04

Melhorando a edição de Jogos

Transcrição

Existe um problema na nossa função "Editar". Quando trocamos a imagem de um jogo e salvamos, parece estar tudo correto. Porém, se voltarmos à janela de edição, a capa anterior continuará lá.

Cada vez que fazemos uma edição, o arquivo é salvo com um nome "capa + id do jogo" e a extensão `.jpg`. Quando salvamos uma edição, a imagem é atualizada no servidor, mas nada muda no navegador.

Isso acontece porque o navegador reconhece que já tem aquele arquivo em memória, e não precisa buscá-lo novamente no servidor. Esse tipo de *cache* na verdade ajuda o usuário a navegar na web com mais facilidade e rapidez.

Mas como resolveremos o problema que esse *cache* gera?

Na função `criar()`, quando o usuário faz o upload de uma imagem, seria mais interessante se ela fosse criada no servidor com um nome único. Dessa forma, se a imagem é substituída, o navegador saberá que precisa atualizá-la.

Uma forma de implementarmos essa funcionalidade é adicionando outra informação única para cada upload - por exemplo, um *timestamp* do momento em que ele é feito.

Para isso, iremos importar a biblioteca `time`, nativa do Pytho. Em seguida, buscaremos o tempo atual com a função `time()`, e passaremos o retorno dela para uma variável `timestamp`. Essa variável então será concatenada no nome do nosso arquivo:

```
@app.route('/criar', methods=['POST',])
def criar():
    nome = request.form['nome']
    categoria = request.form['categoria']
    console = request.form['console']
    jogo = Jogo(nome, categoria, console)
    jogo = jogo_dao.salvar(jogo)

    arquivo = request.files['arquivo']
    upload_path = app.config['UPLOAD_PATH']
    timestamp = time.time()
    arquivo.save(f'{upload_path}/capa{jogo.id}-{timestamp}.jpg')
    return redirect(url_for('index'))
```

Em `editar()`, o método `render_template()` terá que encontrar a imagem com o *timestamp*. Ao invés de tentarmos encontrar uma forma de adivinhar o *timestamp* de cada imagem, podemos buscar sempre por `capa{jogo.id} -`já que toda imagem terá uma `id` associada. Antes de passarmos a imagem para o *template*, vamos recuperar o nome dessa imagem (`nome_imagem`) com uma função `recupera_imagem()` recebendo o `id`:

```
@app.route('/editar/<int:id>')
def editar(id):
    if 'usuario_logado' not in session or session['usuario_logado'] == None:
        return redirect(url_for('login', proxima=url_for('editar')))
    jogo = jogo_dao.busca_por_id(id)
```

```
    nome_imagem =  recupera_imagem(id)
    capa_jogo = f'capa{id}.jpg'
    return render_template('editar.html', titulo='Editando jogo', jogo=jogo,
                           capa_jogo = f'capa{id}.jpg')
```

Agora criaremos a função `recupera_imagem()`. Nela, teremos que percorrer todos os arquivos na pasta de uploads para encontrar uma imagem pelo `id` específico dela. Para isso, usaremos `for nome_arquivo` dentro de uma lista de arquivos.

A lista de arquvivos em "uploads" pode ser acessada com `app.config[]` passando a chave para esse dicionário, que é `UPLOAD_PATH`. Porém, `app.config[]` na verdade retorna uma string, e queremos percorrer os arquivos propriamente ditos. O pacote `os` do Python nos ajuda a solucionar isso com a função `listdir()`, que retorna o nome dos arquivos em um diretório.

Em seguida, verificaremos se `nome_arquivo` tem o prefixo específico que buscamos (capa + `jogo.id`). Se essa condição for verdadeira, retornaremos o nome correto do arquivo:

```
def recupera_imagem(id):
    for nome_arquivo in os.listdir(app.config['UPLOAD_PATH']):
        if f'capa{id}' in nome_arquivo
            return nome_arquivo

app.run(debug=True)
```

Agora, na função `editar()`, podemos passar a imagem correta para o `render_template()` com `nome_imagem`:

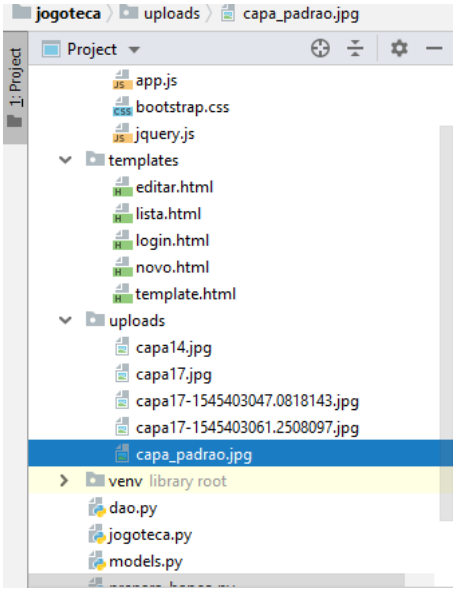
```
@app.route('/editar/<int:id>')
def editar(id):
    if 'usuario_logado' not in session or session['usuario_logado'] == None:
        return redirect(url_for('login', proxima=url_for('editar')))
    jogo = jogo_dao.busca_por_id(id)
    nome_imagem =  recupera_imagem(id)
    capa_jogo = f'capa{id}.jpg'
    return render_template('editar.html', titulo='Editando jogo', jogo=jogo,
                           capa_jogo = nome_imagem)
```

Para prosseguirmos, teremos que alterar também o código de `atualizar()` para incluir o *timestamp*:

```
@app.route('/atualizar', methods=['POST',])
def atualizar():
    nome = request.form['nome']
    categoria = request.form['categoria']
    console = request.form['console']
    jogo = Jogo(nome, categoria, console, id=request.form['id'])
    jogo_dao.salvar(jogo)

    arquivo = request.files['arquivo']
    upload_path = app.config['UPLOAD_PATH']
    timestamp = time.time()
    arquivo.save(f'{upload_path}/capa{jogo.id}-{timestamp}.jpg')
    return redirect(url_for('index'))
```

Se testarmos a função de mudar imagem na nossa aplicação, veremos que nosso problema se mantém: as imagens são enviadas para o servidor, mas o navegador continuará exibindo a que foi incluída em `/novo`.



Consegue adivinhar qual o problema? A função `recupera_imagem()` que criamos busca por `f'capa{id}`, e todas as imagens enviadas para o servidor começam com `f'capa{id}`. Para resolvermos isso, criaremos uma função que, no momento em que o usuário salva uma imagem nova, deleta todas as antigas para o mesmo `id`.

A função utilitária `deleta_arquivo()` deverá receber um `id`. Com ele, poderemos recuperar o nome do arquivo usando a função `recupera_imagem()` e passando novamente o `id`:

```
def deleta_arquivo(id):
    arquivo = recupera_imagem(id)
```

Para deletarmos esse arquivo, podemos usar `os.remove()` passando `arquivo`. Como é uma boa prática passarmos o caminho absoluto do arquivo, iremos juntar o caminho da pasta "uploads" com o nome do arquivo.

Para que isso funcione em qualquer sistema operacional, passaremos um `os.path.join()`, que consegue juntar diferentes *paths* - nesse caso, `app.config['UPLOAD_PATH']` e o nome do nosso arquivo (`arquivo`).

```
def deleta_arquivo(id):
    arquivo = recupera_imagem(id)
    os.remove(os.path.join(app.config['UPLOAD_PATH'], arquivo))
```

Nós queremos que essa função seja executada quando for criado um novo arquivo com um `id` existente na nossa aplicação. Portango, na funçao `atualizar`', antes da criação de um novo arquivo (`arquivo.save`), usaremos o `deleta_arquivo()` passando `jogo.id`.

Testando no navegador, veremos que agora nossa aplicação **jogoteca** consegue atualizar e guardar as imagens corretamente. Com essa solução, conseguimos deixá-la ainda mais funcional - podemos criar novos jogos, deletar, editar e incluir imagens no servidor.

Temos quase um CRUD completo utilizando, basicamente, um único arquivo com bastante código (incluindo bibliotecas CSS, códigos Javascript, etc.). Porém, nosso projeto não parece estar organizado da melhor forma possível.

No arquivo `jogoteca.py`, temos a configuração da aplicação, a configuração e a inicialização do banco de dados, as rotas, as definições de regras para as *views* e os métodos utilitários. Isso pode não ser prejudicial enquanto estamos trabalhando com uma aplicação pequena, porém, conforme ela vai crescendo, é interessante termos vários arquivos, cada um com sua responsabilidade.

Na próxima aula trataremos esse problema. Até lá!