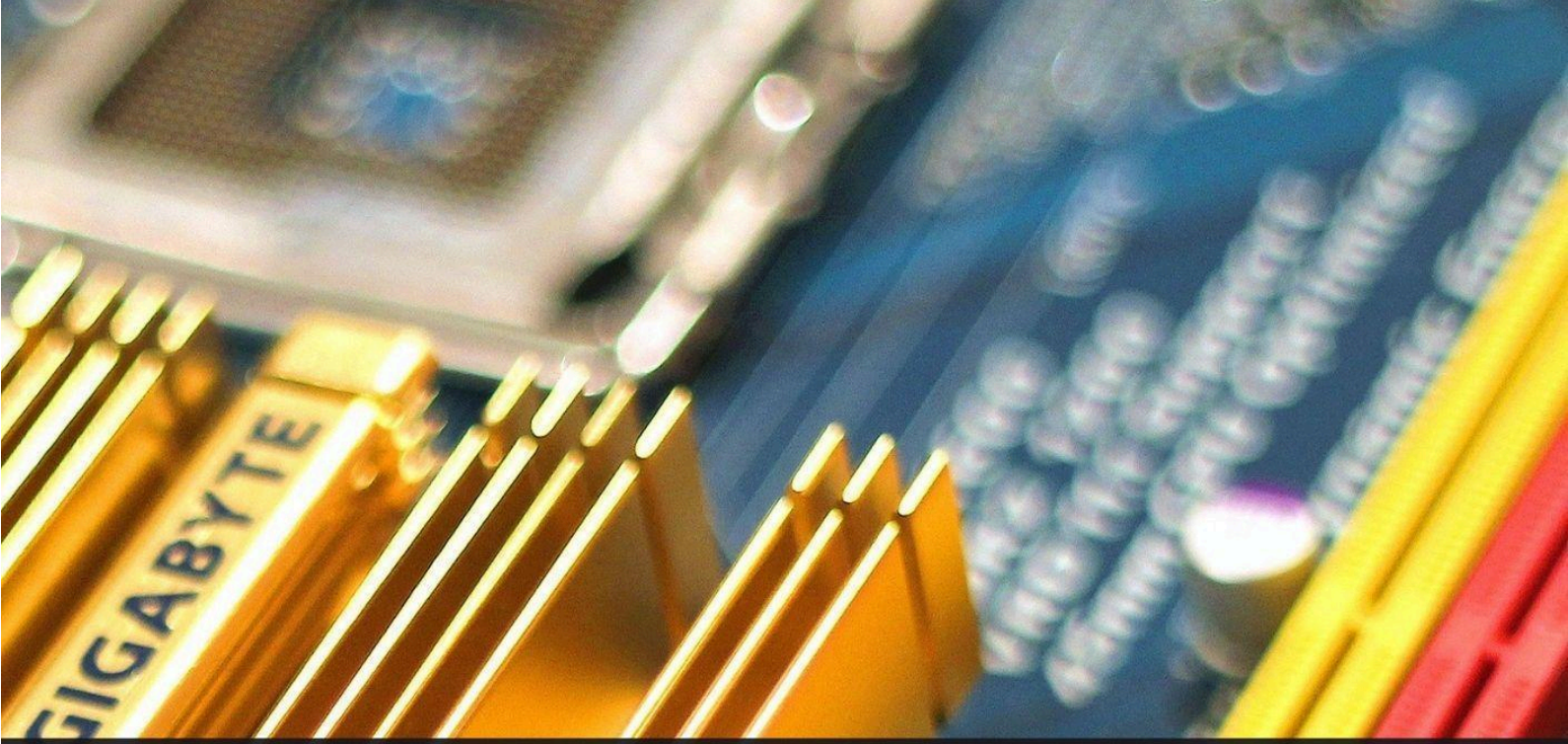
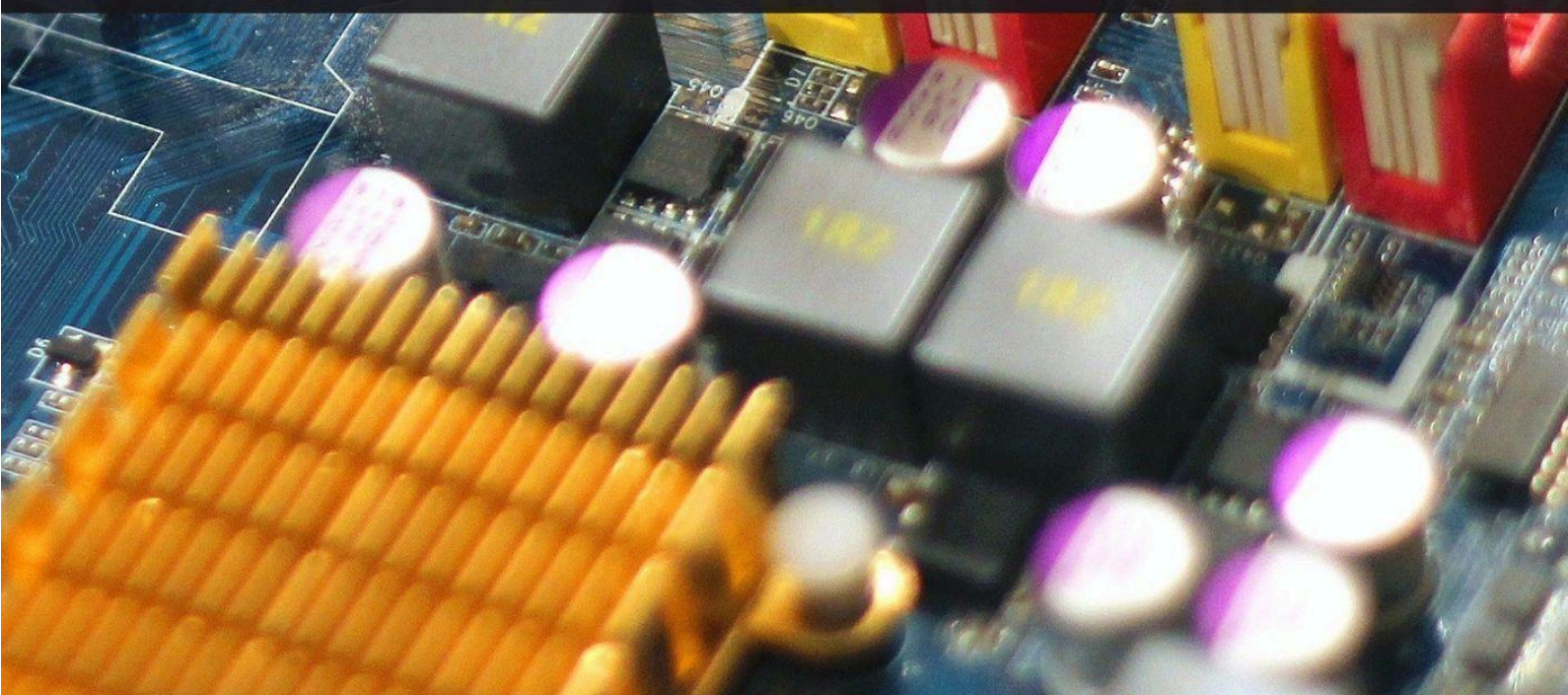


