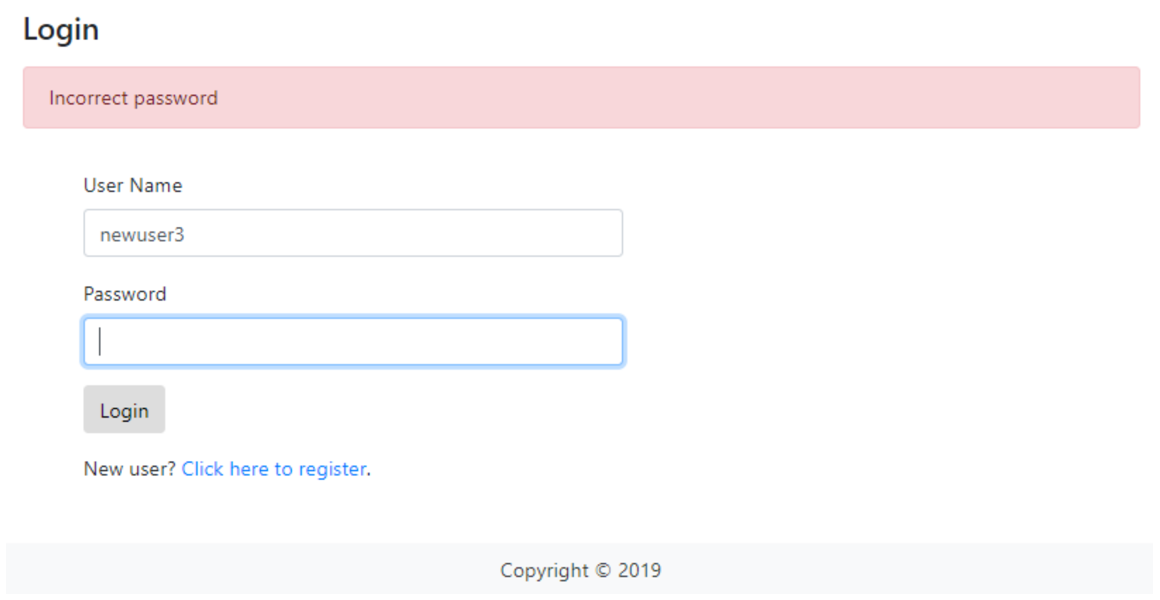
Supporting User Authentication



Incorrect password

User Name

newuser3

Password

New user? Click here to register.

Copyright © 2019

Aims of this lecture

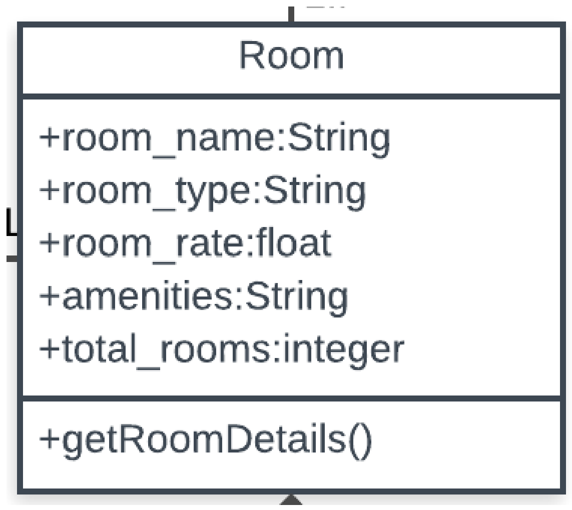
- Authentication of user
  - Password-based authentication

- Supporting Form validations
- Flashing Messages

- *Specific Form fields*
  - *File field*

Recap - Object to Relational Mapping

- Data should be represented as relations (tables).



room_table					
id	room_name	room_type	room_rate	amenities	total_rooms
0111	Standard room with two queen beds	Standard	110	free wifi	30
0112	something	King	120	free wifi, breakfast included	20
0118	something	Queen	110	free wifi	20
0119	Suitable for three adults	Triple	180	free wifi	10

