In run.py,

```
app = create_app(config_mode)
Migrate(app, db)
```

Calls **create_app** function. We have create_app() function in __init__.py.
Imports in **app/__init__.py:**

`from flask import Flask, url_for`	First we imported the **`Flask`** class. An instance of this class will be our WSGI application.To build a URL to a specific function, use the **`url_for()`** function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule.
`from flask_login import LoginManager`	Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.The login manager contains the code that lets your application and Flask-Login work together
`from flask_sqlalchemy import SQLAlchemy`	Used to control SQLAlchemy integration to one or more Flask applications.
`from importlib import import_module`	Required when importing a package.
`from logging import basicConfig, DEBUG, getLogger, StreamHandler`	Configuration for logging.

`from os import path`	The os.path module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is always in one of the different formats.
`from flask_mail import Mail`	The **Flask-Mail** extension provides a simple interface to set up SMTP with your Flask application and to send messages from your views and scripts.
`from flask_bootstrap import Bootstrap`	Flask-Bootstrap packages Bootstrap into an extension that mostly consists of a blueprint named 'bootstrap'. It can also create links to serve Bootstrap from a CDN.
`from apscheduler.schedulers.background import BackgroundScheduler`	BackgroundScheduler is **a scheduler provided by APScheduler** that runs in the background as a separate thread.
`from flask_migrate import Migrate`	**Flask-Migrate** is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic. The database operations are made available through the Flask command-line interface.

```python
def create_app(config, selenium=False):
    app = Flask(__name__, static_folder='base/static')
    app.config.from_object(config)
    # app.config['REMEMBER_COOKIE_DURATION'] = timedelta(seconds=5)
    if selenium:
        app.config['LOGIN_DISABLED'] = True
    # added for migration
    Migrate(app, db)
    register_extensions(app)
    register_blueprints(app)
    configure_database(app)
    configure_logs(app)
    apply_themes(app)
    return app
```

In create_app() function, creates a Flask Application named app.

```python
app = Flask(__name__, static_folder='base/static')
```

The *static* folder contains **assets** used by the templates, including CSS files, JavaScript files, and images. In the example, we have only one asset file, *main.css*.

1. First we imported the **`Flask`** class. An instance of this class will be our WSGI application.
2. Next we create an instance of this class. The first argument is the name of the application's module or package. `__name__` is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.

The most important part of an application that uses Flask-Login is the **LoginManager** class.

Configuring from python files.
Added for migration.

Flask-Migrate is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic. The database operations are made available through the Flask command-line interface.
Flask-Migrate exposes one class called `Migrate`. This class contains all the functionality of the extension.
The two arguments to `Migrate` are the application instance and the Flask-SQLAlchemy database instance. The `Migrate` constructor also takes additional keyword arguments, which are passed to Alembic's `EnvironmentContext.configure()` method.

Calls register_extensions(app)

```
def register_extensions(app):
    db.init_app(app)
    login_manager.init_app(app)
```

Flask-Migrate can be initialized using the `init_app` method.

```
login_manager = LoginManager()
```

LoginManager() is used to hold the settings used for logging in. Instances of Login Manager are not bound to specific apps so you can create one in the main body of your code and then bind it to your app in a factory function.
`login_manager = LoginManager()`

The login manager contains the code that lets your application and Flask-Login work together, such as how to load a user from an ID, where to send users when they need to log in, and the like.

Once the actual application object has been created, you can configure it for login with:

`login_manager.init_app(app)`

Calls register_blueprints(app)

```
def register_blueprints(app):
    for module_name in ('base', 'home'):
        module = import_module('app.{}.routes'.format(module_name))
        app.register_blueprint(module.blueprint)
```

Flask uses a concept of *blueprints* for making application components and supporting common patterns within an application or across applications.The basic concept of blueprints is that they record operations to execute when registered on an application. Flask associates view functions with blueprints when dispatching requests and generating URLs from one endpoint to another.Provide template filters, static files, templates, and other utilities through blueprints.

configure_database(app)

```
def configure_database(app):

    @app.before_first_request
    def initialize_database():
        db.create_all()

    @app.teardown_request
    def shutdown_session(exception=None):
        db.session.remove()
```

before_first_request

Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

Functions decorated with `@app.before_first_request` will run once before the first request to this instance of the application

To create the initial database, just import the `db` object from an interactive Python shell and run the **`SQLAlchemy.create_all()`** method to create the tables and database:

```
>>> from yourapplication import db
>>> db.create_all()
```

there is your database.

A **`Blueprint`** can add handlers for these events that are specific to the blueprint. The handlers for a blueprint will run if the blueprint owns the route that matches the request.

1. Before each request, **`before_request()`** functions are called. If one of these functions return a value, the other functions are skipped. The return value is treated as the response and the view function is not called.
2. If the **`before_request()`** functions did not return a response, the view function for the matched route is called and returns a response.
3. The return value of the view is converted into an actual response object and passed to the **`after_request()`** functions. Each function returns a modified or new response object.
4. After the response is returned, the contexts are popped, which calls the **`teardown_request()`** and **`teardown_appcontext()`** functions. These functions are called even if an unhandled exception was raised at any point above.

db.session.remove

Dispose of the current .session if present.

configure_logs(app)

```
def configure_logs(app):
    # soft logging
    try:
        basicConfig(filename='error.log', level=DEBUG)
        logger = getLogger()
        logger.addHandler(StreamHandler())
    except:
        pass
```

basicConfig do basic configuration for logging system.

getLogger() returns a logger with specified name creation if necessary. If no name is specified , return the root logger.

addHandler adds the specified handler i.e. StreamHandler which writes to a sys.stderr to logger. StreamHandler is a handler class which writes logging records, approximately formatted to a stream.

apply_themes(app)

```
Generates a URL to the given endpoint.
If the endpoint is for a static resource
```

Add support for themes.

