

Урок 2. Создаем приложение для главной страницы

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc99033100)

[ВВЕДЕНИЕ 3](#_Toc99033101)

[НАЧИНАЕМ РЕАЛИЗАЦИЮ СИСТЕМЫ ХРАНЕНИЯ ДАННЫХ 4](#_Toc99033102)

[НАЧИНАЕМ РЕАЛИЗАЦИЮ СИСТЕМЫ АУТЕНТИФИКАЦИИ 4](#_Toc99033103)

[СОЗДАЕМ МОДЕЛИ ПРОЕКТА 5](#_Toc99033104)

[СОЗДАЕМ КОНФИГ ДЛЯ ПРОЕКТА 8](#_Toc99033105)

[СОЗДАЕМ ПЕРВЫЙ БЛЮПРИНТ 9](#_Toc99033106)

[ПОДКЛЮЧАЕМ ШАБЛОНЫ 10](#_Toc99033107)

[ВЫПОЛНЯЕМ ТЕСТОВЫЙ ЗАПУСК ПРОЕКТА 11](#_Toc99033108)

[ЗАКЛЮЧЕНИЕ 13](#_Toc99033109)



# ВВЕДЕНИЕ

Если вспомнить работу с фреймворком Django, то наш проект состоял из приложений, которые были переносимыми компонентами, т.е. мы могли скопировать приложение в любой другой проект, после чего подключить к этому проекту.

Во Flask реализуются так называемые «блюпринты» (англ. blueprints). Это не совсем приложения в стандартном их понимании. Это эскизы или небольшие проекты, из которых мы можем собирать более крупные проекты.

Итак, «блюпринт» не является приложением, но является его макетом. Мы можем создать один макет, определить его конфигурацию и реализовать на его основе любое количество приложений, которые затем зарегистрировать в проекте.

Создадим первый «блюпринт», который обеспечит отображение главной страницы. Но перед этим необходимо реализовать систему хранения данных, в том числе подготовить код моделей, сделать некоторые важные предварительные настройки и т.д.

# НАЧИНАЕМ РЕАЛИЗАЦИЮ СИСТЕМЫ ХРАНЕНИЯ ДАННЫХ

В отличие от Django, во Flask отсутствует встроенный ORM-механизм. Приходится подключать сторонние ORM-библиотеки, например, SQLAlchemy.

Перейдем в файл \_\_init\_\_.py пакета flask\_blog нашего проекта и выполним импорт:

**from** flask\_sqlalchemy **import** SQLAlchemy

Стоит отметить, что стандартная библиотека SQLAlchemy требует предварительной настройки. Если же мы хотим начать работу с ORM максимально быстро, оптимальным вариантом будет установить расширение Flask-SQLAlchemy. Мы его уже установили ранее на основе файла с зависимостями requirements.txt.

Выполним создание объекта класса-конструктора SQLAlchemy. Он необходим для описания моделей.

db = SQLAlchemy()

# НАЧИНАЕМ РЕАЛИЗАЦИЮ СИСТЕМЫ АУТЕНТИФИКАЦИИ

Процедура аутентификации обеспечивает защиту проекта от доступа к его функционалу со стороны неавторизованных пользователей. Можно создать собственную систему аутентификацию с генерацией хешей и т.д. Но это непростая задача, к тому же существуют готовые решения («батарейки»), в которых в которых уже есть встроенные средства для решения задачи аутентификации.

Воспользуемся расширением Flask-Login, которое мы установили ранее, указав в файле requirements.txt.

Перейдем в файл \_\_init\_\_.py пакета flask\_blog нашего проекта и выполним импорт:

**from** flask\_login **import** LoginManager

…

login\_manager = LoginManager()

…

Для подключения системы аутентификации необходимо импортировать класс-конструктор LoginManager и выполнить создание его объекта.

Таким образом, наш файл \_\_init\_\_.py примет следующий вид:

**Листинг 1. flask\_blog/flask\_blog/\_\_init\_\_.py**

**from** flask **import** Flask  
**from** flask\_sqlalchemy **import** SQLAlchemy  
**from** flask\_login **import** LoginManager  
  
  
db = SQLAlchemy()  
login\_manager = LoginManager()  
  
  
**def** create\_app():  
 app = Flask(\_\_name\_\_)  
  
 **return** app

# СОЗДАЕМ МОДЕЛИ ПРОЕКТА

Создадим в пакете flask\_blog модуль models.py и разместим в нем код наших основных классов-моделей. Для начала выполним импорты.

**Листинг 2. flask\_blog/flask\_blog/models.py**

**from** datetime **import** datetime  
**from** flask\_blog **import** db

**from** flask\_login **import** UserMixin

Пока у нас будут две модели-сущности «пользователь» и «запись блога».

Модель для описания сущности «пользователь».

