**Flask Lessons**

1. **Python Virtual Environments**

To address the issue of maintaining different versions of packages for different applications, Python uses the concept of **virtual environments**. A virtual environment is a complete copy of the Python interpreter. When you install packages in a virtual environment, the system-wide Python interpreter is not affected, only the copy is. So, the solution to have complete freedom to install any versions of your packages for each application is to use a different virtual environment for each application. Virtual environments have the added benefit that they are owned by the user who creates them, so they do not require an administrator account.

1. **Steps to create a flask project:**

pip install virtualenv

mkdir flaskdemo

cd flaskdemo

python -m venv demoenv

source demoenv/bin/activate OR demoenv/Scripts/activate

pip install flask

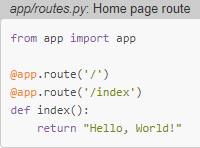
mkdir app



The script above simply creates the application object as an instance of class Flask imported from the flask package. The \_\_name\_\_ variable passed to the Flask class is a Python predefined variable, which is set to the name of the module in which it is used. Flask uses the location of the module passed here as a starting point when it needs to load associated resources such as template files. The app variable is defined as an instance of class Flask in the \_\_init\_\_.py script, which makes it a member of the app package.

1. **Creating routes in Flask**

In Flask, handlers for the application routes are written as Python functions, called view **functions.** View functions are mapped to one or more route URLs so that Flask knows what logic to execute when a client requests a given URL.

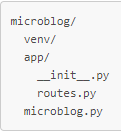


1. **Complete the application**

To complete the application, you need to have a Python script at the top-level that defines the Flask application instance. Let's call this script microblog.py, and define it as a single line that imports the application instance:



Flask application instance is called app and is a member of the app package. Project hierarchy:



1. **To run flask application**:

(demoenv) $ export FLASK\_APP=microblog.py OR set FLASK\_APP=microblog.py

(demoenv) $ flask run

(demoenv) $ set FLASK\_DEBUG=1 #for debug mode

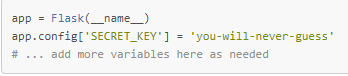
(demoenv) $ flask run #for debug mode

To avoid setting environment variable FLASK\_APP everytime, pip install python-dotenv write the environment variable name and value in a **.flaskenv** file in the top-level directory of the project:



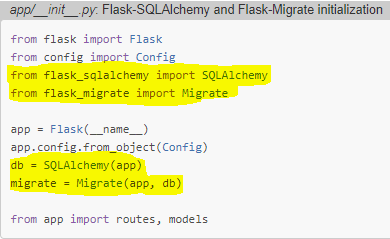
1. **Put configuration in a different file.**

There are several formats for the application to specify configuration options. The most basic solution is to define your variables as keys in app.config, which uses a dictionary style to work with variables.



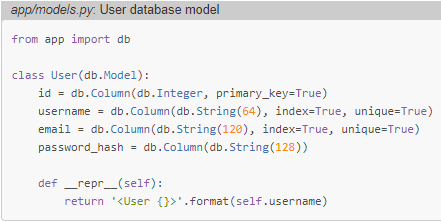
1. **Relational database configuration**

[Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy), an extension that provides a Flask-friendly wrapper to the popular SQLAlchemy package which is a ORM and [Flask-Migrate](https://github.com/miguelgrinberg/flask-migrate) extension is a Flask wrapper for [Alembic](https://bitbucket.org/zzzeek/alembic), a database migration framework for SQLAlchemy. Relational databases are centered around structured data, so when the structure changes the data that is already in the database needs to be **migrated** to the modified structure.



1. **Database models**

The data that will be stored in the database will be represented by a collection of classes, usually called **Database Models**. To create a table structure, write following lines:



The \_\_repr\_\_ method tells Python how to print objects of this class, which is going to be useful for debugging.

1. **What is a migration repository? Why is it needed?**

As the application continues to grow, there is going to be a need to change that structure, very likely to add new things, but sometimes also to modify or remove items. Alembic (the migration framework used by Flask-Migrate) will make these schema changes in a way that does not require the database to be recreated from scratch. To accomplish this seemingly difficult task, Alembic maintains a **migration repository**, which is a directory in which it stores its migration scripts. Each time a change is made to the database schema, a migration script is added to the repository with the details of the change. To apply the migrations to a database, these migration scripts are executed in the sequence they were created.

1. **Creating a migration repository and migration scripts**

Command: flask db init

flask command relies on the FLASK\_APP environment variable to know where the Flask application lives. To generate a migration automatically, Alembic compares the database schema as defined by the database models, against the actual database schema currently used in the database. It then populates the migration script with the changes necessary to make the database schema match the application models. Migration script has two functions called upgrade() and downgrade(). The upgrade() function applies the migration, and the downgrade() function removes it. This allows Alembic to migrate the database to any point in the history, even to older versions, by using the downgrade path.