DESENVOLVIMENTO DE APIs E MICROSSERVIÇOS



11

Arquitetura MVC

Andréia Cristina dos Santos Gusmão

Resumo

Após aprender a construir servidores com flask, vamos estudar como tornar mais fácil a manutenção de nossos códigos, dividindo as responsabilidades em estruturas. Para essa tarefa, vamos utilizar o padrão de projeto MVC (Model, View, Controller), que tem como funcionalidade deixar nossa aplicação independente.

11.1. Estruturando nossos dados

Uma maneira de organizar nosso código, é estruturando-o de forma que dividimos em partes menores, em que cada parte, seja responsável por uma determinada tarefa. Essa divisão não é somente em nível de criar funções, mas sim, em estruturar um arquivo em vários arquivos, de forma que cada um, contenha exatamente o que precisa e que todos conversem entre si.

Existem diversos padrões, que nada mais é, que recomendações de procedimentos já testados e que funcionam, para que possamos ter agilidade, segurança e centralidade em nosso código, os chamados **padrões de projeto** (*design patterns*).

Antes de apresentar o padrão MVC, vamos entender o que são Microsserviços.

11.2. Microsserviços

Vamos conceituar o que é Serviço: **serviço** é uma forma de entregar VALOR ao cliente, facilitando a obtenção de resultados que ele quer alcançar, sem que ele assuma os riscos inerentes.

Assim, **Microsserviços** são pequenos serviços autônomos que funcionam de forma integrada e independente.

As principais características de microsserviços são:

1. **Tamanho pequeno:** possuem foco em fazer apenas uma coisa bem feita; possuem uma Alta coesão: Componentes que são relacionados ficam no mesmo código.

2. **Autônomos:** não é necessário parar o microsserviço *A* para fazer o deploy do microsserviço *B*. Se o microsserviço *A* estiver sobrecarregado, é possível colocar mais cópias dele para atender melhor a alta demanda sem impactar os demais.

3. **Integrados:** podem se comunicar por APIs de maneira síncrona ou por mensagens de maneira assíncrona.

11.3 Blueprints para segregação de microsserviços

Em uma implementação com flask, uma maneira de dividir os microsserviços em módulos independentes é através do uso de blueprints. Ao invés de criar apenas um módulo contendo a descrição de rotas de todas as APIs, como está colocado no exemplo a seguir, é possível criar um módulo para cada escopo de microsserviço.

Veja a seguir, o código do arquivo sala_aula.py antes da segregação, contendo as rotas /alunos e /professores. Esse código já foi explicado na aula anterior (exemplo2.py), portanto, não serão detalhadas as rotas já criadas.

Figura 11.1. Arquivo sala_aula.py antes da segregação.

```python
from flask import Flask, jsonify

app = Flask(__name__)

database = {
    'ALUNO' : [{"id": 1, "nome": "Andreia"},
               {"id": 2, "nome": "Arthur"},
               {"id": 3, "nome": "Pedro"}],

    'PROFESSOR' : [{"id": 1, "nome": "Professor1"},
               {"id": 2, "nome": "Professor2"},
               {"id": 3, "nome": "Professor3"}],
}

@app.route('/alunos')
def getAlunos():
    return jsonify(database['ALUNO'])


@app.route('/professores')
def getProfesores():
    return jsonify(database['PROFESSOR'])


if __name__ == '__main__':
    app.run(host = 'localhost', port = 5002, debug = True)
```

Fonte: do autor, 2021.

