Databases

CITS3403 and CITS5505 - Agile Web Development

Unit Coordinator: Matthew Daggitt

Semester 1, 2024

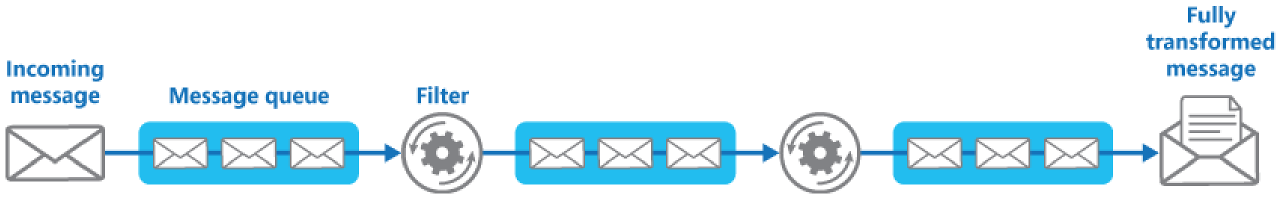
Architecture

Architectural patterns

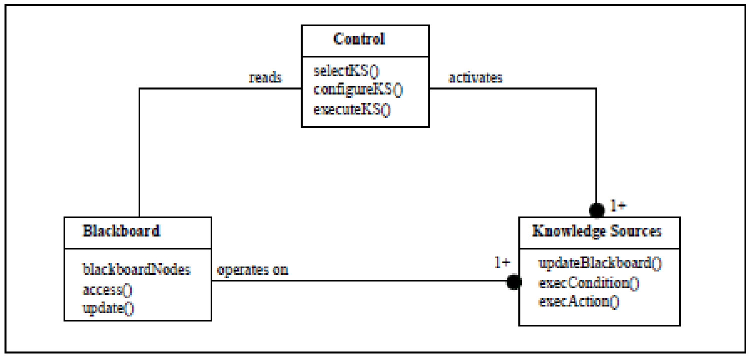
- Design patterns describe re-useable design concepts, particularly in software. They describe how concepts are organized.

- We've already seen one of the most famous architectural design patterns:
  - client-server architecture

- Other architectural design patterns include:
  - pipe and filter

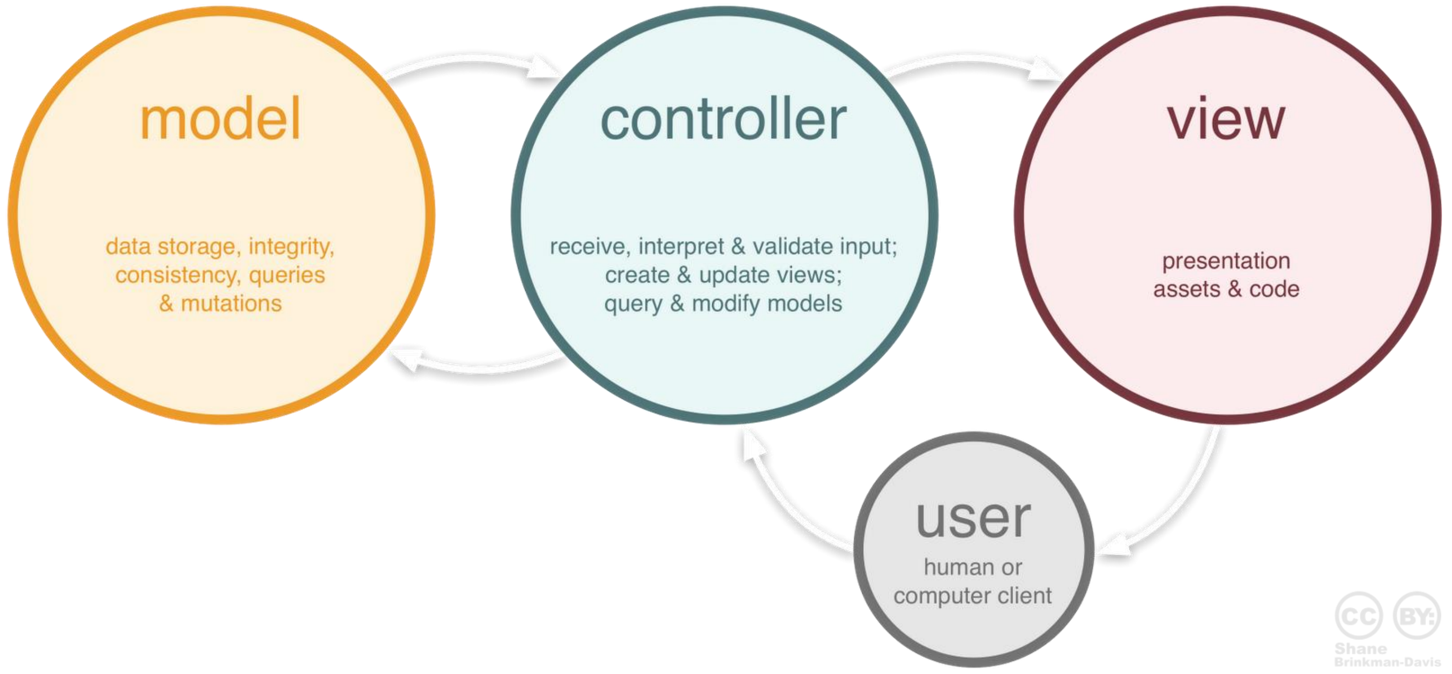


  - blackboard architectures.



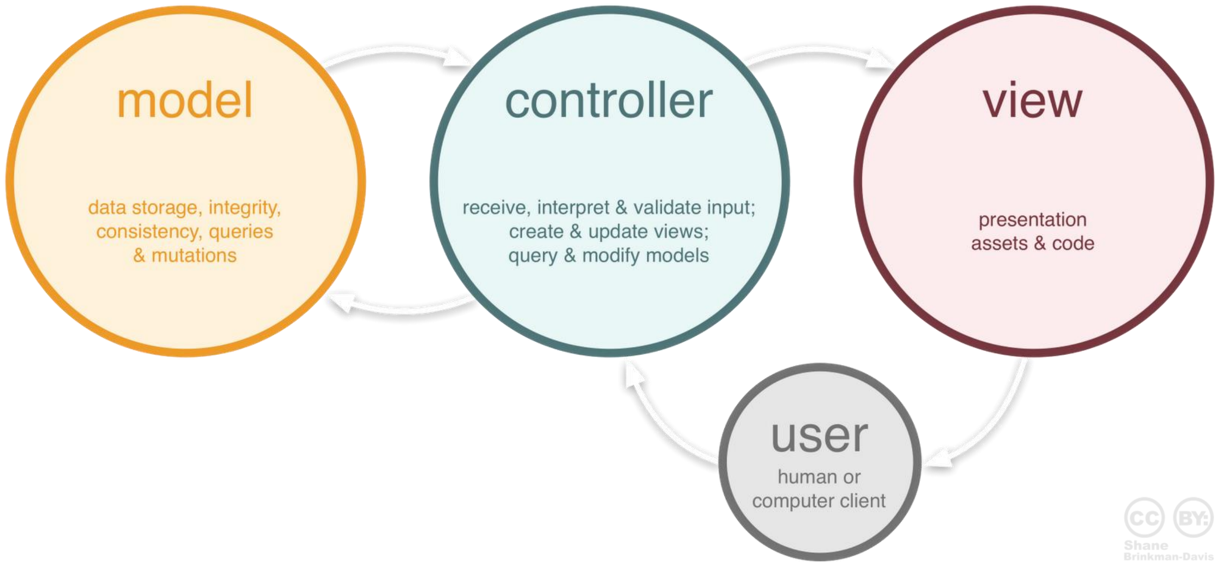
Model-view-controller pattern

- The model-view-controller (MVC) pattern is one of the most popular architectures for server-side web applications.


model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

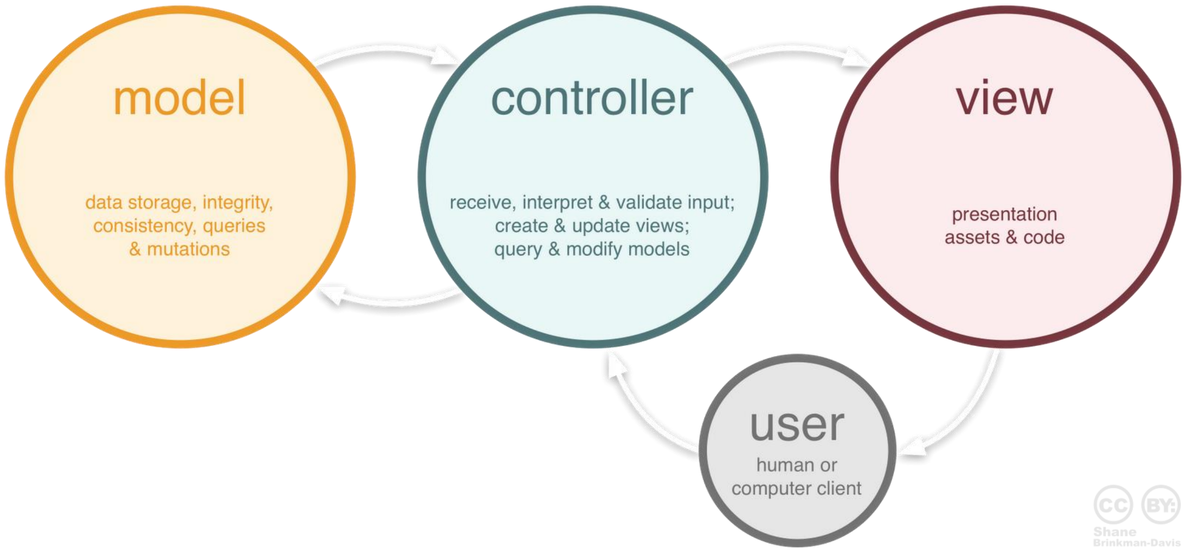
Advantages of MVC

- The division of the web-application in this manner has several advantages:
  - You can alter the view without altering the model.
  - You can swap the database out without altering the view.
  - Specialist developers can work on each bit.
  - Can support multiple views (e.g. web page + mobile app).
  - Easier to test parts in isolation.


