

Python-Atlassian basics MPP Course

About the course

This course will teach you how to interact with Jira, Confluence, Stash/Bitbucket, and other Atlassian DevOps tools using the Python programming language, and Atlassian's freely available API (Application Programming Interface).

This will allow you to:

- Create a website that allows you to quickly pull from Jira's underlying database, and present that data in a table, a spreadsheet, or just about any other format
- Automate any reports you would otherwise have to create manually

About Me

I'm Rick Segrest, a Huntsville native who started my career in Graphic Design with a degree from Auburn. I then went back to UAH to get a Computer Science degree and a Master's in Modeling and Simulation

I have a wife and three kids: two 12-year-old identical twin girls, and a 7-year-old boy

After doing graphic design, I did defense work for 12 years, and then came to NASA as a Jacobs contactor in early 2019. I have done a variety of things, but have been mainly focused on User Interface/User Experience development, and Full-Stack web development using React, JavaScript/TypeScript, and Python/Flask on the backend.

I have supported the ES52 Software Tools Team, along with ED04 Advanced Concepts Office and the ES50 ECLSS four-bed Digital Twin simulation effort.

Before starting the course, you should make sure you can run Python and are able to "pip install" Python libraries on your PC. I also use VSCode throughout this project, so installing VSCode is also very helpful. All of this is freely available, but you may need admin privileges on your PC.

If you are not able to install Python, pip, VSCode, etc. you can use a cloud IDE like codesandbox.io. You will have to sign up, but the free account will allow you to do everything you need to for this project.

If you have any issues, please email me and we can resolve those ahead of time.

A language that was meant to be simple and powerful, and has become one of the most popular iisince it was introduced a little over 30 years ago.

"Python" was named after the "Monty Python" comedy troupe. They created the British TV Show Monty Python's Flying Circus, which made it's way over to the US and developed a cult following. Monty Python also did several movies, including the very popular and often quoted Monty Python & the Holy Grail.

Programmers have a lot of in-jokes from pop culture, but most seem to be references to Monty Python shows and movies, and The Hitchhiker's Guide to the Galaxy book. You may see a lot of Star Trek, Star Wars, Pokemon, and Dr. Who references as well.

Python was first created in 1991

1.0 created in 1991

2.0 created in 2000

3.0 December 2008

3.12.0 is the current version as of October 2023.

Python is simple to read and write -- a good language for beginners

Second-most popular programming language next to JavaScript

- I use JavaScript for front-end web development (React)
- I use Python for back-end, server, and desktop application development

Jira Cloud (vs. Jira Server)

Atlassian had two ways to host Jira and their other tools:

1. You can set up Jira on your own web server, and host it yourself. This is typically what NASA does. This allows you to limit access to exclude anyone outside your organization's network, and has traditionally been considered more secure for this reason.

2. You can use Atlassian's cloud-based service hosting service that allows you to access Jira from anywhere. Since this is hosted on a public IP address, potentially anyone with a user's account information could access it, however, there are better security options now that help secure the data and limit access in other ways.

However, it seems that Jira is no longer supporting the server-based option, and they are attempting to move everyone to the cloud. So, if you are using one of NASA's Server-based Jira portals, they are likely not running the latest version, and therefore you may not be able to do all of the things that the latest API allows you to do.

To access all of the API functionality on a NASA Jira server, you may also need to get approval for elevated privileges.

I set up a Jira Cloud instance with sample data so we can learn the basics.

IDEs (Integrated Development Environment)

Although you can code in Python from a terminal with text and command line interfaces only, it makes it much easier to get a free IDE like Microsoft Visual Studio Code. This is what I generally use.

Other popular IDEs are Eclipse, Atom, Sublime Text, Codium, PyCharm, etc. Most are free or have a free version, and they all work in a similar way, but have slightly different feature sets, and some may have advantages and disadvantages over others.

The main benefits of using an IDE like VSCode are:

- Syntax color-coding and highlighting
- Code file organization
- Debugger integration
- Terminal integration
- Mouse interactions to follow links to references to things like imported source code files
- Errors are underlined in red
- Output or lists of Problems with the code are shown in a auxiliary panel
- "Linting" can be done as you go (PyLint, PEP8 Standard)

For this tutorial project, you can follow along by installing VSCode on your computer or you can use the online IDE called CodeSandbox.

If you navigate to "codesandbox.io" into your browser, and sign up or log in with an existing account, you will see something that looks and acts just like VSCode inside your browser, and you can do much of the development that you would ordinarily do with a VSCode application on your own machine, but all of the IDE functionality and your work-in-progress are there inside the online application, hosted in the cloud.

Obviously this is not a great option for working on sensitive NASA data, but for learning and sharing purposes, it works well.

You can set up a free CodeSandbox account to do the tutorial. Or you can set everything up inside your own IDE on your PC.

Coding Environment: Command-line tools

These command line tools need to be set up on your computer (or Virtual Machine, or cloud IDE):

Python 3

For our coding environment we will need an installation of Python 3, and preferably one of the latest versions (3.9-3.12+ should do).

PIP 3

We will also need "PIP," the Python "Preferred Installer Program." This application lets you install Python libraries using a simple command, and the program will automatically organize and enable updates and configuration "freezes" to make sure others can use the same configuration you have created for your code.

venv

If you are working on multiple Python projects, it is a good idea to create a "virtual environment" for each. This lets you keep each program's Python version and library configuration separate.

git (version control)

A critical tool for development is a version control program. "Git" is by far the most widely-used (93%). Git is free and open source, created by Linus Torvalds, who also wrote the Linux Operating System Kernel. It is generally used via the command line, but also has a free GUI to make it a bit easier to use for a newcomer.

Installing Python locally

The simplest way to do this on Windows is to go to python.org, then "Downloads" and click the big yellow button that says "Download Python *3.12.0".

*3.12.0 is the current version as of this writing, but is frequently updated. The latest version is usually the best to install.

Double-click to run the .exe that you just downloaded (usually in your "Downloads" folder).

Check the box that says "Add python.exe to PATH"

Click "Install Now" (you have to have Admin privileges on your PC)

Take note of the version number. After you install (you might have to restart the terminal or the computer), you can verify that you have installed Python correctly by entering the command:

```
> python --version
```

This should show a version number that matches the version number you just downloaded and installed--since I installed version 3.12.0, my terminal returns:

```
"Python 3.12.0"
```

The PATH Environment Variable

If you do not get the expected output when you enter the "python --version" command, it is likely because Python's binary (bin) folder is not in the list of paths. To add it, you can go to System-> Control Panel-> Advanced-> Edit System Variables, find PATH, and then add the folder that Python is installed in as a new entry in the list of paths (keep the existing entries).

This tells the PC where to look for commands that you enter into the command prompt.

If the PATH variable does not exist at all (this is uncommon), you can "Add" a system variable called PATH.

Verify that PIP is installed

"PIP" (a "recursive acronym" that stands for **PIP Install Packages**) is a tool that automatically locates libraries that we want to install to expand the functionality of Python.

PIP should have been installed with Python if you used the .exe installer from python.org. To verify:

```
> pip --version
```

```
"pip 23.2.1 from <path_to_python> (python 3.12)"
```

The version of python at the end of the output should match the version of Python that you just installed, and the version that showed up with you typed "python --version".

If you do not get a response that prints the PIP version number, you can enter the following commands to download and run a script to install it:

Installing Visual Studio Code

Visual Studio Code, or "VSCode" is a lightweight and easy-to-use code editor, that has a huge library of free extensions and configuration options, which makes it suitable for just about any coding project you can dream up.

Even though it is a Microsoft product, it is open source and free to download and use, as are about 99% of the extensions available.

It is a very different product than Microsoft's much older and proprietary IDE, Visual Studio, even though it has a similar name... The original Visual Studio is much more cumbersome, difficult to use, and expensive, and is intended for very large-scale projects.

There is a license that you have to agree to when you install the product which some may balk at-- if so, you can download Codium or one of the other IDEs that is built on the same source code but has a more open license and data privacy agreement.

To install Visual Studio Code:

Go to code.visualstudio.com/download

Click the big blue button that says "Windows 10, 11"

After the installer downloads, double-click it.

Accept the license agreement. (There is an open source alternative called Codium which has a freer license)

Select "Additional Tasks" -- I would recommend the four "other" options:

- * Add "Open with Code" action to Windows Explorer file context menu

- * Add "Open with Code" action to Windows Explorer directory context menu
- * Register Code as an editor for supported file types
- * Add to PATH (requires shell restart)

Click Next, then Install.

Run VSCode. If it asks for Firewall permission, approve this.

Now you can either close VSCode, or continue to use it.

Installing Git

Git is a project repository manager and version control system, and it has become ubiquitous among developers in 2024.

It is well worth using and understanding if you want to do any kind of development, especially if you plan to work well with others!

To get git, go to <https://git-scm.com/download/win>

Download the installer-- generally you will want the 64-bit, Standalone version.

(If you do not have admin rights, you can use the Portable ("thumbdrive") edition, which can run from any folder and does not make any changes to the Windows Registry or windows OS folders)

Double-click the installer from your Downloads folder.

Click "Yes" if it asks you "Do you want to allow this app to make changes to your device?"

Click "Next" on the (GNU) license page, and then choose where you want to install it (usually the default, C:\Program Files\Git, is fine, unless you don't have permissions to install here.

Next you choose the default editor used by Git. If regularly use "vi" or "vim" you can choose that, otherwise I would recommend changing this to "Visual Studio Code" or another application that you are familiar and comfortable with.

You will then be asked several other installation options during the install process. The defaults should be fine unless you have related experience and a strong personal preference.

After installation, verify that it has been installed successfully by opening a PowerShell terminal (if any are open, close those first), and then type:

```
> git --version
```

(with two dashes before the word "version")

You should then see something like:

```
"git version 2.42.0.windows.2"
```

You should have the option to install both the command line and a GUI version.

The GUI version might be helpful if you are a beginner, but it's good to get used to the basic CLI commands, and understand how they work: **clone**, **commit**, **pull**, **push**, **checkout**, **remote**, **log**, and **branch**, for starters.

Clone the project:

Clone the project repo:

Once you have git installed, open VSCode.

Type <CTRL-`> to open the (Powershell) terminal.

(This is the "grave accent" character, above <TAB> and below <ESC>)

You can also open the terminal by clicking View->Terminal on the top bar menu.

Now go to your HOME directory:

```
> cd %HOME%
```

Another way to do this is to use the Linux/Unix/Posix shorthand, which also works on Windows:

```
> cd ~
```

This is the "tilde" character, which you type by using the same key you use for the "grave accent," while holding <SHIFT>. This command will take you to your HOME directory (e.g. C:\Users\rick\)

```
> mkdir code
```

```
> cd code
```

This "makes" a "code" directory. Then you "cd" (change directory) into that new directory.


```
> git clone https://github.com/rsegrest/atlassian-mpp.git
```

OR, if you have an SSH key set up, you can instead use the command:

```
> git clone git@github.com:rsegrest/atlassian-mpp.git
```

I use this method, but it requires additional one-time preliminary set-up.

You can set up an SSH key within GitHub here:

<https://github.com/settings/keys>

Directions about how to generate an SSH key can be found here:

<https://docs.github.com/authentication/connecting-to-github-with-ssh>

This will pull down my "atlassian-mpp" repo into a new folder.

Python Libraries

We will use a number of various Python libraries. Some are built-in, and some you will install using PIP. I will introduce each of these as we start using them in the course.

The Test-Driven Development approach

I will show how to use a programming approach called "Test-Driven Development," or "TDD".

The idea behind TDD is that before you start writing code, you should have a simple goal in mind, and you write that goal out as a failing unit test.

You then write the code to make the test pass.

Each code should a single step forward. You then incrementally add more tests to achieve the next goal along the path to a complete application.

This is a well-known approach to software development that prevents you from writing a lot of untested code and making a mess. It takes a small amount of discipline, but allows you to prevent hidden bugs from entering into your code.

While it might seem like a lot of overhead and setup when you start doing things this way, you will find it can save you a lot of time in the long run, by exposing problems in your code immediately (or preventing them all together).

Using PyTest to write unit tests for Test-Driven Development

PyTest is a common unit-testing library that Python developers use (another common one is PyUnit). This will allow us to do Test-Driven development. I'll discuss it more when we get to the main project.

Using JSON (JavaScript Object Notation) to represent data

JSON is a plain-text document or string that uses key-value pairs to represent parameters and variables in an object. It was originally designed for JavaScript objects, but has a very similar structure to a Python dictionary, so is also useful in Python.

Python can easily deal with JSON and convert between a Python dictionary datatype to JSON, and vice-versa.

Using Flask Server in Python

Flask is another Python library that allows you to quickly put together a server that can be used for an API or serving web pages, but also allows you to integrate Python functions, scripts, and applications.

Jinja

A popular Python HTML templating library that helps parse and manipulate data in JSON format.

(Django is another popular Python-based alternative)

Using the Atlassian-Python-API to integrate our project with
Jira

This is the library that Atlassian provides that will provide the main functionality for our project to integrate with Jira.

Although we are working with Jira, the API allows you to access any of the other products in the Atlassian suite, such as Confluence or Bitbucket.

Once you get the hang of working with the Jira API, it shouldn't be difficult to work with any of Atlassian's other applications--just review the online documentation to find the functions that you want to access:

<https://atlassian-python-api.readthedocs.io/>

Note that there may be slight differences in the way the API works when you are using a Server-based installation of the Atlassian DevOps Suite, and these installations at NASA are usually not kept updated to the very latest version, so some functions and features may be unavailable, depending on the version.

You may also need to work with the IT group that manages the server to get the level of permissions that you need to access data.

Optional: Create a Docker Container for your Project

Docker containers can be used to run an instance of Jira Server or Atlassian Suite, and can also be used to set up your application in its own environment and sandbox, walled off from the rest of your computer's data and resources.

When you complete your project, you can then use a Dockerfile to make your project portable, so you can reproduce it on one or more servers.

It also allows you to run Linux or another Operating System inside your Windows or Mac computer, and configure that OS with only the tools you need for your application.

