Building a plugin architecture with Python

Maxwell Mapako · Follow
6 min read · Apr 24, 2021

*It's no secret that one of the greatest software design principles when extending functionality is to consider using plugins (where possible)**

I will keep this post short and avoid repetition, if you don't know what a plugin system is then now would be a good time to find out from here. Now that is out of the way, let us create a small use-case that we need a solution for.

Problem statement

Given a broad category of devices that share similar functionality and a communication interface with indecent protocols, in this case let's call them inverters (for a solar system). These inverters can provide us with power readings for input (**solar** and or **battery**), output (to mains/load) and device information/status e.t.c that we can access though some text based. Say for example to change from hybrid mode (**solar** and **grid**) to just offline mode (**solar** and **battery** only).

At this point we know that a monolithic application would be either too large to accommodate all the possible inverter types and protocols, so our best bet would be to create a core application that only knows about the high level features (read device info/status, read power data and send commands to change some settings). We can achieve this by creating small units to handle the complexities of the communication protocols (taking in the core application commands and transforming them into commands the inverter can understand). This approach would allow us to ship a bare application and apply/download a plugin for a specific inverter/s which we have.

Fundamental plugin concepts

A common plugin system typically needs to define some form of contract/s that the plugins can extend and the core application can use as an abstraction layer for communication, this could be in the form of a package with some abstract classes. Typically the application would using the following steps/phases to interact with the plugins:

Discovery

This is the mechanism by which a running application can find out which plugins it has at its disposal. To "discover" a plugin, one has to look in certain places, and also know what to look for.

Registration

This is the mechanism by which a plugin tells an application — "I'm here, ready to do work". Admittedly, registration usually has a large overlap with discovery.

Application Hooks

Hooks are also called "mount points" or "extension points". These are the places where the plugin can "attach" itself to the application, signalling that it wants to know about certain events and participate in the flow. The exact nature of hooks is very much dependent on the application.

Exposing application API to plugins

To make plugins truly powerful and versatile, the application needs to give them access to itself, by means of exposing an API the plugins can use.

Code examples

> *I will keep this section short so for the sake of time saving and simplicity, please see the full sample-project on github*

wax911/plugin-architecture

Discovery Registration Application hooks to which plugins attach Exposing application capabilities back to plugins…

github.com

Let's start off by defining some sort of configuration file, which will tell our application which plugins to load and where to get them: (*settings/configuration.yaml*)

```yaml
registry:
  # Plugin registry settings example
  url: 'https://github.com/{name}/{repository}/releases/download/{tag}'
  name: ''
  repository: ''
  tag: 'latest'
logging:
  # Setting the log level: CRITICAL, ERROR, WARNING, INFO, DEBUG
  level: 'DEBUG'
plugins:
  # Packages to download from registry
  - advance-plugin
  - sample-plugin
```

Discovery

Let us create a few contracts that all plugins will implement/extend, one contract will be used for registering a plugin when it is loaded/registered (will go into more detail later) and the other defines the behaviour of the plugin (what it can do and the information we can expect to get):

```python
from logging import Logger
from typing import Optional, List

from model import Meta, Device


class IPluginRegistry(type):
    plugin_registries: List[type] = list()

    def __init__(cls, name, bases, attrs):
        super().__init__(cls)
        if name != 'PluginCore':
            IPluginRegistry.plugin_registries.append(cls)


class PluginCore(object, metaclass=IPluginRegistry):
    """
    Plugin core class
    """

    meta: Optional[Meta]

    def __init__(self, logger: Logger) -> None:
        """
        Entry init block for plugins
        :param logger: logger that plugins can make use of
        """
        self._logger = logger

    def invoke(self, **args) -> Device:
        """
        Starts main plugin flow
        :param args: possible arguments for the plugin
        :return: a device for the plugin
        """
        pass
```

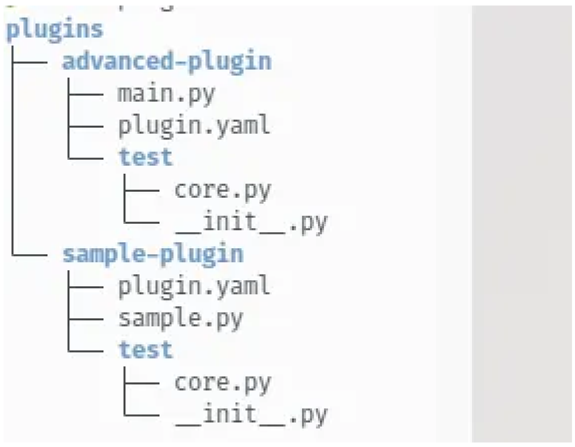
The references models can be found here, these are just data classes for demonstration purposes

> *N.B. IPluginRegistry extends type because we need some sort of mechanism to register plugins which we will not instantiating immediately after the discovery phase, think of this as just keeping information on what class type we're dealing with (we would eventually call plugin_registries[index]() to instantiate the class)*

If we look at the *init(constructor)* block inside our *IPluginRegistry we can see that it* contains something interesting, it checks if the loaded class (this can be considered as part of the hook phase) is not the plugin contract (PluginCore) not to be confused with the *instanceof* operator/function.

