Flask Database & ORM Comprehensive Reference

A complete reference document capturing theory, analogies, and fully commented code examples for all database- and ORM-related topics covered.

---

1. Flash Messages with Categories

Theory: Flask's `flash()` stores messages in the user session. By default, messages are simple strings. When you pass `with_categories=True` to `get_flashed_messages()`, you retrieve `(category, message)` pairs, enabling differentiated styling and behavior.

```
# In your Flask view:
from flask import flash

# Flash an error message under category 'error'
flash('Invalid email address', 'error')
# Flash a success message under category 'success'
flash('Profile updated!', 'success')

# In your Jinja2 template:
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class="flashes">
      {% for category, msg in messages %}
        <!-- 'category' becomes a CSS class, e.g. 'error' or 'success' -->
        <li class="{{ category }}">{{ msg }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

- **Flash flow:** View → `flash()` writes to session → Redirect → Template reads via `get_flashed_messages()`.

---

2. Jinja2 Block Naming & Inheritance

Theory: Blocks define replaceable sections in base templates. Child templates use the same block names to override content.

```
<!-- base.html: defines a 'content' block -->
<!DOCTYPE html>
```

```
<html>
  <body>
    {% block content %}
      <!-- default content if child doesn't override -->
    {% endblock %}
  </body>
</html>

<!-- page.html: extends base and replaces 'content' -->
{% extends 'base.html' %}
{% block content %}
  <h1>Page Title</h1>
  <p>This content appears in place of the base block.</p>
{% endblock %}
```

- **Key point:** Block names must match exactly between base and child.

---

3. `Flask(__name__)` and Project Structure

Theory: The `import_name` argument (`__name__`) tells Flask where your module resides. Flask uses this to locate the `templates/` and `static/` folders relative to your app's root path.

```
from flask import Flask
app = Flask(__name__)
# Internally:
# module = sys.modules[import_name]
# app.root_path = os.path.dirname(module.__file__)
# templates_folder = os.path.join(app.root_path, 'templates')
# static_folder    = os.path.join(app.root_path, 'static')
```

- **Result:** You don't need to hardcode paths; Flask finds assets based on your code's location.

---

4. Environment Configuration via `.env`

Theory: Flask's CLI auto-loads environment variables from a `.env` file (using python-dotenv). Store config and secrets there.

```
FLASK_APP=app.py    # Entry point
FLASK_ENV=development  # Enables debug mode
SECRET_KEY=supersecretkey  # For sessions & CSRF
```

- **Security:** Add `.env` to `.gitignore` if it contains sensitive values.

---

5. SQLite vs. SQLAlchemy ORM

Aspect	sqlite3 (DB-API)	SQLAlchemy ORM
Level	Low-level database driver	High-level toolkit + ORM
Query style	Raw SQL strings	Python class methods & queries
Portability	SQLite only	ORM abstracts DB backend
Safety	Manual parameter binding	Automatic parameter binding
Schema mgmt	Manual `CREATE TABLE`	Automatic via models & migrations

```python
# Raw sqlite3 example:
import sqlite3
conn = sqlite3.connect('blog.db')
cur  = conn.cursor()
cur.execute('SELECT * FROM posts WHERE id=?', (1,))
rows = cur.fetchall()
conn.commit()
conn.close()
```

```python
# SQLAlchemy ORM example:
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy(app)
class Post(db.Model):  # Declarative model
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(150), nullable=False)

post = Post.query.get(1)  # ORM query
print(post.title)
```

---

6. Database Cursors Explained

Theory: A cursor is a DB API object that executes SQL and fetches results.

```
# Using sqlite3
conn = sqlite3.connect('blog.db')
cur = conn.cursor()             # Create cursor tied to connection
cur.execute("""
  CREATE TABLE IF NOT EXISTS posts (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL)
""")                           # Execute DDL
conn.commit()                    # Persist schema change
cur.execute('SELECT * FROM posts')  # Execute DML
for row in cur:                  # Iterate over result rows
    print(row)
cur.close()                      # Clean up cursor
conn.close()                     # Close connection
```

- **Responsibilities:** Sending SQL, managing parameter binding, buffering results.

---

7. Splitting GET vs. POST Views

Theory: You can handle form display and submission in one view (dual-method) or split into two single-method views.

Combined view

```
@app.route('/login', methods=['GET','POST'])
def login():
    if request.method == 'POST':            # Form submission
        # process credentials
    else:                                   # Initial GET
        return render_template('login.html')
```

Separate views (Flask 3.0+)

```
@app.get('/login')
def show_login():
    return render_template('login.html')

@app.post('/login')
```

```
def process_login():
    username = request.form['username']
    # authenticate and redirect
```

- **Advantage:** Clear separation of concerns; no `if` branching inside the view.

---

8. SQLAlchemy Declarative Base & Class Registry

Theory: `declarative_base()` returns a base class (`Base`) that:

1. **Collects metadata** (tables & columns) for table generation.
2. **Registers model classes** (supports polymorphism & lookups).
3. **Resolves relationships** when you use string model names.

