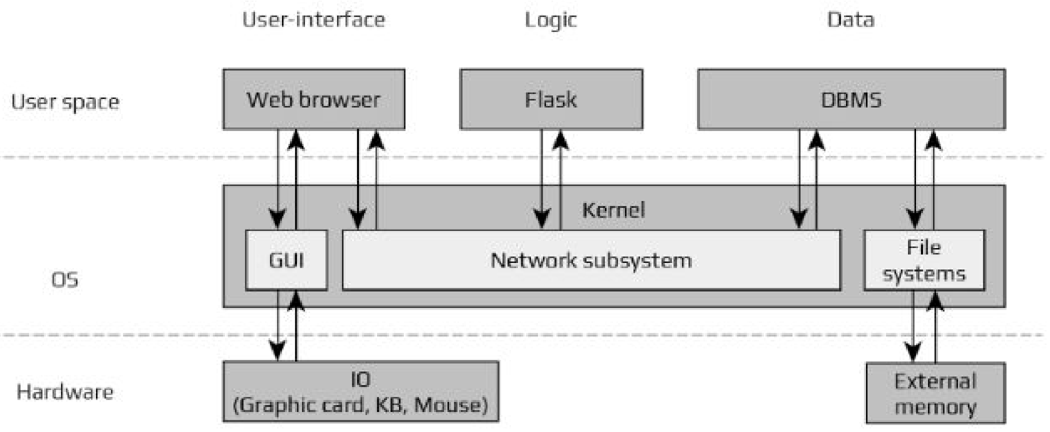
E-14a: Building Web Applications for Data Analysis

# Lab 6: Storing and Manipulating with Data



*Web browser + Flask + PostgreSQL*

## Learning Objectives

- Understand Database modelling
- Working with DDL and DML
- Create, insert into, and change DB tables
- Use Flask extensions
- Creating Web Forms
- Connect Database with Flask
- Style Web Form using CSS
- Communication between frontend and backend
- Request-response cycle with DB component included

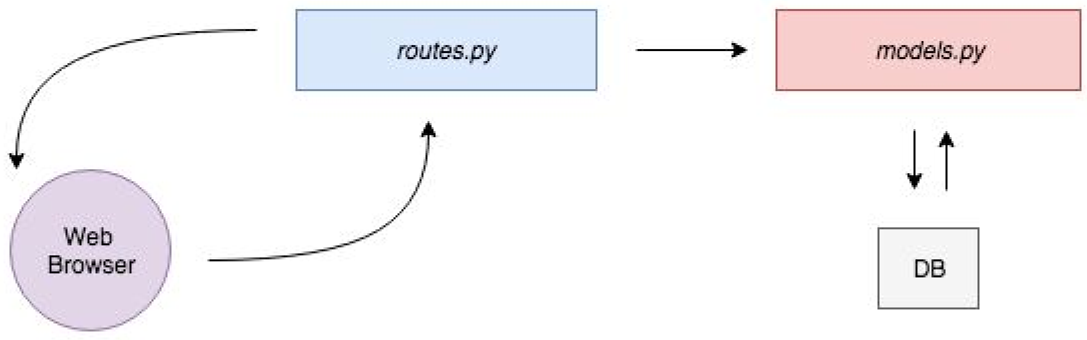
## INTRODUCTION

The databases can be separated into two big groups - those that follow the relational model, and those that do not. The latter group is often called NoSQL, indicating that they do not implement the popular relational query language SQL. While there are great database products in both groups, relational databases are a better match for applications that have structured data while NoSQL databases tend to be better for data that has a less defined structure. In this class we will be working with relational databases only[1].

## Flask and Postgres

In previous week we used techniques to build a flask app containing two static pages. In doing so, we started developing a generalizable workflow that we can use to build out the rest of the app. This workflow is summarized by the request-response cycle. When you type in a URL and press Enter, your request hits *routes.py*. In *routes.py*, the URL is mapped to a python function that fetches web template and its assets and renders it to HTML. That rendered HTML is sent back to the browser, where it displays to the user.

