Server-Side – API REST

Católica de Santa Catarina – Centro Universitário
Jaraguá do Sul - SC, Brasil

Professor: Ph.D. Andrei Carniel (Prof. Andrei)
Contato: andrei.carniel@gmail.com
linktr.ee/andrei.carniel

Revisão - API

- API é a sigla em inglês para *Application Programming Interface*, ou interface de programação de aplicações.

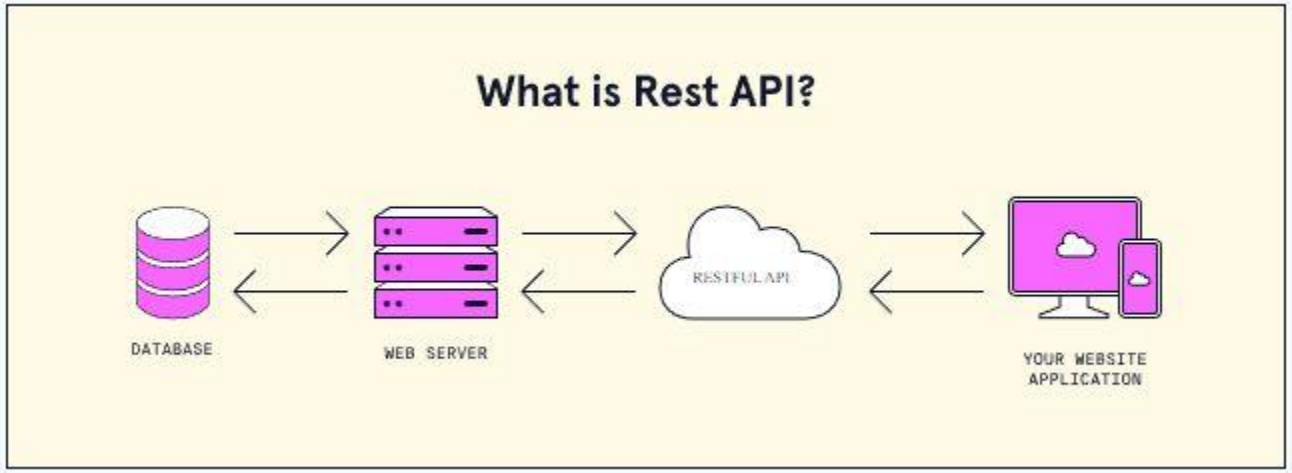
- APIs são conjuntos de regras, protocolos e ferramentas que permitem que diferentes aplicativos se comuniquem entre si.

- Em outras palavras, as APIs são interfaces que permitem que dois softwares se integrem e se comuniquem de maneira padronizada e organizada.

Revisão - REST

- REpresentational State Transfer (REST).

- Veremos agora sobre o paradigma REST e como essa arquitetura agiliza a comunicação entre os componentes da web.



Definição - REST

- REST é um estilo de arquitetura que visa fornecer padrões entre sistemas na web, facilitando assim a comunicação entre os sistemas.

- Os sistemas compatíveis com REST, geralmente são chamados de sistemas **RESTful** e são caracterizados por serem sem estado e separarem as preocupações do cliente e do servidor.

- Veremos o que esses termos significam e por que essas características são úteis para serviços na Web.

- Preste muita atenção: se você está procurando uma carreira em tecnologia, pode ser solicitado que você defina REST durante uma entrevista.

Separação entre Cliente e Servidor

- No estilo de arquitetura REST, a implementação do cliente e a implementação do servidor podem ser feitas independentemente, sem que um saiba sobre o outro.

- Isso significa que o código do lado do cliente pode ser alterado a qualquer momento sem afetar a operação do servidor, e o código do lado do servidor pode ser alterado sem afetar a operação do cliente.

Separação entre Cliente e Servidor

- Contanto que cada lado saiba qual formato de mensagem enviar para o outro, eles podem ser mantidos modulares e separados.
  - “Contrato” ou objeto.

- Separando as questões de interface do usuário das preocupações de armazenamento de dados, melhoramos a flexibilidade da interface entre plataformas e melhoramos a escalabilidade simplificando os componentes do servidor.

- Além disso, a separação permite a cada componente a capacidade de evoluir de forma independente.

