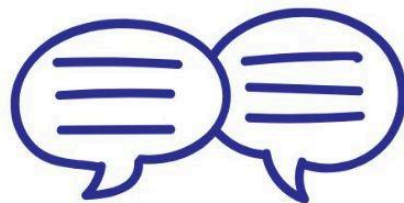




FACULDADE IMPACTA

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DESENVOLVIMENTO WEB



ALEX SOUSA

SÃO PAULO - 04/2024



SUMÁRIO

SUMÁRIO.....	2
Desenvolvimento Web.....	6
Sites de referência.....	6
W3Schools.....	6
MDN Web Docs.....	6
Livros introdutórios.....	6
HTML e CSS.....	6
JavaScript.....	6
Framework Flask.....	6
o que é internet.....	6
Modelo Cliente-Servidor.....	7
Caminho de uma requisição.....	7
Você sabia?.....	8
Estrutura de uma requisição.....	9
Estrutura da URL.....	9
Métodos HTTP.....	10
Cabeçalhos.....	10
Estrutura de uma resposta.....	11
Códigos de status.....	11
Cabeçalhos de resposta.....	11
Corpo da resposta.....	12
HTML.....	12
Sintaxe básica.....	12
Atributos HTML.....	14
Tags aninhadas.....	14
Estrutura básica do documento HTML.....	14
Tipos de tags do HTML.....	15
Tags de configuração.....	15
Tags de marcação de texto.....	16
Tags de estrutura de texto.....	17
Tags de estrutura de documento.....	18
Tags de multimídia.....	18
Você conhece?.....	18
Estrutura básica do documento HTML.....	19
Semântica HTML.....	20
Você quer ler?.....	20
Formulários no HTML.....	20
Coleta de dados do usuário.....	20
Atributo enctype.....	21
Atribuição action.....	21
Atribuição method.....	22
Campos de formulário.....	22
Campos de coleta de texto.....	22

Campos de coleta de opções.....	23
Botões de controle.....	26
Semântica nos formulários.....	27
Validação em formulários.....	27
Aprenda Mais Sobre Form.....	28
Introdução ao CSS.....	28
O que é a linguagem CSS?.....	28
Como usar o CSS.....	28
Sintaxe básica do CSS.....	28
Propriedades básicas do CSS.....	30
Cores.....	31
Seletores CSS.....	32
Seletores por tipo.....	32
Seletores por ID.....	32
Seletores por classe.....	32
Seletores de atributos.....	34
Seletores de agrupamento.....	34
Seletores de posição relativa.....	35
Pseudo-classes e pseudo-elementos.....	36
Vamos praticar?.....	37
Priorização de regras.....	37
Layouts no CSS.....	39
Dimensões e unidades.....	39
Porcentagem.....	39
em e rem.....	40
BoxModel.....	40
Display.....	42
Você quer ler?.....	42
Position.....	42
Media Query.....	43
JavaScript.....	44
Características do JavaScript.....	45
Executando o primeiro código JavaScript.....	45
Regras de sintaxe.....	46
Declaração de variáveis.....	46
Declaração de variáveis.....	47
Operações.....	47
Array.....	48
Object.....	49
Funções.....	50
Estrutura condicional: if-else.....	51
Estruturas de repetição: while, for, do-while.....	52
Você sabia?.....	53
Manipulação do DOM.....	53
O que é o DOM?.....	53

Navegando pelo DOM com o JS.....	54
getElementById.....	54
getElementsByClassName.....	54
getElementsByTagName.....	55
querySelector.....	55
querySelectorAll.....	55
Execução do JS.....	56
async e defer.....	56
Manipulando o DOM com o JS.....	57
Acessando e alterando propriedades.....	57
Acessando e alterando estilos através do DOM.....	58
Acessando e alterando classes através do DOM.....	59
Alterando elementos internos.....	60
Você sabia?.....	61
Eventos em JavaScript.....	62
Escutando eventos com JS.....	62
Principais eventos em JS.....	62
Associação de eventos em JS.....	63
Associação pelos atributos on* com funções nomeadas.....	63
Associação pelos atributos on* com funções anônimas.....	64
Associação com o método addEventListener() e funções nomeadas.....	65
Associação com o método addEventListener() e funções anônimas.....	66
O objeto event.....	67
Você conhece?.....	68
Padrão MVC e Programação no Servidor.....	68
Padrões de Projeto.....	69
Padrão MVC.....	70
Flask Framework.....	70
Pré Requisitos.....	71
HelloWorld: a primeira aplicação MVC no Flask.....	71
Implementando o MVC.....	73
Parametrização de Rotas.....	73
Vamos praticar?.....	74
Server Side Redering: Criação de Páginas Dinâmicas.....	74
Técnicas de Renderização Dinâmica.....	75
Você quer ler?.....	76
Renderização no Servidor com o Flask.....	76
Como usar no Flask.....	76
Imprimindo valores no template.....	76
Estruturas de Controle.....	77
Herança de Templates.....	77
Arquivos Estáticos.....	79
Controle de Sessão - Autenticação e Autorização.....	79
Fluxo de Autenticação.....	80
Envio de dados de autenticação.....	80

Recebendo valores por GET e POST.....	80
Definindo a sessão do usuário.....	81
Controle de Sessão.....	82
Exibindo uma página privada ao usuário.....	82
Expiração do cookie de sessão.....	83
Encerramento da sessão.....	83
Você quer ler?.....	83
Aprimorando o nosso projeto com banco de dados.....	84
Armazenando senhas.....	85
Manipulando a sessão.....	85
Proteção dos controles.....	86
AJAX.....	86
Síncrono vs Assíncrono.....	86
AJAX - Asynchronous JavaScript and XML.....	87
Os objetivos XMLHttpRequest e fetch.....	87
XMLHttpRequest.....	87
fetch API.....	88
Vamos praticar?.....	89
O formato JSON.....	89

Desenvolvimento Web

Sites de referência

W3Schools

W3Schools é um site educacional voltado ao aprendizado de tecnologias Web (HTML, CSS, JavaScript) e outras linguagens de programação. Pode ser usado tanto para consultas de tags e atributos HTML, seletores e propriedades CSS, etc. Além disso possui exercícios e vários exemplos interativos;

<https://www.w3schools.com/>

<https://www.w3schools.com/exercises/>

MDN Web Docs

MDN Web Docs é uma fonte de documentação para desenvolvedores, mantida com o apoio de uma comunidade de desenvolvedores e escritores técnicos, além de hospedar muitos documentos sobre uma grande variedade de assuntos como: HTML, CSS, JavaScript, etc.

<https://developer.mozilla.org/pt-BR/>

Livros introdutórios

HTML e CSS

FREEMAN, E.; FREEMAN, E. Use a cabeça! HTML com CSS e XHTML. 2. ed. Rio de Janeiro: Alta Books, 2015.

JavaScript

DUCKETT, J. Javascript e JQuery: desenvolvimento de interfaces web interativas. 1.ed. Rio de Janeiro: Altabooks, 2016.

Framework Flask

GRINBERG, M. Flask web development: developing web applications with python. 2. ed. O'Reilly Media, 2018.

o que é internet

Grande rede de computadores interligadas, redes de outras redes. vários dispositivos de redes interligados.

Para que esta rede se comunique precisa de um protocolo de comunicação conjunto de regras para troca de informações.

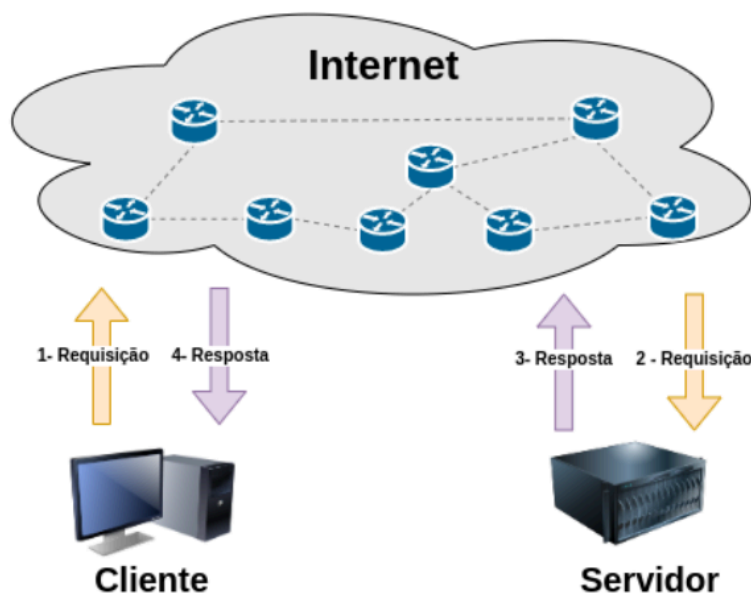
A Internet é formada por diversas redes de computadores conectadas entre si, trocando informações de todo tipo a todo instante. De uma simples mensagem de e-mail até o streaming de um vídeo, temos computadores (além de outros dispositivos conectados) trocando mensagens de forma padronizada e organizada.

Mas como essa troca se dá? Quais tecnologias estão envolvidas? Sabemos que tudo começa no navegador de Internet, o software que todos os computadores e dispositivos móveis possuem para entrar nos sites da Internet. Entender as partes envolvidas nesse processo, mesmo que de forma mais superficial, nos permite ter uma compreensão muito importante para o desenvolvimento de aplicações na web, seja qual for o foco desta aplicação.

Modelo Cliente-Servidor

Em termos gerais, essa troca de mensagens é feita sempre através de um aparelho que requisita (solicita) alguma informação e outro que responde a essa solicitação. Essa forma de comunicação, onde um ponto requisita uma informação e outro responde com ela é representada pelo modelo cliente-servidor [Maia, 2013], conforme pode ser visto na Figura 1.1.

Figura 1.1. Modelo Cliente-Servidor



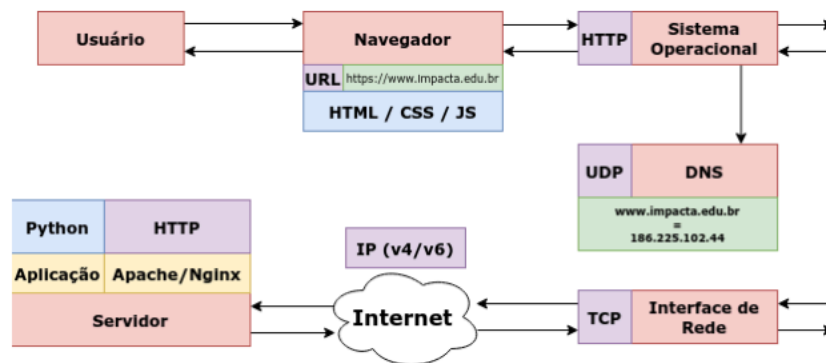
O modelo cliente-servidor abrange uma variedade de sistemas além da web, como por exemplo os caixas eletrônicos, pontos de venda em lojas, sistemas de e-mails, etc. Na Internet, costumeiramente, o cliente será o navegador de Internet e o servidor algum computador que hospeda páginas e outros recursos (imagens, vídeos, arquivos Word ou PDF, arquivos com código CSS, etc) que podem ser solicitados pelo navegador (cliente).

Caminho de uma requisição

Ao digitarmos o endereço do site que queremos acessar no navegador, a requisição feita por ele passa por diversos componentes, tecnologias e protocolos antes de podermos ver o resultado. Entender o papel de alguns destes pedaços ajuda a perceber a complexidade envolvida na construção da web e a construir aplicações com foco no funcionamento do sistema completo.

Vamos simplificar o caminho com os conceitos mais importantes para o desenvolvimento web, mostrados no diagrama da Figura 1.2.

Figura 1.2. Caminho da requisição



Fonte: do autor, 2021.

Tudo começa com o usuário manipulando o navegador para pedir algum recurso (em geral uma página web). Para isso, ele digita o endereço do recurso que ele está procurando na barra de endereço do navegador, usando o formato da URL. Com isso, o navegador empacota o pedido para o sistema operacional, usando o protocolo HTTP como o pacote da mensagem.

Caso o navegador não saiba o IP (Internet Protocol) do domínio digitado (se for a primeira vez que visita o site, por exemplo), o sistema operacional pergunta ao servidor de DNS (Domain Name System), que é o serviço responsável por traduzir domínios para IPs públicos na Internet. Essa conexão tradicionalmente é feita usando o protocolo UDP (User Datagram Protocol) que é um protocolo de transporte de mensagens característico por ser mais rápido por não checar se a informação chegou ou não ao destinatário. O IP é um protocolo de endereçamento na Internet, formado por números de 32 bits (IPv4) ou 128 bits (IPv6).

Após descobrir o IP público do servidor que possui o site pedido, o sistema operacional passa a mensagem para a interface de rede que irá estabelecer uma conexão com o servidor que possui o IP solicitado. A partir daqui, esses dados trafegam pela Internet usando protocolo TCP (Transfer Control Protocol) que, ao contrário do UDP, sempre verifica se a mensagem de fato chegou ao destinatário, fazendo com que seja um protocolo mais robusto (mas com uma perda no desempenho quando comparado com o UDP).

Com o IP e a conexão TCP estabelecida, a requisição HTTP chega ao servidor. Lá, ela precisa ser desempacotada por algum programa que entenda o protocolo HTTP, nesse caso chamado de servidor de aplicação HTTP, onde os exemplos mais famosos são o Apache, Nginx (pronuncia-se “engine-X”) e o Microsoft IIS. Na maioria dos casos esses softwares apenas traduzem a mensagem e repassam para alguma outra aplicação, que irá executar as ações necessárias para retornar a resposta para o cliente.

Com a resposta já formada, o servidor de aplicação HTTP a empacota para o retorno ao cliente (passando por todos os pontos novamente). Chegando no navegador (cliente), ele usa seus recursos para interpretar a resposta e mostrar o resultado ao usuário. Em geral, esses recursos são escritos usando as linguagens que o navegador consegue interpretar, como por exemplo o HTML, CSS e JavaScript.

Você sabia?

A Internet é o local com a maior quantidade de conhecimento aberto do mundo, mas nem sempre foi assim. O conceito foi criado pelo Departamento de Defesa dos EUA, no contexto da Guerra Fria ainda (década de 1960). A ideia era transferir documentos secretos entre pontos estratégicos, descentralizando as informações entre

vários locais. A subdivisão deste departamento que criou o conceito foi a ARPA (Advanced Research Projects Agency), por isso a primeira versão se chamava ARPANET (ESCOLA, s.d.).

Estrutura de uma requisição

A requisição no HTTP é formada basicamente por três grandes partes: a URL, o método HTTP e os cabeçalhos de requisição. Algumas requisições podem possuir um corpo (payload), que abriga dados adicionais enviados ao servidor.

Estrutura da URL

A URL (Uniform Resource Locator) é um sistema de endereçamento de recursos para web. Ela traz informações de localização do servidor na web, do recurso dentro do servidor, e qual protocolo deve ser utilizado para trazer o recurso. A sua estrutura básica e simplificada é:

esquema://domínio:porta/caminho/recurso?query_string#fragmento

O esquema mostra qual protocolo será utilizado na transmissão da mensagem. Dentre os mais comuns, temos como exemplo: http, ftp, smtp, etc.

O domínio é o conjunto de nomes e identificadores mais amigáveis aos humanos para computadores pela Internet. Um domínio em geral corresponde apenas a um endereço IP, embora existam técnicas para permitir múltiplos endereços. Tradicionalmente chamamos de domínio a parte do endereço que compõe o nome do negócio representado (ex: impacta) e os identificadores de negócio e local (.edu para educação e .br para sites no Brasil). Qualquer prefixo na frente do domínio principal (impacta.edu.br) é chamado de subdomínio. Por exemplo: www.impacta.edu.br e account.impacta.edu.br são subdomínios de impacta.edu.br.

A porta é um número inteiro que identifica o caminho lógico por onde uma comunicação de rede está passando. Toda comunicação de rede (e processos nos computadores) passam por uma porta lógica. Por padrão, o HTTP sempre utiliza a porta 80 (ou 443 para o HTTPS, uma versão do HTTP que utiliza criptografia). No navegador, sempre que se utiliza o HTTP e o HTTPS não é necessário escrever suas portas padrão, pois nesses casos ele já utilizará as portas 80 ou 443, implicitamente.

O caminho (também conhecido como path) identifica onde o recurso está dentro do servidor. Este caminho pode ser tanto físico (indicando pastas no servidor) quanto lógico (caminhos configurados dentro da aplicação para encontrar um recurso).

O recurso é o arquivo que se deseja acessar. Nem sempre essa parte da URL vai existir, pois alguns recursos são dinâmicos, ou seja, são gerados em tempo de execução da requisição (ex: lista de chamada de uma turma no dia). Quando o recurso for estático (ex: imagens) essa parte existirá.

A query string é uma forma de passar dados a mais para o servidor, como forma de ajudar na busca de recursos mais específicos. É bastante usada nos buscadores e possui o formato atributo=valor. Essa parte da URL é análoga ao envio de valores através de parâmetros para uma função em uma linguagem de programação, isto é, podemos enviar valores quaisquer para o servidor através da URL.

O fragmento identifica uma parte específica da página que está sendo procurada, em geral um id (identificador único) de algum elemento HTML para que o navegador já entre visualizando o elemento específico.

Métodos HTTP

Os métodos HTTP (ou verbos HTTP) são informações que servem para indicar uma intenção do que pode ocorrer no servidor ao enviar a requisição. Alguns métodos são usados apenas para obtenção de recursos (ex: GET), outros são específicos para alteração do estado da aplicação (ex: PUT, DELETE, POST) (HTTP request methods, s.d.).

Dentre os métodos HTTP mais comuns, destacamos:

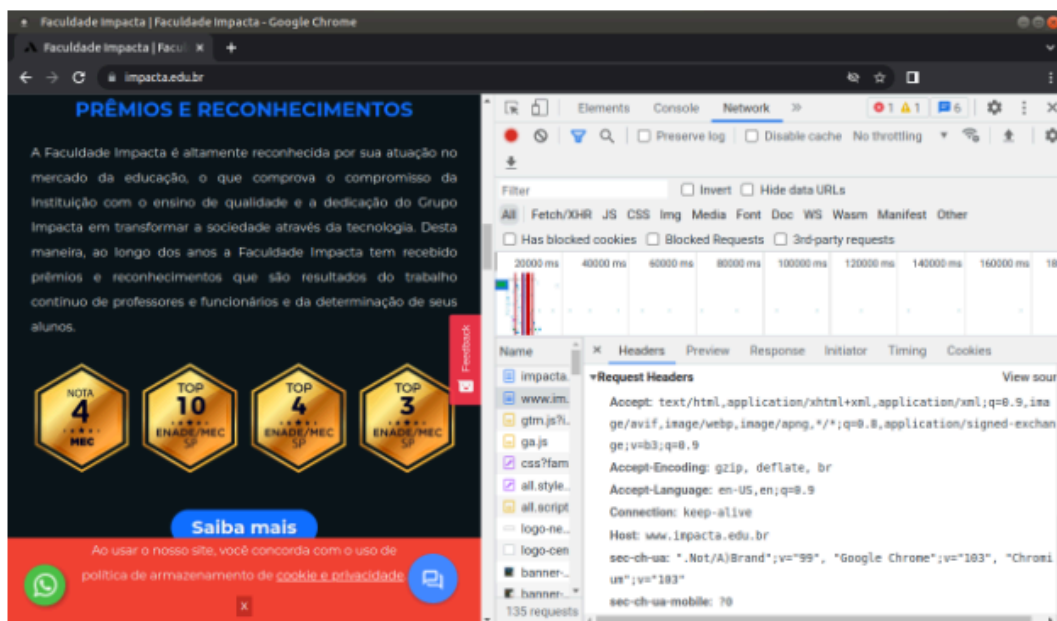
- GET: usado para obter um recurso qualquer do servidor;
- POST: envia dados para serem processados pelo servidor, em geral para criar ou alterar um recurso;
- DELETE: remove um determinado recurso do servidor;
- PUT: atualiza todas as informações de um recurso no servidor;
- PATCH: atualiza parte das informações de um recurso no servidor;

Cabeçalhos

Os cabeçalhos (headers) de requisição são uma série de informações necessárias para a comunicação entre o cliente e o servidor. Essas informações trafegam sempre no formato chave: valor, onde os valores são sempre tratados como dados textuais.

Algumas dessas informações são restritas ao navegador, como por exemplo o user-agent e os cookies. Também é possível criar nossos próprios cabeçalhos com bibliotecas JavaScript para passar informações específicas de nossa aplicação. A Figura 1.3 mostra alguns exemplos de informações contidas no cabeçalho de uma requisição. Essa mesma tela pode ser acessada no menu “Ferramentas do Desenvolvedor”, ou usando o atalho F12 (ou também Control+Shift+I) no navegador Google Chrome.

Figura 1.3. Visualização do cabeçalho de uma requisição usando o navegador Google Chrome



Os cookies são headers mais específicos, pois eles são a única informação que pode sobreviver entre uma requisição e outra. Eles são valores textuais com um nome e o domínio onde eles são aplicados. Toda requisição feita no mesmo domínio carrega todos os cookies registrados nele. O uso dos cookies é uma das maneiras mais clássicas de criar uma sessão de usuário na web. Estudaremos a relação entre sessões de usuário e cookies mais adiante em nosso curso.

Estrutura de uma resposta

As respostas HTTP possuem sempre três partes: código de status, cabeçalhos de resposta e corpo da resposta.

Códigos de status

Os códigos de status indicam se a requisição foi sucedida ou não, e o que aconteceu com ela em ambos os casos. Eles são divididos em números que podem ir de 100 a 599, mas divididos em faixas de 100 de acordo com o seu objetivo. Alguns dos códigos de status mais famosos são o 200 (OK), 404 (Not Found), 400 (Bad Request) e 500 (Internal Server Error).

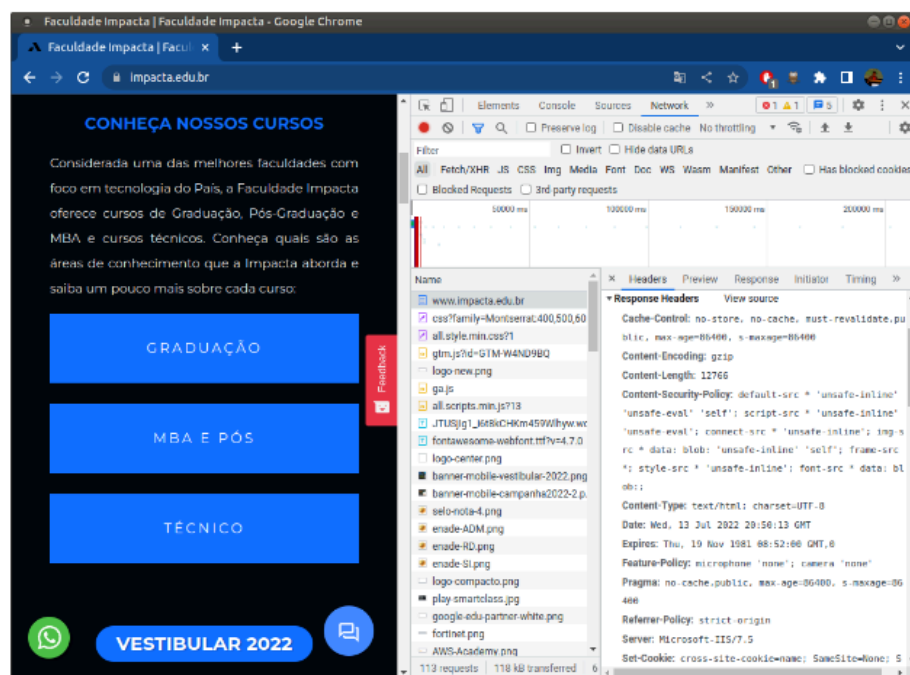
As faixas dos códigos são divididas em (HTTP response status codes, s.d.):

- 100 até 199: códigos informativos. São usados na comunicação do cliente com o servidor com informações intermediárias sobre a comunicação de rede. Requisições com números nessa faixa não são muito comuns no dia a dia;
- 200 até 299: códigos de sucesso. Indicam que a requisição foi bem sucedida no servidor. O sucesso de cada requisição depende muito do método sendo usado;
- 300 até 399: códigos de redirecionamento. Indicam que o navegador precisou tomar uma ação a mais para completar a requisição, como por exemplo, o redirecionamento de páginas, uso do cache, entre outros.
- 400 até 499: códigos de erro do cliente. Mostram que alguma coisa deu errado no cliente (navegador). Situações comuns são: algum problema na comunicação estabelecida pelo navegador, ou falta alguma informação ou condição para a requisição ser concluída (ex: não está logado).
- 500 até 599: códigos de erro do servidor. Indicam que o erro aconteceu no lado do servidor, podendo ser problemas na rede interna ou uma inconsistência na aplicação rodando no servidor.

Cabeçalhos de resposta

Da mesma forma que na requisição, a resposta também possui seus próprios cabeçalhos, no mesmo formato.

Muitos destes cabeçalhos são criados pelo servidor, baseado na resposta. Por exemplo, o cabeçalho Date mostra a data em que a resposta foi gerada e o Server indica o software rodando no servidor. Outros cabeçalhos podem ser criados pela aplicação de acordo com a necessidade dela. Já os cookies são definidos no navegador através do cabeçalho de resposta Set-Cookie. A Figura 1.4 mostra algumas informações disponíveis no cabeçalho de resposta.



Corpo da resposta

O corpo da resposta contempla o recurso que foi requisitado. Esse recurso pode ser qualquer coisa, por exemplo: um documento HTML, uma imagem, um vídeo, um arquivo xml, um arquivo PDF, etc.

HTML

HTML é uma sigla para HyperText Markup Language, que significa linguagem de marcação de hipertexto. Hipertextos são documentos que permitem que o leitor possa ler em uma ordem que ele bem entenda, usando ligações (links) entre os diferentes textos que compõem o hipertexto (HIPERTEXTO, s.d.). Na web, ao adentrar em um site, é possível navegar em diferentes páginas deste site da maneira que quisermos, clicando nos links internos dele. A web é a maior coleção de hipertextos que existe.

O HTML é uma linguagem de marcação, ou seja, é uma linguagem artificial que define um conjunto de sinais e códigos aplicados a um texto ou a dados para definir a sua configuração. A marcação não aparece no trabalho final. Assim, o HTML não é uma linguagem de programação, pela ausência de várias características: variáveis, de estruturas de decisão/repetição, funções, etc. A função do HTML é dar estrutura, significado e semântica para o conteúdo que desejamos mostrar nos sites da web. Podemos usar os marcadores (tags) para indicar um texto como sendo um parágrafo, títulos, citações, tabelas, etc, onde os navegadores usam esses marcadores para mostrar uma página estruturada na tela dos computadores. O navegador é, portanto, o interpretador da linguagem HTML, pois ele interpreta as tags e renderiza (exibe) o conteúdo formatado na tela.

