно избегать двусмысленности. Например, на фонарике есть одна кнопка. По нажатию фонарик включается, нажали еще раз — выключился. Если в фонарике перегорела лампочка, то при нажатии на кнопку не понятно, включаем мы его или нет. Поэтому, вместо одной кнопки выключателя, лучше использовать переключатель (например, checkbox с двумя позициями: «вкл.» и «выкл.»). За исключением случаев, когда состояние задачи очевидно.

4. Имеет смысл делать монотонные интерфейсы. Монотонный интерфейс — это интерфейс, в котором какое-то действие, можно сделать только одним способом. Такой подход обеспечит быструю привыкаемость к программе и автоматизацию действий.

5. Не стоит делать адаптивные интерфейсы, которые изменяются со временем. Так как для выполнения какой-то задачи, лучше изучать только один интерфейс, а не несколько. Пример — стартовая страница браузера Chrome.

6. Если задержки в процессе выполнения программы неизбежны или действие, производимое пользователем очень значимо, важно, чтобы в интерфейсе была предусмотрена сообщающая о них обратная связь. Например, можно использовать индикатор хода выполнения задачи (status bar).

7. ЭИ должны отвечать. Если пользователь произвел клик, то ЭИ должен как-то отозваться, чтобы человек понял, что клик произошел.

При создании интерфейса рекомендуется использовать существующие принципы проектирования пользовательского интерфейса. Далее приводятся шесть принципов, вобравших в себя многое из того, что на данный момент известно о разработке эффективного пользовательского интерфейса. Каждый из них включает в себя несколько связанных между собой идей, более детализированных по сравнению с общими вопросами. Этими общими вопросами являются структура, простота, видимость, обратная связь, толерантность и повторное использование.

Структурная организация пользовательского интерфейса должна быть целесообразной, осмысленной и удобной. Она должна базироваться на четких, целостных моделях, очевидных и распознаваемых пользователями. При этом родственные понятии должны быть связаны, а независимые – разделены. Непохожие элементы должны дифференцироваться, а похожие – выглядеть похоже.

Структурный принцип связан с общей архитектурой интерфейса и напрямую отражает представление о пользовательском интерфейсе как о диалоге между разработчиками и пользователями. Организация хороших интерфейсов продумывается очень тщательно, таким образом, чтобы отражать структуру решаемых системой задач и способ мышления пользователей относительно этих задач. Очень часто, особенно при использовании современных визуальных сред разработки, расположение визуальных компонентов внутри форм или диалогов и их распределение между ними оказывается почти случайным и отражает в лучшем случае последовательность, в которой программистами затрагивались те или иные вопросы. По идее, свойства и функции, которые чаще всего используются совместно или рассматриваются пользователями как связанные друг с другом, должны располагаться вместе или, по крайней мере, должны быть четко и ясно взаимосвязаны. Что же до тех элементов, которые в контексте задачи или в сознании пользователя никак не связаны между собой, то они должны быть разнесены в интерфейсе. Подобное должно быть подобно. Похожая информация должна быть организована с помощью похожих решений, а объекты, обладающие похожим поведением, должны иметь общее представление.

Принцип простоты: следует максимально упрощать управление наиболее распространенными операциями. При этом общение с пользователем должно вестись на понятном для него языке. Должны предоставляться ссылки, логичным образом указывающие на более сложные процедуры. Процесс проектирования интерфейса – это всегда борьба за компромисс. Упрощение чего-то одного неизбежно приводит к усложнению чего-то другого. Если уменьшить количество меню, увеличится число пунктов в каждом из них. Если сделать маленькими все диалоговые окна, включив в них как можно меньше элементов, любое взаимодействие пользователя с системой обернется для него необходимостью обращаться к большому количеству таких окошек.

Невозможно сделать все на свете простым. Следование принципу простоты требует от вас знания того, какие задачи выполняются пользователем наиболее часто и

какие из них, с точки зрения пользователя, проще. Именно такие задачи следует упрощать, чтобы пользователь могбыстро их решить.

Принцип видимости: все функции и данные, необходимые для выполнения данной задачи, должны быть видны, чтобы пользователь не отвлекался на дополнительную и избыточную информацию.

Принцип видимости связан с проектированием таких пользовательских интерфейсов, в которых видны все элементы, нужные для выполнения данной задачи. Цель – перейти от философии WYSIWYG (What You See Is What You Get – что видишь на экране, то и получишь в результате) к философии WYSIWYN (What You See Is What You Need – на экране видишь то, что тебе нужно). Интерфейсы WYSIWYN оставляют видимыми те, и только те элементы, которые действительно нужны пользователю для выполнения операции. С одной стороны, в задачи проектирования входит создание такого интерфейса, на котором были бы явно видны все нужные и важные функции. С другой стороны, хороший интерфейс не должен заваливать пользователя слишком большим количеством возможных вариантов или смущать его избыточной информацией. WYSIWYN-интерфейсы лучше уже тем, что они принимают во внимание ограниченность объема «оперативной памяти» человека и способность узнавать вещи быстрее, чем вспоминать. Нагрузка на долговременную память уменьшается за счет того, что пользователь постоянно видит все необходимые опции и варианты. На кратковременную память нагрузка снижается за счет того, что пользователю не приходится запоминать и затем воспроизводить информацию, содержащуюся в какой-то другой части интерфейса.

