Arquitetura de microkernel (sistema de plugins)

Live de Python # 297

Não iremos falar sobre sistemas operacionais...

Aviso

Não iremos falar sobre sistemas operacionais...

Embora seja por causa deles que essa arquitetura tenha esse nome.

Aviso

Roteiro

1. Definindo a arquitetura

Entendendo o microkernel

2. Pluggy

Biblioteca python para plugins

3. Mão na massa

Um pouco das possibilidades

4. Coisas mais...

trade-offs, recomendações, ...

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alfredo Neto, Alynnefs, Alysson Oliveira, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Artur_farias_, Aurelio Costa, Azmovi, Belisa Arnhold, Beltzery, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Canibasami, Caoptic, Carlos Gonçalves, Carlos Henrique, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, Darcio Alberico, Darcioalberico_sp, David Couto, David Frazao, Dh44s, Diego Guimarães, Dilan Nery, Dunossauro, Edgar, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fábio Belotto, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giuliano Silva, Glauber Duma, Gleidson Costa, Gnomo Nimp, Grinaode, Guibeira, Guilherme Felitti, Guilherme Ostrock, Gustavo Pedrosa, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Idlelive, Igor Souza, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, João Pena, Joao Rocha, Joarez Wernke, Jonas Araujo, Jonatas Leon, Jonatas_silva11, Jose Barroso, Joseíto Júnior, José Predo), Josir Gomes, Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juli4xpy, Juliana Machado, Julio Cesar, Julio Franco, Julio Gazeta, Julio Silva, Kaio Peixoto, Kakaroto, Killfacept, Knaka, Krisquee, Laraalvv, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Polo, Lucas Schneider, Luciano_ratamero, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Marina Passos, Mate65br, Mateusamorim96, Matheus Mendez, Matheus Vian, Medalutadorespacialx, Michael Santos, Mlevi Lsantos, Murilo Carvalho, Nhambu, Oopaze, Otávio Carneiro, Patrick Felipe, Programming, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Shinodinho, Shirakawa, Sommellier_sr, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, Thiago, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tomás Tamantini, Valdir, Varlei Menconi, Vinícius Areias, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitoria Trindade, Vladimir Lemos, Vonrroker, Williamslews, Willian Lopes, Will_sca, Xxxxxxxxx, Yannzin, Zero! Studio

Obrigado você <3

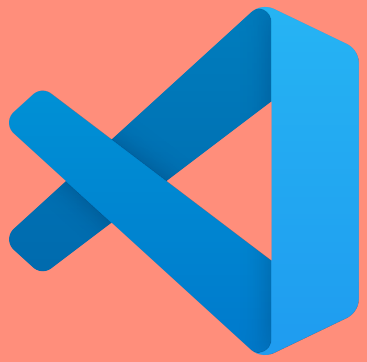
Faça parte da nossa comunidade no telegram: https://t.me/livepython

Definindo a

Arquitetura

Arquitetura de microkernel

Também conhecida como "arquitetura de plugins". Antes da definição formal, vamos a alguns exemplos de softwares que implementam a arquitetura:

Também costumamos chamar de "extensões" ou "adons".

Ou algumas bibliotecas do nosso ecossistema

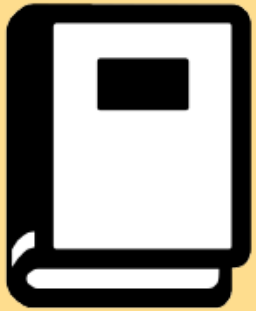
Que suportam plugins / extensões / add-ons / ...

pytest

tox

Apache Airflow

SPHINX



jupyter

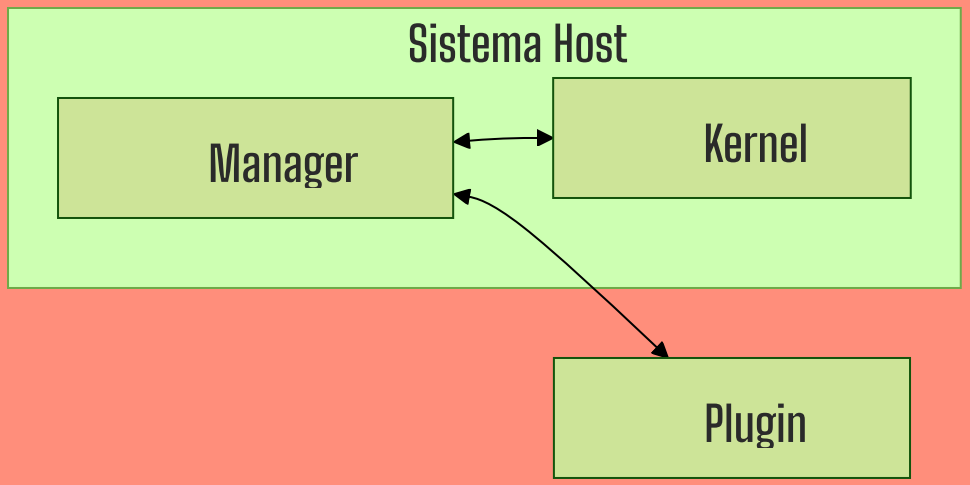
Flask

Topologia

Microkernel é uma estrutura **monolítica*** baseada em dois componentes principais.

