# Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

## Aula 9



## Bem-vindo!

- Nas semanas anteriores, você aprendeu diversas linguagens de programação, técnicas e estratégias.
- Na verdade, esta aula foi muito menos uma aula de *C* ou *Python* e muito mais uma *aula de programação* , de forma que você possa acompanhar as tendências futuras.

- Nas últimas semanas, você aprendeu *como aprender* sobre programação.
- Hoje, vamos passar do HTML e CSS para a combinação de HTML, CSS, SQL, Python e JavaScript para que você possa criar seus próprios aplicativos web.
- Você pode considerar usar as habilidades que aprenderá esta semana para criar seu projeto final.

## servidor http

- Até este ponto, todo o HTML que você viu era pré-escrito e estático.
- Antigamente, quando você visitava uma página, o navegador baixava uma página HTML, que você podia visualizar. Essas são consideradas páginas *estáticas* , pois o que está programado no HTML é exatamente o que o usuário vê e baixa no *lado do cliente* para o seu navegador de internet.
- Páginas dinâmicas referem-se à capacidade do Python e linguagens similares de criar HTML sob demanda. Assim, é possível ter páginas web geradas *no servidor* por código, com base na entrada ou no comportamento dos usuários.
- Você já utilizou `http-server` um servidor para exibir suas páginas da web. Hoje, vamos utilizar um novo servidor capaz de analisar um endereço web e executar ações com base na URL fornecida.
- Além disso, na semana passada, você viu URLs como as seguintes:

  ```
  https://www.example.com/folder/file.html
  ```

  Observe que `file.html` é um arquivo HTML dentro de uma pasta chamada `folder` em `example.com` .

## Frasco

- Esta semana, introduzimos a capacidade de interagir com *rotas* como `https://www.example.com/route?key=value` , onde funcionalidades específicas podem ser geradas no servidor através das chaves e valores fornecidos na URL.
- *Flask* é uma biblioteca de terceiros que permite hospedar aplicações web usando o framework Flask, ou um microframework, dentro do Python.
- Você pode executar o Flask digitando o seguinte comando `flask run` na janela do terminal, no diretório cs50.dev (https://cs50.dev) .
- Para isso, você precisará de um arquivo chamado `flask.yml` `app.py` e outro chamado `flask.yml` `requirements.txt` . `app.py` O arquivo `flask.yml` contém o código que informa ao Flask como executar sua aplicação web. O arquivo `flask.yml` `requirements.txt` inclui uma lista das bibliotecas necessárias para a execução da sua aplicação Flask.
- Aqui está um exemplo de `requirements.txt` :

```
Flask
```

O aviso aparece apenas `Flask` neste arquivo. Isso ocorre porque o Flask é necessário para executar a aplicação Flask.

- Aqui está um aplicativo Flask muito simples em `app.py`:

```
# Says hello to world by returning a string of text

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return "hello, world"
```

Observe que a `/` rota simplesmente retorna o texto `hello, world`.

- Também podemos criar código que implemente HTML:

```
# Says hello to world by returning a string of HTML

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return '<!DOCTYPE html><html lang="en"><head><title>hello</title></head><b
```

Note que, em vez de retornar um texto simples, isso fornece HTML.

- Para melhorar nossa aplicação, também podemos servir HTML baseado em modelos, criando uma pasta chamada `templates` `templates` e um arquivo chamado `templates` `index.html` com o seguinte código dentro dessa pasta:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>hello</title>
    </head>

    <body>
        hello, {{ name }}
    </body>

</html>
```

Observe o caractere duplicado `{{ name }}`, que é um marcador para algo que será fornecido posteriormente pelo nosso servidor Flask.

- Em seguida, na mesma pasta em que a `templates` pasta aparece, crie um arquivo chamado `app.py` e adicione o seguinte código:

```
# Uses request.args.get

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    name = request.args.get("name", "world")
    return render_template("index.html", name=name)
```

Observe que este código define `app` `app` como a aplicação Flask. Em seguida, define a `/` rota de `app` `app` como retornando o conteúdo de `app` `index.html` com o argumento `name` `name`. Por padrão, a `request.args.get` função procurará o ` `name` name` fornecido pelo usuário. Se nenhum nome for fornecido, o padrão será `app` `world`. `@app.route` `name` também é conhecido como um decorador.

- Você pode executar este aplicativo web digitando o comando `flask run` na janela do terminal. Se o Flask não funcionar, verifique se a sintaxe de cada um dos arquivos acima está correta. Além disso, se o Flask ainda não funcionar, certifique-se de que seus arquivos estejam organizados da seguinte forma:

```
/templates
    index.html
app.py
requirements.txt
```

- Depois de iniciar o programa, você será solicitado a clicar em um link. Ao acessar a página, tente adicionar `?name=[Your Name]` o seguinte ao URL base na barra de endereços do seu navegador.

## Formulários

- Aprimorando nosso programa, sabemos que a maioria dos usuários não digitará argumentos na barra de endereços. Em vez disso, os programadores contam com o preenchimento de formulários em páginas da web pelos usuários. Portanto, podemos modificá-lo `index.html` da seguinte forma:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
```

```
        <title>hello</title>
    </head>

    <body>
        <form action="/greet" method="get">
            <input autocomplete="off" autofocus name="name" placeholder="Name"
            <button type="submit">Greet</button>
        </form>
    </body>

</html>
```

Observe que agora é criado um formulário que recebe o nome do usuário e o passa para uma rota chamada `/greet`. `autocomplete` está desativado. Além disso, um `placeholder` com o texto `name` é incluído. Observe também como a `meta` tag é usada para tornar a página da web responsiva para dispositivos móveis.

- Além disso, podemos alterar `app.py` da seguinte forma:

```
# Adds a form, second route

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/greet")
def greet():
    return render_template("greet.html", name=request.args.get("name", "world"
```

Observe que o caminho padrão exibirá um formulário para o usuário inserir seu nome. A `/greet` rota encaminhará o usuário `name` para essa página da web.

- Para finalizar esta implementação, você precisará de outro modelo `greet.html` na `templates` pasta, conforme mostrado a seguir:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>hello</title>
    </head>

    <body>
        hello, {{ name }}
    </body>

</html>
```

Observe que esta rota agora exibirá a saudação ao usuário, seguida de seu nome.

## Modelos

- Nossas duas páginas web, `index.html`a` e `greet.html`b`, possuem grande parte dos mesmos dados. Não seria interessante permitir que o conteúdo fosse único, mas que o layout fosse replicado de uma página para outra?
- Primeiro, crie um novo modelo chamado `layout.html` e escreva o código da seguinte forma:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>hello</title>
    </head>

    <body>
        {% block body %}{% endblock %}
    </body>

</html>
```

  Note que isso `{% block body %}{% endblock %}` permite a inserção de código de outros arquivos HTML.

- Em seguida, modifique o seu `index.html` da seguinte forma:

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

{% block body %}

    <form action="/greet" method="get">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <button type="submit">Greet</button>
    </form>

{% endblock %}
```

  Observe que a linha `{% extends "layout.html" %}` indica ao servidor onde obter o layout desta página. Em seguida, `{% block body %}{% endblock %}` indica qual código deve ser inserido em `layout.html`.

- Por fim, altere `greet.html` da seguinte forma:

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

{% block body %}
    hello, {{ name }}
{% endblock %}
```

Observe como este código é mais curto e compacto.

## Métodos de solicitação

- É possível imaginar cenários em que não seja seguro utilizar o recurso `get`, pois nomes de usuário e senhas apareceriam na URL.
- Podemos utilizar o método `post` para ajudar a resolver esse problema, modificando-`app.py` o da seguinte forma:

```
# Switches to POST

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/greet", methods=["POST"])
def greet():
    return render_template("greet.html", name=request.form.get("name", "world"
```

  Observe que `POST` foi adicionado à `/greet` rota e que usamos `request.form.get` em vez de `request.args.get`.
- Isso instrui o servidor a examinar *mais profundamente* o envelope virtual e a não revelar os itens presentes `post` na URL.
- Ainda assim, este código pode ser aprimorado utilizando uma única rota para ambos, ` `get` a` e `b` `post`. Para isso, modifique o código `app.py` da seguinte forma:

```
# Uses a single route

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        return render_template("greet.html", name=request.form.get("name", "wo
    return render_template("index.html")
```

  Note que ambos os métodos `route` `get` e `post` `route_name` são executados em um único roteamento. No entanto, o método `route_name` `request.method` é utilizado para rotear corretamente com base no tipo de roteamento solicitado pelo usuário.
- Assim sendo, você pode modificar o seu `index.html` da seguinte forma:

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

{% block body %}

    <form action="/" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <button type="submit">Greet</button>
    </form>

{% endblock %}
```

Observe que o formulário `action` foi alterado.

- Ainda assim, existe um erro neste código. Com a nossa nova implementação, quando alguém digita "nenhum nome" no formulário, o `Hello,` campo é exibido sem nome. Podemos melhorar o código editando-o `app.py` da seguinte forma:

```
# Moves default value to template

from flask import Flask, render_template, request

app = Flask(__name__)


@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        return render_template("greet.html", name=request.form.get("name"))
    return render_template("index.html")
```

Observe que `name=request.form.get("name"))` houve uma alteração.

- Por fim, altere `greet.html` da seguinte forma:

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

{% block body %}

    hello,
    {% if name -%}
        {{ name }}
    {%- else -%}
        world
    {%- endif %}

{% endblock %}
```

Observe como `hello, {{ name }}` foi alterado para permitir uma saída padrão quando nenhum nome for identificado.

- Como alteramos muitos arquivos, talvez você queira comparar seu código final com o nosso (https://cdn.cs50.net/2024/fall/lectures/9/src9/hello10/) .

## Calouros IMs

- Frosh IMs ou *froshims* é um aplicativo web que permite aos alunos se inscreverem em esportes intramuros.
- Feche todas as `hello` janelas relacionadas e crie uma pasta digitando `cd` `mkdir froshims` na janela do terminal. Em seguida, digite `cd` `cd froshims` para navegar até essa pasta. Dentro dela, crie um diretório chamado `templates` digitando `cd` `mkdir templates`.
- Em seguida, na `froshims` pasta, digite `code requirements.txt` o código da seguinte forma:

```
Flask
```

  Assim como antes, o Flask é necessário para executar uma aplicação Flask.

- Por fim, digite `code app.py` e escreva o código da seguinte forma:

```
# Implements a registration form using a select menu, validating sport server-

from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/register", methods=["POST"])
def register():

    # Validate submission
    if not request.form.get("name") or request.form.get("sport") not in SPORTS
        return render_template("failure.html")

    # Confirm registration
    return render_template("success.html")
```

  Observe que `failure` existe uma opção para exibir uma mensagem de erro ao usuário caso o campo `name` "ou" `sport` não esteja preenchido corretamente.

- Em seguida, crie um arquivo na `templates` pasta `index.html` com o nome especificado `code templates/index.html` e escreva o código da seguinte forma:

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

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
```

```
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <select name="sport">
            <option disabled selected value="">Sport</option>
            {% for sport in sports %}
                <option value="{{ sport }}">{{ sport }}</option>
            {% endfor %}
        </select>
        <button type="submit">Register</button>
    </form>
{% endblock %}
```

- Em seguida, crie um arquivo digitando o nome `layout.html` e `code templates/layout.html` escreva o código da seguinte forma:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>froshims</title>
    </head>

    <body>
        {% block body %}{% endblock %}
    </body>

</html>
```

- Quarto, crie um arquivo em modelos com `success.html` o seguinte nome:

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

{% block body %}
    You are registered!
{% endblock %}
```

- Por fim, crie um arquivo nos modelos com `failure.html` o seguinte nome:

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

{% block body %}
    You are not registered!
{% endblock %}
```

- Nesta etapa, execute `flask run` e verifique a aplicação.
- Você pode imaginar como gostaríamos de visualizar as diversas opções de inscrição usando botões de opção. Podemos melhorar isso `index.html` da seguinte forma:

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

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        {% for sport in sports %}
```

```
            <input name="sport" type="radio" value="{{ sport }}"> {{ sport }}
        {% endfor %}
        <button type="submit">Register</button>
    </form>
{% endblock %}
```

Observe como `type` foi alterado para `radio`.

- Novamente, ao executar o comando, `flask run` você poderá ver como a interface mudou.
- Você pode imaginar como gostaríamos de aceitar o cadastro de diversos tipos de inscritos. Podemos melhorar isso `app.py` da seguinte forma:

```
# Implements a registration form, storing registrants in a dictionary, with er

from flask import Flask, redirect, render_template, request

app = Flask(__name__)

REGISTRANTS = {}

SPORTS = [
    "Basketball",
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/register", methods=["POST"])
def register():