The problem with this app is in its content - it is still static. Modern web apps demand the use of the dynamic content creation. One side of the problem can be solved by introducing the databases. Here's how a database fits into request-response cycle.



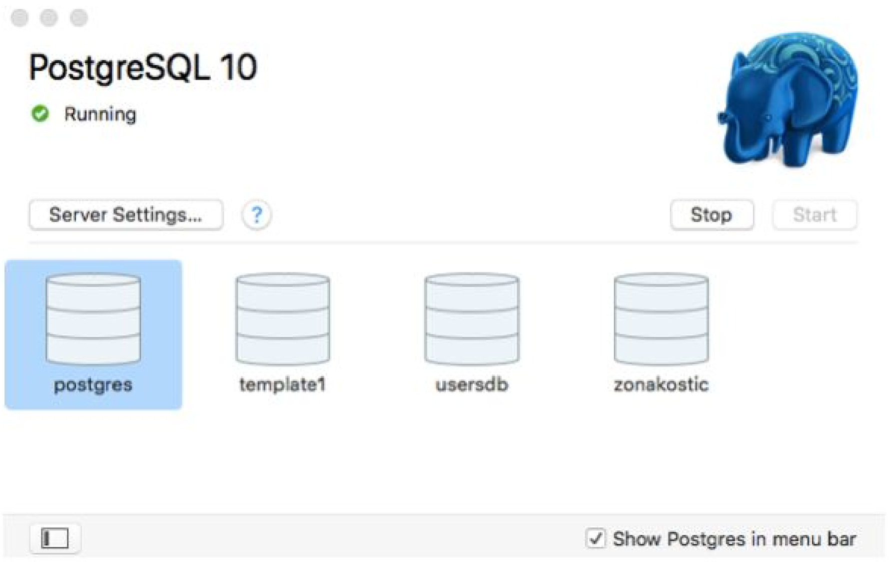
[1] Flask does not support databases natively. This way you have the freedom to choose the database that best fits your application instead of being forced to adapt to a specific one..

When a user types the address and presses Enter, the browser issues a request for that URL. The request hits *routes.py*, where the URL is matched to a python function. The python function can interact with a database. It can write information to the database, like creating a new user. It can read information from that database, like list all users age (or aggregate - return the average age value of all users in the database) . Python function can pass along data from the database to a web template, render it to HTML, and send it back to the browser.

## Create a new DB table

For creating a new database, open a Postges control window, double click to any created database and type:

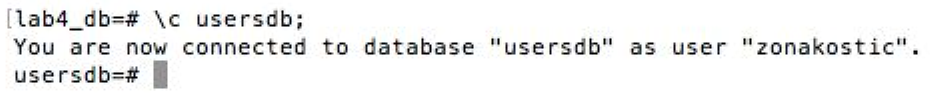
```
CREATE DATABASE usersdb;
```



Inside the “usersdb” database, we need a table where we can safely store users information. We will call this table “users". The information that we will want to store are a user's *first name*, *last name*, *email*, and *password*.

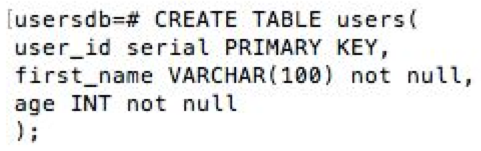
***Important: Make sure you understand that the creation of a database does not mean you are actually connected to that database!***

Let us make sure we are connected to the “usersdb” database. In the Postgres use `\c` and the name of the database.



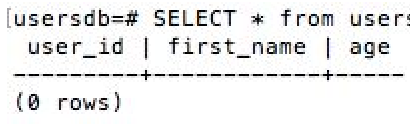
```
[lab4_db=# \c usersdb;
 You are now connected to database "usersdb" as user "zonakostic".
 usersdb=#
```

Create a new users table, using the CREATE TABLE statement (follow the terminal output for more information). The values to be created are *user_id* for the user ID. And then *first_name* and *age*. For the first name we are using the VARCHAR data type and for the age we use INT. For the *user_id* column, we're using the serial data type. This will let us generate an ID for each row that increments automatically.