Docker is a somewhat advanced topic. It won't be covered in-depth here, but I'd encourage anyone who is serious about development or software engineering to look into it and understand what it can do.

Exporting to Common File Formats

Python libraries can export to various formats and produce a file for you to download. It can easily generate a plain text (.txt) or comma-separated-value (.csv) document with no external libraries.

Also, by PIP installing some freely available libraries, you can also generate documents in Microsoft Excel, MS Word, or Adobe Portable Document Format (.pdf) format.

Hello World in Python

Using CodeSandbox.io:

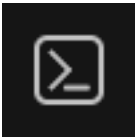
If you have created a new "Python" sandbox in CodeSandbox.io, you will have a file called main.py. Open the file in the editor.

Type the following on the first line, with nothing else:

```
print("Hello world!")
```

Using the Terminal in VSCode or CodeSandbox

Now look at the top edge of the bottom pane, and click on the icon:



Choose "New Terminal" from the drop down.

This will open a Linux command prompt underneath, in a new tab called "Terminal."

Type the following, and hit <RETURN> :

```
python main.py
```

The program should output the following:

```
Hello CodeSandbox!
```

Set up pytest

Create an "app.py" file in the root directory, and then create a "tests" folder, with a Python file where we will start writing tests.

The "touch" command will create a file if none exists. If the file exists, it will only update the timestamp.

```
touch app.py
mkdir tests
cd tests
touch test_app.py
```

Open the test_app.py file and write the following:

```
class TestApp:
    def test_pytest(self):
        assert True == True
```

Then click the Terminal. If you are still in the "tests" folder, go up one directory to the "root" workspace directory:

```
cd ..
```

Type the following to install PyTest. The "-U" option allows the application to be available to any Python project, not just this one:

```
pip install -U pytest
```

Now run PyTest by typing the following:

```
pytest
```

PyTest will then look in your project folders for any files that start with the prefix "test_" (with a .py extension), such as the file "test_app.py" that we created earlier.

After you click enter, PyTest should run and output some results:

```
===== test session starts =====
platform linux -- Python 3.12.0, pytest-7.4.2, pluggy-1.3.0
rootdir: /workspace
collected 1 item

tests/test_app.py . [100%]

===== 1 passed in 0.01s =====
→ /workspace git:(master) |
```

PyTest found one test in our project ("collected 1 item").

Tests were found in the "tests/test_app.py" source code file. There is a single green dot representing our test that passed. On the right side, [100%] shows the progress.

Our test completed immediately, but as you add more tests and more complexity, it might take several seconds to run the tests (if takes longer than something on the order of seconds, you need to improve your tests' efficiency, and you or others working on your project may stop running the test suite due to impatience.)

The green colored bar means that all tests passed that were run.

Using PyTest

Now let's add a failing test. Add the following below the first test case you defined:

```
def test_fail(self):
    assert True == False
```

Obviously this will always fail, just as the first always passes. Now run "pytest" and you see a very different-looking output:

```
→ /workspace git:(master) x pytest
===== test session starts =====
platform linux -- Python 3.12.0, pytest-7.4.2, pluggy-1.3.0
rootdir: /workspace
collected 2 items

tests/test_app.py .F [100%]

===== FAILURES =====
_____ TestApp.test_fail _____

self = <test_app.TestApp object at 0x7f37386d3ce0>

    def test_fail(self):
>     assert True == False
E     assert True == False

tests/test_app.py:6: AssertionError
===== short test summary info =====
FAILED tests/test_app.py::TestApp::test_fail - assert True == False
===== 1 failed, 1 passed in 0.03s =====
→ /workspace git:(master) x
```

Notice that the bar is now red, because you have a failing test. It says in the middle "1 failed, 1 passed."

If you look above this, it has printed the assertion that caused the failure.

```
> assert True == False
```

Then it shows the folder and file where the failing test occurred. Then ":6" means "line number 6." And the error was that we made an assertion that

couldn't resolve to "True" ("AssertionError").

Below this you will see the file name, followed by a green dot (as before), then a red "F." Much like looking at your returned test when you are experiencing a school flashback, the "F" means "failed."

You now have set up PyTest, and you are ready to start writing actual unit tests that will drive our development. Next we can write a meaningful test, and work to get rid of those "F"s.

Reading a Python config file, and our first "fail"

Now we want to change the first test we wrote to create a goal. It will fail now, then we will write the code to make it pass. Then we add to the test, or add a new test, and make the tests pass again.

Second verse, same as the first.

Our first test will be to drive our first functional requirement-- our application should be able to read a Python configuration file.

So, first write a (failing) test case to test that the program can do that. Remove the trivial test cases that we wrote before, and replace them with a new one:

```
class TestApp:

    def test_read_config_file(self):
        config = self.read_config_file()
        assert config != None
```

This test case will call a function (that does not yet exist) called "read_config_file()" and hold whatever the return value is in a variable called "config."

For now, we will write the "read_config_file" function just above the test function. Since the function does not begin with the prefix "test_", pytest will understand that, even though this function is inside a "test_" source code file, it is still not a test function, and shouldn't be run directly by the test runner.

The "self" keyword references the current class "TestApp," so "self.read_config_file" references the function "read_config_file" that is a member of the same class, "TestApp"

So now we write that member function:


```
class TestApp:

    def read_config_file(self):
        return None

    def test_read_config_file(self):
        config = self.read_config_file()
        assert config != None
```

The function is there now, but it is returning "None," which is exactly the opposite of what we want it to do, based on the test case we wrote. So, run "pytest," and that test will now fail.

Then we "assert" that the config variable has been assigned **something** (basically anything).

Well, for the moment our application can't do that, so the test fails.

Passing the Test

The next step is to write the functionality.

First, add the "config" library, which has helper functions to read and write config files. Just enter the following at the terminal:

```
pip install config
```

There are thousands of freely available Python libraries available using PIP which are also just this easy to add to your project. So, before you spend time coding common functionality into an application, check to see if any of the available PIP libraries can do the task in a suitable way.

There are also several other libraries that can accomplish the same function as this "config" library. You might want to check out some alternatives. Some may have additional features or advantages over this one.

Now we start with a config file with all of the basic parameters that we will need:

url: The URL of the API

username: A username that will be used for authentication with the API

token: An authentication token for that username

Create a file called "jira.cfg" in a directory called "configuration" and write the following:

```
url: "https://ricksecrest.atlassian.net"  
username: "rsecrest77@gmail.com"  
token: "<add_api_token_here>"
```

Then change the "read_config_file" function in test_app.py:

```
import config

def read_config_file(self):
    cfg = config.Config('configuration/jira.cfg')
    return cfg
```

Now that we have checked that there is something in the file, we want to check that it can read each parameter:

```
def test_read_config_file(self):
    config = self.read_config_file()
    assert config != None
    assert config['username'] != None
    assert config['url'] != None
    assert config['token'] != None
```

This test should now pass. The "config" object that is returned from the function is a Python dictionary data structure, which has three key / value pairs. We typed the key as a string inside the brackets, and this references each of these values in the object: 'username', 'url', and 'token'.

Next, we can verify that each of these values is a string, as we expect it to be:

```
def test_read_config_file(self):
    config = self.read_config_file()
    assert config != None

    # load each value into a variable
    username = config['username']
    url = config['url']
    token = config['token']

    # check that the value (now held in a variable) was
    assigned, as we did before:
    assert username != None
    assert url != None
    assert token != None

    # check that each is an instance of the "str" (string) data
    type:
    assert isinstance(username, str)
    assert isinstance(url, str)
    assert isinstance(token, str)
```

This is pretty much all there is to reading a configuration file! We will use those values in the next step, where we want to connect to the Jira Cloud API server.

Connecting our Application to Jira

Let's make a new test case called "test_connect_to_jira":

```
def test_connect_to_jira(self):
    config = self.read_config_file()
    username = config['username']
    url = config['url']
    token = config['token']
    url = config['url']
    username = config['username']
    jira = self.connect_with_token()
    assert jira != None
```

Then create a "stub" function above for the new test to call:

```
def connect_with_token(self):
    return None
```

The new "test_connect_to_jira" will fail. Now add the following:

```
def connect_with_token(self):
    config = self.read_config_file()
    username = config['username']
    url = config['url']
    token = config['token']
    jira = Jira(
        url=url,
        username=username,
        password=token,
        cloud=True
    )
    return jira
```

This test does the procedure we want to follow, but it will throw a "NameError: name 'Jira' is not defined."

To clear this error, we will need to "pip install" the atlassian-python-api library, and then import it.

PIP Install atlassian-python-api

This is because "Jira" is a class in the Atlassian Python API library. So we will "pip install" and then "import" that library, as we did "config". At the terminal, enter:

```
pip install atlassian-python-api
```

Then go back to the "test_app.py" source code file and add the new import statement at the top:

```
from atlassian import Jira
```

On this line, "Jira" is a class inside the "atlassian" package. There are also "Confluence," "Bitbucket," etc. classes that allow you to interact with the data inside other Atlassian applications.

This link hosts documentation for the library, showing all of the available classes and functions, and how to use them:

<https://pypi.org/project/atlassian-python-api/>

We are instantiating a new "Jira" class object and saving that instance to the "jira" variable.

When we create the new Jira object, we are sending the url, username, token (instead of a password), and flagging that we are using Jira Cloud (when you want to connect to a Jira Server installation, like most at NASA, you would change this to cloud=False).

Now running "pytest" should pass the tests. If the test passes, our application should now have the ability to connect to the live Jira API.

Other ways to authenticate

Several authentication methods are demonstrated on the online documentation page:

<https://atlassian-python-api.readthedocs.io/>

Username and Password

You can use your username and password to authenticate. However, it's best not to hard-code these values into your application, for many reasons, not least of which that there are massive security concerns.

You can pull these values in the config file, but make sure to lock down the file permissions to where it can't be read by prying eyes, and add it to the .gitignore file so it doesn't end up on a git remote repository server:

```
jira = Jira(  
    url='http://localhost:8080',  
    username='admin',  
    password='admin')
```

OAuth

The following shows an example of using OAuth (Open Authorization) to access the API. This allows you to delegate certain permissions to the application itself (and its users), rather than using your own Jira personal or administration account:

```
oauth_dict = {  
    'access_token': 'access_token',  
    'access_token_secret': 'access_token_secret',  
    'consumer_key': 'consumer_key',  
    'key_cert': 'key_cert'}
```

```
jira = Jira(  
    url='http://localhost:8080',  
    oauth=oauth_dict)
```

Personal Access Token

This is the method we used to authenticate on the ES52 Jira Server. There is a way to generate a Personal Access Token that is tied to your account, and carries all of your permissions.

The comment in the example below describes one method of obtaining a Personal Access Token:

```
# Obtain an API token from: https://id.atlassian.com/manage-profile/  
security/api-tokens  
# You cannot log-in with your regular password to these services.  
  
jira = Jira(  
    url='https://your-domain.atlassian.net',  
    username=jira_username,  
    password=jira_api_token,  
    cloud=True)
```

However, on our Jira installation there is a way to generate a token through the application itself, by clicking on your personal Profile avatar (in the upper-right corner), then Settings > Personal Access Tokens. Copy and save the value (it will only be shown once), and then copy it to a configuration file or another secure location that your code can access.

Request a List of Project from the API

We will now use the API connection to do something useful: Get an array of Python dictionaries.

Each dictionary holds the data for each project in our Jira application's database.

Then we can pull the project names (and other data items) from the project dictionaries.

Add the following test functions to our test class:

```
def test_get_projects(self):
    projects = self.get_projects()
    assert projects != None

def test_get_project_names(self):
    projects = self.get_projects()
    project_names = self.strip_project_names()
    assert project_names != None
    assert len(project_names) > 0
    assert project_names == ['Sample-Kanban']
```

We want to move the "read_config_file" function to a file called "readconfig.py" in the "configuration" directory:

```
import config

def read_config_file():
    cfg = config.Config('configuration/jira.cfg')
    return cfg
```

If you get import errors when you try to import the functions from the new "interface" module, try the following:

1. Make sure there are no errors, including whitespace / indentation problems, or spelling errors.
2. Add a blank file called "__init__.py" to each folder that you want to import from ("tests" and "interface" for our purposes).
3. If you are still getting import errors, try entering the following command at the terminal (make sure you are in the root folder):

```
export PYTHONPATH=.
```

The system environment variable "PYTHONPATH" (with no spaces) tells Python the starting point for which to look for source files and module directories. In this case, the dot/period (".") character refers to the present directory (that you are issuing the command from).

Now create a directory called "interface," with a Python file called "get_projects.py", and another called "connect_to_jira.py":

Move the import statement and the "connect_with_token" function that we wrote into "connect_to_jira.py":

```
from atlassian import Jira
from configuration.readconfig import read_config_file

def connect_with_token():
    config = read_config_file()
    username = config['username']
    url = config['url']
    token = config['token']
    jira = Jira(
        url=url,
        username=username,
        password=token,
        cloud=True
    )
    return jira
```

If you have removed the "connect_with_token" function from the TestApp

class, you can still reference the new location of the function by importing it at the top of the `test_app.py` file:

```
from interface.connect_to_jira import connect_with_token
```

Then fill out the `"get_projects.py"` source file with the following:

```
from interface.connect_to_jira import connect_with_token

def get_projects():
    jira = connect_with_token()
    projects = jira.get_all_projects()
    return projects

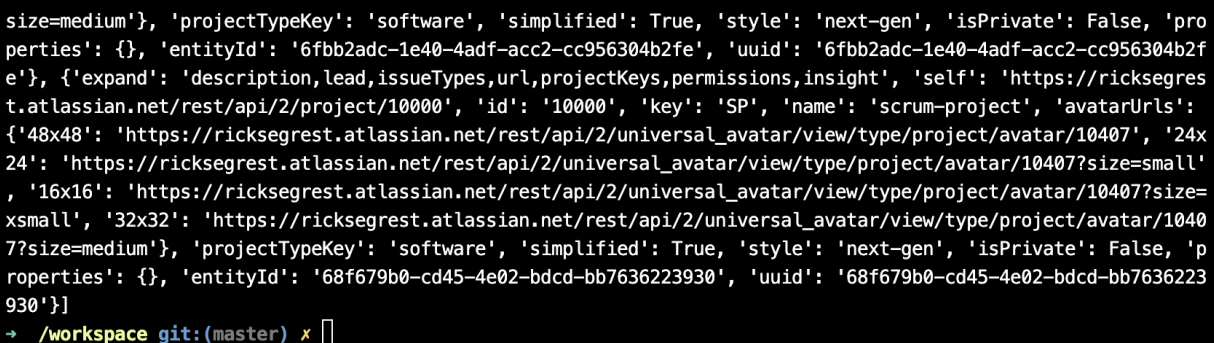
print(get_projects())
```

This source code defines the function to connect to the API and then call `"get_all_projects()"`. It will then return the response from the API, which should be an array of objects (one for each project in the database).