    # Validate name
    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")

    # Validate sport
    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    # Remember registrant
    REGISTRANTS[name] = sport

    # Confirm registration
    return redirect("/registrants")


@app.route("/registrants")
```

```
def registrants():
    return render_template("registrants.html", registrants=REGISTRANTS)
```

Observe que um dicionário chamado `is` `REGISTRANTS` é usado para registrar o que `sport` foi selecionado por ` `REGISTRANTS[name]` . Observe também que ` `registrants=REGISTRANTS` is` passa o dicionário para este modelo.

- Além disso, podemos implementar `error.html` :

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

{% block body %}
    <h1>Error</h1>
    <p>{{ message }}</p>
    <img alt="Grumpy Cat" src="/static/cat.jpg">
{% endblock %}
```

- Além disso, crie um novo modelo `registrants.html` com o seguinte nome:

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

{% block body %}
    <h1>Registrants</h1>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Sport</th>
            </tr>
        </thead>
        <tbody>
            {% for name in registrants %}
                <tr>
                    <td>{{ name }}</td>
                    <td>{{ registrants[name] }}</td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Observe que o `{% for name in registrants %}...{% endfor %}` processo percorrerá cada um dos inscritos. É muito útil poder iterar em uma página web dinâmica!

- Finalmente, crie uma pasta chamada `static` na mesma pasta que `app.py` . Lá, carregue o seguinte arquivo de um gato (https://cdn.cs50.net/2024/fall/lectures/9/src9/froshims4/static/cat.jpg) .
- Execute `flask run` e utilize o aplicativo.
- Você agora tem um aplicativo web! No entanto, existem algumas falhas de segurança! Como tudo é executado no lado do cliente, um invasor poderia alterar o HTML e *invadir* o site. Além disso, esses dados não serão mantidos se o servidor for desligado. Haveria alguma maneira de manter nossos dados mesmo após a reinicialização do servidor?

# Flask e SQL

- Assim como vimos como o Python pode interagir com um banco de dados SQL, podemos combinar o poder do Flask, do Python e do SQL para criar um aplicativo web onde os dados serão persistentes!
- Para implementar isso, você precisará seguir uma série de etapas.
- Primeiro, baixe o seguinte banco de dados SQL (https://cdn.cs50.net/2024/fall/lectures/9/src9/froshims4/froshims.db) para sua `froshims` pasta.
- Execute o comando no terminal `sqlite3 froshims.db` e digite `.schema` para ver o conteúdo do arquivo de banco de dados. Em seguida, digite `SELECT * FROM registrants;` para saber mais sobre o conteúdo. Você notará que atualmente não há registros no arquivo.
- Em seguida, modifique `requirements.txt` da seguinte forma:

```
cs50
Flask
```

- Modifique `index.html` da seguinte forma:

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

{% block body %}
    <h1>Register</h1>
    <form action="/register" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        {% for sport in sports %}
            <input name="sport" type="checkbox" value="{{ sport }}"> {{ sport
        {% endfor %}
        <button type="submit">Register</button>
    </form>
{% endblock %}
```

- Modifique `layout.html` da seguinte forma:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>froshims</title>
    </head>

    <body>
        {% block body %}{% endblock %}
    </body>

</html>
```

- O Garantir `error.html` que apareça é o seguinte:

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

{% block body %}
    <h1>Error</h1>
    <p>{{ message }}</p>
    <img alt="Grumpy Cat" src="/static/cat.jpg">
{% endblock %}
```

- Modifique `registrants.html` para que fique da seguinte forma:

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

{% block body %}
    <h1>Registrants</h1>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Sport</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            {% for registrant in registrants %}
                <tr>
                    <td>{{ registrant.name }}</td>
                    <td>{{ registrant.sport }}</td>
                    <td>
                        <form action="/deregister" method="post">
                            <input name="id" type="hidden" value="{{ registran
                            <button type="submit">Deregister</button>
                        </form>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Observe que um valor oculto `registrant.id` está incluído para que seja possível utilizá-lo `id` posteriormente em `app.py`

- Por fim, modifique `app.py` da seguinte forma:

```
# Implements a registration form, storing registrants in a SQLite database, wi

from cs50 import SQL
from flask import Flask, redirect, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///froshims.db")

SPORTS = [
    "Basketball",
```

```
    "Soccer",
    "Ultimate Frisbee"
]


@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)


@app.route("/deregister", methods=["POST"])
def deregister():

