



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»

ФАКУЛЬТЕТ Робототехника и комплексная автоматизация

КАФЕДРА Системы автоматизированного проектирования (РК-6)

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

на тему:

*«Разработка библиотеки функций на языке Python, реализующей
автоматизированное построение динамических графических
пользовательских интерфейсов в рамках CMS Django»*

Студент РК6-73Б
(Группа)

(Подпись, дата)

А.Р. Василян
(И.О.Фамилия)

Руководитель курсовой работы (проекта)

(Подпись, дата)

А.П. Соколов
(И.О.Фамилия)

Москва, 2022 г.

Оглавление

Введение.....	3
Методы взаимодействия пользователя и ЭВМ.....	3
Ограничительный.....	3
Направляющий	4
Методический подход к созданию универсального пользовательского интерфейса	7
Построение пользовательского интерфейса с использованием интерактивного машинного обучения.....	10
Разработка тестового web-приложения	10
Django	10
Docker	11
Разработка тестового web-приложения	12
Запуск web-приложения на тестовом сервере.....	18
Заключение	20
Список литературы	21

Введение

Интерфейс — это совокупность средств методов и правил взаимодействия управления контроля и т.д. между элементами системы;

Пользовательский интерфейс — это разновидность интерфейсов, в котором одна сторона представлена человеком-пользователем, другая — машиной-устройством. Представляет собой совокупность средств и методов, при помощи которых пользователь взаимодействует с различными чаще всего сложными машинами устройствами и аппаратурой;

Графический пользовательский интерфейс (GUI) — это разновидность пользовательского интерфейса, в котором элементы интерфейса меню кнопки значки списки и т.п., представленные пользователю на дисплее, исполнены в виде графических изображений.

В данной курсовой работе были рассмотрены способы взаимодействия пользователя и ЭВМ, подходы разработки GUI и было разработано несложное web-приложение.

Методы взаимодействия пользователя и ЭВМ

Была изучена статья “Метод построения оконного интерфейса пользователя на основе моделирования пользовательских целей”. При изучении были выделены два метода взаимодействия пользователя и ЭВМ: ограничительный и направляющий.

Ограничительный

Для средств взаимодействия пользователя и ЭВМ с оконным интерфейсом типичным является ограничительный метод. Метод заключается в том, что пользователю предоставляется некий набор операций, благодаря которым он может выполнять определенные действия с описанными в ЭВМ объектами своей деятельности. Операция имеет название, исходные данные и результаты, представляющие собой объекты воздействия операции. Пользователь решает, какую из операций необходимо выбрать в данный момент для выполнения своего задания, выбирает нужную операцию и задает для нее исходные данные. После чего ЭВМ выполняет указанную операцию, активируя соответствующие функции приложения, и выдаёт результаты операции пользователю.

По итогам выполнения очередной операции пользователь решает, какую следующую операцию ему нужно выбрать, передаёт ее ЭВМ на выполнение и т.д. Этот процесс он должен продолжать до тех пор, пока в итоге выполнения операций не будет достигнут желаемый результат, соответствующий выполненному заданию. Следовательно, пользователь должен сам планировать ход выполнения своего задания из предоставляемых ему операций. Обобщенная схема ограничительного метода взаимодействия приведена на рис. 1. Для ограничительного метода считается, что у пользователя присутствуют процедурные знания, необходимые для планирования процесса выполнения своих заданий.

пользователя требуется в ответ на запросы ЭВМ ввести исходные данные, необходимые для этих операций.

Направляющий метод взаимодействия “пользователь-ЭВМ” состоит из следующих основных этапов:

- информирование пользователя о множестве допустимых заданий, которые может выполнять ЭВМ в рамках данного приложения;
- выбор пользователем задания по меню заданий и передача его ЭВМ на выполнение;
- планирование процесса взаимодействия при выполнении задания;
- ввод пользователем данных, необходимых ЭВМ для выполнения задания;
- передача пользователю результатов выполнения задания и их оценка пользователем.

Обобщенная схема направляющего метода взаимодействия приведена на рис. 2. В соответствии с этой схемой пользователь должен выбрать в меню заданий свое задание и передать его ЭВМ на выполнение. Выбранное задание рассматривается в ЭВМ как цель пользователя. Достижение этой цели обеспечивается в ходе последующего диалога с пользователем, в котором инициатива взаимодействия принадлежит ЭВМ.

