Pluggable Architecture

Aly Sivji

@CaiusSivjus
I’m Aly Sivji. @CaiusSivjus on
**breathe**
Pluggable Architecture
Plugins are software components that extend or enhance an existing program.
PLUGINS

PLUGINS EVERYWHERE
EXTENSIONS

Search Extensions in Marketplace

**ENABLED**

- **Thomas Walther**
- **Debugger for Chrome** 4.12.8
  - Debug your JavaScript code in the Chrome browser
  - Microsoft
- **Disable Ligatures** 0.0.8
  - Disable ligatures at the cursor position, or disable ligatures
  - CoenraadS
- **Docker** 1.2.1
  - Makes it easy to create, manage, and debug containers
  - Microsoft
- **Dracula Official** 2.22.1
  - Official Dracula Theme. A dark theme for many editors
  - Dracula Theme
- **GitHub Pull Requests and Issues** 0.16.0
  - Pull Request and Issue Provider for GitHub
  - GitHub
- **GraphQL** 0.2.14
  - GraphQL extension for VSCode adds syntax highlighting
  - Prisma

**DISABLED**

- **45**
- **5K** 5
- **4.6M** 4.5
- **1.2M** 5
- **488K** 4
- **191K** 4
WHAT SHOULD WE DO NEXT?

WOOF

WE NEED TO ADD A PLUGIN SYSTEM...
Benefits
Benefits

- New features are easier to develop
Benefits

- New features are easier to develop
- Separation of Concerns
Benefits

- New features are easier to develop
- Separation of Concerns
- Third-party developers can extend your app
Trade-offs
Trade-offs

- Upfront design cost
Trade-offs

- Upfront design cost
- Additional complexity inside core application
#Goals
Write a third-party plugin from scratch
To catch a fish, you have to think like a fish

I'm wet, and I don't even know it
What this talk is not...
Dynamic Code Patterns: Extending Your Applications with Plugins

presented by
Doug Hellmann
Rose Judge - Plug-in to Python: Extending your applications through the use of plugins
Motivating Example:
Generate API Documentation
Create user

This can only be done by the logged in user.

REQUEST BODY SCHEMA: application/json

```
{
  "pet": {  
    "category": {  
      "name": "string",
      "sub": {  
        "prop1": "string"
      }
    },
    "name": "Ours",
    "photoUrls": {  
      "string"
    },
    "friend": {},
    "tags": {},
    "status": "available",
    "petType": "string"
  },
  "username": "John78",
  "firstName": "John",
  "lastName": "Smith",
  "email": "john.smith@example.com",
  "password": "drowsyP123",
  "phone": "+1-202-555-0192",
  "userStatus": 0
}
```
HELLO, IS IT ME YOU'RE LOOKING FOR?
A pluggable API specification generator. Currently supports the OpenAPI Specification (f.k.a. the Swagger specification).

Features

- Supports the OpenAPI Specification (versions 2 and 3)
- Framework-agnostic
- Built-in support for marshmallow
- Utilities for parsing docstrings

[github.com/marshmallow-code/apispec](https://github.com/marshmallow-code/apispec)
A pluggable API specification generator. Currently supports the OpenAPI Specification (f.k.a. the Swagger specification).

Features

- Supports the OpenAPI Specification (versions 2 and 3)
- Framework-agnostic
- Built-in support for marshmallow
- Utilities for parsing docstrings

github.com/marshmallow-code/apispec
A pluggable API specification generator. Currently supports the OpenAPI Specification (f.k.a. the Swagger specification).

Features

- Supports the OpenAPI Specification (versions 2 and 3)
- Framework-agnostic
- Built-in support for marshmallow
- Utilities for parsing docstrings

[github.com/marshmallow-code/apispec](github.com/marshmallow-code/apispec)
apispec

A pluggable API specification generator. Currently supports the OpenAPI Specification (f.k.a. the Swagger specification).

Features

- Supports the OpenAPI Specification (versions 2 and 3)
- Framework-agnostic
- Built-in support for marshmallow
- Utilities for parsing docstrings

github.com/marshmallow-code/apispec
Writing Plugins

A plugin is a subclass of `apispec.plugin.BasePlugin`.

**Helper Methods**

Plugins provide "helper" methods that augment the behavior of `apispec.APISpec` methods.