Creating a Class that Maps to a Table

- Create a class with base class **`db.Model`**
- List all the columns , their types, and column options

```
class User(db.Model):
    __tablename__='users' # table name
    id = db.Column(db.Integer, primary_key=True) # auto generated

    name = db.Column(db.String(100), index=True, unique=True, nullable=False)
    emailid = db.Column(db.String(100), index=True, nullable=False)
    password_hash = db.Column(db.String(255), nullable=False)
```

```
    def __repr__(self): #string print method
            return "<Name: {}, ID: {}>".format(self.name, self.id)
```

Column Types and Column Options

Column Type Name	Description
Integer	32 bit integer : int
String	Variable length string : str
Text	Variable length string optimized for large unbounded text : str
Date	DateTime.date
Time	DateTime.time
DateTime	DateTime.datetime
Numeric	Fixed point number : Decimal
Boolean	Boolean value

Options	Description
primary_key	Set to True, if column primary key
unique	True, duplicate values in the columns are not allowed – e.g. email id has to be unique in all the rows
index	True, if the column needs to be indexed, if you want any retrieve based on the column, it is good to set it to True.
nullable	True if this column can have empty values
default	A value that would be default. E.g if the column is not nullable, it would be good to assign a default value

Creating a User in Python Terminal

```
from travel import db
from travel.models import User
from travel import create_app

app=create_app()  # create Flask App

ctx= app.app_context()  #create a context (required for python command line executions)
ctx.push()

db.create_all()   # create DB tables

#create first user object
user1 = User(name='FirstUser', emailid='user@x.com', password_hash='test')
db.session.add(user1)

#no ID as it has not been stored in the DB
print(user1)
<Name: FirstUser, ID: None>

db.session.commit()
print(user1)
# has the ID updated
<Name: FirstUser, ID: 1>
```