A ideia é dividir nossa aplicação, por exemplo, podemos criar o arquivo *alunos_api.py* para escrever as rotas das APIs referentes aos alunos e criar *professores_api.py* para as rotas de professores de modo análogo, e assim por diante.

A Figura 11.2 mostra o código do arquivo alunos_api.py, somente com os dados de alunos, referente ao código apresentado na Figura 11.1. O que temos de novo, é que vamos fazer o import do Blueprint, conforme pode ser visto na primeira linha da Figura 11.2: from flask import Flask, jsonify, **Blueprint**

Na linha 3 da Figura 11.2, definimos que iremos utilizar Blueprint, em que alunos_app é o nome que iremos utilizar neste código para construir nossas rotas em flask (linha 14 Figura 11.2). Uma dica, utilizar nomes que representam qual 'app' estamos criando (alunos_app, professores_app, etc).

Figura 11.2. Definindo Blueprint para o arquivo alunos_app.py

```
1   from flask import Flask, jsonify, Blueprint
2
3   alunos_app = Blueprint('alunos_app', __name__ ,
4       template_folder='templates')
5
6
7   database = {
8       'ALUNO' : [{"id": 1, "nome": "Andreia"},
9                  {"id": 2, "nome": "Arthur"},
10                 {"id": 3, "nome": "Pedro"}]
11  }
12
13  #rota /alunos padrão GET: retorna todos os alunos
14  @alunos_app.route('/alunos')
15  def getAlunos():
16      return jsonify(database['ALUNO'])
```

Fonte: do autor, 2021.

A Figura 11.3 representa o código do arquivo professores_api.py, seguindo a mesma lógica da explicação referente à Figura 11.2.

Figura 11.3. Definindo Blueprint para o arquivo professores_app.py.

```
1   from flask import Flask, jsonify, Blueprint
2
3   professores_app = Blueprint('professores_app', __name__ ,
4       template_folder='templates')
5
6   database = {
7       'PROFESSOR' : [{"id": 1, "nome": "Professor1"},
8                      {"id": 2, "nome": "Professor2"},
9                      {"id": 3, "nome": "Professor3"}]
10  }
11
12  #rota /professores padrão GET: retorna todos os professores
13  @professores_app.route('/professores')
14  def getProfessores():
15      return jsonify(database['PROFESSOR'])
```

Fonte: do autor, 2021.

E para finalizar essa parte, apresentamos o arquivo sala_aula.py (Figura 11.4), após a extração das rotas de professores e alunos para isolá-las em seus próprios arquivos (blueprints).

Figura 11.4. Arquivo sala_aula.py após a segregação.

```
1   from flask import Flask, jsonify
2   from alunos_api import alunos_app
3   from professores_api import professores_app
4
5   app = Flask(__name__)
6
7   app.register_blueprint(alunos_app)
8   app.register_blueprint(professores_app)
9
10  if __name__ == '__main__':
11      app.run(host = 'localhost', port = 5002, debug = True)
```

Fonte: do autor, 2021

Note que é possível criar as rotas para alunos, professores e outros serviços em arquivos separados (Figura 11.2 e Figura 11.3). Sendo que, no arquivo principal (Figura 11.4) basta registrá-los com o método: register_blueprint() (linhas 7 e 8 da Figura 11.4), em que passamos como parâmetro o nome da aplicação para as rotas definidas em cada arquivo (alunos_api.py e professor_api.py). Repare que na Figura 11.4 nós removemos todos os códigos referentes às rotas de alunos e professores.

É obrigatório também, fazer o import dos arquivos criados (linhas 2 e 3 da Figura 11.4): from alunos_api import alunos_app, quer dizer que, do arquivo python alunos_api vamos importar para o nosso código principal a definição da aplicação 'alunos_app'. E no arquivo *sala_aula.py*, apenas registramos as *blueprints* de professores e alunos.

Ao executar a aplicação, todas as rotas dos módulos registrados serão consideradas. Para executar, basta acessar o *prompt de comando,* acessar a pasta onde se encontram os arquivos (para esse exemplo c:\dam) e digite: python sala_aula.py (Figura 11.5).

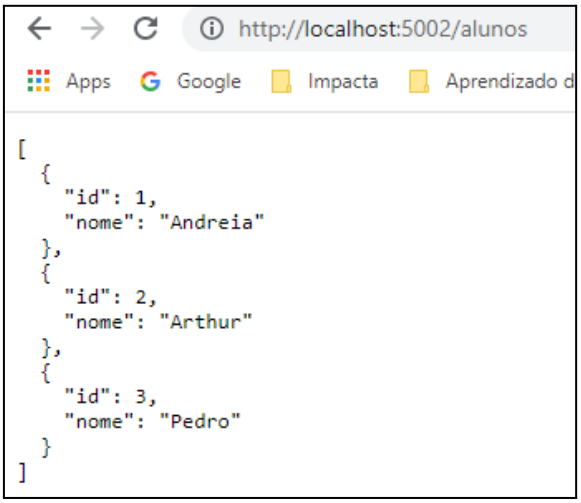
Figura 11.5. Execução do arquivo sala_aula.py.

```
C:\dam>python sala_aula.py
 * Serving Flask app "sala_aula" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 119-118-421
 * Running on http://localhost:5002/ (Press CTRL+C to quit)
```

Fonte: do autor, 2021

Após a execução, podemos testar no navegador, por exemplo a rota @alunos_app.route('/alunos') (método GET) em que serão mostrados na tela, todos os alunos cadastrados: http://localhost:5002/alunos.

