Programação de Sistemas para Internet

Prof. Romerito Campos

Padrão Model-View-Controller

Plano de Aula

- Objetivo: Compreender o padrão de design de arquitetura Model-View-Controller
- Materiais:
  - Código-Fonte - Estudo de Caso 1 - Versão rar
  - Código-Fonte - Estudo de Caso 2 - Versão rar

Conteúdos

- Padrão MVC
- Vantagens
- Componenentes
  - Model
  - View
  - Controller
- Funcionamento

Padrão MVC

Padrão MVC

- O MVC é uma sigla para o **padrão de projeto** Model-View-Controller.
- Este padrão é amplamente utilizado no mundo do desenvolvimento de software
- O objetivo é permitir a escrita da aplicação com base em componentes distintos realizando a separação de preocupações (*separation of concerns*)
- Os componentes básicos são: **Model**, **View** e **Controller**.

Padrão MVC

- As vantagens apontadas são:
  - Separação de preocupações
  - Reusabilidade
  - Escalabilidade
  - Testabilidade
  - Manutenabilidade
- A ideia por trás do uso de padrões de projeto reside na experiências de muitos estudos de caso e a reconhecida eficácia na reutilização da solução (padrão).

Padrão MVC

- Os componentes Model-View-Controller tem responsabilidades distintas como resumido abaixo:
  - **Model**: responsável pela lógica de negócio e trato com os dados da aplicação;
  - **View**: responsável pela apresentação dos dados da aplicação aos clientes;
  - **Controller**: orquestra a interação entre as partes Model e View.
- Estes componentes podem ser vistos como *camadas* distintas na estruturação do código.

Padrão MVC

- Em um contexto de divisão clara entre frontend e backend. Podemos compreender os componentes do MVC da seguinte forma:
  - **Model**: lógica da aplicação que reside no backend
  - **View**: apresentação de dados que reside no frontend
  - **Controller**: orquestração entre frontend e backend

Padrão MVC e Frameworks

Padrão MVC e Frameworks

- Muitos frameworks para desenvolvimento de aplicações Web tem na sua definição arquitetural o padrão MVC.
- O exemplo mais notório é o Laravel.
- Você deve estar se perguntando sobre o Flask e o Django?

Padrão MVC e Frameworks

- O Flask não é estruturado com base em uma arquitetura definida.
- De acordo com a documentação, neste link, o flask não assume algumas decisões de projetos.
- Portanto, há liberdade total para decisões sobre como a aplicação será estruturada. Que camadas deve possuir entre outros fatores.
- Isso não impede que um projeto em Flask assuma uma arquitetura MVC e incorpore soluções já referenciadas pela comunidade.

Padrão MVC e Frameworks

- E quanto ao Django? Aí já temos uma história diferente do Flask.
- O Django implementa uma derivação do MVC que é comumente chamada de MVT (Model-View-Template).
- A mudança é sutil, mas importante. Os novos componentes são:
  - Model: continua com a mesma semântica do MVC;
  - View: tem um papel relacionado ao de controlador, contém a parte da lógica de negócio.
  - Template: camada de apresentação de dados aos clientes (como a View no MVC).

Padrão MVC e Frameworks

- É importante observar que os frameworks não definem os padrões. É exatamente ao contrário.
- O padrão MVC é uma abstração que pode ser utilizada em um framework como Flask de modo a se reutilizar boas práticas e soluções para problemas recorrentes.
- Desta maneira, se justifica, por exemplo, a mudança incorporada no Django onde alguns elementos do MVC clássico estão definidos.

Implementação de MVC com Flask - Exemplo 2

Estudo de Caso 1

- Neste exemplo, temos uma aplicação de cadastro de usuários.
- A estrutura do projeto está conforme ilustrado abaixo:

Exemplo

```
case/
|-- templates/
|   |-- index.html
|   `-- register.html
|-- database
|-- app.py
`-- iniciar_db.py
```

Estudo de Caso 1

- Considerando os componentes do MVC, podemos observar de imediado que os templates podem ser classificados como sendo pertencentes a camada de visão.
- Esta camada lida com a apresentação dos dados ao usuário e pode ser implementada de diferentes formas: páginas web, aplicações desktop ou mobile.
- Observe que estamos analisando a estrutura do projeto utilizando a abstração referente ao MVC.

Estudo de Caso 1

- Agora, temos um desafio mais importante que é o seguinte: onde ficam os controladores e modelos neste exemplo.
- Essa divisão de responsabilidades não fica muito clara quando não adotamos o padrão MVC na construção do software.
- Logo, teremos no arquivo `app.py` partes do código que são de responsabilidades típicas de um Controlador e também aquelas que são típicas de um Modelo.
- Entretanto, isso não está explícito no código.

