Felipe Moz / moz.felipe@gmail.com

4Linux também é consultoria, suporte e desenvolvimento.

Você já deve nos conhecer pelos melhores cursos de linux do Brasil mas a 4Linux também é **consultoria, suporte e desenvolvimento** e executou alguns dos mais famosos projetos do Brasil utilizando tecnologias "open software".

Você sabia que quando um cidadão faz uma aposta nas loterias, saca dinheiro em um ATM (caixa eletrônico), recebe um SMS com o saldo de seu FGTS ou simula o valor de um financiamento imobiliário no "feirão" da casa própria, ele está usando uma infraestrutura baseada em softwares livres com consultoria e suporte da 4Linux?

Serviços

Consultoria:
Definição de Arquitetura, tunning em banco de dados , práticas DEVOPS, metodologias de ensino on-line, soluções open source (e-mail, monitoramento, servidor JEE), alocação de especialistas em momentos de crise e auditoria de segurança.

Suporte Linux e Open Source:
Contratos com regimes de atendimento – preventivo e/ou corretivo - em horário comercial (8x5) ou de permanente sobre-aviso (24x7) para ambientes de missão crítica. Suporte emergencial para ambientes construídos por terceiros.

Desenvolvimento de Software:
Customização visual e funcional de softwares Open Source, consultoria para a construção de ambiente ágeis de Integração Contínua (Java e PHP) e mentoria para uso de bibliotecas, plataformas e ambientes Open Source. Parceiro Oficial Zend.

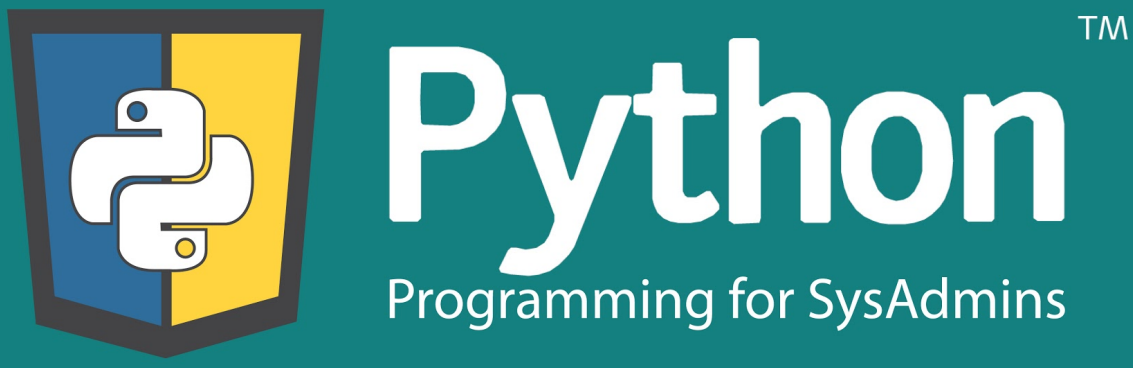
Parceiros Estratégicos

Saiba mais

(11) 2125-4769
www.4linux.com.br/suporte
contato@4linux.com.br

Felipe Moz / moz.felipe@gmail.com

Entender REST API



Curso: 521

Python Programming for SysAdmins

Versão: 4.0

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Índice

Aula 01: Trabalhando com REST API's
1.1 – Entender REST API 3
1.2 - Consumindo REST API 13
1.3 - Consumindo API com Python 23
Aula 02: Construindo uma API com Flask
2.1 - Conhecendo o Flask 32
2.2 - Estrutura básica de uma aplicação 38
2.3 - Criando views 45
2.4 – Dividindo a aplicação em Blueprints 57
Aula 03: Criando uma Model da API com MongoDB
3.1 - Entender o Model 63
3.2 - Trabalhar com o MongoEngine 70
3.3 - Criar API com Flask e MongoDB 79
Aula 04: Gerenciando Repositórios com Gitlab
4.1 - Gerenciando Repositórios com Gitlab 88
4.2 - Administrando o GitLab com Python 99
Aula 05: Administrando e Operando o Docker
5.1 - Introdução ao Docker 112
5.2 - Integrando Docker e Python 122
Aula 06: Python e Jenkins
6.1 - Introdução ao Jenkins 132
6.2 - Criando Jobs com Jenkins 139
6.3 - Jenkins e Python 152
6.4 - Criando Jobs com Python 158
Aula 07: Python e AWS com boto3
7.1 - Introdução a CloudComputing 171
7.2 - Conhecendo AWS 177
7.3 - Configurando EC2 na AWS 188
7.4 - Integração na AWS com Boto3 199
Aula 08: Fazendo conexões SSH com Paramiko
8.1 - Introdução ao SSH 212
8.2 - Conexões SSH com Python 222
Aula 09: Construindo Modulos de Logs
9.1 - Introdução a Logs 229
9.2 - Conhecendo Logs com Python 235
9.3 - Implementando logs na aplicacao 243
Aula 10: Criando Dashboard com Flask
10.1 - Criando Dashboards básicos 253
10.2 - Entendendo o Projeto 261
10.3 - Criando o Dashboard do projeto 267
Aula 11: Construindo o projeto
11.1 - Criando integração com Docker 285
11.2 - Criando integração com Gitlab 295
11.3 - Criando integração com Servidores 307
11.4 - Criando integração com AWS 315
Aula 12: Deploy da aplicação com Ansible
12.1 - Introdução ao Ansible 326
12.2 - Módulos do Ansible 333
12.3 - Playbook de Deploy 341

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Objetivos da Aula

- ✔ Entender as REST API's;
- ✔ Compreender os métodos GET, POST, PUT e DELETE.

3

4LINUX

Entender REST API

Nesta aula iremos entender o conceito de API e aprender os métodos GET, POST, PUT e DELETE.

Anotações:

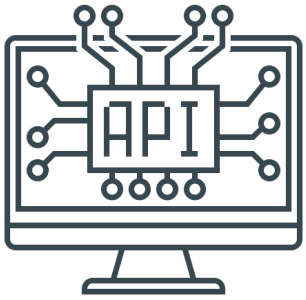
Felipe Moz / moz.felipe@gmail.com

Entender REST API

Introdução

REST API significa Representational State Transfer. É um estilo arquitetural de aplicações WEB com o objetivo de transferir dados.

Com sistemas cada vez mais integrados, é comum trabalharmos com API para fazer a troca de dados entre a aplicação e o cliente (podendo ser um usuário ou uma outra aplicação).



A transferência pode ser feita através de vários protocolos, sendo os mais comuns: SOAP e JSON.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Entendendo o SOAP

SOAP significa Simple Object Access Protocol. Esse protocolo faz a transferência de dados seguindo o formato **XML**.

XML é um formato criado pela **W3C** e significa Extensible Markup Language. É um formato bem parecido com o HTML.

Esse formato também é utilizado para o armazenamento de configurações e transferência de dados.

5 4LINUX

W3C

A W3C é uma organização cujo fim é desenvolver padrões para tecnologias Web. Para fazer esse trabalho a W3C segue uma série de processos que promovem um desenvolvimento de alta qualidade baseado em padrões sobre o consenso da comunidade de desenvolvedores.

Os processos que a W3C seguem para desenvolver esses padrões podem ser encontrados nesse link: https://www.w3.org/2015/Process-20150901/.

Caso você esteja desenvolvendo um site, a W3C também disponibiliza um validador que irá mostrar uma pontuação referente à qualidade do seu código:
https://validator.w3.org/

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Formato do XML

Um exemplo de um arquivo XML para envio de dados:

```
<?xml version="1.0">
  <configuracao aplicacao="dexterops" versao="1.0">
  <git user="devops@dexter.com.br" pass="4linux">
      <repo>git@devops.4linux.com.br</repo>
  </git>
  </configuracao>
```


XML

Apesar do XML ser bem parecido com o HTML eles foram desenvolvidos para diferentes propósitos.

Por exemplo:

- O XML foi desenvolvido para transferir dados, com foco em tentar descrever o que aquele dado significa.
- O HTML foi desenvolvido com foco em exibir dados, principalmente em como esses dados vão aparecer na tela.
- O XML não tem tags pré-definidas, já o HTML sim.

Como o XML não tem tags pré-definidas, ele se torna extensível. Então, você pode enviar um XML através de uma aplicação, solicitando o mesmo XML, porém com os dados preenchidos como, por exemplo, dados de um determinado usuário.

Pode-se enviar um XML somente com o ID do Usuário e o XML é retornado com todos os dados daquele usuário que possui o ID enviado.

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Entendendo o JSON

JSON significa JavaScript Object Notation. É um protocolo leve para transferência de dados e o seu formato tem o mesmo nome **(JSON).** Esse formato foi criado por **Douglas Crockford,** muito conhecido pela contribuição de ferramentas voltadas para JavaScript como JSLint e JSMin. Hoje esse formato JSON é o mais utilizado na criação de **REST API's** pelo seu formato limpo e facilidade na integração com AJAX.

{ json:api }

7 4LINUX

Caso você queira saber mais informações sobre Douglas Crockford, ele possui este site: http://www.crockford.com/.

Ele também realiza uma série de contribuições para o mundo de JavaScript que podem ser encontradas neste link: http://javascript.crockford.com/

Todo o processo para a criação de um arquivo JSON também pode ser encontrado neste site: http://json.org/ , onde são mostradas as estruturas de arquivos JSON, como incluir listas nesses arquivos, as implementações em várias linguagens e quais são os módulos disponíveis para você trabalhar com esse formato em cada uma dessas linguagens.

Exemplo: Em python existem os módulos:

- simplesjson
- pyjson
- ultrajson

Muitos outros podem ser encontrados também no endereço: https://pypi.python.org/pypi

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Formato do JSON

Um exemplo de um arquivo JSON para envio de dados:

```
{"config":
          {"aplicacao":"dexterops","versao":1.0},
          {"git":{
                  "user":"devops@dexter.com.br",
                  "pass":"4linux",
                  "repo":"git@devops.4linux.com.br"}
                  }
 }
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Métodos HTTP

Como a **REST API** trabalha sobre o procolo HTTP, precisamos saber os seus métodos mais comuns para enviar os dados e os status.

Métodos

POST - Criar

GET - Ler

PUT - Atualizar

DELETE - Deletar

Status

200 - Ok

401 - Não autorizado

402 - Proibido

404 - Não encontrado

500 - Erro interno no servidor

9 4LINUX

O protocolo HTTP trabalha na camada de aplicação segundo o modelo OSI. É utilizado para sistemas de informação que trabalham com hipermídia, que são vários tipos de mídia em um único lugar.

Exemplo:

Em uma página de um site pode-se incluir imagens, músicas, textos, vídeos e muitos outros recursos que o HTML permite, que é a base da internet. Para que esse dados trafeguem entre as máquinas servidores e clientes é utilizado o protocolo chamado HTTP (HyperText Transfer Protocol).

Todos os métodos desses procolos são:

- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Métodos e operações

Geralmente associamos os métodos HTTP com as operações CRUD do banco de dados, conforme o exemplo.

Métodos		Operações
POST	→	CREATE
GET	→	RETRIVE
PUT	→	UPDATE
DELETE	→	DELETE

10 4LINUX

O protocolo HTTP possui uma série de códigos de retorno.

Foram citados alguns no slide anterior, mas a classificação deles é a seguinte:

- 1xx – Mensagens informativas.
- 2xx – Mensagens de que a requisição foi bem sucedida.
- 3xx – Informações sobre redirecionamento da requisição.
- 4xx – A requisição feita pelo Client não pode ser atendida.
- 5xx – Ocorreu algum erro no servidor ao efetuar uma requisição válida.

Existe também uma implementação do protocolo HTTP com criptografia, que chamamos de HTTPS (HyperText Transfer Protocol Secure).

O HTTPS é uma implementação do protocolo HTTP com uma camada a mais de segurança que utiliza normalmente do protocolo SSL, que faz a criptografia dos dados através de certificados digitais.

Isso garante que as informações, mesmo sendo interceptadas, não possam ser lidas, garantindo o sigilo em transações bancárias, cartões de créditos, compras online e a passagem de senhas através de formulário.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Entendendo REST API

As **REST API's** são utilizadas para gerenciar recursos dentro de um servidor, então, as **URL's** geralmente são mapeadas com os nomes desses recursos, por exemplo:

/usuarios/	/grupos/	/clientes/
Representa os usuários da sua aplicação.	Representa os grupos da sua aplicação.	Representa os clientes da sua aplicação.

Os recursos são conceitos fundamentais quando se fala de API's.

Um recurso é um objeto associado a um tipo, dado associado ou um relacionamento com outros recursos. É parecido com um objeto quando trabalhamos com programação Orientada a Objetos, em que um objeto tem as suas próprias características quando é chamado. O recurso também também tem.

Exemplo:

/usuarios/1/

Pode retornar:

{“nome””teste”,”email”:”teste@teste.com”,”id”:1}

Estes são exatamente os atributos de uma classe chamada Usuário dentro da programação.

Quando temos um relacionamento entre recursos, eles podem ser representados da seguinte maneira:

/usuarios/1/compras/

Nesse exemplo, temos um relacionamento entre usuários e compras, em que na URL serão apresentadas todas as compras realizadas pelo usuário cujo ID é igual a 1.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender REST API

Entendendo REST API

Podeomos associar essas **URL's** dos recursos com os **métodos HTTP**:

URL's	Métodos	Ações
/usuarios/	POST	Criar usuário
/usuarios/	GET	Listar todos os usuários
/usuarios/1/	PUT	Atualizar um usuário
/usuarios/1/	DELETE	Remover um usuário
/usuarios/1/	GET	Listar um usuário

Normalmente trabalhamos com o tipo JSON quando usamos REST API's. No entanto, alterando o Content-Type, que é um header do protocolo HTTP que especifica qual é o tipo do dado que será retornado, pode-se utilizar os seguintes valores:

apllication/yaml – para o formato YAML
application/json - para o formato JSON
text/plain – para retornar texto puro

Existem muitos outros que também podem ser usados.
Essas configurações podem ser encontradas na internet como MIME Types.
MIME significa Multipurpose Internet Mail Extensions, que são extensões desenvolvidas para qualquer propósito, que vai desde enviar dados em tipo JSON, como em texto puro, ou até mesmo planilhas em EXCEL.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Objetivos da Aula

- ✔ Entender o RestClient
- ✔ Fazer testes utilizando métodos GET, POST, PUT e DELETE.

Consumindo REST API

Nesta aula iremos aplicar na prática os conceitos da aula anterior e criar alguns testes com os métodos HTTP.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Introdução

Vamos agora efetuar alguns testes para ver na prática como as requisições de **API's** funcionam.

No **Firefox** existe uma extensão chamada **RestClient**, que nos permite enviar requisições dos tipos GET, POST, PUT e DELETE para uma determinada URL e setar o cabeçalho da aplicação que, no nosso caso, será um **application/json**. Este é o formato de dados que vamos trabalhar.

O RestClient é uma extensão do firefox feita exatamente para testar a requisição de API's. Existe também uma extensão para o Google Chrome chamada Advanced Rest Client.

Ambas as ferramentas são muito boas para testar as API's.

Ainda é possível efetuar testes com outras ferramentas. Uma muito conhecida é um comando do linux chamado CURL:

CURL - https://curl.haxx.se/

Essa ferramenta serve para fazer as requisições de uma maneira muito parecida com o módulo requests que vamos utilizar no python.

Caso você queira fazer uma requisição utilizando o curl é só digitar o seguinte comando em um terminal linux:

```
curl 127.0.0.1:5000/usuarios/
```

Caso não seja informado o método da requisição, por padrão será enviado um método GET. Se você quiser especificar um método, pode-se utilizar o parâmetro -X.

Exemplo:

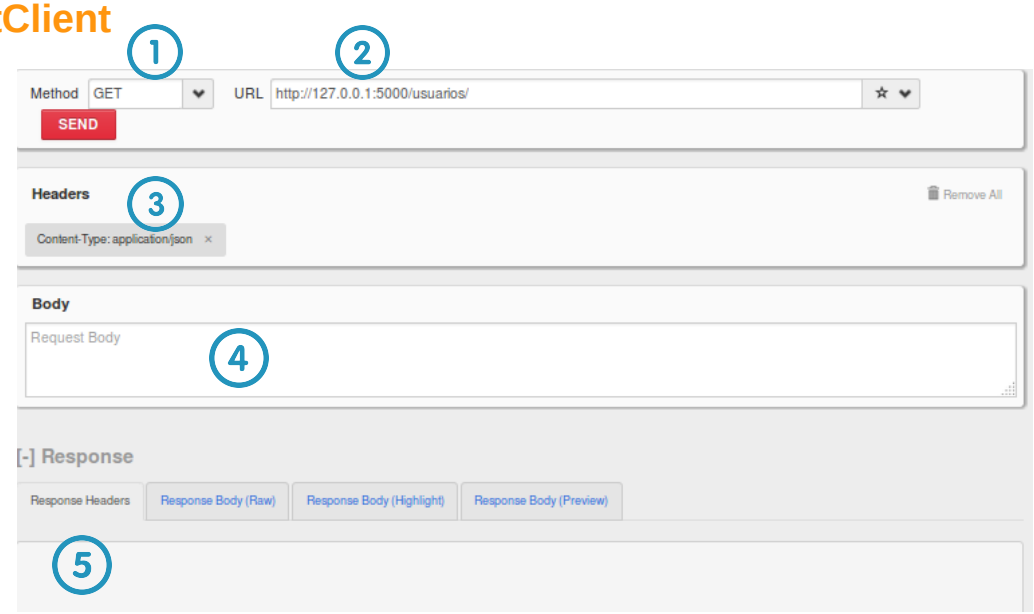
```
curl -X POST http://127.0.0.1:5000/usuarios/
```

Anotações:

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

RestClient



15 4LINUX

Número 1: A opção Method serve para definir o metodo utilizado (GET, POST, PUT, DELETE).

Número 2: A opção URL deve ser preenchida com a URL para fazer as requisições.

Número 3: A opção heards deve ser ajustada para Content-Type:application/json.

Número 4: A opção body são os dados que são enviaados para o servidor.

Número 5: A opção resposte é onde teremos o retorno dos dados do servidor (assim conseguimos saber se foi executado com sucesso ou não.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Testando o GET

O primeiro teste que vamos fazer é um **request** com o método GET na URL http://127.0.0.1:5000/usuarios/

No **body** não teremos nada, uma vez que o método GET é utilizado apenas para receber os dados.

No **Response Headers** devemos ter o código da requisição (200).

No **Response Body** devemos ter uma lista com os usuários cadastrados.

O campo Body nem sempre precisa ter um valor como, por exemplo, em conexões realizadas através do método GET.

No entanto, o Response sempre existirá sendo ele em JSON ou qualquer outro formato que você queira.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Retorno do GET

O seu **response** deve ser parecido com este, caso tenha algum usuário cadastrado. Caso não tenha, podemos cadastrar um.

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)


```
{
  "usuarios": [
    {
      "email": "alisson@4linux.com.br",
      "id": 1,
      "nome": "Alisson"
    },
    {
      "email": "rafael@4linux.com.br",
      "id": 2,
      "nome": "Rafael"
    }
  ]
}
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Cadastrando um novo usuário

Para cadastrar um novo usuário, precisamos mudar o método para **POST** e fazer um **request** na mesma URL, porém, agora precisamos adicionar dados ao **Body** e depois clicar em **Send**.

Body

```
{"nome":"Mariana Albano","email":"mariana@4linux.com.br"}
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Cadastrando um novo usuário

O retorno será o seguinte:

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```
1. {
2.     "message": "Usuario cadastrado com sucesso"
3. }
```

Agora você pode novamente fazer um **GET** na mesma URL e verá que já aparece o usuário que acabou se ser cadastrado.
Guarde o **ID** deste usuário porque vamos precisar para alterar os dados.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Atualizando um usuário

Vamos fazer agora a atualização dos dados de um usuário já cadastrado.

Na URL você precisará informar o **ID** do usuário no final. Ela ficará da seguinte forma: **http://127.0.0.1:5000/usuarios/1/**

No campo **Body** podemos preencher da seguinte forma:
{"nome":"Teste de API","email":"API@4linux.com.br"}

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Retorno da atualização

O **response** agora irá retornar o usuário com os dados já atualizados.

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```
1. {
2.     "email": "API@4linux.com.br",
3.     "id": 8,
4.     "nome": "Teste de API"
5. }
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo REST API

Remover um usuário

Vamos fazer agora a **remoção** de um usuário já cadastrado. Na URL você precisará informar o ID do usuário no final. Ela ficará da seguinte forma: **http://127.0.0.1:5000/usuarios/1/** e o método deverá ser setado como **DELETE**. O response da API será o seguinte:

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```
1. {
2.     "message": "Usuario deletado com sucesso!"
3. }
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Objetivos da Aula

- ✔ Conhecer o módulo requests;
- ✔ Fazer testes com o Python utilizando métodos GET, POST, PUT e DELETE.

Consumindo API com Python

Nesta aula iremos consumir uma API utilizando o módulo “requests”.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Módulo requests

A Linguagem Python possui módulos muito bons para consumir API's. Vamos utilizar um módulo chamado requests.

Para instalar o módulo execute o seguinte comando:

```
# pip3 install requests
```

Com ele pode-se chamar as URL's para acessar os recursos, definir os métodos, definir o Content-Type e uma série de outros elementos.

O módulo requests foi desenvolvido com o objetivo de facilitar o consumo de API's utilizando a linguagem python, tanto que o título do projeto é Requests: HTTP for Humans.

Caso você queira ver a documentação oficinal do módulo, ela pode ser encontrada no seguinte endereço:

http://docs.python-requests.org/en/master/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Módulo requests

Vamos importar o módulo requests da seguinte forma:

```
import requests
```

Para trabalhar com o requests, devemos utilizar a seguinte sintaxe:

```
requests.<METODO>
```


Caso você queira testar todos os métodos disponíveis, existe um site chamado http://httpbin.org/. Nele você pode enviar todos os métodos e ver quais são as respostas.

Exemplo:

Método POST:

print requests.post('http://httpbin.org/post', data = {'key':'value'})

Método PUT:

print requests.put('http://httpbin.org/put', data = {'key':'value'})

Método DELETE:

print requests.delete('http://httpbin.org/delete')

Método HEAD:

print requests.head('http://httpbin.org/get')

Método OPTIONS:

print requests.options('http://httpbin.org/get')

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Busca de usuário

Se eu quiser buscar todos os usuários, posso fazer da seguinte forma:

```
print (requests.get("http://127.0.0.1:5000/usuarios/"))
```

Para buscar um usuário específico, utilizamos a mesma lógica do RESTClient:

```
print (requests.get("http://127.0.0.1:5000/usuarios/1/"))
```


O retorno de ambos os comandos, será <Response [200] >, pois ele retorna somente o Status da requisição. No caso 200, esta requisição foi realizada com sucesso.

Caso você queira ver especificamente o status code de uma requisição, pode-se utilizar o método status_code.

Exemplo:
print (request.get("http://127.0.0.1:5000/usuarios/1/").status_code)

O interessante de retornar somente o status code, é que automaticamente o resultado já é convertido para valor inteiro. Então, você poderia tratar dependendo do número que for retornado.

Exemplo:

```
#!/usr/bin/python3
import requests
response = requests.get("http://127.0.0.1:5000/usuarios/").status_code
if response == 200:
    print ("A requisicao foi realizada com sucesso")
else:
    print ("Ocorreu um erro ao processar requisicao")
```

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Conteúdo da requisição

Para saber o conteúdo da requisição porecisamos chamar o método **_content** deste objeto.

```
req = requests.get("http://127.0.0.1:5000/usuarios/ ")._content

print (req)
```

Desta forma conseguimos ver o conteúdo igual ao que vimos utilizando o RESTClient.

Quando utilizamos _content para retornar o conteúdo da requisição, os usuários serão do tipo str (string) e para trabalharmos com ele devemos utilizar com dic. Para conseguir obter o retorno apenas do usuário e o seu e-mail, podemos fazer desta forma:

```
#!/usr/bin/python3
import requests
response = requests.get("http://127.0.0.1:5000/usuarios/").json()
for u in response["usuarios"]
    print (u["nome"], u["email"])
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Enviando dados

Se for necessário enviar dados (para as requisições PUT e POST) é necessário passar os seguintes parâmetros:

Data: ele deve possuir como valor um json representando os dados que devem ser enviados.

Headers: ele deve possuir como valor o content-type da requisição (igual definimos no RESTClient).

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Enviando dados

```
#!/usr/bin/python3
import requests
import json
data = json.dumps({"nome":"Nome",
                   "email":"nome@4linux.com.br"})
headers = {'content-type':'application/json'}
response = requests.post("http://0.0.0.0:5000/usuarios/",
                         data=data,headers=headers)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Editando dados

Para fazer a atualização de um usuário cadastrado, devemos alterar apenas a URL e passar os dados igual fizemos com o POST.

```
data = json.dumps({"nome":"Alterando Nome",
                   "email":"alterando_nome@4linux.com.br"})
headers = {'content-type':'application/json'}
response = requests.put("http://0.0.0.0:5000/usuarios/1/",
                        data=data,headers=headers)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Consumindo API com Python

Excluindo dados

Para deletar um usuário, basta apenas alterar o método do request para “delete” e informar na URL qual o ID do usuário que seja excluído:

```
requests.delete("http://0.0.0.0:5000/usuarios/1/")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

Objetivos da Aula

- ✔ Compreender o que é Flask;
- ✔ Entender como trabalhar com rotas e views.

Conhecendo o Flask

Nesta aula iremos conhecer o Flask e o seu funcionamento.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

> **Flask** é um microframework para aplicações Web em Python.

O **Flask** nasceu a partir de uma sátira com o framework Bottle, que tem como objetivo o desenvolvimento da sua aplicação em python em um único arquivo, Já o Flask segue o caminho contrário, fazendo com que a sua aplicação seja o mais segmentada possível.

O Flask foi desenvolvido baseado em uma biblioteca em python chamada Werkzeug (http://werkzeug.pocoo.org/), que foi criada justamente para fazer o controle de rotas. Ele também utiliza de uma outra tecnologia chamada Jinja2 (http://jinja.pocoo.org/), que é um sistema gerenciador de templates.

Caso você queira usar essas bibliotecas sem usar o Flask, é possível também.

Um exemplo de case de sucesso com Flask é o Pinterest.
Veja um post do desenvolver da API do Pinterest no seguinte link:

https://www.quora.com/What-challenges-has-Pinterest-encountered-with-Flask

Aqui ele explica que utilizam o Flask desde 2011 e que tem obtido ótimos resultados, pois nele é possível criar várias URL's de API's em poucas linhas.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

Mas por que **Microframework**?

Microframework é um conjunto de recursos essenciais para o funcionamento da sua aplicação.

Exemplos: Controle de Rotas; Templates e Views.

Conforme o seu projeto for crescendo, você pode adicionar novas extensões que te fornecem cada vez mais funcionalidades. Ou seja, você só vai instalar o que você precisa usar!

Os microframeworks é o contrário de um Full Stack Framework, pois neles temos somente os recursos essenciais para o funcionamento da aplicação.

Existem vários microframeworks para diferentes linguagens.

Alguns deles são:

Flask – Python
Sinatra – Ruby
Silex – PHP
Scalatra – Scala
NancyFX – C#/.Net

Já os Full Stack Frameworks, possuem todos os recursos já instalados, como:

- Contas
- Autenticação
- Autorização
- Regras
- ORM
- Validação de Formulários
- Web Template Engine

Em python são bem famosos dois Frameworks, sendo eles: Django e Web2Py.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

Funcionalidades

Imagine que é necessário em nossa aplicação a conexão com Banco. Isto é resolvido de forma fácil com o Flask, pois para cada necessidade você pode instalar extensões específicas.

Exemplos:

- **flask-sqlalchemy** (Faz a integração com o SQLAlchemy);
- **flask-migrate** (Gera migrations para atualizar a estrutura do banco);
- **flask-admin** (Simplifica processo de autenticação);
- **flask-mail** (Módulo para enviar e-mails pela sua aplicação).

Extensões do Flask

No próprio site do Flask existe uma página com a maioria das extensões disponíveis.

Caso você tenha curiosidade em buscar por mais do que as que foram citadas aqui é só acessar o seguinte link:

http://flask.pocoo.org/extensions/

Mas lembre-se de sempre usar somente o que você precisa para o seu projeto e evitar que sejam importadas extensões sem uso, pois isso aumenta o tempo de carregamento da sua aplicação e afeta a performance.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

Funcionamento

O Flask trabalha com rotas e views, não temos um limite para a sua criação, ou seja, podemos criar quantas forem necessárias.

As rotas são as URL's da aplicação, por exemplo:

/ - equivalente a index do site

/usuarios/ - rota de usuarios

/grupos/ - rota de grupos

Rotas

As aplicações modernas utilizam um recurso chamado URL's amigáveis. Elas recebem esse nome, pois são fáceis de decorar. Assim ,você pode acessar a página que você quer diretamente, sem passar pela index do site.

Um exemplo de como as URL's eram em sistemas mais antigos:
http://localhost/?page=contato

Atualmente:
http://localhost/contato

O flask, por padrão, já trabalha nesse segundo formato com URL's amigáveis. Essas URL's são configuradas na aplicação com o nome de rotas.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo o Flask

Funcionamento

As views são as funções que retornam os elementos que são exibidos na página, ou seja, o que o usuário irá ver.

Por exemplo, a rota /usuarios/ possui uma função chamada usuarios:

```
def usuarios():
    return "Hello World"
```

O nome da função não precisa ser o mesmo nome da rota, mas é de costume colocar o nome da função sempre de acordo com a rota.

Views

As views podem retornar vários formatos como json, xml, plain text, yaml e outros.

Quem determina a saída é o content type ao realizar as requests do tipo json, como foi visto na primeira aula.

Para retornar um XML no Flask, você pode fazer da seguinte maneira:

- Importe a classe Response do Flask:

from flask import Response

- No retorno da view coloque da seguinte maneira:

return Response(xml, mimetype="text/xml")

O XML também é bastante utilizado em API's. Então, é importante saber como retornar esse formato de dado.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Objetivos da Aula

- ✔ Entender a aplicação;
- ✔ Criar uma estrutura básica utilizando Flask.

