



Desenvolvimento Web com Python e

django

AGRADECIMENTOS

Agradeço primeiramente à **Deus**, pois sem Ele, eu não teria a capacidade e força para entender nem uma linha de código sequer.

Agradeço à minha **linda esposa**, Luiza, pelo infinidável ânimo e incentivo às minhas loucas iniciativas e à minha família pelo amor e suporte.

Agradeço ao grupo do Facebook **Python Brasil – Programadores**, pelo apoio e suporte à comunidade Python!

SOBRE O AUTOR



Olá pessoal! Aqui quem vos fala é o **Vinícius!**

Bom, minha paixão por computadores e tecnologia despertou desde cedo, quando ainda era bem novo.

Assim como você, **entusiasta de tecnologia**, eu sempre gostei de brinquedos "tecnológicos" (como eu me divertia jogando **Mister Show** e **Pense Bem** com a minha família).

Mas minha brincadeira ia além disso! Eu adorava ver como eles funcionavam.

E como fazer isso sem abrí-los?! **Certo?**

Bom... Isso revoltava minha mãe, que via aquele brinquedo que tinha demorado tanto para escolher, em "pedaços".

Cabos, motores, capacitores e baterias espalhados sempre fizeram parte da decoração do meu quarto.

Mas minha trilha na computação estava apenas começando.

Chegou a época de escolher minha graduação e, como sempre fui fascinado pelo funcionamento dos meus brinquedos eletrônicos, optei por cursar Engenharia de Computação na Universidade de Brasília.

Dentre outras coisas, o curso serviu para aumentar **ainda mais** minha vontade de aprender e desvendar a computação.

Quem nunca se perguntou como pode um processador conter mais de 1 bilhão de transistores? Como 0's e 1's conseguem controlar toda essa máquina extremamente intrigante que é um computador? Essas e outras várias perguntas sempre fazem parte do dia a dia do universo de quem tem sede de aprender um pouco mais sobre computação e tecnologia.

Durante o curso, tive experiência na área de suporte e na área de desenvolvimento, estagiando na própria Universidade e em órgãos públicos. Sem contar na oportunidade de aprender com grandes professores e instrutores que tive por lá.

Os anos se passaram, me formei e estava na hora de decidir o rumo da minha vida profissional, e como bom brasiliense que sou, optei pela

vida de concursado, fazendo concursos de TI para Tribunais, órgãos públicos e empresas públicas.

Dado meu esforço, não demorou muito e passei no concurso para área de **Tecnologia do Banco do Brasil**.

Atualmente eu estou nessa nova empreitada na Python Academy para produzir e disponibilizar conteúdo da mais alta qualidade.

Espero que nosso conteúdo faça você entender Python **DE VERDADE!**

É isso pessoal!

VÁ DIRETO AO ASSUNTO

INTRODUÇÃO.....	8
O FRAMEWORK DJANGO	9
FLUXO DE UMA REQUISIÇÃO NO DJANGO	10
INSTALAÇÃO	12
<i>HELLO WORLD, DJANGO!</i>	13
CONCLUSÃO DO CAPÍTULO.....	22
CAMADA MODEL	23
ONDE ESTAMOS.....	23
CAMADA MODEL.....	25
DB BROWSER FOR SQLITE.....	33
API DE ACESSO A DADOS	35
CONCLUSÃO DO CAPÍTULO.....	38
CAMADA VIEW	39
ONDE ESTAMOS.....	40
CAMADA VIEW.....	41
FUNÇÕES VS CLASS BASED VIEWS	44
FUNÇÕES (<i>FUNCTION BASED VIEWS</i>)	47
CLASSES (CBV - <i>CLASS BASED VIEWS</i>)	53
FORMS.....	65
MIDDLEWARES.....	72
CONCLUSÃO DO CAPÍTULO.....	79

CAMADA TEMPLATE	80
ONDE ESTAMOS.....	81
DEFINIÇÃO DE <i>TEMPLATE</i>	82
CONFIGURAÇÃO	84
Django <i>TEMPLATE LANGUAGE</i>	85
TAGS E FILTROS CUSTOMIZADOS.....	108
BUILT-IN FILTERS	117
CÓDIGO	127
CONCLUSÃO DO CAPÍTULO.....	127
E AGORA?.....	128
REFERÊNCIA	130

INTRODUÇÃO

Django é um *framework* de alto nível, escrito em Python que encoraja o desenvolvimento limpo de aplicações *web*.

Desenvolvido por experientes desenvolvedores, Django toma conta da parte pesada do desenvolvimento *web*, como tratamento de requisições, mapeamento objeto-relacional, preparação de respostas HTTP, para que, dessa forma, você gaste seu esforço com aquilo que **realmente interessa**: suas **regras de negócio!**

Foi desenvolvido com uma preocupação extra em **segurança**, evitando os mais comuns ataques, como *Cross site scripting* (XSS), *Cross Site Request Forgery* (CSRF), *SQL injection*, entre outros.

É bastante **escalável**: Django foi desenvolvido para tirar vantagem da maior quantidade de hardware possível (desde que você queira). Django usa uma arquitetura “zero-compartilhamento”, o que significa que você pode adicionar mais recursos em qualquer nível: servidores de banco de dados, cache e/ou servidores de aplicação.

Para termos uma boa noção do Django como um **todo**, esse *ebook* utiliza uma abordagem *bottom-up* (de baixo para cima): primeiro veremos os conceitos do Django, depois abordaremos a **Camada de Modelos**, depois veremos a **Camada de Views** e, por fim, a **Camada de Templates**.

O *Framework* Django

Como disse anteriormente, o Django é um *framework* para construção de aplicações web em Python.

E, como todo *framework* web, ele é um *framework* MVC (*Model View Controller*), certo?

Bem... Não exatamente!

De acordo com sua documentação, os desenvolvedores o declaram como um *framework* MTV - isto é: *Model-Template-View*.

Mas por que a diferença?

Para os desenvolvedores, as *Views* do Django representam **qual** informação você vê, não **como** você vê. Há uma util diferença.

No Django, uma *View* é uma forma de processar os dados de uma URL específica, pois ela descreve **qual** informação é apresentada, através do processamento descrito pelo desenvolvedor em seu código.

Além disso, é imprescindível separar conteúdo de apresentação – que é **onde** os *templates* residem.

Como disse, uma *View* descreve **qual** informação é apresentada, mas uma *View* normalmente delega para um *template*, que descreve **como** a informação é apresentada.

*Assim, onde o **Controller** se encaixa nessa arquitetura?*

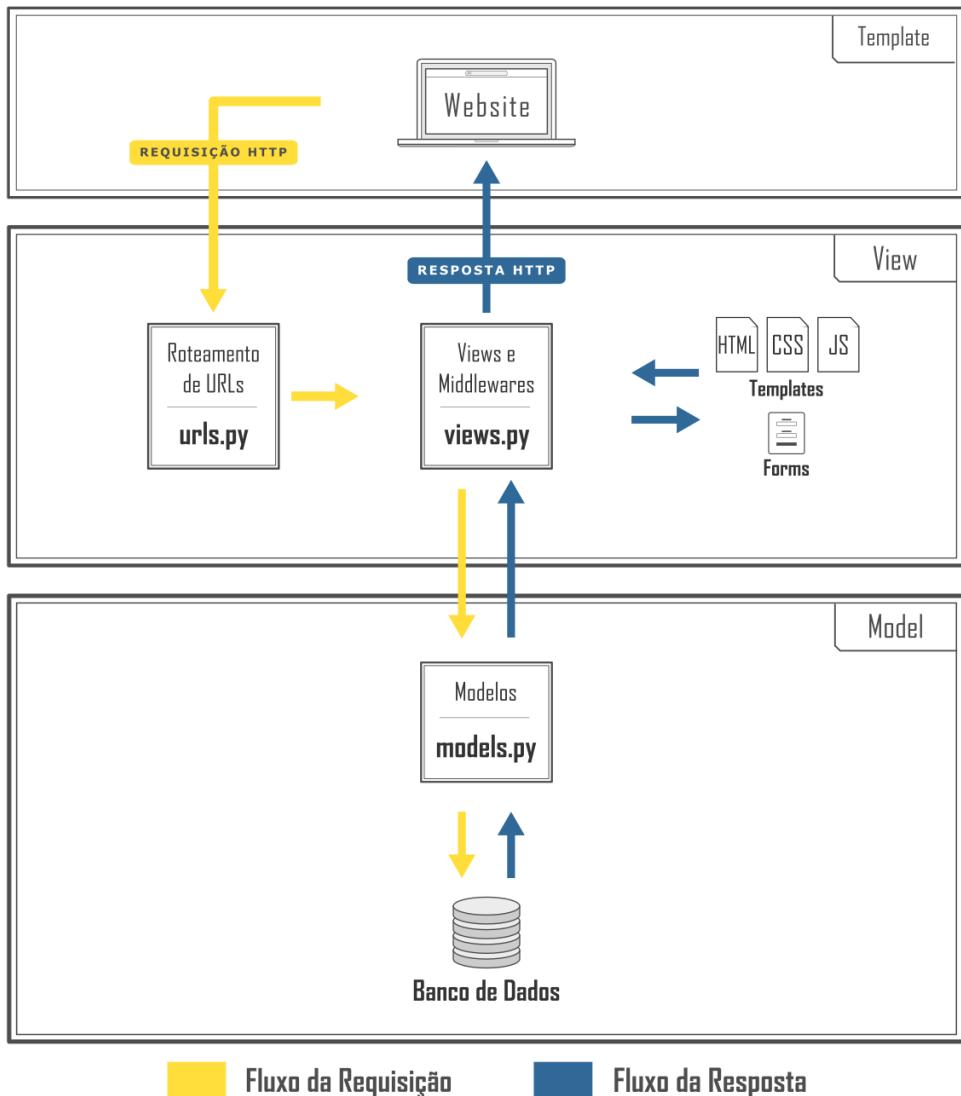
No caso do Django, é o próprio *framework* que faz o trabalho pesado de processar e rotear uma requisição para a *View* apropriada de acordo com a configuração de URL descrita pelo desenvolvedor.

Fluxo de uma requisição no Django

Para ajudar a entender um pouco melhor, vamos analisar o fluxo de uma requisição saindo do *browser* do usuário, passando para o servidor onde o Django está sendo executado e retornando ao *browser* do usuário.

Veja a seguinte ilustração:

ARQUITETURA DO django



O Django é dividido em **três camadas**:

- A Camada de **Modelos**.
- A Camada de **Views**.
- A Camada de **Templates**.

Vamos agora, dar nossos primeiros passos com o Django, começando pela sua **instalação**!

Instalação

Primeiro, precisamos nos certificar que o **Python** e o **pip** (gerenciador de pacotes do Python) estão instalados corretamente.

Vá no seu terminal ou *prompt* de comando e digite o comando **python --version**. Deve ser aberto o terminal interativo do Python (se algo como **bash: command not found** aparecer, é por que sua instalação não está correta).

Agora, digite **pip --version**. A saída desse comando deve ser a versão instalada do pip. Se ele não estiver disponível, faça o *download do instalador nesse link* e execute o código.

Vamos executar esse projeto em um ambiente virtual utilizando o **virtualenv** para que as dependências não atrapalhem as que já estão instaladas no seu computador.

Para saber mais sobre o virtualenv, [leia esse post aqui](#) sobre desenvolvimento em ambientes virtuais.

Após criarmos nosso ambiente virtual, instalamos o Django com:

```
1 pip install django
```

Para saber se a instalação está correta, podemos abrir o terminal interativo do Python (digitando `python` no seu terminal ou prompt de comandos) e executar:

```
1 >>> import django
2 >>> print(django.get_version())
```

A saída deve ser a versão do Django instalada. No meu caso, a saída foi `2.0.7`.

Hello world, Django!

Com tudo instalado corretamente, vamos agora fazer um projeto para que você veja o Django em ação!

Nosso projeto é fazer um sistema de gerenciamento de Funcionários. Ou seja, vamos fazer uma aplicação onde será possível **adicionar, listar, atualizar e deletar** Funcionários.

Vamos começar criando a estrutura de diretórios e arquivos principais para o funcionamento do Django. Para isso, o pessoal do Django fez um comando muito bacana para nós: o `django-admin.py`.

Se sua instalação estiver correta, esse comando já foi adicionado ao seu PATH!

Tente digitar `django-admin --version` no seu terminal (se não estiver disponível, tente `django-admin.py --version`).

Digitando apenas `django-admin`, é esperado que aparece a lista de comandos disponíveis, similar a:

```
1 Available subcommands:  
2  
3 [django]  
4     check  
5     compilemessages  
6     createcachetable  
7     dbshell  
8     diffsettings  
9     dumpdata  
10    flush  
11    inspectdb  
12    loaddata  
13    makemessages  
14    makemigrations  
15    migrate  
16    runserver  
17    sendtestemail  
18    shell  
19    showmigrations  
20    sqlflush  
21    sqlmigrate  
22    sqlsequencereset  
23    squashmigrations  
24    startapp  
25    startproject  
26    test  
27    testserver
```

Por ora, estamos interessados no comando `startproject` que cria um novo projeto com a estrutura de diretórios certinha para começarmos a desenvolver!

Executamos esse comando da seguinte forma:

```
1 django-admin.py startproject helloworld
```

Criando a seguinte estrutura de diretórios:

```
1 /helloworld
2     - __init__.py
3     - settings.py
4     - urls.py
5     - wsgi.py
6 - manage.py
```

Explicando cada arquivo:

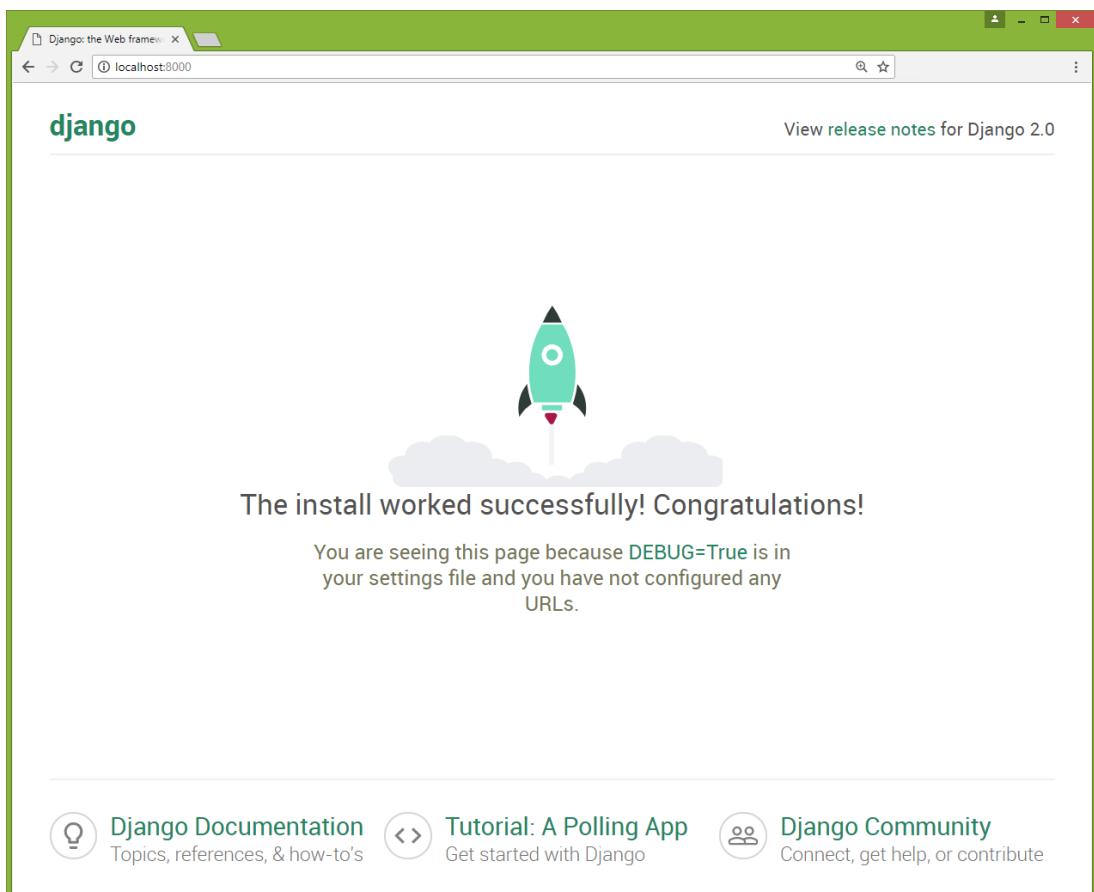
- **helloworld/settings.py**: Arquivo muito importante com as configurações do nosso projeto, como configurações do banco de dados, aplicativos instalados, configuração de arquivos estáticos e muito mais.
- **helloworld/urls.py**: Arquivo de configuração de rotas (ou URLConf). É nele que configuramos **quem** responde a **qual** URL.
- **helloworld/wsgi.py**: Aqui configuramos a interface entre o servidor de aplicação e nossa aplicação Django.

- `manage.py`: Arquivo gerado automaticamente pelo Django que expõe comandos importantes para manutenção da nossa aplicação.

Para testar, vá para a pasta raíz do projeto e execute o comando `python manage.py runserver`.

Depois, acesse seu *browser* no endereço `http://localhost:8000`.

A seguinte tela deve ser mostrada:



Se ela aparecer, nossa configuração está **correta** e o Django está pronto para começarmos a desenvolver!

Agora, vamos criar um **app** chamado **website** para separarmos os arquivos de configuração da nossa aplicação, que vão ficar na pasta **/helloworld**, dos arquivos relacionados ao **website**.

De acordo com a documentação, um **app** no Django é:

*Uma aplicação Web que faz alguma coisa, por exemplo - um blog, um banco de dados de registros públicos ou um aplicativo de pesquisa. Já um **projeto** é uma coleção de configurações e apps para um website em particular.*

Um projeto pode ter vários **apps** e um **app** pode estar presente em diversos projetos.

A fim de criar um novo **app**, o Django provê outro comando, chamado **django-admin.py startapp**.

Ele nos ajuda a criar os arquivos e diretórios necessários para tal objetivo.

Na raíz do projeto, execute:

```
1 django-admin.py startapp website
```

Agora, vamos criar algumas pastas para organizar a estrutura da nossa aplicação. Primeiro, crie a pasta **templates** dentro de **website**.

Dentro dela, crie uma pasta **website** e dentro dela, uma pasta chamada **_layouts**.

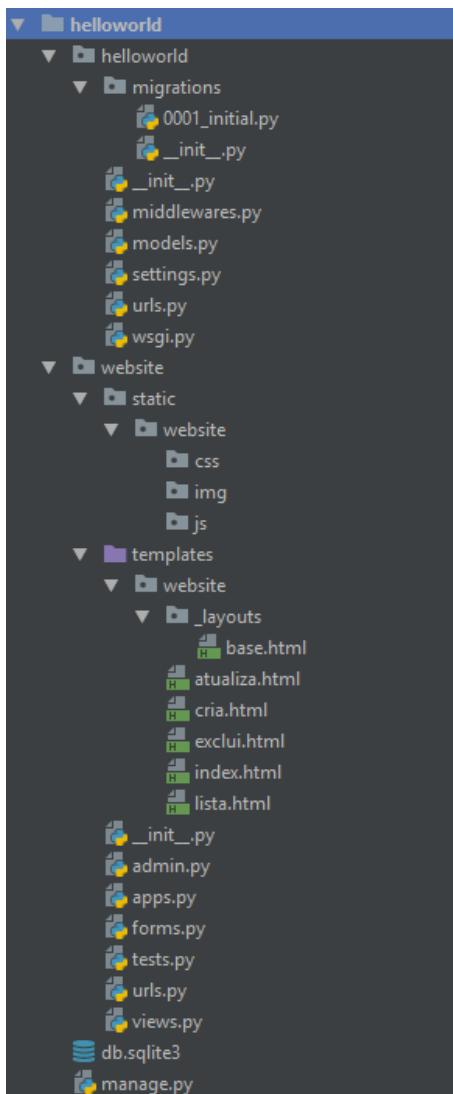
Crie também a pasta **static** dentro de **website**, para guardar os arquivos estáticos (arquivos CSS, Javascript, imagens, fontes, etc). Dentro dela crie uma pasta **website**, por questões de *namespace*. Dentro dela, crie: uma pasta **css**, uma pasta **img** e uma pasta **js**.

Assim, nossa estrutura de diretórios deve estar similar a:

```
1 /helloworld
2     - __init__.py
3     - settings.py
4     - urls.py
5     - wsgi.py
6 /website
7     - templates/
8         - website/
9             - _layouts/
10        - static/
11            - website/
12                - css/
13                - img/
14                - js/
15        - migrations/
16        - __init__.py
17        - admin.py
18        - apps.py
19        - migrations.py
20        - models.py
21        - tests.py
22        - views.py
23 - manage.py
```

Observação: Nós criamos uma pasta com o nome do app (**website**, no caso) dentro das pastas **static** e **templates** para que o Django crie o **namespace** do app. Dessa forma, o Django entende onde buscar os recursos quando você precisar!

Dessa forma, devemos estar com a estrutura da seguinte forma:



Para que o Django gerencie esse app, é necessário adicioná-lo a lista de **apps** instalados. Fazemos isso atualizando a configuração **INSTALLED_APPS** no arquivo de configuração **helloworld/settings.py**

Ela é uma lista e diz ao Django o conjunto de **apps** que devem ser gerenciados no nosso projeto.

É necessário adicionar os **apps** da nossa aplicação à essa lista para que o Django as enxergue. Para isso, procure por:

```
1 INSTALLED_APPS = [  
2     'django.contrib.admin',  
3     'django.contrib.auth',  
4     'django.contrib.contenttypes',  
5     'django.contrib.sessions',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8 ]
```

E adicione **website** e **helloworld**, ficando assim:

```
1 INSTALLED_APPS = [  
2     'django.contrib.admin',  
3     'django.contrib.auth',  
4     'django.contrib.contenttypes',  
5     'django.contrib.sessions',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'helloworld',  
9     'website'  
10 ]
```

Agora, vamos fazer algumas alterações na estrutura do projeto para organizar e centralizar algumas configurações.

Primeiro, vamos passar o arquivo de modelos `models.py` de `/website` para `/helloworld`, pois os arquivos comuns ao projeto vão ficar centralizados no app `helloworld` (geralmente temos apenas um arquivo `models.py` para o projeto todo).

Como não temos mais o arquivo de modelos na pasta `/website`, podemos, então, excluir a pasta `/migrations` e o `migrations.py`, pois estes serão gerados e gerenciados pelo app `helloworld`.

Por fim, devemos estar com a estrutura de diretórios da seguinte forma:

```
1 /helloworld
2   - __init__.py
3   - settings.py
4   - urls.py
5   - wsgi.py
6   - models.py
7 /website
8   - __init__.py
9   - admin.py
10  - apps.py
11  - tests.py
12  - views.py
13 - manage.py
```

Conclusão do Capítulo

Nesse capítulo, vimos um pouco sobre o Django, suas principais características, sua estrutura de diretórios e como começar a desenvolver utilizando-o!

Vimos as facilidades que o comando `django-admin` trazem e como utilizá-lo para criar nosso projeto.

Também o utilizamos para criar `apps`, que são estruturas modulares do nosso projeto, usados para organizar e separar funções específicas da nossa aplicação.

No **próximo capítulo**, vamos falar sobre a Camada *Model* do Django, que é onde residem as entidades do nosso sistema e toda a lógica de acesso a dados!

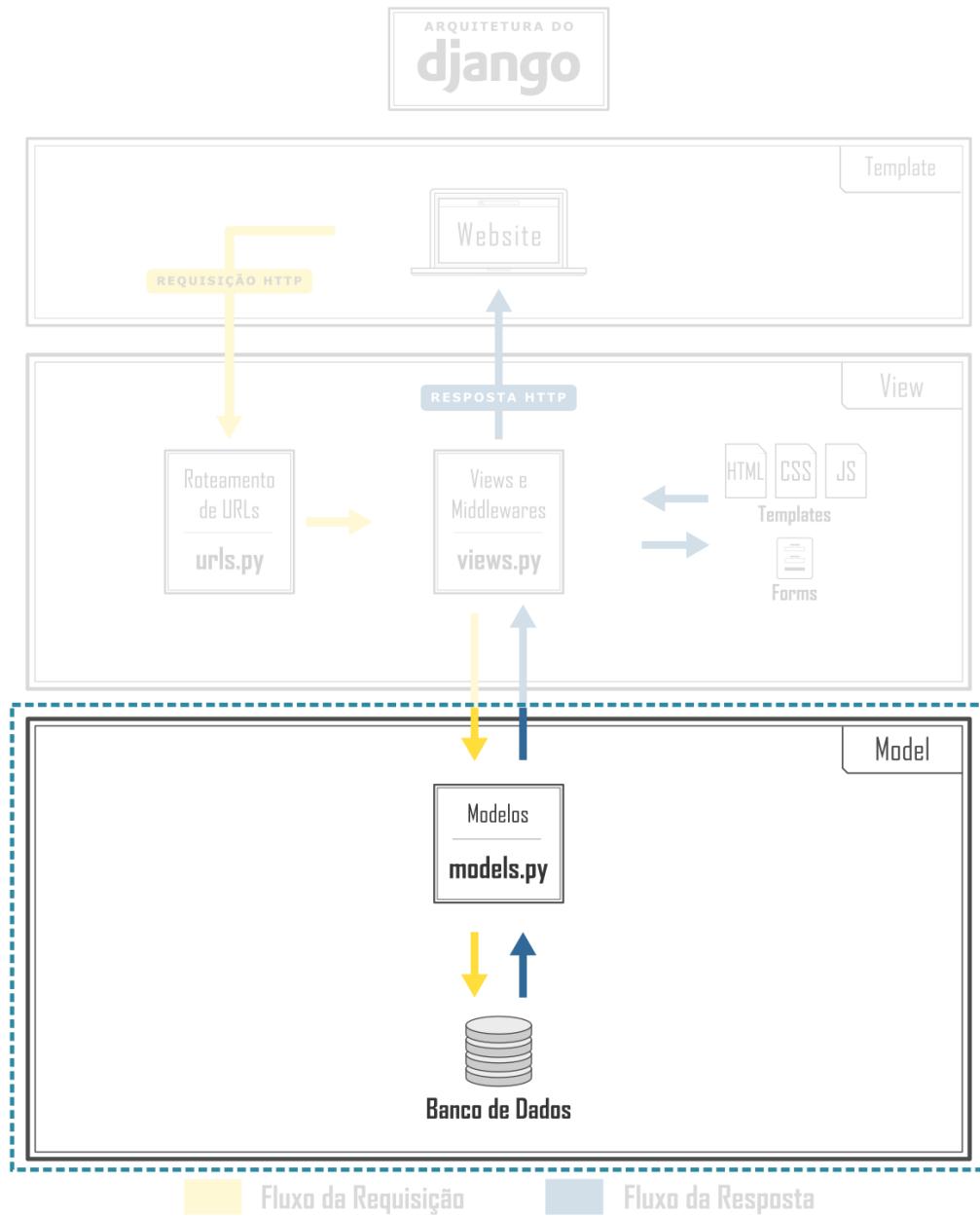
CAMADA *MODEL*

A Camada de Modelos tem uma função essencial na arquitetura das aplicações desenvolvidas com o Django. É nela que descrevemos os campos e comportamentos das entidades que irão compor nosso sistema. Também é nela que reside a lógica de acesso aos dados da nossa aplicação. Vamos ver como é simples manipular os dados do nosso sistema através da poderosa API de Acesso a Dados do Django.

Onde estamos...

No primeiro capítulo, tratamos de conceitos introdutórios do *framework*, uma visão geral da sua arquitetura, sua instalação e a criação do famoso ***Hello World Django-based***.

Agora, vamos tratar da primeira camada do Dango, conforme abaixo:



Vamos mergulhar um pouco mais e conhecer a camada *Model* da arquitetura **MTV** do Django (*Model Template View*).

Nela, vamos descrever, em forma de classes, as **entidades** do nosso sistema, para que o resto (*Template e View*) façam sentido.

Camada *Model*

Vamos começar pelo básico: pela definição de **modelo**!

Um **modelo** é a descrição do dado que será gerenciado pela sua aplicação.

Ele contém os campos e comportamentos desses dados. No fim, cada modelo vai equivaler à uma tabela no banco de dados.

No Django, um modelo tem basicamente duas características:

- É uma classe que herda de `django.db.models.Model`
- Cada atributo representa um campo da tabela

Com isso, Django gera **automaticamente** uma API de Acesso à Dados. Essa API facilita e muito nossa vida quando formos gerenciar (adicionar, excluir e atualizar) nossos dados.

Para entendermos melhor, vamos modelar nosso “*Hello World*”!

Vamos supor que sua empresa está desenvolvendo um sistema de gerenciamento dos funcionários e lhe foi dada a tarefa de modelar e desenvolver o acesso aos dados da entidade **Funcionário**.

Pensando calmamente em sua estação de trabalho enquanto seu chefe lhe cobra diversas metas e dizendo que o *deadline* do projeto foi adiantado em duas semanas você pensa nos seguintes atributos para tal classe:

- Nome
- Sobrenome
- CPF
- Tempo de serviço
- Remuneração

Ok!

Agora, é necessário passar isso para código Python para que o Django possa entender.

No Django, os modelos são descritos no arquivo `models.py`.

Ele já foi criado no Capítulo anterior e está presente na pasta `helloworld/models.py`.

Nele, nós iremos descrever cada atributo (nome, sobrenome, CPF e etc) como um campo (ou **Field**) da nossa classe de Modelo.

Vamos chamar essa classe de **Funcionário**.

Seguindo as duas características que apresentamos (herdar da classe **Model** e mapear os atributos da entidade com os campos), podemos descrever nosso modelo da seguinte forma:

```
1  from django.db import models
2
3  class Funcionario(models.Model):
4
5      nome = models.CharField(
6          max_length=255,
7          null=False,
8          blank=False
9      )
10
11     sobrenome = models.CharField(
12         max_length=255,
13         null=False,
14         blank=False
15     )
16
17     cpf = models.CharField(
18         max_length=14,
19         null=False,
20         blank=False
21     )
22
23     tempo_de_servico = models.IntegerField(
24         default=0,
25         null=False,
26         blank=False
27     )
28
29     remuneracao = models.DecimalField(
30         max_digits=8,
31         decimal_places=2,
32         null=False,
33         blank=False
34     )
35
36     objetos = models.Manager()
```

Explicando esse modelo:

- Cada campo tem um tipo.
- O tipo **CharField** representa uma string.

- O tipo `PositiveIntegerField` representa um número inteiro positivo.
- O tipo `DecimalField` representa um número decimal com precisão fixa (geralmente utilizamos para representar valores monetários).
- Cada tipo tem um conjunto de propriedades, como: `max_length` para delimitar o comprimento máximo da string; `decimal_places` para configurar o número de casas decimais; entre outras (a documentação de cada campo e propriedade pode ser [acessada aqui](#)).
- O campo `objetos = models.Manager()` é utilizado para fazer operações de busca e será explicado ali embaixo!
- **Observação:** não precisamos configurar o identificador `id` - ele é herdado do objeto `models.Model` (do qual nosso modelo herdou)!

Agora que criamos nosso modelo, é necessário executar a criação das tabelas no banco de dados.

Para isso, o Django possui dois comandos que ajudam **muito**: o `makemigrations` e o `migrate`.

O comando `makemigrations`

O comando `makemigrations` analisa se foram feitas mudanças nos modelos e, em caso positivo, cria novas migrações (**Migrations**) para alterar a estrutura do seu banco de dados, refletindo as alterações feitas.

Vamos entender o que eu acabei de dizer: toda vez que você faz uma **alteração** em seu modelo, é necessário que ela seja **aplicada** a estrutura presente no banco de dados.

A esse processo é dado o nome de **Migração!** De acordo com a documentação do Django:

Migração é a forma do Django de propagar as alterações feitas em seu modelo (adição de um novo campo, deleção de um modelo, etc...) ao seu esquema do banco de dados. Elas foram desenvolvidas para serem (a maioria das vezes) automáticas, mas cabe a você saber a hora de fazê-las, de executá-las e de resolver os problemas comuns que você possa vir a ser submetidos.

Portanto, toda vez que você alterar o seu modelo, não se esqueça de executar **python manage.py makemigrations!**

Ao executar esse comando no nosso projeto, devemos ter a seguinte saída:

```
1 $ python manage.py makemigrations
2
3 Migrations for 'helloworld':
4     helloworld\migrations\0001_initial.py
5         - Create model Funcionario
```

Observação: Ao executar pela primeira vez, talvez seja necessário executar o comando referenciando o app os modelos estão definidos, com: **python manage.py makemigrations helloworld**. Depois disso, apenas **python manage.py makemigrations** deve bastar!

Agora, podemos ver que foi criada um diretório chamado `migrations` dentro de `helloworld`.

Nele, você pode ver um arquivo chamado `0001_initial.py`

Ele contém a `Migration` que cria o `model Funcionario` no banco de dados (veja na saída do comando `makemigrations: Create model Funcionario`)

O comando `migrate`

Quando executamos o `makemigrations`, o Django cria o banco de dados e as `migrations`, mas não as executa, isto é: não aplica as alterações no banco de dados.

Para que o Django as aplique, são necessárias três coisas, basicamente:

- 1. Que a configuração da interface com o banco de dados esteja descrita no `settings.py`
- 2. Que os modelos e `migrations` estejam definidos para esse projeto.
- 3. Execução do comando `migrate`

Se você criou o projeto com `django-admin.py createproject helloworld`, a configuração padrão foi aplicada. Procure pela configuração `DATABASES` no `settings.py`.

Ela deve ser a seguinte:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
5     }  
6 }
```

Por padrão, o Django utiliza um banco de dados leve e completo chamado [SQLite](#). Já já vamos falar mais sobre ele.

Sobre os modelos e *migrations*, eles já foram feitos com a definição do [Funcionário](#) no arquivo `models.py` e com a execução do comando `makemigrations`.

Agora só falta **executar o comando `migrate`**, propriamente dito!

Para isso, vamos para a raíz do projeto e executamos: `python manage.py migrate`. A saída deve ser:

```
1 $ python manage.py migrate  
2  
3 Operations to perform:  
4     Apply all migrations: admin, auth, contenttypes, helloworld, sessions  
5  
6 Running migrations:  
7     Applying contenttypes.0001_initial... OK  
8     Applying auth.0001_initial... OK  
9     Applying admin.0001_initial... OK  
10    Applying admin.0002_logentry_remove_auto_add...  
11    Applying contenttypes.0002_remove_content_type_name...  
12    Applying auth.0002_alter_permission_name_max_length...  
13    Applying auth.0003_alter_user_email_max_length...  
14    Applying auth.0004_alter_user_username_opts...
```

```
14 Applying auth.0005_alter_user_last_login_nul...
15 Applying auth.0006_require_contenttypes_0002...
16 Applying auth.0007_alter_validators_add_error_
17 Applying auth.0008_alter_user_username_max_l...
18 Applying auth.0009_alter_user_last_name_max_...
19 Applying helloworld.0001_initial... OK
20 Applying sessions.0001_initial... OK
```

*Calma lá... Havíamos definido apenas **uma** Migration e foram aplicadas 15!!! Por quê???*

Se lembra que a configuração **INSTALLED_APPS** continha vários **apps** (e não apenas os nossos *helloworld e website*)?

Pois então, cada **app** desses contém seus próprios modelos e *migrations*. Sacou?!



Com a execução do comando `migrate`, o Django irá criar diversas tabelas no banco. Uma delas é a referente ao nosso modelo `Funcionário`, similar à:

```
1 CREATE TABLE helloworld_funcionario (
2     "id" serial NOT NULL PRIMARY KEY,
3     "nome" varchar(255) NOT NULL,
4     "sobrenome" varchar(255) NOT NULL,
5     ...
6 );
```

- *Isso está muito abstrato!*

- *Como eu posso ver o banco, as tabelas e os dados na prática?*

DB Browser for SQLite

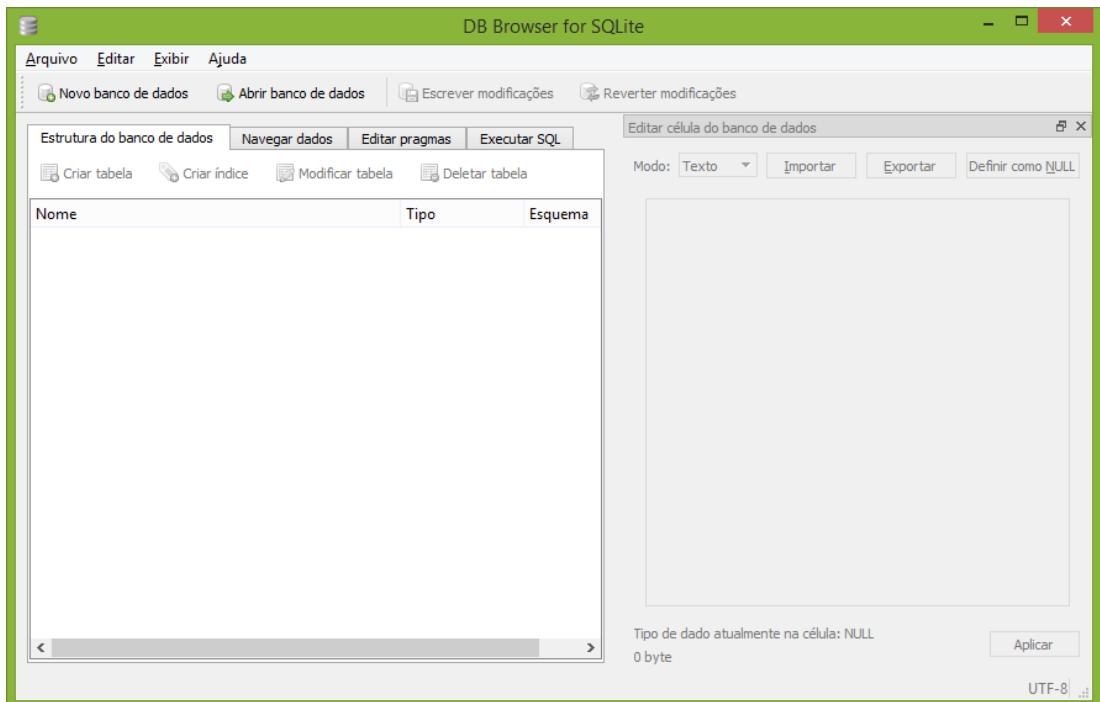
Apresento-lhes uma ferramenta **MUITO** prática que nos auxilia verificar se nosso código está fazendo aquilo que queríamos: o *DB Browser for SQLite*!

Com ele, podemos ver a estrutura do banco de dados, alterar dados em tempo real, fazer *queries*, verificar se os dados foram efetivados no banco e muito mais!

[Clique aqui](#) para fazer o *download* e instalação do software.

Ao terminar a instalação, abra o *DB Browser for SQLite*.

Temos a seguinte tela:



Aqui, podemos clicar em “Abrir banco de dados” e procurar pelo banco de dados do nosso projeto `db.sqlite3` (ele está na raíz do projeto).

Ao importá-lo, teremos uma visão geral, mostrando Tabelas, Índices, *Views* e *Triggers*.

Para ver os dados de cada tabela, vá para a aba “Navegar dados”, escolha nossa tabela `helloworld_funcionario` e...

Voilá! O que temos? NADA 😞

*Calma jovem... Ainda não adicionamos nada! Já já vamos criar as **Views** e **Templates** e popular esse BD!* 😊

API de Acesso a Dados

Com nossa classe **Funcionário** modelada, vamos agora ver a API de acesso à dados provida pelo Django para facilitar **muito** a nossa vida!

Vamos testar a adição de um novo funcionário utilizando o *shell* do Django. Para isso, digite o comando:

```
1 python manage.py shell
```

O shell do Django é muito útil para testar trechos de código sem ter que executar o servidor inteiro!

Para **adicionar** um novo funcionário, basta criar uma instância do seu modelo e chamar o método **save()** (*não desenvolvemos esse método, mas lembra que nosso modelo herdou de **Models**?* Pois é, é de lá que ele veio).

Podemos fazer isso com o código abaixo (no *shell* do Django):

```
1 from helloworld.models import Funcionario
2
3 funcionario = Funcionario(
4     nome='Marcos',
5     sobrenome='da Silva',
6     cpf='015.458.895-50',
7     tempo_de_servico=5,
8     remuneracao=10500.00
9 )
10
11 funcionario.save()
```

E.... Pronto!

O Funcionário **Marcos da Silva** será salvo no seu banco!

NADA de código SQL e queries enormes!!! Tudo simples! Tudo limpo! Tudo Python!

A API de **busca de dados** é ainda mais completa! Nela, você constrói sua *query* à nível de objeto!

Mas como assim?!

Por exemplo, para buscar todos os Funcionários, abra o *shell* do Django e digite:

```
1 funcionarios = Funcionario.objects.all()
```

Se lembra do tal **Manager** que falamos lá em cima? Então, um **Manager** é a interface na qual as operações de busca são definidas para o seu modelo.

Ou seja, através do campo **objetos** podemos fazer *queries* incríveis sem uma linha de SQL!

Exemplo de um *query* um pouco mais complexa:

*Busque todos os funcionários que tenham **mais de 3 anos de serviço**, que ganhem **menos de R\$ 5.000,00 de remuneração** e que **não tenham** Marcos no nome.*

Podemos atingir esse objetivo com:

```
1 funcionarios = Funcionario.objects
2     .exclude(name="Marcos")
3     .filter(tempo_de_servico__gt=3)
4     .filter(remuneracao__lt=5000.00)
5     .all()
```

O método `exclude()` retira linhas da pesquisa e `filter()` filtra a busca!

No exemplo, para filtrar por **maior que** concatenamos a string `__gt` (`gt` = *greater than* = maiores que) ao filtro e `__lt` (`lt` = *less than* = menores que) para resultados menores que o valor passado.

O método `.all()` ao final da *query* serve para retornar **todas** as linhas do banco que cumpram os filtros da nossa busca (também temos o `first()` que retorna apenas o primeiro registro, o `last()`, que retorna o último, entre outros).

Agora, vamos ver como é **simples** excluir um **Funcionário**:

```
1 # Primeiro, encontramos o Funcionário que desejamos deletar
2 funcionario = Funcionario
3     .objects
4     .filter(id=1)
5     .first()
6
7 # Agora, o deletamos!
8 funcionario.delete()
```

Legal, né?!

A atualização também é extremamente simples, bastando buscar a instância desejada, alterar o campo e salvá-lo novamente!

Por exemplo: o funcionário de **id = 13** se casou e alterou seu nome de Marcos da Silva para Marcos da Silva Albuquerque.

Podemos fazer essa alteração da seguinte forma:

```
1 # Primeiro, buscamos o funcionario desejado
2 funcionario = Funcionario
3     .objetos
4     .filter(id=13)
5     .first()
6
7 # Alteramos seu sobrenome
8 funcionario.sobrenome = funcionario.sobrenome + " Albuquerque"
9
10 # Salvamos as alterações
11 funcionario.save()
```

Conclusão do Capítulo

Com isso, concluímos a construção do modelo da nossa aplicação!

Criamos o banco de dados, vimos como visualizar os dados com o *DB Browser for SQLite* e como a API de acesso a dados do Django é simples e poderosa!

No **próximo capítulo**, vamos aprender sobre a Camada View e como podemos adicionar lógica de negócio à nossa aplicação!

CAMADA *VIEW*

Nesse capítulo, vamos abordar a Camada *View* do Django.

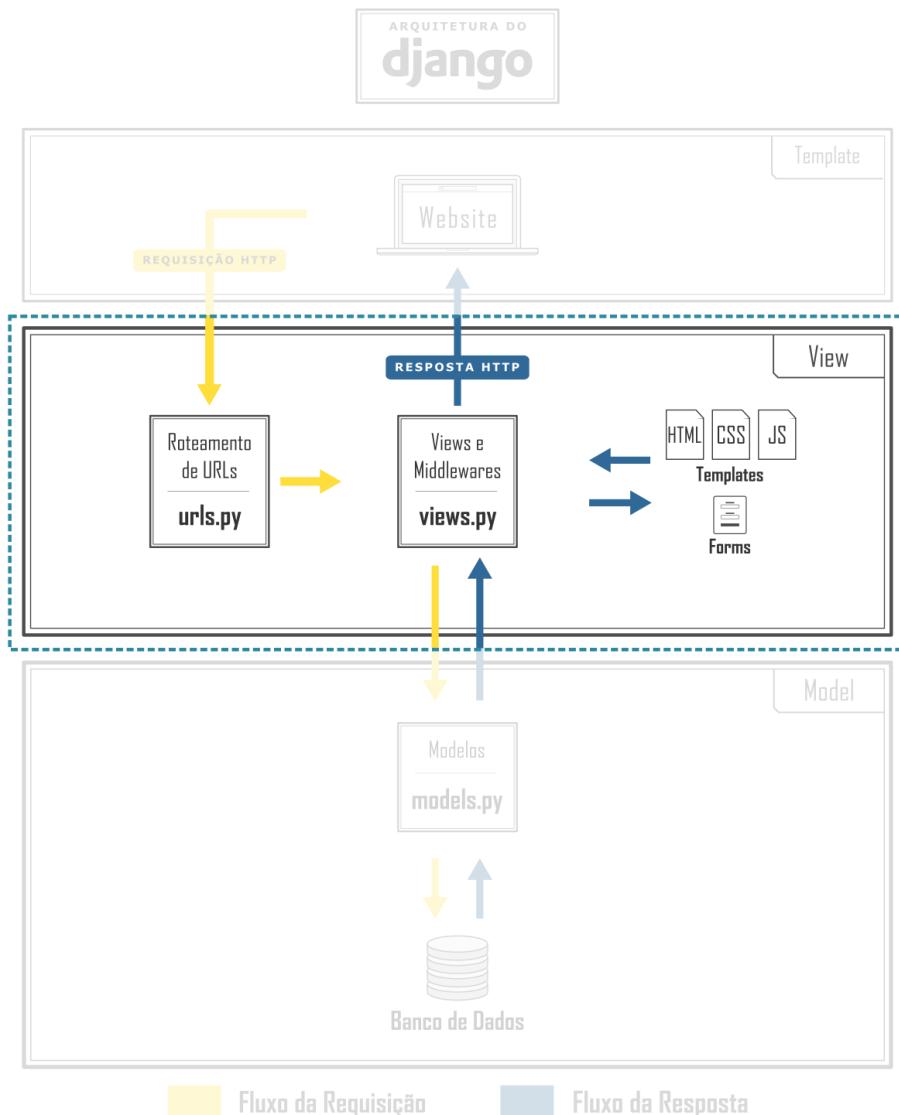
É nela que descreveremos a lógica de negócios da nossa aplicação, ou seja: é nela que vamos descrever os métodos que irão **processar as requisições, formular respostas e enviá-las** de volta ao usuário.

Vamos aprender o conceito das *Views* do Django, aprender a diferença entre *Function Based Views* e *Class Based Views*, como utilizar *Forms*, aprender o que é um *Middleware* e como desenvolver nossos próprios e muito mais.

Então vamos nessa, que esse capítulo está **completo!**

Onde estamos...

Primeiramente, vamos nos situar:



Camada View

Essa camada tem a responsabilidade de processar as **requisições** vindas dos usuários, formar uma **resposta** e enviá-la de volta ao usuário. É aqui que residem nossas **lógicas de negócio**!

Ou seja, essa camada deve: **recepçionar, processar e responder!**

Para isso, começamos pelo **roteamento de URLs**!

A partir da URL que o usuário quer acessar ([/Funcionarios](#), por exemplo), o Django irá rotear a requisição para quem irá tratá-la.

Mas primeiro, o Django precisa ser informado para **onde** mandar a requisição.

Fazemos isso no chamado **URLconf** e damos o nome a esse arquivo, por convenção, de [urls.py](#)!

Geralmente, temos um arquivo de rotas por *app* do Django. Portanto, crie um arquivo [urls.py](#) dentro da pasta [/helloworld](#) e outro na pasta [/website](#).

Como o *app helloworld* é o núcleo da nossa aplicação, ele faz o papel de centralizador de rotas, isto é:

- Primeiro, a requisição cai no arquivo [/helloworld/urls.py](#) e é roteada para o *app* correspondente.
- Em seguida, o URLConf do *app* ([/website/urls.py](#), no nosso caso) vai rotear a requisição para a *view* que irá processar a requisição.

Dessa forma, o arquivo `helloworld/urls.py` deve conter:

```
1 from django.urls.conf import include
2 from django.contrib import admin
3 from django.urls import path
4
5 urlpatterns = [
6     # Inclui as URLs do app 'website'
7     path('', include('website.urls', namespace='website')),
8
9     # Interface administrativa
10    path('admin/', admin.site.urls),
11 ]
```

Assim, o Django irá tentar fazer o *match* (casamento) de URLs primeiro no arquivo de URLs do app *Website* (`website/urls.py`) depois no URLConf da plataforma administrativa.

Pode parecer complicado, mas ali embaixo, quando tratarmos mais sobre Views, vai fazer mais sentido!

A configuração do URLConf é bem simples!

Basta definirmos qual função ou **View** irá processar requisições de **tal** URL. Por exemplo, queremos que:

Quando um usuário acesse a URL raíz /, o Django chame a função `index()` para processar tal requisição.

Vejamos como poderíamos configurar esse roteamento no nosso arquivo `urls.py`:

```
1 # Importamos a função index() definida no arquivo views.py
2 from . import views
3
4 app_name = 'website'
5
6 # urlpatterns contém a lista de roteamentos de URLs
7 urlpatterns = [
8     # GET /
9     path('', views.index, name='index'),
10 ]
```

O atributo `app_name = 'website'` define o namespace do app **website** (lembre-se do décimo nono Zen do Python: **namespaces** são uma boa ideia! - [clique aqui para saber mais sobre o Zen do Python](#)).

O método `path()` tem a seguinte assinatura:

`path(rota, view, kwargs=None, nome=Nome)`.

- **rota**: string contendo a rota (URL).
- **view**: a função (ou classe) que irá tratar essa rota.
- **kwargs**: utilizado para passar dados adicionais à função ou método que irá tratar a requisição.
- **nome**: nome da rota. O Django utiliza o `app_name` mais o nome da rota para nomear a URL. Por exemplo, no nosso caso, podemos chamar a rota raíz `'/'` com `'website:index'` (`app_site = website` e a rota raíz = `index`). Veja mais sobre [padrões de formato de URL](#).

Funções vs *Class Based Views*

Com as URLs corretamente configuradas, o Django irá rotear a sua requisição para onde você definiu. No caso acima, sua requisição irá cair na função `views.funcionarios_por_ano()`.

Podemos tratar as requisições de duas formas: através de funções (*Function Based Views*) ou através de *Class Based Views* (ou apenas **CBVs**).

Utilizando **funções**, você basicamente vai definir uma função que:

- **Recebe** como parâmetro uma requisição (`request`).
- **Realiza** algum processamento.
- **Retorna** alguma informação.

Já as *Class Based Views* são classes que herdam da classe do Django `django.view.generic.base.View` e que agrupam diversas funcionalidades e facilitam a vida do desenvolvedor.

Nós podemos herdar e estender as funcionalidades das *Class Based Views* para atender a lógica da nossa aplicação.

Por exemplo, suponha você quer criar uma página com a **listagem de todos os funcionários**.

Utilizando **funções**, você poderia chegar ao objetivo da seguinte forma:

```
1 def lista_funcionarios(request):
2     # Primeiro, buscamos os funcionários
3     funcionarios = Funcionario.objects.all()
4
5     # Incluímos no contexto
6     contexto = {
7         'funcionarios': funcionarios
8     }
9
10    # Retornamos o template para listar os funcionários
11    return render(
12        request,
13        "templates/funcionarios.html",
14        contexto
15    )
```

Aqui, algumas colocações:

- Toda função que vai processar requisições no Django recebe como parâmetro um objeto `request` contendo os dados da requisição.
- Contexto é o conjunto de dados que estarão disponíveis para construção da página - ou *template*.
- A função `django.shortcuts.render()` é um atalho (*shortcut*) do próprio Django que facilita a renderização de *templates*: ela recebe a própria requisição, o diretório do *template*, o contexto da requisição e retorna o *template* renderizado.

Já utilizando *Class Based Views*, podemos utilizar a `ListView` presente em `django.views.generic` para listar todos os funcionários, da seguinte forma:

```
1 from django.views.generic import ListView
2
3 class ListaFuncionarios(ListView):
4     template_name = "templates/funcionarios.html"
5     model = Funcionario
6     context_object_name = "funcionarios"
```

Perceba que você não precisou descrever a lógica para buscar a lista de funcionários?

É **exatamente isso** que as **Views** do Django proporcionam: elas descrevem o comportamento padrão para as funcionalidades mais simples (listagem, exclusão, busca simples, atualização).

O caso comum para uma listagem de objetos é buscar todo o conjunto de dados daquela entidade e mostrar no template, certo?! É exatamente **isso** que a **ListView** faz!

Com isso, um objeto **funcionarios** estará disponível no seu *template* para iteração.

Dessa forma, podemos então criar uma tabela no nosso *template* com os dados de todos os funcionários:

```
1 <table>
2   <tbody>
3     {% for funcionario in funcionarios %}
4       <tr>
5         <td>{{ funcionario.nome }}</td>
6         <td>{{ funcionario.sobrenome }}</td>
7         <td>{{ funcionario.remuneracao }}</td>
8         <td>{{ funcionario.tempo_de_servico }}</td>
```

```
9   </tr>
10  {% endfor %}
11  </tbody>
12  </table>
```

Vamos falar mais sobre **templates** no próximo artigo!

O Django tem uma diversidade enorme de *Views*, uma para cada finalidade, por exemplo:

- **CreateView**: Para criar de objetos (*É o Create do CRUD*)
- **DetailView**: Traz os detalhes de um objeto (*É o Retrieve do CRUD*)
- **UpdateView**: Para atualização de um objeto (*É o Update do CRUD*)
- **DeleteView**: Para deletar objetos (*É o Delete do CRUD*)

E várias outras muito úteis!

Agora vamos tratar detalhes do tratamento de requisições através de Funções. Em seguida, trataremos mais sobre as *Class Based Views*.

Funções (*Function Based Views*)

Utilizar funções é a maneira mais explícita para tratar requisições no Django (veremos que as *Class Based Views* podem ser um pouco mais complexas pois muita coisa acontece implicitamente).

Utilizando funções, geralmente tratamos primeiro o método HTTP da requisição: foi um **GET**? Foi um **POST**? Um **OPTION**?

A partir dessa informação, processamos a requisição da maneira desejada.

Vamos seguir o exemplo abaixo:

```
1 def cria_funcionario(request, pk):
2     # Verificamos se o método POST
3     if request.method == 'POST':
4         form = FormularioDeCriacao(request.POST)
5
6     if form.is_valid():
7         form.save()
8         return HttpResponseRedirect(reverse('lista_funcionarios'))
9
10    # Qualquer outro método: GET, OPTION, DELETE, etc...
11    else:
12        return render(request, "templates/form.html", {'form': form})
```

O fluxo é o seguinte:

- Primeiro, conforme mencionei, verificamos o método HTTP da requisição no campo `method` do objeto `request` na **linha 3**.
- Depois instanciamos um `form` com os dados da requisição (no caso `POST`) com `FormularioDeCriacao(request.POST)` na **linha 4** (vamos falar mais sobre `Form` já).
- Verificamos os campos do formulário com `form.is_valid()` na **linha 6**.
- Se tudo estiver **OK**, utilizamos o helper `reverse()` para traduzir a rota '`lista_funcionarios`' para `funcionarios/`. Utilizamos isso para retornar um *redirect* para a *view* de listagem na **linha 8**.
- Se for qualquer outro método, apenas renderizamos a página novamente com o método `render()` na **linha 12**.

Deu para perceber que o objeto `request` é essencial nas nossas Views, né?

Separei aqui alguns atributos desse objeto que provavelmente serão os mais utilizados por você:

- `request.scheme`: String representando o esquema (se veio por uma conexão *HTTP* ou *HTTPS*).
- `request.path`: String com o caminho da página requisitada - exemplo: `/cursos/curso-de-python/detalhes`.
- `request.method`: Conforme citamos, contém o método *HTTP* da requisição (***GET***, ***POST***, ***UPDATE***, ***OPTION***, etc).
- `request.content_type`: Representa o tipo MIME da requisição - `text/plain` para texto plano, `image/png` para arquivos .PNG, por exemplo - saiba mais [clicando aqui](#).
- `request.GET`: Um *dict* contendo os parâmetros GET da requisição.
- `request.POST`: Um *dict* contendo os parâmetros do corpo de uma requisição POST.
- `request.FILES`: Caso seja uma página de *upload*, contém os arquivos que foram enviados. Só contém dados se for uma requisição do tipo *POST* e o `<form>` da página *HTML* tenha o parâmetro `enctype="multipart/form-data"`.
- `request.COOKIES`: *Dict* contendo todos os *COOKIES* no formato de string.

Observação: Para saber mais sobre os campos do objeto `request`, dê uma olhada na classe `django.http.request.HttpRequest`!

Debugando uma requisição no PyCharm

Algumas vezes, é interessante você ver o conjunto de dados que estão vindo do browser do usuário para o servidor onde o Django está sendo executado.

Outras vezes, precisamos verificar se está tudo correto, se tudo está vindo como esperado ou se existem erros na requisição.

Uma forma de vermos isso é **debugando** o código, isto é: pausando a execução do código no momento que em que a requisição chega no servidor e vendo os atributos da requisição, verificando se está tudo OK (*ou não*).

Se você utiliza o **PyCharm**, ou alguma outra IDE com debugger, pode fazer os passos que eu vou descrever aqui (*creio que em outra IDE, o processo seja similar*).

Por exemplo, vamos adicionar um *breakpoint* no método de uma *View*. Para isso, clique duas vezes ao lado esquerdo da linha onde quer adicionar o *breakpoint*. O resultado deve ser esse (*linha 75, veja o círculo vermelho na barra à esquerda, próximo ao contador das linhas*):

```
61     )
62
63     return HttpResponseRedirect(self.get_success_url())
64
65
66     # COURSE DETAIL VIEW
67     #
68
69     class CourseDetailView(DashboardMixin, DetailView):
70         template_name = 'dashboard/course/detail.html'
71         model = Course
72
73         def get_context_data(self, **kwargs):
74             # Get super() content
75             context = super().get_context_data()
76
77             # Get course questions
78             context['questions'] = CourseDiscussionQuestion.objects.filter(
79                 course_id=context['course'].id
80             ).all()
81
82             # Put comment form on context
83             context['form'] = CommentForm()
84
85             # All lessons' progress
86             context['user_progress'] = UserCourseProgress.objects.filter(
87                 user_id=self.request.user.id,
88                 course=context['course'],
89             ).all()
90
91             # Last watched lesson
92             last_watched_lesson_id = UserCourseProgress.objects.filter(
93                 user_id=self.request.user.id,
94                 course=context['course'],
95                 accomplished=True
96             ).aggregate(Max('lesson'))
97
98             if last_watched_lesson_id is None:
99                 context['last_watched_lesson'] = None
100            else:
101                context['last_watched_lesson'] = Lesson.objects.filter(
102                    id=last_watched_lesson_id.get('lesson__max')
103                ).first()
104
105            return context
106
107
108     # LESSON DETAIL VIEW
109     #
110
111     class LessonDetailView(DashboardMixin, DetailView):
112         template_name = 'dashboard/lesson/player.html'
113         model = Lesson
114         context_object_name = 'lesson'
115
116         def get_context_data(self, **kwargs):
117             CourseDetailView.get_context_data(self)
```

Com isso, quando você disparar uma requisição no seu browser que venha a cair nessa linha de código, o *debugger* entrará em ação, mostrando as variáveis naquela linha de código.

Nesse exemplo, quando o *debugger* chegou nessa linha, obtive a seguinte saída:

The screenshot shows the PyCharm debugger's variables view. The current frame is labeled 'dj pythonacademy'. The 'request' variable is selected and expanded, showing its attributes:

- login_url = {NoneType} None
- model = {ModelBase} <class 'core.models.Course'>
- object = {Course} Course object (1)
- permission_denied_message = {str} ''
- pk_url_kwarg = {str} 'pk'
- query_pk_and_slug = {bool} False
- queryset = {NoneType} None
- raise_exception = {bool} False
- redirect_field_name = {str} 'next'
- redirect_to_login = {NoneType} None
- request = {WSGIRequest}

The 'request' attribute is expanded, showing its detailed structure:

- COOKIES = {dict}
- FILES = {MultiValueDict}
- GET = {QueryDict}
- META = {dict}
- POST = {QueryDict}
- _cached_user = {User}
- _encoding = {NoneType}
- _messages = {FallbackStorage}
- _post_parse_error = {bool}
- _read_started = {bool}
- _stream = {LimitedStream}
- _upload_handlers = {list}
- body = {bytes}
- content_params = {dict}
 - content_type = {str}
 - csrf_processing_done = {bool}
 - encoding = {NoneType}
- environ = {dict}
 - method = {str}
 - path = {str}
 - path_info = {str}
- resolver_match = {ResolverMatch}
 - scheme = {str}
- session = {SessionStore}
- upload_handlers = {list}
- user = {User}

A partir dessa visão, podemos verificar **todos** os atributos do objeto **request** que chegou no servidor!

*Vão por mim, isso ajuda **MUITO** a detectar erros!*

Dito isso, agora vamos tratar detalhes do tratamento de requisições através de *Class Based Views*.

Classes (CBV - *Class Based Views*)

Conforme expliquei anteriormente, as *Class Based Views* servem para automatizar e facilitar nossa vida, encapsulando funcionalidades comuns que todo desenvolvedor sempre acaba implementando. Por exemplo, geralmente:

- Queremos que quando um usuário vá para página inicial, seja mostrado **apenas uma página** simples, com as opções possíveis.
- Queremos que nossa **página de listagem** contenha a **lista** de todos os funcionários cadastrados no banco de dados.
- Queremos uma **página com um formulário** contendo todos os campos pré-preenchidos para **atualização** de dado funcionário.
- Queremos uma **página de exclusão** de funcionários.
- Queremos um formulário em branco para **inclusão de um novo funcionário**.

Certo?!

Pois é, as **CBVs - Class Based Views** - facilitam isso para nós!

Temos basicamente **duas formas** para utilizar uma **CBV**.

Primeiro, podemos utilizá-las diretamente no nosso `URLConf` (`urls.py`), assim:

```
1 from django.urls import path
2 from django.views.generic import TemplateView
3
4 urlpatterns = [
5     path('', TemplateView.as_view(template_name="index.html")),
6 ]
```

E a segunda maneira, a mais utilizada e mais poderosa, é herdar da *View* desejada e sobrescrever os atributos e métodos na subclasse.

Abaixo, veremos as Views mais utilizadas, e como podemos utilizá-las em nosso projeto.

TemplateView

Por exemplo, para o **primeiro caso**, podemos utilizar a [TemplateView](#) ([acesse a documentação](#)) para renderizar uma página, da seguinte forma:

```
1 class IndexTemplateView(TemplateView):
2     template_name = "index.html"
```

E a configuração de rotas fica assim:

```
1 from django.urls import path
2 from helloworld.views import IndexTemplateView
3
4 urlpatterns = [
5     path('', IndexTemplateView.as_view(), name='index'),
6 ]
```

ListView

Já para o segundo caso, de **listagem de funcionários**, podemos utilizar a [ListView](#) (acesse a documentação).

Nela, nós configuramos o *Model* que deve ser buscado (**Funcionario** no nosso caso), e ela automaticamente faz a busca por todos os registros presentes no banco de dados da entidade informada.

Por exemplo, podemos descrever a *View* da seguinte forma:

```
1 from django.views.generic.list import ListView
2 from helloworld.models import Funcionario
3
4 class FuncionarioListView(ListView):
5     template_name = "website/lista.html"
6     model = Funcionario
7     context_object_name = "funcionarios"
```

Utilizamos o atributo `contexto_object_name` para nomear a variável que estará disponível no contexto do `template` (se não, o nome padrão dado pelo Django será `object`).

E configurá-la assim:

```
1 from django.urls import path
2 from helloworld.views import FuncionarioListView
3
4 urlpatterns = [
5     path(
6         'funcionarios/',
```

```
7     FuncionarioListView.as_view(),
8     name='lista_funcionarios'),
9 ]
```

Isso resultará em uma página **lista.html** contendo um objeto chamado **funcionarios** contendo todos os Funcionários disponível para iteração.

*Dica: É uma boa prática colocar o nome da View como o Model + CBV base. Por exemplo: uma view que lista todos os Cursos, receberia o nome de **CursoListView** (Model = *Curso* e CBV = *ListView*).*

UpdateView

Para a **atualização de usuários** podemos utilizar a **UpdateView** (veja a [documentação](#)). Com ela, configuramos qual o *Model* (atributo **model**), quais campos (atributo **field**) e qual o nome do template (atributo **template_name**), e com isso temos um formulário para atualização do modelo definido.

No nosso caso:

```
1 from django.views.generic.edit import UpdateView
2 from helloworld.models import Funcionario
3
4 class FuncionarioUpdateView(UpdateView):
5     template_name = 'atualiza.html'
6     model = Funcionario
7     fields = [
8         'nome',
9         'sobrenome',
```

```
10     'cpf',
11     'tempo_de_servico',
12     'remuneracao'
13 ]
```

Dica: Ao invés de listar todos os campos em `fields` em formato de lista de strings, podemos utilizar `fields = '__all__'`. Dessa forma, o Django irá buscar todos os campos para você!

Mas de onde o Django vai pegar o `id` do objeto a ser buscado?

O Django precisa ser informado do `id` ou `slug` para poder buscar o objeto correto a ser atualizado. Podemos fazer isso de **duas formas**.

Primeiro, na configuração de rotas (`urls.py`):

```
1 from django.urls import path
2 from helloworld.views import FuncionarioUpdateView
3
4 urlpatterns = [
5     # Utilizando o {id} para buscar o objeto
6     path(
7         'funcionario/<id>',
8         FuncionarioUpdateView.as_view(),
9         name='atualiza_funcionario'),
10
11    # Utilizando o {slug} para buscar o objeto
12    path(
13        'funcionario/<id>',
14        FuncionarioUpdateView.as_view(),
15        name='atualiza_funcionario'),
16 ]
```

Mas o que é slug?

Slug é uma forma de gerar URLs mais **legíveis** a partir de dados já existentes.

Exemplo: podemos criar um campo *slug* utilizando o campo **nome** do funcionário. Dessa forma, as URLs ficariam assim:

- `/funcionario/vinicius-ramos`

E não assim (utilizando o **id** na URL):

- `/funcionario/175`

No campo *slug*, **todos os caracteres** são transformados em minúsculos e os espaços são transformados em hífens, o que dá mais sentido à URL.

A **segunda forma** de buscar o objeto é utilizando (ou **sobrescrevendo**) o método `get_object()` da classe pai `UpdateView`.

A documentação desse método traz (traduzido):

Retorna o objeto que a View irá mostrar. Requer `self.queryset` e um argumento `pk` ou `slug` no URLConf. Subclasses podem sobrescrever esse método e retornar qualquer objeto.

Ou seja, o Django nos dá total liberdade de utilizarmos a **convenção** (quando passamos os parâmetros na configuração da rota

- *URLConf* ou a **configuração** (quando sobrescrevemos o método `get_object()`).

Basicamente, o método `get_object()` deve pegar o `id` ou `slug` da URL e realizar a busca no banco de dados até encontrar o objeto com aquele `id`.

Uma forma de sobrescrevermos esse método na *View* de listagem de funcionários (`FuncionarioListView`) pode ser implementada da seguinte maneira:

```
1 from django.views.generic.edit import UpdateView
2 from helloworld.models import Funcionario
3
4 class FuncionarioUpdateView(UpdateView):
5     template_name = "atualiza.html"
6     model = Funcionario
7     fields = '__all__'
8     context_object_name = 'funcionario'
9
10    def get_object(self, queryset=None):
11        funcionario = None
12
13        # Os campos {pk} e {slug} estão presentes em self.kwargs
14        id = self.kwargs.get(self.pk_url_kwarg)
15        slug = self.kwargs.get(self.slug_url_kwarg)
16
17        if id is not None:
18            # Busca o funcionario apartir do id
19            funcionario = Funcionario
20                .objects
21                .filter(id=id)
22                .first()
23
24        elif slug is not None:
25            # Pega o campo slug do Model
26            campo_slug = self.get_slug_field()
```

```
27      # Busca o funcionario apartir do slug
28      funcionario = Funcionario
29          .objects
30          .filter(**{campo_slug: slug})
31          .first()
32
33
34      # Retorna o objeto encontrado
35      return funcionario
```

Dessa forma, os dados do funcionário estarão disponíveis na variável `funcionario` no *template* `atualiza.html`!

DeleteView

Para deletar funcionários, utilizamos a `DeleteView` ([documentação](#)).

Sua configuração é similar à `UpdateView`: nós devemos informar ao Django qual o objeto queremos excluir via URLConf ou através do método `get_object()`.

Precisamos configurar:

- O *template* que será renderizado.
- O *model* associado à essa *view*.
- O nome do objeto que estará disponível no *template*.
- A URL de retorno, caso haja sucesso na deleção do Funcionário.

Com isso, a *view* pode ser codificada da seguinte forma:

```
1 class FuncionarioDeleteView(DeleteView):
2     template_name = "website/exclui.html"
```

```
3 model = Funcionario  
4 context_object_name = 'funcionario'  
5 success_url = reverse_lazy(  
6     "website:lista_funcionarios"  
7 )
```

O método `reverse_lazy()` serve para fazer a conversão de rotas (similar ao `reverse()`) mas em um momento em que o URLConf ainda não foi carregado (que é o caso aqui)

Assim como na `UpdateView`, fazemos a configuração do `id` a ser buscado no `URLConf`, da seguinte forma:

```
1 urlpatterns = [  
2     path(  
3         'funcionario/excluir/<pk>',  
4         FuncionarioDeleteView.as_view(),  
5         name='deleta_funcionario'),  
6 ]
```

Assim, precisamos apenas fazer um *template* de confirmação da exclusão do funcionário (o link será feito através de um botão “Excluir” que vamos adicionar à página `lista.html` no **próximo capítulo**).

Podemos fazer o *template* da seguinte forma:

```
1 <form method="post">  
2     {% csrf_token %}  
3  
4     Você tem certeza que quer excluir  
5     o funcionário <b>{{ funcionario.nome }}</b>?  
6     <br><br>
```

```
7 <button type="button">
8   <a href="{% url 'lista_funcionarios' %}">
9     Cancelar
10    </a>
11  </button>
12  <button>Excluir</button>
13 </form>
```

Algumas colocações:

- A *tag* do Django `{% csrf_token %}` é obrigatório em todos os *forms* pois está relacionado à proteção que o Django provê ao **CSRF** - *Cross Site Request Forgery* (tipo de ataque malicioso - [saiba mais aqui](#)).
- Não se preocupe com a sintaxe do *template* veremos mais sobre ele no **próximo post!**

CreateView

Nessa *View*, precisamos apenas dizer para o Django o *model*, o nome do *template*, a classe do formulário (vamos tratar mais sobre *Forms* ali embaixo) e a URL de retorno, caso haja sucesso na inclusão do Funcionário.

Podemos fazer isso assim:

```
1 from django.views.generic import CreateView
2
3 class FuncionarioCreateView(CreateView):
4   template_name = "website/cria.html"
```

```
5 model = Funcionario
6 form_class = InsereFuncionarioForm
7 success_url = reverse_lazy(
8     "website:lista_funcionarios"
9 )
```

O método `reverse_lazy()` traduz a View em URL. No nosso caso, queremos que quando haja a inclusão do Funcionário, sejamos redirecionados para a página de listagem, para podermos conferir que o Funcionário foi realmente adicionado.

E a configuração da rota no arquivo `urls.py`:

```
1 from django.urls import path
2 from helloworld.views import FuncionarioCreateView
3
4 urlpatterns = [
5     path(
6         'funcionario/cadastrar/',
7         FuncionarioCreateView.as_view(),
8         name='cadastra_funcionario'),
9 ]
```

Com isso, estará disponível no template configurado (`website/cria.html`, no nosso caso), um objeto `form` contendo os campos do formulário para criação do novo funcionário.

Podemos mostrar o formulário de duas formas.

A **primeira**, mostra o formulário inteiro **cru**, isto é, sem formatação e estilo, conforme o Django nos entrega.

Podemos mostrá-lo no nosso template da seguinte forma:

```
1 <form method="post">
2   {% csrf_token %}
3
4   {{ form }}
5
6   <button type="submit">Cadastrar</button>
7 </form>
```

*Observação: apesar de ser um **Form**, sua renderização não contém as tags **<form></form>** - cabendo a nós incluí-los no template.*

Já a **segunda**, é mais trabalhosa, pois temos que renderizar campo a campo no *template*. Porém, nos dá um nível maior de customização.

Podemos renderizar cada campo do *form* dessa forma:

```
1 <form method="post">
2   {% csrf_token %}
3
4   <label for="{{ form.nome.id_for_label }}>
5     Nome
6   </label>
7   {{ form.nome }}
8
9   <label for="{{ form.sobrenome.id_for_label }}>
10    Sobrenome
11   </label>
12   {{ form.sobrenome }}
13
14  <label for="{{ form.cpf.id_for_label }}>
15    CPF
16  </label>
```

```
17  {{ form.cpf }}  
18  
19  <label for="{{ form.tempo_de_servico.id_for_label }}>  
20    Tempo de Serviço  
21  </label>  
22  {{ form.tempo_de_servico }}  
23  
24  <label for="{{ form.remuneracao.id_for_label }}>  
25    Remuneração  
26  </label>  
27  {{ form.remuneracao }}  
28  
29  <button type="submit">Cadastrar</button>  
30 </form>
```

Nesse *template*:

- `{{ form.campo.id_for_label }}` traz o `id` da tag `<input>` para adicionar à tag `<label></label>`.
- Utilizamos o `{{ form.campo }}` para renderizar apenas um campo do formulário, e não ele inteiro.

Agora vamos aprender mais sobre a utilização do **Form** do Django!

Forms

O tratamento de formulários é uma tarefa que pode ser **bem complexa**.

Considere um formulário com diversos campos e diversas regras de validação: seu tratamento não é mais um processo simples.

Os *Forms* do Django são formas de descrever, em **código Python**, os formulários das páginas HTML, **simplificando e automatizando** seu processo de criação e validação.

O Django trata **três** partes distintas dos formulários:

- **Preparação** dos dados tornando-os prontos para renderização
- **Criação** de formulários HTML para os dados
- **Recepção** e processamento dos formulários enviados ao servidor

Basicamente, queremos uma forma de renderizar em nosso *template* o seguinte código HTML:

```
1 <form action="/insere-funcionario/" method="post">
2   <label for="nome">Your name: </label>
3   <input id="nome" type="text" name="nome" value="">
4   <input type="submit" value="Enviar">
5 </form>
```

E que, ao ser submetido ao servidor, tenha seus campos de entrada validados e, em caso de validação positiva – sem erros, seja inserido no banco de dados.

No centro desse sistema de formulários do Django está a classe **Form**.

Nela, nós descrevemos os campos que estarão disponíveis no formulário HTML.

Para o formulário acima, podemos descrevê-lo da seguinte forma.

```
1 from django import forms
2
3 class InsereFuncionarioForm(forms.Form):
4     nome = forms.CharField(
5         label='Nome do Funcionário',
6         max_length=100
7     )
```

Nesse formulário:

- Utilizamos a classe `forms.CharField` para descrever um campo de texto.
- O parâmetro `label` descreve um rótulo para esse campo.
- `max_length` decreve o tamanho máximo que esse `input` pode receber (100 caracteres, no caso).

Veja os diversos tipos de campos disponíveis [acessando aqui](#).

A classe `Forms.Form` possui um método muito importante, chamado `is_valid()`.

Quando um formulário é submetido ao servidor, esse é um dos métodos que irá realizar a validação dos campos do formulário.

Se tudo estiver **OK**, ele colocará os dados do formulário no atributo `cleaned_data` (que pode ser acessado por você posteriormente para pegar alguma informação - como o nome que foi inserido pelo usuário no campo `<input name='nome'>`).

*Como o processo de validação do Django é **bem complexo**, optei por descrever aqui o essencial para começarmos a utilizá-lo. Para saber mais sobre o funcionamento dos Forms, [acesse a documentação aqui](#).*

Vamos ver agora um exemplo mais complexo com um formulário de inserção de um Funcionário com todos os campos. Para isso, crie o arquivo **forms.py** no *app website*.

Em seguida, e consultando a [documentação](#) dos possíveis campos do formulário, podemos descrever um Form de inserção assim:

```
1 from django import forms
2
3 class InsereFuncionarioForm(forms.Form):
4
5     nome = forms.CharField(
6         required=True,
7         max_length=255
8     )
9
10    sobrenome = forms.CharField(
11        required=True,
12        max_length=255
13    )
14
15    cpf = forms.CharField(
16        required=True,
17        max_length=14
18    )
19
20    tempo_de_servico = forms.IntegerField(
21        required=True
22    )
23
24    remuneracao = forms.DecimalField()
```

Affff, mas o Model e o Form são quase iguais... Terei que reescrever os campos toda vez?

Claro que não, jovem! Por isso o Django nos presenteou com o incrível **ModelForm** 😊

Com o **ModelForm** nós configuramos de qual *Model* o Django deve pegar os campos. A partir do atributo **fields**, nós dizemos quais campos nós queremos e, através do campo `exclude`, os campos que não queremos.

Para fazer essa configuração, utilizamos os metadados da classe interna **Meta**. Metadado (no caso do *Model* e do *Form*) é tudo aquilo que não será transformado em campo, como `model`, **fields**, **ordering** etc (veja mais sobre *Meta options*).

Assim, nosso **ModelForm**, pode ser descrito da seguinte forma:

```
1 from django import forms
2
3 class InsereFuncionarioForm(forms.ModelForm):
4     class Meta:
5         # Modelo base
6         model = Funcionario
7
8         # Campos que estarão no form
9         fields = [
10             'nome',
11             'sobrenome',
12             'cpf',
13             'remuneracao'
14         ]
```

```
15  
16     # Campos que não estarão no form  
17     exclude = [  
18         'tempo_de_servico'  
19     ]
```

Podemos utilizar apenas o campo **fields**, apenas o **exclude** ou os dois juntos e mesmo ao utilizá-los, ainda podemos adicionar outros campos, **independente** dos campos do *Model*.

O resultado será um formulário com todos os campos presentes no **fields**, menos os campos do **exclude** mais os outros campos que adicionarmos.

Ficou confuso? Então vamos ver um exemplo que utiliza todos os atributos e ainda adiciona novos campos ao formulário:

```
1 from django import forms  
2  
3 class InsereFuncionarioForm(forms.ModelForm)  
4  
5     chefe = forms.BooleanField(  
6         label='Chefe?',  
7         required=True,  
8     )  
9  
10    biografia = forms.CharField(  
11        label='Biografia',  
12        required=False,  
13        widget=forms.TextArea  
14    )  
15  
16    class Meta:  
17        # Modelo base
```

```
18     model = Funcionario
19
20     # Campos que estarão no form
21     fields = [
22         'nome',
23         'sobrenome',
24         'cpf',
25         'remuneracao'
26     ]
27
28     # Campos que não estarão no form
29     exclude = [
30         'tempo_de_servico'
31     ]
```

Isso vai gerar um formulário com:

- Todos os campos contidos em `fields`
- Serão retirados os campos contidos em `exclude`
- O campo `forms.BooleanField`, como um `checkbox` (`<input type='checkbox' name='chefe' ...>`)
- Biografia como uma área de texto (`<textarea name='biografia' ...></textarea>`)

Assim como é possível definir **atributos** nos modelos, os campos do formulário **também são customizáveis**.

Veja que o campo `biografia` é do tipo `CharField`, portanto deveria ser renderizado como um campo `<input type='text' ...>`.

Contudo, eu modifiquei o campo configurando o atributo `widget` com `forms.TextArea`.

Assim, ele não mais será um simples *input*, mas será renderizado como um `<textarea></textarea>` no nosso *template*!

Nós veremos mais sobre formulários no **próximo capítulo**, quando formos renderizá-los nos nossos *templates*.

Agora vamos tratar de um componente muito importante no processamento de requisições e formulação das respostas da nossa aplicação: os *Middlewares*.

Middlewares

Middlewares são trechos de códigos que podem ser executados antes ou depois do processamento de requisições/respostas pelo Django.

É uma forma que os desenvolvedores, **nós**, temos para alterar como o Django processa algum dado de entrada ou de saída.

Se você olhar no arquivo `settings.py`, nós temos a lista `MIDDLEWARE` com diversos *middlewares* pré-configurados:

```
1 MIDDLEWARE = [
2     'django.middleware.security.SecurityMiddleware',
3     'django.contrib.sessions.middleware.SessionMiddleware',
4     'django.middleware.common.CommonMiddleware',
5     'django.middleware.csrf.CsrfViewMiddleware',
6     'django.contrib.auth.middleware.AuthenticationMiddleware',
7     'django.contrib.messages.middleware.MessageMiddleware',
8     'django.middleware.clickjacking.XFrameOptionsMiddleware',
9 ]
```

Por exemplo, temos o *middleware* **AuthenticationMiddleware**.

Ele é responsável por adicionar a variável **user** a todas as requisições. Assim, você pode, por exemplo, mostrar o usuário logado no seu *template*:

```
1 <li>
2   <a href="{% url 'profile' id=user.id %}">
3     Olá, {{ user.email }}
4   </a>
5 </li>
```

Você pode pesquisar e perceber que em lugar nenhum em nosso código nós adicionamos a variável **user** ao Contexto das requisições.

Não é muito comum, mas pode ser que você tenha que adicionar algum comportamento **antes** de começar a tratar a Requisição ou **depois** de formar a Resposta.

Portante, veremos agora como podemos criar o nosso próprio *middleware*.

Um *middleware* é um método *callable* (que tem uma implementação do método **__call__()**) que recebe uma **requisição** e retorna uma **resposta** e, assim como uma *View*, pode ser escrito como **função** ou como **Classe**.

Um exemplo de *middleware* escrito como função é:

```
1 def middleware_simples(get_response):
2
3     # Código de inicialização do Middleware
4
5     def middleware(request):
6         # Código a ser executado antes da View e
7         # antes de outros middlewares serem executados
8
9         response = get_response(request)
10
11        # Código a ser executado após a execução
12        # da View que irá processar a requisição
13
14    return middleware
15
16 return middleware
```

E como Classe:

```
1 class MiddlewareSimples:
2     def __init__(self, get_response):
3         self.get_response = get_response
4
5         # Código de inicialização do Middleware
6
7     def __call__(self, request):
8         # Código a ser executado antes da View e
9         # antes de outros middlewares serem executados
10
11        response = self.get_response(request)
12
13        # Código a ser executado após a execução
14        # da View que irá processar a requisição
15
16    return response
17
```

Como cada *Middleware* é executado de maneira encadeada, do topo da lista **MIDDLEWARE** para o fim, a **saída de um é a entrada do próximo.**

O método `get_response()` pode ser a própria *View*, caso ela seja a última configurada no **MIDDLEWARE** do `settings.py`, ou o próximo *middleware* da cadeia.

Utilizando a construção do *middleware* via Classe, nós temos três métodos importantes:

O método `process_view`

Assinatura: `process_view(request, func, args, kwargs)`

Esse método é chamado **logo antes do Django executar a View** que vai processar a requisição e possui os seguintes parâmetros:

- `request` é o objeto `HttpRequest`.
- `func` é a própria *view* que o Django está para chamar ao final da cadeia de *middlewares*.
- `args` é a lista de parâmetros posicionais que serão passados à *view*.
- `kwargs` é o *dict* contendo os argumentos nomeados (*keyword arguments*) que serão passados à *view*.

Esse método deve retornar `None` ou um objeto `HttpResponse`:

- Caso retorne `None`, o Django entenderá que **deve continuar** a cadeia de *Middlewares*.

- Caso retorne `HttpResponse`, o Django entenderá que a resposta **está pronta** para ser enviada de volta e não vai se preocupar em chamar o resto da cadeia de *Middlewares*, nem a *view* que iria processar a requisição.

O método `process_exception`

Assinatura: `process_exception(request, exception)`

Esse método é chamado quando uma *View* lança uma exceção e deve retornar ou `None` ou `HttpResponse`.

Caso retorne um objeto `HttpResponse`, o Django irá aplicar o *middleware* de resposta e o *middleware* de *template*, retornando a requisição ao *browser*.

- `request` é o objeto `HttpRequest`
- `exception` é a exceção propriamente dita lançada pela view.

O método `process_template_response`

Assinatura: `process_template_response(request, response)`

Esse método é chamado logo após a *View* ter terminado sua execução caso a resposta tenha uma chamada ao método `render()` indicando que a resposta possui um *template*.

Possui os seguintes parâmetros:

- `request` é um objeto `HttpRequest`.
- `response` é o objeto `TemplateResponse` retornado pela `view` ou por outro `middleware`.

Agora vamos criar um `middleware` um pouco mais complexo para exemplificar o que foi dito aqui!

Vamos supor que queremos um `middleware` que filtre requisições e só processe aquelas que venham de uma determinada lista de IP's.

Esse `middleware` é muito útil quando temos, por exemplo, um conjunto de servidores com IP fixo que vão se conectar entre si. Você poderia, por exemplo, ter uma configuração no seu `settings.py` chamada `ALLOWED_SERVERS` contendo a lista de IP autorizados a se conectar ao seu serviço.

Para isso, precisamos abrir o cabeçalho das requisições que chegam no nosso servidor e verificar se o IP de origem está autorizado.

Como precisamos dessa lógica **antes** da requisição chegar na `View`, vamos adicioná-la ao método `process_view`, da seguinte forma:

```
1 class FiltraIPMiddleware:  
2  
3     def __init__(self, get_response=None):  
4         self.get_response = get_response  
5  
6     def __call__(self, request):  
7         response = self.get_response(request)  
8  
9         return response
```

```
10
11 def process_view(request, func, args, kwargs):
12     # Lista de IPs autorizados
13     ips_autorizados = ['127.0.0.1']
14
15     # IP do usuário
16     ip = request.META.get('REMOTE_ADDR')
17
18     # Verifica se o IP do está na lista de IPs autorizados
19     if ip not in ips_autorizados:
20         # Se usuário não autorizado > HTTP 403 (Não Autorizado)
21         return HttpResponseForbidden(
22             "IP não autorizado"
23         )
24
25     # Se for autorizado, não fazemos nada
26     return None
```

Depois disso, precisamos registrar nosso *middleware* no arquivo de configurações `settings.py` (na configuração `MIDDLEWARE`):

```
1 MIDDLEWARE =
2     'django.middleware.security.SecurityMiddleware',
3     'django.contrib.sessions.middleware.SessionMiddleware',
4     'django.middleware.common.CommonMiddleware',
5     'django.middleware.csrf.CsrfViewMiddleware',
6     'django.contrib.auth.middleware.AuthenticationMiddleware',
7     'django.contrib.messages.middleware.MessageMiddleware',
8     'django.middleware.clickjacking.XFrameOptionsMiddleware',
9
10    # Nosso Middleware
11    'helloworld.middlewares.FiltrarIPMiddleware',
12 ]
```

Agora, podemos testar seu funcionamento alterando a lista `ips_autorizados`:

- Coloque `ips_autorizados = ['127.0.0.1']` e tente acessar alguma URL da nossa aplicação: devemos conseguir acessar normalmente nossa aplicação, pois como estamos executando o servidor localmente, nosso IP será 127.0.0.1 e, portanto, passaremos no teste.
- Coloque `ips_autorizados = []` e tente acessar alguma URL da nossa aplicação: deve aparecer a mensagem “**IP não autorizado**”, pois nosso IP (127.0.0.1) não está autorizado a acessar o servidor.

Conclusão do Capítulo

Nesse capítulo vimos vários conceitos sobre os tipos de *Views* (funções e classes), os principais tipos de CBV (*Class Based Views*), como mapear suas URL para suas *views* através do *URLConf*, como entender o fluxo da sua requisição utilizando o debug da sua IDE, como utilizar os poderosos **Forms** do Django e como utilizar *middlewares* para adicionar camadas extras de processamento às requisições e respostas que chegam e saem da nossa aplicação.

No próximo capítulo, vamos melhorar a interface com o usuário através da **Camada de *Templates*** do Django!

Capítulo IV

CAMADA TEMPLATE

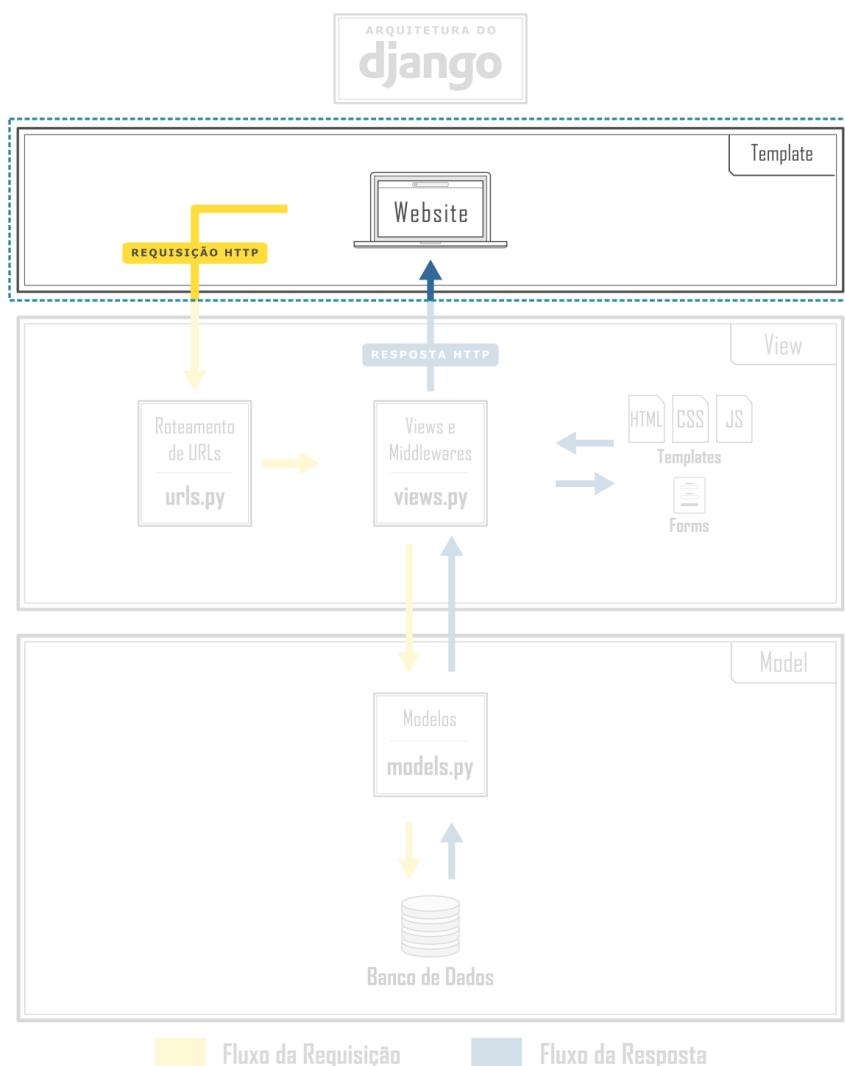
Chegamos ao nosso último capítulo do noss ebook!

Nesse capítulo vamos aprender a configurar, customizar e estender *templates*, como utilizar os filtros e *tags* do Django, como criar *tags* e filtros customizados e um pouquinho de *Bootstrap*, para deixar as páginas **bonitonas**!

A **Camada Template** é quem dá cara à nossa aplicação, isto é, faz a interface com o usuário. É nela que se encontra o código Python, responsável por renderizar nossas páginas web, e os arquivos HTML, CSS e Javascript que darão vida à nossa aplicação!

Onde estamos...

Primeiro, vamos relembrar onde estamos no fluxo requisição/resposta do nosso servidor Django:



Agora, estamos na camada que faz a interface do nosso código Python/Django com o usuário, interagindo, trocando informações, captando dados de *input* e gerando dados de *output*.

SPOILER ALERT: Nesse post vamos concentrar nossos esforços em entender a camada de templates para **construção de páginas**. Nesse momento, não vamos focar na implementação da lógica por trás da Engine de templates, pois acredito que é algo que dificilmente você se verá fazendo, ok?!

Vamos começar pelo começo: **o que é um Template?**

Definição de *Template*

Basicamente, um *template* é um arquivo de texto que pode ser transformado em outro arquivo (um arquivo HTML, um CSS, um CSV, etc).

Um *template* no Django contém:

- **Variáveis** que podem ser substituídas por valores, a partir do processamento por uma *Engine de Templates* (núcleo ou “motor” de *templates*). Usamos os marcadores `{{ variável }}`.
- **Tags** que controlam a lógica do *template*. Usamos com `{% tag %}`.
- **Filtros** que adicionam funcionalidades ao *template*. Usamos com `{{ variável|filtro }}`.

Por exemplo, abaixo está representado um *template* mínimo que demonstra alguns conceitos básicos:

```
1  {# base.html contém o template que usaremos como esqueleto #}
2  {% extends "base.html" %}
3
4  {% block conteudo %}
5      <h1>{{ section.title }}</h1>
6
7  {% for f in funcionarios %}
8      <h2>
9          <a href="{% url 'website:funcionario_detalhe' pk=f.id %}">
10             {{ funcionario.nome|upper }}
11         </a>
12     </h2>
13 {% endfor %}
14 {% endblock %}
```

Alguns pontos importantes:

- **Linha 1:** Escrevemos comentário com a tag `{# comentário #}`. Eles serão processados pelo *Engine* e não estarão na página resultante.
- **Linha 2:** Utilizamos `{% extends "base.html" %}` para estender de um *template*, ou seja, utilizá-lo como base, passando o caminho para ele.
- **Linha 4:** Podemos facilitar a organização do *template*, criando blocos com `{% block nome_do_bloco %}{% endblock %}`.
- **Linha 5:** Podemos interpolar variáveis vindas do servidor com nosso *template* com `{{ secao.titulo }}` - dessa forma, estamos acessando o atributo `titulo` do objeto `secao` (que deve estar no **Contexto** da resposta).
- **Linha 7:** É possível iterar sobre objetos de uma lista através da tag `{% for objeto in lista %}{% endfor %}`.

- **Linha 10:** Podemos utilizar **filtros** para aplicar alguma função à algum conteúdo. Nesse exemplo, estamos aplicando o filtro `upper`, que transforma todos os caracteres da `string` em maiúsculos, no conteúdo de `funcionario.nome`. Também é possível encadear filtros, por exemplo: `{{ funcionario.nome|upper|cut:" " }}`

Para facilitar a manipulação de *templates*, os desenvolvedores do Django criaram uma linguagem que contém todos esses elementos.

Chamaram-na de **DTL - Django Template Language**! Veremos mais dela nesse capítulo!

Para começarmos a utilizar os *templates* do Django, é necessário primeiro **configurar sua utilização**.

E é isso que veremos agora!

Configuração

Se você já deu uma espiada no nosso arquivo de configurações, o `settings.py`, você já deve ter visto a seguinte configuração:

```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [],
5         'APP_DIRS': True,
6         'OPTIONS': {},
7     },
8 ]
```

Mas você já se perguntou **o que essa configuração quer dizer?**

Nela:

- **BACKEND** é o caminho para uma classe que implementa a API de *templates* do Django.
- **DIRS** define uma lista de diretórios onde o Django deve procurar pelos *templates*. A ordem da lista define a ordem de busca.
- **APP_DIRS** define se o Django deve procurar por *templates* dentro dos diretórios dos *apps* instalados em **INSTALLED_APPS**.
- **OPTIONS** contém configurações específicas do **BACKEND** escolhido, ou seja, dependendo do *backend* de *templates* que você escolher, você poderá configurá-lo utilizando parâmetros em **OPTIONS**.

Por ora, vamos utilizar as configurações padrão “de fábrica” pois elas já nos atendem!

Agora, vamos ver sobre a tal **Django Template Language**!

Django Template Language

A *DTL* é a linguagem padrão de *templates* do Django. Ela é simples, porém poderosa.

Dando uma olhada na sua [documentação](#), podemos ver a **filosofia** da **DTL** (traduzido):

Se você tem alguma experiência em programação, ou se você está acostumado com linguagens que misturam código de

*programação diretamente no HTML, você deve ter em mente que o sistema de templates do Django não é simplesmente código Python embutido no HTML. Isto é: o sistema de templates foi desenhado para ser a **apresentação**, e não para conter lógica!*

Se você vem de outra linguagem de programação deve ter tido contato com o seguinte tipo de construção: código de programação adicionado diretamente no código HTML (como PHP).

Isto é o **terror** dos *designers* (e não só deles)!

Ponha-se no lugar de um *designer* que não sabe nada sobre programação. Agora imagina você tendo que dar manutenção nos estilos de uma página **LOTADA** de código de programação?!

Complicado, hein?! 🤯

Agora, nada melhor para aprender sobre a *DTL* do que botando a mão na massa e melhorando as páginas da nossa aplicação, né?!

Observação: nesse post eu vou utilizar o [Bootstrap 4](#) para dar um “tapa no visual”.

Template-base

Nosso *template* que servirá de esqueleto deve conter o código HTML que irá se repetir em todas as páginas.

Devemos colocar nele os trechos de código mais comuns das páginas HTML.

Por exemplo, toda página HTML:

- Deve ter as tags: `<html></html>`, `<head></head>` e `<body></body>`.
- Deve ter os *links* para os arquivos estáticos: `<link></link>` e `<script></script>`.
- Quaisquer outros trechos de código que se repitam em nossas páginas.

Você pode fazer o *download* dos arquivos necessários para o nosso projeto [aqui \(Bootstrap\)](#) e [aqui \(jQuery\)](#), que é uma dependência do *Bootstrap*, ou utilizar os arquivos que eu já baixei e estão na pasta `website/static/`.

Faça isso para todos as bibliotecas externas que queira utilizar (ou utilize um [CDN - Content Delivery Network](#)).

Ok! Agora, com os arquivos devidamente colocados na pasta `/static/`, podemos começar com nosso *template*:

```
1 <!DOCTYPE html>
2 <html>
3 {% load static %}
4 <head>
5   <title>
6     {% block title %}Gerenciador de Funcionários{% endblock %}
7   </title>
8
9   <!-- Estilos -->
10  <link rel="shortcut icon" type="image/png"
11    href="{% static 'website/img/favicon.png' %}">
12  <link rel="stylesheet"
13    href="{% static 'website/css/bootstrap.min.css' %}">
```

```
14 <link rel="stylesheet"
15   href="{% static 'website/css/master.css' %}">
16
17 {% block styles %}{% endblock %}
18 </head>
19
20 <body>
21 <nav class="navbar navbar-expand-lg navbar-light">
22   <a class="navbar-brand" href="{% url 'website:index' %}">
23     
24   </a>
25   <button class="navbar-toggler" type="button" data-toggle="collapse"
26     data-target="#conteudo-navbar" aria-controls="conteudo-navbar"
27     aria-expanded="false" aria-label="Ativar navegação">
28     <span class="navbar-toggler-icon"></span>
29   </button>
30
31   <div class="collapse navbar-collapse" id="conteudo-navbar">
32     <ul class="navbar-nav mr-auto">
33       <li class="nav-item active">
34         <a class="nav-link" href="{% url 'website:index' %}">
35           Página Inicial
36         </a>
37       </li>
38       <li class="nav-item">
39         <a class="nav-link" href="{% url 'website:lista_funcionario' %}">
40           Funcionários
41         </a>
42       </li>
43     </ul>
44   </div>
45 </nav>
46
47 {% block conteudo %}{% endblock %}
48
49 <script src="{% static 'website/js/jquery.min.js' %}"></script>
50 <script src="{% static 'website/js/bootstrap.min.js' %}"></script>
51
52 {% block scripts %}{% endblock %}
53
54 <script src="{% static 'website/js/scripts.js' %}"></script>
55 </body>
56 </html>
```

E vamos as explicações:

- `<!DOCTYPE html>` serve para informar ao *browser* do usuário que se trata de uma página HTML5.
- Para que o Django possa carregar dinamicamente os arquivos estáticos do site, utilizamos a tag `static`. Ela vai fazer a busca do arquivo que você quer e fazer a conversão dos *links* corretamente. Para utilizá-la, é necessário primeiro carregá-la. Fazemos isso com `{% load <modulo> %}`. Após seu carregamento, utilizamos a tag `{%static 'caminho/para/arquivo' %}`, passando como parâmetro a localização relativa à pasta `/static/`.
- Podemos definir quaisquer blocos no nosso *template* com a tag `{% blocknome_do_bloco %}{% endblock %}`. Fazemos isso para organizar melhor as páginas que irão estender desse *template*. Podemos passar um valor padrão dentro do bloco (igual está sendo utilizado na [linha 6](#)) - dessa forma caso não seja definido nenhum valor no *template* filho - é aplicado o valor padrão.
- Colocamos nesse *template* os arquivos necessários para o funcionamento do *Bootstrap*, isto é: o jQuery, o CSS e Javascript do *Bootstrap*.
- O *link* para outras páginas da nossa aplicação é feito utilizando-se a tag `{% url 'nome_da_view' parm1 parm2... %}`. Dessa forma, deixamos que o Django cuide da conversão para URLs válidas!
- O conjunto de tags `<nav></nav>` definem a barra superior de navegação com os *links* para as páginas da aplicação. Esse também é um trecho de código presente em todas as páginas, por isso, adicionamos ao *template*. ([Documentação da Navbar - Bootstrap](#))

E pronto! Temos um *template* base!

Agora, vamos customizar a tela principal da nossa aplicação: a **index.html**!

Página Inicial

Template: `website/index.html`

Nossa tela inicial tem o objetivo de apenas mostrar as opções disponíveis ao usuário, que são:

- *Link* para a página de cadastro de novos Funcionários.
- *Link* para a página de listagem de Funcionários.

Primeiramente, precisamos dizer ao Django que queremos utilizar o *template* que definimos acima como base.

Para isso, utilizamos a seguinte *tag* do Django, que serve para que um *template* estenda de outro:

```
1  {% extends "caminho/para/template" %}
```

Com isso, podemos fazer:

```
1  <!-- Estendemos do template base -->
2  {% extends "website/_layouts/base.html" %}
3
4  <!-- Bloco que define o <title></title> da nossa página -->
5  {% block title %}Página Inicial{% endblock %}
6
```

```

7  <!-- Bloco de conteúdo da nossa página -->
8  {% block conteudo %}
9  <div class="container">
10 <div class="row">
11 <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
12 <div class="card">
13 <div class="card-body">
14 <h5 class="card-title">Cadastrar Funcionário</h5>
15 <p class="card-text">
16   Cadastre aqui um novo <code>Funcionário</code>.
17 </p>
18 <a href="{% url 'website:cadastra_funcionario' %}" 
19   class="btn btn-primary">
20   Novo Funcionário
21 </a>
22 </div>
23 </div>
24 </div>
25 <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
26 <div class="card">
27 <div class="card-body">
28 <h5 class="card-title">Lista de Funcionários</h5>
29 <p class="card-text">
30   Veja aqui a lista de <code>Funcionários</code> cadastrados.
31 </p>
32 <a href="{% url 'website:lista_funcionarios' %}" 
33   class="btn btn-primary">
34   Vá para Lista
35 </a>
36 </div>
37 </div>
38 </div>
39 </div>
40 </div>
41 {% endblock %}

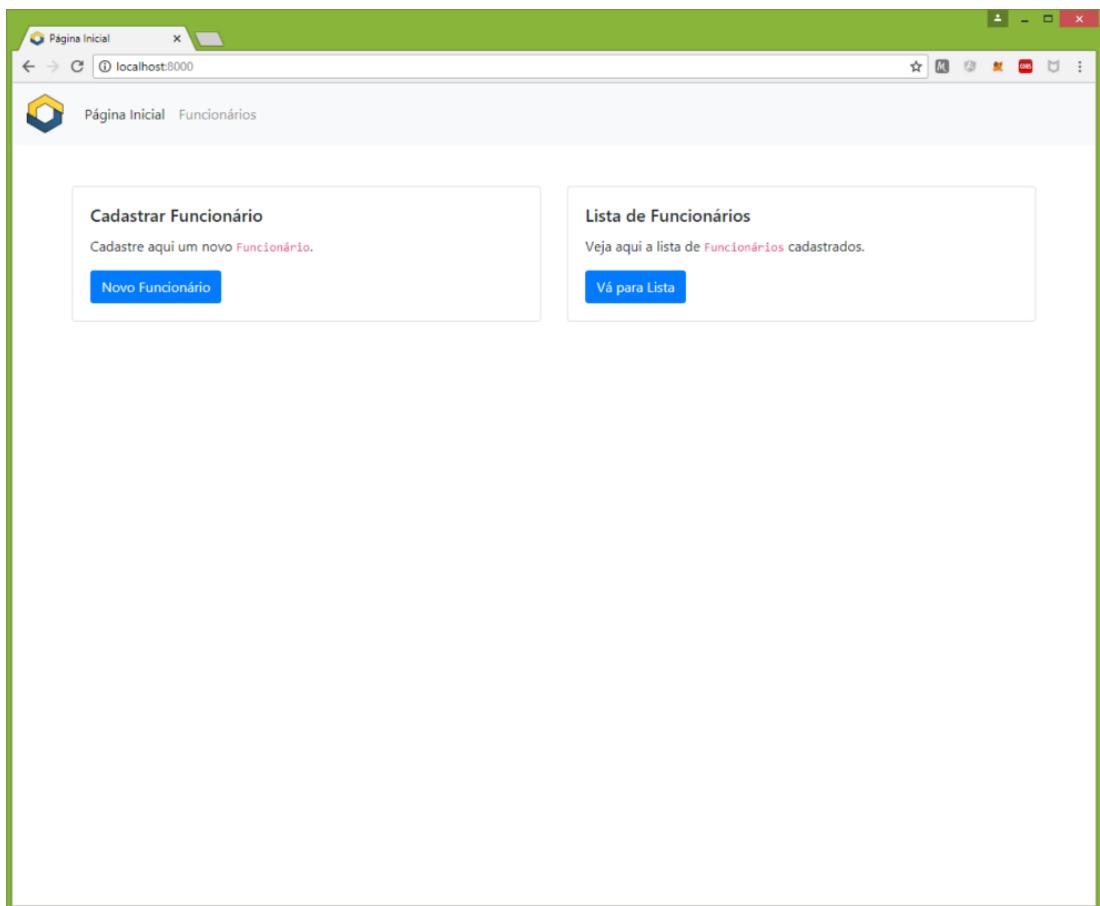
```

Nesse *template*:

- A classe **container do Bootstrap** (linha 9) serve para definir a área útil da nossa página (para que nossa página fique centralizada e não fique ocupando todo o comprimento da tela).

- As classes `row` e `col-*` fazem parte do *sistema Grid do Bootstrap* e nos ajuda a tornar nossa página **responsiva** (que se adapta aos diversos tipos e tamanhos de tela: celular, *tablet*, *desktop* etc...).
- As classes `card*` fazem parte do *component Card do Bootstrap*.
- As classes `btn` e `btn-primary` (*documentação*) são usados para dar o visual de botão à algum elemento.

Com isso, nossa Página Inicial - ou nossa *Homepage* - fica assim:



Top, hein?! 🎉

Agora vamos para a página de cadastro de Funcionários:
a `cria.html`

Template de Cadastro de Funcionários

Template: `website/cria.html`

Nesse *template*, mostramos o formulário para cadastro de novos funcionários.

Se lembra que definimos o formulário `InsereFuncionarioForm` no capítulo passado?

Vamos utilizá-lo nesse template, adicionando-o na `View FuncionarioCreateView`. Dessa forma, ela irá expor um objeto `form` no nosso *template* para que possamos utilizá-lo.

Mas antes de seguir, vamos instalar uma biblioteca que vai nos auxiliar **e muito** a renderizar os campos de *input* do nosso formulário: [a *Widget Tweaks*!](#)

Com ela, nós temos maior liberdade para customizar os campos de *input* do nosso formulário (adicionando classes CSS e/ou atributos, por exemplo).

Para isso, primeiro nós a instalamos com:

```
1 pip install django-widget-tweaks
```

Depois a adicionamos a lista de *apps* instalados, no arquivo `helloworld/settings.py`:

```
1 INSTALLED_APPS = [
2     ...
3     'widget_tweaks',
4     ...
5 ]
```

E, no *template* onde formos utilizá-lo, carregamos ela com `{% load widget_tweaks %}`!

E pronto, agora podemos utilizar a *tag* que irá renderizar os campos do formulário, a `render_field`:

```
1 {% render_field nome_do_campo parametros %}
```

Para alterar como o input será renderizado, utilizamos os parâmetros da *tag*. Dessa forma, podemos alterar o código HTML resultante. Portanto, nosso template pode ser escrito assim:

```
1 {% extends "website/_layouts/base.html" %}
2
3 {% load widget_tweaks %}
4
5 {% block title %}Cadastro de Funcionários{% endblock %}
6
7 {% block conteudo %}
8 <div class="container">
9   <div class="row">
10    <div
11      class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
12        <div class="card">
```

```
13 <div class="card-body">
14   <h5 class="card-title">Cadastro de Funcionário</h5>
15   <p class="card-text">
16     Complete o formulário abaixo para cadastrar
17     um novo <code>Funcionário</code>.
18   </p>
19   <form method="post">
20     <!-- Não se esqueça dessa tag -->
21     {% csrf_token %}
22
23     <!-- Nome -->
24     <div class="input-group mb-3">
25       <div class="input-group-prepend">
26         <span class="input-group-text">Nome</span>
27       </div>
28       {% render_field form.nome class+="form-control" %}
29     </div>
30
31     <!-- Sobrenome -->
32     <div class="input-group mb-3">
33       <div class="input-group-prepend">
34         <span class="input-group-text">Sobrenome</span>
35       </div>
36       {% render_field form.sobrenome class+="form-control" %}
37     </div>
38
39     <!-- CPF -->
40     <div class="input-group mb-3">
41       <div class="input-group-prepend">
42         <span class="input-group-text">CPF</span>
43       </div>
44       {% render_field form.cpf class+="form-control" %}
45     </div>
46
47     <!-- Tempo de Serviço -->
48     <div class="input-group mb-3">
49       <div class="input-group-prepend">
50         <span class="input-group-text">
51           Tempo de Serviço
52         </span>
53       </div>
54       {% render_field form.tempo_de_servico class+="form-control" %}
55     </div>
```

```

56      <!-- Remuneração -->
57      <div class="input-group mb-3">
58          <div class="input-group-prepend">
59              <span class="input-group-text">Remuneração</span>
60          </div>
61          {% render_field form.remuneracao class+="form-control" %}
62      </div>
63
64          <button class="btn btn-primary">Enviar</button>
65      </form>
66  </div>
67  </div>
68  </div>
69  </div>
70  </div>
71 </div>
72 {% endblock %}

```

Aqui:

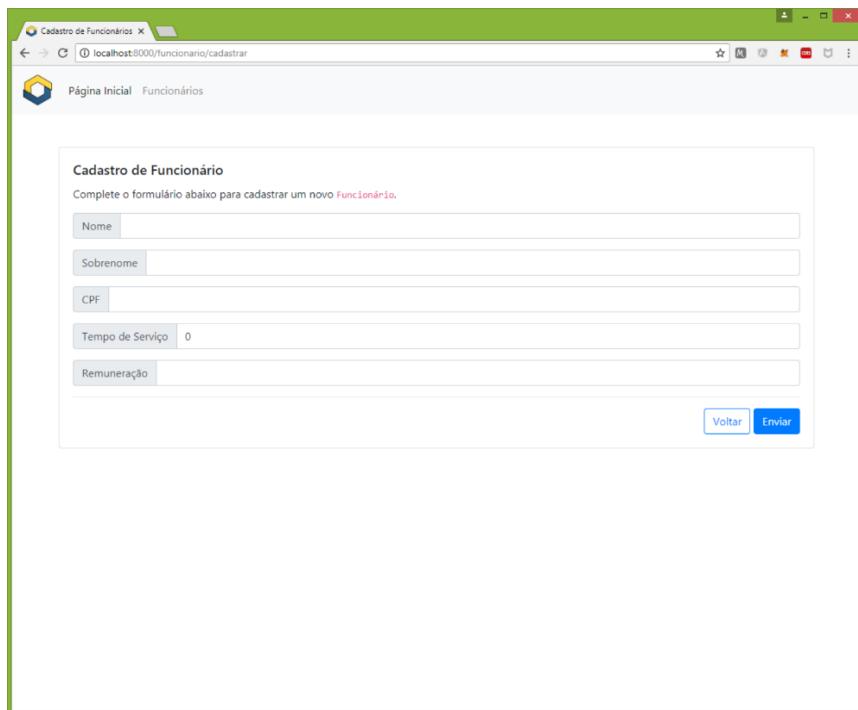
- Utilizamos, novamente as classes `container`, `row`, `col-*` e `card*` do Bootstrap.
- Conforme mencionei no **capítulo passado**, devemos adicionar a tag `{% csrf_token %}` para evitar ataques de *Cross Site Request Forgery*.
- As *classes Input Group do Bootstrap* `input-group`, `input-group-prepend` e `input-group-text` servem para customizar o estilo dos elementos `<input />`.
- Para aplicar a classe `form-control` do Bootstrap, utilizamos `{% render_field form.campo class+='form-control' %}`

Observação: É possível adicionar a classe CSS `form-control` diretamente no nosso Form `InsereFuncionarioForm`, da seguinte forma:

```
1 class InsereFuncionarioForm(forms.ModelForm):
2     nome = forms.CharField(
3         max_length=255,
4         widget=forms.TextInput(
5             attrs={
6                 'class': "form-control"
7             }
8         )
9     )
10    ...
```

Mas eu não aconselho, pois deixa nosso código extremamente acoplado. Veja que para mudar a classe CSS (atributo da interface) teremos que mudar código do backend. Por isso, aconselho a utilização de bibliotecas como o Widget Tweaks, pois alteramos apenas no template!

Com isso, nosso formulário deve ficar assim:



Agora, vamos desenvolver o *template* de listagem de Funcionários.

Template de Listagem de Funcionários

Template: website/lista.html

Nessa página, nós queremos mostrar o conjunto de Funcionários cadastrado no banco de dados e as ações que o usuário pode tomar: atualizar os dados do Funcionário ou excluí-lo.

Se lembra da *view* FuncionarioListView? Ela é responsável por buscar a lista de Funcionários e expor um objeto chamado **funcionarios** para iteração no *template*.

Podemos construir nosso *template* da seguinte forma:

```
1  {% extends "website/_layouts/base.html" %}  
2  
3  {% block title %}Lista de Funcionários{% endblock %}  
4  
5  {% block conteudo %}  
6    <div class="container">  
7      <div class="row">  
8        <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">  
9          <div class="card">  
10            <div class="card-body">  
11              <h5 class="card-title">Lista de Funcionário</h5>  
12  
13              {% if funcionarios|length > 0 %}  
14                <p class="card-text">  
15                  Aqui está a lista de <code>Funcionários</code>  
16                  cadastrados.  
17                </p>
```

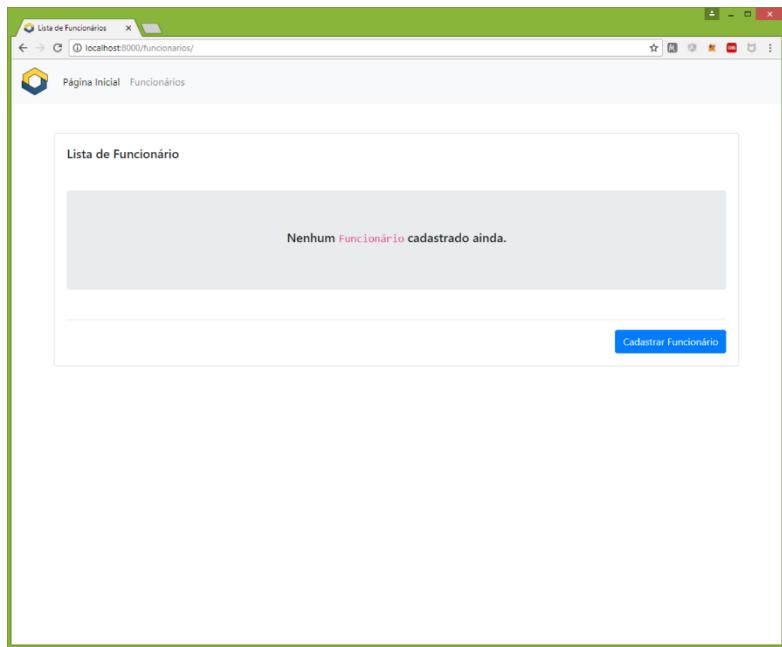
```
18     <table class="table">
19         <thead class="thead-dark">
20             <tr>
21                 <th>ID</th>
22                 <th>Nome</th>
23                 <th>Sobrenome</th>
24                 <th>Tempo de Serviço</th>
25                 <th>Remuneração</th>
26                 <th>Ações</th>
27             </tr>
28         </thead>
29
30         <tbody>
31             {% for f in funcionarios %}
32                 <tr>
33                     <td>{{ f.id }}</td>
34                     <td>{{ f.nome }}</td>
35                     <td>{{ f.sobrenome }}</td>
36                     <td>{{ f.tempo_de_servico }}</td>
37                     <td>{{ f.remuneracao }}</td>
38                     <td>
39                         <a href="{% url 'website:atualiza_funcionario' pk=f.id %}"
40                             class="btn btn-info">
41                             Atualizar
42                         </a>
43                         <a href="{% url 'website:deleta_funcionario' pk=f.id %}"
44                             class="btn btn-outline-danger">
45                             Excluir
46                         </a>
47                     </td>
48                 </tr>
49             {% endfor %}
50         </tbody>
51     </table>
52     {% else %}
53         <div class="text-center mt-5 mb-5 jumbotron">
54             <h5>Nenhum <code>Funcionário</code> cadastrado ainda.</h5>
55         </div>
56     {% endif %}
57     <hr />
58     <div class="text-right">
```

```
57     <a class="btn btn-primary"  
58     href="{% url 'website:cadastra_funcionario' %}">  
59         Cadastrar Funcionário  
60     </a>  
61 </div>  
62 </div>  
63 </div>  
64 </div>  
65 </div>  
66 {% endblock %}
```

Nesse *template*:

- Utilizamos as seguintes classes [do Bootstrap para estilizar as tabelas](#): `table` para estilizar a tabela e `thead-dark` para escurecer o cabeçalho.
- Na **linha 13**, utilizamos o **filtro `length`** para verificar se a lista de funcionários está vazia. Se ela contiver dados, a tabela é mostrada. Se ela estiver vazia, será renderizado o [componente *Jumbotron* do Bootstrap](#) com o texto “*Nenhum Funcionário cadastrado ainda*”.
- Utilizamos a tag `{% for funcionario in funcionarios %}` na **linha 30** para iterar sobre a lista `funcionarios`.
- Nas **linhas 39 e 46** fazemos o *link* para as páginas de atualização e exclusão do usuário.

O resultado, sem Funcionários cadastrados, deve ser esse:



E com um Funcionário cadastrado:

A screenshot of a web browser window titled "Lista de Funcionários". The address bar shows "localhost:8000/funcionarios/". The main content area is a white box titled "Lista de Funcionário". It contains a message "Aqui está a lista de Funcionários cadastrados." followed by a table. The table has columns: ID, Nome, Sobrenome, Tempo de Serviço, Remuneração, and Ações. There is one row of data: ID 3, Nome Vinicius, Sobrenome Ramos, Tempo de Serviço 1, Remuneração 10000.00, and Ações with "Atualizar" and "Excluir" buttons. At the bottom right of the box is a blue button labeled "Cadastrar Funcionário".

ID	Nome	Sobrenome	Tempo de Serviço	Remuneração	Ações
3	Vinicius	Ramos	1	10000.00	Atualizar Excluir

Quando o usuário clicar em “**Excluir**”, ele será levado para a página `exclui.html` e quando clicar em “**Atualizar**”, ele será levado para a página `atualiza.html`.

Vamos agora construir a página de Atualização de Funcionários!

Template de Atualização de Funcionários

Template: `website/atualiza.html`

Nessa página, queremos que o usuário possa ver os dados atuais do Funcionário e possa atualizá-los, conforme sua vontade. Para isso utilizamos a `View FuncionarioUpdateView` que implementamos no capítulo passado.

Ela expõe um formulário com os campos do modelo preenchidos com os dados atuais para que o usuário possa alterar.

Vamos utilizar novamente a biblioteca `Widget Tweaks` para facilitar a renderização dos campos de `input`.

Abaixo, como podemos fazer nosso *template*:

```
1  {% extends "website/_layouts/base.html" %}  
2  
3  {% load widget_tweaks %}  
4  
5  {% block title %}Atualização de Funcionário{% endblock %}  
6  
7  {% block conteudo %}  
8    <div class="container">
```

```
10 <div class="row">
11   <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
12     <div class="card">
13       <div class="card-body">
14         <h5 class="card-title">
15           Atualização de Dados do Funcionário
16         </h5>
17         <form method="post">
18           <!-- Não se esqueça dessa tag -->
19           {% csrf_token %}

20
21           <!-- Nome -->
22           <div class="input-group mb-3">
23             <div class="input-group-prepend">
24               <span class="input-group-text">Nome</span>
25             </div>
26             {% render_field form.nome class+="form-control" %}
27           </div>

28
29           <!-- Sobrenome -->
30           <div class="input-group mb-3">
31             <div class="input-group-prepend">
32               <span class="input-group-text">Sobrenome</span>
33             </div>
34             {% render_field form.sobrenome class+="form-control" %}
35           </div>

36
37           <!-- CPF -->
38           <div class="input-group mb-3">
39             <div class="input-group-prepend">
40               <span class="input-group-text">CPF</span>
41             </div>
42             {% render_field form.cpf class+="form-control" %}
43           </div>

44
45           <!-- Tempo de Serviço -->
46           <div class="input-group mb-3">
47             <div class="input-group-prepend">
48               <span class="input-group-text">Tempo de Serviço</span>
49             </div>
50
51
52
```

```
53      {% render_field form.tempo_de_servico class+="form-control" %}
```

```
54  l" %}
```

```
55  </div>
```

```
56
```

```
57  <!-- Remuneração -->
```

```
58  <div class="input-group mb-3">
```

```
59    <div class="input-group-prepend">
```

```
60      <span class="input-group-text">Remuneração</span>
```

```
61    </div>
```

```
62    {% render_field form.remuneracao class+="form-control" %}
```

```
63  </div>
```

```
64    <button class="btn btn-primary">Enviar</button>
```

```
65  </form>
```

```
66  </div>
```

```
67  </div>
```

```
68  </div>
```

```
69  </div>
```

```
70  </div>
```

```
71  </div>
```

```
72  {% endblock %}
```

Nesse *template*, não temos nada de novo.

Perceba que seu código é similar ao *template* de adição de Funcionários, com os campos sendo renderizados com a *tag render_field*.

Como nossa *View* herda de `UpdateView`, o objeto `form` já vem populado com os dados do modelo em questão (aquele cujo `id` foi enviado ao se clicar no botão de edição).

Sua interface deve ficar similar à:

Atualizar Dados de Funcionário

Nome	Vinicius
Sobrenome	Ramos
CPF	11122233344
Tempo de Serviço	1
Remuneração	5000,00

[Voltar](#) [Enviar](#)

E por último, temos o *template* de exclusão de Funcionários.

***Template* de Exclusão de Funcionários**

Template: [website/exclui.html](#)

A função dessa página é mostrar uma página de confirmação para o usuário antes da exclusão de um Funcionário. Essa página vai concretizar a sua exclusão.

A view que fizemos, a `FuncionarioDeleteView`, facilita bastante nossa vida. Com ela, basta dispararmos uma requisição `POST` para a URL configurada, que o Funcionário será deletado!

Dessa forma, nosso objetivo se resume à:

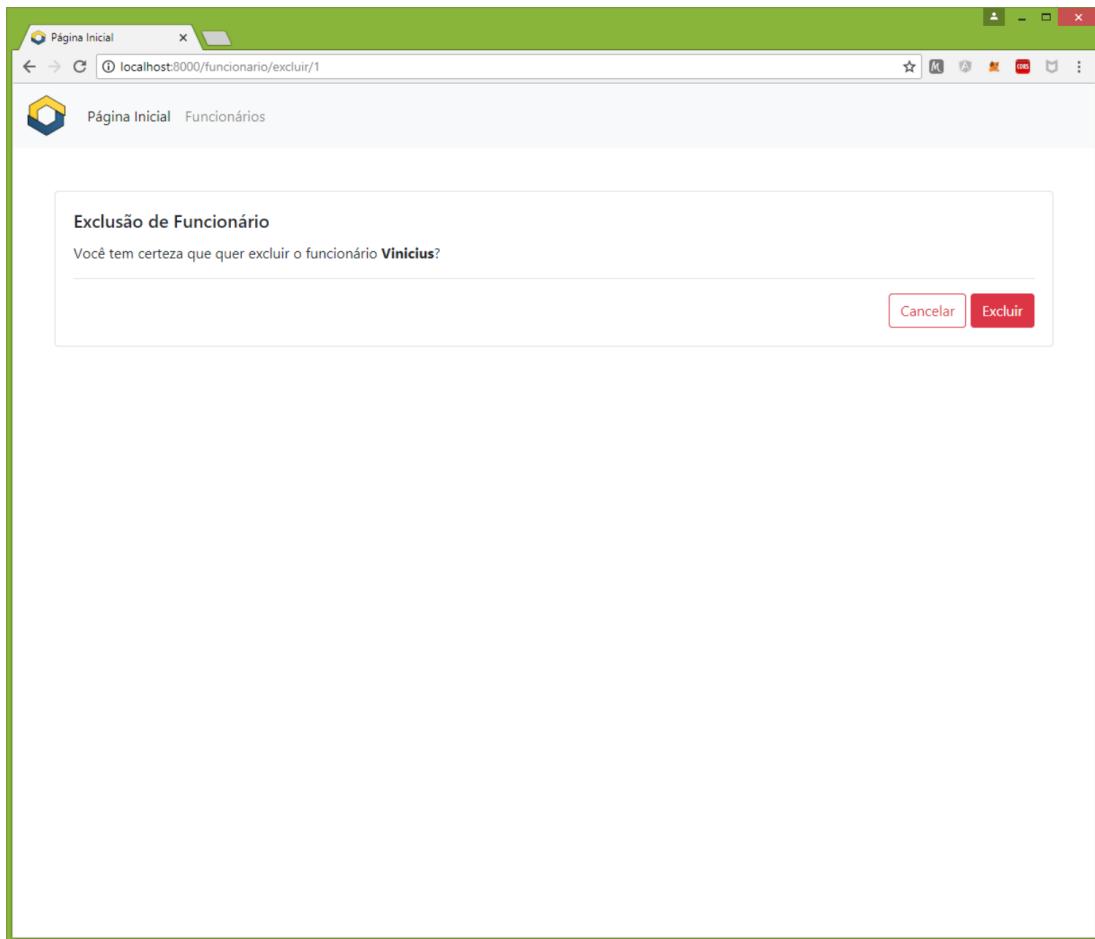
```
1  <!-- Estendemos do template base -->
2  {% extends "website/_layouts/base.html" %}

3
4  <!-- Bloco que define o <title></title> da nossa página -->
5  {% block title %}Página Inicial{% endblock %}

6
7  <!-- Bloco de conteúdo da nossa página -->
8  {% block conteudo %}
9   <div class="container mt-5">
10    <div class="card">
11      <div class="card-body">
12        <h5 class="card-title">Exclusão de Funcionário</h5>
13        <p class="card-text">
14          Você tem certeza que quer excluir o funcionário
15          <b>{{ funcionario.nome }}</b>?
16        </p>
17        <form method="post">
18          {% csrf_token %}
19          <hr />
20          <div class="text-right">
21            <a href="{% url 'website:lista_funcionarios' %}">
22              class="btn btn-outline-danger">
23                Cancelar
24              </a>
25              <button class="btn btn-danger">Excluir</button>
26            </div>
27          </form>
28        </div>
29      </div>
30    </div>
31  {% endblock %}
```

Aqui, nada de novo.

Apenas mostramos o formulário onde o usuário pode decidir excluir ou não o Funcionário, que deve ficar assim:



Pronto!

Com isso, temos todas as páginas do nosso projeto! 😊

Agora vamos ver como construir **tags** e **filtros customizados**!

Tags e Filtros customizados

Sabemos, até agora, que o Django possui uma grande variedade de filtros e *tags* pré-configurados.

Contudo, é possível que, em alguma situação específica, o Django não te ofereça o filtro ou *tag* necessários.

Por isso, ele previu a possibilidade de você **construir seus próprios filtros e tags!**

Portanto, vamos construir uma *tag* que irá nos dizer o **tempo atual formatado** e um **filtro** que irá retornar a **primeira letra da string passada**.

Para isso, vamos começar com a **configuração** necessária!

Configuração

Os filtros e *tags* customizados residem em uma pasta específica da nossa estrutura: a **/templatetags**.

Portanto, crie na raíz do app *website* essa pasta (**website/templatetags**) e adicione:

- Um script **`__init__.py`** em branco (para que o Django enxergue como um pacote Python).
- O script **`tempo_atual.py`** em branco referente à nossa *tag*
- O script **`primeira_letra.py`** em branco referente ao nosso filtro.

Nossa estrutura, portanto, deve ficar:

```
1 - website/
2   ...
3     - templatetags/
4       - __init__.py
5       - tempo_atual.py
6       - primeira_letra.py
7   ...
```

Para que o Django enxergue nossas *tags* e filtros é necessário que o *app* onde eles estão instalados esteja configurada na lista **INSTALLED_APPS** do **settings.py** (no nosso caso, **website** já está lá, portanto, nada a fazer aqui).

Também é necessário carregá-los com o **{% load filtro/tag %}**.

Vamos escolher um para começar: vamos começar com o **filtro**.

Vamos chamá-lo de **primeira_letra** e, quando estiver pronto, iremos utilizá-lo da seguinte maneira:

```
1 <p>{{ valor|primeira_letra }}</p>
```

Filtro **primeira_letra**

Filtros customizados são basicamente funções que recebem um ou dois argumentos. São eles:

- O valor do *input*.

- O valor do argumento - que pode ter um valor padrão ou não receber nenhum valor.

No nosso filtro `{{ valor|primeira_letra }}`:

- `valor` será o `value`.
- Nosso filtro não irá receber argumentos, portanto não foi passado nada para ele.

Para ser um filtro válido, é necessário que o código dele contenha uma variável chamada `register` que seja uma instância de `template.Library` (onde todos os tags e filtros são registrados).

Isso **define** um filtro!

Outra questão importante são as **Exceções**. Como a *engine* de *templates* do Django **não provê tratamento de exceção** ao executar o código do filtro, qualquer exceção será exposta como uma exceção do próprio servidor.

Por isso, nosso filtro deve **evitar lançar exceções** e, ao invés disso, deve retornar um valor padrão.

Vamos ver um exemplo de filtro do Django.

Abra o arquivo `django/template/defaultfilter.py`. Lá temos a definição de diversos filtros que podemos utilizar em nossos templates (eu separei alguns e vou explicar ali embaixo).

Lá temos o exemplo do filtro `lower`:

```
1 @register.filter(is_safe=True)
2 @stringfilter
3 def lower(value):
4     """Convert a string into all lowercase."""
5     return value.lower()
```

Nele:

- `@register.filter(is_safe=True)` é um *decorator* utilizado para registrar sua função **como um filtro para o Django**. Só assim o *framework* vai enxergar seu código (saiba mais sobre *decorators* no nosso post - [Domine Decorators em Python](#)).
- `@stringfilter` é um *decorator* utilizado para dizer ao Django que seu filtro espera uma string como argumento.

Com isso, vamos agora codificar e registrar nosso filtro!

Uma forma de pegarmos a primeira letra de uma string é transformá-la em lista e pegar o elemento de índice `[0]`, da seguinte forma:

```
1 from django import template
2 from django.template.defaultfilters import stringfilter
3
4 register = template.Library()
5
6 @register.filter
7 @stringfilter
8 def primeira_letra(value):
9     return list(value)[0]
```

Nesse código:

- O código `register = template.Library()` é necessário para pegarmos uma instância da biblioteca de filtros do Django. Com ela, podemos registrar nosso filtro com `@register.filter`.
- `@register.filter` e `@stringfilter` são os *decorators* que citei aqui em cima.

E agora vamos testar, fazendo o carregamento e utilização em algutm *template*. Para isso, vamos alterar a tabela do *template website/lista.html* para incluir nosso filtro da seguinte forma:

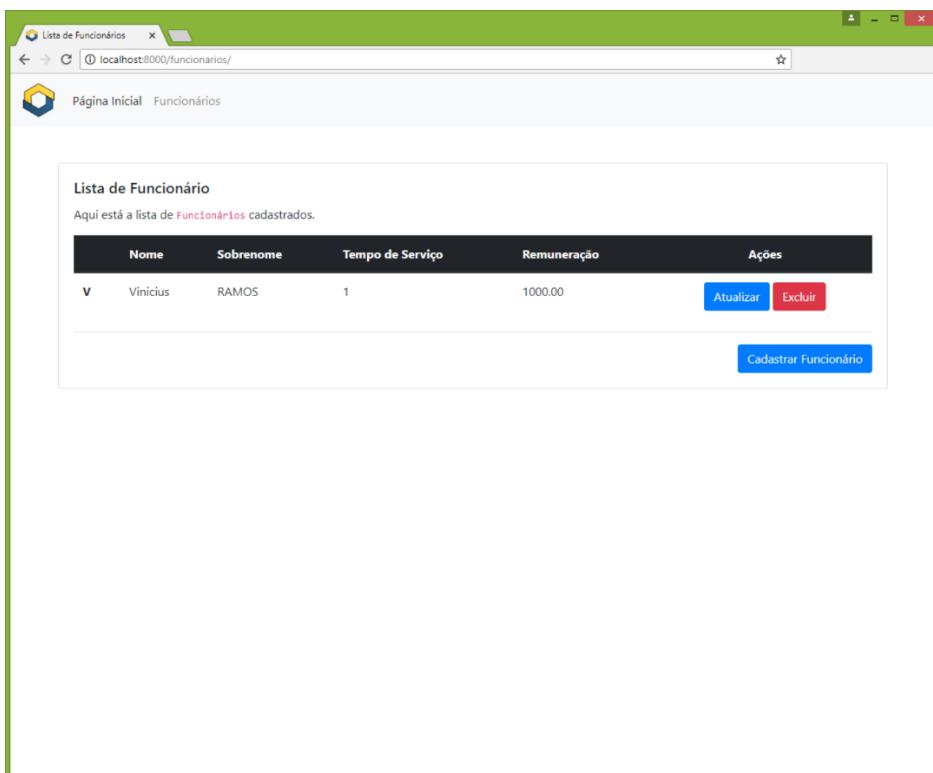
```

1  <!-- Primeiro, carregamos nosso filtro, logo após o extends -->
2  {% load primeira_letra %}
3  ...
4  <table class="table">
5  <thead class="thead-dark">
6    <tr>
7      <th><!-- Retiramos o "ID" aqui --></th>
8      <th>Nome</th>
9      <th>Sobrenome</th>
10     <th>Tempo de Serviço</th>
11     <th>Remuneração</th>
12     <th class="text-center">Ações</th>
13   </tr>
14 </thead>
15 <tbody>
16 {% for f in funcionarios %}<br>
17   <tr>
18     <!-- Aplicamos nosso filtro no atributo funcionario.nome -->
19     <td>{{ f.nome|primeira_letra }}</td>
20     <td>{{ f.nome }}</td>
21     <td>{{ f.sobrenome }}</td>
22     <td>{{ f.tempo_de_servico }}</td>
23     <td>{{ f.remuneracao }}</td>
24     <td class="text-center">
25       <a class="btn btn-primary"

```

```
26     href="{% url 'website:atualiza_funcionario' pk=f.id %}">
27     Atualizar
28   </a>
29   <a class="btn btn-danger"
30     href="{% url 'website:deleta_funcionario' pk=f.id %}">
31     Excluir
32   </a>
33 </td>
34 </tr>
35 {% endfor %}
36 </tbody>
37 </table>
38
39
```

O que resulta em:



E com isso, terminamos nosso **primeiro filtro!**

Agora vamos fazer nossa *tag* customizada: a **tempo_atual**!

Tag **tempo_atual**

De acordo com a documentação do Django, “*tags são mais complexas que filtros pois podem fazer qualquer coisa*“.

Desenvolver uma *tag* pode ser algo bem trabalhoso, dependendo do que você deseja fazer. Mas também pode ser simples.

Como nossa *tag* vai apenas mostrar o tempo atual, sua implementação não deve ser complexa.

Para isso, utilizaremos um “atalho” do Django: a **simple_tag**!

A **simple_tag** é uma ferramenta para construção de *tags* simples (assim como o próprio nome já diz).

Com ela, a criação de *tags* fica similar à criação de filtros, que vimos na seção passada.

Assim como na criação da tag, precisamos incluir uma instância de **template.Library** (para ter acesso à biblioteca de filtros e *tags* do Django), utilizar o *decorator* **@register** (para registrar nossa *tag*) e definir a implementação da nossa função.

Para pegar o tempo atual, podemos utilizar o método **now()** da biblioteca **datetime**. Como queremos formatar a data, também

utilizamos o método `strftime()`, passando como parâmetro a string formatada (`%H` é a hora, `%M` são os minutos e `%S` são os segundos).

Podemos, então, definir nossa *tag* da seguinte forma:

```
1 import datetime
2 from django import template
3
4 register = template.Library()
5
6 @register.simple_tag
7 def tempo_atual():
8     return datetime.datetime.now().strftime('%H:%M:%S')
```

E para utilizá-la, a carregamos com `{% load tempo_atual %}` e a utilizamos em nosso template com `{% tempo_atual %}`.

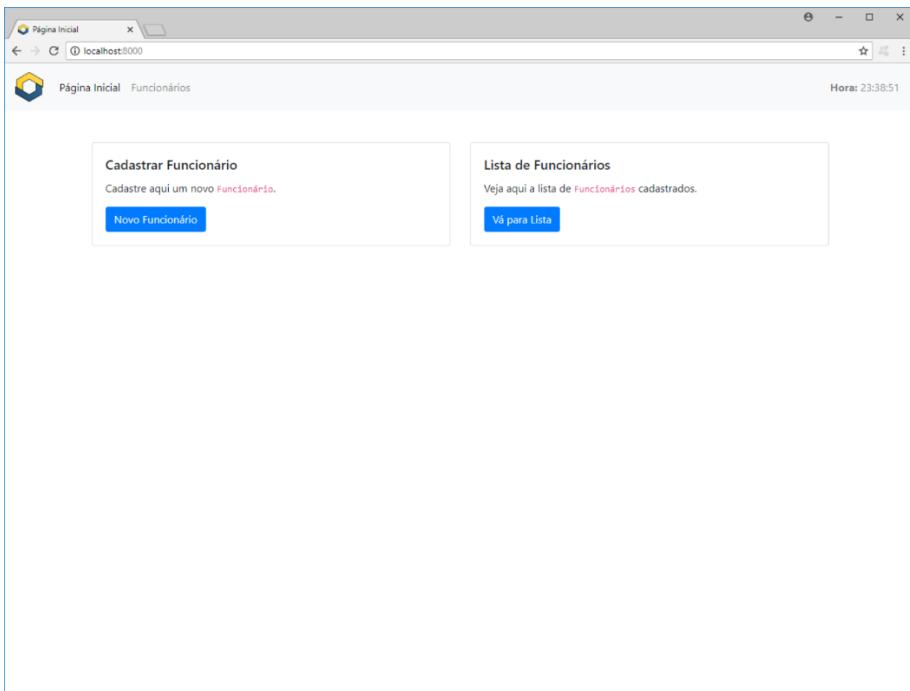
No nosso caso, vamos utilizar nossa *tag* no template-base: o `website/_layouts/base.html`.

Vamos adicionar um novo item à barra de navegação (do lado direito), da seguinte forma:

```
1 <body>
2   <!-- Navbar -->
3   <nav class="navbar navbar-expand-lg navbar-light bg-light">
4     ...
5     <div class="collapse navbar-collapse" id="navbarSupportedContent">
6       <ul class="navbar-nav mr-auto">
7         <li class="nav-item active">
8           <a class="nav-link" href="{% url 'website:index' %}">
9             Página Inicial
10            </a>
11          </li>
```

```
12 <li class="nav-item">
13   <a class="nav-link" href="{% url 'website:lista_funcionarios' %}">
14     Funcionários
15   </a>
16 </li>
17 </ul>
18 <!-- Adicione a lista abaixo -->
19 <ul class="navbar-nav float-right">
20   <li class="nav-item">
21     <!-- Aqui está nosso filtro -->
22     <a class="nav-link" href="#">
23       <b>Hora: </b>{{ tempo_atual | date:"H:i" }}
24     </a>
25   </li>
26 </ul>
27 </div>
28 </nav>
29 ...
```

O resultado deve ser:



Com isso, temos nosso filtro e *tag* customizados!

Agora vamos dar uma olhada nos filtros que estão presentes no próprio Django: os *Built-in Filters*!

Built-in Filters

É possível fazer muita coisa com os filtros que já veem instalados no próprio Django!

Muitas vezes, é melhor você fazer algumas operações no *template* do que fazê-las no *backend*. Sempre verifique a viabilidade de um ou de outro para facilitar sua vida!

Como a lista de *built-in filters* do Django é bem extensa ([veja a lista completa aqui](#)), vou listar aqui os que eu considero mais úteis!

Sem mais delongas, aí vai o primeiro: o **capfirst!!!**

Filtro `capfirst`

O que faz: Torna o primeiro caracter do valor para maiúsculo.

Exemplo:

Entrada: `valor = 'esse é um texto'.`

Utilização:

```
1 {{ valor|capfirst }}
```

Saída:

```
Esse é um texto
```

Filtro **cut**

O que faz: Remove todas as ocorrências do parâmetro no valor passado.

Exemplo:

Entrada: `valor = 'Esse É Um Texto De Testes'`

Utilização:

```
1 {{ valor|cut:" " }}
```

Saída:

```
EsseÉUmTextoDeTestes
```

Filtro **date**

O que faz: Utilizado para formatar datas. Possui uma grande variedade de configurações ([veja aqui](#)).

Exemplo:

Entrada: Objeto `datetime`.

Utilização:

```
1 {{ data|date:'d/m/Y' }}
```

Saída:

```
01/07/2018
```

Filtro **default**

O que faz: Caso o valor seja **False**, utiliza o valor **default**.

Exemplo:

Entrada: **valor = False**

Utilização:

```
1 {{ valor|default:'Nenhum valor' }}
```

Saída:

```
Nenhum valor
```

Filtro **default_if_none**

O que faz: Similar ao filtro **default**, caso o valor seja **None**, utiliza o valor configurado em **default_if_none**.

Exemplo:

Entrada: **valor = None**

Utilização:

```
1 {{ valor|default:'Nenhum valor' }}
```

Saída:

```
Nenhum valor
```

Filtro **divisibleby**

O que faz: Retorna **True** se o valor for divisível pelo argumento.

Exemplo:

Entrada: **valor = 14** e **divisibleby: '2'**

Utilização:

```
1 {{ valor|divisibleby:'2' }}
```

Saída:

```
True
```

Filtro **filesizeformat**

O que faz: Transforma tamanhos de arquivos em valores legíveis.

Exemplo:

Entrada: **valor = 123456789**

Utilização:

```
1 {{ valor|filesizeformat }}
```

Saída:

```
117.7 MB
```

Filtro **first**

O que faz: Retorna o primeiro item em uma lista

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
1 {{ valor|first }}
```

Saída:

```
Marcos
```

Filtro **last**

O que faz: Retorna o último item em uma lista

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
1 {{ valor|last }}
```

Saída:

```
Luiz
```

Filtro **floatformat**

O que faz: Arredonda números com ponto flutuante com o número de casas decimais passado por argumento.

Exemplo:

Entrada: `valor = 14.25145`

Utilização:

```
1 {{ valor|floatformat:"2" }}
```

Saída:

```
14.25
```

Filtro **join**

O que faz: Junta uma lista utilizando a string passada como argumento como separador.

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
1 {{ valor|join:" - " }}
```

Saída:

```
Marcos - João - Luiz
```

Filtro length

O que faz: Retorna o comprimento de uma lista ou string. É muito utilizado para saber se existem valores na lista (se `length > 0`, lista não está vazia).

Exemplo:

Entrada: valor = ['Marcos', 'João']

Utilização:

```
1 {% if valor|length > 0 %}  
2 <p>Lista contém valores</p>  
3 {% else %}  
4 <p>Lista vazia</p>  
5 {% endif %}
```

Saída:

```
<p>Lista contém valores</p>
```

Filtro lower

O que faz: Transforma todos os caracteres de uma string em minúsculas.

Exemplo:

Entrada: valor = PaRaLeLePíPeDo

Utilização:

```
1 {{ valor|lower }}
```

Saída:

```
paralelepípedo
```

Filtro pluralize

O que faz: Retorna um sufixo plural caso o número seja maior que 1.

Exemplo:

Entrada: valor = 12

Utilização:

```
1 Sua empresa tem {{ valor }} Funcionário{{ valor|pluralize:"s" }}
```

Saída:

```
Sua empresa tem 12 Funcionários
```

Filtro random

O que faz: Retorna um item aleatório de uma lista.

Exemplo:

Entrada: valor = [1, 2, 3, 4, 5, 6, 7, 9]

Utilização:

```
1 {{ valor|random }}
```

Sua saída será um valor da lista escolhido randomicamente.

Filtro title

O que faz: Transforma em maiúsculo o primeiro caracter de todas as palavras do texto.

Exemplo:

Entrada: valor = 'primeiro post do blog'

Utilização:

```
1 {{ valor|title }}
```

Saída:

```
Primeiro Post Do Blog
```

Filtro **upper**

O que faz: Transforma em maiúsculo todos caracteres da string.

Exemplo:

Entrada: `valor = texto de testes`

Utilização:

```
1 {{ valor|upper }}
```

Saída:

```
TEXTO DE TESTES
```

Filtro **wordcount**

O que faz: Retorna o número de palavras da string.

Exemplo:

Entrada: `valor = Django é o melhor framework web`

Utilização:

```
1 {{ valor|wordcount }}
```

Saída:

```
6
```

Código

O código completo desenvolvido nesse projeto está **disponível no Github da Python Academy**. [Clique aqui para acessá-lo e baixá-lo!](#)

Para rodar o projeto, execute em seu terminal:

- `pip install -r requirements.txt` para instalar as dependências.
- `python manage.py makemigrations` para criar as **Migrações**.
- `python manage.py migrate` para efetivar as **Migrações** no banco de dados.
- `python manage.py runserver` para executar o servidor de testes do Django.
- Acessar o seu navegador na página `http://localhost:8000` (por padrão).

E pronto... Servidor rodando! 😊

Conclusão do Capítulo

Nesse capítulo vimos como configurar, customizar e estender *templates*, como utilizar os filtros e *tags* do Django, como criar *tags* e filtros customizados e um pouquinho de *Bootstrap*, para deixar as páginas **bonitonas**!

Capítulo V

E AGORA?

Finalmente chegamos ao fim do nosso ebook! Mas, como você sabe, o Django está em constante evolução. Por isso, é bom você se manter atualizado nas novidades lendo, pesquisando e acompanhando o mundo do Django.

Para lhe ajudar, vou colocar aqui algumas referências para você se manter atualizado e também para aprender cada vez mais sobre o Django:

- Site oficial do Django: <https://www.djangoproject.com/>
- Documentação: <https://docs.djangoproject.com/pt-br/2.0/>
- Github do Django: <https://github.com/django/django>

- Twitter do Django: <https://twitter.com/djangoproject>
- Django RSS: <https://www.djangoproject.com/rss/weblog/>
- Lista de e-mails do Django:
<https://groups.google.com/forum/#!forum/django-users>
- Grupo do Facebook:
<https://www.facebook.com/groups/django.brasil/>

E, é claro que não podia faltar, o **Blog da Python Academy**:

<https://pythonacademy.com.br/blog/>

Lá, nós temos conteúdos completos sobre Python, Django, Kivy e muito mais! Conheça e se inscreva na nossa lista de Pythonistas viciados por conteúdo de qualidade!

Capítulo VI

REFERÊNCIA

Django, Documentação do *Django* em Português (largamente utilizada):
<https://docs.djangoproject.com/pt-br/2.0/>.

The Django Book, Read The Docs:
<http://django-book.readthedocs.io/en/latest/>.

How to create custom a custom Django Middleware:
<https://simpleisbetterthancomplex.com/tutorial/2016/07/18/how-to-create-a-custom-django-middleware.html>.

Classy Class Based Views:
<https://ccbv.co.uk/>

DB Browser for SQLite:
<https://sqlitebrowser.org/>

Codementor, Creating Custom Template Tags in Django:
<https://www.codementor.io/hiteshgarg14/creating-custom-template-tags-in-django-application-58wvmqm5f>

Estendendo os Templates, Django Girls:
https://tutorial.djangogirls.org/pt/template_extending/