CyanoConstruct

A general explanation for maintenance

Table of Contents

last updated: September 6, 2020 by Lia Thomson

Directory Structure

- **cyanoConstruct**: the overarching folder; contains all of the Python files
    - **files**: contains folders and files as they are created in order to be sent to users, which are deleted once sent; usually empty
        - **zips**: folder to contain **.zip** files as they are being created and sent to users, which are deleted once sent; usually empty
    - **migrations**: used by Flask-Migrate to perform database migrations
        - **versions**: contains the Alembic scripts used to perform migrations
    - **misc**: contains miscellaneous Python scripts
    - **static**: contains static files, used for the website (not the server)
        - **images**: contains images that are to be displayed on the site
        - **js**: contains JavaScript files for the site pages
        - **styles**: contains CSS files for the site pages
    - **templates**: contains HTML templates for the website

(There is also the **__pycache__** folder, which can be ignored and/or deleted. The **.git** folder and **.gitignore** file are used for Git and GitHub.)

Python File Summaries

All the Python files also have docstrings at the beginning, describing what they do. For easy reference, here is a compiled overview:

- **RunThisFile.py**: the file that should be run to host the server (mainly for testing). Perhaps not the best application structure.
- **__init__.py**: officially contains all of the import statements, and is what is referred to in a "`from cyanoConstruct import ...`" statement.
- **config.py**: used to configure the Flask `app`, mainly containing database information. (Note: it is *not* updated via Git because the testing database used on my computer uses a different configuration than the version on PythonAnywhere.)
- **routes.py**: the routes (i.e. the URLs of the site).
- **routesFuncs.py**: many functions used by **routes.py**. They are stored in a separate file to keep the file lengths manageable.
- **users.py**: defines the UserData class, contains several functions for handling users.
- **component.py**: defines classes for component-associated information.
- **database.py**: the database tables, defines the database.
- **enumsExceptions.py**: defines custom exceptions (i.e. errors). Originally contained custom enumerations, thus the name, but said enums are no longer in use.
- **misc.py**: contains a couple general, short, miscellaneous functions used by other files.
- **misc/dbBackup.py**: backs up the database on PythonAnywhere.

The rest of this document works through the application from the outside in.

The Site: HTML, CSS, JavaScript

This section covers what the end user sees and directly deals with.

HTML and CSS

HTML is a markup language, and CSS is a style language. HTML informs the computer *what* to display on a website (e.g. a paragraph of text or an image), and CSS says *how* those elements are displayed (e.g. their colors, relative sizes). HTML and CSS together determine the appearance of a site.

The HTML and CSS files will only need to be altered if the content of a page is changed or another page is added to the site. W3Schools is a good resource for learning HTML and CSS.

There are only two things to note about the HTML files:

1. The **.html** files in the **templates** folder are templates only, *not* complete HTML files. The functions in the **routes.py** file, through Flask, use variables (e.g. which user is logged in) to "complete" the HTML templates and render them so the user can see and interact with the web page. The Flask Mega-Tutorial has a clear explanation of how templates work.
2. The most useful part of the HTML files are contained within forms as `<input>` elements. There are multiple types of input, like buttons, check boxes, and text boxes. These are where the user inputs information. JavaScript will then gather the information from the `<input>` elements to send to the server.

JavaScript

JavaScript is the scripting language of the internet. It makes sites dynamic and functional.

JavaScript files are stored in the **static/js** folder. There is a **.js** file for every page that requires more than a couple lines of JavaScript. The functions within the CyanoConstuct JavaScript files perform four main types of tasks:

1. They make the site more attractive and usable. For example, the "[Click to show/hide]" functionality on the */library* page works due to JavaScript.
2. They change a web page's contents based on user input. This is mainly seen on the site where there is an "Add New [Thing]" and "Remove Last [Thing]"-type functionality.
3. They check whether the data in a form is valid before sending the information to the server.
4. They send information, either due to a button-press or the submission of a form, to the server, and then handle the output. The output is usually a short message saying the user succeeded or failed.

The first three types are relatively varied and will not be discussed. The last functionality will be discussed below.

The HTTP Request

The most essential part of sending site information to the server is the HTTP Ajax request. It has the general format of:

```
1.    $.ajax({
2.          data : {name1: value1,
3.                name2; value2},
4.          type : 'POST',
5.          url : '/ROUTE'
6.          })
7.    .done(function(data){
8.          //do something with data.output
9.
10.         if(data.succeeded){
11.               //do things if it succeeded
12.         }
13.         else{
14.               //do things if it failed
15.         }
16.   });
```

The parts in **bold** are changed for different functions.

Lines 2-3 define `data` as a JavaScript object, with two name-value pairs. The server accesses values using the corresponding name. For example, if data were set as

```
data : {email: "test@gmail.com"}
```

Then the server could access "email" in the request and get "test@gmail.com".

Line 5 defines the route that `data` is sent to. Routes are explained in the routes section.

