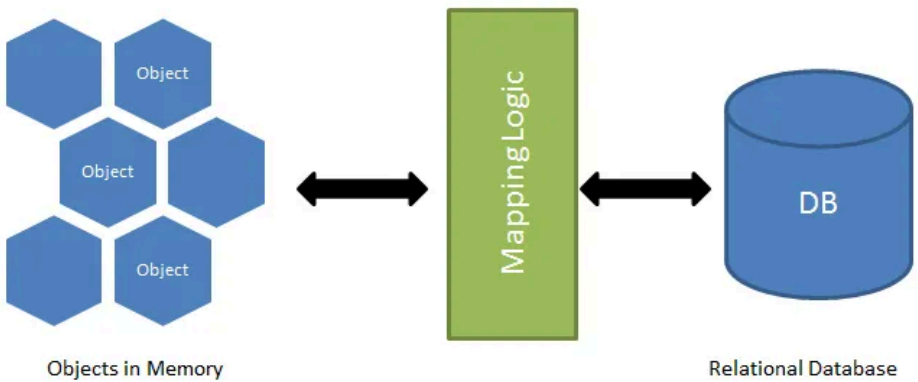
Object Relational Mappers

CST 363

What is an ORM?

- Description of how an object model "maps" to a relational data model
  - Class ⇨ table(s)
  - Object properties (attributes) ⇨ columns or relations



Two Worlds That Do Not Match Perfectly

- Application code:
  - In-memory, fast, short-lived
  - Structured as objects, classes, methods
- Corporate data:
  - Persistent, long-lived
  - Stored in relational databases (RDBMS)
- These worlds “do not speak the same language” out of the box

The Object-Oriented View (in Code)

- We model the domain with:
  - Classes and inheritance
  - Methods and behavior
  - Object references and collections
- We think in terms of:
  - `Customer` has many `Orders`
  - `Order` has many `LineItems`
  - `Product` is a superclass of `Book`, `Movie`, etc.

The Relational View (in the Database)

- We store data using:
  - Tables (relations)
  - Rows (tuples)
  - Columns (attributes)
- Relationships via:
  - Primary keys and foreign keys
  - Join tables (for many-to-many)
- Querying via SQL:
  - Set-based, declarative operations (`SELECT` , `JOIN` , `GROUP BY` , ...)

The Object–Relational Mismatch

- Objects:
  - Identity via object references
  - Rich graphs, cycles, inheritance
- Relations:
  - Identity via primary keys
  - Flat rows, joins for relationships
- Result:
  - “Impedance mismatch” between:
    - How we write code
    - How we store and query data

What an ORM Actually Does

- Maps between objects and relational data:
  - Class ↔ Table
  - Object instance ↔ Row
  - Field/property ↔ Column
  - Object reference ↔ Foreign key / join table
- Responsibilities:
  - Generate SQL for CRUD operations
  - Manage identity and caching (first-level cache)
  - Handle transactions (often integrated with app framework)
- Net effect:
  - Gives the *illusion* of working with an “object database” over an RDBMS

Why Not Just Use an Object Database?

- Pure Object Databases (OODBMS) did not become mainstream:
  - Weak standardization (limited interoperability)
  - Poor fit for analytics and reporting
  - Harder to do ad-hoc, set-based queries over large datasets
- Relational model won for business data:
  - Strong vendor support (IBM, Oracle, Microsoft, PostgreSQL ecosystem)
  - SQL as a common, portable query language
  - Mature tooling for Business Intelligence (BI), reporting, analytics

Schema Design Still Matters

- ORMs do not eliminate the need for good schema design:
  - A messy schema → complicated, fragile mappings
  - A clean schema → simpler, more predictable ORM usage
- Normalization:
  - Reduce redundancy and update anomalies
  - Typically target Third Normal Form (3NF) for core transactional data
- Example:
  - Instead of one big `ordered_books` table with repeated customer/book data:
    - `customers`
    - `books`
    - `orders`
    - `order_lines`

ORMs: Benefits and Trade-Offs (High-Level)

- Benefits:
  - Faster development for common CRUD
  - Less boilerplate SQL in application code
  - Stronger alignment with the domain model (DDD, rich domain objects)
- Trade-offs:
  - Generated SQL can be non-obvious and sometimes inefficient
  - Complex queries may still require hand-written SQL
  - Easy to forget the cost of queries and N+1 problems
- Important mindset:
  - ORM is a tool, not magic
  - You still need to understand the relational model and SQL

Security: SQL Injection

```
def login():
    username = request.args.get('username', '')
    password = request.args.get('password', '')

    conn = get_db()
    cur = conn.cursor()

    # Unsafe: directly interpolating user input into SQL
    query = (
        f"SELECT id, username "
        f"FROM users "
        f"WHERE username = '{username}' "
        f"  AND password = '{password}';"
    )
    cur.execute(query)
    user = cur.fetchone()
    conn.close()
```

Why this is dangerous?