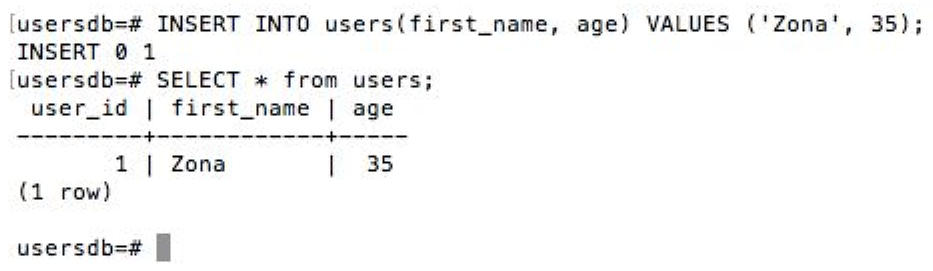
```
[usersdb=# CREATE TABLE users(
 user_id serial PRIMARY KEY,
 first_name VARCHAR(100) not null,
 age INT not null
 );
```

Using a SELECT statement you list all the table rows. As you mighs table is empty:



```
[usersdb=# SELECT * from users;
  user_id | first_name | age
 ---------+------------+-----
 (0 rows)
```

There are no rows in this table yet. Let's use an INSERT INTO statement to add a new user attributes. Now when we run a SELECT statement, we see the table has one row, has one user, and the uid field has auto-incremented.


```
[usersdb=# INSERT INTO users(first_name, age) VALUES ('Zona', 35);
 INSERT 0 1
[usersdb=# SELECT * from users;
  user_id | first_name | age
 ---------+------------+-----
        1 | Zona       |  35
 (1 row)

 usersdb=# 
```

So far, we've created a users table in the database. Next, let's connect this database to a Flask app.

**YOUR TURN**

Create a new database with name *students*. Then, create a new table *student*. Every student entity should have the following values: *user_id*, *first_name*, *last_name*, and *email*.

Email address should be a primary key.

## Connect Flask to a Database

The next step is connecting the Flask app to the *usersdb* database in order to read and write information to/from it. However Flask doesn't come with this functionality natively. Instead of packing in extra functionality like connecting to a database, Flask lets you aid it on as needed using extensions.

Welcome to the Flask extensions registry. Here you can find a list of packages that extend Flask. This list is moderated and updated on a regular basis. If you want your package to show up here, follow the guide on creating extensions.

An extension is a package that adds a specific functionality to a Flask app. There are extensions for adding in functionality for web forms, sending email, many other common tasks in web application development. To add functionality to communicate with a database, we'll use the *Flask-SQLAlchemy* extension. To use this extension, first get back into the app's isolated Python development environment. Once inside the development environment, use `pip` to install *Flask-SQLAlchemy*. So type:

```
pip install flask-sqlalchemy
```

Now that *Flask-SQLAlchemy* is installed, let's use it to connect the Flask app to the Postgres database *usersdb*. Open up *routes.py* and modify it to look like this.

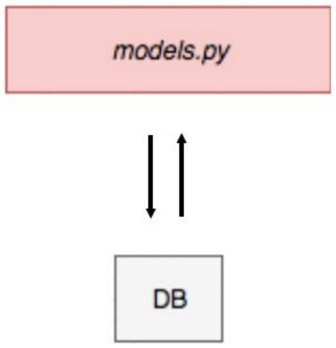
```
from flask import Flask, render_template
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://localhost/usersdb'
```

With this line we configured the Flask app to use the *usersdb* database. We specify this using a database URI. This format is the connection string that *Flask-SQLAlchemy* expects.

