

django

DESENVOLVIMENTO WEB COM **PYTHON** E **DJANGO**

Seu Guia **definitivo** para dominar o framework web mais robusto do Python e conquistar sua vaga no **mercado de trabalho**





EBOOK

DESENVOLVIMENTO WEB COM PYTHON E DJANGO

SUMÁRIO

| | |
|-------------------------------------|-----------|
| CAPÍTULO 1: INTRODUÇÃO | 4 |
| FLUXO DE UMA REQUISIÇÃO | 6 |
| CAPÍTULO 2: INSTALAÇÃO | 8 |
| HELLO WORLD, DJANGO! | 9 |
| CAPÍTULO 3: CAMADA MODEL | 15 |
| ONDE ESTAMOS... | 15 |
| CAMADA MODEL | 16 |
| DB BROWSER FOR SQLITE | 22 |
| API DE ACESSO A DADOS | 23 |
| CAPÍTULO 4: CAMADA VIEW | 27 |
| ONDE ESTAMOS... | 27 |
| CAMADA VIEW | 28 |
| FUNÇÕES vs CLASS BASED VIEWS | 30 |
| CLASS BASED VIEWS | 31 |
| FUNÇÕES (FUNCTION BASED VIEWS) | 33 |
| DEBUGANDO UMA REQUISIÇÃO NO PYCHARM | 34 |
| AS PRINCIPAIS CLASS BASED VIEWS | 36 |
| FORMS NO DJANGO | 42 |
| MIDDLEWARES | 48 |
| CAPÍTULO 5: CAMADA TEMPLATE | 53 |
| ONDE ESTAMOS... | 54 |
| DEFINIÇÃO DE TEMPLATE | 55 |
| CONFIGURAÇÃO | 57 |
| DJANGO TEMPLATE LANGUAGE (DTL) | 57 |
| CONSTRUINDO A BASE DO TEMPLATE | 58 |
| TAGS E FILTROS CUSTOMIZADOS | 72 |
| FILTROS DO DJANGO | 79 |

CAPÍTULO 1

INTRODUÇÃO

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

E antes de mergulharmos no Django, vamos entender primeiro sobre o **Desenvolvimento Web**! No nicho de **Desenvolvimento Web**, o Pythonista tem como objetivo o desenvolvimento de páginas Web, plataformas ou qualquer outra aplicação que seja executada em um navegador - como o Google Chrome e o Firefox - com conexão à internet.

Dentro do **Desenvolvimento Web** existem 2 áreas principais: o **Frontend** e o **Backend**. O Diagrama abaixo exemplifica como os dois interagem e em seguida explicamos cada um deles:

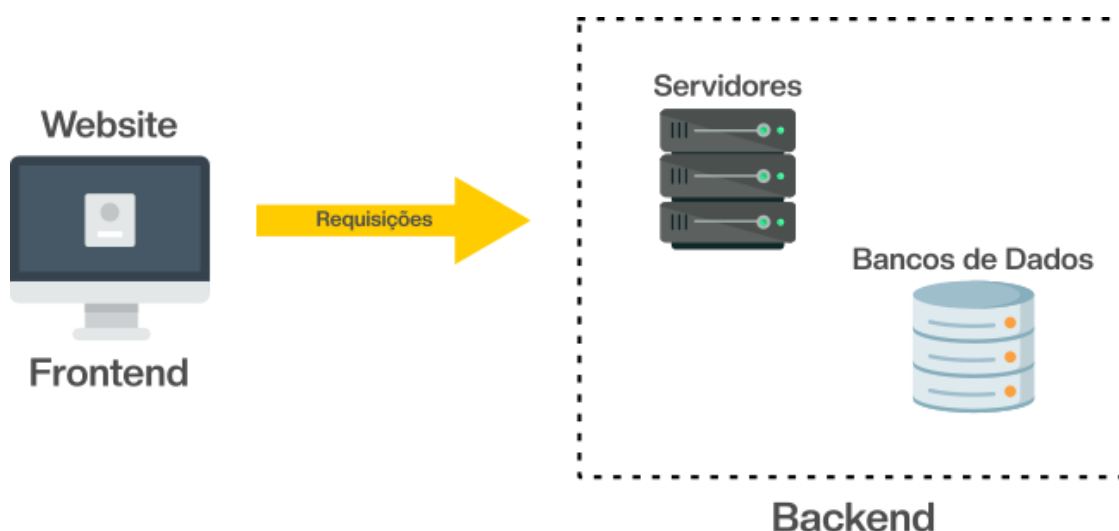


Diagrama Frontend x Backend

- O **Frontend** é constituído por tudo aquilo que o Usuário final vê e interage e é composto, basicamente, das páginas web que esse Usuário terá acesso. Algumas tecnologias geralmente utilizadas são: React, Angular, HTML, CSS e Javascript.
- Já o **Backend**, também chamado de Servidor, tem as seguintes responsabilidades: processa as requisições enviadas pelo Frontend, mantém as Regras de Negócio do sistema e gerencia o acesso ao Banco de Dados. **É aqui que o Python está presente!**

Existem bibliotecas que auxiliam (e muito) no desenvolvimento de aplicações web: os chamados **frameworks**! Um **framework** é um conjunto de ferramentas, técnicas e convenções para facilitar o desenvolvimento de algum tipo específico de aplicação. Sendo assim, temos diversos tipos de frameworks diferentes, por exemplo:

- Temos **Frameworks Frontend**, que facilitam a criação de aplicações web que são executadas nos navegadores dos usuários de um sistema. Exemplos: React, Angular, Vue.js.
- Temos **Frameworks Mobile**, que facilitam o desenvolvimento de aplicações mobile. Exemplo: Ionic e React Native.
- Temos **Frameworks Web**, que facilitam o desenvolvimento de aplicações web, abstraindo detalhes de baixo nível (como protocolos de rede, acesso à banco de dados, tratamento de requisições HTTP). Exemplos: Ruby on Rails (Ruby), Spring (Java), Express (Node.js).

E agora, voltando para o nosso querido Django... 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**: as **regras de negócio da sua aplicação**!

Ele foi desenvolvido com uma preocupação extra em **segurança**, evitando os mais comuns tipos de ataques web, 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.

É utilizado por grandes empresas ao redor do mundo:



E, para que você possa dominar esse framework como um **todo**, utilizaremos uma abordagem **bottom-up** (de baixo para cima), isto é:

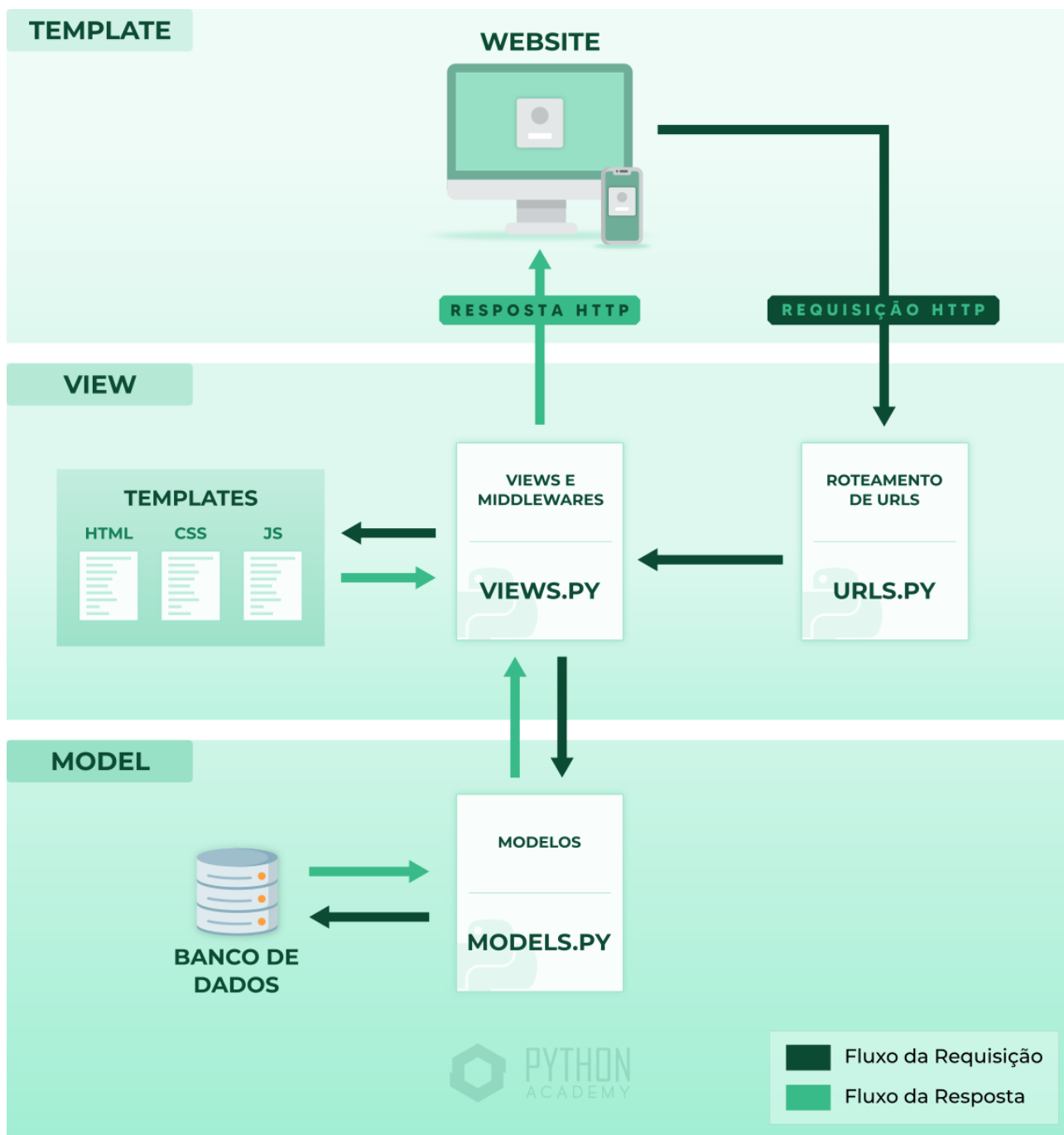
1. Primeiro, vamos começar do começo, **instalando o Django**!
2. Depois abordaremos a **Camada de Modelos**, que é onde fica centralizado o acesso ao Banco de Dados e a modelagem das entidade da nossa aplicação - que chamamos de *Modelos*.
3. Em seguida veremos a **Camada de Views**, que é onde implementamos as regras de negócio da nossa aplicação.
4. Por fim, veremos a **Camada de Templates**, que é a parte do framework responsável por renderizar páginas web, onde utilizaremos HTML, CSS, Javascript e a biblioteca de componentes Bootstrap.

Como disse anteriormente, o Django é um *framework* para construção de aplicações web em Python. Além disso, ele é estruturado em camadas, sendo chamado de um Framework **MTV** - isto é: **Model-Template-View** - que são exatamente as camadas que veremos a seguir.

E para entender como o Django funciona, vamos fazer um Raio-X de uma Requisição, desde o navegador do usuário até o servidor que vai processá-la.

FLUXO DE UMA REQUISIÇÃO

Para ajudar melhor, vamos analisar o fluxo de uma requisição saindo do navegador do usuário, passando para o servidor onde o Django está sendo executado, retornando novamente ao navegador.



O Django é dividido em três camadas: a **Camada de Modelos**, a **Camada de Views** e a **Camada de Templates**. Veremos cada uma nos Capítulos seguintes. Mas agora vamos dar os primeiros passos com o Django, começando pela sua **instalação**! Então ajeita sua cadeira, prepara o café e **vamos nessa**!

CAPÍTULO 2

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:

```
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:

```
import django
print(django.get_version())
```

A saída deve ser a versão do Django que acabou de ser instalada.

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:

```
Available subcommands:
```

```
[django]
  check
  compilemessages
  createcachetable
  dbshell
  diffsettings
  dumpdata
  flush
  inspectdb
  loaddata
  makemessages
  makemigrations
  migrate
  runserver
  sendtestemail
  shell
  showmigrations
  sqlflush
  sqlmigrate
  sqlsequencereset
  squashmigrations
  startapp
  startproject
  test
  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:

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

Criando a seguinte estrutura de diretórios:

```
/helloworld
- __init__.py
- asgi.py
- settings.py
- urls.py
- wsgi.py
- manage.py
```

Explicando cada arquivo:

- `helloworld/asgi.py`: Aqui configuramos a interface entre o servidor de aplicação e nossa aplicação Django, com capacidades assíncronas (não se preocupem com isso por enquanto), através do padrão ASGI.
- `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, através do padrão WSGI
- `manage.py`: Arquivo gerado automaticamente pelo Django que expõe comandos importantes para manutenção da nossa aplicação.

