



Web services em Python

Interface para integração de aplicações, Web services e as tecnologias SOAP e REST, com demonstração do consumo e fornecimento de Web services em Python.

Prof. Denis Gonçalves Cople

Propósito

Compreender os conceitos relacionados a Web services, como **SOA**(arquitetura orientada a serviços), **SOAP** (protocolo simples de acesso a objetos) e **REST**(transferência representacional de estado), demonstrando o fornecimento e consumo de Web services SOAP e REST utilizando a linguagem de programação Python.

Preparação

Para implementar os Web services, será necessário instalar o ambiente de desenvolvimento **Spyder**, para **Python**, e o **Miniconda**, que fornecerá uma linha de comando mais versátil para a instalação de bibliotecas e frameworks, via comando **pip**, além da execução de servidores minimalistas criados com **Flask** e **Django**.

Objetivos

- Descrever os conceitos de Web services.
- Descrever o consumo de Web services por meio do protocolo SOAP utilizando Python.
- Descrever o consumo de Web services RESTful utilizando Python.
- Reconhecer o conceito de API e sua implementação com base em Web services.

Introdução

Antes de falarmos de web services, é importante situá-los no contexto da arquitetura orientada a serviços (SOA, do inglês service oriented architecture). Quando falamos de SOA, estamos falando de um padrão de arquitetura de software, baseado nos princípios da computação distribuída, em que as funcionalidades existentes no software devem estar disponíveis no formato de serviços.

Nesse ponto, temos os web services, que podem ser definidos como uma interface que permite a comunicação entre clientes e serviços; a comunicação entre máquinas; a comunicação entre softwares, podendo ser escritos em diferentes linguagens e residir em diferentes plataformas. Em outras palavras, é por meio de web services que podemos acessar os serviços disponibilizados em softwares construídos utilizando a arquitetura SOA. Temos, então, o SOAP e o REST, que são duas diferentes abordagens que permitem a transmissão de dados nos web services.

Ao longo deste tema, os web services, assim como as abordagens SOAP e REST, serão descritos conceitualmente. Além disso, o seu uso será demonstrado por meio de exemplos práticos, nos quais utilizaremos a linguagem de programação Python.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O que são web services?

Neste vídeo, vamos desvendar o mundo dos Web Services e entender o papel fundamental que desempenham na comunicação entre aplicações web. Aprenderemos sobre os conceitos básicos dos Web Services, sua arquitetura e como eles permitem a integração entre sistemas distribuídos. Discutiremos os diferentes tipos de Web Services, como RESTful e SOAP, e sua importância no desenvolvimento de aplicações modernas.



Conteúdo interativo

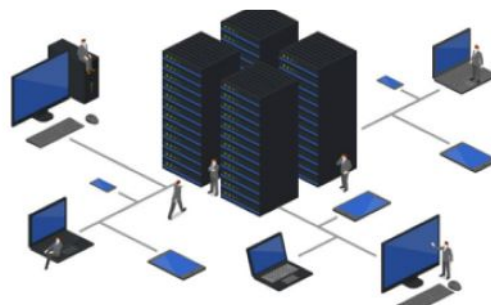
Acesse a versão digital para assistir ao vídeo.

A arquitetura de web services pode ser construída com base nos modelos SOAP e REST, cada um com características próprias, mas com o mesmo objetivo, que seria a exposição de serviços de maneira interoperável na web. Enquanto o SOAP assume um formalismo maior, sendo muito utilizado na comunicação entre empresas, o REST tem uma escrita mais simples, constituindo uma boa solução para a comunicação com o consumidor final. Ao longo deste conteúdo, veremos as características de cada um deles, permitindo que você possa diferenciá-los em termos arquiteturais.

Web services

No início da computação distribuída, com as aplicações distribuídas, a comunicação entre cliente e servidor era restrita a uma rede interna, ficando o servidor responsável por efetuar todo o processamento.

Posteriormente, esse processamento passou a ser feito entre vários servidores, onde era comum o uso de **middlewares** como **CORBA** (common object request broker architecture), **DCOM** (distributed component object model) e **RMI** (remote method invocation), sendo tais middlewares responsáveis por prover a comunicação nos sistemas distribuídos.



Middleware

Middleware é a infraestrutura de software localizada entre o sistema operacional e uma aplicação distribuída. Também pode ser considerado como middleware a camada de software entre o front-end e o back-end de um sistema.

Mais recentemente, as aplicações cliente x servidor migraram para a internet, dando origem aos web services, que surgiram como uma extensão dos conceitos de chamada remota de métodos, presentes nos middlewares já mencionados, para a web. Logo, podemos dizer que os web services são aplicações distribuídas que se comunicam por meio de mensagens. Ou, usando outras palavras, um web service é uma interface que descreve uma coleção de operações acessíveis pela rede por meio de mensagens. Nesse sentido, temos transações e regras de negócio de uma aplicação expostas por meio de protocolos que são acessíveis e compreensíveis por outras aplicações – podendo estas serem escritas em qualquer linguagem de programação, além de residirem em qualquer sistema operacional.

Os web services e sua arquitetura

A arquitetura dos web services é baseada em três componentes. A seguir, vamos conhecer um pouco mais sobre cada um deles.

Provedor de serviços

O primeiro componente, o provedor de serviços, é responsável pela criação e descrição do web service – em um formato padrão, compreensível para quem precise utilizar o serviço – assim como pela sua disponibilização a fim de que possa ser utilizado.

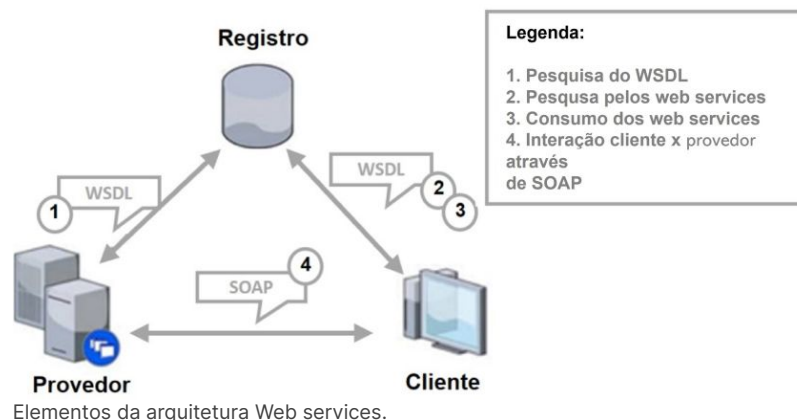
Consumidor de serviços

Esse componente, ou papel, é representado por quem utiliza um Web service disponibilizado em um provedor de serviços.

Registro de serviços

Trata-se de um repositório a partir do qual o provedor de serviços pode disponibilizar seus web services e no qual o consumidor de serviços pode utilizá-los. Em termos técnicos, o registro dos serviços contém informações, como os detalhes da empresa, os serviços por ela oferecidos e a descrição técnica do web services.

A imagem a seguir ilustra como funcionam esses componentes.



WSDL e UDDI

Conforme pôde ser visto anteriormente, além dos elementos já apresentados, há ainda outros que compõem a arquitetura dos Web services, como a **WSDL**, o **SOAP**, assim como a **XML** e a **UDDI**. A seguir, conheceremos um pouco mais sobre as tecnologias WSDL e UDDI. Já o SOAP será visto mais adiante, assim como o REST, em tópicos específicos.

WSDL

A WSDL (*web services description language*) é uma linguagem baseada em XML, cuja função é descrever, de forma automatizada, os serviços do web service por meio de um documento acessível aos clientes que desejam fazer uso do web service. A WSDL é responsável por fornecer as informações necessárias para utilização de um web service, como as operações disponíveis e suas assinaturas.

UDDI

A UDDI (*universal description, discovery and integration*) é responsável por prover um mecanismo para a descoberta e publicação de web services. Nesse sentido, a UDDI contém informações categorizadas sobre as

funcionalidades e serviços disponíveis no web service, permitindo, ainda, a associação de informações técnicas, normalmente definidas com o uso da WSDL, a esses serviços.

SOAP e REST

Inicialmente, no contexto da computação distribuída, eram utilizadas tecnologias como RMI, DCOM e CORBA para a integração de aplicações. Nesse cenário, tais tecnologias obtiveram sucesso quando aplicadas em ambientes de rede locais e homogêneos. Posteriormente, já no ambiente heterogêneo da Internet, outras soluções foram aplicadas por meio da construção de aplicações web escritas em linguagens como Java (JSP), ASP e PHP. Essas aplicações, em termos de integração com outras aplicações, faziam uso de XML.

Embora a XML seja um formato de transmissão de dados padronizado, faltava padronização por parte das empresas em termos de desenvolvimento, utilização de protocolos, transações, segurança etc. Frente a isso, o **W3C** desenvolveu um padrão cujo principal objetivo era prover a interoperabilidade entre aplicações. Tal padrão recebeu o nome de "**Padrões WS-***" e é constituído por especificações para criação de web services baseados no protocolo **SOAP**.



Dica

Veja mais a respeito dessas especificações no site do W3C, conforme está indicado no Explore+

O padrão WS-* é composto por várias especificações, como a **WS-Addressing**, que trata dos mecanismos de transporte para a troca de mensagens nos Web services, e a **WS-Security**, um protocolo que trata da segurança nos Web services, entre outras.

O REST (**Representational State Transfer**), diferentemente do SOAP, não é uma especificação nem foi criado pelo W3C. Em linhas gerais, trata-se de uma forma alternativa, uma arquitetura web para o consumo de web services, e que se baseia na utilização de recursos oferecidos pelo próprio protocolo HTTP.

Modelo SOAP

Neste vídeo, vamos conhecer os web services e sua arquitetura.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

SOAP (simple object access protocol)

O **SOAP** é um protocolo, baseado em definições XML, utilizado para a troca de informações/comunicação em ambiente distribuído. Tal protocolo encapsula as chamadas e os retornos a métodos de web services, trabalhando, principalmente, sobre o protocolo HTTP. Com o uso de SOAP, é possível invocar aplicações remotas utilizando chamadas RPC ou troca de mensagens, sendo indiferente o sistema operacional, a plataforma ou a linguagem de programação das aplicações envolvidas.

Comunicação em SOAP

Web services que fazem uso do protocolo SOAP podem utilizar dois modelos distintos de comunicação:

RPC

Nesse modelo, é possível modelar chamadas de métodos com parâmetros, assim como receber valores de retorno. Com ele, o corpo (body) da mensagem SOAP contém o nome do método a ser executado e os parâmetros. Já a mensagem de resposta contém um valor de retorno ou de falha.

Document

Nesse modelo, o body contém um fragmento XML que é enviado ao serviço requisitado, no lugar do conjunto de valores e parâmetros presente no RPC.

Formato de mensagem

Uma mensagem SOAP é composta por três elementos:

Envelope

Elemento principal (raiz do documento) do XML, responsável por definir o conteúdo da mensagem. É um elemento obrigatório.

Header

Mecanismo genérico que torna possível a adição de características, de informações adicionais, à mensagem. É um elemento opcional, mas que, quando utilizado, deve ser o primeiro elemento do Envelope.

Body

Corpo da mensagem. Contém a informação a ser transportada. Assim como o Envelope, é um elemento obrigatório.

A seguir, podemos ver o fragmento XML contendo os elementos definidos acima.

```
plain-text
```

Observe que, conforme visto no código, o elemento Body pode conter um elemento opcional, o Fault. Tal elemento é usado para transportar mensagens de status e/ou erros.

Exemplo de requisição e resposta utilizando SOAP

Para melhor compreensão, veremos a seguir um exemplo prático de **requisição** e **resposta** de Web service utilizando o protocolo **SOAP**.

Nesse exemplo, será invocado o método **"GetModulosTema"**. Esse método recebe como parâmetro o nome do tema, representado pela variável **"TemaNome"**. Como resposta, são retornados os nomes dos módulos relacionados ao tema informado. O XML contendo o envelope da requisição pode ser visto no código seguinte:

```
plain-text
```

```
Webservices
```

A seguir, é demonstrado o XML do envelope contendo a resposta do método invocado.

```
plain-text
```

```
SOAP e REST
```

```
Utilização de SOAP XML em JAVA
```

```
Utilização de REST JSON em JAVA
```

Modelo REST

Neste vídeo, você verá mais detalhes sobre o protocolo SOAP e sua aplicação.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

REST (representational state transfer)

O REST foi proposto por Roy Fielding, um dos criadores do protocolo HTTP, em 2000, com a premissa de utilizar os recursos oferecidos pelo HTTP. Trata-se de um modelo mais simples que o SOAP, além de não ser um protocolo, mas sim uma arquitetura web, composta pelos seguintes elementos:

- Cliente (do web service)
- Provedor (do web Service)

- Protocolo HTTP

Considerando esses elementos, o consumo de um web service que faz uso de REST tem seu ciclo de vida iniciado com o cliente enviando uma solicitação a um determinado provedor. Esse provedor, após processar a requisição, responde ao cliente. Além disso, o HTTP é o protocolo que define o formato das mensagens enviadas e recebidas, além de também ser responsável pelo transporte dessas mensagens.



Atenção

A exemplo do WSDL, utilizado para web services SOAP, em REST está disponível a WADL (web application description language), cuja função também é a de descrever serviços – nesse caso, os Web services RESTful.

