Topic 10: Model-View-Controller

CITS3403 Agile Web Development

Semester 1, 2023

Architectural Patterns
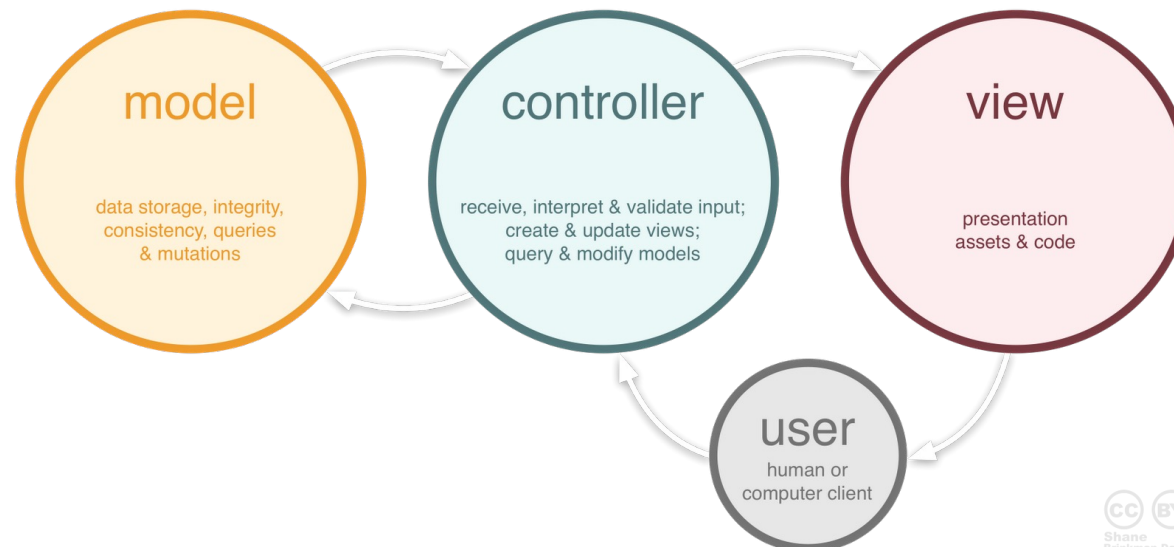
- Design patterns describe re-useable design concepts, particularly in software. They describe how objects are organized to call each other.

- Examples are client-server architecture, pipe and filter, and blackboard architectures.

- Some specific patterns that apply to web applications are *Model View Controller*, *Boundary Control Entity*, *3-Tier Architecture* and *Model View View-Model*.

Model View Controller

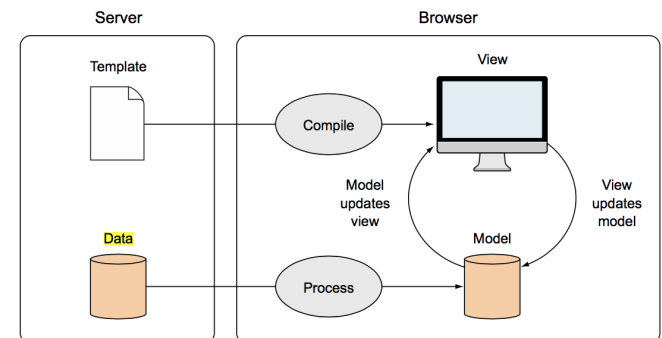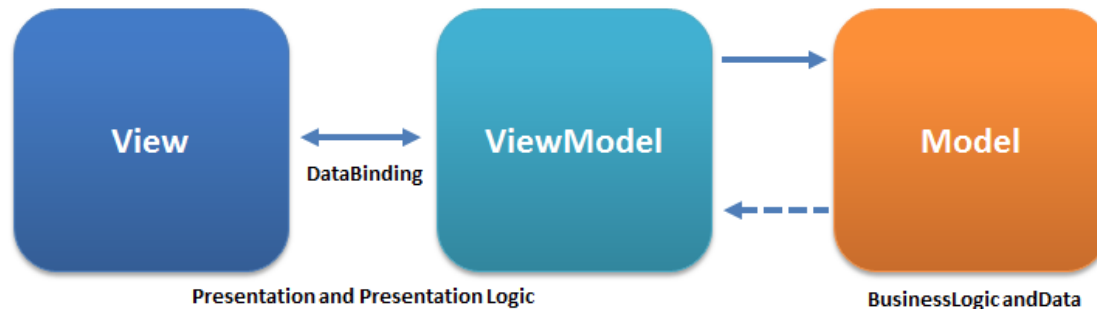- The model view controller patter is one of the most popular for server side web applications.
- The *model* refers to an object referencing an entity in a database.
- The *view* is how that object is presented to the user.
- The *controller* is a linking class that builds the model from the database, prepares the view based on the model, and the updates and saves the models back to the database.