model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

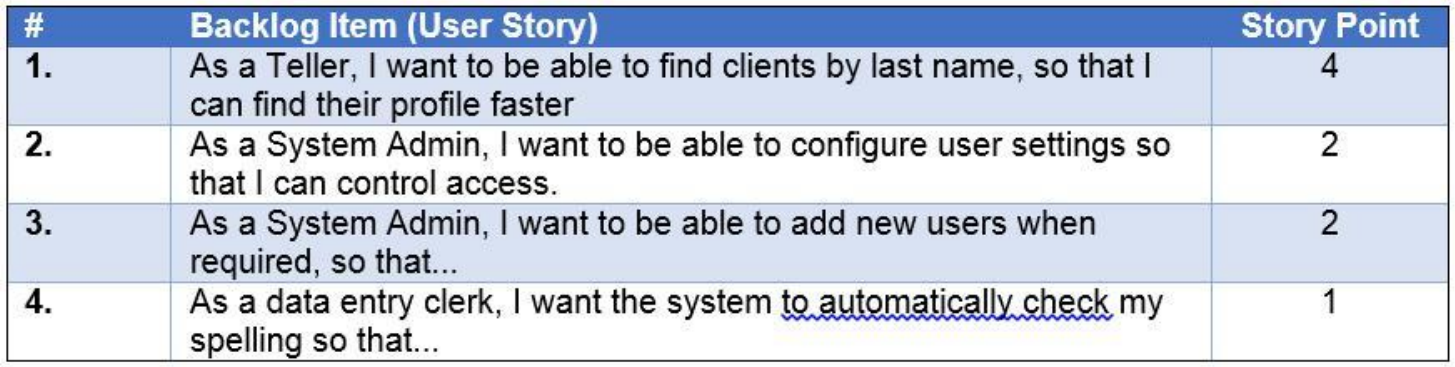
MVC in Agile Web Dev

- The model refers to the Python objects in the application backed by the database (SQLite).
- The view are the HTML pages created from the Jinja templates.
- The controller is the code in `routes.py` that i) links the model to the database, ii) prepares the view based on the model, and iii) the updates and saves the models back to the database.


model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

Designing an MVC architecture

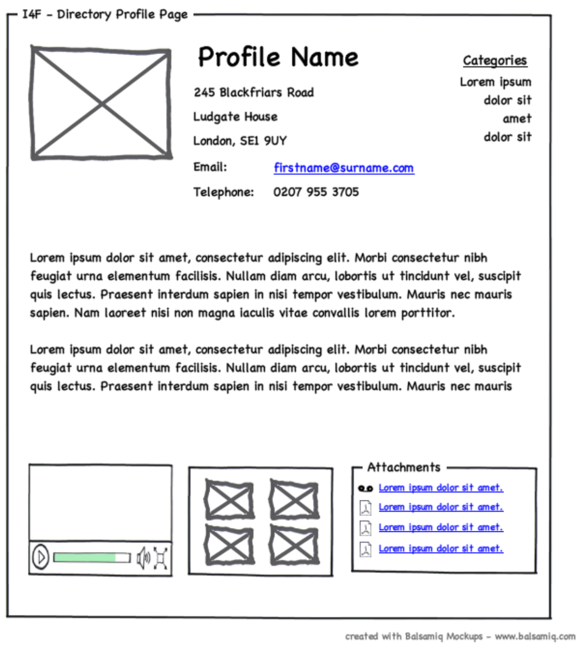
- To design an MVC solution architecture, you need to identify what models, views and controllers you require.
- Recall user stories are simple representations of software requirements.
- In every user story, we can identify *nouns* which could be models, *verbs* which could be routes, and associate a view for the specified user.
- We can then mockup wireframe sketches of view and mock HTTP requests and responses.



#	Backlog Item (User Story)	Story Point
1.	As a Teller, I want to be able to find clients by last name, so that I can find their profile faster	4
2.	As a System Admin, I want to be able to configure user settings so that I can control access.	2
3.	As a System Admin, I want to be able to add new users when required, so that...	2
4.	As a data entry clerk, I want the system to automatically check my spelling so that...	1

Mock-ups of views

- Wireframe drawing show the basic layout and functionality of a user interface.
- There are various tools for building these, or you can draw them by hand.
- A series of wire frame mocks can show the sequence of interfaces used in an application.



Profile Name

245 Blackfriars Road
Ludgate House
London, SE1 9UY

Email: firstname@surname.com
Telephone: 0207 955 3705

Categories
Lorem ipsum
dolor sit
amet
dolor sit

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi consectetur nibh feugiat urna elementum facilisis. Nullam diam arcu, lobortis ut tincidunt vel, suscipit quis lectus. Praesent interdum sapien in nisi tempor vestibulum. Mauris nec mauris sapien. Nam laoreet nisi non magna iaculis vitae convallis lorem porttitor.

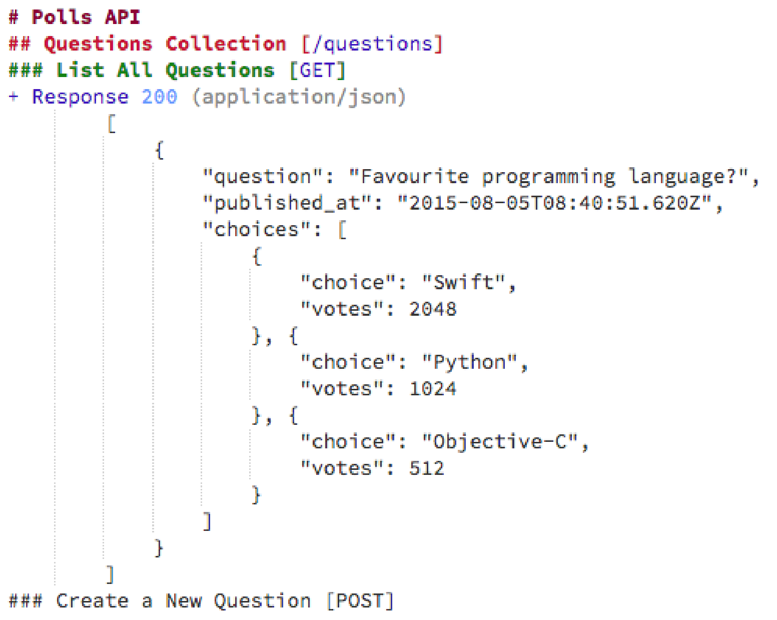
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi consectetur nibh feugiat urna elementum facilisis. Nullam diam arcu, lobortis ut tincidunt vel, suscipit quis lectus. Praesent interdum sapien in nisi tempor vestibulum. Mauris nec mauris

Attachments
- Lorem ipsum dolor sit amet.
- Lorem ipsum dolor sit amet.
- Lorem ipsum dolor sit amet.
- Lorem ipsum dolor sit amet.

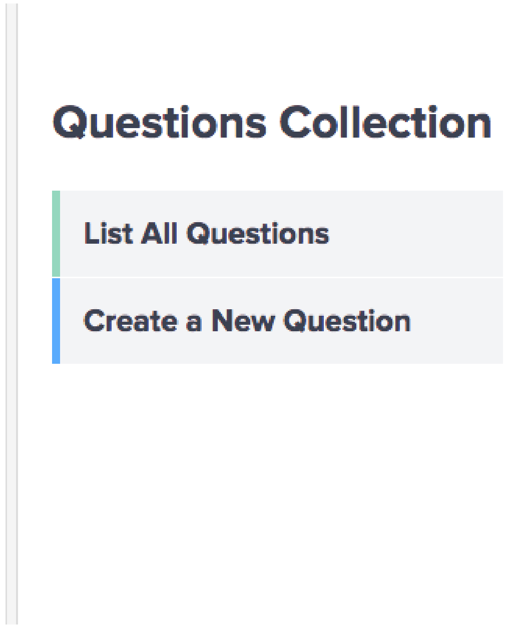
created with Balsamiq Mockups – www.balsamiq.com

Mock-ups of APIs

- You can also mock the typical HTTP requests and responses your app will serve.
- These can be hard coded using tools like Apiary and Mocky (more on this later)



```
# Polls API
## Questions Collection [/questions]
### List All Questions [GET]
+ Response 200 (application/json)
        [
            {
                "question": "Favourite programming language?",
                "published_at": "2015-08-05T08:40:51.620Z",
                "choices": [
                    {
                        "choice": "Swift",
                        "votes": 2048
                    }, {
                        "choice": "Python",
                        "votes": 1024
                    }, {
                        "choice": "Objective-C",
                        "votes": 512
                    }
                ]
            }
        ]
### Create a New Question [POST]
```