Check out the *Flask-SQLAlchemy* docs for more information: https://www.sqlalchemy.org/

## Create Users Models

What we have so far? First, inside the database, we created a users table to store user's information. Second, we used *Flask-SQLAlchemy* to connect Flask to this database. The last thing we need is a data structure that we can use from the app to read and write to the users table. This data structure is called a *model*. We will write this code in a new file called *models.py*.



Open file *models.py* and let us go through this code line by line:

```
from flask_sqlalchemy import SQLAlchemy
```

Import *SQLAlchemy*. Then, create a variable named *db* containing a new usable instance of the *SQLAlchemy* class.

```
db = SQLAlchemy()
```

Then, create a Python class to model the user's attributes.

```
class User(db.Model):
 __tablename__ = 'users'
user_id = db.Column(db.Integer, primary_key=True, autoincrement=True)
first_name = db.Column(db.String(100), nullable=False)
age = db.Column(db.Integer, nullable=False)
```

Lastly, to make everything work together, modify *routes.py* to look like this.

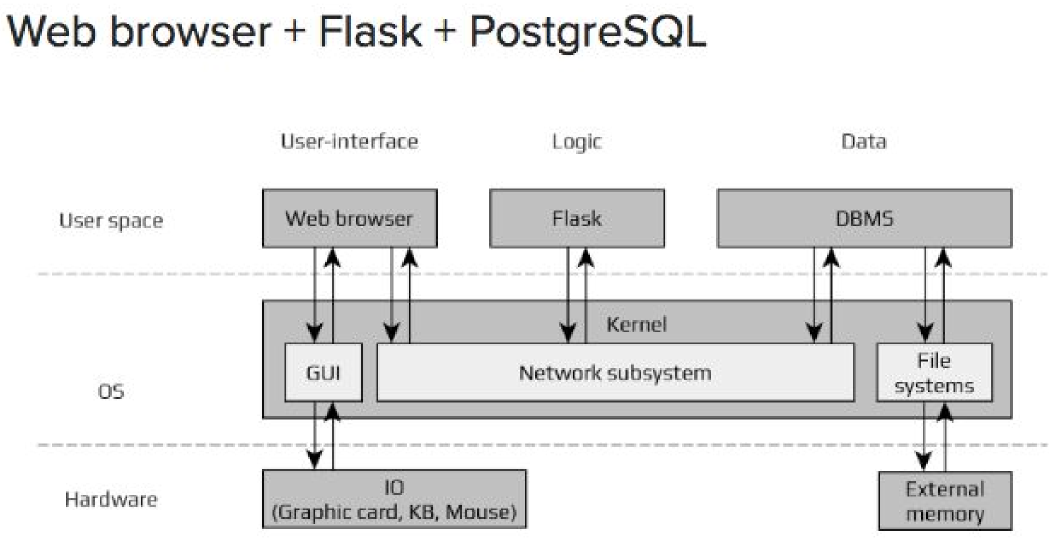
```
from models import db, User
```

First, import the db object from models. Remember, db was the new usable instance of the SQLAlchemy class. Then, use `db.init_app(app)` to initialize the Flask app for using this database setup.

```
db.init_app(app)
```

## Connect Frontend and Backend

In this section, we'll create a web form to allow users directly putting their data into the database, so that we can take in user data and save it to the database. Take a look at the picture to understand how everything will fit together. First, a user will use the form to input the data. The page will be retrieved through an HTTP GET request and will load in the browser. Second, the sign up page will have fields for a user's name, email, and password.



The user will fill out the form and press Enter. When a user presses Enter, the forms will submit to the Flask app through a POST request and will hit *routes.py*. In *routes.py*, a function will check whether the submitted data is valid (this means checking whether all fields are filled out). Finally, if any of the submitted data does not pass validation, the Form page will load again, with helpful error messages. Otherwise, if all fields are valid, we'll save the user's credentials to the database.