Registration

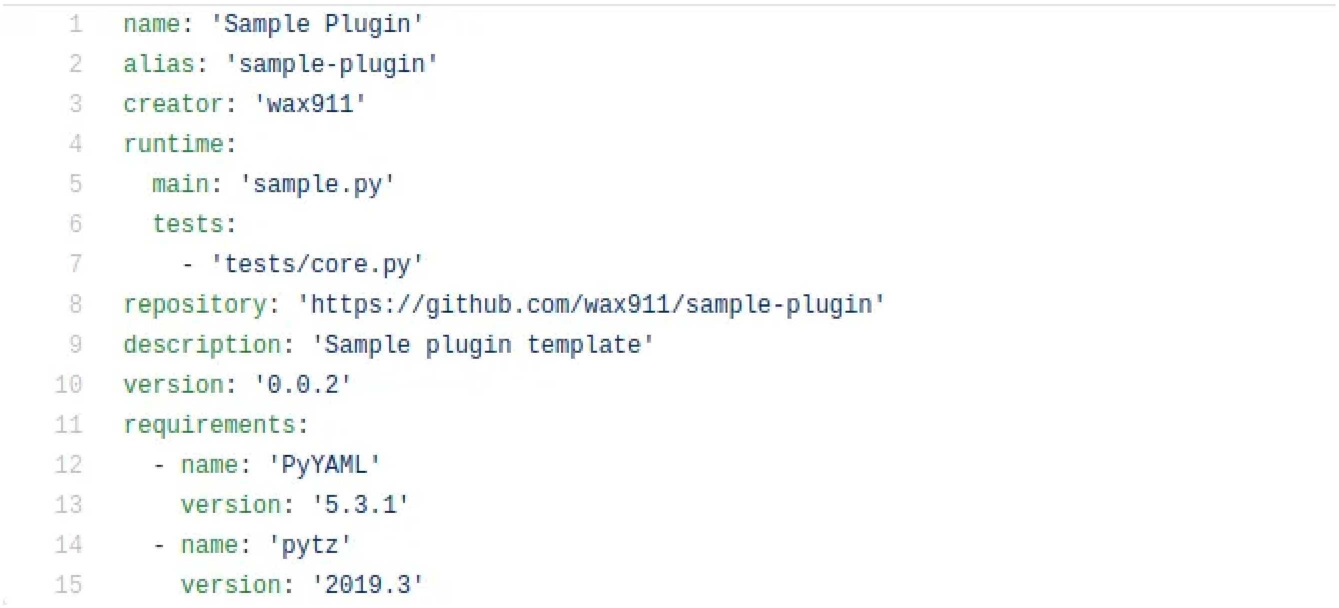
In the sample project all our plugins will be placed in a *plugins* directory, this is to simulate a case where a plugins are downloaded and kept.



```
plugins
├── advanced-plugin
│   ├── main.py
│   ├── plugin.yaml
│   └── test
│       ├── core.py
│       └── __init__.py
└── sample-plugin
    ├── plugin.yaml
    ├── sample.py
    └── test
        ├── core.py
        └── __init__.py
```

Sample plugin directory structure which can be found here

> *Each of the plugins will have a configuration file called plugin.yaml that will contain information regarding which "main" file to load, dependencies which we will install automatically (yes, I know this would be a big security risk in production if plugins are not audited against malicious intent)*



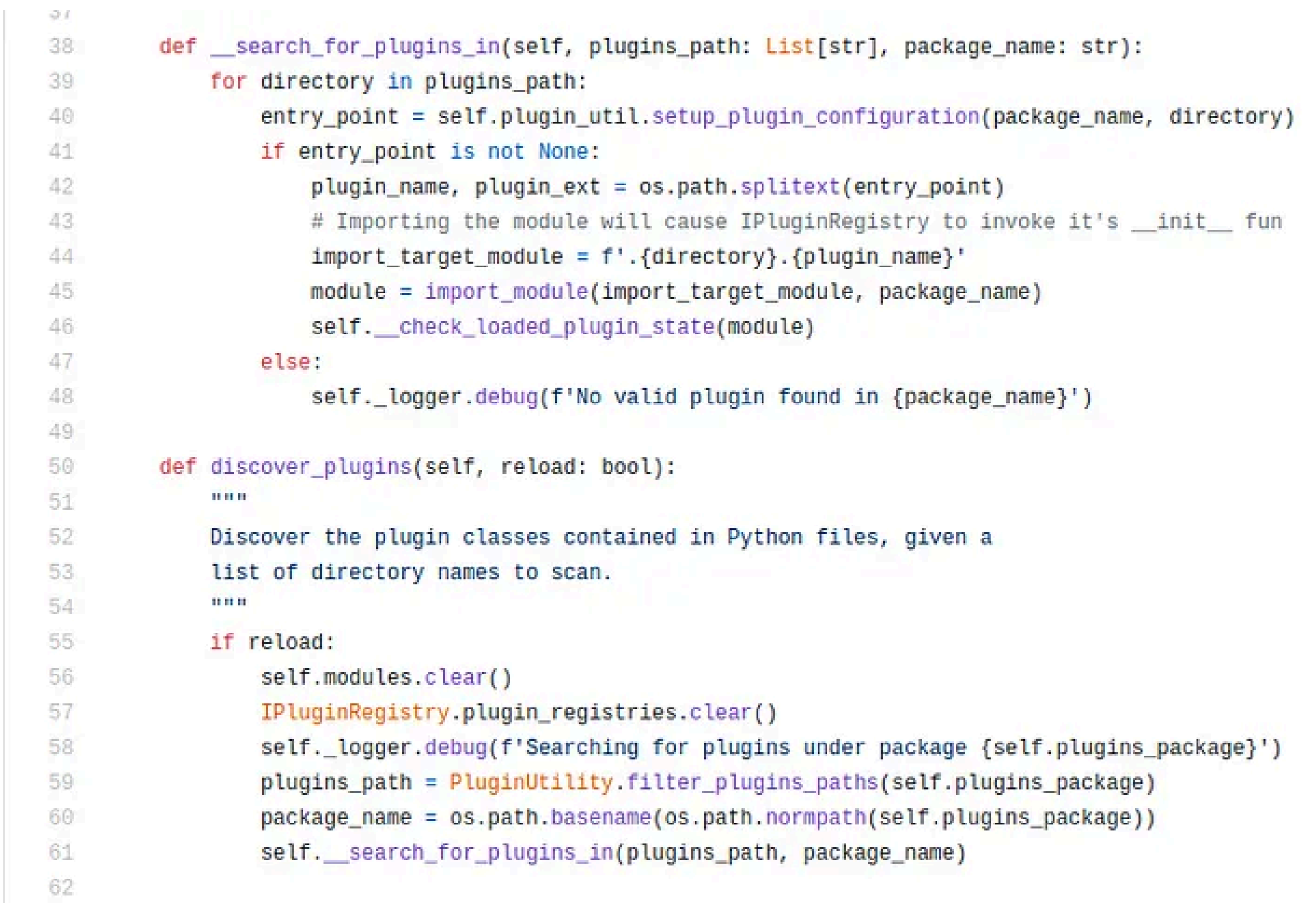
```yaml
name: 'Sample Plugin'
alias: 'sample-plugin'
creator: 'wax911'
runtime:
  main: 'sample.py'
  tests:
    - 'tests/core.py'
repository: 'https://github.com/wax911/sample-plugin'
description: 'Sample plugin template'
version: '0.0.2'
requirements:
  - name: 'PyYAML'
    version: '5.3.1'
  - name: 'pytz'
    version: '2019.3'
```