Separação entre Cliente e Servidor

- Usando uma interface REST, diferentes clientes acessam os mesmos *endpoints* REST, executam as mesmas ações e recebem as mesmas respostas.
  - Site.
  - App.
  - Desktop.
  - Programas para teste.

Stateless

- Os sistemas que seguem o paradigma REST são *stateless* (sem estado), o que significa que o servidor não precisa saber nada sobre o estado em que o cliente está e vice-versa.

- Dessa forma, tanto o servidor quanto o cliente podem entender qualquer mensagem recebida, mesmo sem ver as mensagens anteriores.

- Essa restrição de *stateless* é aplicada por meio do uso de recursos, em vez de comandos.
  - Recursos são os substantivos da Web - eles descrevem qualquer objeto, documento ou coisa que você precise armazenar ou enviar para outros serviços.

Stateless

- Como os sistemas REST interagem por meio de operações padrão em recursos, eles não dependem da implementação de interfaces (um tipo de dados que age como uma abstração para uma classe).

- Essas restrições ajudam os aplicativos RESTful a obter confiabilidade, desempenho rápido e escalabilidade, como componentes que podem ser gerenciados, atualizados e reutilizados sem afetar o sistema como um todo, mesmo durante a operação do sistema.

- Agora, vamos explorar como a comunicação entre o cliente e o servidor realmente acontece quando estamos implementando uma interface RESTful.

Comunicação entre Cliente e Servidor

- Na arquitetura REST, os clientes enviam solicitações para recuperar ou modificar recursos e os servidores enviam respostas a essas solicitações.
  - Paradigma Cliente-Servidor.

- Vamos dar uma olhada nas formas padrão de fazer solicitações e enviar respostas.

Comunicação entre Cliente e Servidor

- Fazendo Requisições (Requests):

- O REST requer que um cliente faça uma solicitação ao servidor para recuperar ou modificar dados no servidor. Uma solicitação geralmente consiste em:
  - Um verbo HTTP, que define que tipo de operação executar.
  - Um cabeçalho, que permite ao cliente passar informações sobre a solicitação.
  - Um caminho para um recurso (Path).
  - Um corpo de mensagem opcional contendo dados.

Comunicação entre Cliente e Servidor

- **Verbos HTTP**:

  - Existem 4 verbos HTTP básicos que usamos em requisições para interagir com recursos em um sistema REST:
    - **GET** — recupera um recurso específico (ex: por id) ou uma coleção de recursos
    - **POST** — cria um novo recurso
    - **PUT** — atualiza um recurso específico (ex: por id)
    - **DELETE** — remove um recurso específico por id

  - Uso por exemplo em CRUDs
    - Create, Read, Update e Delete.

Comunicação entre Cliente e Servidor

- Cabeçalhos e Parâmetros:
  - No cabeçalho da requisição, o cliente envia o tipo de conteúdo que pode receber do servidor.
  - Este é o campo ***Accept***, ele garante que o servidor não envie dados que não possam ser entendidos ou processados pelo cliente.
  - As opções de tipos de conteúdo são Tipos MIME (*Multipurpose Internet Mail Extensions*), sobre os quais você pode ler mais no MDN Web Docs

Comunicação entre Cliente e Servidor

- Tipos MIME, usados para especificar os tipos de conteúdo no campo ***Accept***, consistem em um tipo e um subtipo. Eles são separados por uma barra (/).
  - Por exemplo, um arquivo de texto contendo HTML seria especificado com o tipo *text/html*.
  - Se este arquivo de texto contivesse CSS, seria especificado como *text/css*.
  - Um arquivo de texto genérico seria denotado como *text/plain*.
  - Este valor padrão, *text/plain*, não é um *catch-all*! Se um cliente estiver esperando *texto/css* e receber *text/plain*, ele não será capaz de reconhecer o conteúdo.

Comunicação entre Cliente e Servidor

- Outros tipos e subtipos comumente usados:
  - **image —** image/png, image/jpeg, image/gif.
  - **audio —** audio/wav, audio/mpeg.
  - **video —** video/mp4, video/ogg.
  - **application —** application/json, application/pdf, application/xml, application/octet-stream.

