# 420-951-VA Transaction Web Applications

# Web Programming with Flask

## Author: Denis Rinfret

## ORM and Databases

- **ORM**: *Object-Relational Mapping* (or *Mapper*)
- General steps to use an ORM such as SQLAlchemy, directly or through the Flask-SQLAlchemy wrapper
    i. Define models as classes in Python
    ii. Create the tables corresponding to the models using the ORM
        - this will end up executing SQL statements, such as `CREATE TABLE` statements
        - but you don't have to explicitly write the SQL statements, the ORM will do it for you
    iii. Add instances of the model classes through the ORM
        - `INSERT INTO` statements will be generated and executed by the ORM
    iv. Query the DB through the models or the DB session
        - there are 2 main ways to query the DB:
            a. the legacy interface, using `model.query`
            b. the recommended way (since version 3 of Flask-SQLAlchemy), using `db.session.execute`
        - You don't normally need to use SQL directly while using the ORM, except for some special queries that cannot be (easily) expressed with the ways mentioned above

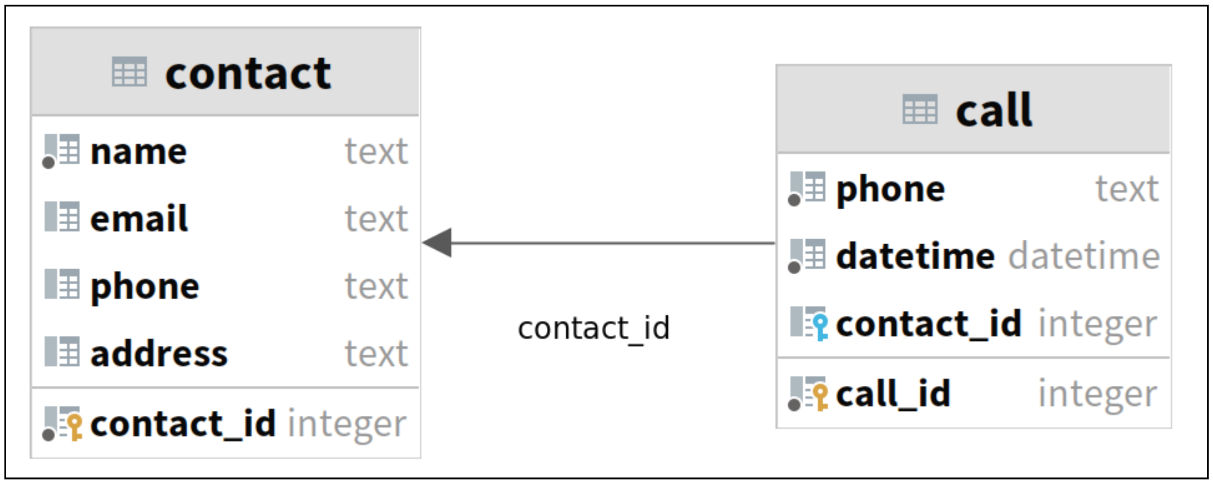
## Project: `flask_mvc`

This is a small project to show how to set up an ORM, define models, and query the models (i.e. query the DB). In this project, the views (the templates) and the controllers (the routes) are kept simple, and the focus is placed on the models. To begin with, there are only 2 models, corresponding to 1 table each, so there are no many-to-many relationships in this example.

## File `models.py`

Doc

Here is a diagram representing the DB:



*contact_call.png*

It is a simplified contacts DB that could be used on a mobile phone. It records the name, the phone number, the address and the email of contacts, and a history phone calls. There's a foreign key from `call` to `contact`, which is optional in case there's no contact associated to a given phone number.

We start by defining the `Contact` model.

```
class Contact(db.Model):
    __tablename__ = 'contact'
    contact_id = db.Column(db.Integer(), primary_key=True, autoincrement=True)
    name = db.Column(db.Text(), nullable=False)
    email = db.Column(db.Text())
    phone = db.Column(db.Text())
    address = db.Column(db.Text())
    calls = db.relationship('Call', backref='contact', lazy=True)

    def __repr__(self):
        return f"<Contact {self.contact_id}: {self.name}>"
```

For now, just ignore the `calls` field, which will be explained later. Setting the `__tablename__` and defining the `__repr__` method is optional, but this way it can make it easier to work with the models in some situations.

We define a field for each attribute in the DB design. Refer to the doc for more details on all the data types available. The constraints are specified as arguments to the `db.Column` constructor. In this model, only the `contact_id` and the name are mandatory, all the other ones are optional (i. e. can be null).

Then we define the `Call` model.

```
class Call(db.Model):
    __tablename__ = 'call'
    call_id = db.Column(db.Integer(), primary_key=True, autoincrement=True)
    phone = db.Column(db.Text(), nullable=False)
    datetime = db.Column(db.DateTime(), nullable=False,
                         default=datetime.datetime.now)
    contact_id = db.Column(db.Integer(), db.ForeignKey('contact.contact_id'))

    def __repr__(self):
        return f"<Call {self.call_id}: {self.phone} {self.contact_id}>"
```

There's an example of setting a default value for a column, in this case the `datetime` column. The default value can be a fixed value, or it can be a function that's going to be called whenever necessary.