## Create Users Forms

In order to create the form Flask, again, uses extensions. To add functionality for web forms, we'll use *Flask-WTF*. To install *Flask-WTF*, we'll use the same sequence of commands as we did when adding *Flask-SQLAlchemy*. Use `pip` to install *Flask-WTF*:

```
pip install flask-wtf;
```

Flask-WTF

Flask-WTF offers simple integration with WTForms. This integration includes optional CSRF handling for greater security.

This is an approved extension.

| | |
|---|---|
| Author: | Anthony Ford (created by Dan Jacob) |
| PyPI Page: | Flask-WTF |
| Documentation: | Read docs @ pythonhosted.org |
| On Github: | ajford/flask-wtf |

## Creating a Web Form - backend

In order to make a web form we need to do two things. First we need to set up a backend where we define the form's fields. Second we need to set up a frontend where we display the form's fields to the user.

Let's create a new file named *forms.py* and write the form there. Then open up *forms.py* and add the following code to it:

```
from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField, SubmitField
from wtforms.validators import DataRequired
```

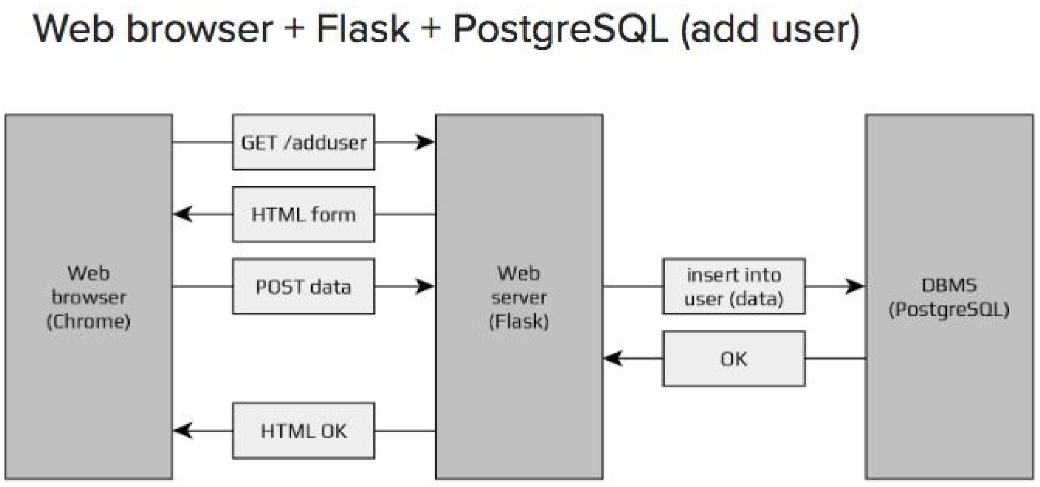
First we import a few helpful classes from Flask WTF. The base form class, a text field, a password field and a submit button. Next we create a new class named UsersForm inheriting from the base form class.  And then we create each field we want in the form.

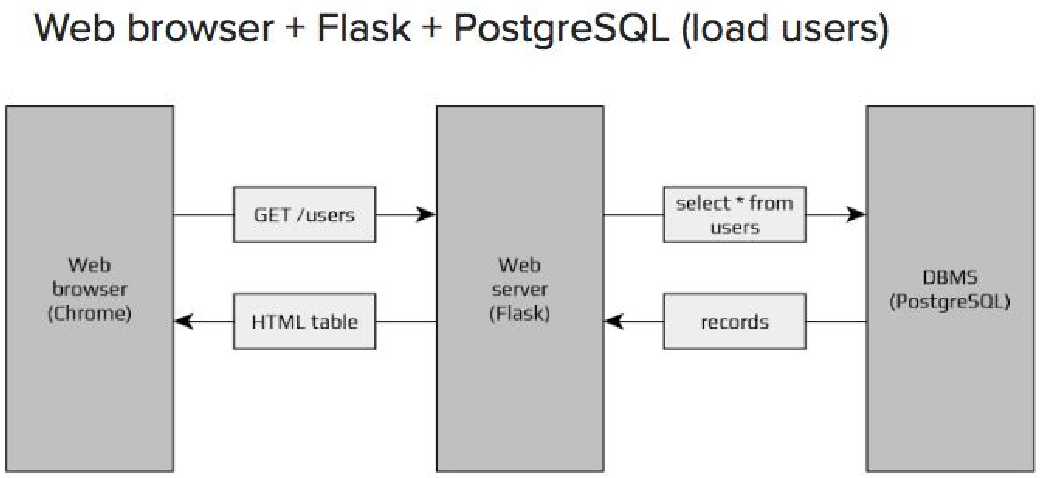
```
class UsersForm(FlaskForm):
  first_name = StringField('First Name', validators=[DataRequired()])
  age = IntegerField('Age', validators=[DataRequired()])
  submit = SubmitField('Enter')
```