Model View ViewModel

- Model View View-Model is a variation of model view controller that is tailor for client side applications and single page applications. Rather than having a controller compose the view a *binder* links the *view* to a *viewmodel*.

- The *view* presents thethe current state of the viewmodel

- The *viewmodel* exposes the data and available operations of the model, and updates the model as required.

- Two way data-binding links the *view* and *viewmodel* without need to link back to the server.

By Ugaya40 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19056842
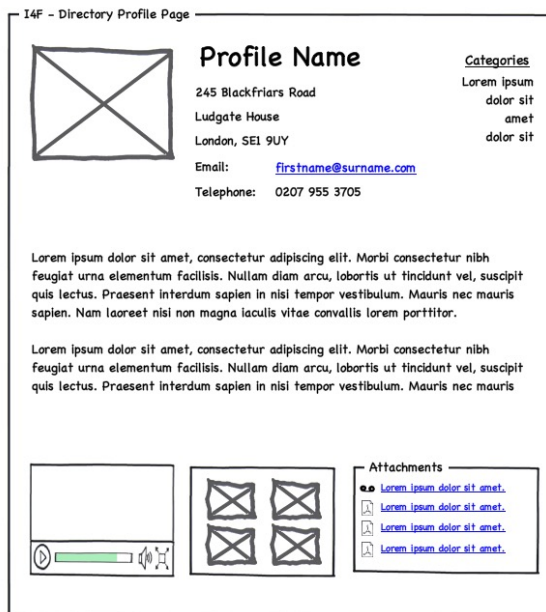
Designing an MVC structure

- We will focus on the MVC architecture as it is most suitable for web applications with server side rendering.
- 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 mock up 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 Websites

- 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.
- 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)

I4F – Directory Profile Page

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

```
# 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]
```
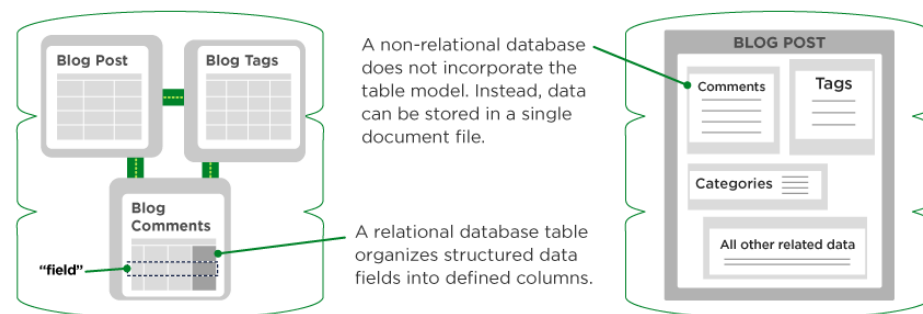
Questions Collection