Estrutura básica de uma aplicação

Nesta aula iremos criar a estrutura básica de uma aplicação, com as suas rotas e views.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Instalação

A instalação do Flask por ser realizada através do pip.

```
# pip3 install flask
```

O próprio Flask já traz um servidor Web interno para que você possa testar as suas aplicações sem precisar de um outro WebServer instalado.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Criando aplicação

Para criar a aplicação em Flask, precisamos apenas de algumas linhas:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def index():
    return "Hello World"

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


Esse é um exemplo de uma aplicação básica em flask.

Se você acessar a página oficial do projeto verá que existe um esquema muito parecido, pois essa é a base e todo o seu projeto pode ser desenvolvido nesse mesmo arquivo.

Porém, o Flask não prega essa filosofia. Recomenda-se que as responsabilidades sejam o mais separadas possível.

Mais a frente veremos o conceito de Blueprints, que irá nos auxiliar e na hora de modelar aplicações grandes.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Iniciando aplicação

Agora execute no terminal:

```
python3 flask_application.py
```

Em seguida, aparecerá:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Isso significa que a sua aplicação já está sendo **executada com sucesso**.
Então, é só acessar esse mesmo endereço pelo navegador e já aparecerá a mensagem.

O Flask possui um servidor de testes interno para que você possa testar a sua aplicação sem a necessidade de instalar um servidor apache ou nginx, melhorando o foco do programador no desenvolvimento da aplicação e não na configuração do ambiente.

Esse servidor pode ser alterado e usado em ambiente de produção. Uma alternativa muito boa é um servidor chamado Tornado, que também é escrito em python:

http://www.tornadoweb.org/en/stable/

Caso você queira utilizá-lo em sua aplicação, é só seguir as instruções do site oficial:

http://flask.pocoo.org/snippets/78/

Anotações:

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Ativando modo Debug

Quando trabalhamos com aplicações em **Flask** é muito comum deixar esse servidor da aplicação sendo executado enquanto estamos programando, porém, quando você fizer as alterações nos arquivos, esse servidor precisa ser reiniciado.

Para fazer com que ele reinicie automaticamente, você pode ir na linha **app.run()** e adicionar o parâmetro **debug=True**:

```
app.run(debug=True)
```


Além das configurações de modo debug, endereço do host e porta, muitas outras configurações podem ser habilitadas e desabilitadas. Todas elas estão listadas no seguinte endereço:

http://flask.pocoo.org/docs/0.12/config/

Além dos parâmetros serem passados através do construtor, como é o caso do slide, essas configurações também podem ser passadas da seguinte maneira:

```
app["DEBUG"] = True
```

Essas configurações nada mais são do que um dicionário que armazena as propriedades da aplicação em flask.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Alterando porta e endereço IP

Por padrão o Flask já executa o servidor interno na porta 5000 e somente em localhost (127.0.0.1), mas é possível alterar essas informações também. A alteração é feita na mesma linha em que colocamos o **debug=True**.

Assim, ficará da seguinte forma:

```
app.run(debug=True,host="0.0.0.0",port=80)
```

O endereço de IP precisa estar entre aspas para que o python considere uma string.

No site oficial também temos uma lista de servidores nos quais você pode executar a sua aplicação em Flask:

http://flask.pocoo.org/docs/0.12/deploying/

Uma das maneiras mais comuns de se colocar a sua aplicação em produção é instalando um servidor apache com o mod_wsgi.

Outra maneira é utilizar o servidor Nginx + Gunicorn.

O Gunicorn é um servidor HTTP já desenvolvido para Python. Todas as informações sobre ele podem ser encontradas no site oficial:

http://gunicorn.org/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Estrutura básica de uma aplicação

Criando novas rotas

Para criar uma nova rota e view, podemos fazer da seguinte forma:

```
@app.route("/usuarios/")

def index_usuarios():

   return "aqui serao listados os usuarios"
```

Agora é só salvar o seu arquivo e tentar acessar pelo navegador com a mesma URL, porém, colocando-se **/usuarios/** para ver se aparecerá essa mensagem na tela.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Objetivos da Aula

- ✔ Entender a aplicação;
- ✔ Criar as rotas da aplicação.

Criando views

Nesta aula iremos criar a estrutura da nossa API, com as rotas e funções respectivas para cada método.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando views

Criando novas rotas

Para criar uma nova rota e view, podemos fazer da seguinte forma:

```
@app.route("/usuarios/")

def index_usuarios():

   return "aqui serao listados os usuarios"
```

Agora é só salvar o seu arquivo e tentar acessar pelo navegador com a mesma URL, porém, colocando-se **/usuarios/** para ver se aparecerá essa mensagem na tela.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Lembrando que as funções nunca podem ter nomes iguais, então se você copiar e colocar uma função para aproveitar o código, não se esqueça de trocar o nome das funções. Caso fiquem iguais, a sua aplicação retornará um erro.

Não existe limite para funções e rotas dentro da sua aplicação e todas elas podem ficar no seu arquivo principal. Ou seja, você pode ter uma aplicação completa em um único arquivo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Quando se trabalha com API's, é necessário tratar as requisições de forma diferente através dos métodos que são usados. Como:

- GET
- PUT
- POST
- DELETE

Então, para a mesma rota, pode-se fazer coisas diferentes, por exemplo, /usuarios/, em que na mesma URL é possível listar todos os usuários ou cadastrar um usuário novo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Já temos a funções **index_usuarios** que responde pela rota **/usuarios/**. Quando não especificamos o método que uma rota deve receber, por padrão é entendido o GET.
Para definir um método fazemos da seguinte maneira:

```
@app.route("/usuarios/",methods=["POST"])
```

O **parâmetro methods** recebe como valor uma lista. Podemos, então, colocar mais de um. Por exemplo:

```
@app.route("/usuarios/",methods=["POST","PUT"])
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Então, podemos utilizar o **RestClient** do firefox novamente para ver se a nossa aplicação já está trabalhando da forma correta.

Deixe a view da seguinte forma:

```
@app.route("/usuarios/",methods=["POST"])
def insert_usuarios():
    return "Aqui serao inseridos os usuarios"
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Podemos passar **variáveis** através das rotas, por exemplo para ver qual usuário deve ser retornado especificamente.

Neste caso também é necessário passar o ID como parâmetro da função que vem logo na sequência.

```
@app.route("/usuarios/<id>/",methods=["GET"])
def select_usuario(id):
        return "usuario selecionado"
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Devemos criar as views referentes as operações **CRUD** com o banco de dados. Então as nossas rotas ficarão da seguinte forma:

```
#create
@app.route("/usuarios/",methods=["POST"])
#retrieve
@app.route("/usuarios/<int:id>/",methods=["GET"])
#update
@app.route("/usuarios/<int:id>/", methods=["PUT"])
#delete
@app.route("/usuarios/<int:id>/",methods=["DELETE"])
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

As funções podem seguir a seguinte sequencia:

```
def insert_usuario(): #create
def select_usuario(): #retrieve
def update_usuario(): #update
def delete_usuario(): #delete
```

A função **index** não precisa ser modificada, uma vez que ela listará todos os usuários cadastrados por padrão.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Agora que já temos as nossas views, precisamos fazer com que elas recebam dados no formato **json** e enviem dados no mesmo formato.

Vimos que o **python** tem muita facilidade em trabalhar com esse formato, uma vez que temos os dicionários, que são praticamente iguais.

O **flask** possui um módulo que faz a conversão de um dicionário para o formato JSON e envia o response para o usuário.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

Na primeira linha do nosso arquivo, existe um **import do Flask**, logo na sequência podemos fazer o import do **módulo jsonify**, que faz a conversão de um dicionário para o formato json. Então, a linha ficará da seguinte forma:

```
from flask import Flask,jsonify
```

Para testar, podemos modificar a view index da nossa aplicação em Flask.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando views

A view então ficará da seguinte maneira:

```
@app.route("/usuarios/"):

def index_usuarios():

        return jsonify({"message":"Aqui serao
                                listados os usuarios"})
```

Faça o teste novamente através do RestClient.

Você verá que a reposta virá em um formato idêntico ao que foi visto na primeira aula.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Objetivos da Aula

- ✔ Entender o que são Blueprints;
- ✔ Dividir a aplicação em Blueprints.

Dividindo a aplicação em Blueprints

Nesta aula iremos aprender o conceito de Blueprints e como podemos aplicar em nossa API.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Introdução

Fizemos as views da parte de usuários, mas precisamos gerenciar grupos também. Pode-se também criar todas as views no mesmo arquivo que irá funcionar sem problemas.

No entando, o Flask possui um recurso muito legal que permite que uma aplicação seja dividida em várias aplicações menores.

> O nome desse recurso é **Blueprints**.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Blueprints

Blueprints são **mini aplicações em Flask** que podem ser integradas à sua aplicação principal com o objetivo de modularizar o seu projeto.

A PEP20 fala sobre o Zen of Python, que é como se fosse a filosofia do Python. Nela há a frase: “Sparse is better than dense” (Esparso é melhor do que denso), ou seja, quanto mais separada a sua aplicação, melhor ela é.

O conceito de **Blueprint** do Flask segue a risca essa filosofia.

Caso você queira saber mais informações sobre a PEP20, é só acessar esse link:

https://www.python.org/dev/peps/pep-0020/

Ou pelo próprio console do python você pode digitar:

```
import this
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Criando uma Blueprint

Crie um arquivo python com o seguinte conteúdo:

```
from flask import Blueprint

grupos = Blueprint('grupos',__name__)

@grupos.route("/grupos/")

def index_grupos():

     return "Index do gerenciamento de grupos"
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Registrando a Blueprint

Agora você precisa registrar essa Blueprint na sua aplicação principal. Para fazer isso você só precisa fazer duas coisas: o import da sua blueprint e registar na sua aplicação principal.
Exemplo:

```
from grupos import grupos

#a próxima linha deve ficar abaixo de app = Flask(__name__)

app.register_blueprint(grupos)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Dividindo a aplicação em Blueprints

Registrando a Blueprint

Agora você pode replicar exatamente o mesmo conteúdo do arquivo principal na parte de grupos.

Outra coisa que deve-se fazer, é remover as rotas referentes ao usuários e criar uma Blueprint separada somente para fazer o gerenciamento desses recursos.

Não existe limite para a quantidade de Blueprints que você pode ter em seu projeto.

Então, lembre-se: “Sparse is better than dense”.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

Objetivos da Aula

- ✔ Entender o conceito de ORM;
- ✔ Compreender o Model.

Entender o Model

Nesta aula iremos conhecer o conceito de ORM e o que é Model.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

ORM significa Object Relationship Mapping. Serve para abstrair a interação entre a aplicação e o banco de dados, poupando o desenvolvedor de ter que executar comandos

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

O **Model** de uma aplicação é o equivalente ao modelo do seu banco de dados, que pode ser tanto em SQL como em noSQL.

O Banco de Dados serve para fazer a persistência dos seus dados para que possam ser utilizados posteriormente.

Como a nossa API a princípio é bem simples, só precisamos controlar usuários e grupos.

Precisamos, então, de duas entidades:

1. Usuários
2. Grupos

Quem vem de outras linguagens de programação, pode estar acostumado a colocar a regra de negócio dentro dos Models da aplicação.

No caso de projetos em Python, você terá arquivos específicos para fazer a regra de negócio e o Model fica especificamente para espelhar o seu banco de dados. Então, nele não teremos nada além das classes que irão gerar as tabelas no banco de dados.

No nosso caso, como é no SQL, teremos somente as classes que irão gerar as collections dentro do MongoDB.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

Quando trabalhamos com banco de dados, precisamos saber também fazer as **cardinalidades**. As mais comuns são:

Um para Um (1 -> 1): cada um dos dois objetos referenciam um ao outro.

Um para Muitos (1 -> N): um objeto pode referenciar vários outros objetos.

Muitos para Muitos (N -> N): vários objetos podem referenciar vários objetos.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

noSQL

Da mesma maneira que precisamos saber quais usuários pertencem à um determinado grupo através do vínculo de chaves primárias e estrangeiras, no noSQL temos como fazer isso fazendo o uso de subdocumentos, adicionando-os dentro de uma lista em um documento principal.

Exemplo:

```
Grupos:
   Nome: Analistas
   Integrantes: [Alisson,Gaby,Leandro,Liliane,Hederson]
```


Caso você queria saber mais sobre esse relacionamento entre documentos e subdocumentos, a documentação oficial do MongoDB é muito boa e completa.

No link abaixo existem alguns exemplos sobre relacionamentos um para muitos utilizando subdocumentos no MongoDB:

https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

Você pode também navegar pela documentação e ver muito mais coisas sobre o MongoDB, que é um dos bancos de dados noSQL atualmente.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entender o Model

Lembrando que o MongoDB trabalha armazenando os dados no formato JSON, o exemplo anterior ficaria da seguinte forma:

```
{
   "_id": 1,
   "nome":"Analistas",
   "integrantes":[{"nome":"Alisson"},{"nome":"Mariana"},
                  {"nome":"Gabriela"}, {"nome":"Edgar"}]
}
```


Apesar de tratarmos os dados no MongoDB da mesma maneira em que trabalhamos com o formato JSON, ele armazena esses dados em disco no formato BSON, que é um formato mais leve, economizando, assim, espaço em disco.

No link abaixo existem mais informações sobre esse formato:
http://bsonspec.org/

No site do MongoDB também existem algumas informações falando sobre o formato JSON e BSON:

https://www.mongodb.com/json-and-bson

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entender o Model

Então, nós sabemos agora que será necessário:

- Um documento chamado Grupos.
- Um subdocumento chamado Integrantes.

O **Documento Grupos** precisará somente de duas informações: nome e integrantes.
E o **Subdocumento Integrantes** também precisará de duas informações: nome e e-mail.

Agora, isso precisa ser refletido no Model.

Apesar de tratarmos os dados no MongoDB da mesma maneira em que trabalhamos com o formato JSON, ele armazena esses dados em disco no formato BSON, que é um formato mais leve, economizando, assim, espaço em disco.

No link abaixo existem mais informações sobre esse formato:
http://bsonspec.org/

No site do MongoDB também existem algumas informações falando sobre o formato JSON e BSON:

https://www.mongodb.com/json-and-bson

Anotações:

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

Objetivos da Aula

- ✔ Entender a aplicação;
- ✔ Criar uma estrutura básica utilizando Flask.

Trabalhar com MongoEngine

Nesta aula iremos conhecer o MongoEngine e construir uma aplicação com conexão no MongoDB utilizando este módulo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

MongoEngine é um ORM (Object Relationship Mapping) para MongoDB.

mongoengine

Como temos uma API simples e enviamos e recebemos dados pelo formato JSON e o MongoDB armazena esses dados no mesmo formato, podemos aproveitar e unir as duas coisas.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

Instalação do módulo

Como a API foi feita utilizando o microframework Flask, pode-se instalar uma extensão do Flask específica para trabalhar com o MongoEngine, mas ele também pode ser utilizado sozinho sem precisar do Flask.

Para instalar utilize o seguinte comando:

```
sudo pip3 install flask-mongoengine
```


Durante o curso utilizaremos uma extensão do Flask chamada Flask-MongoEngine, porém, ele pode ser utilizado sozinho. Para saber mais informações, você pode buscar no seguinte link:

http://docs.mongoengine.org/

As mesmas informações encontradas nessa documentação também podem ser aplicadas ao flask-mongoengine como, por exemplo, os tipos de dados dos campos de um documento.

Nada impede também que você utilize o módulo pymongo, como foi visto no curso Python Fundamentals.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

Criando o Model

Dentro da pasta da aplicação, crie um arquivo chamado **Model.py**. Quando utilizamos extensões do Flask, todos os arquivos são mini aplicações, então, é necessário importar o Flask e a extensão.

Por exemplo:

```
from flask import Flask
from flask_mongoengine import MongoEngine
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

Na sequência, faça a criação da sua aplicação Flask:

```
app = Flask(__name__)

# configure o banco de dados

app.config["MONGODB_SETTINGS"] = {"db":"dexter-api"}

# crie uma instância com o banco de dados

db = MongoEngine(app)
```


Como vimos, o Model é o responsável por fazer essa interação com o banco de dados, isso será feito com o MongoEngine, onde a estrutura será a seguinte:

- A **classe** é corresponde a uma collection e os atributos são as nossas "chaves".

No exemplo do slide acima, foi criada uma conexão com o banco de dados dexter-api, que será utilizado para armazenar as informações que serão inseridas através da API criada na aula 02.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

A classe **Usuarios** ficará da seguinte forma:

```
class Usuarios(db.Document):

    nome = db.StringField()

    email = db.StringField(unique=True)

    data_cadastro =

            db.DateTimeField(default=datetime.now())
```


Os tipos possíveis de campos de um documento podem ser encontrados nessa documentação do MongoEngine:

http://docs.mongoengine.org/en/latest/apireference.html#fields

Podendo eles variar desde StringField,IntField até URLField, que valida se o valor inserido é uma URL.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

No campo e-mail foi colocado um parâmetro **unique=True**. Esse parâmetro serve para dizer ao **MongoDB** que esse campo nunca poderá ser repetido, uma vez que dois usuários não podem utilizar o mesmo e-mail.

No caso do **data_cadastro**, foi passado como parâmetro: **default=datetime.datetime.now()**.

Caso não seja passada a data de cadastro, ele automáticamente irá assumir que a data é a data e hora atual.

Não esqueça de importar o módulo datetime!

Os tipos possíveis de campos de um documento podem ser encontrados nessa documentação do MongoEngine:

http://docs.mongoengine.org/en/latest/apireference.html#fields

Podendo eles variar desde StringField,IntField até URLField, que valida se o valor inserido é uma URL.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

A classe **Grupos** ficará da seguinte forma:

```
class Grupos(db.Document):

    nome = db.StringField(unique=True)

    integrantes = db.ListField()
```

Em Integrantes foi criado do tipo **ListField**, equivalente a uma lista em python.

O tipo do campo Integrantes será um db.ListField que tem como objetivo trabalhar como uma lista em python.

Então, para adicionar novos elementos podemos trabalhar do mesmo jeito que já trabalhamos com as listas.

Exemplo:

```
analistas = Grupos()
analistas.integrantes.append("mariana.albano@4linux.com.br")
```

Como parâmetro pode ser passado o db.EmbeddedDocumentField, que indica que será recebido um subdocumento e como parâmetro ele recebe uma classe que deve conter os valores do subdocumento.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Trabalhar com MongoEngine

Para testar o Model, podemos fazer desta forma:

```
u = Usuarios()
    u.nome = "Janaina"
    u.email = "janaina@4linux.com.br"
    u.save()
    g = Grupos()
    g.nome = "Analistas"
    g.integrantes.append(u)
    g.save()
```


O tipo do campo Integrantes será um db.ListField que tem como objetivo trabalhar como uma lista em python.

Então, para adicionar novos elementos podemos trabalhar do mesmo jeito que já trabalhamos com as listas.

Exemplo:

```
analistas = Grupos()
analistas.integrantes.append(“mariana.albano@4linux.com.br”)
```

Como parâmetro pode ser passado o db.EmbeddedDocumentField, que indica que será recebido um subdocumento e como parâmetro ele recebe uma classe que deve conter os valores do subdocumento.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Objetivos da Aula

- ✔ Modificar a nossa aplicação para trabalhar com o MongoDB;
- ✔ Criar métodos para listar, criar, excluir e deletar usuários.

Criar API com Flask e MongoDB

Nesta aula iremos modificar a aplicação que foi construida nas aulas anteriores para que ela trabalhe com o MongoDB (utilizando o MongoEngine) e então criar os métodos para listar, criar, excluir e deletar os usuários.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Agora que já sabemos como o **MongoEngine** trabalha, podemos juntá-lo ao **Flask** para fazer a persistência dos usuários e grupos na nossa API.

Então, vamos voltar e terminar de codificar a nossa API.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Blueprints

Até este momento temos duas **Blueprints**:

Usuarios: faz a gerencia de rotas e operações de usuários

Grupos: faz a gerencia de rotas e operações de grupos

No entando elas funcionam apenas de forma estática, ou seja, não foram programadas rotinas para fazer a inserção no banco de dados.

Vamos criar a parte de **Usuários**.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Listando os usuários

Para criamos a lista de todos os usuários, é necessário fazer o import do Model desta forma:

```
from model import Usuarios
import json
```

Iremos utilizar o módulo json pois precisamos converter os dados para este formato.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Listar os usuários

Após o import, podemos alterar a view **index_usuários** e adicionar o seguinte código:

```
users = json.loads(Usuarios.objects.to_json())
return jsonify({"usuarios":users})
```

Agora a listagem de usuários que estão cadastrados no mongodb está pronta. Como até o momento não temos nenhum usuário, vamos alterar a view para realizar novos cadastros.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Inserindo novos usuários

Podemos alterar a view **insert_usuarios** correspondente ao método POST, adicionando o seguinte código:

```
usuario = request.get_json()
user = Usuarios()
for u in usuario.keys():
   setattr(user,u,usuario[u])
user.save()
return jsonify({"message":"Usuario cadastrado!"})
```


Adicionando usuários

Podemos substituir o “for” no código acima e utilizar da seguinte forma:

user.nome = “Mariana”
user.email = “mariana.albano@4linux.com.br”

O python possui uma função builtin chamada “setattr” que é responsável por fazer a mesma coisa de forma dinâmica e sua sintaxe é:

setattr(objeto, nome, valor)

Então, se definirmos da seguinte forma:

setattr(x, escola, ‘4linux’)

O resultado será:

x.escola = valor

Utilizamos o setattr pois caso haja alguma modificação na Model e ela passe a ter mais/menos atributos, não será necessário alterar o código e também porque não precisamos ficar deninindo todos os atributos e seus valores correspondentes.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Buscar usuários específicos

Para buscar um usuário em específico, vamos alterar a função select_usuarios.

O código deve ficar desta maneira:

```
user = json.loads(Usuarios.objects(id=id).to_json())

return jsonify({“usuario”:user})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Deletar um usuário

Para deletar um usuário, vamos alterar a função **delete_usuarios**

O código deve ficar desta maneira:

```
user = Usuarios.objects(id=id).first()
user.delete()
return jsonify({“usuario”:”Usuario removido com sucesso”})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criar API com Flask e MongoDB

Update de usuário

Para fazer o update em um usuário é necessário primeiro fazer o filtro (para saber quem deve ser atualizado) e após é feita a atualização.

Vamos alterar a função **update_usuarios**:

```
usuarios = request.get_json()
user = Usuarios.objects(id=id).first()
for u in usuarios.keys():
    setattr(user,u,usuarios[u])
user.save()
return jsonify({“message”:”Usuario atualizado com sucesso!”})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Objetivos da Aula

- ✔ Fazer uma introdução ao Git;
- ✔ Entender o Gitlab;
- ✔ Criar usuários e repositórios.

Gerenciando repositórios com Gitlab

Nesta aula iremos aprender alguns comandos básicos do git e como podemos trabalhar com o Gitlab criando usuários e novos repositórios.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Introdução

Git é versionador, ou seja, um controlador de versão. Ele registra as mudanças feitas em um ou mais arquivos ao logo do tempo.

Através dele, conseguimos desenvolver projetos onde mais de uma pessoa consegue trabalhar simultaneamente sem o risco de que algum arquivo seja sobreescrito ou modificado.

Conseguimos fazer isto pois ele trata os dados como um conjunto de fotografias e sempre que salvamos nosso código é como se uma foto fosse tirada do projeto para uso posterior.

Foi criado por Linus Torvalds em 2002, quando o kernel Linux começou a utilizar um sistema chamado BitKeeper para o controle de versão. Até então eles versionavam o código usando patches de arquivos compactados.

Em 2005 a empresa responsável pelo desenvolvimento comercial do BitKeeper começou a cobrar pelo uso. A comunidade então resolveu desenvolver o próprio versionador baseado nas lições aprendidas no BitKeeper.

É uma ferramenta que possui as seguintes características: Design simples, é utilizado para projetos de pequeno e grande porte, tem um desenvolvimento não linear e trabalha com grande velocidade.

Anotações:

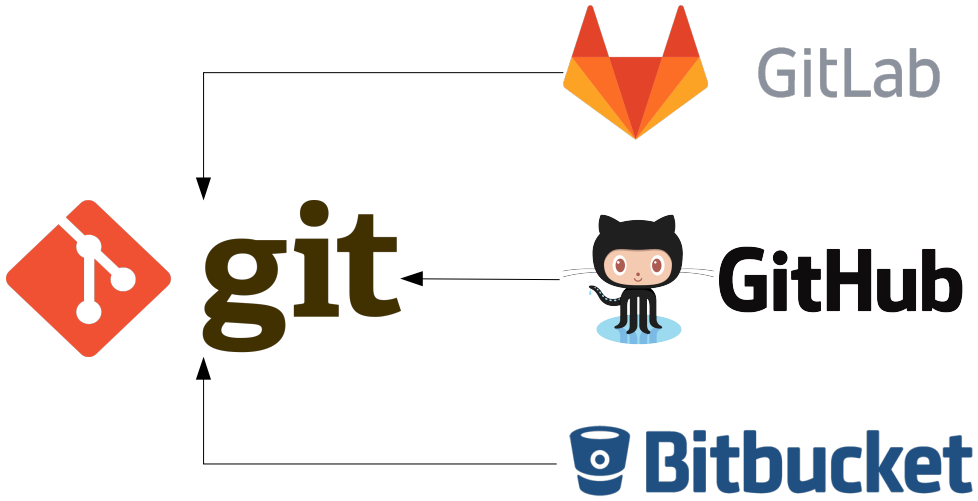
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Interfaces para o git

Todas essas interfaces utilizam o git como “motor”.



Entendendo o Git

Antes de começarmos a trabalhar com outras ferramentas, é importante entender que independente da interface utilizada (gitlab, github, bitbucket ou qualquer outra) por trás ele sempre utilizará o **git** pois essa é a ferramenta de versionamento.

Gitlab: é um gerenciador de repositórios git, possui outras funcionalidades como wiki e issues. O espaço para os repositórios é ilimitado e a quantidade de repositórios também, conseguimos importar projetos de outras plataformas (Github, Bitbucket, Gitorious, entre outras) e o armazenamento é feito em servidores privados.

Github: é uma plataforma de hospedagem de repositórios e está disponível gratuitamente (com um limite de armazenamento) e não permite que crie repositórios privados. O serviço oferece planos pagos e desta forma é possível criar repositórios privados, adicionar desenvolvedores fixos, entre outras coisas.

Bitbucket: é uma ferramenta de hospedagem de projetos, com algumas funcionalidades como: gerenciar, compartilhar ou manter o código privado, podendo trabalhar com até 5 desenvolvedores.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Introdução ao Gitlab

O **Gitlab** é uma aplicação web para gerenciar repositórios privados. Possui a mesma função do Github, porém ele permite a instalação em uma infraestrutura própria e fazer o gerenciamento do seu próprio controlador de versão.

Entendendo o Gitlab

O gitlab é um sistema de controle muito poderoso e utilizado por várias empresas grandes. Na página oficial (https://about.gitlab.com/) aparecem vários dos clientes, como:

- RedHat
- Nasdaq
- Sony
- Lego
- IBM

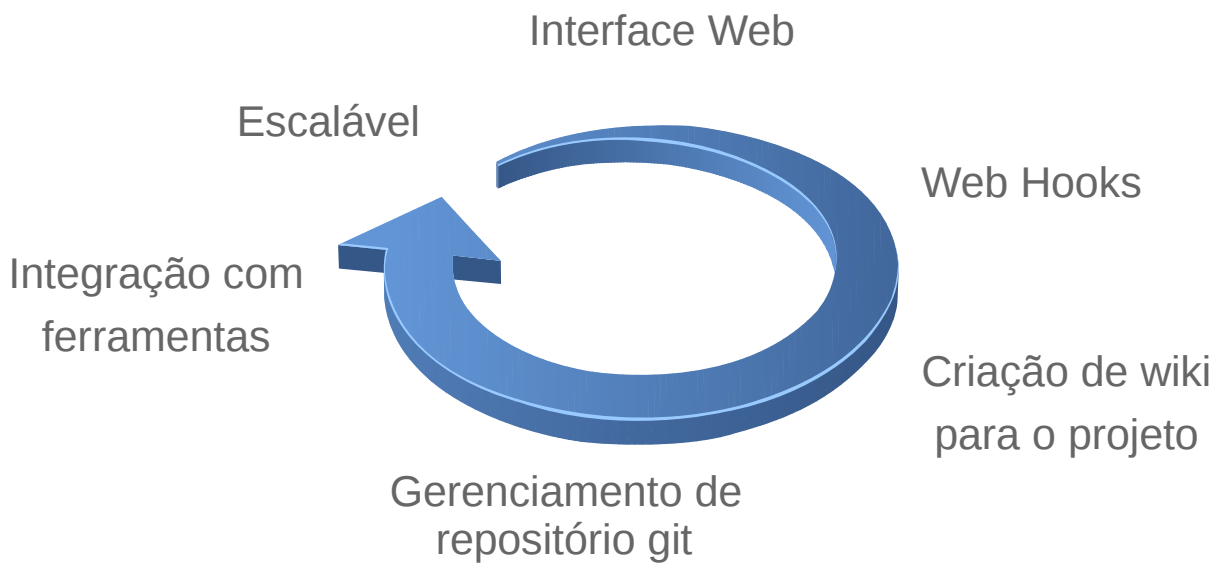
Por ser um sistema muito robusto também o utilizamos na 4linux para os ambientes do EaD internamente e colocamos no curso para mostrar como o gitlab é utilizado e quais são as possíveis integrações.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Características



Características

Além dessas características, o gitlab possui muitas outras, sendo elas listadas no seguinte link:

https://about.gitlab.com/features/

O gitlab possui uma Wiki interna para fazer a documentação dos projetos, navegador de arquivos e Stream de atividades para que você possa saber tudo que aconteceu no ambiente.

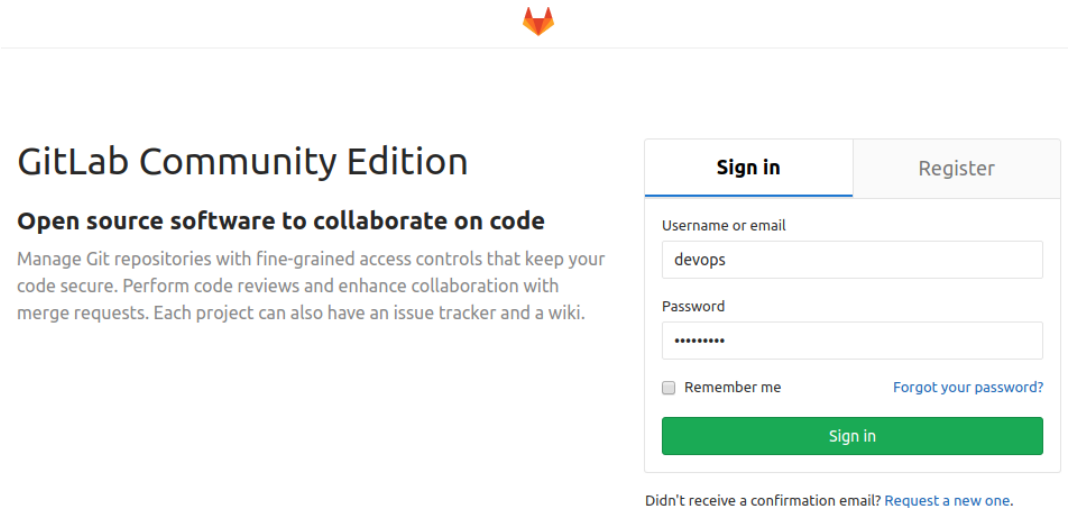
Um dos recursos mais interessantes que ele possui é o WebHooks que utilizaremos no curso para elaborar um ambiente de integração contínua totalmente baseado em ferramentas open source.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Tela inicial do GitLab onde podemos acessar com **usuário devops** e **senha 4linux123**



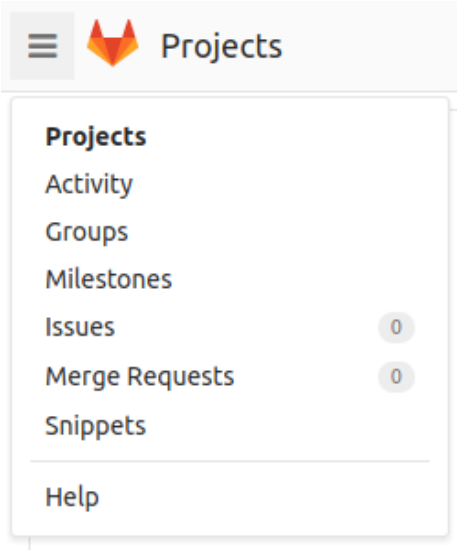
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Interface do GitLab



Projects: acesso aos projetos (repositórios)

Activity: histórico de atividades

Groups: criar e administrar grupos

Milestones: registro de marcos do projeto

Issues: registro de funcionalidades e problemas

Merge Requests: mesclar mudanças no código

Snippets: trechos de códigos armazenados no GitLab

Funcionalidades

Além de gerenciar os repositórios e versionar o código, podemos usar o GitLab para gerenciar Log de atividade (**Activity)**, onde será possível ver quando um código foi enviado ao repositório, quando um Merge foi realizado e afins.

É possível criar um marco para o projeto **(Milestones)** sendo este para resolução de algum problema ou a criação de uma nova funcionalidade. Este recurso pode estar atrelado às **Issues** e ao finalizar uma determinada quantidade de funcionalidades você alcança uma **Milestone**.

Ainda podemos utilizar o GitLab para ter um fluxo de CI/CD (Continuous Integration and Continuous Delivery) completo.

Para maiores informações sobre as funcionalidades do GitLab podemos acessar a documentação oficial: https://docs.gitlab.com/ee/user/index.html#projects

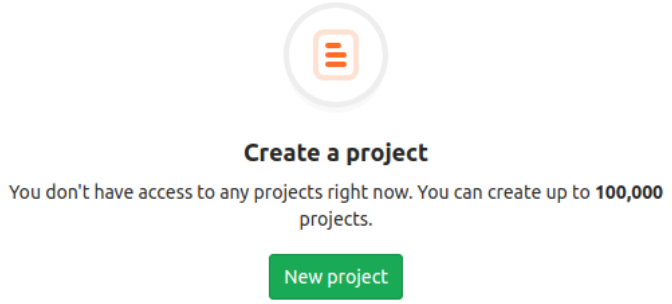
Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Criando um novo projeto

Para criarmos um novo projeto, basta clicarmos em **New project**:



Anotações:

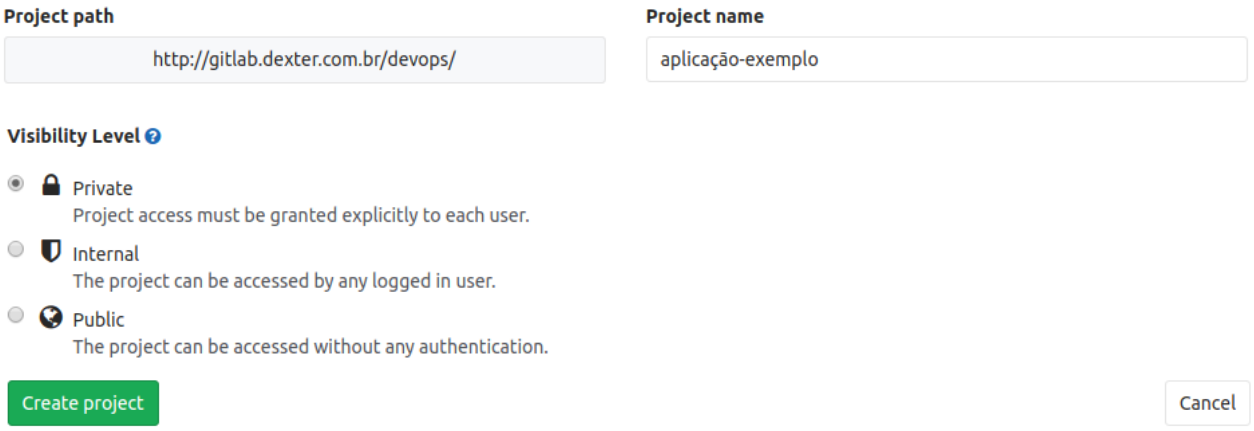
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Nome e visibilidade do projeto

Podemos definir o nome como aplicacao-exemplo em “Project name” e como Private em “Visibility Level”, em seguida crie o projeto.



Um recurso interessante ao criar um projeto no gitlab é o nível de visibilidade que pode ser definido ao projeto.

Um projeto pode ser:

Public – Qualquer possa pode ver o conteúdo do projeto e baixar.

Private – Somente usuários com permissão podem ver e editar o projeto.

Interal – Qualquer usuário que esteja logado conseguirá clonar o projeto.

Existem também opções de importar um projeto já existente do Bitbucket,Github, Gitorious, GoogleCode ou qualquer outro servidor de git que você tenha.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Clonando o projeto

1	Faça um clone do projeto, será criado um diretório com o mesmo nome do projeto. `#git clone http://gitlab.dexter.com.br/devops/aplicacao-exemplo.git`
2	Para listar os repositórios configurados `# git remote show`
3	Verificar as configurações do repositório `# git remote show origin`
4	Mostrar as modificações dentro do repositório `# git diff`

O git permite que você trabalhe com vários repositórios remotos. Caso você queira adicionar um segundo “remote”, você pode utilizar o seguinte comando:

git remote add segundo_remote URL_DO_REPOSITORIO

Assim, você pode mudar a sintaxe do comando **git pull** ou **git push**.

Exemplo:

git push origin master – Vai para o repositório de origem do projeto (o qual ele foi clonado).

git push segundo_remote master – Vai para o segundo repositório remoto que você adicionou.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando repositórios com GitLab

Subindo arquivos para o projeto

1 Adiciona todos os arquivos modificados para fazer o commit

