*Тема 4.2 Веб-фреймворки* (**4+4=8час)**

***Лекционное занятие 1 (1 час)***

При создании Web-проекта на Python встаёт вопрос выбора Фреймворка - программной платформы, облегчающей разработку и монтаж разных компонентов большого программного проекта. Опишем преимущества Фреймворков.

*Проста диагностики и отладки*. Помимо сокращения времени работы, фреймворки упрощают отладку и обслуживание программного обеспечения. Процесс отладки подразумевает пошаговое выполнение кода в поисках точки, в которой была допущена ошибка при написании программы. Некоторые имеют собственную систему тестирования кода, что позволяет программистам выполнять модульные тесты одновременно. Такой процесс дает возможность больше времени посвятить тестированию, а не исправлению ошибок.

*Повышенная эффективность кода*. Фреймворки способствуют повторному использованию кода, что, в свою очередь, повышает его эффективность. Чтобы не писать сложные структуры, содержащие сотни строк кода с нуля, можно обратиться к базе платформы. Используя такой метод разработчик получает код, в котором легко внести изменения и применить дополнительные функции. Также можно создать собственный код, чтобы использовать его в последующих проектах.

*Ускоренная разработка*. Фреймворки содержат базовые модули, удобный интерфейс, библиотеки, среду кодирования и другие инстррументы упрощения работы. Разработчики не заботитятся о сокрытии данных, управлении сессиями, обработке ошибок и аутентификации ‑ платформа справляется с большинством этих задач, что позволяет сразу начать писать код.

При выборе Фреймворка, желательно оценить размер и сложность проекта. Если разработка ‑ система с большим количеством функций, то нужно выбрать full-stack фреймворк, а для небольшого проекта хватит микрофреймворков.

Из наиболее популярных full-stack фреймворков для Python можно выделить: **Django**, **Pyramid**, **Web2py** и т.д. Отметим некоторые их особенности.

**Pyramid** – open-source фреймворк, который дает максимум возможностей разработчику с малыми затратами времени и ресурсов. Интересная возможность фреймворка – работа как с большими, так и с малыми приложениями. Отдельно стоит выделить такие функции Pyramid, как: однофайловые приложения, генерация URL, масштабируемая конфигурация, гибкость аутентификации и авторизации, простая для понимания техническая документация.

**Web2py** поставляется с собственным IDEwhich: есть редактор кода, дебаггер и деплой. Стоит отметить отсутствие необходимости в установке и настройке, работу в Windows, Mac, Google App Engine, Amazon EC2 и любом хостинге с поддержкой Java+Python или Python 2.5–2.7, поддержку разных протоколов, безопасность данных, трекер ошибок, обратную совместимость, позволяя легко работать с приложениями на основе прежних версий фреймворка.

**TurboGears** – open-source full-stack фреймворк веб-приложений, позволяет разработчику быстро создавать расширяемые веб-приложения, управляемые данными. TurboGears поставляется с удобными шаблонами и мощным и гибким ORM. Выделим наиболее важные функции фреймворка: поддержка различных баз данных, поддержка SQLObject и SQLAlchemy, валидация для FormEncode, Pylons как веб-сервер, инструменты командной строки.

Среди микрофрейворков отметим **Flask**, **Bottle**, **CherryPy**.

**Flask** – предоставляется по лицензии BSD. Его разработчики развили идеи фреймворка Sinatra Ruby. Он зависит от инструментария Werkzeug WSGI и шаблона Jinja2. У Flask модульный дизайн, позволяющий подстроить его под разные задачи. «Из коробки» разработчик получает следующие функции:

|  |  |
| --- | --- |
| * встроенный сервер и дебаггер; * шаблоны Jinja2; * поддержка безопасных кукис; | * WSGI 1.0; * Unicode; * возможность подключения к любой ORM |

Создавался фреймворк для поддержки разработчиков веб-приложений, которые получили возможность выбирать расширения исходя из своих задач.

**Bottle** – микрофреймворк, изначально предназначенный для создания API. Bottle отличает единственный source-файл. Его функции по умолчанию включают маршрутизацию, шаблоны, утилиты и базовую абстракцию в стандарте WSGI:

* *маршрутизация* поддерживает запросы на отображение функциональных вызовов, позволяя вам получать чистые и динамические URL-адреса.
* *шаблоны* — много нужных инструментов, работающих «из коробки» вместе с полной поддержкой mako, jinja2 и cheetah.
* *утилиты* — удобный доступ к данным, загрузка файлов, куки, заголовки и другие метаданные, связанные с HTTP.
* *встроенный сервер разработки HTTP*, который поддерживает функции fapws3, bjoern, GAE, CherryPy и любой другой HTTP-сервер с WSGI.

Это отличное решение для прототипирования, изучения структуры веб-фреймворков и создания небольших приложений.

**CherryPy** – минималистичный веб-фреймворк с открытым исходным кодом. Он обеспечивает возможность создания веб-приложений Python, которое ничем не отличается от построения любой другой объектно-ориентированной программы. Web приложение, работающее под CherryPy ‑ приложение Python, использующее свой многопоточный веб-сервер. Приложения CherryPy запускаются в любой операционной системе с поддержкой Python (Windows, MacOS, Linux и т.д.).