Figura 11.6. Resultado da execução da rota /alunos.



Fonte: do autor, 2021

11.4. O que é o padrão MVC

Vimos anteriormente, uma maneira de estruturar nossa aplicação, com uso de Blueprints. Agora, vamos aprender sobre o padrão de projeto MVC. Mas afinal, o que é MVC?

MVC é a sigla para Model (modelo), View (visão) e Controller (controle) que facilita a troca de informações entre a interface do usuário aos dados no banco, fazendo com que as respostas sejam mais rápidas e dinâmicas (ZUCHER, 2020).

MVC é um padrão de desenvolvimento dividido em três partes, em que, outras partes podem ser acrescentadas de acordo com a necessidade. Nessa disciplina vamos focar em **Model e Controller**, pois a visualização (**View**) não é conteúdo dessa disciplina e requer conhecimentos de outras grades (html, javascript, etc).

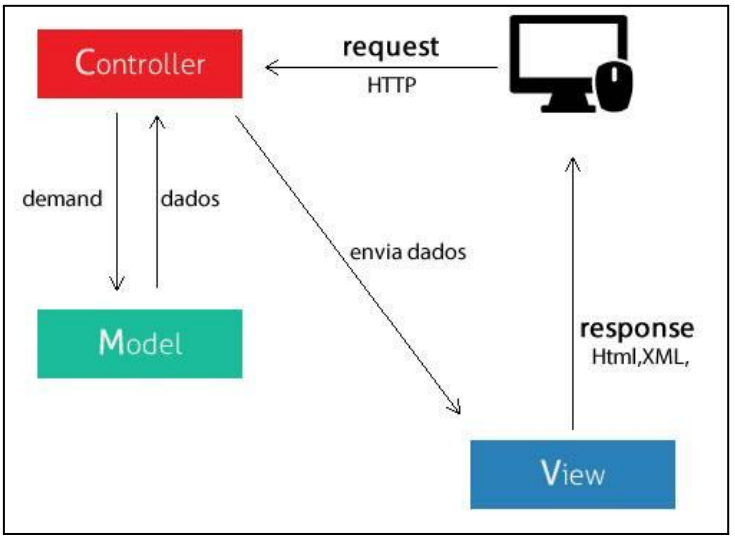
Segundo Ramos (2015), podemos resumir as três camadas do MVC como segue a seguir e apresentado na Figura 11.7.

- **View**: camada de parte visual, o que o usuário vê, exibição dos dados. Interação do usuário. Exemplo: um formulário em html, em que os o usuário digita seus dados, um xml que retorna dados de uma nota fiscal, etc;

- **Model**: camada de manipulação dos dados. É a camada que tem como objetivo fazer a leitura e escrita dos dados e suas validações;

- **Controller**: camada de controle que faz a 'ponte' entre a Model e a View. Recebe as requisições do usuário. Seus métodos são responsáveis por controlar qual model será utilizado e qual view retornará para o usuário.

Figura 11.7. Como o padrão MVC funciona.



Fonte: Ramos, 2015

Mas afinal, para que dividir minha aplicação em 'mais arquivos'? Quais benefícios? Segurança, Organização, Eficiência, Tempo e Transformação (no vídeo, falaremos um pouco de cada um desses benefícios).

11.5. Criando nossa aplicação com MVC

Então, vamos praticar? Vimos na aula passada e início dessa, exemplos com alunos e professores. Continuamos com essas entidades, vamos criar uma aplicação em flask com MVC, com foco nas camadas de **controller** e **model**.

Vamos considerar os dados de entrada, como uma lista de dicionário de alunos, que tenham as informações de id, nome, media, em que 'media' significa a média de todas as disciplinas cursadas pelo aluno em um determinado período letivo.

Vamos então, dividir nossa aplicação em 3 arquivos:

sala_aula_server.py = arquivo principal que contém as rotas em Flask. Como não temos a view, esse arquivo fará o mesmo papel, mesmo que não tenha um visual desenvolvido.

aluno_controller.py = é o arquivo controller, que irá acionar o model, para enviar as requisições de sala_aula_server.py. É onde ocorrem as validações.

aluno_model.py = é o arquivo model, que contém nosso database com os dados de alunos e toda a lógica de negócio.

Considere o nosso database de entrada, com dados de 4 alunos.

```
database = [{"id": 1, "nome": "Andreia", "media": 8.5},
            {"id": 2, "nome": "Arthur", "media": 10},
            {"id": 3, "nome": "Pedro", "media": 10},
            {"id": 4, "nome": "Ana", "media": 7}]
```

Criando nosso CRUD

Vamos demonstrar nosso exemplo com método de inserir (create **C**), ler/buscar (read **R**), alterar/atualizar (update **U**) e para excluir (delete **D**).

As rotas não serão detalhadas nesta aula, pois já vimos com detalhes na aula anterior. O foco é mostrar como dividir nosso código em MVC.

Buscando todos os alunos cadastrados

Considere que, o usuário quer buscar e apresentar na tela, todos os alunos cadastrados. Nessa aula, embora não estamos focando na visualização, definimos aqui, que o arquivo sala_aula_server.py será nossa view.