Line 8 is a comment about doing something with `data.output`. In my functions, the server generally returns two pieces of information to the web page, once the request has been processed: `output` and `succeeded`. `output` is a string of HTML that is displayed to the user, either as an error message or a success message. `succeeded` is a Boolean describing whether the process succeeded. Lines 10-15 have space to do different things depending on the value of `succeeded`.

Routes

Flask

Flask is the web development framework used here. It has many extensions so it has many capabilities. The extensions used here are Flask-Login, Flask-SQLAlchemy, and Flask-Migrate.

A couple variables of note are created directly from the base Flask package:

1. `app`: is an instance of the `Flask` class. It is used to configure Flask-Login, Flask-SQLAlchemy, and Flask-Migrate. `app` is what runs so the web app exists.
2. `session`: is basically the cookies stored on the user's browser. Information is usually stored and accessed in `session` for multi-part processes, e.g. designing a component on */design*.

routes.py and routesFuncs.py

Routes are basically the URLs that exist on the site. All of CyanoConstruct's routes are linked to a specific function in **routes.py**. There are three main types of routes in use here:

1. Ones that display a web page, e.g. */library*. These are meant to be visited and viewed by the user.
2. Ones that produce a ZIP, e.g. */newComponent.zip*. These are not meant to be viewed by the user, but when visited, they return a **.zip** file that is immediately downloaded onto the user's computer.
3. Ones that process information, e.g. */processAssembly*. These are not meant to be viewed directly, but are visited by various JavaScript functions due to user input.

The functions contained in **routesFuncs.py** are used exclusively by the functions within routes.py. They are in a separate file to prevent routes.py from being 3000 lines long.

Routes for Displaying a Web Page

There is one of this kind of route for each visitable page on the site. The */about* page's route is a very simple example.

```
1.    @app.route("/about")
2.    def about():
3.        """Route for /about. The information page."""
4.        return render_template("about.html", loggedIn = checkLoggedIn())
```

Lines 1-2 set the function `about()` as the route for */about*. So, `about()` is called whenever a user tries to visit */about*.

Line 4 renders and returns the web page the user is meant to view. This is done using `render_template()`. `render_template()` takes two arguments here: the template to

render, **about.html**, and `loggedIn`. The template uses `loggedIn` to determine what links the navigation bar should have. Other arguments can be passed in and used by templates to change what information is displayed.

(I use `loggedIn` for the rendering of all templates, so for convenience, I made the function `checkLoggedIn()` to return the proper value, `True` or `False`.)

Routes for ZIP files

There are multiple routes that produce **.zip** files. They are all complicated, so below is a simplified version of the function for */componentZIP.zip*.

```
1.    @app.route("/componentZIP.zip")
2.    def getComponentZIP():
3.          """Create and send the .zip for a component."""
4.          compID = request.args.get("id")
5.          offset = int(request.args.get("timezoneOffset"))
6.
7.          comp = ComponentDB.query.get(compID)
8.          checkPermission(comp)
9.
10.         #make the .zip file
11.         compZIP = comp.getCompZIP(offset)
12.         data = rf.makeZIP(compZIP)
13.
14.         return Response(data,
15.               headers = {"Content-Type": "application/zip",
16.                          "Condent-Disposition": "attachment;
17.                                filename='componentZIP.zip';"})
```

In the function `getComponentZIP()`, a few things happen.

Lines 4-5 get information from the request. This is done through `request.args.get()`. The arguments that can be accessed are attached to the end of a URL.

For example, if using: *cyanoconstruct.com/componentZIP.zip?id=20&timezoneOffset300*

Then, `request.args.get("id")` returns `"20"`, and `request.args.get("timezoneOffset")` returns `"300"`.

Lines 7-8 use the component's ID to retrieve the component from the database, and check whether the user has permission to use that component.

Lines 11-12 produce the **.zip** file and save it as `data`.

Lines 14-17 return `data` as a downloadable file using `Response()`. (I do not understand the settings used, and only change the `filename`, which may not be needed anyway.)

Routes for Processing Information

There are several routes that process information. Their general structure is:

```
1.    @app.route("/ROUTE", methods = ["POST"])
2.    def FUNCTION():
3.          outputStr = ""
4.          validInput = True
5.          succeeded = False
6.
7.          #get data from web form
8.          try:
9.                value1 = request.form["name1"]
10.               value2 = request.form["name2"]
11.         except Exception:
12.               validInput = False
13.               outputStr = "ERROR: invalid input received.<br>"
14.
15.         #process information
16.
17.         return jsonify({"output": outputStr, "succeeded": succeeded})
```

The parts in **bold** are changed for different functions.

Lines 3-5 define `outputStr`, `validInput`, and `succeeded`. These variables are modified throughout the function. `validInput` is a Boolean reflecting whether the input is valid, `succeeded` is a Boolean determining whether the operation succeeded, and `outputStr` is a string message, indicating success or failure, to return to the user.