В CherryPy применимы любые технологии шаблонизации, доступа к данным. Он может обрабатывать статьи, статику, куки, загружать файлы и все остальное, что умеют другие фреймворки. Его отличительные способности:

* простой запуск нескольких HTTP-серверов одновременно;
* мощная конфигурационная система и гибкая система плагинов;
* возможности: кэширование, декодинг, сессии, аутентификация и т.п.;
* возможность работы под Python 2.7+, Python 3.1+, PyPy, Jython и Android.

**Django** – open-source full-stack фреймворк, где можно добавить большинство стандартных функций единым пакетом вместо поиска отдельных библиотек. Среди них – такие востребованные, как аутентификация, URL-маршрутизация, миграция схемы данных и т.п. Django использует ORM для сопоставления объектов с таблицами баз данных. Один и тот же код работает с разными базами данных, так что переход из одной БД в другую становится простой задачей. Основными базами, с которыми работает Django, являются PostgreSQL, MySQL, SQLite и Oracle. Можно добавить и другие, но понадобятся решения сторонних разработчиков.

Базовая структура фреймворка Django включает следующие компоненты:

* *URL-маршрутизаторы*, перенаправляющие HTTP-запрос от браузера или любого иного веб-клиента в представления;
* *Представление*, которое обрабатывает запрос, обращается к модели и сообщает ей, какие именно данные из БД нужно задействовать, чтобы удовлетворить запрос;
* *Модель (менеджер базы данных, ORM)*, «вытаскивающую» нужную информацию из БД и передающую ее представлению;
* *HTML-шаблоны*, которые используются представлением для демонстрации пользователю полученных от модели данных.

Лучше установить Django в виртуальном окружении – virtualenv или pipenv, где проект будет изолирован от других. Большинство приведённых ниже команд указаны для работы в виртуальной среде. Рассмотрим пример с virtualenv:

Установить virtualenv можно двумя способами:

1. При создании проекта в IDE PyCharm 2022.3.2 (Community Edition)

2. При создании проекта с помощью терминала.

*Первый способ:* Создадим каталог project\_django в PyCharm, подключим virtualenv:

|  |  |
| --- | --- |
| **Рис1**. PyCharm: создание нового проекта. | **Рис.2**. Структура проекта. |

*Второй способ:* Создадим вручную каталог project\_django и перейдём в него.

Для установки virtualenv введем в консоли/терминале (Linux, Windows, macOS):

|  |  |
| --- | --- |
| *pip3 install virtualenv*  **Рис.3**. Скриншот установки |  |

Создастся каталог venv/ Python-окружения. Python-библиотеки, установленные при активированном окружении, будет сохраняться в папку venv/

Для активации окружения введем в консоли команду (в разных OS):

|  |  |
| --- | --- |
| *source vproject /bin/activate (linux, macOS)* | *vproject\Scripts\activate.bat (Windows)* |
| **Рис.4**. Терминал с активированным *venv* |  |

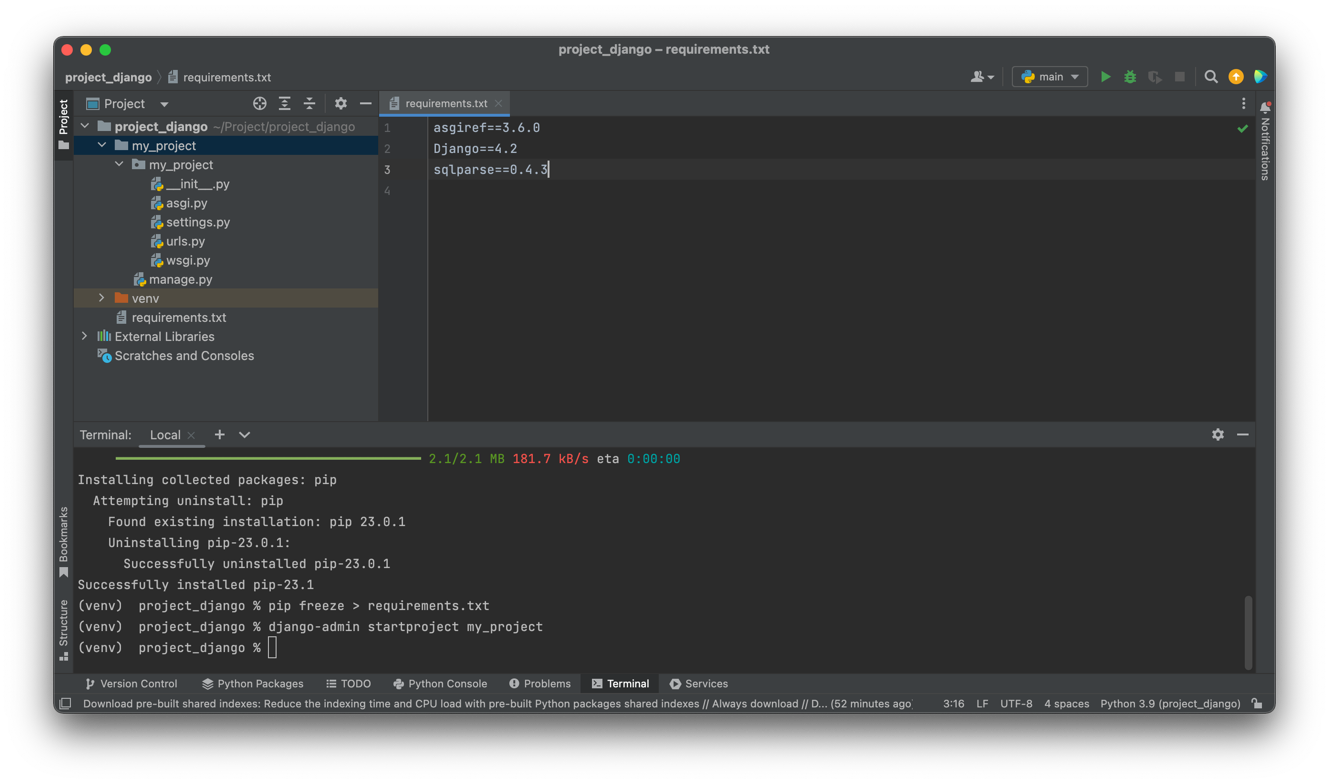
Для деактивации виртуального окружения, выполните команду *deactivate*.