Observe o diálogo entre as camadas mvc, para melhor compreensão da teoria:

- **View**: E aí controller! O usuário acabou de pedir para visualizar todos os alunos cadastrados.
- **Controller**: Certo. Já te envio a resposta. Amigo model, a view disse que o usuário quer ver os alunos cadastrados.
- **Model**: Verifiquei aqui o database. Já estou retornando.
- **Controller**: Obrigado. View, existem alunos cadastrados sim. Vou te enviar e você mostra na tela, ok?
- **View**: Valeu amigo. Mostrando ao usuário.

Model:

Para isso criamos um método **getAll()** que, caso o database retorne True (tem dados) retorna para o usuário o database convertido para json. Caso contrário, retorna None (nada).

Esse código faz parte de qual arquivo **aluno_model.py.**

```
def getAll():
    if database:
        return jsonify(database)
    return None
```

Controller

No nosso arquivo **aluno_controller.py**, temos a função def listar(), que 'chama' getAll() definida em aluno_model.py. Se retornar algum dado, será retornado para a view, senão apresenta a mensagem 'Não existem alunos cadastrados. Verifique!'

Para que possamos utilizar/chamar métodos da classe model, é obrigatório fazer o import: import model.aluno_model as aluno_model. Isso quer dizer que, estamos

importando o arquivo aluno_model que está dentro da pasta model e vamos chamá-la de 'aluno_model'.

```
def listar():
    alunos = aluno_model.getAll()
    if alunos == None:
        return 'Não existem alunos cadastrados. Verifique!'
    return alunos
```

View

No arquivo **sala_aula_server.py** precisamos importar o arquivo controller: import controller.aluno_controller as aluno_controller, e vamos chama-lo de **aluno_controller**. Nesse arquivo criamos as notas rotas em flask e inicializamos a aplicação.

```
import controller.aluno_controller as aluno_controller

app = Flask(__name__)

@app.route('/alunos')
def getAlunos():
    return aluno_controller.listar()
```

Como testar nossa aplicação? Da mesma forma já vista na aula anterior.

Considere que nossos arquivos estão no diretório **c:/dam/mvc**. Vamos acessar o *prompt de comando* e digitar **python sala_aula_server.py**

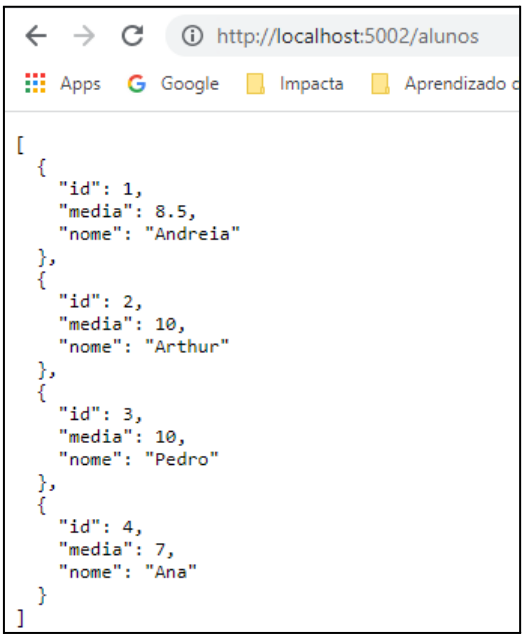
Figura 11.8. Execução no prompt de comando: sala_aula_server.py.

```
C:\dam\mvc>python sala_aula_server.py
 * Serving Flask app "sala_aula_server" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 119-118-421
 * Running on http://localhost:5002/ (Press CTRL+C to quit)
```

Fonte: do autor, 2021

Após a execução do servidor flask iniciar, abrimos o navegador e digitamos http://localhost:5002/alunos. Lembre-se, chamamos pelo nome da rota, que aqui é /alunos, para o método GET. A Figura 11.9 mostra o resultado dessa execução.

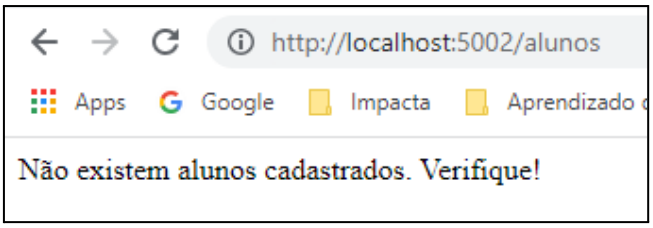
Figura 11.9. Execução da rota /alunos para mostrar todos os alunos cadastrados.



Fonte: do autor, 2021

Considere agora, o nosso database de alunos vazio: database = []. Nesse caso, executando a mesma rota, será retornada uma frase informando que não existem alunos cadastrados, conforme Figura 11.10.

Figura 11.10. Execução da rota /alunos a partir de um database vazio.



Fonte: do autor (2021).

Essa é a lógica do MVC, dividir em camadas as responsabilidades. Seguimos essa mesma lógica para todas as outras opções disponíveis e funcionalidades requeridas.

Buscando um aluno pelo seu id

O usuário vai chamar a rota /alunos passando um parâmetro, que é o id do aluno que se quer pesquisar. Caso esse aluno exista, será apresentado na tela. A Figura 11.11 mostra os códigos para essa busca divididos nas três camadas.

Figura 11.11. Códigos da rota /alunos para busca de um aluno pelo seu id.