List All Questions

Create a New Question

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.
- There are relational databases, document databases, graph databases,and others
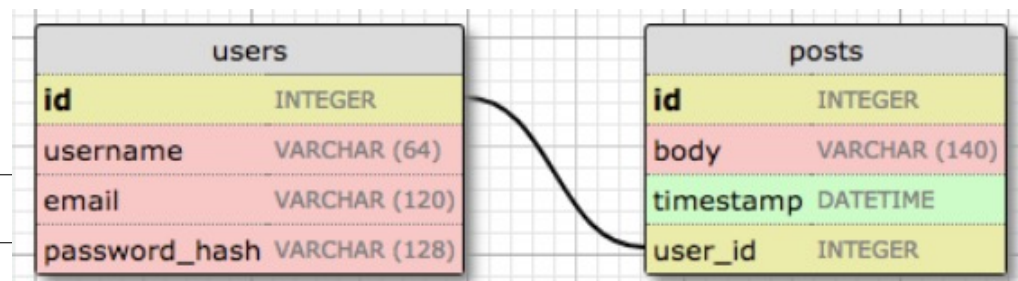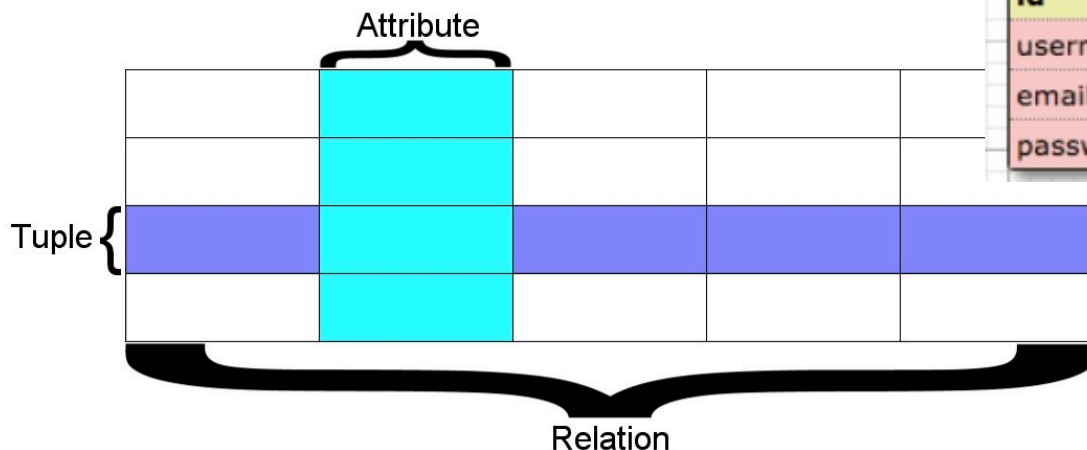- We will focus on relational databases and particularly SQLite, but we will discuss alternatives.

RELATIONAL VS. NON-RELATIONAL DATABASES

Upwork

Blog Post   Blog Tags

A non-relational database does not incorporate the table model. Instead, data can be stored in a single document file.

BLOG POST

Comments   Tags

Blog Comments

Categories

"field"

A relational database table organizes structured data fields into defined columns.

All other related data

- 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 of the table 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.*

Setting up a database

- The DataBase Management System DBMS is an application that controls access to a database.
- 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:$
```

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

```
1  CREATE TABLE contact_groups (
2    contact_id integer,
3    group_id integer,
4    PRIMARY KEY (contact_id, group_id),
5    FOREIGN KEY (contact_id) REFERENCES contacts (contact_id)
6    ON DELETE CASCADE ON UPDATE NO ACTION,
7    FOREIGN KEY (group_id) REFERENCES groups (group_id)
8    ON DELETE CASCADE ON UPDATE NO ACTION
9  );
```

Relational Query Language

- The basic operations of any database system are *Create*, *Read*, *Update* and *Delete* (**CRUD**). The sequential query language (SQL) provides the syntax for performing these 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.

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