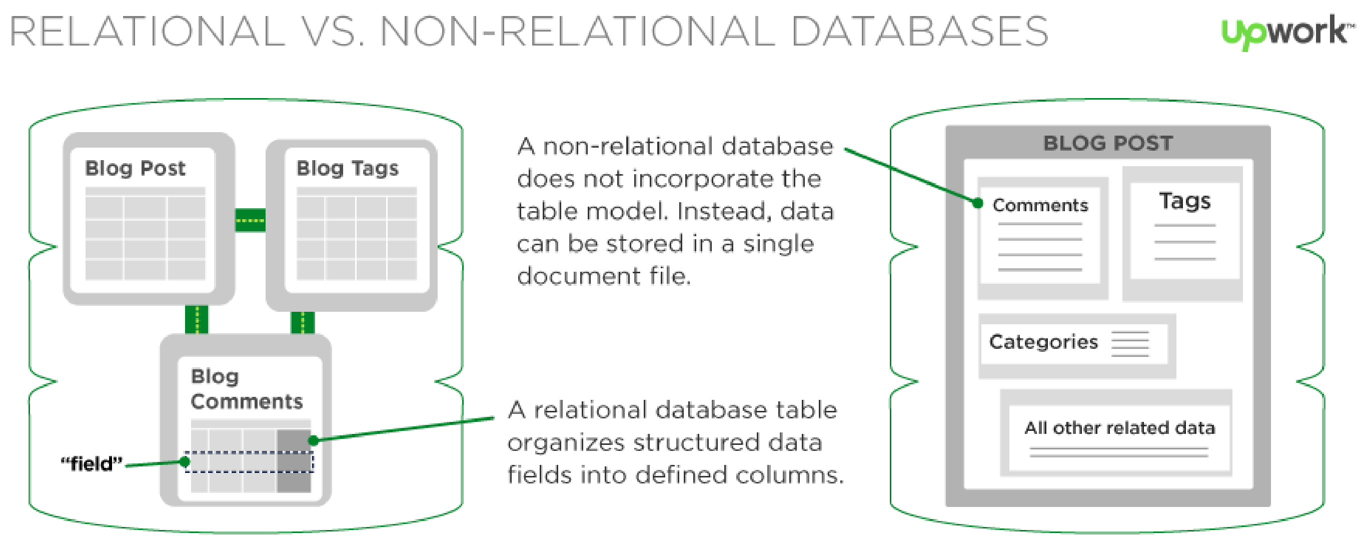
Implementing models

- A model is an object that is paired with an entity in a database.
- There is an Object Relational Mapping (ORM) linking the data in the database to the models in the application.
- The models are only built as needed and update the database as required. Most frameworks include ORM support.
- To build the models, we first need to set up the database.

Databases refresher

Types of databases

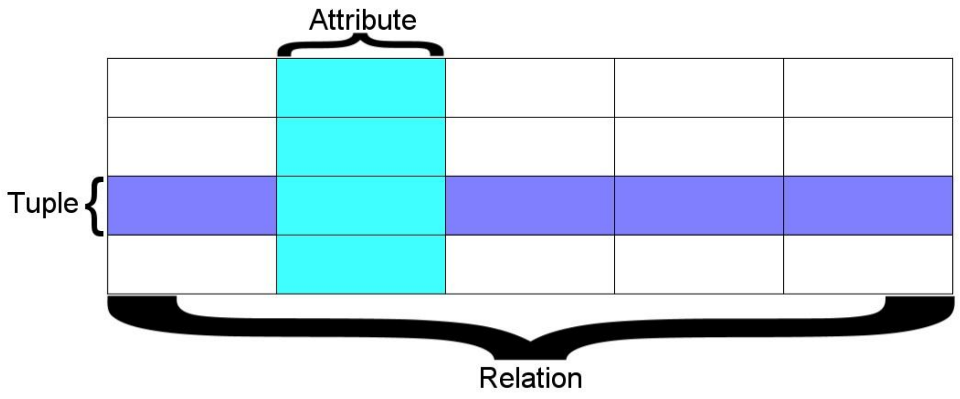
- There are relational databases, document databases, graph databases, and others

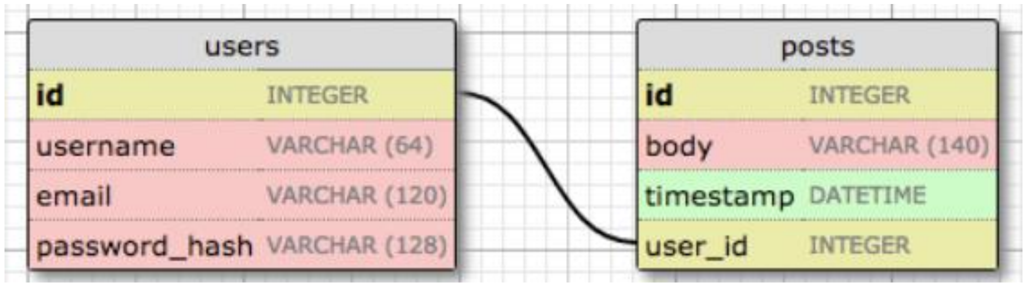

RELATIONAL VS. NON-RELATIONAL DATABASES
upwork
Blog Post
Blog Tags
Blog Comments
"field"
A non-relational database does not incorporate the table model. Instead, data can be stored in a single document file.
A relational database table organizes structured data fields into defined columns.
BLOG POST
Comments
Tags
Categories
All other related data

- Flask is very flexible and can support a variety of databases.

Relational databases

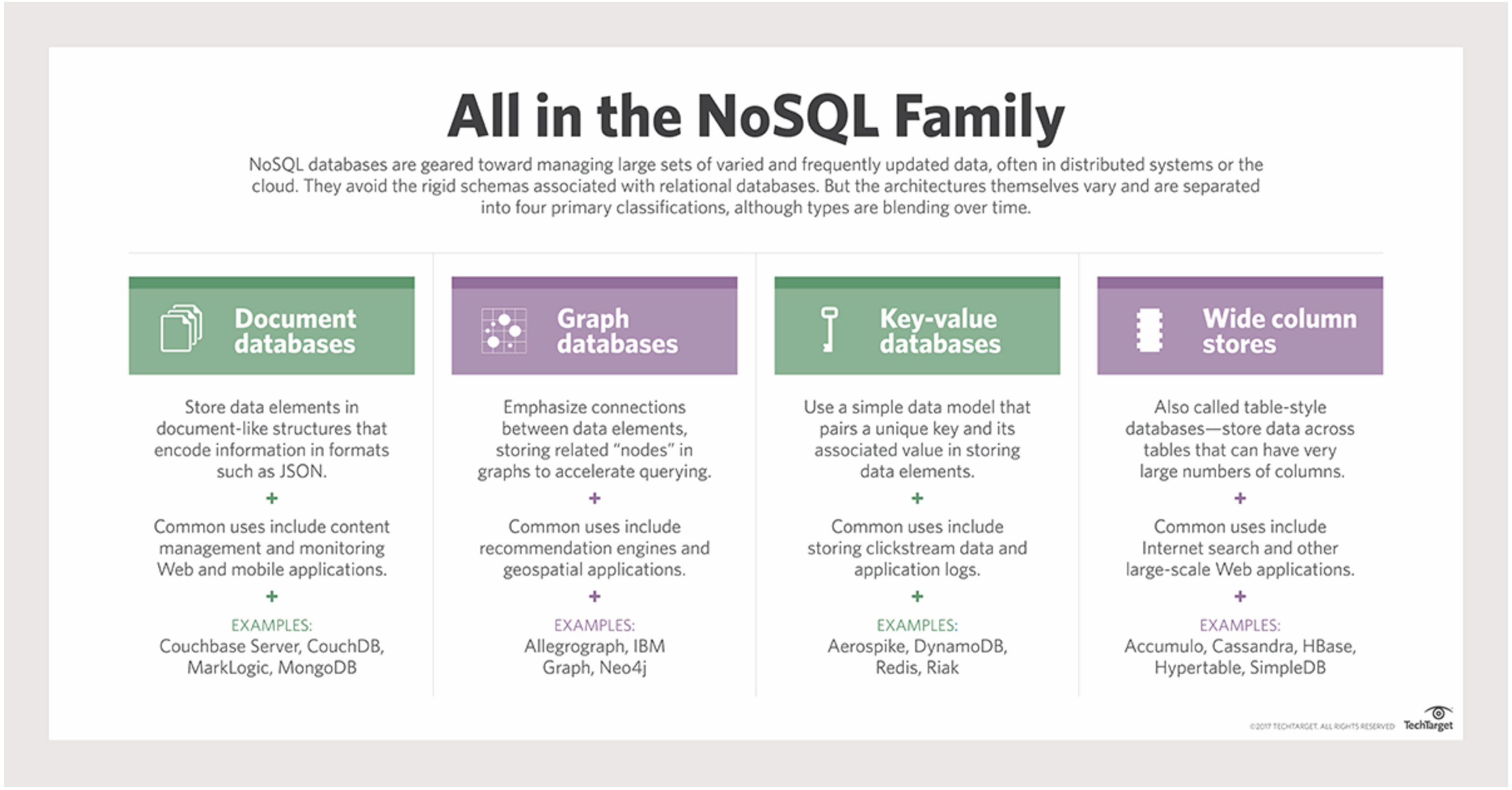
- Relational databases store data as a set of relations, where each relation is represented as a table.
- Each row of the table is an entity, and each column is an attribute of that entity.
- Every relation has an attribute that is unique for every entity in that relation, called the primary key.
- Some relations attributes that are primary keys in other relations. These are called foreign keys.





NoSQL

- NoSQL standards for "not only SQL" and describes non-relational databases.



All in the NoSQL Family

NoSQL databases are geared toward managing large sets of varied and frequently updated data, often in distributed systems or the cloud. They avoid the rigid schemas associated with relational databases. But the architectures themselves vary and are separated into four primary classifications, although types are blending over time.

Document databases

Store data elements in document-like structures that encode information in formats such as JSON.

+

Common uses include content management and monitoring Web and mobile applications.

+

EXAMPLES:
Couchbase Server, CouchDB, MarkLogic, MongoDB

Graph databases

Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.

+

Common uses include recommendation engines and geospatial applications.

+

EXAMPLES:
Allegrograph, IBM Graph, Neo4j

Key-value databases

Use a simple data model that pairs a unique key and its associated value in storing data elements.

+

Common uses include storing clickstream data and application logs.

+

EXAMPLES:
Aerospike, DynamoDB, Redis, Riak

Wide column stores

Also called table-style databases—store data across tables that can have very large numbers of columns.

+

Common uses include Internet search and other large-scale Web applications.

+

EXAMPLES:
Accumulo, Cassandra, HBase, Hypertable, SimpleDB

©2017 TECHTARGET. ALL RIGHTS RESERVED TechTarget

- These can be very useful in some applications, but RDMS are still be far the most popular and general approach.