A primeira versão da linguagem HTML foi escrita em 1993 por Tim Berners-Lee, como forma de resolver o problema de compartilhar suas pesquisas com o seu grupo. Desde então o HTML evoluiu bastante, sendo a versão mais atual, o HTML5, uma revolução em termos de semântica e estrutura na web.

Sintaxe básica

O HTML é composto de marcadores, ou mais comumente chamados de tags, para dar significado semântico a um texto. Esse conteúdo tradicionalmente fica em arquivos com extensão .html, que podem ser manipulados em qualquer editor simples de texto.

As marcações seguem uma estrutura específica e sua própria nomenclatura. Todo marcador (tag) começa com o símbolo < (“menor que”), seguido do identificador da tag, e termina com > (“maior que”).

```
<html>
<head>
<body>
<p>
```

Após o marcador de abertura, podemos incluir algum conteúdo textual. Por fim, devemos fechar o marcador de forma semelhante à marcação de abertura, mas adicionando uma barra (/) entre o símbolo < (menor que) e o identificador da tag.

```
</html>
</head>
</body>
</p>
```

Outro detalhe importante é que o HTML não é sensível a maiúsculas ou minúsculas, ou seja, escrever uma determinada tag como <marcador>, <Marcador> ou <MARCADOR> não fará diferença no funcionamento, mas é uma boa prática escrever o identificador da tag com letras minúsculas. Ao contrário do que ocorre na linguagem Python, o navegador não diferencia espaços a mais que existam nos documentos HTML, sejam quebras de linha, tabulações (tabs) ou apenas espaços que não sejam os separadores de palavras.

Algumas tags não possuem o marcador de fechamento, e em geral são utilizadas para inserir algum conteúdo especial ou efeito na página: são tags ditas autocontidas. Um exemplo é a tag de quebra de linha, que é representada por
 ou
. Note que a barra de fechamento no fim da tag (segunda opção) é opcional no HTML5, e ambas as formas causarão o mesmo efeito.

A junção da tag e seu conteúdo (se houver) é chamado de elemento HTML.

```
<p>Salve este exemplo em um arquivo com extensão .html e abra-o no navegador. Note que o
conteúdo do parágrafo fica entre as tags de abertura e fechamento.</p>
```

Atributos HTML

Dentro das tags HTML podemos colocar informações adicionais que auxiliam o navegador a entender e estruturar melhor o documento. Essas informações são chamadas de atributos HTML. Os atributos são adicionados sempre nas tags de abertura, no formato `nome="valor"`, sendo possível haver mais de um atributo na mesma tag, bastando separá-los por um espaço em branco.

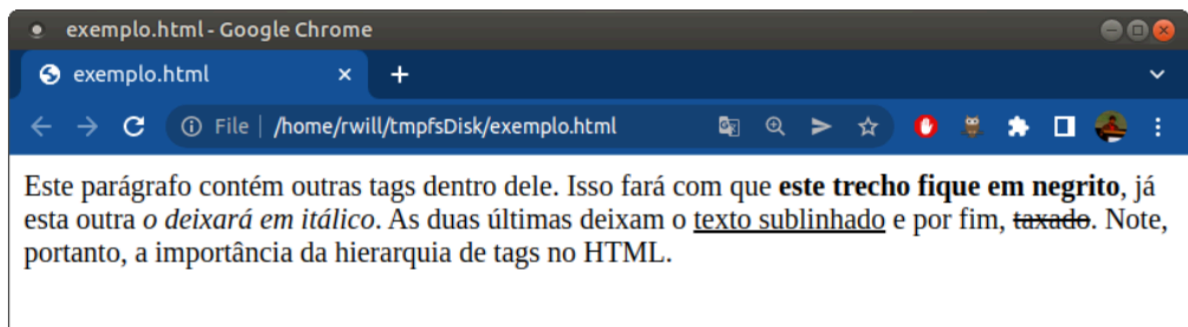
```
<p id="paragrafo" class="principal">Salve este exemplo em um arquivo com extensão .html e abra-o no navegador. Note que o conteúdo do parágrafo fica entre as tags de abertura e fechamento.</p>
```

Existem diversos atributos para as tags com os mais diferentes objetivos, alguns são de escopo global, ou seja, todas as tags podem usar (como os atributos `id` e `class`, por exemplo) e outros são específicos para tags específicas (exemplos: o atributo `href` para as tags `<a>` e `<link>`, ou o atributo `src` para as tags `` e `<script>`). Uma relação mais completa pode ser vista em [HTML Attributes \(s.d.\)](#).

Tags aninhadas

O HTML permite escrever tags dentro de outras tags, o que permite criar uma estrutura hierárquica de marcadores, que definirá a organização do documento (uma página na web). É possível, por exemplo, ter várias tags que marcam textos simples dentro de um parágrafo

```
<p>Este parágrafo contém outras tags dentro dele. Isso fará com que <strong>este trecho fique em negrito</strong>, já esta outra <em>o deixará em itálico</em>. As duas últimas deixam o <ins>texto sublinhado</ins> e por fim, <del>taxado</del>. Note, portanto, a importância da hierarquia de tags no HTML.</p>
```



Estrutura básica do documento HTML

Para definir um documento em HTML, usamos a estrutura básica definida na Figura 2.6. Nela, podemos notar o uso das seguintes tags:

Tag	Descrição
<code><!DOCTYPE html></code>	Define que o tipo do documento é HTML
<code><html> ... </html></code>	Define o início (e o fim) do documento HTML
<code><head> ... </head></code>	Contém metadados, informações e configurações do documento
<code><body> ... </body></code>	Contém o corpo do documento. É aqui onde vamos adicionar elementos visuais (títulos de seção, parágrafos, tabelas, imagens, campos de formulários, etc)
<code><!-- ... --></code>	Define um comentário em HTML. Pode usar várias linhas e seu conteúdo não é renderizado pelo navegador

```

<!DOCTYPE html>
<html>
<head>
<!-- Esta parte contém metadados e informações/configurações sobre o documento -->
</head>
<body>
<!-- Esta parte define o corpo do documento: títulos de seção, parágrafos, tabelas, imagens
etc. -->
</body>
</html>

```

Note que cada uma das tags possui uma função muito específica, e devem aparecer na ordem definida no exemplo. A seguir, explicaremos os principais marcadores (tags) da linguagem, dividindo-as em categorias para facilitar o entendimento.

Tipos de tags do HTML

O HTML5 define mais de 100 tags diferentes para diversos objetivos. Obviamente, estudar todas elas seria um trabalho muito grande, e por esse motivo vamos enfatizar as tags importantes e mais usadas.

Por mais que existam muitas tags, é possível dividi-las em algumas categorias para ajudar a encontrar uma que mais se adeque ao conteúdo que desejamos mostrar. As categorias de tags que dividimos neste curso são: configuração, marcação de texto, estruturas de texto, estruturas de documento e multimídia.

Tags de configuração

São tags escritas diretamente na head do documento, usadas para configurar como o navegador vai interpretar o documento como um todo. As mais comuns desta categoria são: title, meta, link, style e script. As tags style e script são específicas para o uso do CSS e do JavaScript respectivamente, e as detalharemos mais adiante no nosso curso.

A tag title define o título do documento. Esse título sempre será um texto, e irá aparecer na barra de título do navegador (na aba dele).

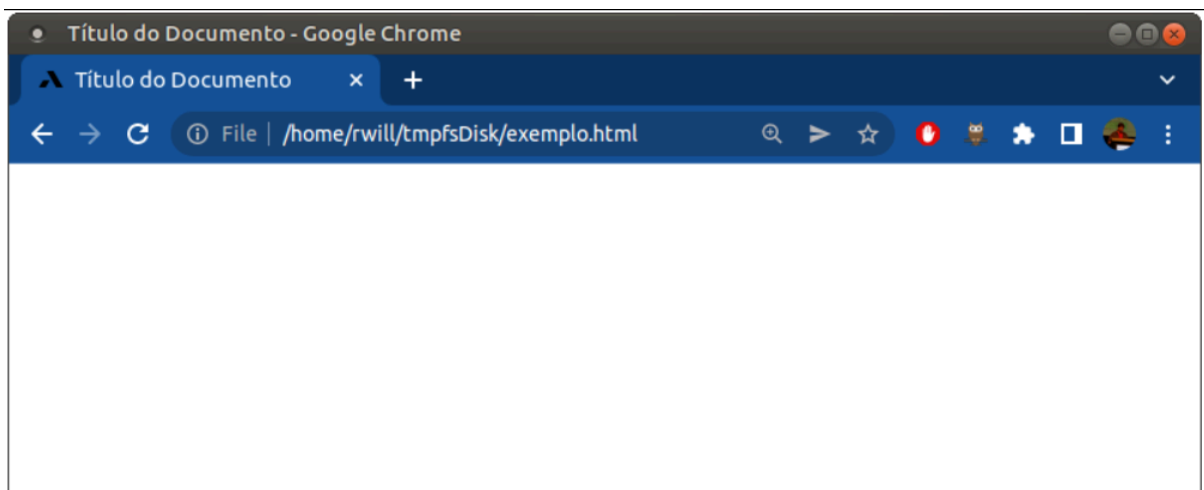
A tag meta é usada para configurar metadados (informações sobre o documento) que são invisíveis ao usuário, mas importantes ao navegador e buscadores: é uma tag autocontida que aceita vários atributos diferentes. O

atributo charset é comumente utilizado para indicar qual codificação de caracteres foi utilizada no arquivo HTML. O recomendado é que seja utilizada a codificação UTF-8.

A tag link é usada para carregar recursos externos usados na visualização pelo navegador, como arquivos CSS ou um ícone para a aba do navegador (favicon).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Título do Documento</title>
  <link rel="icon" href="favicon.ico">
</head>
<body>

</body>
</html>
```



Note que para simular esse exemplo em sua plenitude, você deve ter um arquivo de ícone (uma imagem com extensão ico, png, jpg, etc) com o mesmo nome indicado no atributo href da tag link. O ícone sempre deve ser uma imagem quadrada, e de preferência pequena (16x16, 32x32, 64x64 pixels, etc).

Tags de marcação de texto

São tags utilizadas para marcar um texto específico, sem que haja uma quebra no fluxo do texto em si, como se fosse usado um marca texto físico. Elas são usadas para dar algum tipo de ênfase no texto que desejamos "marcar". As mais importantes são as tags: a, strong, em, ins e del.

A tag a (âncora) é uma das mais utilizadas na Internet, pois é ela que possibilita a navegação entre os diferentes hipertextos, criando os hiperlinks entre eles. O texto interno à tag é o que fica marcado como conteúdo clicável pelo navegador (ganhando a clássica cor azul e sublinhado como estilo padrão).

Para a âncora funcionar, ela precisa do atributo href definido. Este atributo terá a URL do recurso que a âncora está referenciando, onde será navegado quando ela for clicada. Essa URL pode ser tanto absoluta (com protocolo, domínio, etc.) quanto relativa (com apenas o caminho que falta dentro do mesmo domínio). Além

disso, ela pode referenciar um conteúdo interno da página, usando o fragmento da URL com o ID do conteúdo que ela referencia. Veja mais exemplos e detalhes em [HTML a tag](#), (s.d.).

Já as tags `strong`, `em`, `ins` e `del` são usadas para realçar o texto com objetivos diferentes. Cada uma traz um significado semântico e um estilo diferente. A tag `strong` para textos importantes (negrito), `em` para textos enfáticos (itálico), `ins` para textos inseridos (sublinhado) e `del` para textos removidos (tachado).

Tags de estrutura de texto

Essas tags vão definir os elementos textuais presentes nas páginas da web, construindo todo o conteúdo com elementos comuns em textos que vemos em outros lugares, como livros e jornais. As tags mais importantes desta categoria são: `p`, `br`, `h1` até `h6`, `ul`, `ol` e `table`.

A tag `p` (já mostrada) representa uma unidade de um parágrafo textual, com os espaçamentos (margens) previstas.

Como qualquer quebra de linha feita por teclas `enter` são ignoradas pelo navegador, caso seja necessário quebrar a linha no HTML é necessário usar a tag `br`, que é uma tag autocontida (ou seja, basta escrever `
`), que causa o efeito de quebra de linha imediata.

Os títulos de sessão dentro de um documento são chamados de headings no HTML, e são marcados por tags que vão do nível 1 (mais prioritário) ao 6 (menos prioritário), ou seja: `h1`, `h2`, `h3`, `h4`, `h5` e `h6`. Os números nos headings são usados como indicação do nível de sessão, ou seja, o `h2` deve vir após um `h1`, o `h3` após o `h2`, e assim por diante (HTML Headings, s.d.). A Figura 2.8 mostra um exemplo de como ficam os títulos de sessão.

Título h1 - muito importante

Título h2 - menos importante que h1

Título h3 - menos importante que h2

Título h4 - menos importante que h3

Título h5 - menos importante que h4

Título h6 - menos importante que h5

As listas de itens e tabelas são elementos mais complexos, que envolvem mais de um tipo de tag. As listas básicas são divididas em dois tipos: ordenada (definida pela tag `ol`, de "ordered list") e não ordenada (definida

pela tag `ul`, de “unordered list”). A lista ordenada possui como identificador uma sequência, que pode ser numérica ou alfabética. Já a lista não ordenada tem como identificadores símbolos iguais, em geral pontos (bullets). Ambas as listas abrem com suas respectivas tags, mas são seguidas por vários itens marcados com a tag `li` (“list item”), com exemplos em [HTML Lists \(s.d.\)](#).

Já a tabela possui diversos elementos, alguns opcionais, para definir sua estrutura. Comumente começamos por sua tag `table`. Dentro dela colocamos a sequência de linhas que construirão a tabela, usando a tag `tr` (“table row”). Dentro de cada linha (`tr`) colocamos as células que farão parte das linhas, de maneira similar aos programas de planilha. As células podem ser representadas pelas tags `th` (“table header”, para as linhas de cabeçalho) e `td` (“table data”, demais linhas), veja exemplos em [HTML Tables \(s.d.\)](#).

Tags de estrutura de documento

Conforme o documento HTML vai crescendo, surge a necessidade de organizar o código e o conteúdo de acordo com os seus significados e sua semântica. Essa categoria inclui uma grande lista de tags, cada uma com sua função semântica na separação de conteúdos. Algumas das tags desta categoria são: `div`, `section`, `article`, `main`, `header`, `footer` e `nav`.

Todos os marcadores desta categoria são usados para agrupar outras tags. Por exemplo, se estamos montando uma página de notícias, e cada notícia for composta por um heading (título da manchete) e alguns parágrafos, faz sentido que cada uma das notícias esteja dentro de `article`, e um conjunto de notícias (ex: todas de esporte) estejam dentro de uma `section`. Cabeçalhos e rodapés, seja de páginas ou outros elementos, devem estar dentro das tags `header` e `footer` e quaisquer lista de âncoras (links) devem estar dentro de um `nav`, para indicar elementos de navegação. Veremos mais adiante sobre o conteúdo semântico das tags.

Tags de multimídia

As páginas da web não são feitas apenas de textos simples, temos muitos conteúdos de multimídia, como imagens, vídeos e sons e o HTML tem tags específicas para esses casos. As mais importantes nessa categoria são: `img`, `audio` e `video`.

A tag `img` é amplamente utilizada em praticamente todos os documentos HTML. Trata-se de uma tag autocontida que precisa de um atributo chamado `src` para indicar onde o navegador deve procurar a imagem (que pode ser indicada utilizando uma URL absoluta ou relativa). É recomendado que o atributo `alt` seja definido também, como um texto alternativo para exibir uma mensagem quando a imagem não carregar, e também para facilitar a navegação de usuários com dificuldades de visão, além de auxiliar buscadores da Internet.

As tags `audio` e `video` possuem uma estrutura similar. Ambas precisam definir a fonte dos conteúdos a serem tocados, através das tags secundárias `source`. Para ambas é possível definir mais de uma tag `source`, além de ser possível definir outros atributos de controles ou `autoplay` nas tags, como pode ser visto em [Video and audio content \(s.d.\)](#).

Você conhece?

O professor e divulgador pelo Youtube Gustavo Guanabara publica seu material, de forma totalmente gratuita, desde 2013, no seu site pessoal e no seu canal do Youtube. Possui dois cursos bem completos de desenvolvimento Web com HTML e CSS, além de outros temas relacionados a programação e computação geral. Confira mais detalhes no site dele <https://www.cursoemvideo.com/>.

Estrutura básica do documento HTML

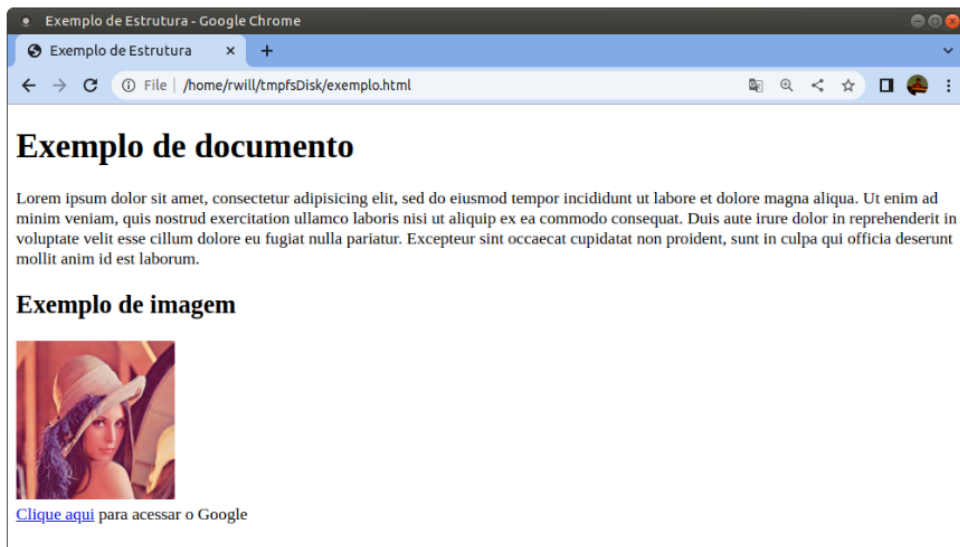
Assim como outros tipos de documentos digitais, o HTML possui uma estrutura própria que devemos seguir. Apesar dessa estrutura não ser obrigatória, uma vez que o navegador consegue entender tags “soltas”, é extremamente recomendável escrevê-la.

Como vimos, algumas tags são usadas para marcar conteúdo que o usuário irá ver: textos, imagens, tabelas, listas, títulos de seção (headings) etc. Essas tags compõem o que chamamos de “corpo” (body) do documento. Outras tags são usadas pelo navegador como configuração, e compõem a “cabeça” (head) do documento, como por exemplo: as tags title (título na aba), link (para importar algum recurso visual para o navegador) ou meta (para configurar algo no navegador, como a codificação a ser usada).

Além do head e do body, todo documento HTML possui mais dois elementos: a tag raiz html e a diretiva DOCTYPE. O HTML é uma linguagem de marcação do tipo SGML (Standard Generalized Markup Language), que é um padrão internacional para linguagens de marcação como o HTML e o XML (eXtensible Markup Language). Na SGML é obrigatório os documentos comecem com a diretiva de tipo de documento (DOCTYPE) e um primeiro elemento raiz, onde o resto do documento será escrito. No caso do HTML, o tipo sempre vem com a marcação `<!DOCTYPE html>` e o elemento raiz sempre é a tag `<html>`.

Portanto, todo documento HTML considerado correto fica com a estrutura representada na Figura 2.9.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Exemplo de Estrutura</title>
</head>
<body>
  <h1>Exemplo de documento</h1>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</p>
  <h2>Exemplo de imagem</h2>
  
  <div>
    <a href="http://www.google.com">Clique aqui</a> para acessar o Google
  </div>
</body>
</html>
```



Semântica HTML

O HTML surgiu como uma maneira de estruturar um documento na web e transferi-lo pela rede. Conforme o conteúdo presente evoluiu e a web pública ficou cada vez mais importante e maior, tornou-se necessário trabalhar mais na marcação deste conteúdo, de forma a facilitar o entendimento dele por mecanismos de busca e por usuários que usem tecnologias assistivas. O HTML na versão 5 trouxe uma revolução em termos de semântica para os documentos, ou seja, no sentido que os conteúdos querem trazer para os usuários que os estão consumindo.

Antes do HTML5 era comum dividir o conteúdo dos sites usando a tag `div`, que era usada apenas para dividir para organizar o código e facilitar a aplicação de estilos. Para facilitar o entendimento do conteúdo do documento HTML, outras tags de divisão com valor semântico foram adicionadas.

Agora pode-se utilizar a tag `section` para dividir o conteúdo de uma mesma página em seções (partes de um todo), ou usar a tag `article` para conteúdos mais independentes. As tags `header` e `footer` identificam com precisão onde está o cabeçalho e o rodapé das páginas, enquanto que a tag `main` identifica onde está o conteúdo principal. A tag `nav` mostra onde estão as opções de navegação, internas ou externas às páginas.

Além da divisão do documento, existem tags para dar conteúdo semântico aos textos específicos. As tags `strong` e `em`, por exemplo, além de deixarem o texto em negrito e itálico, respectivamente, também marcam o texto como algo importante para o documento (no caso do `strong`) e algo enfático (no caso do `em`), onde sua ênfase pode mudar o significado da frase que está inserido (ex: ironia).

Um site com o conteúdo semântico bem definido e dividido ajuda aos buscadores entenderem e referenciar mais a página em específico, sendo que essa construção é uma das bases do Search Engine Optimization (SEO), além de auxiliar na acessibilidade do site por usuários guiados por tecnologias assistivas.

Você quer ler?

Vimos que a semântica é importante na construção de sites mais entendíveis para tecnologias assistivas e os motores de busca. Existem outras coisas que devemos considerar na hora de escrever código HTML para melhorar a acessibilidade, como textos e atributos de tags HTML. Veja mais detalhes em [HTML: Boas práticas em acessibilidade \(s.d.\)](https://developer.mozilla.org/pt-BR/docs/Learn/Accessibility/HTML), disponível no link: <https://developer.mozilla.org/pt-BR/docs/Learn/Accessibility/HTML>.

Formulários no HTML

O HTML permite disponibilizar informação através da criação de interfaces gráficas com forte conteúdo semântico, inclusive recursos de multimídia. Mas além de apresentar dados (saída de dados), também é possível coletar informações (entrada de dados). Os formulários e seus controles surgem com a capacidade de obter as mais diversas formas de informações dos usuários, com controles de coleta de dados e de ações específicas para cada situação, tornando-se ferramentas essenciais para o dia a dia das empresas que utilizam sistemas web na sua operação.

Coleta de dados do usuário

Conforme a Internet foi crescendo e sendo utilizadas para outros negócios diferentes, a capacidade de apenas exibir dados aos usuários limitava o que podia ser feito com as páginas da web. A necessidade de coletar informações dos usuários e enviá-las ao servidor é essencial para a construção de qualquer sistema computacional, seja para confirmar dados específicos e liberar um certo conteúdo ao usuário (autenticação), ou o

cadastro de um endereço para entregas de encomendas, ou mesmo enviar uma simples postagem em uma rede social. Os formulários tornaram-se essenciais para todos os negócios.

Criação de formulários

Um formulário de coleta de dados no HTML é definido através das tags `<form></form>`. Essa tag funciona de maneira similar aos marcadores de estrutura de documento (como a `div`), mas com atributos específicos que controlam o seu funcionamento global. Todos os elementos que serão utilizados para a coleta dos dados devem estar dentro da tag `form`, isto é, entre as tags de abertura e fechamento. O processo de enviar os dados presentes no `form` para o servidor se chama `submeter` (`submit`) os dados ao servidor.

A tag `form` pode ser utilizada com os atributos padrão, mas alguns comportamentos indesejados podem ocorrer. Portanto, entender os três atributos principais do `form` é importante para aprender a maneira que o `form` submete os dados ao servidor, sendo possível e necessário configurar esses atributos conforme a necessidade. São eles: `enctype`, `action` e `method`. A Figura 3.1 mostra o uso da tag `form` com os três atributos, que serão explicados na sequência. Note que, por simplicidade, omitimos as tags `DOCTYPE`, `html`, `head` e `body`, mas você sempre deve utilizá-las de agora em diante.

```
<form method="POST" action="/salvarFormulario" enctype="application/x-www-form-urlencoded">
  <!-- Aqui fica o conteúdo do formulário,
       isto é, os campos que o usuário deve preencher -->
</form>
```

Atributo `enctype`

O processo de submissão faz a coleta dos dados presentes no formulário em formato de texto e os organiza de uma determinada forma para enviar ao servidor. Vale lembrar que o HTTP é um protocolo de troca de mensagens em texto, portanto os valores presentes no formulário precisam ser passados em formato textual. O atributo `enctype` (encoding type ou “tipo de codificação”, em português) especifica como os dados do formulário devem ser codificados quando submetidos (enviados) ao servidor.

Por padrão, isto é, se você não usar o atributo `enctype` de maneira explícita, seu valor é igual a `application/x-www-form-urlencoded`, que basicamente organiza os valores coletados de uma maneira similar aos parâmetros de query string da URL, ou seja, no formato `nome=valor`, onde diferentes nomes e valores serão separados pelo caractere `&`. Por exemplo, se enviarmos o formulário com dois campos distintos chamados `nome` (com o valor “Pedro”) e `sobrenome` (com o valor “Silva”), os dados enviados pelo formulário teriam o formato: `nome=Pedro&sobrenome=Silva`.

Para a maioria dos formulários que criaremos no dia a dia, o valor padrão basta, mas há casos em que será necessário enviar arquivos em conjunto com outros dados, e arquivos em geral são muito grandes para a codificação padrão. Neste caso, deve ser usado o valor `multipart/form-data` para o atributo `enctype`. Com essa codificação, o navegador separa os dados do arquivo e de outros campos em pequenos pedaços para melhor organizar o corpo da mensagem sendo enviada.

Atribuição `action`

Após a coleta dos dados com a codificação expressa no atributo `enctype`, o formulário vai fazer uma requisição para alguma rota (URL) configurada no servidor. Por padrão, o navegador envia os dados para a mesma URL da página que contém o formulário. Ou seja, se o formulário está na URL `https://www.impecta.edu.br/contato`, quando ele for submetido, todos os seus dados irão para a mesma URL (`https://www.impecta.edu.br/contato`).

O atributo `action` serve para configurar para qual URL o formulário será submetido, e só é usado quando desejamos enviar os dados para uma URL diferente da URL da página que o contém. Seu valor segue a mesma regra presente em outros atributos de URL (`src` e `href`), sendo permitido URLs absolutas e relativas.

Atribuição `method`

O atributo `method` especifica qual método será usado para enviar os dados, podendo ter dois valores: `GET` e `POST`. Basicamente, a diferença entre os dois é que o `GET` adiciona os dados na própria URL que será usada para fazer a submissão ao servidor. Esse método possui a desvantagem de deixar todos os dados visíveis, uma vez que estarão presentes na barra de endereço do navegador, e também limita o tamanho dos dados que podem ser enviados, pois todos os navegadores definem um limite máximo que uma URL pode ter.