The last line calls that function, and prints the returned array to the screen. Run it and verify that it looks correct:

```
python interface/get_projects.py
```

You should see a lot of output, similar to below:



```
size=medium'}, 'projectTypeKey': 'software', 'simplified': True, 'style': 'next-gen', 'isPrivate': False, 'pro
properties': {}, 'entityId': '6fbb2adc-1e40-4adf-acc2-cc956304b2fe', 'uuid': '6fbb2adc-1e40-4adf-acc2-cc956304b2f
e'}, {'expand': 'description,lead,issueTypes,url,projectKeys,permissions,insight', 'self': 'https://ricksegres
t.atlassian.net/rest/api/2/project/10000', 'id': '10000', 'key': 'SP', 'name': 'scrum-project', 'avatarUrls':
{'48x48': 'https://ricksegrest.atlassian.net/rest/api/2/universal_avatar/view/type/project/avatar/10407', '24x
24': 'https://ricksegrest.atlassian.net/rest/api/2/universal_avatar/view/type/project/avatar/10407?size=small'
, '16x16': 'https://ricksegrest.atlassian.net/rest/api/2/universal_avatar/view/type/project/avatar/10407?size=
xsmall', '32x32': 'https://ricksegrest.atlassian.net/rest/api/2/universal_avatar/view/type/project/avatar/1040
7?size=medium'}, 'projectTypeKey': 'software', 'simplified': True, 'style': 'next-gen', 'isPrivate': False, 'p
roperties': {}, 'entityId': '68f679b0-cd45-4e02-bdcd-bb7636223930', 'uuid': '68f679b0-cd45-4e02-bdcd-bb7636223
930'}}
→ /workspace git:(master) x
```

It's a lot of data, so outputting it to the terminal isn't particularly useful. The `print` statement creates "standard out" which is streamed to the screen by default.

Using the API Output

We want to parse this output-- extract particular data points. So we are going to use this output as a test case. To do that, we will save it to a file called "project_data.py".

To save the output to a file, we use a little trick-- we redirect standard out to a file. In Linux and PowerShell, this is easy:

In Linux and recent versions of PowerShell, the ">" redirects standard out and replaces the target that you set:

```
python interface/get_projects.py > project_data.py
```

You can append an existing file or target by using the ">>" instead, so existing data isn't overwritten.

After you create the project_data.py file with the test case data, go into the file, and before the opening array bracket, add "project_data =" to assign it to a variable, so you can import the data into another file and do operations on it:

A screenshot of a code editor with a dark background. The title bar at the top says "project_data.py". On the right side of the title bar are icons for a window and a back arrow. The code area shows two lines: line 1 is "project_data = [{"expand": "description,lead,issueTypes,url,proje_" and line 2 is an empty line starting with a bracket "[".

```
project_data.py
```

```
1 project_data = [{"expand": "description,lead,issueTypes,url,proje_  
2
```

At the end of the file, you can try adding the following statements, and then run it directly:

```
print(len(project_data))
```

Then run in the terminal:

```
python project_data.py
```

Then you should see the terminal output a number, which is the length of the test data list (of objects). Verify this and then delete the print line from `project_data.py`.

Another useful thing to print would be the following:

```
print(project_data[0].keys())
```

This will take the first object from the array of projects in our test data, and list all of the data keys that are available, like 'id', and 'name'.

'name' refers to each project's name. We want to write a function to extract that from every single project object, and then return the names as a Python list (of strings):

```
def strip_project_names(projects):
    project_names = map(
        lambda x: x['name'],
        projects
    )
    return list(project_names)
```

This function sends the list of all projects into the built-in python "map" function. The map function iterates through a Python list, and does an operation on each element in the list.

The first argument to map is another function that defines the operations to perform on each element of the list.

The second argument is the list that will be operated on.

The operation function above is defined as an inline, anonymous "lambda" function. (You could also pass a function that has a name and has been defined elsewhere)

The lambda function assigns each element in the array to the 'x' variable (in our case, x is the project object from the array of projects).

It then returns x['name'], which is only the name from the project. So it filters out all the other information in that object that we don't currently want.

"map" then adds each name (a string) to the array "project_names", and the function returns the list of strings-- only the names of the projects without all of that other information.

This case is very simple, but map can handle more complex cases.

This is the way to get the 'id' and 'key' from a single project object:

```
def get_id_from_project(project_object):  
    # check if 'id' exists  
    if 'id' in project_object:  
        return project_object['id']  
    return ''
```

The function will only reach the last line (and return an empty string) if the 'id' is not present in the object that was sent as an object. Otherwise it will return the 'id,' as we expect.

Get key:

```
def get_key_from_project(project_object):  
    if 'key' in project_object:  
        return project_object['key']  
    return ''
```

Set up Flask

app.py is our "main" application entry point.

Here we want to use the Flask library to create a web server. Getting a Flask server started is simple. We can start with a server that just says "Hello World!":

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(debug=True)
```

The `@app.route('/')` annotation tells Flask to run the function defined on the next line (`hello_world`), when someone accesses that route (`/` is the "root route"--just the URL of the server. Later we will add new routes, appended to the end of the server URL.)

Before running the server source file, "pip install" flask:

```
pip install flask
```

Flask uses the "FLASK_APP" environment variable to know from where to kick off your Flask server (Linux / CodeSandbox):

```
export FLASK_APP=app.py
```

PowerShell:

```
$env:FLASK_APP = "app.py"
```

You can then just type "flask run" to run the server:


```
→ /workspace git:(master) export FLASK_APP=app.py
→ /workspace git:(master) flask run
* Serving Flask app 'app.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [10/Oct/2023 21:38:48] "GET / HTTP/1.1" 200 -
```

It shows that the server is running on `http://127.0.0.1:5000`, which is the localhost address, port 5000. If you are running the app on your own PC, you can open a browser and type in that URL (type the "http://" part because modern browsers typically default to "https://"). Hit enter, and you should see something like this in the upper-left corner of the page (with no formatting):

Hello World!

This shows that our server is working. You can configure Flask to change the port if necessary.

If you are running on CodeSandbox.io, it should have opened a small panel with a browser menu.

If you are not seeing this, open the drop down menu in the upper-left corner (with the square icon), select View-> Toggle Dev Tools. A panel should have popped up, and if it doesn't already have a tab that says "Preview: 5000," it should have a [+] icon where you can open that tab (5000 corresponds to the port number that Flask is running on).

(Since CodeSandbox.io runs in the cloud, the web server is also running on

the cloud--localhost on your own machine and browser won't work.)

Flask: Debug Mode

If you run Flask in Debug Mode, it will watch for changes you make and save in the source code, and will recompile and update that site, live.

Additionally, if there are any errors, the error and stack trace will be displayed to the screen. This is certainly not something you want in a production site, but when you are building a site, it can be very useful.

There are two ways to run Flask in debug mode:

1. Set the "FLASK_ENV" environment variable in the terminal before you run the command "flask run."

In Linux (bash shell), including the CodeSandbox terminal:

```
export FLASK_ENV=development
```

PowerShell:

```
$env:FLASK_ENV = "development"
```

2. You can also add "debug=True" as an argument to the app.run command in the source code, where you are starting the server.

```
if __name__ == "__main__":  
    app.run(debug=True)
```

Create an HTML template for Jinja

Create a new directory called templates. Then create a file called "index.html" in that folder.

Start with a basic HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Issue Search</title>
  </head>
  <body>
    <div>
      <div>
        <h2>Issue Search</h2>
        <!-- form goes here -->
      </div>
    </div>
  </body>
</html>
```

Add HTML template to Flask Route

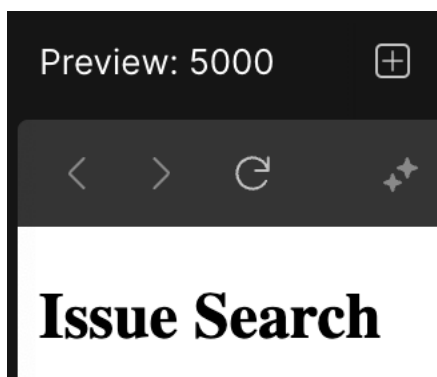
This is the template that we will use to allow Flask to render the webpage. We will change our "root route" in our Flask `app.py` source file to bring in the template:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def home():
    return render_template(
        "index.html"
    )

if __name__ == "__main__":
    app.run(debug=True)
```

Now try running the updated Flask server using "flask run." You should see your new .html template rendered when you navigate to the local server URL and port:



Now we can add a simple dropdown form, which will ultimately allow the users to choose what type of Jira Issue they want to search for.

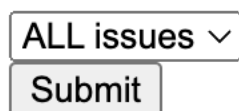
Create an HTML form

To start creating the form, remove the "form goes here" comment line, and add the following. Right now, we will hard-code the options for "issueType," but later we will get the values from the API and dynamically populate them:

```
<form>
  <div>
    <div>
      <label for="issueTypeSelect" />
      <select name="issueTypeSelect"
id="issueTypeSelect">
        <option>ALL issues</option>
        <option>Epic</option>
        <option>Story</option>
        <option>Bug</option>
      </select>
    </div>
  </div>
  <div>
    <button>Submit</button>
  </div>
</form>
```

If you are running in debug mode, the live page should update to look like this:

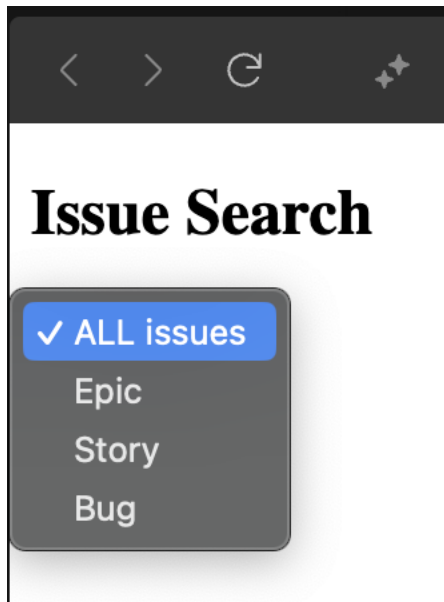
Issue Search



ALL issues ▾

Submit

Click the form, and the options (issue types) should "drop down":



Add basic CSS styling

We can also add some basic CSS formatting to make the page look nicer.

Create a folder called "static", and then another folder inside that one called "styles." Then create a file called "app.css." Before adding styles to the .css file, we will link it to the HTML page we just created. Below "<title>Issue Search</title>" add the following:

```
<meta charset="utf-8">
  <link rel="stylesheet"
href="{{ url_for('static', filename='styles/
app.css') }}">
```

Then open the app.css file in the editor and add the following:

```
html {
  font-family: Arial, Helvetica, sans-serif;
  background-color: gray;
}
```

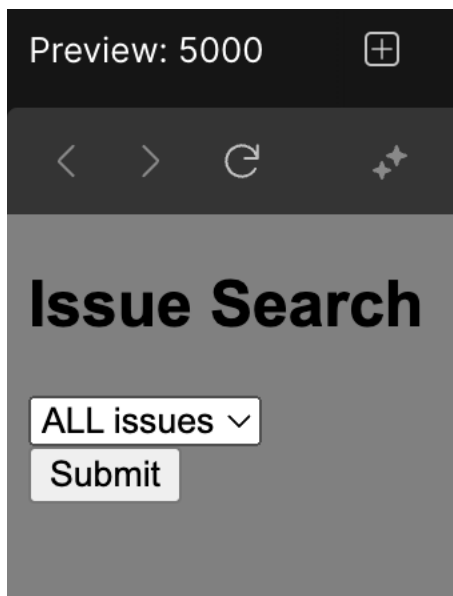
```
#dashboard {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-wrap: wrap;
  width: 80%;
  margin: 0 auto;
  padding: 20%;
}
```

```
.container-full-width {
  display: flex;
  text-align: center;
  height: 100vh;
  width: 90%;
```