Model

```
def getAlunoId(id_consulta):
    for aluno in database:
        if aluno['id'] == id_consulta:
            return aluno
    return None
```

Controller

```
def localizaPorId(id_consulta):
    aluno = aluno_model.getAlunoId(id_consulta)
    if aluno == None:
        return 'Aluno não encontrado'
    return  {"Status": "Aluno encontrado", "Aluno": aluno}
```

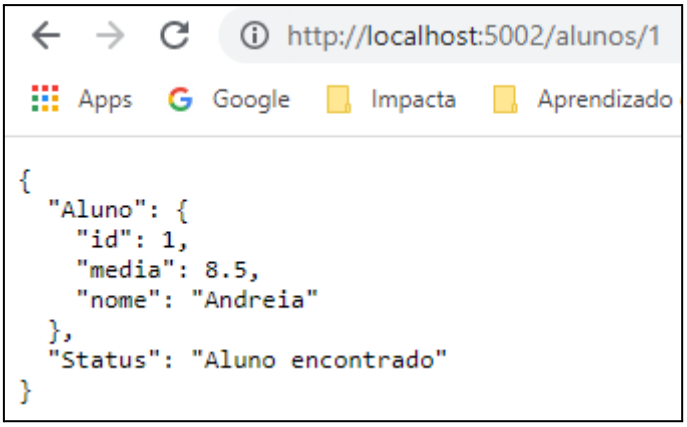
View

```
@app.route("/alunos/<int:id_consulta>", methods=["GET"])
def getAlunoId(id_consulta):
    return aluno_controller.localizaPorId(id_consulta)
```

Fonte: do autor, 2021

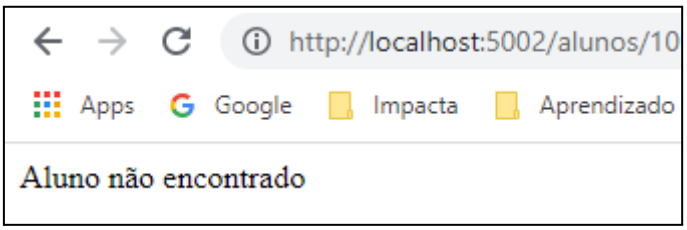
Na Figura 11.12 mostra a execução da rota através do navegador para a busca do aluno de id = 1 (http://localhost:5002/alunos/1) e na Figura 11.13, para o aluno id = 10 (http://localhost:5002/alunos/10), que não existe na nossa base de dados.

Figura 11.12. Resultado da busca de um aluno pelo seu id existente.



Fonte: do autor, 2021

Figura 11.13. Resultado da busca de um aluno pelo seu id inexistente.



Fonte: do autor, 2021

Não entramos nos detalhes da lógica para excluir um aluno pelo seu id, pois já foi explicado na aula anterior. Estamos focando em mostrar **como os códigos devem ser divididos.**

Inserindo um aluno

A inserção de um aluno é feita pelo método POST, não quer parâmetro, mas isso, corpo na requisição.

Figura 11.14. Códigos da rota /alunos para inserção de um aluno.

```
def inserirAluno(aluno):
    novo_aluno = aluno
    database.append(novo_aluno)
    return
```

Model

```
def inserirAluno(aluno):
    aluno_model.inserirAluno(aluno)
    return  listar()
```

Controller

```
@app.route("/alunos", methods=["POST"])
def inserir():
    aluno = request.json
    return aluno_controller.inserirAluno(aluno)
```

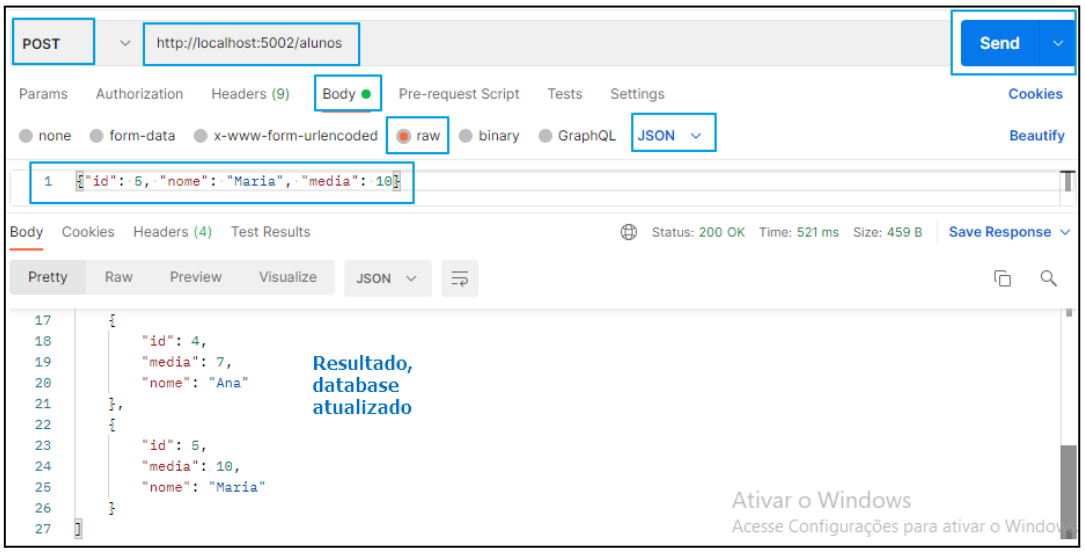
View

Fonte: do autor, 2021

Como testamos o método POST? Pelo POSTMAN. Aliás, os métodos POST, DELETE e PUT. GET é opcional.