1. O Kernel, o núcleo da aplicação; e.
2. Um gerenciador de plugins.

Onde o kernel é responsável por coordenar as chamadas de plugins.



*Não é distribuída.

Plugins?

Embora seja de senso comum o uso de plugins. É bom trabalharmos com alguma definição:

> "Plug-in, software de computador que adiciona novas funções a um programa do programa host sem alterar o próprio programa."
>
> Jonathan Sterne - plug-in (software) - Encyclopedia Britannica

Plugins?

Embora seja de senso comum o uso de plugins. É bom trabalharmos com alguma definição:

> "Plug-in, software de computador que adiciona novas funções a um programa do programa host sem alterar o próprio programa."
>
> Jonathan Sterne - plug-in (software) - Encyclopedia Britannica

Reduzindo ao nosso escopo informalmente:

> "Um pedaço de código, que sem alterar o código original, insere mais funcionalidades."

Nota Pessoal: Sobre a nomenclatura

Eu particularmente não gosto do nome "microkernel".

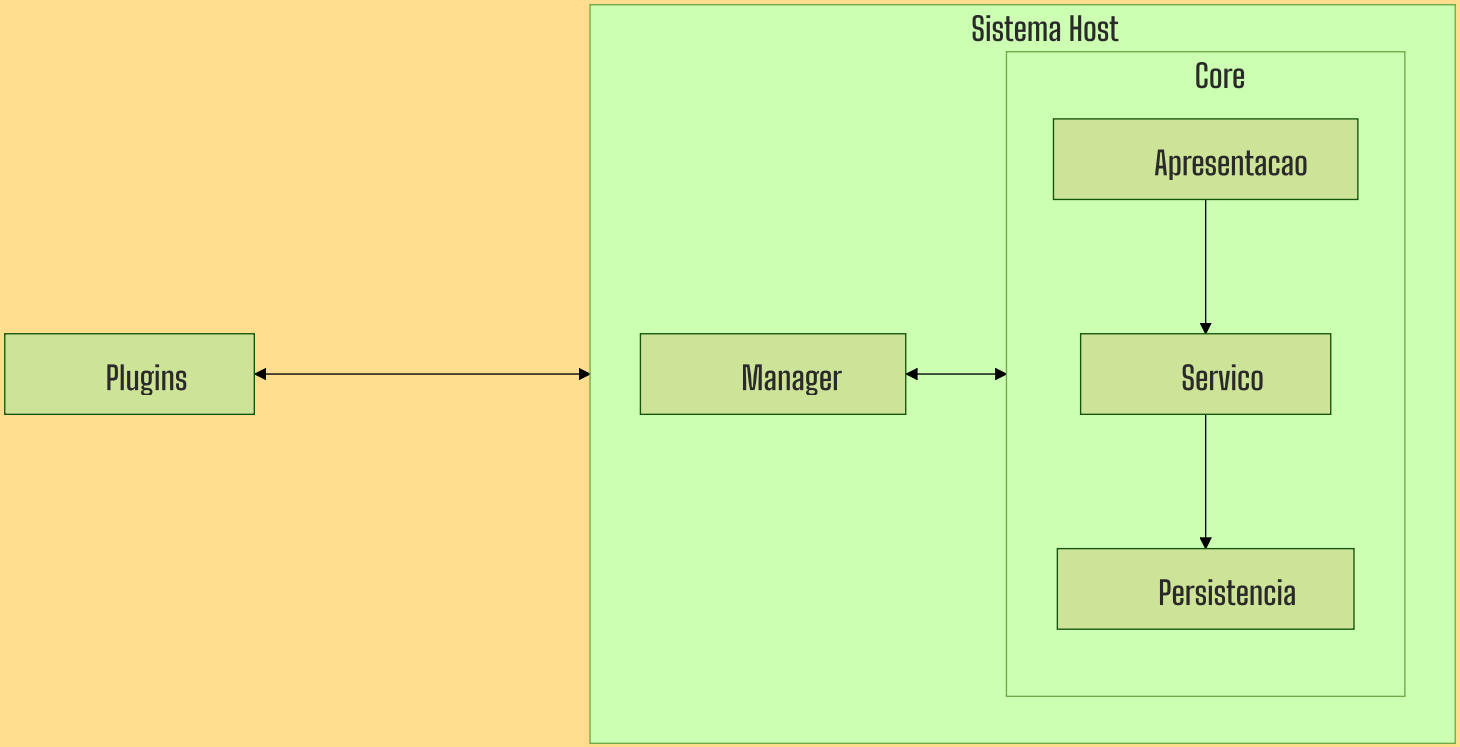
Isso nos dá a falsa impressão de que todo sistema é "micro" e tem toda sua expansão por meio de "plugins". O que não é necessariamente verdade.