```
margin: auto;
}

.centered-frame {
  display: block;
  padding: 1.0rem;
  width: 50vw;
  margin: auto;
  border: 2px solid black;
  background-color: #dde;
  border-radius: 1rem;
}
```

If you save, the background and the heading font should now be different. You might have to restart the Flask server to apply the CSS, even if running in "debug mode":



In our CSS file, we defined several styles.

Styles like "html" apply to all html tags, unless they are overridden by more specific styles.

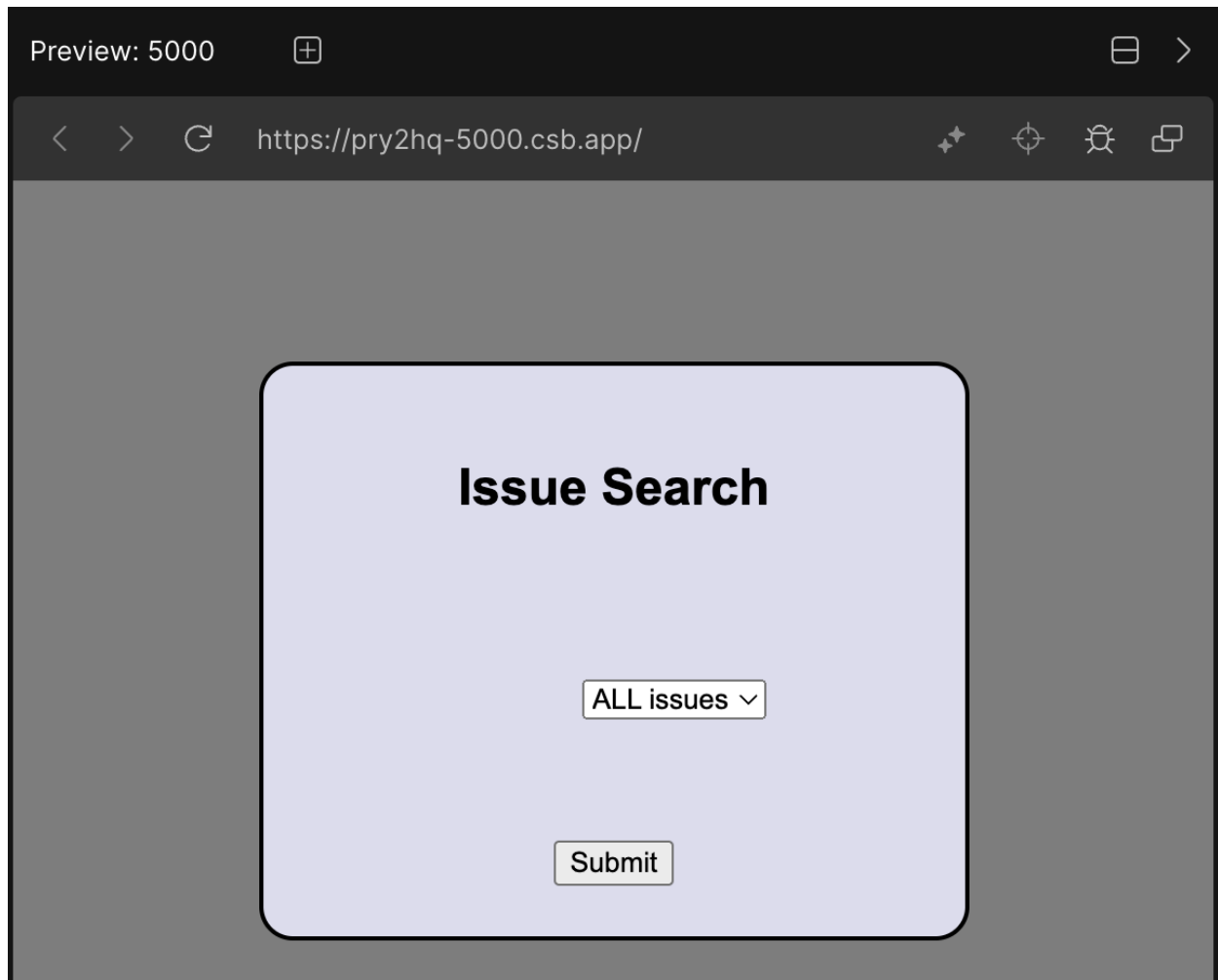
The styles "#dashboard" and ".container-full-width" are using the "#" and "." selectors, which apply styles to an id or class, respectively, that matches the name of the style.

Generally, an id name should only be used on one HTML element tag, but a class name can be used for several elements, if you would like their style to match.

So, to apply the styles we currently have, go back into the app.py file and set the id and/or the class of the tags where you want to apply that style:

```
<body>
  <div class="container-full-width">
    <div class="centered-frame">
      <h2>Issue Search</h2>
      <form>
        <div id="dashboard">
          <div class="dropdown">
            <label
for="issueTypeSelect" />
            <select
name="issueTypeSelect" id="issueTypeSelect">
              <option>ALL issues</
option>
              <option>Epic</option>
              <option>Story</option>
              <option>Bug</option>
            </select>
          </div>
        </div>
        <div id="button_row">
          <button
id="submit_epic_button">Submit</button>
        </div>
      </form>
    </div>
  </div>
</body>
```

If you save and reload or restart, the page should now look like this:



Much better, but things are still a little off.

You might have noticed that we gave the button an id of "submit_epic_button," and its parent "div" was named "button_row," but we don't have styles for those ids. But, we can now add those, and can also add styles that apply to all "label," "button," and "select" elements in the HTML document:

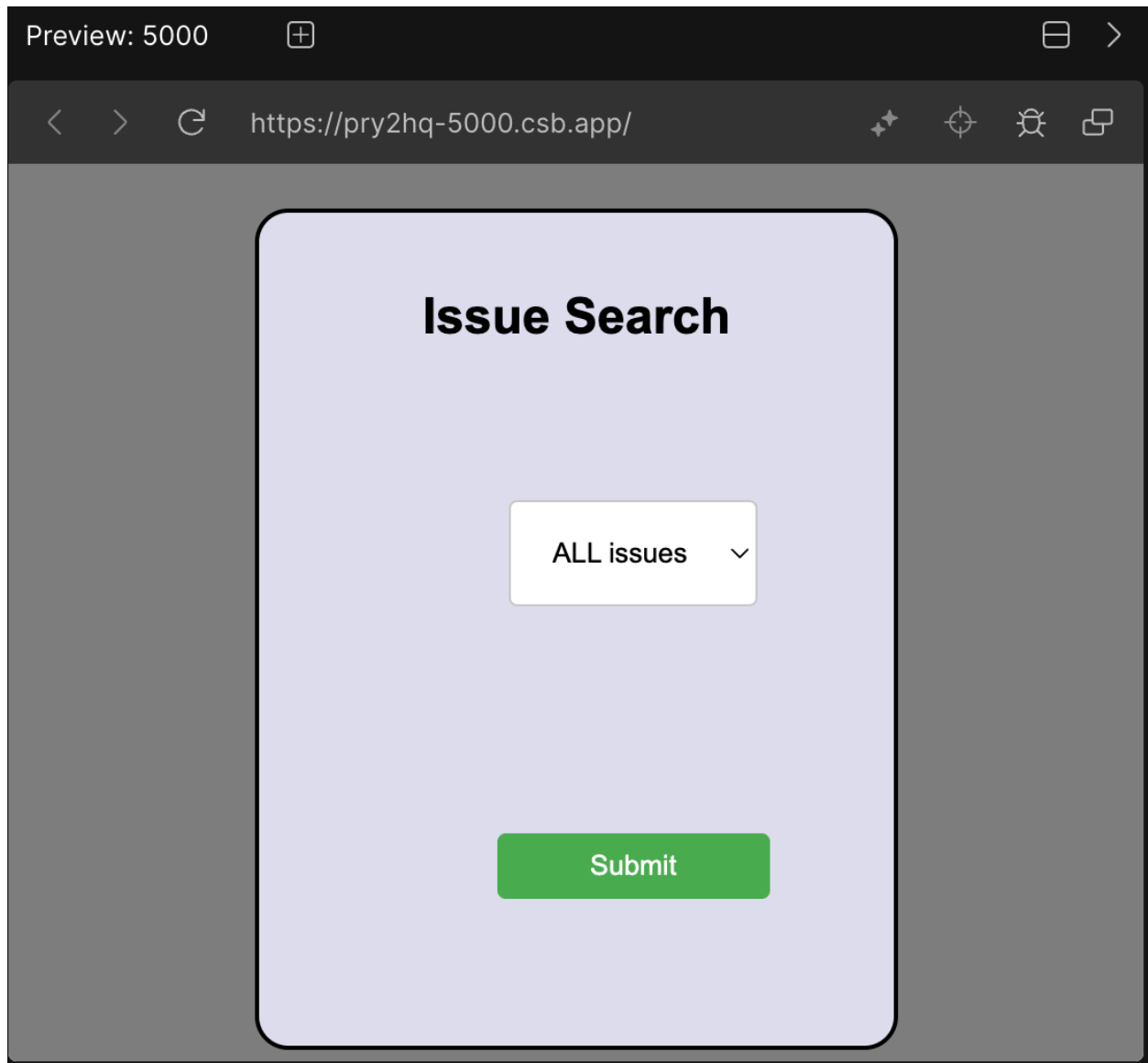
```
label {  
  font-weight: bold;  
  margin-bottom: 1rem;  
}
```

```
select {
```

```
padding: 1rem;  
border: 1px solid #ccc;  
border-radius: 0.25rem;  
}
```

```
button {  
  padding: 0.5rem 1rem;  
  border: none;  
  border-radius: 0.25rem;  
  background-color: #4caf50;  
  color: white;  
  cursor: pointer;  
  width: 60%;  
}
```

```
#button_row {  
  display: flex;  
  justify-content: center;  
  align-items: center;  
  flex-wrap: wrap;  
  width: 80%;  
  margin: 0 auto;  
  padding: 20%;  
}
```




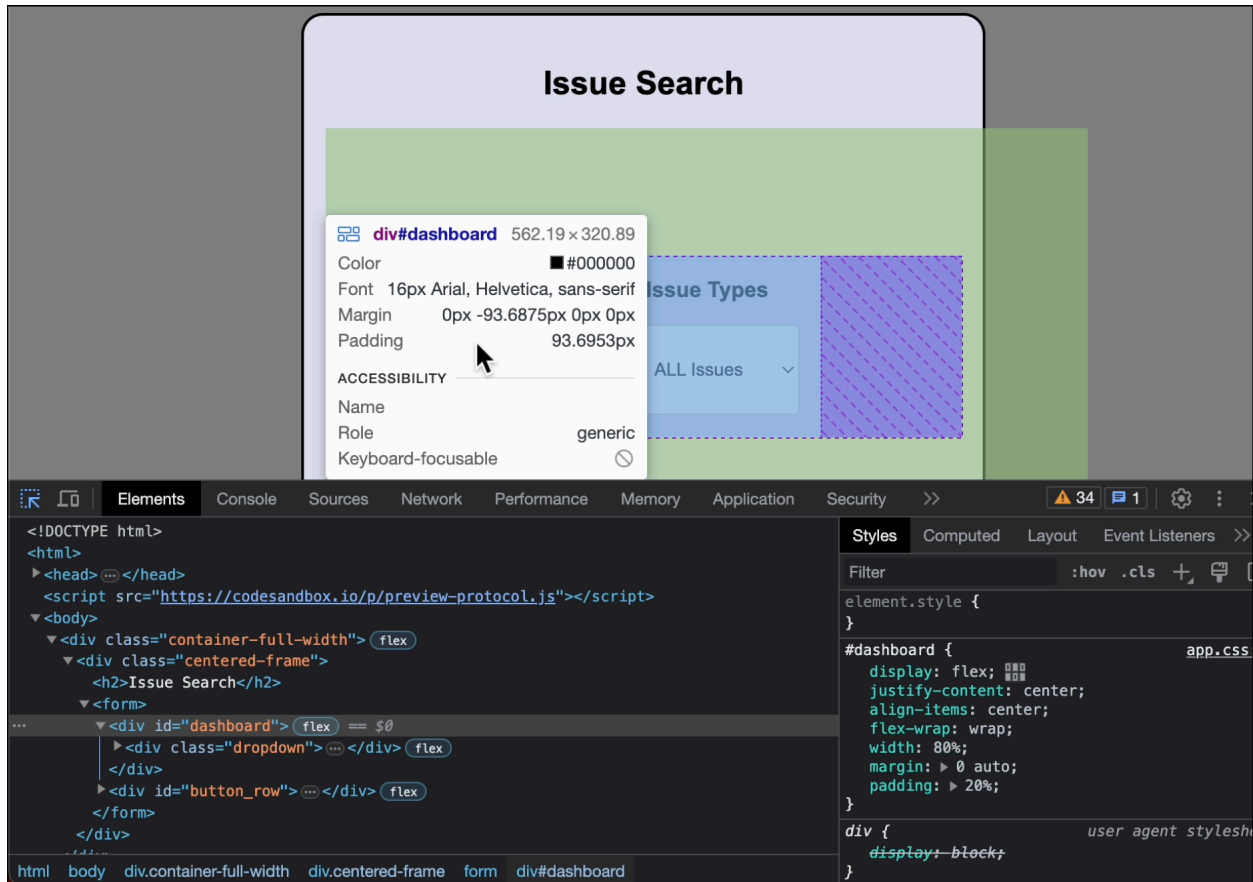
Tweaking CSS using Chrome's Developer Tools

CSS styling can be tricky, especially if you are using a GUI framework or library like Bootstrap. If you are dealing with multiple styles from different source files and libraries, located in different places throughout your project, and you see an element that isn't displaying as you expected it to, sometimes you need a little help to track down what's going wrong, and where.

This is where your browser's development tools are useful. Chrome and Chromium-based browsers have a nice set of tools to help.

To open the developer tool panel, click the "three dots" icon in the upper-left, just under the "X" icon that closes the window. Then hover over "More Tools..." and click "Developer Tools" in the pop-out panel. (Or just hit <CTRL+SHIFT+I> on PC, or <CMD+OPTION+i> on Mac.

On the developer tools panel, you can click the  icon on the left side of the top bar, to allow you to inspect an individual HTML element. Then, when you click on an element that you want to inspect, it will highlight it in the "Elements" tab in the same panel:



So in the above screenshot, I am inspecting the div with the "dashboard" ID. The elements panel shows where it is located in the tag hierarchy. You can expand and collapse nested tags (the div below with the "dropdown" class has been collapsed).

On the right, the "Styles" tab shows the "#dashboard" style that we defined earlier, because that element is using the style.

On the Styles panel, you can change any of the CSS values that are used, and see how those changes affect the live site. You can also see what turning off individual styles would do to the display-- if you roll your mouse over one of the CSS attributes, a checked box will appear, and you can click it to uncheck it, which will strikethrough that attribute, and deactivate that style attribute in the live site.

After experimenting in this way, I decided to remove the "margin" and "padding" attributes from the "#dashboard" style (the margin is how much space you want outside the edge of the element, and padding is space inside the element).

Also, the button is off-center, so on the "#button_row" style, I changed the padding to 0.5rem to center it.

There are several different units that you can use-- including px (pixels), percentages, pt (points-- typically used for typefaces-- there are 72 points in one inch). I typically like to use "rem."

"em" is the width of an em-dash ("--"), which is also the width of a capital letter "M." But "rem" is the width of a "relative em-dash," so the measurements are adjusted relative to the zoom level and resolution / screen size that the user has. This also makes elements more "responsive" when viewed on different devices, and the elements scale together.

After tinkering with the CSS styles, the "select" and ".dropdown" styles now look like this:

```
select {  
  padding: 0.5rem;  
  border: 1px solid #ccc;  
  border-radius: 0.25rem;  
}
```

```
.dropdown {  
  display: flex;  
  flex-direction: column;  
  margin: 0.25rem;  
  padding: 0.25rem;  
}
```

Make Your Page Dynamic by Linking it to the API

Getting the issue types from the API

At the top of the test_app.py file, import the following (which we will create):

```
from interface.get_issue_types import get_issue_types,
strip_issue_types
```

Create a new (failing) test in the test_app.py file called "test_get_issue_types_for_dropdown":

```
def test_get_issue_types_for_dropdown(self):
    issue_types = get_issue_types()
    assert issue_types != None
    assert len(issue_types) > 0

    # we only want the names of the issue types
    issue_names = []
    for issue in issue_types:
        issue_names.append(strip_issue_types(issue))

    # We want to get unique names only
    # casting the list to a set,
    # then back to a list will get rid of
    duplicates
    issue_names = [*set(issue_names)]

    # We will replace this:
    assert issue_names == [
        'unknown'
    ]
```

We hope to get a set of values, but we don't know exactly what will be returned, so just put a single string, 'unknown,' in the expected array.

Now create the source file and function that we want to implement, import, and test:

```
from interface.connect_to_jira import
connect_with_token

def get_issue_types():
    issue_types = []
    jira = connect_with_token()
    issue_types = jira.get_issue_types()
    return issue_types

def strip_issue_types(issue_type):
    return issue_type['name']
```

Now go to the terminal and run the test, but use the "-v" option so that PyTest shows "verbose" output.