An attacker can craft a URL like:

```
/login?username=admin'--&password=anything
```

which makes the SQL look like:

```
SELECT id, username
FROM users
WHERE username = 'admin'--'
  AND password = 'anything';
```

- Everything after `--` is treated as a comment, so the `AND password = ...` check is skipped.
  - effectively logs them in as “admin” without knowing the password.

Demonstration of a Classic Injection

Bypass authentication:

```
/login?username=' OR '1'='1&password=' OR '1'='1
```

```
SELECT id, username
FROM users
WHERE username = '' OR '1'='1'
  AND password = '' OR '1'='1';
```

Since `'1'='1'` is always true, the query returns the first user.

Destructive commands

```
/login?username=anything'; DROP TABLE users;--&password=x
```

```
SELECT id, username
FROM users
WHERE username = 'anything';
DROP TABLE users;--'
  AND password = 'x';
```

This deletes your entire users table.

How an ORM can help here

- Built-in Parameterization
  - Query API parameters are automatically bound for you
    - you never string-interpolate user input into SQL. Injection risks become effectively zero.
- Domain-centric Code
  - Instead of hand-writing raw SQL, you work with, e.g., Python classes and objects.
- Cross-Database Portability
  - You write the same ORM-based code whether you're on SQLite, PostgreSQL, MySQL, or Oracle.
    - No conditional SQL syntax in your application logic.

SQLAlchemy: Two Layers

Python's arguably best ORM

1. **Core** – thin, explicit SQL construction layer.
   - Think "Pythonic SQL string-builder."
   - Includes schema management functions, SQL query builder, and schema inspection utilities

2. **ORM** – builds on Core, adds identity map, sessions, and Python classes that map to tables.

> We'll use the 2.0 style API (released 2023), which is declarative-by-default and async-friendly.

Getting Set Up

With virtual environment activated: `pip install sqlalchemy psycopg[binary] alembic`

- Postgres must be running
- Create a database

Connecting

```
from sqlalchemy import create_engine

engine = create_engine(
    "postgresql+psycopg://postgres:ott3r@localhost:5431/my_db",
    echo=True
)
```

- This is based on our existing PostgreSQL Docker container and a database called `my_db`
- `engine` is cheap; create once, reuse everywhere.
- Alembic is an extension for migrations and schema management
  - Tools like Alembic allow you to version and evolve your schema alongside your code, instead of manually writing `ALTER TABLE` scripts

Declaring Models (1 of 3)

```
from typing import List
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
from sqlalchemy import ForeignKey, Text


class Base(DeclarativeBase):
    pass
```

- `DeclarativeBase` – to define your base class – a single shared foundation for all models
- `Mapped[ ... ]` – marks an attribute as a mapped column or relationship
- `mapped_column( ... )` – defines a column
- `relationship( ... )` – defines ORM relationships between tables

Declaring Models (2 of 3)

```python
class User(Base):
    __tablename__ = 'users'

    id: Mapped[int] = mapped_column(primary_key=True)
    username: Mapped[str] = mapped_column(nullable=False)
    email_address: Mapped[str] = mapped_column(nullable=True)
    comments: Mapped[List["Comment"]] = relationship(back_populates="user")

    def __repr__(self) -> str:
        return f"<User username={self.username!r}>"
```

- `back_populates` creates a bidirectional relationship between `User` and `Comment`
  - It connects both sides of a relationship explicitly, so changes on one side are reflected on the other

Declaring Models (3 of 3)

```
class Comment(Base):
    __tablename__ = 'comments'

    id: Mapped[int] = mapped_column(primary_key=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id"), nullable=False)
    text: Mapped[str] = mapped_column(Text, nullable=False)
    user: Mapped["User"] = relationship(back_populates="comments")

    def __repr__(self) -> str:
        return f"<Comment text={self.text!r} by {self.user.username!r}>"
```

Creating the Schema

Generates and runs `CREATE TABLE ... SQL`

```
Base.metadata.create_all(bind=engine)
```

Unit of Work: Sessions

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    # create a user plus a comment in one go
    new_user    = User(username="joe", email_address="joe@example.com")
    new_comment = Comment(text="First post!", user=new_user)

    session.add(new_user)   # new_comment is cascaded via the relationship
    session.commit()        # both INSERTs happen here
```

Key ideas:

1. **Identity map** – within a session, each row is represented by *one* Python object.
2. **Lazy vs. eager loading** – accessing `customer.orders` may trigger a SELECT.
3. **Transactions** – `Session` wraps every flush in a database transaction.

Querying