If DEFAULT_THEME is set then all calls to

url_for('static', filename='')

will modfify the url to include the theme name

The theme parameter can be set directly in url_for as well:

ex. url_for('static', filename='', theme='')

If the file cannot be found in the /static/<theme>/ location then

the url will not be modified and the file is expected to be

in the default /static/ location

```
@app.context_processor
def override_url_for():
    return dict(url_for=_generate_url_for_theme)

def _generate_url_for_theme(endpoint, **values):
    if endpoint.endswith('static'):
        themename = values.get('theme', None) or \
            app.config.get('DEFAULT_THEME', None)
        if themename:
            theme_file = "{}/{}".format(themename, values.get('filename', ''))
            if path.isfile(path.join(app.static_folder, theme_file)):
                values['filename'] = theme_file
    return url_for(endpoint, **values)
```

To inject new variables automatically into the context of a template, context processors exist in Flask. Context processors run before the template is rendered and have the ability to inject new values into the template context. A context processor is a function that returns a dictionary. The keys and values of this dictionary are then merged with the template context, for all templates in the app.

```
Return app
```

app/base/forms.py

`from flask_wtf import FlaskForm`	Flask-WTF provides your Flask application integration with WTForms.
`from wtforms import TextField, PasswordField`	It is used to represent the text field HTML form element.It is used to take the password as the form input from the user.

`from wtforms.validators import InputRequired, Email, DataRequired`	DataRequired:Checks the field's data is 'truthy' otherwise stops the validation chain. InputRequired:Validates that input was provided for this field. Email:Validates an email address. Requires email_validator package to be installed.

`Login and registration`

```
class LoginForm(FlaskForm):
    username = TextField    ('Username', id='username_login'   , validators=[DataRequired()])
    password = PasswordField('Password', id='pwd_login'        , validators=[DataRequired()])
```

LoginForm has username and password with validators which checks if the field's data is truthy.

```
class CreateAccountForm(FlaskForm):
    username = TextField('Username'     , id='username_create' , validators=[DataRequired()])
    email    = TextField('Email'        , id='email_create'    , validators=[DataRequired(), Email()])
    password = PasswordField('Password' , id='pwd_create'      , validators=[DataRequired()])
```

For registration of users, it has username, email and password text fields.

app/base/models.py

In models.py, we have

- User
- Login
- Tasks
- Subtasks
- Ratings
- Notifications
- Mail
- Bin

`from flask_login import UserMixin`	Flask-Login provides a *mixin* class called `UserMixin` that includes generic implementations that are appropriate for most user model classes. UserMixin is a helper provided by the Flask-Login library to provide boilerplate methods necessary for managing users.
`from sqlalchemy import Binary, Column, Integer, String`	
`import datetime`	
`from app import db, login_manager`	
`from app.base.util import hash_pass`	

user

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

    __tablename__ = 'User'

    id = Column(Integer, primary_key=True)
    username = Column(String(80), unique=True)
    email = Column(String(80), unique=True)
    password = Column(Binary)

    def __init__(self, **kwargs):
        for property, value in kwargs.items():
            # depending on whether value is an iterable or not, we must
            # unpack it's value (when **kwargs is request.form, some values
            # will be a 1-element list)
            if hasattr(value, '__iter__') and not isinstance(value, str):
                # the ,= unpack of a singleton fails PEP8 (travis flake8 test)
                value = value[0]

            if property == 'password':
                value = hash_pass( value ) # we need bytes here (not plain str)

            setattr(self, property, value)

    def __repr__(self):
        return str(self.username)
```

4 columns for user table

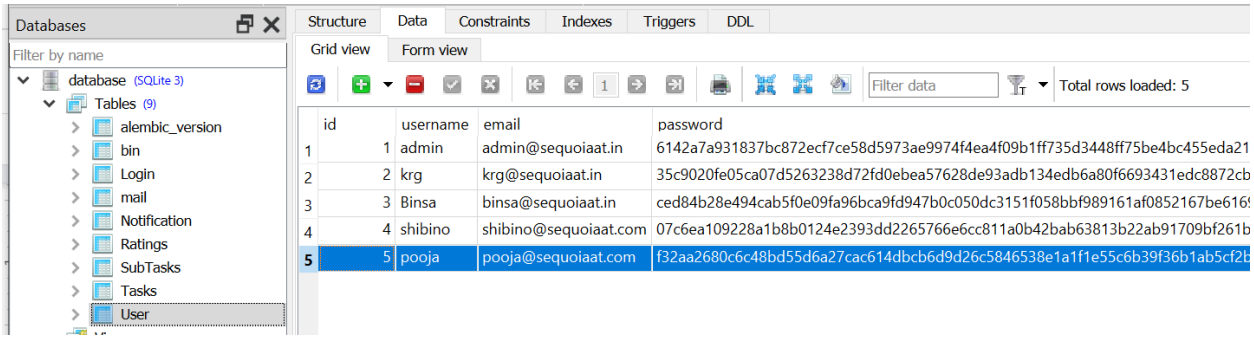
Id(setting id as primary key)

Username(string varchar(80)

Email(string varchar(80)

password(binary) password() returns a binary string

Hash_pass in utils.py



b)Login

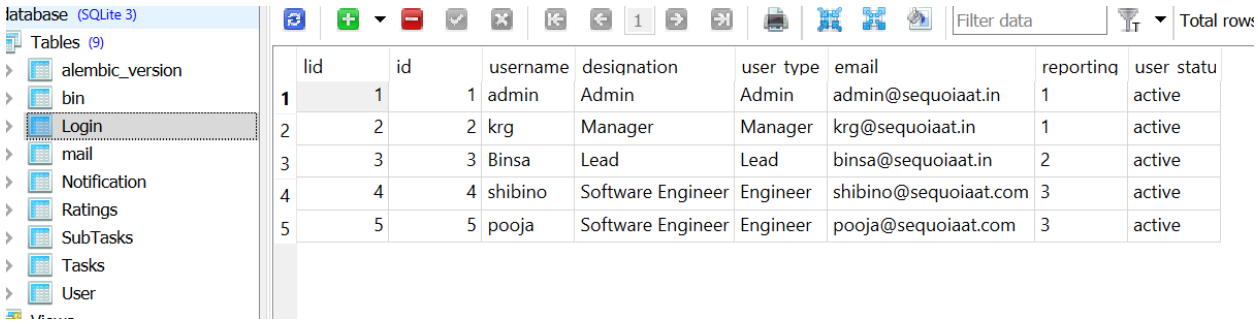
```
class Login(db.Model, UserMixin):