Database Management System - SQLite

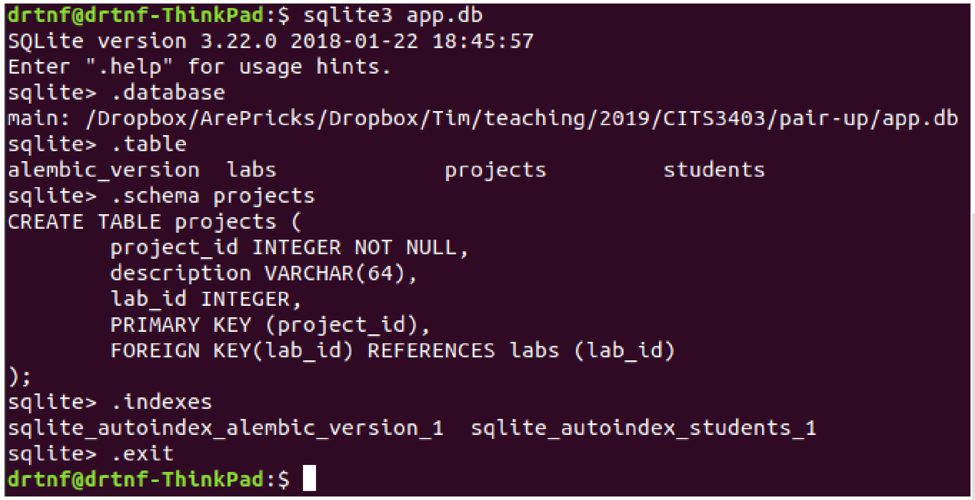
- A Database Management System (DBMS) is an application that controls access to a database.

- We will use the relational database SQLite as our DBMS because:
  - Simple to setup.
  - Small package size.
  - Stores the database as a single file.

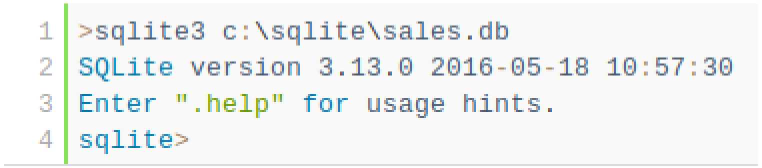
- Disadvantages compared to MySQL:
  - Doesn't scale well
  - Doesn't have inbuilt authentication methods.

Using SQLite

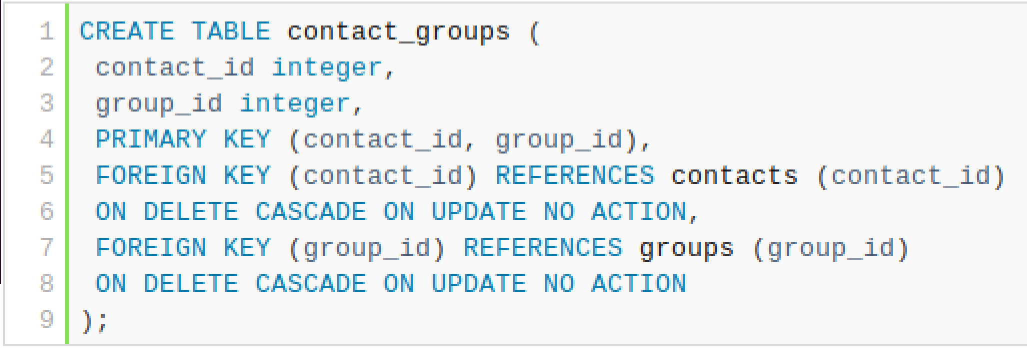
- A database is created, and then we set up schemas for the tables
- The schema of the database is the set of tables (relations) that are defined, the types of the attributes, and the constraints on the attributes. This is the *meta-data* of the database and is not expected to change in the normal usage of the application.
- SQLite commands start with a '.' and can display the metadata (.help to see all commands)



```
drtnf@drtnf-ThinkPad:$ sqlite3 app.db
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
sqlite> .database
main: /Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/app.db
sqlite> .table
alembic_version  labs             projects         students
sqlite> .schema projects
CREATE TABLE projects (
        project_id INTEGER NOT NULL,
        description VARCHAR(64),
        lab_id INTEGER,
        PRIMARY KEY (project_id),
        FOREIGN KEY(lab_id) REFERENCES labs (lab_id)
);
sqlite> .indexes
sqlite_autoindex_alembic_version_1  sqlite_autoindex_students_1
sqlite> .exit
drtnf@drtnf-ThinkPad:$
```



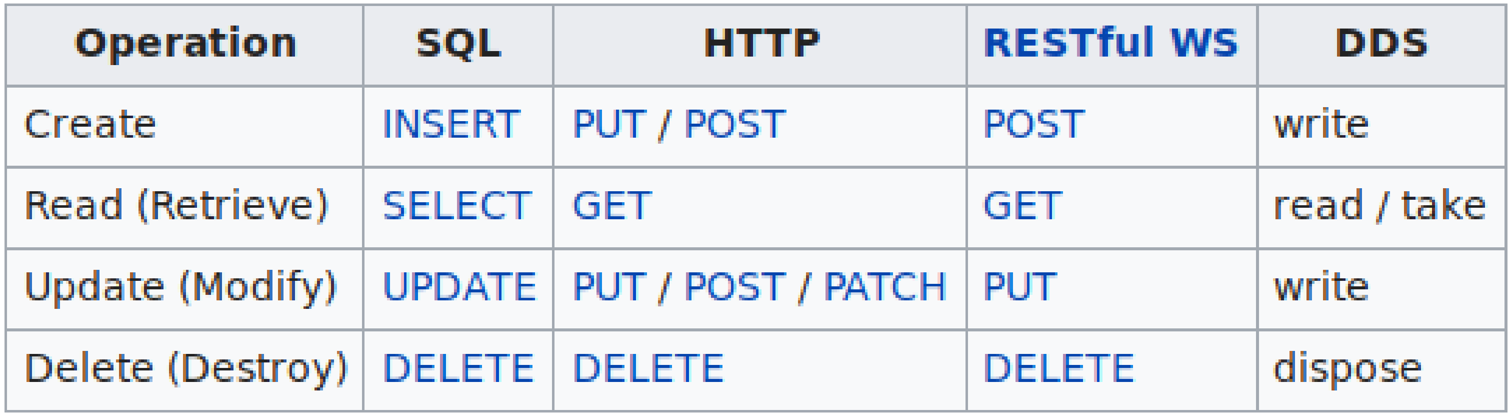
```
>sqlite3 c:\sqlite\sales.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite>
```



```
CREATE TABLE contact_groups (
 contact_id integer,
 group_id integer,
 PRIMARY KEY (contact_id, group_id),
 FOREIGN KEY (contact_id) REFERENCES contacts (contact_id)
 ON DELETE CASCADE ON UPDATE NO ACTION,
 FOREIGN KEY (group_id) REFERENCES groups (group_id)
 ON DELETE CASCADE ON UPDATE NO ACTION
);
```

CRUD

- The basic operations of any persistent storage system are ***C**reate*, ***R**ead*, ***U**pdate* and ***D**elete* (**CRUD**).



Operation	SQL	HTTP	RESTful WS	DDS
Create	INSERT	PUT / POST	POST	write
Read (Retrieve)	SELECT	GET	GET	read / take
Update (Modify)	UPDATE	PUT / POST / PATCH	PUT	write
Delete (Destroy)	DELETE	DELETE	DELETE	dispose

Relational query language

The sequential query language (SQL) provides the syntax for performing these CRUD operations:

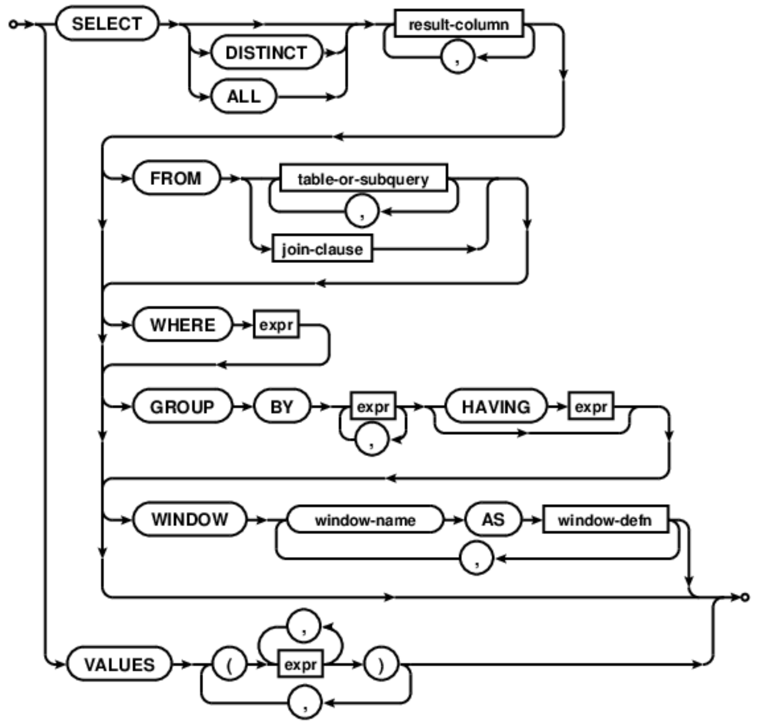
- Create is done using an *insert* statement
- Read is done using the *select* statement
- Update is done using an *update* statement
- Delete is done using a *delete* statement.

```sql
INSERT INTO table1 (
 column1,
 column2 ,..)
VALUES
 (
 value1,
 value2 ,...);
```

```sql
UPDATE table
SET column_1 = new_value_1,
    column_2 = new_value_2
WHERE
    search_condition
ORDER column_or_expression
LIMIT row_count OFFSET offset;
```

```sql
SELECT DISTINCT column_list
FROM table_list
  JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;
```