Na Figura 11.15 mostramos no POSTMAN a inserção de um aluno com os dados: {"id": 5, "nome": "Maria", "media": 10}. Observe que na mesma figura é mostrado o database atualizado, já com o novo aluno inserido.

Figura 11.15. Testando a rota /alunos para inserção de um aluno.



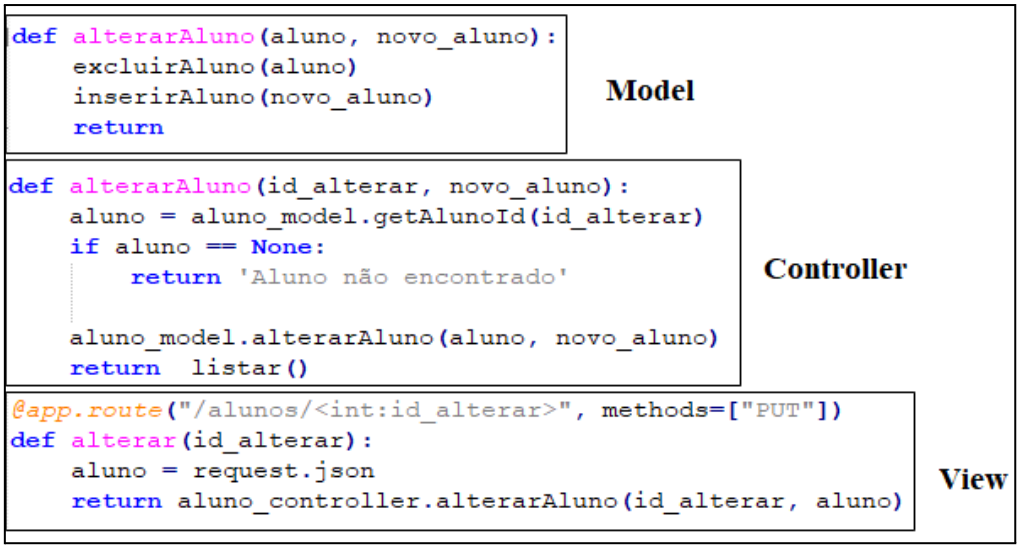
Fonte: do autor, 2021

Alterando um aluno

Para alterar um aluno, passamos como parâmetro **do id do aluno** que seja alterado e os **dados no formato json** do aluno com as devidas alterações. Como não estamos alterando um único atributo, temos que passar todos os atributos.

A Figura 11.16 mostra os códigos para essa alteração e a Figura 11.17 mostra como testar no POSTMAN.

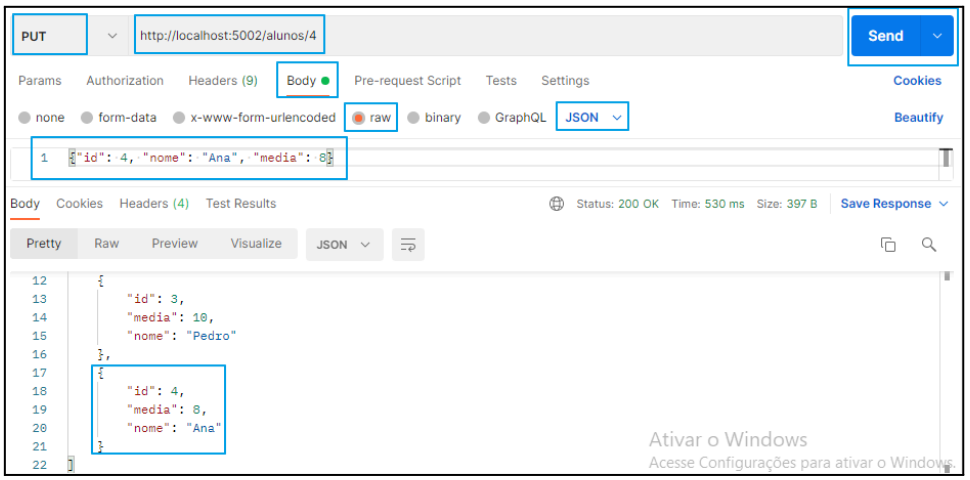
Figura 11.16. Códigos da rota /alunos para alteração de um aluno



Fonte: do autor, 2021

Estamos alterando o aluno de id = 4. Percebemos que a média da aluna estava errada, e por isso vamos passar os novos valores: {"id": 4, "nome": "Ana", "media": 8}. (http://localhost:5002/alunos/4).

Figura 11.17. Testando a rota /alunos para alteração de um aluno



Fonte: do autor, 2021

Excluindo um aluno

Para excluir um aluno, passamos passar como parâmetro **do id do aluno** que queremos excluir. Para esse exemplo, vamos excluir o **aluno de id = 4.**

A Figura 11.18 mostra os códigos para essa exclusão e a Figura 11.19 mostra como testar no POSTMAN.