    __tablename__ = 'Login'

    lid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)
    id = Column(Integer, db.ForeignKey("User.id"), nullable=False)
    username = Column(String(80), unique=True, default='')
    designation = Column(String(80), default='')
    user_type = Column(String(80), default='')
    email = Column(String(80), unique=True, default='')
    # pwd = Column(String(80), unique=True, default='')
    reporting_head = Column(String(80), default='')
    user_status = Column(String(80), default='active')
    db.ForeignKeyConstraint(
        ['id'], ['User.id'],
        use_alter=True, name='login_ibfk_1'
    )

    def __repr__(self):
        return str(self.username)
```

Table arguments other than the name, metadata, and mapped Column arguments are specified using the __table_args__ class attribute. This attribute accommodates both positional as well as keyword arguments that are normally sent to the Table constructor. The attribute can be specified in one of two forms. One is as a dictionary:

```
class MyClass(Base):
    __tablename__ = 'sometable'
```

```
__table_args__ = {'mysql_engine':'InnoDB'}
```

1)lid(type int, setting autoincrement True, can't be null, set as primary key)
2)id(type int,However the foreign key has to be separately declared with the **`ForeignKey`** class,cant be null)
3)username(string varchar(80), unique=True)
4)designation(string varchar(80))
5)user_type(string varchar(80))
6)email(string varchar(80), unique=True)
7)reporting_head(string varchar(80))
8)user_status(string varchar(80))



	lid	id	username	designation	user_type	email	reporting	user_statu
1	1	1	admin	Admin	Admin	admin@sequoiaat.in	1	active
2	2	2	krg	Manager	Manager	krg@sequoiaat.in	1	active
3	3	3	Binsa	Lead	Lead	binsa@sequoiaat.in	2	active
4	4	4	shibino	Software Engineer	Engineer	shibino@sequoiaat.com	3	active
5	5	5	pooja	Software Engineer	Engineer	pooja@sequoiaat.com	3	active

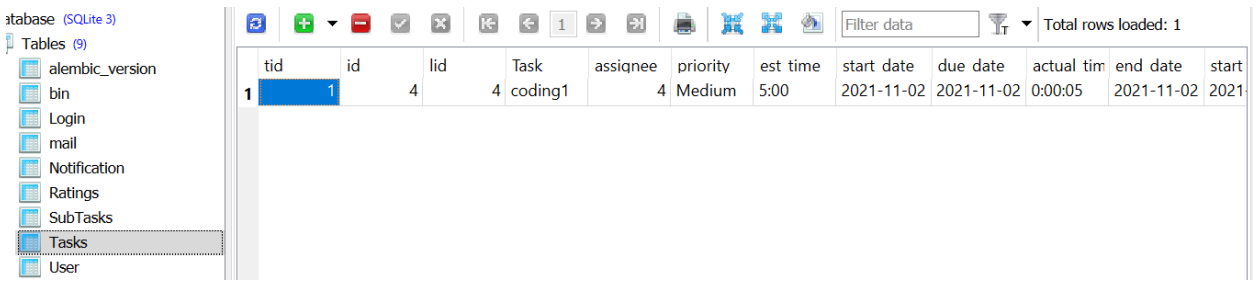
c)Tasks

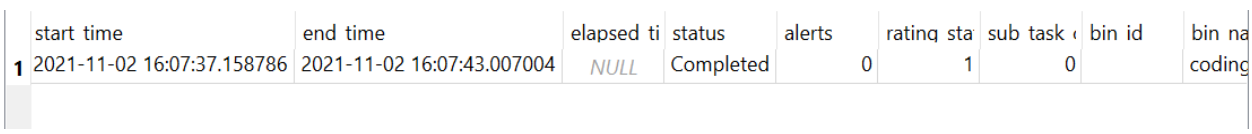
```python
class Tasks(db.Model, UserMixin):

    __tablename__ = 'Tasks'

    tid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)
    id = Column(Integer, db.ForeignKey("User.id"), nullable=False)
    lid = Column(Integer, db.ForeignKey("Login.lid"), default='')
    bin_id = Column(Integer, db.ForeignKey("bin.bin_id"), default='')
    bin_name = Column(String(120), default='')
    Task = Column(String(120), default='')
    assignee = Column(Integer, default='')
    priority = Column(String(50), default='')
    est_time = Column(String(20), default='')
    start_date = Column(db.Date, default='')
    actual_time = Column(String(20), default='', nullable=True)
    due_date = Column(db.Date, default='')
    end_date = Column(db.Date, default='', nullable=True)
    start_time = Column(db.TIMESTAMP, nullable=True)
    end_time = Column(db.TIMESTAMP, nullable=True)
    elapsed_time = Column(String(20), nullable=True)
    #elapsed_time = Column(db.TIMESTAMP, nullable=True)
    status = Column(String(20), default='Todo', nullable=True)
    alerts = Column(Integer, default=0, nullable=True)
    rating_status = Column(Integer, default=0, nullable=True)
    sub_task_count = Column(Integer, default=0, nullable=True)
```

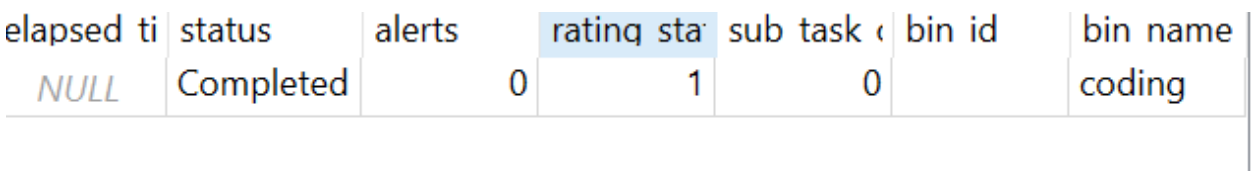
```python
    db.ForeignKeyConstraint(
        ['id'], ['User.id'],
        use_alter=True, name='tasks_ibfk_1'
    )
    db.ForeignKeyConstraint(
        ['lid'], ['Login.lid'],
        use_alter=True, name='tasks_ibfk_2'
    )