Has not been committed/stored in the DB

ID is automatically updated – by default primary key is auto-incremented

Querying Objects

- Run a select query on a model

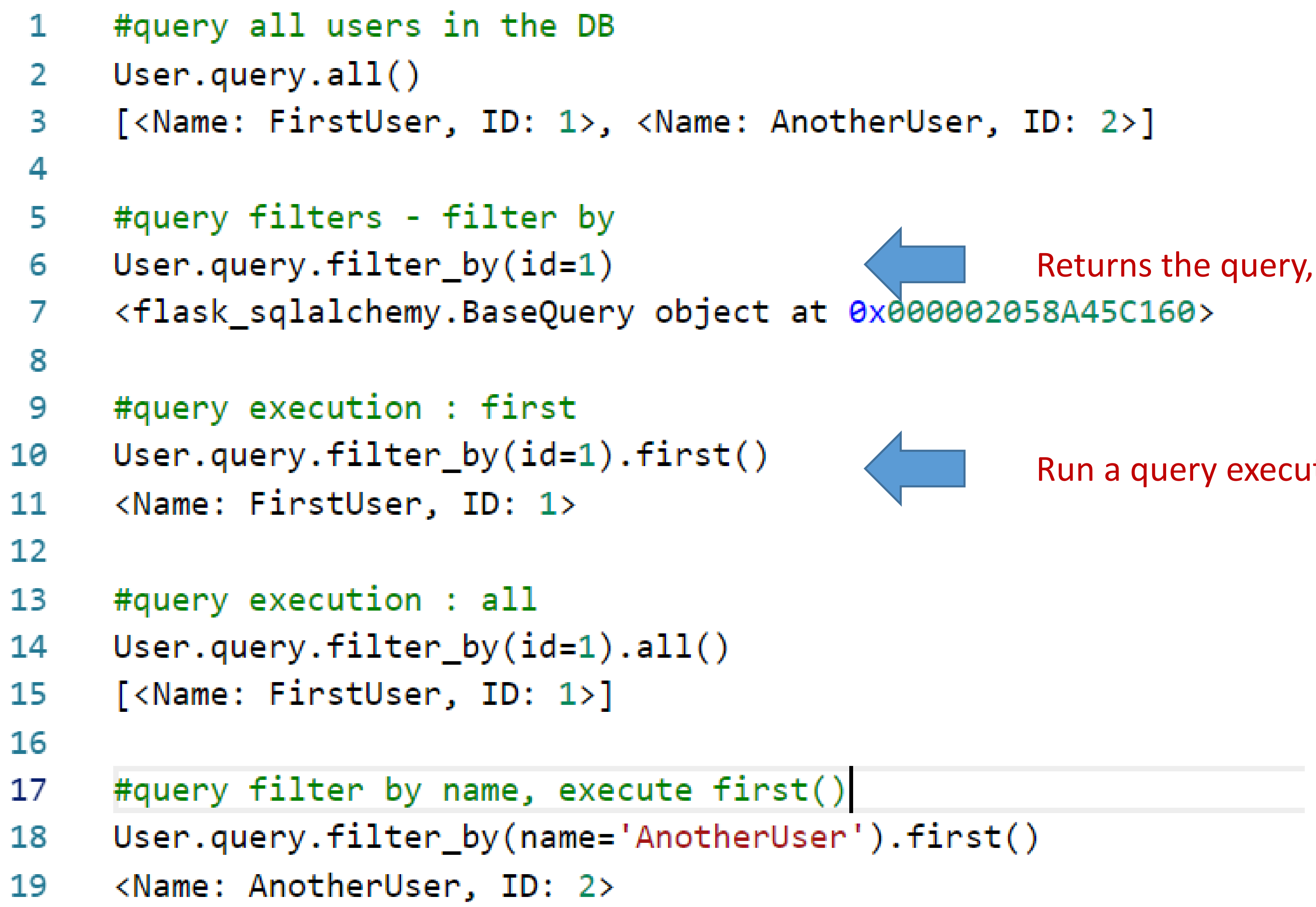
Query filters – Returns a Query

Method	Description
filter_by()	Returns a query based on the filter parameter values
filter()	Returns a query with more flexible query parameter passing
order_by()	Returns the query that orders based on a criteria
group_by()	Returns a query that groups based on the criteria

Query Executors – Executes query

Method	Description
all()	Returns all the results of the query
first()	Returns the first result of the query

Querying User Object in Python Terminal



```
#query all users in the DB
User.query.all()
[<Name: FirstUser, ID: 1>, <Name: AnotherUser, ID: 2>]

#query filters - filter by
User.query.filter_by(id=1)
<flask_sqlalchemy.BaseQuery object at 0x000002058A45C160>

#query execution : first
User.query.filter_by(id=1).first()
<Name: FirstUser, ID: 1>

#query execution : all
User.query.filter_by(id=1).all()
[<Name: FirstUser, ID: 1>]

#query filter by name, execute first()
User.query.filter_by(name='AnotherUser').first()
<Name: AnotherUser, ID: 2>
```

Returns the query, not the row/record

Run a query executor

Model one-to-many relationship

- User may post zero or more comments
- Define the Foreign Key in the Comment Object

```
class Comment(db.Model):
    __tablename__='comments'
    id = db.Column(db.Integer, primary_key=True)
    text = db.Column(db.Text, index=True)
    created_at = db.Column(db.DateTime, default=datetime.now())

    # define the foreign key - refers to <tablename.primarykey>
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

Create a relationship in the class

- Relationship in the primary class (or multiplicity = 1)

- `backref='user'` indicates the attribute in Comment class used to access User