Estrutura dos recursos REST

Na arquitetura REST, os serviços ou recursos disponíveis correspondem a uma **URI** (*uniform resource identifier*) específica e que também é única. Se considerarmos o exemplo visto no protocolo SOAP, podemos dizer que **"GetModulosTema"** é um método pertencente a um recurso – que vamos chamar de **"Tema"**. Logo, a URI para consumo desse serviço seria:

```
plain-text
```

```
http://www.dominio.com.br/tema/GetModulosTema/{nome-do-tema}
```

Considerando então que "Tema" é o nome do recurso, podemos imaginar outros métodos disponíveis no mesmo. Poderíamos ter, por exemplo, um método para listar todos os temas ou um método para inserir um novo tema. Cada um desses serviços teria uma URI própria:

- **Listagem de todos os temas** `http://www.dominio.com.br/tema`
- **Inserção de tema** `http://www.dominio.com.br/tema/CreateTema/{nome-do-tema}`

Uma vez que os web services REST são baseados no protocolo HTTP, a estrutura dos recursos REST, como vimos, provém justamente dos métodos e códigos de retorno HTTP. Isso, em termos práticos, significa dizer que devemos usar os diferentes métodos HTTP de acordo com as operações para manipulação de dados dos recursos que desejamos fazer.



Exemplo

Para recuperar dados, como no caso onde queremos listar todos os temas existentes, devemos usar o método HTTP GET.

A seguir estão listados os métodos HTTP e suas funções em relação à arquitetura REST.

GET

Usado na **recuperação** ou **listagem** de recursos.

POST

Usado na **inclusão** de um recurso.

PUT

Usado na **edição** de um recurso.

DELETE

Usado na **exclusão** de um recurso.

Como já mencionado, REST utiliza os recursos do protocolo HTTP. Logo, em relação às respostas dos serviços, temos disponíveis os códigos de retorno HTTP.

Por exemplo, para verificarmos se um recurso foi atualizado com **sucesso**, devemos verificar se o código HTTP é igual a **200**. Caso algum **erro** tenha ocorrido, teremos o código **400** ou **404**.

Exemplo de requisição e resposta utilizando REST

O consumo de um recurso REST é feito por meio de uma URI. Logo, poderíamos acessar esse recurso até mesmo por meio de um navegador web, sendo a forma mais usual, quando falamos de integração entre aplicações, a implementação de um cliente, por meio de uma linguagem de programação, que acesse o recurso em questão, enviando parâmetros, quando necessário, e tratando o retorno dele. Nesse contexto, vamos usar o mesmo exemplo visto em SOAP e recuperar a listagem de módulos disponíveis para um determinado tema.

Para consumir o serviço, vamos utilizar a seguinte URI:

plain-text

`http://www.dominio.com.br/tema/GetModulosTema/Webservices`

Um possível retorno para essa requisição é visto a seguir:

plain-text

```
{
  "Modulos": [
    {"Nome": "SOAP e REST"},
    {"Nome": "Utilização de SOAP XML em JAVA"},
    {"Nome": "Utilização de REST JSON em JAVA"}
  ]
}
```

Como vimos, as informações retornadas pelo web service consumido estão no formato JSON. Embora não seja o único tipo de dado disponível para o transporte de informações em REST – ou melhor, em mensagens HTTP –, ele é o mais utilizado nessa arquitetura.

Como curiosidade, em requisições REST podemos usar qualquer um dos tipos de conteúdo (*content-type*) a seguir:

- Application/xml;
- Application/json;
- Text/plain;
- Text/xml;
- Text/html.

SOAP UI

Neste vídeo, demonstramos como utilizar a ferramenta SOAP UI para testar um web service do tipo SOAP e a outro do tipo REST, observando a diferença entre eles.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Estudamos algumas definições aplicáveis aos web services. Marque a opção que corresponde a uma dessas definições.

A

Os web services são serviços para integração de aplicações que, para se comunicarem, precisam ser escritos em uma mesma linguagem de programação.

B

Embora não seja obrigatório serem escritos em uma mesma linguagem de programação, os web services, como serviços que integram diferentes aplicações, precisam ser hospedados no mesmo tipo de servidor web e sistema operacional para que a comunicação possa ser estabelecida.

C

Os web services são uma solução utilizada na integração e comunicação entre diferentes aplicações.

D

A integração entre aplicações é uma novidade que só se tornou possível com o advento da internet.

E

A partir do advento da Internet, a integração por meio de web services tornou-se restrita ao ambiente da internet. Ou seja, não é mais possível integrar aplicações hospedadas em uma mesma rede interna.



A alternativa C está correta.

Os web services são uma solução tecnológica para a integração de diferentes aplicações e que independe de fatores como localização das aplicações – se estão em uma mesma rede ou na internet; de linguagens utilizadas em sua escrita; sistemas operacionais ou servidores web utilizados em sua hospedagem.

Questão 2

A respeito dos conceitos relacionados ao protocolo SOAP e da arquitetura REST, é correto afirmar:

A

Deve-se priorizar a utilização do protocolo SOAP, uma vez que se trata de uma especificação organizada e mantida pelo W3C.

B

Embora não seja uma especificação mantida pelo W3C, a arquitetura REST é a solução mais utilizada atualmente, tendo substituído por completo a utilização de SOAP, ficando esta última restrita aos sistemas legados.

C

Ao considerarmos os detalhes de implementação, SOAP e REST são iguais e, portanto, um mesmo código de integração escrito para um funcionará com o outro.

D

A escolha entre SOAP e REST deve considerar que nem toda linguagem de programação tem suporte a ambos, já que o SOAP é um formato fechado, restrito, enquanto REST é um formato aberto.

E

SOAP e REST são tecnologias que cumprem um mesmo papel, o de integrar diferentes aplicações. Em linhas gerais, ambas cumprem o mesmo objetivo e a escolha entre uma ou outra deve considerar detalhes e necessidades específicas de cada projeto.



A alternativa E está correta.

O protocolo SOAP e a arquitetura REST são tecnologias semelhantes, no sentido de cumprirem com um mesmo objetivo, o de integrar diferentes aplicações. Entretanto, há particularidades na implementação de cada uma e a escolha entre elas deve considerar uma série de aspectos, inerentes aos projetos nos quais há necessidade de integração e troca de mensagens entre aplicações.

Construindo web service SOAP em Python

Django e Spyne

Neste vídeo, apresentaremos protocolo SOAP com Python estruturando a aplicação.

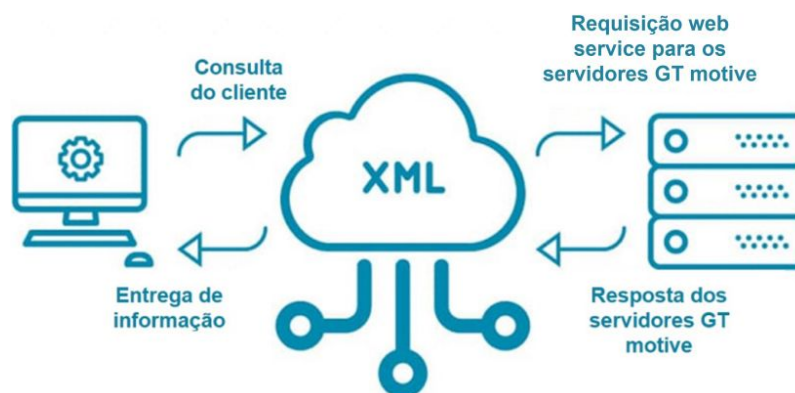


Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Protocolo SOAP com Python

Veremos aqui como codificar um web service utilizando o protocolo SOAP. Também veremos como codificar um cliente para consumir os serviços fornecidos. Com os fundamentos aqui demonstrados, você terá o conhecimento mínimo necessário para criar web services simples, assim como criar aplicações para consumir web services de terceiros a partir do ambiente Python.



Fluxo de requisição e resposta em um Web service SOAP

Antes de iniciar, veja na seção "Preparação" se você possui tudo o que precisaremos para desenvolver nosso web service.

Estruturando a aplicação

Nossa aplicação será um **projeto web** baseado no **Django**, que será responsável por hospedar todos os elementos que serão disponibilizados via protocolo **HTTP**. Nessa aplicação, definiremos os recursos que serão disponibilizados via **SOAP**, além de um componente para o tratamento de requisições simples, cujo papel será o de **Cliente**, ou seja, deverá consumir os recursos definidos em nosso Web service.

Django é um **framework** para criação de aplicativos **Web**, que utiliza o padrão denominado model-template-view, definindo uma estrutura básica para a criação de sites completos por meio do ambiente Python.

Outra ferramenta que utilizaremos será o **Spyne**, uma biblioteca que permite criar servidores do tipo **RPC** (**remote procedure call**), com base em diversos protocolos de entrada e saída, e aqui veremos como utilizá-lo para a comunicação via **SOAP**. Uma das grandes vantagens no uso dessa biblioteca é a geração automática do arquivo **WSDL**, facilitando o acesso por terceiros.

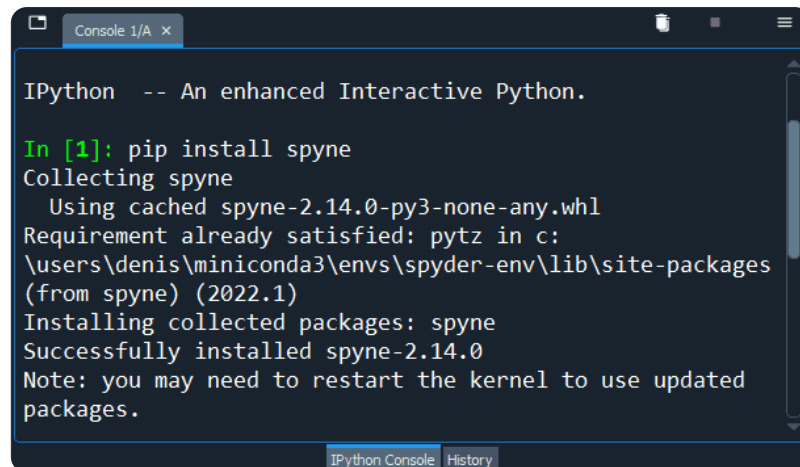
Finalmente, será necessário utilizar a biblioteca **lxml**, que efetua a transformação do formato **XML** para **Python** de forma simples, assim como o caminho contrário.

Inicialmente, devemos instalar todas as ferramentas necessárias por meio do **Miniconda** ou via console interno do **Spyder**, utilizando os comandos apresentados a seguir.

```
plain-text

pip install spyne
pip install lxml
pip install django
```

Parte da instalação das ferramentas pode ser observada a seguir, sendo executada no console interno do **Spyder**.

A screenshot of the Spyder IPython console window. The title bar says 'Console 1/A x'. The text inside the console reads: 'IPython -- An enhanced Interactive Python.', 'In [1]: pip install spyne', 'Collecting spyne', 'Using cached spyne-2.14.0-py3-none-any.whl', 'Requirement already satisfied: pytz in c:\users\denis\miniconda3\envs\spyder-env\lib\site-packages (from spyne) (2022.1)', 'Installing collected packages: spyne', 'Successfully installed spyne-2.14.0', and 'Note: you may need to restart the kernel to use updated packages.'. At the bottom of the console, there are tabs for 'IPython Console' and 'History'.

Instalação de componente a partir do console interno do Spyder.

Com as ferramentas instaladas, vamos criar o diretório DjangoApps e, por meio do console do Miniconda, vamos navegar para ele e criar nosso aplicativo Django, com o nome DisciplinasWS, por meio da sequência de comandos apresentada a seguir.

```
plain-text

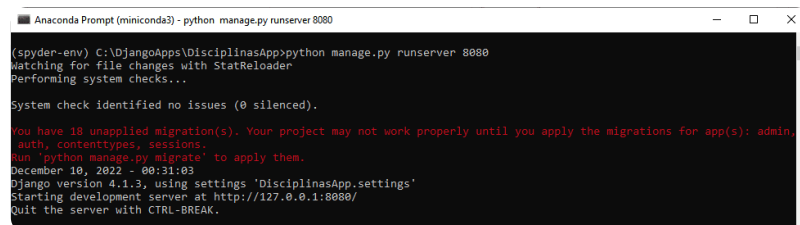
cd /
cd DjangoApps
django-admin startproject DisciplinasApp
```

Em seguida, nosso projeto pode ser executado, utilizando o comando apresentado a seguir, no diretório do aplicativo, e o teste pode ser feito por meio de qualquer navegador, com o acesso ao endereço **http://localhost:8080**.

```
plain-text

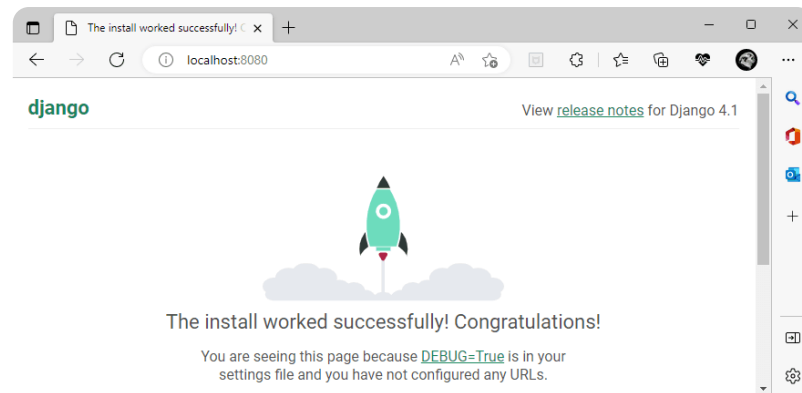
cd DisciplinasApp
python manage.py runserver 8080
```

A execução dos comandos, no console do **Miniconda**, pode ser observada a seguir.