```
# git add --al
```

2 Criar um histórico no git com estas modificações:

```
# git commit -m "Descreva o que foi modificado"
```

3 Envie as modificações para o repositório:

```
# git push origin master
```

Ao trabalhar com "branches" podemos alterar comando da etapa 3 e substituir master para a branch utilizada.

Quando utilizamos o **git add –all**, ele adiciona todos os arquivos modificados ao **commit** que será realizado.

Caso você queira adicionar somente um arquivo, pode-se fazer da seguinte maneira:

git add arquivo1.txt

Depois faça um **commit** e um **git push** normalmente.

Existem muitos outros comandos do git, como checkout, branch, status, etc. Para esse curso esses já serão suficientes para o que queremos fazer.

Porém, vale a pena se aprofundar em comandos do git.

Caso você queira estudar mais, é só consultar na documentação oficial do projeto:

https://git-scm.com/doc

Anotações:

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Objetivos da Aula

- ✔ Criar usuários e projetos;
- ✔ Listar usuários e projetos;
- ✔ Adicionar usuários ao projeto.

Administrando o GitLab com Python

Nesta aula iremos aprender a administrar o GitLab através do Python, iremos criar e listar os usuários e projetos e adicionar novos usuários aos projetos.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Instalação do módulo

A integração com o GitLab é feita com o módulo chamado requests, podemos instalá-lo com o seguinte comando:

```
pip install requests
```

Após a instalação já podemos começar a construir as integrações. Vamos criar um arquivo e criar as funcionalidades mais utilizadas do GitLab.

Sempre utilizamos o comando pip para fazer a instalação desses módulos do python, mas se não anotarmos todos eles, fica difícil de descobrir quando for fazer o deploy da aplicação em um outros servidores.

Para resolver esse problema geralmente criamos um arquivo chamado requirements.txt. Para gerar esse arquivo, execute os seguintes comandos:

pip install pipreqs

E depois:

pipreqs caminho_do_projeto

Será gerado o arquivo requirements.txt. Para instalar os módulos no outro servidor utilize o seguinte comando:

pip install -r requirements.txt

Anotações:

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Autenticação

A primeira coisa a ser feita é pegar o token para autenticação com o servidor. No painel no GitLab vá el: Profile Settings > Account.

Você verá um campo chamado “Private Token” que é o token para autenticação, vamos usá-lo como abaixo:

```
import requests
token = “rsSMxQqJy2s1yVyFsmJe”
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

URL da API

Todas as requisições devem ser feitas no endereço como abaixo:

http://gitlab.dexter.com.br/api/v4/**RECURSO**?private_token=**TOKEN**

Onde recurso é o recurso da API que vamos utilizar, por exemplo: user para usuários.

E token é a chave que obtemos no painel do GitLab.

No exemplo acima, foram mostrados somente os parâmetros obrigatórios para o cadastro de um usuário no Gitlab. Caso você queria passar mais parâmetros além desses que foram listados, você pode consultar essa página da documentação oficial:

http://doc.gitlab.com/ce/api/users.html

Os parâmetros opcionais são:

skype (optional) - Skype ID
linkedin (optional) - LinkedIn
twitter (optional) - Twitter account
website_url (optional) - Website URL
projects_limit (optional) - Number of projects user can create
extern_uid (optional) - External UID
provider (optional) - External provider name
bio (optional) - User's biography
location (optional) - User's location
admin (optional) - User is admin - true or false (default)
can_create_group (optional) - User can create groups - true or false
confirm (optional) - Require confirmation - true (default) or false
external (optional) - Flags the user as external - true or false(default)

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Listando projetos

Para podemos listar os projetos, faremos da seguinte forma:

```
def get_project(self):
  recurso = "projects"
  token = “rsSMxQqJy2s1yVyFsmJe”
  url = "http://gitlab.dexter.com.br/api/v4/%s?
         private_token=%s"%(recurso,token)
  response = json.loads(requests.get(url)
                        ._content.decode('utf-8'))
  projects = [ r['name'] for r in response ]
  return projects
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Listando usuários

Para podemos listar os usuários, podemos fazer bem parecido com a função anterior, precisamos alterar apenas o recurso.

```
def get_users(self):
     recurso = "projects"
     token = “rsSMxQqJy2s1yVyFsmJe”
  url = "http://gitlab.dexter.com.br/api/v4/%s?
         private_token=%s"%(recurso,token)
  response = json.loads(requests.get(url)
                        ._content.decode('utf-8'))
  projects = [ r['name'] for r in response ]
  return projects
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar novos usuários

Para adicionar um novo usuário, precisamos passar 4 parâmetros obrigatórios: nome, username, e-mail e senha.

```
def create_users(self, usuario):
  recurso = 'users'
  token = “rsSMxQqJy2s1yVyFsmJe”'
  url = "http://gitlab.dexter.com.br/api/v4/%s?
         private_token=%s"%(recurso,token)
  usuario = {"name":"User","username":"user",
             "email":"user@dexter.com.br",
             "password":"4linux123"}
  response = json.loads(requests.post(url,
             data=usuario)._content.decode('utf-8'))
  return "Usuario adicionado com sucesso!"
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar novos projetos

Para adicionar um novo projeto, precisamos precisamos apenas passar qual o nome do projeto.

```
def create_project(self, project):
  recurso = "projects"
  token = "rsSMxQqJy2s1yVyFsmJe"
  url = "http://gitlab.dexter.com.br/api/v4/%s?
         private_token=%s"%(recurso,token)
  projeto = {"name":"novo-projeto"}
  response = json.loads(requests.post(url,
             data=project)._content.decode('utf-8'))
  return "Projeto adicionado com sucesso!"
```


Todas as URL's que serão utilizadas durante esta aula estão disponíveis na documentação oficial:

http://doc.gitlab.com/ce/api/

Existem muitos outros recursos na API do que os que são utilizados na aula. No entanto, foram selecionados os mais importantes e os mais utilizados. Pode ser que você precise de algo que não tenha sido abordado durante a aula. Nesse caso, pode-se buscar essas informações na documentação oficial.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar um membro ao projeto

Para adicionar um membro a um projeto existente, vamos criar uma função e ela deve receber o ID do usuário e ID do projeto:

```
def add_project_member(self,project, user):
  url = "http://gitlab.dexter.com.br/api/v4/projects/
         %s/members?private_token=%s"%(project,token)
  member = {"project_id":project,
            "user_id":user,
            "access_level":30}
  response = json.loads(requests.post(url,
             data=member)._content.decode('utf-8'))
  return response
```


Quando adicionamos membros a um projeto via API, é necessário passar o ACESS_LEVEL que, nesse caso, recebe um valor numérico.
No entanto, pela interface do Gitlab, existem as seguintes permissões:

- Guest
- Reporter
- Developer
- Master

Essas permissões são equivalentes aos seguintes números:

10 – Guest
20 – Reporter
30 – Developer
40 - Master

Anotações:

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar um membro ao projeto

Agora podemos utilizar list comprehension para buscar o ID do projeto e dos usuários:

```
if __name__ == '__main__':
  user = [u for u in get_users() if u['username'] == 'user']
  project = [p for p in get_project() if p['name'] == 'novo-projeto']
  add_project_member(project[0]['id'], user[0]['id'])
```

Na função addprojectmember, o access_level foi ajustado como 30 que é o valor equivalente a permissão de desenvolvedor do gitlab.

List Comprehension faz parte do estudo de estrutura de dados. Esse é um recurso utilizado para criação de listas, sendo possível gerar uma nova lista a partir de uma lista com muitos outros elementos.

No exemplo acima, a varável **user e project** irão retornar uma lista com todas as informações do usuário “user” e do projeto “novo-projeto”. Como precisamos apenas do ID para conseguir completar a ação, então podemos acessar o indice 0 (correspondente ao objeto encontrado) e em seguida acessamos a chave “id”.

A documentação oficial do python possui exemplos muito interessantes sobre esses recursos. Você pode vê-los no seguinte link:

https://docs.python.org/2/tutorial/datastructures.html

Anotações:

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Webhooks

O GitLab tem suporte a WebHooks, isto significa que dentro do diretório do projeto podemos configurar uma URL que será acionada dependendo da ação que ocorrer no repositório. Estas ações podem ser:

Push Events – Aciona a URL quando for feito um PUSH no projeto
Tag Push Events – Quando uma nova tag for adicionada
Comments – Quando um comentário for adicionado ao repositório.
Issue Events – Quando uma ISSUE for criada.
Merge Request Events – Quando for feito um Merge entre Branchs.
Build Events – Quando for realizada uma Build do projeto.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar uma webhook

Para adicionar um membro a um projeto existente, vamos criar uma função e ela deve receber o ID do usuário e ID do projeto:

```
def addprojecthook(self,project_id,hook_url):
  token = "rsSMxQqJy2s1yVyFsmJe"
  url = "http://gitlab.dexter.com.br/api/v4/projects/
         %s/hooks?private_token=%s"%(project_id, token)
  data = {"url":hook_url,"push_events":True}
  response = json.loads(requests.post(url,
             data=data)._content.decode('utf-8'))
  return “WebHook criada com sucesso!”
```


Na lista abaixo temos todos os parâmetros obrigatórios e opcionais ao cadastrar uma webhook:

id (required) - The ID or NAMESPACE/PROJECT_NAME of a project
url (required) - The hook URL
push_events - Trigger hook on push events
issues_events - Trigger hook on issues events
merge_requests_events - Trigger hook on merge_requests events
tag_push_events - Trigger hook on push_tag events
note_events - Trigger hook on note events
enable_ssl_verification - Do SSL verification when triggering the hook

Caso você queira verificar na documentação oficial ela pode ser encontrada no seguinte endereço:

http://doc.gitlab.com/ce/api/projects.html#add-project-hook

Anotações:

Felipe Moz / moz.felipe@gmail.com

Administrando o GitLab com Python

Adicionar uma Webhook

Agora podemos chamar a funçao passando o ID do projeto e a URL:

```
if __name__ == '__main__':
  project = [p for p in get_project() if p['name'] == 'novo-projeto']
  addprojecthook(project[0]['id'],
                 'http://jenkins.dexter.com.br:8080/gitlab/build_now')
```

No nosso caso, a Webhook só será ativada quando for realizado um **PUSH** no repositório, o que ocasionará o **deploy** da nossa aplicação no futuro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Objetivos da Aula

- ✔ Fazer uma introdução ao Docker;
- ✔ Criar containers;
- ✔ Gerenciar containers;
- ✔ Network com Docker;

Introdução ao Docker

Nesta aula iremos aprender como criar e gerenciar containers em Docker, como criar uma network e trabalhar com elas.

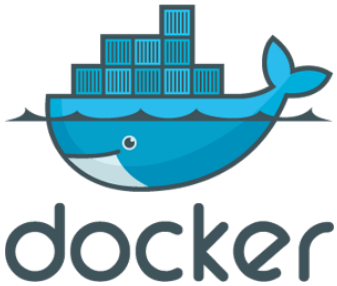
Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Introdução

Docker é uma ferramenta para gerenciar containers. Os containers do Docker 'enjaulam' o software em um filesystem completo que contém tudo o que precisa para rodar Códigos, aplicações, ferramentas de sistema e bibliotecas, basicamente, tudo o que pode ser instalado em um servidor.
Diferente de uma virtualização, este método se utiliza do mesmo kernel do sistema operacional Host.



O Docker utiliza Linux Containers (LXC). Containers é um método de virtualização rodando um sistema operacional isolado (containers) sobre um máquina hospedeira (Máquina Host). Diferente de uma virtualização, este método se utiliza do mesmo kernel do sistema operacional Host. Basicamente, container é um processo isolado através de namespaces e um chroot (**Change Root**).

Executar o chroot é tão simples e rápido quanto iniciar um novo processo no sistema operacional. Porém, o chroot de um sistema operacional requer muita I/O de disco, podendo assim tornar o processo lento. Para evitar esse tipo de problema, o Docker utiliza conceitos de imagens que contém a estrutura básica do sistema operacional e, para cada container, ele utiliza um **diff** de um container base.

O Docker utiliza o AuFS para o sistema de arquivos. É um sistema de arquivo em camadas, dando a possibilidade de ter uma parte destinada para leitura e outra parte para escrita, além do compartilhamento de diretórios entre os containers.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Vantagens na utilização de Container

Leveza

O Container executa sobre uma máquina que compartilha o mesmo Kernel do Sistema Operacional. Assim, este é iniciado instantaneamente e utiliza a memória RAM de uma forma mais eficiente.

Segurança

O Container isola as aplicações uma das outras, adicionando uma camada de segurança para a aplicação.

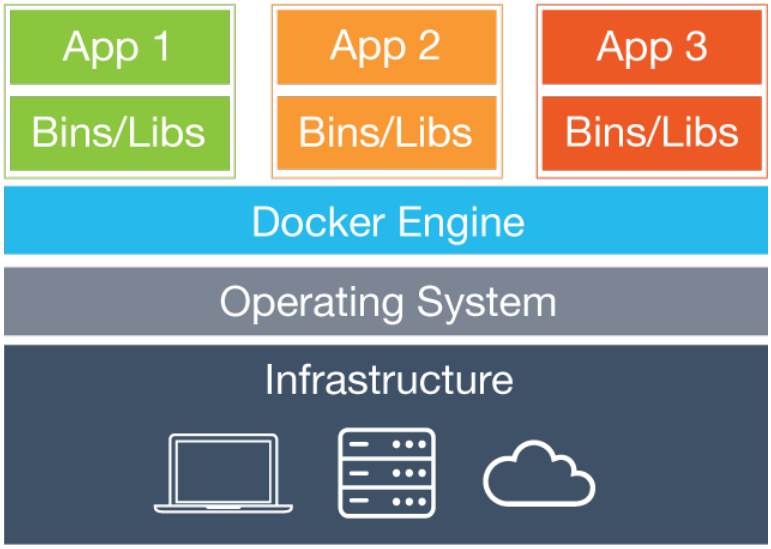
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Arquitetura do Docker Engine



O Docker Engine é responsável por criar e executar os containers do Docker, ele atua entre a máquina Host e os containers.

A Daemon é comunicada pelo cliente através da própria API para gerenciar os containers. Esse cliente pode ser instalado na mesma máquina, que é o padrão utilizado nas distribuições (comando docker). O Docker Engine também pode ser acessado externamente através da rede, normalmente utilizado em ambientes de cluster ou interfaces de gerenciamento.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Criando o primeiro container

1	Criar um container com o nome "**hello**" `# docker run --hello debian echo "Hello Container"`
2	Listar todos os containers criados pelo Docker (inclusive os parados) `# docker ps -a`
3	Listar apenas os containers em execução `# docker ps`
4	Iniciar o container criado `# docker start hello`

Docker run:
O Comando **run** é utilizado para rodar um comando em um novo container.

Syntaxe do comando run:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG…]
```

Vamos analisar cada parâmetro do primeiro comando acima:

```
$ docker run –name hello debian echo "Hello Container"
```

A Opção –**name** é utilizada para definir o nome do container. No caso **hello**, todo container precisa de uma imagem como base para os arquivos do sistema. Essa imagem contém os binários, bibliotecas e diretórios básicos do sistema operacional. No exemplo acima, estamos utilizando a imagem do **debian**. Por último, temos o comando a ser executado dentro do container. **echo "Hello World".**

Docker PS:
O comando **ps** é o comando utilizado para listar os containers. Ao executar o comando sem nenhum parâmetro, o mesmo retorna somente os containers, que estão rodando no sistema. Para ver todos os containers, sejam eles parados ou ativos, utilizamos o parâmetro **-a.**

Docker start:
O comando **start** é usado para iniciar os containers do docker. Podemos iniciar mais de um container no mesmo comando, como por exemplo:

```
$ docker start container1 container2 container3 …
```

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Gerenciando containers

1. Remover o container criado anteriormente
```
# docker rm hello
```

2. Criar o container novamente
```
# docker run -dti --name hello debian /bin/bash
```

3. Inspecione o container para obter informações
```
# docker inspect hello
```

4. Pare o container criado
```
# docker stop hello
```


Docker rm:

O comando **rm** é utilizado para remover os containers, assim como com o comando **start** podemos remover mais de um container executando um mesmo comando, como por exemplo:

```
$ docker rm container1 container2 container3
```

Docker inspect:

O comando **inspect** é utilizado para retornar informações sobre o container. Através dele é possível visualizar todas as informações disponíveis sobre um container criado pelo docker. Por padrão, o comando retorna as informações sobre o formato JSON. Através desse comando podemos visualizar o ID do Container, Data de Criação, comando executado, qual o status do container no momento da execução do comando, o ID da imagem do container, informações sobre a rede do container, utilização de recursos e muitas outras informações disponibilizadas pelo Docker Engine. Exemplo do comando **inspect**:

```
$ docker inspect container1
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Gerenciando containers

5 Inicie o container criado anteriormente

```
# docker start hello
```

6 Conecte ao terminal do container para interagir com o interpretador de comando

```
# docker attach hello
```

7 Desconecte do terminal do container

```
# CTRL^P e CTRL^Q
```

Atenção: Caso você digite 'exit' para sair do container, o mesmo será parado.

Docker attach:
O comando **attach** é utilizado para se conectar ao container, usando o nome ou ID do container. Ao se conectar ao container você pode visualizar a saída do comando. Caso esteja habilitado com a opção **-i,** é possível interagir com o container.

Para desconectar do container sem parar sua execução precisamos utilizar a sequência de teclas **CTRL^P + CTRL^Q.** Se utilizamos os comandos padrões do bash para sair do interpretador (**exit** ou **CTRL^D)**, iremos parar a execução do container.

Mas por que eu não posso sair simplesmente executando o **exit?**

Os containers do Docker permanecem executando enquanto o comando passado na criação do mesmo esteja rodando. Como vimos no nosso primeiro exemplo, quando utilizamos o comando echo "Hello Container"**,** o container foi criado e, após o término da execução do comando echo, o container é parado. Este é o comportamento de um container no Docker. Ele continua vivo enquanto o comando que foi designado a ele estiver sendo executado. Sendo assim, quando utilizamos o exit do **bash**, estamos na verdade parando o interpretador, por consequência, parando o container. Podemos utilizar o comando **start** para iniciar o container novamente.

No exemplo do slide, temos dois parâmetros novos passados ao criar o container:

```
$ docker run -dit --name hello debian /bin/bash
```

O parâmetro **-i (** –**interactive na** sua forma estendida) é utilizado para deixar a entrada de dados ao container aberta, caso você não esteja conectado (attached) no container em questão.

O parâmetro **-t** é utilizado para alocar um pseudo-tty para o container. Assim, podemos interagir com o interpretador de comando passando comandos para o **/bin/bash**, da mesma forma que usamos um terminal e uma máquina virtual ou máquina física.

O parâmetro **-d** (deattaached) é para ser utilizado caso você não queira se conectar no container no momento de sua criação.

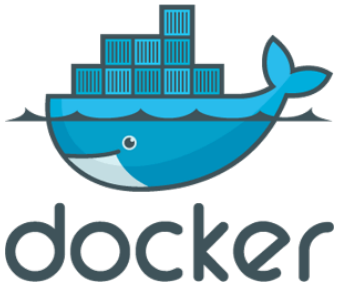
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Network com Docker

O **Docker Engine** providencia uma rede interna entre os containers. Essa rede interna é criada automaticamente durante a instalação do pacote. Para comunicação dos containers com a rede externa da máquina, o Docker cria uma interface de rede virtual que se comunica com a rede da máquina hospedeira, sendo que é utilizado o gateway para os containers.



O Docker utiliza Linux Containers (LXC). Containers é um método de virtualização rodando um sistema operacional isolado (containers) sobre um máquina hospedeira (Máquina Host). Diferente de uma virtualização, este método se utiliza do mesmo kernel do sistema operacional Host. Basicamente, container é um processo isolado através de namespaces e um chroot (**Change Root**).

Executar o chroot é tão simples e rápido quanto iniciar um novo processo no sistema operacional. Porém, o chroot de um sistema operacional requer muita I/O de disco, podendo assim tornar o processo lento. Para evitar esse tipo de problema, o Docker utiliza conceitos de imagens que contém a estrutura básica do sistema operacional e, para cada container, ele utiliza um **diff** de um container base.

O Docker utiliza o AuFS para o sistema de arquivos. É um sistema de arquivo em camadas, dando a possibilidade de ter uma parte destinada para leitura e outra parte para escrita, além do compartilhamento de diretórios entre os containers.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Gerenciando network dos containers

1. Crie um container com o nome webs recebendo conexões pela porta 80 da máquina host
```
# docker run -it -p 80:80 --name webs nginx /bin/bash
```

2. Dentro do container, inicie o servido do Nginx
```
# service nginx start
```

3. Desconecte do container sem parar a sua execução
```
# CTRL^P e CTRL^Q
```

Podemos mapear a porta da máquina host para os containers com parâmetros **-p** na criação dos containers. Qualquer conexão na porta especificada será redirecionada para o container em questão.

Docker rm:

O comando **rm** é utilizado para remover os containers, assim como com o comando **start** podemos remover mais de um container executando um mesmo comando, como por exemplo:

```
$ docker rm container1 container2 container3
```

Docker inspect:

O comando **inspect** é utilizado para retornar informações sobre o container. Através dele é possível visualizar todas as informações disponíveis sobre um container criado pelo docker. Por padrão, o comando retorna as informações sobre o formato JSON. Através desse comando podemos visualizar o ID do Container, Data de Criação, comando executado, qual o status do container no momento da execução do comando, o ID da imagem do container, informações sobre a rede do container, utilização de recursos e muitas outras informações disponibilizadas pelo Docker Engine. Exemplo do comando **inspect**:

```
$ docker inspect container1
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Docker

Gerenciando network dos containers

1 Crie uma rede interna com o nome de dexterlan

```
# docker network create dexterlan
```

2 Crie um container para teste utilizando a nova rede criada

```
# docker run --it --name node01 --hostname node01 --net dexterlan
  net-teste /bin/bash
```

3 Conectar um container que já foi criado a nova rede

```
# docker network connect dexterlan webs
```


Pode-se especificar qual o range de IP's em que essa rede vai trabalhar. Para fazer isso, execute o seguinte comando:

docker network create dexterlan –subnet=172.18.0.0/16

Podemos também definir um IP fixo para o container que será executado:

docker run -it --name node01 --hostname node01 --net dexterlan --ip 172.18.0.2 net-teste /bin/bash

A rede padrão do docker chamada docker0 trabalha no range:

172.17.0.0/16

Então, a partir da versão1.9.0 torna-se opcional criar os containers na rede padrão do Docker ou não.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Objetivos da Aula

- ✔ Entender a API do docker;
- ✔ Listar, iniciar, parar e remover containers;
- ✔ Criar novos containers.