```
comments = db.relationship('Comment', backref='user')
```

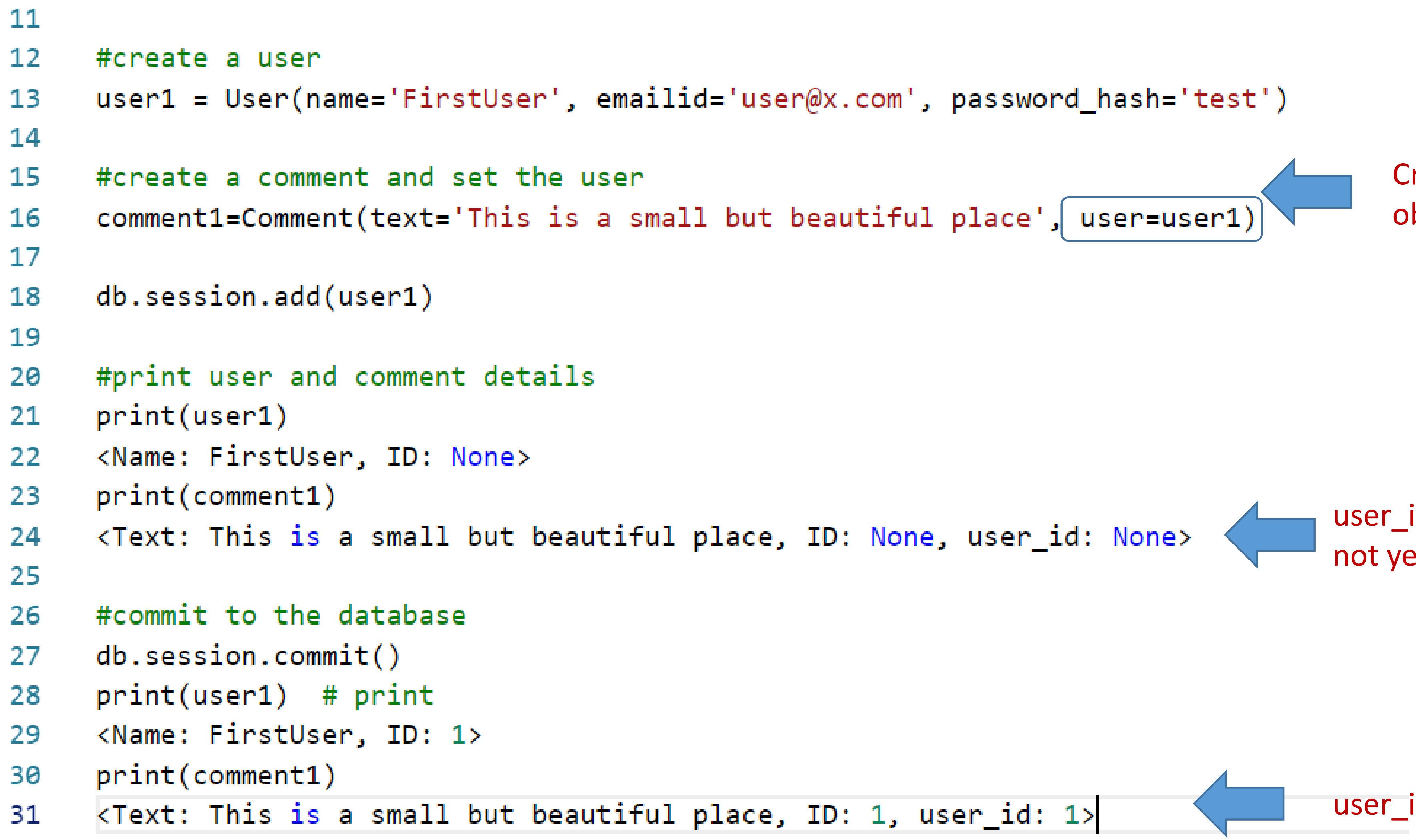
```
john = User(…)
john.comments
```

Name of the class, having the foreign key
Not the table name

user is the name of the attribute to access in the Comment clas

```
comment1 = Comment(…)
comment1.user
```

One to many relationship



```

#create a user
user1 = User(name='FirstUser', emailid='user@x.com', password_hash='test')