O método `POST`, por outro lado, envia os dados de maneira “escondida”, dentro do corpo da requisição, e por esse motivo não possui restrição de tamanho para os dados que serão enviados. É o método ideal para enviar dados sensíveis como senhas, números de cartões de crédito, e outras informações de cunho pessoal dos usuários.

Por padrão, todo formulário envia os dados com o método `GET`, pois esse foi o primeiro a existir. Entretanto, na maioria das vezes será necessário configurar os formulários com o atributo `method` com o valor `POST`.

Campos de formulário

O formulário será a região de coleta dos dados do usuário, mas precisamos de controles específicos que os usuários possam interagir para colocar seus dados. Na grande maioria das vezes, o que será coletado é um texto em algum formato (texto, números, datas, etc), mas em alguns casos queremos limitar as escolhas do usuário ou coletar textos mais específicos.

Os campos de formulário, também chamados de `inputs`, são divididos basicamente em três categorias: coleta de texto, coleta de opções e botões de controle. A grande maioria dos campos é representado pela tag autocontida `<input>`, usada para colocar o controle em uma determinada posição no formulário. As tags `input` (e suas variações, que veremos adiante) possuem vários atributos específicos, mas as mais importantes são os atributos: `name` e `type`.

O atributo `name` permite que o navegador colete as informações com nomes a serem enviados ao servidor, similar às variáveis que existem nas linguagens de programação. Sem o atributo `name`, o navegador não conseguiria criar uma identificação única para determinado valor, e portanto não seria possível distinguir os diferentes valores que o usuário deseja enviar ao servidor durante uma requisição. Já o atributo `type` permite configurar o tipo de dado esperado no `input`, como por exemplo: texto, e-mail, senhas, URLs, etc.

Campos de coleta de texto

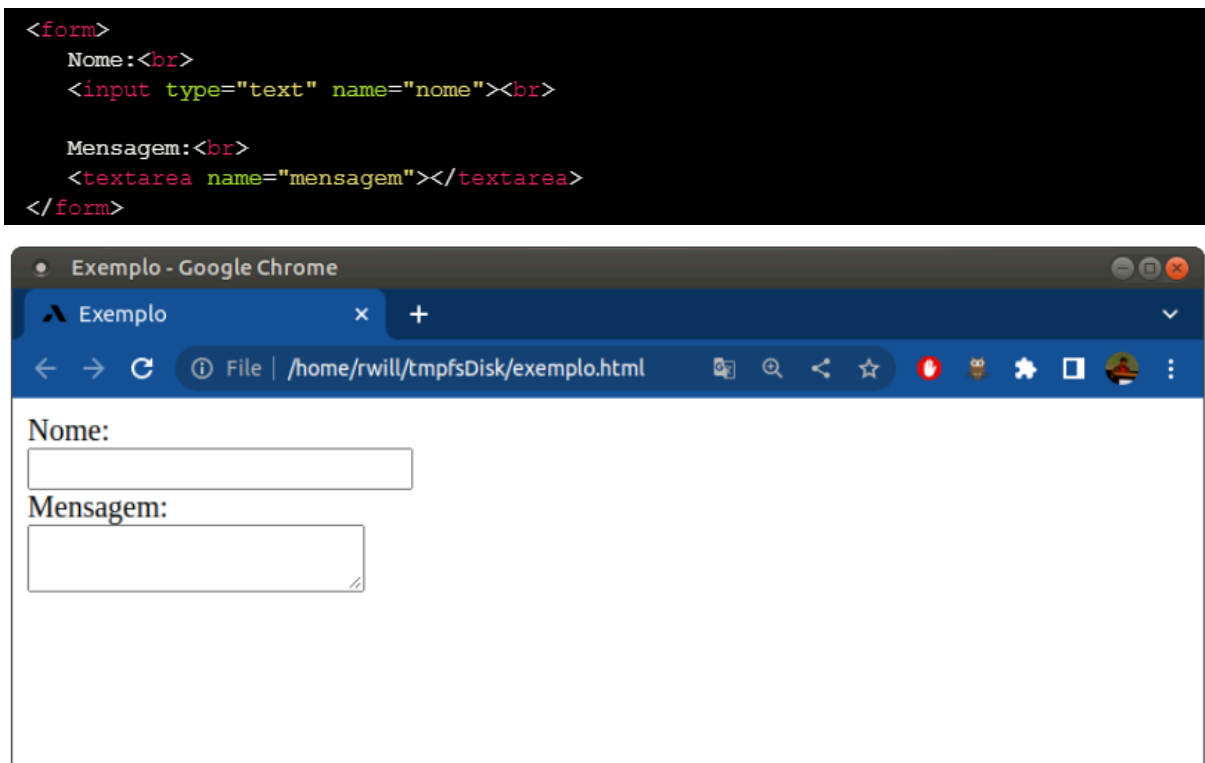
Para coleta de texto temos dois grandes tipos: textos de uma única linha e textos de múltiplas linhas. Para textos de uma única linha usamos a tag `input` com a variação do `type` de acordo com o dado que queremos obter. Por padrão, o atributo `type` tem o valor `text`, que é o campo mais elementar e aceita qualquer texto em uma única linha.

Caso seja necessário coletar algum tipo mais específico de dado, existem diversos tipos pré-definidos no HTML. Essas variações no type para textos serve para adicionar uma capacidade do navegador sugerir valores baseados no tipo e no histórico do usuário, bem como uma validação nativa se o dado está escrito corretamente.

As variações mais comuns do type são:

- text: valor padrão, usado quando omitimos o atributo type;
- password: campo de texto para senhas (esconde o valor digitado);
- email: campo de texto que valida e-mails;
- url: campo de texto que valida URLs;
- date: campo de texto que valida datas;
- datetime-local: campo de texto que valida data e hora;
- time: campo de texto que valida horas e minuto;
- number: campo de texto que aceita apenas números;
- Mais exemplos destes inputs podem ser consultados em [HTML input types \(s.d.\)](#).

Para textos de múltiplas linhas, precisamos usar um elemento diferente, definido pela tag `<textarea></textarea>`. Essa tag define uma caixa de texto que torna possível inserir quebras de linha no texto escrito. Note que, diferentemente da tag input, a tag textarea possui abertura e fechamento. Qualquer conteúdo dentro da tag é interpretado como sendo o valor inicial de texto nela ([HTML textarea tag, s.d.](#)). Mesmo sendo outra tag, por ser uma variação da tag input, ela também deve ter o atributo name configurado. A Figura 3.2 ilustra o uso das tags input e textarea dentro de um formulário.



Campos de coleta de opções

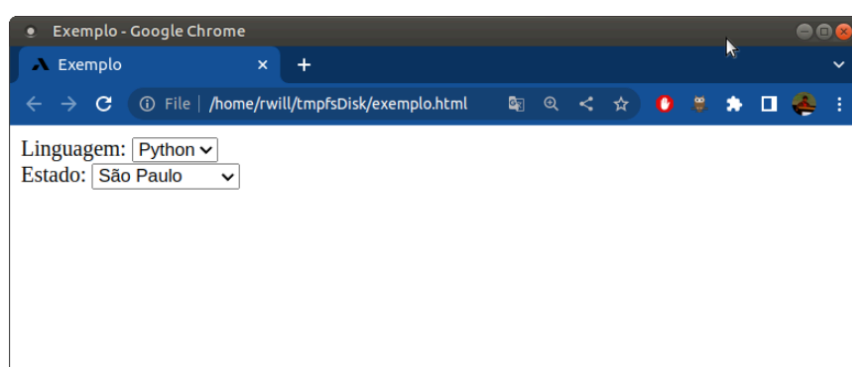
Também é possível restringir as opções do usuário a uma lista pré-definida de opções. Temos algumas maneiras de apresentar e restringir essas opções: caixas de seleção e caixas de marcação múltiplas ou únicas.

A caixa de seleção é uma tag diferente, pois passamos a ela uma lista de opções possíveis a serem escolhidas (em uma estrutura similar às listas ordenadas/não ordenadas). A caixa de seleção é definida pela tag `<select></select>`. Note que ela possui abertura e fechamento, e assim como as tags `input` e `textarea` deve ter o atributo `name` configurado.

A lista de opções disponíveis para o usuário escolher, é definida usando várias tags `<option></option>`. Cada tag `option` deve ter um conteúdo textual para mostrar ao usuário e, opcionalmente, pode ter um valor escondido que será enviado ao servidor caso a opção seja selecionada (definido pelo atributo `value`). Por exemplo, na Figura 3.3 temos duas caixas de seleção. Na primeira, qualquer opção escolhida envia o conteúdo textual diretamente na submissão, mas na segunda caixa o valor enviado é o que estiver no atributo `value` da opção marcada. A primeira `option` é a que vem selecionada por padrão ao carregar a página, mas isso pode ser alterado através do atributo booleano `selected`. Note que usamos `div`s para separar as duas caixas de seleção (opcional), e também uma tag nova chamada `label` (opcional, mas é uma boa prática utilizá-la), que pode ser usada com qualquer campo de formulário e será explicada mais adiante neste material.

```
<form method="POST">
  <div>
    <label for="se-linguagem">Linguagem:</label>
    <select name="linguagem" id="se-linguagem">
      <option>Java</option>
      <option>PHP</option>
      <option selected>Python</option>
    </select>
  </div>

  <div>
    <label for="se-estado">Estado:</label>
    <select name="estado" id="se-estado">
      <option value="SP">São Paulo</option>
      <option value="MG">Minas Gerais</option>
      <option value="RJ">Rio de Janeiro</option>
    </select>
  </div>
</form>
```



No exemplo da Figura 3.3, ao selecionar qualquer linguagem de programação, o texto selecionado será enviado ao servidor, isto é, um dos três valores listados: Java, PHP ou Python. Já ao selecionar um estado, o valor presente no atributo `value` do estado selecionado é que será enviado ao servidor, isto é: ou SP, ou MG, ou RJ. Em resumo, quando não usamos o atributo `value` na tag `option`, o valor enviado é o texto da `option`, e quando definimos o `value`, o seu valor é que será enviado na requisição.

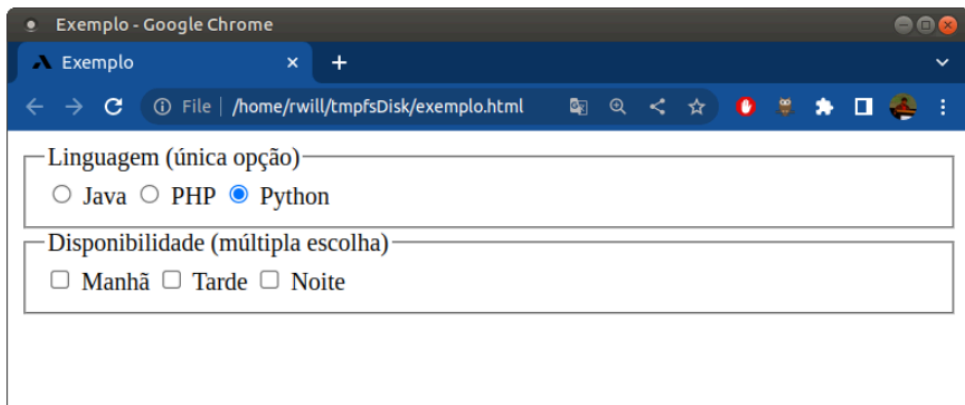
Caso seja interessante mostrar todas as opções disponíveis sem que o usuário precise consultar a caixa de seleção, similar ao que ocorre em provas e pesquisas, podemos usar as caixas de marcação. Temos dois tipos: marcação única e múltipla. Nestes casos, ambas são variações de tipos da tag input, já comentada acima. Usamos o type com valor radio para marcação única e com o valor checkbox para marcações múltiplas.

De maneira similar aos demais campos de formulários, também precisamos definir o atributo name para cada checkbox ou radio, mas no caso desses componentes, devemos usar o mesmo valor de name dentro de um mesmo grupo. Isso é importante para informar ao navegador que aqueles radios fazem parte do mesmo grupo, e com isso só será possível marcar um deles. Já no caso dos checkboxes, é possível marcar mais de um componente dentro de um mesmo grupo, e todos os valores marcados serão enviados com o mesmo name dentro daquele grupo, como se fosse uma lista de valores.

Por fim, cada checkbox ou radio precisa definir o valor que será enviado durante a submissão para o servidor, caso aquele componente esteja selecionado. Isso é feito através do atributo value. Note que esse valor fica oculto no navegador (ele não aparece para o usuário). Também é possível marcar quais componentes virão marcados por padrão, através do atributo booleano checked. A Figura 3.4 mostra o uso das caixas de marcação. As tags fieldset e legend servem para organizar melhor as caixas de marcação e são opcionais, e serão explicadas mais adiante neste material.

```
<form method="POST">
  <fieldset>
    <legend>Linguagem (única opção)</legend>
    <input type="radio" name="linguagem" id="ra-java" value="Java">
    <label for="ra-java">Java</label>
    <input type="radio" name="linguagem" id="ra-php" value="PHP">
    <label for="ra-php">PHP</label>
    <input type="radio" name="linguagem" id="ra-python" value="Python" checked>
    <label for="ra-python">Python</label>
  </fieldset>

  <fieldset>
    <legend>Disponibilidade (múltipla escolha)</legend>
    <input type="checkbox" name="disp" id="ra-manha" value="M">
    <label for="ra-manha">Manhã</label>
    <input type="checkbox" name="disp" id="ra-tarde" value="T">
    <label for="ra-tarde">Tarde</label>
    <input type="checkbox" name="disp" id="ra-noite" value="N">
    <label for="ra-noite">Noite</label>
  </fieldset>
</form>
```



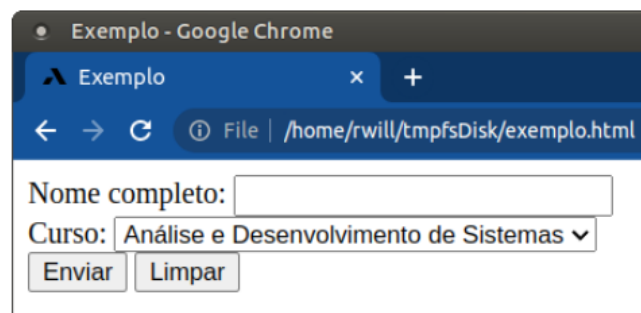
Existem ainda os campos de seleção de arquivos, que permitem que sejam anexados arquivos para enviar ao servidor. Os type file e image cuidam de criar um campo que permite escolher um arquivo do seu computador (ou imagem) para anexar no formulário.

Botões de controle

Para enviar um formulário, precisamos de um controle que o usuário use para dizer que ele preencheu tudo que era necessário e agora é o momento de enviar os dados ao servidor. Os botões de controle permitem que o usuário decida quando determinada ação deve ser executada. Eles podem ser definidos tanto pela tag `<input>`, como também pela tag `<button></button>`. Diferentemente da tag `input`, o `button` possui abertura e fechamento, com o conteúdo interno sendo usado para formatar o botão. Existem três tipos de botões em HTML, e eles são diferenciados pelo atributo `type`: `submit`, `reset` e `button`. Cada um desses tipos possui funções específicas:

- `submit`: usado para submeter (enviar) o formulário ao servidor. Ao clicar nele, o navegador efetua as validações dos dados escritos no formulário (veremos mais como isso funciona), codifica os dados obtidos seguindo seu `enctype`, e envia a requisição para a URL especificada no `action`, usando o método especificado no `method`. Todos os dados contidos na tag `form` que este botão estiver contido serão enviados, mesmos os dados escondidos;
- `reset`: faz com que todos os campos dentro do formulário voltem ao seu estado original. Em geral isso faz com que se esvazie tudo o que foi digitado e selecionado, mas caso algum campo tenha sido definido com algum valor padrão (através atributo `value`), esse valor é o que será colocado no campo. No caso das caixas de seleção, o valor com o atributo `selected` voltará a ser selecionado, e no caso das caixas de marcação (radios e checkboxes), os valores com o atributo `checked` voltarão a ser marcados;
- `button`: não faz nada por padrão, sendo utilizado apenas para ações customizadas com o A linguagem JavaScript.

```
<form method="POST">
  <div>
    <label for="tx-nome">Nome completo:</label>
    <input type="text" name="nome" id="tx-nome">
  </div>
  <div>
    <label for="se-curso">Curso:</label>
    <select name="curso" id="se-curso">
      <option value="ADS">Análise e Desenvolvimento de Sistemas</option>
      <option value="CC">Ciência da Computação</option>
      <option value="SI">Sistemas de Informação</option>
    </select>
  </div>
  <div>
    <button type="submit">Enviar</button>
    <button type="reset">Limpar</button>
  </div>
</form>
```



A screenshot of a Google Chrome browser window titled "Exemplo - Google Chrome". The address bar shows the file path "/home/rwill/tmpfsDisk/exemplo.html". The form is rendered with the following elements:

- A text input field labeled "Nome completo:".
- A dropdown menu labeled "Curso:" with the selected option "Análise e Desenvolvimento de Sistemas".
- Two buttons: "Enviar" and "Limpar".

Semântica nos formulários

Sempre que apresentamos informações no HTML devemos nos preocupar com o conteúdo semântico do que é apresentado. Esse mesmo princípio também vale quando construímos formulários, pois melhora a usabilidade geral do formulário para todos os usuários, incluindo os que usam tecnologias assistivas.

O uso de campos adequados para cada tipo de dado que se deseja coletar é o primeiro passo para formulários mais corretos do ponto de vista semântico. Por exemplo, o número de unidades federativas no Brasil é limitado (27 atualmente), portanto é muito mais adequado listá-las em uma caixa de seleção do que solicitar que o usuário digite o nome de algum estado, evitando erros de grafia. Outro exemplo é quando solicitamos o sexo do usuário e ele deve informar somente uma opção, sem ambiguidade, sendo adequado o uso de caixas de marcação do tipo radio. Mas além disso, há outras duas tags que ajudam no quesito da semântica: label e fieldset.

A tag `<label></label>` serve para criarmos um texto associado a um campo existente. Para isso, o label deve ter o texto no seu conteúdo através do atributo `for`, onde seu valor é igual ao valor do `id` do campo associado. O `id` é o único atributo que garante que o label esteja associado a um único campo. A importância do label se dá para ajudar em tecnologias assistivas em ler o nome do campo para saber o que deve ser escrito dentro dele.

Já a tag `<fieldset></fieldset>` é utilizada para agrupar vários campos que tenham o mesmo objetivo, por exemplo, campos de endereço (rua, número, cidade, estado, etc.) ou opções diferentes de um mesmo assunto (alternativas de uma questão). Dentro de um fieldset deve haver uma tag `<legend></legend>` com o nome do agrupamento (exemplo: título da questão) e todos os campos que se referem ao que o fieldset quer agrupar.

Tradicionalmente usamos fieldset com agrupamentos de vários radios ou checkboxes.

Validação em formulários

Antes de enviar os dados do servidor, é interessante validar o máximo possível se o que está escrito ou selecionado no formulário faz sentido com o que era esperado, economizando requisições desnecessárias. O HTML possui maneiras de indicar uma série de regras para validações básicas dos dados presentes nos formulários.

A primeira validação ocorre pelo tipo. Se o campo tem um `type` muito específico, como `email`, `number` ou `url`, o próprio navegador já mostra uma mensagem de erro caso o dado não esteja escrito da maneira correta. Além disso, temos alguns atributos que podem ser incluídos nos campos para auxiliar na validação (`Form data validation`, s.d.):

`required`: atributo booleano; quando presente, faz com que o navegador verifique se o campo está vazio antes de enviar os dados para o servidor e avisa caso esteja vazio;

`maxlength` ou `minlength`: para tipos textuais (incluindo `textarea`) esses atributos requisitam um número máximo ou mínimo de caracteres escritos no campo para serem aceitos para submissão;

`min` ou `max`: de maneira similar, para o campo `number`, podemos definir um valor mínimo ou máximo para o que o usuário irá colocar no campo;

`pattern`: podemos definir uma expressão regular (regex) para verificar se o que está escrito bate com o esperado, comumente usado para números de telefones.

Aprenda Mais Sobre Form

A course about HTML forms to help you improve your web developer expertise. <https://web.dev/learn/forms/>

Introdução ao CSS

O HTML é uma linguagem de marcação que permite definir a estrutura de um documento com os seus recursos visuais e textuais, mas nos dá pouco controle de como mostrar esses recursos. O CSS é uma linguagem que complementa o HTML, pois permite definir os estilos (apresentação) do documento através de um conjunto de regras, e sem alterar o conteúdo semântico das páginas web.

O que é a linguagem CSS?

A linguagem HTML é usada para definir quais elementos visuais deseja-se exibir nas páginas web. Ela também permite definir a semântica deste conteúdo, mas a sua capacidade de formatar e embelezar o que está sendo mostrado é limitada. Quando é necessário estilizar o conteúdo, empregamos uma outra tecnologia conhecida por CSS, que é um acrônimo para Cascading Style Sheets (em português: Folhas de Estilo em Cascata).

Assim como o HTML, o CSS também passou por diversas revisões para adicionar mais capacidades e flexibilidade à linguagem. Atualmente, é possível criar animações que, somadas a algumas funcionalidades da linguagem JavaScript, acabaram por substituir outras tecnologias usadas para criar animações (como por exemplo o Adobe Flash). Essa flexibilidade se dá nas centenas de propriedades que o CSS define. Entender todas elas de uma única vez é impraticável, inclusive muitas propriedades ainda serão adicionadas ao longo do tempo com a evolução dos navegadores. Portanto, o mais importante é entender como o CSS é aplicado, como funciona seu mecanismo de seleção, entender as propriedades mais utilizadas e as maneiras de incorporá-lo ao código HTML.

Como usar o CSS

O CSS é uma linguagem diferente, usada para definir regras de estilos aos elementos presentes em um documento HTML, e por este motivo não é considerada uma linguagem de programação. Toda regra CSS deve ser aplicada sobre um (ou mais) elementos do HTML, ou seja, o funcionamento do CSS é intimamente ligado ao HTML, apesar do HTML existir sem o CSS.

Comumente escrevemos código CSS em um arquivo de texto separado (externo), com extensão .css. Também é possível escrever diretamente dentro do arquivo HTML de duas formas: a interna, onde o código CSS deve ficar contido entre as tags <style></style>, por sua vez contida dentro do head da página, e a maneira inline, na qual é possível escrever propriedades CSS diretamente no atributo style de uma tag HTML. Cada uma dessas formas possui seu objetivo, vantagens e desvantagens, mas a maneira externa (arquivo separado) é a mais recomendada e mais amplamente utilizada.

Quando definimos um arquivo CSS externo, o mesmo deve ser importado dentro do documento HTML. Para isso, utiliza-se a tag <link>, informando o valor stylesheet no atributo rel, de forma similar ao exemplo a seguir: <link rel="stylesheet" href="arquivo.css">.

Sintaxe básica do CSS

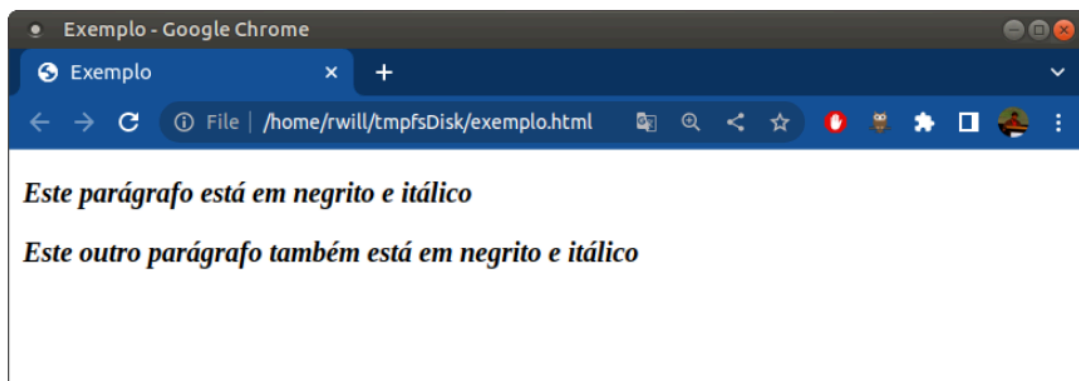
As regras definidas no CSS são descritas através de propriedades, onde cada propriedade será escrita no formato nome e valor, separadas pelo caractere dois pontos (:). Por exemplo, se quisermos deixar o texto de um parágrafo no HTML em negrito, devemos definir a regra `font-weight: bold`, já para fonte em itálico definimos a regra `font-style: italic`. É possível escrever várias propriedades CSS para um mesmo elemento, separando-as por ponto e vírgula (;).

Se usarmos a maneira inline de declarar o CSS, escrevemos quantas propriedades quisermos diretamente no atributo `style` (ex: `<p style="font-weight: bold; font-style: italic">Este parágrafo está em negrito e itálico</p>`).

O problema do inline é que o reaproveitamento de regras não existe. Caso quiséssemos que todos os parágrafos tenham a mesma formatação, teríamos que copiar todas as propriedades deste para os outros parágrafos, dificultando a manutenção. Para evitar este problema, o CSS permite escrever uma regra mais geral que seja aplicada a todos os parágrafos. No head do documento podemos usar a tag `<style></style>` que permite escrever código CSS diretamente no HTML, de forma a escrever regras aplicáveis a página inteira.

No caso dessa separação, precisamos dizer sobre quais elementos HTML aquelas propriedades serão aplicadas, utilizando um seletor de elementos. A sintaxe muda um pouco nesse caso. A Figura 4.1 mostra como ficaria o exemplo anterior escrevendo CSS separadamente (dentro da tag `style`). Neste caso, uma regra CSS é composta de um seletor de elementos HTML, depois abre-se uma região de propriedades com o par de chaves ({ }) e dentro dessa região escrevemos a lista de propriedades a ser aplicada nos elementos selecionados, todas separadas por ponto e vírgula (;).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Exemplo</title>
  <style>
    p {
      font-weight: bold;
      font-style: italic;
    }
  </style>
</head>
<body>
  <p>Este parágrafo está em negrito e itálico</p>
  <p>Este outro parágrafo também está em negrito e itálico</p>
</body>
</html>
```



A sintaxe do CSS utilizada dentro da tag style na Figura 4.1 é a mesma utilizada para separar o CSS em arquivos externos (com extensão .css). Como você deve ter notado, o CSS interno facilita a manutenção e reaproveitamento de código, mas ainda não possibilita que uma mesma regra seja aplicada no site inteiro, isto é, em outros documentos HTML. Por isso é indicada a separação do código CSS em um arquivo externo, que precisa ser importado (através da tag link) para que suas regras passem a valer na página. A Figura 4.2 exemplifica a definição e importação de um arquivo CSS externo. O resultado visual no navegador é idêntico ao da Figura 4.1, e por isso foi omitido.