There are five types of helper methods:
Helper Methods

Plugins provide “helper” methods that augment the behavior of `apispec.APISpec` methods.

There are five types of helper methods:

- Schema helpers
- Parameter helpers
- Response helpers
- Path helpers
- Operation helpers

Helper functions modify `apispec.APISpec` methods. For example, path helpers modify `apispec.APISpec.path`.
The `init_spec` Method

`BasePlugin` has an `init_spec` method that `APISpec` calls on each plugin at initialization with the `spec` object itself as parameter. It is no-op by default, but a plugin may override it to access and store useful information on the `spec` object.

A typical use case is conditional code depending on the OpenAPI version, which is stored as `openapi_version` on the `spec` object. See source code for `apispec.ext.marshmallow.MarshmallowPlugin` for an example.
Next Steps

To learn more about how to write plugins:

- Consult the Core API docs for BasePlugin.
- View the source for an existing apispec plugin, e.g. FlaskPlugin.
- Check out some projects using apispec: https://github.com/marshmallow-code/apispec/wiki/Ecosystem
class CategorySchema(Schema):
    id = fields.Int()
    name = fields.Str(required=True)

class PetSchema(Schema):
    categories = fields.List(fields.Nested(CategorySchema))
    name = fields.Str()

@app.route("/random")
def random_pet():
    """A cute furry animal endpoint.
    ""
    get:
    description: Get a random pet

    responses:
    200:
    description: Return a pet
    content:
    application/json:
    schema: PetSchema

    """

    # Hardcoded example data
    pet_data = {
        "name": "sample_pet_" + str(uuid.uuid1()),
        "categories": [{"id": 1, "name": "sample_category"}],
    }
    return PetSchema().dump(pet_data)
class CategorySchema(Schema):
    id = fields.Int()
    name = fields.Str(required=True)

class PetSchema(Schema):
    categories = fields.List(fields.Nested(CategorySchema))
    name = fields.Str()

@app.route("/random")
def random_pet():
    """A cute furry animal endpoint.
    """
    get:
    description: Get a random pet
    responses:
    200:
    description: Return a pet
    content:
    application/json:
        schema: PetSchema

    # Hardcoded example data
    pet_data = {
        "name": "sample_pet_" + str(uuid.uuid1()),
        "categories": [{"id": 1, "name": "sample_category"}],
    }
    return PetSchema().dump(pet_data)
class FlaskPlugin(BasePlugin):
    """APISpec plugin for Flask"""

    ...

def path_helper(self, operations, *, view, app=None, **kwargs):
    """Path helper that allows passing a Flask view function."""

    import pdb; pdb.set_trace()

    rule = self._rule_for_view(view, app=app)
    operations.update(yaml_utils.load_operations_from_docstring(view.__doc__))
    if hasattr(view, "view_class") and issubclass(view.view_class, MethodView):
        for method in view.methods:
            if method in rule.methods:
                method_name = method.lower()
                method = getattr(view.view_class, method_name)
                operations[method_name] = yaml_utils.load_yaml_from_docstring(
                )

    return self.flaskpath2openapi(rule.rule)
class FlaskPlugin(BasePlugin):
    """APISpec plugin for Flask"""

    ...

def path_helper(self, operations, *, view, app=None, **kwargs):
    """Path helper that allows passing a Flask view function."""

    import pdb; pdb.set_trace()

    rule = self._rule_for_view(view, app=app)
    operations.update(yaml_utils.load_operations_from_docstring(view.__doc__))
    if hasattr(view, "view_class") and issubclass(view.view_class, MethodView):
        for method in view.methods:
            if method in rule.methods:
                method_name = method.lower()
                method = getattr(view.view_class, method_name)
                operations[method_name] = yaml_utils.load_yaml_from_docstring(
                    )

    return self.flaskpath2openapi(rule.rule)
app = falcon.API()

class RandomPetResource:
    def on_get(self, req, resp):
        """A cute furry animal endpoint."
        
        description: Get a random pet
        responses:
        200:
            description: A pet to be returned
            schema: PetSchema
        """

        pet = None
        resp.media = pet

        random_pet_resource = RandomPetResource()
        app.add_route("/random", random_pet_resource)
apispec plugin that generates OpenAPI specification (aka Swagger) for Falcon web applications.