#create a comment and set the user
comment1=Comment(text='This is a small but beautiful place', user=user1)

db.session.add(user1)

#print user and comment details
print(user1)
<Name: FirstUser, ID: None>
print(comment1)
<Text: This is a small but beautiful place, ID: None, user_id: None>

#commit to the database
db.session.commit()
print(user1)  # print
<Name: FirstUser, ID: 1>
print(comment1)
<Text: This is a small but beautiful place, ID: 1, user_id: 1>
```

Created the two objects

user_id is None as it is not yet stored in the DB

user_id after commit

Model one-to-one relationship

- Booking has a single Owner

1. Define the Foreign Key in the Booking Object
2. Define the owner attribute in the Booking Object

```
class Booking(db.Model):
    __tablename__='bookings'
    id = db.Column(db.Integer, primary_key=True)

    #one to one relationship – specify the foreign key
    owner_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    #specify the relationship – important
    owner = db.relationship('User', foreign_keys=[owner_id])
```

Questions?

<u>Running db commands from the terminal/command line</u>

1. cd (change directory) to the folder containing the project
2. run python.exe – need to know the python path

User Authentication

- Authentication: Access the application by stating who the user is (login)

- Access based on authentication
    - User can access some or all functions of the application without a login (or as anonymous user)

    - If user needs to login for some pages/functions – direct user to login page

    - Keep track of the user throughout the session (user is using web application)

Supporting Password-based Authentication

- User Forms – Login/Register (FlaskForms)

- HTML Template to show forms

- View function: Register
  - Create a user

- View function: Login
  - Verify user name and password

Create a Login Form

- FlaskForm
- Fields - String and Password field
- Validator – InputRequired checks if the form input was provided
    - List of validators can be passed

```
class LoginForm(FlaskForm):
  username = StringField('User Name', validators=[InputRequired()])
  password = PasswordField('Password', validators=[InputRequired()])
  submit = SubmitField('Login')
```

Create Registration Form

- FlaskForm
- Fields - String and Password field
- Validator – EqualTo: name of the field that should match with the current field
- Validator – Email to check input is of valid email format

```
class RegisterForm(FlaskForm):
  username = StringField('User Name', validators=[InputRequired()])
  email = StringField('Email ID', validators=[InputRequired(),Email() ])
  #password field
  password = PasswordField('Password', validators=[InputRequired()])
  #validator to check if the user entry is equal to password
  confirm = PasswordField('Confirm Password',
          validators=[EqualTo('password', message='Re-enter same as Password')])

submit = SubmitField('Register')
```

HTML template to show form

- Bootstrap to render the form

```
<div class="col-md-6">
{{wtf.quick_form(form)}}
</div>
```

- Reuse the same HTML for registration and login

Test Login/Register forms

Message flashing

- The flashing system makes it possible to record a message at the end of a request and access it next request and only next request.

