

Flask

Desenvolvendo aplicações Web com o Framework Flask - Aula 4

Um pouco mais sobre Templates com Jinja2 – Parte 2

Flask – Templates com Jinja2

O que veremos nesta aula

Nesta aula serão abordados conceitos básicos de templates Jinja2 à partir da perspectiva do Flask. Também veremos como criar aplicativos com templates modulares e extensíveis.

Veremos:

- Templates Jinja2
- Usar o Bootstrap
- Composição de bloco e herança de layout
- Criando um processador de contexto customizado
- Criando um filtro personalizado Jinja2
- Criando uma macro personalizada para formulários
- Formatação de data e hora com Moment.js

Flask – Templates com Jinja2

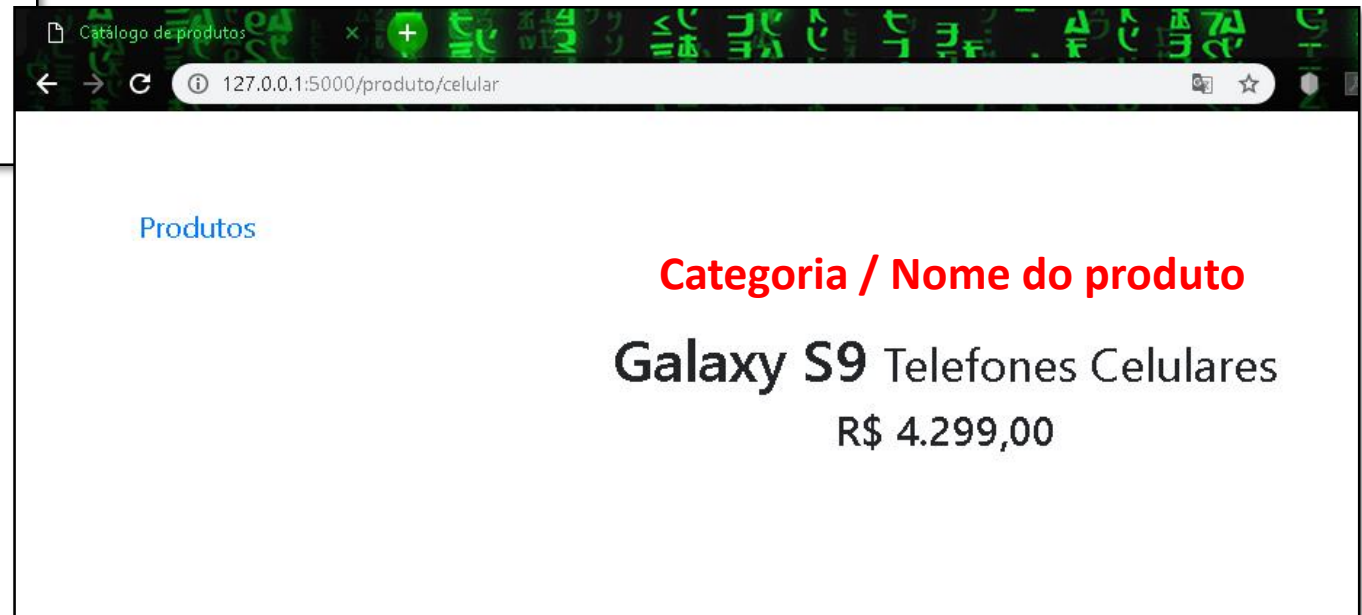
Criando um processador de contexto customizado

Às vezes, podemos querer calcular ou processar um valor diretamente nos modelos.

Jinja2 mantém a noção de que o processamento da lógica deve ser tratado nas visualizações (*views*) e não nos modelos (*models*) e, portanto, mantém os modelos limpos. Um processador de contexto se torna uma ferramenta útil nesse caso. Podemos passar nossos valores para um método; isso será processado em um método Python e nosso valor resultante será retornado. Portanto, estamos essencialmente apenas adicionando uma função ao contexto do modelo (graças ao *Python* por nos permitir passar funções como qualquer outro objeto).