plugin.yaml configuration file for the sample plugin

```python
from logging import Logger

from engine import PluginCore
from model import Meta, Device


class SamplePlugin(PluginCore):

    def __init__(self, logger: Logger) -> None:
        super().__init__(logger)
        self.meta = Meta(
            name='Sample Plugin',
            description='Sample plugin template',
            version='0.0.1'
        )

    @staticmethod
    def __create_device() -> Device:
        return Device(
            name='Sample Device',
            firmware=0xa2c3f,
            protocol='SAMPLE',
            errors=[0x0000]
        )

    def invoke(self, command: chr) -> Device:
        self._logger.debug(f'Command: {command} -> {self.meta}')
        device = self.__create_device()
        return device
```

sample.py the main sample plugin file which can be found here

Our sample plugins extends *PluginCore* and expects a logger as a dependency from the application (*This is to simulate the main application providing dependencies to its plugins, this could be in the form of a configured network client, a database wrapper to provide read only functionality or anything really*) and has an invoke method which is called by the core application to run commands on the destination device.



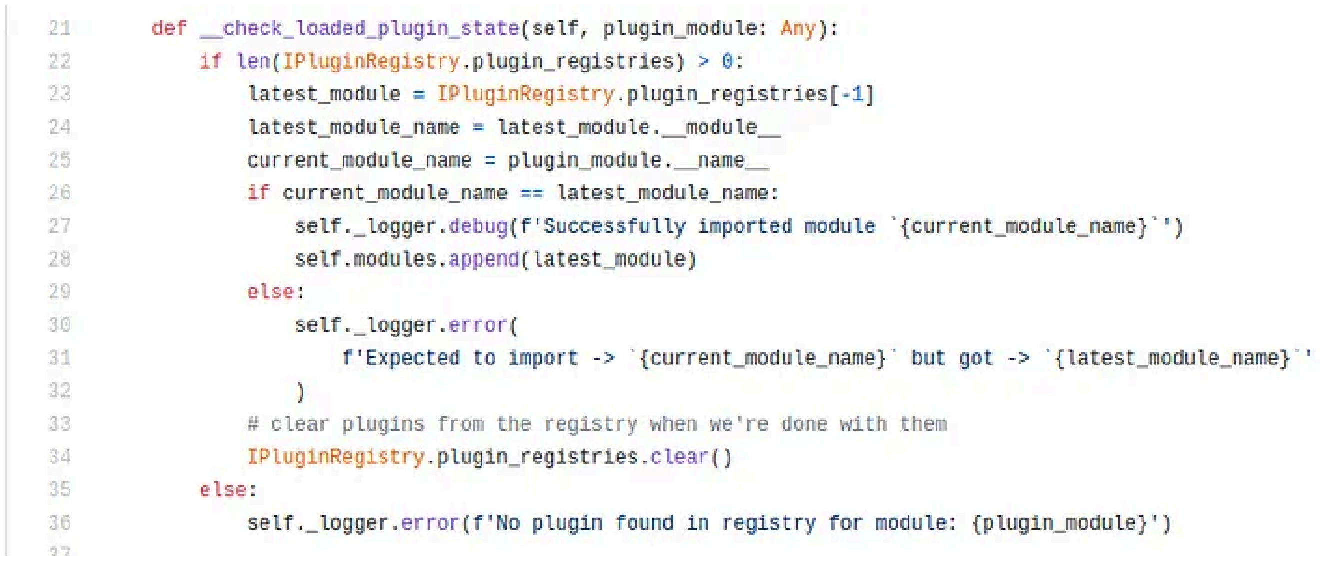
```python
    def __search_for_plugins_in(self, plugins_path: List[str], package_name: str):
        for directory in plugins_path:
            entry_point = self.plugin_util.setup_plugin_configuration(package_name, directory)
            if entry_point is not None:
                plugin_name, plugin_ext = os.path.splitext(entry_point)
                # Importing the module will cause IPluginRegistry to invoke it's __init__ fun
                import_target_module = f'.{directory}.{plugin_name}'
                module = import_module(import_target_module, package_name)
                self.__check_loaded_plugin_state(module)
            else:
                self._logger.debug(f'No valid plugin found in {package_name}')

    def discover_plugins(self, reload: bool):
        """
        Discover the plugin classes contained in Python files, given a
        list of directory names to scan.
        """
        if reload:
            self.modules.clear()
            IPluginRegistry.plugin_registries.clear()
            self._logger.debug(f'Searching for plugins under package {self.plugins_package}')
            plugins_path = PluginUtility.filter_plugins_paths(self.plugins_package)
            package_name = os.path.basename(os.path.normpath(self.plugins_package))
            self.__search_for_plugins_in(plugins_path, package_name)

