

Лекция 3.

Дополнительные возможности Flask



Оглавление

Flask-SQLAlchemy	3
Введение	3
➤ Что такое Flask-SQLAlchemy?	3
➤ Зачем использовать Flask-SQLAlchemy?	4
Установка и настройка	4
➤ Установка Flask-SQLAlchemy	4
➤ Настройка подключения к базе данных	4
➤ Разделение проекта на отдельные компоненты	5
Создание моделей	6
➤ Определение классов моделей	6
➤ Описание полей моделей	7
➤ Создание связей между моделями	8
➤ Создание таблиц в базе данных	8
Работа с данными	9
➤ Создание записей	9
➤ Изменение записей	10
➤ Удаление записей	10
➤ Наполнение тестовыми данными	11
➤ Получение данных из базы данных	11
➤ Фильтрация данных	12
Заключение по работе с Flask-SQLAlchemy	14
Flask-WTForm	14
Введение	14
Зачем использовать Flask-WTF?	14
Установка и настройка	15
Установка Flask-WTF	15
Настройка защиты от CSRF-атак	15
Создание форм в WTForms	16
Определение классов форм	16
Описание полей форм	16
Валидация данных формы	17
Использование форм WTForms в приложении	18
Отображение форм на страницах приложения	18
Обработка данных из формы	19
Заключение по работе с WTForms	21

На этой лекции мы

1. Узнаем про работу с базами данных посредством Flask-SQLAlchemy
2. Разберёмся с созданием форм средствами Flask-WTForm

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали про экранирование пользовательских данных
2. Разобрались с генерацией url адресов
3. Изучили обработку GET и POST запросов
4. Узнали несколько полезных функций Flask
5. Разобрались с cookie файлами и сессиями

Подробный текст лекции

Flask-SQLAlchemy

Введение

➤ Что такое Flask-SQLAlchemy?

Flask-SQLAlchemy — это расширение Flask, которое облегчает работу с базами данных в приложениях, написанных на Flask. Оно предоставляет удобный интерфейс для работы с базами данных, а также мощные инструменты для создания и управления моделями данных.

➤ Зачем использовать Flask-SQLAlchemy?

Безусловно, использование баз данных является неотъемлемой частью многих веб-приложений. Flask-SQLAlchemy позволяет легко и быстро создавать и управлять базами данных, что делает его идеальным выбором для разработки веб-приложений любого уровня сложности.

Кроме того, Flask-SQLAlchemy обладает множеством преимуществ по сравнению с другими ORM-библиотеками. Он предоставляет более простой и понятный синтаксис для работы с базами данных, а также обладает отличной производительностью и масштабируемостью.

Пример использования Flask-SQLAlchemy можно привести на основе создания блога. В блоге нужно хранить информацию о пользователях, статьях, комментариях и т.д. Flask-SQLAlchemy поможет легко создать и управлять базой данных для хранения всех этих данных.

Установка и настройка

➤ Установка Flask-SQLAlchemy

Для установки Flask-SQLAlchemy необходимо выполнить команду:

```
pip install Flask-SQLAlchemy # Windows
pip3 install Flask-SQLAlchemy # Unix
```

После этого можно импортировать его в свой проект:

```
from flask_sqlalchemy import SQLAlchemy
```

➤ Настройка подключения к базе данных

Для настройки подключения к базе данных необходимо указать адрес базы данных, а также ее тип и некоторые другие параметры. Например, для подключения к базе данных SQLite можно использовать следующий код:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///mydatabase.db'
db = SQLAlchemy(app)

...
```

В данном примере мы создаем объект app класса Flask, указываем адрес базы данных в параметре SQLALCHEMY_DATABASE_URI и создаем объект db класса SQLAlchemy.

Адрес базы данных может быть различным в зависимости от ее типа и места расположения. В данном примере мы используем базу данных SQLite, которая хранится в файле mydatabase.db в текущей директории.

Также можно использовать другие типы баз данных, такие как MySQL, PostgreSQL и другие. Для этого необходимо указать соответствующий адрес и параметры подключения.

Например, для подключения к базе данных MySQL можно использовать следующий код:

```
...
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql+pymysql://username:password@hostname/database_name'
db = SQLAlchemy(app)
...
```

Здесь мы указываем адрес базы данных в формате:

mysql+pymysql://username:password@hostname/database_name, где

username и password - это логин и пароль для подключения к базе данных, hostname - адрес сервера базы данных, а database_name - имя базы данных.

При подключении к PostgreSQL используется аналогичная строка. Изменяется лишь начало:

postgresql+psycopg2://username:password@hostname/database_name

➤ Разделение проекта на отдельные компоненты

Чтобы Flask проект не превратился один файл гигантского размера, вынесем работу с БД в отдельный файл models.py.

На текущем этапе в нём будут следующие строки:

```
from flask_sqlalchemy import SQLAlchemy


db = SQLAlchemy()
```

Внутри основного файла проекта оставим следующий код:

```
from flask import Flask
from flask_lesson_3.models import db

app = Flask(__name__)
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///mydatabase.db'
db.init_app(app)
```

 **Важно!** Теперь класс не получает приложение Flask app при инициализации. Для инициализации баз данных необходимо выполнить строку `db.init_app(app)`

Создание моделей

При работе с Flask-SQLAlchemy необходимо определить модели данных, которые будут использоваться в приложении. Модель - это класс, который описывает структуру таблицы в базе данных.

➤ Определение классов моделей

Для определения модели необходимо создать класс, который наследует от класса `Model` из библиотеки `SQLAlchemy`. Название класса должно соответствовать названию таблицы в базе данных.

Наполняем кодом `models.py`

Пример:

```
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime

db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True,
        nullable=False)
    email = db.Column(db.String(120), unique=True,
        nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

    def __repr__(self):
        return f'User({self.username}, {self.email})'
```

В этом примере определена модель `User`, которая имеет четыре поля: `id`, `username`, `email` и `created_at`. Поле `id` является первичным ключом таблицы и автоматически генерируется при добавлении записи в таблицу. Поля `username` и `email` являются

строками с ограничением на уникальность и обязательность заполнения. Поле `created_at` содержит дату и время создания записи и автоматически заполняется текущей датой и временем при добавлении записи.

➤ Описание полей моделей

Для описания полей модели используются классы-типы данных из библиотеки SQLAlchemy. Существуют следующие типы данных:

- Integer — целое число
- String — строка
- Text — текстовое поле
- Boolean — булево значение
- DateTime — дата и время
- Float — число с плавающей точкой
- Decimal — десятичное число
- Enum — перечисление значений
- ForeignKey — внешний ключ к другой таблице

Рассмотрим ещё один пример таблицы:

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    content = db.Column(db.Text, nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey('user.id'),
        nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

    def __repr__(self):
        return f'Post({self.title}, {self.content})'
```

В этом примере определена модель `Post`, которая имеет пять полей: `id`, `title`, `content`, `author_id` и `created_at`. Поля `title` и `content` являются строками и обязательны для заполнения. Поле `author_id` является внешним ключом к таблице пользователей (`User`) и ссылается на поле `id` этой таблицы. Поле `created_at` содержит дату и время создания записи и автоматически заполняется текущей датой и временем при добавлении записи.

К модели пользователя добавим следующую строку:

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

Так мы (а точнее наш код) понимаем какие посты принадлежат конкретному пользователю.

➤ Создание связей между моделями

Для создания связей между моделями используется поле ForeignKey. Оно указывает на поле первичного ключа связанной таблицы.

```
class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.Text, nullable=False)
    post_id = db.Column(db.Integer, db.ForeignKey('post.id'),
        nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey('user.id'),
        nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

    def __repr__(self):
        return f'Comment({self.content})'
```

В этом примере определена модель Comment, которая имеет пять полей: id, content, post_id, author_id и created_at. Поля content и post_id являются обязательными для заполнения. Поле post_id является внешним ключом к таблице постов (Post) и ссылается на поле id этой таблицы. Поле author_id является внешним ключом к таблице пользователей (User) и ссылается на поле id этой таблицы. Поле created_at содержит дату и время создания записи и автоматически заполняется текущей датой и временем при добавлении записи.

➤ Создание таблиц в базе данных


Остался финальный этап. Напишем функцию, которая создаст таблицы через консольную команду. Заполняем основной файл проекта

```
...
from flask_lesson_3.models import db, User, Post, Comment
...

@app.cli.command("init-db")
def init_db():
    db.create_all()
    print('OK')
...
```


Из `models` импортировали все созданные классы таблиц. Без этого импорта функция `create_all` может не увидеть какие таблицы необходимо создать. Далее создали функцию, которая будет вызвана командой в консоли:

```
flask init-db
```

 **Внимание!** Если команда в консоли выдает ошибку, проверьте что у вас есть `wsgi.py` файл в корневой директории проекта и он верно работает. Например его код может быть таким:

```
from flask_lesson_3.app_01 import app

if __name__ == '__main__':
    app.run(debug=True)
```

Мы рассмотрели основные аспекты создания моделей в Flask-SQLAlchemy. Были описаны классы моделей, поля моделей и создание связей между моделями.