Flask – Templates com Jinja2

Criando um processador de contexto customizado



Flask – Templates com Jinja2

Criando um processador de contexto customizado

Veja por exemplo, como faríamos para exibir o nome do produto desta forma:

Categoria / Nome do produto

```
@product_blueprint.context_processor
def my_processor():
    def nome_completo(produto):
        return '{0} / {1}'.format(produto['categoria'], produto['descricao'])
    return {'nome_completo': nome_completo}
```

Um contexto é simplesmente um dicionário que pode ser modificado para adicionar ou remover valores. Qualquer método decorado com `@product_blueprint.context_processor` deve retornar um dicionário que atualize o contexto real.


Flask – Templates com Jinja2

Criando um processador de contexto customizado

Podemos usar o processador de contexto anterior da seguinte forma: `{{ nome_completo(produto) }}`

Vamos adicionar ao nosso aplicativo para a listagem de produtos (em `my_app/templates/produto.html`) da seguinte maneira:

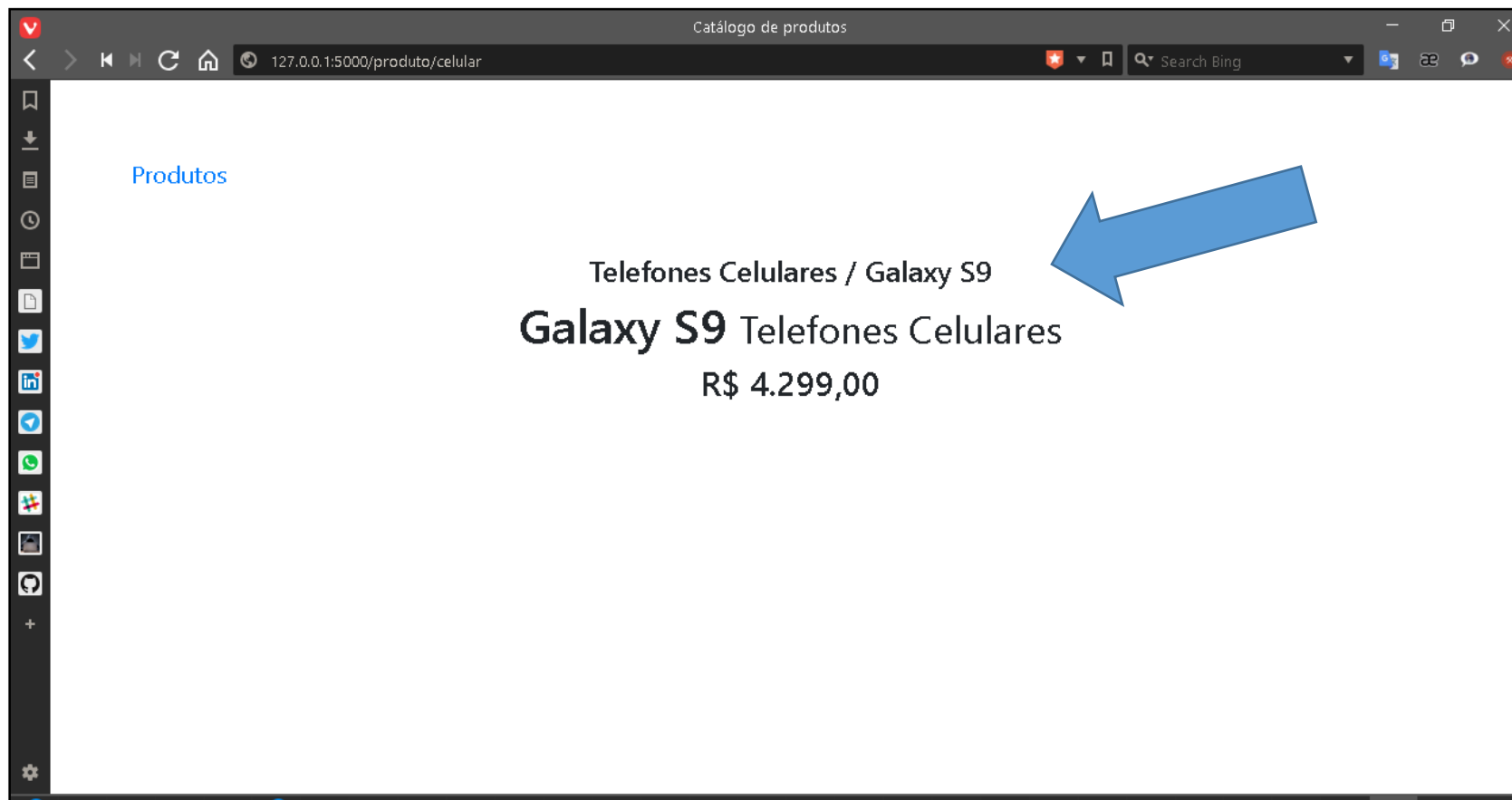
```
{% extends 'home.html' %}
{% block container %}
    <div class="top-pad">
        <h4>{{ nome_completo(produto) }}</h4>
        <h1>{{ produto['descricao'] }}
            <small>{{ produto['categoria'] }}</small>
        </h1>
        <h3>R$ {{ produto['valor'] }}</h3>
    </div>
{% endblock %}
```