Lines 8-13 try to get data from the $ajax request. The values are accessed using `request.form[`**`name`**`]`.

Line 15 is where the data would be processed, and is where whatever complicated operations needed occur. `outputStr` and `succeeded` are modified as necessary.

Line 17 returns `outputStr` and `succeeded` to the site using `jsonify()`. The site, using JavaScript, can then handle this information. Sometimes additional information is sent.

Types of Objects

There are multiple unique types of objects. They are not strictly objects, in the computing sense, but rather ways data is grouped. They have slightly strange names by which they are referenced extensively in the code comments, which is why they are explained here.

Named sequence

A named sequence represents a sequence of DNA. It contains: a DNA sequence, a name, and a type (i.e. promoter, RBS, GOI, terminator). A named sequence does not have a position, unlike components, which are derived from named sequences.

The `NamedSequence` class stores data about a named sequence temporarily. Likewise, the `NamedSequenceDB` database table stores a named sequence in the database. Each entry in the `NamedSequenceDB` is linked to a user.

Spacer data

Spacer data represents a set of spacers for a single component. It contains: a DNA sequence for the left spacer, a DNA sequence for the right spacer, the position it is for, and whether it is for a last component (referred to in code as `isTerminal`).

The `SpacerData` class stores data about a pair of spacers temporarily. The `SpacerDataDB` database table stores spacer data in the database. Every `SpacerDataDB` is linked to a `ComponentDB`.

Primer data

Primer data represents the primers for a component. It contains: whether primers were found (`False` if primers were skipped OR if no primers are possible with the desired TM), the TM of the left and right primers, the GC content of the left and right primers, and the DNA sequences of the left and right primers, 5' to 3'.

The `PrimerData` class calculates and stores data about primers. A `PrimerData` is linked to a `SpacerData` in order to add the spacer sequences to the end of the primer sequences. The `PrimerDataDB` database table stores primer information in the database. Every PrimerDataDB is linked to a `ComponentDB`, but not a `SpacerDataDB`.

Component

A component stores information about a component. It is linked to a named sequence, spacer data, and primer data. It does not contain other information.

There is a `ComponentDB` database table. Each entry in the database is linked to: one `NamedSequenceDB`, one `SpacerDataDB`, one `PrimerDataDB`, and one `UserDataDB`.

User data

User data stores information about users.

There is a `UserDataDB` database table. It contains information used to identify the user. The `UserData` class is a wrapper class that is used to deal with `UserDataDB`.

The Dreaded Database

The database is created and managed using Flask-SQLALchemy. The database object that is referenced throughout the Python files is named `db`. PythonAnywhere hosts a MySQL-type database for the official site, but I use a SQLite-type database on my own computer for testing.

Database tables (i.e. how the database is defined) are in the **database.py** file. These tables are classes that inherit from `db.Model`. Each table has columns defined using `db.Column()`. A column is a piece of information (e.g. an integer) attached to each entry (row).

Whenever the tables' columns are modified, a database migration must be performed using Flask-Migrate.

Please reference the Flask-Migrate, Alembic, Flask-SQLAlchemy, and SQLAlchemy documentation to learn how to modify the database. I do not understand it well enough to explain it, and find making changes to be an error-inducing struggle. Avoid altering the database if possible, unless you know what you're doing.

The methods in the database tables (e.g. `getHTMLdisplay()`) *can* be modified without having to perform migrations or worrying about random database-related errors. It is changing the columns that causes problems.

One thing to note is that I could not get SQLAlchemy's relationship capability to work, so instead use entry ID's to link entries in the database.

Reference Information

Links for various packages and resources used extensively.

Flask

Flask is the main package used.

Documentation: https://flask.palletsprojects.com/en/1.1.x/

This is a very thorough tutorial on Flask. I referenced chapters one through five extensively.

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Flask Extensions

Flask-Login is used for handling logged-in users.

Documentation: https://flask-login.readthedocs.io/en/latest/

Flask-SQLAlchemy is used for the database. It is a wrapper for SQLAlchemy.

Flask-SQLAlchemy documentation: https://flask-sqlalchemy.palletsprojects.com/en/2.x/

SQLAlchemy documentation: https://docs.sqlalchemy.org/en/13/

Flask-Migrate is used to perform database migrations. It is a wrapper for Alembic. I usually make modifications to the Alembic scripts it produces.

Flask-Migrate documentation: https://flask-migrate.readthedocs.io/en/latest/

Alembic documentation: https://alembic.sqlalchemy.org/en/latest/

Other

Google Sign-In is used to sign in and register users.

Documentation: https://developers.google.com/identity/sign-in/web/

In general, I find W3Schools is a good resource about HTML, CSS, and JavaScript.

Site: https://www.w3schools.com/

PythonAnywhere hosts CyanoConstruct.

Site: https://www.pythonanywhere.com/