```
flash('User successfully registered')
flash (error)
```

- Access the message in the templates

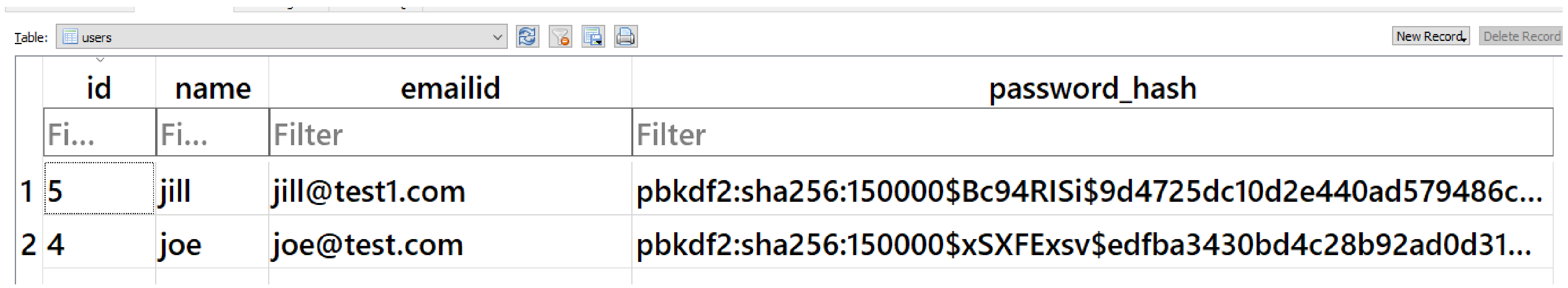
```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    {% for message in messages %}
    <div class="alert alert-info">
        {{ message }}
    </div>
    {% endfor %}
  {% endif %}
{% endwith %}
```

Access the message in the templates

Register function

- Create a new user
  - Get the name, email-id, password from the form
  - Create user in the database

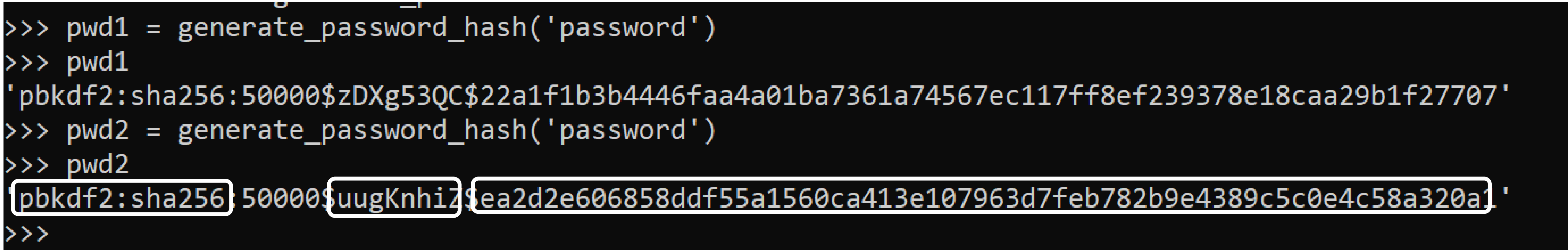
- Store the password
  - Use a hash function to ensure that password cannot be read



	id	name	emailid	password_hash
	Fi...	Fi...	Filter	Filter
1	5	jill	jill@test1.com	pbkdf2:sha256:150000$Bc94RISi$9d4725dc10d2e440ad579486c...
2	4	joe	joe@test.com	pbkdf2:sha256:150000$xSXFExsv$edfba3430bd4c28b92ad0d31...

Password Hashing

- A password hashing function takes a password as input, adds a random *salt* and then applies cryptographic transformation(s)



```
>>> pwd1 = generate_password_hash('password')
>>> pwd1
'pbkdf2:sha256:50000$zDXg53QC$22a1f1b3b4446faa4a01ba7361a74567ec117ff8ef239378e18caa29b1f27707'
>>> pwd2 = generate_password_hash('password')
>>> pwd2
'pbkdf2:sha256:50000$uugKnhiZ$ea2d2e606858ddf55a1560ca413e107963d7feb782b9e4389c5c0e4c58a320a1'
>>>
```

- Generates a unique value for the same password
- One way function – you can check if the password is correct

Register function

```python
def register():
    form = RegisterForm()
    if form.validate_on_submit():
      print('Register form submitted')

      #get username, password and email from the form
      uname =form.username.data
      pwd = form.password.data
      email=form.email.data

     # create password hash
      pwd_hash = generate_password_hash(pwd)

    #create a new user model object
      new_user = User(name=uname, password_hash=pwd_hash, emailid=email)
      db.session.add(new_user)
      db.session.commit()

    #commit to the database and redirect to HTML page
      return redirect(url_for('auth.register'))
```

Get username, password, email from the form

Create a password hash
Security function provided by Flask

Create User and commit to DB

Redirect to same URL '/register'

Login function

- Get the username and password
- Check if username exists
- Compare the password entered by user is the same as the password store in the database

Login function