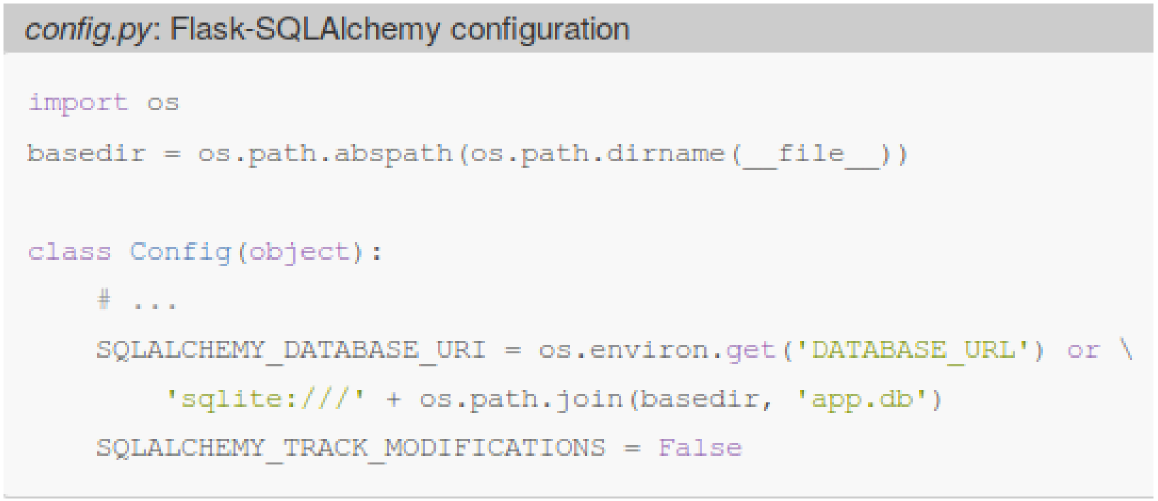
```sql
DELETE
FROM
 table
WHERE
 search_condition;
```



Databases in Flask

Linking to a database with SQL-Alchemy

- Now we have an SQLite database set up, we would like to link it into our application.
- We will use SQL-Alchemy for ORM.
- We need to install the packages `flask-sqlalchemy` and `flask-migrate`
- We will keep the database in a file called `app.db`, in the root of our app, and include this in a new `config.py`

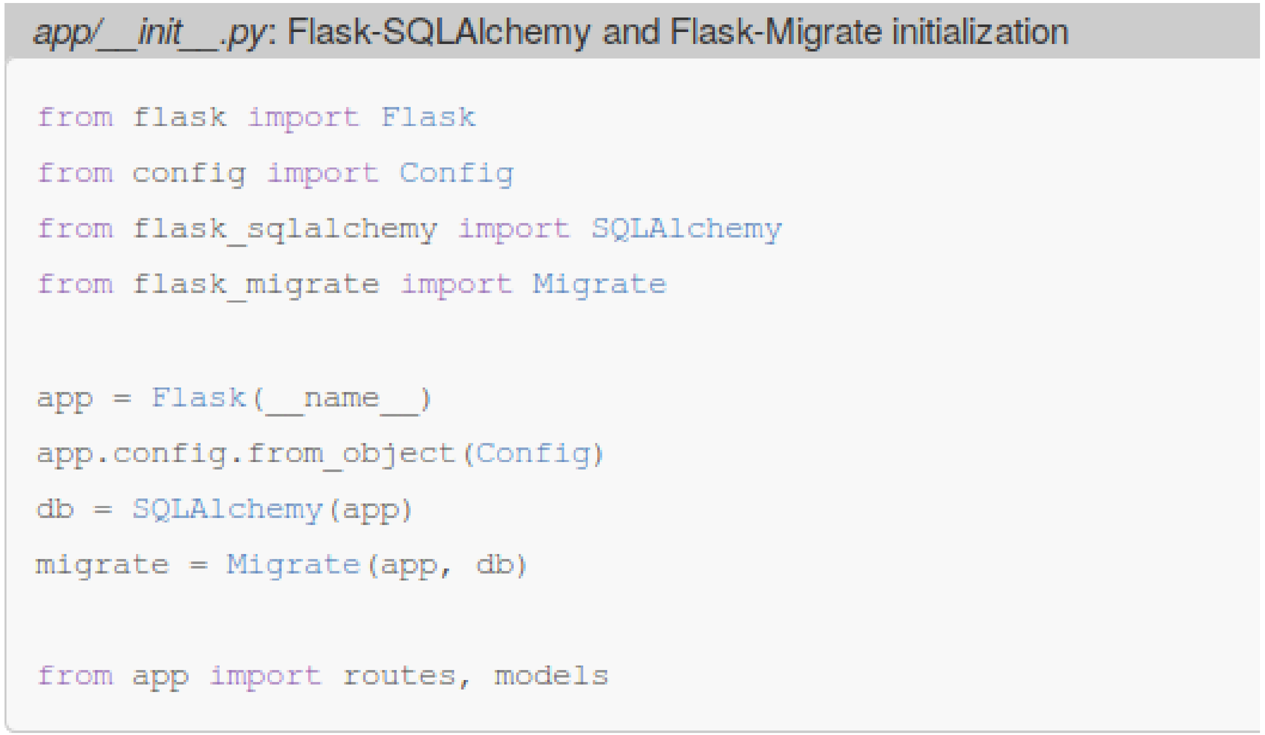


config.py: Flask-SQLAlchemy configuration

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Linking to a database with SQL-Alchemy

- Next, we update __init__.py to create an SQLAlchemy object called `db`, create a `migrate` object, and import a module called `models` (which we will write)

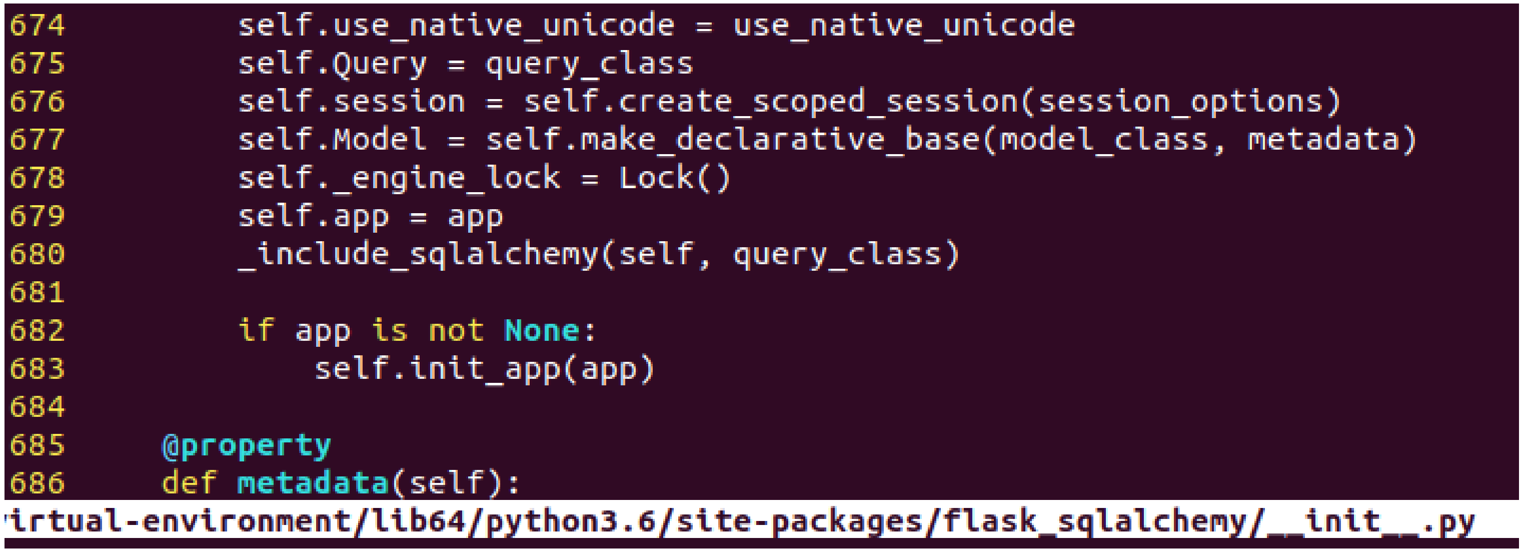


app/__init__.py: Flask-SQLAlchemy and Flask-Migrate initialization

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

Creating models with SQL-Alchemy

- To see what these modules are doing, you can find the source code in the virtual environment directory.



```
        self.use_native_unicode = use_native_unicode
        self.Query = query_class
        self.session = self.create_scoped_session(session_options)
        self.Model = self.make_declarative_base(model_class, metadata)
        self._engine_lock = Lock()
        self.app = app
        _include_sqlalchemy(self, query_class)

        if app is not None:
            self.init_app(app)

    @property
    def metadata(self):
irtual-environment/lib64/python3.6/site-packages/flask_sqlalchemy/__init__.py
```

Creating models with SQL-Alchemy

- To build a model we import `db`, the instance of the `SQLAlchemy` class, and our models are then all defined to be subclasses of `db.Model`.

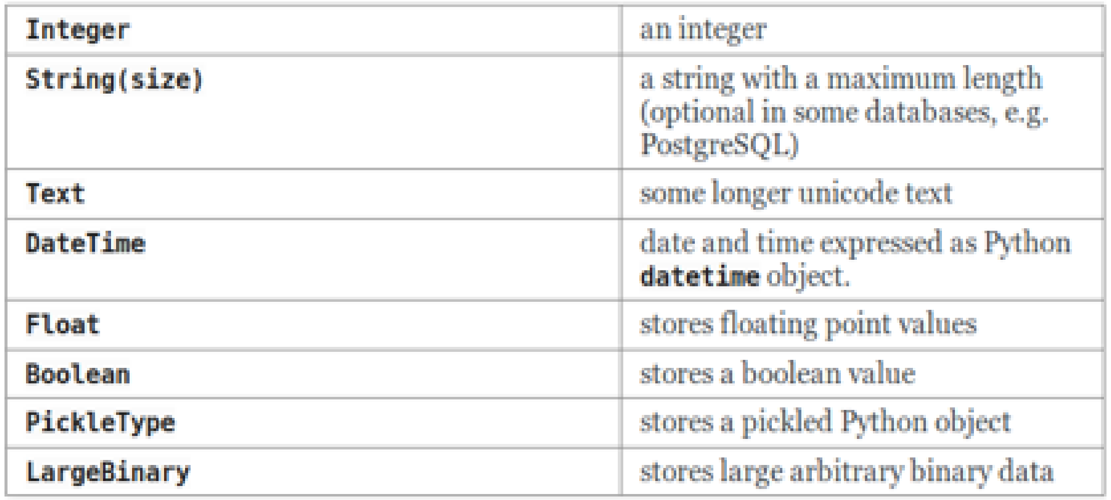
```python
class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'),
        nullable=False)
```

- Each model corresponds to a table in the database.