```
Arquivo: estilos.css

p {
    font-weight: bold;
    font-style: italic;
}

Arquivo: HTML

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Exemplo</title>
  <!-- A tag link a seguir importa o arquivo CSS para este arquivo HTML -->
  <link rel="stylesheet" href="estilos.css">
</head>
<body>
  <p>Este parágrafo está em negrito e itálico</p>
  <p>Este outro parágrafo também está em negrito e itálico</p>
</body>
</html>
```

Propriedades básicas do CSS

O CSS define uma infinidade de propriedades, mas algumas delas são muito usadas no dia a dia, portanto é importante conhecê-las. A seguir categorizamos algumas das principais.

- Propriedades que permitem mudar a forma como um texto é apresentado:
- font-family: indica a fonte a ser usada. Algumas observações importantes:
 - A fonte usada precisa estar presente (instalada) no computador do usuário. Por isso não utilize fontes “obscuras”;
 - É possível especificar várias fontes separadas por vírgulas. O navegador usa a primeira que o utilizador tenha;
 - Se o nome da fonte tiver mais de uma palavra é necessário usar aspas;
 - O último valor desta propriedade deverá ser uma classe de fonte mais genérica como: serif, sans-serif, cursive, fantasy, monospace.
- font-size: indica o tamanho da fonte;
- font-weight: indica se a fonte é negrito ou não. Alguns valores possíveis: normal, bold, bolder, lighter, valores de 100 a 900 (aumentando de 100 em 100).
- font-style: indica se o texto deve ser escrito em itálico ou não. Alguns valores possíveis: normal, italic, oblique.
- text-decoration: indica se o texto deve ser sublinhado ou não. Alguns valores possíveis: none, underline, overline, line-through;

- `text-transform`: muda a capitalização das letras. Alguns valores possíveis: `none`, `capitalize`, `uppercase`, `lowercase`;

Propriedades que permitem alinhar texto (não funcionam para alinhar outros elementos!):

- `text-align`: permite alinhar o texto dentro de um elemento. Alguns valores possíveis são: `center`, `left`, `right`, `justify`;
- `text-indent`: permite indentar a primeira linha de um parágrafo. Os valores desta propriedade podem ser um comprimento ou uma percentagem;

Propriedades que permitem mudar o espaçamento entre as várias componentes do texto (letras, palavras, linhas, parágrafos, etc):

- `letter-spacing` e `word-spacing`: permitem mudar o espaçamento entre letras ou entre palavras. Os valores permitidos são um comprimento ou `normal`; `line-height`: altera a altura de uma linha sem alterar o tamanho da fonte. Os valores permitidos são um comprimento, uma percentagem ou `normal`;

Propriedades que permitem definir cores:

- `color`: permite modificar a cor de um texto;
- `background-color`: permite modificar a cor de fundo de um elemento.

Cores

As cores são representadas por uma composição de três canais de cores primárias: vermelho (red), verde (green) e azul (blue), compondo a sigla RGB. São usados 8 bits para cada uma dessas cores, e portanto cada um dos três canais de cor utiliza números variando de 0 a 255. Usamos a notação `rgb(RED, GREEN, BLUE)` onde colocamos valores numéricos no lugar dos nomes das cores. Podemos usar também a notação hexadecimal, onde temos três números hexadecimais (que vão de 00 até FF) para representar a composição RGB. A Tabela 4.1 mostra algumas cores e suas representações nas duas notações.

Cor	RGB	Hexadecimal
Vermelho (red)	<code>rgb(255, 0, 0)</code>	<code>#FF0000</code>
Verde (green)	<code>rgb(0, 255, 0)</code>	<code>#00FF00</code>
Azul (blue)	<code>rgb(0, 0, 255)</code>	<code>#0000FF</code>
Preto (black)	<code>rgb(0, 0, 0)</code>	<code>#000000</code>
Branco (white)	<code>rgb(255, 255, 255)</code>	<code>#FFFFFF</code>
Amarelo (yellow)	<code>rgb(255, 255, 0)</code>	<code>#FFFF00</code>
Púrpura (purple)	<code>rgb(128, 0, 128)</code>	<code>#800080</code>

Existem outras formas de representar cores, independente da maneira escolhida, ela será sempre interpretada na versão `rgb()`. Usualmente nos nossos códigos CSS usamos a representação em hexadecimal por ser mais concisa.

Uma maneira simplificada de definir cores mais comuns é através de valores constantes. Nos navegadores existem ao menos 140 cores com nomes já pré-definidos em inglês (`red`, `green`, `blue`, `black`, `white`, etc.).

Seletores CSS

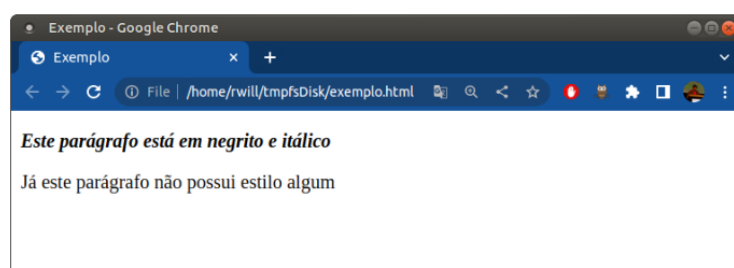
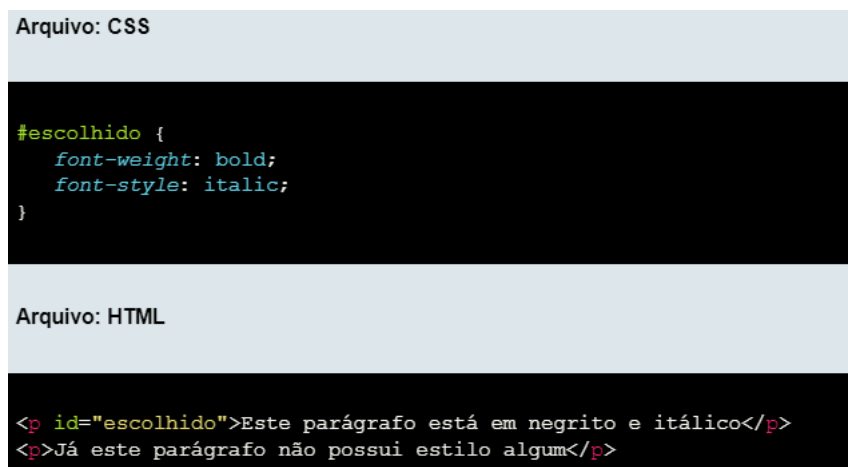
Ao separar as regras CSS das tags HTML (seja dentro da tag `style` ou em arquivos externos) precisamos informar em quais elementos as propriedades serão aplicadas. Para isso usamos os seletores CSS. Esses seletores possuem uma sintaxe própria muito poderosa e flexível para encontrar elementos HTML das mais diversas maneiras. Vamos revisar os seletores mais importantes. Uma lista mais completa pode ser consultada em [CSS Selector Reference](#) (s.d.).

Seletores por tipo

É o seletor mais intuitivo, e corresponde ao seletor usado no arquivo CSS do exemplo da Figura 4.2. O CSS permite que usemos o nome da tag para indicar que todas as ocorrências daquela tag no documento HTML seguirão as mesmas regras definidas por este seletor.

Seletores por ID

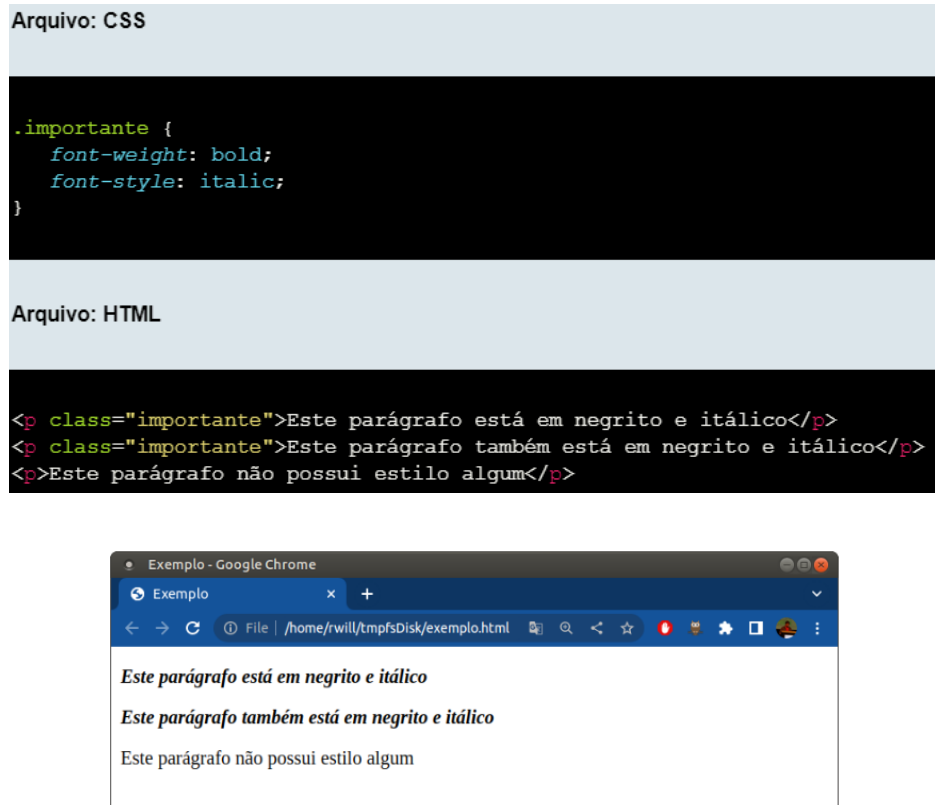
Em muitas situações não desejamos estilizar todas as ocorrências de uma tag, mas sim um único elemento específico do documento. Uma das formas de se alcançar isso é definir um identificador único para aquele elemento através do atributo `id`. Desta forma, o CSS permite definir regras específicas para esse elemento através do seletor de ID, que possui a sintaxe `#id-do-elemento`. A Figura 4.3 mostra um exemplo de seletor por ID. Por simplicidade, a partir deste exemplo iremos omitir as tags de estrutura do documento HTML.



Seletores por classe

O seletor por id permite selecionar apenas um elemento específico do documento HTML. Se quisermos aplicar um determinado estilo a vários elementos, mas não todos do mesmo tipo, podemos usar o seletor por classe.

Toda tag HTML permite definir um atributo chamado class, cuja função é criar categorias (ou classes) que queiramos usar no CSS, para flexibilizar o uso dos seletores. Por exemplo, podemos criar uma categoria de parágrafos importantes, que terá como estilo estarem em negrito e itálico ao mesmo tempo. Colocamos a propriedade class nos que terão este estilo e usamos o seletor .nome-da-classe na regra do CSS. A Figura 4.4 mostra um exemplo do uso de seletor por classe. Note que os dois primeiros parágrafos pertencem à mesma categoria (chamada de “importante”), mas o último parágrafo não.



O atributo class é um dos poucos que permitem diversos valores no HTML, ou seja, é possível um elemento HTML ter diversas classes declaradas, bastando para isso separá-las com espaços. Com isso, é possível que regras definidas por seletores de classes diferentes sejam aplicadas a um mesmo elemento.

Outro detalhe importante é que as classes não são restritas a nenhum tipo de tag específica, ou seja, no exemplo da Figura 4.4 se existisse um elemento h1 com classe importante, ele também teria seu texto exibido em negrito e itálico. Se quisermos ter certeza de que apenas elementos p de classe importante recebam esse estilo, é possível combinar os seletores, escrevendo p.importante. Nesse tipo de combinação, o tipo da tag sempre vem antes do seletor de classe.

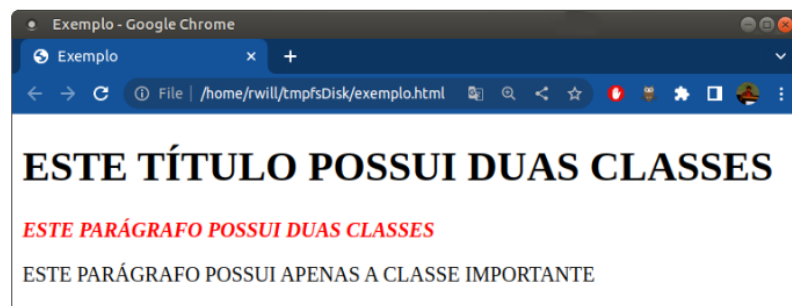
Por fim, também é possível agrupar classes diferentes para criar regras novas. Por exemplo, podemos criar uma regra que parágrafos que sejam de classe urgente e importante tenham todas negrito, itálico e cor vermelha. Para isso, escrevemos a regra: p.importante.urgente. Neste caso, a ordem entre os seletores de classe não importa, mas o de tipo vem antes, sempre. Podemos suprimir o seletor de tipo e escrever apenas .importante.urgente, nesse caso o seletor se refere a qualquer elemento que possua as duas classes ao mesmo tempo. A Figura 4.5 ilustra um exemplo com agrupamento de classes. Note que os dois seletores definem regras para a classe importante, enquanto que o último serve tanto para a classe importante como também para a classe urgente, mas só é aplicado para parágrafos. O texto entre /* */ indica comentários no código, que serão ignorados pelo navegador.

Arquivo: CSS

```
.importante {  
  /* Funciona para qualquer tag com a classe "importante" */  
  text-transform: uppercase;  
}  
  
p.importante.urgente {  
  /* Funciona somente para parágrafos (tag p) com as classes "importante" e "urgente" */  
  font-weight: bold;  
  font-style: italic;  
  color: red;  
}
```

Arquivo: HTML

```
<h1 class="importante urgente">Este título possui duas classes</h1>  
<p class="importante urgente">Este parágrafo possui duas classes</p>  
<p class="importante">Este parágrafo possui apenas a classe importante</p>
```



Seletores de atributos

O CSS também permite selecionar elementos por atributos HTML específicos e valores específicos. Usamos o nome do atributo entre colchetes para selecionar elementos que possuam o atributo especificado. Por exemplo, o seletor `a[href]` seleciona todos os elementos `a` (âncoras) que tenham o atributo `href` definido.

É possível selecionar elementos cujos atributos possuem algum valor específico, (exemplo: `a[href="#paragrafo"]`), ou que contenham uma parte do valor (exemplo: `a[href*="impacta"]`) e outras possibilidades (CSS Attribute Selectors, s.d.).

Seletores de agrupamento

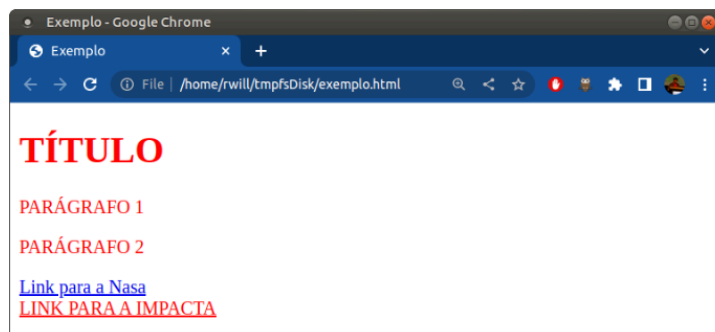
Caso tenhamos um mesmo conjunto de propriedades que deve ser aplicado a diferentes seletores, podemos reaproveitar código e agregar vários seletores em um único. Para isso, separamos os vários seletores usando vírgulas (.). Quaisquer seletores podem ser agrupados (por tipo, ID, classe, etc), e a ordem do agrupamento não faz diferença no seletor. A Figura 4.6 ilustra um exemplo de seletores agrupados.

Arquivo: CSS

```
h1, p, .link-comum {
  color: red;
  text-transform: uppercase;
}
```

Arquivo: HTML

```
<h1>Título</h1>
<p>Parágrafo 1</p>
<p>Parágrafo 2</p>
<a href="http://www.nasa.gov">Link para a Nasa</a><br>
<a href="http://www.impacta.edu.br" class="link-comum">Link para a Impacta</a>
```



Seletores de posição relativa

É possível também escrever seletores de posição relativa a outros elementos. Por exemplo, podemos pegar um elemento que esteja dentro de outro usando os seletores de descendência, ou elementos irmãos de outros (dentro do mesmo elemento) usando seletores de adjacência (CSS Combinators, s.d.).

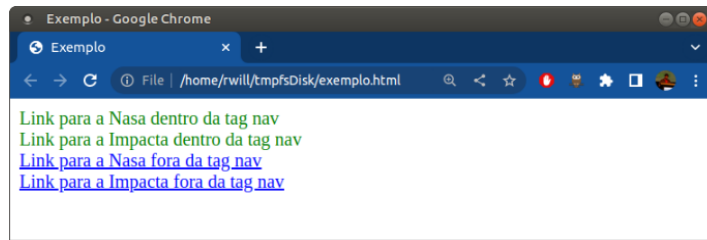
O seletor `nav` a seleciona todos os elementos que estão dentro de um elemento `nav`. Podemos usar qualquer outra combinação, como por exemplo `div p`, para selecionar todos os parágrafos dentro de tags `div`, ou então `div#importante p.urgente`, para selecionar todos os parágrafos com o atributo `class` `urgente` dentro da `div` com `id` `importante`. A Figura 4.7 mostra um exemplo deste seletor.

Arquivo: CSS

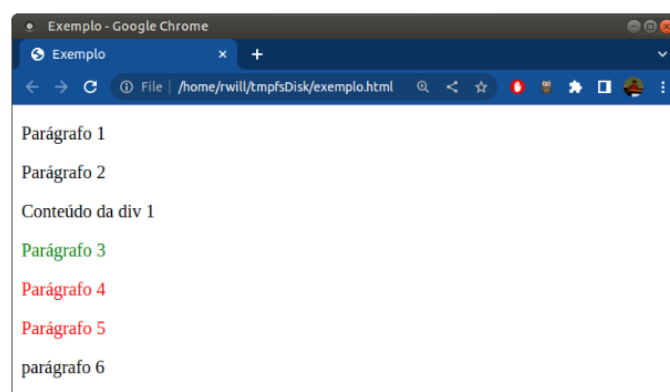
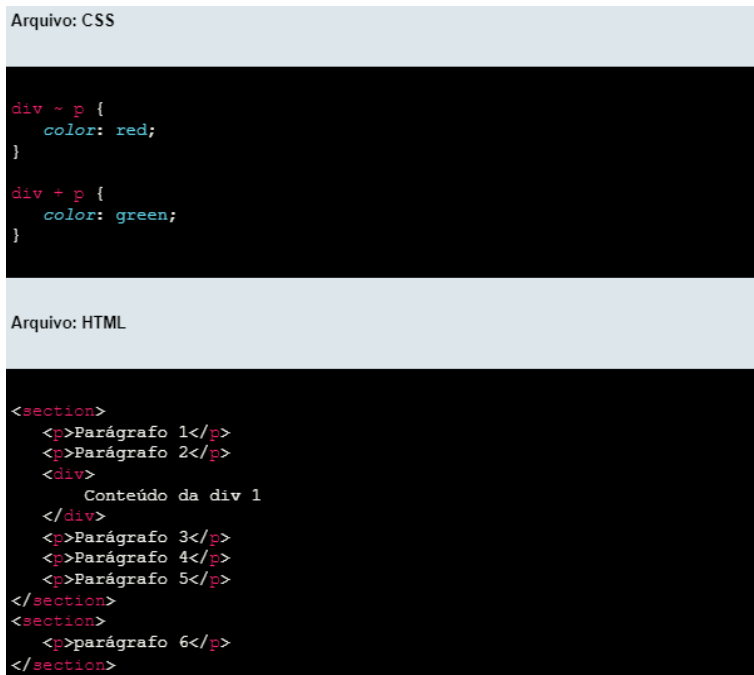
```
nav a {
  color: green;
  text-decoration: none;
}
```

Arquivo: HTML

```
<nav>
  <a href="http://www.nasa.gov">Link para a Nasa dentro da tag nav</a><br>
  <a href="http://www.impacta.edu.br">Link para a Impacta dentro da tag nav</a>
</nav>
<a href="http://www.nasa.gov">Link para a Nasa fora da tag nav</a><br>
<a href="http://www.impacta.edu.br">Link para a Impacta fora da tag nav</a>
```



Existem também os seletores de adjacência ~ e +. No primeiro seletor (exemplo: div ~ p) seleciona todo parágrafo que esteja depois de uma div, desde que tenham o mesmo pai. Já o seletor div + p seleciona apenas os parágrafos que estejam definidos imediatamente após tags div.



Pseudo-classes e pseudo-elementos

As pseudo-classes e os pseudo-elementos são categorias adicionais que colocamos em seletores para ter mais flexibilidade na seleção dos elementos, sem alterar o HTML que já existe. As pseudo-classes permitem definir estilos que dependem do estado dos elementos. Por exemplo, é possível alterar estilos se um determinado elemento é o primeiro filho de um outro (exemplo: p:first-child), ou se um elemento a (âncora) já foi visitado ou

não (a:visited), ou se o cursor do mouse está em cima de um link (a:hover), ou se algum campo de formulário está com foco (input:focus), etc. (CSS Pseudo-classes, s.d.).

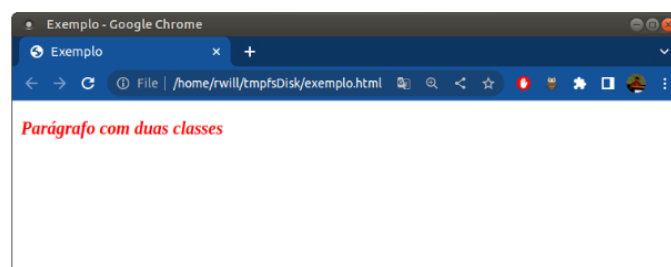
Já os pseudo-elementos permitem definir estilos para partes específicas do elemento selecionado, como a primeira linha de um parágrafo (p::first-line) ou a primeira letra de um item de lista (li::first-letter) (CSS Pseudo-elements, s.d.).

Vamos praticar?

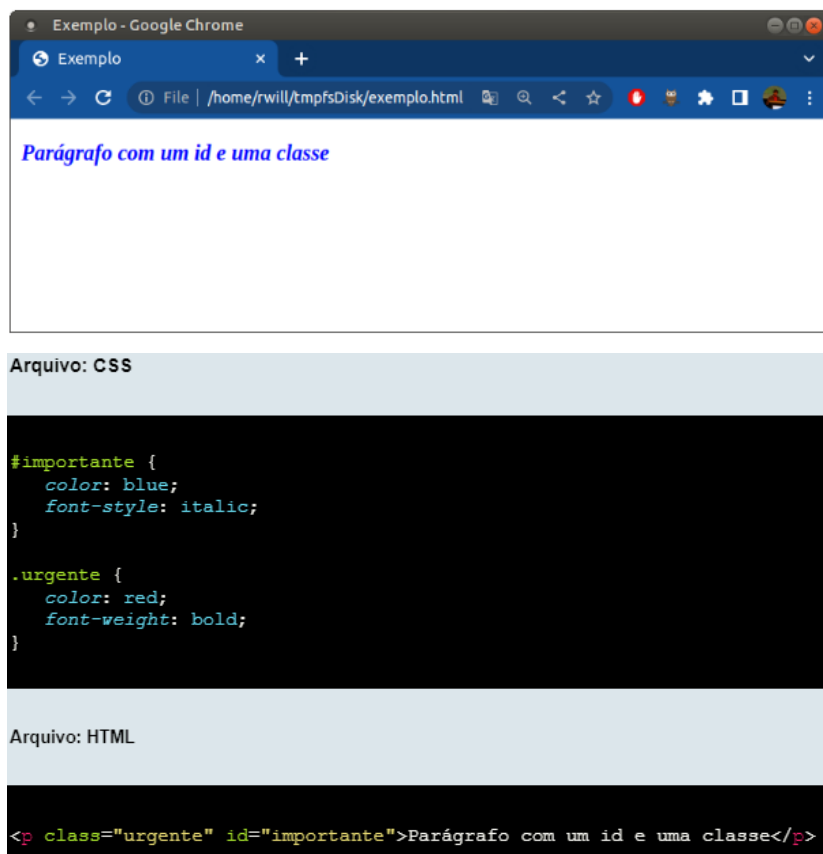
Para melhor entender como funciona essa grande quantidade de seletores, existe um jogo online para esse aprendizado. O jogo é feito inteiramente em HTML, CSS e JS (tem link para o código fonte) e treina em praticamente todas as combinações de seletores. Acesse ele em: <https://flukeout.github.io/>.

Priorização de regras

Conforme nossas regras CSS vão crescendo e se espalhando, não é difícil esbarrar em regras que podem ser conflitantes, como mostrado no exemplo da Figura 4.9. Neste caso, as regras por classe urgente e importante aplicam estilos diferentes e parcialmente conflitantes no parágrafo.



O navegador processa as regras CSS na ordem que ele encontra. Portanto, a primeira regra (elementos de classe importante) é aplicada ao parágrafo e ele ganha fonte azul e itálico. Já a segunda regra (elementos de classe urgente) aplica no elemento uma fonte vermelha e negrito. Neste momento, o navegador substitui a cor azul pela vermelha, uma vez que há conflito, e mantém o estilo de fonte em itálico e negrito, pois nenhuma dessas propriedades conflitam. Considere agora o exemplo da Figura 4.10, onde há uma inversão do ocorrido.



Neste exemplo o parágrafo possui ID e classe, e há um conflito na propriedade color dos seletores por ID e por classe. Observe que a ordem não importou e o parágrafo ficou azul, referente à primeira regra. Isso se deve ao mecanismo de priorização de regras do CSS, que leva em consideração três aspectos (Cascade and inheritance, s.d.):

1. Posição das Regras
2. Especificidade dos Seletores
3. Importância

Antes de aplicar as regras definidas no CSS, o navegador faz um ranking de regras mais específicas, e as aplica a partir da menos específica até a mais específica. Esse grau de especificidade é baseado em valores que o CSS atribui para cada tipo de seletor contido na regra. Não somos obrigados a saber esses valores (o navegador cuida disso), mas é importante saber que:

- Seletores de tipo e pseudo-elementos são os menos específicos;
- Seletores de classe, atributos ou pseudo-classes são mais específicos que os anteriores;
- Seletor por ID é o mais específico de todos;
- Usar regras no atributo style (definição inline) é o jeito mais específico de todos;

Portanto, no exemplo anterior, a cor azul permaneceu, pois o seletor por ID é mais específico do que o de classe, portanto foi aplicado após a de classe, resultando na fonte azul. Combinar diferentes tipos de seletores aumenta o grau de especificidade, fazendo uma regra ser aplicada antes da outra, independente da ordem de declaração.

Existe uma palavra chave para fazer uma determinada propriedade ganhar em importância de todas as outras e prevalecer nesse ranking que o navegador faz. A declaração `!important`, feita logo após o valor de uma propriedade, desfaz todas essas regras de especificidade da propriedade marcada com ela para garantir que seja aplicada. Entretanto, seu uso NÃO é recomendado sempre que possível, pois ela torna a depuração de problemas e a manutenção do código CSS muito complicada.

Layouts no CSS

Vimos como usar o CSS para dar estilos aos nossos elementos HTML, alterando cores e textos, usando as regras CSS e os seletores. Agora, para se pensar em layouts nas páginas HTML, precisamos entender as principais propriedades que configuram a maneira que os elementos se dispersam pela tela do navegador. Todo elemento HTML possui valores padrão para essas propriedades, veremos como alterar elas para conseguir efeitos específicos.

Dimensões e unidades

Para entender como criar layouts nas páginas, primeiro é necessário entender como os elementos ocupam espaço na janela. Como eles sempre são dispostos de maneira bidimensional, consideramos duas medidas: altura (`height`) e largura (`width`), duas propriedades fundamentais em CSS. Mesmo que não tenhamos definido valores para essas propriedades, o navegador calcula de acordo com suas regras de display (que veremos mais à frente) e o conteúdo interno do elemento.

Essas medidas (calculadas ou informadas) e qualquer outra informação de tamanho (como os tamanhos de fonte) usam por padrão a unidade da computação gráfica pixel (abreviada como `px` no CSS). Um pixel é a menor medida utilizada em computação gráfica, e representa um dos pequenos quadradinhos que todas as telas possuem e são usados para renderizar as imagens. Em geral, a qualidade das telas é medida pela quantidade disponível de pixels. Por exemplo, um monitor Full HD possui 1920 pixels de largura e 1080 de altura. Já um monitor 4K possui 3840 pixels de largura e 2160 de altura.

A unidade pixel é considerada absoluta, ou seja, ela usa uma quantidade fixa de espaço da tela, assim como outras unidades menos utilizadas na web (milímetros, polegadas, pontos, etc.). Com a popularização dos tablets, smartphones e outras telas de diferentes tamanhos, não se recomenda o uso das medidas absolutas. Para melhor ajustar o conteúdo em vários formatos de tela diferentes, recomenda-se o uso de medidas relativas. Existem diversas medidas relativas, mas vamos focar nas três principais para construção de layouts: a porcentagem, o `em` e o `rem`. Mais informações sobre essas e outras unidades em (CSS VALUES AND UNITS, s.d.). Uma lista concisa com unidades absolutas e relativas pode ser consultada em (CSS UNITS, s.d.). A seguir, explicamos as principais unidades de medida relativa.

Porcentagem

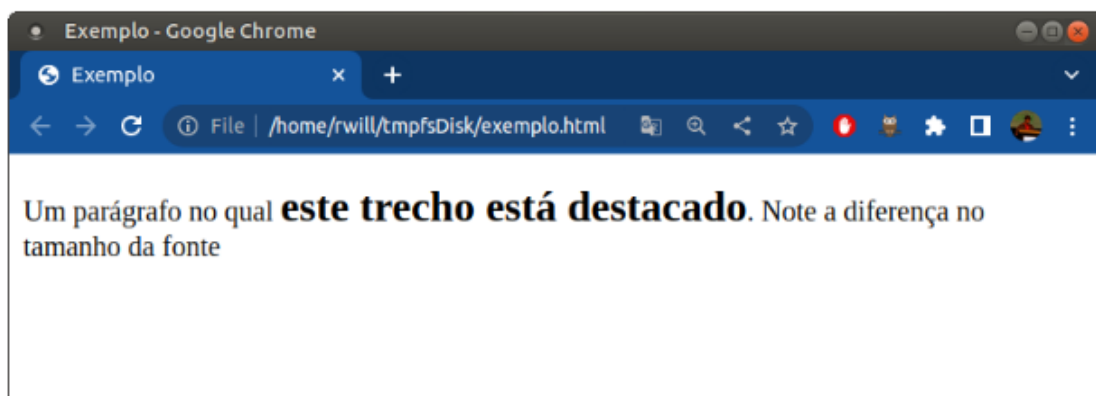
A medida de porcentagem é sempre relativa à mesma medida no elemento pai. Na Figura 5.1 temos um exemplo, onde o tamanho da fonte da tag `` dentro de um parágrafo (elemento pai) será de 150%. Por padrão, os navegadores usam 16px para definir o tamanho da fonte (este é o tamanho do texto do parágrafo). Sendo assim, o texto dentro da tag `` terá tamanho 150% maior (ou seja, 24px). Caso o tamanho da fonte do parágrafo seja modificado para 20px, por exemplo, o trecho destacado pelo `` tem que continuar 150% maior, e assim terá seu tamanho recalculado para 30px. Esse comportamento também vale para qualquer propriedade que envolva medidas (como `width` e `height`).

