Implementation of website

The web application was coded in Python using the Flask Framework. We are grateful for professor De Martini and his colleges to teaching key points of Python and Flask Framework. Flask allowed us for an easy implementation of the database through SQLAlchemy database as well as an direct interaction with the front-end using Jinja2. It also provided useful extensions such as Flask-Login to manage the user logins and WTForms to manage the forms on the website. Pycharm was used to structure the application.

Backend

The Back End of a website is typically coded in a programming language like, python, ruby etc which you will not see since the code is on the server and not sent to the client like your browser. And we planned to divide backend of the web application can be divided into three main categories:

* The Database – SQLAlchemy
* The Routes – Jinja2
* The Forms – WTForms Extension

The database is implemented with a library in Flask called. This help us to define a table in the database, using the similar object oriented approach, by defining classes. For instance the class User which we used in our website implementation is implemented as follows:

class User(db.Model, UserMixin):  
 id = db.Column(db.Integer, primary\_key=True)  
 username = db.Column(db.String(20), unique=True, nullable=False)  
 email = db.Column(db.String(120), unique=True, nullable=False)  
 password = db.Column(db.String(60), nullable=False)  
 meetings = db.relationship('Meeting', backref='author', lazy=True)

def \_\_repr\_\_(self):  
 return 'User ({},{}!)'.format(self.username, self.email)

The User class has some supplementary methods in order to define the properties of the current user. These are part of the Flask-Login extension that helps manage user sign in sessions. With this extension we we were able to limit the access to various pages of the site for only the users that were currently signed in.

Another class is called Meeting. This class represents the meeting class parameters like id, subject, date\_posted and etc). This class is interconnected with class User. Means that each user can create it’s own meeting and it will be added to database. In our website user can see how many meetings he or she created

class Meeting(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 subject = db.Column(db.String(100), nullable=False)  
 date\_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)  
 description = db.Column(db.Text, nullable=False)  
 location = db.Column(db.String, nullable=False)  
 date = db.Column(db.Text, nullable=False)  
 time = db.Column(db.String, nullable=False)  
 user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

class Current\_Meeting represents the meeting created by unique user\_id. In our program when user creates it is own meeting another user can join and to show how many users are connected and prevent error in joining created database current\_meeting

class Current\_Meeting(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 meeting\_id = db.Column(db.Integer)  
 user\_id = db.Column(db.Integer, nullable=False)

Our last class is FileContent. In this class user can share study materials with others. After uploading file another user has a chance to download uploaded material. In this class we used path model which makes easier to operate with other relationships.

class FileContent(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String(300))  
 path = db.Column(db.String)  
 user\_id = db.Column(db.String, nullable=False)

Flask-WTF extension provides integration with the WTForms library for python. Meet up friends project we used 3 forms in the web application (RegistrationForm, LoginForm, MeetingForm):

class RegistrationForm(FlaskForm):  
 username = StringField('Username',validators=[DataRequired(Length), Length(min=4, max=20)])  
 email = StringField('Email',validators=[DataRequired(Length), Email()])  
 password = PasswordField('Password', validators=[DataRequired(),Length(min=6)])  
 confirm\_password = PasswordField('Confirm Password', validators=[DataRequired(), EqualTo('password')])  
 submit = SubmitField('Sign Up')  
  
 def validate\_username (self, username):  
 user = User.query.filter\_by(username=username.data).first()  
 if user:  
 raise ValidationError('That username is already taken. Please choose another one')  
  
 def validate\_email(self, email):  
 user = User.query.filter\_by(email=email.data).first()  
 if user:  
 raise ValidationError('That email is already taken. Please choose another one')  
  
  
class LoginForm(FlaskForm):  
 email = StringField('Email',validators=[DataRequired(Length), Email()])  
 password = PasswordField('Password', validators=[DataRequired(), Length(min=6)])  
 submit = SubmitField('Login')  
  
class MeetingForm(FlaskForm):  
 subject = StringField('Subject', validators=[DataRequired()])  
 description = TextAreaField('Description', validators=[DataRequired()])  
 location = StringField('Location', validators=[DataRequired()])  
 date =StringField('Date', validators=[DataRequired()])  
 time = StringField('Time', validators=[DataRequired()])  
 submit = SubmitField('Submit')

As we can see, every class is a subclass of the class FlaskForm, meaning that it inherits all the characteristics of the superclass, for example the input type of a form, such as StringField or SubmitField. WTForms is a powerful tool also because it performs all the validation checks required for the fields, so there’s no need to use javascript in the HTML pages.