- Each column, defined using `db.Column`, corresponds to a column in the table.

Types of columns

- The first argument defines the type of the column and there are many types available:



`Integer`	an integer
`String(size)`	a string with a maximum length (optional in some databases, e.g. PostgreSQL)
`Text`	some longer unicode text
`DateTime`	date and time expressed as Python `datetime` object.
`Float`	stores floating point values
`Boolean`	stores a boolean value
`PickleType`	stores a pickled Python object
`LargeBinary`	stores large arbitrary binary data

- There are various types of optional column arguments including:
  - `nullable`
  - `primary key`
  - `foreign key`
  - `unique, index` etc.

Creating models with SQL-Alchemy

- `db.relationship` is a function that defines attributes based on a database relationship, i.e. links between rows in different tables.

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', backref='person', lazy=True)
```

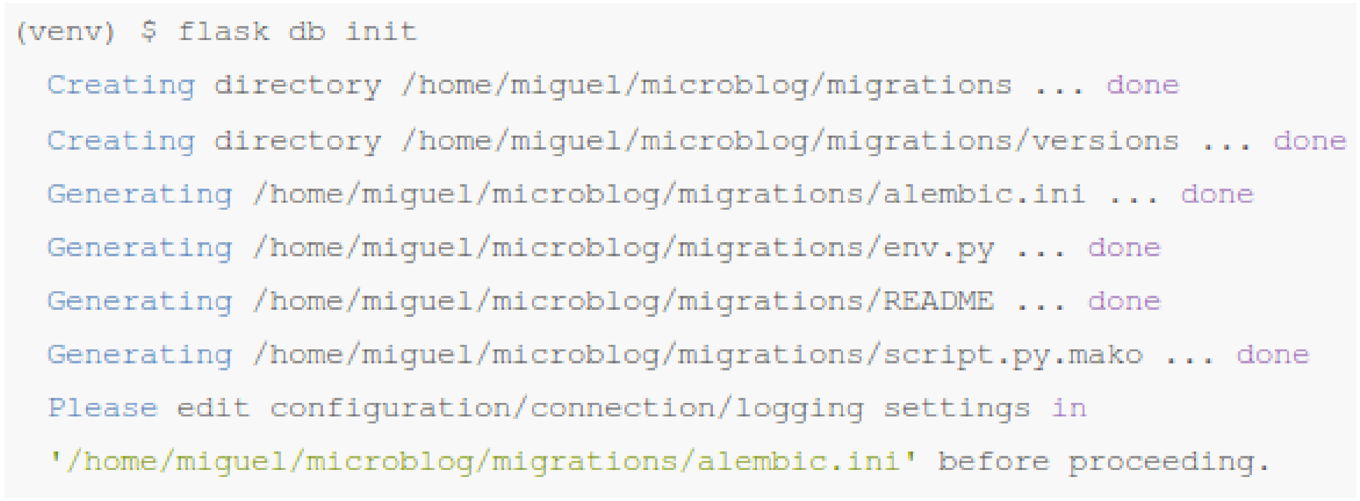
- The first argument is the name of the model that is being referenced.

Database initialisation

- Flask-SQLAlchemy can automatically extract the schema for the database tables from the models.

- The following command will initialise a database to synchronize with the models you have defined.

```
flask db init
```



```
(venv) $ flask db init
  Creating directory /home/miguel/microblog/migrations ... done
  Creating directory /home/miguel/microblog/migrations/versions ... done
  Generating /home/miguel/microblog/migrations/alembic.ini ... done
  Generating /home/miguel/microblog/migrations/env.py ... done
  Generating /home/miguel/microblog/migrations/README ... done
  Generating /home/miguel/microblog/migrations/script.py.mako ... done
  Please edit configuration/connection/logging settings in
  '/home/miguel/microblog/migrations/alembic.ini' before proceeding.
```

Adding entities

- We are now able to access the models from within the flask shell.
- The command `flask shell` will start the shell, and then we can import the models.
- We can create instances of the models and add them to the `db` object, using `db.session.add`

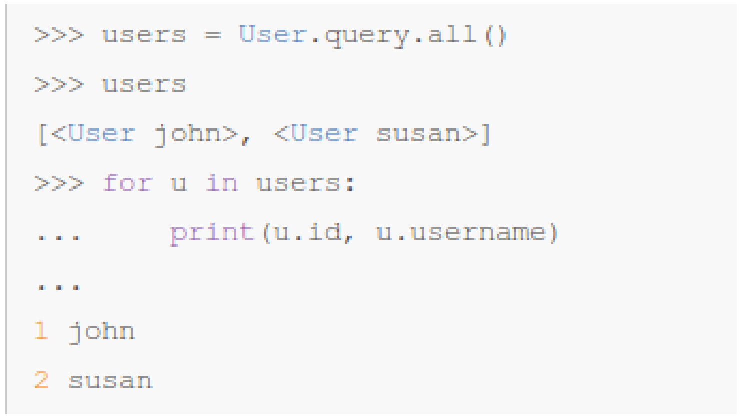
```
>>> u = User(username='susan', email='susan@example.com')
>>> db.session.add(u)
```

- Note, that just like `git add`, these changes are not yet in the database. We've merely notified the DBMS that these objects exist.
- The `db.session` object will synchronize all changes with the database when we call `db.session.commit`:

```
>>> db.session.commit()
```

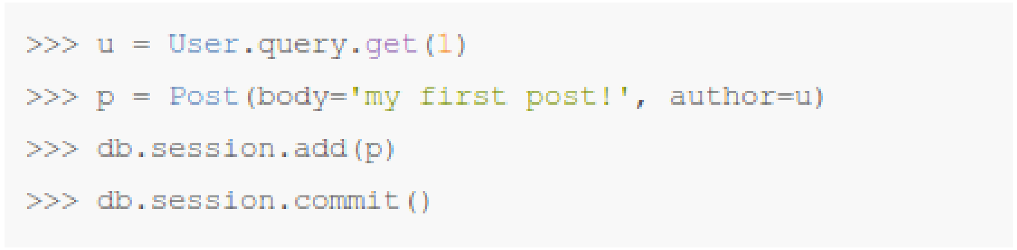
Basic queries

- We can extract entities from the database using a query which wraps an SQL select statement. For example:
  - `<model>.query.all()` or alternatively `session.query(<model>).all()` will return all entities of type model.



```
>>> users = User.query.all()
>>> users
[<User john>, <User susan>]
>>> for u in users:
...     print(u.id, u.username)
...
1 john
2 susan
```

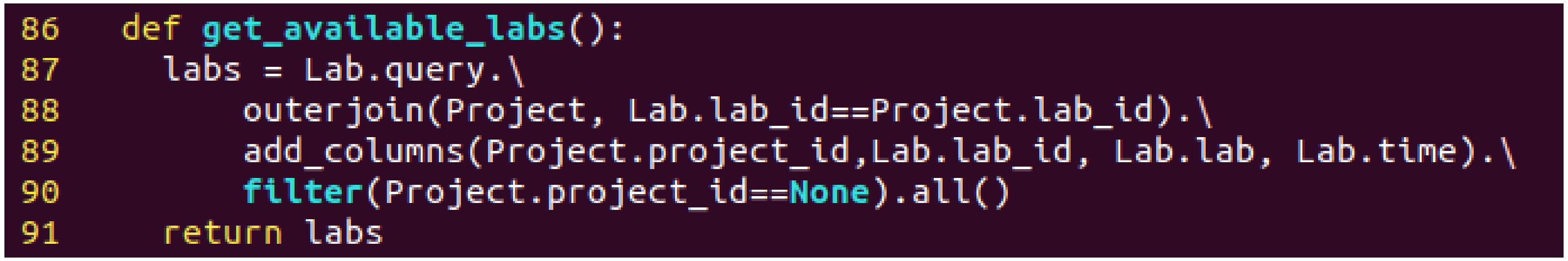
  - `<model>.query.get(i)` will get the ith instance of the model.



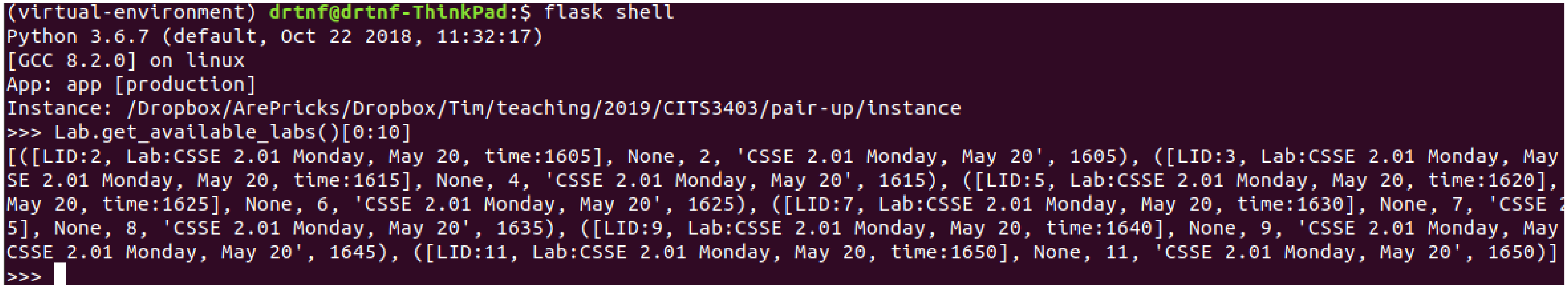
```
>>> u = User.query.get(1)
>>> p = Post(body='my first post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