Figura 11.18. Códigos da rota /alunos para exclusão de um aluno

```
def excluirAluno(aluno):
    database.remove(aluno)
```
Model

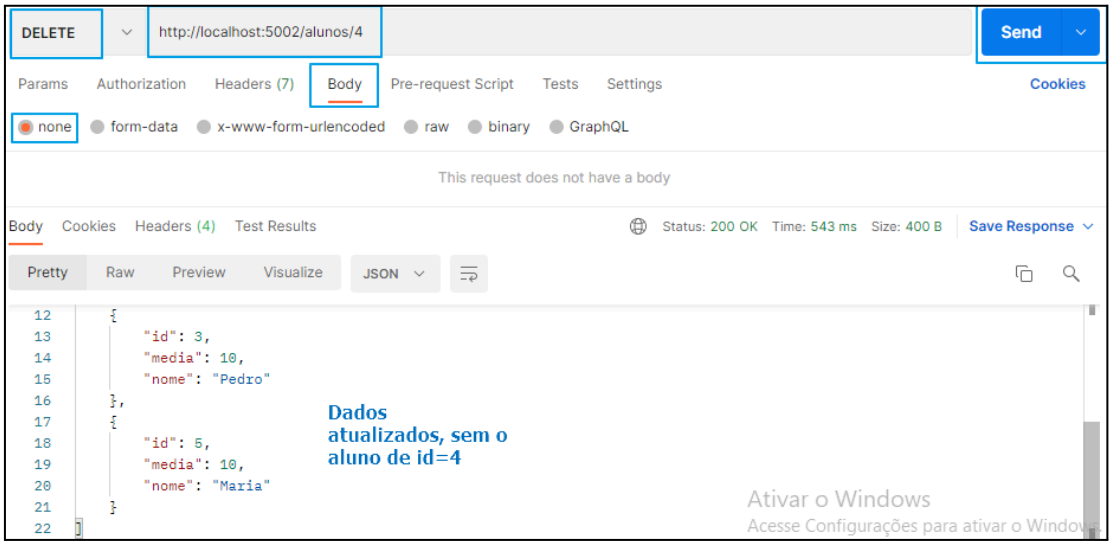
```
def excluirPorId(id_deletar):
    aluno = aluno_model.getAlunoId(id_deletar)
    if aluno == None:
        return 'Aluno não encontrado'

    aluno_model.excluirAluno(aluno)
    return  listar()
```
Controller

```
@app.route("/alunos/<int:id_deletar>", methods=["DELETE"])
def excluir(id_deletar):
    return aluno_controller.excluirPorId(id_deletar)
```
View

Fonte: do autor, 2021

Figura 11.19. Testando a rota /alunos para exclusão de um aluno



Fonte: do autor, 2021

Caso tentar excluir um aluno que não existe, por exemplo, id = 40, deve mostrar na tela que não existe o aluno, ou 'Aluno não encontrado'.

Buscando aluno(s) com a maior média

Vamos criar uma nota **rota /maior_media**, para que possamos localizar no database qual é a maior média entre os alunos cadastrados. Depois, retornar os alunos com a maior média.

Figura 11.20. Códigos da rota /maior_media para buscar os alunos com maior média

Model

```
def getAlunoMaiorMedia():
    maior_media = 0
    alunos_maior_media = []
    for aluno in database:
        if aluno['media'] >= maior_media:
            if aluno['media'] > maior_media:
                alunos_maior_media = []
            maior_media = aluno['media']
            alunos_maior_media.append(aluno)

    if alunos_maior_media:
        return jsonify(alunos_maior_media)
    return None
```

Controller

```
def localizarPorMaiorMedia():
    alunos = aluno_model.getAlunoMaiorMedia()
    return alunos
```

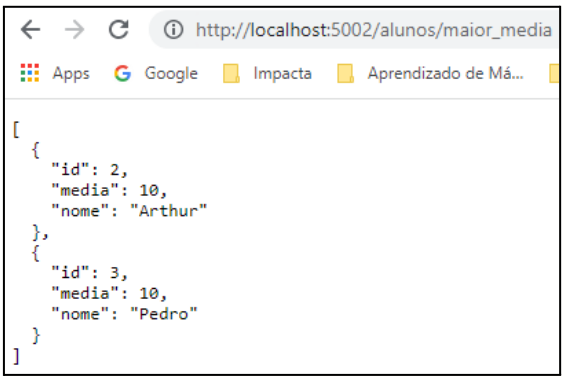
View

```
@app.route("/alunos/maior_media", methods=["GET"])
def getAlunoMaiorMedia():
    return aluno_controller.localizarPorMaiorMedia()
```

Fonte: do autor, 2021

Como é utilizado o método GET na rota /maior_media, podemos testar a execução direto no navegador digitando http://localhost:5002/alunos/maior_media. O resultado é mostrado na Figura 11.21.

Figura 11.21. Resultado da rota /maior_media



Fonte: do autor, 2021.

Apresentamos os métodos básicos de CRUD, mas podem aperfeiçoar, com novas buscas, de acordo com a necessidade.

Não usaremos aqui, exemplo com professores, mas analogamente podem criar os códigos para praticar, considerando, por exemplo, os atributos id, nome e ano_admissao.

11.6. Código completo

Os arquivos encontram-se como material complementar desta aula. Referente ao código com mvc, é a pasta mvc.zip. Os demais, são referentes ao uso de blueprint.

Referências

Flask BluePrints. Disponível em: https://flask.palletsprojects.com/en/1.0.x/blueprints/. Acesso em 20 out 2021.

Ramos, Allan. **O que é MVC**? Disponível em: https://tableless.com.br/mvc-afinal-e-o-que/. 2015. Acesso em 20 out 2021.

Zucher, Vitor. **O que é padrão MVC? Entenda arquitetura de softwares!** Disponível em: https://www.lewagon.com/pt-BR/blog/o-que-e-padrao-mvc/.2020. Acesso em 20 out 2021.