**BioPath**

**Environment Setup:**

Required Software (you will need to download these when setting up the environment):

Django, we used version 2.1.3

Python 3

Pip 3

Git

VisualStudio code, or other prefered IDE

Directions:

1. We had Jason create a virtual environment on the Gonzaga servers for us, you will likely want to do the same.
2. Once you navigate to the location you would like to develop in, clone our repository. <https://github.com/sjoyce2/bioChemApp_SeniorDesign>
3. If you enter ls you should see a directory: **bioChemApp\_SeniorDesign**
4. Enter the directory
5. If you haven’t already, install python3 and pip3, these will be used to install other software.
6. Once you have successfully installed python3 and pip3, activate the virtual environment.
7. **klarson7@igt4-biochem**:**~/seniorProj/bioChemApp\_SeniorDesign**$ source seniorProjenv/bin/activate
8. Your command line prompt should look like: (seniorProjenv)**klarson7@igt4-biochem**:**~/seniorProj/bioChemApp\_SeniorDesign**$
9. Now download django:

(seniorProjenv) **klarson7@igt4-biochem**:**~/seniorProj/bioChemApp\_SeniorDesign**$ pip3 install django

1. Follow the prompts to complete the installation.
2. Now check if your environment works:

(seniorProjenv) **klarson7@igt4-biochem**:**~/seniorProj/bioChemApp\_SeniorDesign**$ python3 manage.py check

1. If it works you’ll see: System check identified no issues (0 silenced).
2. If it doesn’t make sure you have properly installed pip3 and python3.
3. Also make sure you installed the latest version of Django. You should have Django 2 or later.
4. Once you have no issues detected you can type: python3 manage.py runserver
5. You should see a line saying something like: Starting development server at http://147.222.165.81:8080/
6. Go to that ip address and you should be able to interact with the website.

Accessing website when server is running:

Go to http://147.222.165.81:8080

**Important Files:**

Manage.py:

Manage.py is created by django when the project is created and is located at the root directory it is the first file that runs when you run the server. It specifies what version of Django to use and any other connected software.

Settings.py:

Settings.py is also created by django and is located inside the seniorProj directory. It contains all the settings for your project and links the project to your database. It also links to SendGrid which allows users to reset their passwords over email.

Urls.py:

Urls.py is another important django file. There are 2 urls.py, one in the seniorProj directory and one in the testApp directory. The one in the testApp directory is what we use for our website. Here you supply all of the url patterns that are recognized and link those urls to a particular view in views.py. You can think of urls.py as a router, it takes in the url from the browser and routes you to the appropriate view. There is one url for each of our webpages.

Views.py:

Views.py is another important django file. It controls what code runs when a particular webpage is opened. For example when the modelChoice screen is opened the urls.py routes us to the modelChoice function in views.py and that function determines what data to send to the webpage. Also if views is the concept of models. In django models are references to specific tables in the database (see our database schema below). To continue with our example, modelChoice links to the Model table in the database.

**\*NOTE\* from now on when I use the word Model, this refers to the Model table in the database and when I use the word model, this refers to django’s concept of models.**

In Views.py you are also introduced to the concept of querysets which are django’s version of sql queries. Querysets look like this:

privateModels = Model.objects.all().filter(userID\_id\_\_exact = user.id)

publicModels = Model.objects.all().filter(public\_\_exact = True)

In the example above, pulled from the modelChoice function in views.py, there are 2 querysets created. One, is grabbing all rows from the Model table where the userID equals a specific value and the other is grabbing all rows from the Model table where the public column is set to true.

For more information on querysets see:

<https://docs.djangoproject.com/en/2.2/topics/db/queries/>

<https://docs.djangoproject.com/en/2.1/ref/models/querysets/>

In Views.py you also set the data to send to the context. The context is sent to you webpage and can be accessed in your JavaScript and HTML files. So if you want specific data to appear on your webpage, this is how you do it. At the bottom of each function in views you will see a variable, context. This variable is and all its contents are sent to the html file it is linked to.

For example in the modelChoice function in views.py:

context = {

'userID': user.id,

'models': privateModelsList,

'publicModels' : publicModelsList

}

return render(request, 'modelChoice.html', context=context)

This context variable is then accessed by doing the following:

**Option 1)**

In the html file create a script section and inside the script tags have something like:

var myModels = [];

{% for model in models %}

myModels.push(['{{ model.modelName }}', '{{model.id}}']);

{% endfor %}

The curly brackets and percent signs signify that you are accessing the data sent from views.py.

The code in the example above creates a list, myModels, which is populated with the data from models, which was sent in the context variable.

**Option 2)**

In the html file you can use, {{}}, to access the context variable. For example, in modelEdit.html you can see this being done. We then use the data in the context variable to create JSON objects that we will use in the javascript.

{{ modules|json\_script:"db-modules" }}

{{ substrates|json\_script:"db-substrates" }}

{{ products|json\_script:"db-products" }}

{{ model\_num|json\_script:"modelNum"}}

{{ xCoorNext|json\_script:"x-coor-next" }}

{{ yCoorNext|json\_script:"y-coor-next" }}

{{ pubModel|json\_script:"pubModel" }}

Views.py is an extremely important file that handles all GET and POST requests and connects to the database. For more information go to: <https://docs.djangoproject.com/en/2.2/topics/http/views/>

Forms.py:

Forms.py is where you enter the forms you create in your html files. The Django concept of forms if very similar to html forms and they are primarily used to handle POST requests. In our code we have 2 primary POST requests, one for signing up and the other for saving a new Module to the user’s Model. In forms.py we have a form created for both of these instances. The form class in forms.py contains variables for each of the values we need the form to retrieve. The SignUpForm is inherited from django, but the SaveModuleForm was created by us. For more information on django forms see: <https://docs.djangoproject.com/en/2.2/topics/forms/>

Models.py:

Lastly we have models.py which is another important django file that is used to create models, which are representations of the database tables. There is one model per database table. In our project we have 4 primary models. Models, Modules, Products and Substrates. For the most part, every column in your database table becomes a variable in your model in models.py. For more information see: <https://docs.djangoproject.com/en/2.2/topics/db/models/>

Other:

There are a couple other files create by Django, like admin.py and tests.py and apps.py, but these have not changed at all since we first created the project.