```
from sqlalchemy import select

with Session(engine) as session:
    stmt = (
        select(Comment)
        .join(Comment.user)
        .where(User.username == "joe")
        .order_by(Comment.id.desc())
        .limit(5)
    )
    recent_comments = session.scalars(stmt).all()
    # → gives you a List[Comment]
```

- `select()` → SQL is built safely.
- `.scalars()` flattens the `Row` objects to the first column (the `Order`).

Lazy, Eager, and "N + 1"

The "N + 1" problem is a common performance pitfall when using an ORM with lazy-loaded relationships.

- One query to load the parent objects
- N additional queries to load each child collection

> Hence, 1 + N queries total—every time you iterate, you pay the cost of a separate round-trip to the database.

How it happens

```
with Session(engine) as session:
    users = session.scalars(select(User).limit(3)).all()
    for u in users:
        # Lazy: when you first access `u.comments`, SQLAlchemy issues:
        #   SELECT * FROM comments WHERE comments.user_id = <u.id>
        print([c.text for c in u.comments])
```

1. Query #1
   `SELECT * FROM users LIMIT 3;`
2. Query #2 (for user 1)
   `SELECT * FROM comments WHERE user_id = 1;`
3. Query #3 (for user 2)
   `SELECT * FROM comments WHERE user_id = 2;`
4. Query #4 (for user 3)
   `SELECT * FROM comments WHERE user_id = 3;`

N + 1 (cont.)

That’s four queries to get three users and their comments.

- Round-trip latency multiplies with each extra query.
- Under load, the database sees a storm of tiny queries rather than a few bigger ones.
- Becomes especially painful when N (number of parents) grows.

> Use eager loading so that related rows are fetched in bulk

Mitigate the N + 1:

```
from sqlalchemy.orm import selectinload

stmt = (
    select(User)
    .options(selectinload(User.comments))
    .limit(3)
)

with Session(engine) as session:
    users = session.scalars(stmt).all()
    for u in users:
        print([c.text for c in u.comments])
        # ← no extra queries: comments are loaded in bulk
```

SQLAlchemy + Pandas ❤️‍🔥

- A Pandas `DataFrame` is a 2D labeled data structure in Python (like a spreadsheet or SQL table)
  - a collection of rows and columns with mixed data types (int, float, string, etc.).
  - Built on top of NumPy, optimized for performance and memory.
  - Commonly used for data analysis, cleaning, filtering, grouping, and visualization prep.
  - Can be created from: Lists or dictionaries, CSV, Excel, SQL databases, etc.
- Common use cases of this powerful combo:
  - Read SQL query results into a `DataFrame`
  - Write a `DataFrame` into a database table
  - Run analytical queries directly with Pandas on top of your database

Example 1: Read from database into DataFrame

```
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine("postgresql+psycopg2://user:pass@localhost/dbname")

df = pd.read_sql("SELECT * FROM users", con=engine)
```

- Under the hood, Pandas uses SQLAlchemy to connect.
- You can also pass a full select() object from SQLAlchemy Core.

Example 2: Write DataFrame to database

```
df.to_sql("users", con=engine, if_exists="replace", index=False)
```

- This creates or replaces a table called `users`.
- `if_exists="append"` adds rows to the table instead.

Example 3: Use ORM to get data, then convert to DataFrame

```
from sqlalchemy.orm import Session
from models import User

with Session(engine) as session:
    users = session.query(User).all()

# Convert list of ORM objects to DataFrame
df = pd.DataFrame([{
    "id": user.id,
    "username": user.username,
    "email": user.email_address
} for user in users])
```

- This is useful if you're using the ORM and want full control over fields.

Migrations with Alembic (Brief)

1. `alembic init alembic`
2. Edit `alembic.ini` to point at the same URL.
3. `alembic revision --autogenerate -m "add order tables"`
4. `alembic upgrade head`

By putting every schema change into a migration script and committing it, you:

- Have a clear, chronological history of how your schema evolved.
- Can reproduce the same database structure on any machine (dev, test, CI, production).
- Avoid accidental data loss or “works on my machine” drift.

When to *Skip* the ORM

- Use the ORM for everyday create-read-update-delete and straightforward query work—its object-mapping and identity-tracking save you boilerplate.
- When you need full control of SQL, want bulk-friendly operations, or are squeezing out every last microsecond of performance, reach for Core (or plain SQL strings) instead.

Trade-offs at a Glance

Strengths	**Watch out for…**
Faster development & fewer lines of code	Hidden queries → *N + 1* problems
Compile-time model validation	Harder to squeeze the last % of performance
Safe parameter binding (no SQL-injection)	Learning curve: sessions, identity map
Easy migrations via Alembic	May tempt you to ignore good schema design

Further Reading & Exercises

Reading – SQLAlchemy 2.0 tutorial sections: ORM Quick Start, Relationship Patterns.