Работа с данными

После определения моделей в Flask-SQLAlchemy можно начать работу с данными в базе данных. Давайте рассмотрим основные методы для создания, изменения и удаления записей, а также получения данных из базы данных и их фильтрацию.

➤ Создание записей

Для создания новой записи в базе данных необходимо создать объект модели и добавить его в сессию базы данных. После этого нужно вызвать метод `commit()` для сохранения изменений.

```
...
@app.cli.command("add-john")
def add_user():
    user = User(username='john', email='john@example.com')
    db.session.add(user)
    db.session.commit()
    print('John add in DB!')
...
```

В этом примере создается новый объект модели User с именем пользователя "john" и электронной почтой "john@example.com". Затем объект добавляется в сессию базы данных и сохраняется с помощью метода commit().


Как вы уже догадались для выполнения функции необходимо выполнить в консоли команду flask add-john

➤ Изменение записей

Для изменения существующей записи нужно получить ее из базы данных, изменить нужные поля и вызвать метод commit().

```
...
@app.cli.command("edit-john")
def edit_user():
    user = User.query.filter_by(username='john').first()
    user.email = 'new_email@example.com'
    db.session.commit()
    print('Edit John mail in DB!')
...
```

В этом примере получаем объект модели User по имени пользователя "john", изменяем его электронную почту на "new_email@example.com" и сохраняем изменения с помощью метода commit().

 **Внимание!** Если бы база данных позволяла хранить несколько пользователей с одинаковыми username, в переменную user попал бы один пользователь благодаря методу first().

➤ Удаление записей

Для удаления записи нужно получить ее из базы данных, вызвать метод delete() и затем вызвать метод commit().

```
...
@app.cli.command("del-john")
def del_user():
    user = User.query.filter_by(username='john').first()
    db.session.delete(user)
    db.session.commit()
    print('Delete John from DB!')
...
```

В этом примере получаем объект модели User по имени пользователя "john", удаляем его из базы данных с помощью метода delete() и сохраняем изменения с помощью метода commit().

➤ Наполнение тестовыми данными

Перед тем как двигаться добавим в базу несколько тестовых пользователей и их статей.

```
@app.cli.command("fill-db")
def fill_tables():
    count = 5
    # Добавляем пользователей
    for user in range(1, count + 1):
        new_user = User(username=f'user{user}',
            email=f'user{user}@mail.ru')
        db.session.add(new_user)
        db.session.commit()

    # Добавляем статьи
    for post in range(1, count ** 2):
        author = User.query.filter_by(username=f'user{post %
count + 1}').first()
        new_post = Post(title=f'Post title {post}',
            content=f'Post content {post}', author=author)
        db.session.add(new_post)
        db.session.commit()
```

Вначале мы записываем в БД count пользователей. А далее генерируем статьи, которые они написали.

➤ Получение данных из базы данных

Для получения данных из базы данных необходимо использовать метод query() модели. Этот метод возвращает объект запроса, который можно дополнить фильтрами и другими параметрами.

```
...
@app.route('/users/')
def all_users():
    users = User.query.all()
    context = {'users': users}
    return render_template('users.html', **context)
...
```

В этом примере получаем все объекты модели User из базы данных с помощью метода all(). Пробросим их в шаблон, где выводим имена пользователей и электронные адреса.

```
{% extends 'base.html' %}

{% block title %}
    {{ super() }} - {{ title }}
{% endblock %}

{% block content %}
    <div class="row">
        {% for user in users %}
            <p class="col-12 col-md-6">Имя пользователя: {{
user.username }}<br>
            Email пользователя: {{ user.email }}</p>
        {% endfor %}
    </div>
{% endblock %}
```

➤ Фильтрация данных

Для фильтрации данных можно использовать метод filter() объекта запроса. Этот метод принимает условия фильтрации в виде аргументов или объектов-атрибутов модели.


```
...
@app.route('/users/<username>/')
def users_by_username(username):
    users = User.query.filter(User.username == username).all()
    context = {'users': users}
    return render_template('users.html', **context)
...
```

В этом примере получаем все объекты модели User из базы данных, у которых имя пользователя равно username из строки запроса, с помощью метода filter() и выводим их имена пользователей и электронные адреса используя прежний шаблон.