```
CSS

.destacado {
  font-size: 150%;
}

HTML

<p>Um parágrafo no qual <strong class="destacado">este trecho está destacado</strong>. Note a diferença no tamanho da fonte</p>
```



em e rem

As medidas em e rem se popularizaram com o conceito de sites responsivos, principalmente pensando nos dispositivos móveis. A ideia é que ambas as medidas são relativas, similares a porcentagem, mas ao invés de ser relativa à medida em que a unidade está sendo aplicada, ambas são sempre relativas ao tamanho da fonte. Mesmo que usemos a unidade no width, ela será calculada referente ao tamanho da fonte e não da medida width. A diferença entre ambas, é que o em é relativa ao tamanho da fonte do próprio elemento, já a rem é relativa ao tamanho da fonte da tag raiz (<html>).

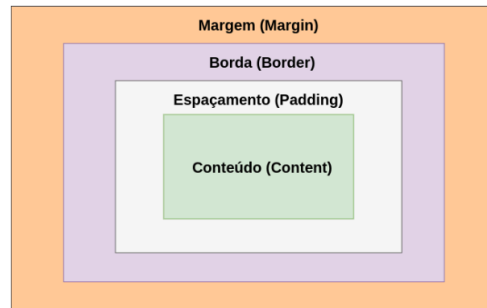
Ambas unidades são interessantes pois conseguimos fazer com que todas as medidas respeitem a mudança do tamanho da fonte do navegador, seja por pessoas que preferem (ou precisam) de letras maiores ou em dispositivos móveis que precisam aumentar o tamanho da fonte (em pixels) para conseguir mostrar seus textos. O cálculo é relativamente simples: 1em equivale a 1 vez o tamanho da fonte do elemento (por padrão, 16px), 2em equivale a 2 vezes o tamanho (por padrão, 32px) e assim por diante. O rem calcula o tamanho de forma similar, mas olhando para o tamanho da fonte do navegador (que pode ser mudado colocando um font-size na tag <html>).

BoxModel

Além das medidas width e height, os elementos HTML ainda possuem outras medidas que afetam a quantidade de espaço que eles ocupam. Todas essas medidas juntas formam o que chamamos de modelo caixa (BoxModel). Este modelo consiste em algumas regiões que formam camadas em volta do elemento HTML em questão, conforme mostra a Figura 5.2.

A parte mais interna se refere ao espaço que o conteúdo do elemento ocupa (o texto dentro de um parágrafo, ou a imagem definida pela tag , por exemplo). Logo após temos uma medida de preenchimento (padding),

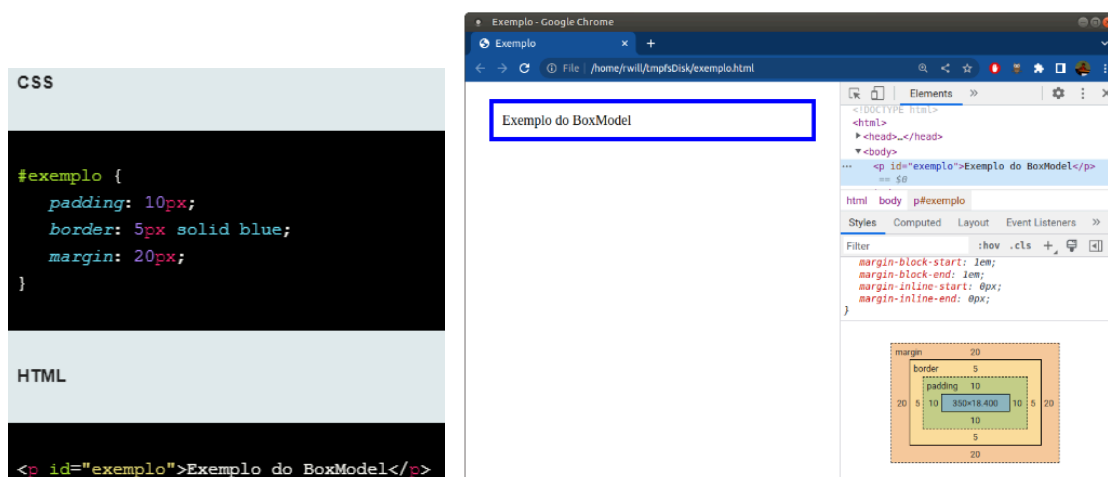
que é um espaço que se cria entre o conteúdo do elemento e seu limite. Neste limite, podemos criar uma borda (border) que ocupa algum espaço também. Fora da borda (e agora fora do elemento), podemos criar um último espaço chamado de margem (margin), que “afasta” o elemento dos demais. Todas essas medidas podem ser definidas com um tamanho geral, o que afeta todos os 4 lados de maneira igual, ou por direção (top, bottom, right, left), dando origem a várias outras propriedades derivadas, como por exemplo: margin-top, margin-bottom, margin-right, margin-left, padding-top, padding-bottom, padding-right, padding-left, border-top-width, border-bottom-width, border-right-width, border-left-width.



A Figura 5.3 mostra um exemplo do uso do box model. Note que a propriedade border, em especial, necessita que 3 valores sejam definidos: a espessura da borda, o estilo da borda (sólida), e a sua cor. Podemos também visualizar como o box model foi calculado ao redor de um determinado elemento, como mostrado ao inspecionar o parágrafo (canto inferior direito) do Chrome.

Outro aspecto importante ao explorar o box model é que, quando definimos a altura e largura de um elemento usando as propriedades height e width, por padrão esses valores serão aplicados somente na região do conteúdo, e não em toda a caixa ao redor do elemento. Isso significa que quando adicionamos espaçamento interno (padding), borda e margem, o elemento será maior do que o que foi definido nas propriedades height e width. Para que toda a caixa fique do tamanho exato definido nessas propriedades, é necessário usar a propriedade CSS box-sizing com o valor border-box. Os valores possíveis dessa propriedade são:

- **content-box:** valor padrão, conforme especificado pela norma CSS. As propriedades altura e largura (height e width) são medidas incluindo só o conteúdo do elemento, sem considerar padding, border, margin.
- **border-box:** indica que o padding e border serão incluídos no cálculo das medidas de altura e largura. Detalhe importante: a propriedade margin não é considerada neste cálculo!



Display

Como visto anteriormente, alguns elementos do HTML naturalmente fazem o conteúdo se separar verticalmente, quebrando linhas e jogando tudo para baixo do próprio elemento. Outros não atrapalham o fluxo do texto, se ajustando ao tamanho do conteúdo que estão marcando. A propriedade do CSS que comanda esse efeito chama-se `display`. Temos dois valores básicos para o `display` que quase todas as tags possuem por padrão: `inline` e `block`.

Elementos que possuem `display inline` são aqueles que não interrompem o fluxo da informação, sem nenhum tipo de quebra de linha, e possuem seu tamanho ajustado de acordo com o tamanho do seu conteúdo apenas. Exemplos desses elementos são os criados com as tags ``, `<a>` e ``. Uma característica do `inline` é que ele não permite alteração dos valores de `width` e `height`. Qualquer alteração nessas propriedades não reflete no tamanho do elemento, pois ele continua ocupando apenas o tamanho do seu conteúdo.

Já elementos do tipo `block` ocupam todo espaço lateral disponível no navegador, independente do tamanho do seu conteúdo, jogando todo o resto do documento para baixo dele. Exemplos são os elementos com tag `<p>`, `<div>` e `<h1>` (até `<h6>`). O `display block` permite alterar a altura e largura do elemento (através das propriedades `height` e `width`), mas mesmo que haja espaço à direita do elemento (com a diminuição da largura), a região horizontal fica reservada para ele, jogando os outros elementos para baixo dele.

Existem outros dois tipos de `display` muito usados que são o `inline-block` e o `none`. O valor `inline-block` é uma mistura dos dois primeiros apresentados. Ele por padrão se ajusta ao tamanho do conteúdo do elemento (como o `inline`), mas permite alterar os valores de largura e altura livremente (como o `block`), liberando espaço à direita para outros elementos. Já o valor `none` acaba escondendo o elemento por completo, fazendo com que ele não ocupe espaço algum (mas ainda existe no navegador), mesmo que alteremos os valores de altura e largura, nada muda no espaço ocupado.

Outros detalhes e valores podem ser vistos em (CSS DISPLAY PROPERTY, s.d.). Dois valores modernos que auxiliam na construção de novos layouts são o `flex` e o `grid`, mas explicá-los foge do escopo deste material.

Você quer ler?

Os valores mais modernos de `display`, o `flex` e o `grid` foram criados para facilitar na construção de layouts fluídos em uma e duas dimensões, respectivamente. Eles são a chave dos layouts fluídos e responsivos mais recentes. O problema é que ambos requerem estudos específicos, pois com eles vem uma lista grande de outras propriedades que fazem parte da especificação, recomendamos a leitura dos seus respectivos tutoriais: <https://css-tricks.com/snippets/css/a-guide-to-flexbox> e <https://css-tricks.com/snippets/css/complete-guide-grid>.

Position

Outra propriedade que permite alterar a disposição dos elementos é a `position`. Por padrão, os elementos HTML são posicionados um após o outro, seguindo a ordem de declaração no arquivo e o `display` definido para cada elemento. Esse posicionamento pode ser mudado pela propriedade `position` que possui os seguintes valores possíveis:

- **static:** valor padrão. Faz com que o elemento fique fixo na sua posição definida no arquivo html;
- **relative:** permite posicionar o elemento em relação ao pai dele;
- **fixed:** posição fixa em relação à tela do navegador;

- **absolute:** permite posicionar o elemento em relação a um pai que não seja static;
- **sticky:** se a posição do scroll do navegador está antes do elemento, ele se comporta como relative; se chegou no elemento ou passou dele, comporta-se como fixed.

Para qualquer valor diferente do static, podemos definir onde o elemento deverá ficar posicionado usando uma distância das quatro direções possíveis: top, right, left e bottom. Por exemplo, se colocamos position: relative e as propriedades top: 10px, o elemento fica a 10 pixels abaixo da sua posição original (CSS POSITION PROPERTY, s.d.).

Media Query

O Media Query permite criar layouts mais responsivos. Através dele é possível aplicar certos estilos para determinadas telas ou situações. Trata-se de uma maneira de definir uma condição para aplicação de estilos CSS, que só será verdadeira quando o dispositivo possuir as capacidades definidas na regra do Media Query. Por exemplo, podemos dispor um menu em locais diferentes, a depender se o usuário está acessando o site a partir de um computador ou de um celular, de maneira que o espaço disponível na tela seja melhor aproveitado para exibir o conteúdo principal da página.

Usamos a notação @media media_type (media_features) para passar ao navegador as condições de aplicação do estilo. O media_type é um identificador do que vai ser testado. Por padrão, o navegador testa todas, mas as opções mais comuns são: screen (tela do aparelho), print (para impressoras) e tv (para aparelhos de TV).

As media_features são capacidades a serem testadas no tipo de mídia especificado. É uma lista relativamente extensa, mas alguns comuns são: max-width (testa se é menor ou igual a uma unidade de tamanho), min-width (testa se é maior ou igual a uma unidade de tamanho), e orientation (a orientação da tela: portrait, modo retrato, ou então landscape, modo paisagem). Então, a Media Query @media screen and (max-width: 550px) { ... } terá seus estilos aplicados se o tamanho atual da janela do navegador for até 550px.

```
CSS

body {
  background-color: bisque;
}

div.nao-importante {
  background-color: yellow;
  height: 80px;
}

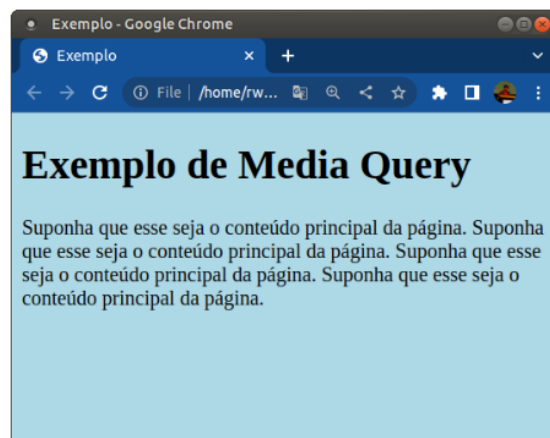
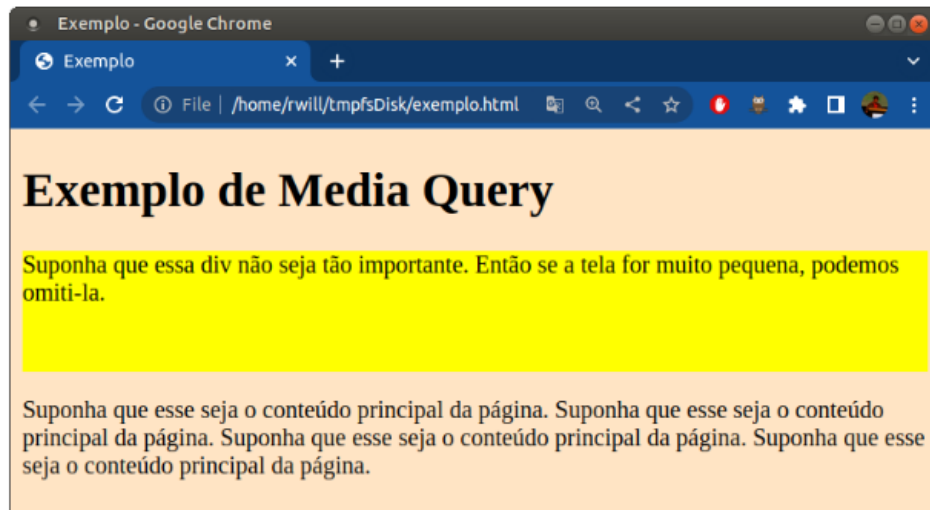
@media screen and (max-width: 600px) {
  body {
    background-color: lightblue;
  }

  div.nao-importante {
    display: none;
  }
}
```

```
HTML

<h1>Exemplo de Media Query</h1>
<div class="nao-importante">Suponha que essa div não seja tão importante. Então se a tela for muito pequena,
podemos omiti-la.</div>
<p>Suponha que esse seja o conteúdo principal da página. Suponha que esse seja o conteúdo principal da
página. Suponha que esse seja o conteúdo principal da página. Suponha que esse seja o conteúdo principal da
página.</p>
```

Na Figura 5.4 temos um exemplo do uso do Media Query. Por padrão, a cor de fundo da página é o valor `bisque`, e a `div` com a classe “não-importante” aparece na página. Mas quando o tamanho da tela diminui (largura menor ou igual a 600px), valem as regras do Media Query: a cor de fundo da página muda para o valor `lightblue`, e a propriedade `display` da `div` muda para o valor `none`, o que faz com que a `div` desapareça. Faça o teste: rode o código do exemplo e modifique a largura da tela.



JavaScript

Juntas, as tecnologias HTML e CSS permitem mostrar qualquer conteúdo gráfico no navegador da forma que desejarmos, com o poder semântico do HTML e a estilização do CSS para posicionarmos e decorarmos os elementos da melhor maneira. Entretanto, quando desejamos adicionar comportamentos mais específicos de acordo com a interação do usuário no navegador, é necessário algo mais poderoso. O JavaScript é uma linguagem de programação inventada para rodar diretamente no navegador e permitir que ele execute tarefas mais complicadas que não são possíveis de implementar somente com HTML e CSS.

O JavaScript foi criado na década de 1990 como uma forma de trazer inteligência às páginas da web (DEGROAT, 2019). No início, era usado essencialmente para validar formulários antes destes serem submetidos ao servidor. Isto é, o JavaScript nasceu como uma tecnologia do lado do cliente: uma linguagem de programação, cujo interpretador é o navegador de Internet. Com o passar dos anos o JavaScript se popularizou, chegando agora a

todas as plataformas, e hoje em dia é possível utilizá-lo inclusive para programar o lado do servidor com o NodeJS.

Com vários navegadores diferentes, além do NodeJS, implementando a sua própria versão do JavaScript, foi necessário criar uma especificação de como a linguagem deve ser e se comportar, chamado de ECMAScript. Essa especificação é revisada anualmente (em geral) trazendo mais padronização e funcionalidades para a linguagem.

Características do JavaScript

O JavaScript é uma linguagem de programação relativamente simples e dinâmica, com poucos entraves para começar a programar. As plataformas que deram início a sua utilização são os navegadores, mas hoje o JavaScript é amplamente utilizado no servidor (e outros dispositivos). Dizemos que o JavaScript é uma linguagem interpretada, dinamicamente e fracamente tipada.

Ser uma linguagem interpretada significa que ela precisa de um programa para executar suas instruções, que geralmente é o navegador, mas também pode ser motor do NodeJS. Isso permite que pequenas alterações no nosso código já sejam executadas assim que possível. O código até acaba sendo compilado em algum estágio da execução, mas isso fica completamente invisível para os desenvolvedores, por isso dizemos que ela se comporta como uma linguagem interpretada (igual ao Python).

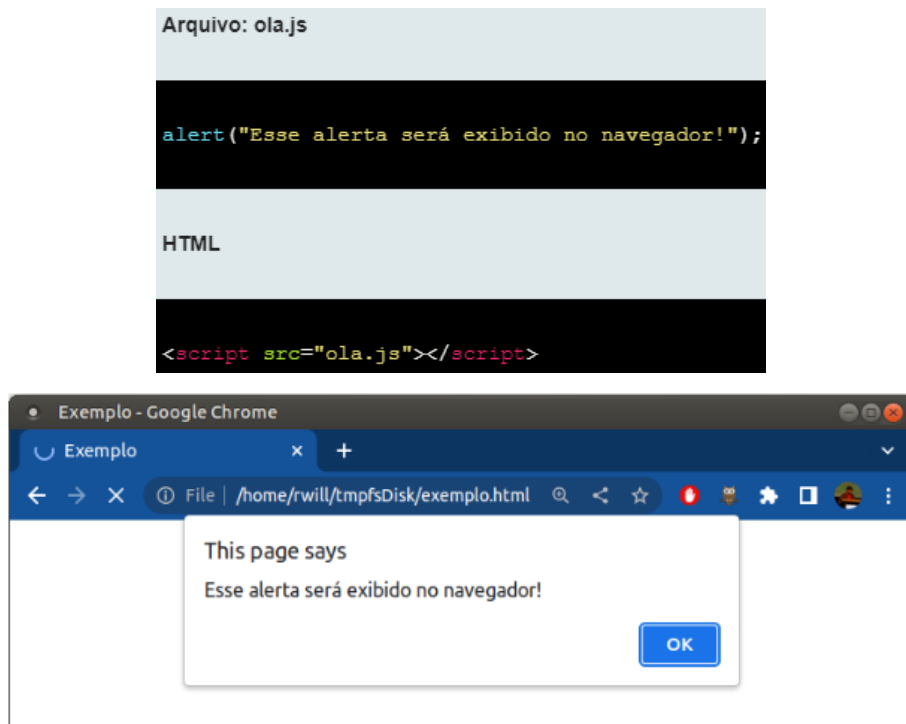
Ao declarar uma variável em JavaScript, é necessário apenas informar o nome da variável e o seu valor, enquanto que o seu tipo não é explicitamente declarado: ele é deduzido através do valor. Linguagens com essa característica são ditas dinamicamente tipada. Além disso, operações com tipos distintos são permitidos no JavaScript, pois ele converte um dos valores da operação automaticamente para um tipo compatível com o outro operando, permitindo que a operação seja realizada sem erros. Linguagens com essa característica também são ditas fracamente tipadas. Essa é uma diferença do Python, onde operações com tipos muito distintos (inteiro e string, ou string e lista, por exemplo) resultam em erros de execução (uma vez que Python é uma linguagem fortemente tipada).

Executando o primeiro código JavaScript

Em nosso curso, vamos nos ater apenas aos navegadores, mas boa parte do código que vamos escrever funcionará no NodeJS também. Para ser executado nas nossas páginas, o código JavaScript precisa estar contido entre as tags `<script></script>`, ou pode ser escrito em um arquivo próprio e carregado pelo atributo `src`. A tag `script` pode aparecer em qualquer parte do arquivo HTML, ou seja, ela pode estar tanto no `head` da página, como também no `body`.

Uma função simples para testarmos o primeiro exemplo é a função `alert()`, que exibe uma mensagem (uma string passada por parâmetro) no navegador. Você pode escrever o seguinte código em um arquivo HTML: `<script>alert("Esse alerta será exibido no navegador!");</script>`. Teste e veja o que acontece!

A Figura 6.1 mostra como podemos executar o JavaScript a partir de um arquivo externo. Neste exemplo, o conteúdo do arquivo contém apenas uma linha, que chama a função `alert()` com a mesma mensagem anterior. Note que no HTML, mesmo carregando um script externo, ainda precisamos abrir e fechar a tag `script`, sem adicionar qualquer comando entre as tags de abertura e fechamento. Similar ao CSS, não é recomendado colocar códigos JavaScript no HTML, salvo algumas exceções.



Regras de sintaxe

O JavaScript possui uma sintaxe similar a de linguagens como: Java, C#, e de certa forma C e PHP. No entanto, diferente dessas linguagens, no JavaScript não é obrigatório colocar ponto e vírgula no final dos comandos (embora seja altamente recomendável para não criar o hábito de esquecer do ponto e vírgula quando você for programar em linguagens em que ele é obrigatório). Em JS, escrever `alert("Olá");` e `alert("Olá")` produz o mesmo resultado.

Outro aspecto importante é que a indentação não faz diferença para o interpretador do JavaScript, embora sirva para organizar o seu código e melhorar sua legibilidade. Comentários no código podem ser feitos de duas formas, como mostrado na Figura 6.2.