    # Forget registrant
    id = request.form.get("id")
    if id:
        db.execute("DELETE FROM registrants WHERE id = ?", id)
    return redirect("/registrants")


@app.route("/register", methods=["POST"])
def register():

    # Validate name
    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")

    # Validate sports
    sports = request.form.getlist("sport")
    if not sports:
        return render_template("error.html", message="Missing sport")
    for sport in sports:
        if sport not in SPORTS:
            return render_template("error.html", message="Invalid sport")

    # Remember registrant
    for sport in sports:
        db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", name,

    # Confirm registration
    return redirect("/registrants")


@app.route("/registrants")
def registrants():
    registrants = db.execute("SELECT * FROM registrants")
    return render_template("registrants.html", registrants=registrants)
```

Observe que a `cs50` biblioteca está sendo utilizada. Uma rota está incluída para `register` o `post` método. Essa rota receberá o nome e o esporte obtidos do formulário de inscrição e executará uma consulta SQL para adicionar o nome `name` e o esporte `sport` à `registrants` tabela. As `deregister` rotas levam a uma consulta SQL que obterá o ID do usuário `id` e utilizará essas informações para cancelar o registro desse indivíduo.

- Você pode executar o comando `flask run` e examinar o resultado.

- Se você quiser baixar nossa implementação, `froshims` pode fazê-lo aqui (https://cdn.cs50.net/2024/fall/lectures/9/src9/froshims5/) .
- Você pode ler mais sobre o Flask na documentação do Flask (https://flask.palletsprojects.com) .

## Cookies e sessão

- `app.py` A *visão* é considerada o que os usuários veem. O *modelo é* como os dados são armazenados e manipulados. Juntos, esses elementos são chamados de *MVC* (modelo, visão, controlador).
- Embora a implementação anterior `froshims` seja útil do ponto de vista administrativo, permitindo que um administrador de back-office adicione e remova indivíduos do banco de dados, é fácil imaginar como esse código não é seguro para implementar em um servidor público.
- Por um lado, pessoas mal-intencionadas poderiam tomar decisões em nome de outros usuários clicando no botão de cancelamento de registro – apagando, efetivamente, a resposta registrada deles do servidor.
- Serviços da web como o Google usam credenciais de login para garantir que os usuários tenham acesso apenas aos dados corretos.
- Podemos implementar isso nós mesmos usando *cookies* . Cookies são pequenos arquivos armazenados no seu computador que permitem que ele se comunique com o servidor e diga: "Sou um usuário autorizado que já fez login". Essa autorização por meio do cookie é chamada de *sessão* .
- Os cookies podem ser armazenados da seguinte forma:

  ```
  GET / HTTP/2
  Host: accounts.google.com
  Cookie: session=value
  ```

  Aqui, um `session` ID é armazenado com um identificador específico `value` que representa essa sessão.
- Na forma mais simples, podemos implementar isso criando uma pasta chamada `login` e adicionando os seguintes arquivos.
- Primeiro, crie um arquivo com `requirements.txt` o seguinte conteúdo:

  ```
  Flask
  Flask-Session
  ```

  Observe que, além de `Flask` , também incluímos `Flask-Session` , que é necessário para dar suporte às sessões de login.
- Em segundo lugar, em uma `templates` pasta, crie um arquivo `layout.html` com o seguinte formato:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>login</title>
    </head>

    <body>
        {% block body %}{% endblock %}
    </body>

</html>
```

Observe que isso proporciona um layout muito simples, com um título e um corpo de texto.

- Terceiro, crie um arquivo na `templates` pasta `index.html` com o seguinte nome:

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

{% block body %}

    {% if name -%}
        You are logged in as {{ name }}. <a href="/logout">Log out</a>.
    {%- else -%}
        You are not logged in. <a href="/login">Log in</a>.
    {%- endif %}

{% endblock %}
```

Observe que este arquivo verifica se `session["name"]` ele existe (explicado mais detalhadamente `app.py` abaixo). Se existir, exibirá uma mensagem de boas-vindas. Caso contrário, recomendará que você acesse uma página para fazer login.

- Quarto, crie um arquivo chamado `login.html` e adicione o seguinte código:

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

{% block body %}

    <form action="/login" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" typ
        <button type="submit">Log In</button>
    </form>

{% endblock %}
```

Observe que este é o layout de uma página de login básica.