More complicated queries

- We can also perform the following operations using the query syntax:
  - inner joins - `query.join()`
  - left-outer-joins - `query.outerjoin()`
  - filter - `filter_by()`
  - sort - `order_by()`



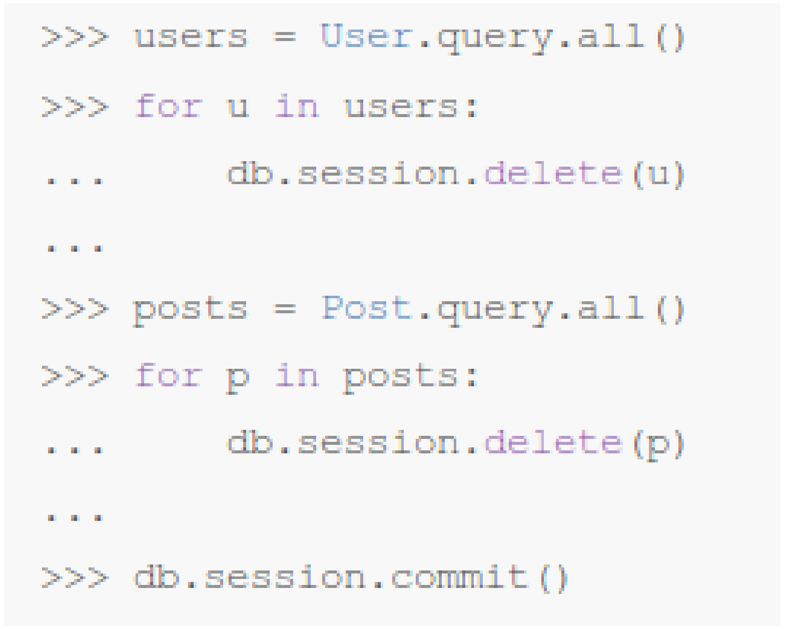
```python
def get_available_labs():
  labs = Lab.query.\
      outerjoin(Project, Lab.lab_id==Project.lab_id).\
      add_columns(Project.project_id,Lab.lab_id, Lab.lab, Lab.time).\
      filter(Project.project_id==None).all()
  return labs
```



```
(virtual-environment) drtnf@drtnf-ThinkPad:$ flask shell
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
App: app [production]
Instance: /Dropbox/ArePricks/Dropbox/Tim/teaching/2019/CITS3403/pair-up/instance
>>> Lab.get_available_labs()[0:10]
[([LID:2, Lab:CSSE 2.01 Monday, May 20, time:1605], None, 2, 'CSSE 2.01 Monday, May 20', 1605), ([LID:3, Lab:CSSE 2.01 Monday, May
SE 2.01 Monday, May 20, time:1615], None, 4, 'CSSE 2.01 Monday, May 20', 1615), ([LID:5, Lab:CSSE 2.01 Monday, May 20, time:1620],
May 20, time:1625], None, 6, 'CSSE 2.01 Monday, May 20', 1625), ([LID:7, Lab:CSSE 2.01 Monday, May 20, time:1630], None, 7, 'CSSE
5], None, 8, 'CSSE 2.01 Monday, May 20', 1635), ([LID:9, Lab:CSSE 2.01 Monday, May 20, time:1640], None, 9, 'CSSE 2.01 Monday, May
CSSE 2.01 Monday, May 20', 1645), ([LID:11, Lab:CSSE 2.01 Monday, May 20, time:1650], None, 11, 'CSSE 2.01 Monday, May 20', 1650)]
>>>
```

Deleting entities

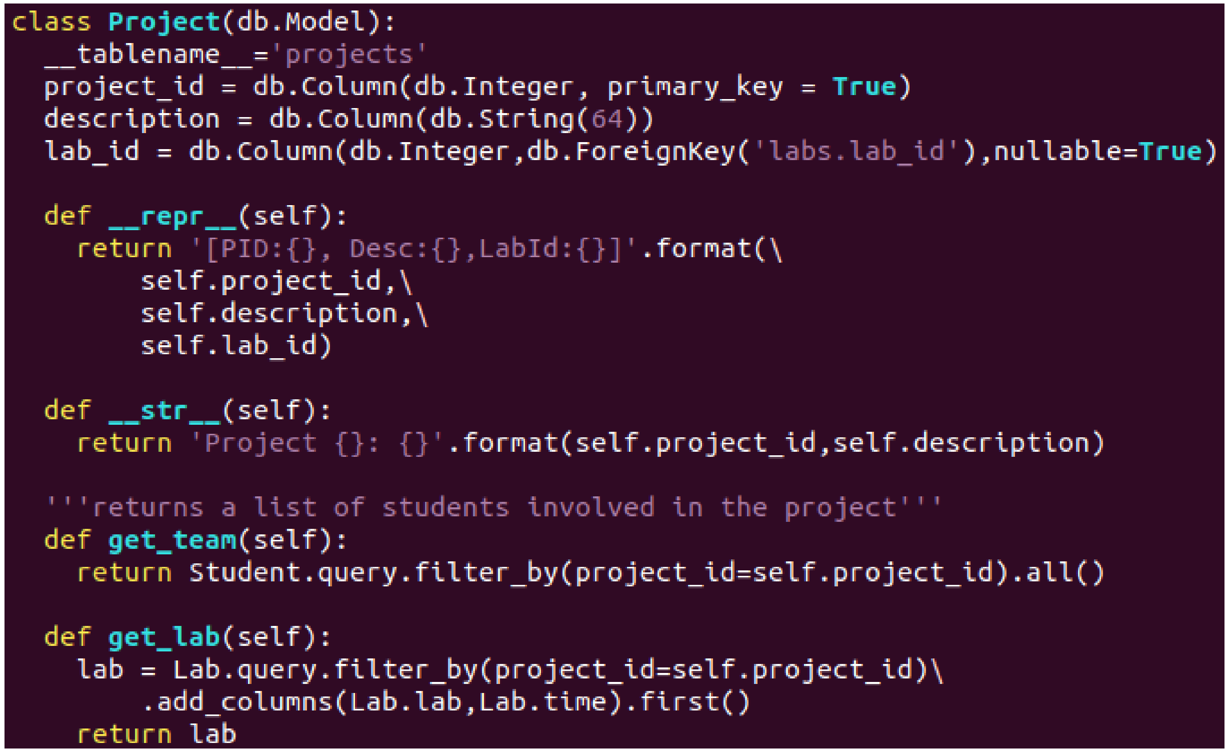
- We can delete instances of the models using `db.session.delete`



```
>>> users = User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
```

Adding helper methods in the models

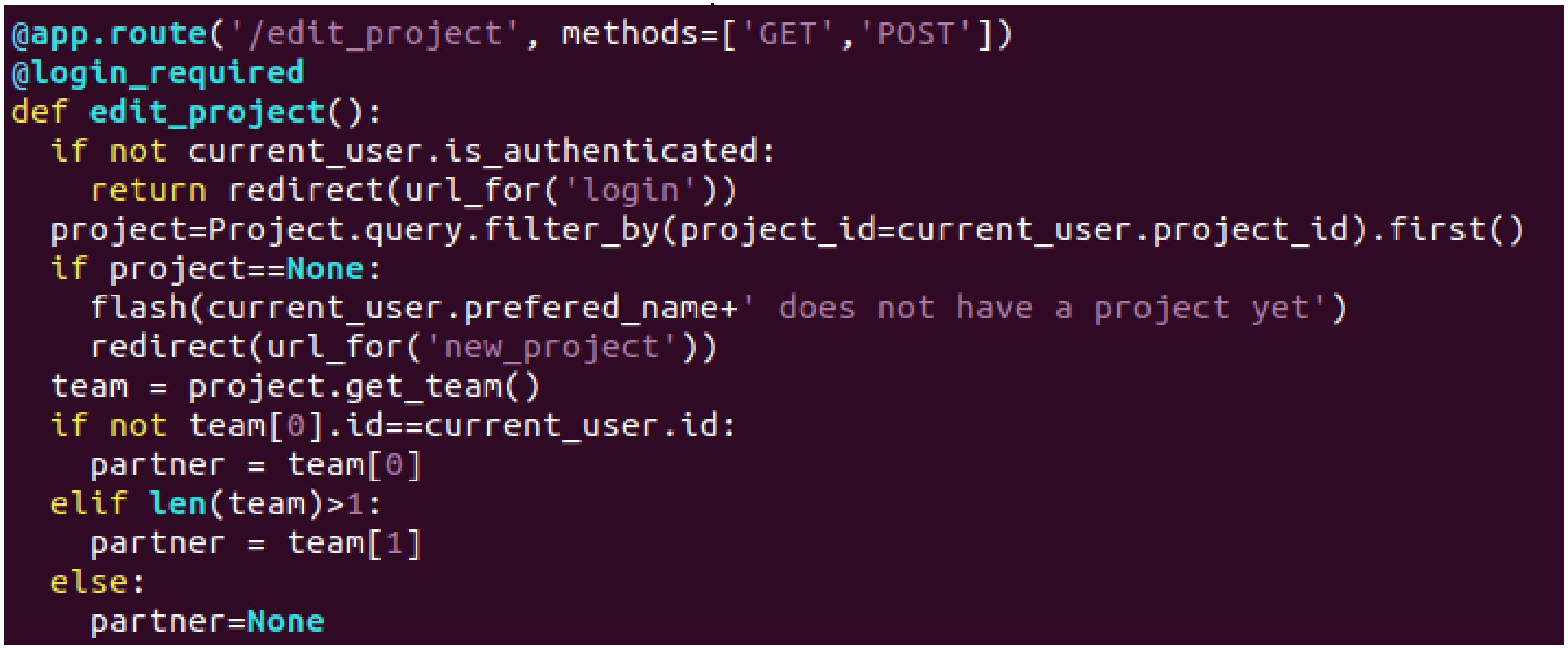
- The model classes are Python classes just like any other.
- Therefore, it is often useful to define helper methods on the classes to improve encapsulation:



```
class Project(db.Model):
  __tablename__='projects'
  project_id = db.Column(db.Integer, primary_key = True)
  description = db.Column(db.String(64))
  lab_id = db.Column(db.Integer,db.ForeignKey('labs.lab_id'),nullable=True)

  def __repr__(self):
    return '[PID:{}, Desc:{},LabId:{}]'.format(\
        self.project_id,\
        self.description,\
        self.lab_id)

  def __str__(self):
    return 'Project {}: {}'.format(self.project_id,self.description)

  '''returns a list of students involved in the project'''
  def get_team(self):
    return Student.query.filter_by(project_id=self.project_id).all()

  def get_lab(self):
    lab = Lab.query.filter_by(project_id=self.project_id)\
        .add_columns(Lab.lab,Lab.time).first()
    return lab
```