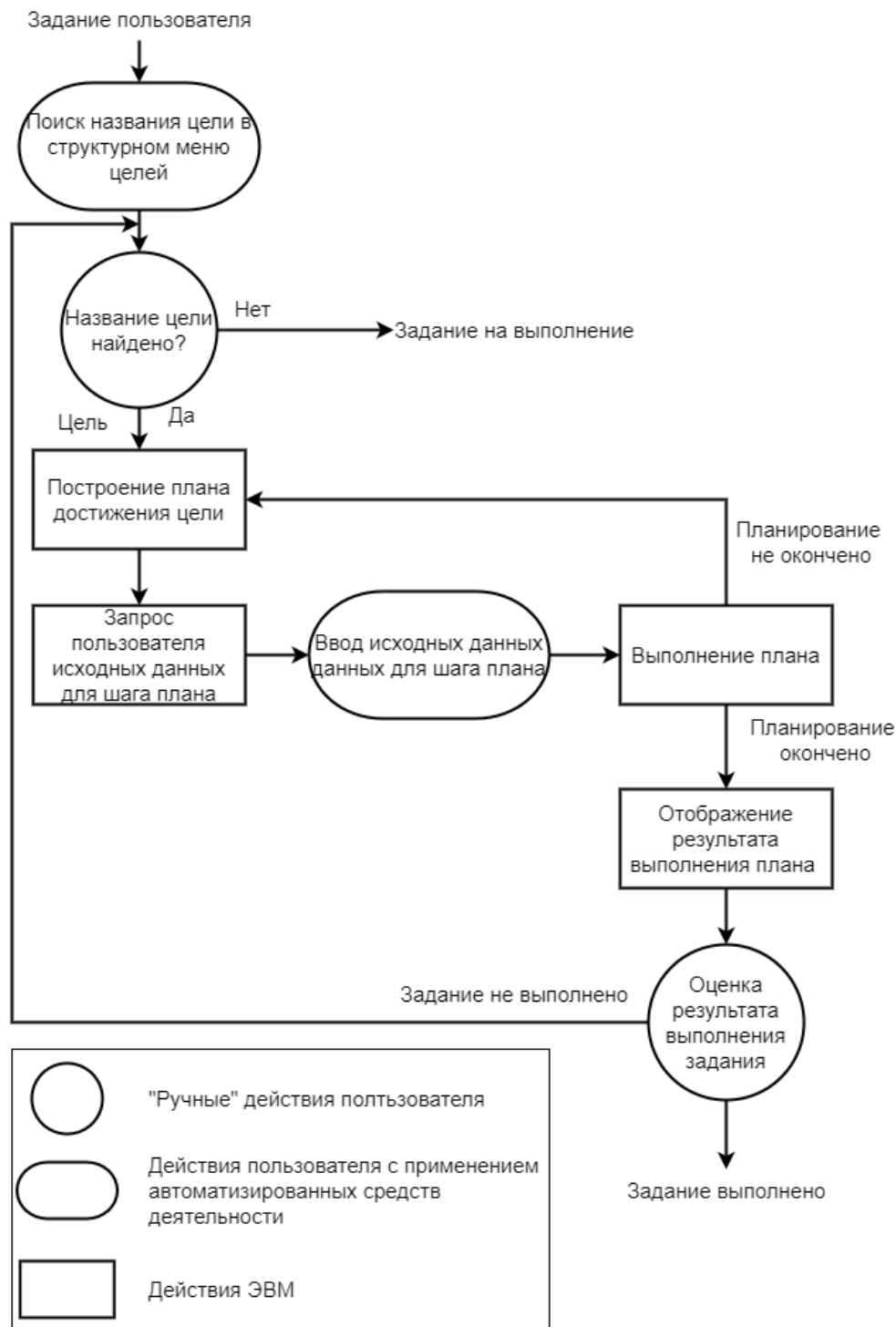


Рис. 2. Обобщённая схема направляющего метода взаимодействия “пользователь-ЭВМ” для оконного интерфейса.

В отличие от схемы ограничительного метода, в рассматриваемой схеме существует только один маршрут движения, т. е. здесь нет зависимости результата от степени пользовательской подготовки. После передачи пользовательской цели в ЭВМ дальнейшее взаимодействие не требует инициативы пользователя вплоть до этапа оценки результата достижения цели. Планирование процесса взаимодействия на множестве ДТ осуществляет ЭВМ в соответствии с ДТ-моделью диалога. Данные, необходимые ЭВМ для достижения принятой от пользователя цели, вводятся пользователем только по запросу, т.е. на этой стадии он выступает в роли реагирующего участника.

Итак, главное отличие направляющего метода взаимодействия “пользователь-ЭВМ” по сравнению с ограничительным методом состоит в том, что инициатива взаимодействия по достижению цели пользователя принадлежит не пользователю, а ЭВМ. Это означает, что ЭВМ играет активную роль в целенаправленном взаимодействии по выполнению задания пользователя.

Сопоставление схемы направляющего метода взаимодействия (рис. 1) со схемой ограничительного метода (рис. 2) позволяет наглядно судить об упрощении работы пользователя за счет того, что пользователю уже не требуется знать, как выполнить то или иное задание, поскольку планирование процесса взаимодействия выполняет не он сам, а ЭВМ.