```
from sqlalchemy.orm import declarative_base
Base = declarative_base()

class Post(Base):
    __tablename__ = 'posts'
    id = Column(Integer, primary_key=True)
    title = Column(String)
```

- **Analogy:** `Base` is a factory's blueprint board. Every model subclass pins its blueprint there, so the factory (SQLAlchemy) knows what to build.

---

9. Core vs. ORM Comparison

Theory:

- **Core:** You build SQL expressions manually, similar to writing raw SQL in Python.
- **ORM:** You map Python classes to tables, working with objects instead of query strings.

Core example:

```
from sqlalchemy import select
stmt = select(posts_table).where(posts_table.c.id == 1)
with engine.connect() as conn:
    row = conn.execute(stmt).fetchone()
    print(row['title'])
```

ORM example:

```
post = session.query(Post).get(1)
print(post.title)
```

---

10. Relationships in Detail

One-to-Many (User → Posts)

```
class User(Base):
    __tablename__ = 'users'
    id    = Column(Integer, primary_key=True)
    # 'posts' returns list of Post objects where Post.author_id == User.id
    posts = relationship('Post', back_populates='author')

class Post(Base):
    __tablename__ = 'posts'
    id        = Column(Integer, primary_key=True)
    author_id = Column(Integer, ForeignKey('users.id'))  # enforces referential
integrity
    # 'author' returns the single User object matching author_id
    author    = relationship('User', back_populates='posts')
```

- **Flow:**
- `ForeignKey` sets up SQL schema link.
- `relationship` creates Python properties for navigation.
- `back_populates` names must match on both sides.

Many-to-Many (Posts ↔ Tags)

```
# Association table: no model class needed
post_tags = Table(
    'post_tags', Base.metadata,
    Column('post_id', ForeignKey('posts.id'), primary_key=True),
    Column('tag_id',  ForeignKey('tags.id'), primary_key=True)
)

class Post(Base):
    __tablename__ = 'posts'
    id   = Column(Integer, primary_key=True)
    title= Column(String)
    # 'tags' lists Tag objects via post_tags linking
    tags = relationship('Tag', secondary=post_tags, back_populates='posts')
```

```
class Tag(Base):
    __tablename__ = 'tags'
    id   = Column(Integer, primary_key=True)
    name = Column(String, unique=True, nullable=False)
    # 'posts' lists Post objects via the same association
    posts= relationship('Post', secondary=post_tags, back_populates='tags')
```

- ``: tells SQLAlchemy which intermediate table to use.
- ``: ensures two-way sync of object lists.

---

11. Model Methods & Mixins

Adding Behavior to Models

```
class Post(Base):
    __tablename__ = 'posts'
    id           = Column(Integer, primary_key=True)
    title        = Column(String, nullable=False)
    body         = Column(Text, nullable=False)
    is_published = Column(Boolean, default=False)

    def publish(self):
        """
        Mark post as published and save to DB.
        Raises ValueError if already published.
        """
        if self.is_published:
            raise ValueError("Post is already published.")
        self.is_published = True
        session = Session.object_session(self)  # get current DB session
        session.add(self)                       # mark object changed
        session.commit()                        # persist UPDATE

    def __repr__(self):
        return f"<Post id={self.id} title={self.title!r}>"  # helpful debug info
```

Reusable Mixins

```
class TimestampMixin:
    """
    Adds automatic created_at and updated_at timestamps.
    """
    created_at = Column(DateTime, default=datetime.utcnow)
```

```
    updated_at = Column(
        DateTime,
        default=datetime.utcnow,
        onupdate=datetime.utcnow  # auto-update timestamp on row change
    )

class Post(TimestampMixin, Base):
    __tablename__ = 'posts'
    id    = Column(Integer, primary_key=True)
    title = Column(String, nullable=False)
    body  = Column(Text, nullable=False)
```

- **Mixin benefit:** Write timestamp logic once, apply to any model.

---

12. Data Validation Patterns

Property Setters

```
class User(Base):
    __tablename__ = 'users'
    _email = Column('email', String, nullable=False)

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        """
        Validate email format and normalize.
        """
        cleaned = value.strip().lower()
        if '@' not in cleaned:
            raise ValueError("Invalid email address")
        self._email = cleaned
```

- **Mechanism:** Python properties intercept attribute access/assignment to enforce rules.

WTForms Integration

```
from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, BooleanField
from wtforms.validators import DataRequired, Length
```

```
class PostForm(FlaskForm):
    title = StringField(
        'Title',
        validators=[DataRequired(message="Title cannot be empty."),
Length(max=150)]
    )
    body  = TextAreaField(
        'Body',
        validators=[DataRequired(message="Body cannot be empty.")]
    )
    is_published = BooleanField('Publish now')
```

- **Usage in view:**

```
form = PostForm()
if form.validate_on_submit():
    # Access sanitized form.title.data, form.body.data
```

- **Benefit:** Consolidates form rendering and validation rules.

End of Comprehensive Reference