- Por exemplo, um cliente acessando um recurso com id 23 em um recurso de artigos em um servidor pode enviar uma solicitação **GET** como esta:

```
GET /articles/23
Accept: text/html, application/xhtml
```

Paths

- As solicitações devem conter um caminho (*Path*) para um recurso no qual a operação deve ser executada.
  - Nas APIs RESTful, os caminhos devem ser projetados para ajudar o cliente a saber o que está acontecendo.

- Convencionalmente, a primeira parte do caminho deve ser a forma plural do recurso. Isso mantém os caminhos aninhados simples de ler e fáceis de entender.

- Um caminho como `fashionboutique.com/customers/223/orders/12` é claro no que aponta, mesmo que você nunca tenha visto esse caminho específico antes, porque é hierárquico e descritivo.
  - Podemos ver que estamos acessando o **pedido** com **id** 12 para o **cliente** com **id** 223.

Paths

- Os caminhos devem conter as informações necessárias para localizar um recurso com o grau de especificidade necessário.
  - Ao se referir a uma lista ou coleção de recursos, nem sempre é necessário adicionar um id.
  - Por exemplo, uma requisição POST para o caminho `fashionboutique.com/customers` não precisaria de um identificador extra, pois o servidor irá gerar um id para o novo objeto.

- Se estivéssemos tentando acessar um único recurso, precisaríamos anexar um id ao caminho. Por exemplo:
  - **GET** `fashionboutique.com/customers/:id` — recupera o item no recurso *customers* com o id especificado.
  - **DELETE** `fashionboutique.com/customers/:id` — exclui o item no recurso *customers* com o id especificado.

Enviando respostas

- *Content Types*:
  - Nos casos em que o servidor está enviando uma carga de dados (*payload*) para o cliente, o servidor deve incluir um *content-type* (tipo de conteúdo) no cabeçalho da resposta.
  - Esse campo de cabeçalho alerta o cliente sobre o tipo de dados que está enviando no corpo da resposta.
    - Esses tipos de conteúdo são tipos MIME, assim como estão no campo ***accept*** do cabeçalho da solicitação.
    - O *content-type* que o servidor envia de volta na resposta deve ser uma das opções que o cliente especificou no campo ***accept*** da solicitação.

Enviando respostas

- Por exemplo, quando um cliente está acessando um recurso com id 23 em um recurso de artigos com esta solicitação GET:

```
GET /articles/23 HTTP/1.1
Accept: text/html, application/xhtml
```

- O servidor pode enviar de volta o conteúdo com o cabeçalho de resposta:

```
HTTP/1.1 200 (OK)
Content-Type: text/html
```

- Isso significaria que o conteúdo solicitado está sendo retornado no corpo da resposta com um *content-type text/html*, que o cliente disse que poderia aceitar.

Response Codes

- As respostas do servidor contêm códigos de status para alertar o cliente sobre informações sobre o sucesso (ou não) da operação.
- Como desenvolvedor, você não precisa conhecer todos os códigos de status (existem muitos), mas deve conhecer os mais comuns e como são usados:

Status code	Meaning
200 (OK)	This is the standard response for successful HTTP requests.
201 (CREATED)	This is the standard response for an HTTP request that resulted in an item being successfully created.
204 (NO CONTENT)	This is the standard response for successful HTTP requests, where nothing is being returned in the response body.
400 (BAD REQUEST)	The request cannot be processed because of bad request syntax, excessive size, or another client error.
403 (FORBIDDEN)	The client does not have permission to access this resource.
404 (NOT FOUND)	The resource could not be found at this time. It is possible it was deleted, or does not exist yet.
500 (INTERNAL SERVER ERROR)	The generic answer for an unexpected failure if there is no more specific information available.

Response Codes

- Para cada verbo HTTP, há códigos de status esperados que um servidor deve retornar após o sucesso:
  - **GET** — *return 200 (OK)*
  - **POST** — *return 201 (CREATED)*
  - **PUT** — *return 200 (OK)*
  - **DELETE** — *return 204 (NO CONTENT)*

- Se a operação falhar, retorne o código de status mais específico possível correspondente ao problema encontrado.

Exemplos de Solicitações e Respostas

- Digamos que temos um aplicativo que permite visualizar, criar, editar e excluir clientes e pedidos de uma pequena loja de roupas hospedada em `fashionboutique.com`.