Integrando Docker e Python

Nesta aula iremos aprender a administrar o Docker através do Python, iremos criar, listar iniciar parar e remover containers.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Liberando a API do Docker

Para liberar a API do Docker é necessário alterar o arquivo:

/lib/systemd/system/docker.service

Na linha em que está **ExecStart**, deve-se adicionar a seguinte configuração e reiniciar o serviço:

-H tcp://0.0.0.0:2376

Com essa opção a API Docker pode ser acessada por qualquer endereço. Também é possível restringi-la para somente a rede interna ou somente para um IP em específico.

Caso o seu sistema seja baseado em SysV, o arquivo a ser modificado é diferente. Então deve editar o arquivo:
/etc/default/docker

Na linha DOCKER_OPTS deve ser adicionado o parâmetro:
-H 0.0.0.0:2376

Feito isso, devemos reiniciar o docker:
service restart docker

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Instalação do módulo

Após liberar a API no servidor, devemos instalar o módulo chamado **docker-py**, podemos instalar utilizando o seguinte comando:

```
pip install docker-py
```

Agora podemos criar um arquivo para trabalharmos com a API e vermos as operações que podem ser realizadas com o Docker.

O github do módulo docker-py pode ser acessado no seguinte link:

https://github.com/docker/docker-py

É muito interessante seguir esses repositórios no github, pois sempre há novas atualizações e novos recursos que podem ser úteis para utilizarmos em nossos projetos.

Você pode também abrir alguma **issue** para o projeto caso você encontre um bug no módulo que está utilizando.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Conectando com o servidor

A primeira coisa que precisamos fazer é conectar com o servidor Docker. Para isto, podemos colocar as seguintes linhas de código em um arquivo:

```
from docker import Client
cli = Client(base_url="tcp://192.168.0.2:2376")
```

Podemos colocar os códigos dentro de blocos try/except para tratar as mensagens caso ocorra algum erro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Listando containers criados

Para podermos listar os containers em execução, utilizamos a seguinte instrução:

```
for c in cli.containers():
    print (c)
```

Caso queira mostrar todos os containers (inclusive os que estão parados) basta passar o parâmetro all=True

```
for c in cli.containers(all=True):
    print (c)
```


O retorno do código acima será um JSON com diversas informações sobre o container, podemos "limpar" estes dados e mostrar as informações de forma mais clada, como por exemplo, para mostrar apenas o nome do container e a imagem que está utilizando:

```
#!/usr/bin/python3

from docker import Client

class DockerClass(object):

    def __init__(self):
        self.cli = Client(base_url="tcp://192.168.0.2:2376")

    def list_all(self):
        for c in self.cli.containers(all=True):
            print ("Nome:" ,c['Names'][0].replace('/',''))
            print ("Imagem:" ,c['Image'])
            print ("===========")


if __name__ == '__main__':
    d = DockerClass()
    d.list_all()
```

Anotações:

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Criar novo container

Para criar um novo container, podemos criar o seguinte código:

```
cli.create_container(name='hello',
                     command='/bin/bash',
                     image='debian')
```

A função create_container() pode receber todos os parametros igual passados na linha de comando. No exemplo acima, colocamos o nome como 'hello', a imagem como 'debian' e o comando como '/bin/bash'.

Podemos usar outros parâmetros, como por exemplo:

stdin_open=True

detach=True

tty=True

Estes equivalem a opção -tdi na linha de comando. E se precisarmos definir o hostname, podemos colocar a opção:

hostname='exemplo.com.br'

A lista de parâmetros para se criar um container é bem extensa.

Caso você queira saber todos os parâmetros eles estão disponíveis no link abaixo:

https://docker-py.readthedocs.org/en/latest/api/#create_container

Não só os parâmetros estão disponíveis como tem também informações sobre o retorno dessa função.

Caso você não queria utilizar esse módulo, todas as operações podem ser realizadas através da API do próprio Docker, conforme a documentação:

https://docs.docker.com/engine/reference/api/docker_remote_api_v1.22/

No entanto, é interessante a utilização do módulo, pois ele facilita o trabalho e faz com que seja necessário escrever menos código. Caso contrário, será necessário utilizar algum outro módulo como o **requests,** consumir a API e trabalhar como **response**.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Iniciar e parar containers

Podemos iniciar e parar containers passando o seu nome na função, como abaixo:

```
# iniciar container
cli.start('hello')
# parar container
cli.stop('hello')
```

Também podemos utilizar o ID do container para as operações acima:

```
cli.start(container=id)
```


Um outro projeto interessante também do Docker é o Docker Swarm, que pode ser encontrado no seguinte link:

https://docs.docker.com/swarm/swarm-api/

O Docker Swarn é geralmente utilizado em conjunto com o Docker Machine para que você possa gerenciar Clusters em Docker, o que pode se tornar necessário dependendo do tamanho do seu ambiente.

Caso você queira mais informações sobre o Docker Machine, elas podem ser encontradas no link abaixo:

https://docs.docker.com/machine/overview/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Executando comandos no container

É comum termos a necessidade de executarmos comandos em um container, para isto podemos fazer o seguinte código:

```
exec = cli.exec_create(container.get("Id"),
                       "git clone http://repo_exemplo.git")
cli.exec_start(exec.get("Id"))
```


Sempre que é realizada uma chamada de um método **exec_create** , ele retorna um ID para a execução e para que ela seja executada é necessário utilizar o seguinte método:

exec_start(exec_id)

Sendo o **exec_id** o ID da execução retornado pela chamada do **exec_create**, podemos passar também da seguinte maneira:

cli.exec_start(cli.exec_create(container.get("Id"),"git clone http://repo_exemplo.git").get("Id"))

Anotações:

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Ver informações do container

É possível também saber as informações de um container como, por exemplo, rede, endereço de IP, nome do container, etc.

```
print cli.inspect_container(container.get("Id"))
```

Será retornado um dicionário com os valores. Pode-se acessar qualquer informação a partir dele.

Uma das partes mais importantes na hora de trabalhar com Docker, é inspecionar os containers, pois assim é possível saber, por exemplo, o IP que um container está utilizando naquele momento.

Antes de surgir o comando **docker networks** era utilizado na maioria das vezes a rede padrão do Docker. Uma vez que um container era "Stoppado", ele perdia o IP do qual havia sido designado. Quando esse container voltava ao ar o Docker Engine, dava a ele o próximo IP disponível, então, se o IP do container é algo importante para o que você está fazendo, sempre era necessário fazer a inspeção do container para saber qual era o IP que ele havia recebido.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Integrando Docker e Python

Removendo container

Uma vez que o container não é mais necessário, pode-se exclui-lo também através de API. No entanto, é necessário parar um container caso ele esteja sendo executado para só depois efetuar a sua exclusão.

```
cli.stop('hello')
cli.remove_container('hello')
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução Jenkins

Objetivos da Aula

- ✔ Fazer uma introdução ao Jenkins;
- ✔ Conhecer o painel;
- ✔ Conhecer plugins;
- ✔ Conhecer Pipeline do Jenkins.

Introdução ao Jenkins

Nesta aula iremos aprender como criar e gerenciar Jobs no Jenkins, entenderemos como ele funciona e como podemos automatizar deploys.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Introdução

Jenkins é um servidor de automação escrito em java e que disponibiliza centenas de plugins para a automatização de deploys de aplicação, testes e integração com outros sistemas.

Atualmente está entre as melhores ferramentas de integração contínua que exitem no mercado.

Caso você queira fazer a sua própria instalação do Jenkins, ele pode ser baixado no site oficial: https://jenkins.io/

Está disponível para as principais distribuições como:

- Ubuntu
- Debian
- CentOS
- openSUSE

Também é possível baixar uma imagem do Docker com o Jenkins já instalado:

https://hub.docker.com/r/jenkinsci/jenkins/

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Características do Jenkins

- Interface web.
- Extensível (plugins).
- Multiplataforma (Desenvolvido em Java)
- Integração com a maioria dos SCM's.

Continuous Integration:

Continuous Integration ou **Integração Continua** é uma prática de Desenvolvimento onde os membros de um time integram seu trabalho frequentemente, normalmente essa integração é realizada diariamente. Cada integração é verificada por uma tarefa ou build automatizada para detectar erros de integração da forma mais rápida possível. A integração contínua diminui a probabilidade de erros dentro de um projeto.

Continuous Delivery

Continuous Delivery ou **Entrega Contínua** é uma prática de desenvolvimento usada na construção de software, de uma maneira que você pode disponibilizar mudanças em seu ambiente de produção a qualquer momento.

Em um ambiente de entrega contínua, são realizadas mudanças pequenas e com maior frequência de Deploys, o que diminui o risco de erros em ambientes de produção, simplesmente pelo fato de serem pequenas mudanças.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Tela inicial do Jenkins

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Verificar plugins disponíveis

O Jenkins possui diversos plugins que podem ser utilizados para incrementar funcionalidades e ajudar na automação. Para visualizar os plugins, clique em "**Gerenciar Jenkins**" e em seguida em "**Gerenciar Plugins**".

Anotações:

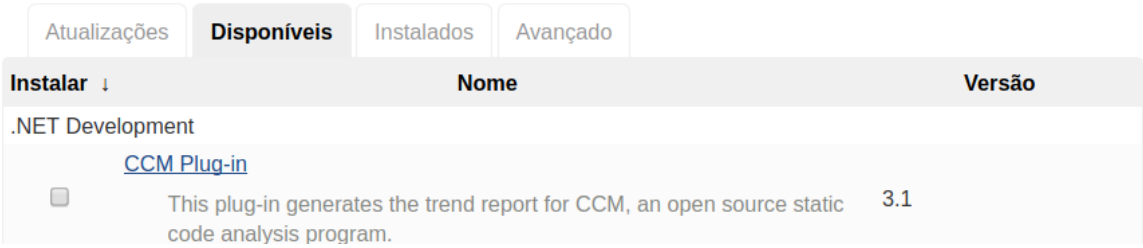
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Verificar plugins disponíveis

Podemos instalar novos Plugins clicando na aba "**disponíveis**".



Na aba "**Instalados**" podemos gerenciar os plugins que estão sendo utilizados atualmente.

Atualmente temos certa de 1000 plugins disponíveis na comunidade do Jenkins, eles facilitam a construção de um Job, como por exemplo:

coverage jenkins – utilizado para gerar gráficos.
https://wiki.jenkins.io/display/JENKINS/Cobertura+Plugin

Maven plugin – oferece uma integração avançada para deploy de projetos Maven.
https://wiki.jenkins.io/display/JENKINS/Maven+Project+Plugin

Git plugin - permite a integração com o Git.
https://wiki.jenkins.io/display/JENKINS/Git+Plugin

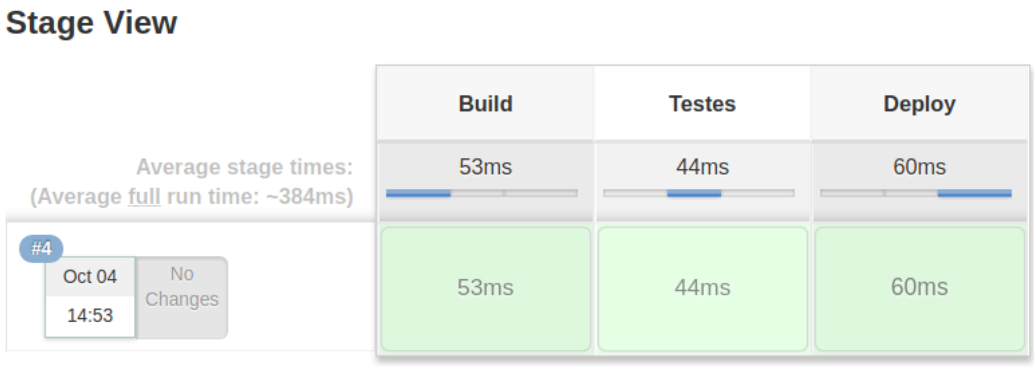
Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Jenkins

Criar Pipeline no Jenkins

Um dos recursos muito utilizados no Jenkins é a Pipeline, que permite a construção deploy em um único fluxo de trabalho.



Jenkins Pipeline foi adicionada na versão 2.0 do Jenkins e temos uma visualização muito melhor do fluxo de trabalho, tal como monitoramento de qual tarefa foi executada sem problemas, qual está em execução e qual ainda será executada.

Podemos criar um Pipeline seguindo os seguintes passos:

- No DashBoard do Jenkins, clique em “Novo Job”, selecione “Pipeline”.
- Na opção “Definition” (dentro de Pipeline), selecione: Pipeline script.

Podemos preencher o campo do scrip da seguinte forma:

```
node {
    stage 'Build'
        echo "build"
    stage 'Testes'
        echo "testes"
    stage 'Deploy'
        echo "deploy"
}
```

Onde “stage” serão os estados para visualização e abaixo os comandos que serão executados em cada um deles.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Objetivos da Aula

- ✔ Criar Jobs;
- ✔ Executar Jobs;
- ✔ Criar Jobs parametrizados.

Criando Jobs com Jenkins

Nesta aula iremos aprender como criar e gerenciar Jobs no Jenkins, entenderemos como ele funciona e como podemos automatizar deploys.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

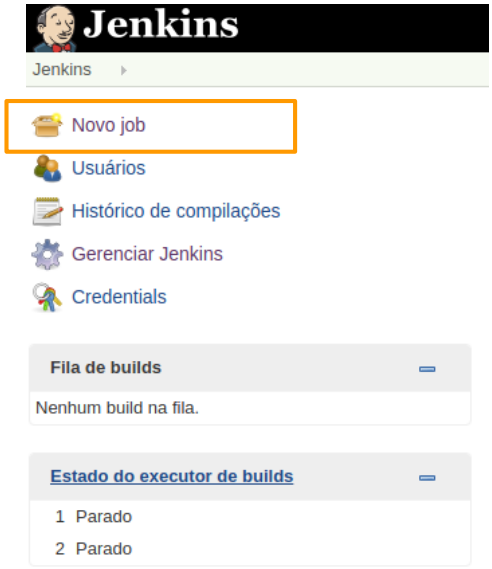
Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Criar job no Jenkins

Os **Jobs** ou **Projects** são tarefas gerenciadas e monitoradas pelo Jenkins.

Para criar, basta criar em "Novo job", como no exemplo ao lado.



Anotações:

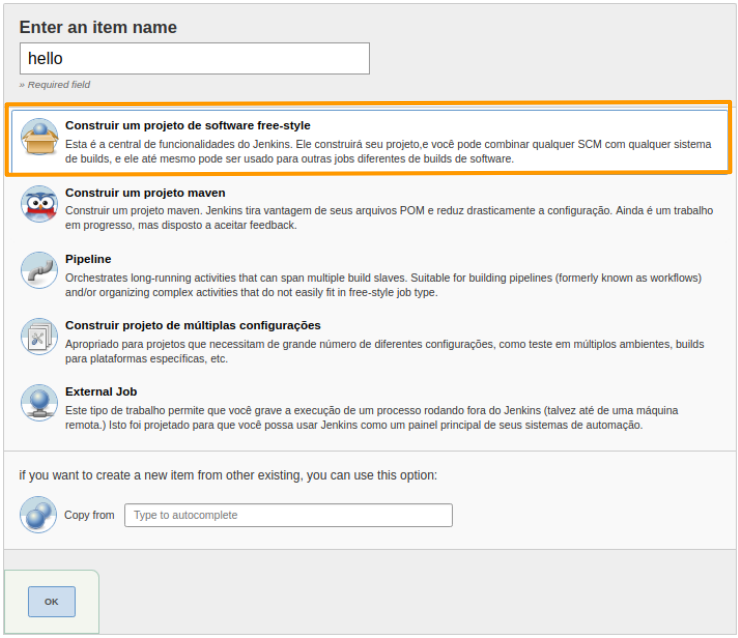
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Criar job no Jenkins

Defina o nome para: **Hello Dexter**, selecione a opção: **Construir um projeto de software free-style** e clique em **OK**.



Tipos de Jobs

Projetos de software Free-Style: São projetos para um propósito geral que disponibilizam maiores flexibilidades de ferramentas e tarefas no Jenkins.

Projetos Maven: É uma Job especializada para construção de Projetos utilizando o Maven. O Jenkins reconhece os arquivos de configuração do Maven para construir os projetos. Maven é um software para automatizar a construção e dependências de projetos de software, especialmente utilizado para projetos Java.

Pipeline: Um conjunto de plugins capaz de implementar e integrar pipelines de entrega contínua.

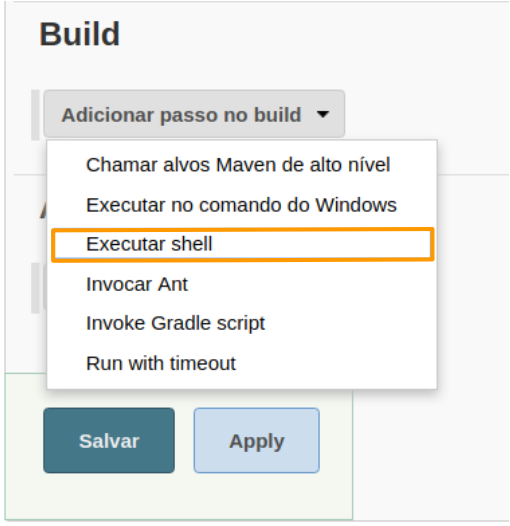
Construir um Projeto de Múltiplas Configurações: Define um projeto que pode ser executado em várias outras configurações. Normalmente utilizado para ambientes de teste de uma aplicação em ambientes diferentes, com banco de dados diferentes, máquinas diferentes.

External Jobs: Usado para monitorar a execução de processos não interativos, como tarefas da crontab do sistema e procmail.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Para adicionar um passo ao projeto para executar uma shell, basta clicar em "**Executar shell**". Onde iremos definir o comando a ser executado.



Os Steps de uma Job são comandos a serem executados em sequência pelo Jenkins durante a execução de uma Job.

Esses comandos podem ser executados ela Shell do Linux diretamente selecionando a opção **Executar Shell**.

No mundo de desenvolvimento é muito comum também utilizar a opção Invocar Ant.

O Ant é uma biblioteca em java desenvolvida com o objetivo de compilar projetos em Java. No entanto, ele funciona muito bem para outras linguagens como python e PHP. No curso PHP Orientado a Objetos da 4Linux, ele é utilizado para executar os comandos do PHPLint, PHPCodeSniffer, PHPUnit, entre outros, com o objetivo de verificar se o código em PHP está dentro dos padrões de programação e se existem erros de Sintaxe.

Para ter mais informações sobre o Ant, o seguinte link pode ser consultado:

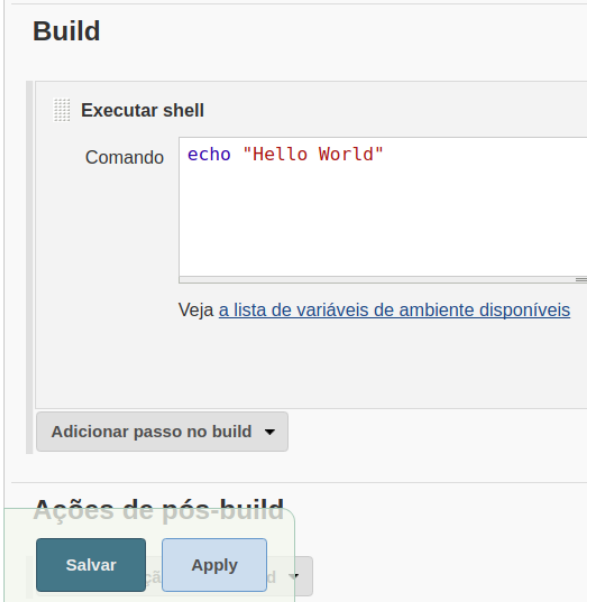
http://ant.apache.org/

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

No campo "**comando**" podemos inserir o comando a ser executado. Podemos incluir mais de um comando selecionando novamente o "**Adicionar um passo no build**".

Clique no botão **Salvar** para finalizar a criação da nossa primeira Job.

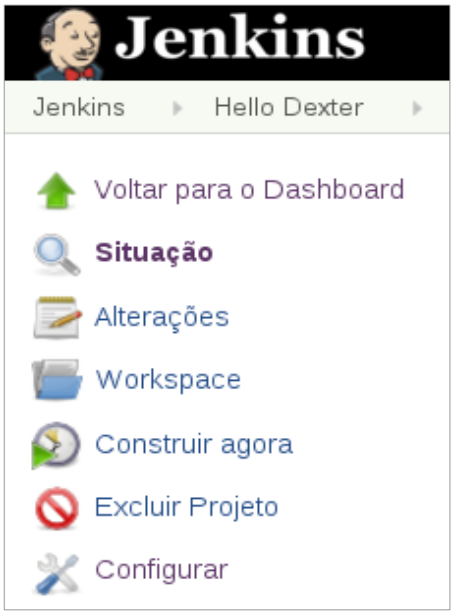


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins



Menu do projeto

Alterações: Histórico de mudanças.

Workspace: Arquivos criados pelas Builds.

Construir agora: Executar uma tarefa manualmente

Excluir Projeto: Delevar um projeto

Configurar: Reconfigurar o projeto

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Iniciar uma tarefa

Vamos executar manualmente a nossa primeira tarefa no Jenkins, para fazer isto basta clicar em "**Construir agora**" no menu do projeto.

Esta execução deve ficar marcada no histórico de Builds:

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Iniciar uma tarefa

Vamos executar manualmente a nossa primeira tarefa no Jenkins, para fazer isto basta clicar em "**Construir agora**" no menu do projeto.

Esta execução deve ficar marcada no histórico de Builds:

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Verificando a Saída do Console

Após a execução da Build, podemos verificar a saída dos comandos clicando em "**Saída do Console**".

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

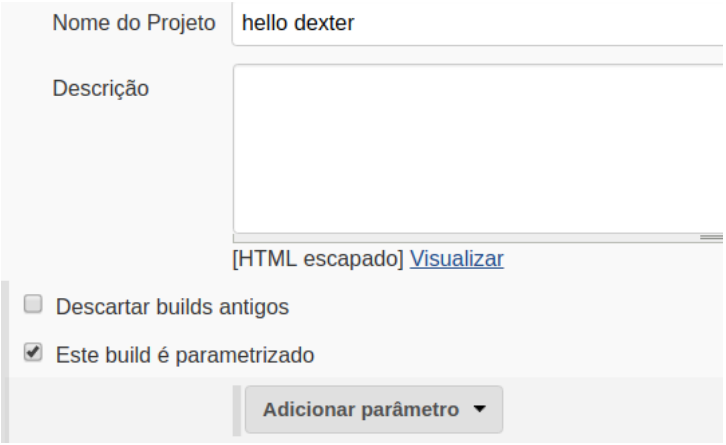
Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Parâmetros no Build

Pode-se passar parâmetros para que uma **build** seja executada.

Para fazer, isso clique em **Voltar ao Projeto** e depois em **Configurar**. Existe uma opção chamada **Este build é parametrizado**.



É um recurso muito interessante utilizar Jobs parametrizadas, pois assim podemos criar como se fosse uma Job genérica e, através de parâmetros, definir onde ela vai executar os Steps.

Por exemplo, poderíamos criar uma Job que instale o apache em um container Docker. Então, o step dessa job poderia ser:

```
docker exec -ti $container apt-get install apache2 -y
```

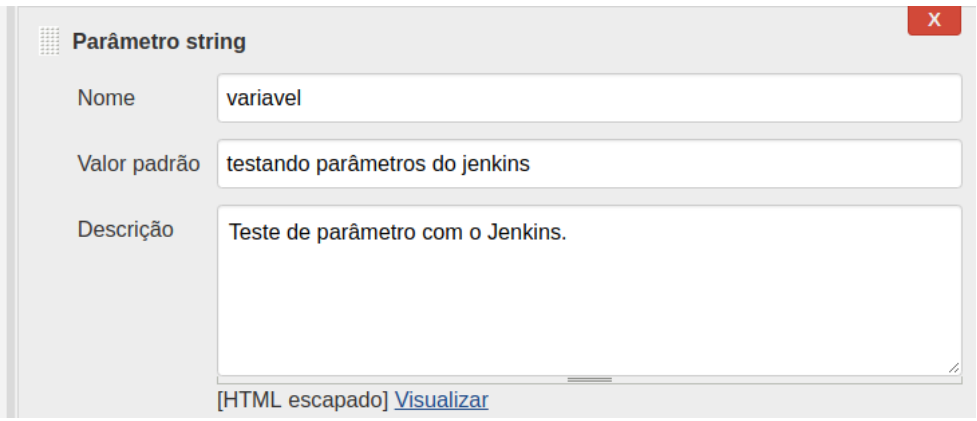
Sendo o $container um parâmetro da Job e o nome do container pode ser passado quando essa job for executada. Assim, a mesma job serve para instalar o apache em N containers.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Ao clicar em **Adicionar Parâmetro** e selecione **Parâmetro String**.
Em nome será o nome da variável usada em alguns passos da sua Build e o valor padrão é o valor que ela deve possuir caso não seja informado nenhum.



O nome da Job é o nome da variável que será chamada no step, como na página anterior em que chamamos a variável de:

$container

Assim, o nome do parâmetro seria **container** e o valor default é passado para que ele seja assumido caso um parâmetro não seja informado. No caso da job de instalação do apache, não é interessante deixar uma job com um valor default.

Então, pode-se criar um Step que verifique o conteúdo dessa variável e, se ele for o valor padrão, automaticamente a Job falhará, pois o apache deve ser instalado sempre em um container.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Em **Executar Shell** faça a alteração do comando e coloque da seguinte forma:

Não existe limite de parâmetros para uma job. Você pode criar, por exemplo, uma job que execute comandos, sendo os parâmetros definidos da seguinte maneira:

comando args

Na variável **comando** seria passado o comando executado e na **args** os argumentos desse comando. Um exemplo pode ser a instalação do apache. O conteúdo das variáveis ficaria da seguinte maneira:

comando = apt-get
args = install apache2 -y

No step poderia ficar assim:
docker exec -ti $container $comando $args

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Jenkins

Veja que agora a opção **Construir** virou **Construir com parâmetro** e o valor padrão já está preenchido na caixa de texto. Coloque como parâmetro: **Python e Jenkins** e clique em construir.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Objetivos da Aula

- ✔ Entender a API do Jenkins;
- ✔ Criar novos Jobs;
- ✔ Iniciar jobs.

Integrando Docker e Python

Nesta aula iremos aprender a administrar o Docker através do Python, iremos criar, listar iniciar parar e remover containers.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Instalação do módulo

Para fazer a integração com o Jenkins, podemos utilizar um módulo chamado **python-jenkins**:

```
pip install python-jenkins
```

Veremos duas funcionalidades que já são suficientes para criar muitos ambientes automatizados: criar e executar Jobs.
Mas através deste módulo é possível praticamente gerenciar tudo do Jenkins.

O módulo python-jenkins foi desenvolvido pelo OpenStack com o objetivo de gerenciar jobs e nodes do Jenkins.

O repositório está hospedado no Github:

https://github.com/openstack/python-jenkins

Durante essa aula o trabalho será focado nas jobs, mas se você quiser saber mais informações a respeito desse módulo, segue abaixo o link da documentação oficial:

http://python-jenkins.readthedocs.org/en/latest/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Conexão com o servidor

Após realizar a instalação do módulo, podemos criar a conexão com o servidor Jenkins:

```
import jenkins

srv = jenkins.Jenkins('http://jenkins.dexter.com.br:8080')
print (srv.get_version())
```

O script acima deverá retornar a versão do Jenkins na tela.

Quando instanciamos o objeto **srv** a partir da classe **Jenkins** , o servidor não é consultado, assim não é possível saber se a conexão com o Jenkins foi realizada com sucesso. Esse processo somente define no objeto qual é o servidor em que o Jenkins está instalado.

Para fazer o teste de conexão é comum utilizar o método **.get_version().** Assim, o módulo **fazer uma conexão com o servidor do Jenkins** para saber qual é a versão atual do Jenkins sendo executada na máquina, caso ele não consiga conectar automaticamente, retornará um erro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Criando uma nova Job

Para criar uma Job, podemos adicionar abaixo do srv.get_version() o seguinte código:

```
print (srv.create_job('Python-Job',
                      jenkins.EMPTY_CONFIG_XML))
```

O primeiro parâmetro do método **create_job** é o nome do Job que será criado e o segundo deve receber um arquivo XML com as configurações que serão aplicadas. Podemos usar o **jenkins.EMPTY_CONFIG_XML** para criar um job em branco.

O atributo EMPTY_CONFIG_XML nada mais é do que um XML, sendo este o conteúdo:

```
<project>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <scm class='jenkins.scm.NullSCM'/>
  <canRoam>true</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <triggers class='vector'/>
  <concurrentBuild>false</concurrentBuild>
  <builders/>
  <publishers/>
  <buildWrappers/>
</project>
```

Anotações:

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Executar uma Job

Para executar uma Job já existente, podemos utilizar a seguinte instrução:

```
srv.build_job('Python-Job')
```

Caso seja um Job com parâmetros, eles podem ser passados na forma de dicionário:

```
srv.build_job('Python-Job',
              {"container":"server.dexter.com.br"})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Jenkins e Python

Listar Jobs criadas

Para listar quais são as Jobs que estão criadas no Jenkins, podemos utilizar a seguinte instrução:

```
for u in srv.get_jobs():
    print (u['fullname'])
```

Podemos verificar também informações específicas sobre um Job:

```
print (srv.get_job_info('Python-Job'))
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Objetivos da Aula

- ✔ Criar Jobs no Jenkins;
- ✔ Conhecer o formato XML;
- ✔ Trabalhar com XML na criação de Jobs;

Criando Jobs com Python

Nesta aula iremos aprender a criar Jobs completas no Jenkins, passando arquivos XML e assim criar os steps automaticamente.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Criar Jobs utilizando XML

É difícil escrever uma **Job** totalmente por xml, então uma das soluções que se pode adotar, é criar uma Job manualmente pela interface do **Jenkins**, copiar o xml dessa Job que fica armazenado em **/var/lib/jenkins/jobs** no servidor onde o Jenkins está instalado e usar esse xml como template.

Vamos criar uma Job vinculada a um repositório Git e com um Step, isto servirá como modelo para utilizarmos com o Python posteriormente.

É interessante saber criar as jobs a partir de um arquivo XML, mas nem sempre é necessário pegar o XML de uma job existente, editá-lo e depois criar uma job.

É possível fazer a cópia de uma job com um novo nome. Para isso, você pode utilizar o seguinte método:

```
server.copy_job('job', 'job_copy')
```

Se você quiser zerar a configuração dessa job, pode-se utilizar o seguinte comando:

```
server.reconfig_job('job_copy', jenkins.RECONFIG_XML)
```

Anotações:

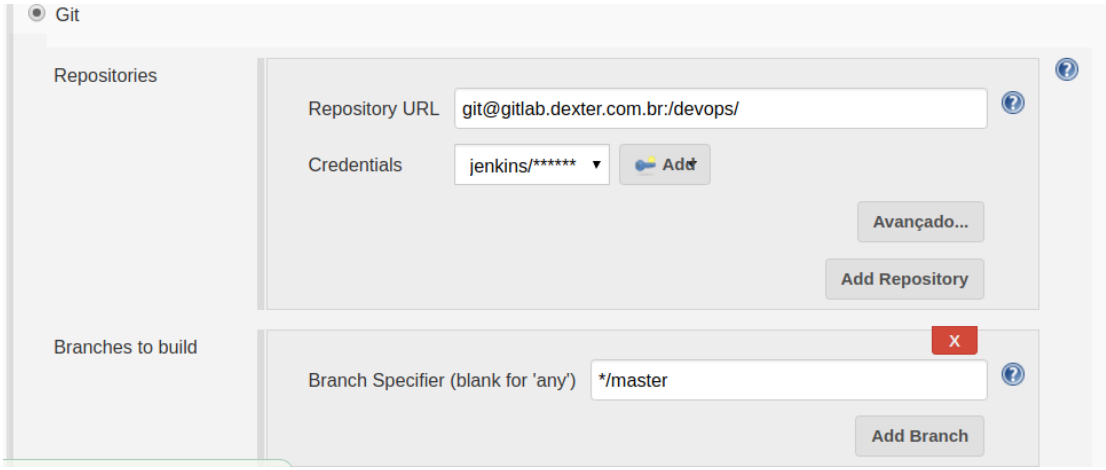
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Configuração do Git

A configuração do git pode ficar da seguinte maneira:



Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Adicionar um novo STEP

Podemos adicionar conforme visto a seguir:

Após salvar a Job, podemos verificar o arquivo XML.

Quando adicionamos um comando no step, esse comando não é restrito a somente um. Você pode, por exemplo, inserir um shell script dentro do espaço em branco. Se eu fosse instalar um wordpress, poderia colocar os seguintes comandos:

```
apt-get update
apt-get install git apache2 php5 mysql-server mysql-client -y
git clone https://github.com/WordPress/WordPress
```

Todos eles seriam executados na sequência e todos no mesmo step.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Entendendo o XML

Existe uma série de **Tags** dentro do arquivo config.xml, mas por enquanto precisamos entender basicamente duas:

```
<userRemoteConfigs>
   <hudson.plugins.git.UserRemoteConfig>
     <url>git@gitlab.dexter.com.br</url>
     <credentialsId>jenkins</credentialsId>
   </hudson.plugins.git.UserRemoteConfig>
</userRemoteConfigs>
```

As Tags acima são as configurações do repositório Git

Nessa seção (userRemoteConfig) ficarão armazenadas as configurações do sistema de controle de versão que você pode utilizar, sendo os mais comuns:

- Git
- SVN
- CVS

Porém, é possível instalar novos plugins e utilizar outro SCM que você queira como, por exemplo, mercurial.

Você pode encontrar informações sobre ele no seguinte link:

https://wiki.jenkins-ci.org/display/JENKINS/Mercurial+Plugin

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Entendendo o XML

Outra tag importante é a **builders**

```
<builders>
  <hudson.tasks.Shell>
     <command>comando_exemplo</command>
  </hudson.tasks.Shell>
</builders>
```

Estas são referentes ao **steps** da Job, nesta seção poderemos adicionar novos steps.

Nessa seção **builders** são armazenados todos os Steps configurados na Job.

Os Steps possíveis são:

- Executar no comando do Windows.
- Executar Shell.
- Invoke Ansible Ad-Hoc Command.
- Invoke Ant.
- Call Maven Targes.
- Call Builds on other projets.

Esse último é bem interessante, pois você pode configurar um **pipeline** de Jobs. Assim, se uma job for executada com sucesso ela pode invocar uma próxima job e assim por diante como, por exemplo, o deploy de uma aplicação que pode seguir o seguinte fluxo:

Clone Project → Build → Unit Test → Deploy

Cada uma dessas etapas é uma job diferente.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Alterando o repositório

A parte de alterar o repositório, podemos fazer um simples replace, do valor que está atualmente no XML para um novo, vamos subtituir pelo que foi criado na aula de Gitlab e gravar em um arquivo temporário:

```
with open('xml_exemplo.py','r') as f:
  config_xml = f.read().replace('git@gitlab.dexter.com.br',
   "git@gitlab.dexter.com.br:devops/aplicacao-exemplo.git")

# arquivo temporário
with open("/tmp/temp_job.xml",'w') as f:
  f.write(config_xml)
```


Nem sempre é necessário gravar a job temporária em um arquivo. Todas as modificações podem ser durante o processamento do script, mas é interessante saber que existe essa possibilidade, pois pode ser que seja necessário ter vários templates de jobs disponíveis dependendo da quantidade de projetos que você trabalhar.

No slide foi gravado o arquivo no /tmp mas também é possível através do pytthon criar um diretório e armazenar o arquivo dentro dele. Para fazer isso você precisará utilizar um módulo chamado **os**.

Por exemplo:
Import os

os.mkdir("pasta1",777)

O método **mkdir** recebe dois parâmetros, sendo um o caminho do diretório a ser criado e o segundo a permissão desse diretório.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Instalando módulo

Agora na questão dos **Steps da Build** já é um pouco mais complicado, pois é necessário fazer a leitura do XML, encontrar os elementos e adicioná-los no lugar certo. Para fazer isso, precisamos utilizar um módulo específico para trabalhar com arquivos do tipo XML.

```
sudo apt-get install python-lxml
```


O módulo **lxml** do python nesse caso foi instalado via **apt-get,** pois ele necessita de outras dependências do sistema das quais o pip não consegue resolver, uma vez que ele consegue gerenciar dependências somente dos módulos do python.

O Módulo lxml foi escolhido para a aula, pois é o mais robusto dos módulos. No entanto, existem outros que são relativamente simples de serem usados. Alguns deles são:

- Minidom
- Pulldom
- pyXML

Caso você queria ver uma lista mais completa, é só acessar o link a seguir:

https://wiki.python.org/moin/PythonXml

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Alterando os Steps do Jobs

A primeira coisa que devemos fazer para conseguir alterar os steps é importar o módulo lxml, abrir o arquivo que queremos trabalhar e transformá-lo em um objeto do tipo lxml, podemos fazer o seguinte:

```
from lxml import etree
with open("temp_job.xml",'r') as f:
    config_xml = f.read()
root = etree.XML(config_xml)
```


A parte mais interessante do módulo **ElementTree**, é que ele consegue tratar os arquivos XML's como se fossem listas. Assim, quando você quiser adicionar um novo elemento é só efetuar um **elemento.append(filho)**, e será adicionado ao final da lista o novo elemento.

Porém, sempre for necessário gravar em um arquivo é necessário converter esse objeto para String. Para isso execute a instrução abaixo:

convertido = etree.tostring(objeto_element_tree)

Depois, grave em um arquivo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

A partir da variável root podemos encontrar todos os elementos. No nosso caso, por exemplo, é interessante buscar por um elemento chamado **builders**, pois é nele que estão os comandos que serão executados pela Job.

```
for b in root.findall("builders"):
    builder = b
```


O método **findall** do **ElementTree** faz a busca no xml inteiro pela tag **builders.** No nosso caso, isso funciona pois só existe uma tag no documento inteiro. Se houvesse mais de uma, poderia ser necessário analisar a lista resultante da instrução **root.findall()** e ver quais foram retornados, pois esse laço que fizemos armazena na variável builder o último elemento encontrado.

Você pode utilizar também o método **find().** Se ele retornar o primeiro elemento que for encontrado no seu arquivo xml, a sintaxe dele é basicamente igual a do **findall()**.

O método find é utilizado para buscar o primeiro elemento dentro de todo o arquivo XML. Para ver os filhos de um elemento em específico, ou seja os elementos que podem existir dentro desse elemento, é utilizado o método iterchildren(). Por exemplo:

```
for c in buider.iterchildren():
    print b.tag,b.text
Saida:
 <Element command at 0x7f24a8989e9368>
```

Toda a documentação do ElementTree está disponível no link abaixo:

https://docs.python.org/2/library/xml.etree.elementtree.html

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Para adicionar um **novo filho** à um elemento já existente, precisa-se primeiro criar o elemento e depois fazer um **append** ao elemento PAI. O **ElementTree**, que é a função que importamos do módulo lxml, considera o arquivo xml com listas, então, pode-se usar as mesmas operações das listas para tratar esse tipo de arquivo.

Exemplo:

```
novo_comando = etree.Element("command")
novo_comando.text  = "echo testando novo comando"
    builder.append(novo_comando)
```


Quando vamos criar um subelemento para adicionar a uma lista, existem duas formas de se fazer isso. Você pode criar um objeto **etree.SubElement** ou um objeto **etree.Element.** Na teoria o correto é pegar um elemento já existente e adicionar a ele um subElement.

Mas pode-se também criar um objeto do tipo **etree.Element** e, depois, fazer um **objeto_pai.append(novo_elemento)**, como é o exemplo mostrado no slide.

No nosso caso o elemento **builder** foi buscado anteriormente e agora recebe como filho o elemento **command**.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

Todas as alterações no XML foram feitas somente em tempo de execução do script. Assim que ele for finalizado essas alterações serão perdidas, então, é necessário fazer a **persistência** em um arquivo.

Podemos usar a mesma função **open** para escrever em um arquivo, mas ela recebe como parâmetro uma **string**. Então, é necessário converter o xml em string para depois gravar no arquivo.

Exemplo:

```
with open('job_modificada.xml','w') as f:
            f.write(etree.tostring(root))
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Jobs com Python

No nosso caso não será necessário fazer isso, uma vez que o xml é passado como parâmetro para a função **create_job()**, uma string contendo o XML.

Então, o código pode ficar da seguinte maneira:

```
srv.create_job("NovaJob",etree.tostring(root))
```

A Job já deve ser criada automaticamente com os parâmetros que foram definidos através do script.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Cloud Computing

Objetivos da Aula

- ✔ Entender o que é computação na Nuvem;
- ✔ Conhecer algumas das suas características;
- ✔ Conhecer os tipos e modelos da Cloud Computing.

Introdução a Cloud Computing

Nesta aula iremos aprender o conceito de Cloud Computing, conhecer suas características e conhecer os tipos e modelos de Cloud existentes.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Cloud Computing

Introdução

Cloud Computing ou Computação em Nuvem é um formato computacional, no qual aplicativos, dados e recursos de TI são disponibilizados aos usuários como serviço, por meio da Internet, e pagos de acordo com a utilização dos recursos disponibilizados.

Computação nas nuvens refere-se à possibilidade de acessar arquivos e executar diferentes tarefas pela internet. Ou seja, você não precisa instalar aplicativos no seu computador para tudo, pois pode acessar diferentes serviços online para fazer o que precisa, já que os dados não se encontram em um computador específico, mas sim em uma rede.

Uma vez devidamente conectado ao serviço online, é possível desfrutar suas ferramentas e salvar todo o trabalho que for feito para acessá-lo, depois, de qualquer lugar. Justamente por isso que o seu computador estará nas nuvens, pois você poderá acessar os aplicativos a partir de qualquer computador que tenha acesso á internet.

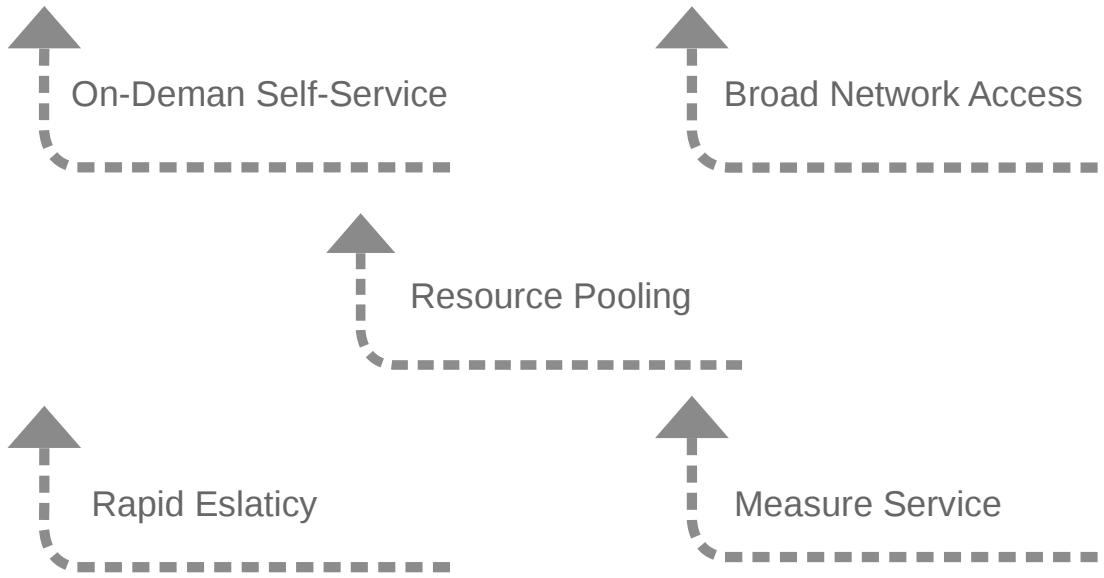
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Cloud Computing

Características da Computação na Nuvem



On-demand Self-Service: O cliente pode usar os serviços da nuvem e, caso achar necessário, aumentar ou diminuir as capacidades computacionais alocadas, armazenamento de dados, uptime do servidor, etc. Tudo isso, sem interação humana com o provedor de serviços utilizando API ou softwares.

Broad Network Access: Ter um amplo acesso a rede significa que os serviços de nuvem são acessíveis de qualquer plataforma. São utilizadas várias tecnologias e metodologias que possuem uma interoperabilidade do sistema. Assim, o cliente pode acessar os serviços de um smartphone, desktop, laptop ou qualquer outro dispositivo.

Resource Pooling: Os recursos computacionais da nuvem ficam reunidos geograficamente. Seus recursos virtuais são dinamicamente atribuídos ou retribuídos para que o cliente confirme a sua demanda. O cliente não possui controle sobre a real localização dos recursos que serão utilizados, tendo somente uma informação ampla como o país em que se encontra o Datacenter.
Os tipos de recursos são: armazenamento, processamento, memória, banda e máquinas virtuais. Até mesmo nuvens privadas tendem a reunir seus recursos entre partes das organizações.

Rapid Elasticy: Elasticidade é definida como a capacidade de alocar mais ou menos recursos no momento em que for necessário, com agilidade. Na ótica do consumidor, a nuvem parece ser infinita, pois ele pode adquirir quanto mais ou menos poder computacional for necessário para suas aplicações. Essa é uma das principais características que torna a Cloud Computing um serviço muito atrativo.

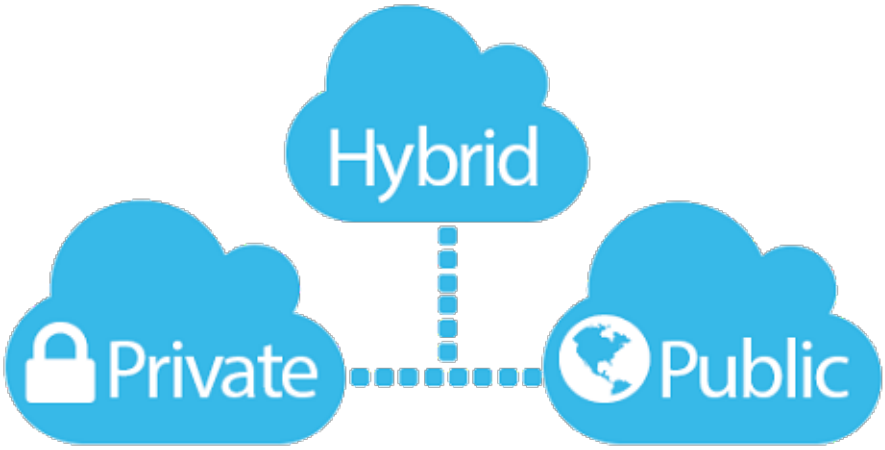
Measured Service: Todos os serviços são controlados e monitorados automaticamente pela nuvem, de maneira que fica tudo transparente tanto para o consumidor quanto para o fornecedor. Isso ajuda o consumidor a otimizar sua utilização da nuvem de acordo com sua produção e ajuda o provedor na hora de cobrança dos recursos.

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Cloud Computing

Tipos de Cloud: pública, privada e híbrida



Public Cloud (Nuvem Pública)

Um formato de cloud em que qualquer usuário conectado a internet pode utilizar e comprar seus recursos. Esse hardware normalmente é compartilhado entre os clientes, o que, em alguns casos ,pode gerar problema de segurança.

Private Cloud (Nuvem Privada)

Um formato de cloud em que apenas uma empresa ou grupo é responsável por utilizar e, na grande maiorias dos casos, gerenciar. Os recursos são dedicados ao uso dos funcionários ou parceiros de negócios da empresa.

Community Cloud (Nuvem Comunitária)

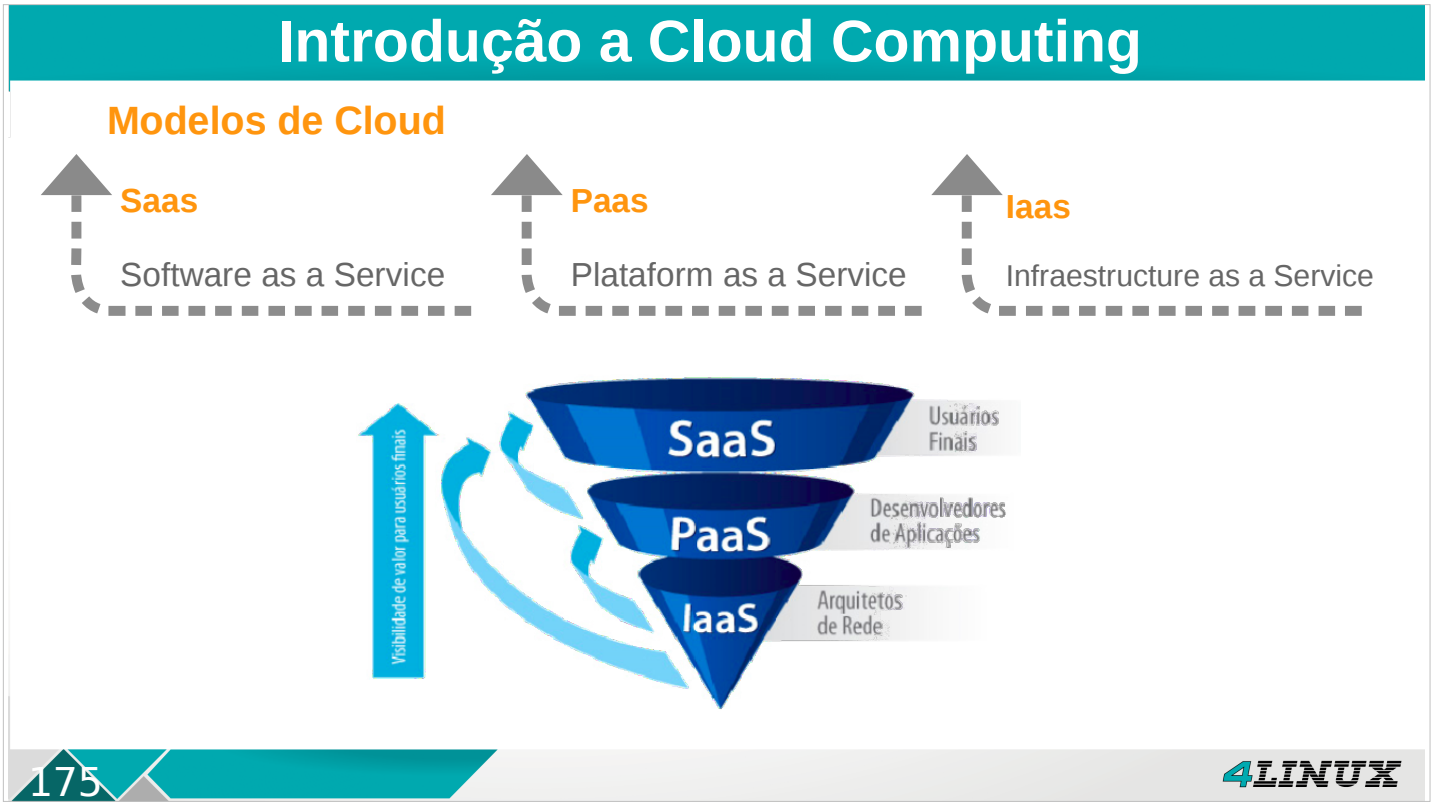
A Cloud Comunitária é compartilhada por várias organizações que partilham interesses como a missão, requisitos de segurança, políticas, entre outros. Funciona como uma rede de hospitais, por exemplo. Pode ser administrada pelas próprias empresas ou por terceiros.

Hybrid Cloud (Nuvem Híbrida)

Cloud Híbrida ou Nuvem Híbrida é uma composição de duas ou mais nuvens, sejam elas privadas, públicas ou comunitárias.

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com



Esses são os 3 principais modelos de serviços em Cloud Computing. Existem muitos outros serviços, mas vamos focar nesses.

A diferença básica entre esses tipos de serviços em cloud é a quantidade de serviços que você gerencia.

IaaS – Significa Infrastructure as a Service, traduzindo Infraestrutura como Serviços. Nesse tipo de serviço, o mais comum é se contratar um servidor cloud, que é um servidor virtual, onde você seleciona o sistema operacional que quer usar e, a partir daí, todo o gerenciamento da máquina é de responsabilidade do contratante.

PaaS – Significa Platform as a Service, traduzindo Plataforma como Serviços. Nesse tipo é muito comum se contratar uma instância de banco de dados relacionais, bancos noSQL ou até mesmo armazenamento de arquivos, como é o caso do S3 da Amazon.

SaaS – Significa Software as a Service, traduzindo Software como Serviço. Nessa categoria são contratados serviços em específico como, por exemplo, um ecommerce, no qual você não precisa configurar absolutamente nada técnico. A responsabilidade que o contratante tem é somente de cadastrar e vender os produtos.

Felipe Moz / moz.felipe@gmail.com

Introdução a Cloud Computing

Separação dos Modelos de Cloud

Local	IaaS	PaaS	SaaS
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking

Outro gerencia — Você gerencia

O slide acima mostra exatamente a quantidade serviços que o contratante tem que administrar, separados entre as camadas de serviços de cloud computing, nos quais a administração desses serviços vai diminuindo de acordo com a categoria.

No SaaS, por exemplo, o cliente não precisa ter muito conhecimento técnico. Ele pode, simplesmente, contratar e usar. Na maioria das vezes, o serviço pode ser cancelado sem custo algum, como são nos casos de plataforma de ecommerce.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Objetivos da Aula

- ✔ Conhecer a AWS;
- ✔ Conhecer os serviços disponíveis;
- ✔ Criar conta no nível gratuíto.

Conhecendo a AWS

Nesta aula iremos conhecer um pouco sobre a AWS e seus serviços, como estão disponíveis no nível gratuíto e criar uma conta.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

AWS (Amazon Web Services) é uma coleção de serviços em nuvem, também conhecido como web services, e oferecidos pela Amazon.

A empresa possui serviços operando em 12 regiões diferentes espalhadas pelo mundo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

AMI (Amazon Machine Image) é um Serviço de imagens utilizado para a criação de máquina (EC2) no ambiente Cloud da Amazon.

É uma appliance virtual que serve como base para o sistema operacional que irá ser usada em instâncias EC2.

EC2 (Elastic Cloud Compute) é um serviço web que disponibiliza máquina virtuais (Instâncias EC2) no ambiente da Amazon. As instâncias do EC2 são criados a partir das AMI's.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

VPC (Virtual Private Cloud) é um serviço que disponibiliza um ambiente lógico privado dos Serviços da Amazon. Utilizando VPC você pode criar uma segurança a nível lógico e gerenciar redes e subredes das instâncias dos serviços.

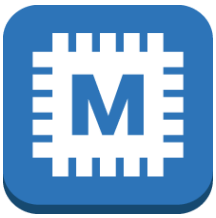
S3 (Simple Storage Service) é um serviço que disponibiliza armazenamento de objetos de forma segura e escalável. O S3 pode ser usado para armazenar basicamente qualquer coisa.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS



ELC (Elastic Cache) é um serviço que disponibiliza o armazenamento em cache de memória escalável. Você pode escolher entre dois engines, uma instância do Memcache ou do Redis Server.

RDS (Relational Database Service) é um serviço que disponibiliza instâncias de Banco de Dados, de forma escalável, backup e Failover automático. Disponibiliza instâncias de PostgreSQL, MariaDB, MySQL, entre outros bancos.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Cloud Formation é um serviço da Amazon que oferece a capacidade para provisionar serviços de forma fácil através de arquivos de configurações. Você escreve um arquivo no formato JSON, descrevendo os serviços e os parâmetros de configurações e provisiona o ambiente.

Lambda é um serviço que permite a você executar códigos sem a necessidade de gerenciar ou criar servidores. Você pode configurar o lambda para executar códigos a partir de eventos disparados por outros serviços da Amazon.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Nível Gratuito da AWS:

Serviços disponibilizados gratuitamente por um ano.

http://aws.amazon.com/pt/free/

Nível gratuito da AWS

O nível gratuito da Amazon Web Services (AWS) foi concebido para permitir que você obtenha experiência prática com os serviços da Nuvem AWS. O nível gratuito da AWS inclui serviços com um nível gratuito disponível por 12 meses após a data do seu cadastro na AWS, além de ofertas de serviço adicionais que não expiram automaticamente ao final do período de 12 meses do nível gratuito da AWS.

Depois de criar a sua conta da AWS, você poderá usar gratuitamente qualquer um dos produtos e serviços listados a seguir, dentro de determinados limites de utilização.

Você pode começar hoje e aproveitar automaticamente o nível gratuito da AWS seguindo as etapas abaixo:

1. Cadastre-se para abrir uma conta da AWS.
2. Insira o endereço de faturamento e as informações do cartão de crédito. Você não será cobrado, a menos que ultrapasse os níveis de uso gratuito.
3. Comece a usar os serviços em nuvem da AWS escolhendo qualquer um dos produtos a seguir.

183 4LINUX

Essa conta gratuita que pode-se fazer na AWS é muito interessante para Start ups, pois como tem recursos limitados, uma instância da AWS pode ser utilizada por até um ano sem custo algum. É muito útil quando se quer ter testar a divulgação de algum produto. É possível, por exemplo, hospedar uma landing page em um EC2 ou até mesmo utilizar outros serviços, como serão mostrados nos próximos slides.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Nível gratuito EC2:

750 Horas por mês de uso de instância t2.micro Linux RHEL.

(1 Instância/mês ou 2 Instâncias/duas semanas)

Expira em 12 meses após o cadastro.

Nível gratuito Elastic Load Balancer:

750 Horas por mês.

15 GB de processamento de dados.

Expira em 12 meses após o cadastro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

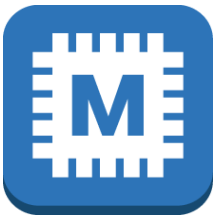
Nível gratuito S3:

5 GB de armazenamento padrão.

20.000 requisições GET.

2.000 requisições PUT.

Expira em 12 meses após o cadastro.



Nível gratuito ElastiCache:

750 horas de uso de instância cache.t2.micro, suficiente para execução contínua a cada mês.

Expira em 12 meses após o cadastro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Nível Gratuito RDS:

750 horas de uso de instância db.t2.micro Single-AZ.

20 GB de armazenamento de banco de dados: qualquer combinação de propósito geral (SSD) ou magnético.

10.000.000 de E/S.

Expira em 12 meses após o cadastro.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo a AWS

Nível Gratuito CloudWatch:

10 métricas personalizadas e **10 alertas.**

1.000.000 de requisição de API.

5 GB de ingestão de dados de log e **5 GB** de arquivos de dados de log.

3 paineis com até **50 m**étricas por mês para cada um deles.

Não expira ao final do período de 12 meses do nível gratuito.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Objetivos da Aula

- ✔ Conhecer o painel da AWS;
- ✔ Criar e gerenciar instancias do EC2;
- ✔ Criar credenciais para trabalhar com API.

Configurando EC2 na AWS

Nesta aula iremos conhecer o painel da AWS, criar e gerenciar instancias EC2 e criar as credenciais para trabalhar com a API da AWS.

Anotações:

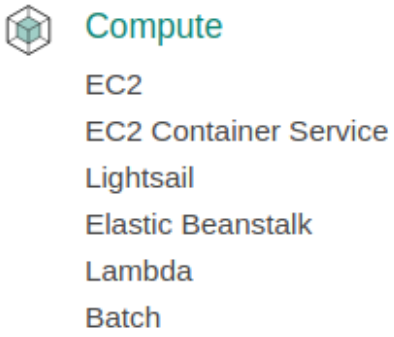
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Menu AWS

No menu de serviços da AWS, selecione a opção EC2 para gerenciar a criar novas instancias:



Toda a configuração de serviços da AWS pode ser realizada através do próprio painel, simplesmente com alguns cliques é possível ter uma infraestrutura completa e redundante.

Mas é necessário ter algum conhecimento técnico também para poder entender alguns conceitos como, por exemplo, Subnets, Redirects, EBS, Zonas, entre outros.

No próximos slides temos as informações sobre como criar instâncias do tipo EC2, mas a amazon tem muitos serviços,inclusive certificações próprias para melhorar esses serviços.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Na página principal do **EC2**, você pode observar um painel com informações sobre as configuração do serviço como, por exemplo, quantas instâncias estão ativas no momento, quantos Elastic Ips, Security Group, entre outras informações.