When the user visits the URL with the option to add a new user, an *add-user* page containing this form will show up. This means we need to create a new URL mapping in *routes.py*. Open up the *routes.py* file and import the newly created form:

```
from forms import UsersForm
```

If you are interested in how request-response cycle looks like in both cases - load users and add users - take a look at the following 2 images.





Next, we will configure the form to protect against a security exploit called *cross site request forgery* - CSRF. Type `app.secret_key="e14a-key"`. This secret key *Flask-WTF* can use to generate secure forms.

The first portion of the *routes.py* code should look like this:

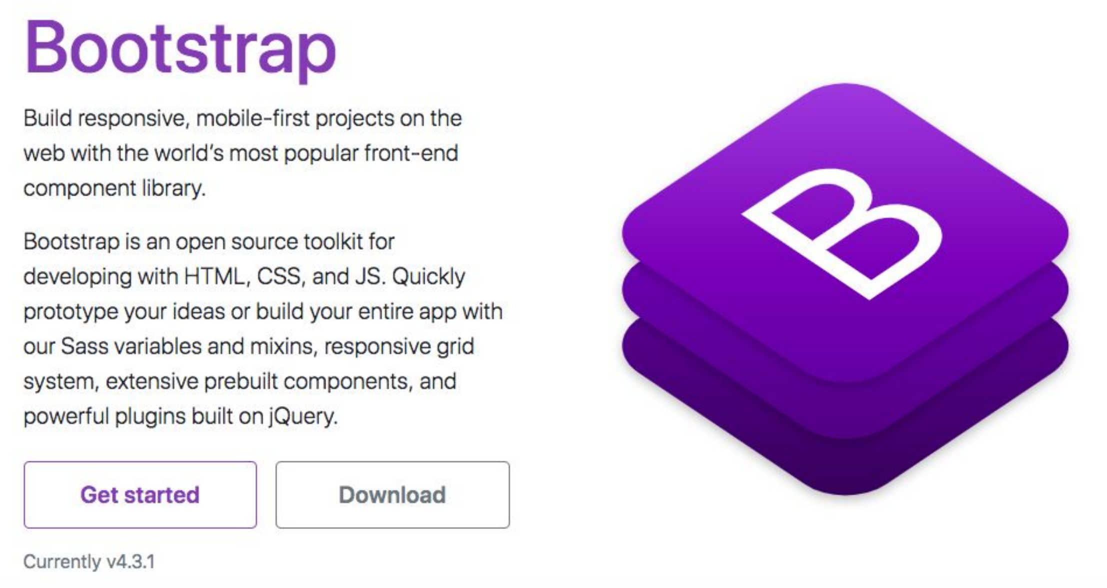
```
from flask import Flask, render_template
from models import db, User
from forms import UsersForm

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://localhost/usersdb'
db.init_app(app)

app.secret_key = "e14a-key"
```

## Creating a Web Form - frontend


# Bootstrap

Build responsive, mobile-first projects on the web with the world's most popular front-end component library.

Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery.

Currently v4.3.1

Rather than coding from scratch, frameworks enable you to utilize ready made blocks of code to help you get started. They give you a solid foundation for what a typical web project requires and usually they are also flexible enough for customization.

Bootstrap is an example of open source HTML, JS and CSS framework. It is one of the most widely used frameworks, easy to understand, and it provides a great documentation with many examples. Here is a summary of the main aspects of Bootstrap:

- Open source HTML, CSS, and JS framework
- Provides a base styling for commonly used HTML elements
- **The grid** system helps you to create multi-column and nested layouts
- Extensive list of pre-styled components
- Customizable: All CSS rules can be overridden by your own rules
- Compatible with the latest versions of all major browsers

**Web Forms**

Web forms should have different sections on the web page or different html page/s. Let us create a new template for users form named *add_user.html*. Remember, we placed templates inside the Templates folder. Open *add_user.html* and copy/paste the following code:

```
<form method="POST" action="/add-user">
        {{ form.hidden_tag() }}
        <div class="form-group">
          {{ form.first_name.label }}
          {% if form.first_name.errors %}
            {% for error in form.first_name.errors %}
              <p class="error-message">{{ error }}</p>
            {% endfor %}
          {% endif %}
          {{ form.first_name }}
        </div>
        <div class="form-group">
            {{ form.age.label }}
            {% if form.age.errors %}
                {% for error in form.age.errors %}
                <p class-"error-message">{{ error }}</p>
                {% endfor %}
            {% endif %}
            {{ form.age(class="form-control") }}
        </div>
        {{ form.submit(class="btn-primary") }}
      </form>
```

To understand the above code, we need to remind ourselves of template inheritance. Jinja2 has a template inheritance feature that specifically addresses the problem of dynamic content creation. Let us observe this portion (**bold**) that creates text field (element) for inputting user's first name:

```
      <div class="form-group">
          {{ form.first_name.label }}
          {% if form.first_name.errors %}
            {% for error in form.first_name.errors %}
              <p class="error-message">{{ error }}</p>
            {% endfor %}
          {% endif %}
          {{ form.first_name }}
        </div>
```

The text element has been defined by calling the object *form* (*UsersForm* from *forms.py*) and passing *first_name (StringField)* as a text element and **`first_name.label`** as a string for the label.

Note {{ url_for('static', filename='css/main.css') }}

We are using Flask function URL4 to generate a URL for us. URL4 is telling Flask to go to the static folder and look for the file given by the filename.

## Saving a New User

Go back to the *routes.py* and add the following:

```
@app.route('/add-user', methods=['GET', 'POST'])
def add_user():

    form = UsersForm()
    if request.method == 'GET':
        return render_template('add_user.html', form=form)
    else:
        if form.validate_on_submit():
            first_name = request.form['first_name']
            age = request.form['age']
            new_user = User(first_name=first_name, age=age)
            db.session.add(new_user)
            db.session.commit()
            return redirect(url_for('index'))
```

What the above code does? After importing the user class from *models.py* we need to create a new useful instance of the user object, and initialize it with the data from the *Add New User* form. We added `methods = ['GET', 'POST']` so we need to create if/else statement to distinguish between a get and a post request. If a form has been submitted, a post request has happened:

```
if request.method == 'GET':
      return render_template('add_user.html', form=form)
else: "Success!!!"
```

We are then, passing the values from the form fields (elements) to the variables:

```
first_name = request.form['first_name']
age = request.form['age']
```

In the next step we are calling the *Users* object and passing the previously collected field values:

```
new_user = User(first_name=first_name, age=age)
```

And then we have to insert the new user object into the users table and save it. You can do this by using `db` session add, and `db` session commit.