- Poderíamos criar uma API HTTP que permite que um cliente execute estas funções:
  - Se quiséssemos visualizar todos os clientes, a solicitação ficaria assim:

    ```
    GET http://fashionboutique.com/customers
    Accept: application/json
    ```

  - Um possível cabeçalho de resposta seria:

    ```
    Status Code: 200 (OK)
    Content-type: application/json
    ```

- Seguido dos dados dos clientes solicitados no formato `application/json`.

Exemplos de Solicitações e Respostas

- Criar um novo cliente postando os dados:

```
POST http://fashionboutique.com/customers
Body:
{
  “customer”: {
    “name” = “Fulano Ciclano”,
    “email” = “fulano.ciclano@catolicasc.org.br”
  }
}
```

- O servidor então gera um id para aquele objeto e o retorna para o cliente, com o seguinte cabeçalho:

```
201 (CREATED)
Content-type: application/json
```

Exemplos de Solicitações e Respostas

- Para visualizar um único cliente, recuperamos ele especificando o ID desse cliente:

```
GET http://fashionboutique.com/customers/123
Accept: application/json
```

- Um possível cabeçalho de resposta seria:

```
Status Code: 200 (OK)
Content-type: application/json
```

- Seguido pelos dados do recurso do cliente com id 23 no formato `application/json`.

Exemplos de Solicitações e Respostas

- Podemos atualizar esse cliente colocando os novos dados:

```
PUT http://fashionboutique.com/customers/123
Body:
{
  “customer”: {
    “name” = “Fulano Ciclano da Silva”,
    “email” = “fulano.ciclano@catolicasc.org.br”
  }
}
```

- Um possível cabeçalho de resposta teria o Código de Status: `200 (OK)`, para notificar o cliente que o item com id 123 foi modificado.

Exemplos de Solicitações e Respostas

- Também podemos excluir esse cliente especificando seu id:

```
DELETE http://fashionboutique.com/customers/123
```

- A resposta teria um cabeçalho contendo o Código de Status: `204` (SEM CONTEÚDO), notificando o cliente que o item com id 123 foi deletado, e nada no corpo da resposta.

Atividades

Prática - Flask

- Para desenvolvermos uma API RESTFul de modo rápido no Python, podemos usar o Framework Flask.

- https://flask.palletsprojects.com/en/2.2.x/

Atividade

- Ler a documentação básica do Flask:
  - https://flask.palletsprojects.com/en/2.2.x/quickstart/#a-minimal-application

- Ideia é entender alguns princípios básicos descritos.
- 15 minutos.

Material complementar

- Learn REST: A RESTful Tutorial:

  https://restapitutorial.com/

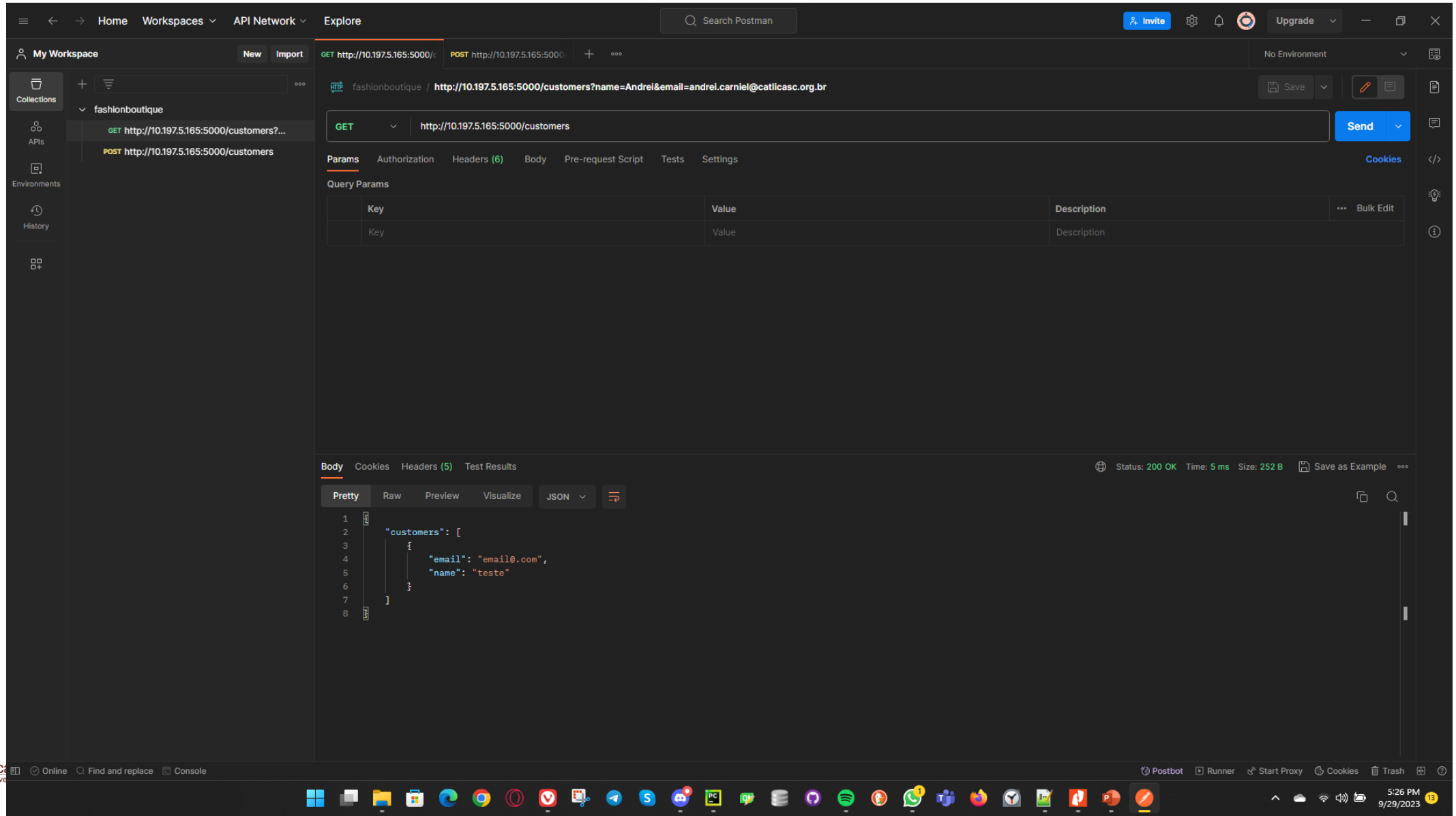
- JSON API: Explained in 4 minutes (+ EXAMPLES)