```

PluginUseCase discover and load/hook plugins

> *The above code snippet is from the PluginUseCase class where the application will search the plugin directory for a configuration file* (plugin.yaml) *which is handled in PluginUtility (for all the installation and dependency resolution details), and finally loaded/registered after all checks are done.*

You can read more about what *import_module* is from here

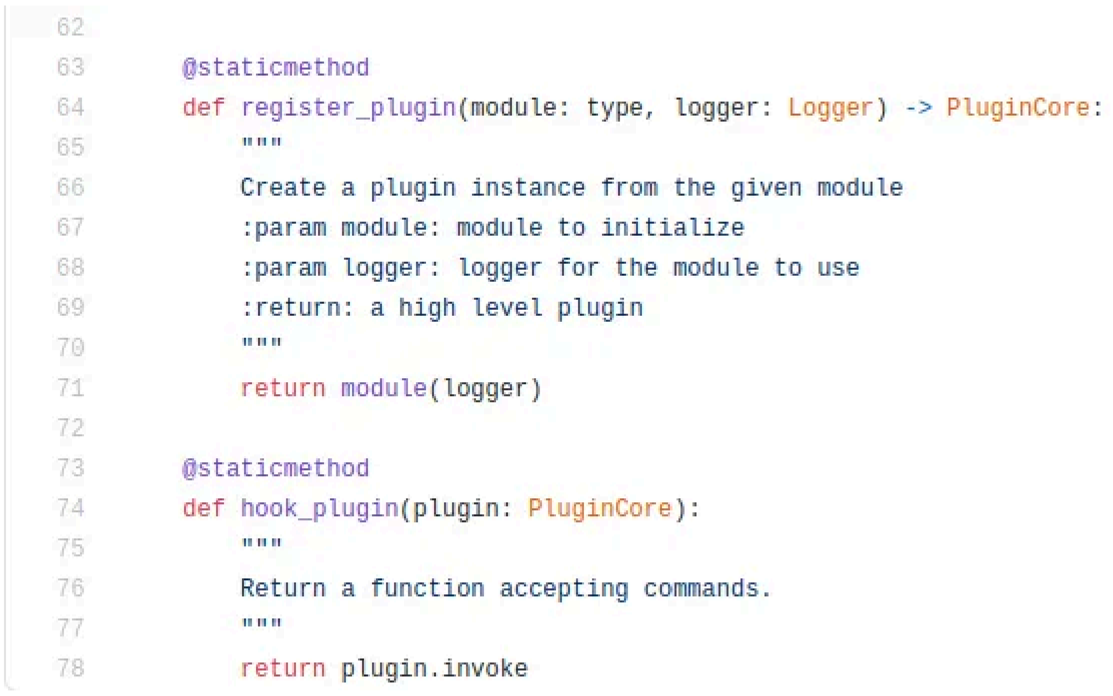
Application hooks and exposing the API



```python
    def __check_loaded_plugin_state(self, plugin_module: Any):
        if len(IPluginRegistry.plugin_registries) > 0:
            latest_module = IPluginRegistry.plugin_registries[-1]
            latest_module_name = latest_module.__module__
            current_module_name = plugin_module.__name__
            if current_module_name == latest_module_name:
                self._logger.debug(f'Successfully imported module `{current_module_name}`')
                self.modules.append(latest_module)
            else:
                self._logger.error(
                    f'Expected to import -> `{current_module_name}` but got -> `{latest_module_name}`'
                )
            # clear plugins from the registry when we're done with them
            IPluginRegistry.plugin_registries.clear()
        else:
            self._logger.error(f'No plugin found in registry for module: {plugin_module}')
```

Plugins are loaded through **__search_for_plugins** at this point

After the plugins are loaded we clean up the global registry (IPluginRegistry) since all loaded plugins will be stored in the PluginUseCase scope, and we're ready to use our plugins through the following functions which our core application will call on.



```python

    @staticmethod
    def register_plugin(module: type, logger: Logger) -> PluginCore:
        """
        Create a plugin instance from the given module
        :param module: module to initialize
        :param logger: logger for the module to use
        :return: a high level plugin
        """
        return module(logger)

    @staticmethod
    def hook_plugin(plugin: PluginCore):
        """
        Return a function accepting commands.
        """
        return plugin.invoke
```