A screenshot of the Anaconda Prompt window. The title bar says 'Anaconda Prompt (miniconda3) - python manage.py runserver 8080'. The text inside the prompt shows the execution of 'python manage.py runserver 8080'. It displays the Django version (4.1.3), the settings used ('DisciplinasApp.settings'), and the starting development server at 'http://127.0.0.1:8080/'. It also shows a message about 10 unapplied migrations and a prompt to run 'python manage.py migrate'.

Execução do projeto DisciplinasApp no Miniconda.

Testando o acesso por meio de um **navegador**, deveremos ter o resultado apresentado a seguir.



Acesso ao aplicativo DisciplinasApp por meio do navegador

Como podemos observar na estrutura criada pelo Django, temos o diretório **DisciplinasApp**, com o arquivo **manage.py**, para inicialização e gerenciamento do servidor, e um arquivo padrão de banco de dados **SQLite**. Dentro do primeiro diretório, temos outro com o mesmo nome, definindo um pacote no formato **Python**, onde podemos destacar o arquivo **urls.py**, responsável pelas rotas do aplicativo.

A estrutura da aplicação pode ser vista a seguir:

```
plain-text
- DisciplinasApp
  - db.sqlite3
  - manage.py
  - DisciplinasApp
    - __pycache__
    - __init__.py
    - asgi.py
    - settings.py
    - urls.py
    - wsgi.py
```

Cliente e provedor SOAP

Clientes Zeep para SOAP

Neste vídeo, veremos os principais aspectos em relação ao cliente SOAP.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Provedor SOAP

Com o aplicativo criado, vamos interromper sua execução, com o uso de **CTRL+C** no console do Miniconda, e adicionar o arquivo **repositorio.py** no diretório **DisciplinasApp** interno, o mesmo de **urls.py**, utilizando o código da listagem seguinte.

python

```
class DisciplinasRepo:
    def __init__(self):
        self.repo = {
            "Webservices":
                ["Conceitos de Web services",
                 "Utilizando SOAP em Java",
                 "Utilizando REST em Java"],
            "Programação Servidor com Java":
                ["Webserver Tomcat",
                 "App Server GlassFish",
                 "Servlet e JSP"],
            "JPA e JEE":
                ["Tecnologia JPA",
                 "Enterprise Java Beans",
                 "Arquitetura MVC"]
        }
    def getTemas(self):
        return self.repo.keys()
    def getModulosTema(self, tema):
        return self.repo[tema]
```

A classe **DisciplinasRepo** é apenas um repositório de dados baseado em um **dicionário** do Python, onde as chaves são temas e os valores são listas de módulos para cada tema. Temos ainda os métodos **getTemas**, para retornar os temas a partir do dicionário, e **getModulosTema**, que tem como parâmetro um tema, retornando os módulos do tema fornecido. Agora vamos adicionar outro arquivo no mesmo diretório, com o nome **provedor.py**, onde adotaremos o código da listagem seguinte.

```
python

# -*- coding: utf-8 -*-
from spyne import Application, rpc, ServiceBase, Unicode
from spyne import Iterable
from spyne.protocol.soap import Soap11
from spyne.server.django import DjangoApplication
from django.views.decorators.csrf import csrf_exempt

from DisciplinasApp.repositorio import DisciplinasRepo

disciplinas = DisciplinasRepo()

class DisciplinasService(ServiceBase):
    @rpc(_returns=Iterable(Unicode))
    def getTemas(ctx):
        for i in disciplinas.getTemas():
            yield i
    @rpc(Unicode, _returns=Iterable(Unicode))
    def getModulosTema(ctx,tema):
        for i in disciplinas.getModulosTema(tema):
            yield i

application = Application([DisciplinasService],
    tns='disciplinas.app.ws',
    in_protocol=Soap11(validator='lxml'),
    out_protocol=Soap11()
)

# Este módulo deve estar em um pacote do aplicativo Django
disciplinas_ws = csrf_exempt(DjangoApplication(application))
```

Aqui temos um código bem mais complexo, no qual definimos nosso web service SOAP. A classe **DisciplinasService**, derivada de **ServiceBase**, contém a implementação dos serviços, por meio de métodos anotados como RPC.

A anotação **RPC** deve indicar os tipos dos parâmetros de entrada, na ordem em que aparecem no método associado, e o tipo do retorno, na cláusula **_returns**. Para o método **getTemas**, não temos parâmetros na chamada SOAP, e retornamos uma lista (**Iterable**) de texto (**Unicode**), enquanto **getModulosTema** tem um parâmetro de texto na requisição.

Podemos observar que o método **getTema** tem apenas um parâmetro **ctx**, que receberá o contexto de execução, sem parâmetros de requisição, e retorna as chaves do repositório, com o uso de uma estrutura **for** e **yield**. O mesmo comportamento ocorre para **getModulosTema**, mas agora com o parâmetro adicional **tema**, fornecido como **Unicode** a partir da **requisição**, retornando os módulos associados no dicionário.

Após definir a classe de serviço, devemos utilizá-la em um aplicativo (**Application**), definindo uma expressão para o **tns**, protocolo de entrada **SOAP** com validador **lxml**, e protocolo de saída como **SOAP**. Ao final, registramos **disciplinas_ws** como um recurso **DjangoApplication**, baseado no aplicativo definido anteriormente.

Falta apenas definir a rota de acesso ao serviço, no arquivo **urls.py**, que deverá ficar como o que é apresentado na listagem seguinte.


```
python

from DisciplinasApp.provedor import disciplinas_ws
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('disciplinasws/', disciplinas_ws)
]
```

Tudo que fizemos foi importar o componente **disciplinas_ws**, e associá-lo, por meio do método **path**, ao caminho **disciplinasws**. Executando o projeto, poderemos verificar o descritor **WSDL**, que é gerado de forma automática pelo **spyne**, com a utilização de um navegador e acesso ao endereço **http://localhost:8080/disciplinasws?wsdl**.

Cliente SOAP

Para facilitar a criação do cliente, vamos inicialmente instalar a biblioteca **zeep**, que permite inferir nos métodos do web service a partir de seu descritor **WSDL**. O comando para instalar a biblioteca é apresentado a seguir.

```
plain-text

pip install zeep
```

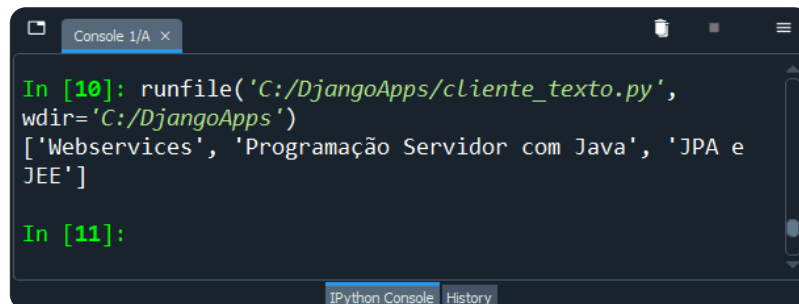
Criaremos um cliente com interface de **linha de comando**, fora do aplicativo Django, no arquivo **cliente_texto.py**, de acordo com a listagem apresentada a seguir.

```
python

# -*- coding: utf-8 -*-
from zeep import Client
client = Client('http://localhost:8080/disciplinasws?wsdl')
result = client.service.getTemas()
print(result)
```

Note como a utilização da biblioteca é extremamente simples, instanciando um cliente a partir do endereço do descritor, e efetuando as chamadas como se fossem simples chamadas locais de métodos. Na prática, temos a implementação do padrão **Proxy**, com a delegação de cada chamada para o servidor, e obtenção do resultado, de forma totalmente transparente.

Executando o arquivo no **Spyder**, poderemos observar os temas sendo exibidos no console interno, como é apresentado a seguir.



```
Console 1/A x

In [10]: runfile('C:/DjangoApps/cliente_texto.py',
wdir='C:/DjangoApps')
['Webservices', 'Programação Servidor com Java', 'JPA e
JEE']

In [11]:
```

Execução do cliente de teste em modo texto.

Como nosso passo final, vamos criar um sistema web padrão, com acesso ao nosso serviço via cliente zeep, aproveitando a estrutura fornecida pelo Django.

Criando a interface web

Interface web para o serviço

A arquitetura do Django utiliza templates HTML alimentados a partir de funções do Python, e devemos iniciar a definição de nossa camada **web** criando o diretório com o nome **templates**, ao nível da raiz do projeto, no qual se encontra o arquivo **manage.py**. Após criar o diretório, será necessário adicionar seu nome ao conjunto **DIRS**, de **TEMPLATES**, no arquivo **settings.py**, que se encontra no diretório **DisciplinasApp interno**, como você pode observar.

```
python

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ['templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

Ainda no diretório DisciplinasApp interno, vamos criar o arquivo **views.py**, utilizando o código da listagem apresentada a seguir.

```
python

from zeep import Client
from django.shortcuts import render

def index(request):
    cliente = Client('http://localhost:8080/disciplinasws?wsdl')
    temas = cliente.service.getTemas()
    context = {
        'temas': temas
    }
    return render(request, 'index.html', context=context)
```

Temos apenas a definição de uma função **index**, recebendo uma requisição HTTP em **request**, com a obtenção dos temas por meio de um cliente **zeep**, e envio dos temas recuperados para um template HTML, por meio do dicionário **context**. O retorno da função ocorre com base na função **render**, tendo como argumentos a requisição, o nome do template e o contexto.

Ainda precisamos criar o arquivo **index.html**, no diretório **templates**, definido anteriormente, utilizando o código da listagem seguinte.

plain-text

Temas

```
{% for t in temas %}

• {{ t }}

{% endfor %}
```

Temos um arquivo HTML comum, com algumas partes que serão substituídas pelo Django, no qual a estrutura **for** repetirá o trecho **li** para cada tema **t** do conjunto **temas**, fornecido pelo contexto. Internamente a cada tag **li**, será colocada a descrição do tema, por meio do valor de **t**.

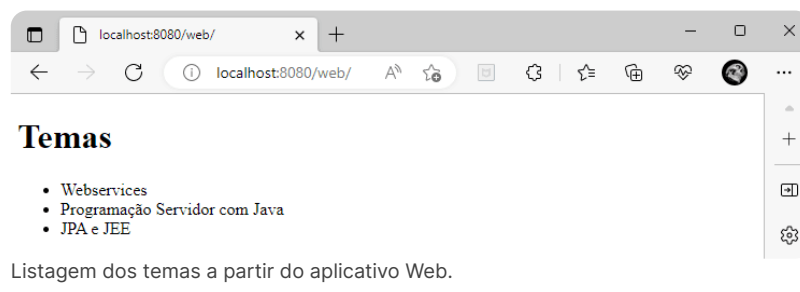
O último passo será uma nova alteração em **urls.py**, para definir a nova **rota** web, apontando para a função **index**.

```
python

from DisciplinasApp.provedor import disciplinas_ws
from DisciplinasApp.views import index
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('disciplinasws/', disciplinas_ws),
    path('web/', index)
]
```

Executando novamente o aplicativo, e acessando o endereço **http://localhost:8080/web** em um navegador, devemos ter o resultado apresentado a seguir.



Com relação à exibição dos **módulos** de cada tema, vamos modificar o arquivo **views.py**, de acordo com a listagem seguinte.

```
python

from zeep import Client
from django.shortcuts import render
from django.forms import Form, ChoiceField

class ModulosForm(Form):
    tema = ChoiceField(label='Tema', choices=[])

def index(request):
    cliente = Client('http://localhost:8080/disciplinasws?wsdl')
    temas = cliente.service.getTemas()
    context = {
        'temas': temas
    }
    return render(request, 'index.html', context=context)

def modulos(request):
    cliente = Client('http://localhost:8080/disciplinasws?wsdl')
    if request.method == 'POST':
        tema = request.POST['tema']
        modulos = cliente.service.getModulosTema(tema)
        context = {'modulos': modulos, 'tema': tema}
        return render(request, 'modulos_lista.html', context)
    else:
        form = ModulosForm()
        form.fields['tema'].choices = [tuple([x,x])
                                       for x in cliente.service.getTemas()]
        return render(request, 'modulos_form.html', {'form': form})
```

Acrescentamos uma classe **ModulosForm**, descendente de **Form**, para encapsular os campos de um formulário HTML. Na prática, temos apenas um componente do tipo **ChoiceField**, que equivalerá a uma lista de seleção no formulário, inicialmente sem as opções preenchidas, já que necessitam de um acesso ao nosso web service.

Em seguida, definimos a função **modulos**, tendo a requisição HTTP no parâmetro **request**, e nela implementamos dois comportamentos:

GET

Este modo retorna o formulário para a seleção do tema.

POST

Este modo retorna a lista de módulos a partir do tema selecionado.

Se a requisição é do tipo **POST**, obtemos o valor do tema a partir da mesma, e usamos o cliente zeep para obter seus módulos, com a chamada para **getModulosTema**. Em seguida, o tema e os módulos são colocados em um dicionário de contexto, ocorrendo o redirecionamento para **modulos_lista.html** com a passagem do **contexto** criado.

Já no modo **GET**, instanciamos um objeto do tipo **ModulosForm**, e preenchemos as opções para a seleção do **tema** a partir de uma chamada para **getTemas** pelo cliente zeep, observando que foi necessário transformar em tuplas com valor e descrição, no caso equivalentes. Em seguida, redirecionamos para **modulos_form.html**, com a passagem da instância de **ModulosForm** no atributo **form** do contexto.

Agora precisamos das páginas HTML, que serão colocadas no diretório templates, a começar por **modulos_form.html**, cujo código é apresentado a seguir.

plain-text

Selecione o Tema

```
{% csrf_token %}
{{ form }}
```

Temos uma página bastante simples, onde o formulário tem seus componentes de entrada gerados a partir da referência ao valor de **form**, fornecido a partir do contexto. Temos ainda uma instrução **csrf token**, que inclui um token no formulário para evitar ataques do tipo **CSRF**, ou Cross-Site Request Forgery.

Com relação à segunda página, com o nome **modulos_lista.html**, seu código é apresentado na listagem seguinte.

plain-text

Módulos de {{ tema }}

```
{% for m in modulos %}

• {{ m }}

{% endfor %}
```

Temos uma codificação muito semelhante à que foi utilizada para a listagem de temas, neste caso usando a estrutura **for** para os **módulos**, além da recuperação do **tema** em si para a exibição no título da página.

Ainda falta acrescentar a rota **web/modulo**, apontando para a nova função de tratamento, no arquivo **urls.py**.