- Por fim, crie um arquivo chamado `app.py` e escreva o código da seguinte forma:

```
from flask import Flask, redirect, render_template, request, session
from flask_session import Session


```

```
# Configure app
app = Flask(__name__)

# Configure session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)


@app.route("/")
def index():
    return render_template("index.html", name=session.get("name"))


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        session["name"] = request.form.get("name")
        return redirect("/")
    return render_template("login.html")


@app.route("/logout")
def logout():
    session.clear()
    return redirect("/")
```

Observe as *importações* modificadas no início do arquivo, incluindo `<session>` `session`, que permitirão o suporte a sessões. Mais importante ainda, observe como `<session>` `session["name"]` é usado nas rotas ` <session> `login` ` e ` <session>`. A rota `<session>` atribuirá o nome de login fornecido e o atribuirá a `<session>` . No entanto, na rota `<session>`, o logout é implementado limpando o valor de `<session>` `. `logout` `login` `session["name"]` `logout` `session`

- A `session` abstração permite garantir que apenas um usuário específico tenha acesso a dados e recursos específicos em nosso aplicativo. Ela permite assegurar que ninguém aja em nome de outro usuário, seja para o bem ou para o mal!
- Se desejar, você pode baixar nossa implementação (https://cdn.cs50.net/2024/fall/lectures/9/src9/login/) de `login`.
- Você pode ler mais sobre sessões na documentação do Flask (https://flask.palletsprojects.com/en/stable/api/#flask.session) .

## Carrinho de compras

- Passando para um exemplo final de utilização da capacidade do Flask de habilitar uma sessão.
- Examinamos o seguinte código para `store` . `app.py` O código exibido foi o seguinte:

```
from cs50 import SQL
from flask import Flask, redirect, render_template, request, session
from flask_session import Session
```

```
# Configure app
app = Flask(__name__)

# Connect to database
db = SQL("sqlite:///store.db")

# Configure session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)


@app.route("/")
def index():
    books = db.execute("SELECT * FROM books")
    return render_template("books.html", books=books)


@app.route("/cart", methods=["GET", "POST"])
def cart():

    # Ensure cart exists
    if "cart" not in session:
        session["cart"] = []

    # POST
    if request.method == "POST":
        book_id = request.form.get("id")
        if book_id:
            session["cart"].append(book_id)
        return redirect("/cart")

    # GET
    books = db.execute("SELECT * FROM books WHERE id IN (?)", session["cart"])
    return render_template("cart.html", books=books)
```

Observe que isso `cart` é implementado usando uma lista. Itens podem ser adicionados a esta lista usando os `Add to Cart` botões em `books.html`. Ao clicar em um desses botões, o `post` método é invocado, onde o ID `id` do item é anexado ao `cart`. Ao visualizar o carrinho, invocando o `get` método, o SQL é executado para exibir uma lista dos livros no carrinho.

- Também vimos o conteúdo de `books.html`:

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

{% block body %}

    <h1>Books</h1>
    {% for book in books %}
        <h2>{{ book["title"] }}</h2>
        <form action="/cart" method="post">
            <input name="id" type="hidden" value="{{ book['id'] }}">
            <button type="submit">Add to Cart</button>
        </form>
    {% endfor %}
```

```
{% endblock %}
```

Observe como isso cria a possibilidade de `Add to Cart` usar para cada livro `for book in books`.

- Você pode ver o restante dos arquivos que dão suporte a essa `flask` implementação no código-fonte (https://cdn.cs50.net/2024/fall/lectures/9/src9/store/) .

## Shows

- Analisamos um programa pré-desenhado chamado `shows`, em `app.py`:

```
# Searches for shows using LIKE

from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + request
    return render_template("search.html", shows=shows)
```

Observe como a `search` rota permite uma forma de pesquisar um título `show`. Essa pesquisa procura por títulos `LIKE` diferentes daquele fornecido pelo usuário.

- Também analisamos `index.html`:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>shows</title>
    </head>

    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="text">

        <ul></ul>

        <script>
            let input = document.querySelector('input');
```

```
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
                let shows = await response.json();
                let html = '';
                for (let id in shows) {
                    let title = shows[id].title.replace('<', '&lt;').replace('
                    html += '<li>' + title + '</li>';
                }
                document.querySelector('ul').innerHTML = html;
            });
        </script>

    </body>

</html>
```

Observe que o JavaScript `script` cria uma implementação de autocompletar, onde os títulos que correspondem à pesquisa `input` são exibidos.