  https://www.youtube.com/watch?v=N-4prIh7t38

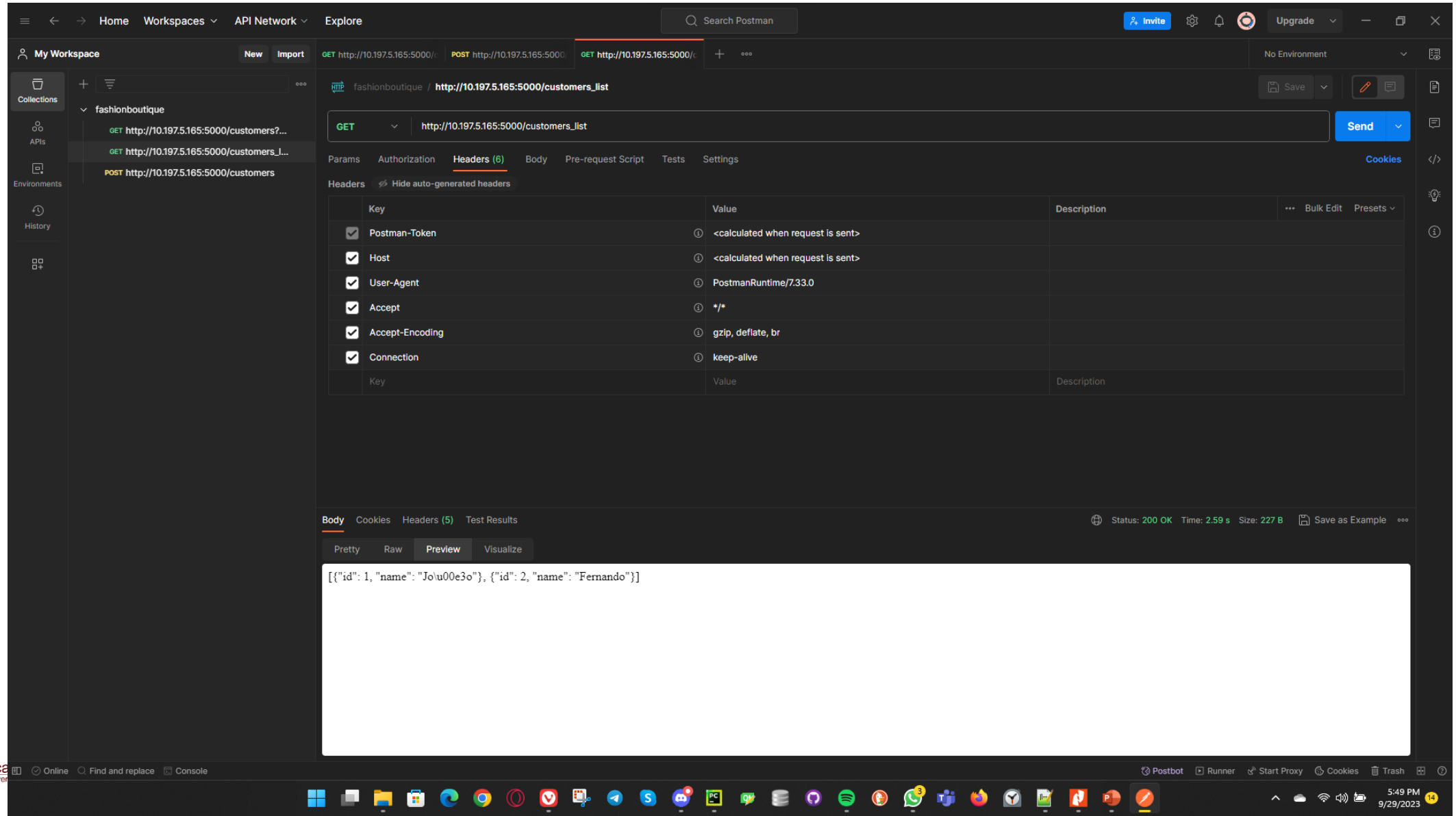
Colocando em prática o exemplo

- Anteriormente vimos o exemplo da `fashionboutique.com`.
- Agora vamos implementar com o Flask os métodos descritos anteriormente
  - Slides “Exemplos de Solicitações e Respostas”.
- As duas primeiras funções estão detalhadas no seguinte código:
  - *main_07.py* e *Customer.py.*
  - Podemos testar via Postman.
- Faça o clone do repositório e vamos colocar para funcionar
- Note que é utilizado JSON como meio de transferir os dados
- Vocês deverão implementar:
  - Busca de cliente por ID.
  - Alteração de cliente por ID.
  - Remoção de cliente por ID.