Plugin instance creation and invocation

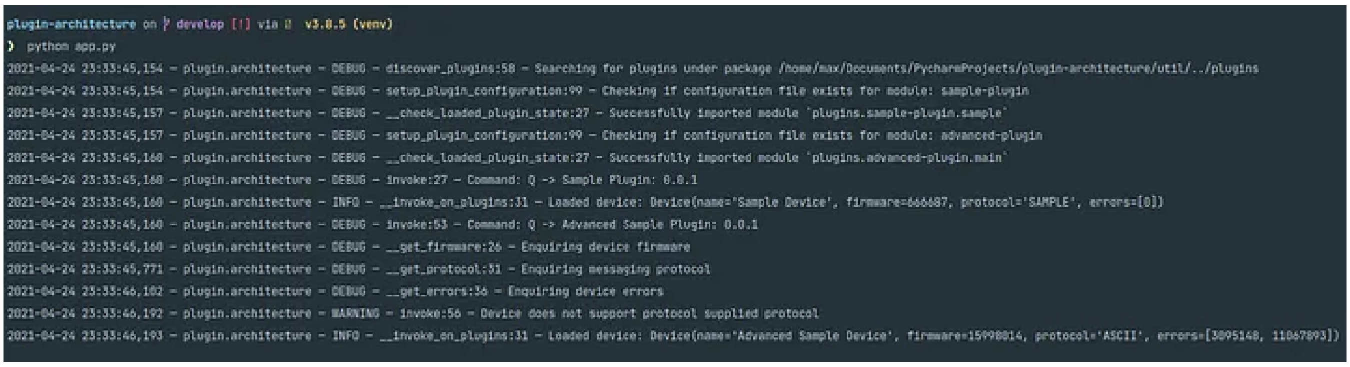
> *Arguably one could omit the* **register_plugin** *and* **hook_plugin** *functions and directly do this inline in the the calling core application module, but for the sake of demonstrating the phases separately we're doing it this way*

```python
class PluginEngine:
    _logger: Logger

    def __init__(self, **args) -> None:
        self._logger = LogUtil.create(args['options']['log_level'])
        self.use_case = PluginUseCase(args['options'])

    def start(self) -> None:
        self.__reload_plugins()
        self.__invoke_on_plugins('Q')

    def __reload_plugins(self) -> None:
        """Reset the list of all plugins and initiate the walk over the main
        provided plugin package to load all available plugins
        """
        self.use_case.discover_plugins(True)

    def __invoke_on_plugins(self, command: chr):
        """Apply all of the plugins on the argument supplied to this function
        """
        for module in self.use_case.modules:
            plugin = self.use_case.register_plugin(module, self._logger)
            delegate = self.use_case.hook_plugin(plugin)
            device = delegate(command=command)
            self._logger.info(f'Loaded device: {device}')
```

Application core module which can be found here

In this example we're just calling calling all the loaded plugins, depending on our use-case we may just want one plugin and that's easily done by simply specifying a single plugin in our *configuration.yaml* file.

Running the project and we'd get the following output:

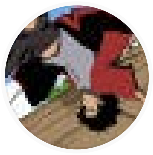


```
plugin-architecture on  develop [!] via   v3.8.5 (venv)
❯ python app.py
2021-04-24 23:33:45,154 - plugin.architecture - DEBUG - discover_plugins:58 - Searching for plugins under package /home/max/Documents/PycharmProjects/plugin-architecture/util/../plugins
2021-04-24 23:33:45,154 - plugin.architecture - DEBUG - setup_plugin_configuration:99 - Checking if configuration file exists for module: sample-plugin
2021-04-24 23:33:45,157 - plugin.architecture - DEBUG - __check_loaded_plugin_state:27 - Successfully imported module `plugins.sample-plugin.sample`
2021-04-24 23:33:45,157 - plugin.architecture - DEBUG - setup_plugin_configuration:99 - Checking if configuration file exists for module: advanced-plugin
2021-04-24 23:33:45,160 - plugin.architecture - DEBUG - __check_loaded_plugin_state:27 - Successfully imported module `plugins.advanced-plugin.main`
2021-04-24 23:33:45,160 - plugin.architecture - DEBUG - invoke:27 - Command: Q -> Sample Plugin: 0.0.1
2021-04-24 23:33:45,160 - plugin.architecture - INFO - __invoke_on_plugins:31 - Loaded device: Device(name='Sample Device', firmware=666687, protocol='SAMPLE', errors=[0])
2021-04-24 23:33:45,160 - plugin.architecture - DEBUG - invoke:53 - Command: Q -> Advanced Sample Plugin: 0.0.1
2021-04-24 23:33:45,160 - plugin.architecture - DEBUG - __get_firmware:26 - Enquiring device firmware
2021-04-24 23:33:45,771 - plugin.architecture - DEBUG - __get_protocol:31 - Enquiring messaging protocol
2021-04-24 23:33:46,102 - plugin.architecture - DEBUG - __get_errors:36 - Enquiring device errors
2021-04-24 23:33:46,192 - plugin.architecture - WARNING - invoke:56 - Device does not support protocol supplied protocol
2021-04-24 23:33:46,193 - plugin.architecture - INFO - __invoke_on_plugins:31 - Loaded device: Device(name='Advanced Sample Device', firmware=15998014, protocol='ASCII', errors=[3095148, 11067893])
```

Thank you for reading and let me know if you have any questions :)

> *Credits for inspiration: https://eli.thegreenplace.net/2012/08/07/fundamental-concepts-of-plugin-infrastructures*