You are using the following Amazon EC2 resources in the US East (N. Virginia) region:

0 Running Instances	0 Elastic IPs
0 Dedicated Hosts	0 Snapshots
0 Volumes	0 Load Balancers
2 Key Pairs	9 Security Groups
0 Placement Groups	

Esse é um overview da sua conta na AWS.

Running Instances – São as suas instâncias sendo executadas no momento.

Dedicated hosts – São máquinas dedicadas na sua conta.

Volumes – São basicamente a quantidade de discos que você tem para armazenar dados.

Key Pairs – Par de chaves para acessar as Instâncias por SSH.

Elastic IPS – São a quantidade de IP's públicos que você tem no momento. Esses elastic IP's são trocados quando uma instância é reiniciada. Caso você precise de um IP fixo, será necessário contratar um IP dedicado.

Snapshots – São pontos de restauração para as suas instâncias.

Security Grupos – São grupos com regras de firewall para as suas instâncias.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Agora que temos o nosso par de chaves configurado, vamos criar a nossa instância EC2.
Selecione a opção **Launch Instances** para criar novas instâncias do serviço.

O caminho para a instalação de um EC2 é basicamente: Next → Next → Finish, caso você não tenha que efetuar configurações específicas.

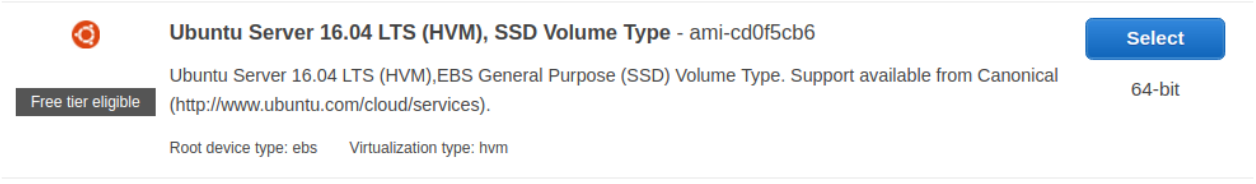
No nosso caso podemos utilizar toda a configuração padrão, pois posteriormente toda a administração será feita por um módulo do python e não precisaremos mais acessar o painel.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Para o nosso laboratório, iremos utilizar AMI do Ubuntu Server 16.04. Selecione a imagem clicando no botão **SELECT**.



Note que a opção que selecionamos é HVM, que é uma máquina virtualizada totalmente independente do seu Hypervisor, servidor físico que hospedará a nossa instância Cloud. Podemos chamar isso de Full Virtualization, onde a máquina virtualizada é totalmente independente da máquina que virtualizou. É isso que permite que seja virtualizado um sistema Windows em cima de servidor Linux.

Já PV que significa ParaVirtualization. Você não consegue instanciar servidores cloud com um sistema operacional diferente do hypervisor. No caso do linux, por exemplo, as máquinas virtualizadas sempre utilizarão o mesmo kernel da máquina hospedeira, como é o caso do Docker. No entanto, este tipo é mais rápido e consome menos recursos do que o formato HVM.

Ambos os formatos são bons, sendo que cada um atende a um propósito.

Anotações:

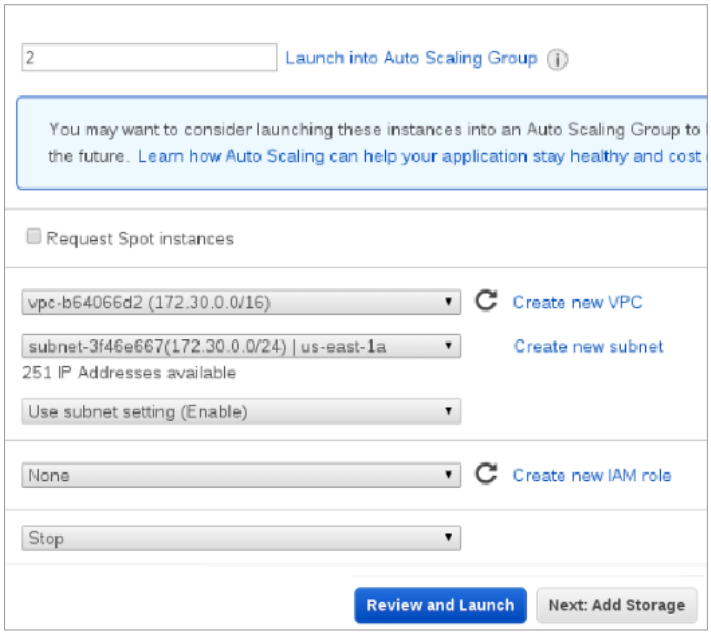
Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Insira o número **2** para criar duas instâncias idênticas.

Selecione a **VPC** e **Subnet** configurada.

Após a configuração, clique em **Next: Add Storage**.



As VPC's na amazon são redes que você pode criar para as suas instâncias EC2. Assim, elas podem ficar isoladas logicamente para que uma não tenha acesso a outra, o que acaba trazendo mais segurança para as suas instâncias.

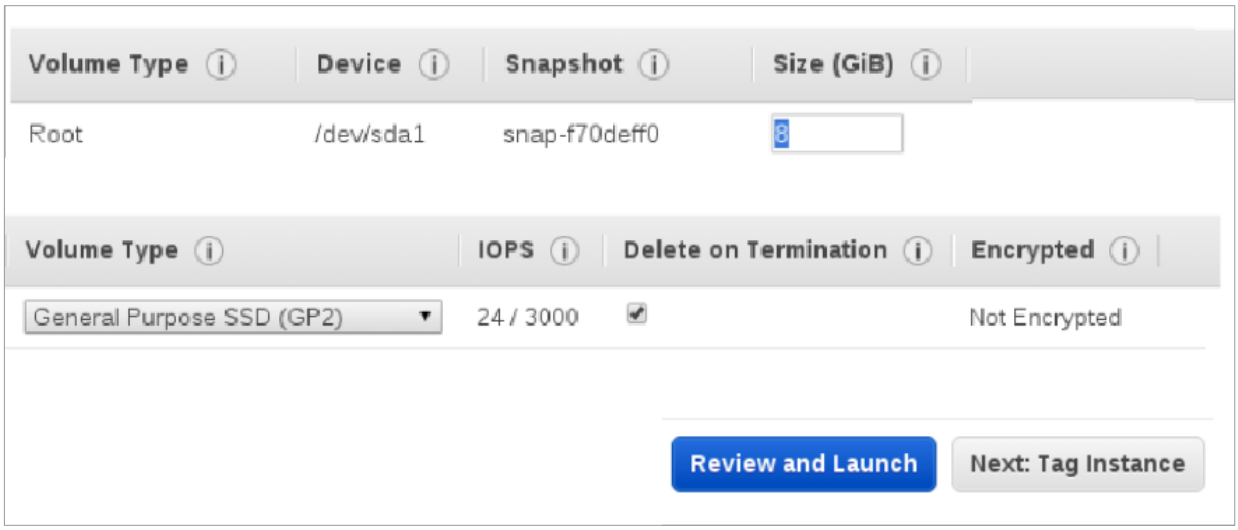
Mas essa configuração de segurança também pode ser feita através de subnets, assim, as máquinas precisarão ter rotas configuradas e compartilhar um gateway em comum para trafegar dados entre elas.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Clique em “Review and Lauch” para finalizar a criação do EC2.



Quando criamos volumes para as instâncias, é muito comum utilizar os tipos SSD, pois eles apresentam maior performance na gravação e leitura de arquivos.

Quando selecionamos esse tipo de SSD, ele ainda tem 2 subcategorias, que são:

General Purpose - É o melhor custo beneficio, balanceando performance e reço.

Provisioned IOPS – É o formato mais rápido, porém, o seu custo é mais elevado se comparado ao primeiro tipo. É recomendado a ser utilizado em ambientes críticos.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Criar credenciais

Ao criar uma credencial de acesso através do Boto3, é necessário ter um **IAM user** da Amazon.

Para fazer isso, clique em **Services > IAM**.

IAM significa Identity and Access Management. Elas servem para que possamos criar automações em nossos scripts, pois não convém disponibilizar o login e a senha do painel em todos os scripts que forem executar tarefas na AWS.

Com as chaves IAM pode-se ter um par de chaves específico para cada script e, caso você queria cortar o acesso de um determinado usuário ou sistema, é só desabilitar a chave IAM.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Selecione o usuário que você quer utilizar para fazer a integração.

	User Name	Groups	Password	Password Last Used	Access Keys	Creation Time
☐	aws_controle	1		N/A	1 active	2014-06-03 12:58 UTC...
☐	controller	1	✔	Never	1 active , 1 inactive	2013-03-11 16:05 UTC...

Após selecionar o usuário, você já cairá na aba **Security Credentials**. Nela você deverá chegar a uma chave de acesso.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Recapitulando

- ✔ Conhecer a API do Jenkins;
- ✔ Criar novas Jobs;
- ✔ Listar as Jobs criadas;
- ✔ Iniciar Jobs.

Configurando EC2 na AWS

Nesta aula conhecemos a API do Jenkins, criamos, iniciamos e iniciamos Jobs.
Vimos como podemos listar as Jobs criadas e buscar informações sobre a mesma.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Configurando EC2 na AWS

Access Keys

Use access keys to make secure REST or Query protocol requests to any AWS service API. For your protection, you should never share your secret keys with anyone. In addition, industry best practice recommends frequent key rotation. Learn more about Access Keys

Create Access Key

Access Key ID	Created	Last Used	Last Used Service	Last Used Region	Status	Actions
AKIAIESZBUDI74JYF35Q	2013-03-12 11:52 UTC-0300	N/A	N/A	N/A	Inactive	Make Active | Delete
AKIAIIFJBSSYUIK3UG4A	2016-03-07 16:25 UTC-0300	2016-03-07 20:52 UTC-0300	ec2	us-east-1	Active	Make Inactive | Delete

Após clicar em gerar chave será fornecido o **Access Key ID** e o **Secret Access Key**.

Não esqueça de **selecionar a zona** para criadas as suas máquinas.

Esses serão os dados necessários para fazer a autenticação por script.

198 4LINUX

Zonas da AWS são regiões geográficas onde os seus serviços serão instalados. Quando se trabalha com clusters convém que cada uma das máquinas que você crie seja em uma zona diferente, pois, assim, a chance de indisponibilidade diminui muito.

Caso você queira uma lista com as zonas e regiões da AWS, elas podem ser encontradas no link abaixo:

http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Objetivos da Aula

- ✔ Conhecer o módulo Boto3;
- ✔ Configurar autenticação;
- ✔ Criar e listar instâncias EC2

Gerenciando AWS com Boto3

Nesta aula iremos conhecer o módulo que fará a integração com a AWS, o Boto3, faremos a sua configuração e criaremos algumas instâncias EC2.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Configurando autenticação

Em máquinas Linux, é necessário criar um arquivo que irá armazenar essas credenciais.

Crie um diretório chamado **.aws** no diretório Home do seu usuário.

Exemplo:

```
# mkdir ~/.aws
```

Dentro deste diretório você precisará de 2 arquivos: um chamado **credentials** e outro chamado **config**. Onde armazenarão a chave de acesso e a região em que você vai criar as instâncias

Como o curso é totalmente baseado em Linux, abordaremos na aula a configuração em ambientes linux somente. Caso você seja um usuário de windows, é possível também efetuar a configuração pelo PowerShell. No link abaixo existe mais informações sobre:

http://docs.aws.amazon.com/powershell/latest/userguide/specifying-your-aws-credentials.html

Lembre-se que as chaves estão sendo configuradas dentro do diretório home em que você está logado, então, o seu script só funcionará para esse usuário.

Caso você queira conectar com outro usuário, precisará fazer a mesma configuração, ou tente efetuar uma configuração genérica, mas é interessante manter um usuário que irá gerenciar as suas rotinas de DevOps.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para editar o arquivo **credentials**, digite em seu terminal:

```
# vim ~/.aws/credentials
```

O conteúdo do arquivo deve ser o seguinte:

```
[default]
aws_access_key_id = CHAVEEXEMPLO
aws_secret_access_key = ZXCÇL@#!CZXC!@#
```

Não esqueça de substituir os valores acima para os que foram gerados na hora de criar a chave de acesso!

Todas as informações a respeito da configuração do seu ambiente podem ser encontradas na documentação oficial:

http://boto3.readthedocs.org/en/latest/guide/quickstart.html#configuration

Uma vez realizadas essas configurações, você já pode criar o seu script e automatizar a criação do seu ambiente na AWS.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para editar o arquivo **config**, digite em seu terminal:

```
# vim ~/.aws/config
```

O conteúdo do arquivo deve ser o seguinte:

```
[default]
region=us-east-1
```

Lembrando que esta é a região você especificou ao criar a chave, agora já pode-se utilizar o **Boto3** para fazer a integração com a AWS.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Esse módulo serve para integrar qualquer um dos serviços da Amazon. No nosso caso, vamos utilizar o **EC2**.

```
ec2 = boto3.resource("ec2")
```

Para buscar todas as instâncias, pode-se utilizar o seguinte recurso:

```
instances = ec2.instances.all()
```


Durante a aula não teremos como abordar todos os métodos relacionados às instâncias EC2. Caso você queira ver uma lista completa ela pode ser encontrada no seguinte link:

http://boto3.readthedocs.org/en/latest/reference/services/ec2.html

Lembre-se que você pode descobrir isso através da própria linguagem python, usando o método **__dict__** que mostra todos os atributos e valores de um objeto instanciado, ou por exemplo o **dir()** que é mostrado no slide que mostra os métodos.

Pode-se também através do console do python executar as seguintes instruções:

```
>> Import boto3
>>
>> help(boto3)
```

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para buscar uma instância em específico, pode-se utilizar o método **filter**. Exemplo:

```
instances = ec2.instances.filter(
            Filter[
                  {"Name":"instance-state-name",
                   "Values":["stopped"]
                  }
               ])
```

Assim serão retornadas todas as instâncias cujo o estado é igual a **stopped**, ou seja, as instâncias que estejam paradas no momento.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para criar uma **nova instância**, pode-se utilizar a seguinte instrução:

```
new_instance = ec2.create_instances(
                        ImageId='ami-b2e3c6d8',
                        MinCount=1,
                        MaxCount=1)
```

O **ImageId** deve ser uma imagem disponível na AWS. Este AMI deve ser verificado no próprio site da Amazon.
MinCount e **MaxCount** são a quantidade de instâncias que você quer criar. Nesse caso, criamos somente uma em InstanceType, a máquina **t2.nano**, que é o plano mais barato da amazon.

Todos os atributos de uma instância EC2 podem ser parametrizados através do boto3. Nesse exemplo passamos somente os obrigatórios para se criar uma instância, mas se você precisar de mais parâmetros, pode-se verificar na documentação oficial conforme o link abaixo:

http://boto3.readthedocs.org/en/latest/reference/services/ec2.html#EC2.Subnet.create_instances

Caso você precise criar muitas instâncias, é importante saber que existe um limite por região. Para saber qual é esse limite você pode acessar a documentação oficial no seguinte link:

http://aws.amazon.com/pt/ec2/faqs/#How_many_instances_can_I_run_in_Amazon_EC2

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Na variável **new_instance** foi armazenado o ID da nova instância.

Quando uma nova instância é criada, ela é inicializada automaticamente. Caso você queira parar essa instância, você pode utilizar a seguinte instrução:

```
stop_instance = ec2.instances.filter(InstanceIds=['i-
e7ae2364']).stop()
E caso queira remove-la você pode-se simplesmente alterar o método.
terminate_instance = ec2.instances.filter(
                     InstanceIds=['i-e7ae2364']).terminate()
```


Convém saber como parar e iniciar as instâncias EC2, pois como elas são cobradas por hora, pode ser necessário parar uma instância com o objetivo de reduzir custos. Isso é muito comum em ambientes Web em que existem picos de acesso em determinados horários.

Exemplo:
Você possui um e-commerce que tem uma média de 10 mil acessos diários, porém, quando chega o período entre as 18:00 e 20:00 esses acessos dobram para 20 mil. assim é necessário ligar uma segunda instância durante esse período e desligá-la após as 20:00, que é quando os acessos diminuem.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

É importante também gerenciar os **security groups**, uma vez que dependendo da aplicação que você for publicar em uma instância em **EC2**, pode-se precisar de portas diferentes das que vem na política padrão.
Para criar um **security group** utilize a seguinte instrução:

```
sg = ec2.create_security_group(GroupName="Python-SG",
                               Description="Grupo criado atraves do boto3")
```

A variável SG irá armazenar o valor do security group. Agora você pode criar uma instância e já associá-la a esse **security group**.

Essa é uma das partes mais importantes em se gerenciar um EC2, pois é muito comum efetuar a instalação de novos serviços na máquina e esquecer de liberar as portas de conexão na sua instância.

Por padrão a única porta de acesso que é liberada é a 25, que se refere aos serviços de SSH para que seja possível conectar na instância e efetuar a instalação dos serviços. Então, se você for instalar um servidor Web será necessário liberar a porta 80 e 443 se for utilizar HTTPS. Caso seja um servidor de e-mail precisará liberar as portas 25,143,110, 993,995, 587, o que pode ser uma tarefa árdua dependendo da quantidade de máquinas que você possui e da quantidade de portas que será necessário liberar.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para criar uma nova instância, pode-se utilizar a seguinte instrução:

```
new_instance = ec2.create_instances(
                        ImageId='ami-b2e3c6d8',
                        MinCount=1,
                        MaxCount=1.
                        SecurityGroupIds=[“sg-dc918cb6”])
```

Para validar o **security group**, você pode utilizar a seguinte instrução:

```
instance = ec2.instances.filter(InstanceId(InstanceIds=[“i-c7b53844”]))
      print next([ i.security_groups for i in instance ])
```


Nesse slide é criada uma instância com um security group especifico. É interessante sempre ter mais de um security group, pois você pode criar configurações padronizadas, como por exemplo:

Security Group – Email
Secuirty Group – WebServer
Security Group – Cpanel

A tarefa de ficar adicionando e removendo portas pode ser efetuada somente uma vez, então, se você já sabe qual é o serviço que irá ser executado naquela instância, já crie-a no security group específico.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Caso queira **listar todos os security groups** existentes, pode-se utilizar as seguintes instruções:

```
for s in ec2.security_groups.all():
   print s.group_name, s.group_id
```

Assim você pode saber todos os **security groups** existentes sem precisar logar no console da AWS.

Podemos também **adicionar e remover regras** desse security group.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para **adicionar uma regra**, você deverá seguir as seguintes instruções:

```
sg = ec2.SecurityGroup("sg-dc918cb6")
```

Essa instrução irá instanciar um objeto do tipo SecurityGroup. Agora você pode **adicionar e remover regras**. Exemplo:

```
sg.authorize_ingress(FromPort=80,ToPort=80,
                     CidrIP="0.0.0.0/0",IpProtocol="tcp")
```


A regra acima irá liberar a porta, sendo o CidrIP o endereço que será liberado para trafegar por essa porta na sua instância.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Gerenciando AWS com Boto3

Para **remover uma regra** é necessário mudar somente o método. Exemplo:

```
sg.revoke_ingress(FromTo=80,ToPort=80,CidrIp="0.0.0.0/0",IpProtocol="tcp")
```

Agora se você verificar no seu console da AWS, verá que não existe mais essa adicionada ao security group.
Caso você não precise mais desse Security Group basta **deletá-lo**, desde que não exista nenhuma instância ativa utilizando-o.

```
sg.delete()
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Objetivos da Aula

- ✔ Entender o que é conexão SSH;
- ✔ Conhecer algumas das suas características;
- ✔ Realizar a conexão entre os servidores.

Introdução ao SSH

Nesta aula iremos aprender o conceito de SSH, onde veremos algumas de suas características e como podemos realizar a conexão utilizando SSH.

Anotações:

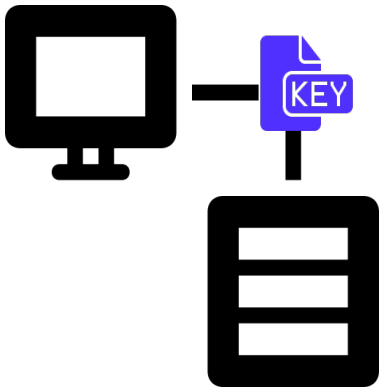
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Introdução

O **Serviço SSH** (Secure SHell) é usado para realizar Acesso Remoto de forma Segura. Basicamente toda a transferência de dados do servidor ao client (e vice-versa) é criptografada, utilizando SSH conseguimos garantir: autenticação, criptografia e integridade.



Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Segurança

1/4	Após a primeira conexão, ele armazena a identidade do Servidor (known_hosts) para garantir que você sempre acessará o servidor correto. Caso a identidade seja alterada, ele alertará.
2/4	O cliente transmite as informações de autenticação usando criptografia forte de 256 bits.
3/4	Todos os dado usam uma criptografia de 256 bits, o que torna, praticamente, impossível decifrar os dados.
4/4	Como SSH criptografa tudo, ele pode servir de tunelamento para outros protocolos inseguros (Tunelamento).

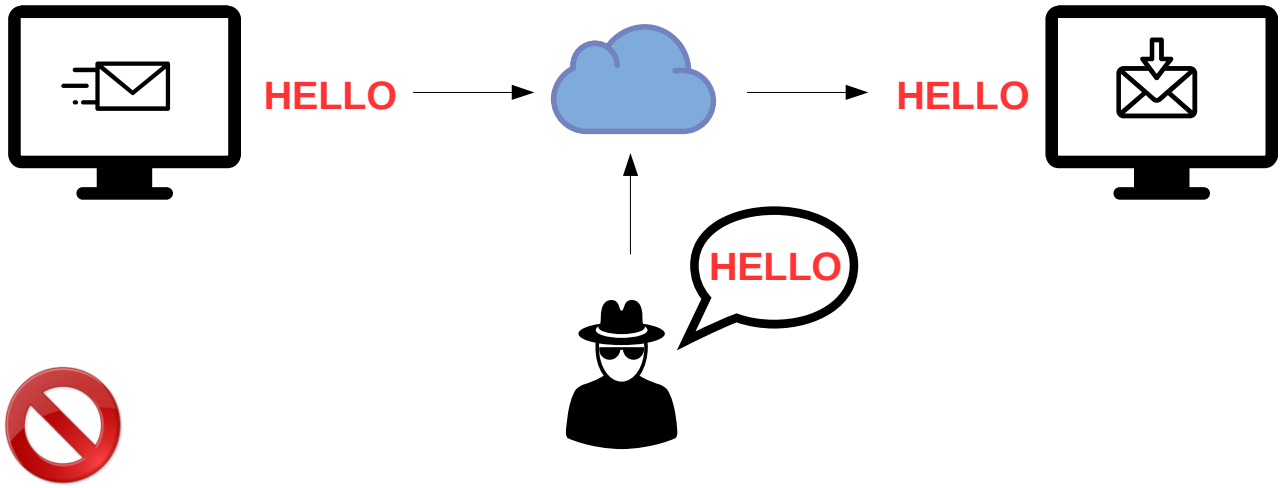
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Comunicação Insegura (sem criptografia)



Sistemas não criptografados

Existem sistemas de comunicação que não utilizam criptografia como por exemplo o FTP e o TELNET, o SSH surgiu como uma alternativa a este tipo de sistema

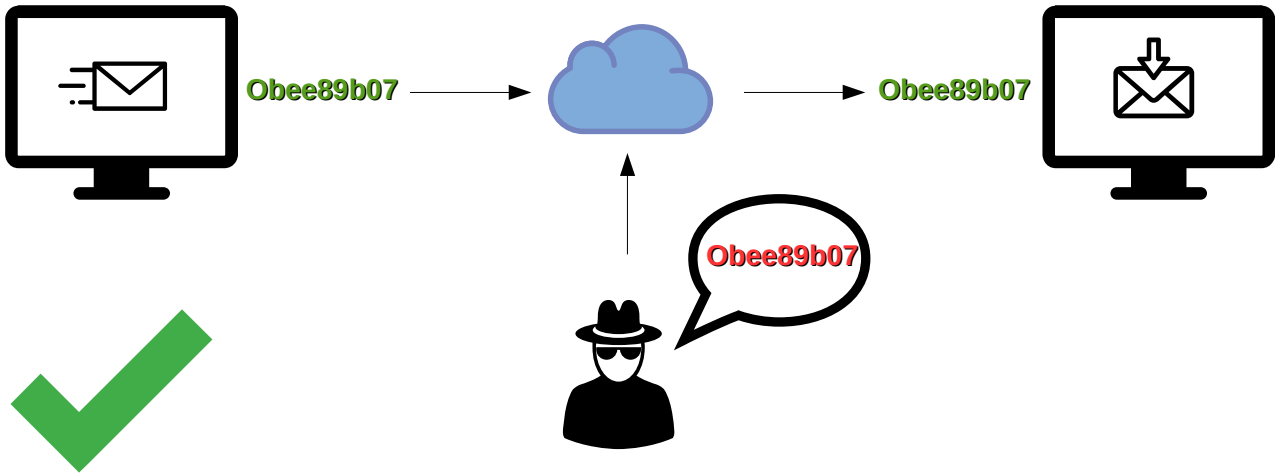
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Comunicação Segura (com criptografia)



Por que o SSH é um protocolo tão seguro?

O SSH é considerado seguro por utilizar algumas funcionalidades de transmissão e autenticação de usuários, provavelmente a principal dessas funcionalidades é o sistema de chaves criptográficas, onde uma chave criptográfica pública é utilizada para autenticar um computador remoto, Esse sistema de chaves também pode ser utilizado para autenticar usuários no sistema.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

SSH

Existe uma variedade de ferramentas que podem ser usadas para romper ou interceptar dados de uma comunicação com o objetivo de conseguir acesso a um sistema como, por exemplo, usar um sniffer para capturar dados que estão trafegando na rede;

Com o SSH essa ameaça é quase nula, pois o cliente e o servidor SSH usam assinaturas digitais para verificar a sua identidade. Além disso, toda a comunicação entre eles é criptografada.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

SSH & Cloud

É praticamente impossível falar de Cloud sem SSH.

Com o avanço do mercado de Cloud, o SSH passou a ser uma ferramenta vital para o SysAdmin quando o assunto é administrar servidores remotos.

É exatamente por isso que é extremamente importante configurar esse serviço de forma correta, visando sempre a segurança do acesso ao sistema.

SSH & Cloud

Ao contratarmos serviços na nuvem é muito comum que o cliente receba do fornecedor do serviço uma chave com a qual será feita a conexão no servidor, esta chave é acionada através do comando **"ssh"** com o parâmetro **"-i"** seguido do caminho para a chave que será responsável pela autenticação do usuário.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Conexão SSH

1 Conectando ao SSH

```
# ssh root@docker.dexter.com.br
```

```
The authenticity of host '192.168.200.20 (192.168.200.20)'
can't be established.
RSA key fingerprint is
55:02:60:1e:6c:27:cf:00:5a:73:ce:c5:47.
Are you sure you want to continue connecting (yes/no)?
yes
```

No primeiro acesso, será solicitado que você aceite a identificação do servidor que será armazenada no arquivo **~/.ssh/known_hosts**.

Arquivos de configuração SSH

Além do arquivo known_hosts, o SSH tem outros arquivos de configuração:

sshd_config – arquivo de configuração que o servidor irá ler ao iniciar.
authorized_keys – arquivo que específica quais usuários podem fazer login em um servidor (utilizam a chave pública).
ida_usa – chave privada para conexão SSH.
id_rsa.pub – chave pública para conexão SSH.

A chave (publica e privada) é gerada em um host onde a chave pública deve ficar no servidor e a chave privada ficará no host para a conexão ser realizada.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Verificando o arquivo known_hosts:

1 Visualizando o arquivo

```
# cat ~/.ssh/known_hosts
```

```
|1|qVbvPw6ghpuz6toM54Hkinsqb5w=|1Wm1Kxadw8KBv7vPdq2S60YTGkg=
ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyN
TYAAABBBLArNkwLwxUAZpvnz3WaV38tVvMJkKvXXiQBNWq8bSQTJLtb6DBEGP
IiMTzigoEO0QKb2b46tavV3cvu2KL6zvQ=
```

O próximo acesso ao servidor, não será mais necessário digitar “sim” para autenticar pois já existe a autenticação no arquivo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Intrudução ao SSH

Conexão SSH

Quando fazemos a conexão SSH pela primeira vez em um servidor, sempre será necessário aceitar o “fingerprint” para que a chave seja armazenada dentro do arquivo known_hosts.

Se o servidor que iremos conectar já possui uma chave no arquivo, então a conexão é feita sem a necessidade de aceitar esta chave novamente.

Este arquivo também é importante quando faremos a conexão utilizando o Python, pois teremos que consultá-lo ou aceitar a chave para conseguir realizar a conexão.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Objetivos da Aula

- ✔ Conhecer o módulo Paramiko;
- ✔ Realizar conexões SSH.

Conexões SSH com Python

Nesta aula iremos conhecer o módulo Paramiko e entender como podemos fazer conexões SSH.

Anotações:

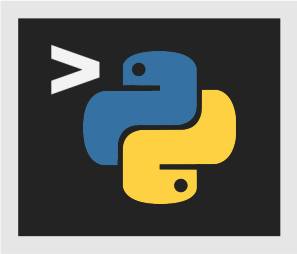
Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Introdução

Existe um módulo em Python que serve para fazer conexões via SSH, que é muito utilizado no mundo Devops.



Este módulo é usado por ferramentas como Ansible e Fabric, que fazem conexões SSH para o gerenciamento de configurações.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Instalação do módulo

Podemos fazer a instalação do módulo Paramiko da seguinte forma:

```
pip install paramiko
```

Agora vamos criar um arquivo chamado “ssh.py” para podemos começar a fazer as conexões SSH com este módulo.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Criando conexão

Para usar o paramiko, precisamos instanciar um objeto do tipo SSHClient.