Методический подход к созданию универсального пользовательского интерфейса

Была изучена статья “Методический подход к созданию универсального пользовательского интерфейса”

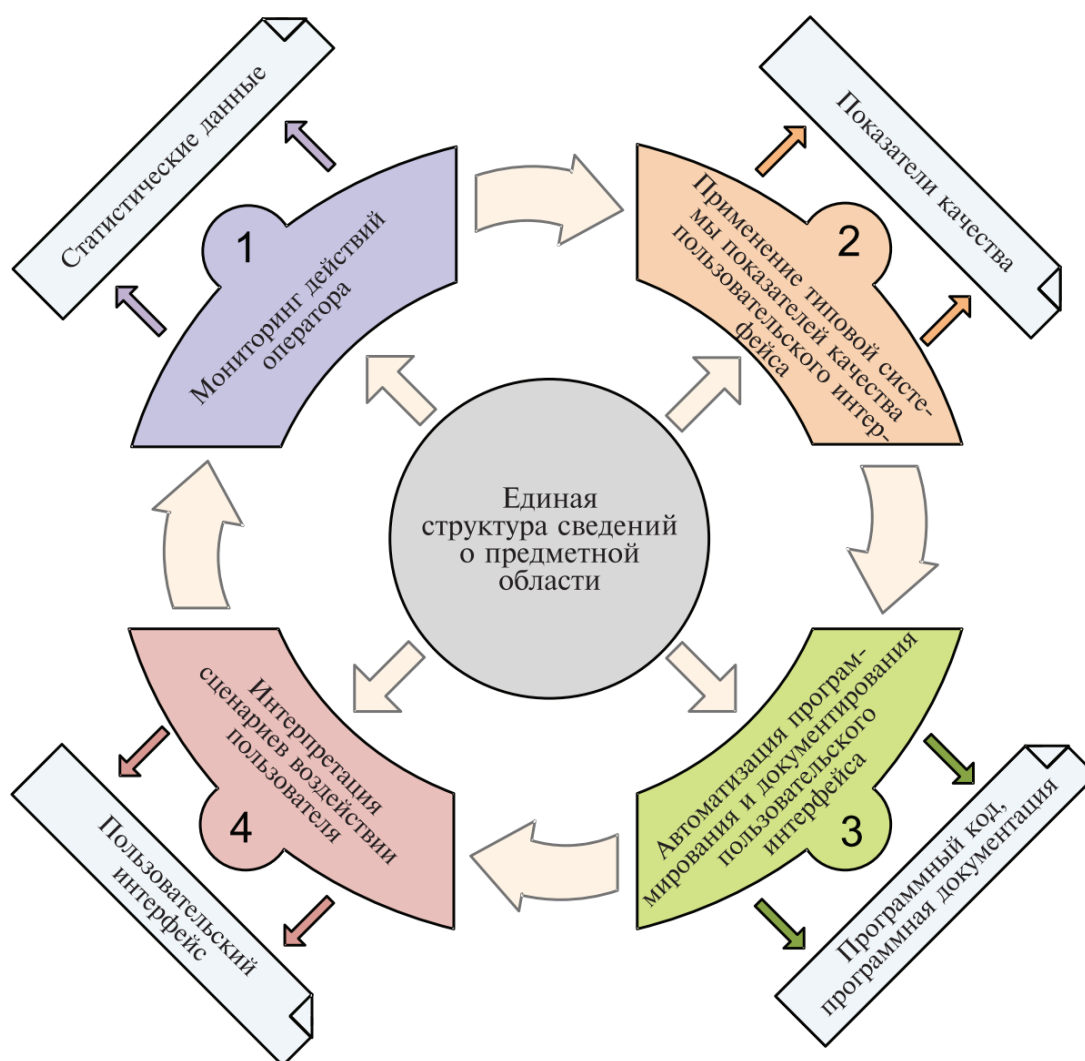


Рис. 3. Составные элементы подхода к созданию универсального средства построения пользовательского интерфейса программных средств

Мониторинг действий оператора (блок 1 на рис. 3) позволяет осуществлять сбор и накопление статистики деятельности оператора во время

эксплуатации программных средств. Оператор вводит входные параметры с помощью технических средств ввода данных: клавиатуры, манипулятора «мышь», фотокамеры, микрофона, сканера, специализированных панелей кнопок и переключателей, внешних носителей информации и т. п. Каждая атомарная операция основывается на низкоуровневых сигналах от средства ввода, преобразуемых программной средой в воздействия ввода оператора за счет событийной обработки.

Оценка качества пользовательского интерфейса обеспечивается за счет:

- применения типовой системы показателей качества (блок 2 на рис. 3);
- разработки методики оценки качества;
- выполнения мероприятий по оценке показателей качества.

Система показателей качества пользовательского интерфейса основывается на сведениях представленных типов:

- аппаратные события – события, которые возникают от действия всех технических средств при работе с периферийным оборудованием, а также вследствие использования каналов связи и удаленных подключений;
- события приложения – события, связанные с операциями получения, обработки и преобразования данных, обеспечения политики безопасности и уровней доступа, т. е. все эти события появляются в процессе функционирования программы в программной среде;
- пользовательские события – события, предусмотренные программой, происходят в результате воздействий пользователя на элементы управления интерфейса.

Величины, с помощью которых можно определить качество выполнения операции:

- среднее время выполнения операции;
- число ошибок (количество итераций), возникающих за период работы;
- среднее время возникновения ошибки.

Показатели качества пользовательского интерфейса выражаются в формате:

- время выполнения операций (действий);
- число действий, необходимых для выполнения операции;
- число атомарных операций, необходимых для выполнения действия;
- число пустых воздействий, выполненных оператором;
- число ошибочных атомарных операций.

“Автоматизация программирования и документирования пользовательского интерфейса” (блок 3 на рис. 3) подразумевает возможность автоматизированного документирования интерфейса программы.