Python3 Python Plugins Design Patterns



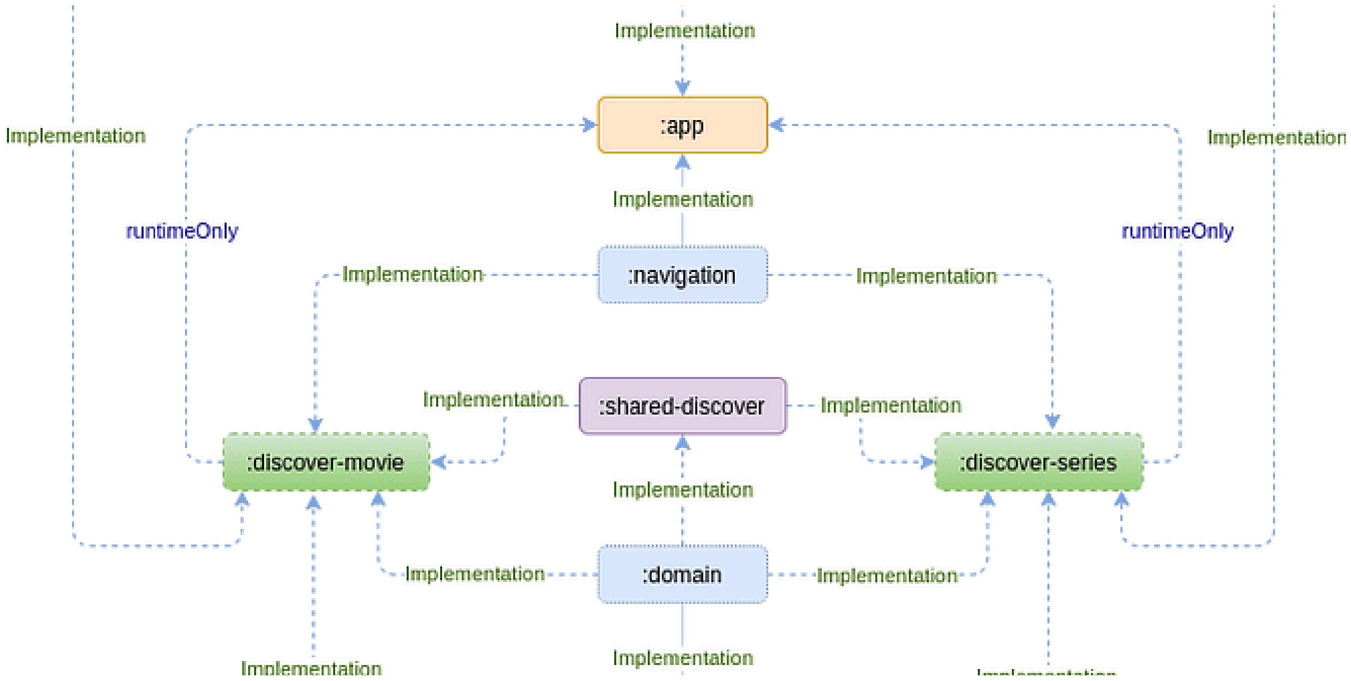
Follow

Written by Maxwell Mapako

21 Followers

Freelancer and open source developer, growing my skills one byte a time :)

More from Maxwell Mapako



Maxwell Mapako

Navigating through runtime only modules and thinking outside of the box with androidx.startup

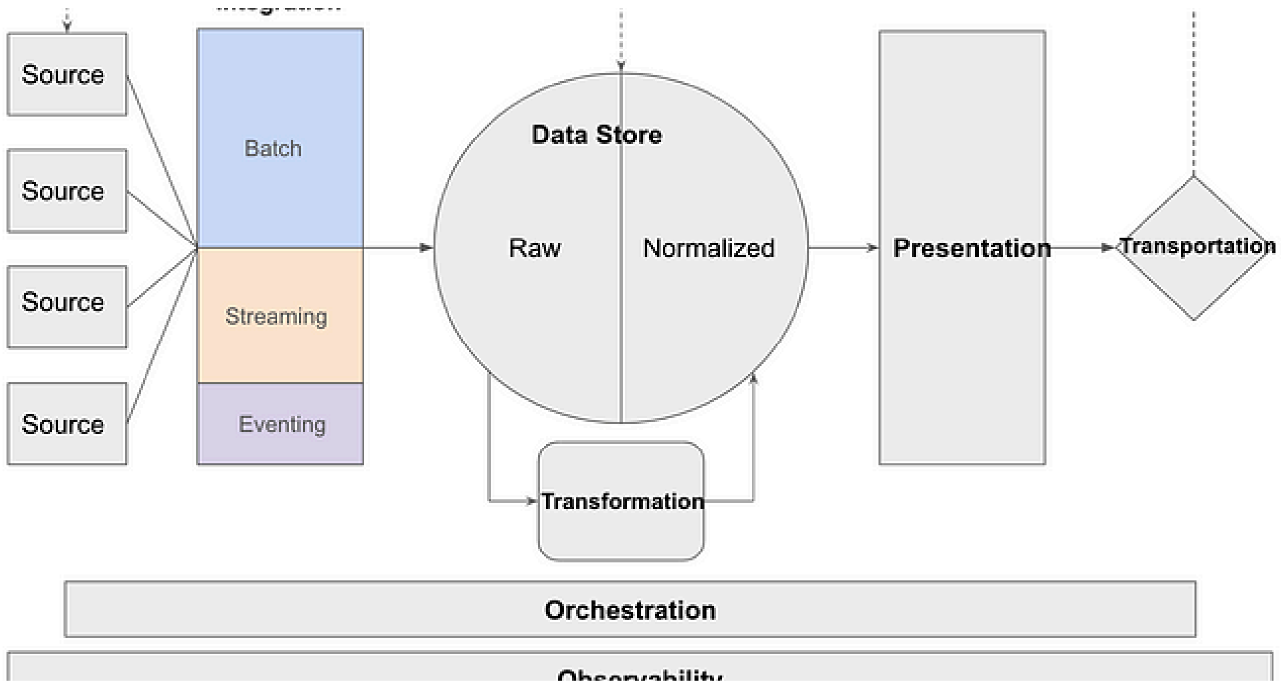
androix.startup is a new addition to the Jetpack suite of libraries. Read more about it here

6 min read · Nov 20, 2020

26

See all from Maxwell Mapako

Recommended from Medium



Dave Melillo in Towards Data Science

Building a Data Platform in 2024

How to build a modern, scalable data platform to power your analytics and data science projects (updated)

9 min read · Feb 6, 2024

2.5K 33

```
{
  "name": "JSON is slow!",
  "blog": true,
  "writtenAt": 1695884403,
  "topics": ["JSON", "Javascript"]
}
```

Vaishnav Manoj in DataX Journal

JSON is incredibly slow: Here's What's Faster!

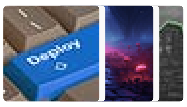
Unlocking the Need for Speed: Optimizing JSON Performance for Lightning-Fast Apps and Finding Alternatives to it!

16 min read · Sep 28, 2023

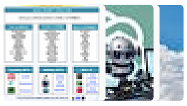
14.4K 170

Lists

Coding & Development
11 stories · 521 saves


Predictive Modeling w/ Python
20 stories · 1030 saves

Practical Guides to Machine Learning
10 stories · 1232 saves


ChatGPT
21 stories · 534 saves

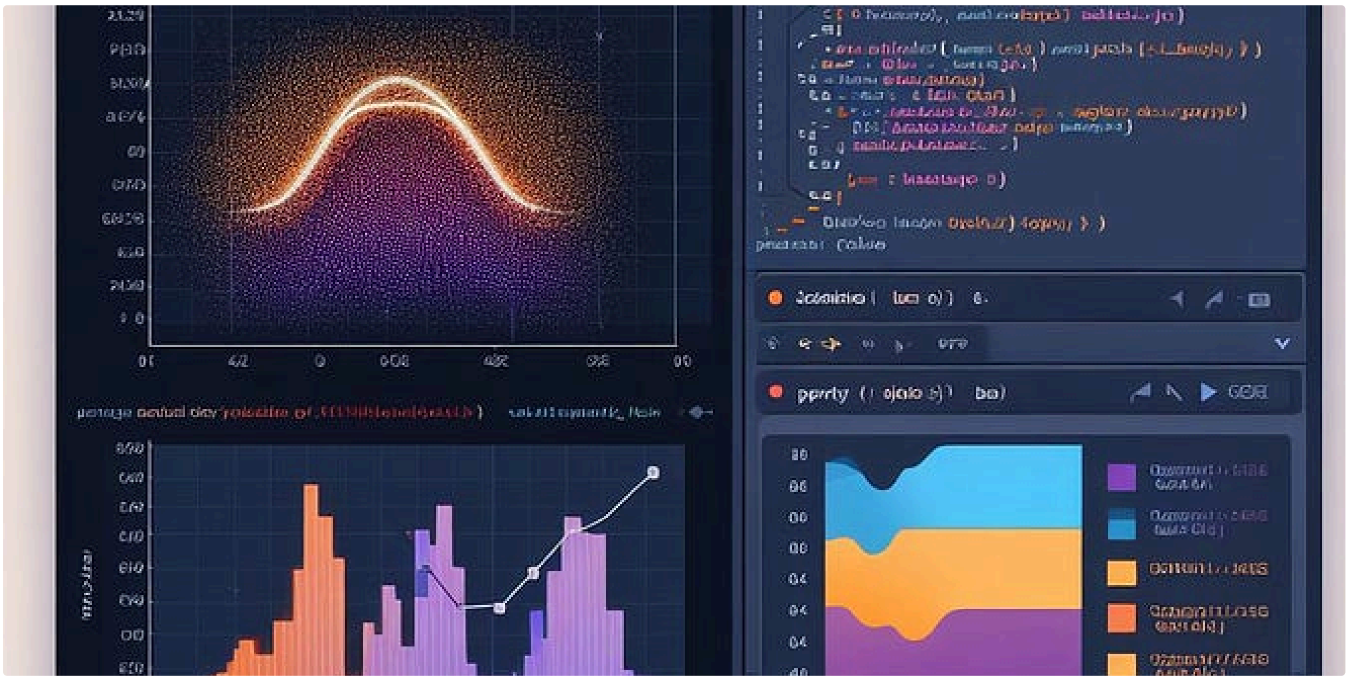
Anmol Tomar in CodeX

Say Goodbye to Loops in Python, and Welcome Vectorization!