```
from paramiko.client import SSHClient
import paramiko
client = SSHClient()
```

Esse objeto tem todos os métodos e propriedades necessárias para que possamos executar as conexões via ssh.

Ao executar o script abaixo é possível ver que um objeto SSHclient foi criado e a posição na memória em que ele está armazenado:

```
#!/usr/bin/python3

from paramiko.client import SSHClient
import paramiko

client = SSHClient()

print (client)
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

O **SSH** precisa que seja aceito o **fingerprint** quando acessamos um servidor pela primeira vez. Este fingerprint serve para garantir a autenticidade de um servidor. Para carregar os fingerprints dos servidores já acessados, utilizamos o seguinte comando:

```
client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

A segunda linha é responsável por aceitar o fingerprint automaricamente (caso seja o primeiro acesso ao servidor).

Agora que esses parâmetros foram adicionados, o script ficou dessa maneira:

```
#!/usr/bin/python3

from paramiko.client import SSHClient
import paramiko

client = SSHClient()

client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy)

print (client)
```

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Estabelecer conexão

Agora só resta estabelecer uma conexão. Para isso, utilizamos o método connect.

```
client.connect('192.168.0.2',
                    username='root', password='4linux')
```

Podemos executar comandos remotamente:

```
stdin,stdout,stderr = client.exec_command('ls -la')
```


O método acima retorna 3 valores: entrada padrão, saída padrão e saída de erro e, com eles, podemos saber se um comando foi executado com sucesso ou não.

gora que todos os parâmetros já foram ajustados e a conexão está ativa, podemos executar um comando remotamente e ver a sua saída:

```
#!/usr/bin/python3

from paramiko.client import SSHClient
import paramiko

client = SSHClient()

client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

client.connect("192.168.0.2")
stdin, stdout, stderr = client.exec_command("ls -la")

print (stdout.read())
```

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conexões SSH com Python

Os mesmos métodos existem tanto para entrada padrão, saída padrão como para saída de erro.

```
stderr.channel.recv_exit_status()
```

Este método irá retornar um valor inteiro. Se o comando foi executado com sucesso, retorno 0, caso contrário, um valor maior.

```
stdout.read()
```

O método acima irá exibir a mensagem retornada pelo comando executado.

Para saber se um comando executou com sucesso podemos usar o exit_status().

No Exemplo abaixo o comando ls foi digitado errado para gerar um erro:

```
#!/usr/bin/python

from paramiko.client import SSHClient
import paramiko
client = SSHClient()
client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect("192.168.0.2")
stdin, stdout, stderr = client.exec_command("lsx -la")

if stderr.channel.recv_exit_status() != 0:
    print (stderr.channel.recv_exit_status())
    print (stderr.read())
else:
    print (stdout.read())
```

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Objetivos da Aula

- ✔ Introdução aos Logs;
- ✔ Entender o formato de Logs;
- ✔ Arquivos de Log.

Introdução a Logs

Nesta aula iremos aprender alguns conceitos sobre Logs, o formato mais utilizado e onde são armazenados os seus arquivos.

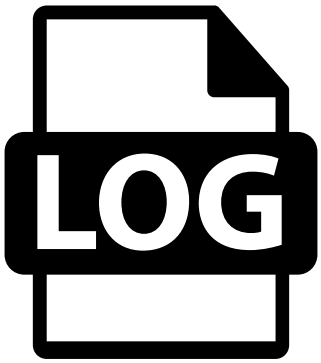
Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Introdução

Logs são formas de registrar informações relevantes sobre eventos que acontecem em seu sistema. Podem ser desde informações de Debug como até mesmo erros críticos que podem parar o funcionamento.



Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Formato dos Logs

Os **Logs** geralmente são gravados em arquivos de texto e devem ter informações importantes, como:

- Horário em que o evento aconteceu.
- Qual foi o evento ocorrido.
- Qual foi a data em que ocorreu.
- Nível de criticidade da informação.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Essas são algumas características básicas que um Log, para ser eficiente, deve conter. Caso você queria saber mais informações que podem ser colocadas nos Logs, a **ISO 27001** tem uma seção específica para isso.

Os Logs acabam sendo importante para a aplicação pois nele irá conter todos os registros das atividades que podem nos ajudar quando ocorrer algum problema com a aplicação.

A ISO 27001 é um padrão criado especificamente para cuidar de segurança da informação. Nele há uma série de boas práticas que, ao serem colocadas em prática, faz com que a sua organização tenha sistemas mais seguros e com boas políticas de auditório.

Caso você queira saber mais sobre isso, verifique no site oficial:

http://www.iso.org/iso/home/standards/management-standards/iso27001.htm

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Podemos implementar o Log no acesso ao servidor (com paramiko), onde ficaria da seguinte maneira:

```
[ 30/03/2016 17:14 ] ERROR – Falhou ao acessar servidor
```

Podemos criá-lo seguindo exatamente esse formato:

```
[ DATA HORA ] NIVEL DO LOG –  EVENTO OCORRIDO
```

Em arquivos de Log é muito comum também guardar qual foi o usuário que executou determinado evento, mas para o nosso caso isso já será o suficiente, uma vez que somente nós vamos executar esse script.

Os formatos de logs podem ser customizados da maneira que você achar melhor e com os dados que você achar relevante.

Os logs também podem ser customizados. Dependendo da aplicação os arquivos de log também podem ser definidos.

No caso de sistema linux, temos alguns logs que são específicos da aplicação e todos os logs referentes ao kernel ficam em um único arquivo. Sendo ele:

/var/log/messages

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução a Logs

Por padrão as distribuições Linux guardam os **arquivos de Log** dentro do diretório:

```
/var/log/
```

Esse é um padrão estabelecido pela **FHS** (Filesystem Hierarchy Standard), que tem como objetivo padronizar a estrutura de diretórios do Linux. No nosso caso vamos criar uma pasta chamada **Logs** e nela vamos armazenar essas informações.

A FHS é um sistema de hierarquia de arquivos usado por muitos sistemas operacionais, principalmente o Linux.
Mas também temos outros como:

- Solaris
- FreeBSD
- Hurd
- Outros

Caso você queira saber mais sobre a FHS, pode-se acessar o seguinte link:
http://www.pathname.com/fhs/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Objetivos da Aula

- ✔ Conhecer o módulo Logging;
- ✔ Criar logs para aplicação.

Conhecendo Logs com Python

Nesta aula iremos conhecer o módulo Logging e entender como podemos implementar Logs na aplicação.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Instalação do módulo

Para gerar os arquivos de Log precisamos de um módulo do python específico pra isso chamado **Logging**.

Podemos fazer a instalação do módulo da seguinte forma:

```
pip3 install logging
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Criando Logs

Crie um arquivo chamado Log.py para ser responsável pelos logs, dentro dele coloque o seguinte conteúdo:

```
#!/usr/bin/python3
import logging

class Logs(object):
    def __init__(self):
        pass
```


O módulo logging atualmente é o mais utilizado para efetuar essa geração de Logs em seus scripts feitos em Python.

Para ver a documentação oficial e toda a configuração disponível, você pode acessar o seguinte link:

https://docs.python.org/2/library/logging.html

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

No contrutor da classe **Logs**, adicione as seguintes linhas:

```
logging.basicConfig(filename="logs/app.log",
                    level=logging.INFO,
                    format="%(asctime)s %(message)s",
                    datefmt="[ %d/%m/%Y - %H:%M:%S]")
```

Com essa instrução nós definimos:

Arquivo de Log que estará em logs/dexter.logs.

A **chave level** que foi passada como parâmetro, que diz a partir de qual nível de Log será registrado no arquivo.

O módulo logging atualmente é o mais utilizado para efetuar essa geração de Logs em seus scripts feitos em Python.

Para ver a documentação oficial e toda a configuração disponível, você pode acessar o seguinte link:

https://docs.python.org/2/library/logging.html

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Os níveis de Logs possíveis são os seguintes:

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG
- NOTSET

No slide anterior, como foi setado o **Level INFO**, irão aparecer mensagens deste nível para cima, ou seja, de info até critical.

Quando se trabalha com mensagens de log, cada uma delas tem um nível de criticidade dentro do log e você pode definir a partir de qual nível as mensagens irão ser registradas. Abaixo temos elas listadas do nível mais alto para o mais baixo:

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG
- NOTSET

Se você definiu o level do seu log como INFO, irão aparecer respectivamente as mensagens de INFO, WARNING, ERROR e CRITICAL.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

A chave **format** indica o formato do Log que será a data e hora no formato **ascii**. Poderia ser, por exemplo, em Unixtimestamp, mas o **ascii** é o mais legível.
A ordem em que a data e hora irão aparecer foi definido em **datefmt**. Suas variaveis significam:

- %d - Dia
- %M - Minuto
- %H - Hora
- %Y - Ano
- %m - Mês
- %S - Segundo

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Quando você quiser gerar um Log, será necessário executar a seguinte instrução:

```
logging.NIVEL_DO_LOG(“Mensagem”)
```

Pensando nisso, podemos criar um método para gerar o log da seguinte maneira:

```
def log_info(self):
    logging.info("Gerando log da aplicacao")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Conhecendo Logs com Python

Veja dentro da pasta logs que foi criado o arquivo app.log. Abra e veja o conteúdo, que ficou da seguinte forma:

```
[ 11/10/2017 – 11:50:54 ] Gerando log da aplicacao
```

Para ter apenas um log de erro na aplicação, podemos alterar o **logging.basicConfig** e alterar o parâmetro level para **logging.ERROR**, desta forma as informações não irão mais aparecer, apenas os eventos de **ERROR** ou **CRITICAL**.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Objetivos da Aula

- ✔ Criar uma pequena aplicação;
- ✔ Implementar logs.

Implementando Logs na aplicação

Nesta aula iremos criar uma aplicação e implementar logs para obter informações e erros.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Criando aplicação

Vamos criar um script para conectar no servidor Docker e ver a memória disponível, para isto utilizaremos o módulo paramiko. Crie um arquivo chamado conexao.py com o seguinte conteúdo:

```
from paramiko.client import SSHClient
import paramiko
class Conexao(object):
    def __init__(self):
        pass
    def comando(self):
        pass
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Dentro do método construtor podemos colocar a conexão, utilizando um try/except para tratativa de erros:

```
try:
   self.client = SSHClient()
   self.client.load_system_host_keys()
   self.client.set_missing_host_key_policy
              (paramiko.AutoAddPolicy())
   self.client.connect('192.168.0.2',username='root',
                       password='4linux')
   print("Conexão feita com sucesso!")
except Exception as e:
   print("Conexão falhou!")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Dentro do método comando podemos colocar o seguinte código:

```
try:
  stdin,stdout,stderr = self.client.exec_command('free -m')
  if stderr.channel.recv_exit_status() != 0:
     print (stderr.channel.recv_exit_status())
     print (stderr.read().decode('utf-8'))
  else:
     print (stdout.read().decode('utf-8'))
  print("Comando executado com sucesso!")
except Exception as e:
  print("Comando falhou!")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Neste momento temos uma conexão SSH, que irá acessar o servidor Docker e executar o comando “free -m”.

Para obtermos o resultado, podemos chamar a classe e o método comando da seguinte forma:

```
if __name__ == '__main__':
    ssh = Conexao()
    ssh.comando()
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Criando Logs

Agora podemos implementar os logs em nosso script, para isto, devemos fazer o import do módulo logging:

```
import logging
```

Após fazermos o import deste módulo, devemos realizar a sua configuração e definir quais tipos de logs queremos em nossa aplicação, neste caso iremos usar INFO.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Criando Logs

Agora podemos implementar os logs em nosso script, para isto, devemos fazer o import do módulo logging:

```
import logging
```

Após fazermos o import deste módulo, devemos realizar a sua configuração e definir quais tipos de logs queremos em nossa aplicação, neste caso iremos usar INFO.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Configurando log

Dentro do método construtor (acima do try), vamos configurar o log:

```
def __init__(self):
  logging.basicConfig(filename="app.log",
                      level=logging.INFO,
      format="[%(asctime)s] %(levelname)s - %(message)s",
                      datefmt="%d/%m/%Y - %H:%M:%S")
  try:
     .........
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Agora podemos substituir todos os prints da aplicação por "logging.LEVEL".

Dentro do try colocaremos:

```
logging.info("...")
```

Dentro do except será:

```
logging.error("...")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Implementando logs na aplicação

Desta forma irá aparecer todas as informações de conexão dentro do log. Podemos também deixar que o log grave apenas as mensagens de erro, desta forma, devemos alterar o **logging.basicConfig**:

```
level=logging.ERROR
```

Agora, quando executamos o script podemos ver os logs sendo criados dentro de app.log

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Objetivos da Aula

- ✔ Criar aplicação em Flask;
- ✔ Renderizar HTML.

Criando Dashboard

Nesta aula iremos criar uma aplicação em Flask e entender como ele renderiza o HTML.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Aplicação em Flask

Esse é o corpo inicial de uma aplicação em **Flask**

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def index():
    return "Flask response"

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


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Renderizando HTML

Para renderizar um html, é necessário criar uma pasta chamada **templates** no mesmo diretório onde criamos o Dashboard.py.
Agora teremos que alterar duas linhas:

```
from flask import Flask,render_template
```

O **return** ficará assim:

```
return render_template("index.html")
```

O **index.html** deve ser cirado dentro da pasta chamado **templates** e deve ter qualquer conteúdo

Toda a parte do **render_template** e como chamar os arquivos html através do Flask estão documentadas na página oficial do projeto:

http://flask.pocoo.org/docs/0.10/quickstart/

O Flask utiliza uma linguagem de template chamada **Jinja2** que pode auxiliar e muito durante a criação das suas páginas. Toda a documentação dessa linguagem pode ser encontrada no site oficial do projeto:

http://jinja.pocoo.org/docs/dev/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

O conteúdo do arquivo **html** pode ficar como abaixo:

```
<html>

<body>

     <p>HTML da Aplicação em Flask</p>

</body>

</html>
```

Agora execute a aplicação e veja que já está funcionando!

É recomendado que se saiba pelo menos o básico de HTML para fazer essa aula, já que o nosso objetivo não é ensinar HTML e sim mostrar de forma breve como criar uma interface Web para você poder deixar os seus sistemas mais amigáveis. Caso você não saiba HTML, existe a documentação da W3C que pode ser consultada em caso de dúvidas:

http://www.w3schools.com/html/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Passando variáveis

Vamos modificar a nossa aplicação e passar variáveis. Altere a sua rota da seguinte forma:

```
def index():
    animais = [{'nome':'Papagaio', 'especie':'Ave'},
               {'nome':'Arara', 'sobrenome':'Ave'},
               {'nome':'Leao', 'sobrenome':'Mamifero'}]

    return render_template('index.html',
                           animais=animais)
```

Note que estamos enviando para o HTML a variável “animais” que é uma lista.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Agora que modificamos a rota, podemos começar a alterar o index.html. Vamos criar uma tabela com os dados de animais:

```
<div class='container'>
<table class="table table-bordered table-striped">
  <thead class="thead-inverse">
    <tr>
      <th>Nome</th>
      <th>Sobrenome</th>
    </tr>
  </thead>
… … …
```

O código acima é referente as colunas.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Para podemos criar os registros, faremos da seguinte forma:

```
<tbody>
    {% for animal in animais %}
    <tr>
      <td>{{ animal["nome"] }}</td>
      <td>{{ animal["especie"] }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
</div>
```

Acima vemos como o jinja2 utiliza as variáveis que recebe do Flask.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando Dashboard

Para ter interfaces mais ricas, vamos utilizar um framework CSS chamado **Bootstrap**, que atualmente é um dos mais utilizados no mundo e muito fácil de ser utilizado.

Nesta primeira aplicação podemos utilizar um recurso chamado MaxCDN, que basicamente apenas iremos colocar os links de JavaScript e CSS no final da aplicação.

Podemos obter esses links através do site:

https://getbootstrap.com/docs/3.3/getting-started/

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Objetivos da Aula

- ✔ Entender o projeto;
- ✔ Criar a sua estrutura.

Entendendo o Projeto

Nesta aula entender o projeto final e criar a sua estrutura.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Entendendo o sistema

O DexterOps é um sistema focado em realizar a integração das ferramentas utilizadas pela Dexter Courier. O sistema deve conter estes módulos:

- ✔ Servidores
- ✔ Jenkins
- ✔ Gitlab
- ✔ Docker
- ✔ AWS

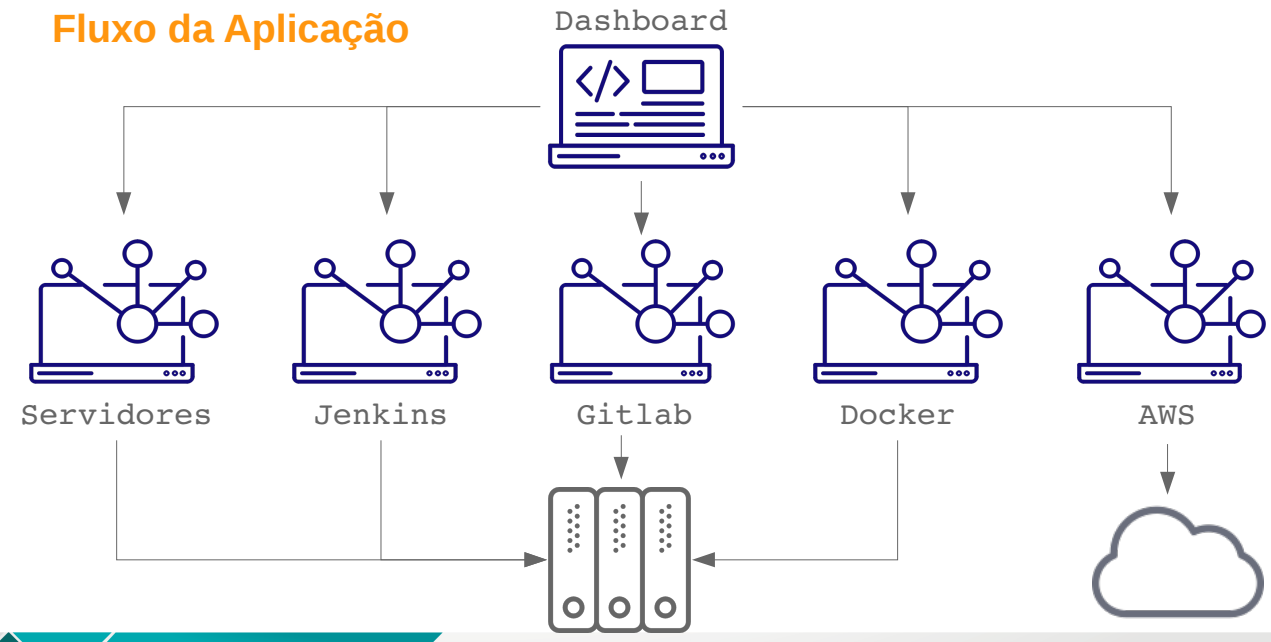
Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Fluxo da Aplicação



Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Funcionalidades

Ao finalizar o projeto poderemos gerenciar todas as ferramentas por um único painel, dentro dela teremos as funcionalidades:

GitLab
Listar, criar e remover usuários e repositórios

Jenkins
Listar, criar, remover e iniciar Jobs

Servidores
Informações (load, disco e memória)

Docker
Listar, criar, remover, iniciar e parar container

AWS
Criar e excluir Máquinas e SG

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Criando a estrutura

Durante as aulas criamos arquivos de integração com o Gitlab, Docker, Jenkins e AWS. Vamos aproveitar estes arquivos e fazer os ajustes necessários para que o nosso BackEnd seja feito.

Para os nossos módulos podemos criar um diretório chamado **"Modules"** e criar os arquivos dentro dele.

Por enquanto devemos ter estes quatro modulos dentro do diretório.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Entendendo o Projeto

Exemplo da estrutura

Dentro do diretório principal do projeto, teremos as seguintes divisões:

- /modules: será responsável pelos módulos utilizados
- /static: será responsável por armazenar os arquivos de css e javascript
- /templates: será responsável pelas páginas em html

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

Objetivos da Aula

- ✔ Entender a estrutura de dashboard;
- ✔ Organizar a estrutura do projeto.

Criando Dashboard do projeto

Nesta aula iremos entender a estrutura do Dashboard e organiza-lo em nosso projeto.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

Bootstrap

Neste projeto para ter uma interface mais rica, estamos utilizando o Bootstrap. Para fizemos o download do JavaScript e CSS através do seguinte link:

```
http://getbootstrap.com.br/getting-started/
```

Após acessar o site, basta clicar em “Download Bootstrap”. O mesmo conteúdo do arquivo baixado pode ser encontrado dentro do diretório “static”.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

Bootstrap

Neste projeto iremos utilizar o Bootstrap e para isto podemos fazer o download do JavaScript e CSS através do seguinte link:

```
http://getbootstrap.com.br/getting-started/
```

Após acessar o site, basta clicar em “Download Bootstrap” e o conteúdo do arquivo baixado deverá ficar dentro do diretório “static”.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

Organizando o projeto

Agora que já sabemos de onde vem o JavaScript e o CSS, os arquivos em HTML ficam dentro de um diretório chamado “templates”. Temos este diretório pois é onde o Flask irá buscar tais arquivos.

Dentro deste, o arquivo “base.html” que será responsável pelo nosso menu e os outros são referentes as páginas específicas, como por exemplo: docker.html, aws.html entre outros.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

URLs

As seguintes URLs foram colocadas no Dashboard, então devemos criar as mesmas no Flask:

/aws/ - gerencia do painel AWS

/aws/new_sg/ - cria um novo Security Group

/aws/new_ec2/ - cria um novo EC2

/gitlab/ - gerencia do gitlab

/gitlab/new_user/ - cria novos usuários

/gitlab/new_project/ - cria novos projetos

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

URLs

As URLs para o docker e para a gerencia de servidores:

/container/ - gerencia dos containers
/container/stop/ - para um container
/container/start/ - inicia um container
/container/remove/ - remove um container
/container/new/ - cria novos containers
/servidores/ - mostra informações sobre servidores

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

URLs

As URLs para o docker e para a gerencia de servidores:

/container/ - gerencia dos containers
/container/stop/ - para um container
/container/start/ - inicia um container
/container/remove/ - remove um container
/container/new/ - cria novos containers
/servidores/ - mostra informações sobre servidores

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando o Dashboard do projeto

Criação do HTML

Caso você queira ver passo a passo da criação do HTML para esta aplicação, pode acompanhar nas anotações o código de cada arquivo criado.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com


```
<html>
<head>
<title>Gerencia da DexterOps</title>
</head>
<body>
   <nav class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
         <div class="navbar-header">
            <a class="navbar-brand" href="/index/">Inicio</a>
         </div>
         <div id="navbar" class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
               <li class="{% if request.path == '/container/' %} active {% endif %}">
                  <a href="#" class="dropdown-toogle" data-toggle="dropdown"
                    role="button" aria-haspopup="true" area-extended="false">Docker
                  <span class="caret"></span></a>
                  <ul class="dropdown-menu">
                     <li><a href="/container/">Gerencia do Docker</a></li>
                     <li><a href="/container/new/">Novo container</a></li>
                  </ul>
               </li>
            </ul>
            <ul class="nav navbar-nav">
               <li class="{% if request.path == '/gitlab/' %} active {% endif %}">
                  <a href="#" class="dropdown-toogle" data-toggle="dropdown"
                    role="button" aria-haspopup="true" area-extended="false">GitLab
                  <span class="caret"></span></a>
                  <ul class="dropdown-menu">
                     <li><a href="/gitlab/">Gerencia do GitLab</a></li>
                     <li><a href="/gitlab/new_user/">Novo Usuário</a></li>
                     <li><a href="/gitlab/new_project/">Novo Projeto</a></li>
                  </ul>
               </li>
            </ul>
            <ul class="nav navbar-nav">
                <li>
                  <a href="/servidores/" role="button" aria-haspopup="true" area-extended="false">Servidores</a>
                </li>
            </ul>
            <ul class="nav navbar-nav">
               <li class="{% if request.path == '/aws/' %} active {% endif %}">
                  <a href="#" class="dropdown-toogle" data-toggle="dropdown"
                    role="button" aria-haspopup="true" area-extended="false">AWS
                  <span class="caret"></span></a>
                  <ul class="dropdown-menu">
                     <li><a href="/aws/">Gerencia da AWS</a></li>
                     <li><a href="/aws/new_ec2/">Novo EC2</a></li>
                     <li><a href="/aws/new_sg/">Novo SG</a></li>
                  </ul>
               </li>
            </ul>
         </div>
      </div>
   </nav>
   {% block content %}
   {% endblock %}
   <link rel="stylesheet" href="/static/css/bootstrap.min.css" />
   <script language="javascript" src="/static/js/docker.js"></script>
   {% block jscode %}
   {% endblock %}

</body>
</html>
```

Felipe Moz / moz.felipe@gmail.com

```
{% extends "base.html" %}

{% block content %}


<br>
<br>
<div id="exTab2" class="container">
    <ul class="nav nav-tabs">
        <li class="active">
            <a href="#1" data-toggle="tab">EC2</a>
        </li>
        <li><a href="#2" data-toggle="tab">SECUTIRY GROUP</a>
        </li>
    </ul>

    <div class="tab-content ">
        <div class="tab-pane active" id="1">
            <div class="panel-body">

                <table class="table">
                    <th>ID</th>
                    <th>Tipo</th>
                    <th>Security Group</th>
                    {% for instance in instances %}
                    <tr>
                        <td>{{instance.id}}</td>
                        <td>{{instance.instance_type}}</td>
                        <td>{{instance.security_groups[0]['GroupName']}}</td>
                    </tr>
                    {% endfor %}
                </table>
            </div>
        </div>
        <div class="tab-pane" id="2">

            <div class="panel-body">
                <table class="table">
                    <th>Nome</th>
                    <th>ID</th>

                    {% for sg in security_groups %}
                    <tr>
                        <td>{{sg.group_name}}</td>
                        <td>{{sg.group_id}}</td>
                    </tr>
                    {% endfor %}
                </table>
            </div>
        </div>
    </div>
</div>


{% endblock %}
{% block jscode %}
<link rel="stylesheet" href="/static/css/bootstrap.min.css"/>
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com

```
{% extends "base.html" %}

{% block content %}
{% if message %}
<br>
<br>
<div class="alert alert-info" role="alert">
  {{message}}
</div>
{% endif %}
<br>
<br>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading"><b>Create EC2</b></div>
            <div class="panel-body">
                <form class="form-horizontal" method="post" action="/aws/new_ec2/">
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Imagem</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="imagem" placeholder="Digite o nome do EC2">
                      </div>
                    </div>
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Security Group</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="sg" placeholder="Digite o Security Group">
                      </div>
                    </div>


                    <div class="form-group">
                      <div class="col-sm-offset-2 col-sm-10">
                        <button type="submit" class="btn btn-success">Criar</button>
                      </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>

<link rel="stylesheet" href="/static/css/bootstrap.min.css" />
<script language="javascript" src="/static/js/jquery.min.js"></script>
<script language="javascript" src="/static/js/docker.js"></script>
</body>
</html>
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com

```
{% extends "base.html" %}

{% block content %}
{% if message %}
<br>
<br>
<div class="alert alert-info" role="alert">
  {{message}}
</div>
{% endif %}
<br>
<br>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading"><b>Create Security Group</b></div>
            <div class="panel-body">
                <form class="form-horizontal" method="post" action="/aws/new_sg/">
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Nome</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="nome" placeholder="Digite o nome do SG">
                      </div>
                    </div>
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Descrição</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="descricao" placeholder="Digite a descrição">
                      </div>
                    </div>


                    <div class="form-group">
                      <div class="col-sm-offset-2 col-sm-10">
                        <button type="submit" class="btn btn-success">Criar</button>
                      </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>

<link rel="stylesheet" href="/static/css/bootstrap.min.css" />
<script language="javascript" src="/static/js/jquery.min.js"></script>
<script language="javascript" src="/static/js/docker.js"></script>
</body>
</html>
{% endblock %}
```

```
{% extends "base.html" %}

{% block content %}


<br>
<br>
   <div class="container">
      <div class="panel panel-default">
         <div class="panel-heading"><b>Running Containers</b></div>
         <div class="panel-body">
            <table class="table">
            <th>Nome</th>
            <th>Endereço</th>
            <th>Status</th>
            <th>Ações</th>
            {% for c in containers %}
            <tr>
               <td>{{c.Names[0]}}</td>
               <td>{{c.NetworkSettings.Networks.bridge.IPAddress}}</td>
               <td>{{c.Status}}</td>
               <td>
                  <a href="" id="{{c.Id}}" class="start-container">
                  <span class="glyphicon glyphicon-play"></span>
                  </a>
                  <a href="" id="{{c.Id}}" class="stop-container">
                  <span class="glyphicon glyphicon-stop"></span>
                  </a>
                   <a href="" id="{{c.Id}}" class="remove-container">
                  <span class="glyphicon glyphicon-remove"></span>
                  </a>
               </td>
            </tr>
            {% endfor %}
            </table>
         </div>
      </div>
   </div>

{% endblock %}
{% block jscode %}
<link rel="stylesheet" href="/static/css/bootstrap.min.css" />
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com

Felipe Moz / moz.felipe@gmail.com


```
{% extends "base.html" %}

{% block content %}
{% if message %}
<br>
<br>
<div class="alert alert-info" role="alert">
  {{message}}
</div>
{% endif %}
<br>
<br>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading"><b>Create Containers</b></div>
            <div class="panel-body">
                <form class="form-horizontal" method="post" action="/container/new/">
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Nome</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="container-name" placeholder="Digite o nome do
container">
                      </div>
                    </div>
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Imagem</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="container-image" placeholder="Digite a imagem.
Ex: debian:latest">
                      </div>
                    </div>
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Comando</label>
                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="container-command" placeholder="Digite o
comando. Ex: /bin/bash">
                      </div>
                    </div>


                    <div class="form-group">
                      <div class="col-sm-offset-2 col-sm-10">
                        <button type="submit" class="btn btn-success">Criar</button>
                      </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>

<link rel="stylesheet" href="/static/css/bootstrap.min.css" />
<script language="javascript" src="/static/js/jquery.min.js"></script>
<script language="javascript" src="/static/js/docker.js"></script>
</body>
</html>
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com


```
{% extends "base.html" %}

{% block content %}


<br>
<br>

<div id="exTab2" class="container">
   <ul class="nav nav-tabs">
      <li class="active">
         <a href="#1" data-toggle="tab">Projetos</a>
      </li>
      <li><a href="#2" data-toggle="tab">Usuários</a>
      </li>
   </ul>