Command to create migration script: **flask db migrate -m “any suitable message for changes”**

Command to apply the changes to the database: **flask db upgrade**

Command to undo the last changes to the database: **flask db downgrade**

If models in your application are modified, then a new migration script (flask db migrate) is generated. Apply the changes to development database (flask db upgrade). Add the migration script to source control and commit it.

When new version of the application is to be released to the production server, all you need to do is grab the updated version of your application, which will include the new migration script, and run flask db upgrade. Alembic will detect that the production database is not updated to the latest revision of the schema and run all the new migration scripts that were created after the previous release. flask db downgrade command undoes the last migration.

1. **Creating a one to many foreign key relation**

Create another table post to show one-to-many relation. The field timestamp has utcnow function. The function is directly passed. This ensures that there are uniform timestamps regardless of where the users are located. These timestamps will be converted to the user's local time when they are displayed.

The user\_id field was initialized as a foreign key to user.id, which means that it references an id value from the users table. The user part is the name of the database table for the model. In a db.relationship() call, the model is referenced by the model class, which typically starts with an uppercase character, while in other cases such as this db.ForeignKey() declaration, a model is given by its database table name, for which SQLAlchemy automatically uses lowercase characters and, for multi-word model names, snake case.

 if a user is stored in u, the expression u.posts will run a database query that returns all the posts written by that user. The first argument to db.relationship is the model class that represents the "many" side of the relationship.

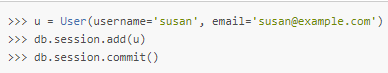
The backref argument defines the name of a field that will be added to the objects of the "many" class that points back at the "one" object. This will add a post.author expression that will return the user given a post.



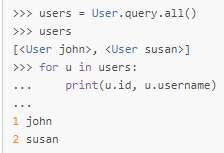
An author of a post using the **author** virtual field instead of having to deal with user IDs

1. **Interacting with database in shell**

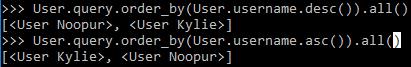
Changes to a database are done in the context of a session, which can be accessed as db.session. Multiple changes can be accumulated in a session and once all the changes have been registered, issue a single db.session.commit(), which writes all the changes atomically. If at any time while working on a session there is an error, a call to db.session.rollback() will abort the session and remove any changes stored in it. The important thing to remember is that changes are only written to the database when db.session.commit() is called.



The database can answer a query that returns all the users:



User.query.get(1) -> fetch by id



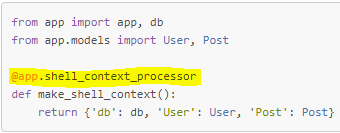
To delete records:



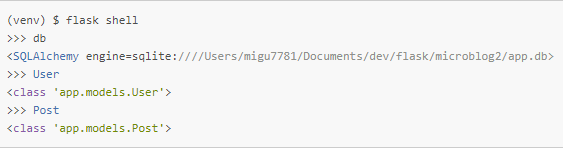
1. **Flask shell command**

The purpose of **flask shell** command is to start a Python interpreter in the context of the application. With a regular interpreter session, the app symbol is not known unless it is explicitly imported, but when using flask shell, the command pre-imports the application instance. The nice thing about flask shell is not that it pre-imports app, but that we can configure a "shell context", which is a list of other symbols to pre-import.

The following function in microblog.py creates a shell context that adds the database instance and models to the shell session:



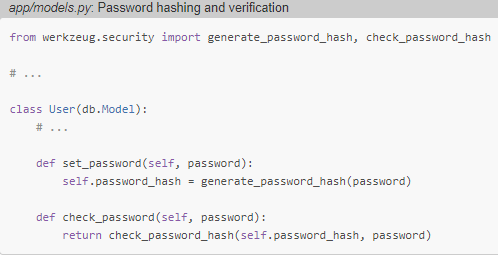
The app.shell\_context\_processor decorator registers the function as a shell context function. When the flask shell command runs, it will invoke this function and register the items returned by it in the shell session. The reason the function returns a **dictionary** and not a list is that for **each item you must also provide a name under which it will be referenced in the shell, which is given by the dictionary keys**. Adding the shell context processor function we can work with database entities without having to import them.



If you try the above and get NameError exceptions when you access db, User and Post, then the make\_shell\_context() function is not being registered with Flask. The most likely cause of this is that you have not set FLASK\_APP=microblog.py in the environment.

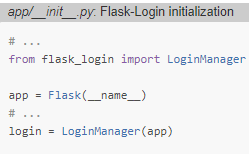
1. **Hashing in Flask (Login functionality)**

To secure login, password is not stored. Instead hash of the password is. It is a unique string generated. Flask has a core dependency called werkzeug which provides hash generation as well as checking hash.



1. **Login**

[**Flask-Login**](https://flask-login.readthedocs.io/) extension manages the user logged-in state, so that for example users can log in to the application and then navigate to different pages while the application "remembers" that the user is logged in. It also provides the "remember me" functionality that allows users to remain logged in even after closing the browser window.