Apispec uses three sources of information. Basic information is directly given to `APISpec()`. The plugin reads information about paths from the Falcon app. Information about an object could be given by `marshmallow` specification.

**Installation**

```bash
pip install falcon-apispec
```

Optionaly:

```bash
pip install marshmallow
```

Works with `apispec v1.0+`. 
falcon-apispec 0.4.0

pip install falcon-apispec

Falcon plugin for apispec documentation generator.

Released: May 10, 2020
I SEE
A PATTERN
IT'S LIKE POETRY

IT RHYMES
Anatomy of a Plugin System
Require host application
Communication channel between host and plugin
Register with the host application
Loaded dynamically at runtime
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "common.middleware.RequestUuidMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
    "common.middleware.SuperuserCanViewDebugToolbarInProductionMiddleware",
]
Respond when called by the host application
Designing a Plugin System
Plugin System Checklist

- Requires host application
- Communication channel between host and plugin
- Register with the host application
- Load plugins dynamically at runtime
- Respond when called upon by the host application
Case Study: Git Stats
apispec plugin that generates OpenAPI specification (aka Swagger Docs) for Falcon web applications.

28 commits
1 branch
4 releases
9 contributors

Latest commit a5e665e 20 days ago

- falcon_apispec: Bump version for release
- scripts: Tools and notes for cutting releases (#20)
- tests: Add support for Falcon Resource suffixes (#19)
- .gitignore: Add support for Falcon Resource suffixes (#19)
$ python cli.py --help

usage: cli.py [-h] --url URL

Fetch statistics from Online Git Repo

optional arguments:
   -h, --help    show this help message and exit
   --url URL     URL to repository: https://addr..
$ python cli.py --url
https://github.com/alysivji/falcon-apispec

Description: apispec plugin that generates OpenAPI specification (aka Swagger Docs) for Falcon web applications.
Stars: 29
Forks: 14
Open Issues: 3
Last Activity: 2020-05-22 18:08:12+00:00
Git Stats MVP Requirements

- Support GitHub and GitLab upon release
  - Will have to eventually support BitBucket

- Identify provider given URL

- Use API to download statistics
Host
Application
```python
class RepoDetails(NamedTuple):
    organization: str
    repo: str

>>> # https://github.com/alysivji/falcon-apispec
>>> project = RepoDetails("alysivji", "falcon-apispec")
>>> project
RepoDetails(organization='alysivji', repo='falcon-apispec')
```
class RepoStatistics(NamedTuple):
    id: int
    description: str
    stars: int
    forks: int
    open_issues: int
    last_activity: datetime

>>> stats = RepoStatistics(5723246, "apispec plugin for Falcon", 29, 14, 3, yesterday)

>>> stats
RepoStatistics(id=5723246, description='apispec plugin for Falcon', stars=29, forks=14, open_issues=3, last_activity=datetime.datetime(2020, 6, 14, 10, 48, 1, 165391))
url = "https://github.com/alysivji/falcon-apispec"
if "github.com" not in url.lower():
    raise ValueError("Not a valid GitHub url")

repo = url.lower().split("github.com/ ")[1]
project_url = f"https://api.github.com/repos/{repo}"
response = requests.get(project_url)

data = response.json()
result = RepoStatistics(
    id=data["id"],
    description=data["description"],
    stars=data["stargazers_count"],
    forks=data["forks"],
    open_issues=data["open_issues"],
    last_activity=data["pushed_at"],
)

print(result)
url = "https://github.com/alysivji/falcon-apispec"
if "github.com" not in url.lower():
    raise ValueError("Not a valid GitHub url")

repo = url.lower().split("github.com/")[1]
project_url = f"https://api.github.com/repos/{repo}"
response = requests.get(project_url)

data = response.json()
result = RepoStatistics(
    id=data["id"],
    description=data["description"],
    stars=data["stargazers_count"],
    forks=data["forks"],
    open_issues=data["open_issues"],
    last_activity=data["pushed_at"],
)

print(result)
url = "https://github.com/alysivji/falcon-apispec"
if "github.com" not in url.lower():
    raise ValueError("Not a valid GitHub url")

repo = url.lower().split("github.com/")[1]
project_url = f"https://api.github.com/repos/{{repo}}"
response = requests.get(project_url)

data = response.json()
result = RepoStatistics(
    id=data["id"],
    description=data["description"],
    stars=data["stargazers_count"],
    forks=data["forks"],
    open_issues=data["open_issues"],
    last_activity=data["pushed_at"],
)

print(result)
url = "https://github.com/alysivji/falcon-apispec"
if "github.com" not in url.lower():
    raise ValueError("Not a valid GitHub url")

repo = url.lower().split("github.com/")[1]
project_url = f"https://api.github.com/repos/{repo}"
response = requests.get(project_url)

data = response.json()
result = RepoStatistics(
    id=data["id"],
    description=data["description"],
    stars=data["stargazers_count"],
    forks=data["forks"],
    open_issues=data["open_issues"],
    last_activity=data["pushed_at"],
)

print(result)
Description: apispec plugin that generates OpenAPI specification (aka Swagger Docs) for Falcon web applications.

Stars: 29
Forks: 14
Open Issues: 3
Last Activity: 2020-05-22 18:08:12+00:00
Host Application

Plugin
class BasePlugin:
class BasePlugin:
    def __init__(self, repo):
        self.repo = repo

    def __repr__(self):
        return f"<{self.__class__.__name__}>"
class BasePlugin:
    def __init__(self, repo):
        self.repo = repo

    def __repr__(self):
        return f"<{self.__class__.__name__}>

@staticmethod
def check(domain) -> bool:
    raise NotImplementedError
class BasePlugin:
    def __init__(self, repo):
        self.repo = repo

    def __repr__(self):
        return f"<{self.__class__.__name__}>"

@staticmethod
def check(domain) -> bool:
    raise NotImplementedError

def repo_stats(self) -> RepoStatistics:
    raise NotImplementedError
class GitHubPlugin(BasePlugin):
class GitHubPlugin(BasePlugin):
    @staticmethod
    def check(domain):
        return domain.lower() == "github.com"
class GitHubPlugin(BasePlugin):
    @staticmethod
    def check(domain):
        return domain.lower() == "github.com"

    def repo_stats(self) -> RepoStatistics:
        project_url = f'https://api.github.com/repos/{self.repo}"
        response = requests.get(project_url)
        data = response.json()

        return RepoStatistics(
            id=data["id"],
            description=data["description"],
            stars=data["stargazers_count"],
            forks=data["forks"],
            open_issues=data["open_issues"],
            last_activity=data["pushed_at"],
        )
Host Application
class GitLabPlugin(BasePlugin):
class GitLabPlugin(BasePlugin):
    @staticmethod
    def check(domain):
        return domain.lower() == "gitlab.com"
class GitLabPlugin(BasePlugin):
    @staticmethod
    def check(domain):
        return domain.lower() == "gitlab.com"

    def repo_stats(self) -> RepoStatistics:
        encoded_repo = quote_plus(self.repo)
        project_url = f"https://gitlab.com/api/v4/projects/{encoded_repo}"
        response = requests.get(project_url)
        data = response.json()

        return RepoStatistics(
            id=data["id"],
            description=data["description"],
            stars=data["star_count"],
            forks=data["forks_count"],
            open_issues=None,
            last_activity=data["last_activity_at"],
        )
Host Application
plugins = [GitHubPlugin, GitLabPlugin]
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)

    def _parse_url(self, self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
url = "https://github.com/alysivji/falcon-apispec"
client = GitApiClient(url)

stats = client.get_stats()
print(stats)
Description: apispec plugin that generates OpenAPI specification (aka Swagger Docs) for Falcon web applications.