```
python

from DisciplinasApp.provedor import disciplinas_ws
from DisciplinasApp.views import index,modulos
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('disciplinasws/', disciplinas_ws),
    path('web/', index),
    path('web/modulo/', modulos)
]
```

Com tudo pronto, podemos acessar o endereço **http://localhost:8080/web/modulo**, por meio de um navegador, **selecionar** o tema, e clicar em **Exibir**, para visualização dos dados na segunda página. O resultado dessas ações pode ser verificado nas imagens seguintes.



Ao fim deste módulo, temos dois componentes Python funcionais: um web service SOAP e um cliente para o serviço no modelo web, ambos hospedados em um aplicativo Django.

Web service SOAP com Python

Neste vídeo, assista ao exemplo da construção de um provedor SOAP com Django e Spyne, seguido de uma linha de criação de um cliente de linha de comando com o zeep e a WSDL do serviço.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Estudamos sobre a criação de um provedor e um cliente de web services que fazem uso do protocolo SOAP. Nesse contexto, é correto afirmar que

A

nas situações em que precisamos recuperar informações de outra aplicação deveremos, obrigatoriamente, desenvolver tanto o provedor quanto o cliente referente ao web service.

B

as únicas ferramentas disponíveis para a criação de provedores e clientes de web services são a IDE Spyder e os projetos do tipo Django.

C

em termos de arquitetura, no que diz respeito ao provedor e ao cliente, é indiferente em qual camada os arquivos relacionados ao fornecimento de métodos/serviços ficarão hospedados.

D

na prática, a separação entre provedor e cliente serve apenas para cumprir com a especificação definida pelo W3C, onde consta que deverão sempre, em qualquer projeto, estar presentes o provedor, o registro e o cliente de serviços.

E

a criação do provedor e do cliente para web services é restrita ao tipo de projeto ou necessidade específica de nossa aplicação. Em outras palavras, se a necessidade em questão consiste em apenas consumir informações de outra aplicação, precisamos desenvolver apenas o cliente, já que o provedor estará do lado da aplicação que consultaremos.



A alternativa E está correta.

Os exemplos demonstrados, incluindo os projetos Python utilizados, tiveram papel de demonstrar a lógica relacionada à criação de um provedor e de um cliente para o consumo de web services SOAP. Logo, há outras formas de realizar tal implementação, devendo, então, ser consideradas as teorias aplicadas ao longo dos exemplos, inclusive para o acesso às APIs de terceiros existentes na web.

Questão 2

Considerando a arquitetura utilizada na implementação do provedor SOAP, escolha a alternativa correta.

A

Os dados de uma aplicação SOAP devem sempre ser armazenados no formato de listas do Python. Isso torna a aplicação mais leve e mais rápida.

B

Não é possível realizar a evolução de um provedor SOAP após a sua primeira implementação. Logo, se precisarmos de novas funcionalidades, deveremos definir um novo provedor.

C

Os testes dos serviços disponibilizados no provedor são restritos ao ambiente do próprio provedor, podendo ser realizados apenas por meio das funcionalidades da IDE utilizada em seu desenvolvimento.

D

Os métodos disponíveis em um provedor de Web services SOAP são altamente customizáveis, desde a quantidade e tipo de parâmetros que recebem, aos dados que retornam. Logo, é possível, por meio deles, implementar diferentes regras de negócio e devolver de dados simples a dados complexos.

E

Os métodos de um provedor de web service SOAP não podem receber parâmetros.



A alternativa D está correta.

Por se tratar de serviços ou métodos implementados em uma linguagem de programação – em nosso caso utilizamos Python –, as restrições relacionadas a esses recursos são as mesmas da linguagem de programação, quando existirem. Logo, é possível implementar regras complexas, em diferentes tipos de negócio e que recebam e devolvam diferentes tipos de dados.

Definindo a persistência dos dados

Vamos ver como configurar um aplicativo servidor **RESTful**, com base no **Flask**, que tem uma utilização mais simples que o Django, e vamos persistir os dados em um banco **PostgreSQL**, via **SQL Alchemy** e conector **psycopg**, utilizados por padrão pelo Flask. Após criar o servidor, vamos definir um cliente para sua utilização, com base na biblioteca **requests**, e alteraremos o nosso exemplo de aplicativo web, já criado anteriormente, para que acesse o novo serviço.



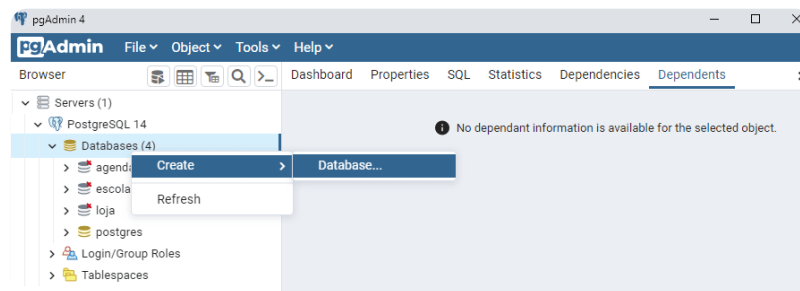
Atenção

Antes de iniciar, veja na seção "Preparação" se você possui instalado o PostgreSQL e sua ferramenta de administração PgAdmin. Download disponível em <https://www.postgresql.org/download/>

Definindo o banco de dados

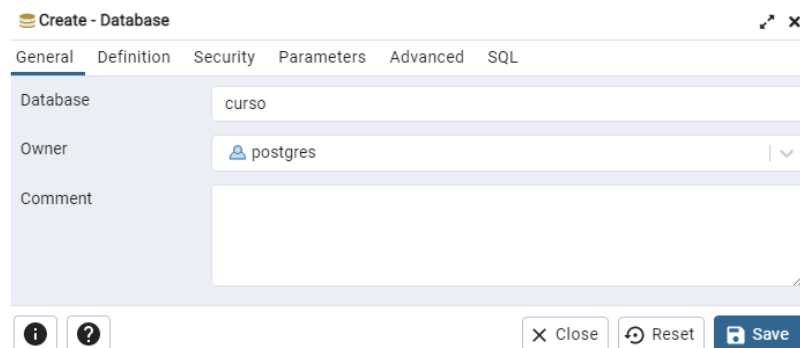
Em uma arquitetura dividida em camadas, como a Model, View e Controller, a camada **Model** está relacionada ao conjunto de entidades e componentes de acesso ao banco de dados. É onde ficam as classes **DAO**, e onde é efetuado o **mapeamento objeto-relacional**, mas para iniciar todo o processo, precisamos de uma base de dados constituída.

Inicialmente, vamos criar um banco de dados no PostgreSQL, por meio do **PgAdmin**, com o nome **curso**. A opção para **criação de banco** pode ser observada na imagem seguinte.



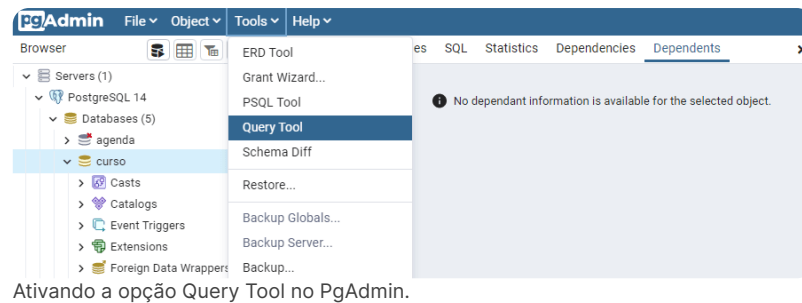
Criação de banco de dados no PostgreSQL

Na janela que será aberta, basta digitar o nome do banco, que no caso é **curso**, aceitando o usuário **postgres** como **owner**, e clicar em **Save**.



Captura de tela do PostgreSQL.

Com o banco criado e selecionado, vamos abrir a ferramenta para digitação dos comandos SQL, por meio do menu **Tools**, opção **Query Tool**, como pode ser observado a seguir.



Ativando a opção Query Tool no PgAdmin.

Já podemos criar a estrutura de nossas tabelas, por meio dos comandos da listagem seguinte, que devem ser digitados na área de edição da janela **Query Tool**, com execução por meio da tecla **F5**, ou clique no ícone de "play". Para que a sequência de comandos seja executada de uma só vez, será necessário selecionar todo o conteúdo antes de utilizar F5.

```
sql

CREATE TABLE TEMA(
    TEMA_ID INTEGER NOT NULL PRIMARY KEY,
    TEMA_NOME VARCHAR(255)
);

CREATE TABLE MODULO(
    MODULO_ID INTEGER NOT NULL PRIMARY KEY,
    MODULO_NOME VARCHAR(255),
    TEMA_ID INTEGER
);

ALTER TABLE MODULO ADD FOREIGN KEY(TEMA_ID) REFERENCES TEMA;
```

Após a execução dos comandos, teremos duas tabelas, **TEMA** e **MODULO**, com identificador e nome para cada entidade, e o relacionamento entre módulos e temas por meio de uma chave estrangeira (**FOREIGN KEY**). Após definir a estrutura de nosso banco de dados, vamos inserir alguns valores nas tabelas, onde o código seguinte contém uma sugestão de valores, com base nos dados usados em exemplos anteriores.

sql

```
INSERT INTO TEMA(TEMA_ID, TEMA_NOME)
VALUES(1, 'Programacao Servidor com Java');
INSERT INTO TEMA(TEMA_ID, TEMA_NOME)
VALUES(2, 'JPA e JEE');
INSERT INTO TEMA(TEMA_ID, TEMA_NOME)
VALUES(3, 'Webservices');
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(1, 'Webserver Tomcat', 1);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(2, 'App Server GlassFish', 1);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(3, 'Servlet e JSP', 1);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(4, 'Tecnologia JPA', 2);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(5, 'Entrepise Java Beans', 2);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(6, 'Arquitetura MVC', 2);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(7, 'Conceitos Web services', 3);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(8, 'Utilizando SOAP em Java', 3);
INSERT INTO MODULO(MODULO_ID, MODULO_NOME, TEMA_ID)
VALUES(9, 'Utilizando REST em Java', 3);
```

Após executar as inserções, a partir do editor do Query Tool, podemos dar início à codificação da camada **Model**, no ambiente do **Python**, em nosso próximo passo.

Definindo a camada Model

No ambiente do Python, o **SQLAlchemy** é uma ótima ferramenta para efetuar o mapeamento objeto-relacional (**ORM**). Por meio de um pequeno conjunto de classes e atributos especiais, essa biblioteca permite definir as entidades do sistema de forma a compatibilizar com qualquer tipo de banco de dados escolhido como back-end.

Vamos instalar a biblioteca por meio do comando seguinte.

```
plain-text

pip install sqlalchemy
```

Com a biblioteca instalada, nosso primeiro passo na definição da camada Model será a criação das classes de nossas entidades. Vamos criar um diretório com o nome **curso** para colocar os arquivos do projeto, e adicionar ao diretório o arquivo **entidades.py**, com o conteúdo seguinte.

```
python

# -*- coding: utf-8 -*-
from sqlalchemy import Column
from sqlalchemy import ForeignKey
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Tema(Base):
    __tablename__ = "tema"
    tema_id = Column(Integer, primary_key=True)
    tema_nome = Column(String(255))
    modulos = relationship(
        "Modulo", back_populates="tema", cascade="all, delete-orphan"
    )

class Modulo(Base):
    __tablename__ = "modulo"
    modulo_id = Column(Integer, primary_key=True)
    modulo_nome = Column(String(255))
    tema_id = Column(Integer, ForeignKey("tema.tema_id"))
    tema = relationship("Tema", back_populates="modulos")
```

Inicialmente temos uma classe **Base**, no modelo declarativo, definindo a estrutura básica para o mapeamento dos metadados necessários. A partir dela, derivamos as classes **Tema** e **Modulo**, com a especificação da tabela associada a cada uma no atributo especial **__tablename__**.

Os atributos são definidos por meio da classe **Column**, escolhendo o tipo do campo na tabela, como **Integer** ou **String**, além de possibilitar a marcação como chave primária (**primary_key**) e a especificação de chaves estrangeiras (**ForeignKey**). Temos ainda as declarações **relationship**, aqui utilizadas para configurar o relacionamento do tipo **um-para-muitos**, com um atributo do tipo Tema na classe Modulo e uma **lista** de objetos do tipo Modulo na classe Tema.

Precisamos de um **driver** de conexão com o **PostgreSQL**, que será instalado por meio do comando apresentado a seguir.