The `contact_id` column in the `Call` model is a foreign key to the `Contact` model. A `db.ForeignKey` constraint is given to specify the foreign key. So given a call, it will be easy to find the contact for the call (if any). In some situations, it may be necessary to do the reverse, to know which calls correspond to a specific contact, or in other words, find all the calls with a given `contact_id` . Defining the `calls` field in `Contact` model will make this easy since, given an instance of `Contact` , we will be able to access all the calls associated to the contact through the `calls` field. It is marked as a `backref` since it is, in a sense, the reverse of the foreign key from `Call` .

`calls` is not going to be a column in the `contact` table in the DB. Instead, a query will be executed to find all the calls corresponding to the contact. It is labelled as *lazy* because if not, we could end up query all the calls of all contacts when we may not need them, and it could cause bad performance on large databases. The `calls` field will be available automatically with the legacy query interface, but not with the `db.session.execute` method. More on this later.

## File `manage.py`

This file is used when we start a flask shell.

After creating a new model, if the corresponding table in the DB doesn't already exist, we need to create it. We could write a `CREATE TABLE` statement to create, but it would be easy to make a mistake and end up with a table that doesn't exactly match the model, because of non-matching names or data types for example. The easiest way is to tell SQLAlchemy to automatically create the tables corresponding to our models. It will only create the missing tables, so it is easy to add a new model and create its corresponding table without messing up the existing models/tables.

The `manage.py` file includes a function that will be called automatically if we start a *flask shell*. Open a terminal window in Pycharm, then execute the command `flask shell`. From there, you will be able to call `db.create_all()` to create the tables for the new models without touching the existing tables. Don't forget to commit your changes after creating the tables.

```
from app import app
from models import db, Contact, Call


@app.shell_context_processor
def make_shell_context():
    return dict(app=app, db=db, Contact=Contact, Call=Call)
```

## Adding and querying data through the flask shell

It is possible to add data to the DB through the flask shell. The methods used are the same as the one we will need to use in the routes. The difference is where the data comes from. In the routes, the data might come from the forms, while in the flask shell, we will probably just make up some data for testing. It is also possible to write a script that will add some data to the DB through the ORM.

1. Create an instance of `Contact`: `contact = Contact(name='Denis')`
2. Add it to the DB: `db.session.add(contact)`
3. Repeat if desired, possibly by specifying more column values: `contact = Contact(name='Bob', email='bob@example.com')`
4. Commit your changes: `db.session.commit()`
5. You can also commit after each call to `add`
6. You will not know immediately if there are any issues with the new contacts, for example you will not know if there are broken constraints after you add the new contact
7. You will only know after you commit, that's why it might be better to commit after each `add`
8. You can use `db.session.rollback()` to undo all your changes since the last commit

You can add data to `Call` in a similar fashion.

```
import datetime

db.session.add(Call(phone='1112223333'))
db.session.add(Call(phone='5552223333', contact_id=1))
db.session.add(Call(phone='5552223333',
                    datetime=datetime.datetime(2023, 1, 1),
                    contact_id=1))
db.session.commit()
```

**Querying with the legacy interface**

Doc

With this interface, we need to go through the model to get the data. To get all the contacts, we can do `contacts = Contact.query.all()` . We will get a list of `Contact` objects. Instead of `all()` , we can do `first()` to get only the first result. We won't get a list in this, but just an instance of `Contact` , or `None` if there aren't any results. With `first()` , an empty list will be returned if there aren't any results.

We can use `filter_by` on `query` for simple equality filters, such as `contacts = Contact.query.filter_by(name='Bob').all()` .

With `filter` instead of `filter_by` , we need to refer to the model's fields to specify condition, we cannot use `name` directly. The equivalent query to the previous is `contacts = Contact.query.filter(Contact.name=='Bob').all()` . Note the double equal `==` here. As opposed to `filter_by` , `filter` accepts a condition, not a value for a specific column. Therefore, `filter` is much more flexible than `filter_by` . You can use `where` instead of `filter` if you prefer.

To filter on `null` or `not null` values, since we are using Python, we have to use `None` instead of `null` .

```
contacts = Contact.query.filter(Contact.email == None).all()
contacts = Contact.query.filter(Contact.email != None).all()
contacts = Contact.query.filter(Contact.email.isnot(None)).all()
```

The last 2 are equivalent, and opposite of the first one.