```
pytest -v
```

The test will fail, but hopefully we will get a result:

```
E      AssertionError: assert ['Task', 'Epi...btask', 'Bug'] == ['Subtask', '...Task', 'Epic']
E      At index 0 diff: 'Task' != 'Subtask'
E      Full diff:
E      - ['Subtask', 'Story', 'Bug', 'Task', 'Epic']
E      + ['Task', 'Epic', 'Story', 'Subtask', 'Bug']

tests/test_app.py:53: AssertionError

===== short test summary info =====
FAILED tests/test_app.py::TestApp::test_get_issue_types_for_dropdown - AssertionError: assert ['Task', 'Epi...btask', 'Bug'] == ['Subtask', '...Task', 'Epic']
===== 1 failed, 2 passed in 0.58s =====
```

So now we can see what issue types the API is returning. Copy and paste the values in alphabetical order, and sort the result before the comparison to make sure that the order matches:

```
# Names are in different order
# Want to sort before comparison
issue_names.sort()
```

```
assert issue_names == [  
    'Bug',  
    'Epic',  
    'Story',  
    'Subtask',  
    'Task',  
]
```

Now the test should pass, and will continue to pass until the Jira admin adds or removes an issue type. (Generally this does not happen often, once the project is set up in Jira.)

But, we aren't quite done-- we want users to have an option for the search to return "ALL" types of issues, i.e. they don't want to filter on issues. We'd like for this option to appear first, and be the default selection.

```
# Names are in different order  
# Want to sort before comparison  
issue_names.sort()  
assert issue_names == [  
    'ALL',  
    'Bug',  
    'Epic',  
    'Story',  
    'Subtask',  
    'Task',  
]
```

Now we will write a function in the `get_issue_types.py` source file called "get_issue_type_names_for_dropdown." This function will leverage `get_issue_types`, that we wrote before, and it will prepend "ALL" to the issue names returned by the Jira API Server. We will also add the move the lines that filter duplicates from our test case to this new function.

Here is the simplified test, with the check for "ALL" in the first position of the

array:

```
def test_get_issue_types_for_dropdown(self):
    issue_names = get_issue_type_names_for_dropdown()

    expected_issue_names = [
        'ALL',
        'Bug',
        'Epic',
        'Story',
        'Subtask',
        'Task',
    ]

    assert issue_names == expected_issue_names
```

Here is the new function that is called, which calls the `get_issue_types` function, takes the results, removes duplicates, sorts the unique names, and then returns them in an array, along with the "ALL" option in the first position.

```
def get_issue_type_names_for_dropdown():
    issue_types = get_issue_types()
    issue_names = []
    for issue in issue_types:
        issue_names.append(strip_issue_types(issue))
    # Want to get unique names only
    issue_names = [*set(issue_names)]
    issue_names.sort()
    # prepend ALL to the list
    return ['ALL', *issue_names]
```

The new test should pass. Now we will use these values to populate the select dropdown.

Using our new function to dynamically populate a search options dropdown

To populate the "select" HTML element, we will add some simple JavaScript

to our HTML index template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Issue Search</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="{{ url_for('static',
filename='styles/app.css') }}">
    <!-- ADD the "script" open and close tags below: -->
    <script type="text/javascript">

  </script>
</head>
```

To get the data from the API to the HTML document via Python, we need to call our `get_issue_type_names_for_dropdown` function from the Flask server, and pass it the data we get back to Flask's `render_template` function via a second argument:

```
import json
from flask import Flask, render_template
from interface.get_issue_types import
get_issue_type_names_for_dropdown
app = Flask(__name__)

@app.route('/')
def home():
    issue_types = get_issue_type_names_for_dropdown()
    print('issue_types', issue_types)
    return render_template(
        "index.html",
        issue_types=issue_types,
    )

if __name__ == "__main__":
    app.run(debug=True)
```

Now we can replace the hard-coded options from within our select tag, and replace it with another JavaScript snippet, which will send the array from our

issue_types variable to a function we are about to write:

```

        <div class="dropdown">
            <label for="issueTypeSelect">Issue
Types</label>
            <select name="issueTypeSelect"
id="issueTypeSelect">
                <script type="text/javascript">
                    populateSelect(
                        {{ issue_types | safe }},
                        '#issueTypeSelect'
                    );
                </script>
            </select>
        </div>

```

Inside the first script tag (in the "head" section of the HTML document, we will add the **populateSelect** function:

```

function populateSelect(elementArray, selectId) {
    var selector = document.querySelector(selectId);
    var htmlString = '';
    for (var i = 0; i < elementArray.length; i += 1)
    {
        htmlString = htmlString +
        '<option>'+elementArray[i]+'</option>';
    }
    selector.innerHTML = htmlString;
}

```

This **populateSelect** function takes the elementArray takes two arguments:

1. **elementArray**-- this is "issuetypes" and will come from our **get_issue_types_for_dropdown** function. The double-braces surrounding the variable denotes that it is a Jinja template. The "| safe" tells Jinja that the special characters in the strings should not be treated as "escape characters," (i.e. it should be interpreted as plain text only).

2. **selectId** -- this is the id that we named the select element, which in this case is "#issueTypeSelect" (the "#" character just tells the application that the name is an ID. If it was a class name, it would be prefixed with a "." character)

Select an Issue Type and use it in an Issue Query.

Next we want to allow the user to use the issue type selection to query the API and retrieve a set of Jira "issues" that match the "issue type" (e.g. Epic, Story, Bug, etc.).

First, we want to create a new route, "/results" that also takes a URL argument afterward-- the name of the issue type that you want to filter on, e.g.:

`https://<host_name>:5000/results/bug/`

Where "bug" is the name of the issue type to filter.

```
@app.route("/results/<issue_type>")
def results(issue_type=None):
    """Creates endpoint for showing issues from search in HTML
    table format
    or, if do_export is True, creates spreadsheet and
    returns it
    """
    # results_table = build_html_table(issuetype)
    results_table = []
    return render_template(
        "results.html",
        results_table=results_table
    )
```

Build a new HTML template for results

Create a directory called **output**, and create a file called **create_html.py**, and also an empty file called **__init__.py**.

Create a new test source file called **test_create_html.py** in the same **tests** folder where the previous test file is.

Create another file in the templates folder called **results.html**. For now, all this template will do is display an HTML table that we send, using the **results_table** Jinja variable:

```
<!DOCTYPE HTML>
<html>
  <head>
    <link rel="stylesheet" href={{ url_for('static',
filename='styles/app.css') }}>
  <body>
    <div class="results_container">
      {{ results_table | safe }}
    </div>
  </body>
</html>
```

Now, we will create a helper function that we can use to send results in an two-dimesional array structure. Each row in the table will be in an inner array, and then there will be an outer array of array rows, something like the following:

```
[
  ['row1 data', '123', '456'],
  ['row2 data', '789', '10-11-12'],
]
```

Add an import statement that will point to **create_html** that data as a variable at the top of the **test_create_html.py** file:

```
from output.create_html import create_table_body_string
```

Next, we can create a new test class with driver data and an expected result,

and then a simple two-dimensional array like the one above. We will then write a function that should iterate through each cell in the array and add the HTML tags to make it into an HTML table:

```
from output.create_html import create_table_body_string

driver_array_for_table = [['row1 data', '123', '456'], ['row2
data', '789', '10-11-12']]
expected_table_body_string = '<table><tbody><tr><td>row1 data/>
<td>123</td><td>456</td></tr><tr><td>row2 data</td><td>789/>
<td>10-11-12</td></tr>'

class TestCreateHtml:
    def test_create_table_body_string(self):
        table_body_string =
create_table_body_string(driver_array_for_table)
        assert table_body_string == expected_table_body_string
```

Now go to the **create_html.py** file and create a stub function that we will build upon:

```
def create_table_body_string(table_data):
    body_string = ''
    return body_string
```

Run the **test_create_html** test case, and the test should fail. So we will fix it.

For this function, we need to write two loops, the first to go through each "row" (outer array) in the two-dimensional array, and inside that loop, we will write a similar loop to go through each cell in the row. The function will wrap the rows in `<tr> ... </tr>` HTML tags, and each data items in the row with `<td> ... </td>` tags (`<tr>` stands for "table row," and `<td>` is "table data"). The table also needs to be wrapped in `<table><tbody> .. </tbody></table>` tags.

(Later we will add a `<thead> ... </thead>` section above the table body for column headings)

Let's start with the outer **table** tag, and a **tr** tag for each row:

```
def create_table_body_string(table_data):  
    body_string = '<table><tbody>'
```



```
    for row in table_data:  
        body_string += '<tr>'
```



```
        # TODO: Write second loop to handle cells in each row
```



```
        body_string += '</tr>'
```



```
    body_string += '</table></tbody>'
```



```
    return body_string
```

Now replace the "TODO" comment with the following:

```
    for cell in row:  
        body_string += '<td>' + cell + '</td>'
```

So the final function will look like this:

```
def create_table_body_string(table_data):  
    body_string = '<table><tbody>'
```



```
    for row in table_data:  
        body_string += '<tr>'
```



```
        for cell in row:  
            body_string += '<td>' + cell + '</td>'
```



```
        body_string += '</tr>'
```



```
    body_string += '</tbody></table>'
```



```
    return body_string
```

The loops use Python's "for ... in" syntax. The first loop, "for row in table_data" takes the **table_data** two-dimensional array, and pulls out each inner array, calling it "row". Then "for cell in row" takes in each **row** and pulls out each element of that array, calling it **cell**.

Notice that the first assignment to the **body_string** variable used the standard "=" operator, but then we use the "+=" to append the string, so that the "<table><body>" prefix is not overwritten.

Then, within the "for cell in row" loop, the "+" (concatenate) operator is used to combine the variable string held in **cell** inside of an open and close **td** tag.

With HTML tags, you want to close them in the reverse order that you open them, e.g. "<table><tbody>" opens the table, then opens the table's body, which is nested inside the table tag. But then the table body should be closed first, then the table itself: "</tbody></table>".

Even if you are familiar with HTML, it is easy to miss things like this when writing your function, especially since you are not seeing the HTML in the format you normally would, with the line breaks and whitespace.

Finally, the result of all of that looping and appending is returned from the function.

Now we can run the test again. This time it should pass!

We now have a way to create a (very ordinary) table to send to our results route.

You can see what happens if you use the test data to call our new function in the app.py source file. At the top, import the new function:

```
from output.create_html import create_table_body_string
```

Then, change the "results" route:

```
@app.route("/results/<issue_type>")
def results(issue_type=None):
    """Creates endpoint for showing issues from search in HTML
    table format
```

```

        or, if do_export is True, creates spreadsheet and
returns it
        """
        results_table = create_table_body_string([
            ['row1 data', '123', '456'],
            ['row2 data', '789', '10-11-12'],
        ])

    return render_template(
        "results.html",
        results_table=results_table
    )

```

Run the Flask application and then enter the URL: "http://127.0.0.1:5000/results/all"

Right now we are not using the `<issue_type>` variable ("all"), but the route expects you to put something there.

What you should see on the screen is our test data laid out in table format, but since there is no styling, there are no borders on the table, so our values are just floating in the corner. At least they are lined up in rows and columns, though, which is progress!



```

row1 data 123 456
row2 data 789 10-11-12

```

Now we will want to add functionality to bring in the data from the API, then pull out the specific data points we are interested in. We will also add column headers and styling, to make the table readable.

Getting a List of Issues from Jira

Now we will write a Jira query to get all of the issues. In the **interface** folder, create a new python source file called **get_issues.py** and add the following code:

```
from interface.connect_to_jira import connect_with_token

def get_issues(issue_type='ALL'):
    issues = []
    jira = connect_with_token()
    linked_issues = jira.jql('')
    return linked_issues

if __name__ == "__main__":
    results = get_issues()
    print('sample_issues = ' + str(results))
```

We gave this file a "main" routine so that we can call it directly, and have it spit out the results. We want to write this to a file as we did before, so we are also assigning it to a variable called **sample_issues** .

Because the API returns the results as a Python Dictionary data type, the 'str()' function casts the dictionary to a string, which can be redirected to a file:

```
python interface/get_issues.py > ./tests/sampleddata/
sample_issues.py
```

Now open up the file. The output is difficult to read, because there are none of the line breaks or other white space that we normally use to organize the source-- The entire data set is on a single line.