```
1  INSERT INTO table1 (
2    column1,
3    column2 ,..)
4  VALUES
5    (
6    value1,
7    value2 ,....);
```
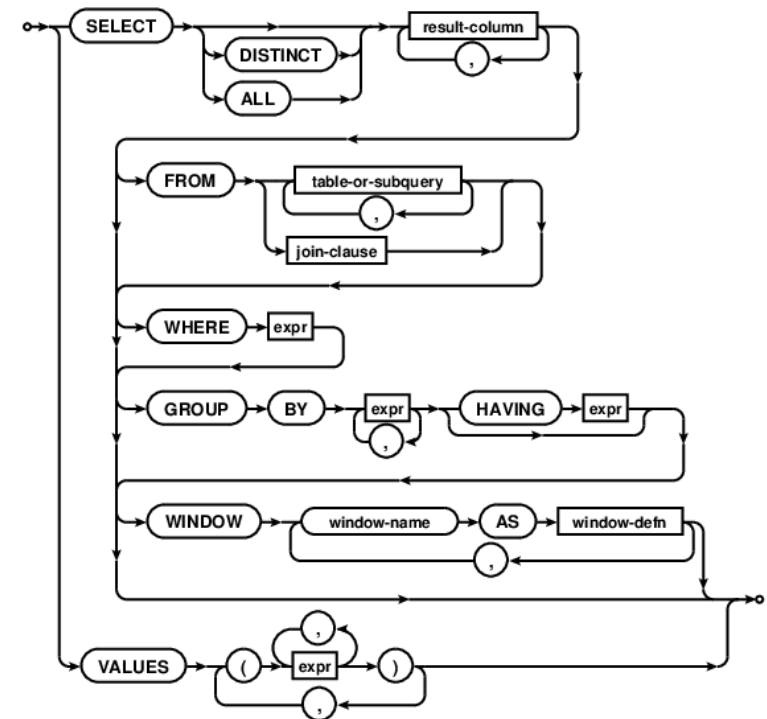
```
1  UPDATE table
2  SET column_1 = new_value_1,
3      column_2 = new_value_2
4  WHERE
5      search_condition
6  ORDER column_or_expression
7  LIMIT row_count OFFSET offset;
```

```
1  SELECT DISTINCT column_list
2  FROM table_list
3    JOIN table ON join_condition
4  WHERE row_filter
5  ORDER BY column
6  LIMIT count OFFSET offset
7  GROUP BY column
8  HAVING group_filter;
```

```
1  DELETE
2  FROM
3    table
4  WHERE
5    search_condition;
```

NoSQL

- NOSQL standards for not only SQL, and describes non-relational databases.
- These can be very useful in some applications, but RDMS are still be far the most popular and general approach.

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	Graph databases	Key-value databases	Wide column stores
Store data elements in document-like structures that encode information in formats such as JSON.	Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.	Use a simple data model that pairs a unique key and its associated value in storing data elements.	Also called table-style databases—store data across tables that can have very large numbers of columns.
+	+	+	+
Common uses include content management and monitoring Web and mobile applications.	Common uses include recommendation engines and geospatial applications.	Common uses include storing clickstream data and application logs.	Common uses include Internet search and other large-scale Web applications.
+	+	+	+
EXAMPLES: Couchbase Server, CouchDB, MarkLogic, MongoDB	EXAMPLES: Allegrograph, IBM Graph, Neo4j	EXAMPLES: Aerospike, DynamoDB, Redis, Riak	EXAMPLES: Accumulo, Cassandra, HBase, Hypertable, SimpleDB

©2017 TECHTARGET. ALL RIGHTS RESERVED TechTarget

Linking Models into an App

- Now we have a database setup, we would like to link it into our application. We will use SQL-Alchemy for ORM with SQLite. Alternatively, we could use pymongo with Mongo or py2neo with Neo4J.
- We need to install `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 `config.py`
- 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)
- The `models` classes define the database schema.