**Листинг 3. flask\_blog/flask\_blog/models.py**

**class** User(db.Model, UserMixin):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 username = db.Column(db.String(20), unique=**True**, nullable=**False**)  
 email = db.Column(db.String(120), unique=**True**, nullable=**False**)  
 image\_file = db.Column(db.String(20), nullable=**False**,

default=**'default.png'**)  
 password = db.Column(db.String(60), nullable=**False**)  
 posts = db.relationship(**'Post'**, backref=**'author'**, lazy=**True**)  
  
**def** \_\_repr\_\_(self):  
 **return f"Пользователь('{**self.username**}',**

**'{**self.email**}', '{**self.image\_file**}')"**

Модель для описания сущности «запись блога»:

**Листинг 4. flask\_blog/flask\_blog/models.py**

**class** Post(db.Model):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 title = db.Column(db.String(100), nullable=**False**)  
 date\_posted = db.Column(db.DateTime, nullable=**False**,  
 default=datetime.utcnow)  
 content = db.Column(db.Text, nullable=**False**)  
 user\_id = db.Column(db.Integer, db.ForeignKey(**'user.id'**),

nullable=**False**)  
  
 **def** \_\_repr\_\_(self):  
 **return f"Запись('{**self.title**}', '{**self.date\_posted**}')"**

Для начала разберем, что означает каждый импорт:

**from** flask\_blog **import** db

Объект класса-конструктора SQLAlchemy. Он необходим нам для описания структуры наших моделей. Но судя по импорту сам объект мы должны создать где-то в проекте.

А мы помним, что все важные инициализации мы выполняем на уровне пакета проекта в файле \_\_init\_\_.py, поэтому дополним файл \_\_init\_\_.py кодом:

**Листинг 5. flask\_blog/flask\_blog/\_\_init\_\_.py**

**from** flask\_sqlalchemy **import** SQLAlchemy  
  
…

db = SQLAlchemy()  
  
**def** create\_app():  
 app = Flask(\_\_name\_\_)  
  
 db.init\_app(app)  
 …

Возвращаемся вновь к коду моделей:

**from** flask\_login **import** UserMixin

Здесь мы вспомним про так называемые «миксины» («примеси»). Это классы, обеспечивающие добавление дополнительных возможностей (атрибутов и методов) в другие классы. Плюс такого подхода в том, что «миксины» не переопределяют методы других классов, они действительно выполняют роль примесей, когда нужно добавить в класс некоторый функционал.

**class** User(db.Model, UserMixin):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 username = db.Column(db.String(20), unique=**True**, nullable=**False**)  
 email = db.Column(db.String(120), unique=**True**, nullable=**False**)  
 image\_file = db.Column(db.String(20), nullable=**False**,  
 default=**'default.png'**)  
 password = db.Column(db.String(60), nullable=**False**)  
 posts = db.relationship(**'Post'**, backref=**'author'**, lazy=**True**)

Набор атрибутов класса-модели User интуитивно понятный. Вопросы вызывают некоторые параметры атрибутов.

nullable=**False**

Запрещена установка значения типа Null.

posts = db.relationship(**'Post'**, backref=**'author'**, lazy=**True**)

Ссылка на запись (пост), оставленную пользователем в блоге.

backref=**'author'**

Это указание на запись в модели Post, чтобы можно было отслеживать кто автор записи блога.

lazy=**True**

Означает, что связанные записи наших двух таблиц («пользователи» и «посты») будут загружаться параллельно.

**class** Post(db.Model):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 title = db.Column(db.String(100), nullable=**False**)  
 date\_posted = db.Column(db.DateTime, nullable=**False**,  
 default=datetime.utcnow)  
 content = db.Column(db.Text, nullable=**False**)  
 user\_id = db.Column(db.Integer, db.ForeignKey(**'user.id'**),

nullable=**False**)  
  
 **def** \_\_repr\_\_(self):  
 **return f"Запись('{**self.title**}', '{**self.date\_posted**}')"**

Набор атрибутов модели Post интуитивно понятен.

# СОЗДАЕМ КОНФИГ ДЛЯ ПРОЕКТА

Добавим в пакет проекта служебный модуль config.py.

**Листинг 6. flask\_blog/flask\_blog/config.py**

**class** Config:  
 SQLALCHEMY\_DATABASE\_URI = **'sqlite:///site.db'** SECRET\_KEY = **'5791628bb0b13ce0c676dfde280ba245'**

Параметр SQLALCHEMY\_DATABASE\_URI содержит ссылку на путь до базы данных, который будет использоваться для подключения к этой базе данных.

Параметр SECRET\_KEY соответствует секретному ключу, используемому для подписи сеансовых cookie-файлов с целью защиты их от подделки. Секретный ключ может иметь случайное значение, главное, скрыть его от злоумышленника.

# СОЗДАЕМ ПЕРВЫЙ БЛЮПРИНТ

Создадим в пакете проекта flask\_blog директорию main для нашего первого и главного приложения и инициализируем директорию main в качестве пакета.

Добавим в пакет main модуль routes.py.