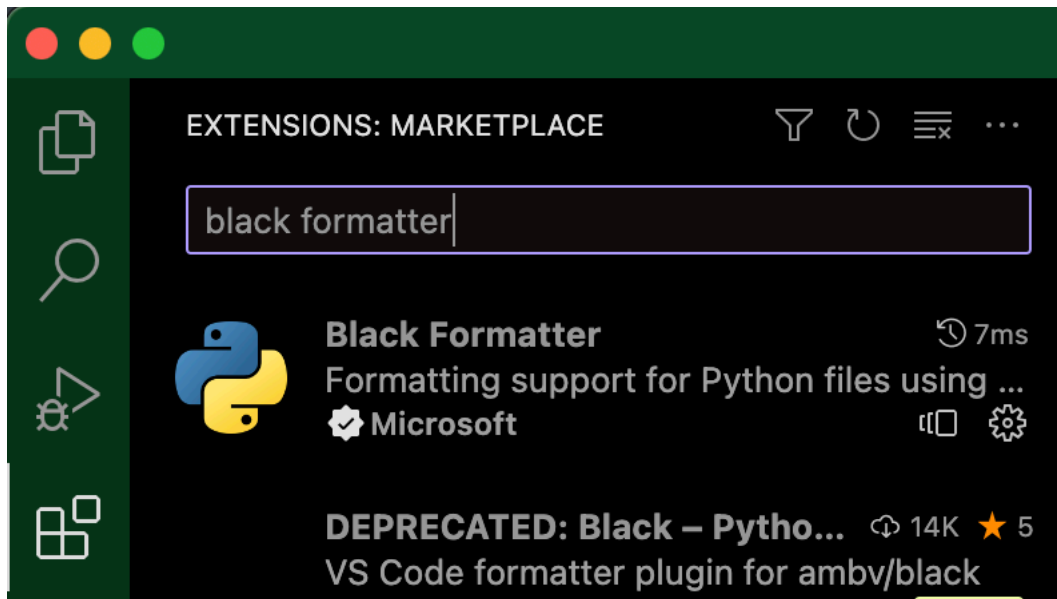

```

1 sample_issues = {'expand': 'schema,names', 'startAt': 0, 'maxResults': 50,
  'total': 10, 'issues': [{ 'expand': 'operations,versionedRepresentations,
    editmeta,changelog,customfield_10010.requestTypePractice,renderedFields',
    'id': '10005', 'self': 'https://ricksegregest.atlassian.net/rest/api/2/issue/
    10005', 'key': 'SP-5', 'fields': { 'statuscategorychangedate':
    '2023-07-08T13:19:28.845-0500', 'issuetype': { 'self': 'https://ricksegregest.
    atlassian.net/rest/api/2/issuetype/10001', 'id': '10001', 'description':
    'Stories track functionality or features expressed as user goals.',
    'iconUrl': 'https://ricksegregest.atlassian.net/rest/api/2/universal_avatar/
    view/type/issuetype/avatar/10315?size=medium', 'name': 'Story', 'subtask':
    False, 'avatarId': 10315, 'entityId':
    'a95fdbca-67d9-4388-863c-d7c76b4c3b19', 'hierarchyLevel': 0}, 'timespent':
    None, 'customfield_10030': None, 'customfield_10031': None, 'project':
    { 'self': 'https://ricksegregest.atlassian.net/rest/api/2/project/10000', 'id':
    '10000', 'key': 'SP', 'name': 'scrum-project', 'projectTypeKey': 'software',
    'simplified': True, 'avatarUrls': { '48x48': 'https://ricksegregest.atlassian.
    net/rest/api/2/universal_avatar/view/type/project/avatar/10407', '24x24':
    'https://ricksegregest.atlassian.net/rest/api/2/universal_avatar/view/type/
    project/avatar/10407?size=small', '16x16': 'https://ricksegregest.atlassian.
    net/rest/api/2/universal_avatar/view/type/project/avatar/10407?size=xsmall',
    '32x32': 'https://ricksegregest.atlassian.net/rest/api/2/universal_avatar/view/
    type/project/avatar/10407?size=medium'}}, 'customfield_10032': [],
    'customfield_10033': None, 'fixVersions': [], 'customfield_10034': None,

```

If you are using VSCode, you can fix this by using a formatter extension called "Black Formatter."

If you don't already have this extension installed, you can install it through the Extensions marketplace -- click the icon that looks like the fourth one down, then search for "Black Formatter" and install the current version (published by Microsoft).



Now you can go back to the sample_issues.py code file, then right-click inside the editor window, and select "Format Document" in the context menu. If it asks you to set a default formatter (at the top), you can choose "Black Formatter." If it doesn't format the code correctly, you might try selecting "Format Document With..." in the context menu, and then choose "Black Formatter." You should then see something like this:

```

1 sample_issues = {
2     "expand": "schema,names",
3     "startAt": 0,
4     "maxResults": 50,
5     "total": 10,
6     "issues": [
7         {
8             "expand": "operations,versionedRepresentations,editmeta,changelog",
9             "id": "10005",
10            "self": "https://ricksegregest.atlassian.net/rest/api/2/issue/10005",
11            "key": "SP-5",
12            "fields": {
13                "statuscategorychangedate": "2023-07-08T13:19:28.845-0500",
14                "issuetype": {
15                    "self": "https://ricksegregest.atlassian.net/rest/api/2/issue/10001",
16                    "id": "10001",
17                    "description": "Stories track functionality or features expected in a future release",
18                    "iconUrl": "https://ricksegregest.atlassian.net/rest/api/2/issue/10001/icon.png",
19                    "name": "Story",
20                    "subtask": False,
21                    "avatarId": 10315,
22                    "entityId": "a95fdbca-67d9-4388-863c-d7c76b4c3b19",
23                    "hierarchyLevel": 0

```

Now let's add a main class to this file, so we can pull out the first issue from the issues list, and later use that to drive some of the functions we are going to write:

```

if __name__ == "__main__":
    issues = sample_issues['issues'][0]
    print('first_sample_issue = ' + str(issues))

```

Now run this file, and redirect the output to another new source file:

```
python tests/sampleddata/sample_issues.py > tests/sampleddata/first_sample_issue.py
```

Open up this new file, and format it as you did before. Then it should look something like this:

```
1 first_sample_issue = {
2     "expand": "operations,versionedRepresentations,editmeta,changelog,customf:
3     "id": "10005",
4     "self": "https://ricksegregest.atlassian.net/rest/api/2/issue/10005",
5     "key": "SP-5",
6     "fields": {
7         "statuscategorychangedate": "2023-07-08T13:19:28.845-0500",
8         "issuetype": {
9             "self": "https://ricksegregest.atlassian.net/rest/api/2/issuetype/10
10             "id": "10001",
11             "description": "Stories track functionality or features expressed
12             "iconUrl": "https://ricksegregest.atlassian.net/rest/api/2/universa
13             "name": "Story",
14             "subtask": False,
15             "avatarId": 10315,
16             "entityId": "a95fdbca-67d9-4388-863c-d7c76b4c3b19",
17             "hierarchyLevel": 0,
18         },
19         "timespent": None,
20         "customfield_10030": None,
21         "project": {
22             "self": "https://ricksegregest.atlassian.net/rest/api/2/project/100
23             "id": "10000",
```

This is still a lot of data. There are only a few important data points that we will want to display on the screen in our results table, so next, we need to write functions that will automate pulling those bits of data out of each issue's dictionary object.

When we parsed the config object, we pulled individual values out of a Python dictionary object, by key. We are basically going to do the same thing to the Jira Issue dictionary object.

- Time Created
- Priority
- Issue Type
- Status
- Creator's Name
- Summary

There are plenty of other data points that you may or may not be interested

in, and it is up to you which ones you want to present in your report. You can even get fancy and allow the end users to customize which fields they want to see in their reports!

The next step is to create tests and a new functions, and we can use the first sample issue to drive that function so we know what results to expect. Because each function should only do one thing, we will create functions to pull each individual data point as a string.

There will be another function to collect the strings from those functions and return them as a Python List (array), which will map to a single row in our table.

This function will then be sent each issue object from the API by another function, which will build the List of rows.

If we look closely at the sample issue object, we know what values should be returned when we get the data points we are interested in:

Data	Value we expect	Issue Object Keys
Jira Key (for this issue)	"SP-5"	['key']
Time Created	"2023-07-08T13:19:28.687-0500"	['fields'] ['created']
Priority	"Medium"	['fields'] ['priority']['name']
Issue Type	"Story"	['fields'] ['issuetype'] ['name']
Status	"To Do"	['fields']['status'] ['statusCategory'] ['name']
Creator's Name	"Rick Segrest"	['fields'] ['creator'] ['displayName']
Summary	"Another story"	['fields'] ['summary']

So when we build the row data for this issue object, we expect it to look like this:

```
[
    'SP-5',
    '2023-07-08T13:19:28.687-0500',
    'Medium',
    'Story',
    'To Do',
    'Rick Segrest',
]
```

Using this, we will build a new test case, and then write our function to make it pass. Create a new file in the output directory called **parse_object.py**, and then :

```
from tests.sampledata.first_sample_issue import
first_sample_issue as my_issue
from output.parse_object import get_rowdata_from_issue

class TestParseJson:
    expected_datarow = [
        'SP-5',
        '2023-07-08T13:19:28.687-0500',
        'Medium',
        'Story',
        'To Do',
        'Rick Segrest',
        'Another story'
    ]

    def test_get_rowdata_from_issue(self):
        datarow = get_rowdata_from_issue(my_issue)
        assert datarow == self.expected_datarow
```

Run the failing test, then write the function (in **parse_object.py**) to get the test to pass:

```
def get_rowdata_from_issue(issue_object):  
    jira_key = issue_object['key']  
    time_created = issue_object['fields']['created']  
    priority = issue_object['fields']['priority']['name']  
    issue_type = issue_object['fields']['issuetype']['name']  
    status = issue_object['fields']['status']['statusCategory']  
    ['name']  
    creator = issue_object['fields']['creator']['displayName']  
    summary = issue_object['fields']['summary']  
    return [  
        jira_key,  
        time_created,  
        priority,  
        issue_type,  
        status,  
        creator,  
        summary,  
    ]
```

Sending the issue objects from the list, one-by-one

At the top, import a new function that we will define, and then a new test:

```
from output.parse_object import get_rowdata_from_issue,
build_row_list_from_issue_obj_list

# ... Other code ...
```

```
def test_build_row_list_from_issue_list(self):
    row_list =
build_row_list_from_issue_obj_list(api_response)

    assert row_list[0] == self.expected_datarow
    assert row_list == ''
```

For this test, we are asserting that the result will be equivalent to an empty string, which, of course, it won't.

Write the function build_row_list_from_issue_list in the output/parse_object.py file:

```
def build_row_list_from_issue_obj_list(api_response):
    row_list = []
    issues = api_response['issues']

    for issue in issues:
        row_list.append(get_rowdata_from_issue(issue))

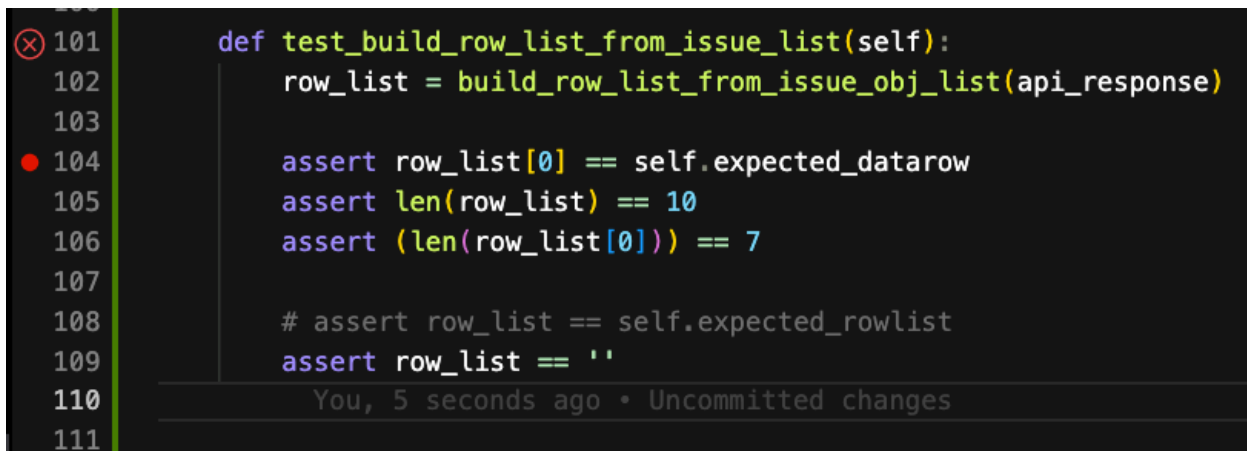
    return row_list
```

Since you are sending the full API response, the function gets the issues list out of that response on the second line-- there is some other metadata there that we don't care about right now.

Debugging Your Unit Tests in VSCode

I want to run this test using the debugger, and set a breakpoint right after the line where the `row_list` variable is set.

To do this, click the mouse to the left of the line number just after the variable is returned from the `build_row_list_from_issue_obj_list` function. This should add a red dot next to that line number (#104 in my source file, but it may be different in yours). This indicates that you have set a break point on line 104.



