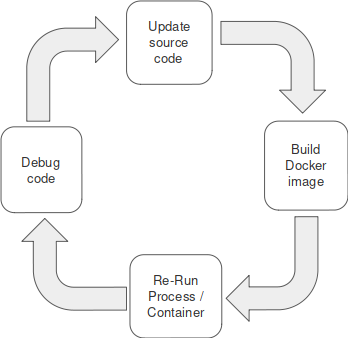
Here, we review the development cycle of the project and discuss in more detail how to apply code updates and debug failures of the containerized Python services. The goal is to analyze how to speed up these recurrent phases of the development process such that we get a similar experience to the local development one.

## Applying Code Updates

In general, our containerized development cycle consists of writing/updating code, building, running and debugging it.



For the building and running phase, as most of the time we actually have to wait, we want these phases to go pretty quick such that we focus on coding and debugging.

We now analyze how to optimize the build phase during development. The build phase corresponds to image build time when we change the Python source code. The image needs to be rebuilt in order to get the Python code updates in the container before launching it.

We can however apply code changes without having to build the image. We can do this simply by bind-mounting the local source directory to its path in the container. For this, we update the compose file as follows:

*docker-compose.yaml*

...

app:

build: app

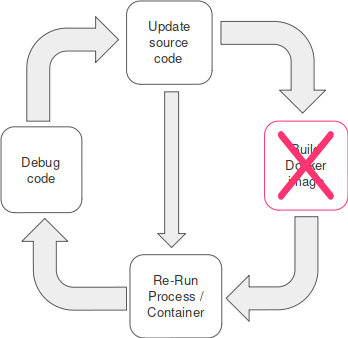
restart: always

volumes:

- ./app/src:/code

…

With this, we have direct access to the updated code and therefore we can skip the image build and restart the container to reload the Python process.



Furthermore, we can avoid re-starting the container if we run inside it a reloader process that watches for file changes and triggers the restart of the Python process once a change is detected. We need to make sure we have bind-mounted the source code in the Compose file as described previously.

In our example, we use the Flask framework that, in debugging mode, runs a very convenient module called the reloader. The reloader watches all the source code files and automatically restarts the server when it detects that a file has changed. To enable the debug mode we only need to set the debug parameter as below:

*server.py*

server.run(debug=True, host='0.0.0.0', port=5000)

If we check the logs of the *app* container we see that the flask server is running in debugging mode.

$ docker-compose logs app

Attaching to project\_app\_1

app\_1 | \* Serving Flask app "server" (lazy loading)

app\_1 | \* Environment: production

app\_1 | WARNING: This is a development server. Do not use it in a production deployment.

app\_1 | Use a production WSGI server instead.

app\_1 | \* Debug mode: on

app\_1 | \* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

app\_1 | \* Restarting with stat

app\_1 | \* Debugger is active!

app\_1 | \* Debugger PIN: 315-974-099

Once we update the source code and save, we should see the notification in the logs and reload.

$ docker-compose logs app

Attaching to project\_app\_1

app\_1 | \* Serving Flask app "server" (lazy loading)

...

app\_1 | \* Debugger PIN: 315-974-099

app\_1 | \* Detected change in '/code/server.py', reloading

app\_1 | \* Restarting with stat

app\_1 | \* Debugger is active!

app\_1 | \* Debugger PIN: 315-974-099

## Debugging Code

We can debug code in mostly two ways.

First is the old fashioned way of placing print statements all over the code for checking runtime value of objects/variables. Applying this to containerized processes is quite straightforward and we can easily check the output with a *docker-compose logs* command.

Second, and the more serious approach is by using a debugger. When we have a containerized process, we need to run a debugger inside the container and then connect to that remote debugger to be able to inspect the instance data.

We take as an example again our Flask application. When running in debug mode, aside from the reloader module it also includes an interactive debugger. Assume we update the code to raise an exception, the Flask service will return a detailed response with the exception.