1. **Preparing User model for Flask-login**

The **Flask-Login** extension works with the application's user model and expects following properties and methods to be implemented in it.

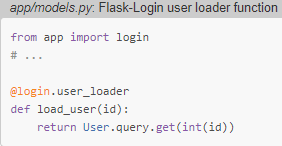
* is\_authenticated: a property that is True if the user has valid credentials or False otherwise.
* is\_active: a property that is True if the user's account is active or False otherwise.
* is\_anonymous: a property that is False for regular users, and True for a special, anonymous user.
* get\_id(): a method that returns a unique identifier for the user as a string (unicode, if using Python 2).

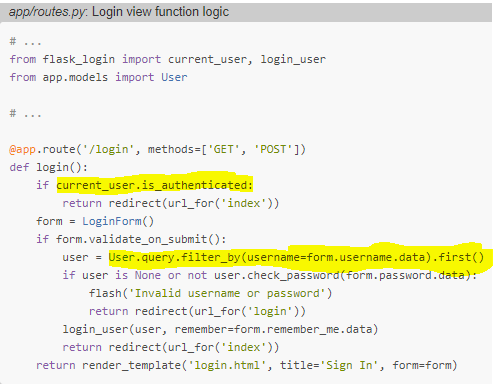
Implementing these four is easy, but since the implementations are generic, Flask-Login provides a **mixin** class called UserMixin that includes generic implementations that are appropriate for most user model classes.



1. **Loading User in session**

Flask-Login keeps track of the logged in user by storing its unique identifier in Flask's user session, a storage space assigned to each user who connects to the application. Each time the logged-in user navigates to a new page, Flask-Login retrieves the ID of the user from the session, and then loads that user into memory. The extension expects that the application will configure a user loader function, that can be called to load a user given the ID. The user loader is registered with Flask-Login with the **@login.user\_loader** decorator. The id that Flask-Login passes to the function as an argument is going to be a string, so databases that use numeric IDs need to convert the string to integer.

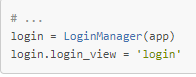




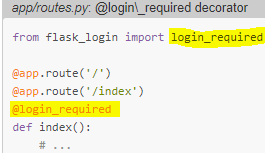
The result of filter\_by() is a query that only includes the objects that have a matching username. For one or zero results, first() is used which will return the user object if it exists, or None if it does not. login\_user() function will register the user as logged in, so that means that any future pages the user navigates to will have the current\_user variable set to that user.

1. **Require login**

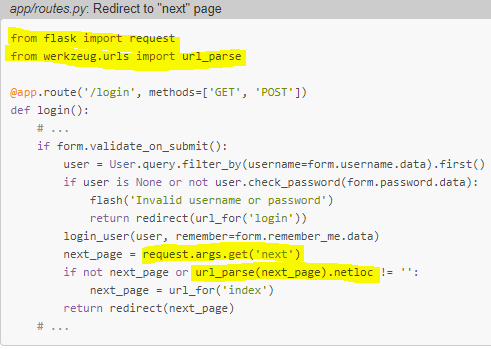
If a user who is not logged in tries to view a protected page, Flask-Login will automatically redirect the user to the login form, and only redirect back to the page the user wanted to view after the login process is complete. Add in app/init.py:

login is the name of view function that handles login

@login\_required decorator is added to a view function below the @app.route decorators from Flask, the function becomes protected and will not allow access to users that are not authenticated.



The @login\_required decorator redirects to the login page and includes some more info so that after login the user could go to the requested page. If the user navigates to /index, for example, the @login\_required decorator will intercept the request and respond with a redirect to /login, but it will add a query string argument to this URL, making the complete redirect URL /login?next=/index*.*



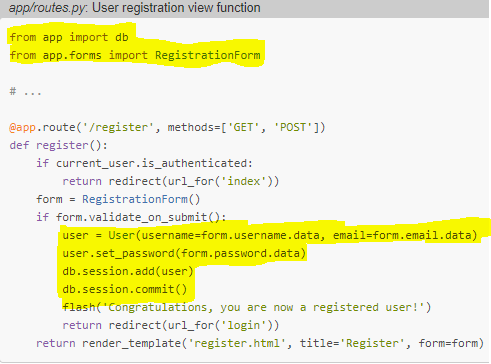
**request** variable contains all the information that the client sent with the request.**request.args** attribute exposes the contents of the query string in a friendly dictionary format. General structure of a URL: scheme://netloc/path;parameters?query#fragment

.netloc returns empty for a relative url. An attacker could insert a URL to a malicious site in the next argument.