Основные мероприятия жизненного цикла пользовательского интерфейса:

- задание требований;
- проектирование;
- разработка, программирование;

- оценка качества и испытания;
- документирование (описание).

Каждое мероприятие жизненного цикла интерфейса взаимосвязано с UML моделями, описывающими интерфейс, которые в ходе итерационного процесса разработки корректируются и используются для получения документов.

В соответствии с подходом предусматривается автоматизация программирования макетов пользовательского интерфейса, основанная на таком выборе паттерна проектирования программного кода, который позволит инкапсулировать механизмы обработки данных и управления от элементов управления интерфейса, но при этом обеспечит связь с моделью данных и мониторинг действий оператора.

Можно использовать UML модель сценариев применения пользовательского интерфейса (модель сценариев воздействий пользователя (СВП)), описывающая различные стороны функционирования программы, которая выражает на определенном уровне абстракции порядок взаимодействия пользователя с программным средством. Модель используется для декомпозиции и формирования однозначного понимания сведений по совокупности функций и режимов работы разрабатываемого программного средства, а также для обеспечения возможности автоматизированного документирования и построения кода интерфейса программы.

С помощью UML модели СВП решаются следующие задачи:

- прототипирование и/или макетирование пользовательского интерфейса при разработке программного средства;
- формализация существующего пользовательского интерфейса;
- описание деятельности пользователя при эксплуатации программных средств, выраженное в виде набора осуществленных сценариев воздействий на элементы управления программы.

Отображение некоторого абстрактного сценария осуществляет механизм его интерпретации (блок 4 на рис. 3) в стандартные программные процедуры, характерные для выбранной программноаппаратной платформы. Наличие интерпретатора предписывает применение некоторой формальной логики (языка), с помощью лексем которой выражаются любые сценарии. Одна лексема описывает типовую атомарную операцию над элементом управления пользовательского интерфейса, идентифицирующие сведения о которой содержатся в параметрах лексемы.

Каждая атомарная операция, которая вызвана воздействиями пользователя, на диаграмме СВП представляет собой дугу графа, вершины графа — элементы управления пользовательского интерфейса, веса графа — комплексные показатели, характеризующие:

- количество выполнений атомарной операции;
- время выполнения атомарной операции;
- количество ошибок, связанных с выполнением атомарной операции.

Построение пользовательского интерфейса с использованием интерактивного машинного обучения

На первом этапе производится сбор входных данных. В качестве таких данных будут выступать частота, последовательность, достигаемый результат и время между применениями рассматриваемых функций. Исследованием в данной области занимается человеко-компьютерное взаимодействие, где в настоящее время основную роль играет машинное обучение.

На основании собранных данных проводится обучение, целью которого является сократить путь для достижения конкретного результата, сокращение количества шагов и затрачиваемого времени для выполнения идентичных задач. Для обучения используется алгоритм дерева градиентного повышения.

Алгоритм обучения выбирает лучший порог для каждого. Использование матрицы Гессiana и весов позволяет вычислять прирост информации, вызванный применением каждой функции и правила принятия решения для узла.

Обучение может производиться на любом приложении с графическим пользовательским интерфейсом, имеющем длинные цепочки выполнения действий, например, пакет офисных приложений. По результатам обучения строится последовательность действий для достижения необходимого результата. При её построении учитывается время поиска элемента интерфейса, его доступность (подмножество действий необходимых к выполнению для получения доступа к данному элементу), соответствие описания и ожидаемого результат (использование других элементов, требующих большего числа шагов для достижения результата).

На основании полученных результатов вносятся корректировки в существующий интерфейс, после чего обучение продолжается.

Разработка тестового web-приложения

Django

Django — это высокоуровневый Python веб-фреймворк для бэкенда, который позволяет быстро создавать безопасные и поддерживаемые веб-сайты. Фреймворк — это программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Главная цель фреймворка Django — позволить разработчикам вместо того, чтобы снова и снова писать одни и те же части кода, сосредоточиться на тех частях своего приложения, которые являются новыми и уникальными для их проекта.

Достоинства Django:

- Масштабируемый. Django использует компонентную архитектуру, то есть каждая её часть независима от других и, следовательно, может быть заменена или изменена, если это необходимо. Чёткое разделение частей

означает, что Django может масштабироваться при увеличении трафика, путём добавления оборудования на любом уровне;

- Разносторонний. Django может быть использован для создания практически любого типа веб-сайтов;
- Безопасный. Django помогает разработчикам избежать многих распространённых ошибок безопасности, предоставляя фреймворк, разработанный чтобы «делать правильные вещи» для автоматической защиты сайта. Например, Django предоставляет безопасный способ управления учётными записями пользователей и паролями, избегая распространённых ошибок, таких как размещение информации о сессии в файлы cookie, где она уязвима или непосредственное хранение паролей вместо хэша пароля;
- Переносным. Django написан на Python, который работает на многих платформах;
- Удобным в сопровождении. Код Django написан с использованием принципов и шаблонов проектирования, которые поощряют создание поддерживаемого и повторно используемого кода.