```
plain-text

pip install psycopg2-binary
```

Agora é necessário definir uma conexão com o PostgreSQL, utilizando o driver que acabamos de instalar. Vamos criar o arquivo **corsodb.py**, com o conteúdo apresentado a seguir.

```
python

from sqlalchemy import create_engine

postgresql_engine = create_engine(
    "postgres://postgres:admin123@127.0.0.1/curso",
    pool_reset_on_return=None,
)
```

Com base no método **create_engine**, definimos uma conexão com nosso banco de dados, por meio de uma **string de conexão**, especificando o tipo de banco (**postgresql**), usuário (**postgres**), senha (**admin123**),

endereço do servidor (**127.0.0.1**) e instância (**curso**). A partir desse objeto de conexão, denominado **postgresql_engine**, podem ser definidos elementos **Session**, para que sejam efetuadas as diversas operações sobre o banco de dados.

Agora vamos criar o arquivo **dao.py**, com as classes de gerência para módulos e temas, de acordo com a listagem apresentada a seguir.

```
python

# -*- coding: utf-8 -*-
from sqlalchemy import select
from cursodb import postgresql_engine
from entidades import Tema, Modulo
from sqlalchemy.orm import Session

session = Session(postgresql_engine)

class TemaDAO:
    def obterTodos(self):
        stmt = select(Tema)
        return session.scalars(stmt)
    def obter(self, tema_id):
        tema = session.get(Tema, tema_id)
        return tema
    def incluir(self, tema):
        session.add(tema)
        session.commit()
    def excluir(self, tema_id):
        tema = session.get(Tema, tema_id)
        session.delete(tema)
        session.commit()

class ModuloDAO:
    def obterTodos(self):
        stmt = select(Modulo)
        return session.scalars(stmt)
    def obterTodosTema(self, tema_id):
        stmt = select(Modulo).where(Modulo.tema_id == tema_id)
        return session.scalars(stmt)
    def incluir(self, modulo):
        session.add(modulo)
        session.commit()
    def excluir(self, modulo_id):
        modulo = session.get(Modulo, modulo_id)
        session.delete(modulo)
        session.commit()
```

Note que temos um objeto **session**, relacionado à conexão com o PostgreSQL, utilizado para as consultas, via método **scalars**, acréscimo de registro (**add**), remoção (**delete**), e confirmação das alterações (**commit**).

As consultas são baseadas em chamadas declarativas, via elementos **select** e **where**, como no método **obterTodosTema**, onde ocorre o retorno de todos os módulos para um tema específico, com base na condição de igualdade entre o atributo **tema_id** e o parâmetro de mesmo nome, enquanto nos métodos **obterTodos**, em ambas as classes, ocorre a seleção de todas as entidades, sem restrições. As classes de gerência seguem o padrão DAO (**data access object**), concentrando as operações de persistência para determinada entidade em cada uma delas. Note que as classes **TemaDAO** e **ModuloDAO** apresentam métodos muito similares para consulta, exclusão e inclusão, apenas com a modificação do tipo de entidade e campo identificador.

Para a **inclusão**, basta utilizar **add**, com a passagem de uma **entidade**, seguido da confirmação por meio de **commit**. Já para a **exclusão**, precisamos obter a entidade a partir do identificador, com a utilização do método **get**, remover do banco por meio do **delete**, e confirmar via **commit**.

Nossa camada **Model** está completa, e já podemos implementar o web service **RESTful**.

Criação da camada Model

Neste vídeo, assista ao processo de criação da camada Model, desde a definição da base no PostgreSQL até a codificação no Python, utilizando a biblioteca SQLAlchemy.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Criando um web service RESTful em Python

Neste vídeo, veremos os principais aspectos em relação a criação do web service RESTful.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Criação do web service RESTful

Para a criação de web services RESTful, utilizaremos o **Flask**, uma alternativa mais simples que o Django para a criação de servidores. Sua instalação é feita por meio do comando seguinte.

```
plain-text  
pip install Flask
```

A biblioteca Flask traz todos os recursos necessários para o tratamento de **requisições HTTP** e transformação para o formato **JSON**, ou seja, permite a criação de Web services RESTful, sem a necessidade de um servidor externo. Adicionalmente, as rotas são definidas via anotações, que são aplicadas aos métodos para tratamento das requisições.

Agora vamos criar o arquivo **cursoapp.py**, utilizando o conteúdo da listagem seguinte.

```

python

# -*- coding: utf-8 -*-
from flask import Flask, jsonify, request
from dao import TemaDAO, ModuloDAO
from entidades import Tema, Modulo

app = Flask(__name__)

dao_tema = TemaDAO()
dao_modulo = ModuloDAO()

@app.route('/tema')
def get_temas():
    temas = []
    for t in dao_tema.obterTodos():
        temas.append({"tema_id": t.tema_id,
                      "tema_nome": t.tema_nome})
    return jsonify(temas)

@app.route('/tema/')
def get_tema(tema_id):
    t = dao_tema.obter(tema_id);
    return jsonify({"tema_id": t.tema_id,
                  "tema_nome": t.tema_nome})

@app.route('/tema', methods=['POST'])
def add_tema():
    tema_json = request.get_json()
    tema = Tema(tema_id=tema_json['tema_id'],
               tema_nome=tema_json['tema_nome'])
    dao_tema.incluir(tema)
    return '', 204

@app.route('/tema/', methods=['DELETE'])
def del_tema(tema_id):
    dao_tema.excluir(tema_id)
    return '', 204

@app.route('/modulo/')
def get_modulos(tema_id):
    modulos = []
    for m in dao_modulo.obterTodosTema(tema_id):
        modulos.append({"modulo_id": m.modulo_id,
                      "modulo_nome": m.modulo_nome,
                      "tema_id": m.tema_id})
    return jsonify(modulos)

@app.route('/modulo/', methods=['POST'])
def add_modulo(tema_id):
    modulo_json = request.get_json()
    print(modulo_json)
    modulo = Modulo(modulo_id=modulo_json['modulo_id'],
                   modulo_nome=modulo_json['modulo_nome'],
                   tema_id=modulo_json['tema_id'])
    if modulo.tema_id != tema_id:
        raise Exception("URL inconsistente com novo modulo")
    dao_modulo.incluir(modulo)
    return '', 204

```

Inicialmente temos a definição da aplicação, com base em um objeto **Flask**, e as instâncias para **TemaDAO** e **ModuloDAO**. Os objetos DAO serão utilizados para intermediar as operações sobre o banco de dados, com retorno de entidades e coleções nos métodos de consulta.

No método **get_temas**, respondemos à rota **tema** no modo **GET**, ocorrendo uma conversão das entidades recuperadas via **obterTodos** para uma lista de dicionários, de forma a permitir o uso do método **jsonify** para a construção da resposta no formato JSON, com o conjunto de temas presentes na tabela. Temos o mesmo tipo de operação no método **get_tema**, agora recebendo o valor de **tema_id** na rota **parametrizada**, assim como em **get_modulos**, acessado por meio da rota **modulo**, também **parametrizada**, que retorna todos os módulos de um tema específico.



Comentário

Segundo o padrão REST, o modo POST está relacionado à inclusão, como pode ser observado nos métodos **add_tema** e **add_modulo**, com a obtenção dos dados por meio do método **get_json** de request, geração da instância de entidade necessária, e chamada para o método **incluir** do DAO correto. Ao final, temos um retorno vazio, com código 204, indicando o sucesso da inclusão, e no caso específico do módulo, a rota é parametrizada com o identificador do tema, ocorrendo uma exceção quando esse identificador não corresponde aos dados fornecidos para inclusão.

Finalmente, foi definido o método **del_tema**, recebendo o identificador do tema a partir da rota **parametrizada**, no modo **DELETE**, com uma simples chamada para o método de exclusão do DAO e retorno do indicador de **sucesso**, com código **204**, ao final. O servidor é iniciado por meio do comando apresentado a seguir, que precisa ser executado no ambiente do **Miniconda**, no diretório do projeto (**curso**).

plain-text

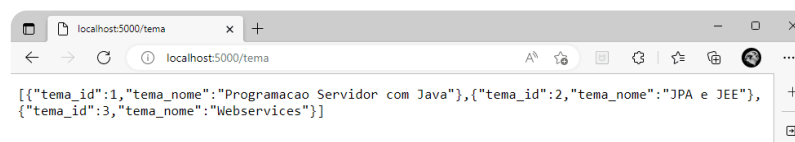
```
flask --app cursoapp run
```

A execução do comando pode ser observada a seguir.

```
Anaconda Powershell Prompt (miniconda3)
(spyder-env) PS C:\FlaskApps\curso> flask --app cursoapp run
* Serving Flask app 'cursoapp'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production
WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Execução do aplicativo Flask no Miniconda.

Nosso servidor responde, por padrão, na porta 5000, e um teste simples pode ser feito com o acesso ao endereço **http://localhost:5000/tema**, como pode ser observado na próxima imagem.



Teste da rota tema através do navegador.

Criação do cliente REST

Para acessar nossos web services RESTful, tudo que precisamos é a biblioteca **requests**, que deve ser instalada por meio do comando apresentado a seguir.

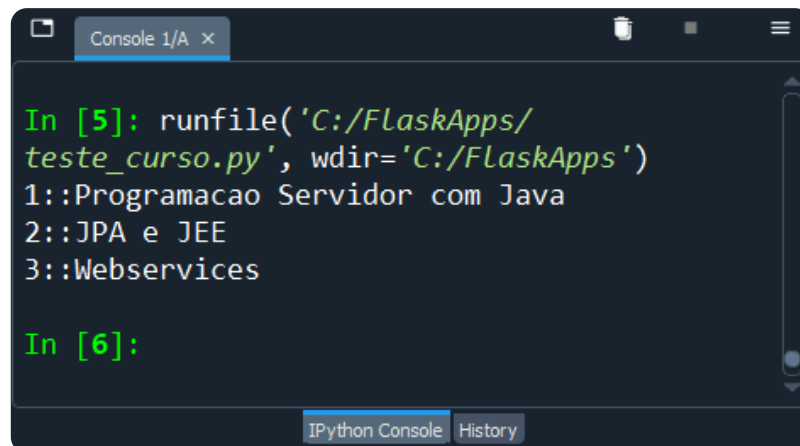

```
plain-text  
pip install requests
```

Vamos iniciar com um pequeno teste, criando o arquivo **teste_curso.py**, utilizando o conteúdo apresentado a seguir.

```
python  
  
# -*- coding: utf-8 -*-  
import requests  
import json  
  
request = requests.get("http://localhost:5000/tema")  
temas = json.loads(request.content)  
  
for t in temas:  
    print(str(t['tema_id'])+"::"+t['tema_nome'])
```

Recebemos no objeto **request** a chamada para **http://localhost:5000/tema**, no modo **GET**, com a extração dos temas a partir do conteúdo (**content**), via função **loads**, da biblioteca **json**, padrão para o Python. Os dados são recebidos na forma de uma lista de dicionários, e temos a impressão dos dados por meio de uma estrutura for.

O resultado da execução, no console interno do **Spyder**, pode ser observado a seguir.



```
Console 1/A x  
In [5]: runfile('C:/FlaskApps/  
teste_curso.py', wdir='C:/FlaskApps')  
1::Programacao Servidor com Java  
2::JPA e JEE  
3::Webservices  
  
In [6]:
```

Execução do cliente de teste.

Executando pequenas alterações em nosso projeto Django inicial, podemos alterar o acesso aos dados para nosso web service RESTful. Para não perder a versão original do projeto, podemos copiá-lo, do diretório **DisciplinasApp** para um novo diretório com o nome **DisciplinasAppREST**. Feita a cópia, vamos criar o arquivo **cliente.py** no diretório **DisciplinasApp**, interno ao diretório **DisciplinasAppREST**, que equivale ao pacote principal do projeto Django, utilizando o código da listagem apresentada a seguir.

```
python

# -*- coding: utf-8 -*-
import requests
import json

class Cliente:
    def buscar_temas(self):
        request = requests.get("http://localhost:5000/tema")
        return json.loads(request.content)
    def buscar_tema(self,tema_id):
        request = requests.get(f"http://localhost:5000/tema/{tema_id}")
        return json.loads(request.content)
    def inserir_tema(self,tema):
        request = requests.post("http://localhost:5000/tema",
                                json=tema)

        return request.ok
    def excluir_tema(self,tema_id):
        request = requests.delete(f"http://localhost:5000/tema/{tema_id}")
        return request.ok
    def buscar_modulos(self,tema_id):
        request = requests.get(f"http://localhost:5000/modulo/{tema_id}")
        return json.loads(request.content)
    def inserir_modulo(self,tema_id,modulo):
        request = requests.post(f"http://localhost:5000/modulo/{tema_id}",
                                json=modulo)

        return request.ok
```

Nossa classe **Cliente** concentra todas as chamadas ao web service, por meio de **requests**, como **buscar_temas**, com retorno de todos os temas, **buscar_tema**, retornando o tema identificado por **tema_id**, e **buscar_modulos**, que fornece os módulos relacionados ao **tema_id**. Como são métodos de **consulta**, utilizam a função **get**, e as rotas parametrizadas são definidas por meio da substituição de **tema_id** em uma **string formatada**.