1. **Registration**

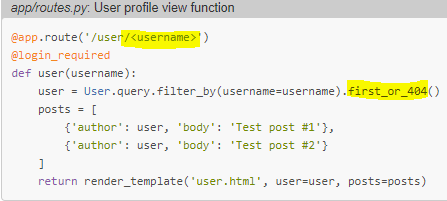
Create a new class for Registration. Two methods in this class called validate\_username() and validate\_email(). When you add any methods that match the pattern **validate\_<field\_name>**, WTForms takes those as custom validators and invokes them in addition to the stock validators. In this case I want to make sure that the username and email address entered by the user are not already in the database, so these two methods issue database queries expecting there will be no results. In the event a result exists, a validation error is triggered by raising ValidationError.



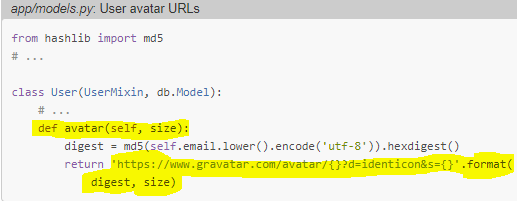


1. **User Profiles and avatars**

If the client browser requests URL /user/susan, the view function is going to be called with the argument username set to 'susan'. first\_or\_404(), which works exactly like first() when there are results, but in the case that there are no results automatically sends a [404 error](https://en.wikipedia.org/wiki/HTTP_404) back to the client.

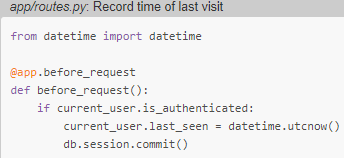


To create an avatar for user, use gravatar.com. The url takes a hash generated from email id, ‘d’ parameter for what type of icon and ‘s’ is for size.



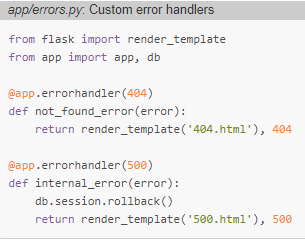
1. **Adding last seen and about me fields**

The **@before\_request** decorator from Flask registers the decorated function to be executed right before the view function. checks if the current\_user is logged in, and in that case sets the last\_seen field to the current time. A server application needs to work in consistent time units, and the standard practice is to use the UTC time zone. There is no db.session.add() before the commit, consider that when you reference current\_user, Flask-Login will invoke the user loader callback function, which will run a database query that will put the target user in the database session. So, you can add the user again in this function, but it is not necessary because it is already there.



1. **Render pages for Errors**

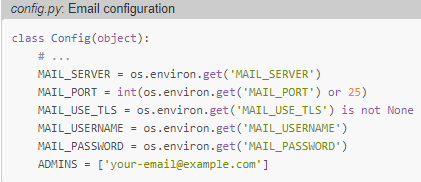
To declare a custom error handler, the **@errorhandler**decorator is used.



The second value is the error code, while in view functions default value is 200.

1. **Sending mail**

The configuration variables for email include the server and port, a boolean flag to enable encrypted connections, and optional username and password. The five configuration variables are sourced from their environment variable counterparts. If the email server is not set in the environment, then I will use that as a sign that emailing errors needs to be disabled. The email server port can also be given in an environment variable, but if not set, the standard port 25 is used. Email server credentials are by default not used but can be provided if needed. The ADMINS configuration variable is a list of the email addresses that will receive mails.



1. **Adding Logger**

Flask uses Python's **logging** package to write its logs, and this package already can send logs by email. To send email, add a [SMTPHandler](https://docs.python.org/3.6/library/logging.handlers.html" \l "smtphandler)instance to the Flask logger object, which is app.logger.



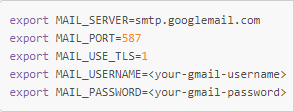
This code enables the email logger when the application is running without debug mode, which is indicated by app.debug being True, and also when the email server exists in the configuration. The code above creates a SMTPHandler instance, sets its level so that it only reports errors and not warnings, informational or debugging messages, and finally attaches it to the app.logger object from Flask.

There are two approaches to test this feature:

1. **Use** the SMTP debugging server from Python. This is a fake email server that accepts emails, but instead of sending them, it prints them to the console. Open a second terminal, and type: **python -m smtpd -n -c DebuggingServer localhost:8025**

Leave the debugging SMTP server running and go back to your first terminal and set export MAIL\_SERVER=localhost and MAIL\_PORT=8025 in the environment. Set FLASK\_DEBUG variable to 0 or not set at all, since the application will not send emails in debug mode.