Docker

Docker — программное обеспечение с открытым исходным кодом, применяемое для разработки, тестирования, доставки и запуска веб-приложений в средах с поддержкой контейнеризации. Он нужен для более эффективного использования системы и ресурсов, быстрого развертывания готовых программных продуктов, а также для их масштабирования и переноса в другие среды с гарантированным сохранением стабильной работы.

Основной принцип работы Docker — контейнеризация приложений. Этот тип виртуализации позволяет упаковывать программное обеспечение по изолированным средам — контейнерам. Каждый из этих виртуальных блоков содержит все нужные элементы для работы приложения. Это дает возможность одновременного запуска большого количества контейнеров на одном хосте.

Достоинства использования Docker:

- Минимальное потребление ресурсов — контейнеры не виртуализируют всю операционную систему (ОС), а используют ядро хоста и изолируют программу на уровне процесса;
- Скоростное развертывание — вспомогательные компоненты можно не устанавливать, а использовать уже готовые docker-образы (шаблоны);
- Удобное скрывание процессов — для каждого контейнера можно использовать разные методы обработки данных, скрывая фоновые процессы;
- Работа с небезопасным кодом — технология изоляции контейнеров позволяет запускать любой код без вреда для ОС;
- Простое масштабирование — любой проект можно расширить, внедрив новые контейнеры;

- Удобный запуск — приложение, находящееся внутри контейнера, можно запустить на любом docker-хосте;
- Оптимизация файловой системы — образ состоит из слоев, которые позволяют очень эффективно использовать файловую систему.

Определения Docker:

- Docker-образ (Docker-image) — файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера;
- Docker-контейнер (Docker-container) — это легкий, автономный исполняемый пакет программного обеспечения, который включает в себя все необходимое для запуска приложения: код, среду выполнения, системные инструменты, системные библиотеки и настройки;
- Docker-файл (Docker-file) — описание правил по сборке образа, в котором первая строка указывает на базовый образ. Последующие команды выполняют копирование файлов и установку программ для создания определенной среды для разработки;
- Docker-клиент (Docker-client / CLI) — интерфейс взаимодействия пользователя с Docker-демоном. Клиент и Демон — важнейшие компоненты “движка” Докера (Docker Engine). Клиент Docker может взаимодействовать с несколькими демонами;
- Docker-демон (Docker-daemon) — сервер контейнеров, входящий в состав программных средств Docker. Демон управляет Docker-объектами (сети, хранилища, образы и контейнеры). Демон также может связываться с другими демонами для управления сервисами Docker;
- Том (Volume) — эмуляция файловой системы для осуществления операций чтения и записи. Она создается автоматически с контейнером, поскольку некоторые приложения осуществляют сохранение данных;
- Реестр (Docker-registry) — зарезервированный сервер, используемый для хранения docker-образов;
- Docker-хаб (Docker-hub) или хранилище данных — репозиторий, предназначенный для хранения образов с различным программным обеспечением. Наличие готовых элементов влияет на скорость разработки;
- Docker-хост (Docker-host) — машинная среда для запуска контейнеров с программным обеспечением;
- Docker-сети (Docker-networks) — применяются для организации сетевого интерфейса между приложениями, развернутыми в контейнерах.

Разработка тестового web-приложения

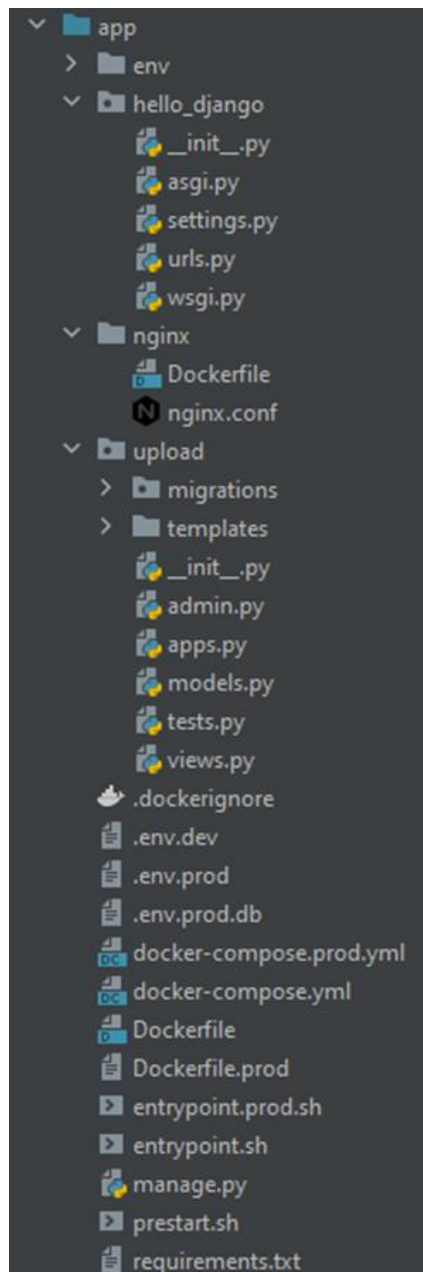


Рис. 4. Содержания проекта

Файл `settings.py` содержит в себе все настройки проекта. Здесь регистрируются приложения, задаётся размещение статичных файлов, настройки базы данных и т.д.

Файл `urls.py` задаёт ассоциации url адресов с представлениями.