**Листинг 7. flask\_blog/flask\_blog/main/routes.py**

**from** flask **import** render\_template, Blueprintmain = Blueprint(**'main'**, \_\_name\_\_)  
  
  
@main.route(**"/"**)  
@main.route(**"/home"**)  
**def** home():  
 **return** render\_template(**'home.html'**)

В приведенном листинге мы указываем, что создаем макет приложения («блюпринт»), т.е. если сравнивать с Django, мы создаем первое приложение проекта. Использование «блюпринтов» похоже на работу с пространством имен, как это было в Django. Мы создавали набор url-шаблонов для каждого приложения, т.е. у каждого приложения был свой набор связанных url-маршрутов.

Код также содержит простейший контроллер и маршрутизацию. Контроллер обрабатывает запрос перехода на главную страницу проекта.

Теперь нам необходимо зарегистрировать наш первый «блюпринт».

Перейдем в файл \_\_init\_\_() пакета проекта и добавим следующие выражения:

**Листинг 8. flask\_blog/flask\_blog/\_\_init\_\_.py**

**…**  
  
**def** create\_app():  
 app = Flask(\_\_name\_\_)  
   
 **from** flask\_blog.main.routes **import** main  
 app.register\_blueprint(main)

Это похоже на то, как мы регистрируем файлы urls.py уровня приложений в файле urls.py уровня проекта в Django. Также можно найти сходством с регистрацией самих приложение в модуле settings.py Django-проекта.

Параллельно подключим конфигурационный файл для нашего проекта и инициализируем взаимодействие проекта с базой данных.

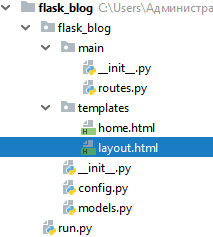
**Листинг 9. flask\_blog/flask\_blog/\_\_init\_\_.py**

**…**  
**from** flask\_blog.config **import** Config  
…  
**def** create\_app():  
 …  
 app.config.from\_object(Config)  
 …

# ПОДКЛЮЧАЕМ ШАБЛОНЫ

Наша цель – сделать больший упор на бэкенд, поэтому для реализации интерфейса можно найти и применить готовый шаблон или воспользоваться материалами курса.

Создадим в пакете проекта директорию templates, в которую поместим базовый шаблон layout.html и шаблон-наследник home.html.

Указанные шаблоны пока можно взять из готового варианта проекта, поскольку вручную их создавать – достаточно трудоемкий процесс.

# ВЫПОЛНЯЕМ ТЕСТОВЫЙ ЗАПУСК ПРОЕКТА

Запустим проект, перейдем на главную страницу увидим ошибку:

В терминале PyCharm более детальная информация об ошибке: 

Действительно, в базовом шаблоне layout.html присутствует переменная контекста current\_user:

{% if current\_user.is\_authenticated %}  
 <**h4**>Hello, {{ current\_user.username }}!</**h4**>  
{% endif %}

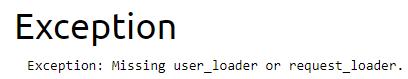
Но переменная не определяется, поэтому мы получаем ошибку.

Для решения проблемы нам необходимо выполнить регистрацию логин-менеджера. Объект логин-менеджера мы уже создали ранее.

В файл \_\_init\_\_.py добавим следующие выражения:

**Листинг 10. flask\_blog/flask\_blog/\_\_init\_\_.py**

**from** flask\_login **import** LoginManager  
  
…  
login\_manager = LoginManager()  
  
  
**def** create\_app():  
 …  
 login\_manager.init\_app(app)  
 **…**

Теперь при запуске возникает другая ошибка:

Нам нужно добавить еще один блок кода в модуль с моделями.

**Листинг 11. flask\_blog/flask\_blog/models.py**

**from** flask\_blog **import** db, login\_manager  
  
  
@login\_manager.user\_loader  
**def** load\_user(user\_id):  
 **return** User.query.get(int(user\_id))

А также добавим импорт модели пользователя в модуль routes.py директории main.

**Листинг 12. flask\_blog/flask\_blog/main/routes.py**

**from** flask\_blog.models **import** User

Это необходимо для того, чтобы при загрузке страницы отработал наш декоратор:

@login\_manager.user\_loader

Перезапустим приложение и перейдем на главную страницу:



# Вид сверху на рукопожатие над деловыми документамиВид сверху на рукопожатие над деловыми документамиЗАКЛЮЧЕНИЕ

По итогам второго урока мы подготовили наши первые классы-модели, создали, настроили и подключили первое приложение проекта. Для этого приложения мы создали макет – «блюпринт». Настроили простейший контроллер и маршрутизацию для него. Мы подготовили несколько первых шаблонов. Также мы выполнили тестовый запуск проекта, выявили и устранили несколько ошибок.

Текущая структура проекта приведена на рис. 1.

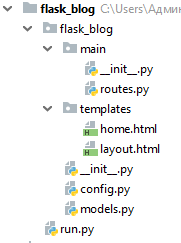


Рис. 1. Структура проекта