1. **Configure** a real email server



MAIL\_SERVER=smtp.gmail.com #kindly correct

Enable less secure apps and IMAP

Check following details also in your code:

smtp.gmail.com

Requires SSL: Yes

Requires TLS: Yes (if available)

Requires Authentication: Yes

Port for SSL: 465

Port for TLS/STARTTLS: 587

1. **Logging in a File**

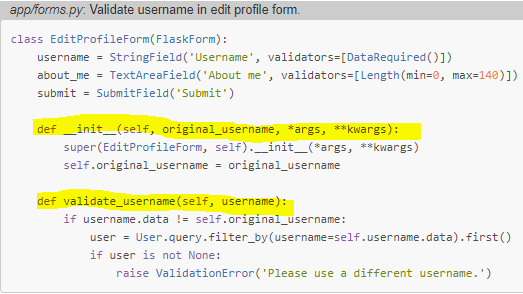


The RotatingFileHandler class is nice because it rotates the logs, ensuring that the log files do not grow too large when the application runs for a long time. In this case I'm limiting the size of the log file to 10KB, and I'm keeping the last ten log files as backup. logging.Formatter class provides custom formatting for the log messages. Logging categories are DEBUG, INFO, WARNING, ERROR and CRITICAL in increasing order of severity.

1. **Handling internal exceptions**

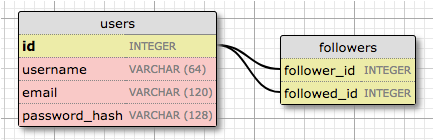
Changes in edit profile form to check that the username doesn’t already exist in the db.

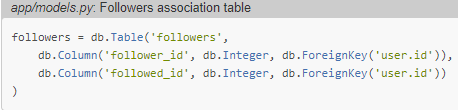
A custom validation method, but there is an overloaded constructor that accepts the original username as an argument. This username is saved as an instance variable, and checked in the validate\_username() method. If the username entered in the form is the same as the original username, then there is no reason to check the database for duplicates.



1. **Representing followers**

The representation of a many-to-many relationship requires the use of an auxiliary table called an ***association table***. A relationship in which instances of a class are linked to other instances of the same class is called a *self-referential relationship.*





Direct translation of the association table from the diagram above. Not declaring this table as a model, like for the users and posts tables. Since this is an auxiliary table that has no data other than the foreign keys, it is created without an associated model class.

Declare the many-to-many relationship in the users table:



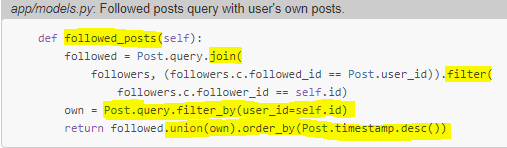
* 'User' is the right-side entity of the relationship (the left side entity is the parent class). Since this is a self-referential relationship, I must use the same class on both sides.
* secondary configures the association table that is used for this relationship, which I defined right above this class.
* primaryjoin indicates the condition that links the left side entity (the follower user) with the association table. The join condition for the left side of the relationship is the user ID matching the follower\_id field of the association table. The followers.c.follower\_id expression references the follower\_id column of the association table.
* secondaryjoin indicates the condition that links the right side entity (the followed user) with the association table. This condition is similar to the one for primaryjoin, with the only difference that now I'm using followed\_id, which is the other foreign key in the association table.
* backref defines how this relationship will be accessed from the right side entity. From the left side, the relationship is named followed, so from the right side I am going to use the name followers to represent all the left side users that are linked to the target user in the right side. The additional lazy argument indicates the execution mode for this query. A mode of dynamic sets up the query to not run until specifically requested, which is also how I set up the posts one-to-many relationship.
* lazy is like the parameter of the same name in the backref, but this one applies to the left side query instead of the right side.

1. **Adding and removing followers**

The follow() and unfollow() methods use the append() and remove() to follow and unfollow respectively. The is\_following() method issues a query on the followed relationship to check if a link between two users already exists.



1. **Obtaining posts from followers**



Post is joined with followers table. The join condition is that followed\_id field of the followers table must be equal to the user\_id of the posts table. The results are filtered by follower\_id that it must be same as current user. Then it is union with the user’s self-posts. Then it is ordered by decreasing timestamp.

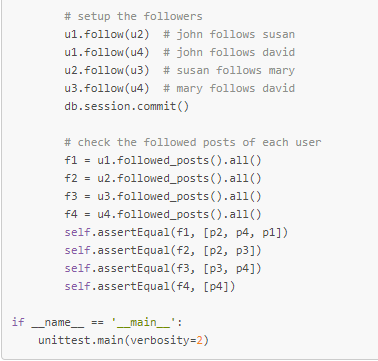
1. **Unit testing**

Python built in package for unit testing is unittest.

The setUp() and tearDown() methods are special methods that the unit testing framework executes before and after each test respectively. In setUp(), to prevent the unit tests from using the regular database used for development, changing the application configuration to sqlite:// , SQLAlchemy uses an in-memory SQLite database during the tests. The db.create\_all() call creates all the database tables.