Estudo de Caso 1

- O papel de um modelo é servir de camada de acesso a dados e também servir como ponto para inclusão de lógica da aplicação. Observe o código abaixo:

Exemplo

```
@app.route('/')
def index():
    conn = get_connection()
    users = conn.execute("SELECT * FROM users").fetchall()
    return render_template('index.html', users = users)
```

- Na definição dessa rota, temos várias coisas acontecendo:
  - uma requisição chegou e precisa ser tratada;
  - é necessário acessar dados no banco;
  - é necessário preparar a resposta para o usuário com os dados obtidos.
- Consegue perceber que temos responsabilidades diferentes nesta simples operação?
- Quem deve receber as requisições? Quem deve interagir com o banco quando for necessário? Quem prepara a resposta?

- As respostas são as seguintes:
  - Quem deve receber as requisições? **Controlador**;
  - Quem deve interagir com o banco quando for necessário? **Modelo**;
  - Quem prepara a resposta? **Controlador**
- Você deve estar se perguntando o seguinte? Então o MVC é somente saber quem vai ser responsável por cada etapa do processo de interação do usuário com a aplicação?
- Sim, é necessário saber quem faz o que (Separação de preocupações - *Separation of concerns*).

- Entretanto, é necessário ir mais além. É preciso estruturar a aplicação para que esta separação fique clara e fácil de manter (manutenção e teste do código).
- Como fazer isso? Bem... Alguns frameworks já são construídos com base no princípio da separação de preocupações e até implementam o MVC.
- A proposta do Flask é entregar o mínimo e deixar o programador aplicar os padrões que desejar, assim como incluir os recursos que achar necessário.

Estudo de Caso 1

- Sim... antes de esquecer o Estudo de Caso 1, vale salientar que não temos um Controlador em si.Tampouco temos um modelo Usuário.
- Mesmo assim, temos acesso ao banco de dados da aplicação permite realizar consultas e registros de novos usuários.

Exemplo

```
# trecho de código de acesso ao banco no arquivo app.py
conn = get_connection()
conn.execute("INSERT INTO users(email, nome) VALUES (?,?)", (email, nome))
conn.commit()
conn.close()
```

Implementação de MVC com Flask - Exemplo 2

Estudo de Caso 2

- Neste exemplo, vamos assumir algumas coisas básicas:
  - Utilizaremos módulos e pacotes(revisar)
  - Vamos estruturar a aplicação para incorporar os conceitos do MVC
- O exemplo continua sendo o problema do registro de usuários do Estudo de Caso 1.
- Entretanto, vamos ter uma estrutura de projetos bem diferente e interessante. Veja o próximo slide.

O projeto está disponível aqui - Versão rar

Exemplo

```
case/
|-- templates/
|   `-- users/
|       |-- register.html
|       `-- index.html
|-- controllers/
|   |-- __init__.py
|   `-- UserController.py
|-- models/
|   |-- __init__.py
|   `-- user.py
|-- database
|-- templates
|-- app.py
`-- __init__.py
```

- Como você deve ter observado, temos alguns diretórios e arquivos novos no projeto.
- A pasta `controllers` assim como a pasta `models` são bem sugestivas. Elas armazenam os controladores e modelos da aplicação, respectivamente. Em breve vamos examiná-las.
- A pasta `templates` continua sendo local padrão para a busca de arquivos HTML no projeto. Apenas adicionamos uma pasta `users` para indicar os arquivos a respeito de usuários.
- Por fim, temos alguns arquivos `__init__.py` que indicam que estamos tratando de pacotes. Veremos mais sobre isso.

- Para executar esta aplicação continuaremos como antes:

Exemplo

```
flask run --debug
```

- No entanto, se você observar o conteúdo do arquivo `app.py` notará uma diferença:

Exemplo

```
from case2 import app
from case2.controllers import UserController
```

- Este é o novo arquivo `app.py` que temos para o Estudo de Caso 2.

- O código abaixo é do arquivo `__init__.py` que está na pasta `case2`:

Exemplo

```
from flask import Flask
app = Flask(__name__)
```

- Criamos um pacote chamado case2 (pasta do projeto). Adicionamos `__init__.py` e definimos a variável da aplicação(`app`).

- Desta maneira, no arquivo `app.py` podemos fazer referência a variável `app`.