Quanto aos métodos **inserir_tema** e **inserir_modulo**, eles utilizam a função **post**, relacionada ao modo POST do HTTP, com a passagem dos valores para as novas entidades no parâmetro **json**. Devemos observar que a inclusão de um módulo é feita por meio de uma rota parametrizada pelo identificador do tema.

Finalmente, o método **excluir_tema** executa no modo DELETE do HTTP, por meio da chamada para a função **delete**, na rota parametrizada pelo identificador do tema. Note que as funções de inserção e exclusão sempre retornam o atributo **ok** da requisição que, na verdade, é o indicador de sucesso no tratamento da mesma pelo servidor REST.

Agora vamos alterar o arquivo **views.py**, que deverá estar no mesmo diretório de **cliente.py**, para o conteúdo da listagem seguinte.

```
python

# -*- coding: utf-8 -*-
from DisciplinasApp.cliente import Cliente
from django.shortcuts import render
from django.forms import Form, ChoiceField

class ModulosForm(Form):
    tema = ChoiceField(label='Tema', choices=[])

def index(request):
    clienteREST = Cliente()
    temas = clienteREST.buscar_temas()
    context = {
        'temas': [x['tema_nome'] for x in temas]
    }
    return render(request, 'index.html', context=context)

def modulos(request):
    clienteREST = Cliente()
    if request.method == 'POST':
        tema_id = request.POST['tema']
        modulos = clienteREST.buscar_modulos(tema_id)
        tema = clienteREST.buscar_tema(tema_id)
        context = {'modulos': [x['modulo_nome'] for x in modulos],
                   'tema': tema['tema_nome']}
        return render(request, 'modulos_lista.html', context)
    else:
        form = ModulosForm()
        form.fields['tema'].choices = [
            tuple([x['tema_id'], x['tema_nome']])
            for x in clienteREST.buscar_temas()
        ]
        return render(request, 'modulos_form.html', {'form': form})
```

Na função **index**, instanciamos um **cliente** para o servidor REST, e utilizamos os temas obtidos na chamada para **buscar_temas** na construção da lista de temas fornecidos no contexto. Ocorre uma transformação, com a extração do valor de **tema_nome** para cada tema recebido, e o resto do processo ocorre da forma original, com a passagem para o template **index.html**.

Com relação à modificação efetuada na função **modulos**, também incluímos um cliente REST para lidar com os dados, ocorrendo a alimentação do campo de seleção do formulário, no modo **GET**, com base nos campos **tema_id** e **tema_nome** dos temas, obtidos via **buscar_temas**. Já no modo **POST**, agora recebendo **tema_id** por meio do parâmetro **tema** da requisição, recuperamos o tema e os módulos associados, a partir das chamadas **buscar_tema** e **buscar_modulos**, ambas utilizando o valor de **tema_id**, e alimentamos os dados do contexto com os valores obtidos.

Com essas alterações pontuais, podemos executar o aplicativo **DisciplinasAppREST**, tendo a certeza de obter os mesmos resultados da versão original, mas agora com os dados obtidos a partir do servidor REST, que, por sua vez, acessa o PostgreSQL para recuperação dos valores. Note que o comando de execução não muda, mas deve ser digitado no diretório do novo projeto.

```
Anaconda Prompt (miniconda3) - python manage.py runserver 8080
(spyder-env) C:\DjangoApps\DisciplinasAppREST>python manage.py runserver 8080
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the
migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
December 11, 2022 - 03:03:06
Django version 4.1.3, using settings 'DisciplinasApp.settings'
Starting development server at http://127.0.0.1:8080/
Quit the server with CTRL-BREAK.
```

Execução do novo projeto Django no Miniconda.

Empregando web services no mundo real

Bibliotecas requests e JSON

Neste vídeo, apresentaremos os principais assuntos em relação a criação do cliente REST.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Web services e sua utilização no 'mundo real'

As aplicações para fornecimento e consumo de web services, tanto **SOAP** quanto **REST**, que criamos ao longo deste conteúdo, são compostas por códigos funcionais, que utilizam diversos recursos da linguagem Python e que podem ser adaptados para utilização em projetos, tanto voltados para fins de estudo quanto para fins profissionais, em aplicações reais.

Com base nessa premissa, discutiremos, a partir de agora, em quais situações do dia a dia podemos fazer uso de web services. Nesse sentido, veremos alguns casos de uso mais simples, e que você poderá utilizar caso esteja iniciando na área de desenvolvimento – e outros mais avançados – para você que já tem alguma experiência no desenvolvimento de aplicações. Para começar, vamos recapitular alguns conceitos já vistos e, em seguida, discutiremos os casos de uso em si.

Recapitulando conceitos

Web services

Um web service é um aplicativo projetado para suportar interoperabilidade entre máquinas por meio de uma rede (W3C, 2004). Logo, os web services são agentes de software cuja função é trocar mensagens, possuindo, para isso, um conjunto de funcionalidades definidas.

Partindo dessa definição, podemos ver que as aplicações que criamos atendem às funcionalidades nelas mencionadas, ou seja, trata-se de aplicações, agentes de software, com a função de disponibilizar serviços – os provedores SOAP e REST – e consumir serviços – os clientes SOAP e REST.

Software

Quando falamos de aplicações, estamos falando de softwares. Em relação a estes, os mesmos podem ser categorizados em alguns tipos, como:

- Softwares de sistema
- Software de aplicação
- Software científico e de engenharia
- Software embutido

- Software para linhas de produtos
- Software para web
- Software de inteligência artificial
- Outros

Dentre essas categorias, cabe ressaltar a definição para o software de aplicação: Segundo Pressman (2016), são programas desenvolvidos para execução em um negócio específico ou em uma empresa específica. Essa definição cabe bem dentro das aplicações que desenvolvemos, uma vez que abordamos um tema específico, voltado para a área acadêmica e no qual manuseamos algumas classes como "Tema" e "Modulo", naturais do negócio em si.

API

Uma interface de programação de aplicação, segundo Jacobson (2012), é uma maneira de duas aplicações de computador se comunicarem, uma com a outra, em uma rede (predominantemente a internet) usando uma linguagem comum entendida por ambas. As APIs seguem uma especificação e isso implica em:

Definição

O provedor da API descreve exatamente qual funcionalidade a API oferecerá.

Disponibilidade

O provedor da API descreve quando e como a funcionalidade estará disponível.

Restrições

O provedor da API pode estabelecer restrições técnicas, legais ou comerciais, como limites de utilização, entre outras.

Responsabilidades

Um contrato deve ser estabelecido entre o provedor da API e quem dela faz uso, ficando o último comprometido a seguir as regras de uso estabelecidas pelo provedor.

Unindo conceitos

Os três conceitos apresentados são de suma importância para abordarmos o uso de web services no mundo real, uma vez que se tratam de conceitos intrinsecamente relacionados, como podemos perceber nas suas respectivas definições. Logo, é comum, no dia a dia, confundirmos os conceitos de web services e APIs, por exemplo. Frente a isso, cabe destacar algumas de suas diferenças:

- Podemos dizer que todos os web services são APIs, mas nem todas APIs são web services.
- Uma API pode usar outras formas de comunicação, além das utilizadas pelos web services (SOAP, REST e XML-RPC).
- Uma API, diferentemente do web service, não precisa de uma rede para funcionar.

Utilizando web services no mundo real

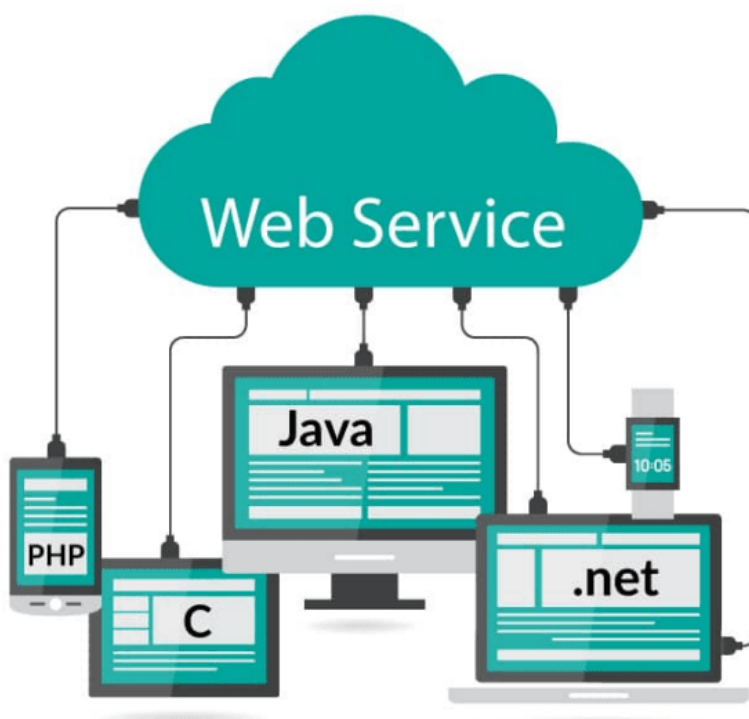
Tomando como base as aplicações que criamos anteriormente, podemos pensar em algumas formas de como utilizá-las no mundo real. Por exemplo: poderíamos ter um website que exibisse informações sobre os cursos oferecidos por uma instituição de ensino. Nesse site, dentre as funcionalidades disponíveis, poderia haver uma para listar todos os cursos e respectivas disciplinas, assim como o conteúdo de cada uma delas.



Atenção

Nesse caso, poderíamos consumir o web service desenvolvido (em SOAP ou REST) tanto a partir do "server side" – carregando as informações no carregamento da página – quanto a partir do "client side" – a partir de formulários de pesquisa e filtro de resultados carregando informações através de AJAX.

Considerando o exemplo, você poderia questionar sobre o motivo de utilizarmos web services para exibir o conteúdo em questão, uma vez que poderíamos utilizar outras técnicas, como consultas diretas ao banco de dados ou até mesmo páginas estáticas, sem o uso de web services.



Consumo de Web services utilizando linguagens Server Side.

Nesse caso, a justificativa para a utilização de web services está em podermos disponibilizar o conteúdo em questão para outras aplicações além do website. Por meio deles poderíamos alimentar um aplicativo mobile, assim como fornecer informações para um sistema de terceiros, como um portal que reúna informações dos cursos de diversas instituições de ensino, por exemplo. Logo, a escolha por uma arquitetura baseada em web services deve levar esses fatores em conta.

Imaginando agora outra situação real e um pouco mais avançada, na qual se aplique a utilização de web services, temos a arquitetura de **microserviços**, uma abordagem de desenvolvimento de softwares que prega a decomposição de aplicações em uma gama de serviços, os quais serão disponibilizados por meio de uma interface de APIs.

Em comparação com a abordagem tradicional de desenvolvimento de aplicações, normalmente monolíticas, ou seja, todos os módulos e funcionalidades em um bloco único, a abordagem baseada em microserviços prega que as aplicações sejam desmembradas em componentes mínimos e independentes que, embora separados, trabalhem juntos para realizarem as mesmas tarefas. Entre as vantagens da abordagem baseada em microserviços, temos:

Compartilhamento

Capacidade de compartilhar de processos e funções semelhantes entre várias aplicações.

Escalabilidade

Maior flexibilidade para acréscimo de novas funcionalidades.

Disponibilidade

Com a aplicação não sendo formada por um único bloco, diminui o risco de indisponibilidade total dela.

Além das vantagens descritas, e agora com foco no processo de desenvolvimento, há outras vantagens como:

- Mais facilidade na criação de testes unitários
- Maior frequência de deploy e facilidade para implementação da entrega contínua.
- Otimização no monitoramento e identificação de erros.

Exemplos de APIs comumente utilizadas

A seguir, são listados alguns exemplos de APIs comumente utilizadas nas mais diversas aplicações no dia a dia:

APIs de pagamento

Serviços que fornecem meios de pagamento e contemplam toda a segurança relacionada a esse processo, além de também oferecerem serviços extras como identificação de fraudes, entre outros.

APIs de redes sociais

A maioria das redes sociais fornecem APIs públicas para o consumo de suas informações, além da realização de outras ações, como login baseado nas credenciais da rede social, publicações e compartilhamentos.

APIs de localização

Um bom exemplo desse tipo de API é o Google Maps.

Outros exemplos

Consulta de CEP; consulta de previsão de tempo; conversão de moedas; serviços de câmbio; serviços de plataformas de e-commerce; chatbots etc.

Verificando o aprendizado

Questão 1

Sobre a arquitetura do provedor e cliente REST estudada, selecione a alternativa correta.

A

A arquitetura estudada é uma entre muitas possíveis de serem aplicadas para a criação de provedores e clientes de web services REST.

B

A especificação REST determina que a arquitetura do provedor REST utilize, obrigatoriamente, dados provenientes de um banco PostgreSQL.

C

O consumo de serviços REST, em Python, é restrito a aplicações do tipo web.