Stars: 29
Forks: 14
Open Issues: 3
Last Activity: 2020-05-22 18:08:12+00:00
Host
Application
Host Application
class BitBucketPlugin(BasePlugin):
    @staticmethod
    def check(domain):
        return domain.lower() == "bitbucket.org"

    def repo_stats(self) -> RepoStatistics:
        project_url = f"https://api.bitbucket.org/2.0/repositories/{self.repo}"
        response = requests.get(project_url)
        data = response.json()

        return RepoStatistics(
            id=data["uuid"],
            description=data["description"],
            stars=None,
            forks=None,
            open_issues=None,
            last_activity=parse_dt(data["updated_on"])
        )
plugins = [GitHubPlugin, GitLabPlugin, BitBucketPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
plugins = [GitHubPlugin, GitLabPlugin, BitBucketPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
Recall: Plugin System Checklist

- Requires host application
- Communication channel between host and plugin
- Register with the host application
- Load plugins dynamically at runtime
- Respond when called upon by the host application
Plugin System Checklist

- Requires host application
Plugin System Checklist

- Requires host application
Plugin System Checklist

✓ Requires host application
Plugin System Checklist

- Communication channel between host and plugin
Plugin System Checklist

- Communication channel between host and plugin

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

✓ Communication channel between host and plugin

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Register with the host application
Plugin System Checklist

- Register with the host application

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

✓ Register with the host application

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

- Loaded dynamically at runtime
Plugin System Checklist

- Loaded dynamically at runtime

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlsplit(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

✓ Loaded dynamically at runtime

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

- Respond when called upon by the host application

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
            else:
                raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlsplit(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

✓ Respond when called upon by the host application

```python
plugins = [GitHubPlugin, GitLabPlugin]

class GitApiClient:
    def __init__(self, url):
        domain, self.repo = self._parse_url(url)
        for plugin in plugins:
            if plugin.check(domain):
                self.plugin = plugin(self.repo)
                return
        else:
            raise ValueError(f"{domain} not supported")

    def _parse_url(self, url):
        url_parts = urlparse(url)
        parts = url_parts.path.split("/")
        return url_parts.netloc, RepoDetails(parts[1], parts[2])

    def get_stats(self) -> RepoStatistics:
        return self.plugin.repo_stats()
```
Plugin System Checklist

✓ Requires host application
✓ Communication channel between host and plugin
✓ Register with the host application
✓ Load plugins dynamically at runtime
✓ Respond when called upon by the host application
Plugin System Checklist

- Requires host application
- Communication channel between host and plugin
- Load plugins dynamically at runtime
- Register with the host application
- Respond when called upon by the host application
Plugin Systems in the Wild
Extending Django...

- Custom Model Fields
- Custom Lookups
- Custom Storage System
- Custom Cache Backend
- Custom Tags and Templates
- Custom Management Commands
- Custom Auth
- Custom User Model
- Writing Your Own Middleware
- Django Signals
  - READ THIS FIRST
Writing Reusable Applications
Writing Custom Middleware -- Django
Middleware is a framework of hooks into Django’s request/response processing.
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
]
HttpRequest

SecurityMiddleware

SessionMiddleware

CsrfViewMiddleware

AuthenticationMiddleware

view function

HttpResponse
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
class RequestUuidMiddleware:
    """Add uuid to each request"""
class RequestUuidMiddleware:
    """Add uuid to each request"""
    def __init__(self, get_response):
        self.get_response = get_response
class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        request_id = request.headers.get("X-Request-ID")
        if not request_id:
            request_id = str(uuid.uuid4())
class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        request_id = request.headers.get("X-Request-ID")
        if not request_id:
            request_id = str(uuid.uuid4())
        request.request_id = request_id
        response = self.get_response(request)
class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        request_id = request.headers.get("X-Request-ID")
        if not request_id:
            request_id = str(uuid.uuid4())
        request.request_id = request_id
        response = self.get_response(request)
        return response
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "common.middleware.RequestUuidMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
]
class TestRequestUuidMiddleware:
    def test_request_has_uuid(self, client):
        response = client.get("/healthcheck")
        request = response.wsgi_request
        assert getattr(request, "request_id") is not None

    def test_request_does_not_have_guid_field(self, client):
        response = client.get("/healthcheck")
        request = response.wsgi_request
        with pytest.raises(AttributeError):
            getattr(request, "request_guid")

    def test_request_has_request_id_header(self, client):
        headers = {"HTTP_X-Request-ID": "abc"}
        response = client.get("/healthcheck/", **headers)
        request = response.wsgi_request
        assert request.request_id == "abc"
Extending Flask...