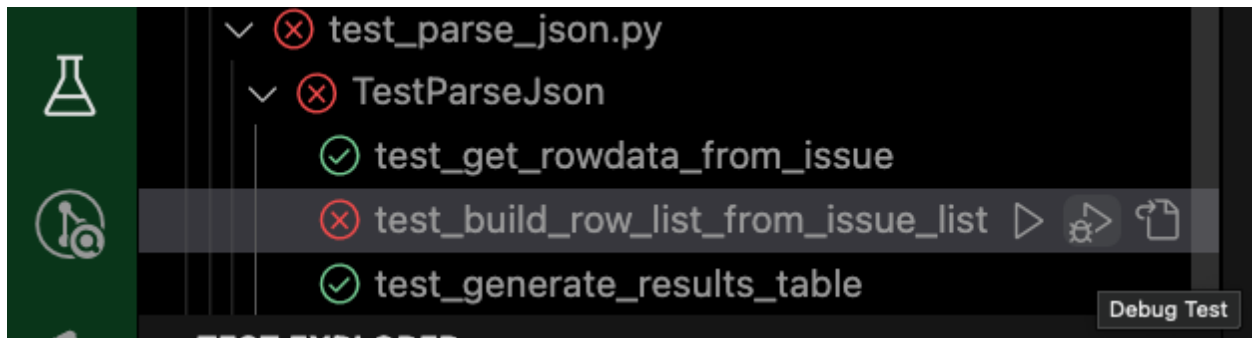
```
101 def test_build_row_list_from_issue_list(self):
102     row_list = build_row_list_from_issue_obj_list(api_response)
103
104     assert row_list[0] == self.expected_datarow
105     assert len(row_list) == 10
106     assert (len(row_list[0])) == 7
107
108     # assert row_list == self.expected_rowlist
109     assert row_list == ''
110
111
```

You, 5 seconds ago • Uncommitted changes

Now run a "Debug Test" on the `test_build_row_list_from_issue_list` test case. When you hover the pointer over the test case name, it will show up as a "play" triangle with a image of an actual bug in the lower-left corner:

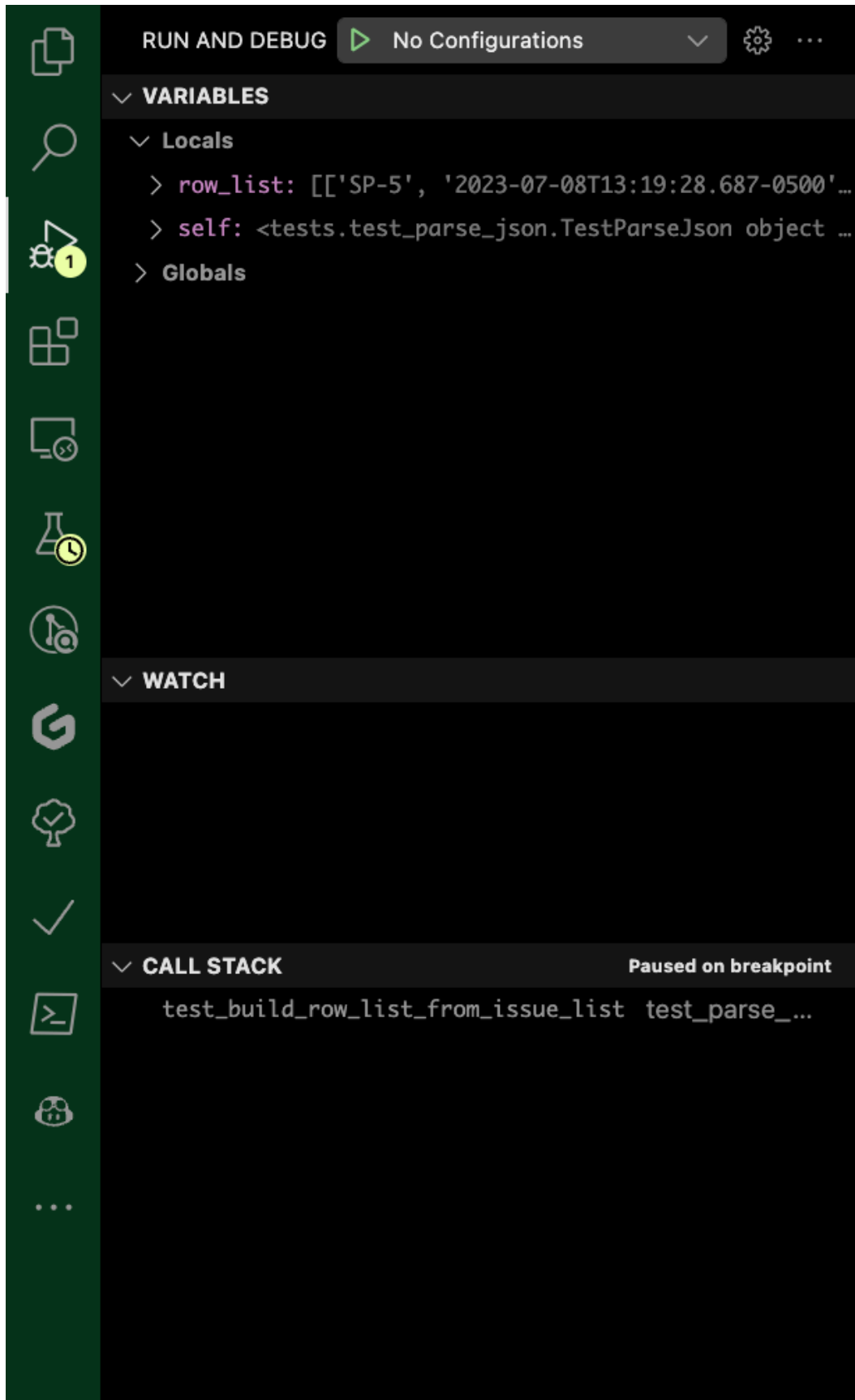


You can see the icon in context below:

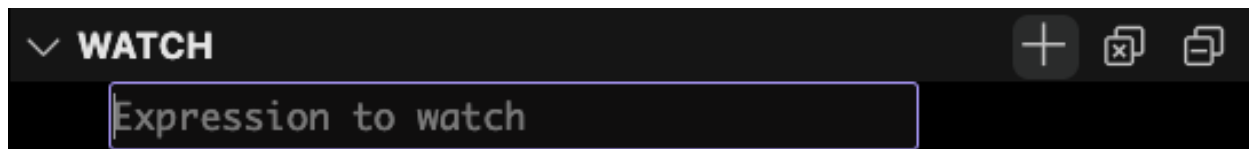


Click on the Debug Test icon. The debugger will start, then run the test, and then halt progress and wait for you when it hits the breakpoint.

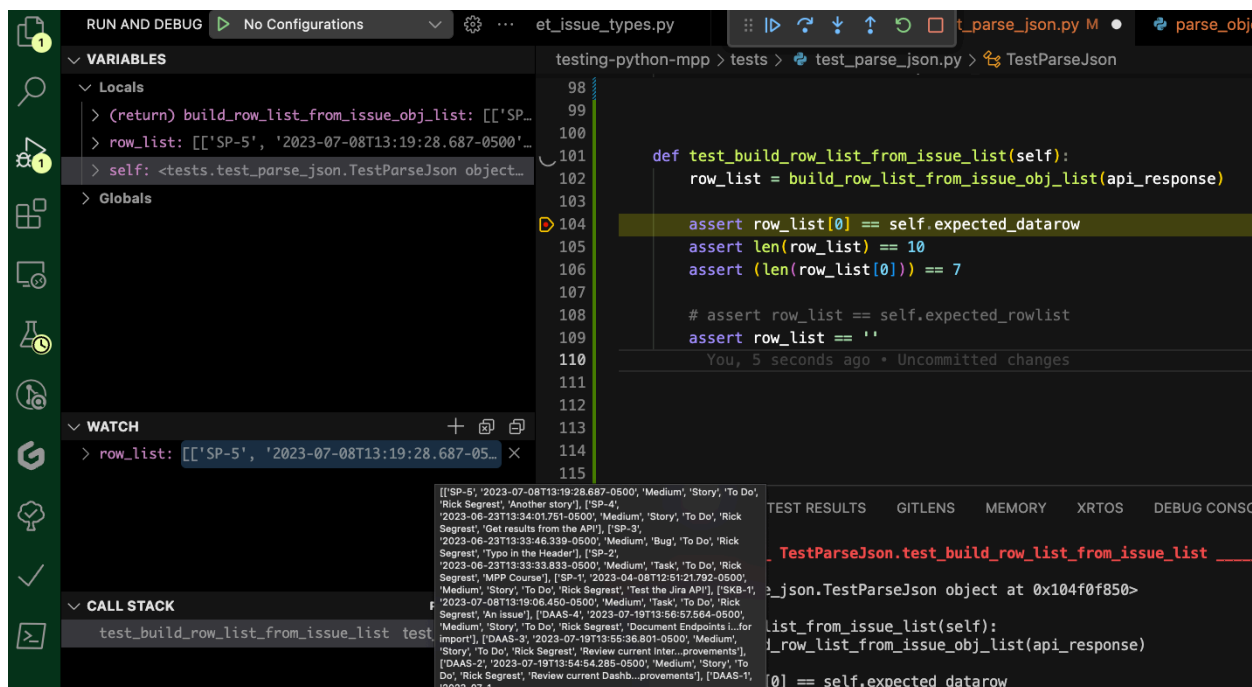
Now we want to ask the debugger to "watch" the row_list variable. To do this, hover over the **WATCH** bar in the **Testing** panel.



A few icons should now be displayed, so click the one that looks like a "+". Then in the field, type row_list -- all lowercase, with the underscore, exactly as it appears in the code (you can also copy and paste to make sure you get longer symbol names correct). Then hit <ENTER>.



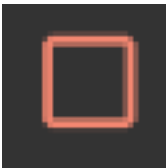
Now the debugger will display the value of row_list when it is assigned. Whenever a variable you watch is changed, that will also be reflected in the watch panel. This can be a powerful technique to understand what is going on in the guts of your program.



Now hover over the value of row_list in the watch panel, and go to the top of the test_parse_json.py file, and assign that to a variable called expected_row_list.

Look closely at the data and make sure it looks like you'd expect. If so, assert that the `row_list` should equal `self.expected_rowlist` (add `self`, since it is a member of the class).

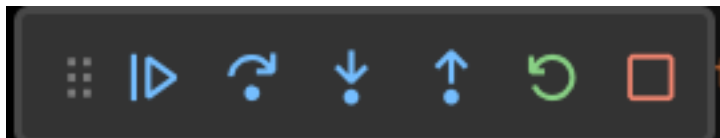
Now you can stop the debugger by clicking the red square in the debugger control strip, which should appear at the top of the screen while the debugger is running, right in the center:



Then just run the test (without debugging), and it should pass.

Debugger controls

The debugger control strip has six buttons:



It's good to know how to use it, as it can be very useful to get yourself out of a pickle-- even experienced programmers make typos and silly mistakes sometimes! It can also be useful for reverse engineering someone else's code, to gain a better understanding of how it works.

Here is a summary of what each of the controls do (from left to right):

1. **Continue/pause**: If the program execution has paused on a breakpoint, this button will prompt the debugger to continue -- to start running the program again. If the program is running, the debugger will pause at the current line it is executing. You can then add watch variables, breakpoints

at specific lines, or create breakpoints based on a comparison. You can also add code, or step through the program execution line-by-line.

2. **Step over**: When the program execution is paused, you can step through the program using this button, but any functions that the debugger encounters will be executed all in one step.
3. **Step into**: Instead of "stepping over," this button will cause the debugger to go into the next function it encounters, and continue to execute line-by-line .
4. **Step out**: This option will execute all of the remaining lines of the function the debugger is currently paused in, and then pause at the next line of the calling function.
5. **Restart**: This will terminate the program execution and start the debugger again from the beginning
6. **Stop**: Stop running the debugger (and the program)

Get the results in table form

We can now send the list of rowdata into the function we wrote to generate an HTML table--we tested it with meaningless data, but now we can see what it generates with real data from Jira.

Add a test case called something like test_generate_results_table(self): and have it call the function create_table_body_string in output/create_html.py. It should

```
def test_generate_results_table(self):  
    row_list =  
build_row_list_from_issue_obj_list(api_response)  
    html_table = create_table_body_string(row_list)  
    assert html_table == ''
```

We can then use the debugger again to see what the function returns. Repeat the same steps as we did in the last test case to watch the variable html_table, analyze it, and add it as an expected_html_table at the top of the test class. Then change the last line:

```
assert html_table == self.expected_html_table
```

Now we can take this generated table, and temporarily copy and paste it into an HTML template. Duplicate the results.html file in the templates folder. Replace the line:

```
{{ results_table | safe }}
```

with the string that we assigned to expected_html_table (and remove the quotes). Your screen should look like this:



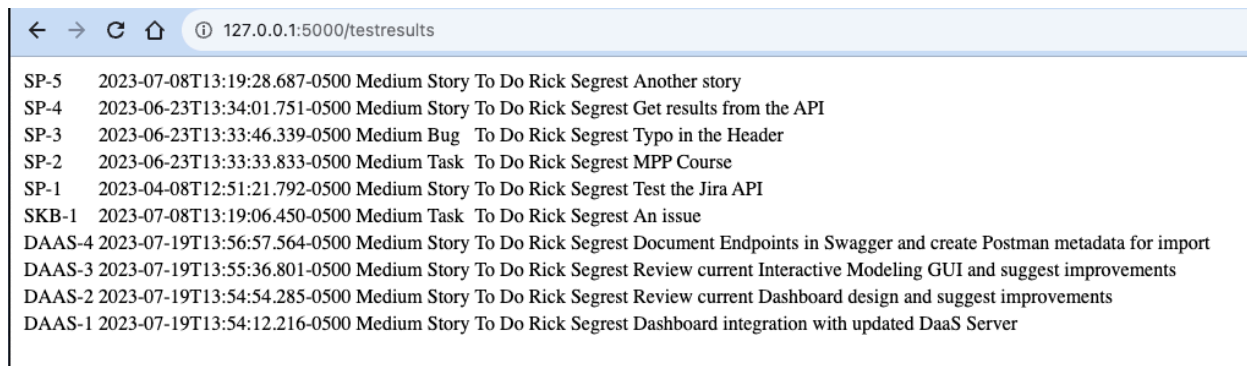
```

1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
5   <body>
6     <div class="results_container">
7       <!-- Copy and paste "expected_html_table" inside this div: -->
8       <table><tbody><tr><td>SP-5</td><td>2023-07-08T13:19:28.687-0500</td><td>Medium</td><td>Story</td></tr></tbody></table>
9     </div>
10  </body>
11 </html>

```

The **table** is all on one line, and will run off the right side of the editor panel, but that is okay.