Рассмотрим ещё один вариант фильтрации данных

```
@app.route('/posts/author/<int:user_id>/')
def get_posts_by_author(user_id):
    posts = Post.query.filter_by(author_id=user_id).all()
    if posts:
        return jsonify([{'id': post.id, 'title': post.title,
'content': post.content, 'created_at': post.created_at} for post
in posts])
    else:
        return jsonify({'error': 'Posts not found'})
```


Мы создаем маршрут /posts/author/<int:user_id>, который принимает ID пользователя в качестве параметра. Внутри маршрута мы используем метод filter_by для фильтрации постов по ID автора и метод all для получения всех найденных постов. Если посты найдены, мы возвращаем их данные в формате JSON, иначе возвращаем ошибку.

 **Внимание!** Для того, чтобы вернуть JSON объект используется функция jsonify. Её необходимо импортировать из модуля Flask перед использованием.

Финальный пример фильтрации данных

```
@app.route('/posts/last-week/')
def get_posts_last_week():
    date = datetime.utcnow() - timedelta(days=7)
    posts = Post.query.filter(Post.created_at >= date).all()
    if posts:
        return jsonify([{'id': post.id, 'title': post.title,
'content': post.content, 'created_at': post.created_at} for post
in posts])
    else:
        return jsonify({'error': 'Posts not found'})
```

Создаем маршрут /posts/last-week, который возвращает все посты, созданные за последнюю неделю. Внутри маршрута мы используем модуль datetime для вычисления даты, которая была неделю назад, и метод filter для фильтрации постов по дате создания. Если посты найдены, мы возвращаем их данные в формате JSON, иначе возвращаем ошибку.

 **Важно!** Для работы без ошибок, добавьте строку импорта в начале файла:

```
from datetime import datetime, timedelta
```

Заключение по работе с Flask-SQLAlchemy

Flask-SQLAlchemy — это мощный инструмент для работы с базами данных в приложениях Flask. Он предоставляет простой и удобный интерфейс для создания моделей, выполнения запросов и управления данными.

Мы рассмотрели основные функции Flask-SQLAlchemy, такие как создание моделей, работу с данными, получение и фильтрацию данных. Мы также рассмотрели создание запросов к базе данных с помощью SQLAlchemy ORM.

Flask-SQLAlchemy позволяет разработчикам быстро и легко создавать и поддерживать базы данных в своих приложениях Flask. Он также обеспечивает безопасность и надежность работы с данными.

Flask-WTForm

Введение

Flask-WTForm — это модуль для Flask, который предоставляет инструменты для работы с формами веб-приложений на Python. Flask-WTForm позволяет легко создавать и обрабатывать формы, валидировать данные, защищать приложение от атак CSRF и многое другое.

Зачем использовать Flask-WTF?

Создание форм вручную может быть утомительным и трудоемким процессом, особенно если вы хотите создать несколько форм. Flask-WTF упрощает этот процесс, предоставляя множество инструментов, которые позволяют быстро создавать и обрабатывать формы.

Кроме того, Flask-WTF предоставляет механизмы валидации данных, что позволяет легко проверять правильность заполнения формы. Это особенно важно при работе с конфиденциальной информацией, такой как пароли или номера кредитных карт.

Flask-WTF также обеспечивает защиту от атак CSRF (межсайтовой подделки запросов), что является важным аспектом безопасности веб-приложений.

Установка и настройка

Установка Flask-WTF

Для установки Flask-WTF необходимо выполнить команду:

```
pip install Flask-WTF
```

После установки модуля его можно импортировать в приложение Flask:

```
from flask_wtf import FlaskForm
```

Настройка защиты от CSRF-атак

Защита от CSRF-атак в Flask-WTF осуществляется с помощью генерации токена, который добавляется к каждой форме. При отправке формы этот токен проверяется на сервере, чтобы убедиться, что запрос был отправлен с того же сайта.

Для включения защиты от CSRF-атак в Flask-WTF необходимо установить секретный ключ приложения. Этот ключ используется для генерации токена и должен быть достаточно длинным и случайным.