Para testar, vá para a pasta raiz 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:



django

View [release notes](#) for Django 4.0



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



Django Documentation

Topics, references, & how-to's



Tutorial: A Polling App

Get started with Django



Django Community

Connect, get help, or contribute

Se ela aparecer, nossa configuração está **correta** e estamos prontos para começarmos a desenvolver nossa aplicação!

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.

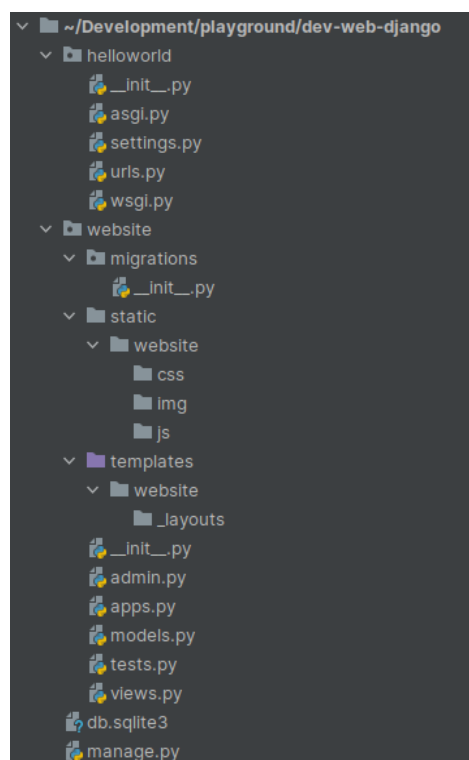
Na raiz do projeto, execute:

```
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** (isto é feito por questões de convenção do próprio *framework*). Também dentro de **static**, crie: uma pasta **css**, uma pasta **img** e uma pasta **js**.

Assim, sua estrutura de diretórios deve estar similar a:



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!

Para que o Django enxergue esse app que acabamos de criar, é necessário adicioná-lo à lista de **apps** instalados do Django. 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:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

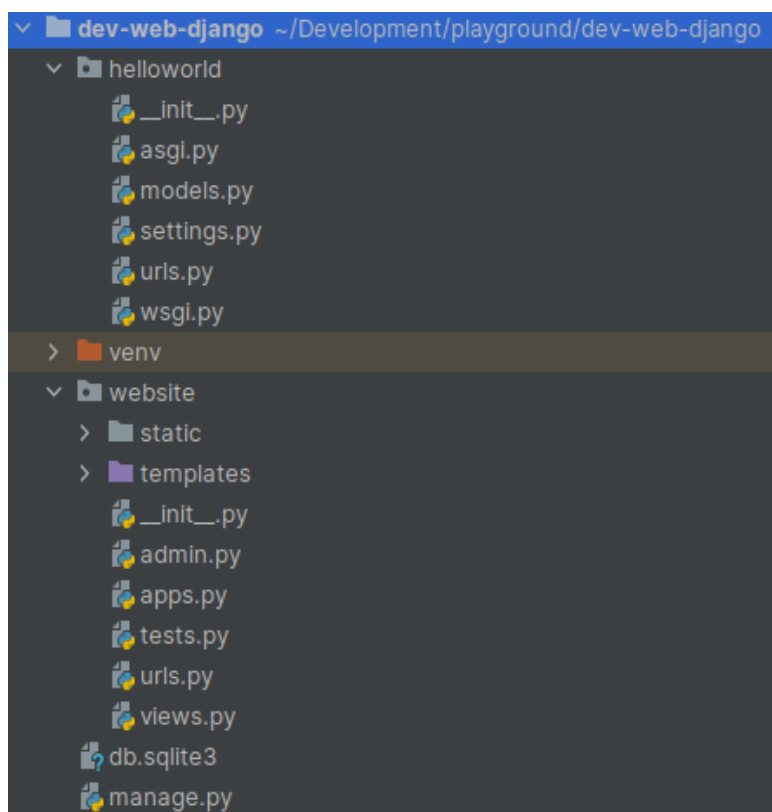
E adicione **website** e **helloworld**, ficando assim:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'helloworld',  
    'website'  
]
```

Agora, vamos fazer algumas alterações na estrutura do projeto para organizar e centralizar algumas configurações. Essa é uma configuração pessoal que eu sempre faço nos projetos que desenvolvo e que facilita a vida quando desenvolvemos utilizando Django.

1. 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 para projetos menores e medianos temos apenas um arquivo **models.py** para o projeto todo. A separação do arquivo de modelos geralmente só ocorre em projetos grandes).
2. 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**.
3. Crie um arquivo de rotas **urls.py** em **/website**. Vamos alterá-lo posteriormente.

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



CONCLUSÃO DO CAPÍTULO

Neste 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!

CAPÍTULO 3

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 e que serão traduzidas em **Tabelas** do nosso Banco de Dados.

Também é nela que reside a lógica de acesso aos dados da nossa aplicação. Neste capítulo, você verá como o Django facilita nossa vida ao manipular os dados do nosso sistema através da poderosa API de Acesso a Dados.

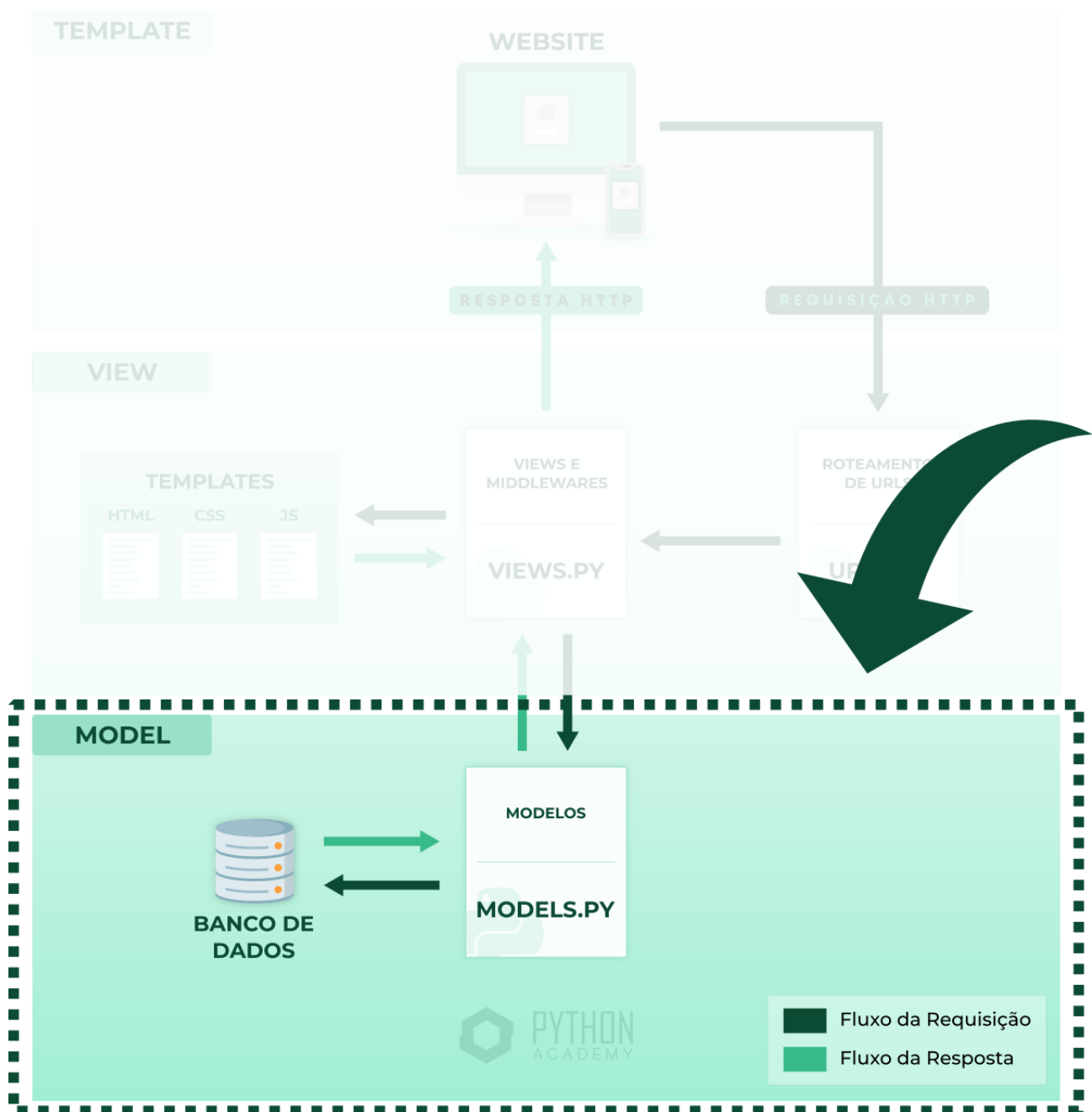
Mas primeiro, vamos nos situar!

ONDE ESTAMOS...

No primeiro capítulo, tratamos de conceitos introdutórios do *framework*, uma visão geral da sua arquitetura

Já no segundo capítulo, fizemos a instalação do Django e a criação do famoso **Hello World** em *Django*.

Agora, vamos tratar da primeira camada do Django, a **Camada Model**!



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** - também chamado de **entidade do sistema** - é a descrição do dado que será gerenciado pela sua aplicação.

Ele contém os campos e comportamentos desses dados. No fim, cada modelo vai ser transformado em uma tabela no banco de dados: processo que é feito pelo próprio Django, portanto não se preocupe!

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, o Django gera **automaticamente** uma API (*Application Programming Interface*) de **Acesso a Dados**. Essa API foi desenhada para facilitar e muito nossa vida quando formos gerenciar (adicionar, excluir e atualizar) os dados da nossa aplicação.

Para entendermos melhor, vamos modelar a principal entidade do sistema que vamos desenvolver: a entidade **Funcionário**!

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

Agora, é necessário traduzir 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 anteriormente (herdar da classe `Model` e mapear os atributos da entidade através de campos), podemos descrever nosso modelo da seguinte forma:

```
from django.db import models

class Funcionario(models.Model):
    nome = models.CharField(
        max_length=255,
        null=False,
        blank=False
    )

    sobrenome = models.CharField(
        max_length=255,
        null=False,
        blank=False
    )

    cpf = models.CharField(
        max_length=14,
        null=False,
        blank=False
    )

    tempo_de_servico = models.IntegerField(
        default=0,
        null=False,
        blank=False
    )

    remuneracao = models.DecimalField(
        max_digits=8,
        decimal_places=2,
        null=False,
        blank=False
    )

    objetos = models.Manager()
```

E agora vamos à explicação deste 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 definir 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 em seguida!
- **Observação:** não precisamos configurar o identificador `id` - ele é herdado automaticamente ao herdar de `django.db.models.Model`!

Toda vez que alteramos os modelos da nossa aplicação Django é necessário gerar uma **Migração** que vai atualizar as tabelas do nosso banco de dados.

Nós fazemos isso através de dois comandos muito importantes que o Django traz para nós através do script `manage.py`: o comando `makemigrations` e o comando `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 de tabelas do 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 (na maioria das vezes) **automáticas**, mas cabe a você saber a hora de fazê-las, executá-las e resolver os problemas comuns que possam vir a acontecer.*

Portanto, toda vez que você alterar um arquivo de modelo, não se esqueça de executar `python manage.py makemigrations`!

Ao executar esse comando, devemos ter a seguinte saída:

```
python manage.py makemigrations

Migrations for 'helloworld':
  helloworld\migrations\0001_initial.py
  - Create model Funcionario
```

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

Agora, perceba que foi criado um diretório chamado `migrations` dentro da pasta `helloworld`.

Nele, você pode ver um arquivo chamado `0001_initial.py`: ele contém a `Migration` que possibilita a criação do `model Funcionario` no banco de dados!

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 **realmente** as alterações no banco de dados.

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

1. Que a configuração da interface com o banco de dados esteja descrita no arquivo `settings.py`
2. Que os Modelos e Migrações 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:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Por padrão, o Django utiliza um banco de dados leve 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**, para realmente alterar a estrutura do Banco de Dados!