    def __repr__(self):
        return str(self.assignee)
```

1)stid =Integer, autoincrement=True, nullable=False, primary_key=True)
2)id = Column(Integer, db.ForeignKey("User.id"), nullable=False)
3)tid = Column(Integer, db.ForeignKey("Tasks.tid"), nullable=False)
4) lid = Column(Integer, db.ForeignKey("Login.lid"))

5)sub_task = Column(String(120))
6)assignee = Column(Integer')
7)priority = Column(String(50))
8)est_time = Column(String(20))
9)start_date = Column(db.Date,)
10)actual_time = Column(String(20), nullable=True)
11) end_date = Column(db.Date, nullable=True)
12)start_time = Column(db.TIMESTAMP, nullable=True)
13)end_time = Column(db.TIMESTAMP, nullable=True)
14) status = Column(String(20), default='Todo', nullable=True)
15)alerts = Column(Integer, default=0, nullable=True)
16)rating_status = Column(Integer, default=0, nullable=True)







d)Ratings

```python
class Ratings(db.Model, UserMixin):

    __tablename__ = 'Ratings'

    rid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)
    id = Column(Integer, db.ForeignKey("User.id"), default='')
    tid = Column(Integer, db.ForeignKey("Tasks.tid"), default='')
    lid = Column(Integer, db.ForeignKey("Login.lid"), default='')
    user_rating = Column(Integer, default='')
    lead_rating = Column(Integer, default='')
    manager_rating = Column(Integer, default='')
    user_comment = Column(String(150), default='')
    lead_comment = Column(String(150), default='')
    manager_comment = Column(String(150), default='')
    rev_user_rating = Column(Integer, default='')
    rev_lead_rating = Column(Integer, default='')
    rev_manager_rating = Column(Integer, default='')
    rev_user_comment = Column(String(150), default='')
    rev_lead_comment = Column(String(150), default='')
    rev_manager_comment = Column(String(150), default='')
    final_rating = Column(Integer, default='')
    db.ForeignKeyConstraint(
        ['tid'], ['Tasks.tid'],
        use_alter=True, name='Rate_ibfk_1'
    )
```

```python
db.ForeignKeyConstraint(
    ['lid'], ['Login.lid'],
    use_alter=True, name='Rate_ibfk_2'
)
db.ForeignKeyConstraint(
    ['id'], ['User.id'],
    use_alter=True, name='Rate_ibfk_3'
)

def __repr__(self):
    return str(self.final_rating)
```

1)rid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)

2)id = Column(Integer, db.ForeignKey("User.id"))

3)tid = Column(Integer, db.ForeignKey("Tasks.tid"))

4)lid = Column(Integer, db.ForeignKey("Login.lid"))

5) user_rating = Column(Integer)

6)lead_rating = Column(Integer)

7)manager_rating = Column(Integer)

8)user_comment = Column(String(150))

9)lead_comment = Column(String(150))

10)manager_comment = Column(String(150))

11)rev_user_rating = Column(Integer)

12)rev_lead_rating = Column(Integer)

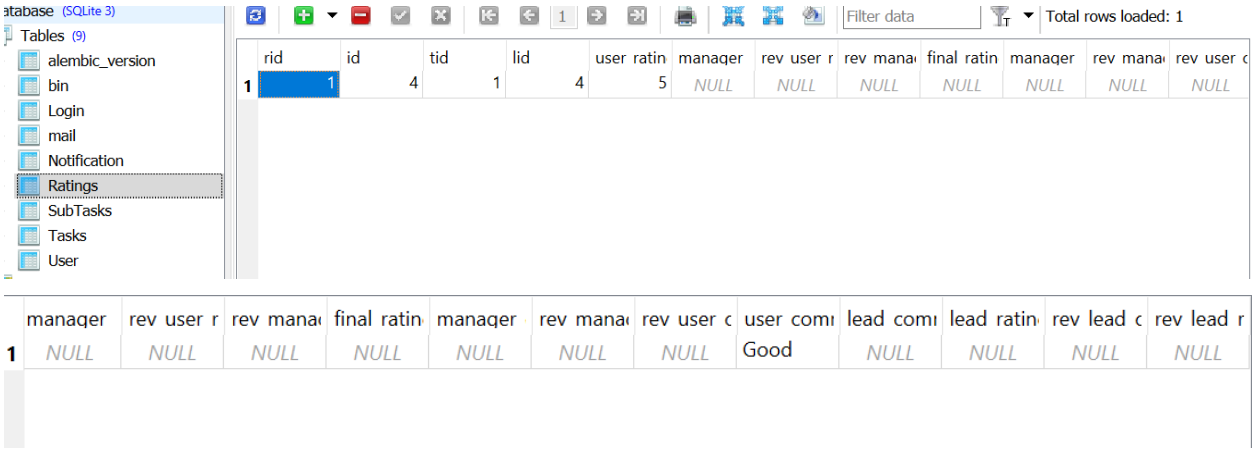
13)rev_manager_rating = Column(Integer)

14)rev_user_comment = Column(String(150))

15)rev_lead_comment = Column(String(150))

16)rev_manager_comment = Column(String(150))

17)final_rating = Column(Integer)



e)Notifications

```python
class Notifications(db.Model, UserMixin):