Flask – Templates com Jinja2

Criando um processador de contexto customizado

Veja o resultado:



Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Variáveis podem ser modificadas por filtros. Os filtros são separados da variável por um símbolo de pipe (|) e podem ter argumentos opcionais entre parênteses.

Por exemplo, para colocar a primeira letra de uma variável em maiúsculas podemos usar `{{ variavel|capitalize }}`

Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Veja alguns filtros comumente usados, disponibilizados pelo Jinja2.

Nome do filtro	Descrição
safe	Renderiza o valor sem aplicar escape.
capitalize	Converte o primeiro caractere do valor para letra maiúscula e o restante para letras minúsculas.
lower	Converte o valor para letras minúsculas.
upper	Converte o valor para letras maiúsculas.
title	Converte o primeiro caractere de cada palavra do valor para letra maiúscula.
trim	Remove espaços em branco no início e no final do valor.
striptags	Remove qualquer tag HTML do valor antes de renderizar.

Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

O filtro *safe* é interessante e merece destaque. Por padrão, o *Jinja2* escapa todas as variáveis por questões de segurança. Por exemplo, se uma variável estiver definida com o valor '`<h1>Hello</h1>`', o *Jinja2* renderizará a *string* como '`<h1>Hello</h1>`', o que fará o elemento `h1` ser exibido, e não interpretado pelo navegador. Muitas vezes, será necessário exibir o código *HTML* armazenado em variáveis, e, nesses casos, o filtro *safe* deve ser usado. Mas, jamais use o filtro *safe* em valores que não são confiáveis, por exemplo, em um texto fornecido por usuários em formulários *web*.

Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Agora vamos ver como criar um filtro personalizado. Não precisamos necessariamente criar um processador de contexto como fizemos anteriormente para exibir o descritivo do produto. Podemos simplesmente escrever um filtro para obter o mesmo resultado; Isso tornará as coisas muito mais limpas. Um filtro pode ser gravado para exibir o nome descritivo do produto, conforme mostrado a seguir:

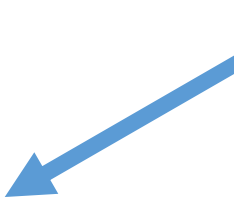
```
@product_blueprint.app_template_filter('nome_completo')
def nome_completo_filtro(produto):
    return '{0} / {1}'.format(produto['categoria'], produto['descricao'])
```

Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Podemos usar o filtro desta forma:

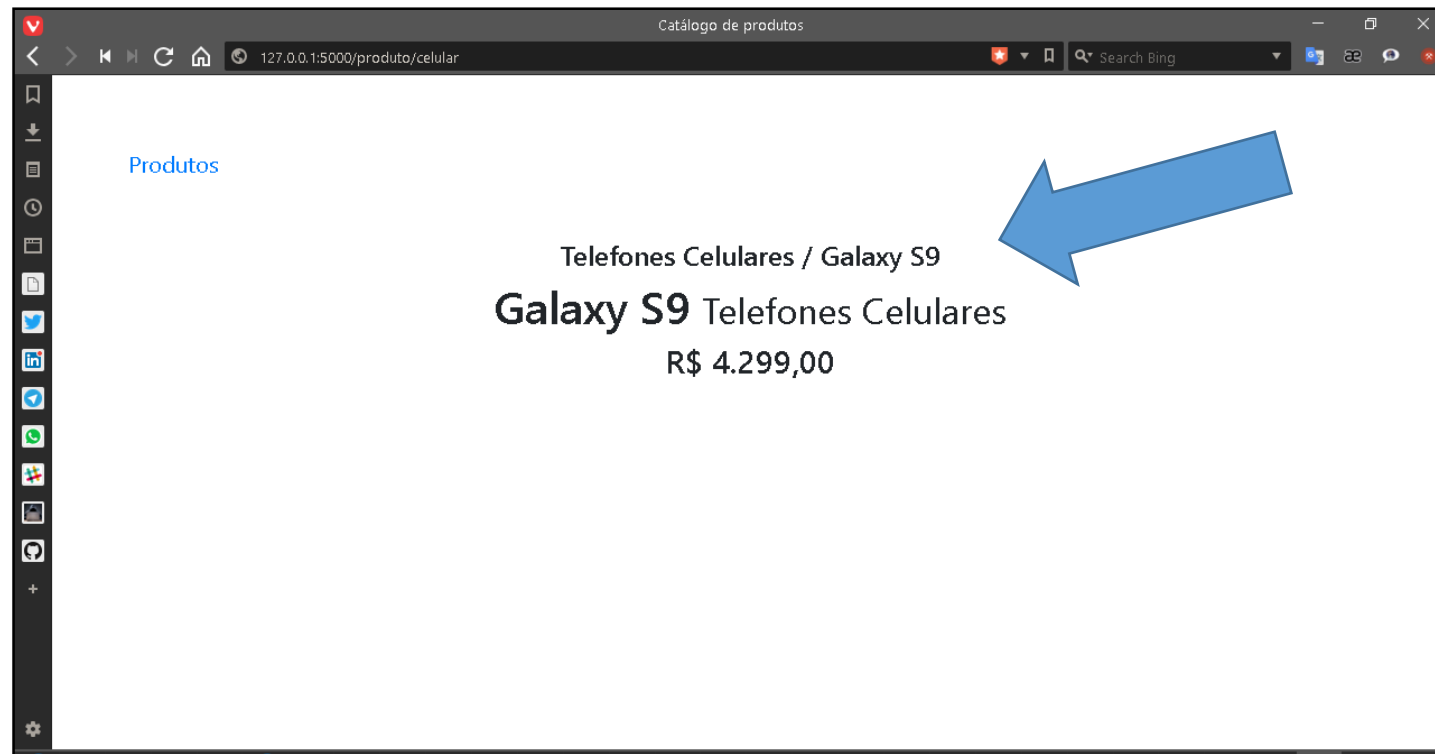
```
{% extends 'home.html' %}
{% block container %}
    <div class="top-pad">
        <h4>{{ produto|nome_completo }}</h4>
        <h1>{{ produto['descricao'] }}
            <small>{{ produto['categoria'] }}</small>
        </h1>
        <h3>R$ {{ produto['valor'] }}</h3>
    </div>
{% endblock %}
```



Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

O resultado será o mesmo obtido pelo processador de contexto que criamos no exemplo anterior.



Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Vamos agora utilizar um filtro para formatar o valor do produto com base no idioma local.

Para isso vamos utilizar o módulo *ccy*, que é um módulo *Python* para formatar valores monetários. Este módulo compila um dicionário de objetos monetários contendo informações úteis na análise financeira.

Primeiro vamos instalar o *ccy* usando:

\$ pip install ccy

<https://pythonhosted.org/ccy/currencies.html>

Flask – Templates com Jinja2

Criando um filtro personalizado Jinja2

Temos que alterar nossa view acrescentando a importação do `ccy` e o código a seguir:

```
import ccy
from flask import request
```

```
@product_blueprint.app_template_filter('formato_moeda')
def formato_moeda_filtro(valor):
    codigo_moeda = ccy.countryccy('US') # 'BR'
    return '{0} {1}'.format(codigo_moeda, valor)
```

Agora temos que trocar em `produto.html` a marcação que exibe o valor de:

```
<h3>R$ {{ produto['valor'] }}</h3>
```

Para:

```
<h3>{{ produto['valor']|formato_moeda }}</h3>
```

CONTINUA...