1. После активации среды, установим Django 4.2: *pip install Django*

Темерь можно приступить к создаванию приложений.

|  |  |  |
| --- | --- | --- |
| **Рис.5**. Установка в терминале:  Сохраним связи в файл requirements.txt:  *pip freeze > requirements.txt* | |  |
| **Рис.6**. Интерфейс структуры проекта: |  | |

1. В папке проекта появился файл с зависимостями. Перейдем к созданию проекта Django следующей командой: *django-admin startproject my\_project*



**Рис.7**. Интерфейс PyCharm общий вид.

Django создаст проект, и в нашем каталоге появится папка *my\_project*

1. Для запуска локального сервера, введем: *python manage.py runserver*

***Практическое занятие 1 (1 час)***

1. Откроем PyCharm 2022.3.2 (Community Edition).
2. Настроим среду: установим *virtualenv* создания изолированного окружения.

Установить virtualenv можно двумя способами:

a. При создании проекта в PyCharm 2022.3.2 (Community Edition)

b. При создании проекта в терминале, или просто в командной строке.

|  |  |  |  |
| --- | --- | --- | --- |
| **Рис. 8**. Интерфейс PyCharm создание нового проекта:  Ниже показана его структура: |  | | |
| **Рис. 9**. Интерфейс PyCharm структуры проекта:  Для установки в терминале (Linux, Windows, macOS) введем команду: *pip3 install virtualenv* | | |  |
| **Рис. 10**. Интерфейс при установке: | |  | |

Создается каталог venv/ для Python-окружения. Любая Python-библиотека, установленная при активированном окружении, будет сохраняться в папку venv/

Для активации окружения введем команду: *vproject\Scripts\activate.bat (Windows)*

или *source vproject /bin/activate (linux, macOS).*

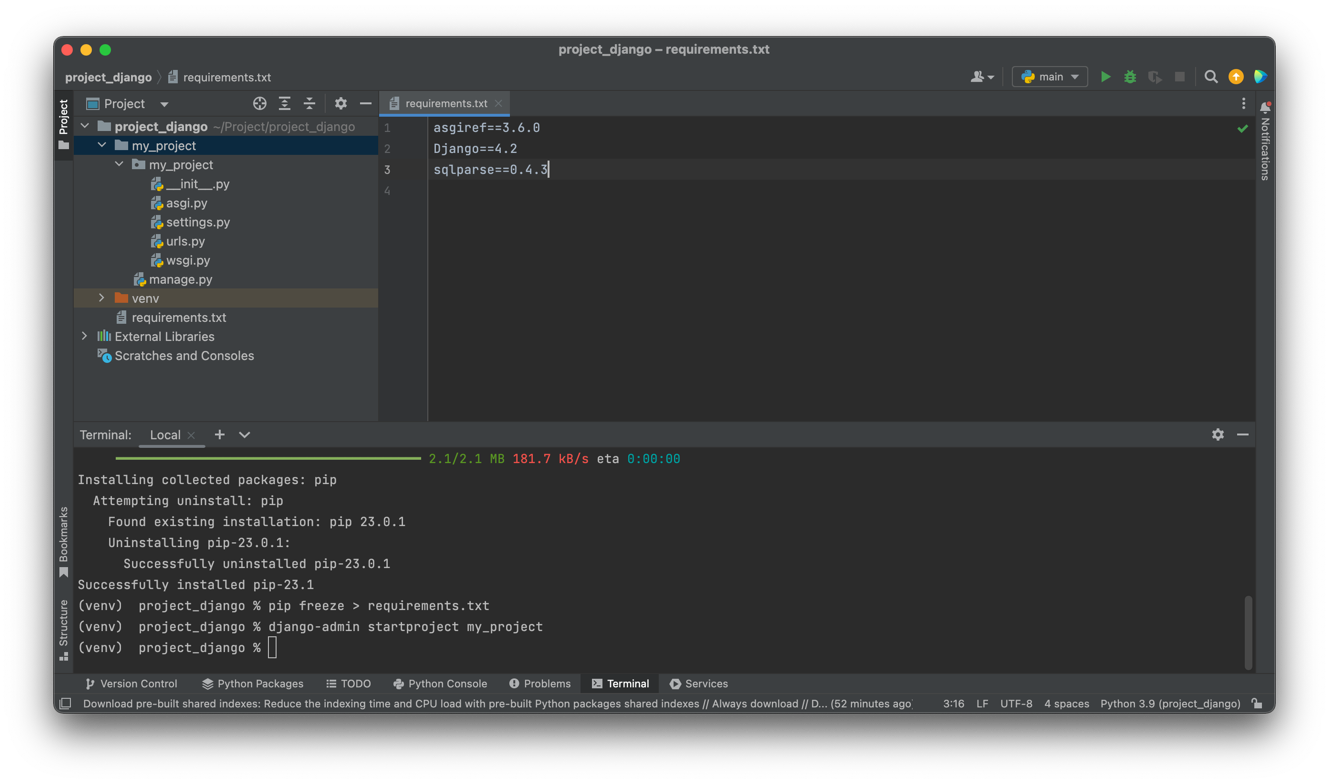
|  |  |
| --- | --- |
| **Рис. 11**. Окно терминала с активированным окружением *venv* |  |