**Routes**

The main working logic of the backend is routing between the html pages and the server. In Flask you can route a url to a specific function as can be seen below.

@app.route('/register', methods=['GET', 'POST'])

The Get or Get/Post methods define whether the information is only being retrieved or also sent to the server. Every time the user request the URL example.html the following part of code will be executed. The backend has to return a page to the browser, and to do that it will use a command called “render template”, sending also the dynamic variables in the HTML code.

return render\_template('register.html', title='Register', form=form)

Different pages can be rendered depending on predefined conditions. As an example I can render the sign in page to a user who is not logged in and is trying to access the profile.html page. One of the routes from the applications is concerned with the signup page and is follows:

@app.route('/register', methods=['GET', 'POST'])  
def register():  
 if current\_user.is\_authenticated:  
 return redirect('/')  
 form = RegistrationForm()  
 if form.validate\_on\_submit():  
 hashed\_password = bcrypt.generate\_password\_hash(form.password.data)  
 user = User(username=form.username.data, email=form.email.data, password=hashed\_password)  
 db.session.add(user)  
 db.session.commit()  
 user = User.query.filter\_by(email=form.email.data).first()  
 if user and bcrypt.check\_password\_hash(user.password, form.password.data):  
 login\_user(user)  
 return redirect('/')  
 return render\_template('register.html', title='Register', form=form)

First of all, we can see that things take two completely different paths de- pending on whether the request method is a GET or a POST. In the first case it means that the user is just asking for the registration page, and they haven’t submitted the form yet. Once they do, the request will be a POST request and they will be redirected to their profile following a successful registration or to the same page again in case of an unsuccessful one.

Most routes in the application have a similar structure with functions for specific purposes such as displaying the available meetings to a user or the descriptions of attendees coming to that meeting. As we see below first user enters required information then submit it. System checks do all information entered correctly or not. After checking entered information new meeting will be added to the database. What is a database. A database is an organized collection of [data](https://en.wikipedia.org/wiki/Data_(computing)" \o "Data (computing)), generally stored and accessed electronically from a computer system.  db.session.add(meeting)– stores data to database

@app.route("/meeting/new\_meeting", methods=['GET', 'POST'])  
@login\_required  
def new\_meeting():  
 form = MeetingForm()  
 if form.validate\_on\_submit():  
 meeting = Meeting(subject=form.subject.data, description=form.description.data, location=form.location.data,  
 date=form.date.data, time=form.time.data, author=current\_user)  
 db.session.add(meeting)  
 db.session.commit()  
 flash('Your meeting created!', 'success')  
 return redirect(url\_for('meetings'))  
 return render\_template('new\_meeting.html', title='New Meeting', form=form, legend='New Meeting')

**Frontend**

Web pages were written in HTML and CSS using Jinja2 templating language for the dynamic parts of the page. The dynamic part of the webpage goes inside the curly brackets.

{% for meeting in meetings %}

{% if messages %}

Jinja2 also allows the web pages to inherit from each other. We used this concept to create a base structure of the website, with the background, the logo, the menu and other objects that are constant. This is achieved by creating a block in the base structure.

{% block title %}  
 {% if title %}  
 Meetupfriends - {{ title }}  
 {% else %}  
 Meetupfriends  
 {% endif %}  
{% endblock %}

Other web pages can then inherit from the base webpage as follows:

{% extends "layout.html" %}

{% extends "bootstrap/base.html" %}

{% import "bootstrap/wtf.html" as wtf %}

Using this structure makes it easier to manage the website especially when basic structural changes need to be made. Also we used bootstrap to in order to give style to our for our website. Writing from scratch takes a lot of time, so thanks to Bootstrap to help and prevent spending more time and effort for style of website

List of Tools

Pycharm

BootStrap

Flask

HTML

CSS

WhatsApp

Google Drive