`wsgi.py` используется для определения связи между Django приложением и веб-сервером.

`manage.py` используется для создания приложений, работы с базами данных и для запуска отладочного сервера.

В файле `Dockerfile.prod` описана последовательность команд, которые надо выполнить. На основе этого файла будет создан образ (docker image). В `Dockerfile.prod` используется многоступенчатая сборка (multi-stage build), чтобы уменьшить конечный размер образа. `builder` — это временный образ, которое используется для сборки Python. Затем он копируется в конечный производственный образ, а образ `builder` отбрасывается. Так же был создан пользователь `app` без полномочий `root`. По умолчанию Docker запускает

контейнерные процессы как root внутри контейнера, и если кто-то получит доступ на сервер и к контейнеру, то проникший на сервер пользователь тоже станет root, что, очевидно, не желательно. Таким образом увеличивается безопасность.

Вначале указан образ, на котором будем основываться. В нашем случае python 3.9. Устанавливается рабочая директория в контейнере с помощью WORKDIR. Далее устанавливаются переменные окружения:

- PYTHONDONTWRITEBYTECODE означает, что Python не будет пытаться создавать файлы .рус;
- PYTHONUNBUFFERED гарантирует, что вывод консоли выглядит знакомым и не буферизируется Docker.

Затем команды для установки соответствующих пакетов, необходимых для Psycopg2. Из директории, где находится Dockerfile.prod, копируется файл зависимостей requirements.txt в рабочую директорию контейнера. Далее с помощью команды RUN исполняются перечисленные в ней команды, что приводит к установкам зависимостей. Затем копируется вся дериктория проекта в контейнер.

Были созданы папки staticfiles и mediafiles, так как docker-compose монтирует именованные тома как root. И так как используется пользователь app не обладает полномочиями root, таким образом можно получить ошибку отказа в разрешении при запуске команды collectstatic, если каталог еще не существует.

Листинг 1. Dockerfile.prod

```
# BUILDER
FROM python:3.9.6-alpine as builder

# установка рабочей директории
WORKDIR /usr/src/app

# установка переменных окружения
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# установите зависимости psycopg2
RUN apk update \
    && apk add postgresql-dev gcc python3-dev musl-dev

RUN pip install --upgrade pip
RUN pip install flake8
COPY . .
#RUN flake8 --ignore=E501,F401 .

# установка зависимостей
COPY ./requirements.txt .
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /usr/src/app/wheels -r
requirements.txt

# FINAL

FROM python:3.9.6-alpine

# создание каталога для пользователя app
```

```

RUN mkdir -p /home/app

# создание пользователя app
RUN addgroup -S app && adduser -S app -G app

# создание необходимых директорий
ENV HOME=/home/app
ENV APP_HOME=/home/app/web
RUN mkdir $APP_HOME
RUN mkdir $APP_HOME/staticfiles
RUN mkdir $APP_HOME/mediafiles
WORKDIR $APP_HOME

# установка зависимостей
RUN apk update && apk add libpq
COPY --from=builder /usr/src/app/wheels /wheels
COPY --from=builder /usr/src/app/requirements.txt .
RUN pip install --no-cache /wheels/*

# копирование entrypoint.prod.sh
COPY ./entrypoint.prod.sh .
RUN sed -i 's/\r$//g' $APP_HOME/entrypoint.prod.sh
RUN chmod +x $APP_HOME/entrypoint.prod.sh

# копирование проекта в контейнер
COPY . $APP_HOME

# chown файлам пользователя
RUN chown -R app:app $APP_HOME

# переход к пользователю app
USER app

# запуск entrypoint.prod.sh
ENTRYPOINT ["/home/app/web/entrypoint.prod.sh"]

```

Был создан файл `docker-compose.prod.yml` (листинг 2). `docker-compose` позволяет управлять многоконтейнерностью. То есть можно запустить сразу несколько контейнеров, которые будут работать между собой. В нашем случае мы создаем контейнер с базой данных `db`, наш основной контейнер `web` и `nginx`.

Листинг 2. `docker-compose.prod.yml`

```

version: '3.8'

services:
  web:
    build:
      context: ./
      dockerfile: Dockerfile.prod
    command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8080
    volumes:
      - static_volume:/home/app/web/staticfiles
      - media_volume:/home/app/web/mediafiles
    env_file:
      - ./env.prod
    depends_on:
      - db
  db:
    image: postgres:13.0-alpine
    volumes:

```

```

    - postgres_data:/var/lib/postgresql/data/
  env_file:
    - ../env.prod.db
nginx:
  build: ./nginx
  volumes:
    - static_volume:/home/app/web/staticfiles
    - media_volume:/home/app/web/mediafiles
  ports:
    - 8084:80
    - 443:443
  depends_on:
    - web

volumes:
  postgres_data:
  static_volume:
  media_volume:

```

В самом начале указывается версия docker-compose. Далее указываются services (контейнеры).