Para isso, vamos para a raiz do projeto (onde está o script **manage.py**) e executamos: **python manage.py migrate**. A saída deve ser:

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, helloworld, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add...
  Applying contenttypes.0002_remove_content_ty...
  Applying auth.0002_alter_permission_name_max...
  Applying auth.0003_alter_user_email_max_leng...
  Applying auth.0004_alter_user_username_opts...
  Applying auth.0005_alter_user_last_login_nul...
  Applying auth.0006_require_contenttypes_0002...
  Applying auth.0007_alter_validators_add_erro...
  Applying auth.0008_alter_user_username_max_l...
  Applying auth.0009_alter_user_last_name_max...
  Applying helloworld.0001_initial... OK
  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*. Por isso que ao executar o comando **migrate** o Django aplicou tudo que estava aguardando ser aplicado. *Sacou?!*



Com a execução do comando **migrate**, o Django irá executar diversos comandos SQL para criar a estrutura necessária para execução da nossa aplicação. Uma delas é a tabela referente ao nosso modelo **Funcionário**, similar à:

```
CREATE TABLE helloworld_funcionario (  
    "id" serial NOT NULL PRIMARY KEY,  
    "nome" varchar(255) NOT NULL,  
    "sobrenome" varchar(255) NOT NULL,  
    ...  
);
```

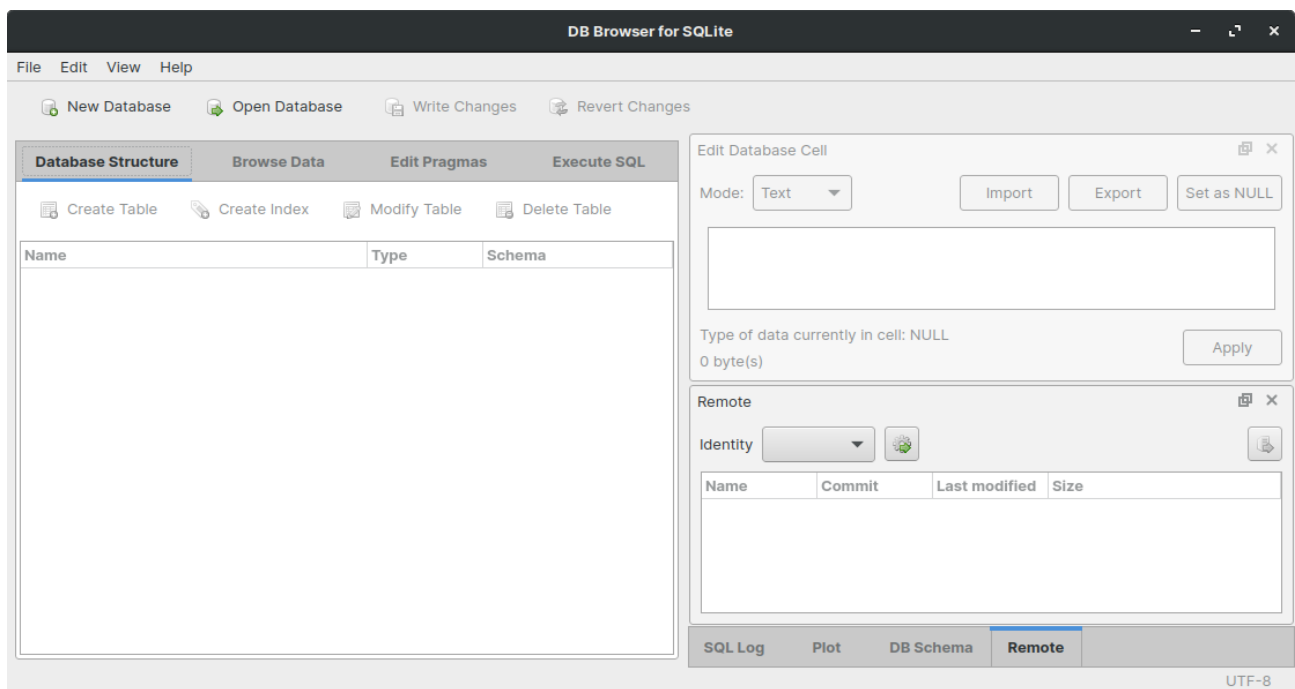
E agora veremos como podemos analisar o banco de dados que o Django criou de forma visual, através da aplicação **DB Browser for SQLite**!

DB BROWSER FOR SQLITE

Apresento-lhes uma ferramenta **MUITO** prática que nos auxilia verificar a estrutura do nosso Banco de Dados: o **DB Browser for SQLite**!

Com ele, podemos ver a estrutura do banco de dados, alterar dados em tempo real, fazer *queries* (consultas), 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*. Você deve ter 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 raiz do projeto).

Ao importar o banco de dados, 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...

Voilà! 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 e já instalada no Banco de Dados, vamos agora ver a API de acesso à dados provida pelo Django que vai facilitar **muito** a nossa vida!

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

```
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):

```
from helloworld.models import Funcionario

funcionario = Funcionario(
    nome='Marcos',
    sobrenome='da Silva',
    cpf='015.458.895-50',
    tempo_de_servico=5,
    remuneracao=10500.00
)

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!

É importante ressaltar que antes da conclusão com sucesso do método `.save()`, o registro não possui um **identificador único** (também chamado de **ID** ou **PK** - **Primary Key**). Após a chamada ao método `.save()` podemos visualizar o ID do registro da seguinte forma:

```
print(funcionario.id)

# Saída deve ser: 1
```

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:

```
funcionarios = Funcionario.objetos.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:

```
funcionarios = Funcionario.objetos
    .exclude(nome="Marcos")
    .filter(tempo_de_servico__gt=3)
    .filter(remuneracao__lt=5000.00)
    .all()
```

O método **exclude()** retira linhas da pesquisa (no nosso caso, vai excluir os registros que contenham “Marcos” no nome) e **filter()** filtra a busca, de acordo com os filtros que passamos!

No exemplo, para filtrar por **maior que**, adicionamos a string **__gt** (**gt** do inglês **greater than**. Em português “maior que”) e **__lt** (**lt** do inglês **less than**. Em português, “menor que”) aos campos.

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**:

```
# Primeiro, encontramos o Funcionário que desejamos deletar
funcionario = Funcionario.objetos.get(id=1)

# Agora, o deletamos!
funcionario.delete()
```

Aqui temos um método novo, o **.get()**! Com ele, podemos passar o **identificador único** do registro que queremos encontrar. Em seguida, podemos chamar o método **.delete()** para deletar um registro do banco de dados!

Simples, né?!

A atualização de campos 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 no banco de dados da seguinte forma:

```
# Primeiro, buscamos o funcionário desejado
funcionario = Funcionario.objetos.get(id=13)

# Alteramos seu sobrenome
funcionario.sobrenome = f"{funcionario.sobrenome} Albuquerque"

# Salvamos as alterações
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 adicionamos lógica de negócio à nossa aplicação Django!

CAPÍTULO 3

CAMADA VIEW

Neste capítulo vamos abordar a Camada *View* do Django, que é onde descrevemos as lógicas de negócio da nossa aplicação!

É nesta camada que vamos desenvolver os métodos que irão:

- **Processar** as Requisições HTTP que chegarem à nossa aplicação;
- **Formular** Respostas HTTP; e
- **Enviá-las** de volta ao usuário.

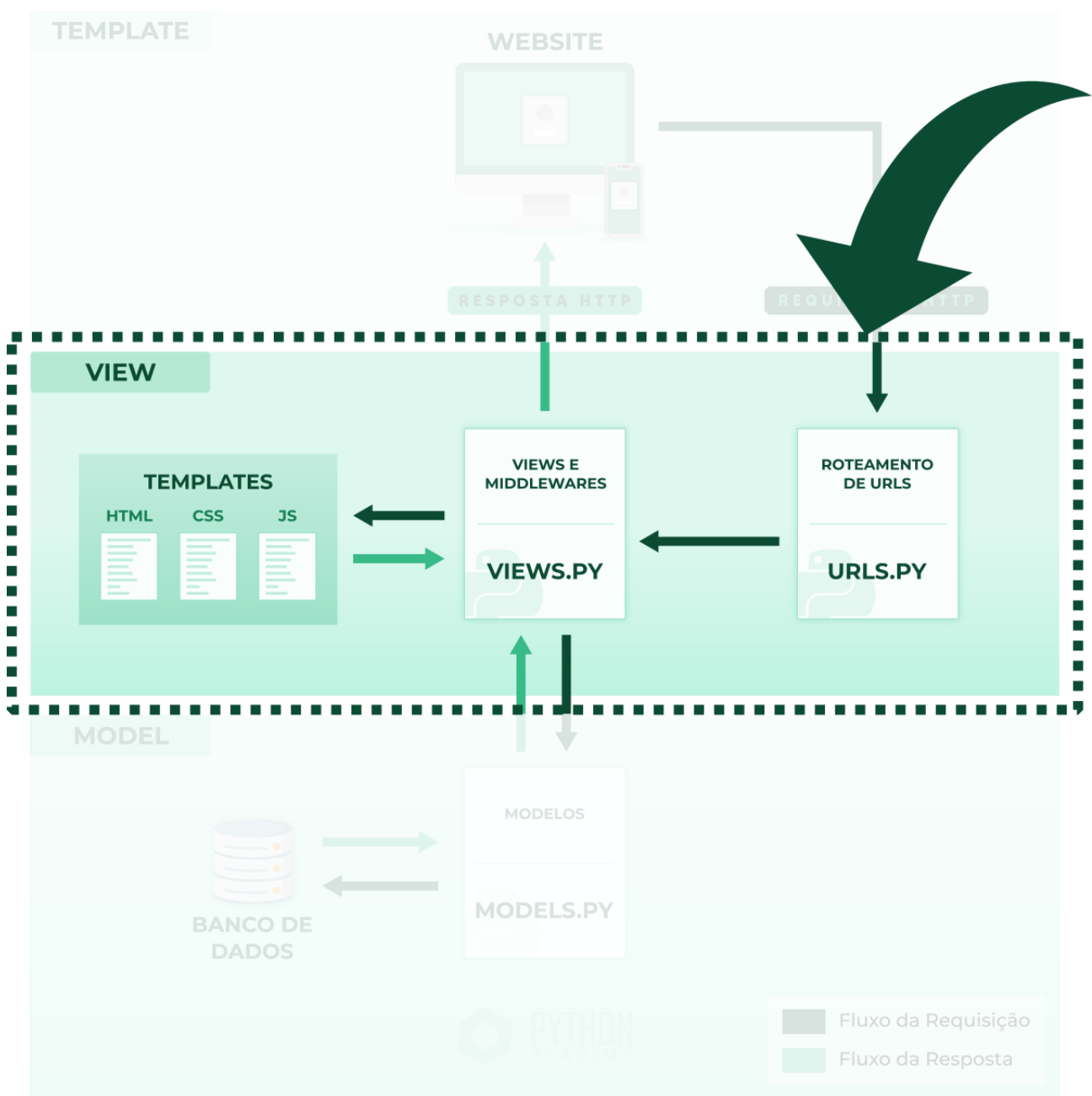
Vamos aprender o conceito das poderosas *Views* do Django, aprender a diferença entre **Function Based Views (FBV)** e **Class Based Views (CBV)**, como utilizar os **Forms** do Django, aprender o que são **Middlewares**, como desenvolvê-los e muito mais.

Então vamos nessa, que esse capítulo está **repleto de código e muito conteúdo!**

Mas antes, você já sabe: vamos nos situar para saber onde estamos, dentro do ciclo de vida de uma Requisição HTTP dentro do Framework Django.

ONDE ESTAMOS...

Primeiramente, vamos nos situar:



CAMADA VIEW

A principal responsabilidade desta camada é a de processar as **requisições** vindas dos usuários, formar uma **resposta** e enviá-la de volta ao usuário. E é nesta camada que residem as **lógicas de negócio** da nossa aplicação.

Essa camada deve: receber, processar e responder!

Na etapa de **recepção** das Requisições, um dos primeiros passos é determinar qual trecho de código a processará, através do que chamamos de **roteamento de URLs!**

A partir da URL que o usuário quiser acessar (**/funcionarios**, por exemplo), o Django irá rotear a Requisição para quem irá tratá-la. Mas primeiro, o Django precisa ser informado sobre **qual código** processa **qual rota**. 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. Por isso criamos o arquivo **urls.py** dentro da pasta **/website**, lá nos capítulos iniciais deste Ebook. 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 processá-la.

Traduzindo em código, fazemos isso da seguinte, alterando o arquivo, o arquivo **helloworld/urls.py**:

```
from django.urls.conf import include
from django.contrib import admin
from django.urls import path

urlpatterns = [
    # Inclui as URLs do app website
    path('', include('website.urls', namespace='website')),

    # Interface administrativa
    path('admin/', admin.site.urls),
]
```

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. Se não houver o casamento de URLs entre o que está configurado nas rotas do Django e o que o usuário quer acessar, um erro **HTTP 404 NOT FOUND** será retornado ao usuário, significando que a página - ou rota - não foi encontrada.

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 tentar acessar a URL raiz da nossa aplicação /, o Django chame a função `index()` para processar tal requisição.

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

```
# Importamos a função index() definida no arquivo views.py
from . import views

app_name = 'website'

# urlpatterns contém a lista de roteamentos de URLs
urlpatterns = [
    # GET /
    path('', views.index, name='index'),
]
```

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](#)).

A função auxiliar `path()` tem a seguinte assinatura:

```
path(rota, view, kwargs=None, name='')
```

Destrinchando cada parâmetro:

- **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.
- **name**: 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 raiz `'/'` com `'website:index'` (`app_site = website` e a rota raiz = `index`). Veja mais sobre [padrões de formato de URL](#).

FUNÇÕES vs CLASS BASED VIEWS

Com as URLs corretamente configuradas, o Django irá rotear Requisições para onde você definiu. No caso acima, sua requisição irá ser processada pela função `views.index()`.

Podemos tratar as requisições de duas formas: através de Views desenvolvidas através de **funções** (*Function Based Views*) ou Views desenvolvidas através de **classes** (*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, geralmente uma **Resposta HTTP**.

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

CLASS BASED VIEWS

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

Para entender as diferenças das Funções e das *Class Based Views*, vamos fazer um exemplo. Suponha você quer criar uma página com a **listagem de todos os funcionários**. Utilizando **funções**, você poderia chegar a esse objetivo da seguinte forma:

```
from django.shortcuts import render
from helloworld.models import Funcionario

def lista_funcionarios(request):
    # Primeiro, buscamos os funcionarios
    funcionarios = Funcionario.objetos.all()

    # Incluimos no contexto
    contexto = {'funcionarios': funcionarios}

    # Retornamos o template para listar os funcionários
    return render(request, "templates/funcionarios.html", contexto)
```

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 na página web que será retornada ao usuário.

- 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:

```
from django.views.generic import ListView
from helloworld.models import Funcionario

class ListaFuncionarios(ListView):
    template_name = "templates/funcionarios.html"
    model = Funcionario
    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 facilitam o desenvolvimento de Views para os casos mais comuns (como 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 acesso. Dessa forma, podemos - por exemplo - criar uma tabela no nosso *template* com os dados de todos os funcionários, assim:

```
<table>
<tbody>
{% for funcionario in funcionarios %}
<tr>
<td>{{ funcionario.nome }}</td>
<td>{{ funcionario.sobrenome }}</td>
<td>{{ funcionario.remuneracao }}</td>
<td>{{ funcionario.tempo_de_servico }}</td>
</tr>
{% endfor %}
</tbody>
</table>
```

*Não se preocupe com a sintaxe do código acima! Vamos falar mais sobre **templates** no próximo capítulo!*

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

- **CreateView**: Para criar de objetos (É o **C**reate do **CRUD**)
- **DetailView**: Traz os detalhes de um objeto (É o **R**etrieve do **CRUD**)
- **UpdateView**: Para atualização de um objeto (É o **U**ppdate do **CRUD**)
- **DeleteView**: Para deletar objetos (É o **D**eleete 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, por baixo dos panos).

Geralmente ao utilizar funções para tratar Requisições, o primeiro passo é verificar **qual foi o método HTTP utilizado**: foi um **GET**? Foi um **POST**? Um **OPTION**?

A partir dessa informação, processamos a Requisição da maneira desejada. Vamos seguir o exemplo abaixo:

```
def cria_funcionario(request, pk):
    # Verificamos se o método POST
    if request.method == 'POST':
        form = FormulárioDeCriacao(request.POST)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('lista_funcionarios'))

    # Qualquer outro método: GET, OPTION, DELETE, etc...
    else:
        return render(request, "templates/form.html", {'form': form})
```

O fluxo é o seguinte:

- Primeiro, conforme mencionei, verificamos o método HTTP da requisição no atributo **method** do objeto **request**.
- Depois instanciamos um **form** com os dados da requisição (no caso **POST**) com **FormulárioDeCriacao(request.POST)** na **linha 4** (vamos falar mais sobre **Form** mais para frente).

- 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 `funcionários/`. Utilizamos isso para redirecionar o usuário para a view de listagem da aplicação.
- 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 [clcando 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.
- `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`!

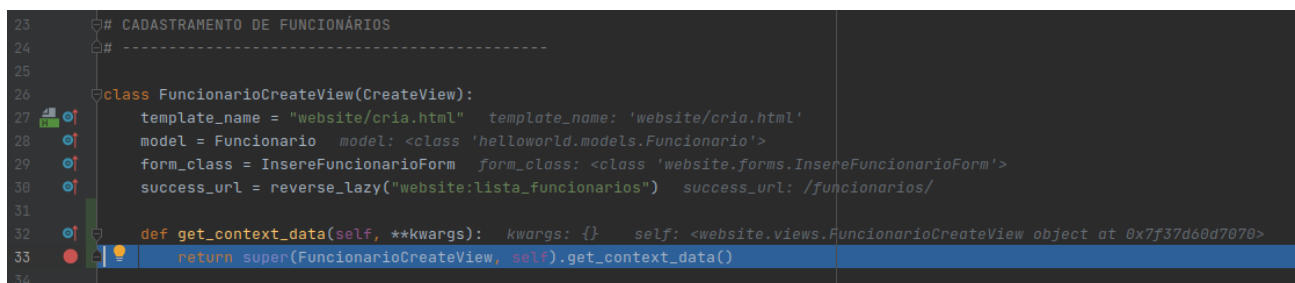
DEBUGANDO UMA REQUISIÇÃO NO PYCHARM

Algumas vezes, é interessante você ver o conjunto de dados que está chegando do usuário para o Django. 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 em que a requisição chega no servidor e analisando seus atributos, verificando se está tudo OK (ou *não*).

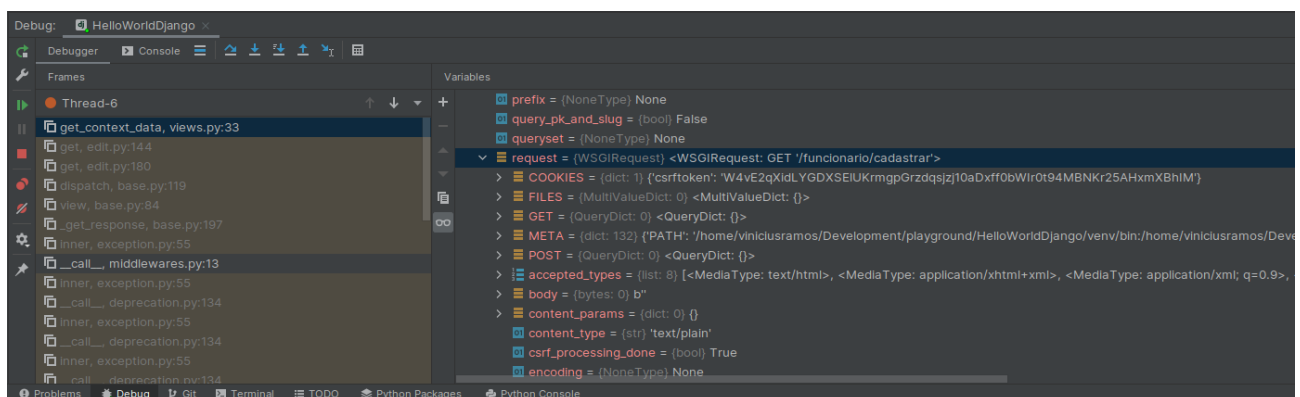
Se você utiliza o **PyCharm**, ou alguma outra IDE com **debugger**, pode fazer os passos que eu vou descrever aqui (*creio que em outra IDE, o processo seja similar*).

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 33, veja o **círculo vermelho** na barra à esquerda, próximo ao contador das linhas):



Com isso, quando uma requisição for enviada do navegador do usuário e que venha a passar nessa linha de código, o *debugger* entrará em ação, mostrando as variáveis naquela linha de código.

Nesse exemplo, quando o *debugger* chegou nessa linha, é possível inspecionar todos os valores atuais na Requisição, contexto, ambiente e mais:



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

Confie em mim, isso ajuda MUITO a detectar erros!

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

AS PRINCIPAIS CLASS BASED VIEWS

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

- Queremos que quando um usuário pedir a página inicial, seja mostrado **apenas uma página** simples, com as opções possíveis.
- Queremos que a 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 dados de um 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`), através do método estático `as_view`, dessa forma:

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    path('', TemplateView.as_view(template_name="index.html")),
]
```

- E a **segunda maneira**, a mais utilizada e mais poderosa, é criar uma Classe herdando da *View* desejada e sobrescrevendo os atributos e métodos nessa subclasse criada, alterando sua lógica para atingir seus objetivos.

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

TemplateView

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

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

Configurando as rotas da seguinte maneira:

```
from django.urls import path  
from helloworld.views import IndexTemplateView  
  
urlpatterns = [  
    path('', IndexTemplateView.as_view(), name='index'),  
]
```

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 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:

```
from django.views.generic.list import ListView  
from helloworld.models import Funcionario  
  
class FuncionarioListView(ListView):  
    template_name = "website/lista.html"  
    model = Funcionario  
    context_object_name = "funcionarios"
```

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

E configuramos sua rota da seguinte maneira:

```
from django.urls import path  
from helloworld.views import FuncionarioListView
```

```
urlpatterns = [
    path('funcionarios/', FuncionarioListView.as_view(), name='lista_funcionarios')
]
```

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

Dica: É uma boa prática colocar o nome da View no formato: 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 registros** 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 de registros do modelo definido.

No nosso caso:

```
from django.views.generic.edit import UpdateView
from helloworld.models import Funcionario

class FuncionarioUpdateView(UpdateView):
    template_name = 'atualiza.html'
    model = Funcionario
    fields = [
        'nome',
        'sobrenome',
        'cpf',
        'tempo_de_servico',
        'remuneracao'
    ]
```

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**):

```
from django.urls import path
from helloworld.views import FuncionarioUpdateView

urlpatterns = [
    # Utilizando o {id} para buscar o objeto
    path(
        'funcionario/<id>',
        FuncionarioUpdateView.as_view(),
        name='atualiza_funcionario'),

    # Utilizando o {slug} para buscar o objeto
    path(
        'funcionario/<slug>',
        FuncionarioUpdateView.as_view(),
        name='atualiza_funcionario'),
]
```

Mas o que é slug?

Slug é uma forma de gerar URLs mais **legíveis** a partir de dados já existentes, transformando todas as letras para minúsculas e todos os espaços para hífens.

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

- **/funcionario/vinicius**

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

- **/funcionario/175**

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

A documentação deste 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 buscar no banco de dados o registro com aquele **id**.

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

```
from django.views.generic.edit import UpdateView
from helloworld.models import Funcionario

class FuncionarioUpdateView(UpdateView):
    template_name = "atualiza.html"
    model = Funcionario
    fields = '__all__'
    context_object_name = 'funcionario'

    def get_object(self, queryset=None):
        funcionario = None

        # Os campos {pk} e {slug} estão presentes em self.kwargs
        id = self.kwargs.get(self.pk_url_kwarg)
        slug = self.kwargs.get(self.slug_url_kwarg)

        if id is not None:
            # Busca o funcionario apartir do id
            funcionario = Funcionario.objects.filter(id=id).first()

        elif slug is not None:
            # Pega o campo slug do Model
            campo_slug = self.get_slug_field()

            # Busca o funcionario apartir do slug
            funcionario = Funcionario.objects.filter(**{campo_slug: slug}).first()

        # Retorna o objeto encontrado
        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 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:

```
class FuncionarioDeleteView(DeleteView):
    template_name = "website/exclui.html"
    model = Funcionario
    context_object_name = 'funcionario'
    success_url = reverse_lazy(
        "website:lista_funcionarios"
    )
```

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 pelo Django (que é o caso aqui).

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

```
urlpatterns = [
    path(
        'funcionario/excluir/<pk>',
        FuncionarioDeleteView.as_view(),
        name='deleta_funcionario'
    )
]
```

Assim, precisamos apenas fazer um *template* de confirmação da exclusão do funcionário. Podemos fazer este *template* da seguinte forma:

```
<form method="post">
{% csrf_token %}

Você tem certeza que quer excluir o funcionário <b>{{ funcionario.nome }}</b>?

<br><br>

<button type="button">
  <a href="{% url 'lista_funcionarios' %}">Cancelar</a>
</button>
<button>Excluir</button>
</form>
```

Algumas colocações:

- A tag do Django `{% csrf_token %}` é obrigatória 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 deste *template*, pois veremos mais sobre ele no **próximo capítulo**!

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:

```
from django.views.generic import CreateView

class FuncionarioCreateView(CreateView):
    template_name = "website/cria.html"
    model = Funcionario
    form_class = InsereFuncionarioForm
    success_url = reverse_lazy("website:lista_funcionarios")
```

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.

A configuração da rota no arquivo `urls.py` pode ser feita da seguinte forma:

```
from django.urls import path
from helloworld.views import FuncionarioCreateView

urlpatterns = [
    path(
        'funcionario/cadastrar/',
        FuncionarioCreateView.as_view(),
        name='cadastra_funcionario',
    )
]
```

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.

E agora trataremos da forma que o *framework* traz para construção de Formulários em código HTML, os **Forms** do Django!

FORMS NO DJANGO

Podemos utilizar o formulário do Django nas páginas HTML de duas formas. A **primeira**, mostra o formulário inteiro **"cru"**, isto é, sem formatação e sem estilo, conforme o Django nos entrega.

Podemos utilizá-lo no nosso *template* da seguinte forma:

```
<form method="post">
  {% csrf_token %}

  {{ form }}

  <button type="submit">Cadastrar</button>
</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:

```
<form method="post">
  {% csrf_token %}

  <label for="{{ form.nome.id_for_label }}">Nome</label>
  {{ form.nome }}

  <label for="{{ form.sobrenome.id_for_label }}">Sobrenome</label>
  {{ form.sobrenome }}

  <label for="{{ form.cpf.id_for_label }}">CPF</label>
  {{ form.cpf }}

  <label for="{{ form.tempo_de_servico.id_for_label }}">Tempo de Serviço</label>
  {{ form.tempo_de_servico }}

  <label for="{{ form.remuneracao.id_for_label }}">Remuneração</label>
  {{ form.remuneracao }}

  <button type="submit">Cadastrar</button>
</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.

Esse *template* será renderizado em uma página HTML no navegador do usuário do nosso sistema. Após ser apresentado, o formulário será preenchido e então submetido de volta ao nosso servidor. E agora vem a parte mais complexa quando desenvolvermos Formulários utilizando o Django: **tratamento de dados!**

O tratamento dos dados enviados no 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:

```
<form action="/insere-funcionario/" method="post">
  <label for="nome">Nome: </label>
  <input id="nome" type="text" name="nome" value="">
  <input type="submit" value="Enviar">
</form>
```

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 e no caso de falha na validação, que possamos mostrar isso ao usuário.

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

Nela, nós descrevemos os campos que estarão disponíveis no formulário HTML. Por exemplo, podemos descrever o formulário acima da seguinte forma:

```
from django import forms

class InsereFuncionarioForm(forms.Form):
    nome = forms.CharField(
        label='Nome do Funcionário',
        max_length=100
    )
```

Neste formulário:

- Utilizamos a classe **forms.CharField** para descrever um campo de texto.
- O parâmetro **label** descreve um rótulo para esse campo.

- `max_length` descreve 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:

```
from django import forms

class InsereFuncionarioForm(forms.Form)

    nome = forms.CharField(
        required=True,
        max_length=255
    )

    sobrenome = forms.CharField(
        required=True,
        max_length=255
    )

    cpf = forms.CharField(
        required=True,
        max_length=14
    )

    tempo_de_servico = forms.IntegerField(
        required=True
    )

    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 o *Model* que servirá como base do Formulário; os campos que queremos a partir do atributo **fields**, e, através do campo **exclude**, os campos que não queremos.

Para fazer essa configuração, utilizamos uma classe interna, chamada **Meta**. Através dela, é possível configurar uma série de comportamentos do **ModelForm**, como o modelo que será utilizado (atributo **model**), os campos (através do atributo **fields**), a forma de ordenação (através do atributo **ordering**) e mais ([veja mais sobre Meta options](#)).

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

```
from django import forms

class InsereFuncionarioForm(forms.ModelForm):
    class Meta:
        # Modelo base
        model = Funcionario

        # Campos que estarão no form
        fields = [
            'nome',
            'sobrenome',
            'cpf',
            'remuneracao'
        ]

        # Campos que não estarão no form
        exclude = [
            'tempo_de_servico'
        ]
```

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 avulsamente.

Ficou confuso?

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

```
from django import forms

class InsereFuncionarioForm(forms.ModelForm)
    chefe = forms.BooleanField(
        label='Este Funcionário exerce função de Chefia?',
        required=True,
    )

    biografia = forms.CharField(
        label='Biografia',
        required=False,
        widget=forms.TextArea
    )

    class Meta:
        # Modelo base
        model = Funcionario

        # Campos que estarão no form
        fields = [
            'nome',
            'sobrenome',
            'cpf',
            'remuneracao'
        ]

        # Campos que não estarão no form
        exclude = [
            'tempo_de_servico'
        ]
```

Isso vai gerar um formulário com:

- Todos os campos contidos em `fields`
- Sem 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, nós modificamos o campo, através da configuração `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 em nossos `templates`.

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

MIDDLEWARES

Middlewares são trechos de códigos que podem ser executados antes ou depois do processamento de Requisições/Respostas pelas *Views* da nossa aplicação. É uma forma que nós temos para alterar como o Django processa algum dado de entrada ou de saída.

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

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

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`:

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

Portanto, veremos agora como podemos criar um *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:

```
def middleware_simples(get_response):  
  
    # Código de inicialização do Middleware  
    def middleware(request):  
        # Código a ser executado antes da View e  
        # antes de outros middlewares serem executados  
  
        response = get_response(request)  
  
        # Código a ser executado após a execução  
        # da View que irá processar a requisição  
  
        return response  
  
    return middleware
```

E como Classe:

```
class MiddlewareSimples:  
    def __init__(self, get_response):  
        self.get_response = get_response  
  
    # Código de inicialização do Middleware  
    def __call__(self, request):  
        # Código a ser executado antes da View e  
        # antes de outros middlewares serem executados  
  
        response = self.get_response(request)  
  
        # Código a ser executado após a execução  
        # da View que irá processar a requisição  
  
        return response
```

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

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

O MÉTODO `process_view`

Assinatura do método: `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 um objeto da Classe `HttpRequest`, do próprio Django.
- `func` é a própria *View* que o Django está prestes a 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 do método: `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 navegador do usuário.

- `request` é o objeto `HttpRequest`
- `exception` é a exceção que foi lançada pela *view*.

O MÉTODO `process_template_response`

Assinatura do método: `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 IPs 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:

```
class FiltraIPMiddleware:

    def __init__(self, get_response=None):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)
        return response

    def process_view(request, func, args, kwargs):
        # Lista de IPs autorizados
        ips_autorizados = ['127.0.0.1']

        # IP do usuário
        ip = request.META.get('REMOTE_ADDR')
```

```
# Verifica se o IP do está na Lista de IPs autorizados
if ip not in ips_autorizados:
    # Se usuário não autorizado > HTTP 403 (Não Autorizado)
    return HttpResponseForbidden("IP não autorizado")

# Se for autorizado, não fazemos nada
return None
```

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

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    # Nosso Middleware
    'helloworld.middlewares.FiltroIPMiddleware',
]
```

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

- Coloque **ips_autorizados = ['127.0.0.1']** e tente acessar alguma URL da aplicação: devemos conseguir acessar normalmente nossa aplicação, pois como estamos executando o servidor localmente, nosso IP será exatamente igual à 127.0.0.1 e, portanto, passaremos no teste condicional que desenvolvemos no Middleware.
- Agora 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 mais autorizado a acessar o servidor, mostrando que nossa lógica funcionou corretamente!

CONCLUSÃO DO CAPÍTULO

Neste capítulo vimos vários conceitos importantes: vimos os tipos de *Views* (funções e classes), os principais tipos de CBV (*Class Based Views*), como mapear URLs para as *views* da aplicação através do **URLConf**, como utilizar os poderosos **Forms** do Django, *Middlewares* e muito mais!

No próximo capítulo, veremos a camada da nossa aplicação, que é quem faz a Interface com o Usuário, a **Camada Template**.

CAMADA *TEMPLATE*

O foco deste capítulo será a Camada Template da arquitetura do Django. Neste capítulo vamos aprender a configurar, customizar e estender *templates*. Também veremos como utilizar os filtros e *tags* que o próprio Django nos disponibiliza, assim como criar *tags* e filtros **customizados** e Middlewares, que são peças muito importantes no desenvolvimento de aplicações que utilizam o Django!

Além disso, veremos como customizar o visual de páginas web com o famoso *Bootstrap*, que dará uma identidade visual profissional às páginas da nossa aplicação!

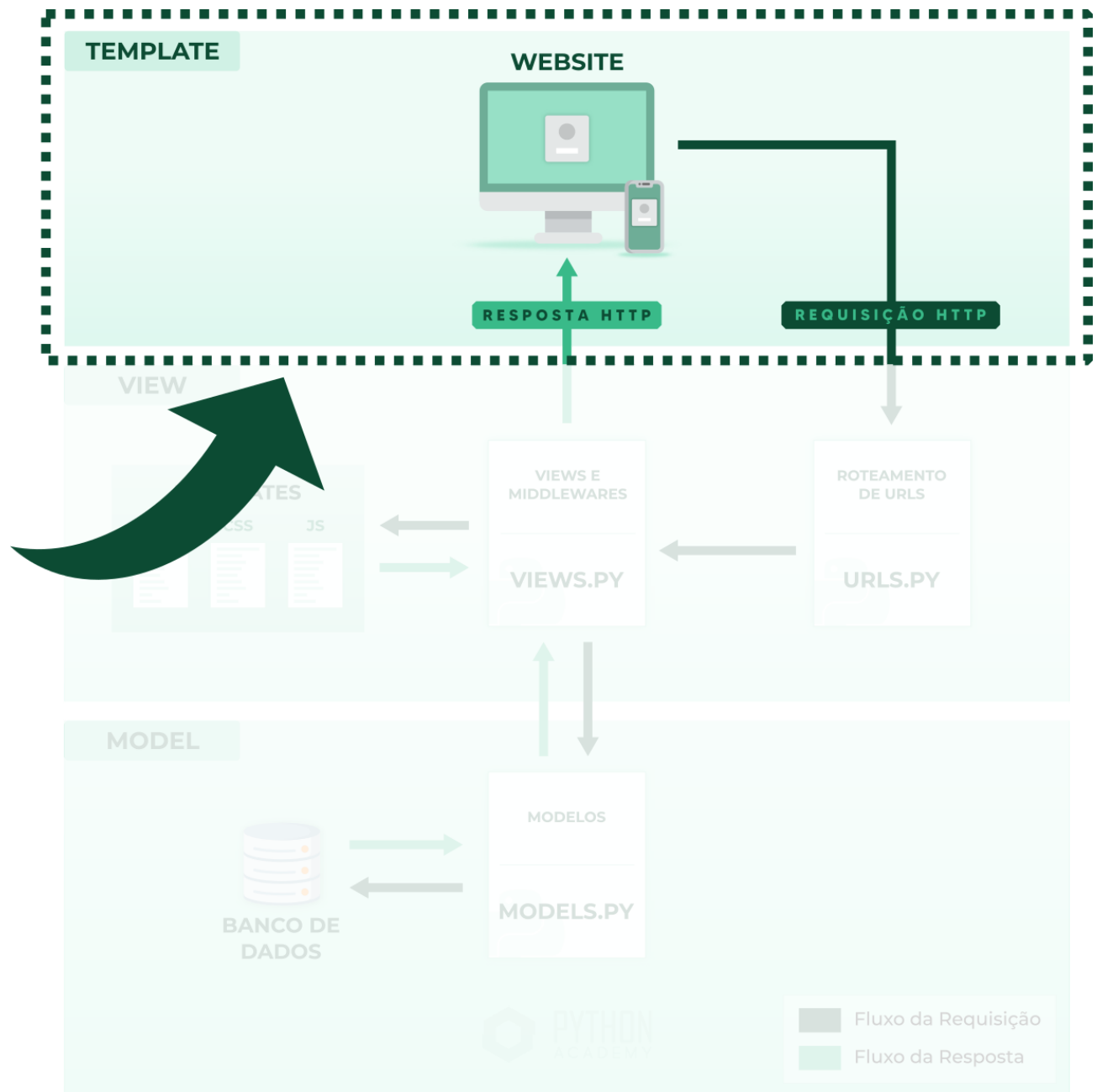
A **Camada Template** tem uma importância muito grande nas aplicações Django, pois é ela quem dá cara ao nosso sistema, isto é, faz a **interface com o usuário**. É nela que se encontra o código Python - responsável por renderizar nossas páginas - e os arquivos HTML, CSS e Javascript - que darão vida à nossa aplicação!

Contudo, vale ressaltar que as aplicações web vem sofrendo uma mudança em sua arquitetura. Antigamente, era muito comum que todo código da aplicação web vivesse apenas no **Backend**, como uma aplicação apenas - com o servidor renderizando as páginas da aplicação. Hoje em dia, é muito comum haver essa separação entre **Backend** e **Frontend**, com duas aplicações distintas: o **Backend** servindo o **Frontend**, através de - geralmente - uma API, enquanto o **Frontend** é desenvolvido como uma aplicação à parte, utilizando frameworks consagrados como React, Angular ou Vue.js, que consumirão dados do **Backend**.

Isso não quer dizer que a camada Template do Django não tem mais utilização! Muitos projetos ainda a utilizam, pois ela é muito poderosa, mas seu uso vem diminuindo.

ONDE ESTAMOS...

Primeiro, vamos relembrar onde estamos no fluxo de Requisição/Resposta da nossa aplicação Django:



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

Antes de mergulhar nessa Camada, vamos começar pelo começo, respondendo à seguinte pergunta: **o que é um *Template***?

DEFINIÇÃO DE TEMPLATE

Basicamente, um *template* é um arquivo **base** que pode ser transformado em outro arquivo (um arquivo HTML, um CSS, um CSV, etc), através do processo de **interpolação de código**.

Um *template* no Django contém:

- **Variáveis** que podem ser substituídas por valores, a partir do seu processamento por uma *Engine* de *Templates* (núcleo ou “motor” de *templates*). Para se usar variáveis em templates, usamos marcadores iniciados com chaves, dessa forma: `{{ variável }}`.
- **Tags** que controlam a lógica de renderização do *template*. Usamos as chaves e o símbolo de porcentagem, dessa forma: `{% tag %}`.
- **Filtros** que adicionam funcionalidades ao *template*. Usamos com o caracter chamado “pipe”, dessa forma: `{{ variável|filtro }}`.

Entenda **interpolação** como o processo de se misturar códigos em linguagens diferentes (por exemplo HTML com código Python) adicionando funcionalidade, e tendo como saída apenas código em uma das duas linguagens.

Não entendeu ainda?

Pense o seguinte: você talvez já saiba que HTML não tem estruturas de repetição como o **for** ou **while** do Python, correto?

E se fosse possível criar um código que misturasse HTML com o **for** do Python para criar uma estrutura de repetição dentro do código HTML, como o código abaixo?!

```
<h1>Teste de interpolação</h1>
```

```
for dado in [1, 2, 3, 4]:  
  <p>{{ dado }}</p>
```

É exatamente isso que a **interpolação** faz (com uma pequena diferença de sintaxe): adiciona funcionalidades de um código a outro!

Agora vamos à um exemplo com a real sintaxe de interpolação utilizando a **Engine de templates do Django**!

```
{# base.html contém o template que usaremos como esqueleto #}
{% extends "base.html" %}

{% block conteudo %}
<h1>{{ section.title }}</h1>

{% for f in funcionarios %}
<h2>
<a href="{% url 'website:funcionario_detalhe' pk=f.id %}">
  {{ funcionario.nome|upper }}
</a>
</h2>
{% endfor %}
{% endblock %}
```

Agora vamos à explicação:

- **Linha 1:** Escrevemos comentário com a tag `{# comentário #}`. Eles serão processados pelo *Engine* e não estarão presentes 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 em nosso *template* utilizando `{{ 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 alguma variável. Nesse exemplo, estamos aplicando o filtro `upper` - que transforma todos os caracteres de uma *string* em maiúsculos - ao 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 e a chamaram de **DTL** - *Django Template Language*! Veremos mais dela neste capítulo!

Para começarmos a utilizar os *templates* do Django, é necessário primeiro **configurar sua utilização**. E é isso que veremos agora!

CONFIGURAÇÃO

O nosso arquivo de configuração `settings.py` contém a seguinte configuração, que define qual Engine (também chamada de Backend) fará o processamento dos templates da nossa aplicação:

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [],  
        'APP_DIRS': True,  
        'OPTIONS': {},  
    },  
]
```

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* utilizado, 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 (DTL)

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?!

Por esse motivo que o template **não deve conter lógica de negócio**, apenas lógica que altere a **apresentação dos dados**!

Agora, nada melhor para aprender sobre a *DTL* do que botando a mão na massa e melhorando as páginas da nossa aplicação. E para deixar as páginas visualmente agradáveis, vamos utilizar o famoso [Bootstrap](#)!

CONSTRUINDO A BASE DO TEMPLATE

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 de páginas HTML. Por exemplo, toda página HTML:

- Deve ter as tags: `<html></html>`, `<head></head>` e `<body></body>`.
- Deve ter os *links* para arquivos estáticos: `<link></link>` e `<script></script>`.

Você pode fazer o *download* dos arquivos necessários para o nosso projeto [aqui \(Bootstrap\)](#) e [aqui \(jQuery\)](#) - que é uma dependência do *Bootstrap*.

Faça isso para todas 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*:

```

<!DOCTYPE html>
<html>
{% load static %}
<head>
<title>
    {% block title %}Gerenciador de Funcionários{% endblock %}
</title>

<!-- Estilos -->
<link rel="shortcut icon" type="image/png" href="{% static 'website/img/favicon.png' %}">
<link rel="stylesheet" href="{% static 'website/css/bootstrap.min.css' %}">
<link rel="stylesheet" href="{% static 'website/css/master.css' %}">

{% block styles %}{% endblock %}
</head>

<body>
<nav class="navbar navbar-expand-lg navbar-light bg-white">
    <a class="navbar-brand" href="{% url 'website:index' %}">
        
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
        data-target="#conteudo-navbar" aria-controls="conteudo-navbar"
        aria-expanded="false" aria-label="Ativar navegação">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="conteudo-navbar">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="{% url 'website:index' %}">
                    Página Inicial
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'website:lista_funcionario' %}">
                    Funcionários
                </a>
            </li>
        </ul>
    </div>
</nav>

{% block conteudo %}{% endblock %}

<script src="{% static 'website/js/jquery.min.js' %}"></script>
<script src="{% static 'website/js/bootstrap.min.js' %}"></script>

{% block scripts %}{% endblock %}

<script src="{% static 'website/js/scripts.js' %}"></script>
</body>
</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 e fazemos isso através do código

`{% load <modulo> %}`. Após carregá-la, utilizamos a *tag* da seguinte maneira: `{%static 'caminho/para/arquivo' %}`, passando como parâmetro a localização relativa à pasta `/static/`.

- Podemos definir quaisquer blocos no nosso *template* com a *tag* `{% block nome_do_bloco %}{% endblock %}`. Fazemos isso para organizar melhor as páginas que irão estender esse *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 - o valor padrão é aplicado.
- Colocamos os arquivos necessários para o funcionamento do *Bootstrap* nesse *template*, isto é: o jQuery, o CSS e o 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:

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

Com isso, podemos fazer:

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

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

<!-- Bloco de conteúdo da nossa página -->
{% block conteudo %}
<div class="container">
  <div class="row">
    <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
      <div class="card">
        <div class="card-body">
          <h5 class="card-title">Cadastrar Funcionário</h5>
          <p class="card-text">
            Cadastre aqui um novo <code>Funcionário</code>.
          </p>
          <a href="{% url 'website:cadastra_funcionario' %}"
            class="btn btn-primary">
            Novo Funcionário
          </a>
        </div>
      </div>
    </div>
    <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
      <div class="card">
        <div class="card-body">
          <h5 class="card-title">Lista de Funcionários</h5>
          <p class="card-text">
            Veja aqui a lista de <code>Funcionários</code> cadastrados.
          </p>
          <a href="{% url 'website:lista_funcionarios' %}"
            class="btn btn-primary">
            Vá para Lista
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

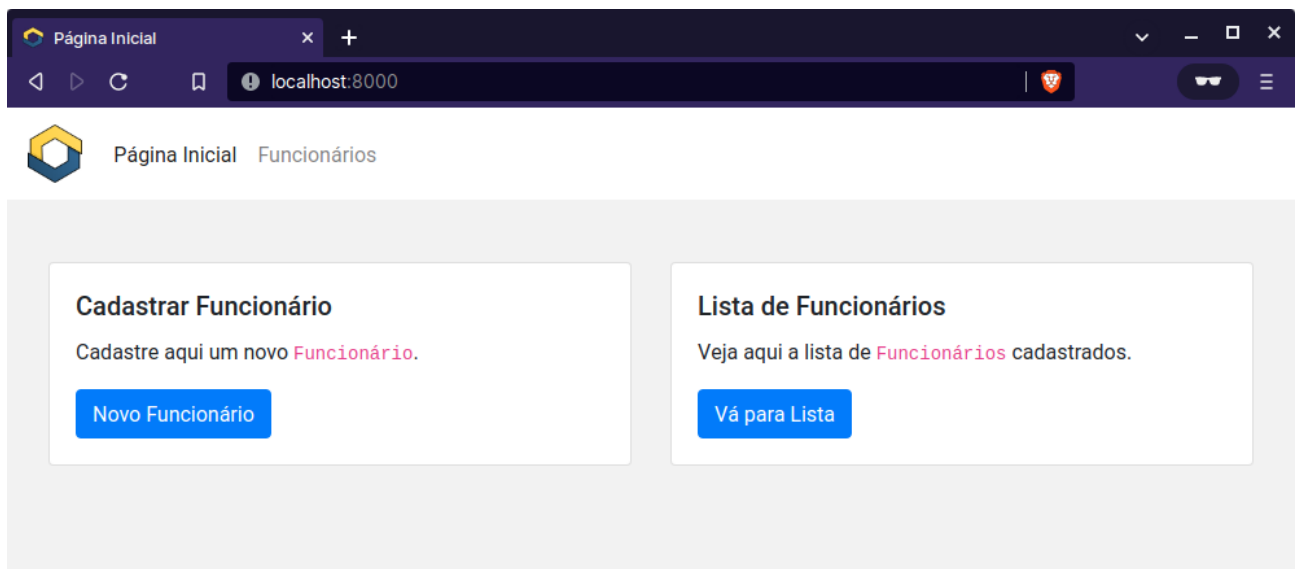
Nesse *template*:

- 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). É muito importante você

se atentar a esse detalhe para não comprometer a experiência do usuário, caso ele acesse em aparelhos diferentes!

- As classes `card*` fazem parte do [componente Card do Bootstrap](#).
- As classes `btn` e `btn-primary` ([documentação](#)) são usadas para dar 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!

TEMPLATE DE CADASTRO DE FUNCIONÁRIOS

Template: `website/cadastre-funcionario.html`

Nesse *template*, mostraremos o formulário para cadastro de novos funcionários ao usuário do sistema.

Lembra que definimos o Form do Django `InsereFuncionarioForm` no capítulo passado?

Vamos utilizá-lo no template que criaremos em seguida, adicionando-o na View `FuncionarioCreateView`. Dessa forma, esta View 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:

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

Depois disso, adicione-a à lista de *apps* instalados, no já conhecido arquivo `helloworld/settings.py`:

```
INSTALLED_APPS = [  
    ...  
    'widget_tweaks'  
]
```

E, no *template* onde formos utilizá-lo, carregamos ela com a *tag* `load`, da seguinte forma: `{% load widget_tweaks %}!`

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

```
{% 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.

Assim, podemos escrever nosso template de cadastro de Funcionários da seguinte forma:

```
{% extends "website/_layouts/base.html" %}  
  
{% load widget_tweaks %}  
  
{% block title %}Cadastro de Funcionários{% endblock %}  
  
{% block conteudo %}  
<div class="container">  
  <div class="row">  
    <div  
      class="col-lg-12 col-md-12 col-sm-12 col-xs-12">  
      <div class="card">  
        <div class="card-body">  
          <h5 class="card-title">Cadastro de Funcionário</h5>  
          <p class="card-text">  
            Complete o formulário abaixo para cadastrar  
            um novo <code>Funcionário</code>.  
          </p>  
          <form method="post">  
            <!-- Não se esqueça dessa tag -->
```

```

{% csrf_token %}

<!-- Nome -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">Nome</span>
  </div>
  {% render_field form.nome class+="form-control" %}
</div>

<!-- Sobrenome -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">Sobrenome</span>
  </div>
  {% render_field form.sobrenome class+="form-control" %}
</div>

<!-- CPF -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">CPF</span>
  </div>
  {% render_field form.cpf class+="form-control" %}
</div>

<!-- Tempo de Serviço -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">
      Tempo de Serviço
    </span>
  </div>
  {% render_field form.tempo_de_servico class+="form-control" %}
</div>

<!-- Remuneração -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">Remuneração</span>
  </div>
  {% render_field form.remuneracao class+="form-control" %}
</div>

  <button class="btn btn-primary">Enviar</button>
</form>
</div>
</div>
</div>
</div>
</div>
{% endblock %}

```

Alguns pontos importante sobre o formulário acima:

- 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 uma classe ao campo, utilizamos o símbolo de adição `+=` no atributo `class: {% render_field form.campo class+='classe' %}`

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

```
class InsereFuncionarioForm(forms.ModelForm):
    nome = forms.CharField(
        max_length=255,
        widget=forms.TextInput(
            attrs={
                'class': "form-control"
            }
        )
    )
    ...
```

Mas essa é uma **péssima ideia** porque bagunça código CSS dentro de código Python. **Não faça isso!**

Nosso formulário deve ficar assim:

The screenshot shows a web browser window with the address bar displaying 'localhost:8000/funcionario/cadastrar'. The page has a navigation bar with 'Página Inicial' and 'Funcionários'. The main content area is titled 'Cadastro de Funcionário' and contains a form with the following fields and values:

| Field | Value |
|------------------|----------------|
| Nome | João |
| Sobrenome | Carlos |
| CPF | 123.456.789-00 |
| Tempo de Serviço | 5 |
| Remuneração | 15000 |

At the bottom right of the form are two buttons: 'Voltar' and 'Enviar'.

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

TEMPLATE DE LISTAGEM DE FUNCIONÁRIOS

Template: `website/lista.html`

Nesta página, nós queremos mostrar o conjunto de Funcionários cadastrados no banco de dados e as ações que o usuário da aplicação pode tomar, que são: atualizar os dados de um Funcionário ou excluí-lo.

Você 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:

```
{% extends "website/_layouts/base.html" %}

{% block title %}Lista de Funcionários{% endblock %}

{% block conteudo %}
<div class="container">
  <div class="row">
    <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
      <div class="card">
        <div class="card-body">
          <h5 class="card-title">Lista de Funcionário</h5>

          {% if funcionarios|length > 0 %}
          <p class="card-text">
            Aqui está a lista de <code>Funcionários</code>
            cadastrados.
          </p>

          <table class="table">
            <thead class="thead-dark">
              <tr>
                <th>ID</th>
                <th>Nome</th>
                <th>Sobrenome</th>
                <th>Tempo de Serviço</th>
                <th>Remuneração</th>
                <th>Ações</th>
              </tr>
            </thead>

            <tbody>
              {% for f in funcionarios %}
              <tr>
                <td>{{ f.id }}</td>
                <td>{{ f.nome }}</td>
                <td>{{ f.sobrenome }}</td>
                <td>{{ f.tempo_de_servico }}</td>
                <td>{{ f.remuneracao }}</td>
                <td>
                  <a href="{% url 'website:atualiza_funcionario' pk=f.id %}"
```

```

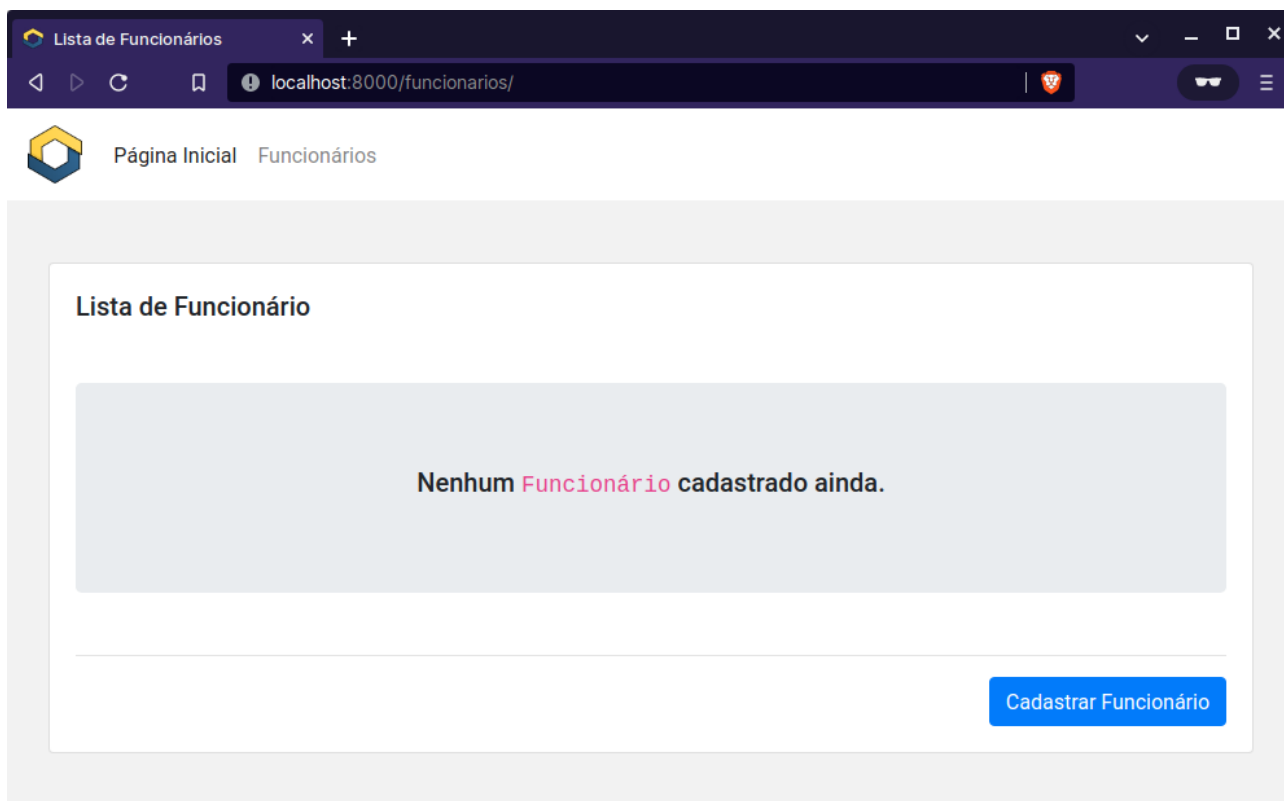
        class="btn btn-info">
        Atualizar
    </a>
    <a href="{% url 'website:deleta_funcionario' pk=f.id %}"
        class="btn btn-outline-danger">
        Excluir
    </a>
</td>
</tr>
</tr>
{% endfor %}
</tbody>
</table>
{% else %}
<div class="text-center mt-5 mb-5 jumbotron">
    <h5>Nenhum <code>Funcionário</code> cadastrado ainda.</h5>
</div>
{% endif %}
<hr />
<div class="text-right">
    <a class="btn btn-primary"
        href="{% url 'website:cadastra_funcionario' %}">
        Cadastrar Funcionário
    </a>
</div>
</div>
</div>
</div>
</div>
</div>
</div>
{% endblock %}

```

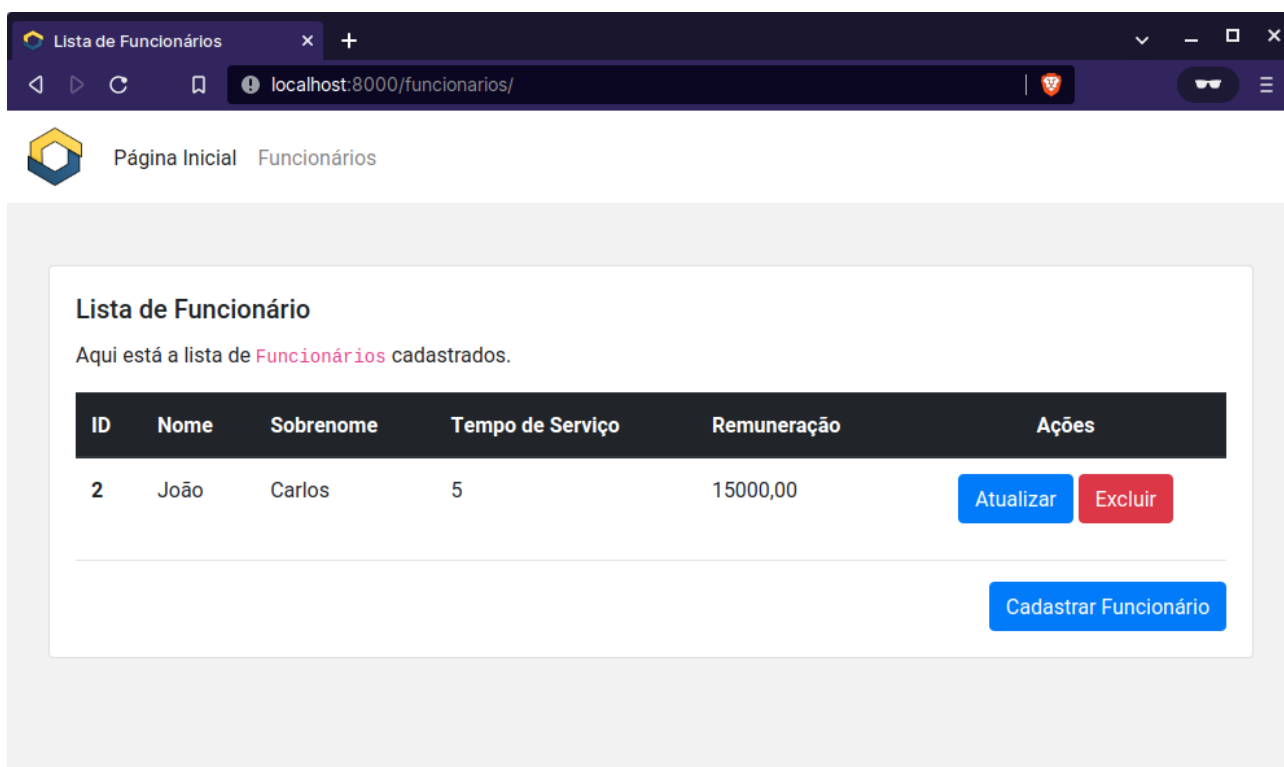
Nesse *template*:

- 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, uma caixa com o texto “*Nenhum Funcionário cadastrado ainda*” será mostrada.
- 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:



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*.

Veja no código abaixo como podemos fazer nosso *template*:

```
{% extends "website/_layouts/base.html" %}

{% load widget_tweaks %}

{% block title %}Atualização de Funcionário{% endblock %}

{% block conteudo %}
<div class="container">
  <div class="row">
    <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
      <div class="card">
        <div class="card-body">
          <h5 class="card-title">
            Atualização de Dados do Funcionário
          </h5>
          <form method="post">
            <!-- Não se esqueça dessa tag -->
            {% csrf_token %}

            <!-- Nome -->
            <div class="input-group mb-3">
              <div class="input-group-prepend">
                <span class="input-group-text">Nome</span>
              </div>
              {% render_field form.nome class+="form-control" %}
            </div>

            <!-- Sobrenome -->
            <div class="input-group mb-3">
              <div class="input-group-prepend">
                <span class="input-group-text">Sobrenome</span>
              </div>
              {% render_field form.sobrenome class+="form-control" %}
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

```

<!-- CPF -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">CPF</span>
  </div>
  {% render_field form.cpf class+="form-control" %}
</div>

<!-- Tempo de Serviço -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">Tempo de Serviço</span>
  </div>
  {% render_field form.tempo_de_servico class+="form-control" %}
</div>

<!-- Remuneração -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text">Remuneração</span>
  </div>
  {% render_field form.remuneracao class+="form-control" %}
</div>
<button class="btn btn-primary">Enviar</button>
</form>
</div>
</div>
</div>
</div>
</div>
{% endblock %}

```

Nesse *template*, não temos nada de novo.

Perceba que o 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 clicar no botão de edição).

Sua interface deve ficar similar à:

Atualização de Funcionário x +

localhost:8000/funcionario/3

Página Inicial Funcionários

Atualizar Dados de Funcionário

Nome João

Sobrenome Carlos

CPF 123.456.789-00

Tempo de Serviço 5

Remuneração 15000.00

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:

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

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

<!-- Bloco de conteúdo da nossa página -->
{% block conteúdo %}
<div class="container mt-5">
<div class="card">
```

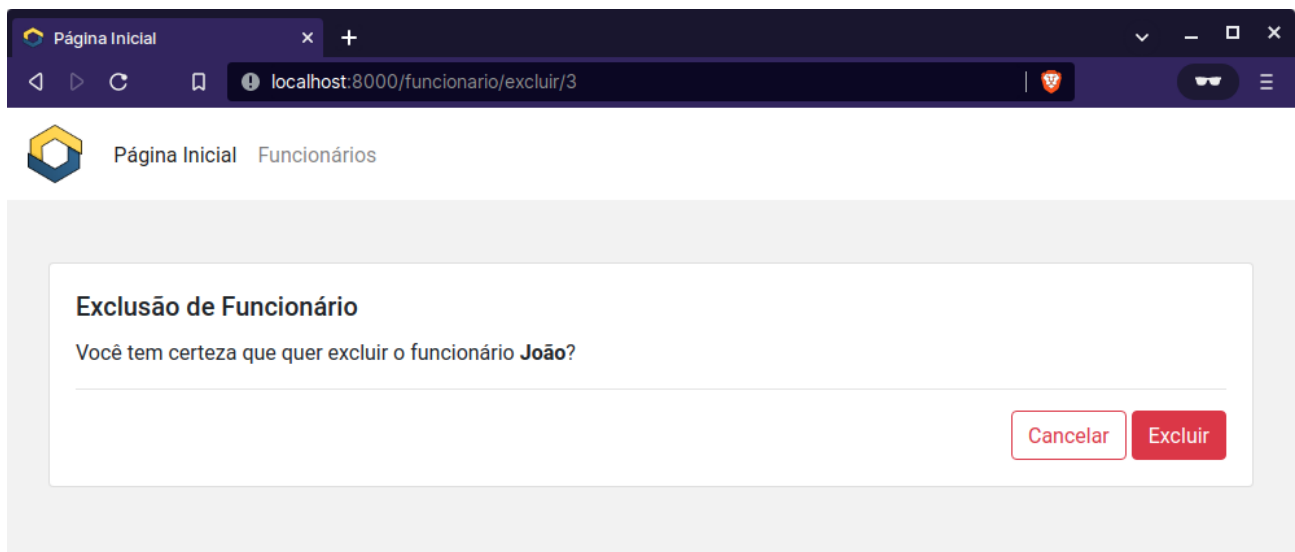
```

<div class="card-body">
  <h5 class="card-title">Exclusão de Funcionário</h5>
  <p class="card-text">
    Você tem certeza que quer excluir o funcionário <b>{{ funcionario.nome }}</b>?
  </p>
  <form method="post">
    {% csrf_token %}
    <hr />
    <div class="text-right">
      <a href="{% url 'website:lista_funcionarios' %}" class="btn btn-outline-danger">
        Cancelar
      </a>
      <button class="btn btn-danger">Excluir</button>
    </div>
  </form>
</div>
</div>
</div>
{% endblock %}

```

Aqui, novamente 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!

Você com certeza aprendeu bastante nessa caminhada! Mas calma que ainda não terminou, ainda temos mais conteúdo para que você fique craque em Django! 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**! E já que ele dispõe dessa capacidade, vamos explorá-la construindo uma **tag** que irá nos dizer o **tempo atual formatado** e um **filtro** que irá retornar **a primeira letra da string passada**.

Mas primeiro, vamos começar com a **configuração** necessária!

CONFIGURAÇÃO

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

Sendo assim, crie na raiz do *app website* essa pasta (**website/templatetags**) e adicione:

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

Nossa estrutura, portanto, deve ficar:

```
- website/  
  ...  
  - templatetags/  
    - __init__.py  
    - tempo_atual.py  
    - primeira_letra.py  
  ...
```

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 %}**.

E já que temos que escolher um para começar: vamos começar desenvolvendo o **filtro**.

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

```
<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.

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.

Para entender melhor, vamos ver um exemplo de filtro nativo do próprio 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`:

```
@register.filter(is_safe=True)
@stringfilter
def lower(value):
    """Convert a string into all lowercase."""
    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 *post* do Blog da Python Academy: [Domine Decorators em Python](#)).

- `@stringfilter` é um *decorator* utilizado para dizer ao Django que seu filtro espera uma string como argumento.

Agora que viu um filtro real do Django, vamos **codificar** e **registrar** nosso próprio filtro!

Uma forma de pegarmos a primeira letra de uma string é através da indexação, acessando o índice `[0]`, da seguinte forma:

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def primeira_letra(value):
    return 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 algum *template*. Para isso, vamos alterar a tabela do *template* `website/lista.html` para incluir nosso filtro da seguinte forma:

```
<!-- Primeiro, carregamos nosso filtro, logo após o extends -->
{% load primeira_letra %}
...
<table class="table">
  <thead class="thead-dark">
    <tr>
      <th><!-- Retiramos o "ID" aqui --></th>
      <th>Nome</th>
      <th>Sobrenome</th>
      <th>Tempo de Serviço</th>
      <th>Remuneração</th>
      <th class="text-center">Ações</th>
    </tr>
  </thead>
  <tbody>
    {% for f in funcionarios %}
```

```

<tr>
  <!-- Aplicamos nosso filtro no atributo funcionario.nome -->
  <td>{{ f.nome|primeira_letra }}</td>
  <td>{{ f.nome }}</td>
  <td>{{ f.sobrenome }}</td>
  <td>{{ f.tempo_de_servico }}</td>
  <td>{{ f.remuneracao }}</td>
  <td class="text-center">
    <a class="btn btn-primary"
      href="{% url 'website:atualiza_funcionario' pk=f.id %}">
      Atualizar
    </a>
    <a class="btn btn-danger"
      href="{% url 'website:deleta_funcionario' pk=f.id %}">
      Excluir
    </a>
  </td>
</tr>
{% endfor %}
</tbody>
</table>

```

O que resulta em:

Lista de Funcionários

Aqui está a lista de **Funcionários** cadastrados.

| Nome | Sobrenome | Tempo de Serviço | Remuneração | Ações | |
|------|-----------|------------------|-------------|----------|---|
| J | João | Carlos | 5 | 15000,00 | Atualizar Excluir |

[Cadastrar Funcionário](#)

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` - como a própria tradução já diz: “simples tag” - é uma ferramenta para construção de *tags* simples. Com ela, a criação de *tags* fica similar à criação de filtros, que vimos na seção passada.

Primeiro, precisamos incluir uma instância de `template.Library` (para ter acesso à biblioteca de filtros e *tags* do Django). Em seguida, 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:

```
import datetime
from django import template

register = template.Library()

@register.simple_tag
def tempo_atual():
    return datetime.datetime.now().strftime('%H:%M:%S')
```

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

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

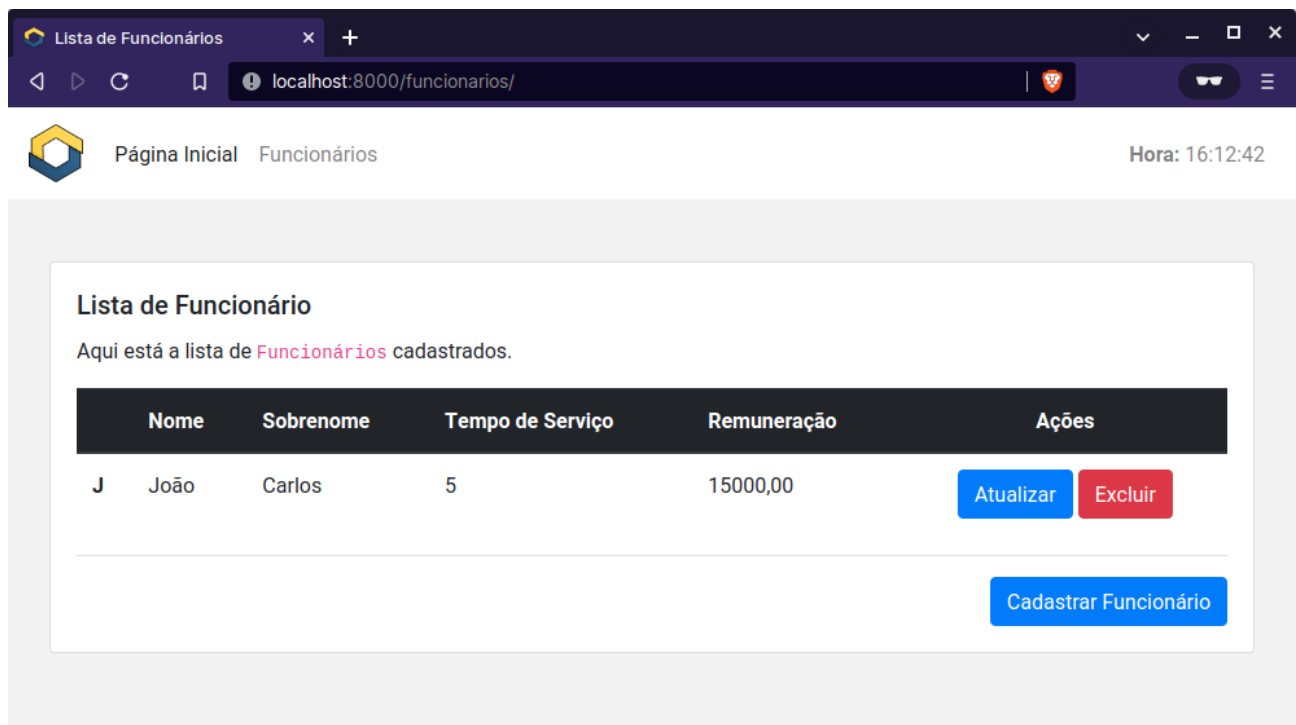
Vamos adicionar um novo item à barra de navegação (do lado direito), da seguinte forma:

```

<body>
<!-- Navbar -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
...
<div class="collapse navbar-collapse" id="navbarSupportedContent">
<ul class="navbar-nav mr-auto">
<li class="nav-item active">
<a class="nav-link" href="{% url 'website:index' %}">
Página Inicial
</a>
</li>
<li class="nav-item">
<a class="nav-link" href="{% url 'website:lista_funcionarios' %}">
Funcionários
</a>
</li>
</ul>
<!-- Adicione a lista abaixo -->
<ul class="navbar-nav float-right">
<li class="nav-item">
<!-- Aqui está nosso filtro -->
<a class="nav-link" href="#"><b>Hora: </b>{% tempo_atual %}</a>
</li>
</ul>
</div>
</nav>
...

```

O resultado deve ser:



Lista de Funcionários

Aqui está a lista de **Funcionários** cadastrados.

| | Nome | Sobrenome | Tempo de Serviço | Remuneração | Ações |
|---|------|-----------|------------------|-------------|---|
| J | João | Carlos | 5 | 15000,00 | Atualizar Excluir |

[Cadastrar Funcionário](#)

Veja a hora do lado direito superior, na barra de navegação (**Hora: 16:12:42**)!

As possibilidades são infinitas!

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*!

FILTROS DO DJANGO

É possível fazer muita coisa com os filtros que já vêm instalados no próprio Django. Muitas vezes, é melhor você fazer algumas operações no *template* do que fazê-las no *backend* (desde que sejam operações de apresentação, **apenas**). Sempre verifique a viabilidade de um ou de outro para facilitar sua vida!

Como a lista de *built-in filters* do Django é beeeeeem 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:

```
{{ 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:

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

Saída:

FILTRO **date**

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

Exemplo:

Entrada: Objeto **datetime**.

Utilização:

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

Saída:

```
01/07/2018
```

FILTRO **filesizeformat**

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

Exemplo:

Entrada: **valor** = 123456789

Utilização:

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

Saída:

```
117.7 MB
```

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:

```
{{ 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:

```
{{ 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:

```
{% if valor|length > 0 %}  
  <p>Lista contém valores</p>  
{% else %}  
  <p>Lista vazia</p>  
{% 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:

```
{{ 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:

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

Saída:

```
Sua empresa tem 12 Funcionários
```

FILTRO **upper**

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

Exemplo:

Entrada: **valor** = texto de testes

Utilização:

```
{{ 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:

```
{{ 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

Neste 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!**

FINALIZAÇÃO

UM ATÉ BREVE...

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/>
- Github do Django: <https://github.com/django/django>
- Twitter do Django: <https://twitter.com/djangoproject>
- Django RSS: <https://www.djangoproject.com/rss/weblog/>
- Grupo do Facebook: <https://www.facebook.com/groups/django.brasil/>

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

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