Деактивация виртуального окружения выполняется командой *deactivate*.

1. Установим Django 4.2: *pip install Django.* Перейдем к созданию приложения.

|  |  |
| --- | --- |
| **Рис. 12**. Установка Django в терминале  Сохраним связи в файл requirements.txt:  *pip freeze > requirements.txt* |  |
| **Рис. 13**. Структура проекта в PyCharm |  |

1. В папке проекта появился файл с зависимостями. Перейдем к созданию проекта Django (пример-в терминале): *django-admin startproject my\_project*



**Рис. 14**. Интерфейс PyCharm общий вид

Django создаст проект, и в нашем каталоге появится папка *my\_project*

1. Для запуска локального сервера, введем: *python manage.py runserver*

Перейдя в браузере на адрес 127.0.0.1:8000, видим следующее:

|  |  |
| --- | --- |
| **Рис. 15**. Начальная страница Django | **Рис. 16**. Результат HTTP запросов |

***Лекционное занятие 2 (1 час)***

Python Django ‑ фреймворк быстрого создания веб-страниц. Его называют «фреймворк с батарейками», поскольку имеет встроенные функции, в частности *интерфейс администратора*, *базу данных* по умолчанию – SQLite3 и т.д. В результате при создании веб-сайта, где нужны: 1)аутентификация пользователя, 2)панель управления веб-сайта, 3)формы, 4)загрузка файлов и т.д., то Django предоставляет для этого пользователю готовые инструменты.

Django разработан в 2003 - 2005 годах командой, создающей газетные веб-сайты. Создав несколько, они стали повторно использовать коды и шаблоны, что в июле 2005 г. эволюционировло в фреймворк «Django» с открытым кодом.

Еще раз укажем основные достоинства данного фреймворка.

* Есть подробная документация и высокая масштабируемость.
* Используется ведущими разработчиками: Instagram, Disqus, Spotify, Youtube…
* Один из самых простых для изучения, есть «батарейки» в комплекте. Это платформа быстрой веб-разработки даже полнофункциональных веб-приложений.
* Django работает в Python с библиотечными функциями веб-очистки, машинного обучения, обработки изображений и т.д., легко интегрируемых в веб-приложения!

|  |  |
| --- | --- |
| **Рис. 17**. Архитектура Django:  Django основан на архитектуре MVT и состоит из следующих частей: |  |

* **Модель**: интерфейс данных, отвечающий за их хранение - логическая структура, являющаяся основой приложения и представленная базой данных (обычно реляционной, такой как MySQL, Postgres).
* **Вид**: интерфейс, видимый в браузере, представлен HTML/CSS/Javascript и Jinja.
* **Шаблон**: состоит из статических частей HTML и специального синтаксиса, описывающего, как будет вставлено динамическое содержимое.

Опишем структуру более подробно.

*Определение данных модели (models.py)*.

Веб-приложения Django обрабатывают данные через объекты Python, называемые моделями. Модели определяют структуру хранимых данных, включая типы полей и, возможно, их максимальный размер, значения по умолчанию, параметры списка выбора, текст справки для документации, текст меток для форм и т. д. Описание модели не зависит от используемой базы данных – они работают в любой из них. После выбора нужной базы данных, не следует напрямую обращаться к ней – программист просто пишет структуру модели и другой код, а Django выполняет всю работу по обращению к базе данных за программиста.

*Запросы данных (views.py).*

Модель Django предоставляет простой набор API запросов для поиска в базе данных. Возможен поиск по нескольким полям, используя различные критерии (такие как exact («точный»), case-insensitive («без учёта регистра»), greater than («больше чем») и т.д.), и может поддерживать сложные выражения (например, программист может задать поиск в U11, где начало команды «Fr», а конец «al»).

*Вывод данных (HTML-шаблоны).*

Системы шаблонов позволяют указать структуру выходного документа, используя заполнители для данных, вставляемых при генерировании страницы. Шаблоны часто используют для создания HTML, но также могут создавать другие типы документов. Django «из коробки» поддерживает как собственную систему шаблонов, так и другую популярную библиотеку Python под названием Jinja2 (она может быть использована для поддержки других систем, если это необходимо).