Putting it altogether in routes.py

- We can now respond to requests for data, by building models from the database, and then populating views with the data.



```
@app.route('/edit_project', methods=['GET','POST'])
@login_required
def edit_project():
  if not current_user.is_authenticated:
    return redirect(url_for('login'))
  project=Project.query.filter_by(project_id=current_user.project_id).first()
  if project==None:
    flash(current_user.prefered_name+' does not have a project yet')
    redirect(url_for('new_project'))
  team = project.get_team()
  if not team[0].id==current_user.id:
    partner = team[0]
  elif len(team)>1:
    partner = team[1]
  else:
    partner=None
```

- As the code is getting complex, it is a good idea to have a `controllers.py` class and move the big functions there, rather than handling everything in routes.py

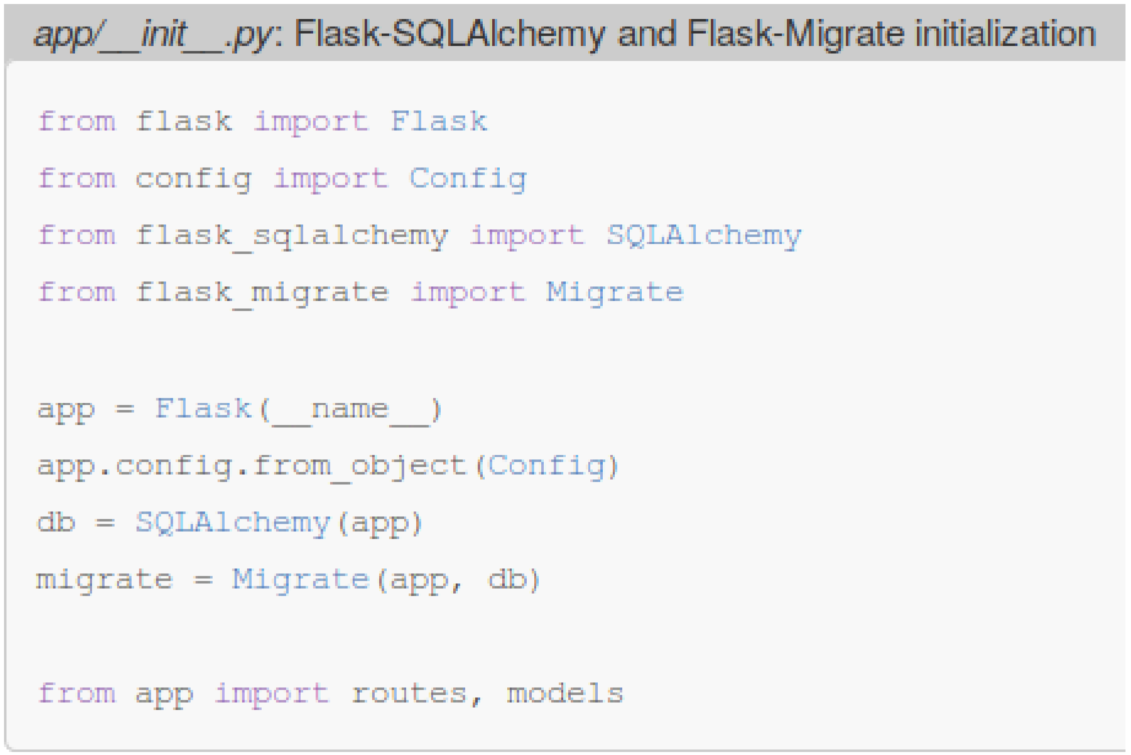
Database migration in Flask

Database migration

- As you develop your application, you will end up frequently altering your models.
- However, you also need to update the database in a way that doesn't break your existing data.
- The answer is a migration framework that acts like version control for your database, allowing you to move forward or backward in time between different versions at will.
- Each change to the database is accompanied with upgrade and downgrade scripts that can be used to move the database from and to the previous version of the models.

Database migration in Flask

- SQLAlchemy uses the Alembic migration framework, which is wrapped by the `flask-migrate` package we installed earlier.



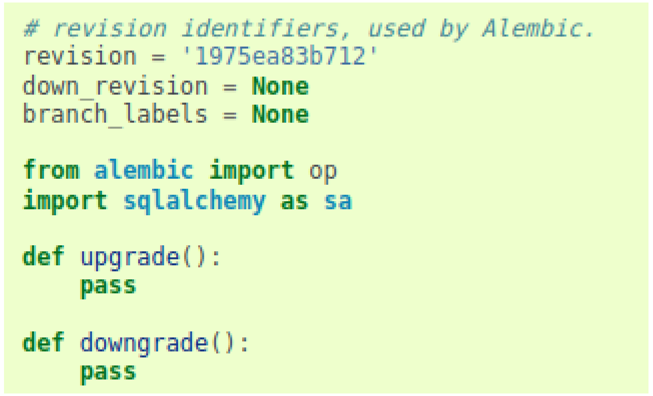
app/__init__.py: Flask-SQLAlchemy and Flask-Migrate initialization

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

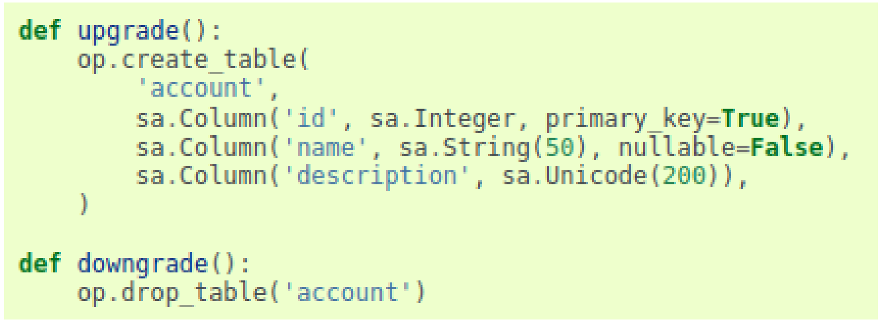
Creating a database migration manually

- A database migration can be created manually by using the `flask db revision` command.

- This creates a script where the `upgrade` and `downgrade` methods can be filled out.



```python
# revision identifiers, used by Alembic.
revision = '1975ea83b712'
down_revision = None
branch_labels = None

from alembic import op
import sqlalchemy as sa

def upgrade():
    pass

def downgrade():
    pass
```

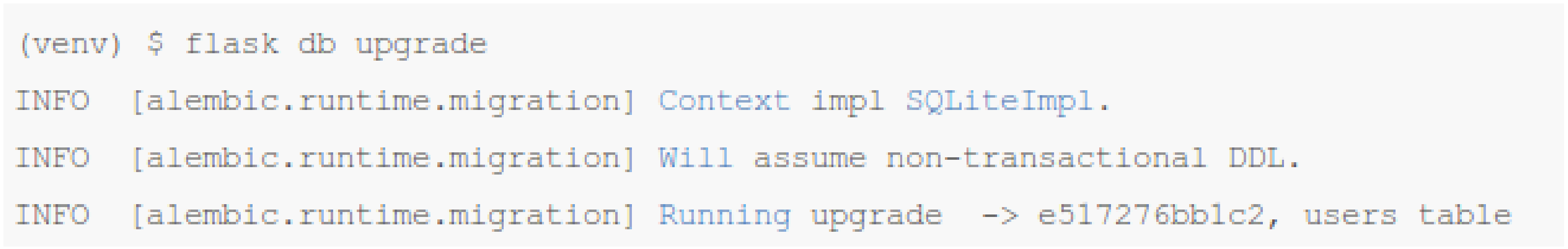


```python
def upgrade():
    op.create_table(
        'account',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('name', sa.String(50), nullable=False),
        sa.Column('description', sa.Unicode(200)),
    )

def downgrade():
    op.drop_table('account')
```

- A migration script can (sometimes!) be created automatically by using the `flask db migrate` command. This compares the updated `Model` classes with the current state of the database.

Database migration

- `flask db upgrade` applies that scripts to the database to bring it up to date.



```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> e517276bb1c2, users table
```

- `flask db downgrade` rolls the changes back.

- This allows us to keep the database schema and the models in sync.

- To get the highest marks in the group project you will need to perform at least one database migration!

Other

Storing images on webservers

- You *can* store images in your database directly using the `Blob` column type.
- This is not very performant as databases are not designed for storing this sort of data.
- In small-scale projects a better approach is to store the images in the filesystem of your server and then store a path to that file in the database.
- In large-scale projects, it's better to store the images off the server entirely on special purpose hardware, as server hardware is not optimized to do heavy IO operations.

Real-life webservers

- When your app reaches a certain size, your database must support many more things. This includes:
  - Adding indexes to allow more performant queries on frequently used data.
  - Optimising data layout depending on whether it is most often read from or written to.
  - Caching to avoid performing the same queries repeatedly.
  - Security to stop malicious users or applications accessing your database even if they compromise the server.
  - Backups to avoid losing your database if your server fails.
  - Auditing to keep track of when a database was altered and by whom.