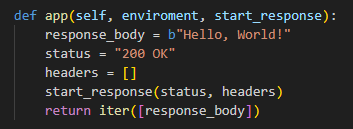
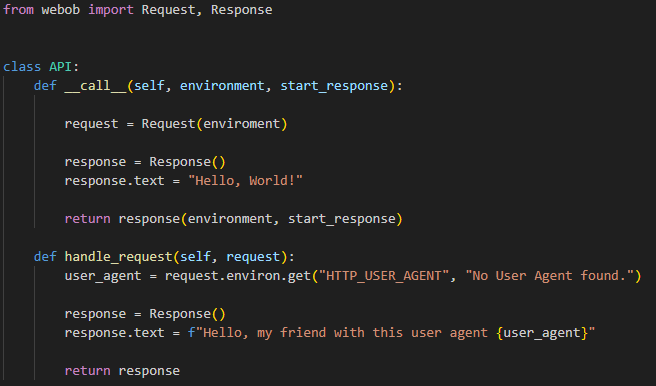
**ANOTAÇÕES PARA O PROJETO PRINCIPAL**



Essa é a forma mais básica de criar um código para um framework web que retornará um “*Hello, World!*” em uma página web. A ferramenta usada para criar o servidor WSGI foi a waitress. Segue abaixo outras anotações:

**environment:** contém várias informações importantes do cliente: url informada, navegador, parâmetros, IPs e outros itens relacionados

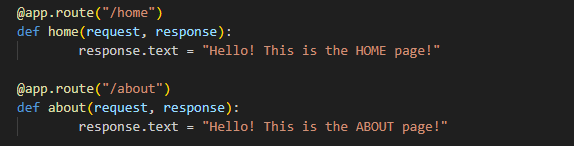
**start\_response:** é uma função callback que é necessária para iniciar uma resposta HTTP, seguido dela temos a resposta de conteúdo que é retornada pelo servidor.

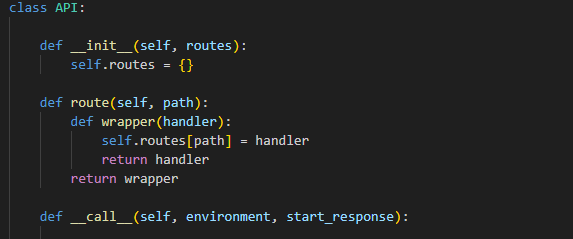


**WebOb:** Biblioteca que fornece objetos HTTP, que é o caso do código acima é Request e Response, para lidar com requisições e respostas do servidor.

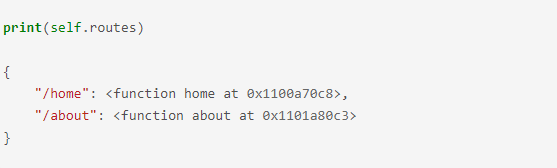
**\_\_call\_\_:** Uma função que faz o mesmo que a função app mostrada anteriormente, mas esta é feita utilizando os objetos Request e Response da biblioteca WebOb.

**handle\_request:** Função para lidar com requests. (Obs.: Na figura acima, o “response = Response(environment)” deveria ser “response = self.handle\_request(request)” para rodar direito e receber a função.

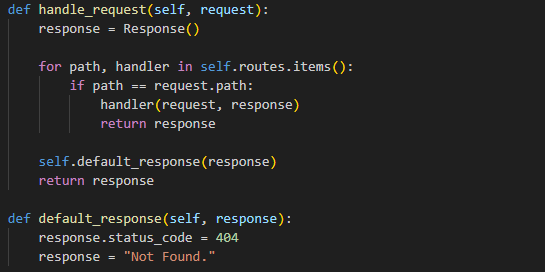




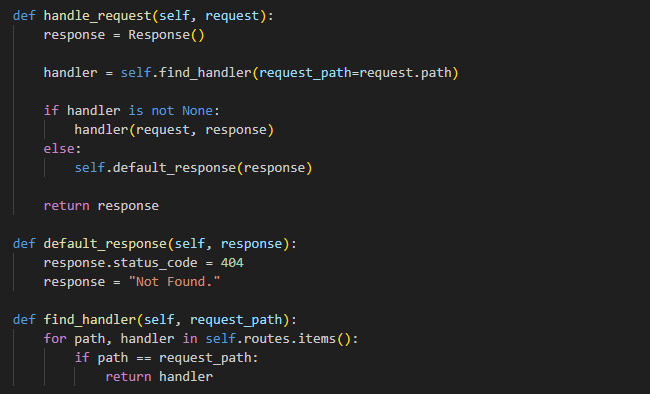
Na figura acima, estamos criando novas rotas que dão acesso a diferentes páginas da mesma aplicação. A função **route** tem o dever de armazenar paths com chaves e handlers como valores das chaves. Ficando mais ou menos assim:



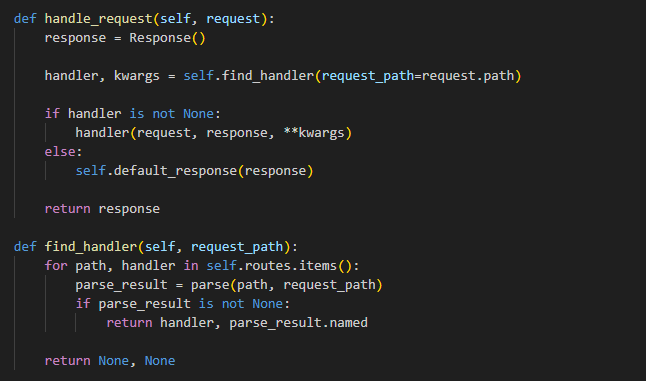
Agora, quando uma request chega, precisamos verificar seu path, encontrar um handler apropriado, chamar esse handler e retornar uma response apropriada. Como vocês podem verificar abaixo:



Como podem ver, a função recebe um request, verifica se existe um path que bate com esse request, então invoca um handler que dará uma response específica. Além disso, foi criada uma função de erro padrão, a **default\_response**, ela que informará que a página não foi encontrada.

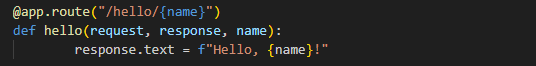


Essa função **find\_handler** é só uma forma de encontrar o handler caso não tenha nenhum. Além disso, ele diminui o trabalho da função **handle\_request**.

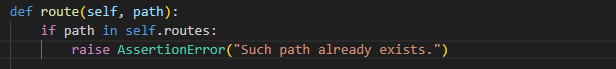


**\*\*kwargs:** Variável que permite a passagem de vários argumentos para alguma função.

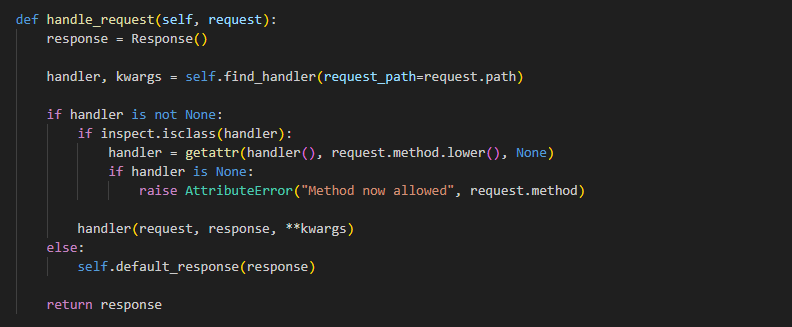
**parse:** Veio da biblioteca de mesmo nome serve para relacionar informações textuais, no caso, para saber qual nome exibir na página hello/{nome}.

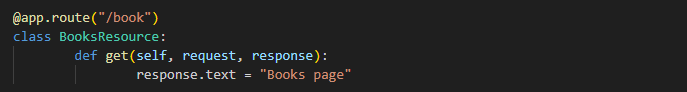


A nova função **find\_handler** retorna seu handler como anteriormente, mas também retorna um parse\_result (Esse que será usado para exibir o nome). Já na nova função **handle\_request** adicionamos o \*\*kwargs, que receberá as quantas variáveis textuais de nome o usuário quiser e que serão exibidas na página hello/{nome}.

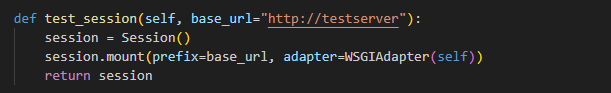


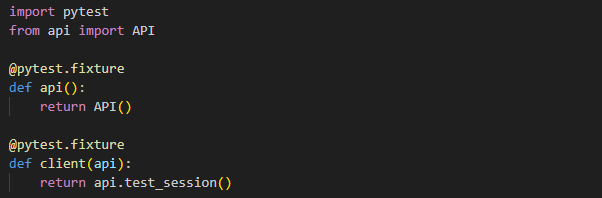
Só uma condição para casa o criador do código criar mais de uma rota com o mesmo nome.

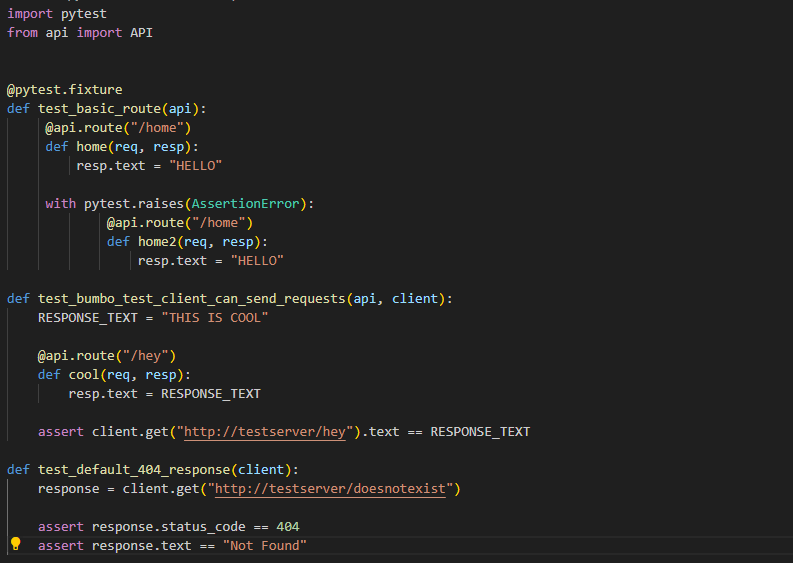




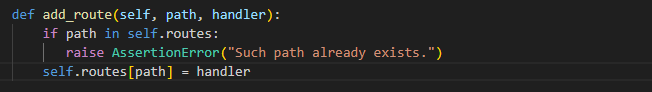
Essa parte foi criada para que fosse possível adicionar rotas no estilo de classes dentro do framework.

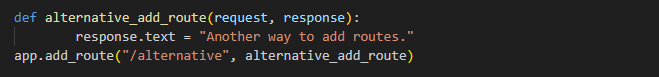


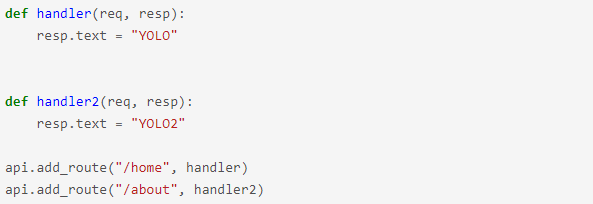




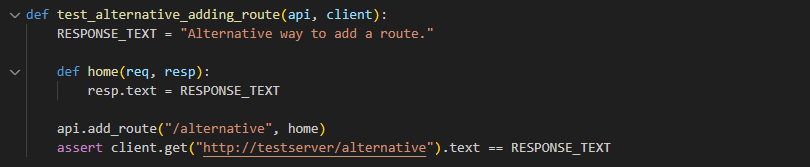
Confesso que não entendi completamente para que serve essa parte. Mas aparentemente são para Unit Tests, uma forma de testar routes sem a necessidade de rodar o servidor. O primeiro código está em api.py, o segundo em conftest.py e o terceiro em framework\_test.py.



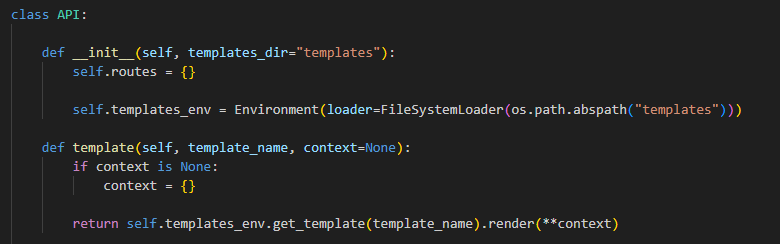


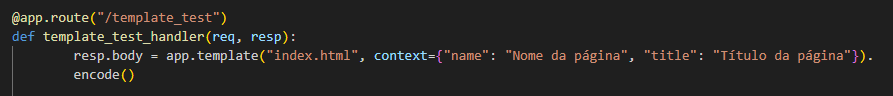


Também foi adicionado uma maneira alternativa de adicionar routes. O criador do tutorial queria fazer algo mais parecido com o Django.



Inclusive foi feito um teste para saber se essa funcionalidade estava funcionando, e ele retornou tudo certo.





Foi adicionado um suporte para templates utilizando as bibliotecas **OS** e **Jinja2**. Primeiro estabelecemos um atributo da classe API para armazenar os template.

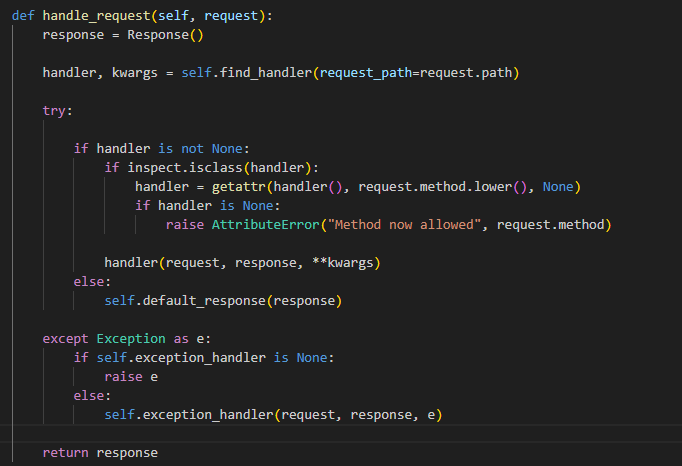
**Environment():** função para ajudar a carregar templates no código.

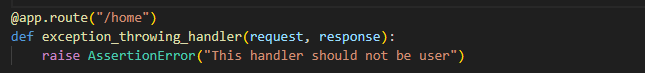
**FileSystemLoader():** Carrega templates direto do sistema ou diretório local.

A variável **template\_name** representa o nome do template que será passado e a **context** serve para receber variáveis que poderão ser usadas no template. A propósito, no final da linha com o resp.body (não é resp.text) o **.encode()** serve porque o resp.body espera por bytes, mas a função template retorna uma string unicode.

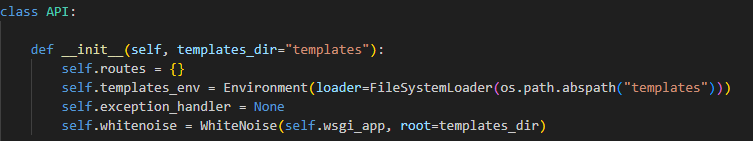
Sobre o template, é só criar um arquivo .html comum e armazenar ele na pasta templates, no mesmo diretório, você poderá retirar qualquer template dela.

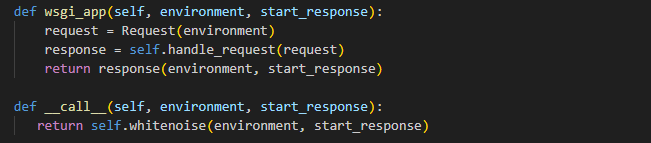






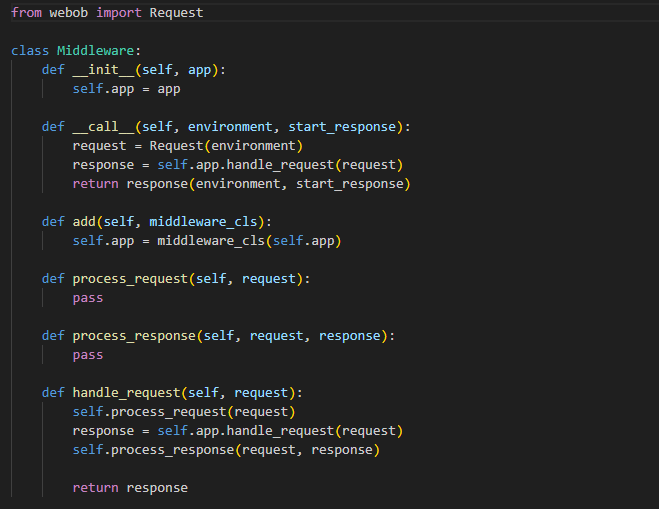
Essa parte se refere a criação de exception handlers customizáveis. Podendo ser mensagens de erro como “Algo deu errado! Por favor contate nosso suporte.” ou qualquer página do gênero.



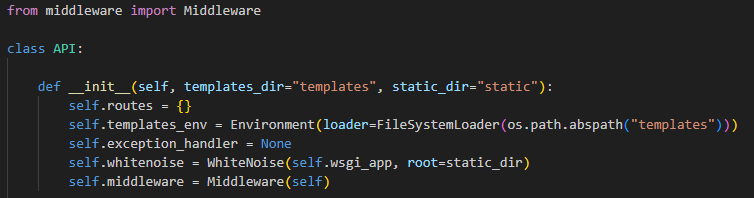


Sobre o código acima, ele serve para dar suporte a arquivos estáticos que serão usados pelos arquivos .html, ou seja, permite arquivos javascript ou css, por exemplo. Foi utilizada a biblioteca **WhiteNoise** para isso. Para testar, basta interligar os arquivos a partir do arquivo .html que será escolhido de template para as páginas (No meu caso, eu selecionei o diretório dos arquivos estáticos como sendo o diretório “template” mesmo, mas você pode mudar para deixar mais organizado).

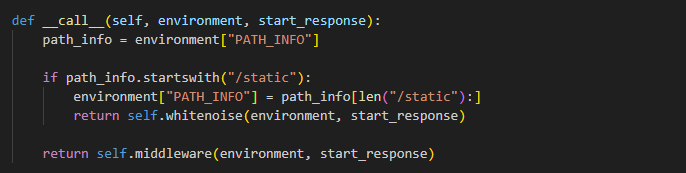
# middleware.py



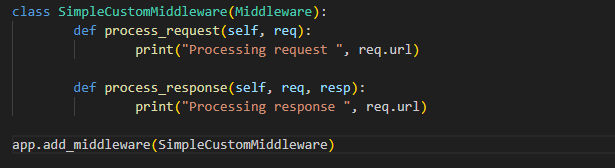
# api.py



# api.py



# app.py



Por fim, o tutorial mostra como criar middlewares simples para a aplicação. Middlewares são wrappers em torno de nosso aplicativo WSGI que têm a capacidade de modificar solicitações e respostas. Middlewares são componentes que pode modificar uma solicitação e/ou resposta HTTP e é projetado para ser encadeado para formar um pipeline de mudanças comportamentais durante o processamento da solicitação. Exemplos de tarefas de middleware podem ser registro de solicitações e autenticação HTTP.

Na função \_\_call\_\_ em api.py, houve uma modificação para a aplicação entender quando se deve chamar a função WhiteNoise(). Já em app.py, temo exemplo de como criar um middleware simples que notifica todas as requisições da aplicação no terminal.