Рассмотрим пример настройки защиты от CSRF-атак:

```
from flask_wtf.csrf import CSRFProtect

app = Flask(__name__)
app.config['SECRET_KEY'] = 'mysecretkey'
csrf = CSRFProtect(app)
```

В этом примере мы создаем объект csrf и передаем ему приложение Flask. Затем мы устанавливаем секретный ключ приложения. После этого защита от CSRF-атак будет включена для всех форм в приложении.

Если вы хотите отключить защиту от CSRF-атак для определенной формы, вы можете использовать декоратор csrf.exempt:

```
from flask_wtf.csrf import CSRFProtect

app = Flask(__name__)
app.config['SECRET_KEY'] = 'mysecretkey'
csrf = CSRFProtect(app)
```

```
@app.route('/form', methods=['GET', 'POST'])
@csrf.exempt
def my_form():
    ...
```

В этом примере мы используем декоратор `exempt` объекта `csrf` для отключения защиты от CSRF-атак для формы, которая обрабатывается в функции `my_form()`. Защита от CSRF-атак является важным аспектом безопасности веб-приложений. Flask-WTF предоставляет простые и эффективные механизмы защиты от таких атак, которые можно легко настроить в приложении Flask.

Создание форм в WTForms

WTForms — это библиотека Python, которая позволяет создавать HTML-формы, а также проводить их валидацию. Flask-WTF использует WTForms для создания форм.

Определение классов форм

Для создания формы с помощью Flask-WTF необходимо определить класс формы, который наследуется от класса `FlaskForm`. Каждое поле формы определяется как экземпляр класса, который наследуется от класса `Field`.

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username',
        validators=[DataRequired()])
    password = PasswordField('Password',
        validators=[DataRequired()])
```

В данном примере определен класс `LoginForm`, который наследуется от `FlaskForm`. Внутри класса определены два поля: `username` и `password`. Поле `username` является строковым полем, а поле `password` — полем для ввода пароля. Оба поля обязательны для заполнения, так как им передан валидатор `DataRequired`.

Описание полей форм

WTForms предоставляет множество типов полей для формы. Вот некоторые из них:

- StringField — строковое поле для ввода текста;
- IntegerField — числовое поле для ввода целочисленных значений;
- FloatField — числовое поле для ввода дробных значений;
- BooleanField — чекбокс;
- SelectField — выпадающий список;
- DateField — поле для ввода даты;
- FileField — поле для загрузки файла.

Рассмотрим ещё один пример создания форм:

```
from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField, SelectField
from wtforms.validators import DataRequired

class RegisterForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired()])
    age = IntegerField('Age', validators=[DataRequired()])
    gender = SelectField('Gender', choices=[('male', 'Мужчина'),
                                           ('female', 'Женщина')])
```

В данном примере определен класс RegisterForm, который наследуется от FlaskForm. Внутри класса определены три поля: name, age и gender. Поле name является строковым полем, поле age — числовым, а поле gender — выпадающим списком. В списке выбора есть две опции: male и female.

Валидация данных формы

WTForms позволяет проводить валидацию данных формы. Для этого можно использовать готовые валидаторы, такие как DataRequired, Email, Length и другие. Также можно написать свой собственный валидатор.

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email, EqualTo

class RegistrationForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=6)])
    confirm_password = PasswordField('Confirm Password', validators=[DataRequired(),
                                                                        EqualTo('password')])
```

В данном примере определен класс `RegistrationForm`, который наследуется от `FlaskForm`. Внутри класса определены три поля: `email`, `password` и `confirm_password`. Поле `email` проверяется на наличие данных и на соответствие формату `email`. Поле `password` проверяется на наличие данных и на минимальную длину (6 символов). Поле `confirm_password` проверяется на наличие данных и на соответствие значению поля `password`.



Важно! Для правильной работы кода необходимо отдельно установить валидатор электронной почты. Для этого достаточно выполнить команду:

```
pip install email-validator
```



Внимание! В валидатор `EqualTo` передаётся строковое имя переменной, т.е. то, что стоит слева от знака равно, а не название поля

`WTForms` — мощная библиотека для создания HTML-форм и их валидации в `Flask`. Определение классов форм является основой работы с библиотекой. Описание полей форм и проведение их валидации позволяют создавать надежные и удобные для пользователей формы.