D

Segundo a especificação REST, é necessário instalar um servidor para armazenar tanto o provedor quanto o Cliente REST.

E

Como REST é uma arquitetura simples, que preza pela leveza, não é possível consumir dados complexos, dados que estejam, por exemplo, armazenados em mais de duas diferentes tabelas de bancos de dados.



A alternativa A está correta.

Não existe uma arquitetura padrão, em termos de linguagem de programação e seus recursos específicos, definida pela especificação REST para o fornecimento e consumo de web services REST.

Questão 2

Estudamos a sintaxe, a anatomia de serviços escritos e consumidor na arquitetura REST. Nesse contexto, selecione a afirmativa correta.

A

Os serviços disponíveis em uma arquitetura REST são caracterizados por possuírem uma única URI de acesso comum.

B

Um recurso, ou método, consumível na arquitetura REST é identificado pela sua URI, ou seja, pelo endereço do recurso, onde informamos o nome do método e os seus parâmetros, quando necessários.

C

Por ser uma arquitetura mais simples, quando comparada ao SOAP, por exemplo, é possível, ao utilizarmos REST, definirmos novos métodos no lado cliente, mesmo que esses não existam no provedor.

D

A única limitação existente na arquitetura REST é a que obriga que todos os métodos disponíveis utilizem o mesmo protocolo HTTP.

E

A arquitetura REST permite que novos protocolos de transmissão, diferentes dos fornecidos pelo protocolo HTTP, sejam criados no provedor e utilizados pelo cliente na transmissão dos dados das aplicações.



A alternativa B está correta.

Um serviço REST pode conter inúmeros métodos, cada um com sua própria URI. Além disso, é possível utilizar diferentes métodos HTTP nos diferentes métodos existentes.

Diferenciando bibliotecas, APIs e frameworks

Neste vídeo, apresentaremos os principais assuntos em relação às bibliotecas, APIs e Frameworks.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Em sistemas computacionais, temos um grande conjunto de estruturas de dados, funções e procedimentos, que podem ser organizados em módulos, disponibilizando operações para algum domínio de utilização específico. Esses módulos, denominados **bibliotecas**, constituem o ferramental de reuso mais comum no mundo da programação.

São muitos os exemplos de bibliotecas no mercado, para as mais diversas plataformas, como **JQuery**, amplamente utilizada na construção de front-end para web, **Retrofit**, que facilita a construção de clientes Java ou Kotlin para web services, e **Pandas**, para análise de dados no ambiente do Python, apenas para citar alguns exemplos.

Os **frameworks**, por sua vez, definem soluções completas, customizáveis para determinados domínios de utilização e, ao contrário das bibliotecas, que apenas disponibilizam os recursos de forma passiva, controlam a maioria das operações necessárias, permitindo que o programador se preocupe apenas com os aspectos específicos do negócio. Por exemplo, com a utilização do **Angular**, temos a construção de front-end do tipo SPA (single page application) em Type Script, com ótimos recursos de vinculação de dados, enquanto o **Flask** permite criar web services baseados em funções do Python, com as rotas configuradas por meio de anotações.



Comentário

Com relação ao conceito de API, pode ser definido como a exposição de um conjunto de funcionalidades de um determinado sistema para ferramentas externas. Um bom exemplo é a API de um sistema operacional, que viabiliza a programação com múltiplas threads em programas criados na linguagem C, bem como a automatização do uso de Word e Excel, ambos produtos da Microsoft, com base em Java Script.

Atualmente os **web services** constituem um dos melhores exemplos de API, pois permitem a definição de conjuntos de serviços, com base em formatos de texto interoperáveis, enquanto sustentam complexos sistemas, ocultos do ambiente exterior. Logo, podem ser considerados como interfaces para programação de aplicativos, já que as chamadas disponibilizadas ativarão diversos processos internos para a conclusão das operações relacionadas.

Aumentando a produtividade

Injeção de dependências e inversão de controle

Na definição de um framework, **injeção de dependências** e **inversão de controle** são recursos que permitem grande aumento da produtividade na construção de sistemas. Eles fornecem um meio simples para configurar o relacionamento das classes com o framework e delegar atividades para execução no núcleo comum do ambiente.

Ao trabalhar com bibliotecas, elas são importadas e efetuamos as chamadas para as funções necessárias no fluxo de execução do programa, mas quando usamos um framework, temos um container que se responsabiliza pela execução de boa parte do processo, o que é definido como **inversão de controle**. Uma grande vantagem dessa abordagem é o **baixo acoplamento**, já que o container pode ser substituído facilmente, sem causar impacto no código do sistema, além de permitir que o programador se preocupe exclusivamente com a lógica do negócio.

Outra funcionalidade oferecida pelos frameworks é a inclusão de recursos do container, como serviços e conexões com bancos de dados, nos objetos do sistema, com base em metodologias simples, o que é conhecido como **injeção de dependência**. Exemplos são a injeção de serviços nas classes Type Script do Angular, a partir dos parâmetros do construtor, ou a captura de uma requisição HTTP na função de tratamento, quando usamos o **Django**, como ocorre na função da listagem seguinte, onde **request** é alimentado (injetado) pelo framework.

```
python

def index(request):
    clienteREST = Cliente()
    temas = clienteREST.buscar_temas()
    context = {
        'temas': [x['tema_nome'] for x in temas]
    }
    return render(request, 'index.html', context=context)
```

Agora vamos construir uma **API** no estilo **REST**, com base no **Flask**, semelhante aos exemplos que foram implementados anteriormente, onde a inversão de controle é utilizada na definição de rotas, por meio de **anotações**, também conhecidas como **decorações**. Como as ferramentas já foram instaladas em passos anteriores, vamos criar o diretório **tarefasAPI**, e iniciar a codificação de nosso projeto imediatamente, com todos os arquivos sendo gerados nesse diretório.

O primeiro arquivo do projeto receberá o nome **tarefa.py**, e vai representar o formato de dados de nossa API, com base em uma classe denominada **Tarefa**, definindo uma tarefa genérica, como pode ser observado na listagem seguinte. Nossa classe terá apenas os atributos para definição de **código**, **título** e **descrição**, como pode ser observado na listagem apresentada a seguir.

```
python

from sqlalchemy import Column
from sqlalchemy import String
from sqlalchemy.orm import declarative_base

Base = declarative_base()
class Tarefa(Base):
    __tablename__ = "tarefa"
    codigo = Column(String(4), primary_key=True)
    titulo = Column(String(50))
    descricao = Column(String(255))
```

Utilizamos o **SQLAlchemy** para definir uma entidade **Tarefa**, com persistência em uma tabela de mesmo nome, composta pelas colunas **codigo**, **titulo** e **descricao**, todas do tipo texto, sendo utilizado o código como chave primária. A adoção do modelo declarativo torna muito simples e intuitivo o mapeamento objeto-relacional, como já vimos em exemplos anteriores.

Para utilizar o **PostgreSQL**, vamos criar o arquivo **corsodb.py**, semelhante ao que foi criado para o exemplo de cursos, mas agora com a criação da tabela, caso ainda não exista. A listagem com o código modificado é apresentada a seguir.

```
python

from sqlalchemy import create_engine, Table, Column
from sqlalchemy import MetaData, String

postgresql_engine = create_engine(
    "postgresql://postgres:admin123@127.0.0.1/curso",
    pool_reset_on_return=None,
)
metadata = MetaData(bind=postgresql_engine)
tb_tarefas = Table('tarefa', metadata,
    Column('codigo', String(4), primary_key=True),
    Column('titulo', String(50), nullable=False),
    Column('descricao', String(255))
)
metadata.create_all()
```

A novidade aqui é a criação da tabela no banco de dados, e para isso temos a definição de um objeto do tipo **Table**, denominado **tb_tarefa**, com o nome da tabela e seus campos. Além desse objeto, temos um objeto do tipo **MetaData**, associado à conexão com o PostgreSQL, e utilizado na definição de **tb_tarefa**. Com a chamada para o método **create_all**, todos os objetos do tipo **Table** que se ligaram à **metadata** serão verificados, sendo criadas as tabelas quando necessário.

Já podemos completar nossa camada **Model**, com a criação do arquivo **tarefaDAO.py**, onde será utilizado o código da listagem seguinte.

```
python

from sqlalchemy import select
from cursodb import postgresql_engine
from tarefa import Tarefa
from sqlalchemy.orm import Session

session = Session(postgresql_engine)

class TarefaDAO:
    def obterTodos(self):
        stmt = select(Tarefa)
        return session.scalars(stmt)
    def obter(self, codigo):
        tarefa = session.get(Tarefa, codigo)
        return tarefa
    def incluir(self, tarefa):
        session.add(tarefa)
        session.commit()
    def excluir(self, codigo):
        session.delete(self.obter(codigo))
        session.commit()
    def alterar(self, tarefa):
        session.merge(tarefa)
        session.commit()
```

A classe **TarefaDAO** utiliza o objeto do tipo **Session**, inicializado com base na engine associada ao PostgreSQL, para todas as tarefas de consulta e persistência. Por exemplo, para a consulta de todas as tarefas (**obterTodos**), utilizamos **scalars**, com base em uma chamada **select**, enquanto para a consulta pela chave primária (**obter**), efetuamos uma chamada para **get**, com a passagem do código fornecido por meio do parâmetro.

Com relação às operações de persistência, temos a inclusão baseada no método **add**, exclusão efetuada via **delete** e alteração por meio de **merge**. Em todos os casos é necessário confirmar a operação por meio de uma chamada para **commit**.

Já podemos definir nossa **API** do tipo **REST**, com base no **Flask**, criando o arquivo **tarefasAPI.py**, onde será utilizado o código da listagem seguinte.

```
python

from flask import Flask, jsonify, request
from tarefaDAO import TarefaDAO
from tarefa import Tarefa

app = Flask(__name__)
dao_tarefa = TarefaDAO()

@app.route('/tarefa')
def get_tarefas():
    tarefas = []
    for t in dao_tarefa.obterTodos():
        tarefas.append({'codigo': t.codigo,
                        'titulo': t.titulo,
                        'descricao': t.descricao})
    return jsonify(tarefas)

@app.route('/tarefa', methods=['POST'])
def add_tarefa():
    tarefa_json = request.get_json()
    tarefa = Tarefa(codigo=tarefa_json['codigo'],
                    titulo=tarefa_json['titulo'],
                    descricao=tarefa_json['descricao'])
    dao_tarefa.incluir(tarefa)
    return '', 204

@app.route('/tarefa/', methods=['DELETE'])
def del_tarefa(codigo):
    dao_tarefa.excluir(codigo)
    return '', 204
```

Em termos práticos, temos apenas funções comuns, com anotações indicando a rota e o método HTTP utilizado, o que permitirá a **inversão de controle** pelo framework **Flask**. Nosso repositório é baseado em uma tabela do PostgreSQL, acessada por meio do DAO, sendo definidas as funções **obterTarefas**, para retorno das tarefas em uma lista, **incluirTarefa**, para adicionar uma tarefa a partir dos dados da requisição, e **excluirTarefa**, para remoção a partir do código.

Para incluir uma tarefa, por meio do método **add_tarefa**, que responde no modo **POST**, obtemos a informação via **get_json**, transformamos em um objeto do tipo **Tarefa**, e acionamos o método de inclusão do DAO.



Atenção

Com relação ao método **del_tarefa**, que utiliza o modo **DELETE**, a exclusão da tarefa ocorre com base no código que é fornecido por meio da rota parametrizada. Em ambos os casos respondemos ao solicitante com o código 204 do protocolo HTTP, indicando sucesso.

Devemos observar que a consulta exige a transformação dos dados recebidos a partir do DAO em uma lista de dicionários, viabilizando o retorno no formato **JSON**, com uma chamada para a função **jsonify**, tendo a lista transformada como parâmetro.

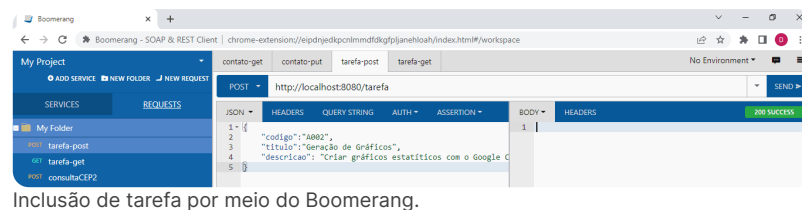
Já podemos executar nossa API, no console do **Miniconda**, por meio do comando seguinte, que deve ser digitado no diretório do projeto.

```
plain-text  
flask --app tarefasAPI.py run --port 8080
```

Com a simples execução de nosso servidor teremos a criação da tabela **tarefa** no banco de dados **curso**, o que poderá ser verificado por meio do **pgAdmin**. Devido ao uso do objeto de metadados, no arquivo **corsodb.py**, a tabela é criada sempre que não for encontrada no PostgreSQL.

Ao trabalhar com uma API, não temos uma interface de usuário constituída, mas apenas um conjunto de serviços disponibilizados para outras plataformas. Para testar as APIs do tipo REST, sem a criação de um aplicativo cliente, podemos utilizar o Postman ou o plugin **Boomerang**, do navegador **Chrome**, sendo este último escolhido pela praticidade.