Now temporarily make a new route, and see if this data that we generated will render:



SP-5	2023-07-08T13:19:28.687-0500	Medium	Story	To Do Rick Segrest Another story
SP-4	2023-06-23T13:34:01.751-0500	Medium	Story	To Do Rick Segrest Get results from the API
SP-3	2023-06-23T13:33:46.339-0500	Medium	Bug	To Do Rick Segrest Typo in the Header
SP-2	2023-06-23T13:33:33.833-0500	Medium	Task	To Do Rick Segrest MPP Course
SP-1	2023-04-08T12:51:21.792-0500	Medium	Story	To Do Rick Segrest Test the Jira API
SKB-1	2023-07-08T13:19:06.450-0500	Medium	Task	To Do Rick Segrest An issue
DAAS-4	2023-07-19T13:56:57.564-0500	Medium	Story	To Do Rick Segrest Document Endpoints in Swagger and create Postman metadata for import
DAAS-3	2023-07-19T13:55:36.801-0500	Medium	Story	To Do Rick Segrest Review current Interactive Modeling GUI and suggest improvements
DAAS-2	2023-07-19T13:54:54.285-0500	Medium	Story	To Do Rick Segrest Review current Dashboard design and suggest improvements
DAAS-1	2023-07-19T13:54:12.216-0500	Medium	Story	To Do Rick Segrest Dashboard integration with updated DaaS Server

So the generated table is rendering correctly, but it has zero styling right now. There are no borders, so all of the data is running together. Color and headings, and a little padding inside the cells will also help make it more readable.

First let's add headings. For now we can hard-code these so we can style them along with the rest of our page, but later we can generate the headings along with the rest of the table. The **<thead>** section goes between the **<table>** and **<tbody>** tags, and each heading is inside a **<th>** tag, like this:

```

<table>
  <thead>

```



```

<th>Jira Key</th>
<th>Created</th>
<th>Priority</th>
<th>Issue Type</th>
<th>Status</th>
<th>Created By</th>
<th>Summary</th>
</thead>
<tbody>

```

The [results.html](#) and [test_results.html](#) files are already pointing to the styles/app.css file, but no styles are defined that apply to our table-- it has no classes or IDs set inside the tags, and there are no styles defined for the tags [table](#), [tr](#), [td](#), etc. Let's define those now, in the [style.css](#) file, giving the table and cells borders. Also, we will center the content in each cell, and center the table on the page. We will give the cells a light colored background, and alternate from a white color to a light-blue color, which will help your eyes follow the row across the screen. This is done with a conditional selector, [tr:nth-child\(even\)](#), which only applies to the even numbered rows in the table.

We will then give the page itself a dark background, using the [html](#) tag:

```

table {
  border: 3px solid black;
  background-color: #def;
  text-align: center;
  margin: 0 auto;
  border-spacing: 0px;
}

th {
  padding: 0.5rem;
  color: white;
  border-left: 1px solid black;
  border-right: 1px solid black;
  outline: none;
  background-color: #840;
  margin: 0px;
  margin-bottom: 2px;
  border-bottom: 2px solid black;
}

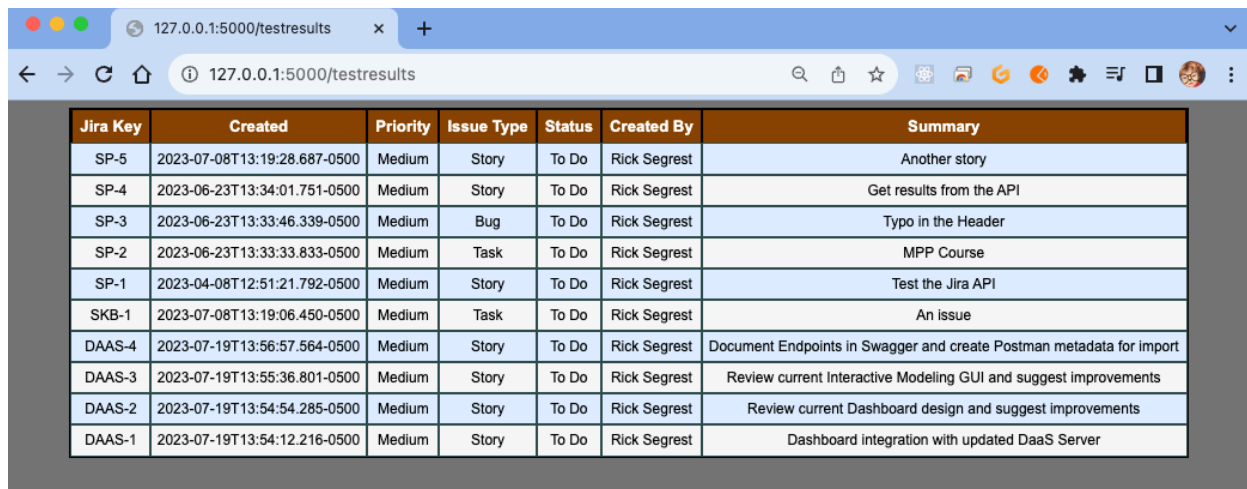
```

```
td {  
  outline: 1px solid darkslategray;  
  padding: 0.5rem;  
}
```

```
tr:nth-child(even) {  
  background-color: whitesmoke;  
}
```

```
html {  
  background-color: #777;  
}
```

Now the table is much more readable



Jira Key	Created	Priority	Issue Type	Status	Created By	Summary
SP-5	2023-07-08T13:19:28.687-0500	Medium	Story	To Do	Rick Segrest	Another story
SP-4	2023-06-23T13:34:01.751-0500	Medium	Story	To Do	Rick Segrest	Get results from the API
SP-3	2023-06-23T13:33:46.339-0500	Medium	Bug	To Do	Rick Segrest	Typo in the Header
SP-2	2023-06-23T13:33:33.833-0500	Medium	Task	To Do	Rick Segrest	MPP Course
SP-1	2023-04-08T12:51:21.792-0500	Medium	Story	To Do	Rick Segrest	Test the Jira API
SKB-1	2023-07-08T13:19:06.450-0500	Medium	Task	To Do	Rick Segrest	An issue
DAAS-4	2023-07-19T13:56:57.564-0500	Medium	Story	To Do	Rick Segrest	Document Endpoints in Swagger and create Postman metadata for import
DAAS-3	2023-07-19T13:55:36.801-0500	Medium	Story	To Do	Rick Segrest	Review current Interactive Modeling GUI and suggest improvements
DAAS-2	2023-07-19T13:54:54.285-0500	Medium	Story	To Do	Rick Segrest	Review current Dashboard design and suggest improvements
DAAS-1	2023-07-19T13:54:12.216-0500	Medium	Story	To Do	Rick Segrest	Dashboard integration with updated DaaS Server

Generate Headings

Since we hard-coded the headings, we want to create a function to generate them automatically.

Copy and paste the hard-coded `<thead>` tag (and what's inside) into the `test_create_html.py`, and eliminate the line breaks and white space. Put quotes around it and assign it to a variable called `expected_headings_string`, and also create one called `expected_table_string` that add the expected `<thead>` tags between the `<table>` and `<tbody>` tags.

We'll also move all of our expected inputs and outputs into a file inside a `test/sampled_data/` folder called `driver_data.py` (don't call it "test_data," otherwise PyTest will think it is a test source file. Then move the variables we need for the tests into that file, and import them into any test cases that need access to them:

```
from tests.sampledata.driver_data import expected_rowlist,
expected_html_table, expected_table_body_string,
expected_table_headings, expected_table_string, headings_list,
expected_datarow
```

Then create a `test_create_headings_string` and a `test_create_table` test case, that will call the `create_headings` and `create_table` function that we will write in `create_html.py`. This is simple compared to what we've done in the past-- write the `create_headings` class to accept a list of headings as strings and generate the `<thead>` section of the table. In this case the heading list would be:

```
headings_list = ['Jira Key', 'Created', 'Priority', 'Issue
Type', 'Status', 'Created By', 'Summary']
```

Since the `create_table` function will now generate the `table` tags and call the two other functions to generate the `thead` and `tbody` sections, remove the open and close `table` tags from the `expected_table_body_string`.

Since this is similar to what we have done in the past, try it yourself, or refer to the source code repo.

Dynamically Generate Results

We have been looking at the **test results** route, where the HTML is hard-coded, but we now have all of the pieces in place to dynamically generate a results page. So let's do that next.

In `app.py`, at the top block of import statements, add an import for the **get_issues** function from **interface/get_issues**:

```
from interface.get_issues import get_issues
```

Go back to the **results/<issue_type>** route. In the function definition, set the default value to "ALL." Import **get_issues** from the interface module

```
@app.route("/results/<issue_type>")
def results(issue_type="ALL"):
    """Creates endpoint for showing issues from search in HTML
    table format
    or, if do_export is True, creates spreadsheet and
    returns it
    """
    api_response = get_issues(issue_type)
    headings_list = ['Jira Key', 'Created', 'Priority', 'Issue
    Type', 'Status', 'Created By', 'Summary']
    results_table = create_table(api_response, headings_list)

    return render_template(
        "results.html",
        results_table=results_table
    )
```

Try navigating to the URL **<127.0.0.1:5000/results/ALL>** and check whether you see a table with Jira results in it -- it should look very much like the one we were working with on the **test results** route.

If it does, you can delete the **test_results** route from **app.py**. Next, we will use the value from the drop-down selection to display only issues that match the

Issue Type we select.

When we pick an issue type from the dropdown, we need a way to use that value. Then we want to use that value in the query, or filter out any issues that don't match.

You can do this either by formatting the query you send to the Jira API, or you can get all of the results and use the Python "filter" function to do the filtering.

Below, the query is empty (returning all issues), if the **issue_type** is equal to 'ALL,' but if the **issue_type** is set to any other value, it will build a query where to assert that the issue's "type" should be equal to the string value we send to the function's argument.

(Confusingly, since the query is written in a different language called **JQL**-- Jira Query Language, the equivalence test operator is a single equal sign (=), where as in Python and JavaScript, the equivalence test operator is a double equal sign (==), because the single equal is to assign a value to a variable.)

```
def get_issues(issue_type='ALL'):  
    issues = []  
    jira = connect_with_token()  
    jql_request = ''  
    if issue_type != 'ALL':  
        jql_request += "type = '"+issue_type+'"  
  
    issues = jira.jql(jql_request)  
    return issues
```

This function uses a different method to come up with the same result, by looping through the object list and testing to see if the specific field matches the **issue_type** argument.

```
def filter_by_issue_type_using_loop(issue_type='ALL',
issue_object_list=[]):
    filtered_list = []
    if issue_type != 'ALL':
        for issue in issue_object_list:
            if issue['fields']['issuetype']['name'] ==
issue_type:
                filtered_list.append(issue)
    else:
        filtered_list = issue_object_list
    return filtered_list
```

Finally, this function uses Python's **filter** function. **Filter** takes two arguments, the list of items to filter, and then a function to define how to filter the list. While you can define the function elsewhere and pass the name into **filter**, frequently you will see an anonymous function passed in using the lambda syntax, as seen below:

```
def apply_filter_func_to_issue_list(issue_type='ALL',
issue_object_list=[]):
    if (issue_type == 'ALL'):
        return issue_object_list
    return list(filter(lambda issue: issue['fields']
['issuetype']['name'] == issue_type, issue_object_list))
```

The lambda function assigns the objects from the list to the **issue** variable, one at a time, and then performs the test to see if that issue's fields->issuetype->name data matches the **issue_type** we passed as an argument, then will decide whether to filter that object out of the list, depending on whether that comparison returns **True** or **False**.

The **filter** function returns a data structure of type **filter**, but we cast the result back to a list as we are returning it.

You don't need all three of these, so you can choose one to handle the filtering, depending on your preference. However, queries to a remote API can be slow, so getting all of the issues and then filtering them using Python may prevent having to make and process multiple queries.

Making the Time Stamps Readable

One thing you might have noticed in the table is that the timestamps are difficult to read:

2023-07-08T13:19:28.687-0500

This is a string that follows the ISO time format. It can be cast to a Python datetime format, and then you can reformat it in any way that you want. Here is an example:

```
def format_iso_datetime_string(dt_str):  
    dt = datetime.fromisoformat(dt_str)  
    return dt.strftime("%Y-%m-%d %I:%M:%S %p")
```

...

More Improvement Ideas

There are still plenty of improvements to be made. For example:

- Click headers to sort rows alphabetically based on a specific data parameter, or by time if the user clicks on **Created By**.
- Color code cells red or green based on whether data is above or below a specific threshold
- Add an **Export** button to download the data in CSV, Excel, Word, or PDF format
- Add additional filter parameters as dropdown options
- Add a search feature
- Chart and/or Graph metrics from data

Click header to sort

Add an **onclick** handler to each of the **<th>** tags to trigger a sort function--you can run the data through Python's **sort** function on the list of issue objects, and write a comparison operator based on which field the user wants to sort on.

Export to CSV

CSV is a very simple format that can be opened in any Spreadsheet application (like Excel). It is a string format, and each cell is separated by a comma, while rows are separated by a line break. You can format the data into CSV in a similar manner to how we generated the HTML, or you can use a library like Python's built-in **csv** library:

<https://docs.python.org/3/library/csv.html>

Export to Excel

The Excel format is much more flexible, but also much more flexible. Modern versions of Excel use an XML-based format, so if you open any .xlsx document in a text editor, you can see the format.

xlsx-writer is one library that is available to "pip install":

<https://xlsxwriter.readthedocs.io>

Luckily, there are several freely available Python libraries that can be used to make Excel document generation easy.

This is just a small sample of the things you can do with the data from Jira. For other ideas, you can use Google, search open source projects on GitHub.com, Gitlab.com, etc., and look at available Python libraries.

There are also many ways to use JavaScript to create dynamic and interactive webpages. React and other front-end frameworks can add a lot of capability to your web application. And, just like Python has "pip" where you can easily add functions through libraries, React is built on Node.js technology, which allows you to easily install JavaScript frameworks through the Node Package Manager (**npm**).

END #