Напомним, что Django предоставляет также:

* **Формы**: HTML, используются при сборе данных для обработки на сервере. Django упрощает создание, проверку и обработку формы.
* **Аутентификация пользователя и разрешения**: Django включает надёжную систему аутентификации и авторизации пользователей с учётом безопасности.
* **Кеширование**: выполнение динамического контента выполняется намного чаще (и медленнее!), чем со статического. Django обеспечивает гибкое кеширование частей отображаемой страницы, ускоряя повторное обращение к ним.
* **Админ-панель**: панель администрирования в Django включена по умолчанию при создании приложения уже в основном блоке, упрощая работу администратору сайта для создания, редактирования и просмотра любых данных сайта.
* **Сериализация данных (преобразование в последовательную форму)**: Django упрощает сериализацию и обслуживание данных в форматах XML или JSON. Это полезно при создании веб-сервисов (сайтов для отображения данных сторонними приложениями/сайтами, сами ничего не отображая) или при создании сайта, где клиентский код обрабатывает весь поток данных.

**Базовая структура Django**

Проект с использованием Django может содержать несколько приложений. На первом этапе необходимо выделить понятия «проект» и «приложение»:

• **Приложение:** модуль проекта. Его можно связать с набором функций или же группой пользователей. Например, если создавать соцсеть, то следует разработать отдельно приложения пользователей, владельцев групп и модераторов контента.

• **Проект:** вся программа, состоящая из одного или нескольких приложений.

Разделение функций проекта по разным приложениям позволяет использовать один код в разных проектах, быстро добавляя нужную функциональность.

**Создание приложения Django**

Продолжая работать в каталоге проекта project\_django, создадим приложение hello, для чего в терминале введем команду: *python manage.py startapp hello*

|  |  |
| --- | --- |
| Django создаст новый каталог приложения, а дерево структуры проекта будет выглядеть как на **Рис. 18**.:  Каталог приложения hello содержит файлы:  \_\_init\_\_.py – для обработки каталога как пакета Python;  admin.py – настройки для страниц администратора Django;  apps.py – настройки для конфигурации приложения;  models.py – серия классов, преобразуемых в базы данных;  tests.py – классы для тестирования;  views.py – инструменты выбора данных для отображения. |  |

После установки hello, необходимо прописать его в настройках проекта, для чего в settings.py добавим следующий код в INSTALLED\_APPS:

*INSTALLED\_APPS = [*

*'django.contrib.admin', 'django.contrib.auth',*

*'django.contrib.contenttypes', 'django.contrib.sessions',*

*'django.contrib.messages', 'django.contrib.staticfiles',*

*'hello',*

*]*

Проект теперь знает, что созданное приложение hello существует.

Для отображения в браузере пропишем код главной страницы в формате html.

Зададим представления и код html в файле views.py. Представления в Django – набор функций/классов в views.py. Каждая функция/класс содержит логику, обрабатываемую при каждом посещении другого URL.

В каталоге приложения hello откроем views.py, содержащий следующий код:

*from django.shortcuts import HttpResponse*

*def hello(request):*

*return HttpResponse("""<!DOCTYPE html>*

*<html lang="en">*

*<head>*

*<meta charset="UTF-8">*

*<title> web-приложение </title>*

*</head>*

*<body>*

*<h1>Моё первое web-приложение</h1>*

*</body>*

*</html>""")*

Функция представления принимает один аргумент request. Это создаст HttpRequestObject при загрузке страницы, содержащий информацию о запросе - метод, который может принимать несколько значений, включая GET и POST.

Теперь пропишем маршруты. Откроем urls.py в папке my\_project и добавим код:

*from django.contrib import admin*

*from django.urls import path, include*

*urlpatterns = [*

*path('admin/', admin.site.urls),*

*path('', include('hello.urls')),*

*]*

При переходе на адрес нашего приложения, модуль вызовет urls.py в hello, и зарегистрирует любые URL-адреса, определенные в файле hello/urls.py.

Последним этапом создадим urls.py в папке *hello,* содержащий следующий код:

*from django.urls import path*

*from hello import views*

*urlpatterns = [*

*path('', views.hello, name='hello'),*

*]*

В терминале/консоли запустим локальный сервер: *python manage.py runserver*

|  |  |
| --- | --- |
| Введя в браузере 127.0.0.1:8000, увидим:  **Рис. 19**. Страница приложения hello |  |

***Практическое занятие 2 (1 час)***

Выполняем все дествия как в разделе «Создание приложения Django» лекции.

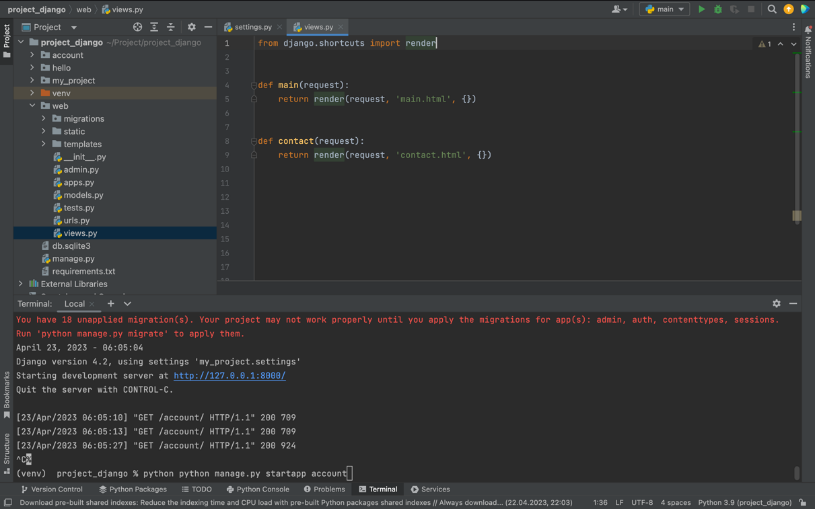
***Лекционное занятие 3 (1 час)***

Вначале создается базовая структура приложения, функции представления и маршруты, далее подключаются шаблоны и контент.