**HTML, JavaScript and CSS:**

Templates:

Inside of the testApp directory there is a templates directory. This directory holds all of the html files for the project. These html files are called by the views functions. At the bottom of each views function there is a return statement. This return statement, sends a request, calls a specific html file and sends the context variable. For example if you look at the return statement in the modelChoice function in views:

return render(request, 'modelChoice.html', context=context)

As you can see the modelChoice function is calling modelChoice.html, which means this html file will be rendered. Each html file extends indexLogged.html. But other than that we use standard html throughout. However if you are having trouble identifying what html code is effecting a particular object on screen, open the developers console on Chrome while running the website, go to the elements tab, and select the icon in the upper left. Or use the shortcut to Toggle Inspect Element Mode, Command + Shift + C. Then hover each element and you should be able to see the html that affects that element in the developers console.

Static:

Inside the testApp directory there is a static directory. This directory holds all of the JavaScript, CSS, images and fonts files. Each JavaScript and CSS file follows the same naming convention as the html files, however if it is unclear which JavaScript file goes to each Html file, there is a line in each html file, encapsulated in a script tag, that links the JavaScript file. All of the css files are linked in the indexLogged.html file.

JavaScript Files:

Note - each JavaScript file should be well commented, however below is a description of each file and what it does.

**modelEdit.js:**

When the window loads, main() is triggered to run. main() calls 5 functions: createSliders(), addValues(), reset(), render() and window.requestAnimationFrame().

createSliders():

As the name suggests, this function creates the sliders on the modelEdit page of the website. The sliders are created dynamically based on what Model the user is currently looking at. There is a slider for each enzyme in the Model.

addValues():

This function creates the lists representing what will be drawn to the canvas.

reset():

This function resets the values of each slider when the page is reloaded.

render():

Draws the model to the canvas.

window.requestAnimationFrame():

This is a function used to animate the Model. This function is repeatedly called while the page is loaded.

**moduleEdit.js:**

The first function to run in this file is init(). This function is run when the moduleEdit page is loaded. The next step is to determine if the Model the user is looking at is public or not. If the Model is private the user should be able to to save the Module to the Model, but if the Model is public this feature needs to be disabled. The init() function will also grab the necessary elements from the Html, so they can be modified or trigger different functions.

The init function will call createErrorCheckArrays(), replaceUnderscores(), displaySnackbarHelp(), enableAndDisableBtns(), and setArraysRepresentingReaction().

CreateErrorCheckArrays(): This function grabs the data sent in the context variable from the views function and stores them in lists. The specific variables it looks at are, allSubs, allProds and allModules. These variables represent every reaction in the completed model and are used for error checking the reactions that the user enters.

replaceUnderscores(): For some reason if we send a string to the database that has spaces and retrive it with a Django queryset, there are a bunch of issues. So as a work around, any time a string with spaces is sent to the database, the spaces are replaced with underscores. So, this function does the opposite. It looks for underscores in the myEnzymes, myProducts and mySubstrates lists and replaces them with spaces.

displaySnackBarHelp(): This function triggers a snackbar to appear when the page loads and gives a short hint for what the user should do.

EnableAndDisableBtns(): Check if the Model is public. If it is disable the save button. If it isn’t set the saveBtn to type, button. The saveBtn has the type submit because it is responsible for sending the POST request to views and trigger the saving of the new Module. However, this should only be allowed if the user has entered a valid reaction, so we set the saveBtn’s type to button until the user creates a valid reaction and the POST request would be valid.

setArraysRepresentingReaction(): This function triggers all of the other functions neccesary for displaying the reaction on the canvas, performing error checking and more. The first thing it does it take the myProducts, myEnzymes and mySubstrates lists and updates the following variables:

countProducts

countSubstrates

checkedSubsNames

checkedProdsNames

checkedEnzsNames

checkedSubsAbbr

checkedProdsAbbr

These variables are then used to determine if the reaction is valid and what should be displayed on the canvas. The next function to be called is validateReaction(), which as the name indicates, validates the reaction. If the reaction is valid checkSubsEnzProds() and displayReaction() are called. These functions check the appropriate checkboxes and radio buttons that correspond to the reaction and display the reaction on the canvas.   
  
All of that happens when the page is loaded. After that there are a series of onClickListeners that trigger this process to repeat when the user interacts with different on screen elements.

**modelChoice.js:**

This file uses JQuery. It also creates 2 sections, one for “Your Models” and one for “Our Models”. “Your Models” holds the private versions of each of the public models that are specific to the current user. “Our Models” hold all of the public models and are completely filled out. Each section has a button for each of the models. These buttons are generated dynamically when the page is loaded.

**main.js:**

This file has simple functions that change to look of buttons on the screen based on whether the user has the mouse hovered over the button or not. It also controls what drops down in the text boxes and has suggestions when you click on text boxes. This functionally shouldn’t need to be changed.