```
JavaScript

// Comentário de uma única linha

/*
Comentário de
múltiplas linhas!
*/
```

Declaração de variáveis

Para declarar uma variável, escrevemos o nome que queremos dar a ela à direita de uma palavra chave, que pode ser `var`, `let` ou `const` (VARIABLES, s.d.). Basicamente, as palavras chave `var` (maneira antiga, define variáveis sem escopo de bloco) e `let` (maneira mais nova e recomendada, define variáveis com escopo de bloco)

são usadas para criar variáveis que podem ter o seu valor alterado ao longo da execução do programa. Já a palavra `const` cria variáveis constantes, as quais obrigatoriamente precisam de um valor atribuído em sua declaração e não poderão mais ser alteradas ao longo do programa.

Como dito anteriormente, não há declaração de tipo de maneira explícita. O JavaScript deduz o tipo da variável através do seu valor. A Figura 6.3 mostra exemplos de declarações de variáveis, com ou sem valor.

JavaScript

```
var nome = "João";
let texto = 'Strings podem ser escritas com aspas simples ou duplas';
const nota_maxima = 10;
let valor_total = 35.23;
let aprovado = true;
let variavel_sem_valor_inicial;
```

Declaração de variáveis

O JavaScript oferece cinco tipos de dados primitivos. São eles (ESTRUTURA DE DADOS DO JAVASCRIPT, s.d.):

- `string`
- `number`
- `boolean`
- `null`
- `undefined`

A `string` no JavaScript funciona como na maioria das linguagens mais modernas. É uma cadeia de caracteres entre aspas (duplas ou simples), utilizadas para representar textos quaisquer.

Os números (inteiros ou de ponto flutuante) fazem parte do mesmo tipo: `number`. O JavaScript distingue se o número é inteiro ou decimal, e isso fica transparente para o desenvolvedor. O tipo `number` é representado internamente com 64 bits, usando o padrão IEEE 754.

Valores booleanos são representados pelas palavras chave `true` ou `false` (com todas as letras minúsculas). Um `boolean` ocupa 1 byte na memória e são usados geralmente em estruturas condicionais da linguagem.

No JavaScript temos dois tipos “vazios” que podem ser usados, mas com características diferentes: `undefined` e `null`. O `undefined` é o valor padrão quando declaramos variáveis sem valor atribuído (exemplo: `let nome;`). Usamos ele para representar o vazio propriamente dito no JavaScript. Já o `null` é geralmente utilizado apenas pelos desenvolvedores, a linguagem não devolve esse valor em momento algum, e em geral representa um retorno de função que não funcionou ou o esvaziamento de uma variável.

Operações

O JavaScript implementa várias das operações matemáticas e booleanas clássicas, bem como as expressões de comparação. A Tabela 6.1 resume vários deles (JAVASCRIPT OPERATORS REFERENCE, s.d.).

Operador	Descrição	Exemplo	Resultado
+	Soma	3 + 4	7
-	Subtração	8 - 2	6
*	Multiplicação	3 * 4	12
/	Divisão	5 / 2	2.5
%	Módulo (resto da divisão)	5 % 2	1
++	Incremento	6++	7
--	Decremento	7--	6
==	Igualdade	7 == 7	true
!=	Não igualdade	7 != 8	true
>	Maior	9 > 1	true
>=	Maior ou Igual	9 >= 9	true
<	Menor	-1 < 3	true
<=	Menor ou Igual	3 <= 3	true
&&	Lógico E	true && false	false
	Lógico OU	true false	true
!	Lógico NÃO	!true	false

Por conta da maneira que o JavaScript interpreta os valores (devido a sua tipagem fraca), podemos ter igualdades com resultados não esperados. Por exemplo, comparar 2 == "2" (2 inteiro igual a 2 string) resulta em true. Isso se dá pelo fato do JavaScript assumir algum tipo compatível quando realizamos operações com tipos diferentes (em geral string). Se quisermos ter certeza de que o valor e o tipo estão iguais, usamos a igualdade estrita: 2 === "2", que neste caso retorna false.

Array

O JavaScript possui nativamente um objeto para guardar conjuntos de valores indexados pela sua posição, chamado de array (vetor). Para criar um novo vetor, basta declará-lo abrindo e fechando colchetes, semelhante às listas do Python. A Figura 6.4 mostra um exemplo onde definimos dois vetores: um vazio (sem elementos), e outro inicializado com quatro elementos. Vetores em JavaScript podem ter elementos de quaisquer tipos de dados dentro da sua estrutura, sejam eles primitivos ou outros objetos (outros vetores, por exemplo).


```

JavaScript

// Definição de um array vazio:
let vetor_vazio = [];

// Definição de um array povoado com 4 elementos:
let vetor = [1, 2, 3, 4];

// Modificando o elemento no índice 2. O array ficará: [1, 2, 9, 4]
vetor[2] = 9;

/* O JS permite adicionar elementos novos desta maneira. O array ficará: [1, 2, 9, 4, 7] */
vetor[4] = 7;

// Acrescenta 6 ao final do vetor: [1, 2, 9, 4, 7, 6]
vetor.push(6);

// Remove o último elemento do vetor: [1, 2, 9, 4, 7]
vetor.pop();

// O atributo length retorna o tamanho do vetor
let tamanho = vetor.length;

```

Quando é necessário acessar algum elemento, é possível acessá-lo usando o índice da sua respectiva posição (iniciando pelo 0). Por exemplo, se quisermos trocar o valor de uma posição específica, podemos fazer a atribuição pelo índice: `vetor[2] = 9`, como mostrado na Figura 6.4. Podemos inclusive incluir um valor em um índice novo, como na atribuição `vetor[4] = 7` do nosso exemplo. O tamanho atual do vetor é dado pelo atributo `length`, ou seja, ao final do exemplo, a variável `tamanho` receberia o valor 5.

Por serem objetos complexos, os arrays possuem diversos métodos que podem ser utilizados para fazer operações neles, como por exemplo:

- `push()`: coloca o elemento passado por parâmetro no fim do vetor;
- `pop()`: remove o último elemento do vetor e retorna o valor removido;
- `concat()`: retorna um novo vetor concatenando o vetor de origem com o vetor passado como parâmetro no método;
- `indexOf()`: retorna o primeiro índice de ocorrência do valor passado por parâmetro. Caso o valor não exista no vetor, retorna -1;

Mais detalhes e métodos podem ser consultados em (ARRAY, s.d.).

Object

O JavaScript também permite a criação de objetos similares aos dicionários de outras linguagens de programação (como por exemplo: Python, C#), os quais permitem armazenar a informação como pares de chave e valor (OBJECT, s.d.). Esse tipo de estrutura permite buscar a informação pela sua chave, ao invés da posição (índice), como é o caso nos arrays. Esse tipo de dado é chamado em JS de Object.

Uma variável do tipo object é definida usando chaves pareadas: `let objeto = {}`. Internamente, seu funcionamento é baseado nas tabelas de espalhamento (hash tables), associando chaves pesquisáveis aos valores (como se fossem índices de vetores). As chaves podem ser qualquer string válida e os valores podem ser de qualquer tipo do JavaScript. A Figura 6.5 mostra exemplos de como criar, ler e modificar valores de um objeto.

JavaScript

```
// Definindo um objeto vazio:
let objeto_vazio = {};

// Definindo um objeto populado:
let aluno = {"nome": "João", "matricula": 1234, "notas": [6.0, 8.0, 10.0]};

// Imprime 1234 no console:
console.log(aluno["matricula"]);

// Também imprime 1234 no console:
console.log(aluno.matricula);

// Muda o nome do aluno para "Maria":
aluno["nome"] = "Maria";

// Também muda o nome do aluno para "Pedro":
aluno.nome = "Pedro";

// Criando um novo atributo:
aluno["cpf"] = 98765432100;

// Também cria um novo atributo:
aluno.formado = false;
```

Podemos acessar ou atribuir novos valores internos de maneira similar ao array: `aluno["matricula"]` retorna o valor 1234. Também é possível usar a notação de ponto, como se estivéssemos acessando o atributo de um objeto. Ou seja, `aluno.matricula` também retorna 1234. Uma chave já existente pode ter seu valor associado modificado de maneira similar, como em: `aluno["nome"] = "Maria"`, ou com a notação de ponto: `aluno.nome = "Pedro"`. Por fim, novas chaves podem ser definidas, sendo necessário apenas fazer uma atribuição a uma chave inexistente (exemplo na Figura 6.5).

É comum haver confusão entre o tipo `object` e strings no formato JSON (JavaScript Object Notation). O JSON se refere a uma notação de objetos em formato de texto que, por sua simplicidade, é amplamente utilizada para passar informações pela rede. Ou seja, um JSON é apenas um texto em um determinado formato, o `object` é a representação binária de objetos gerais no JavaScript (na memória RAM).

Funções

Assim como outras linguagens de programação, podemos definir funções para modularizar nossos programas e reaproveitar código. No caso do JavaScript, as funções são cidadãos de primeira classe (first-class citizen), que significa que podemos fazer operações em cima delas, em específico atribuí-las a variáveis da maneira que quisermos.

Comumente declaramos funções no JavaScript de duas maneiras: com ou sem nome (funções anônimas). A primeira maneira usamos a palavra chave `function`, o nome que queremos dar a função (ex: `calcular`), a lista de parâmetros que a função espera entre parênteses (vazio se não houver) e o par de chaves (`{ }`) para delimitar o bloco da função. Na segunda forma, anônima, declaramos a função sem o nome dela, mas associando-a imediatamente a uma variável para não perder a sua referência. A Figura 6.6 mostra exemplos das duas formas de definição de funções. Ambas são equivalentes, mas se você rodar o programa passo a passo, notará que a atribuição em `const adicao = function (x, y)` será executada como uma atribuição de variável qualquer.

JavaScript

```
// Definindo uma função com nome:
function somar(a, b) {
  let c = a + b;
  return c;
}

// Definindo uma função sem nome:
const adicao = function (x, y) {
  let z = x + y;
  return z;
}

// Chamando as funções e guardando o seu resultado:
let r1 = somar(5, 3);
let r2 = adicao(5, 3);
```

Para executar uma função, devemos usar o nome dela para chamá-la e passar a lista de parâmetros entre parênteses. Um detalhe interessante é que, diferente de outras linguagens, não é obrigatório passar todos os parâmetros, pois o JavaScript colocará o valor undefined nos parâmetros faltantes.

Funções em JS também não declaram o tipo de retorno e nem precisam ter retorno. Se desejamos que uma função retorne algum valor, devemos usar a palavra chave return, o que fará com que a função pare imediatamente naquele ponto e retorne o valor. Caso não haja nenhum retorno, a função retornará o valor undefined.

Estrutura condicional: if-else

O JavaScript possui também a estrutura de decisão clássica if-else (mais detalhes em IF...ELSE - JAVASCRIPT, s.d.). Ao criar uma condicional com a palavra chave if, a condição lógica deve sempre estar entre parênteses, enquanto que os comandos que serão executados caso a condição seja verdadeira devem estar dentro do bloco definido por chaves ({ ... }). A Figura 6.7 ilustra um exemplo de bloco if-else.

JavaScript

```
/* Programa que imprime "Aprovado!" ou "Reprovado!" no console,
a depender da situação do aluno. Ao testar, modifique as duas
variáveis abaixo para entender o funcionamento do bloco if-else: */
let nota = 6;
let frequencia = 75;

if (nota >= 6 && frequencia >= 75) {
  // Bloco que será executado caso o if seja true
  console.log("Aprovado!");
} else {
  // Bloco que será executado caso o if seja false
  console.log("Reprovado!");
}
```

Diferente do Python (e de outras linguagens como PHP, por exemplo), o JavaScript não possui o comando `elif`. Quando necessário, é possível concatenar várias condições extras usando as palavras chave `else if` (com a condição lógica dentro de parênteses e chaves para definir o bloco) e um único `else` (sem parênteses e com chaves para definir o seu bloco). Um exemplo dessa estrutura encontra-se na Figura 6.8.

```
JavaScript

let temperatura = 20;

if (temperatura <= 0) {
    console.log("Água em estado sólido");
} else if (temperatura < 100) {
    // Esse bloco simula um "elif"
    console.log("Água em estado líquido");
} else {
    console.log("Água em estado gasoso");
}
```

Assim como no Python, as condições booleanas também podem tratar de valores vazios ou não vazios. Ou seja, `null`, `undefined`, strings vazias e o valor numérico `0` são considerados falsos, enquanto que todos os outros valores são considerados verdadeiros (por estarem "preenchidos").

Estruturas de repetição: `while`, `for`, `do-while`

O JavaScript define três estruturas de repetição: `while`, `for` e o bloco `do-while`. Geralmente usamos o `while` quando a condição de parada é indefinida, ou seja, não tem um número certo de repetições. Assim como no comando `if`, devemos escrever a condição lógica do `while` sempre entre parênteses e o código que será repetido deverá estar dentro do seu bloco, definido entre chaves (WHILE - JAVASCRIPT, s.d.). O bloco `do-while` é semelhante, mas no `while` a condição lógica é testada antes de entrar no bloco, enquanto que no `do-while` ela só é testada no final. Isto significa que o código do bloco `do-while` será executado pelo menos uma vez. A Figura 6.9 mostra um exemplo com a sintaxe das duas estruturas.

```
JavaScript

let i = 1;

while (i <= 5) {
    /* Este bloco será repetido enquanto i for menor ou igual a 5 */
    console.log(i); // imprime o valor de i
    i++; // incrementa i em 1 unidade
}

i = 1;
do {
    /* Esse bloco será executado pelo menos uma vez, e repetido enquanto i for menor ou igual a 5 */
    console.log(i); // imprime o valor de i
    i++;
} while (i <= 5);
```

Já o laço definido pelo comando `for` pode ser usado de duas maneiras: a clássica (parecida com a do Java) e o `for of` (parecida com a do Python). O `for` clássico possui três entradas entre parênteses e separadas por ponto e vírgula: inicialização da variável de controle, condição de parada, e tamanho do passo a cada repetição (FOR - JAVASCRIPT, s.d.). Já o `for of` é mais simples (FOR...OF - JAVASCRIPT, s.d.), criamos uma variável de controle que assumirá todos os valores do array, em ordem. A Figura 6.10 ilustra os dois exemplos do uso do laço `for`, que percorrem e imprimem todos os elementos da lista definida pela variável `nome`.

```
JavaScript

let nomes = ["Ana", "João", "Maria", "Pedro"];

/* O código a seguir percorre toda a lista e imprime cada um de seus valores: */
for (let i = 0; i < nomes.length; i++) {
    console.log(nomes[i]);
}

/* No código a seguir, a variável nome assume cada um dos nomes contidos na lista: */
for (let nome of nomes) {
    console.log(nome);
}
```

Você sabia?

Por conta da maneira que o JavaScript foi projetado, a questão de ser fracamente tipado, o JavaScript possui alguns comportamentos curiosos (como o `5 == "5"` ser `true`), mas alguns são mais curiosos e confusos (mas com explicação na concepção da linguagem), como o fato que o operador `typeof` (que retorna o tipo de um valor/variável) do `null` ser `object`. Veja mais em: <https://imasters.com.br/desenvolvimento/javascript-o-lado-obscurito>.

Manipulação do DOM

Vimos como o JS funciona em termos de estrutura e sintaxe, agora vamos entender como integrá-lo com elementos HTML de um documento Web. Para isso, é necessário compreender a estrutura que o navegador constrói, através do modelo de documento por objetos (DOM), e também as funções que o JavaScript disponibiliza para manipular essa estrutura.

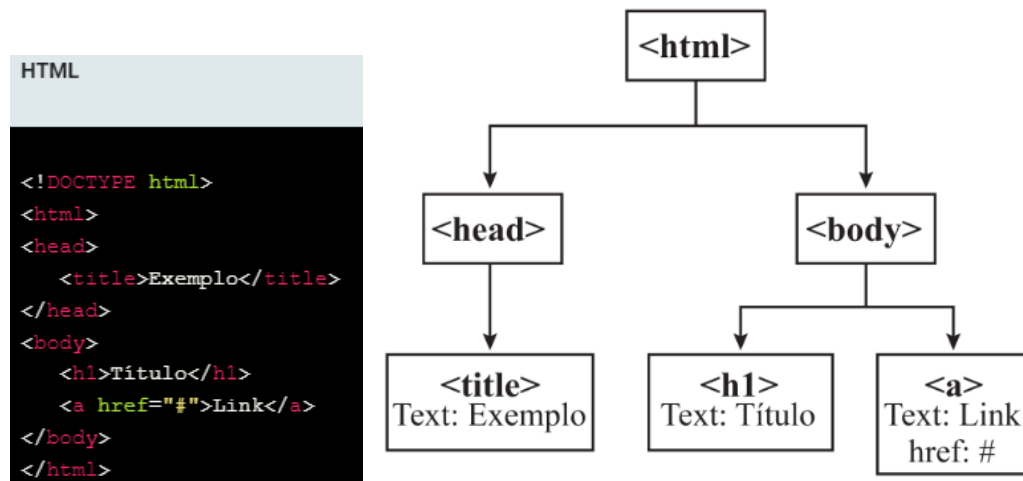
O JavaScript é uma linguagem flexível e dinâmica, e permite que diversas tarefas simples sejam implementadas de maneira rápida e descomplicada. Apesar de hoje em dia ela também poder ser usada para programar o lado do servidor, sua intenção primordial é a de funcionar em conjunto com o HTML e o CSS para trazer capacidades mais modernas à web. Veremos como, a partir de funções nativas do JavaScript no navegador, podemos navegar nos elementos HTML que estão disponíveis e como podemos manipulá-los livremente.

O que é o DOM?

Quando o HTML é carregado pelo navegador após uma requisição, é iniciada a fase de análise e tradução do código HTML (chamado em inglês de HTML parsing), onde o navegador interpreta o código HTML linha a linha, “entendendo” as tags e traduzindo-as para os elementos que aparecerão no navegador (visuais ou de configuração). Na prática, esses elementos são representados por objetos na memória RAM pelo navegador, e como o HTML é intrinsecamente uma representação hierárquica de elementos (uma vez que tags são colocadas

dentro de outras tags), o navegador disponibiliza essa estrutura como uma árvore de objetos contidos na memória. Essa representação em árvore dos elementos HTML de um documento é chamada de Document Object Model (DOM), ou Modelo de Documento por Objetos.

A Figura 7.1 ilustra o exemplo de uma pequena página HTML e sua representação do DOM. Cada caixinha representa um objeto em memória, com suas próprias propriedades e métodos, construídos a partir da representação de suas tags no código HTML.



Esses objetos em memória permitem usar funções e variáveis presentes no JavaScript para navegar nesta árvore de objetos e poder alterá-la da maneira que desejarmos (JAVASCRIPT HTML DOM, s.d.).

Navegando pelo DOM com o JS

No ambiente do navegador, o JavaScript possui algumas variáveis globais disponíveis a todo momento. Uma das mais importantes é aquela que traz a referência ao DOM e sua árvore de elementos: a variável `document`. Tudo que será buscado ou alterado em termos de conteúdo do navegador será feito neste objeto. A seguir, detalharemos os principais métodos internos do objeto `document` para buscar elementos na árvore.

`getElementById`

Esse é o método mais simples para buscar um elemento na árvore de elementos do DOM. Sua sintaxe é `document.getElementById(id)`, onde passamos o id de algum elemento HTML existente na página. O método retorna um objeto com a representação daquele elemento (devendo ser guardado em uma variável), ou o valor `null`, caso não exista nenhum elemento HTML com aquele id.

`getElementsByClassName`

Apesar da facilidade do método `getElementById()`, seu uso nem sempre é possível em determinadas situações, pois implicaria que cada um dos elementos da página tivessem um id atribuído para si. Outra impraticabilidade é o caso em que desejamos recuperar um conjunto de elementos com características similares, mas o `getElementById()` só permite acessar um elemento de cada vez. O método `document.getElementsByClassName(class)` permite selecionar um conjunto de elementos pelo nome da classe a qual eles pertencem.

Note que a palavra “Elements” no nome do método está no plural, pois será retornado um objeto do tipo `HTMLCollection` (similar a um vetor e pode ser tratada como tal). Cada índice corresponde a um elemento HTML, e sua posição no vetor corresponde à ordem em que ele aparece no código HTML.

`getElementByTagName`

Outra maneira de selecionar um conjunto de elementos no JavaScript é através do método `document.getElementsByTagName(tag)`, onde `tag` é o nome da tag que desejamos selecionar. Esse método também retorna um objeto `HTMLCollection` (similar a um vetor), com todos os elementos definidos pela tag informada no parâmetro, na ordem em que eles aparecem no código HTML.

`querySelector`

O método `document.querySelector(seletor)` é uma maneira mais moderna para buscar elementos da árvore do DOM. Ele recebe um seletor CSS por parâmetro, e retorna um objeto que faz referência ao primeiro elemento do código HTML que corresponde ao seletor usado.

`querySelectorAll`

O método `document.querySelectorAll(seletor)` é similar ao `querySelector()`, mas ao invés de retornar a primeira ocorrência, retorna um objeto do tipo `NodeList` (similar a um vetor), contendo referência a todos os elementos na página que correspondem ao seletor passado por parâmetro.

A Figura 7.2 mostra um exemplo com 3 parágrafos e 3 divs em HTML, e um código JavaScript que utiliza os métodos vistos para acessar alguns desses elementos. Com os objetos retornados, podemos usar a linguagem JavaScript para modificar vários aspectos desses elementos (veremos mais adiante).

HTML

```
<p class="importante">Parágrafo 1</p>
<p id="par2">Parágrafo 2</p>
<p class="importante">Parágrafo 3</p>
<div class="importante">Div 1</div>
<div class="comum">Div 2</div>
<div id="div3">Div 3</div>
```

JavaScript

```
/* Retorna um objeto que representa o "Parágrafo 2" */
const objeto1 = document.getElementById("par2");

/* Retorna um vetor com os objetos: "Parágrafo 1", "Parágrafo 3" e "Div 1", nessa ordem */
const vetor1 = document.getElementsByClassName("importante");

/* Retorna um vetor com os objetos: "Div 1", "Div 2" e "Div 3", nessa ordem */
const vetor2 = document.getElementsByTagName("div");

/* Retorna um objeto que representa o "Parágrafo 1" (primeira ocorrência de um parágrafo com a classe "importante") */
const objeto2 = document.querySelector("p.importante");

/* Retorna um vetor com todos os objetos que possuem a classe "importante": "Parágrafo 1", "Parágrafo 3" e "Div 1", nessa ordem */
const vetor3 = document.querySelectorAll(".importante");
```

Independente do método utilizado, é importante notar que para o JavaScript conseguir selecionar um elemento específico (ou vários), esses elementos precisam existir e estarem carregados no DOM. Mesmo parecendo óbvio, esse aviso se deve ao fato de que o JavaScript pode ser executado antes de o HTML ter sido carregado, podendo acarretar em erros estranhos. Isso se deve a maneira como o navegador interpreta o HTML e o JavaScript.

Execução do JS

Como dito anteriormente, quando o navegador recebe o documento HTML, ele inicia a interpretação (parsing) do HTML linha a linha, entendendo os elementos HTML. Mas quando o HTML chega em algum recurso como arquivos CSS ou JS, essa tradução é pausada enquanto o arquivo é baixado e seu conteúdo também traduzido (parsed) e executado (no caso do JS).

Como tanto os arquivos JS quanto CSS são elementos de configuração do navegador (para a correta visualização e execução do site), a recomendação é colocá-los no head do documento HTML. Isso é ideal para o CSS, pois permite que as regras do CSS sejam entendidas antes de mostrar os elementos, aplicando o estilo no momento correto. Já para o JavaScript, isso traz um problema: como o arquivo é baixado e executado ainda durante a tradução do head, não há conteúdo no body para ser procurado pelo JavaScript, resultando em buscas no DOM que devolvem vetores vazios ou null em lugares indesejados.

Para contornar esse problema, os programadores usavam uma de duas soluções: colocavam as tags de script no final do body, garantindo assim que todos elementos HTML já tenham sido carregados corretamente, ou associando uma função JavaScript, com a execução do script inteiro, a um evento que o navegador dispara quando o DOM está pronto (veremos mais sobre eventos mais adiante).

Como forma de otimizar o carregamento de scripts, surgiram duas configurações possíveis de serem colocadas na tag script: o `async` e o `defer`.

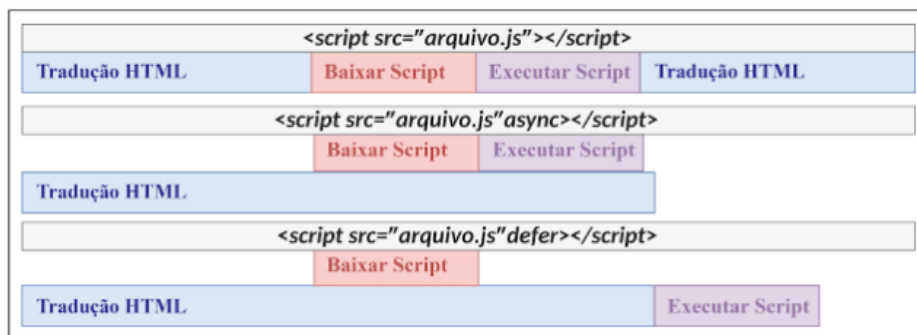
async e defer

Os atributos `async` e `defer` surgiram como forma de otimizar o uso de JavaScripts nas nossas páginas, que se tornaram predominantes na web. Ambos possuem funções diferentes para tipos de scripts diferentes.

O atributo `async` torna todo o processo de baixar e traduzir o JavaScript em um processo assíncrono, ou seja, o navegador faz ambas as coisas ao mesmo tempo em que termina de analisar o código HTML. Neste caso, não há um aumento substancial no tempo de carregamento da página por conta dos scripts (a menos que eles sejam muito grandes). Esse atributo é recomendado para scripts que não precisam do DOM já estar carregado para funcionar, pois é possível que o script encerre sua execução antes que o DOM esteja pronto.

O atributo `defer` faz com que o processo de baixar os arquivos de JavaScript seja assíncrono (assim como o `async`), mas sua tradução e execução são feitas após o carregamento completo do HTML e o DOM esteja pronto. Dessa forma, o navegador garante que scripts que precisam manipular o DOM sejam carregados de maneira mais eficiente (assíncrona), mas sua execução vai ocorrer apenas quando o DOM estiver pronto.

A Figura 7.3 mostra a diferença entre os atributos `async` e `defer`, e como é a ordem de carregamento e tradução do HTML.



Manipulando o DOM com o JS

Vimos que o JavaScript possibilita navegar no DOM usando os métodos `getElementById`, `getElementsByClassName`, `getElementsByTagName`, `querySelector` e `querySelectorAll`. Esses métodos retornam objetos (ou vetores de objetos) que podem ser guardados em variáveis para usarmos posteriormente (Exemplo: `const titulo = document.querySelector("#titulo")`). Com as referências aos objetos, podemos começar a manipular os elementos no DOM.

Todos os objetos retornados nos métodos de busca obedecem a interface `HTMLElement` (HTMLELEMENT, s.d.), garantindo acesso aos atributos presentes no HTML e a métodos específicos. Depois dessa interface, cada um dos elementos HTML possui uma interface própria com seus métodos e atributos específicos (Exemplo: a tag `a` possui a interface `HTMLAnchorElement`).

Toda e qualquer alteração nesses objetos reflete imediatamente no que está representado no DOM, seja de maneira visual (mudar a cor de um parágrafo, por exemplo) ou apenas na parte lógica (associar aquele elemento a um escutador de eventos, como veremos mais adiante).

Acessando e alterando propriedades

Assim como em outras linguagens de programação, os atributos e métodos dos objetos do DOM são acessados pelo operador ponto (.) e o nome do atributo equivalente no HTML. A Figura 7.4 mostra um exemplo de como podemos manipular (ler e atribuir) valores a esses atributos. Note que o comando `console.log(objeto.id)` vai imprimir no console do navegador o valor "titulo", que é o id do elemento. Já o comando `console.log(objeto.title)` imprime uma string vazia, pois originalmente não havíamos definido o atributo `title` na tag de abertura do `h1`.

Ainda com o exemplo da Figura 7.4, note que é possível atribuir valores (strings) a esses atributos. O interessante aqui é que essa ação mudará o elemento `h1` carregado em memória RAM pelo navegador (a versão original salva no disco continuará a mesma, sem modificações). Atribuir uma string ao atributo `objeto.title` fará com que o navegador adicione o atributo `title` no código HTML carregado em memória. Você pode conferir essa mudança ao inspecionar esse elemento no menu "Ferramentas do desenvolvedor", como mostrado na Figura 7.4. O mesmo ocorre quando alteramos o texto do elemento `h1` ao atribuir uma string ao atributo `objeto.textContent`. Isso faz com que o navegador substitua o texto original do `h1` pelo novo texto atribuído pelo JavaScript.

