Django Tutorial Parte 5: Criando nossa home page

Agora estamos prontos para adicionar o código que exibe nossa primeira página completa - uma home page do site LocalLibrary. A página inicial mostrará o número de registros que temos para cada tipo de modelo e fornecerá links de navegação na barra lateral para nossas outras páginas. Ao longo do caminho, obteremos experiência prática ao escrever mapas e visualizações básicos de URL, obter registros do banco de dados e usar modelos.

Pré-requisitos:	Leia a Introdução ao Django. Conclua os tópicos do tutorial anterior (incluindo Django Tutorial Part 4: Django admin site).
Objetivo:	Aprender a criar mapas e visualizações de URL simples (onde nenhum dado é codificado no URL), obtenha dados de modelos e crie modelos.

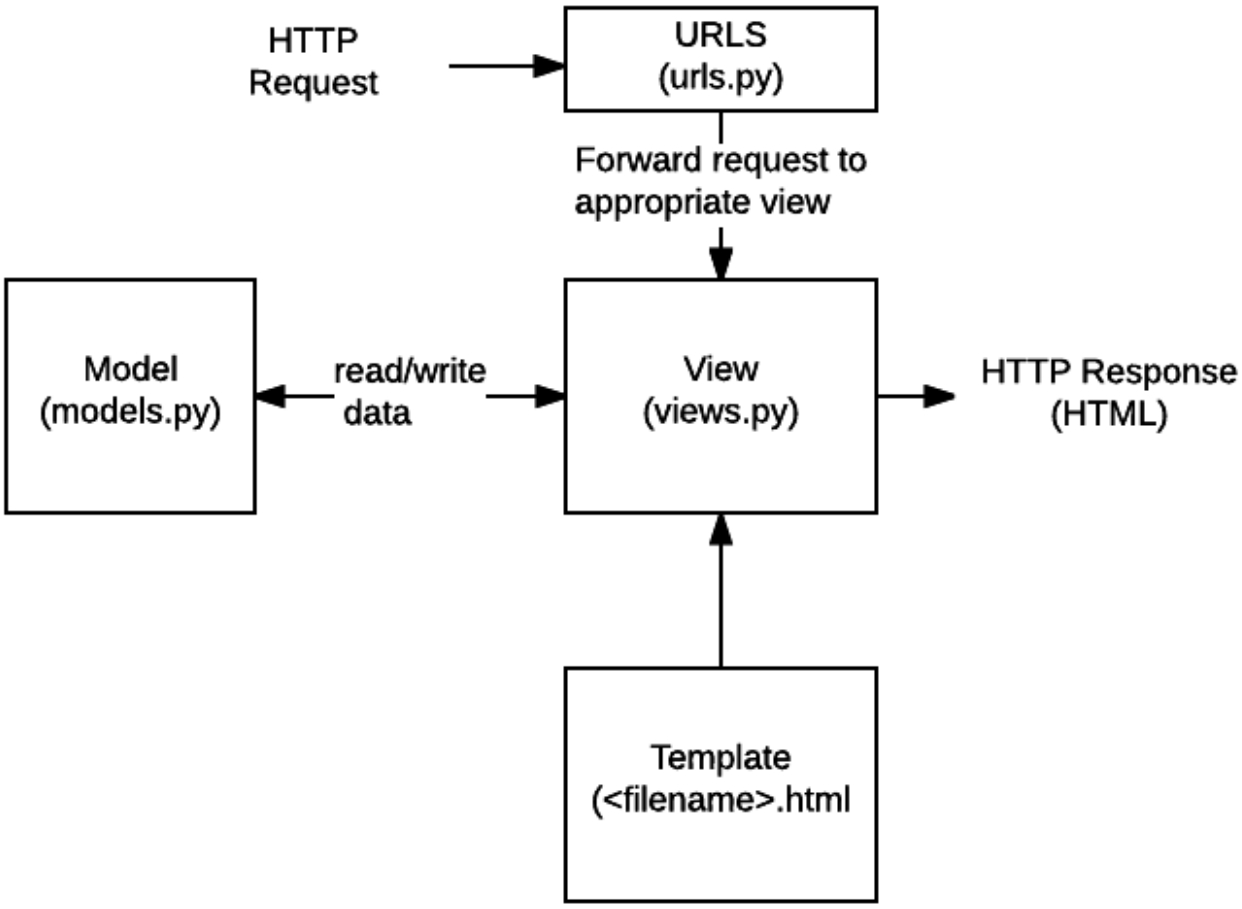
Visão Global

Depois de definirmos nossos modelos e criarmos alguns registros iniciais da biblioteca para trabalhar, é hora de escrever o código que apresenta essas informações aos usuários. A primeira coisa que precisamos fazer é determinar quais informações queremos exibir em nossas páginas e definir os URLs a serem usados para retornar esses recursos. Em seguida, criaremos um mapeador de URLs, visualizações e modelos para exibir as páginas.

O diagrama a seguir descreve o fluxo de dados principal e os componentes necessários ao manipular solicitações e respostas HTTP. Como já implementamos o modelo, os principais componentes que criaremos são:

- Mapeadores de URL para encaminhar os URLs suportados (e qualquer informação codificada nos URLs) para as funções de exibição apropriadas.
- View functions para obter os dados solicitados dos modelos, crie páginas HTML que exibem os dados e retorne as páginas ao usuário para visualização no navegador.

- Templates para usar ao renderizar dados nas visualizações.



Como você verá na próxima seção, temos 5 páginas para exibir, o que é muita informação para documentar em um único artigo. Portanto, este artigo se concentrará em como implementar a página inicial e abordaremos as outras páginas em um artigo subsequente. Isso deve fornecer uma boa compreensão completa de como os mapeadores, visualizações e modelos de URL funcionam na prática.

Definindo os URLs do recurso

Como esta versão do LocalLibrary é essencialmente somente leitura para usuários finais, precisamos fornecer uma página de destino para o site (uma página inicial) e páginas que exibam visualizações de lista e detalhes de livros e autores.

As URLs que iremos precisar na nossa página são:

- `catalog/` — A página inicial (index).
- `catalog/books/` — Uma lista de todos os livros.
- `catalog/authors/` — Uma lista de todos os autores.
- `catalog/book/<id>` — A exibição de detalhes de um livro específico, com uma chave primária de campo `<id>` (o padrão). Por exemplo, o URL do terceiro livro adicionado à lista será `/catalog/book/3`.
- `catalog/author/<id>` — A exibição de detalhes para o autor específico com um campo de chave primária de `<id>`. Por exemplo, o URL do 11º autor adicionado à lista será `/catalog/author/11`.

Os três primeiros URLs retornarão a página de índice, a lista de livros e a lista de autores. Esses URLs não codificam nenhuma informação adicional e as consultas que buscam dados no banco de dados sempre serão as mesmas. No entanto, os resultados retornados pelas consultas dependerão do conteúdo do banco de dados.

Por outro lado, os dois URLs finais exibirão informações detalhadas sobre um livro ou autor específico. Esses URLs codificam a identidade do item a ser exibido (representado por `<id>` acima). O mapeador de URLs extrairá as informações codificadas e as passará para a visualização, e a visualização determinará dinamicamente quais informações serão obtidas do banco de dados. Ao codificar as informações no URL, usaremos um único conjunto de mapeamento de URL, uma visualização e um modelo para lidar com todos os livros (ou autores).

> **Nota**: Com o Django, você pode construir suas URLs da maneira que desejar - você pode codificar informações no corpo da URL, como mostrado acima, ou incluir `GET` parâmetros no URL, por exemplo `/book/?id=6`. Qualquer que seja a abordagem usada, os URLs devem ser mantidos limpos, lógicos e legíveis, conforme recomendado pelo W3C.
> A documentação do Django recomenda informações de codificação no corpo da URL para obter um melhor design da URL.

Conforme mencionado na visão geral, o restante deste artigo descreve como construir a página index.

Criando a página index

A primeira página que criaremos é a página index (`catalog/`). A pagina index incluirá algum HTML estático, juntamente com "contagens" geradas de diferentes registros no banco de dados. Para fazer isso funcionar, criaremos um mapeamento de URL, uma visualização e um modelo.

> **Nota**: Vale a pena prestar um pouco de atenção extra nesta seção. A maioria das informações também se aplica às outras páginas que criaremos.

Mapeamento de URL

Quando criamos o esqueleto do website, atualizamos o arquivo **locallibrary/urls.py** para garantir que sempre que um URL que comece com `catalog/` é recebido, o módulo URLConf `catalog.urls` processará a substring restante.

O seguinte snippet de código de **locallibrary/urls.py** inclui o modulo `catalog.urls`:

```
urlpatterns += [
    path('catalog/', include('catalog.urls')),
]
```

> **Nota**: Sempre que o Django encontra a função de importação `django.urls.include()`, divide a string da URL no caractere final designado e envia a subsequência restante para o módulo URLconf incluído para processamento adicional.

Também criamos um arquivo de espaço reservado para o modulo *URLConf*, chamado **/catalog/urls.py**. Adicione as seguintes linhas a esse arquivo:

```
urlpatterns = [
    path('', views.index, name='index'),
]
```

A função `path()` define o seguinte:

- Um padrão de URL, que é uma sequência vazia: `''`. Discutiremos detalhadamente os padrões de URL ao trabalhar em outras visualizações.
- A view function that will be called if the URL pattern is detected: `views.index`, which is the function named `index()` in the **views.py** file.

A função `path()` também especifica um parâmetro de nome, que é um identificador exclusivo para esse mapeamento de URL específico. Você pode usar o nome para "reverter" o mapeador, ou seja, para criar dinamicamente um URL que aponte para o recurso que o mapeador foi projetado para manipular. Por exemplo, podemos usar o parâmetro name para vincular à nossa home page a partir de qualquer outra página adicionando o seguinte link em um modelo:

```
<a href="{% url 'index' %}">Home</a>.
```

> **Nota**: Podemos codificar o link como em `<a href="/catalog/">Home</a>`), mas se alterarmos o padrão da nossa página inicial, por exemplo, para `/catalog/index`) os modelos não serão mais vinculados corretamente. Usar um mapeamento de URL invertido é muito mais flexível e robusto.

View (function-based)

Uma view é uma função que processa uma solicitação HTTP, busca os dados necessários no banco de dados, renderiza os dados em uma página HTML usando um modelo HTML e, em seguida, retorna o HTML gerado em uma resposta HTTP para exibir a página ao usuário. A visualização do índice segue esse modelo - ele busca informações sobre o número de `Book`, `BookInstance`, disponibilidade de `BookInstance` e registros de `Author` que temos no banco de dados e passa essas informações para um modelo para exibição.

Abra **catalog/views.py** e observe que o arquivo já importa o render() da função shortcuts para gerar arquivo HTML usando um modelo e dados:

```
from django.shortcuts import render

# Create your views here.
```

Cole as seguintes linhas na parte inferior do arquivo:

```
from catalog.models import Book, Author, BookInstance, Genre

def index(request):
    """View function for home page of site."""
```

```
    # Generate counts of some of the main objects
    num_books = Book.objects.all().count()
    num_instances = BookInstance.objects.all().count()


    # Available books (status = 'a')
    num_instances_available = BookInstance.objects.filter(status__exact=

    # The 'all()' is implied by default.
    num_authors = Author.objects.count()

    context = {
        'num_books': num_books,
        'num_instances': num_instances,
        'num_instances_available': num_instances_available,
        'num_authors': num_authors,
    }

    # Render the HTML template index.html with the data in the context v
    return render(request, 'index.html', context=context)
```

A primeira linha importa as classes de models que usaremos para acessar dados em todas as nossas visualizações.

A primeira parte da função view busca o número de registros usando o atributo `objects.all()` nas classes de modelo. Também recebe uma lista de objetos de `BookInstance` que possuem um valor de 'a' (Disponibilidade) no campo status. Você pode encontrar mais informações sobre como acessar os dados do modelo em nosso tutorial anterior Django Tutorial Part 3: Using models > Searching for records.

No final da função view chamamos a função `render()` para criar uma página HTML e retornar a página como resposta. essa função de atalho envolve várias outras funções para simplificar um caso de uso muito comum. A função `render()` aceita os seguintes parâmetros:

- o objeto `request` original, que é um `HttpRequest`.
- um modelo HTML com espaços reservados para os dados.
- uma variável `context`, que é um dicionário Python, contendo os dados a serem inseridos nos espaços reservados.

Falaremos mais sobre modelos e variáveis `context` na próxima seção. Vamos criar nosso template para que possamos exibir algo para o usuário!

Template

Um template é um arquivo de texto que define a estrutura ou o layout de um arquivo (como uma página HTML), usa espaços reservados para representar o conteúdo real.

Django irá procurar automaticamente templates na pasta chamada '**templates**' em sua aplicação. Por exemplo, na exibição de index que acabamos de adicionar, a função `render()` espera encontrar o arquivo ***index.html*** *em* **/locallibrary/catalog/templates/** e gera um erro se o arquivo não estiver presente.

Você pode verificar isso salvando as alterações anteriores e acessando `127.0.0.1:8000` no seu navegador - ele exibirá uma mensagem de erro bastante intuitiva: "`TemplateDoesNotExist at /catalog/`", e outros detalhes.

> **Nota**: Com base no arquivo de configurações do seu projeto, o Django procurará templates em vários locais, pesquisando nos aplicativos instalados por padrão. Você pode descobrir mais sobre como o Django encontra templates e quais formatos ele suporta no the Templates section of the Django documentation.

Estendendo templates

O index template precisará de marcação HTML padrão para head e a body, juntamente com as seções de navegação para criar um link para as outras páginas do site (que ainda não criamos) e para as seções que exibem dados introdutórios de texto e livro.

Grande parte da estrutura HTML e de navegação será a mesma em todas as páginas do nosso site. Em vez de duplicar o código padrão em todas as páginas, você pode usar a linguagem de modelagem do Django para declarar um modelo base e depois estendê-lo para substituir apenas os bits que são diferentes para cada página específica.

O seguinte snippet de código é um template base de amostra de um arquivo **base_generic.html**. Em breve, criaremos o modelo para a LocalLibrary. O exemplo abaixo inclui HTML comum com seções para um título, uma barra lateral e o conteúdo principal marcado com as template tags de nome `block` e `endblock`, mostrado em negrito. Você pode deixar os blocos vazios ou incluir o

nome `block` e `endblock`, mostrado em negrito. Você pode deixar os blocos vazios ou incluir o conteúdo padrão a ser usado ao renderizar páginas derivadas do modelo.

> **Nota**: Template *tags* são funções que você pode usar em um modelo para percorrer as listas, executar operações condicionais com base no valor de uma variável e assim por diante. Além das template tags, a sintaxe template permite que você faça referência a variáveis que são passadas para a template a partir da view e use filtros de template para formatar variáveis (por exemplo, para converter uma sequência em minúscula).

```
<!DOCTYPE html>
<html lang="en">
<head>
  {% block title %}<title>Local Library</title>{% endblock %}
</head>
<body>
  {% block sidebar %}<!-- insert default navigation text for every page
  {% block content %}<!-- default content text (typically empty) -->{% e
</body>
</html>
```

Ao definir um template para uma visualização específica, primeiro especificamos o template base usando a template tag `extends` — veja o exemplo de código abaixo. Em seguida, declaramos quais seções do template queremos substituir (se houver), usando seções `block`/`endblock` como no template base.

Por exemplo, o trecho de código abaixo mostra como usar a template tag `extends` e substituir o block `content`. O HTML gerado incluirá o código e a estrutura definidos no template base, incluindo o conteúdo padrão que você definiu no block `title`, mas o novo block `content` no lugar do padrão.

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Local Library Home</h1>
  <p>Welcome to LocalLibrary, a website developed by <em>Mozilla Develop
{% endblock %}
```

O template base LocalLibrary

Usaremos o seguinte snippet de código como modelo básico para o site *LocalLibrary*. Como você pode ver, ele contém algum código HTML e define blocos para `title`, `sidebar`, e `content`. Temos um título padrão e uma barra lateral padrão com links para listas de todos os livros e autores, ambos colocados em blocos para serem facilmente alterados no futuro.

> **Nota**: Também introduzimos duas template tags adicionais: `url` e `load static`. Essas tags serão explicadas nas próximas seções.

Crie um novo arquivo ***base_generic.html*** em **/locallibrary/catalog/templates/** e cole o seguinte código no arquivo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  {% block title %}<title>Local Library</title>{% endblock %}
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootst
  <!-- Add additional CSS in static file -->
  {% load static %}
  <link rel="stylesheet" href="{% static 'css/styles.css' %}">
</head>
<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-sm-2">
      {% block sidebar %}
        <ul class="sidebar-nav">
          <li><a href="{% url 'index' %}">Home</a></li>
          <li><a href="">All books</a></li>
          <li><a href="">All authors</a></li>
        </ul>
     {% endblock %}
      </div>
      <div class="col-sm-10 ">{% block content %}{% endblock %}</div>
    </div>
```

```
    </div>
  </div>
</body>
</html>
```

O template inclui CSS de Bootstrap para melhorar o layout e a apresentação da página HTML. O uso do Bootstrap (ou outra estrutura da Web do lado do cliente) é uma maneira rápida de criar uma página atraente que é exibida bem em diferentes tamanhos de tela.

O template base também faz referência a um arquivo css local (**styles.css**) que fornece estilo adicional. Criar um arquivo **styles.css** em **/locallibrary/catalog/static/css/** e cole o seguinte código no arquivo:

```
.sidebar-nav {
    margin-top: 20px;
    padding: 0;
    list-style: none;
}
```

O template index

Crie um novo arquivo HTML ***index.html*** em **/locallibrary/catalog/templates/** e cole o seguinte código no arquivo Esse código estende nosso modelo base na primeira linha e substitui o padrão block `content` para o template.

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Local Library Home</h1>
  <p>Welcome to LocalLibrary, a website developed by <em>Mozilla Develop
  <h2>Dynamic content</h2>
  <p>The library has the following record counts:</p>
  <ul>
    <li><strong>Books:</strong> {{ num_books }}</li>
    <li><strong>Copies:</strong> {{ num_instances }}</li>
    <li><strong>Copies available:</strong> {{ num_instances_available }}
    <li><strong>Authors:</strong> {{ num_authors }}</li>
  </ul>
{% endblock %}
```

Na seção *Dynamic content*, declaramos espaços reservados (*variáveis de template*) para as

Na seção *Dynamic content*, declaramos espaços reservados (*variáveis de template*) para as informações da exibição que queremos incluir. As variáveis são colocadas entre chaves (guiador), como mostrado em negrito no exemplo de código.

> **Nota:** Você pode reconhecer facilmente variáveis de template e template tags (funções) - as variáveis são colocadas entre chaves (`{{ num_books }}`), e as tags são colocadas em chaves simples com sinais de porcentagem (`{% extends "base_generic.html" %}`).

O importante a ser observado aqui é que as variáveis são nomeadas com as *chaves* que passamos para o dicionário `context` na função `render()` da nossa view (veja a amostra abaixo). As variáveis serão substituídas pelos *valores* associados quando o modelo for renderizado.

```
context = {
    'num_books': num_books,
    'num_instances': num_instances,
    'num_instances_available': num_instances_available,
    'num_authors': num_authors,
}

return render(request, 'index.html', context=context)
```

Referenciando arquivos estáticos nos templates

É provável que seu projeto use recursos estáticos, incluindo JavaScript, CSS e imagens. Como a localização desses arquivos pode não ser conhecida (ou pode mudar), o Django permite que você especifique a localização em seus modelos em relação a configuração global `STATIC_URL`. O site padrão do esqueleto define o valor de `STATIC_URL` para '`/static/`', mas você pode optar por hospedá-los em uma rede de entrega de conteúdo ou em outro local.

Dentro do template que você chama primeiro na template tag `load` especificando "static" para adicionar a biblioteca de modelos, conforme mostrado no exemplo de código abaixo. Você pode então usar a template tag `static` e especifique o URL relativo ao arquivo necessário.

```
<!-- Add additional CSS in static file -->
{% load static %}
```

```
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

Você pode adicionar uma imagem à página de maneira semelhante, por exemplo:

```
{% load static %}
<img src="{% static 'catalog/images/local_library_model_uml.png' %}" alt
```

> **Nota**: Os exemplos acima especificam onde os arquivos estão localizados, mas o Django não os serve por padrão. Configuramos o servidor da web de desenvolvimento para exibir arquivos modificando o mapeador de URL global (**/locallibrary/locallibrary/urls.py**) quando criamos o esqueleto do website, mas ainda precisamos ativar a veiculação de arquivos na produção. Veremos isso mais tarde.

Para obter mais informações sobre como Trabalhar com arquivos estaticos, consulte Gerenciando arquivos estáticos na documentação do Django.

Vinculando as URLs

O template base abaixo introduziu a template tag `url`.

```
<li><a href="{% url 'index' %}">Home</a></li>
```

Essa tag aceita o nome de uma função `path()` chamado em **urls.py** e os valores para quaisquer argumentos que a view associada receberá dessa função e retorna um URL que você pode usar para vincular ao recurso.

Configurando onde encontrar os templates

Você precisa dizer ao Django para procurar seus templates na pasta de templates. Para fazer isso, adicione o diretório de templates ao objeto TEMPLATES editando o arquivo **settings.py**, como mostrado em negrito, no seguinte exemplo de código:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates'),
        ]
```

```
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Com o que se parece?

Neste ponto, criamos todos os recursos necessários para exibir a página index. Execute o servidor (`python3 manage.py runserver`) e abra http://127.0.0.1:8000/ no seu navegador. Se tudo estiver configurado corretamente, seu site deverá ter a seguinte captura de tela.

.

> **Nota:** Os links **All books** e **All authors** ainda não funcionarão porque os caminhos, visualizações e modelos para essas páginas não estão definidos. Acabamos de inserir espaços reservados para esses links no template `base_generic.html`.

Desafie-se

Temos duas tarefas para testar a sua familiaridade com as consultas de modelos, views e templates

1. O modelo de base da BibliotecaLocal inclui um bloco de `título`. Substitua este bloco no modelo de índice e crie um novo título para a página.
2. > **Dica:** A seção Extendendo Templates explica como criar blocos e extender um bloco em outro template.

3. Modifique a view para gerar contagens para gêneros e livros que contenham uma palavra específica (case insensitive), e passe o resultado para o `contexto`. Isso é feito de

maneira semelhante à criação e uso de `num_books` e `num_instances_available`. Em seguida, atualize o template do index para incluir essas variáveis.

Resumo

Acabamos de criar a página inicial do nosso site - uma página HTML que exibe uma série de registros do banco de dados e links para outras páginas ainda a serem criadas. Ao longo do caminho, aprendemos informações fundamentais sobre mapeadores de url, views, consulta do banco de dados com modelos, passagem de informações para um modelo a partir de uma view e criação e extensão de templates.

No próximo artigo, continuaremos sobre esse conhecimento para criar as quatro páginas restantes de nosso site.

Veja também

- Escrevendo sua primeira aplicação Django, parte 3: View e Templates (documentação do Django)
- Despachante de URL (Django docs)
- Funções das Views (DJango docs)
- Templates (Django docs)
- Gerenciando arquivos estáticos (Django docs)
- Funções de atalho do Django (Django docs)

Nesse Módulo

- Introdução ao Django
- Configurando um ambiente de desenvolvimento Django
- Tutorial Django: Website de uma Biblioteca Local
- Tutorial Django Parte 2: Criando a base do website
- Tutorial Django Parte 3: Utilizando models
- Tutorial Django Parte 4: Django admin site
- Tutorial Django Parte 5: Criando nossa página principal
- Tutorial Django Parte 6: Lista genérica e detail views
- Tutorial Django Parte 7: Sessões de Framework
- Tutorial Django Parte 9: Trabalhando com formulários
- Tutorial Django Parte 10: Testando uma aplicação web Django

- Tutorial Django Parte 11: Implantando Django em produção
- Segurança de aplicações web Django
- DIY Django mini blog

Last modified: 26 de ago. de 2020, by MDN contributors

Change your language

Português (do Brasil) Change language