- Custom Commands
- Flask Extensions
- Modular Applications with Blueprints
- Custom Middleware
Writing Custom Middleware -- Flask
WSGI middleware to wrap your Flask instances and introduce changes at the layer between your Flask application and your HTTP server.
class RequestUuidMiddleware:
    """Add uuid to each request"""
class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, app):
        self.app = app
class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
from werkzeug.wrappers import Request

class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        request = Request(environ)
from werkzeug.wrappers import Request

class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        request = Request(environ)
        request_id = request.headers.get("X-Request-ID")
        if not request_id:
            request_id = str(uuid.uuid4())
from werkzeug.wrappers import Request

class RequestUuidMiddleware:
    """Add uuid to each request"""

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        request = Request(environ)
        request_id = request.headers.get("X-Request-ID")
        if not request_id:
            request_id = str(uuid.uuid4())
        environ["request_id"] = request_id
        return self.app(environ, start_response)
from flask import Flask
from middleware import RequestUuidMiddleware

app = Flask(__name__)
app.wsgi_app = RequestUuidMiddleware(app.wsgi_app)
@pytest.fixture(scope="session")
def app():
    app = Flask(__name__)
    app.wsgi_app = RequestUuidMiddleware(app.wsgi_app)

    @app.route('/
    def hello_world():
        return request.environ.get("request_id")

    ctx = app.app_context()
    ctx.push()
    yield app
    ctx.pop()

@ pytest . fixture ( scope = " session " )
def client(app):
    client = app.test_client()
    yield client

    def testMiddleware(client):
        result = client.get("/")
        assert len(result.data) > 0

    def testMiddlewareWithRequestID(client):
        result = client.get("/", headers={"X-Request-ID": "abc"})
        assert result.data == b"abc"
Extending pytest...

- Pytest Fixtures
- Hooks
pytest Fixture Model
pytest Fixture Model

- Test fixtures set up the test environment and return it to its original state
pytest Fixture Model

- Test fixtures set up the test environment and return it to its original state
- Fixtures are functions pytest runs before and after tests
  - Decorated with `@pytest.fixture`
pytest Fixture Model

- Test fixtures set up the test environment and return it to its original state.
- Fixtures are functions pytest runs before and after tests:
  - Decorated with `@pytest.fixture`
- Can inject fixtures into test function as input arguments:
  - Searches current module then `conftest.py`
pytest Fixture Model

- Test fixtures set up the test environment and return it to its original state
- Fixtures are functions pytest runs before and after tests
  - Decorated with `@pytest.fixture`
- Can inject fixtures into test function as input arguments
  - Searches current module then `conftest.py`
- Fixture Use Cases
  - Setting up database to preconfigured state; cleaning up after tests are run
  - Monkeypatching external dependency with a known value for duration of test
  - [Adding Function Arguments to pytest Fixtures](https://docs.pytest.org/en/stable/fixture.html) aka Factories as Fixtures
from my_project.app import create_app

def app():
    """Session-wide test `Flask` application.
    Establish an application context before running the tests.
    """
    app = create_app(testing=True)
    ctx = app.app_context()
    ctx.push()

    yield app

    ctx.pop()

@ pytest.fixture(scope="module")
def client(app):
    """Create Flask test client where we can trigger test requests to app""
    client = app.test_client()
    yield client
def test_ping_event(client, create_headers, subscription_payload):
    data = subscription_payload()
    headers = create_headers(data, event="ping", is_json_data=True)

    response = client.post(
        "/github/event-subscription",
        headers=headers,
        json=data
    )

    assert response.status_code == 200
pytest Hooks
Hook-based Plugins

- Hooks identify points where application can be extended
  - Developers need to think about this when designing their plugin system
Hook-based Plugins

- Hooks identify points where application can be extended
  - Developers need to think about this when designing their plugin system

- When the host program loads, the enabled plugins are registered for the hooks they care about
Hook-based Plugins

- Hooks identify points where application can be extended
  - Developers need to think about this when designing their plugin system

- When the host program loads, the enabled plugins are registered for the hooks they care about

- When hook is triggered, all functions registered for a hook get notified
Writing a pytest Hook Plugin
Writing a pytest Hook Plugin

- Figure out what you want to build
Writing a pytest Hook Plugin

- Figure out what you want to build
- Find hook we can use to implement desired behavior
## pytest Hooks: Role Call