   <div class="tab-content ">
      <div class="tab-pane active" id="1">
         <div class="panel-body">

            <table class="table">
               <th>Nome</th>
               <th>Data de criação</th>
               <th>Último commit</th>
               {% for p in projects %}
               <tr>
                  <td>{{p.name}}</td>
                  <td>{{p.created_at.strftime('%d/%m/%Y %H:%M')}}</td>
                  <td>{{p.last_activity_at.strftime('%d/%m/%Y %H:%M')}}</td>
               </tr>
               {% endfor %}
            </table>
         </div>
      </div>
      <div class="tab-pane" id="2">

         <div class="panel-body">
            <table class="table">
               <th>Nome</th>
               <th>Username</th>
               <th>E-mail</th>
               {% for u in users %}
               <tr>
                  <td>{{u.name}}</td>
                  <td>{{u.username}}</td>
                  <td>{{u.email}}</td>
               </tr>
               {% endfor %}
            </table>
         </div>
      </div>
   </div>
</div>


{% endblock %}
{% block jscode %}
<link rel="stylesheet" href="/static/css/bootstrap.min.css"/>
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com


```
{% extends "base.html" %}

{% block content %}
{% if message %}
<br>
<br>
<div class="alert alert-info" role="alert">
  {{message}}
</div>
{% endif %}
<br>
<br>

<div class="container">
    <div class="panel panel-default">
        <div class="panel-heading"><b>Create Project</b></div>
            <div class="panel-body">
                <form class="form-horizontal" method="post" action="/gitlab/new_project/">
                    <div class="form-group">
                      <label class="col-sm-2 control-label">Nome</label>

                      <div class="col-sm-10">
                        <input type="text" class="form-control" name="project-name" placeholder="Digite o nome do
projeto">
                      </div>
                    </div>
                    <div class="form-group">
                      <div class="col-sm-offset-2 col-sm-10">
                        <button type="submit" class="btn btn-success">Criar</button>
                      </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>

<link rel="stylesheet" href="/static/css/bootstrap.min.css" />
<script language="javascript" src="/static/js/jquery.min.js"></script>
<script language="javascript" src="/static/js/docker.js"></script>
</body>
</html>
{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com

INDEX.HTML

```
{% extends "base.html" %}

{% block content %}
<br>
<br>

<center><img src="/static/imgs/dexter.png"/>
   <br>
   <br>

<b><h4>Bem-vindo ao Painel de Gerenciamento da Dexter Courier</h4></b>

   <script src="/static/js/jquery.min.js"></script>
   <script src="/static/js/bootstrap.min.js"></script>
   <link rel="stylesheet" href="/static/css/bootstrap.min.css" />
   <script language="javascript" src="/static/js/docker.js"></script>
   {% block jscode %}
   {% endblock %}
</center>


{% endblock %}
```

SERVIDORES.HTML

```
{% extends "base.html" %}

{% block content %}
{% if message %}
<br>
<br>
<div class="alert alert-info" role="alert">
  {{message}}
</div>
{% endif %}
<br>
<br>
<div class="container">
   <div>
      <h2>Status dos servidores</h2>
      <p></p>
   </div>
   <div>
     <h4>Load / Uptime</h4>
   </div>
   <table class="table table-bordered">
      <thead>
         <tr>
            <th class="danger">Gitlab</th>
            <th class="info">Docker</th>
            <th class="success">Jenkins</th>
          </tr>
      </thead>
      <tbody>
         <tr>
         {% for l in load %}
            <td>{{ l }}</td>
         {% endfor %}
         </tr>
      </tbody>
```

Felipe Moz / moz.felipe@gmail.com


```
  </table>
    <div>
      <h4>Memória</h4>
    </div>
    <table class="table table-bordered">
        <thead>
            <tr>
                <th class="danger">Gitlab</th>
                <th class="info">Docker</th>
                <th class="success">Jenkins</th>
             </tr>
        </thead>
        <tbody>
            <tr>
            {% for m in memoria %}
                <td>{{ m }}</td>
            {% endfor %}
            </tr>
        </tbody>
    </table>
    <div>
      <h4>Disco</h4>
    </div>
    <table class="table table-bordered">
        <thead>
            <tr>
                <th class="danger">Gitlab</th>
                <th class="info">Docker</th>
                <th class="success">Jenkins</th>
             </tr>
        </thead>
        <tbody>
            <tr>
            {% for d in disco %}
                <td>{{ d }}</td>
            {% endfor %}
            </tr>
        </tbody>
    </table>
</div>

{% endblock %}
```

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

Objetivos da Aula

- ✔ Criar o módulo para o Docker;
- ✔ Criar as rotas para o Docker.

Criando integração com Docker

Nesta aula iremos criar o módulo e a blueprint para fazer a integração com o Docker.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

DockerModule

Vamos criar um arquivo chamado DockerModule responsável pela integração com a API do Docker. Podemos começar criando a sua conexão:

```
from docker import Client
class DockerModule:
  def __init__(self):
    try:
      self.cli = Client("tcp://192.168.0.2:2376", version="auto")
    except Exception as e:
      print ('Falhou ao conectar no docker: ', e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

Após a conexão, podemos criar os métodos para listar, iniciar e parar:

```
def list_containers(self):
        containers = self.client.containers(all=True)
        return containers

    def start_container(self, id):
        res = self.client.start(container=id)
        return res

    def stop_container(self, id):
        res = self.client.stop(container=id)
        return res
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

O método para deletar container:

```
def delete_container(self, container):
        self.stop_container(container)
        self.client.remove_container(container=container)
        return "Container removido com sucesso"
```

Lembrando que para excluirmos um container ele deve estar parado, desta forma temos que garantir que ele atenderá este requisito.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

O método para criar container ficará da seguinte forma:

```
def create_container(self, **kwargs):
  self.client.pull(kwargs.get("image"))
  res = self.client.create_container(name=kwargs.get("name"),
                                     image=kwargs.get("image"),
                                     command=kwargs.get("command"),
                                     stdin_open=True,
                                     detach=True,
                                     tty=True)
        return res
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

Blueprint Docker

Agora que já criamos o módulo para o Docker, podemos criar as suas rotas. Dentro do diretório Blueprints, crie um arquivo chamado "docker.py". A primeira rota será responsável por listar os containers:

```
from flask import Blueprint, request, render_template, jsonify
from Modules.DockerModule import DockerModule
container = Blueprint('container', __name__)

@container.route("/container/")
def docker_ger():
    dm = DockerModule()
    containers = dm.list_containers()
    return render_template("docker.html", containers=containers)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

Podemos criar a rota de parar containers, no qual deve receber do HTML o ID e será liberado apenas o método POST neste caso:

```
@container.route("/container/stop/", methods=["POST"])
def stop():
    container_id = request.form["id"]
    dm = DockerModule()
    dm.stop_container(container_id)
    return jsonify({"message": "Container parado com sucesso!"})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

A rota para iniciar e para containers ficará bem parecida com a anterior:

```
@container.route("/container/start/", methods=["POST"])
def start():
    container_id = request.form["id"]
    dm = DockerModule()
    dm.start_container(container_id)
    return jsonify({"message": "Container iniciado com sucesso!"})

@container.route("/container/remove/", methods=["DELETE"])
def delete_container():
    container_id = request.form["id"]
    dm = DockerModule()
    res = dm.delete_container(container_id)
    return jsonify({"message": res})
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

A rota para criar novos containers:

```
@container.route("/container/new/", methods=["GET", "POST"])
def container_new():
    try:
        if request.method == "POST":
            name = request.form["container-name"]
            image = request.form["container-image"]
            command = request.form["container-command"]
            dm = DockerModule()
            dm.create_container(name=name,
                                image=image,
                                command=command)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Docker

O código abaixo deve ficar dentro do bloco de código do IF:

```
return render_template("docker_new.html",
                        message="Criado com sucesso")
        else:
            return render_template("docker_new.html")
    except Exception as e:
        return render_template("docker_new.html",
                               message="Erro: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Objetivos da Aula

- ✔ Criar o módulo para o Gitlab;
- ✔ Criar as rotas para o Gitlab.

295

4LINUX

Criando integração com Gitlab

Nesta aula iremos criar o módulo e a blueprint para fazer a integração com o Gitlab.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

GitlabModule

Assim como no Docker, vamos criar um GitlabModule responsável pela integração com a API do Gitlab. Podemos fazer os seguintes imports:

```
import requests
import json
from datetime import datetime

class GitlabModule(object):
   def __init__(self):
      pass
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Podemos criar o método para listar os projetos:

```
def list_project(self):
    self.token = "rsSMxQqJy2s1yVyFsmJe"
    self.recurso = "projects"
    self.url = "http://192.168.0.3/api/v4/%s/?private_token=%s"
                %(self.recurso, self.token)

    response = json.loads(requests.get(self.url)._content.
                                                decode('utf-8'))
    return response
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

O método para criar novos projetos ficará da seguinte forma:

```
def createprojects(self,**dados):
  try:
    self.token = "rsSMxQqJy2s1yVyFsmJe"
    self.recurso = "projects"
    self.url = "http://192.168.0.3/api/v4/%s?private_token=%s"
                %(self.recurso, self.token)
    res = json.loads(requests.post(self.url, data=dados)._content.
                                                decode('utf-8'))
    return res
  except Exception as e:
    return e
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

O método para criar novos usuários:

```
def createuser(self, **kwargs):
  try:
    self.token = "rsSMxQqJy2s1yVyFsmJe"
    self.recurso = "users"
    self.url = "http://192.168.0.3/api/v4/%s?private_token=%s"
                % (self.recurso, self.token)
    res = json.loads(requests.post(self.url, data=kwargs)._content.
                                                    decode('utf-8'))
    return res
  except Exception as e:
   return (e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Agora podemos criar o método para listar usuários:

```
def list_user(self):
  self.token = "rsSMxQqJy2s1yVyFsmJe"
  self.recurso = "users"
  self.url = "http://10.100.0.113/api/v4/%s/?private_token=%s" %
             (self.recurso, self.token)

  response = json.loads(requests.get(self.url)._content.decode())
  return response
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Blueprint Gitlab

Agora que já criamos o módulo para o Gitlab, podemos criar as suas rotas. Dentro do diretório Blueprints, crie um arquivo chamado "gitlab.py". Podemos fazer os seguintes imports:

```
from flask import Blueprint, render_template, request
import datetime

from Modules.GitlabModule import GitlabModule

gitlab = Blueprint('gitlab', __name__)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Podemos criar a rota para listar as informações de usuários e projetos:

```
@gitlab.route("/gitlab/")
def gitlab_ger():
  gm = GitlabModule()
  lista = gm.list_project()
  projects = [{'name': l.get('name'), 'created_at':
              l.get('created_at'),'last_activity_at':
              l.get('created_at')} for l in lista]
  users = gm.list_user()
  return render_template("gitlab.html", projects=projects,
                                        users=users)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

A rota para criar novos projetos:

```
@gitlab.route("/gitlab/new_project/", methods=["GET",
                                                "POST"])
def gitlab_newproject():
  try:
    if request.method == "POST":
      name = request.form['project-name']
      gm = GitlabModule()
      res = gm.createprojects(name=name)
      return render_template("gitlab_newproject.html",
                              message="Criado com sucesso")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

O código abaixo deve ficar dentro do bloco de código do Try:

```
try:
  .........
  else:
    return render_template("gitlab_newproject.html")
except Exception as e:
    return render_template("gitlab_newproject.html",
                           message="Erro: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Para criar um novo usuário, devemos ter nome, username, senha e e-mail, então temos que receber estas informações do HTML:

```
@gitlab.route("/gitlab/new_user/", methods=["GET", "POST"])
def gitlab_newuser():
    try:
        if request.method == "POST":
            name = request.form['name']
            username = request.form['username']
            password = request.form['password']
            email = request.form['email']
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Gitlab

Agora podemos chamar o método do GitlabModule e criar o usuário:

```
    gm = GitlabModule()
    res = gm.createprojects(name=name, username=username,
                            password=password, email=email)
    return render_template("gitlab_newuser.html",
                           message="Usuario criado com sucesso")
  else:
    return render_template("gitlab_newuser.html")
except Exception as e:
  return render_template("gitlab_newuser.html",
                         message="Erro: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

Objetivos da Aula

- ✔ Criar o módulo para conexão com os Servidores;
- ✔ Criar as rotas para as informações de Servidores.

Criando integração com Servidores

Nesta aula iremos criar o módulo e a blueprint para fazer a integração com os Servidores.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

SSHModule

Para criar a integração com os servidores, iremos utilizar o módulo Paramiko, podemos começar com os seguintes imports:

```
import paramiko
from paramiko import SSHClient


class SSHModule(object):
   def __init__(self):
      pass
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

Dentro do método construtor, podemos ter as opções para ver e aceitar o known hosts:

```
def __init__(self):
    try:
        self.client = SSHClient()
        self.client.load_system_host_keys()
        self.client.set_missing_host_key_policy(paramiko.
                                                AutoAddPolicy())
    except Exception as e:
        return "Erro: %s" %e
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

Agora podemos criar o método para verificar o loadavg das máquinas:

```
def load(self):
  try:
    dados = []
    self.maquinas = ['192.168.0.2','192.168.0.3',
                     '192.168.0.4']
    for maquina in self.maquinas:
      self.client.connect(maquina,
                          username='root',
                          password='4linux')
      stdin,stdout,stderr = self.client.exec_command('uptime')
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

O código abaixo deve ficar dentro do bloco de código do for:

```
  if stderr.channel.recv_exit_status() != 0:
      raise Exception(stderr.channel.recv_exit_status())
      dados.append(stderr.read().decode('utf-8'))
  else:
      dados.append(stdout.read().decode('utf-8'))
    return dados
except Exception as e:
    return e
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

Agora podemos criar outros dois métodos, chamados: memoria e disco, onde devemos alterar apenas a seguinte linha:

```
def memoria(self):
      .........
      stdin,stdout,stderr = self.client.exec_command('free -m')
      .........

def disco(self):
      .........
      stdin,stdout,stderr = self.client.exec_command('df -h /')
      .........
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

Blueprint para SSH

A Blueprint para SSH deve ter apenas uma rota:

```
from Modules.SSHModule import SSHModule
from flask import Blueprint, render_template

servidores = Blueprint('servidores', __name__)

@servidores.route("/servidores/")
def ssh_load():
    try:
        sm = SSHModule()
        load = sm.load()
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com Servidores

O código abaixo deverá ficar dentro do bloco de código do try:

```
    memoria = sm.memoria()
    disco = sm.disco()
    return render_template("servidores.html",
                           load=load,
                           memoria=memoria,
                           disco=disco)
except Exception as e:
    return render_template("servidores.html",
           message="Falhou ao gerar informações: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

Objetivos da Aula

- ✔ Criar o módulo para conexão com os AWS;
- ✔ Criar as rotas para integração com AWS.

Criando integração com AWS

Nesta aula iremos criar o módulo e a blueprint para fazer a integração com o painel da AWS.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

AWSModule

Para criar a integração com o painel da AWS, iremos utilizar o módulo boto3, podemos começar com os seguintes imports:

```
import boto3

class AWSModule(object):

    def __init__(self):
        self.ec2 = boto3.resource("ec2")
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

Para listar as instâncias EC2 e os Security Groups podemos criar um método para cada um, respectivamente:

```
def list_instances(self):
    return self.ec2.instances.all()

def list_security_groups(self):
    return self.ec2.security_groups.all()
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

Para criar uma nova instancia do tipo EC2, podemos utilizar o seguinte código:

```
def create_instance(self,image,sg=None):
        try:
            self.ec2.create_instances(ImageId=image,
                                      InstanceType="t2.micro",
                                      MinCount=1,
                                      MaxCount=1,
                                      SecurityGroupIds=sg)
            return "EC2 criado com sucesso!"
        except Exception as e:
            return e
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

E para criar um novo Security Group vamos utilizar o seguinte método:

```
def create_security_group(self, nome, descricao):
   try:
     sg = self.ec2.create_security_group(GroupName=nome,
                                         Description=descricao)
     return "Grupo criado com sucesso!"
   except Exception as e:
     return e
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

Blueprint para AWS

Para criar a Blueprint para AWS, podemos começar com o seguintes imports:

```
from flask import Blueprint, render_template, request
from Modules.AWSModule import AWSModule

aws = Blueprint('aws', __name__)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

A rota para listar informações sobre EC2 e Security Groups:

```
@aws.route("/aws/")
def aws_ger():
    am = AWSModule()
    instances = am.list_instances()
    security_groups = am.list_security_groups()
    return render_template('aws.html',
                           instances=instances,
                   security_groups=security_groups)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

A rota para criar novas instâncias EC2:

```
@aws.route('/aws/new_ec2/', methods=['GET','POST'])
def create_ec2():
    try:
        if request.method == 'POST':
            imagem = request.form['imagem']
            sg = request.form['sg']
            am = AWSModule()
            am.create_instance(imagem, sg)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

O código abaixo deverá ficar dentro do bloco de código do try:

```
        return render_template("aws_newec2.html",
                    message="EC2 criado com sucesso")
    else:
        return render_template("aws_newec2.html")
except Exception as e:
    return render_template("aws_newec2.html",
                           message="Erro: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

Para criarmos os Security Groups, podemos utilizar:

```
@aws.route('/aws/new_sg/', methods=['GET','POST'])
def create_sg():
    try:
        if request.method == 'POST':
            nome = request.form['nome']
            descricao = request.form['descricao']
            am = AWSModule()
            am.create_security_group(nome, descricao)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Criando integração com AWS

O código abaixo deverá ficar dentro do bloco de código do try:

```
        return render_template("aws_newsg.html",
                message="SG criado com sucesso")
    else:
        return render_template("aws_newsg.html")
except Exception as e:
    return render_template("aws_newsg.html",
                           message="Erro: %s" % e)
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Objetivos da Aula

- ✔ Entender o que é Ansible;
- ✔ Entender a configuração;
- ✔ Testar a conectividade.

Introdução ao Ansible

Nesta aula iremos conhecer o Ansible, entender os seus arquivos de configuração e testar a conectividade entre dois hosts.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Ansible é uma ferramenta para automatizar o provisionamento de serviços cloud, gerenciamento de configuração, Deploys de aplicações e orquestração de serviços.

O Ansible foi desenvolvido pensando na simplicidade e facilidade de uso. Este é um dos fatores que está levando os Administradores de Sistema a escolher o Ansible para gerenciar o parque de máquinas. Outro foco do Ansible é a segurança e confiabilidade, utilizando o OpenSSH para transporte e comunicação dos nodes.

A simplicidade ajuda em ambientes de qualquer tamanho, sejam eles ambientes de empresas pequenas ou ambientes de empresas multinacionais, se tornando fácil o gerenciamento das máquinas e tornando possível a qualquer funcionário de TI ler as tarefas do Ansible de forma simples.

O Ansible gere as máquinas de uma forma sem agente (agente-less) pelo openssh. O OpenSSH está presente em todas as distribuições linux. O Ansible é descentralizado e depende somente das credenciais OS existentes para controlar o ambiente. Também podemos utilizar Kerberos ou LDAP para gerenciamento de autenticações centralizadas.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Características do Ansible

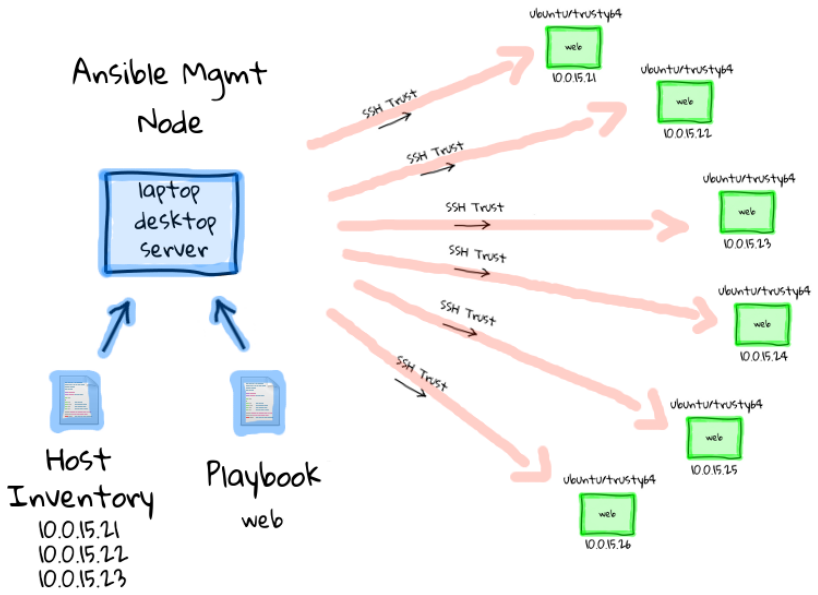
- Arquitetura Máster to nodes.
- Usa uma linguagem de configuração simples usando o formato YAML.
- Não necessita do uso de agentes instalados pois utiliza o SSH.
- Não necessita do uso de configuração de segurança customizada.
- Simples de realizar manutenção.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible



Ansible se conecta no nodes especificados e envia pequenos programas, que são chamados de Módulos do Ansible. Esse programas são escritos para controlar os recursos e estados do sistema. Ansible executa os programas no servidor utilizando SSH por padrão e remove o módulo enviado após o termino da execução.

A bibliotecas de módulos pode ser armazenada em qualquer máquina, e não precisa de nenhum servidor, damons ou banco de dados.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Inventário do Ansible

O arquivo de **Inventário (Inventory) do Ansible** armazena a lista de servidores gerenciados e acessados por ele. Por padrão, o arquivo de configuração é o **/etc/ansible/hosts**. Caso haja necessidade, podemos alterar este caminho no arquivo de configuração "/etc/ansible/ansible.cfg".

Dentro do arquivo de inventário podemos definir um host e também definir um grupo de hosts utilizando as seções separadas por "[]".

Exemplo de um arquivo de inventário.

```
mail.dexter.com.br

[webservers]
node01.dexter.com.br
node02.dexter.com.br   maxClient=709   user=torvalds

[dbservers]
db1.dexter.com.br   ansible_ssh_private_key_file=/etc/keys/db1.pem
db2.dexter.com.br   ansible_ssh_private_key_file=/etc/keys/db2.pem
```

No arquivo de inventário, podemos especificar uma máquina colocando o hostname ou endereço ip, ou podemos especificar um grupo utilizando as chaves. No exemplo acima, podemos escrever um playbook especificando os grupos ***webservers*** e todos os módulos especificando o que será executado nos dois servidores(node01 e node02).

Também podemos especificar variáveis pelo inventário e passar essas variáveis de acordo com node. No exemplo abaixo estamos passando a variável MaxClient com valor 709, e variável user com valor torvalds:

```
node02.dexter.com.br   maxClient=709   user=torvalds
```

Anotações:

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Alterando o inventário

Vamos alterar o arquivo de inventário e acrescentar na última linha a máquina do docker, como abaixo:

```
# vim /etc/ansible/hosts

[docker]
192.168.0.2 ansible_user=root
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Introdução ao Ansible

Testando conectividade

Para podermos testar a conectividade, podemos executar o seguinte comando:

```
# ansible all -m command -a 'ls /'
```

Neste caso a chave de conexão SSH para a conexão já foi configurada.

Para configurarmos a conexão entre os hosts, podemos seguir os passos abaixo:

1 – Criar a chave para conexão:

```
ssh-keygen

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id_rsa): /etc/keys/key.pem

Enter passphrase (empty for no passphrase): <enter>

Enter same passphrase again: <enter>
```

```
2 – Copiar a chave pública para o host:

ssh-copy-id -i /etc/keys/key.pem.pub root@192.168.0.2
```

```
3 – Editar o arquivo de configuração:

# vim /etc/ansible/ansible.cofg

private_key_file = /etc/keys/key.pem
```

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Objetivos da Aula

- ✔ Conhecer Módulos do Ansible;
- ✔ Criar Playbook.

Módulos do Ansible

Nesta aula iremos conhecer alguns módulos do Ansible e criar nossa primeira Playbook.

Anotações:

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Módulos do Ansible

Módulos, também conhecidos por "plugins de tarefas", é o que de fato é executado em cada tarefa do nosso playbook.

Veja ao lado alguns módulos core do Ansible:

- yum
- command
- shell
- copy
- apt
- service
- template
- git

Os módulos do ansible são onde a mágica acontence, eles são responsável por executar as tarefas designadas a eles nos nodes gerenciados pelo ansible. Por padrão o módulos estão localizados no diretório */urs/share/ansible*.

Podemos desenvolver os nosso próprios módulos em qualquer linguagem de programação, sendo que o ansible utiliza o JSON para se comunicar com módulos.

Lista de Módulos do Ansible:

- **apt:** gerencia instalação de pacotes utilizando o APTITUDE (debian/ubuntu);
- **yum:** gerencia instalação de pacotes utilizando o YUM (RedHat/CentOS/Fedora);
- **package:** gerencia instalação de pacotes utilizando o APTITUDE/YUM (o pacote deve possuir o mesmo nome em ambas as distribuições);
- **command:** Executa um comando em node remoto;
- **shell:** Executa um shell script dentro da máquina, após realizar as transferências;
- **service:** Gerencia serviços em máquina remotas;
- **copy:** copia arquivo na máquina local para o node remoto;
- **git:** gerencia repositório do git;
- **unarchive:** Descompactar arquivos na máquina remota;
- **mount:** Gerencia os dispositivos montados na máquina;
- **template:** Gerencia templates no ansible;
- **ec2:** Gerencia instancias ec2 ambiente cloud da Amazon;
- **docker_container:** Gerencia containers de docker.

Link: *http://docs.ansible.com/ansible/list_of_cloud_modules.html*

Anotações:

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Playbooks do Ansible

São arquivos do Ansible em que você descreve as tarefas a serem executadas nos servidores.

Em um nível mais básico, os Playbooks podem ser usados para gerenciar as configurações e realizar deploys nas máquinas remotas.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Primeiro Playbook

Em nosso primeiro playbook, vamos automatizar a instalação do PIP. Podemos criar um arquivo com chamado “playbook.yml” com o seguinte conteúdo:

```
# vim playbook.yml
# Playbook de exemplo:

- name: Primeiro playbook
  hosts: all
  tasks:
     - name: Instalar o PIP
       apt: name=python-pip update_cache=yes state=present
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Executando o playbook

Para executar o playbook, podemos utilizar o comando “ansible-playbook”:

```
# ansible-playbook playbook.yml
```

Podemos utilizar o módulo command para validar se o módulo foi instalado:

```
# ansible all -m command -a 'pip --version'
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Variáveis

Em alguns casos precisamos que nosso playbook possua variáveis.

Estas variáveis podem ser definidas dentro do próprio playbook ou podemos passa-las durante a execução dos playbooks.

Para realizar o teste, podemos criar um arquivo chamado "usuarios.yml" como abaixo:

```
# vim usuarios.yml
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Este playbook fará a criação de usuários no servidor, então podemos utilizar o seguinte código:

```
---
# Playbook para criar usuario
- name: Criacao de usuarios nos servidores
  hosts: all
  vars:
    - username: sysadmin

  tasks:
    - name: Criando os usuarios nos servidores
      user: name={{ username }} shell=/bin/bash createhome=yes
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Módulos e Playbooks

Quando executarmos o playbook, ele irá criar automaticamente o usuário e usará a variável que foi definida dentro do arquivo.

Para passarmos uma variável através da linha de comando, podemos utilizar a seguinte instrução:

```
# ansible-playbook -e username=developer users.yml
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Objetivos da Aula

- ✔ Entender a infraestrutura;
- ✔ Criar a playbook de deploy.

Playbook de Deploy

Nesta aula iremos entender como foi criada a infraestrutura e como podemos criar um script de deploy para o nosso projeto.

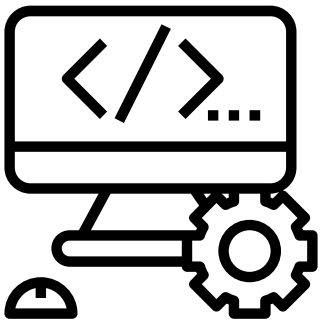
Anotações:

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Entendendo a estrutura

O deploy será feito em um container chamado gerencia dentro da máquina Docker. O nosso playbook fará o deploy dentro de um volume configurado **/var/lib/docker/volumes/projeto-final/** e o container possui a configuração de apache e redirecionamento de porta para receber a aplicação.



Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Criando Playbook

Podemos criar um arquivo chamado “playbook.yml” onde ele deverá fazer as seguintes tasks:

- ✔ Instalar o GIT
- ✔ Fazer o git clone do projeto para o volume
- ✔ Alterar permissão do diretório
- ✔ Fazer o restart do Apache

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Podemos iniciar o nosso playbook definindo onde ele será executado e as variáveis de usuário e senha, estes campos devem ser substituídos pelos dados de acesso ao Git:

```
---
- name: Deploy do Projeto Final
  hosts: localhost
  become: true
  become_method: sudo
  vars:
    - user: usuario
    - pwd: 'senha'
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Instalar o GIT

Agora podemos definir as nossas tasks, podemos começar com a instalação do GIT:

```
tasks:
    - name: Instalar o git
      apt: name={{ item  }}
      with_items:
        - git
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Após a insalação do GIT, faremos o clone do projeto:

```
- name: Baixar o projeto
      git:
        repo: 'http://{{ user | urlencode }}:{{pwd | urlencode }}@URL'
        dest: /var/lib/docker/volumes/projeto-final/gerencia
        version: master
        accept_hostkey: yes
        force: yes
```

Vamos substituir a **URL** pelo nosso projeto do Gitlab.

Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

E agora podemos alterar a permissão do diretório e fazer o restart do apache:

```
- name: Alterando permissao do diretorio
  shell: docker exec gerencia chown -R www-data.www-data /opt/gerencia

- name: Restart do apache
  shell: docker exec gerencia service apache2 restart
```

Agora o script pode ser executado com:

```
ansible-playbook playbook.yml
```


Anotações:

Documento produzido pela 4Linux – Todos os direitos reservados.

Felipe Moz / moz.felipe@gmail.com

Playbook de Deploy

Recapitulando

- ✔ Conhecer a estrutura criada;
- ✔ Criar o script para deploy da aplicação.

Playbook de Deploy

Nesta aula criamos uma playbook para fazer o Deploy da aplicação.

Anotações: