# Flask Blueprint Application Setup - Software Engineering Lesson

## Learning Objectives

By the end of this lesson, students will understand:

1. Flask application structure and organization
2. Blueprint pattern for modular applications
3. Database integration with Flask-SQLAlchemy
4. Configuration management
5. Application factory pattern
6. Migration handling

---

## Project Overview

This lesson demonstrates building a modular Flask application using Blueprints - a scalable approach for organizing large Flask applications. We'll build a foundation that can be extended into a County Services Portal.

### 🏗️ Final Project Structure

```
fs_prep/
├── config.py                 # Configuration management
├── requirements.txt           # Project dependencies
├── run.py                    # Application entry point
├── lesson.md                 # This lesson file
├── app/                      # Main application package
│   ├── __init__.py          # Application factory
│   ├── extensions.py        # Flask extensions
│   ├── main/                # Main blueprint (homepage, etc.)
│   │   ├── __init__.py
│   │   └── views.py
│   ├── auth/                # Authentication blueprint
│   │   ├── __init__.py
│   │   └── routes.py
│   ├── api/                 # API blueprint
│   │   ├── __init__.py
│   │   └── routes.py
│   └── models/              # Database models
│       └── __init__.py
└── instance/                # Instance-specific files
    └── county_services.db   # SQLite database
```

---

## Step 1: Environment Setup and Dependencies

### Understanding Requirements

The `requirements.txt` file defines all the Python packages our application needs:

```
Flask==3.1.0                 # Core web framework
Flask-SQLAlchemy==3.1.1      # Database ORM integration
Flask-Security-Too==5.4.3    # Authentication and authorization
Flask-RESTful==0.3.10        # REST API development
Flask-Migrate==4.0.7         # Database migration management
Flask-WTF==1.2.1             # Form handling and CSRF protection
WTForms==3.1.2               # Form validation
Flask-Mail==0.10.0           # Email functionality
Flask-Bootstrap==3.3.7.1     # Bootstrap CSS framework integration
Flask-FontAwesome==0.1.5     # Icon integration
Flask-Moment==1.0.6          # Date/time formatting
python-dotenv==1.0.1         # Environment variable management
bcrypt==4.2.0                # Password hashing
Pillow==10.4.0               # Image processing
reportlab==4.2.5             # PDF generation
```

**Key Concepts:**

- **Flask**: Lightweight WSGI web application framework
- **SQLAlchemy**: Object-Relational Mapping (ORM) for database operations
- **Blueprints**: Modular components for organizing Flask applications
- **Migrations**: Version control for database schema changes

---

## Step 2: Configuration Management

### `config.py` - Centralized Configuration

```python
import os
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

class Config:
    # General settings
    SECRET_KEY = os.getenv('SECRET_KEY')
    SQLALCHEMY_DATABASE_URI = os.getenv('DATABASE_URL')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

**Teaching Notes:**

1. **Environment Variables**: Keep sensitive data (secrets, database URLs) out of code
2. **Configuration Class**: Centralizes all app settings in one place
3. **python-dotenv**: Loads environment variables from `.env` file for development
4. **SQLALCHEMY_TRACK_MODIFICATIONS = False**: Disables event system (saves memory)

**Best Practices:**

- Never commit `.env` files to version control
- Use different configurations for development, testing, and production
- Environment variables make applications cloud-ready

---

## Step 3: Flask Extensions Management

### `app/extensions.py` - Extension Initialization

```python
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

db = SQLAlchemy()
migrate = Migrate()
```

**Teaching Notes:**

1. **Extension Pattern**: Initialize extensions outside the application factory
2. **Lazy Initialization**: Extensions are created but not bound to app instance yet
3. **Import Separation**: Prevents circular imports in complex applications

**Why This Approach?**

- **Testability**: Easy to use different configurations for testing
- **Modularity**: Extensions can be selectively enabled/disabled
- **Circular Import Prevention**: Common issue in Flask applications

---

## Step 4: Blueprint Structure

### Understanding Blueprints

Blueprints are Flask's way of organizing applications into modules. Each blueprint represents a functional area of your application.

#### `app/main/views.py` - Main Blueprint

```python
from flask import Blueprint

main_bp = Blueprint('main_bp', __name__)

@main_bp.route('/', methods=['GET'])
def home():
    """Endpoint to handle the home route."""
    return {"Welcome to the home page!"}, 200
```

**Teaching Notes:**

1. **Blueprint Creation**: `Blueprint('name', import_name)`
2. **Route Decorators**: `@blueprint.route()` instead of `@app.route()`
3. **Return Format**: JSON response with HTTP status code
4. **Docstrings**: Document what each endpoint does

#### `app/auth/routes.py` - Authentication Blueprint

```python
from flask import Blueprint

auth_bp = Blueprint('auth_bp', __name__, url_prefix='/auth')

@auth_bp.route('/', methods=['GET'])
def auth_page():
    """Endpoint to handle the home route."""
    return "Welcome to the home page!", 200
```

**Teaching Notes:**

1. **URL Prefix**: All routes in this blueprint will be prefixed with `/auth`
2. **Naming Convention**: Clear, descriptive blueprint names
3. **Separation of Concerns**: Authentication logic separated from main application

### `app/api/routes.py` - API Blueprint

```python
from flask import Blueprint

api_bp = Blueprint('api_bp', __name__, url_prefix='/api')

@api_bp.route('/status', methods=['GET'])
def status():
    """Endpoint to check the API status."""
    return "status:API is running with no issue!", 200
```

**Teaching Notes:**

1. **API Prefix**: `/api` prefix clearly identifies API endpoints
2. **Status Endpoint**: Common pattern for health checks
3. **RESTful Design**: Following REST conventions for API design

---

## Step 5: Application Factory Pattern

### `app/__init__.py` - Application Factory

```python
from flask import Flask
from app.extensions import db, migrate
from config import Config

def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    # Initialize extensions with app instance
    db.init_app(app)
    migrate.init_app(app, db)
    # Register blueprints
    from app.main.views import main_bp
    from app.auth.routes import auth_bp
    from app.api.routes import api_bp

    app.register_blueprint(main_bp)
    app.register_blueprint(auth_bp, url_prefix='/auth')
    app.register_blueprint(api_bp, url_prefix='/api')

    # Create database tables
    with app.app_context():
        db.create_all()
    print("Creating database tables...")

    return app
```

**Teaching Notes:**

1. **Factory Function**: Creates and configures Flask app instance
2. **Configuration Loading**: `app.config.from_object()` loads settings
3. **Extension Binding**: `extension.init_app(app)` binds extensions to app
4. **Blueprint Registration**: Makes blueprint routes available to the app
5. **Application Context**: Required for database operations
6. **Database Creation**: Automatically creates tables on startup

**Benefits of Application Factory:**

- **Testing**: Easy to create app instances with different configurations
- **Multiple Environments**: Different configs for dev/test/prod
- **Extension Management**: Clean initialization order

---

## Step 6: Application Entry Point

### `run.py` - Development Server

```
from app import create_app

app = create_app()
```

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

**Teaching Notes:**

1. **Entry Point**: How the application starts
2. **Debug Mode**: Enables auto-reloading and detailed error pages
3. **Development Server**: Flask's built-in server for development only
4. **Production Note**: Never use `debug=True` in production

---

## Step 7: Understanding the File Structure

### Empty `__init__.py` Files

```
# app/main/__init__.py
# app/auth/__init__.py
# app/api/__init__.py
# app/models/__init__.py
```

**Teaching Notes:**

1. **Python Packages**: `__init__.py` makes directories into Python packages
2. **Import Path**: Enables `from app.main import something`
3. **Empty Files**: Can be empty but must exist for package recognition
4. **Future Expansion**: Placeholder for package-level imports and initialization

---

## Step 8: Database Integration

### Current Database Setup

The application creates a SQLite database automatically:

- **Location**: `instance/county_services.db`
- **Creation**: Automatic via `db.create_all()` in application factory
- **Management**: Flask-Migrate handles schema changes

### Key Database Concepts:

1. **ORM**: Object-Relational Mapping translates Python objects to database tables
2. **Migrations**: Version control for database schema
3. **SQLite**: File-based database perfect for development
4. **Instance Folder**: Special Flask folder for instance-specific files

---

## Step 9: Testing the Application

### Running the Application