Command to run test suite: python tests.py

1. **Pagination**

To paginate the results, replace the all() with paginate() query method. The paginate method can be called on any query object from Flask-SQLAlchemy. It takes three arguments:

* the page number, starting from 1
* the number of items per page
* an error flag. If True, when an out of range page is requested a 404 error will be automatically returned to the client. If False, an empty list will be returned for out of range pages

user.followed\_posts().paginate(1,20, False).items

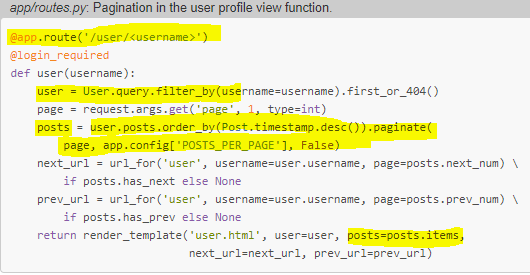
The return value from paginate is an object of a Pagination class from Flask-SQLAlchemy. The items attribute of this object contains the list of items in the requested page. Few other attributes of Pagination object:

* has\_next: True if there is at least one more page after the current one
* has\_prev: True if there is at least one more page before the current one
* next\_num: page number for the next page
* prev\_num: page number for the previous page



The **next\_url** and **prev\_url** in these two view functions are going to be set to a URL returned by url\_for() only if there is a page in that direction. If the current page is at one of the ends of the collection of posts, then the **has\_next** or **has\_prev** attributes of the Pagination object will be False, and in that case the link in that direction will be set to *None*.

We can add any keyword arguments to **url\_for()** function, and if the names of those arguments are not referenced in the URL directly, then Flask will include them in the URL as query arguments.



To get the list of posts from the user, ***user.posts relationship is a query*** that is already set up by SQLAlchemy as a result of the db.relationship() definition in the User model. I take this query and add a order\_by() clause so that I get the newest posts first, and then do the pagination exactly like I did for the posts in the index and explore pages.

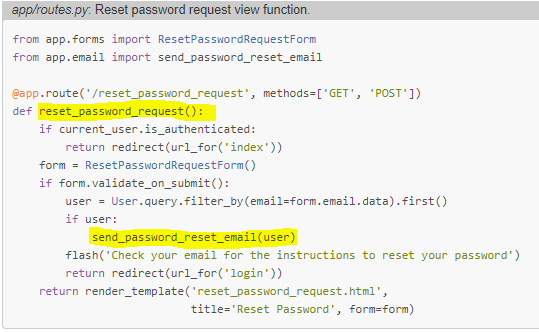
1. **Reset password and generate token**

Pip install flask-mail

Pip install pyjwt

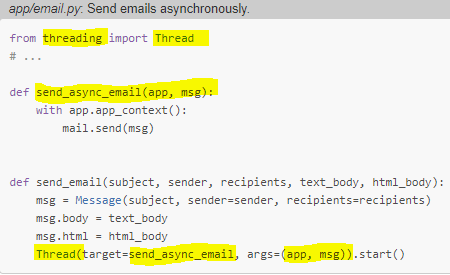
Pyjwt to generate JSON web tokens to send to reset link

Create view function and html templates for reset password:



1. **Sending mails asynchronously**

The mail sending takes a lot of time, so we do it asynchronously:



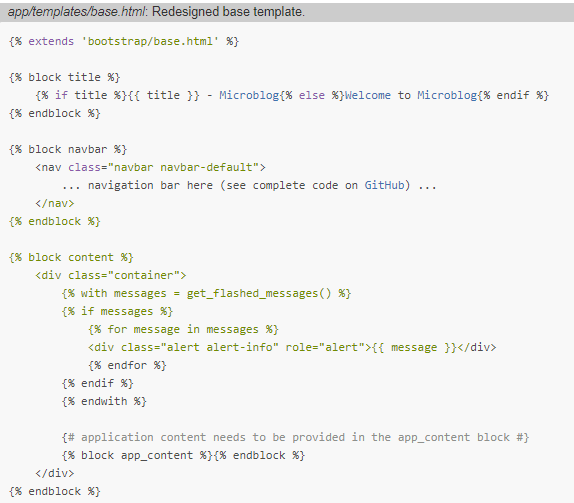
The send\_async\_email function now runs in a background thread, invoked via the Thread()class in the last line of send\_email(). With this change, the sending of the email will run in the thread, and when the process completes the thread will end and clean itself up.

Flask uses **contexts** to avoid having to pass arguments across functions. There are two types of contexts, the *application context* and the *request context*. In most cases, these contexts are automatically managed by the framework, but when the application starts custom threads, contexts for those threads may need to be manually created.

There are many extensions that require an application context to be in place to work, because that allows them to find the Flask application instance without it being passed as an argument. The reason many extensions need to know the application instance is because they have their configuration stored in the app.config object. This is exactly the situation with Flask-Mail. The mail.send() method needs to access the configuration values for the email server, and that can only be done by knowing what the application is. The application context that is created with the with app.app\_context() call makes the application instance accessible via the current\_app variable from Flask.