Another interesting case to exercise is the interactive debugging where we place breakpoints in the code and do a live inspection. For this we need an IDE with Python and remote debugging support. If we choose to rely on Visual Studio Code to show how to debug Python code running in containers we need to do the following to connect to the remote debugger directly from VSCode.

First, we need to map locally the port we use to connect to the debugger. We can easily do this by adding the port mapping to the Compose file:

*docker-compose.yaml*

...

app:

build: app

restart: always

volumes:

- ./app/src:/code

ports:

- 5678:5678

…

Next, we need to import the debugger module in the source code and make it listen on the port we defined in the Compose file. We should not forget to add it to the dependencies file also and rebuild the image for the app service to get the debugger package installed. For this exercise, we choose to use the *ptvsd* debugger package that VS Code supports.

*server.py*

...

import ptvsd

ptvsd.enable\_attach(address=('0.0.0.0', 5678))

...

*requirements.txt*

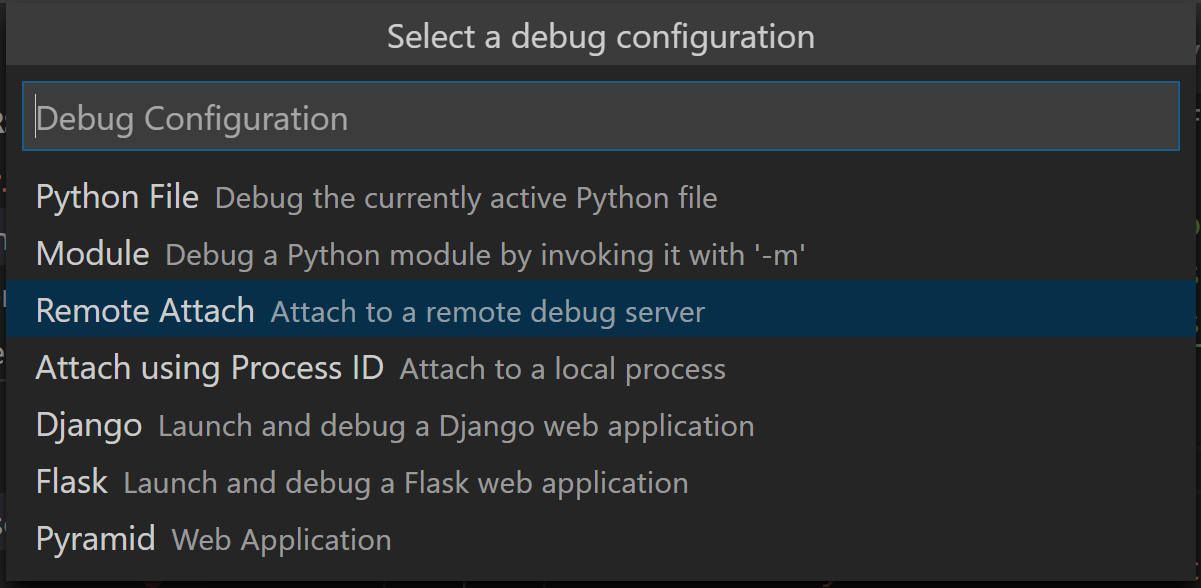
Flask==1.1.1

mysql-connector==2.2.9

ptvsd==4.3.2

We need to remember that for changes we make in the Compose file, we need to run a compose down command to remove the current containers setup and then run a docker-compose up to redeploy with the new configurations in the compose file.

Finally, we need to create a ‘Remote Attach’ configuration in VS Code to launch the debugging mode.



The *launch.json* for our project should look like:

{

"version": "0.2.0",

"configurations": [

{

"name": "Python: Remote Attach",

"type": "python",

"request": "attach",

"port": 5678,

"host": "localhost",

"pathMappings": [

{

"localRoot": "${workspaceFolder}/app/src",

"remoteRoot": "/code"

}

]

}

]

}

We need to make sure we update the path map locally and in the container.

Once we do this, we can easily place breakpoints in the IDE, start the debugging mode based on the configuration we created and, finally, trigger the code to reach the breakpoint.