```
# Navigate to project directory
cd /home/chatelo/Documents/Africode_Academy/fs_prep

# Install dependencies
pip install -r requirements.txt

# Run the application
python run.py
```

### Testing Endpoints

1. **Home Page**: `GET http://localhost:5000/`
2. **Auth Page**: `GET http://localhost:5000/auth/`
3. **API Status**: `GET http://localhost:5000/api/status`

---

## Step 10: Key Architectural Decisions

### 1. Modular Design

- **Benefits**: Easy to maintain, test, and scale
- **Trade-offs**: More files and complexity for simple apps
- **Best For**: Applications expected to grow

### 2. Blueprint Organization

- **By Feature**: `auth/`, `api/`, `main/`
- **Benefits**: Clear separation of concerns
- **Scalability**: Easy to add new features

### 3. Configuration Management

- **Environment-based**: Different settings for different environments
- **Security**: Sensitive data in environment variables
- **Flexibility**: Easy to modify without code changes

### 4. Extension Pattern

- **Lazy Loading**: Extensions initialized separately from app creation
- **Testing**: Easy to mock or replace extensions
- **Modularity**: Pick and choose needed functionality

---

## Step 11: Next Steps and Extensions

### Immediate Enhancements

1. **Templates**: Add HTML templates for web pages
2. **Forms**: Implement WTForms for user input
3. **Models**: Define database models in `app/models/`
4. **Authentication**: Implement user registration and login
5. **Error Handling**: Add custom error pages

### Advanced Features

1. **User Roles**: Implement role-based access control
2. **API Authentication**: Add token-based authentication
3. **Testing**: Unit and integration tests
4. **Deployment**: Production deployment configuration
5. **Logging**: Application logging and monitoring

---

## Step 12: Common Patterns and Best Practices

### 1. Import Organization

```
# Standard library imports
import os
```

```python
from datetime import datetime

# Third-party imports
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

# Local application imports
from app.extensions import db
from app.models.user import User
```

### 2. Error Handling

```python
@app.errorhandler(404)
def not_found_error(error):
    return jsonify({'error': 'Not found'}), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return jsonify({'error': 'Internal server error'}), 500
```

### 3. Configuration Classes

```python
class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')

class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///dev.db'

class ProductionConfig(Config):
    DEBUG = False
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

---

## Summary

This lesson covered the fundamental concepts of building a modular Flask application:

1. ✅ **Project Structure**: Organized, scalable file organization
2. ✅ **Blueprints**: Modular application components

3. ✅ **Application Factory**: Flexible app creation pattern
4. ✅ **Configuration Management**: Environment-based settings
5. ✅ **Database Integration**: SQLAlchemy ORM setup
6. ✅ **Extension Management**: Lazy loading of Flask extensions

### Key Takeaways:

- **Modularity**: Break applications into logical components
- **Separation of Concerns**: Each module has a specific purpose
- **Scalability**: Structure supports growth and team development
- **Best Practices**: Follow Flask conventions and patterns
- **Maintainability**: Clear, organized code structure

### Next Class Preview:

In the next lesson, we'll extend this foundation to build database models, implement user authentication, and create a complete web application with templates and forms.

---

## Additional Resources

### Documentation

- Flask Documentation
- Flask-SQLAlchemy Documentation
- Blueprint Documentation

### Best Practices

- Flask Patterns
- Application Factories
- Large Applications

---

**Remember**: This is a foundation. Real-world applications require additional considerations like security, testing, logging, and deployment strategies. The modular structure we've built here makes adding these features much easier!