Object-Relational Mapping

Reading / References

- SQLAlchemy ORM Tutorial[1]
- ORM Object states[2]
- Relationship loading strategies[3]

Notes

ORM Philosophy

In most applications we want to work and think with **objects** and the **relationships** between them. So for instance we might have *student objects* and *course objects* and also *enrollment objects* which contain in them a student object and a course object and maybe some more information about them. We may want to change the information of an enrollment for a student, for example when they withdraw from a class. We might want to check that each student is enrolled in a specific number of courses, and never exceed 4.5 credits, etc.

This is all what we might call the **business logic** of our application. This is the level at which we want to think about our application, and we want to write code that talks in these terms, so that it is easy for someone to read that code and understand the key business decisions. We often also refer to this understanding of our main objects as the **domain model**.

By contrast, at some point we need to *persist* our data to a database. Databases typically don't talk in terms of "objects" and "relationships". They have tables, columns, primary keys, tuples, foreign keys and a host of other limitations. If possible, we would rather not have to program in those terms.

This is where Object-Relational Mapping enters. After some initial setup, ORM takes care of the nitty gritty details of converting between "business logic" and "database queries". For example it converts the statement "add this student to the system" to an appropriate INSERT query that would actually carry out the creation.

This comes at a price: There are often complicated situations that are somewhat slow in an ORM setting but would be faster if we were to write the SQL query directly. The good news is that most ORM systems live in harmony with more "core" systems, so you can write the majority of your application in ORM form, only occasionally resorting to more direct queries.

This also comes with tremendous benefits. For example, if you suddenly want to completely change your storage, and move from one kind of database to a completely

[1] https://docs.sqlalchemy.org/en/latest/orm/tutorial.html

[2] https://docs.sqlalchemy.org/en/latest/orm/session_state_management.html#session-object-states

[3] https://docs.sqlalchemy.org/en/latest/orm/loading_relationships.html

different kind that maybe doesn't use SQL queries the same way, then most of your business logic does not need to change at all.

In an ORM setting you typically *declare* a **mapping** between a certain class in your application and a certain table. You then leave many of the details up to the system to handle.

ORM Elements

Sessions In ORM our **domain model** consists of a set of **classes** that implement the required behavior. Each class is typically **mapped** to a database table by means of a **mapper**. Our classes typicaly inherit from a base class that is responsible for handling that mapping behavior and other similar setup functionalities.

Work in ORM is maintained via a **session**. A session represents a group of operations that need to happen. It contains information about new objects that are to be created, updates to existing objects, as well as objects that are to be deleted. This is often implemented by the so called **unit of work** pattern.

Changes that exist in a session are not written to the database yet, they are just scheduled to be written. When a **commit** (typically called a **flush**) occurs, every change in the session will be converted to appropriate SQL queries and will be executed. We can also "back out" of the changes in the session by doing a **rollback**.

Object States Objects in an ORM application can be in one of five **states** at any given time (see here[4]):

transient Transient objects are not in any session, nor saved in the database. This is the state that objects are in when we first create them.

pending Pending objects have been added to a session (or have changes to them in the session), but the session has not been *flushed* yet. So these changes exist in your program and your session, but they have not been stored to the database yet. If our application was to terminate abruptly, these objects would be lost.

persistent Persistent objects are in the session and have been flushed to the database at some point. So they have a certain state in the database. They may still have some local changes that have not been flushed yet, and therefore differ from the corresponding database entries. The objects are typically called **dirty** at this point.

deleted Deleted objects are scheduled for deletion in the session, but a flush hasn't occured yet. These objects may go back to being persistent if a *rollback* happens.

detached Detached objects may correspond to entries in the database, but are currently not in any session. As such, their relation to database information is unknown. Deleted objects pass to the detached state after a flush occurs.

The terminology may differ slightly from system to system.

[4] https://docs.sqlalchemy.org/en/latest/orm/session_state_management.html#session-object-states

Relationships Another essential component of an ORM system is the relationships between classes. There are roughly three kinds of relationships, which differ in the number of elements that can map from each side to the other.

- **one-to-one** relationships occur when one object/entity from the one class can correspond to *at most one* object/entity from the other class. They are not very common. But imagine for example that every department at the college typically has a *department head* which is an instructor. And the same instructor cannot be the head of more than one department. But on occasion a department may be "head-less". So a one-to-one relationship should maybe more correctly be called "one-to-zero-or-one".
- **one-to-many** and **many-to-one** relationships are one of the two most common types of relationships. In these relationships one entity from one class can relate to many entities from the other class, but not the other way around. For example each academic term (e.g. Fall 2018) has a number of course sections related to it. But the converse is not true: Each course section is offered at a specific academic term.
- **many-to-many** relationships are the other most common type of relationship, and they are considerably harder to implement. They occur in instances where each entity from the one class may relate to many entities of the other class, and vice versa. As an example, Imagine the relationship between students and course sections. Each student enrolls in multiple sections, and also each section has multiple students enrolled in it.