    __tablename__ = 'Notification'

    nid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)
    receiver = Column(String(50), default='')
    message = Column(String(150), default='')
    time = Column(db.Date,  nullable=False, default=datetime.datetime.utcnow)
    read_msg = Column(String(10), default='')
    read_msg_count = Column(Integer, default='')
    pending_msg_count = Column(Integer, default='')
    total_msg_count = Column(Integer, default='')

    def __repr__(self):
        return str(self.pending_msg_count)
```

1)nid = Column(Integer, autoincrement=True, nullable=False, primary_key=True)

2)receiver = Column(String(50)

3)message = Column(String(150))

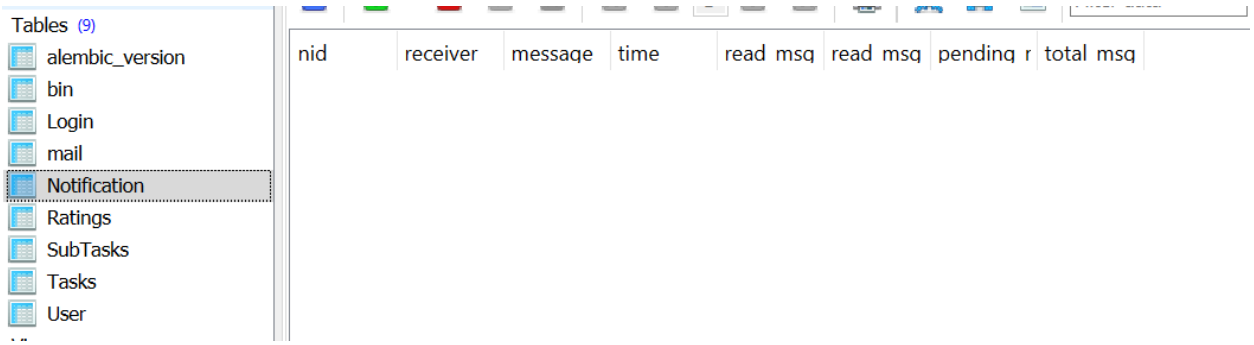
4)time = Column(db.Date, nullable=False, default=datetime.datetime.utcnow)

5)read_msg = Column(String(10))

6)read_msg_count = Column(Integer)

7) pending_msg_count = Column(Integer)

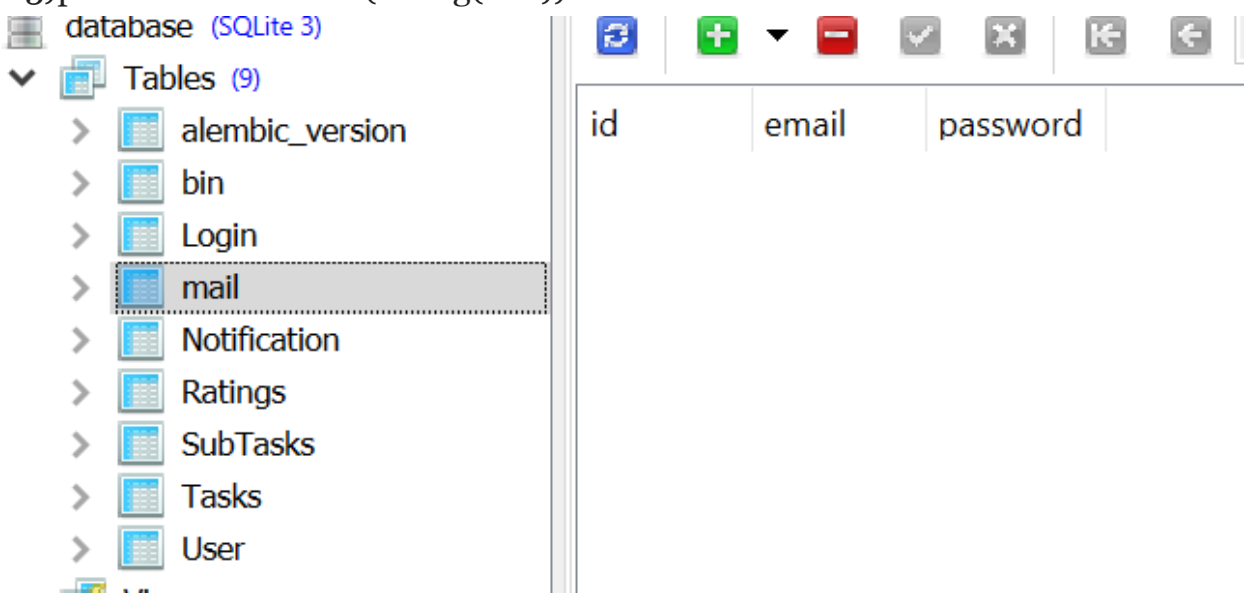
8)total_msg_count = Column(Integer)



f)Mail

```python
class Mail(db.Model, UserMixin):

    __tablename__ = 'mail'

    id = Column(Integer, default='', primary_key=True)
    email = Column(String(100), default='')
    password = Column(String(100), default='')
```

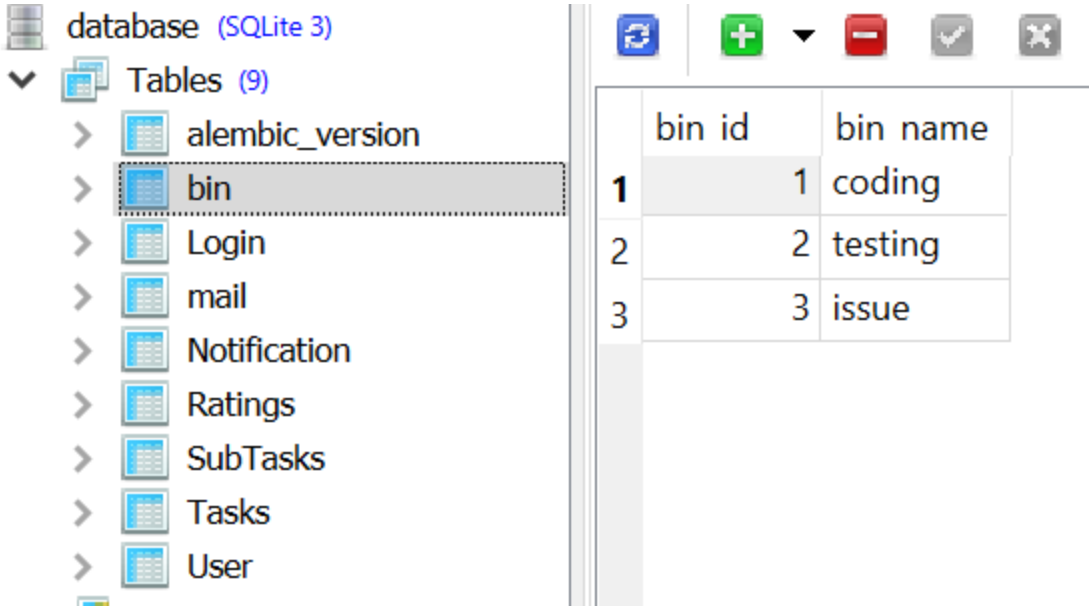
1) id = Column(Integer,, primary_key=True)
2) email = Column(String(100))
3)password = Column(String(100))



g)Bin