<table>
<thead>
<tr>
<th>Bootstrapping Hooks</th>
<th>Initialization Hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>pytest_load_initial_conftests</td>
<td>pytest_addoption</td>
</tr>
<tr>
<td>pytest_cmdline_parse</td>
<td>pytest_addhooks</td>
</tr>
<tr>
<td>pytest_cmdline_main</td>
<td>pytest_configure</td>
</tr>
<tr>
<td></td>
<td>pytest_unconfigure</td>
</tr>
<tr>
<td></td>
<td>pytest_sessionstart</td>
</tr>
<tr>
<td></td>
<td>pytest_sessionfinish</td>
</tr>
<tr>
<td></td>
<td>pytest_plugin_registered</td>
</tr>
</tbody>
</table>
## pytest Hooks: Role Call

<table>
<thead>
<tr>
<th>Test Running Hooks</th>
<th>Collection Hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>pytest_runtestloop</td>
<td>pytest_collection</td>
</tr>
<tr>
<td>pytest_runtest_protocol</td>
<td>pytest_addhooks</td>
</tr>
<tr>
<td>pytest_runtest_logstart</td>
<td>pytest_collect_directory</td>
</tr>
<tr>
<td>pytest_runtest_setup</td>
<td>pytest_collect_collect_file</td>
</tr>
<tr>
<td>pytest_runtest_call</td>
<td>pytest_pycollect_makemodule</td>
</tr>
<tr>
<td>pytest_runtest_teardown</td>
<td>pytest_pycollect_makeitem</td>
</tr>
<tr>
<td>pytest_runtest_makereport</td>
<td>pytest_generate_tests</td>
</tr>
<tr>
<td>pytest_pyfunc_call</td>
<td>pytest_make_parametrize_id</td>
</tr>
<tr>
<td></td>
<td>pytest_collection_modifyitems</td>
</tr>
<tr>
<td></td>
<td>pytest_collection_finish</td>
</tr>
</tbody>
</table>

---

**pytest Hooks: Role Call**

- **Test Running Hooks**
  - pytest_runtestloop
  - pytest_runtest_protocol
  - pytest_runtest_logstart
  - pytest_runtest_setup
  - pytest_runtest_call
  - pytest_runtest_teardown
  - pytest_runtest_makereport
  - pytest_pyfunc_call

- **Collection Hooks**
  - pytest_collection
  - pytest_addhooks
  - pytest_collect_directory
  - pytest_collect_collect_file
  - pytest_pycollect_makemodule
  - pytest_pycollect_makeitem
  - pytest_generate_tests
  - pytest_make_parametrize_id
  - pytest_collection_modifyitems
  - pytest_collection_finish
pytest Hooks: Role Call

Reporting Hooks

```python
pytest_collectstart
pytest_make_collect_report
pytest_itemcollected
pytest_collectreport
pytest_deselected
pytest_report_header
pytest_report_collectionfinish
pytest_report_teststatus
pytest_terminal_summary
pytest_fixture_post_finalizer
pytest_fixture_setup
pytest_warning_captured
pytest_runtest_logreport
pytest_assertrepr_compare
pytest_assertion_pass
```
pytest Hooks: Role Call

Debugging / Interaction hooks

pytest_internalerror
pytest_keyboard_interrupt
pytest_exception_interact
pytest_enter_pdb
TODO make picture
def pytest_addoption(parser):
    parser.addoption("--fast",
                    action="store_true",
                    default=False,
                    help="Exclude tests marked as slow",
                    )
def pytest_addoption(parser):
    parser.addoption("--fast",
        action="store_true",
        default=False,
        help="Exclude tests marked as slow",
    )

def pytest_collection_modifyitems(items, config):
    """Deselect tests marked as slow if --fast is set."""

    if config.option.fast is False:
        return

    selected_items = []
    deselected_items = []

    for item in items:
        if item.get_closest_marker("slow"):
            deselected_items.append(item)
        else:
            selected_items.append(item)

    config.hook.pytest_deselected(items=deselected_items)
    items[:] = selected_items
pytest

Test session starts

Platform: darwin -- Python 3.8.1, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
Rootdir: ~/third-party-plugins/pytest-plugins, Inifile: pytest.ini

Collected 2 items

tests/test_main.py .. [100%]

2 passed in 1.02s

```python
def config_option(*args, **kwargs):
    if config_option.fast is False:
        return

    selected_items = []
    deselected_items = []

    for item in items:
        if item.get_closest_marker("slow"):
            deselected_items.append(item)
        else:
            selected_items.append(item)

    config.hook.pytest_deselected(items=deselected_items)
    items[:] = selected_items
```
pytest

============== test session starts ===============
platform darwin -- Python 3.8.1, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
rootdir: ~/third-party-plugins/pytest-plugins, inifile: pytest.ini
collected 2 items

tests/test_main.py ..