Pense em um sistema `completo` e `operacional`, que funciona de forma `independente`. Mas, que pode `expandir funcionalidades` via plugins/extensões/ad-ons/...

Talvez o nome **arquitetura de plugin** faça mais jus ao que estamos tentando dizer.

Topologia

O que nos daria algo como:



Um sistema completo que leva os plugins a um nível "complementar"

Definição da arquitetura

Agora que já tiramos questões "estranhas" do caminho, podemos definir:

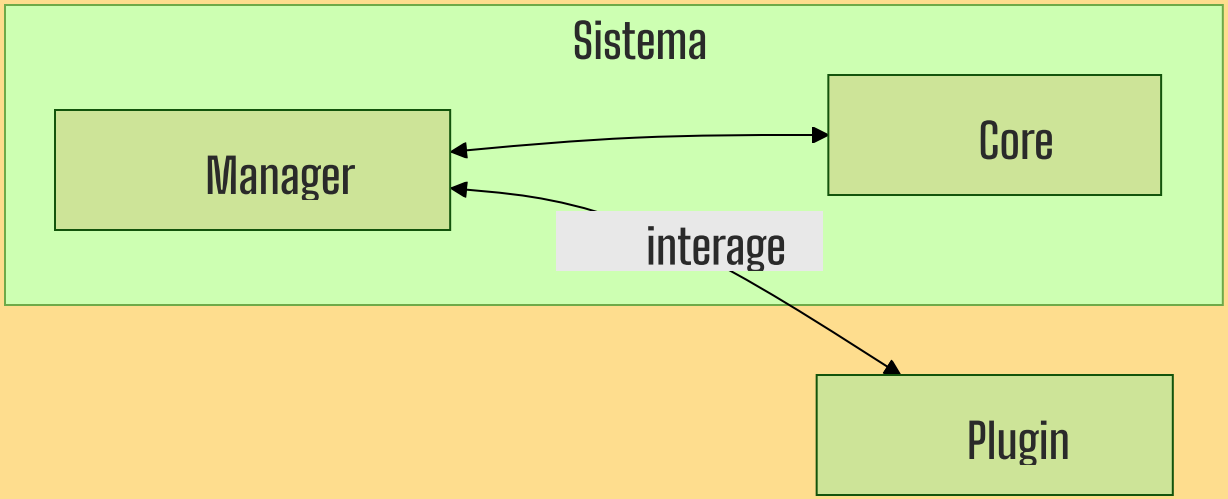
> "O estilo de arquitetura microkernel é uma arquitetura flexível e extensível que permite que um desenvolvedor ou usuário final adicione facilmente funcionalidades e recursos adicionais a um aplicativo existente na forma de extensões, ou "plug-ins", sem impactar a funcionalidade principal do sistema."

Mark Richards - Software Architecture Patterns, 2nd Edition - 2022

Host / Manager de plugins

O gerenciador de plugins tem algumas responsabilidades claras, como:

- Definir contratos de plugins
- Registrar plugins
- Ser a interface de chamada dos plugins
- Encontrar plugins



Contratos

Durante a criação de plugins, o manager deve fornecer uma interface conhecida e uniforme, na qual os plugins serão implementados. Ou seja, um grupo definido de:

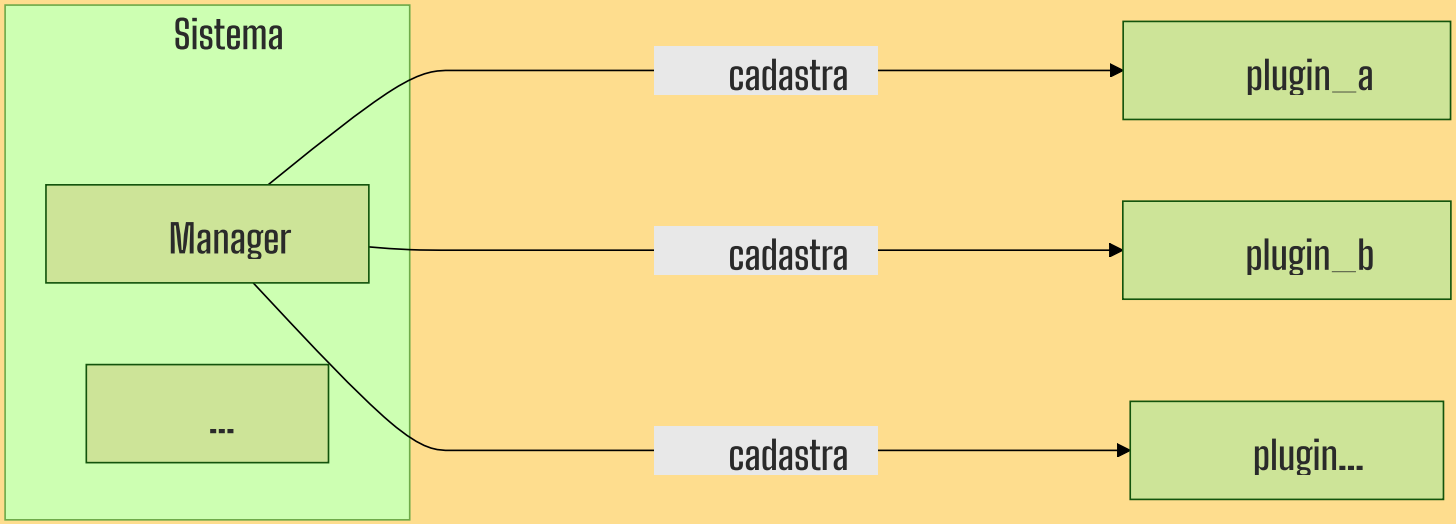
- Funções / Classes
- Módulos
- Tipos
- ...

```python
@plugin_spec
def transferencia(src: User, dst: User) -> bool: ...

@plugin_spec
def parse(data: str) -> str: ...

@plugin_spec
class Xpto: ...
```

Registro de plugins

O manager cadastra os plugins que serão usados na aplicação:

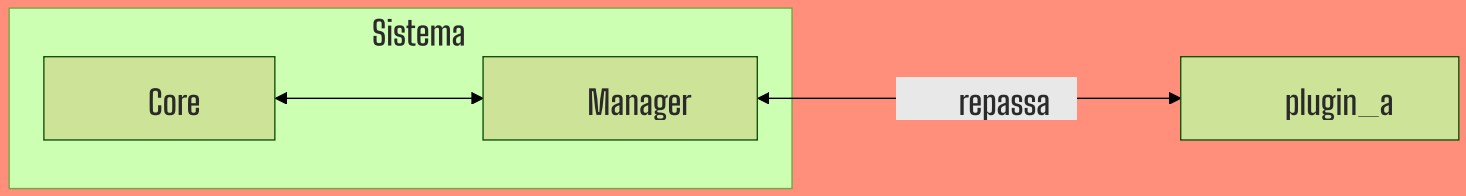
```
pm = PluginManager()

# Registro dos plugins
pm.register(plugin_a)
pm.register(plugin_b)
```



Interface de chamadas

Todas as chamadas passam pelo manager, algo como:

```
pm = PluginManager()

# Registro dos plugins
pm.register(plugin_a)
pm.register(plugin_b)

# se `a` e `b` implementarem parse, os dois serão chamados
def host_parse_func():
    results = pm.parse(data)
    ...
```



Descoberta dos plugins

Não existe uma forma única de fazer a descoberta e cada gerenciador de plugins implementa isso de forma diferente. Algumas formas comuns:

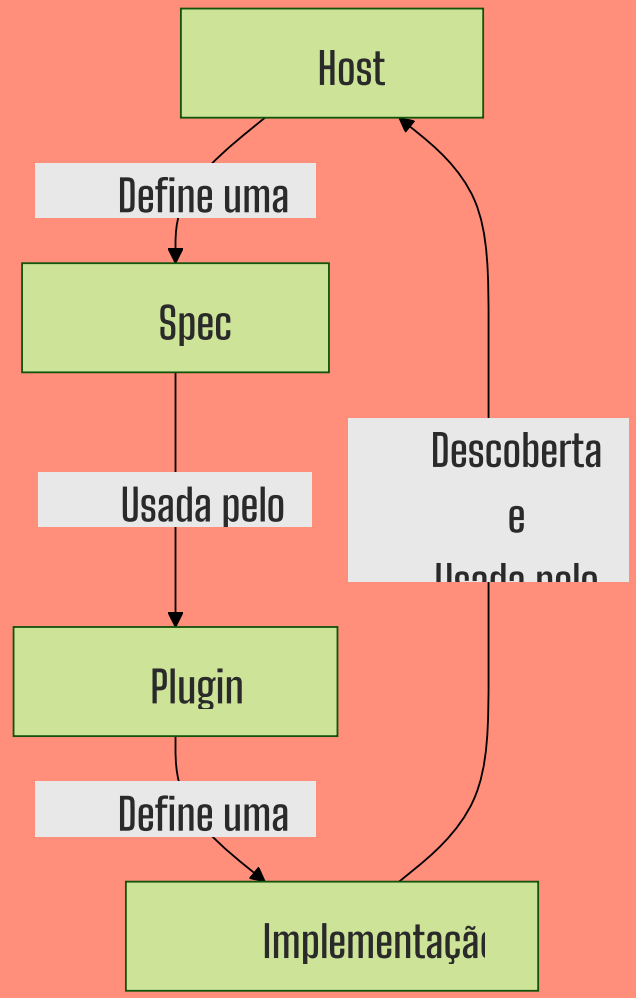
- Eles podem estar disponíveis em diretórios
  - Fora do src
  - Dentro do src
  - ...
- Eles podem estar no seu sitepackage (venv)
- Eles podem ter um package manager do app
- ...

Geralmente existe um `namespace` que engloba o nome da sua aplicação, algo como:

`meuapp.plugin`, `meuapp_plugin`, ...

Ou então um arquivo de configuração um ini/toml/yaml/...

Formando um fluxo parecido com este



Fluxograma roubado do Travis Hathaway Writing Plugin Friendly Python Applications

Os plugins

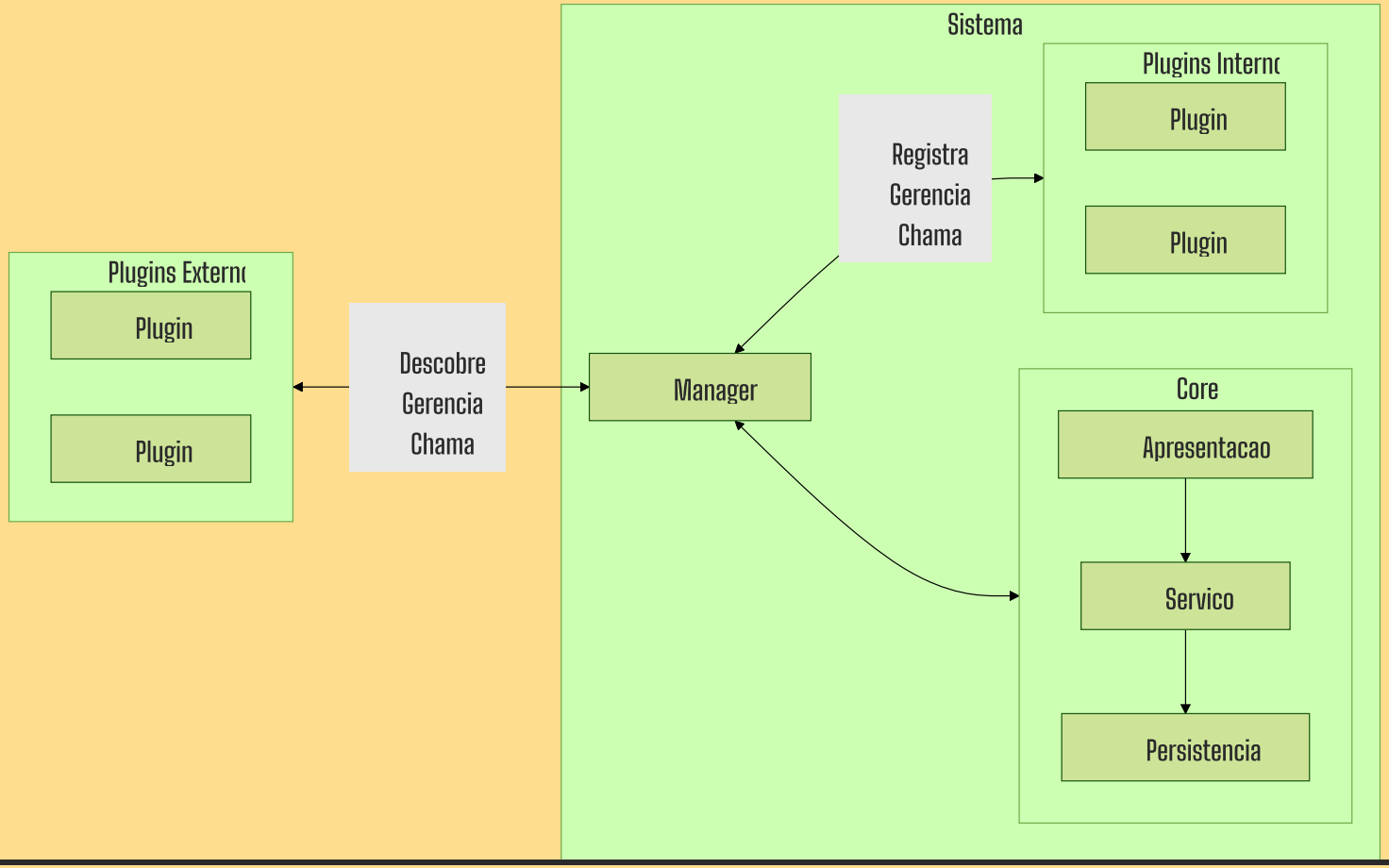
Os plugins são códigos que implementam os specs (contratos) do kernel. Algo como:

```python
# Spec no core
@plugin_spec
def clean(data: str) -> str: ...

# ---

# Implementação
@plugin_impl
def clean(data: str) -> str:
    return ''.join(t for t in data.strip().split() if t not in DENY_LIST)
```