```
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
```

```
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
```

SQLAlchemy Models

- To build a model we import db (the instance of SQLAlchemy) and our models are then all defined to be subclasses of `db.Model`
- To see what these modules are doing, you can find the source code in the virtual environment directory.
- `db.Column` is a class used to specify the type and constraints of each column in the table.
- `db.relationship` is a function that defines attributes based on a database relationship.

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

```
674        self.use_native_unicode = use_native_unicode
675        self.Query = query_class
676        self.session = self.create_scoped_session(session_options)
677        self.Model = self.make_declarative_base(model_class, metadata)
678        self._engine_lock = Lock()
679        self.app = app
680        _include_sqlalchemy(self, query_class)
681
682        if app is not None:
683            self.init_app(app)
684
685    @property
686    def metadata(self):
```
`irtual-environment/lib64/python3.6/site-packages/flask_sqlalchemy/__init__.py`

```python
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)

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)
```

Database Initialisation

- This allows us to define the database schema, but we still need to link it to the database. Flask provides some utilities to do this.
- `flask db init` will initialise a database to synchronize with the models you have defined.
- `flask db migrate` will use *alembic* to create a migration script that applies changes to the datatbase.
- `flask db upgrade` applies that script to the database (and `downgrade` to roll the changes back.)
- This allows us to keep the database schema and the models in sync.

```
app/models.py: Posts database table and relationship

from datetime import datetime
from app import db


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)


class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

```
(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.
```

```
(venv) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on '['email']'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on '['username']'
  Generating /home/miguel/microblog/migrations/versions/e517276bb1c2_users_table.py ... done
```

```
(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
```

Alchemy Syntax

- We are now able to access the models from within the flask shell.
- `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()`
- The `db.session` object will synchronize with the database when we `commit` or `flush`
- We can extract entities from the database using a query.
- `<model>.query.all()` or `session.query(<model>).all ()` will return all entities of type model.

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

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

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

```
>>> 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()
```

SQL-Alchemy Queries

- The `query` object is used to wrap an SQL select statement.
- `query.get()` will extract a single element by id, and `query.all()` will return the full collection.
- We can also perform inner joins (`query.join()`), left-outer-joins (`query.outerjoin()`), and filter (`filter_by()`)and sort (`order_by()`) the results in the query syntax.

```python
query = (model.Session.query(model.Entry)
         .join(model.ClassificationItem)
         .join(model.EnumerationValue)
         .filter_by(id=c.row.id)
         .order_by(model.Entry.amount) # This row :)
        )
```

```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)]
>>>
```