- web. Сбор проекта (build) делается на основе Dockerfile. Далее указывается команда для запуска сервера. Указываются volumes, что значит, что всё из указанных директорий используется в контейнере. Далее указываются порты (сервер Django запускается в контейнере на порте 1337 и этот порт перебрасывается на нашу хост машину). И в конце сервиса web указывается зависимость от сервиса db, то есть web не сможет работать без db.
- db. Выбирается образ postgres. Далее указываются volumes, чтобы сохранять наши данные. Так же настраиваются переменные среды.
- nginx. Чтобы он действовал как обратный прокси-сервер для обработки клиентских запросов, а также для обслуживания статических файлов. Для работы Nginx была создана новая директория с соответствующим названием. В данной директории были созданы файлы Dockerfile (листинг 3) и nginx.conf (листинг 4).

Листинг 3. Dockerfile в директории nginx

```

FROM nginx:1.21-alpine
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
# удаляем настройки по умолчанию
# копируем наши настройки

```

Листинг 4. nginx.conf

```

server {
    listen 80;
    location /static/ {
        alias /home/app/web/staticfiles/;
    }
    location /media/ {
        alias /home/app/web/mediafiles/;
    }
    location / {
        proxy_pass http://web:8080;
        proxy_set_header Host $http_host;
    }
}

```



```

        proxy_set_header Connection "upgrade";
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Upgrade $http_upgrade;
        proxy_connect_timeout      600;
        proxy_send_timeout          600;
        proxy_read_timeout          600;
        send_timeout                 600;
    }
}

```

Команды, представленные на листинге 5, позволяют получить переменные окружения записанные в файл `.env.prod` (листинг 7) и записать их в переменные в файле проекта `settings.py`.

Листинг 5. Обновлённые переменные в `settings.py`

```

SECRET_KEY = os.environ.get("SECRET_KEY")

DEBUG = int(os.environ.get("DEBUG", default=0))

ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS").split(" ")

```

`SECRET_KEY` — это секретный ключ, который используется Django для поддержки безопасности сайта.

`DEBUG`. Включает подробные сообщения об ошибках, вместо стандартных HTTP статусов ответов. Должно быть изменено на `False` на сервере, так как эта информация очень много расскажет взломщикам.

`ALLOWED_HOSTS` — это список хостов/доменов, для которых может работать текущий сайт.

Так же в `setings.py` были обновлены настройки базы данных (листинг 6) на основе переменных окружения, определённых в `.env.prod`.

Листинг 6. Обновлённые переменные в `settings.py`

```

DATABASES = {
    "default": {
        "ENGINE": os.environ.get("SQL_ENGINE", "django.db.backends.sqlite3"),
        "NAME": os.environ.get("SQL_DATABASE", BASE_DIR / "db.sqlite3"),
        "USER": os.environ.get("SQL_USER", "user"),
        "PASSWORD": os.environ.get("SQL_PASSWORD", "password"),
        "HOST": os.environ.get("SQL_HOST", "localhost"),
        "PORT": os.environ.get("SQL_PORT", "5432"),
    }
}

```

Листинг 7. Содержимое файла `.env.prod`

```

DEBUG=0
SECRET_KEY=change_me
DJANGO_ALLOWED_HOSTS=localhost 195.19.40.68 [::1]
SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=hello_django_prod
SQL_USER=hello_django
SQL_PASSWORD=hello_django
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres

```

Для работы с медиафайлами был создан новый модуль Django под названием `upload`, то есть была создана новая директория `upload` (новый модуль создаётся командой в терминале: “`docker-compose exec web python manage.py startapp upload`”) и добавлен новый модуль в `INSTALLED_APPS` в `settings.py`.

В созданной директории был изменен файл `views.py` (листинг 9), была создана папка для шаблонов “`templates`” и в ней был создан новый шаблон `upload.html` (листинг 10). Так же был изменен файл `app/hello_django/urls.py` (листинг 8).

Листинг 8. `urls.py`

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
from upload.views import image_upload

urlpatterns = [
    path("", image_upload, name="upload"),
    path("admin/", admin.site.urls),
]

if bool(settings.DEBUG):
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Листинг 9. `views.py`

```
from django.shortcuts import render
from django.core.files.storage import FileSystemStorage

def image_upload(request):
    if request.method == "POST" and request.FILES["image_file"]:
        image_file = request.FILES["image_file"]
        fs = FileSystemStorage()
        filename = fs.save(image_file.name, image_file)
        image_url = fs.url(filename)
        print(image_url)
        return render(request, "upload.html", {
            "image_url": image_url
        })
    return render(request, "upload.html")
```

Листинг 10. `upload.html`

```
{% block content %}
<form action="{% url 'upload' %}" method="post" enctype="multipart/form-data">
    {% csrf_token %}
    <input type="file" name="image_file">
    <input type="submit" value="submit" />
</form>
{% if image_url %}
    <p>File uploaded at: <a href="{{ image_url }}">{{ image_url }}</a></p>
    
{% endif %}
{% endblock %}
```

Запуск web-приложения на тестовом сервере

После входа на сервер через консоль и запуска приложения, при переходе по `http://195.19.40.68:8084`, открывается страница (рис. 5). Можно выбрать файл (рис.6). И после нажатия на `submit` страница обновляется, и теперь на ней

присутствует ссылка на выбранное изображение, которое теперь сохранено в контейнере, и само изображение (рис. 7).