Use Vectorization—a super-fast alternative to loops in Python

· 5 min read · Dec 28, 2023

4.95K 60



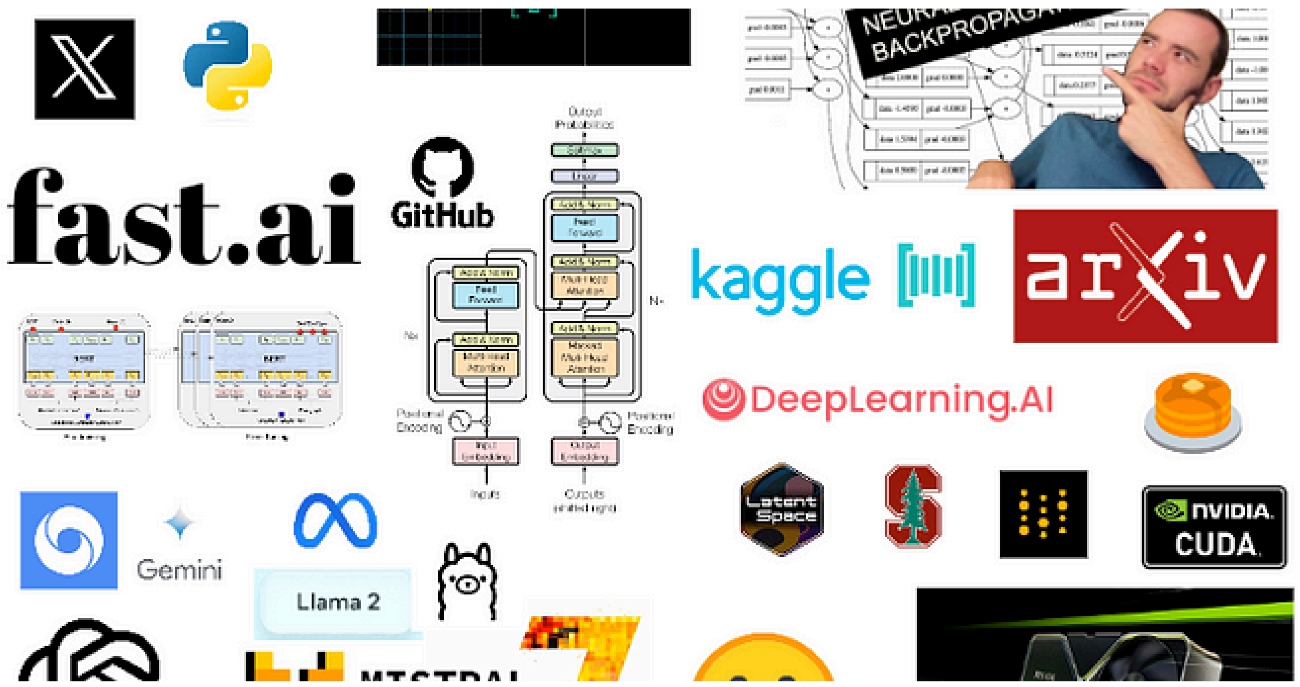
Daniel Wu

Elevate Your Python Data Visualization Skills: A Deep Dive into Advanced Plotly Techniques with...

Data visualization is a crucial aspect of data analysis and exploration. It helps in gaining insights into the underlying patterns, trends...

8 min read · Nov 21, 2023

287 1



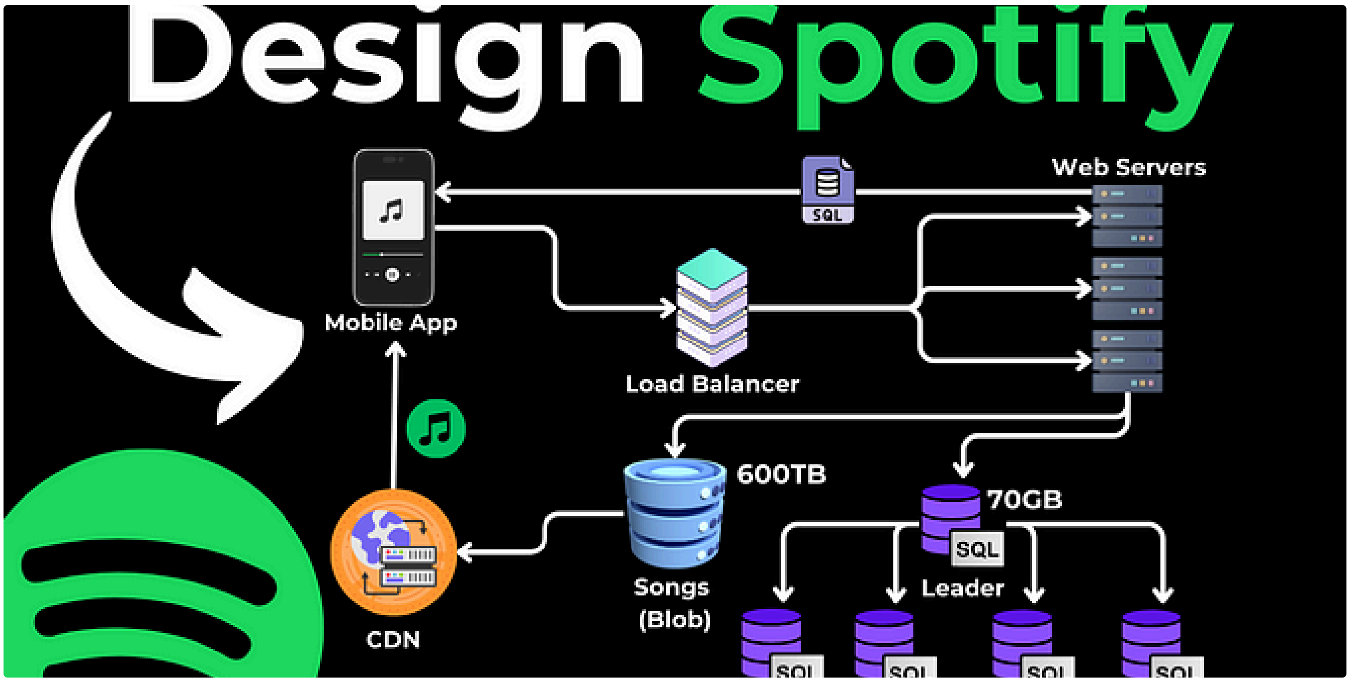
Benedict Neo in bitgrit Data Science Publication

Roadmap to Learn AI in 2024

A free curriculum for hackers and programmers to learn AI

11 min read · Mar 11, 2024

7.7K 83



Hayk Simonyan in Level Up Coding

System Design Interview Question: Design Spotify

High-level overview of a System Design Interview Question - Design Spotify.

6 min read · Feb 21, 2024

4K 29

See more recommendations