Eles podem tanto serem parte do código funcional da aplicação. Como um módulo, ou serem instalados à parte.

De volta à topologia

Resultando em algo como...



Dinâmicos / Estáticos / Tardios

O Manager pode interagir com os plugins de diversas formas.

- Ao nível de compilação: uma distribuição do software com os plugins
- Ao nível de instalação: eles podem ser instalados a parte

Os plugins podem ser "plugados" de formas diferentes:

- Estática: A inicialização do software encontra e carrega os plugins
- Dinâmica: Podem ser habilitados e desabilitados em tempo de execução
- Tardia (lazy): Carregados durante a ação

Quando usar isso?

Existem diversos cenários onde essa arquitetura brilha.

- **Sistemas modulares**: um sistema com partes que podem ser desenvolvidas separadamente
- **Sistemas multiplataformas**: que têm partes específicas de código que mudam em sistemas diferentes
- **Partes do sistema que mudam muito**: nlocos que interagem com cosias externas e que podem mudar
- **Sistemas extensíveis**: que precisam ser estendidos por outros times ou então pela comunidade
- E muito mais...

Tem usos diversos...

Variações da arquitetura

Existem diversas implementações conhecidas de plugins. Algumas baseadas em:

- Microsserviços: Onde os serviços podem ser plugados e desplugados via containers externos. Se comunicando via HTTP / RPC
- Eventos(ldp#294): Onde diferentes subscribers são alocados como plugins
- Pipe and Filter: Onde existem direcionamentos dinâmicos via plugins.

A existência de plugins geralmente é complementar a qualquer arquitetura.

Como fazer isso em python?

Existem diversas formas de aplicar essa arquitetura em python. Usando python puro ou até mesmo bibliotecas externas.

- **importlib**: pode varrer bibliotecas instaladas no sistema e/ou importar módulos dinamicamente. `importlib.metadata.entry_points` (3.10+)
  - Isso por si só valheria uma live sobre **importlib**
- **entrypoints**: biblioteca que tem o comportamento da importlib em 3.10-.
- **stevedore**: sistema de plugins feito no topo da importlib.
- **pluggy**: sistema de plugins largamente adotado na comunidade (pytest, tox, ...)
- ...

Sistema de plugins com

Pluggy

Pluggy

"Um sistema de plugins minimalista pronto para produção"*

- **Primeira release**: 2015
- **Release atual**: 1.6.0 (maio de 2025)
- **Licença**: MIT
- Mantido pelo time do **Pytest**
- **286.419.501** de downloads no último mês
- **Campanha**: https://opencollective.com/pytest

Para instalar:

```
pip install pluggy
```

*Descição do git

Pluggy

O Pluggy fornece os componentes "básicos" de que precisamos para criar um sistema de plugins de forma simples e bastante eficiente.

Vamos nos concentrar inicialmente em 3 objetos:

- `pluggy.PluginManager`: faz o manuseio do dos plugins:
  - Registrar
  - Encontrar
  - Intercepta as chamadas
- `pluggy.HookspecMarker`: Cria um "namespace" para o host e decora funções de especificação
- `pluggy.HookimplMarker`: Usa o "namespace" nos plugins e decora as funções de implementação

Um exemplo inicial [0]

Beleza... Decidimos que algo no nosso sistema vai ser um plugin. Algo como uma função de validação de dados:

```python
import pluggy

hookspec = pluggy.HookspecMarker('app_bolado')


class BoladoSpec:
    @hookspec
    def validate(self, data): ...
```

A classe `BoladoSpec` é um namespace (ldp#238), a especificação é no método `validate`, que foi decorado por `hookspack`.