Использование форм `WTForms` в приложении

В предыдущем пункте мы рассмотрели, как создавать формы с помощью `WTForms`. Теперь рассмотрим, как использовать эти формы в приложении `Flask`.

Отображение форм на страницах приложения

Для отображения формы на странице приложения необходимо создать объект формы в представлении и передать его в шаблон.

```
from flask import render_template, request
from forms import LoginForm
```

```
@app.route('/login/', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if request.method == 'POST' and form.validate():
        # Обработка данных из формы
        pass
    return render_template('login.html', form=form)
```

В данном примере определен маршрут /login, который обрабатывает GET и POST запросы. В представлении создается объект LoginForm, который передается в шаблон login.html с помощью функции render_template. Если метод запроса POST и данные формы проходят валидацию, то выполняется обработка данных из формы. Шаблон login.html должен содержать тег form с указанием метода и адреса для отправки данных формы, а также поля формы с помощью тегов input.

```
{% extends "base.html" %}

{% block content %}
    <h1>Login</h1>
    <form method="POST" action="{{ url_for('login') }}">
        {{ form.csrf_token }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>
            <input type="submit" value="Login">
        </p>
    </form>
{% endblock %}
```

Внутри блока content определен тег form с методом POST и адресом /login. Для каждого поля формы вызывается соответствующий метод объекта формы (например, form.username для поля username) с указанием размера поля.

Обработка данных из формы

Для обработки данных из формы необходимо получить данные из объекта request и провести их валидацию с помощью метода validate() объекта формы.

```

from flask import render_template, request
from forms import RegistrationForm

@app.route('/register/', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if request.method == 'POST' and form.validate():
        # Обработка данных из формы
        email = form.email.data
        password = form.password.data
        print(email, password)
        ...
    return render_template('register.html', form=form)

```

В данном примере определен маршрут /register, который обрабатывает GET и POST запросы. В представлении создается объект RegistrationForm, который передается в шаблон register.html с помощью функции render_template. Если метод запроса POST и данные формы проходят валидацию, то выполняется обработка данных из формы. Данные из полей формы можно получить с помощью свойств data объекта формы. Например, для поля email можно получить значение следующим образом: form.email.data.

Рассмотрим шаблон register.html

```

{% extends "base.html" %}

{% block content %}
<h1>Login</h1>
<form method="POST" action="{{ url_for('register') }}">
    {{ form.csrf_token }}
    {% for field in form if field.name != 'csrf_token' %}
    <p>
        {{ field.label }}<br>
        {{ field }}
        {% if field.errors %}
        <ul class="alert alert-danger">
            {% for error in field.errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
        {% endif %}
    </p>
    {% endfor %}

```

```
<p>
    <input type="submit" value="Register">
</p>

</form>
{% endblock %}
```

В отличие от шаблона login.html мы не указываем поля явно. После стандартного вывода csrf токена создаём цикл для по всем полям формы за исключением токена. Для каждого поля выводится метка и окно поле ввода. Отдельно проверяем наличие ошибок ввода и если они есть, в цикле выводим все ошибки для каждого из полей. Таким образом мы динамически формируем страницу регистрации. А в случае неверного ввода данных пользователем, сразу сообщаем ему об ошибках.

WTForms позволяет легко создавать и валидировать HTML-формы в приложении Flask. Для отображения форм на страницах приложения необходимо создать объект формы в представлении и передать его в шаблон. Для обработки данных из формы необходимо получить данные через post запрос и провести их валидацию с помощью метода validate() объекта формы.

Заключение по работе с WTForms

WTForms — это мощный инструмент для создания и валидации HTML-форм в приложении Flask. Он позволяет легко определять поля формы, устанавливать правила валидации и обрабатывать данные из формы.

Мы рассмотрели основные возможности WTForms и показали, как его использовать в приложении Flask. Мы создали простую форму входа и форму регистрации, а также рассмотрели процесс валидации данных из формы.

WTForms имеет множество дополнительных функций, таких как поддержка многоязычности, кастомизация полей формы и многое другое. Вы можете ознакомиться с дополнительными возможностями в официальной документации WTForms.

Использование WTForms в приложении Flask позволяет значительно упростить процесс создания и валидации HTML-форм. Это помогает сократить время разработки и повысить качество кода.

Вывод

На этой лекции мы:

1. Узнали про работу с базами данных посредством Flask-SQLAlchemy
2. Разобрались с созданием форм средствами Flask-WTForm

Домашнее задание

Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.