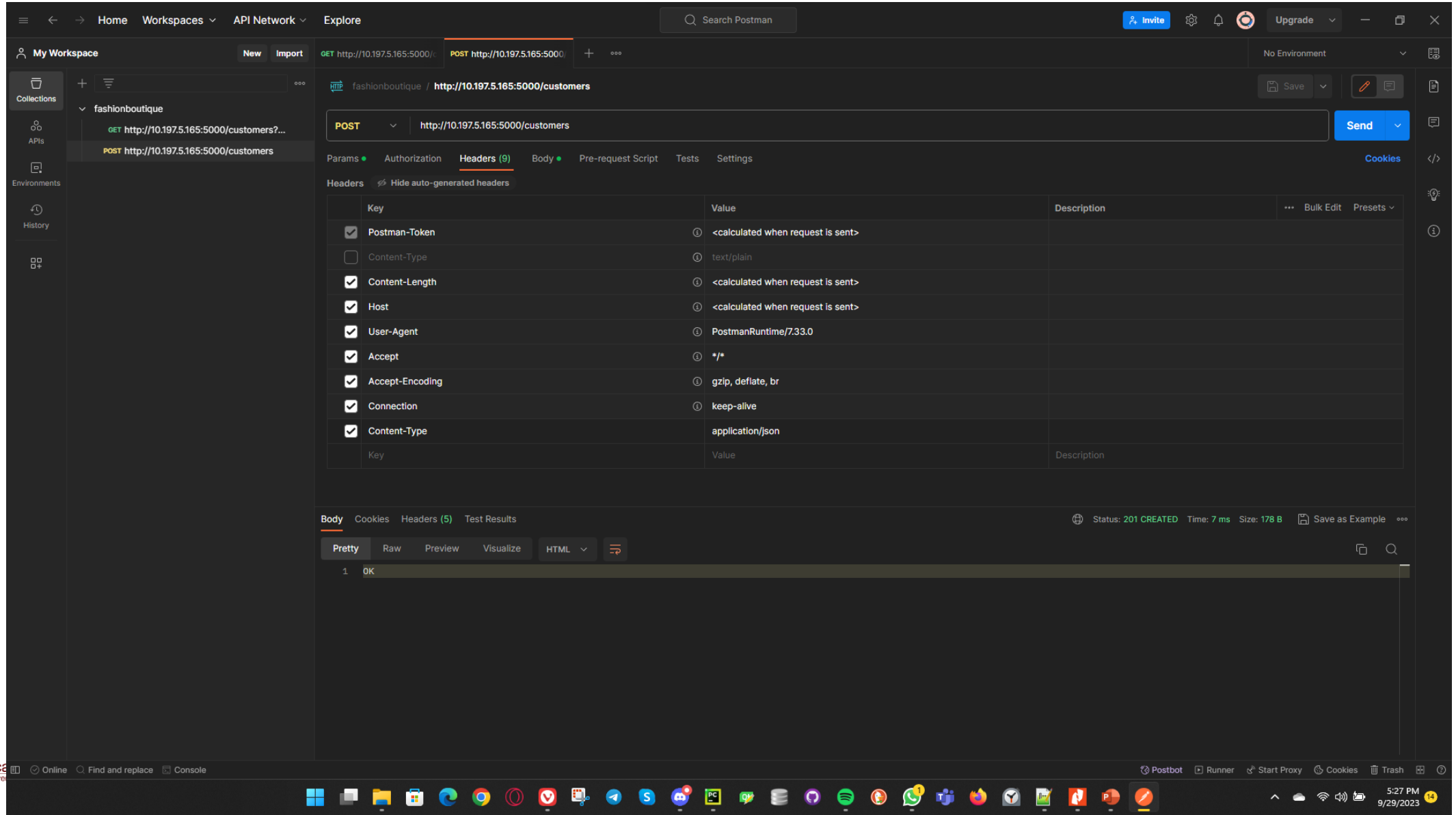
Postman - GET


Home
Workspaces
API Network
Explore
Search Postman
Invite
Upgrade
My Workspace
New
Import
GET http://10.197.5.165:5000/c
POST http://10.197.5.165:5000/
No Environment
Collections
APIs
Environments
History
fashionboutique
GET http://10.197.5.165:5000/customers?...
POST http://10.197.5.165:5000/customers
fashionboutique / http://10.197.5.165:5000/customers?name=Andrei&email=andrei.carniel@catlicasc.org.br
Save
GET
http://10.197.5.165:5000/customers
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 5 ms
Size: 252 B
Save as Example
Pretty
Raw
Preview
Visualize
JSON
{
"customers": [
{
"email": "email@.com",
"name": "teste"
}
]
}
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Trash
5:26 PM
9/29/2023

Postman - GET


Home
Workspaces
API Network
Explore
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
APIs
Environments
History
fashionboutique
GET http://10.197.5.165:5000/customers?...
GET http://10.197.5.165:5000/customers_l...
POST http://10.197.5.165:5000/customers
No Environment
fashionboutique / http://10.197.5.165:5000/customers_list
Save
GET
http://10.197.5.165:5000/customers_list
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Headers
Hide auto-generated headers
Key
Value
Description
Bulk Edit
Presets
Postman-Token
<calculated when request is sent>
Host
<calculated when request is sent>
User-Agent
PostmanRuntime/7.33.0
Accept
/
Accept-Encoding
gzip, deflate, br
Connection
keep-alive
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 2.59 s
Size: 227 B
Save as Example
Pretty
Raw
Preview
Visualize
[{"id": 1, "name": "Jo\u00e3o"}, {"id": 2, "name": "Fernando"}]
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Trash
5:49 PM
9/29/2023

Postman – POST (1/2)



Postman – POST (2/2)