When querying for calls, you can access the `contact_id` field of a call to know which contact it corresponds to. You will get an integer or `None` if there's no associated contact. You can also access the `contact` attribute of a call to get an instance of a contact the call corresponds to. It’s not a real field, there’s no column or columns in `call` containing the full contact data. `contact` in this case is really a getter that’s going to query the `contact` model and build a full instance the corresponding contact.

```
call = Call.query.filter_by(call_id=1).first()
print(call)  # prints <Call 1: 1112223333 None>
print(call.contact_id)  # prints None
print(call.contact)  # prints None
call = Call.query.filter_by(call_id=1).first()
print(call)  # prints <Call 2: 5552223333 1>
print(call.contact_id)  # prints 1
print(call.contact)  # prints <Contact 1: Denis>
```

When querying by primary key, it is more convenient, and possibly faster, to use the `get` method. Because we query by primary key, we know for sure we cannot get more than 1 result, so we get a `Contact` instance, or `None` if the contact was not found. Alternatively, when in a route, you can use `get_or_404` . You use it the same way as `get` , except that instead or returning `None` is not found, a *404 Not Found* will be returned from the route.

```
contact = Contact.query.get(1)
print(contact)  # prints <Contact 1: Denis>
contact = Contact.query.get(4)
print(contact)  # prints None
```

From a contact instance, we can use the `backref` to access all the calls associated to a contact. Here again, the ORM will query the DB to get these associated calls.

```
contact = Contact.query.get(1)
print(contact)  # prints <Contact 1: Denis>
print(contact.calls)  # prints [<Call 2: 5552223333 1>, <Call 3: 5552223333 1>]
```

Although very convenient, we have to be careful using the legacy interface. It can become very inefficient over large databases. Creating model instances can be expensive if the number of rows in the results is large, and/or if the number of columns in the table is large. If, for example, we only need 3 columns of a table out of, say, 50 columns, then creating an instance will require all 50 columns values, even if we are going to use only 3. If the number of rows is large, that’s even worse.

The `backref` is defined as lazy to avoid building the list representing the `backref` all the time, possibly for nothing.

**Querying with `db.session.execute`**

Doc

Doc

The big difference between the two querying methods is that with the legacy interface, we get instances or lists of instances of the model classes, while with `db.session.execute` , we can get list of rows more easily, similar to what we would get if we were executing SQL statements directly on the DB. We still get SQL injection protection, but model instances are not created automatically in all cases. This is less convenient in some cases, but it can be more efficient on large tables and databases since we don’t need to retrieve all columns all the time. And we can control more precisely the queries that are sent to the DB.

In the first example, we get all the calls. If we loop directly on the `result` , we get tuples, each containing 1 `Call` instance.

```
result = db.session.execute(db.select(Call))
for res in result:
    print(res)
# prints
# (<Call 1: 1112223333 None>,)
# (<Call 2: 5552223333 1>,)
# (<Call 3: 5552223333 1>,)
```

If we loop on `result.scalars()` , we get the `Call` instances directly instead.

```
result = db.session.execute(db.select(Call))
for res in result.scalars():
    print(res)
# prints
# <Call 1: 1112223333 None>
# <Call 2: 5552223333 1>
# <Call 3: 5552223333 1>
```

If we don’t want to retrieve all the columns, we can do something like this.

```
result = db.session.execute(db.select(Call.call_id, Call.datetime))
for res in result:
    print(res)
# prints
# (1, datetime.datetime(2023, 1, 4, 21, 22, 16, 777713))
# (2, datetime.datetime(2023, 1, 4, 21, 22, 40, 763128))
# (3, datetime.datetime(2023, 1, 1, 0, 0))
```

Because we did not specify whole models to select, we get tuples of data instead of model instances.

## Joins

With the legacy interface, when we use `call.contact` , we are implicitly doing a join for the specific `Call` instance. Similarly, accessing `contact. calls` involves a join. We can also use the `join` method on the result of query to perform joins, similar to the examples included below using `execute` .

To be able to do any kind of join, we can use a `select` with multiple tables. But be careful, if you don't specify any join conditions, using either the `join` or the `where` methods, then you will end up with a cartesian product.

```
result = db.session.execute(db.select(Call, Contact))
for res in result:
    print(res)
# prints
# (<Call 1: 1112223333 None>, <Contact 1: Denis>)
# (<Call 1: 1112223333 None>, <Contact 2: Bob>)
# (<Call 2: 5552223333 1>, <Contact 1: Denis>)
# (<Call 2: 5552223333 1>, <Contact 2: Bob>)
# (<Call 3: 5552223333 1>, <Contact 1: Denis>)
# (<Call 3: 5552223333 1>, <Contact 2: Bob>)
```

All the following 3 ways give the same results.

```
result = db.session.execute(db.select(Call, Contact)
                            .where(Call.contact_id == Contact.contact_id))
result = db.session.execute(db.select(Call, Contact).join(Contact))
result = db.session.execute(db.select(Call, Contact).join(Contact,
                                                          Call.contact_id ==
        Contact.contact_id))
for res in result:
    print(res)
# prints
# (<Call 2: 5552223333 1>, <Contact 1: Denis>)
# (<Call 3: 5552223333 1>, <Contact 1: Denis>)
```

The first way will produce the following SQL query. It’s really a cartesian product followed by a filter on a condition, which is equivalent to a join.

```
SELECT *
FROM call,
     contact
WHERE call.contact_id == contact.contact_id
```

The other 2 ways will produce the same SQL query, but with the last one, you have the flexibility to specify the join condition you want.

```
SELECT *
FROM call
         INNER JOIN contact ON call.contact_id == contact.contact_id
```