```
HTML

<h1 id="titulo">Um título qualquer</h1>

JavaScript

const objeto = document.getElementById("titulo");

/* Acessando (e imprimindo no console) o valor do atributo id */
console.log(objeto.id);

/* Acessando (e imprimindo no console) o valor do atributo title */
console.log(objeto.title);

/* Atribuindo um valor ao atributo title */
objeto.title = "Esse texto foi atribuído pelo JavaScript";

/* Modificando o texto do h1 */
objeto.textContent = "Esse será o novo texto exibido pelo elemento h1";
```



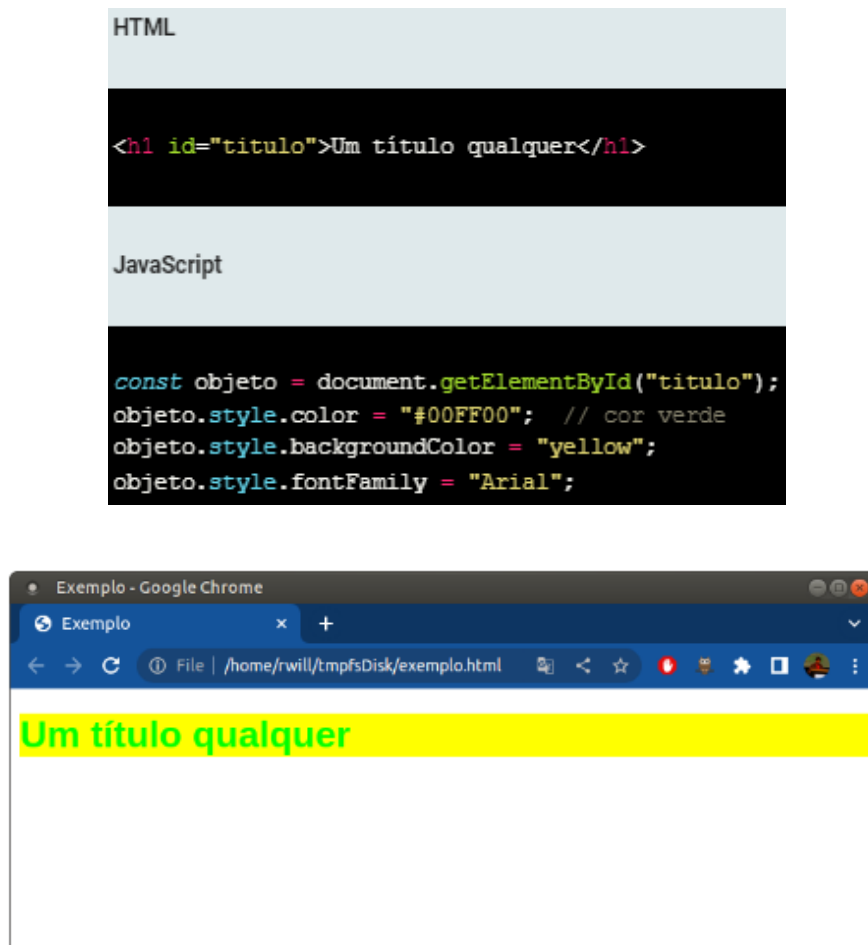
O conteúdo do elemento pode ser obtido e alterado por dois atributos diferentes, a depender do objetivo: `textContent` e `innerHTML`. A diferença mais básica entre ambas, é que o `textContent` ignora tags HTML que estejam no conteúdo, enquanto a `innerHTML` não. Por exemplo, no elemento `<p>Isso é um parágrafo!</p>` o `textContent` voltaria o texto “Isso é um parágrafo!”, já o `innerHTML` retornaria “Isso é um `um parágrafo!`”. Para a associação de novos valores, a lógica funciona de maneira parecida. Se associarmos um novo valor ao atributo `textContent` com uma tag HTML (Exemplo: `objeto.textContent = “Novo texto”`), as tags internas são interpretadas como texto puro e não são interpretadas como novos elementos no DOM. Já usando o atributo `innerHTML`, elas seriam adicionadas como novos elementos no DOM.

Acessando e alterando estilos através do DOM

Existem atributos que permitem modificar os estilos de um elemento, mas o acesso a eles é um pouco diferente. Temos três tipos de estilos a serem observados: os que são definidos no atributo `style`, os que são definidos em classes que o elemento faz parte e os que são herdados por não estarem definidos em nenhum lugar específico.

O atributo `style` representa um objeto do tipo `CSSStyleDeclaration` (`CSSSTYLEDECLARATION`, s.d.), que por sua vez possui vários outros atributos que fazem referência às propriedades do CSS. Por exemplo, para alterar a cor de um texto, podemos usar o atributo `objeto.style.color`. Já para alterar a cor de fundo, devemos usar `objeto.style.backgroundColor`. Note que neste último caso, o nome da propriedade CSS seria `background-color`,

mas como o nome de variáveis (e também de atributos) do JavaScript não podem conter hifens, o nome desses atributos é definido juntando-se as duas palavras e deixando a primeira letra da segunda palavra em maiúscula (no caso, `backgroundColor`).



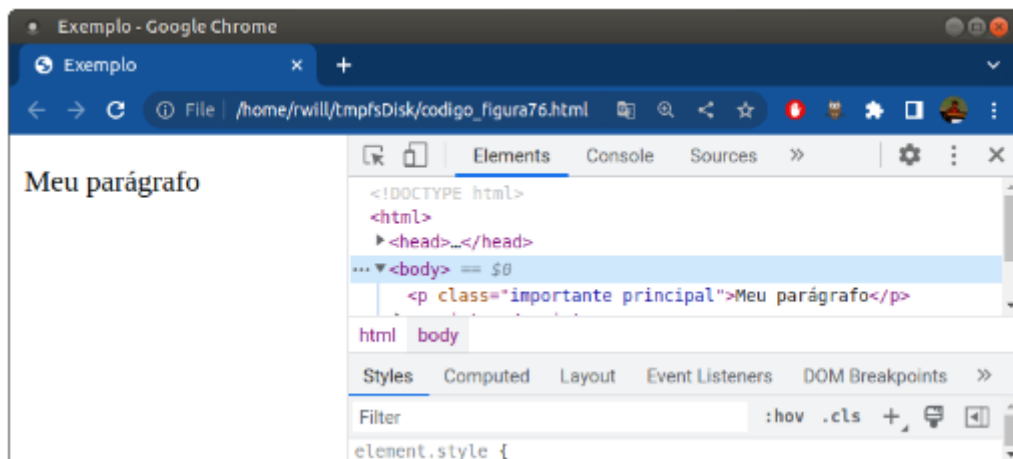
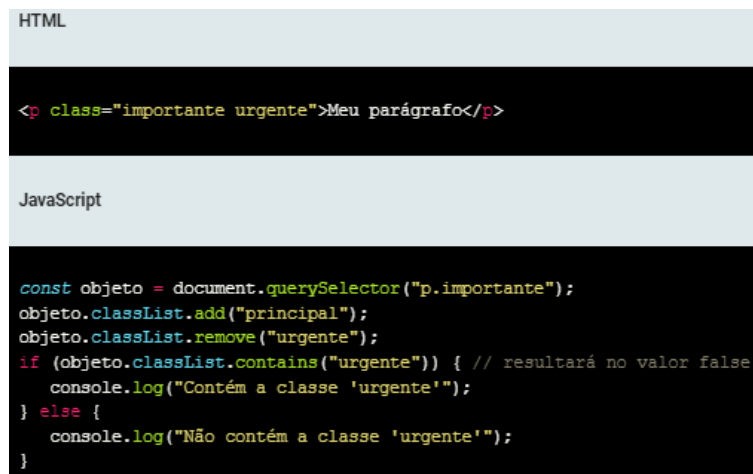
A Figura 7.5 exemplifica a aplicação de estilos através do DOM. Observe que qualquer alteração nos valores dos subatributos do atributo `style` irá refletir automaticamente na apresentação do elemento no navegador. Aqui cabe uma observação importante: sempre utilize o CSS para estilizar uma página Web, pois é uma linguagem própria para este fim. A alteração de estilos no DOM deve ser feita em situações muito específicas (Exemplo: mudar a cor de um texto para vermelho ao clicar num botão, ou exibir algum elemento da página que antes estava invisível após alguma interação do usuário, etc).

Já para os estilos herdados ou definidos em classes, temos que obter os estilos calculados do elemento, ou seja, os estilos “finais” atribuídos ao elemento, calculados após o navegador aplicar todas as regras definidas pelo código CSS (juntando estilos internos e externos). Para obter esses valores, usamos a função global `getComputedStyle(objeto)`, onde `objeto` representa uma variável que faz referência a um objeto do DOM (`WINDOW.GETCOMPUTEDSTYLE()`, s.d.). Essa função retorna um objeto do tipo `CSSStyleDeclaration`, mas seus atributos não podem ser alterados pelo JavaScript e seus valores são sempre atualizados conforme algum outro estilo mude (devido a alguma mudança no código CSS). Os valores obtidos estarão sempre em unidades absolutas, então unidades relativas serão transformadas em pixel, e cores nas suas representações em rgb.

Acessando e alterando classes através do DOM

É possível acessar e modificar o valor do atributo class do HTML através da representação do elemento no DOM. Como ele é multivalorado (um elemento em HTML pode fazer parte de várias classes ao mesmo tempo) podemos manipulá-lo através de dois atributos: `className` e `classList`. O primeiro (`className`) utiliza uma representação em string. Toda alteração (inclusão ou remoção de classes) só pode ser feita com operações sobre strings, não sendo muito prático para manipulações mais complexas.

Já o atributo `classList` representa as classes como um vetor especial, do tipo `DOMTokenList` (`DOMTokenList`, s.d.), com todas as classes definidas no elemento HTML e alguns métodos práticos para manipular seus valores: `.add()` (adiciona uma classe nova), `.remove()` (remove uma classe específica, se existir) e `.contains()` (verifica se uma classe existe no vetor). A Figura 7.6 mostra um exemplo do uso desses métodos.



Alterando elementos internos

Conforme já comentamos, uma das maneiras de adicionar novos elementos internos a um elemento existente é atribuindo valores ao atributo `innerHTML`, onde definimos os elementos como um trecho de código com tags HTML, em formato de string. Uma outra maneira mais flexível e programática, é criar seus elementos usando os métodos presentes no JavaScript e adicioná-los ao elemento pai.

Vamos imaginar uma lista não ordenada de elementos (`ul`) que já possua os itens Minas Gerais e Rio de Janeiro e queremos adicionar um terceiro elemento: São Paulo. Esse exemplo está ilustrado na Figura 7.7, onde a variável `objeto` guarda a representação da lista (tag `ul`) em memória. Usamos o método

`document.createElement(nome)`, onde o parâmetro `nome` representa o identificador de uma tag qualquer, para criar um objeto que representa aquele elemento HTML no JavaScript. Esse método retorna uma referência ao novo objeto criado (no caso do exemplo, um objeto que representa um elemento definido pela tag `li`). Podemos então usar quaisquer outros atributos para manipular esse objeto, como o `textContent` para adicionar conteúdo textual, ou `style` para adicionar estilos ao objeto criado, dentre outros. Vale lembrar que esse objeto ainda não faz parte do DOM, pois acabou de ser criado e está apenas em memória.

```
HTML

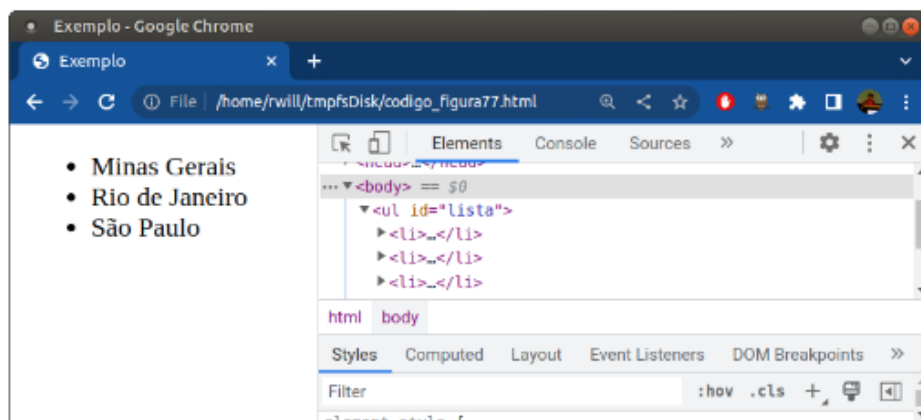
<ul id="lista">
  <li>Minas Gerais</li>
  <li>Rio de Janeiro</li>
</ul>

JavaScript

const objeto = document.querySelector("#lista");

/* Cria um novo objeto em memória, que representa o elemento HTML definido pelas tags <li></li> */
const novo_item = document.createElement("li");
novo_item.textContent = "São Paulo"; // define o texto desse elemento

/* Adiciona o novo objeto ao elemento pai (neste caso, o elemento definido pelas tags <ul></ul>).
Como novo_item representa um elemento <li>, um novo item será acrescentado ao final da listagem */
objeto.appendChild(novo_item);
```



Para acrescentar o novo objeto criado ao DOM, precisamos adicioná-lo ao elemento pai usando o método `document.appendChild(obj)`, onde o parâmetro `obj` é o novo objeto criado. Assim, esse novo elemento será acrescentado ao final do elemento pai. Se quisermos adicionar em uma posição específica, podemos usar o `document.insertBefore(obj_novo, obj_posterior)` (HTML DOM `insertBefore()`, s.d.). É possível também remover um elemento qualquer, usando o método `remove` no próprio objeto (Exemplo: `objeto.remove()`) retirando-o inteiramente do DOM.

Você sabia?

Atualmente o JavaScript está bastante padronizado entre os mais diferentes navegadores e plataformas, mas nem sempre foi assim. Logo após o seu surgimento no fim da década de 1990, cada navegador existente implementou o seu equivalente do JavaScript (que na época era da NetScape), tornando a programação web

para diferentes navegadores um terror para os desenvolvedores. Era muito comum ver condicionais para cada navegador (if ie), pois as funções mudavam de nome ou comportamento dependendo do navegador utilizado. Nesse contexto, a biblioteca jQuery ganhou enorme relevância. Criada em 2006, ela veio com uma sintaxe muito fácil de ser entendida e programada e ela mesmo cuidava desse problema de integração entre os navegadores. Se quiser saber mais sobre o jQuery e sua importância na época, leia o seguinte artigo: <https://pt.khanacademy.org/computing/computer-programming/html-js-jquery/jquery-dom-access/a/history-of-jquery>.

Eventos em JavaScript

O JavaScript é uma linguagem de programação completa que é executada no navegador, permitindo melhorar a interação do usuário com a página web. É possível implementar qualquer lógica com variáveis, funções e diversos tipos de dados. Mas a característica que torna essa linguagem tão importante para a web é poder vincular uma variável a um elemento HTML, e tratar aquele elemento como um objeto em memória. Outra característica importante é a programação e tratamento de eventos, algo muito importante quando trabalhamos com interface gráfica e que permite vincular a execução de uma função a uma ação do usuário.

Quando trabalhamos com interface gráfica (no nosso caso, a página web) é necessário programar a resposta do sistema para cada interação do usuário. Essas interações podem ser, por exemplo: o clique do mouse em um botão, a passagem do mouse por cima de um elemento qualquer, a mudança de conteúdo de algum componente (especialmente em elementos definidos com a tag input), o apertar de uma tecla, etc. Cada uma dessas interações (além de várias outras existentes) gera o que chamamos de evento. E o JavaScript permite programar a resposta a um evento com a execução de uma função.

Escutando eventos com JS

Uma das grandes vantagens do JavaScript é a possibilidade de reagir a certas interações do usuário com as nossas páginas. Praticamente tudo que o usuário manipula na página é passível de reação. Esse sistema é chamado de sistema de eventos no JavaScript, onde eventos são avisos de interações específicas do usuário com a página carregada.

O JavaScript possui uma programação que possibilita escutar a manifestação desses eventos em cada elemento HTML, adicionando “escutadores” (do inglês: listeners) nos elementos que desejamos vincular alguma reação a uma determinada interação do usuário. Os listeners são funções que criamos para serem executadas toda vez que um determinado evento acontecer no nosso elemento HTML especificado.

A associação entre um listener e o seu evento é chamado de registro de listener, e pode ser feito de duas maneiras: usando atributos específicos que iniciam pelo prefixo on* ou utilizando o método `addEventListener()` sobre o elemento que desejamos vincular determinado evento.

Principais eventos em JS

O JavaScript disponibiliza diversos eventos, ou seja, podemos capturar vários tipos de interações diferentes do usuário. A Tabela 8.1 descreve alguns dos principais eventos.

Evento	Descrição
<i>click</i>	Quando o elemento é clicado com o mouse
<i>mouseover</i>	Quando o mouse passa por cima do elemento
<i>mouseout</i>	Quando o usuário afasta o mouse de um elemento
<i>keydown</i>	Quando uma tecla é pressionada
<i>change</i>	Quando o valor do elemento de formulário é alterado
<i>focus</i>	Quando o elemento de formulário ganha foco
<i>focusout</i>	Quando o elemento de formulário perde o foco
<i>load</i>	Quando o navegador termina o carregamento da página
<i>resize</i>	Quando o visitante redimensiona a janela do navegador

Vale mencionar que a associação de eventos a um elemento não é algo exclusivo do JavaScript, pois a maioria das linguagens de programação que lidam com interface gráfica possuem seus próprios modelos de eventos. Cada modelo possui suas especificidades, mas em geral a ideia por trás da maioria delas é executar uma função como resposta a determinada ação do usuário.

Associação de eventos em JS

A seguir, descrevemos as principais formas de associar eventos a um elemento em HTML. Duas delas utilizam os atributos com prefixo `on*`, sendo uma delas através de funções nomeadas e a outra com funções anônimas. As outras duas formas utilizam o método `addEventListener()`, sendo uma delas com funções nomeadas e a outra funções anônimas.

É possível criar outras variações dessas formas, por exemplo, ao associar o “escutador” do evento a uma arrow function (uma forma simplificada de definir funções pequenas) (JAVASCRIPT ARROW FUNCTION, s.d.). Como trata-se apenas de uma mudança na notação da função, não usaremos arrow functions em nossos exemplos.

Associação pelos atributos `on*` com funções nomeadas

Quando construímos uma função com nome, é possível vincular a sua execução a um evento com o prefixo `on*` através de uma atribuição simples. O nome da função se comporta como uma variável que guarda o endereço de memória da região onde está armazenada a função.

A Figura 8.1 mostra um exemplo de como essa vinculação ocorre. Nela, temos duas funções chamadas `exibe_mensagem()` que exibe um alerta na tela, e a função `atualiza_mensagem()` que incrementa uma variável contadora e adiciona um texto na tag `span`. No fim do código em JavaScript, fazemos duas atribuições com os nomes dessas funções (sem aspas, pois não estamos chamando as funções, e sim copiando o endereço de memória delas): o endereço da função `exibe_mensagem` é copiado no atributo `onclick` do objeto `botao` e o endereço da função `atualiza_mensagem` é copiado no atributo `onfocus` do objeto `campo_texto`. Se você retirar o foco do campo de texto e depois clicar nele novamente, a mensagem será atualizada com o novo valor do contador.

HTML

```
<input type="text" id="txt" placeholder="Digite alguma coisa..">
<span id="msg" style="color: red;"></span>
<br><br>
<button type="button" id="btn">Clique em mim</button>
```

JavaScript

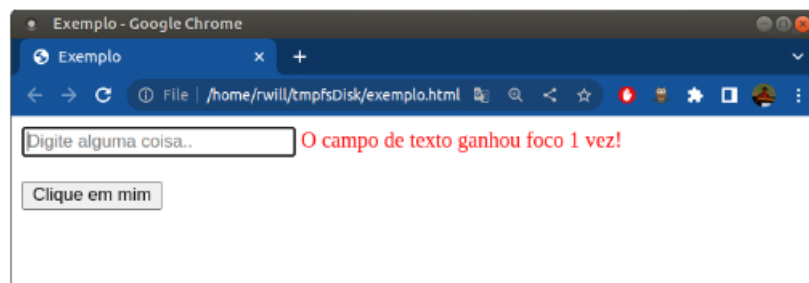
```
const campo_texto = document.querySelector("input#txt");
const mensagem = document.querySelector("span#msg");
const botao = document.querySelector("button#btn");
let contador = 0;

function exibe_mensagem() {
    alert("Você clicou no botão!");
}

function atualiza_mensagem() {
    contador += 1;
    if (contador == 1) {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vez!";
    } else {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vezes!";
    }
}

/* Vincula a função exibe_mensagem() ao evento de click (clique) do botão */
botao.onclick = exibe_mensagem;

/* Vincula a função atualiza_mensagem() ao evento de focus (foco) do campo de input */
campo_texto.onfocus = atualiza_mensagem;
```



Associação pelos atributos on* com funções anônimas

Podemos usar os atributos com prefixo on* para fazer referência direta a funções anônimas. Lembre-se que uma função é armazenada na memória como um objeto qualquer, então nada impede que o seu endereço seja guardado diretamente em um atributo de outro objeto em JavaScript (neste caso, um atributo com prefixo on*).

A Figura 8.2 mostra como podemos vincular funções anônimas a eventos, através desses atributos. Note que o código é muito semelhante ao da Figura 8.1, mas as funções não possuem nomes. À primeira vista o código parece pouco estranho, mas esse tipo de construção é perfeitamente válido em JavaScript, e é comum encontrar exemplos assim na literatura especializada. Em termos de execução, os códigos das Figuras 8.1 e 8.2 são equivalentes, isto é, as mensagens na tela serão exatamente iguais.

HTML

```
<input type="text" id="txt" placeholder="Digite alguma coisa..">
<span id="msg" style="color: red;"></span>
<br><br>
<button type="button" id="btn">Clique em mim</button>
```

JavaScript

```
const campo_texto = document.querySelector("input#txt");
const mensagem = document.querySelector("span#msg");
const botao = document.querySelector("button#btn");
let contador = 0;

/* Vincula uma função anônima ao evento de click (clique) do botão */
botao.onclick = function () {
    alert("Você clicou no botão!");
}

/* Vincula uma função anônima ao evento de focus (foco) do campo de input */
campo_texto.onfocus = function () {
    contador += 1;
    if (contador == 1) {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vez!";
    } else {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vezes!";
    }
}
```

Associação com o método `addEventListener()` e funções nomeadas

O método `addEventListener()` fornece uma maneira mais moderna de associar eventos a uma função. Para vincular um evento qualquer, temos dois parâmetros básicos: o nome do evento (passado como string, e sem o prefixo `on`), e o endereço da função que será executada por aquele evento. A diferença deste método em relação aos atributos com prefixo `on*` é que ele permite adicionar mais de uma função a um mesmo evento. As funções serão executadas na ordem em que foram vinculadas àquele evento.

Outra observação importante é que alguns navegadores não suportam certos tipos de eventos através dos atributos com prefixo `on*` (Exemplo: `focusout`, que na data em que esse material foi escrito, não havia suporte ao atributo `onfocusout` pelos navegadores Chrome, Safari e Opera [`focusout` Event [S.d.]]).

A Figura 8.3 ilustra como podemos associar eventos através do método `addEventListener()` dos objetos `botao` e `campo_texto`. Note mais uma vez que passamos os nomes das funções como um dos parâmetros do método, mas sem adicionar abre e fecha aspas após seus nomes. Estamos enviando o endereço de memória das funções para o método, e não realizando uma chamada direta a elas. Quem vai se encarregar de chamar essas funções é o “escutador” de eventos (event listener) desses objetos, quando esses eventos forem acionados pelo usuário. A execução do código é semelhante aos das Figuras 8.1 e 8.2.

HTML

```
<input type="text" id="txt" placeholder="Digite alguma coisa..">
<span id="msg" style="color: red;"></span>
<br><br>
<button type="button" id="btn">Clique em mim</button>
```

JavaScript

```
const campo_texto = document.querySelector("input#txt");
const mensagem = document.querySelector("span#msg");
const botao = document.querySelector("button#btn");
let contador = 0;

function exibe_mensagem() {
    alert("Você clicou no botão!");
}

function atualiza_mensagem() {
    contador += 1;
    if (contador == 1) {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vez!";
    } else {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vezes!";
    }
}

/* Vincula a função exibe_mensagem() ao evento de click (clique) do botão */
botao.addEventListener("click", exibe_mensagem);

/* Vincula a função atualiza_mensagem() ao evento de focus (foco) do campo de input */
campo_texto.addEventListener("focus", atualiza_mensagem);
```

Associação com o método `addEventListener()` e funções anônimas

Entre todas as maneiras de vincular funções a eventos, essa é a mais interessante. Ao invés de passar o nome da função (que contém seu endereço) ao método `addEventListener()`, podemos criar uma função anônima no lugar do parâmetro. Essa operação funciona e faz sentido: primeiro, a função anônima será armazenada como um objeto na memória RAM, e o seu endereço será retornado após a sua definição. Esse endereço é imediatamente passado por parâmetro para o método `addEventListener()`, que será executado em seguida, vinculando assim esse objeto criado (a função anônima) com o evento passado como string para o método.

A sintaxe básica é: `objeto.addEventListener("nome_do_evento", function () { corpo_da_função });`, e a Figura 8.4 ilustra como podemos usar essa forma com a mesma lógica dos exemplos das Figuras 8.1, 8.2 e 8.3. É muito comum encontrar códigos semelhantes na literatura, e a desvantagem deste método é que não podemos reaproveitar a função. Por outro lado, a associação do listener e a lógica da função ficam mais próximas no código.

HTML