Linking in with views and controllers

- We can now respond to requests for data, by building models from the database, and then populating views with the data.
- As the code is getting complex, it is a good idea to have a Controllers.py class, rather than handling everything in routes.py

```python
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
```

```python
@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
    form=ProjectForm()#initialise with parameters
    form.lab.choices= get_labs(project.lab_id)
    if form.validate_on_submit():#for post requests
        lab=Lab.query.filter_by(lab_id=form.lab.data).first()
        if lab is None or not (lab.lab_id==project.lab_id or lab.is_available()):
            flash("Lab not available")
        else:
            project.description = form.project_description.data
            project.lab_id=lab.lab_id
            db.session.add(project)
            db.session.commit()
            return redirect(url_for("index"))
    return render_template('edit_project.html', student=current_user, partner=partner, project=project,
```

```html
{% extends "base.html" %}

{% block content %}
<h2>Edit Project</h2>

<div class="container">
    <h4>{{student.prefered_name}}
        {% if not partner == None %}
            and
            {{partner.prefered_name}}
        {% endif %}'s Project Page
    </h4>

    <form name='registerProject' action='' method='post' novalidate>
        <div class='form-group'>
            {{form.hidden_tag()}}
            <p>
            {{ form.project_description.label }}<br>
            {{ form.project_description(size=20, default=project.description) }}
            {% for error in form.project_description.errors %}
            <span style="color:red;">[{{ error}}]</span>
            {% endfor %}
            </p>
            <p>
            {{ form.lab.label }}<br>
            {{ form.lab}}
            </p>
            <p> {{ form.submit() }}</p>
        </div>
    </form>
    <h6>Cannot change partner's with in a project. To dissolve a team, delete
{% endblock %}
```
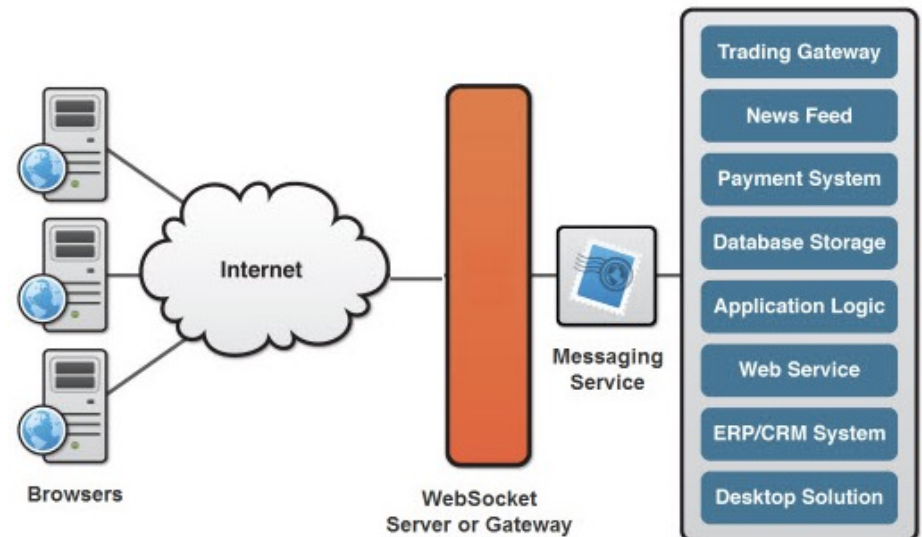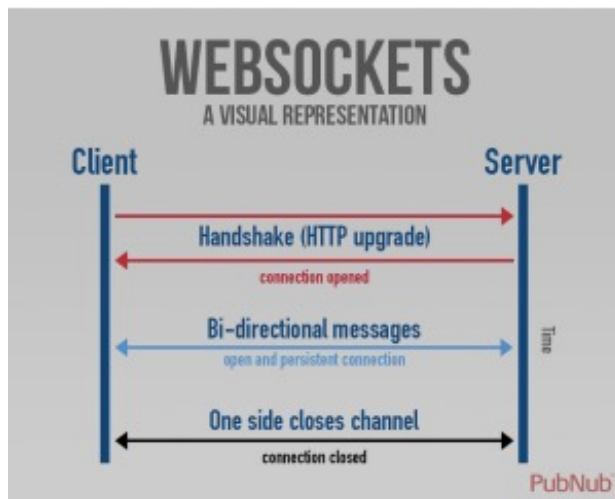
Websockets

- HTTP requests satisfy the 6 REST fundamentals, but many web applications depend on real time interaction.

- Websockets were standardise in 2011 as a means to provide full duplex communication.

- WebSockets allow your client-side JavaScript to open a persistent connection (stream) to the server.

- This allows real time communication in the application without having to send HTTP requests.

SocketIO

- Websockets are supported in Flask via the package flask-socketIO (see https://flask-socketio.readthedocs.io/en/latest/)

- SocketIO is good for message passing chat or distributed games.

- For direct video and audio, WebRTC can be used (peer-to-peer).

- Clients can connect to a socket on a server, and then the server can push messages to clients.

- The client has a *listener* architecture so it will respond to the push immediately.

Sockets in a Flask Project

- Sockets mirror the routes architecture of a Flask project, but instead of listening for requests, they listen for messages and actions, and broadcast to all listening clients.

- The server works as a common blackboard for the session (or room) and the clients implement a listening architecture via jQuery.

- The socketIO architecture maintains rooms that users/processes can subscribe to.

- Clients and server interact by emitting events including *join, status, message,* and *leave.* You can also create customised events for clients to create and receive.

- We will follow a simple demonstration from Miguel Grinberg taken from: https://github.com/miguelgrinberg/Flask-SocketIO-Chat

Setting up the server

- We use a similar architecture. A main folder called *main*, containing a *forms.py* for registration, *routes.py* for handling login, and a *events.py* file for handling the socket events.