```python
class Bin(db.Model, UserMixin):

    __tablename__ = 'bin'

    bin_id = Column(Integer,  autoincrement=True, nullable=False, primary_key=True)
    bin_name = Column(String(100), default='')
```

1)bin_id = Column(Integer, autoincrement=True, nullable=False, primary_key=True)
2)bin_name = Column(String(100))



Once the actual application object has been created, you can configure it for login with:

```
login_manager.init_app(app)
```

You will need to provide a **user_loader** callback. This callback is used to reload the user object from the user ID stored in the session. It should take the **unicode** ID of a user, and return the corresponding user object.

By specifying user_loader, we can query who the current login user is.In this way, we will judge whether the user can login or not.

We need to take control of landing page rights. We set up the REST API to increase, delete and modify to use for the login. Only the API of the query can be easily visible.

Custom Login using Request Loader

Sometimes you want to login users without using cookies, such as using header values or an api key passed as a query argument. In these cases, you should use the **request_loader** callback. This callback should behave the same as your

user_loader callback, except that it accepts the Flask request instead of a user_id.

app/base/utils.py

`hashlib`	The Python hashlib module is **an interface for hashing messages easily**. This contains numerous methods which will handle hashing any raw message in an encrypted format. The core purpose of this module is to use a hash function on a string, and encrypt it so that it is very difficult to decrypt it.
binascii	The binascii module contains **a number of methods to convert between binary and various ASCII-encoded binary representations**.

```
def hash_pass( password ):
    """Hash a password for storing."""
    salt = hashlib.sha256(os.urandom(60)).hexdigest().encode('ascii')
    pwdhash = hashlib.pbkdf2_hmac('sha512', password.encode('utf-8'),
                                  salt, 100000)
    pwdhash = binascii.hexlify(pwdhash)
    return (salt + pwdhash) # return bytes
```

Hashing a password for storing.

- `hash_password` : Encodes a provided password in a way that is safe to store on a database or file.The first thing it does is generate some random salt that should be added to the password. That's just the `sha256` hash of some random bytes read from `os.urandom` . It then extracts a string representation of the hashed salt as a set of hexadecimal numbers (`hexdigest`).

- The salt is then provided to `pbkdf2_hmac` together with the password itself to hash the password in a randomized way. As `pbkdf2_hmac` requires bytes as its input, the two strings (password and salt) are previously encoded in pure bytes. The salt is encoded as plain ASCII, as the hexadecimal representation of a hash will only contain the 0-9 and A-F characters. While the password is encoded as `utf-8` , it could contain any character.

```
def verify_pass(provided_password, stored_password):
    """Verify a stored password against one provided by user"""
    stored_password = stored_password.decode('ascii')
    salt = stored_password[:64]
    stored_password = stored_password[64:]
    pwdhash = hashlib.pbkdf2_hmac('sha512',
                                  provided_password.encode('utf-8'),
                                  salt.encode('ascii'),
                                  100000)
    pwdhash = binascii.hexlify(pwdhash).decode('ascii')
    return pwdhash == stored_password
```

he resulting `pbkdf2` is a bunch of bytes, as you want to store it into a database; you use `binascii.hexlify` to convert the bunch of bytes into their hexadecimal representation in a string format. `Hexlify` is a convenient way to convert bytes to strings without losing data. It just prints all the bytes as two hexadecimal digits, so the resulting data will be twice as big as the original data, but apart from this, it's exactly the same as the converted data.

In the end, the function joins together the hash with its salt. As you know that the `hexdigest` of a `sha256` hash (the salt) is always 64 characters long, by joining them together, you can grab back the salt by reading the first 64 characters of the resulting string. This will permit `verify_password` to verify the password and verify whether the salt used to encode it is required.

Once you have your password, `verify_password` can then be used to verify provided passwords against it. So it takes two arguments: the hashed password and the new password that should be verified. The first thing `verify_password` does is extract the salt from the hashed password (remember, you placed it as the first 64 characters of the string resulting from `hash_password`).

The extracted salt and the password candidate are then provided to `pbkdf2_hmac` to compute their hash and then convert it into a string with `binascii.hexlify` . If the resulting hash matches with the hash part of the previously stored password (the characters after the salt), it means that the two passwords match.

If the resulting hash doesn't match, it means that the provided password is wrong. As you can see, it's very important that you make the salt and the password available together, because you'll need it to be able to verify the password and a different salt would result in a different hash and thus you'd never be able to verify the password.

app/base/routes.py

`import re`	Support for regular expressions.
`from types import FunctionType`	The type of user-defined functions and functions created by lambda expressions
`from flask import json, jsonify, render_template, redirect, request, url_for, session, flash, request`	jsonify is a function in Flask's `flask.json` module. `jsonify` serializes data to JavaScript Object Notation (JSON) format, wraps it in a Response object with the application/json mimetype. To render a template you can use the **`render_template()`** method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. To redirect a user to another endpoint, use the **`redirect()`** function; To build a URL to a specific function, use the **`url_for()`** function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule. In addition to the request object there is also a second object called **`session`** which allows you to store information

	specific to a user from one request to the next. To flash a message use the **`flash()`** method, to get hold of the messages you can use **`get_flashed_messages()`** which is also available in the templates
`import datetime`	