```python
def login():
  form = LoginForm()
  error=None
  if(form.validate_on_submit()):
    user_name = form.username.data
    password = form.password.data
    u1 = User.query.filter_by(name=user_name).first()

        #if there is no user with that name
    if u1 is None:
      error='Incorrect user name'
    #check the password - notice password hash function
    elif not check_password_hash(u1.password_hash, password): # takes the hash and password
      error='Incorrect password'
    if error is None:
    #all good, redirect to main page
      return redirect(url_for('main.index'))
    else:
      print(error)
      flash(error)
    #it comes here when it is a get method
  return render_template('user_form.html', form=form, heading='Login')
```

Query the database for the user using user name

Check the user password hash and password entered

If username and password is correct, redirect to the main page

Questions?

Check for an authenticated user

- Check if the user is authenticated in all view functions where a logged-in user is required

```
if not 'user' in session or session['user'] is None:
  # redirect to login page
else:
  # continue as required
```

- Redirect to login form if not authenticated
- This check for every route that needs login can be tedious and error prone

Flask login provides support for login management

- pip install Flask-Login
- Derive User class from class UserMixin
    - Should have an attribute called **id** *(user_id, userid will not work)*

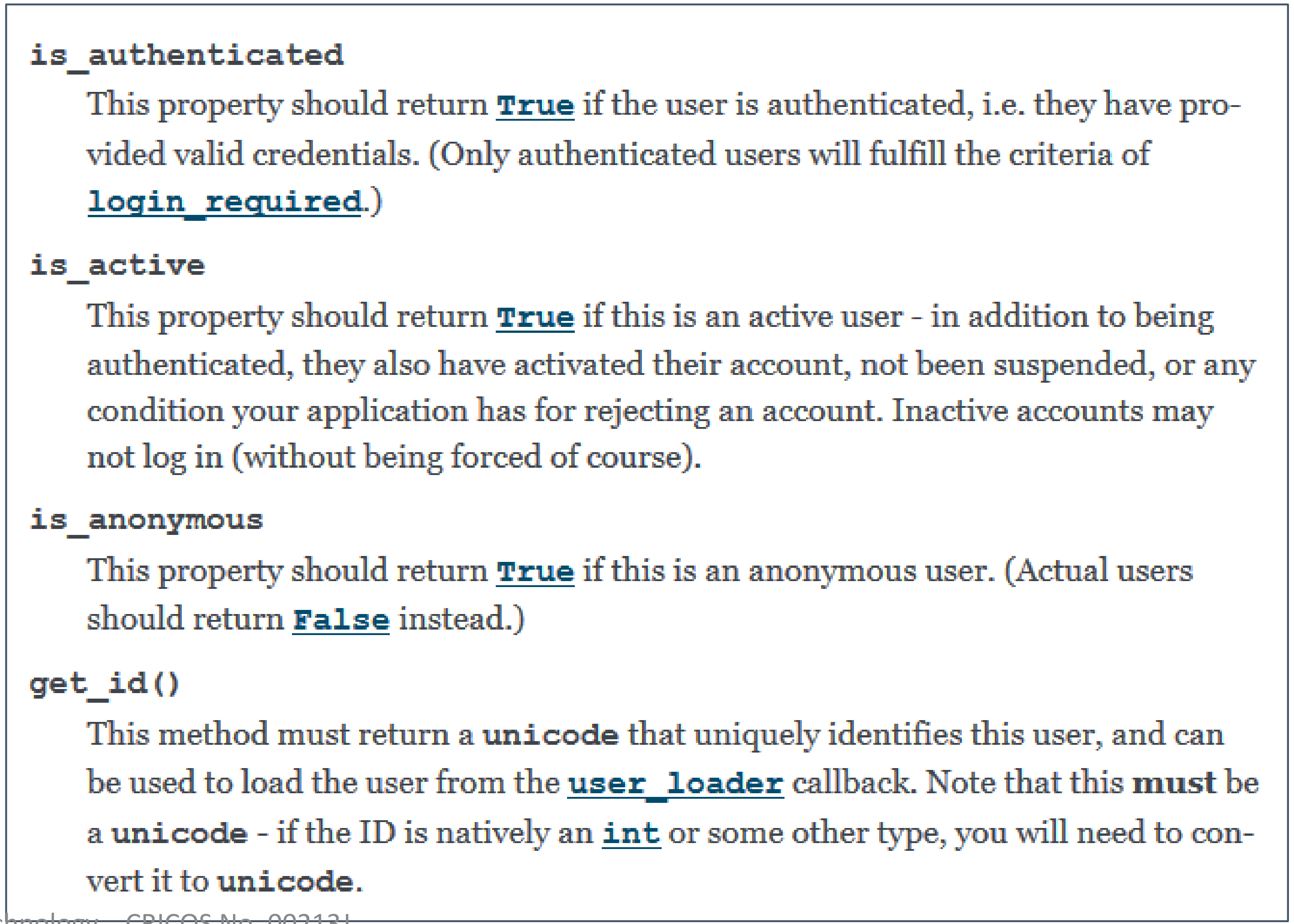
```
from flask_login import UserMixin