Принцип обратной связи: сообщайте пользователям о действиях системы, ее реакциях, изменениях состояния или ситуации, об ошибках и исключениях, которые важны для них. Сообщения должны быть четкими, краткими, однозначными и написанными на языке, понятном пользователю.

Хорошие пользовательские интерфейсы находятся в диалоге с пользователями, сообщая им о том, что происходит в системе. Принцип обратной связи указывает разработчикам некоторые правила этого диалога. Практичные системы информируют пользователя о множестве вещей. К примеру, они должны позволять ему узнавать о том, как воспринимаются вводимые им данные. Всякий раз, когда меняется внутреннее состояние системы, и это может оказать какое-либо влияние на работу пользователя, его следует уведомлять об этом, особенно если меняется интерпретация системой его действий. Разумеется, пользователь должен знать о действиях, которые запрещены или игнорируются. При этом принцип обратной связи не может служить оправданием созданию бесконечных окошек сообщений. Информирование пользователя – не самоцель, а способ организации диалога в компактной и естественной форме. Пользователям также требуются сообщения об ошибках и исключительных ситуациях. Во многих программах эти сообщения, к сожалению, неинформативны и способны ввести в заблуждение. Можно иногда встретить даже оскорбляющие сообщения, после прочтения которых пользователю может стать не по себе. Вряд ли человек вдохновится,

скажем, такой надписью: «Неправильно! Введите корректные данные!». Такое сообщение не только неявно предполагает, что пользователь – какой-то нехороший человек, но и, по сути дела, не дает никакой информации. Здесь не сказано, что именно неправильно и почему. Грамотно составленные сообщения об ошибках – это еще один пример хорошей организации общения с пользователем. Рекомендации здесь можно дать такие: краткость; язык, понятный пользователю; простота понимания. Прежде всего информативным должен быть заголовок сообщения. Он должен в сжатой форме описывать проблему, а уже само сообщение должно раскрывать подробности и предлагать способы решения или последовательность корректирующих действий.

Принцип толерантности: интерфейс должен быть гибким и толерантным. Ущерб, наносимый ошибками пользователей, необходимо снижать за счет возможности отмены и повтора действий и за счет предотвращения появлений этих ошибок путем анализа различных форматов ввода и разумной интерпретации любых разумных действий.

Интерфейс можно делать более или менее толерантным в зависимости от того, какие данные проверяются и когда. Проверка всех полей разом по окончании ввода данных – практика распространенная и иногда оправданная. При этом толерантности системе добавит автоматическая подсветка поля с неправильными данными, установка на него курсора, плюс короткое, информативное сообщение в строке состояния. Больше всего пользователи страдают от программ, которые по окончании ввода во все поля проверяют всю форму, и в случае неправильных данных хотя бы в одном из полей пользователь оказывается снова один на один с пустым бланком. Казалось бы, такое решение ужасно нелепо, но оно встречается очень часто, в том числе и в коммерческих программах.

В целом проверка неиспользуемых полей или полей, которые никак не обрабатываются системой и представляют интерес только для пользователей (в том виде, в каком они были введены), ведет к снижению толерантности ПО. Например, проверка того, что в поле примечаний присутствуют только буквенно-цифровые символы, избыточна. Кроме того, если пользователь вдруг захочет выделить что-нибудь в этом поле спецсимволом или с помощью псевдографики, у него возникнут проблемы.

Принцип повторного использования: следует многократно использовать внутренние и внешние компоненты и принципы поведения системы, поддерживая устойчивость осмысленно, а не просто за счет избыточности. Это способствует уменьшению объема информации, которую пользователям приходится запоминать и о которой приходится думать каждый раз заново.

Применяя повторное использование внешних и внутренних компонентов и решений, распространяющихся на всю систему, разработчик может создать не только более целостный интерфейс, но и более дешевый продукт. Стремление к одной лишь устойчивости повышает стоимость разработки, да и в ряде случаев оказывается сизифовым трудом. Нужно стремиться к устойчивости в контексте решаемых системой задач и области ее использования, устойчивость ни в коем случае не может быть

самоцелью, иначе она может выглядеть несколько глуповато и даже приводить, как ни странно, к неудачным с точки зрения непротиворечивости системам.

Многие принятые стандарты и общепризнанные компоненты пользовательского интерфейса являют собой примеры неудавшихся попыток реализации устойчивости. Стандартные программные платформы порой навязывают разработчика плохо продуманные и неудачно спроектированные решения. Самые обычные диалоговые окна, скажем, могут обеспечивать целостность и при этом быть весьма низкосортными, выбор стандартных горячих клавиш оказывается случайным и никак не соответствующим принципу устойчивости.

Источники:

1. А.С. Саутин «Человеко-машинное взаимодействие»: методические указания к выполнению лабораторных работ.
2. Ю. В. Вайнилович «Интегрированные информационные системы предприятий»: методические рекомендации к лабораторным работам.
3. Проектирование графического интерфейса пользователя: https://www.bsuir.by/m/12_101523_1_108313.pdf
4. Статья «Бизнес-логика на страже качества»: https://www.software-testing.ru/library/testing/test-management/62-business-logic