1. **Flask-Bootstrap**

Use extension flask-bootstrap for using bootstrap in flask. It provides a base template over which we create customized templates.

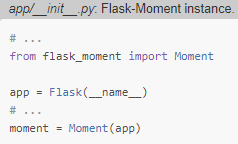


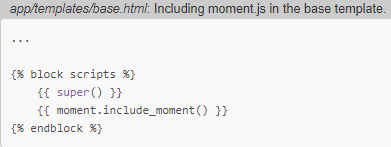
The bootstrap/base.html template provides the basic structure of the page, which includes the Bootstrap framework files. This template exports a few blocks for derived templates such as title, navbar and content.

I'm going to change my *base.html* template to derive from *bootstrap/base.html* and provide implementations for the title, navbar and content blocks. In turn, *base.html* will export its own app\_content block for its derived templates to define the page content.

1. **Timezone conversions**

Timezone stored in db must be in same timezone while when it is rendered to user, it must be in their local timezones. [Moment.js](http://momentjs.com/) is a small open-source JavaScript library that takes date and time rendering to another level, as it provides every imaginable formatting option, and then some. Flask-Moment, a small Flask extension makes it very easy to incorporate moment.js into your application. Pip install flask\_moment





Moment.js must be in every template so we include it in the base template using moment.include\_moment() method. The scripts block is another block exported by Flask-Bootstrap's base template. This is the place where JavaScript imports are to be included. This block already comes with some content defined in the base template. To add moment.js along with the content of base template and this is achieved with the super() statement, which preserves the content from the base template.

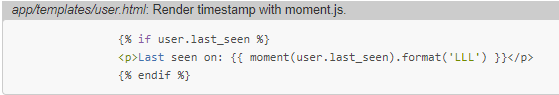
1. **Using Moment.js**

Moment.js makes a moment class available to the browser. The first step to render a timestamp is to create an object of this class, passing the desired timestamp in [ISO 8601](http://en.wikipedia.org/wiki/ISO_8601) format: **t=moment(‘2017-09-28T21:45:23Z’)**

ISO 8601 standard format for dates and times is as follows: {{ year }}-{{ month }}-{{ day }}T{{ hour }}:{{ minute }}:{{ second }}{{ timezone }}



Argument to **moment**() is now a Python datetime object and not an ISO 8601 string. The moment() call issued from a template also automatically generates the required JavaScript code to insert the rendered timestamp in the proper place of the DOM.

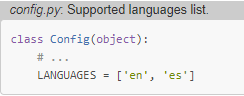




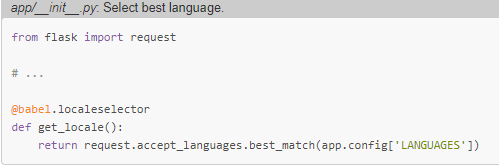
1. **Working with translations**

Pip install flask-babel

To keep track of the list of supported languages, add a configuration variable:



The Babel instance provides a localeselector decorator. The decorated function is invoked for each request to select a language translation to use for **that** **request**:



Flask’s request object provides a high-level interface to work with the [Accept-Language](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Language) header that clients send with a request. This header specifies the client language and locale preferences as a weighted list.

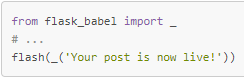


This says that Danish (da) is the preferred language (with default weight = 1.0), followed by British English (en-GB) with a 0.8 weight, and as a last option generic English (en) with a 0.7 weight.

To select the best language, you need to compare the list of languages requested by the client against the languages the application supports, and using the client provided weights, find the best language. The logic to do this is somewhat complicated, but it is all encapsulated in the best\_match() method, which takes the list of languages offered by the application as an argument and returns the best choice.

1. **Marking Texts to Translate**

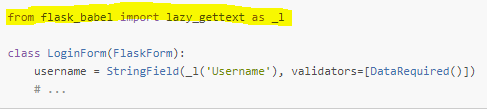
The normal workflow when making an application available in multiple languages is to mark all the texts that need translations in the source code. After the texts are marked, **Flask-Babel** will scan all the files and extract those texts into a separate translation file using the [gettext](https://www.gnu.org/software/gettext/) tool. The way texts are marked for translation is by wrapping them in a function call that as a convention is called \_(), just an underscore.



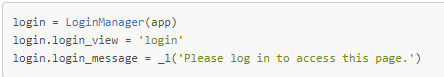
The idea is that the \_() function wraps the text in the base language (English in this case). This function will use the best language selected by the function decorated with the @localeselector function to find the correct translation for a given client. The \_() function then returns the translated text, which in this case will become the argument to flash().



Some string literals are assigned outside of a request, usually when the application is starting up, so at the time these texts are evaluated there is no way to know what language to use. An example of this is the labels associated with form fields. The only solution to handle those texts is to find a way to delay the evaluation of the string until it is used, which is going to be under an actual request. Flask-Babel provides a *lazy evaluation* version of \_() that is called lazy\_gettext():



\_l function wraps the text in a special object that triggers the translation to be performed later, when the string is used.