Um exemplo inicial [1]

Agora que já temos uma especificação, podemos criar alguns plugins que implementam a spec:

```python
import pluggy

hookimpl = pluggy.HookimplMarker('app_bolado')
```

```python
class PluginBolado:
    @hookimpl
    def validate(self, data):
        print(f'Bolado - {data}')
```

```python
class PluginMaisBolado:
    @hookimpl(specname='validate')
    def outro_nome(data):
        print(f'MaisBolado - {data}')
```

As classes `Plugin*Bolado` são classes somente para o namespace (ldp#238). O que é decorado pelo `hookimpl` são de fato as coisas que serão chamadas.

Um exemplo inicial [2]

Agora que temos especificações e plugins, podemos brincar com o Manager:

```
# Inicia o manager no projeto
pm = pluggy.PluginManager('app_bolado')

# Adiciona a spec
pm.add_hookspecs(BoladoSpec)

# Registra os plugins
pm.register(PluginBolado)
pm.register(PluginMaisBolado)

# chama o `validate` dos plugins
pm.hook.validate(data='Live de Python')
```

Com isso, os valores devem aparecer no shell *-*

Como testar isso?

Os testes são extremamente simples, pois você pode chamar os hooks diretamente:

```
def test_hook_xxx():
    plugin = PluginBolado()
    result = plugin.validate(data='test')

    assert result == 'test'
```

O decorador de `hookimpl` não altera a execução do método/função

Opções para specs

As especificações podem ter opções bastante valiosas. Que facilitam o desenvolvimento:

- `firstresult=True`: executa até o primeiro que retornar diferente de `None`
- `historic=True`: garante que o hook seja chamado imediatamente para plugins registrados agora e no futuro.
- `warn_on_impl=Warning(...)`: marca um hook como deprecado
- `warn_on_impl_args={...}`: marca parâmetros como deprecados em hooks

O uso seria algo como:

```
@hookspec(historic=True)
def validate(data):
    print(f'Bolado - {data}')

pm.hook.validate.call_historic(kwargs={'data': 'a'})
```

Opções para implementações

As implementações também tem opções bastante valiosas:

- `wrapper=True`: indica que a implementação do hook é um wrapper, ou seja, envolve a execução do hook original.
- `tryfirst=True`: tenta executar esta implementação primeiro, antes das outras.
- `trylast=True`: tenta executar esta implementação por último, após as outras.

Aqui um exemplo de wrapper para toda a validação:

```
@hookimpl(wrapper=True, specname='validate')
def validate_wrap(self, data):
    print('Antes')
    yield data
    print('Depois')
```

Monitoramento do manager

O manager pode validar coisas antes e depois da execução dos hooks:

```
def before_hook(hook_name, hook_impls, kwargs):
    print(f'Antes do hook "{hook_name}". Argumentos: {kwargs}')


def after_hook(result, hook_name, hook_impls, kwargs):
    print(f'Depois do hook "{hook_name}". Resultado: {result}')


pm.add_hookcall_monitoring(before_hook, after_hook)
```

Descoberta de plugins externos

Tudo que fizemos até agora foi baseado no registro de plugins internos. Mas, como o `pluggy` descobre os plugins externos?

```
pm = pluggy.PluginManager('app_bolado')
pm.add_hookspecs(hookspecs)
pm.load_setuptools_entrypoints('app_bolado')
```

Isso implica em termos pacotes instalados que tenham `entry-points` nomeados com `app_bolado`:

```
# pyproject.toml
[project.entry-points.app_bolado]
plugin_externo_bolado = "bolado:plugin"
```

Referência packing pyproject.toml pipa. Também pode ser usado o `setup.py`, você pode procurar depois :)

Mão na massa

Vendo / criando algo

Alguns projetos

Para sairmos da teoria e implementar algo minimamente legal. Poderíamos olhar os fontes de diversos projetos que usam o `pluggy`:

- pytest: src/_pytest/hookspec.py - src/_pytest/config/__init__.py
- tox: src/tox/plugin/spec.py - src/tox/plugin/manager.py
- devpi: server/devpi_server/hookspecs.py - server/devpi_server/main.py

Poderíamos tentar implementar algo em algum desses projetos, mas venho trabalhando no `sociopyta`, que se baseia em pluggy e tem um código bem menor do que isso.

Então, queria guiar vocês por esse caminho:

https://dunossauro.codeberg.page/sociopyta/

Vamos implementar um plugin de POST HTTP

Algo bem simples inicialmente:

```python
# __init__.py
from pluggy import HookimplMarker

hookimpl = HookimplMarker('sociopyta')


@hookimpl
def post(text, images, config) -> tuple[str, str] | None:
    print('POST HTTP - Funcionou')
    return 'http_post', True
```

Pois nossa maior preocupação agora será na criação do pacote. Para que vocês possam fazer plugins para qualquer outro projeto.