```
from flask_login import
current_user, login_required,
login_user, logout_user
```

flask_login.**current_user**

A proxy for the current user.

flask_login.**login_required**(*func*)If you decorate a view with this, it will ensure that the current user is logged in and authenticated before calling the actual view. (If they are not, it calls the **LoginManager.unauthorized** callback.)

flask_login.**login_user**(*user*, *remember=False*, *duration=None*, *force=False*, *fresh=True*)[source]

Logs a user in. You should pass the actual user object to this. If the user's **is_active** property is False, they will not be logged in unless **force** is True.

flask_login.**logout_user**()[source]

Logs a user out. (You do not need to pass the actual user.) This will also clean up the remember me cookie if it exists.

`from sqlalchemy.sql.expression import null`	
`from sqlalchemy.sql.functions import func`	
`from app import db, login_manager`	

`from app.base import blueprint`	
`from app.base.forms import LoginForm, CreateAccountForm`	
`from app.base.util import verify_pass`	Verify a stored password against one provided by the user.
`from app.base.models import Ratings, SubTasks, User, Tasks, Login`	

`from copy import deepcopy`	Deepcopy operation on arbitrary Python objects.

`from flask import jsonify`	
`import smtplib`	SMTP/ESMTP client class
`from email.mime.multipart import MIMEMultipart`	Base class for MIME Multipart/*type messages.
`from email.mime.text import MIMEText`	Class for generating text/*type MIME documents.

`from email.mime.base import MIMEBase`	Base class for MIME Specializations.
`from email import encoders`	Encoding and related functions.

```
@blueprint.route('/')
def route_default():
    return redirect(url_for('base_blueprint.login'))
```

By default redirected to the login page.

Login and Registration.

```python
## Login & Registration
@blueprint.route('/login', methods=['GET', 'POST'])
def login():
    print("login function called")
    # session.permanent = False
    login_form = LoginForm(request.form)
    if 'login' in request.form:
        # read form data
        username = request.form['username']
        password = request.form['password']
        # Locate user
        # user = User.query.filter_by(username=username).first()
        user = User.query.filter(func.lower(User.username) == func.lower(username)).first()
        # Check the password
        if user and verify_pass( password, user.password):
            login_user(user)
            return redirect(url_for('base_blueprint.route_default'))
        # Something (user or pass) is not ok
        return render_template( 'login/login.html', msg='Wrong user or password', form=login_form)
```

```python
if not current_user.is_authenticated:
    return render_template('login/login.html', form=login_form, segment = 'index')
elif current_user.is_authenticated:
    existingUser = Login.query.filter(Login.id==session['_user_id']).first()
    if not existingUser:
        return redirect(url_for('base_blueprint.profile'))
    else:
        session['loginId'] = Login.query.filter_by(id=session['_user_id']).first().lid
        session['user_type'] = Login.query.filter_by(id=session['_user_id']).first().user_type
        session['username'] =  Login.query.filter_by(id=session['_user_id']).first().username
        session['reporting_head'] =  Login.query.filter_by(id=session['_user_id']).first().reporting_head
        if session['user_type'] == 'Admin':
            sql_select_Query = "select L.id, L.username, L.designation, L.email, L.user_status, U.username as reporting_head fro
            data = db.engine.execute(sql_select_Query)
            users = data.fetchall()
            sql_select_Query = "select * from bin order by bin_name asc"
            data = db.engine.execute(sql_select_Query)
            bins = data.fetchall()
            sql_select_Query = "Select email from Login where user_type!='Engineer'"
            data = db.engine.execute(sql_select_Query)
            head = data.fetchall()
            return render_template('/admin-index.html', users=users, bins=bins, head=head, segment = 'index')
        else:
            return redirect(url_for('home_blueprint.index'))
```

sql_select_Query = "select L.id, L.username, L.designation, L.email, L.user_status, U.username as reporting_head from Login L, User U where U.id=L.reporting_head and designation!='Admin'"

Reading form data

a) Username
b) password

Locate user

Flask-SQLAlchemy to query from a database of users

Check the password

```
if user and verify_pass( password, user.password):

        login_user(user)

        return redirect(url_for('base_blueprint.route_default'))

    # Something (user or pass) is not ok

    return render_template( 'login/login.html', msg='Wrong user or password',
form=login_form)
```

If user and verify_pass(verifying password against one provided by user)

Login_user logs a user in.You should pass the actual user object to this, return redirect to route_default url.

Else

Leaving message wrong password

In routes.py, login() reads the password from the form and takes user datas from the database, checks the user password(db) and the entered password using verify_pass() method in routes.py.
(1) if found same-> login, else incorrect password entry -> redirect to login page
(2) (a) Not authenticated -> redirect to login page
(b) Authenticated -> takes the data of existing user from db using user_id
(c) Not existing user -> new user -> profile page.
Else take the datas from the db(lid, usertype, username, reporting_head,
The Session is **the time between the client logs in to the server and logs out of the server**. The data that is required to be saved in the Session is stored in a temporary directory on the server.

creating user details after initial registration

```python
@blueprint.route('/profile', methods=['GET', 'POST'])
def profile():
    print("profile function called")
    # sendMail(request.form['username'], request.form['email'])
    userid = session['_user_id']
    existingUser = Login.query.filter(Login.id==userid).first()

    if request.method == 'GET':
        if not existingUser:
            return render_template('login/profile.html', existingUser=existingUser, segment = 'index')
        else:
            return render_template('login/profile.html', existingUser=existingUser, segment = 'index')
    else:
        # create_user_from_admin(request.form['username'], request.form['email'])
        create = create_user_from_admin(request.form)
        if create:
            return create
        new_form = deepcopy(dict(request.form))
        reportingHeadID = User.query.filter(User.email==new_form['reporting_head']).first().id
        new_form['reporting_head'] = reportingHeadID
        query = f"SELECT id FROM User where email = '{new_form['email']}'"
        data = db.engine.execute(query)
        id = data.fetchone()[0]
        new_form['id'] = id
```