While this section in general avoids discussing concepts at the level of the database, it is worth mentioning how these three kinds of relationships would typically be represented in the database.

one-to-one Foreign key constraint along with a unique constraint

```
CREATE TABLE departments (
    ...
    head  VARCHAR(40)  UNIQUE  FOREIGN KEY REFERENCES instructors(login),
    ...
)
```

many-to-one Foreign key without a unique constraint

```
CREATE TABLE sections (
    ...
    term INT      FOREIGN KEY REFERENCES  terms(id),
    ...
)
```

many-to-many Using a third "associations" table that links through foreign keys to both tables

```
CREATE TABLE enrollments (
    ...
    section INT   FOREIGN KEY REFERENCES  sections(id),
```

```
        student INT   FOREIGN KEY REFERENCES  students(id),
        ...
    )
```

When working with relationships in an ORM setting, you typically can access the relationship from one end or the other. So for example in the many-to-many example above, a student object may actually contain a list of the section objects that the student is enrolled in, and conversely a section object may contain a list of the students enrolled in it.

When you set up relationships, you have an option to make then **one-directional** or **bi-directional**. Our example above is bi-directional: We can access the relationship from either end. But for example we could easily imagine a setup where section objects do not themselves contain a list of all the student objects enrolled in them; we can only access the enrollments by looking at the student side.

ORM In SQLAlchemy

Now we will discuss how ORM is implemented in SQLAlchemy. We need to start with the usual setup of an engine:

```
from sqlalchemy import *
## Set up however you need to
engine = create_engine('sqlite:///:memory:', echo=True)
```

The next essential component is to create a "base class" that knows how to set up ORM. SQLAlchemy provides a system called declarative_base that has all the needed functionality:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Now we will build classes for our entities, and will make them extend the functionality provided by Base.

```
class Student(Base):
    __tablename__ = 'en_students'

    id      = Column(Integer, primary_key = True)
    login   = Column(String(20), unique=True, nullable=False)
    first   = Column(String(20))
    last    = Column(String(20))
    credits = Column(Integer, default=0)
    gpa     = Column(Float(precision=32), default=0)

    def __repr__(self):
        return "Student<%s %s>" % (self.first, self.last)

Student.__table__                # Shows us the corresponding Table construction
Base.metadata.create_all(engine)  # Creates the tables in the database
```

Now let's create a student object. We would do it in a standard way. The Base class that we inherit from provides a simple constructor, that expects key-value pairs and creates corresponding entries from them:

```
student = Student(last="Turing", first="Alan", login="turinga37")
student      # See the printout we defined above via __repr__
student.last
student.id == None      # Student has no id yet.
insp = inspect(student)      # An inspection object for student
insp.transient      # true
insp.persistent     # false
```

In order to work with the system, we need to create a section. First we must create a "session maker" class:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

Then we can use this to create a session when we need one:

```
session = Session()
```

This session can be used to manage the interaction with the database. Let us start by adding our student object to the session:

```
session.add(student)
insp.transient      # false
insp.pending        # true
student.id == None # Still no id
```

If we try to now perform a query, for example, the session will automatically issue a flush and save this student to the database. For example, let's ask for a student whose last name is "Turing":

```
alan = session.query(Student).filter_by(last="Turing").first()
```

There is of course a whole lot more to querying, look at the notes[5].

One important property of ORM systems is known as **identity**: Once an object is known to the system by a specific primary key, any query will return *that exact same object*, and not a different copy of that object. In our instance, the two objects student, which we created directly, and alan, which we obtained from the query, are literally the same object:

```
alan is student
alan.first = "Alamo"
student.first      # Also is Alamo now
insp.persistent    # True
session.dirty      # The name change is not yet saved to the database.
session.flush()
session.dirty      # Empty now
```

NOTE: flush vs commit: In SQLAlchemy a session manages "transactions" with the database. A transaction is a set of changes that the database knows about and remembers and maintains, but it has also given you the option to possibly undo all these changes. A **flush** moves any changes to objects in SQLAlchemy into the current *transaction buffer*, in other words it makes the database aware of them. This

[5] https://docs.sqlalchemy.org/en/latest/orm/tutorial.html#querying

makes future queries know about them and have access to them. However, the transaction is not yet completed. You can complete the transaction by issuing a **commit** (session.commit()). This closes the current transaction and permanently writes the changes to the database. Alternatively, you can use session.rollback() to **roll back** the entire transaction.

Setting up relationships Let's look at the basic relationship patterns[6] we have in SQLAlchemy and how to set them up.

We have already defined a Student class. We will also consider a Course class that represents course sections being offered. We will develop two kinds of relationships between students and courses:

- Students can have one "favorite" course, but the same course can be favorited by many students. This is a many-to-one relationship.
- Students can enroll in courses, and each course can have many students enrolled in it. This is a many-to-many relationship.