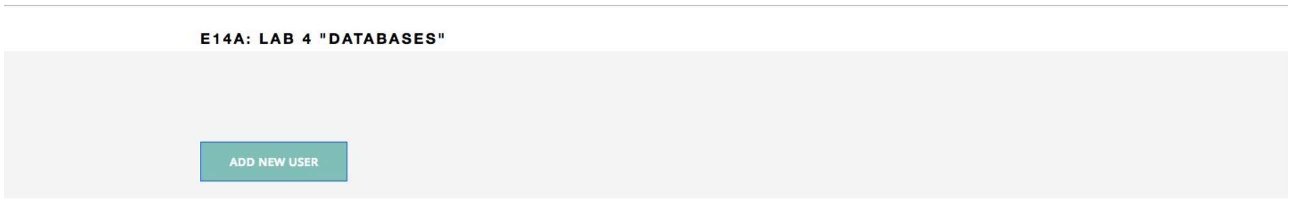
```
db.session.add(new_user)
db.session.commit()
```

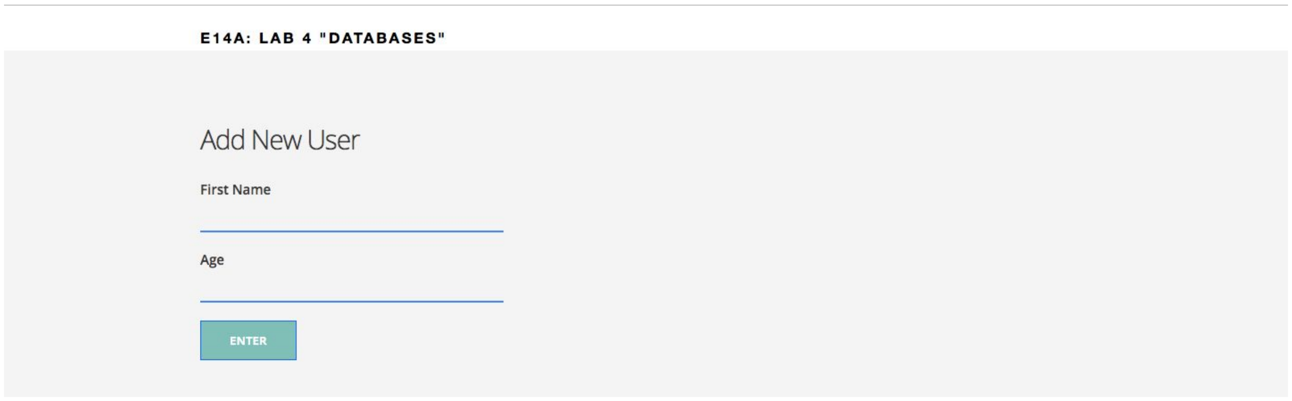
After commit we would like to go back to the index.html page this is what has been enabled with this line of code:

```
return redirect(url_for('index'))
```

*Important: You may see an error message that says, "No module named psycopg2." If you do, this means that the library psycopg2 is missing and you have to install it. Use pip!*

We created a simple web form where users can enter their basic information and store it into the database. Check your database with *SELECT * from users;* to make sure your newly entered data are in. These are the final result screenshots:





## Weekly Challenge: **CR(U)D**

You have already developed the **C** (create) of the **CRUD** functionality in your Flask application. Now, you should extend the application with **Read** and **Delete** functionalities. That way, you'll be able to manipulate the data in the "users" table without having to type SQL commands.

1. **READ** functionality should be realized as an HTML table that contains all database users' data. Also, this functionality should be implemented on a separate HTML page and it should contain a link to an already existing CREATE page (add-user).

2. **DELETE** link on READ page should lead to a page that deletes selected record from database and redirects user to the READ page.

3. Use JavaScript built-in confirmation dialog function to prevent from accidentally deleting a record (1 pts).

4. Develop an **UPDATE** functionality that allows you to edit user's details (4 pts).

### Credits and Additional Resources

- The Flask Mega-Tutorial Part III: Web Forms: link
- The Flask Mega-Tutorial Part IV: Database: link