|  |  |  |
| --- | --- | --- |
| Продолжая работать в папке project\_django, создадим приложение web: *python manage.py startapp web*  Django создаст каталог приложения, а структура проекта соответствует **Рис.20**:  Важно помнить: название приложения не должно совпадать с папкой проекта, иначе возникнет ошибка:  *CommandError: '%app%' conflicts with the name of an existing Python module and cannot be used as an app name. Please try another name.*  Например, если папку назвали my\_site, приложение должно носить другое название – например blog. | |  |
| После того как приложение web установлено, пропишем его в настройках проекта Django, для чего в settings.py добавим строку кода под INSTALLED\_APPS:  Далее откроем web\view.py, где располагаются функции представления, которые обеспечивают функционирование логики приложения. | *INSTALLED\_APPS = [*  *'django.contrib.admin',*  *'django.contrib.auth',*  *'django.contrib.contenttypes',*  *'django.contrib.sessions',*  *'django.contrib.messages',*  *'django.contrib.staticfiles',*  *'hello',*  *'web',*  *]* | |

Представление – это «тип» веб-страницы в приложении Django, который обычно выполняет определенную функцию и имеет определенный шаблон.

В Django веб-страницы и другой контент доставляются с помощью представлений. Каждое представление - функция Python (или метод, в случае представлений на основе классов). Django выберет представление, проверив запрошенный URL-адрес (точнее, часть URL-адреса после имени домена).



**Рис. 21**. Файл представления view.py

В файле уже есть одна строка: from django.shortcuts import render.

|  |  |
| --- | --- |
| Добавим обработку https и функции представления для страниц, например, Главная страница, Контакты и т.д, после чего содержимое примет вид: | from django.shortcuts import render  def main(request):  return render(request, 'main.html', {})  def contact(request):  return render(request, 'contact.html', {}) |

Данный пример является тестовым, в дальнейшем его можно изменять!

|  |  |
| --- | --- |
| Далее в папке web создаем urls.py для сопоставления маршрутов и страниц, добавив в него код: | from django.urls import path  from web import views  urlpatterns = [  path('', views.main, name='main'),  path('main/', views.contact, name='main'),  path('contact/', views.contact, name='contact'),  ] |
| После этого добавим в urls.py, находящемся в папке проекта my\_project, импорт include, а в urlpatterns – путь к ссылкам приложения web: | from django.contrib import admin  from django.urls import path, include  urlpatterns = [  path('admin/', admin.site.urls),  path('', include('hello.urls')),  path('web/', include('web.urls')),  ] |

Строка path('web/', include('web.urls')) указывает, что web лишь раздел портала.

|  |  |
| --- | --- |
| Каждую страницу приложения можно создавать вручную. Первый вариант –решать эту задачу «в лоб», тогда можно было бы написать в функции:  Но такой вариант не удобен и не практичен. | def index(request):  return HttpResponse('''<!DOCTYPE html>  <html>  <head>  <title></title>  </head>  <body>  </body>  </html>''') |

Каждый веб-фреймворк нуждается в удобном способе генерации файлов HTML. В Django за это отвечают шаблоны ‑ файлы HTML, связанные между собой и содержащие базовые операции. Шаблоны в Django могут наследоваться и их блоки могут переопределяться. Проект Django можно настроить с несколькими шаблонизаторами (или без них, где шаблоны не нужны). Django предоставляет встроенные серверные модули для своей системы шаблонов, названной языком шаблонов Django (DTL). Серверные программы для других языков шаблонов могут быть доступны у третьих лиц. Можно написать и свой бэкэнд (программную логику для обеспечения работы приложений), но требуется высокий уровень знания математики, баз данных и пр.

Django определяет стандартный API для загрузки и рендеринга шаблонов независимо от серверной части. Загрузка состоит из поиска шаблона для данного идентификатора и его предварительной обработки, обычно компилируя его в образ в памяти. *Рендеринг означает интерполяцию шаблона с помощью контекстных данных и возврат результирующей строки.*

Как было сказано, нужно импортировать встроенный в Django шаблонизатор:

**from** django.shortcuts **import** render

Здесь функция render отвечает за обработку шаблонов и выдачу их в формате готового HTTP клиенту. Можно прописать эту функцию в представлении index. В самом простом варианте это будет выглядеть так:

def index(request):

return render(request, '<путь к шаблону>')

Здесь первый параметр обязательно должен быть ссылкой request, а второй – путь к файлу шаблона. По умолчанию Django ищет шаблоны в папке templates.

Но есть одна проблема: *при сборке всего проекта все шаблоны от всех приложений помещаются в единую папку templates проекта и если окажется в разных приложениях два одинаковых файла (с одинаковыми именами), то возникнет неопределенность (будет взят первый попавшийся файл).* Чтобы этого не произошло, в templates приложения принято создавать еще один подкаталог с именем приложения, и уже в него помещать шаблоны. Тогда при сборке подкаталог целиком перенесется в каталог templates проекта и коллизий имен не произойдет.