We start with the favorite relationship. Here's how it might look:

```
class Course(Base):
    __tablename__ = "en_courses"

    id       = Column(Integer, primary_key=True)
    prefix   = Column(String(4), nullable=False)
    no       = Column(String(20), nullable=False)
    title    = Column(String(55), nullable=False)
    credits = Column(Integer, nullable=False, default=4)
    __table_args__ = (
        UniqueConstraint('prefix', 'no', name="fullCode")
    )
    favoritedBy = relationship("Student", order_by=Student.id,
                               back_populates="favoriteCourse")
    #
    def __repr__(self):
        return "Course<%s%s %s>" % (self.prefix, self.no, self.title)
    #
    def isFullCredit(self):
        return self.credits == 4

# Back in student class:
    # Here are the "favorites" relationship bits
    favorite_id = Column(Integer, ForeignKey("en_courses.id"))
    # And we talk about the relationship:
    favoriteCourse = relationship("Course", back_populates="favoritedBy")
```

So there are two parts to the definition:

- Making sure you have the appropriate foreign keys set up.

[6] https://docs.sqlalchemy.org/en/latest/orm/basic_relationships.html#relationship-patterns

- Using the relationship method to create the actual relationship variables. If you want the relationship to be bidirectional, then you need to define the variables on both sides, like we did above with the favoritedBy and favoriteCourse variables.

We can then access a student's favorite course by doing for example:

```
alan.favoriteCourse
```

And conversely if we have a course c in mind, we can find all the students favoriting it by doing:

```
c.favoritedBy      # A list of students
```

Setting up many-to-many relationships In order to establish a many-to-many relationship, we have to take some extra steps:

- First we create a extra table that holds pairs of related entities, via their primary keys. It will contain foreign keys pointing to the other two entities' tables.
- Then we set up relationships from each of the two entities to that table. We use the secondary=... parameter to the relationship call to point to the table to be used for the linkage.

For example, let's suppose that we wanted the favorites relationship above between students courses and courses to be many-to-many: Each student can favorite more than one course, and each course can be favorited by more than one student. Here's how that might be set up:

```
# We first create a table:
favoritesTbl = Table('en_favorites', Base.metadata,
    Column('student_id', ForeignKey('en_students.id'), primary_key=True),
    Column('course_id', ForeignKey('en_courses.id'), primary_key=True))

# Inside the Student class definition:
    favoriteCourses = relationship("Course",
                                   secondary=favoritesTbl,
                                   back_populates="favoritedBy")

# Inside the Course class definition:
    favoritedBy = relationship("Student",
                               secondary=favoritesTbl,
                               back_populates="favoriteCourses")
```

As an example, of using this, suppose we want to find out all the course that the student bob has favorited. we could do:

```
bob.favoriteCourses
```

And if we want to find out all students who favorite the same course as bob, we could do a list comprehension:

```
[ student
    for course in bob.favoriteCourses
    for student in course.favoritedBy
    if student != bob ]
```

Loading strategies When we're dealing with many relationships, one important consideration is the loading of objects from the database.

For example consider the above setting of students, courses, and students choosing courses as favorites:

- Bob has put down CS220 as a favorite. Zoe has done the same, and so have 100 other students.
- We are interested in retrieving the Bob object. Along with it comes the list of courses that Bob has favorited, via the favoriteCourses relationship. So this brings along the CS220 object.
- But the CS220 object includes the favoritedBy relationship, so for it to be fully loaded would require that we include all those students with it, like Zoe and many more.
- These students also have favorite courses, so to fully load them we'll need to load even more course objects, and their related student objects and so on and so forth.

Clearly this is unsustainable. We need a way to load just the Bob object, along with perhaps an inkling of the fact that there is a list of courses that Bob has favorited, but without fully loading those course objects unless our desired query somehow needs them. There are various **loading strategies** we can employ in such a situation, documented in this page[7]. The broader classification is between **lazy loading**, **eager loading** and **no loading**.

These loading strategies can be set up when the relationship is defined, or they can be activated at specific queries.

lazy loading In lazy loading, there will be no SELECT components created for objects that are not the primary queried object, unless the corresponding object is being accessed by the Python code.

Lazy loading is the default behavior.

joined (eager) loading In joined eager loading, a JOIN statement will be included to the query of the source object, and therefore the targetted collection will be pre-populated.

subquery (eager) loading In subquery eager loading, a subsequent query is generated for the relationships, restricted using the initial SELECT query in the FROM clause as a subquery. This keeps the initial query simple, as it does not involve any joins, but makes the subsequent queries more complicated.

select IN loading In select IN loading, a subsequent query is generated for the relationships, restricted using an IN clause to target the primary keys of the elements in the initial query.

[7] https://docs.sqlalchemy.org/en/latest/orm/loading_relationships.html

raise loading Raise loading behaves similar to lazy loading, except that it simply raises an exception when the extra query is about to happen, to guard against too many unwanted lazy loads.

no loading The no loading setting simply sets the attributed to empty, never loading the corresponding relationship's values.