class User(db.Model,UserMixin):
```

1

Step1: User is-a UserMixin Class

- UserMixin has default implementation to track user state



is_authenticated

This property should return True if the user is authenticated, i.e. they have provided valid credentials. (Only authenticated users will fulfill the criteria of login_required.)

is_active

This property should return True if this is an active user - in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account. Inactive accounts may not log in (without being forced of course).

is_anonymous

This property should return True if this is an anonymous user. (Actual users should return False instead.)

get_id()

This method must return a **unicode** that uniquely identifies this user, and can be used to load the user from the user_loader callback. Note that this **must** be a **unicode** - if the ID is natively an int or some other type, you will need to convert it to **unicode**.

Step 2: Initialise LoginManager

```
#initialize the login manager
login_manager = LoginManager()

#set the name of the login function that lets user login
# in our case it is auth.login (blueprintname.viewfunction name)
login_manager.login_view='auth.login'


login_manager.init_app(app)
```

2

1. Create LoginManager
2. *Set the view function name for login*
3. Initialise LoginManager

Step 3: Retrieve user given a user ID: user_loader

```
#create a user loader function takes userid and returns User
    from .models import User
    @login_manager.user_loader

    def load_user(user_id):
        return User.query.get(int(user_id))
```

3

- Login manager functionality requires the decorator @login_manager.user_loader

- Query the user
  - get(identifier) returns the object directly

Step 4: Store the user information

- Store the information about the **User** using the login_user function
  - **User** object retrieved from the database after the successful login by a user

```
#all good, set the login_user
      login_user(logged_user)
```

4

Login support to a URL route

- @login_required decorator

```
@mainbp.route('/')
@login_required # decorator to ensure login
def index():
```

Logout function

- log_out() support available

```
@bp.route('/logout')
def logout():
  logout_user()
  return 'Successfully logged out user'
```

Access the current_user/logged-in user

```
{% if current_user.is_authenticated %}
       <a class="nav-item nav-link disabled text-muted" href="#">
              <span>Welcome, {{ current_user.name }}</span></a>
       <a class="nav-item nav-link" href="{{ url_for('auth.logout') }}">Log Out</a>
{% else %}
           <a class="nav-item nav-link" href="{{ url_for('auth.register') }}">Register</a>
           <a class="nav-item nav-link" href="{{ url_for('auth.login') }}">Log In</a>
{% endif %}
```

Questions?

Summary

- Authentication of user
  - Password-based authentication

- Supporting Form validations
- Flashing Messages

- *Specific Form fields*
  - *File field*

Thank you!