- The socketio includes a decorator to match incoming messages with python methods.

- We don't use models, as there is no persistence here.

```python
from flask import session
from flask_socketio import emit, join_room, leave_room
from .. import socketio


@socketio.on('joined', namespace='/chat')
def joined(message):
    """Sent by clients when they enter a room.
    A status message is broadcast to all people in the room."""
    room = session.get('room')
    join_room(room)
    emit('status', {'msg': session.get('name') + ' has entered the room.'}, room=room)


@socketio.on('text', namespace='/chat')
def text(message):
    """Sent by a client when the user entered a new message.
    The message is sent to all people in the room."""
    room = session.get('room')
    emit('message', {'msg': session.get('name') + ':' + message['msg']}, room=room)


@socketio.on('left', namespace='/chat')
def left(message):
    """Sent by clients when they leave a room.
    A status message is broadcast to all people in the room."""
    room = session.get('room')
    leave_room(room)
    emit('status', {'msg': session.get('name') + ' has left the room.'}, room=room)
```

Implementing the front-end

- We use jQuery to send events to the server, listen for events coming from the server, and update the DOM accordingly.

Flask-SocketIO-Chat: Chatroom

```
<Tim has entered the room.>
<Miguel has entered the room.>
Tim:Hi Miguel, thanks for the excellent tutorials!
Miguel:No worries Tim. I hope your students find them useful
```

Enter your message here

Leave this room

```html
1  <html>
2    <head>
3      <title>Flask-SocketIO-Chat: {{ room }}</title>
4      <script type="text/javascript" src="//code.jquery.com/jquery-1.4.2.min.js"></script>
5      <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/socket.io/1.3.6/socket.io.min.js"></script>
6      <script type="text/javascript" charset="utf-8">
7        var socket;
8        $(document).ready(function(){
9          socket = io.connect('http://' + document.domain + ':' + location.port + '/chat');
10         socket.on('connect', function() {
11           socket.emit('joined', {});
12         });
13         socket.on('status', function(data) {
14           $('#chat').val($('#chat').val() + '<' + data.msg + '>\n');
15           $('#chat').scrollTop($('#chat')[0].scrollHeight);
16         });
17         socket.on('message', function(data) {
18           $('#chat').val($('#chat').val() + data.msg + '\n');
19           $('#chat').scrollTop($('#chat')[0].scrollHeight);
20         });
21         $('#text').keypress(function(e) {
22           var code = e.keyCode || e.which;
23           if (code == 13) {
24             text = $('#text').val();
25             $('#text').val('');
26             socket.emit('text', {msg: text});
27           }
28         });
29       });
30       function leave_room() {
31         socket.emit('left', {}, function() {
32           socket.disconnect();
33
34           // go back to the login page
35           window.location.href = "{{ url_for('main.index') }}";
36         });
37       }
38     </script>
39   </head>
40   <body>
41     <h1>Flask-SocketIO-Chat: {{ room }}</h1>
42     <textarea id="chat" cols="80" rows="20"></textarea><br><br>
43     <input id="text" size="80" placeholder="Enter your message here"><br><br>
44     <a href="#" onclick="leave_room();">Leave this room</a>
45   </body>
46 </html>
```

Other applications for sockets

- Sockets can be used for distributing real time events such as real-time scoreboards, stock prices, or weather.

- Implementing user-ids and sessions (next lecture) can allow you to have private chats between two users.

- Socket.io allows you to group sockets into namespaces and rooms, which allows you to control who can access and post messages.

```python
from flask_socketio import join_room, leave_room

@socketio.on('join')
def on_join(data):
    username = data['username']
    room = data['room']
    join_room(room)
    send(username + ' has entered the room.', room=room)

@socketio.on('leave')
def on_leave(data):
    username = data['username']
    room = data['room']
    leave_room(room)
    send(username + ' has left the room.', room=room)
```