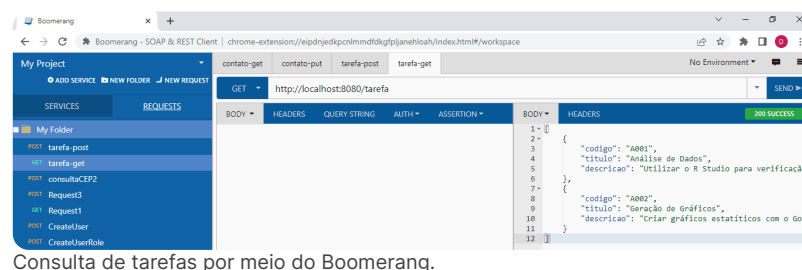
Inicialmente devemos efetuar a inclusão de algumas tarefas no repositório, com a criação de uma requisição pela opção **New Request**, escolha do endereço **http://localhost:8080/tarefa**, no modo **Post**, e modificação do formato de Body para **JSON**. Os dados da nova tarefa devem ser digitados e, com o clique na opção **Send**, o código **200** do HTTP indicará sucesso.



Os dados de cada tarefa devem ser fornecidos no formato JSON, como no exemplo da listagem seguinte, sendo necessário o fornecimento dos valores para todos os atributos da classe.

```
python  
{  
    "codigo": "A002",  
    "titulo": "Geração de Gráficos",  
    "descricao": "Criar gráficos estatísticos com o Google Charts"  
}
```

Após inserir algumas tarefas, repetindo o processo anterior com a modificação dos dados que serão enviados, vamos criar uma requisição para o mesmo endereço, em modo **Get**, e corpo vazio. Ao clicar em **Send**, devemos ter, além do indicador de sucesso, o retorno de todas as tarefas enviadas, na forma de um vetor JSON.



Criando clientes Python para web services Rest

Interoperabilidade

Em meados da década de 1990, com a expansão dos sistemas computacionais, começaram a ser oferecidas diferentes formas padronizadas para armazenar, recuperar e processar dados. Com base em **conversores** de dados, os arquivos de um determinado fabricante eram convertidos para um determinado formato, de forma que outro fabricante pudesse ler. Esse processo surgiu da necessidade de **compartilhamento** de dados entre aplicativos distintos.

Ao longo do tempo, diferentes formas de compartilhamento de informação surgiram, tendo como elementos relevantes os **bancos de dados** e **protocolos de rede** padronizados. Bancos de dados como MySQL e Oracle podem ser acessados por plataformas distintas simultaneamente, e protocolos como RPC (remote procedure call) viabilizaram a execução de procedimentos em máquinas remotas, não importando a plataforma do cliente.

Surge o conceito de **interoperabilidade**, que define a capacidade de um sistema oferecer recursos e serviços para outros sistemas, com independência de plataforma, baseada em meios padronizados de comunicação.

A criação de uma API deve ser guiada pela interoperabilidade, já que o objetivo é oferecer funcionalidades para clientes distintos a partir do sistema existente.

Atualmente a interoperabilidade é garantida pela utilização de formatos de dados padronizados, como **XML** e **JSON**, pois adotam modo texto, podendo ser processados facilmente em qualquer plataforma cliente. Nossos **web services**, tanto SOAP quanto RESTful, definem APIs com serviços baseados nessa estratégia de interoperabilidade, muito adequada ao ambiente de rede, onde temos grande heterogeneidade e a necessidade de transparência frente aos firewalls.

Clientes Python para web services REST

O uso de métodos como **PUT** e **DELETE**, além da adoção de **JSON** para os dados, pode dificultar testes tradicionais, baseados em formulários HTML, mas existem muitas soluções viáveis, como o uso do aplicativo Postman, ou de chamadas AJAX a partir de bibliotecas JQuery. No entanto, a forma mais eficaz para verificar a funcionalidade da solução é a criação de um aplicativo Python que se comunique com nossa API.



Comentário

Bibliotecas como Axios, no ambiente NodeJS, e Retrofit, para a plataforma Java, permitem o uso de todos os métodos do HTTP de forma simplificada. No ambiente do Python, as funções de requests efetuam o acesso via HTTP, enquanto a biblioteca json permite efetuar o mapeamento dos dados no formato JSON para listas de dicionários.

Vamos iniciar a definição do cliente com a criação do diretório **tarefasClient**, onde ficarão todos os arquivos de código-fonte. No diretório tarefasClient incluiremos o arquivo **cliente.py**, com o código apresentado a seguir.

```
python

import requests
import json

class Tarefa:
    def __init__(self, dados):
        self.codigo = dados['codigo']
        self.titulo = dados['titulo']
        self.descricao = dados['descricao']
    def json(self):
        return {'codigo': self.codigo,
                'titulo': self.titulo,
                'descricao': self.descricao}

class ClienteTarefa:
    def __init__(self):
        self.baseURL = 'http://localhost:8080/tarefa'
    def obterTarefas(self):
        tarefas = []
        retorno = requests.get(self.baseURL)
        for t in json.loads(retorno.content):
            tarefas.append(Tarefa(t))
        return tarefas
    def incluirTarefa(self, tarefa):
        requests.post(self.baseURL, json=tarefa.json())
    def excluirTarefa(self, codigo):
        requests.delete(f'{self.baseURL}/{codigo}')
```

A primeira classe do arquivo (**Tarefa**) é apenas um modelo de dados que permitirá trabalhar de acordo com o paradigma orientado a objetos. Ela apresenta um **construtor** que recebe um **dicionário** e associa os valores aos atributos do objeto, além de um método **json**, que retorna os valores dos atributos na forma de um dicionário.

Em seguida, temos a classe **ClienteTarefa**, que representa o cliente de nossa API, com o atributo **baseURL** apontando para o endereço **http://localhost:8080/tarefa**. Os métodos da classe utilizarão esse atributo como endereço de base nas chamadas efetuadas via biblioteca **requests**. Veja a seguir mais informações sobre outros métodos.

obterTarefas

Neste método, efetuamos uma requisição **GET** para o endereço de base, recebendo os dados na forma de uma **lista de dicionários**, e transformamos o conteúdo recebido em uma lista de objetos do tipo **Tarefa**, a qual é retornada ao final.

incluirTarefa

Já neste método, o mesmo endereço é utilizado, agora no modo **POST**, com a passagem de uma tarefa no formato **JSON**, o que causará a inclusão da tarefa ao nível do servidor.

excluirTarefa

Neste método, finalmente, o endereço de base é combinado com o **código** da tarefa, em uma chamada do tipo **DELETE**, para que o registro seja removido no servidor.

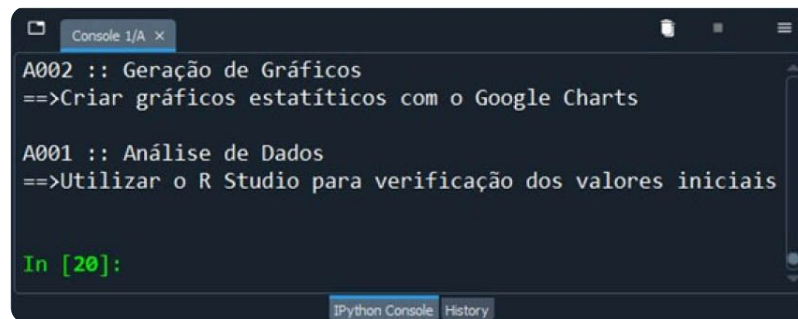
Para testar nosso cliente, vamos criar o arquivo **principal.py**, com o código da listagem seguinte.

```
python

# -*- coding: utf-8 -*-
from cliente import ClienteTarefa

cli = ClienteTarefa()
for t in cli.obterTarefas():
    print(f'{t.codigo} :: {t.titulo}');
    print(f'==>{t.descricao}');
    print();
```

Podemos observar a simplicidade do código, com uma chamada para **obterTarefas**, e a iteração na lista de objetos do tipo **Tarefa**, com a impressão formatada de seus dados. Considerando que nosso servidor ainda esteja em execução, com os dados adicionados por meio do Boomerang, podemos executar o novo arquivo, obtendo uma saída similar à da próxima imagem.



Execução do teste para o cliente REST no console interno do Spyder.

O encapsulamento das chamadas em uma classe permite essa visão simplificada do uso da API, com a execução dos métodos dando a impressão de simples chamadas locais, enquanto temos requisições HTTP na comunicação entre o cliente e o servidor. Criando uma base de clientes para múltiplas APIs, nossos sistemas poderão contar com serviços diversos, como a consulta de CEP, ou lista de projetos do GitHub, apenas para citar alguns exemplos, os quais serão incluídos de forma direta, demonstrando o grande nível de reuso na adoção da metodologia.

Continuando a análise da utilização de nosso cliente REST, para incluir uma tarefa teríamos um trecho de código similar ao que é apresentado a seguir.

```
python

cli = ClienteTarefa()
novaTarefa = Tarefa({'codigo':'T001',
                    'titulo':'Apenas Teste',
                    'descricao':'Este e um teste'})
cli.incluirTarefa(novaTarefa)
```

A exclusão seria ainda mais simples, como podemos observar a seguir.

```
python

cli = ClienteTarefa()
cli.excluirTarefa('T001')
```

Analisando os exemplos, chegamos à conclusão de que o uso de **requests** permite a criação de clientes **REST** de forma simples, abstraindo completamente de detalhes como o conhecimento acerca do protocolo HTTP, bem como a biblioteca **json** facilita o processo de conversão entre o formato JSON e as estruturas do Python.

Criação de API REST e cliente com Python

Neste vídeo, veja a construção de aplicativo cliente-servidor, adotando a arquitetura REST, com base em Flask, requests e biblioteca JSON.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Verificando o aprendizado

Questão 1

Por meio do Flask podemos definir uma API REST de forma simples, sem a necessidade de um servidor, como Apache, JBoss ou Payara. Por meio de anotações, a implementação de toda a funcionalidade relacionada à exposição de serviços é delegada para o framework, segundo uma técnica denominada

A

interoperabilidade.

B

inversão de controle.

C

polimorfismo.

D

sobrecarga.

E

injeção de dependência.



A alternativa B está correta.

Ao trabalhar com um framework, temos um container que se responsabiliza pela execução de boa parte do processo, o que é denominado Inversão de Controle. Podemos ainda utilizar algum recurso do container na programação, por meio da injeção de dependência.

Questão 2

A biblioteca requests permite a criação de clientes para o protocolo HTTP, incluindo APIs do tipo REST, de forma muito simples, utilizando funções equivalentes aos nomes dos métodos do protocolo HTTP. Como o corpo da requisição deve ser preenchido para o envio dos dados?

A

Concatenando o valor na URL, por meio de uma string formatada.

B

Utilizando a função delete.

C

Definindo um envelope na sintaxe XML.

D

Por meio do parâmetro JSON da função.

E

Preenchendo o elemento header.



A alternativa D está correta.

As funções get, post, put e delete definem o tipo de método HTTP utilizado, com a especificação da URL no primeiro parâmetro. No caso do envio de dados, como na inclusão via método POST, para a arquitetura REST, esses dados devem ser enviados por meio de um segundo parâmetro, com o nome json.

Considerações finais

Ao longo deste conteúdo, descrevemos os conceitos teóricos relativos aos web services e, especificamente, ao protocolo SOAP e à arquitetura REST. Além disso, esses conceitos foram aplicados, de forma prática, na construção de aplicações provedoras e consumidoras (clientes) de serviços nas tecnologias em questão, permitindo, assim, ao aluno empregar o conhecimento adquirido.

Também discorremos sobre a aplicabilidade de web services como agentes de software em cenários do "mundo real", e finalizamos com os conceitos relacionados à definição e utilização de APIs, levando para a prática por meio da construção de uma API REST baseada no servidor minimalista oferecido pelo framework Flask.

Podcast

Ouçá agora sobre a importância do uso de web services para a troca de dados entre as aplicações.



Conteúdo interativo

Acesse a versão digital para ouvir o áudio.

Explore +

Pesquise os diversos padrões definidos pela W3C, incluindo definições arquiteturais para Web services SOAP (WS-*) no site da W3C.

Verifique o artigo **Como criar um aplicativo Web usando o Flask em Python 3**, de Abdelhadi Dyourri, oferecido pela Digital Ocean, que aborda detalhes da implementação com Flask.

Acesse a **Documentação Oficial do Flask**, e conheça os detalhes de sua API, além de diferentes formas de utilização do framework.

Acesse a **Documentação Oficial do Django**, com alguns tutoriais e detalhes acerca da utilização do framework.

Acesse a **Documentação Oficial do PostgreSQL**, com uma abordagem completa acerca do banco de dados e da sintaxe SQL específica.

Referências

BLOKDYK, G. **SOAP Simple Object Access Protocol - The Ultimate Step-By-Step Guide**. Australia: 5StarCooks, 2018.

ELMAN, J.; LAVIN, M. **Django Essencial: usando REST, Web sockets e backbone**. São Paulo: Novatec, 2015.

GRINBERG, M. **Desenvolvimento Web com Flask**. São Paulo: Novatec, 2018.

JACOBSON, D.; BRAIL, G.; WOODS, D. **APIs**: a strategy guide. USA: O'Reilly, 2012.

MARRS, T. **JSON at work**. USA: O'Reilly, 2017.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software**: uma abordagem profissional. Porto Alegre: McGraw-Hill – Artmed, 2016.

W3C. **W3C working group note 11**. W3C – Web services Architecture. Publicado em: fev. 2014. Consultado na Internet em: 10 jan. 2023.