- Você pode ver o restante dos arquivos desta implementação no código-fonte (https://cdn.cs50.net/2024/fall/lectures/9/src9/shows3/) .

## APIs

- Uma *interface de programação de aplicativos* , ou *API,* é um conjunto de especificações que permite a interação com outro serviço. Por exemplo, podemos utilizar a API do IMDB para interagir com o banco de dados deles. Podemos até integrar APIs para lidar com tipos específicos de dados que podem ser baixados de um servidor.
- Aprimorando o modelo anterior `shows` , e observando uma melhoria em relação a ele `app.py` , constatamos o seguinte:

```
# Searches for shows using Ajax

from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/search")
def search():
    q = request.args.get("q")
    if q:
        shows = db.execute("SELECT * FROM shows WHERE title LIKE ? LIMIT 50",
    else:
```

```
        shows = []
    return render_template("search.html", shows=shows)
```

Observe que a `search` rota executa uma consulta SQL.

- Ao observar `search.html`, você perceberá que é muito simples:

```
{% for show in shows %}
    <li>{{ show["title"] }}</li>
{% endfor %}
```

Observe que ela fornece uma lista com marcadores.

- Por fim, `index.html` observe que o código *AJAX* é utilizado para realizar a pesquisa:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>shows</title>
    </head>

    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="search">

        <ul></ul>

        <script>
            let input = document.querySelector('input');
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
                let shows = await response.text();
                document.querySelector('ul').innerHTML = shows;
            });
        </script>

    </body>

</html>
```

Observe que um ouvinte de eventos é utilizado para consultar dinamicamente o servidor e obter uma lista que corresponda ao título fornecido. Isso localizará a `ul` tag no HTML e modificará a página da web de acordo para incluir a lista de correspondências.

- Você pode ler mais na documentação do AJAX (https://api.jquery.com/category/ajax/) .

## JSON

- *JavaScript Object Notation* , ou *JSON,* é um arquivo de texto contendo dicionários com chaves e valores. É uma forma simples e fácil de usar computacionalmente para obter grandes quantidades de dados.

- JSON é uma forma muito útil de obter dados do servidor.
- Você pode ver isso em ação no `index.html` que examinamos juntos:

```
<!DOCTYPE html>

<html lang="en">

    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>shows</title>
    </head>

    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="text">

        <ul></ul>

        <script>
            let input = document.querySelector('input');
            input.addEventListener('input', async function() {
                let response = await fetch('/search?q=' + input.value);
                let shows = await response.json();
                let html = '';
                for (let id in shows) {
                    let title = shows[id].title.replace('<', '&lt;').replace('
                    html += '<li>' + title + '</li>';
                }
                document.querySelector('ul').innerHTML = html;
            });
        </script>

    </body>

</html>
```

Embora o texto acima possa parecer um tanto enigmático, ele fornece um ponto de partida para você pesquisar sobre JSON por conta própria e descobrir como ele pode ser implementado em suas próprias aplicações web.

- Além disso, analisamos `app.py` como a resposta JSON é obtida:

```
# Searches for shows using Ajax with JSON

from cs50 import SQL
from flask import Flask, jsonify, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


@app.route("/")
def index():
    return render_template("index.html")
```

```
@app.route("/search")
def search():
    q = request.args.get("q")
    if q:
        shows = db.execute("SELECT * FROM shows WHERE title LIKE ? LIMIT 50",
    else:
        shows = []
    return jsonify(shows)
```

Observe como o `jsonify` comando é usado para converter o resultado em um formato legível e aceitável pelas aplicações web contemporâneas.

- Você pode ler mais na documentação JSON (https://www.json.org/json-en.html) .
- Em resumo, agora você tem a capacidade de desenvolver seus próprios aplicativos web usando Python, Flask, HTML e SQL.

## Resumindo

Nesta lição, você aprendeu como utilizar Python, SQL e Flask para criar aplicações web. Especi

- Frasco
- Formulários
- Modelos
- Métodos de solicitação
- Flask e SQL
- Cookies e sessão
- APIs
- JSON

Nos vemos na próxima aula, a última deste semestre, no Teatro Sanders (https://websites.harvard.edu/memhall/home-2/buildings/sanders-theatre/) !