============== 2 passed in 1.02s ===============

pytest --fast

============== test session starts ===============
platform darwin -- Python 3.8.1, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
rootdir: ~/third-party-plugins/pytest-plugins, inifile: pytest.ini
collected 2 items / 1 deselected / 1 selected

tests/test_main.py .

=============== 1 passed, 1 deselected in 0.01s ================

Writing Your Next Plugin
Is there a plugin system?
Find and copy existing plugins
Sandbox Development Environment
Add breakpoints to host application code
Add logging
CONFORMITY
Sometimes it's just easier not to fit in.
Testing Plugins -- Integration Tests
Testing Plugins -- Testing Matrix
Let’s Recap!
Plugins are software components that extend or enhance an existing program.
Plugin System Checklist

- Requires host application
- Communication channel between host and plugin
- Register with the host application
- Load plugins dynamically at runtime
- Respond when called upon by the host application
Tips for writing your next plugin

- Read the documentation
- Find and copy existing plugins
- Create a Sandbox Development Environment
- Add breakpoints in host application codebase
- Add logging
- Test using integration test
- Test all versions you plan to support
TAKE A STEP BACK
Breathe
Hard Problem

- #1 Easier
- #2 Easier
- #3 Easier

Easy  Easy  Easy  Easy  Easy  Easy  Easy
YOU GOT THIS
Resources -- Videos

- Darlene Wong: [How to Write Pytest Plugins](#)
- Doug Hellman: [Extending Your Applications with Plugins](#)
- Floris Bruynooghe: [The hook-based plugin architecture of py.test](#)
- Raphael Pierzina: [Advanced pytest](#)
- Rose Judge: [Plug-in to Python](#)
- Sandi Metz: [Go Ahead, Make a Mess](#)
Resources -- Websites / Blogs

- Django Docs: Index
- Flask Docs: Extension Development
- pluggy: the pytest plugin system
- pytest Docs: Writing plugins
- Omar Elgabry: Plug-in Architecture
- Stevedore Documentation
Resources -- Books


- “Gang of Four”. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. 1st ed. Boston, MA: Addison-Wesley Professional
Thank You

Github: alysivji/talks

Twitter: @CaiusSivjus

Blog: https://alysivji.github.io

Acknowledgements (Easter Egg)

- ChiPy

- AS, ES, CF, CL, TD, LG, SI, JO, RB, AS
Thank You

Github: alysivji/talks

Twitter: @CaiusSivjus

Blog: https://alysivji.github.io

Appendix

Slides below here do not fit into current form of presentation.
Plugin Development Tip #1

Sandbox Development Environment
Extending pytest...

<table>
<thead>
<tr>
<th>Fixtures</th>
<th>Hooks</th>
</tr>
</thead>
</table>

Extending pytest...

<table>
<thead>
<tr>
<th>Fixtures</th>
<th>Hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>- reusable test code</td>
<td></td>
</tr>
<tr>
<td>- operate within test functions</td>
<td></td>
</tr>
</tbody>
</table>
Extending pytest...

**Fixtures**
- reusable test code
- operate within test functions

**Hooks**
- customize how pytest works
- add new functionality to pytest
Ending

Like everything else in programming, once we deconstruct the problem into smaller chunks, we can reason about implementation details clearly.

We assume things are more difficult than they appear. This is especially true for problems we have not seen before.
Key Notes

- there are design patterns (strategy) we can use, but they all follow the same theme
- Aside: Design Patterns provide a blueprint
  - The design we built uses the strategy pattern
  - Strategy pattern
- break down concepts to primary parts of Object Oriented Programming
  - Design patterns make sense when you think of underlying components
- Abstract Base Classes make sense, but it’s beyond the scope of this talk
  - If you do make a plugin system, use ABCs