Exemplo

```
# arquivo app.py
from case2 import app
from case2.controllers import UserController
```

- No mesmo arquivo da apliação (`app.py`) temos um segundo import que é `UserController` .
- Seguimos o mesmo princípio de importação de módulos. Neste caso importamos o módulo `UserController` do pacote `controllers` .
- Note que o pacote `controllers` é um pacote dentro de `case2` .
- O pacote controller tem o seguinte conteúdo:
  - Um arquivo `__init__.py` e um arquivo `UserController.py`
- O arquivo `controllers\__init__.py` contém apenas um import para `UserController.py` , tornando o módulo disponível quando o pacote é utilizado.

Controladores

- No código desta aplicação, temos de fato uma separação clara das funções de um controlador.
- Fisicamente, há uma pasta para inclusão dos controladores.
- Esta pasta chamada `controllers` é um pacote e possui dois arquivos: `__init__.py` e `UserController.py`.
- O arquivo mais interessante neste momento é o `UserController.py`

Exemplo

```python
# Código de UserController.py
from flask import render_template, redirect, url_for, request, flash
from case2 import app
from case2.database import get_connection
from case2.models.user import User

@app.route('/users')
def index():
    return render_template('users/index.html', users=User.all())

@app.route('/register', methods=['POST', 'GET'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        nome= request.form['nome']
        if not email:
            flash('Email é obrigatório')
        else:
            user = User(email, nome)
            user.save()
            return redirect(url_for('index'))
    return render_template('users/register.html')
```

- Você pode estar se peguntando, então esse é controlador?
- Sim. Note que o controlador tem o papel de orquestrar a recepção da requisição, repassar para o modelo algum pedido de dados. Receber esses dados do modelo e repassar para camada de visão.
- Quando uma requisição POST chega para `register` o controlador obtém os dados da requisição e a partir daí a camada de modelo opera. Criando um novo usuário e salvando os dados no banco.

Exemplo

```
# trecho de register em UserController.py
user = User(email, nome)
user.save()
```

- O próximo passo é devolver o comando da operação para camada de visão que vai preparar a página de retorno para o usuário.
- No exemplo, este passo é um redirecionamento para página de listagem de usuários.

Exemplo

```
# trecho de register em UserController.py
return redirect(url_for('index'))
```

- Observe que estas divisões de camadas são abstratas. Vamos ter código de camadas diferentes trabalhando no mesmo bloco de código. Entretanto, estão logicamente separadas.

- Observe que no controlador importamos classes e funções de outras camadas, como é o caso do importe para o modelo de usuários.

Exemplo

```
from case2.models.user import User
```

- Esta classe é a responsável, neste estudo, por interagir com o banco de dados.
- Ela está definida em outro local da aplicação que examinaremos agora.

Modelos

- O único modelo deste exemplo está na pasta models.
- Esta pasta também é tratada como um pacote e contém os seguintes arquivos: `__init__.py` e `user.py`
- O arquivo `__init__.py` não possui código até o momento.
- O conteúdo de `user.py` consiste da classe que representa um Usuário: `User`.

Exemplo

```python
# Código-Fonte de User
from case2.database import get_connection

class User:
    def __init__(self, email, nome):
        self.email = email
        self.nome = nome

    def save(self):
        conn = get_connection()
        conn.execute("INSERT INTO users(email, nome) values(?,?)", (self.email, self.nome))
        conn.commit()
        conn.close()
        return True

    @classmethod
    def all(cls):
        conn = get_connection()
        users = conn.execute("SELECT * FROM users").fetchall()
        return users
```

- Observe que já trabalhamos com modelos com representação semelhante a este exemplo.
- O modelo em questão possui acesso ao banco de dados ao usar a função `get_connection` que é importada.
- Podemos fazer um exercício simples de imaginação e considerar que, de acordo com o problema, novas classes serão definidas.
- O local para armazenar estas novas classes é justamente no pacote de modelos.
- Onde cada modelo tem sua responsabilidade bem definida.

Visão

- Neste exemplo, a camada de visão é bem simples.
- Nela temos os templates da aplicação (arquivos HTML).
- Algumas funções ajudam a camada de visão a entregar o resultado final processado através do uso do controlador e modelo. Como é o caso da função `render_template`.
- A função `url_for` também tem um papel interessante na camada de visão.

Resumo

O padrão Model-View-Controller (MVC) é uma arquitetura amplamente utilizada no desenvolvimento de software que busca a separação de responsabilidades dentro de uma aplicação. Seus principais componentes são:

- Model – Lida com a lógica de negócio e o acesso aos dados.
- View – Responsável pela apresentação das informações ao usuário.
- Controller – Atua como intermediário, recebendo requisições, coordenando o fluxo entre o Model e a View e retornando as respostas.

> As vantagens do MVC incluem: manutenibilidade, reutilização, testabilidade e escalabilidade.