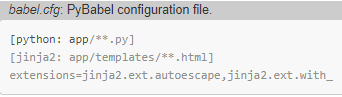
Test in templates is also marked similarly.



1. **Extracting text to translate**

After putting \_() and \_l() in the application, use pybabel command to extract the text to be translated to a .pot(portable object template) file. This is a text file that includes all the texts that were marked as needing translation. The purpose of this file is to serve as a template to create translation files for each language.

The extraction process needs a small configuration file that tells pybabel what files should be scanned for translatable texts.



To extract:



The pybabel extract command reads the configuration file given in the -F option, then scans all the code and template files in the directories that match the configured sources, starting from the directory given in the command (the current directory or. in this case). By default, pybabel will look for \_() as a text marker, but I have also used the lazy version, which I imported as \_l(), so I need to tell the tool to look for those too with the -k \_l. The -o option provides the name of the output file.

Create a translation for each language that will be supported in addition to English by command:



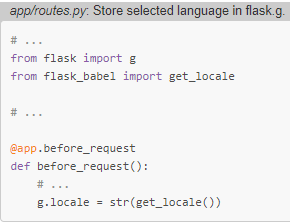
The pybabel init command takes the *messages.pot* file as input and writes a new language catalog to the directory given in the -d option for the language specified in the -l option. Install all the translations in the *app/translation’s* directory, because that is where Flask-Babel will expect translation files to be by default. The command will create an ***es***subdirectory inside this directory for the Spanish data files. There will be a new file named *app/translations/es/LC\_MESSAGES/messages.po*, that is where the translations need to be made. Repeat the above command with each of the language codes you want, so that each language gets its own repository with a *messages.po* file. This messages.po file that is created in each language repository uses a format that is the de facto standard for language translations, the format used by the [gettext](http://www.gnu.org/software/gettext/) utility.

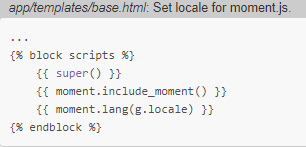
The *messages.po* file is a sort of source file for translations. When you want to start using these translated texts, this file needs to be *compiled* into a format that is efficient to be used by the application at run-time.



This operation adds a *messages.mo* file next to *messages.po* in each language repository. The *.mo* file is the file that Flask-Babel will use to load translations for the application. In case of some translations missed out, use:





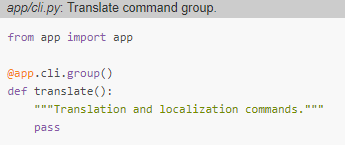
to convert timestamps also in local language

1. **Custom commands to use with flask**

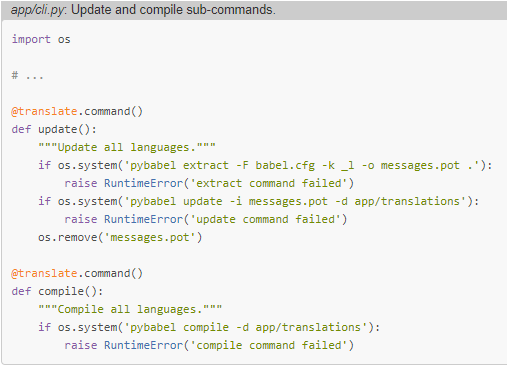
Add application-specific commands to flask

* flask translate init LANG to add a new language
* flask translate update to update all language repositories
* flask translate compile to compile all language repositories

The babel extract step is not going to be a command, because generating the *messages.pot*file is always a pre-requisite to running either the init or the update commands, so the implementation of these commands will generate the translation template file as a temporary file. Flask relies on [Click](http://click.pocoo.org/5/) for all its command-line operations. Commands like translate, which are a root for several sub-commands are created via the app.cli.group() decorator.



The name of the command comes from the name of the decorated function, and the help message comes from the docstring. Since this is a parent command that only exists to provide a base for the sub-commands, the function itself does not need to do anything.



Note how the decorator from these functions is derived from the translate parent function. This may seem confusing, since translate() is a function, but it is the standard way in which Click builds groups of commands. Same as with the translate() function, the docstrings for these functions are used as help message in the --help output.

If the command errors, then I raise a RuntimeError, which will cause the script to stop. The update() function combines the extract and update steps in the same command, and if everything is successful, it deletes the *messages.pot* file after the update is complete, since this file can be easily regenerated when needed again.



This command uses the @click.argument decorator to define the language code. Click passes the value provided in the command to the handler function as an argument, and then I incorporate the argument into the init command. To add a new language, you use:

(venv) $ flask translate init <language-code>

To update all the languages after making changes to the \_() and \_l() language markers:

(venv) $ flask translate update

To compile all languages after updating the translation files:

(venv) $ flask translate compile

1. **sfaff**