И, далее, в функции render указывается путь к этому шаблону, например:

def index(request): return render(request, 'my\_site/index.html')

Шаблон Django представляют собой текстовый документ или строку Python, размеченную в языке шаблонов Django. Некоторые конструкции распознаются и интерпретируются механизмом шаблонов. Основные из них – переменные и теги.

Шаблон выводится с контекстом. Визуализация заменяет переменные значениями, видимыми в контексте, и выполняет теги. Остальное выводится как есть.

Синтаксис языка шаблонов Django включает четыре конструкции.

* ***Переменные***, выводят значение из контекста, представляющего собой dict-подобный объект, отображающий ключи к значениям, размещенный между {{ и }}.
* ***Тэги***, которые обеспечивают произвольную логику в процессе рендеринга. Это определение заведомо расплывчатое. Например, тег может выводить контент, служить структурой управления, например. оператор «if» или цикл «for», захват содержимого из базы данных или даже разрешение доступа к другим тегам шаблона. Теги заключаются в символы {% и %}.

|  |  |  |
| --- | --- | --- |
| ***Фильтры*** – преобразование значений переменных или аргументов тегов. Пример: | def cut(value, arg):  """Удаляем все значения аргумента из строки"""  return value.replace(arg, '') | |
| А вот пример использования этого фильтра: | | {{ somevariable|cut:"0" }} |

* ***Комментарии,*** которые содержат невыводимый текст между {# и #}

***Практическое занятие 3 (1 час)***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. Перейдем в каталог с проектом project\_django. 2. Создадим приложение web: *python manage.py startapp web*   Django создаст новый каталог приложения, структура проекта будет выглядеть как на **Рис.22**: | | | |  |
| Откроем файл settings.py и добавим строки под INSTALLED\_APPS:   1. В каталоге нашего приложения web создадим новую папку **templates**, где и будут хранится шаблоны страниц нашего приложения. | | | *INSTALLED\_APPS = [*  *'django.contrib.admin',*  *'django.contrib.auth',*  *'django.contrib.contenttypes',*  *'django.contrib.sessions',*  *'django.contrib.messages',*  *'django.contrib.staticfiles', 'hello', 'web',]* | |
| 1. Создадим в **templates** файл **main.html** со следующим кодом (много лишних строк, которые при файла вручную, не пишут): | | *<!DOCTYPE html><html lang="en">*  *<head> <meta charset="UTF-8">*  *<title>Главная страница</title></head>*  *<body><h1>Главная страница</h1></body>*  *</html>* | | |
| 1. Создадим в **templates** файл **contact.html** с кодом (здесь убрали «лишние» инструкции): | | *<html>*  *<head><title>Контакты</title></head>*  *<body><h1>Контакты</h1></body> /html>* | | |
| 1. Зададим представления в views.py, то есть в каталоге web создадим **views.py** со следующим кодом: | | from django.shortcuts import render  def main(request):  return render(request, 'main.html', {})  def contact(request):  return render(request, 'contact.html', {}) | | |
| 1. Создадим файл **urls.py** в **my\_project**, добавив код: | | from django.contrib import admin  from django.urls import path, include  urlpatterns = [ path('admin/', admin.site.urls),  path('', include('hello.urls')),  path('web/', include('web.urls')), ] | | |
| 1. Создадим файл urls.py в папке web, добавив код: | from django.urls import path  from web import views  urlpatterns = [ path('main/', views.main, name='main'),  path('contact/', views.contact, name='contact'), ] | | | |

1. Запустим наш локальный сервер из терминала: *python manage.py runserver*

|  |  |
| --- | --- |
| В браузере на адресах 127.0.0.1:8000/web/main и 127.0.0.1:8000/web/contact увидим (**Рис.23**):  На рисунке: сверху – вывод по запросу страницы **main**, снизу – страницы **contact** |  |
|  |

***Лекционное занятие 4 (1 час)***

Веб-сайты обычно нуждаются в дополнительных файлах, таких как изображения, JavaScript или CSS. Работа с файлами должна быть построена так, чтобы это не снижало производительность приложения. В Django файлы называют «статическими» и есть инструмент для работы с ними: *django.contrib.staticfiles*.

|  |  |  |
| --- | --- | --- |
| Включим *django.contrib.staticfiles* в INSTALLED\_APPS и зададим в settings.py *STATIC\_URL = '/static/'* | | *INSTALLED\_APPS = [ 'django.contrib.admin',*  *'django.contrib.auth', 'django.contrib.sessions', 'django.contrib.contenttypes', 'django.contrib.messages',*  *'django.contrib.staticfiles', 'web']* |
| В шаблонах используйте тег static для URL пути через настроенный STATICFILES\_STORAGE: | {% load static %}  <img src="{% static 'my\_app/example.jpg' %}" alt="My image"> | |

Необходимо организовать хранение статических файлов в папке с именем static в разрабатываемом приложении, например, web/static/example.jpg.

Удобно хранить файлы каждого типа в отдельных папках: изображения – в images, а стили – в css. Можно создать папки и для других типов файлов.

Далее используем эти файлы в шаблоне, для чего в начале файла шаблона определим инструкцию: {% load static %}, при этом код ввести после DOCTYPE.