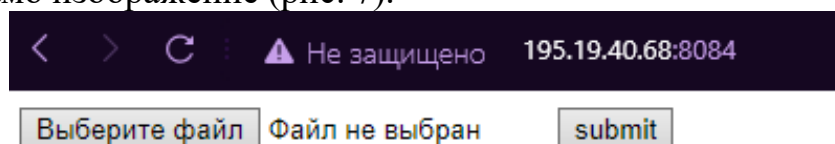


Рис. 5. Страница для загрузки фотографии

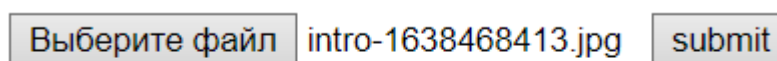



Рис. 6. Фотография выбрана



Файл загружен по ссылке: /media/IMG_20230123_174752_936.png



Рис. 7. Страница после нажатия на submit

Далее проверим работу базы данных. Входим в контейнер и создаём суперпользователя (рисунок 8).

```

PS D:\PyCharm\django-on-docker\app> docker-compose exec web sh
/usr/src/app # ls
Dockerfile          docker-compose-old.yml  entrypoint.prod.sh    hello_django          requirements.txt
Dockerfile.prod     docker-compose.prod.yml  entrypoint.sh         manage.py
catalog            docker-compose.yml      env                   nginx
/usr/src/app # python manage.py createsuperuser
Username (leave blank to use 'root'): Arthur
Email address:
Password:
Password (again):
Superuser created successfully.
/usr/src/app #

```

Рис.8 Создание суперпользователя

Переходим по адресу <http://195.19.40.68:8084/admin> и вводим имя и пароль суперпользователя. Вход выполнен (рисунок 9).

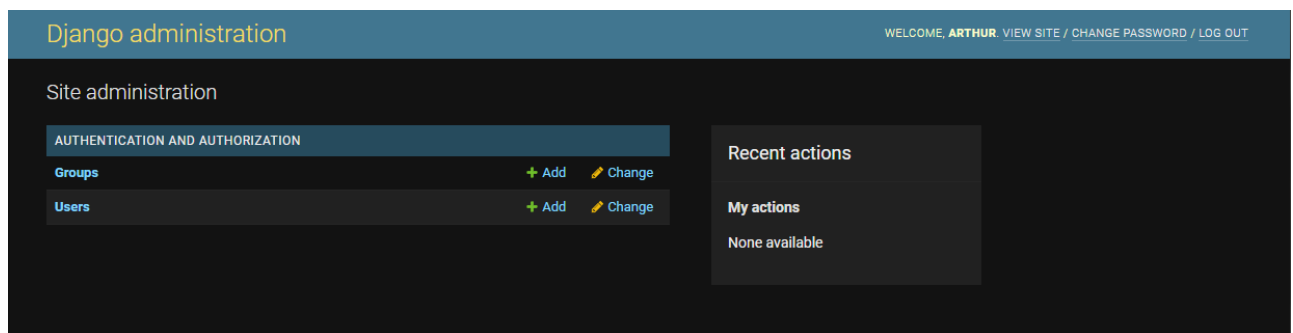


Рис. 9 Вход успешно выполнен

Проверка на создание таблиц по умолчанию на рисунке 10.

```

PS D:\PyCharm\django-on-docker\app> docker-compose exec db psql --username=hello_django --dbname=hello_django_dev
psql (13.0)
Type "help" for help.

hello_django_dev=# \l

               List of databases
  Name          | Owner          | Encoding | Collate   | Ctype      | Access privileges
-----+-----+-----+-----+-----+-----
hello_django_dev | hello_django   | UTF8     | en_US.utf8 | en_US.utf8 |
postgres        | hello_django   | UTF8     | en_US.utf8 | en_US.utf8 |
template0       | hello_django   | UTF8     | en_US.utf8 | en_US.utf8 | =c/hello_django +
                |                |          |            |            | hello_django=CTc/hello_django
template1       | hello_django   | UTF8     | en_US.utf8 | en_US.utf8 | =c/hello_django +
                |                |          |            |            | hello_django=CTc/hello_django
(4 rows)

```

Рис. 10. Проверка таблиц

Заключение

Были изучены существующие подходы разработки GUI, было первое теоретическое и практическое знакомство с Django и Docker, после чего было разработано несложное web-приложение.

Дальнейшие поставленные задачи:

- Изучение существующего web-приложения comwps (проект по разработке web-клиента для доступа к подсистемам РВС GCD и другим программным системам);

- Доработка библиотеки `ruscomsdk` (SDK для программных реализаций сложных вычислительных методов в рамках графоориентированной технологии GBSE) в части возможности генерации GUI;
- Интеграция разработки в состав web-приложения `comwps` и тестирование работоспособности созданных программных средств.

Список литературы

[1] Крехтунова Д., Ершов В., Муха В., Тришин И., Василян А. Р. Разработка систем инженерного анализа и ресурсоемкого ПО (`rndhps`): Научно-исследовательские заметки. / Под редакцией Соколова А.П. [Электронный ресурс] - Москва: 2021. - 85 с. URL: <https://arch.rk6.bmstu.ru> (облачный сервис кафедры РК6)