Criação do pacote

Algo extremamente simples seria trabalhar assim:

```
[project]
dependencies = [
  "pluggy",
  "sociopyta @ git+https://codeberg.org/dunossauro/sociopyta.git"
]

[project.entry-points.sociopyta]
http_post = "sociopyta_http_post_plugin"

[build-system]  # poderia ser qualquer outro :)
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"
```

Um laboratório

Só pra gente testar o que acontece :)

```
poetry new --flat projetinho
poetry add git+https://codeberg.org/dunossauro/sociopyta.git
poetry add caminho/do/plugin
```

Testando de fato:

```
$ sociopyta post --text "Auuuu"

POST HTTP - Funcionou
[('http_post', 'True')]
```

trade-offs,
recomendações e

Coisas Mais

Observações gerais

Se você quiser trabalhar com o Manager, algumas observações:

- Crie um Factory para ele:
  - Vai facilitar as chamadas em N lugares do código.
  - Unifica o lugar onde os plugins são registrados
  - Você pode trabalhar estratégias como lazy
  - ...

Uma função deve ser suficiente:

```
def get_plugin_manager():
    pm = pluggy.PluginManager('nome_app')
    pm.add_hookspecs(hookspecs)
    pm.load_setuptools_entrypoints('nome_app')
    pm.register(plugin_interno)
    return pm
```

Integrações

Na maioria dos casos, queremos integrar coisas já existentes em um sistema de plugins. Nesses casos, alguns padrões serão muito úteis:

- **OCP** (Open-Closed Principle): compor o objeto externo em um novo
- **Wrappers**:
  - **Proxy**: manipular as chamadas que podem e não podem do externo
  - **Decorator*: criar envólucros em chamadas externas
  - **Adapter**: transformar as entradas e saídas externas nos padrões da spec

Trade-offs [0]

Embora o microkernel seja extremamente versátil e expansível, muitos problemas podem aparecer com ele:

- **Avanço do core**: conforme o sistema host avança, nem sempre os plugins **externos** acompanham as mudanças principais.
  - Atenção aos deprecateds
  - Evitar alterar APIs já estabelecidas
  - Mais atenção ao criar / alterar specs
- **Plugins externos problemáticos**: não é possível garantir que coisas que não foram feitas pelo time principal tenham a mesma qualidade. O que pode gerar um ecossistema problemático.
- **Documentação**: não que seja um trade-off, mas a atenção ao detalhes deve ser maior. Toda a comunicação precisa ser efetiva.

Trade-offs [1]

Mais alguns pontos de atenção:

- **Gerenciamento de configuração**: a configuração pode se tornar extensa, com múltiplas camadas adicionadas pelos plugins
- **"Pluginização"**: Tendência de criar mais plugins, desmodularizando partes do sistema onde não seria necessário
- **Complexidade de implementação**: implementações simples podem se tornar complicadas, já que é preciso lidar com código de plugins externos, cujo comportamento pode ser desconhecido e imprevisível.

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Referências

- devpi/devpi: Python PyPi staging server and packaging, testing, release tool. Disponível em: https://github.com/devpi/devpi/tree/main. Acesso em: 15 set. 2025.
- DUNOSSAURO. sociopyta. Disponível em: https://codeberg.org/dunossauro/sociopyta. Acesso em: 15 set. 2025.
- FORD, Neal. Fundamentals of software architecture. Rio de Janeiro, RN: Alta Books, 2022.
- pluggy — pluggy 0.1.dev96+gfd08ab5 documentation. Disponível em: https://pluggy.readthedocs.io/en/stable/. Acesso em: 11 set. 2025.
- Plug-in | Electricity, Power, Efficiency | Britannica. Disponível em: https://www.britannica.com/technology/plug-in. Acesso em: 14 set. 2025.
- pytest-dev/pytest: The pytest framework makes it easy to write small tests, yet scales to support complex functional testing. Disponível em: https://github.com/pytest-dev/pytest. Acesso em: 15 set. 2025.
- tox-dev/tox: Command line driven CI frontend and development task automation tool. Disponível em: https://github.com/tox-dev/tox. Acesso em: 15 set. 2025.
- Travis Hathaway: Writing Plugin Friendly Python Applications. , 20 jun. 2023. Disponível em: https://www.youtube.com/watch?v=d40tBcqopAI. Acesso em: 14 set. 2025
- Writing your pyproject.toml - Python Packaging User Guide. Disponível em: https://packaging.python.org/en/latest/guides/writing-pyproject-toml/. Acesso em: 15 set. 2025.