```python
# new_form['id'] = session['_user_id']
# session['user_type'] = new_form['user_type']
userLogin = Login(**new_form)
db.session.add(userLogin)
db.session.commit()
# existingUser = Login.query.filter(Login.id==userid).first()
# str(Login.query.filter(Login.id==userid).first().user_type)
sql_select_Query = "select L.id, L.username, L.designation, L.email, L.user_status, U.username as reporting_head from Login L, User
data = db.engine.execute(sql_select_Query)
users = data.fetchall()
# return render_template('login/profile.html', existingUser=existingUser, segment = 'index')
usersList = [
{'id': i['id'], 'designation': i['designation'], 'reporting_head': i['reporting_head'], 'username': i['username'],
 'user_status': i['user_status'], 'email': i['email']}
for i in users
]
return json.dumps(usersList)
```

```
select L.id, L.username, L.designation, L.email, L.user_status, U.username
as reporting_head from Login L, User U where U.id=L.reporting_head and
designation!='Admin
```

Taking existingUser from db with userid.

1)If request.method == GET:

a) If not an existing user return to profile page

2)Else Calls create_user_from_admin(request.form)

In create_user_form_admin() function,

Getting username and email

```python
def create_user_from_admin(form):
    username  = form['username']
    email     = form['email'   ]
    user = User.query.filter_by(username=username).first()
    if user:
        # return render_template( 'login/register.html', msg='Username already registered')
        return "Username already registered"
    user = User.query.filter_by(email=email).first()
    if user:
        # return render_template( 'login/register.html', msg='Email already registered')
        return "Email already registered"

    # else we can create the user
    Dict = {'username': username, 'email': email, 'password': 'Sequoia123$'}
    user = User(**Dict)
    db.session.add(user)
    db.session.commit()
    return
```

Getting the user from db using username.
If existing user return username already registered.
Getting the user from db using email.
If existing user return email already registered.
Else create the user.
Entering the username, password, email as dict.
db.session.add(user)-> place an object in the session.
db.session.commit() -> flush pending changes and commit the current transaction.
New_form keeps a deep_Copy of form for request.
Taking reporting_head from database.
Getting id using fetchone.This method returns one record as a tuple, If there are no more records then it returns `None`.
Add userLogin to db.
`fetchall()` method of cursor object to fetch the records. json.dumps() function converts a Python object into a json string of users_list.

Create_user

```python
@blueprint.route('/create_user', methods=['GET', 'POST'])
def create_user():
    login_form = LoginForm(request.form)
    create_account_form = CreateAccountForm(request.form)
    if 'register' in request.form:
        username  = request.form['username']
        email     = request.form['email'   ]
        user = User.query.filter_by(username=username).first()
        if user:
            return render_template( 'login/register.html', msg='Username already registered', form=create_account_form)
        user = User.query.filter_by(email=email).first()
        if user:
            return render_template( 'login/register.html', msg='Email already registered', form=create_account_form)
        # else we can create the user
        user = User(**request.form)
        db.session.add(user)
        db.session.commit()
        admin_profile(username, email)
        return render_template( 'login/register.html', msg='User created please <a href="/login">login</a>', form=create_account_form)
    else:
        return render_template( 'login/register.html', form=create_account_form, segment = 'index')
```

With the datas from login form.
Calls CreateAccountForm
Takes the username, email, password.
For ‘register’ in requestform
Getting the user from db using username.
If existing user return username already registered.
Getting the user from db using email.
If existing user return email already registered.
Else create the user.
Calls admin_profile
Insert into login table.
After insertion giving msg user created successfully.

Logout

```python
@blueprint.route('/logout')
def logout():
    logout_user()
    session.clear()
    return redirect(url_for('base_blueprint.login'))
```

flask_login.**logout_user**()
Logs a user out. (You do not need to pass the actual user.) This will also clean up the remember me cookie if it exists.
Clears the session.
Redirect to login page.

Shutdown

```python
@blueprint.route('/shutdown')
def shutdown():
    func = request.environ.get('werkzeug.server.shutdown')
    if func is None:
        raise RuntimeError('Not running with the Werkzeug Server')
    func()
    return 'Server shutting down...'
```

`Stop the flask web server, from the context of the flask app.`
shutdown the flask service

```python
## Errors

@login_manager.unauthorized_handler
def unauthorized_handler():
    return render_template('errors/403.html'), 403

@blueprint.errorhandler(403)
def access_forbidden(error):
    return render_template('errors/403.html'), 403

@blueprint.errorhandler(404)
def not_found_error(error):
    return render_template('errors/404.html'), 404

@blueprint.errorhandler(500)
def internal_error(error):
    return render_template('errors/500.html'), 500
```

```python
@blueprint.route('/new_task', methods=['GET', 'POST'])
def new_task():
    print("new_task function is called")
    if request.method == 'POST':
        # taskId = request.args.get('taskId', '')
        today = datetime.date.today()
        # if sub!=0:
        #     cursor.execute("UPDATE tasks set sub={} where tid={}".format(sub,sub))
        # if not due_date :
        #     due_date = today
        taskStatus = request.form['status']
        ETC = request.form['etc']
        hrs = int(ETC.split(":")[0])
        mts = int(ETC.split(":")[1])
        startDate = datetime.datetime.now().strftime("%Y-%m-%d")
        startDate = datetime.datetime.strptime(startDate, '%Y-%m-%d')
        due_date = datetime.datetime.strptime(request.form['duedate'], '%Y-%m-%d')
        startTime = datetime.datetime.now()
        if taskStatus == "InProgress" :
            # startDate = datetime.datetime.now().strftime("%Y-%m-%d")
            # startTime = datetime.datetime.now()
            endTime = datetime.datetime.now() + datetime.timedelta(hours=hrs, minutes=mts)
            endDate = endTime.strftime("%Y-%m-%d")
            endDate = datetime.datetime.strptime(endDate, '%Y-%m-%d')
```

The request method is POST.