Чтобы указать путь к статическим файлам используют выражения типа:

{% static "путь к файлу внутри папки static" %}

Для загрузки изображений и стилей в шаблон, удобен такой код *main.html*:

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

*{% load static %}*

*<link rel="stylesheet" type="text/css" href="{% static 'style.css' %}">*

*<meta charset="UTF-8"> <title>Главная страница</title>*

*</head> <body><h1>Главная страница</h1>*

*<img src="{% static '/wallpaper.jpeg' %}" alt="My image"/></body> </html>*

|  |  |  |  |
| --- | --- | --- | --- |
| Cоздадим папку static в приложенииweb.  На **Рис. 24** показана структура созданного приложения:  Создадим файл style.css, со следующим кодом: h1{color:green;} | | |  |
| В my\_project отредактируем файл urls.py, чтобы приложение сопоставило маршруты и страницы: | | from django.contrib import admin  from django.urls import path, include  urlpatterns = [  path('admin/', admin.site.urls),  path('', include('hello.urls')),  path('web/', include('web.urls')),] | |
| Далее откроем web\view.py. Здесь размещены функции представления для работы логики приложения: | from django.shortcuts import render  def main(request):  return render(request, 'main.html', {})  def contact(request):  return render(request, 'contact.html', {}) | | |

Для загрузки JavaScript добавим следующую строку кода в файл index.html.

{% load static %}

<script src="{% static '/js/script.js' %}"

Например:

*// index.html*

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

<head> <meta charset="UTF-8"> <title>Index</title>

{% load static %}

<script src="{% static '/js/script.js' %}" type="text/javascript"></script>

</head> <body> </body> </html>

*// script.js*

alert("Hello, Welcome to Javascript");

В Django можно наследовать шаблоны, позволяя создать базовый шаблон. Он содержит все общие элементы сайта и задает блоки, которые могут переопределять дочерние шаблоны. Дочерний шаблон может модифицировать родительский.

Любой участок шаблона может быть обернут в блочный тег, соблюдая при этом определенные правила. Блоку дается имя. Например:

{% block content %}

тело блока

{% endblock %}

При помощи тега extend программист указывает, какой шаблон необходимо уточнять. Расширяя шаблон, можно переопределить любые блоки, которые есть в родительском шаблоне. Все, что находится вне этих блоков, будет пропущено.

В папке templates приложения web создадим базовый шаблон base.html:

*<html> <html lang="en"><head> <title>{% block title %}{% endblock %}</title>*

*<link rel="stylesheet" type="text/css" href="{%static 'style.css' %}"> </head>*

*<body>{% block content %}{% endblock %}</body></html>*

Подключим базовый шаблон в файле settings.py

*TEMPLATES = [{ 'BACKEND': 'django.template.backends.django.DjangoTemplates',*

*'DIRS': ["web/templates/"], 'APP\_DIRS': True,*

*'OPTIONS': { 'context\_processors': ['django.template.context\_processors.debug',*

*'django.template.context\_processors.request', 'django.contrib.auth.context\_processors.auth',*

*'django.contrib.messages.context\_processors.messages', ], }, }, ]*

|  |  |
| --- | --- |
| Отредактируем main.html с учётом базового шаблона: | *{% extends "base.html" %}*  *{% block title %}Главная страница*  *{% load static %}{% endblock %}*  *{% block content %}*  *<h1>Главная страница</h1>*  *<img src="{% static '/wallpaper.jpeg' %}" alt="My image"/>*  *{% endblock %}* |

***Практическое занятие 4 (1 час)***

1. Перейдем в каталог с проектом project\_django.
2. Продолжим работу с приложение web.
3. Определим STATIC\_URL в файле settings.py: *STATIC\_URL = '/static/'*
4. Cоздадим папку static в приложенииweb. Структура – см. Рис.24
5. Создадим в ней файл style.css, который содержит код: *h1{color:green; }*

Добавим файл изображения wallpaper.jpeg в папку static приложения web

1. Создадим в папке templates файл base.html со следующим кодом:

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

*<head> <title>{% block title %}{% endblock %}</title>*

*{% load static %}<link rel="stylesheet" type="text/css" href="{% static 'style.css' %}">*

*</head><body> {% block content %}{% endblock %} </body></html>*

1. Отредактируем файлы main.html и contact.html в templates:

|  |  |
| --- | --- |
| main.html | *{% extends "base.html" %}*  *{% block title %}Главная страница{% load static %}*  *{% endblock %}*  *{% block content %}<h1>Главная страница</h1>*  *<img src="{% static '/wallpaper.jpeg' %}" alt="My image"/>*  *{% endblock %}* |
| contact.html | *{% extends "base.html" %}*  *{% block title %}Контакты*  *{% endblock %}*  *{% block content %}*  *<h1>Контакты </h1>*  *{% endblock %}* |

1. Запустим локальный сервер: *python manage.py runserver* и идем к адресам 127.0.0.1:8000/web/main и 127.0.0.1:8000/web/contact, увидев следующее:

|  |  |
| --- | --- |
| 127.0.0.1:8000/web/main  (картинка любая с именем wallpaper.jpeg) |  |
| 127.0.0.1:8000/web/contact |  |

Теория и порядок выполнения практической работы **4.2.4** дан выше.

*(Здесь пример для PyCharm в MacOS. В файле* ***reinstall.docx*** *‑ пример для Windows).*

**Эти картинки (скриншоты) могут быть отчетом по работе**