```
<input type="text" id="txt" placeholder="Digite alguma coisa..">
<span id="msg" style="color: red;"></span>
<br><br>
<button type="button" id="btn">Clique em mim</button>
```

JavaScript

```
const campo_texto = document.querySelector("input#txt");
const mensagem = document.querySelector("span#msg");
const botao = document.querySelector("button#btn");
let contador = 0;

/* Passa uma função anônima como parâmetro, vinculando-a ao evento de click (clique) do botão */
botao.addEventListener("click", function () {
    alert("Você clicou no botão!");
});

/* Passa uma função anônima como parâmetro, vinculando-a ao evento de focus (foco) do campo de input */
campo_texto.addEventListener("focus", function () {
    contador += 1;
    if (contador == 1) {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vez!";
    } else {
        mensagem.textContent = "O campo de texto ganhou foco " + contador + " vezes!";
    }
});
```

O objeto event

Todo evento gerado pelo JavaScript vem acompanhado de um objeto especial com informações sobre aquele evento específico. Esse objeto é chamado de objeto de evento (event object), e pode ser acessado adicionando um parâmetro com um nome qualquer à função associada a um evento. Comumente esse parâmetro é chamado de event, evt, ou simplesmente e, ficando a critério do desenvolvedor escolher o nome mais apropriado.

A depender do tipo de evento, esse objeto pode ser do tipo Event, ou alguma especialização desse tipo, como por exemplo MouseEvent (HTML DOM MOUSEEVENT, s.d.), para eventos que envolvem manipulação do mouse (Exemplo: evento mouseover), ou KeyboardEvent (HTML DOM KEYBOARDEVENT, s.d.), para eventos que envolvem manipulação do teclado (Exemplo: evento keydown). Há outros tipos mais específicos, como por exemplo, PointerEvent, FocusEvent, etc.

Cada um desses objetos específicos acompanha um conjunto de atributos, cujos valores podem ser acessados para obter informações adicionais sobre o evento. Também acompanham um conjunto de métodos, que podem alterar o comportamento daquele evento. A Figura 8.5 ilustra o uso do objeto de evento: a função `exibe_mensagem(event)` está associada ao evento de click do botão e utiliza os atributos `clientX` e `clientY` do objeto event para saber as coordenadas x e y do clique do mouse naquele botão. Já a função `verifica_tecla(event)` está associada ao evento de keydown do campo de texto. Toda vez que uma tecla é pressionada nesse campo, esse evento é disparado. A função então verifica se o valor do atributo `key` do objeto event, que contém o caractere pressionado, está na faixa entre '0' e '9'. Se essa verificação for verdadeira, chamamos o método `preventDefault()` do objeto event, que fará com que a ação padrão do evento seja cancelada. A ação padrão do evento keydown é adicionar a tecla pressionada ao campo de texto. Em resumo, o campo de texto do exemplo não aceita valores entre 0 e 9, mas aceita qualquer outra tecla.

HTML

```
<input type="text" id="txt" placeholder="Digite alguma coisa..">
<br><br>
<button type="button" id="btn">Clique em mim</button>
```

JavaScript

```
const campo_texto = document.querySelector("input#txt");
const botao = document.querySelector("button#btn");

function exibe_mensagem(event) {
    alert("Você clicou no botão nas coordenadas x=" + event.clientX + " e y=" + event.clientY);
}

function verifica_tecla(event) {
    /* Verifica se a tecla pressionada está entre 0 e 9 */
    if (event.key >= '0' && event.key <= '9') {
        /* Se a tecla estiver entre 0 e 9, imprima a ação padrão do evento:
        ou seja, não permita que o valor digitado seja inserido ao campo de texto */
        event.preventDefault();
    }
}

botao.addEventListener("click", exibe_mensagem);
campo_texto.addEventListener("keydown", verifica_tecla);
```

Você conhece?

O desenvolvedor e consultor em softwares especializado em aplicações JavaScript, Erick Wendel possui um portfólio de cursos (alguns gratuitos), vídeos, participações de comunidades e fóruns de JavaScript, sendo nomeado pela Microsoft como MVP (Most valuable Professional) em Javascript, além de ser formado pela Faculdade Impacta de Tecnologia em 2016. Se quiser acompanhar mais do Erik, veja o seu site pessoal (com links para suas mídias sociais): <https://erickwendel.com/>.

O método `preventDefault()` é extremamente útil quando queremos cancelar alguma ação em JavaScript. Considere, por exemplo, que temos um botão do tipo submit em um formulário. Podemos associar uma função ao clique deste botão, e dentro da função podemos verificar se os dados do formulário são válidos. Se houver alguma inconsistência, chamamos o método `preventDefault()` a partir do objeto de evento de clique, cancelando o envio do formulário ao servidor.

O JavaScript também possui uma programação que permite criar eventos customizados (chamados de eventos sintáticos) e usá-los da maneira que desejarmos, mas esse assunto vai além do escopo da nossa disciplina. Para ler mais a respeito, acesse (Creating and triggering events, s.d.).

Padrão MVC e Programação no Servidor

Os sistemas que rodam em servidores possuem toda a diversidade de problemas e soluções clássicas, que vem de anos de desenvolvimento de aplicações neles. Essas soluções padronizadas para problemas comuns são conhecidas como Padrões de Projeto (do inglês: Design Patterns). Um dos padrões arquiteturais mais conhecidos e amplamente usado é o MVC (Model-View-Controller). Vamos entender como esse padrão funciona e como ele se encaixa na programação em servidor usando a linguagem de programação Python e o microframework Flask.

Com as linguagens HTML e CSS podemos desenvolver um site para exibir informações quaisquer. Esse site se comportará como um “livro” digital, mostrando informações estáticas e sem muita interação com o usuário. A linguagem de programação JavaScript pode ser usada para melhorar essa interação, mas ela também possui suas limitações, pois será executada somente do lado do cliente (interpretada pelo navegador).

Imagine agora que desejamos desenvolver um sistema e disponibilizá-lo na web. Da mesma forma, utilizaremos o HTML, CSS e JavaScript para criar páginas web e exibir informações aos usuários desse sistema. Mas um sistema necessita de muito mais que isso: é necessário implementar toda a regra de negócio, toda a lógica por trás do funcionamento do sistema, integração com algum SGBD (sistema gerenciador de banco de dados), etc. Para isso, precisamos de um servidor web que execute toda a lógica por trás do nosso sistema, através de uma linguagem de programação.

Pense que esse servidor web executará o seu programa desenvolvido em Python. Mas toda vez que for necessário executar um print (comando que exibe informações na tela), ele envia uma resposta em HTML para o navegador que requisitou que aquela lógica fosse executada. Internamente, o Python trata o HTML como uma string, mas para o navegador, aquele código em HTML criará uma interface gráfica.

Nas seções a seguir, vamos entender como é possível desenvolver uma aplicação deste tipo, através de um módulo chamado Flask.

Padrões de Projeto

Um software consiste em várias instruções escritas em uma linguagem de programação, armazenadas em memória e executadas por um processador. O primeiro software criado é datado de 1948, criado por John von Neumann na Inglaterra, apesar de um ter sido concebido em forma lógica por Ada Lovelace em 1842 (SUPERINTERESSANTE, 2011). Passado tanto tempo de desenvolvimento de software, muitos problemas comuns no desenvolvimento apareceram, muitos deles recorrentes em outros domínios de aplicação dos softwares. Para esses problemas, uma série de soluções surgiram, igualmente recorrentes. Essas soluções para problemas recorrentes e padronizados são chamadas de Padrões de Projeto (Design Patterns), termo que ficou conhecido no livro do GoF (Gang of Four) Design Patterns: Elements of Reusable Object-Oriented Software (JESUS, 2019). Neste livro estão catalogados 23 padrões mais comuns à época, usados no desenvolvimento de programas orientados a objetos.

Com a evolução dos softwares, bem como os problemas encontrados no seu desenvolvimento, mais padrões foram aparecendo e, com isso, temos mais de 80 padrões diferentes atualmente, com mais categorias aparecendo (como a arquitetural). Alguns padrões caíram em desuso, pela evolução natural das plataformas e linguagens.

Inicialmente foram levantados 23 padrões divididos em três categorias: criacional, estrutural e comportamental. Os padrões de criação (creational) são aqueles construídos para ajudar na criação de objetos. Dependendo do nível de complexidade de uma classe, a criação do objeto pode ser uma tarefa muito complicada ou depender de muitos dados, padrões como o Factory Method existem para facilitar esta tarefa.

Os padrões estruturais (structural) estabelecem maneiras mais flexíveis de trabalhar com composições de objetos, de forma a criar grandes estruturas de objetos. O padrão decorator é um bom exemplo, criando funcionalidades extras de maneira dinâmica a um objeto, sem que haja necessidade de se recorrer a herança.

Os padrões comportamentais (behavioral) têm como objetivo facilitar a comunicação e a passagem de dados entre objetos. São usados para deixar essa integração entre objetos mais fácil de ser mantida. O padrão Observer é um exemplo deste tipo, onde outros objetos que estejam observando o objeto original serão imediatamente avisados de qualquer alteração de estado neste objeto.

Todos esses padrões têm um escopo um pouco mais reduzido, a nível de código ou classes, para resolver um problema específico de negócio. Conforme os sistemas começaram a ficar maiores em número de linhas de

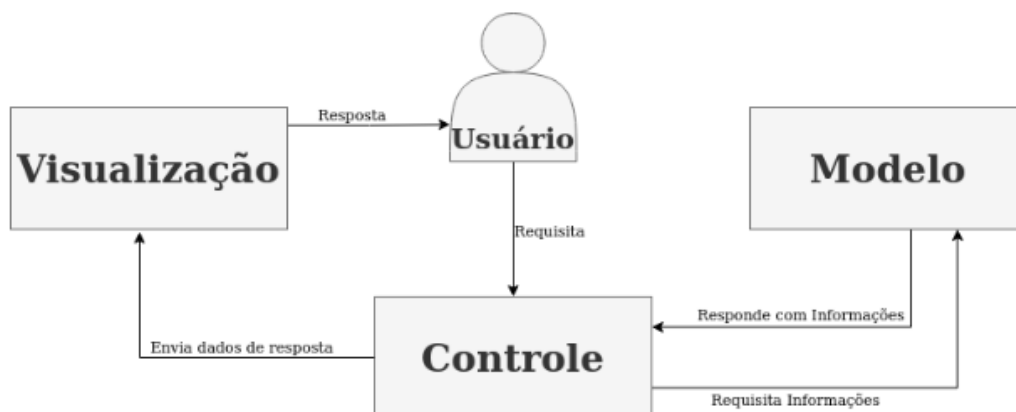
código e classes, outros padrões para o nível mais macro do projeto foram criados, chamados de padrões arquiteturais. O mais famoso e utilizado destes é o Padrão MVC.

Padrão MVC

O padrão arquitetural MVC (Model-View-Controller) foi criado por um engenheiro da Xerox, Trygve Reenskaug, em 1979. A ideia era criar um padrão de estrutura de aplicações feitas com a linguagem Smalltalk, criando uma estrutura em camadas para diminuir o acoplamento e aumentar a coesão entre as classes no projeto, facilitando a manutenção do sistema (MEDEIROS, 2013).

O padrão consiste nas três camadas que dão o seu nome: Modelo (Model), Visualização (View) e Controle (Controller), cada camada é responsável por uma responsabilidade no sistema:

- O **model** (modelo) representa o negócio que a aplicação foi construída para lidar. Nele, colocamos desde modelos de dados (como classes que mapeiam tabelas de banco de dados) a códigos de lógica de negócio. O modelo é o que traz valor e necessidade do negócio para o software. Tudo que for decisão do ponto de vista do negócio modelado fica nesta camada;
- O **view** (visualização) é a camada responsável por interagir com o usuário diretamente. Ela tem a responsabilidade de mostrar os dados da melhor maneira para o usuário e de coletar informações e interações dele e encaminhá-las para a aplicação. No caso da web, o navegador e as páginas são a visualização, com o HTML, CSS e JavaScript;
- O **controller** (controle) é a camada intermediária entre o modelo e a visualização. Ela é responsável pelo fluxo de dados e de trabalho da aplicação. Ela recebe as informações e interações da visualização via requisições HTTP e escolhe qual funcionalidade do modelo responderá por essas interações. Ao final do processamento, o controle ainda é responsável por decidir o caminho da resposta do processamento, devolvendo dados para a camada de visualização montar as páginas da melhor maneira. A Figura 9.1 mostra como funciona essa comunicação entre as camadas.



Flask Framework

O Flask é um micro framework escrito em Python para a construção de aplicações web. O micro vem do fato de que o framework em si é bem pequeno, concentrando-se apenas nas tarefas mais importantes, deixando pontos de extensão para que novas capacidades sejam acopladas a ele de outras maneiras.

Pela sua simplicidade, o Flask está pronto para ser usado nas mais diversas aplicações em produção, com amplas capacidades de escalabilidade. Usado por grandes empresas de tecnologia como Airbnb, Uber, Reddit e Mozilla (ROCHA, 2018).

Pré Requisitos

Para a utilização do Flask, é necessário ter uma instalação funcional do interpretador Python na versão 3.5 (ou superior) e o pacote Flask instalado. Se o Python tiver sido instalado com o seu gerenciador de pacotes padrão (pip), basta usar o comando descrito na Figura 9.2, caso contrário, a documentação do Flask (disponível em: <https://flask.palletsprojects.com/en/2.2.x/installation/>) mostra como instalar ele de outras maneiras.

```
# Instalação no Windows (através do cmd):  
py -m pip install -U flask  
  
# Instalação no Linux ou Mac (através do Terminal):  
python3 -m pip install -U flask
```

Mesmo não sendo obrigatório, recomendamos que todos os exemplos e execuções do Flask sejam feitas usando um ambiente virtual do Python (virtualenv, pipenv, etc). Além disso, que seja instalado um linter (ferramenta de validação de código do Python) como o Pylint ou Flake8.

HelloWorld: a primeira aplicação MVC no Flask

A construção de uma aplicação Flask se resume basicamente a criação e execução de um objeto específico. Para construir uma pequena aplicação, vamos criar um arquivo app.py no nosso projeto, onde a aplicação será criada. A Figura 9.3 mostra a estrutura básica usada para construir uma aplicação em flask.

Code:

Arquivo: app.py

```
# Importação da classe Flask a partir do módulo flask
```

```
from flask import Flask
```

```
# Definição do objeto que representa a aplicação
```

```
app = Flask(__name__)
```

```
# Cria a rota /, acessível a partir de: http://127.0.0.1:5000/
```

```
@app.route('/')
```

```
def index():
```

```
    return "Olá mundo!"
```

```
# Cria a rota /saudacao, acessível a partir de: http://127.0.0.1:5000/saudacao
```

```
@app.route('/saudacao')
```

```
def saudacao():
```

```
    s = "<h1>Bem-vindo!</h1><p>Essa página contém tags html.</p>"
```

```
    return s
```

```
# Inicialização do servidor
```

```
if __name__ == '__main__':
```

```
    app.run()
```

Todos os objetos, classes e funções que serão usadas no Flask se encontram no módulo flask e serão importados direto desse módulo. A aplicação é um objeto criado a partir da classe Flask (note o F maiúsculo), importada do módulo citado, com o comando: `from flask import Flask`.

Na sequência, é necessário criar o objeto da aplicação, associando ele a uma variável, que nos nossos exemplos vamos sempre chamar de `app`. A classe Flask possui vários parâmetros no seu construtor, mas só um deles é obrigatório. Esse parâmetro indica um nome de importação para a nossa aplicação (representada pelo objeto `app`), indicando como essa aplicação será importada em outros módulos. Enquanto a aplicação possuir um único módulo simples, é recomendado usar a variável global do Python `__name__`, portanto a construção deste objeto será definida com o comando: `app = Flask(__name__)`.

Após definir o objeto que representa a aplicação, devemos definir funções em Python que vão responder a requisições HTTP. Essas funções fazem o papel de controles (controllers) do MVC, ou seja, elas recebem as requisições e implementam código que coordena o que será feito com elas. Para que o Flask associe a execução de uma função com o acesso a uma URL específica, devemos registrar essa informação através de um decorador (decorator) específico do Flask, chamado `@app.route()` (veja mais sobre decorators em (Introdução aos “decorators” do Python, 2012)). Note que o `app` deste comando é o nome do objeto que representa a nossa aplicação, e `route()` é um método interno deste objeto. O parâmetro deste decorador é a rota (path) que será respondida pelo controle decorado. O decorador deve aparecer na linha anterior à definição de cada função, e desta forma cada função responderá à rota informada. Sendo assim, ao executar o código definido na Figura 9.3, teremos:

- `http://127.0.0.1:5000/` - acessar a rota `/` (barra) pelo navegador fará com que a função associada a ela, chamada `index()`, seja executada. Essa função retorna a string “Olá mundo!”;
- `http://127.0.0.1:5000/saudacao` - acessar a rota `/saudacao` pelo navegador fará com que a função `saudacao()` seja executada. Essa função cria uma string chamada `s` que contém texto entre tags HTML. Ao receber esse código, o navegador interpreta essas tags e exibe os elementos gráficos desejados (um título de seção e um parágrafo). Esse exemplo ilustra onde queremos chegar: misturar HTML (linguagem entendida pelo navegador) com Python (linguagem que usaremos para programar a lógica do servidor).

Por fim, é necessário executar a aplicação para ela começar a responder as requisições. Para isso, usamos o método `run()` no objeto `app` (`app.run()`). Como esse método só será executado na execução do script com a aplicação, é comum ele estar dentro do bloco `main` do Python (abaixo do comando: `if __name__ == '__main__':`). Para evitar ficar reiniciando a aplicação toda vez que estivermos mexendo nos arquivos Python, podemos passar o parâmetro nomeado `debug=True` no método `run()` para iniciar a aplicação em modo de depuração (debug), assim qualquer alteração em arquivos Python já é carregada para a execução da aplicação diretamente. Note que a porta padrão na URL é a 5000. Esse é o valor padrão do Flask, pois é uma porta que provavelmente não será usada por nenhuma outra aplicação que está rodando paralelamente no computador, evitando conflitos na hora de criar o servidor. Esse valor pode ser alterado através do parâmetro nomeado `port=VALOR` no método `run()`, onde `VALOR` deve ser um número entre 1 e 65535. Normalmente deixamos o valor padrão, mas caso seja necessário mudar, é recomendado que você nunca utilize um valor de porta abaixo de 1024, pois são portas comumente reservadas para outras aplicações.

Ao rodar o arquivo `app.py`, algumas informações serão exibidas no terminal (ou console), sendo a mais importante um endereço para entrarmos e ver o resultado. Acesse as URLs `http://127.0.0.1:5000/` ou `http://localhost:5000/` para ver a mensagem Olá Mundo! na janela do navegador. O nome `localhost` (assim como o IP especial 127.0.0.1) indica para o computador que as requisições vão para um servidor na própria máquina.

Implementando o MVC

No exemplo da Figura 9.3 fizemos uma pequena aplicação juntando tudo no mesmo arquivo. Agora vamos definir outra aplicação um pouco mais complexa, que separa os códigos pensando nas camadas do MVC. Por enquanto, como a aplicação é muito simples, não vamos nos preocupar com o modelo (model). O código desta aplicação está disponível em: <https://github.com/rwill-impacta/flask-criacao-templates>

Note que criamos uma pasta chamada templates (ela deve ter exatamente este nome), onde ficarão os arquivos HTML. O Flask já vem configurado para buscar os arquivos HTML neste diretório. Essa pasta só é acessível dentro do servidor, ou seja, não é possível acessar nenhum arquivo dentro dela através da URL vinculada ao servidor.

Outra pasta contida no projeto chama-se static, e como o nome sugere, é a pasta onde ficarão arquivos estáticos (CSS, JavaScript, imagens, documentos e quaisquer outros formatos de arquivo que desejamos compartilhar). Diferente do primeiro diretório, tudo que é colocado dentro desta pasta é acessível a partir da URL do servidor, através do caminho /static. Por exemplo, o arquivo estilos.css que está contido neste diretório pode ser acessado através da URL relativa /static/estilos.css (por um arquivo HTML dentro do projeto), ou pela URL absoluta <http://127.0.0.1:5000/static/estilos.css> (através do navegador).

No Flask usaremos arquivos HTML como templates (modelos para visualização de conteúdo), e por isso eles ficarão em um diretório com esse mesmo nome. Cada um desses templates fará o papel de uma visualização (view) do modelo MVC. Para poder usar um template no controle, devemos importar a função `render_template()` do pacote flask (através do comando: `from flask import render_template`) e, no lugar do retorno da string, colocar a execução da função `render_template()` com o nome do arquivo a ser usado como template (`return render_template('nome_do_arquivo.html')`).

Resumidamente, a função `render_template()` lê o conteúdo do arquivo, interpreta-o como uma string, e retorna esta string para a linha de código que a chamou. Veremos no próximo capítulo que, na verdade, essa função faz muito mais do que simplesmente ler o conteúdo do arquivo: ela também é capaz de interpretar uma linguagem usada para modelar templates no Flask. Essa é uma característica importantíssima no servidor, pois permite que criemos páginas dinâmicas, isto é, páginas que mudam de conteúdo ao longo do tempo.

Parametrização de Rotas

Uma característica muito útil do Flask é a possibilidade de parametrizar rotas. Isso permite simplificar a definição de rotas que seguem um padrão similar (Exemplos: `/curso/devweb` e `/curso/poo`). Note que essa característica poderia ser alcançada também se utilizássemos a Query String (Exemplos: `/curso?nome=devweb` e `/curso?nome=poo`), mas esse método não é indicado para páginas que devem ser indexadas nos mecanismos de busca, pois eles comumente ignoram a parte da URL que contém a Query String. Para construir URLs parametrizáveis e indexadas, vamos usar o conceito de Path Parameter (FULLSOUR, 2017).

A ideia dos Path Parameter é criar regiões no caminho (path) da URL que virem parâmetros de entrada nos nossos controles. Nesse caso, vamos usar o slug: a última parte da URL (que servirá de forma semelhante a um recurso), identificando o conteúdo que será mostrado, usando um texto amigável para URLs (tudo em minúsculas, sem espaços e acentos). Para exemplificar o uso dos Path Parameters, construímos um novo exemplo disponível em: <https://github.com/rwill-impacta/flask-path-parameters>

Neste novo projeto em Flask, a função Python `curso()` será executada todas as vezes que alguém acessar a rota `/curso`. Mas quando acessamos a rota `/curso/devweb`, o servidor chamará a função `curso_por_nome()`. Isso porque definimos a rota dessa função como `/curso/<nome>`, onde `<nome>` é o slug que representa o nome (sigla) de um curso qualquer que adicionaremos na rota (No exemplo anterior, o parâmetro `nome` receberia o valor string `devweb`). Note que a função `curso_por_nome()` recebe um parâmetro chamado `nome`, que é o mesmo valor que colocamos no slug entre os símbolos `<` (menor que) e `>` (maior que). Em resumo, o parâmetro `nome` daquela função captura qualquer valor colocado ao final da URL (após o prefixo `/curso/`), e assim podemos dar o tratamento necessário para qualquer rota com esse prefixo dentro do controlador. Assim, não é necessário criar um controle diferente para cada curso, basta identificarmos o slug passado na URL para buscar no nosso catálogo de cursos.

Por padrão, esses parâmetros são interpretados como Strings, mas o Flask permite definir exatamente o seu tipo, bastando para isso informar o tipo na frente do nome dado ao slug, separado por dois pontos [Variable Rules, [S.d.]]:

- `<int:param>`: para inteiros positivos;
- `<float:param>`: para decimais positivos;
- `<path:param>`: igual a string, mas aceita barras no meio;
- `<uuid:param>`: aceita objetos do tipo UUID.

Independente do seu tipo, para cada path parameter definido em uma rota, a função do controle daquela rota receberá o parâmetro de mesmo nome como entrada. Por exemplo, no projeto que exemplifica o uso dos path parameters, a função `curso_com_dois_parametros()` recebe os parâmetros `nome` e `ano`, pois sua rota possui dois slugs: `/curso/<nome>/<int:ano>`. Ou seja, qualquer rota formada por esse padrão (Por exemplo: `/curso/programacao/2022`, ou `/curso/web/1997`, ou mesmo `/curso/abc/2`) será tratada pela função `curso_com_dois_parametros()`, enquanto que os parâmetros `nome` e `ano` da função terão os tipos string e inteiro, respectivamente. Por outro lado, a rota `/curso/devweb/27a` não será tratada por nenhum controle configurado, pois o valor `27a` não é um valor inteiro.

Vamos praticar?

No exemplo deste material, precisamos usar bibliotecas que não estão por padrão no Python, sendo necessário instalar elas (usando o pip). Fizemos isso direto na instalação global do Python (do sistema operacional), nem sempre isso é recomendável ou possível (às vezes nem somos admins do computador). Recomenda-se fortemente adotar os ambientes virtuais do Python, que são ferramentas que permitem criar pequenas instalações do nosso Python de maneira personalizada por projeto, permitindo que possamos instalar o que quiser nessa pequena instalação, sem prejudicar a instalação global do Python. Veja mais em: <https://pythonacademy.com.br/blog/python-e-virtualenv-como-programar-em-ambientes-virtuais>.

Server Side Redering: Criação de Páginas Dinâmicas

Aprendemos a usar HTML, CSS e JavaScript para construir páginas com conteúdo semântico, estilizadas e dinâmicas, mas todo este conteúdo precisa estar definido previamente para o navegador renderizá-lo. E se quisermos que o conteúdo mudasse de acordo com o estado da aplicação? Uma das técnicas mais clássicas e ainda em grande utilização é a renderização no servidor (ou Server Side Rendering) onde usamos alguma linguagem de servidor que nos ajude a gerar o HTML que precisamos para mostrar nosso conteúdo.

Quando falamos em termos de MVC pensando em sistemas web, associamos imediatamente a camada de visualização com as tecnologias do cliente: HTML, CSS e JavaScript, pois elas são responsáveis por estruturar,

estilizar e trazer vida e inteligência para as páginas na web. Porém esse conteúdo é chamado de estático, pois esses arquivos já estão prontos para entrega no lado do servidor.

Conforme movemos os negócios para o mundo digital, se torna mais comum que nosso conteúdo seja cada vez mais dinâmico, ou seja, ele muda a todo momento, não sendo possível entregar um único HTML com o mesmo conteúdo a todo momento. Nesse sentido, precisamos usar tecnologias adicionais que permitam juntar o conteúdo dinâmico (dados em um banco de dados, por exemplo) com o HTML, CSS e JavaScript para apresentar esses dados aos usuários. E dependendo dos objetivos e características da aplicação, podemos empregar diferentes técnicas de renderização, ou até misturar várias delas em diferentes partes da aplicação.

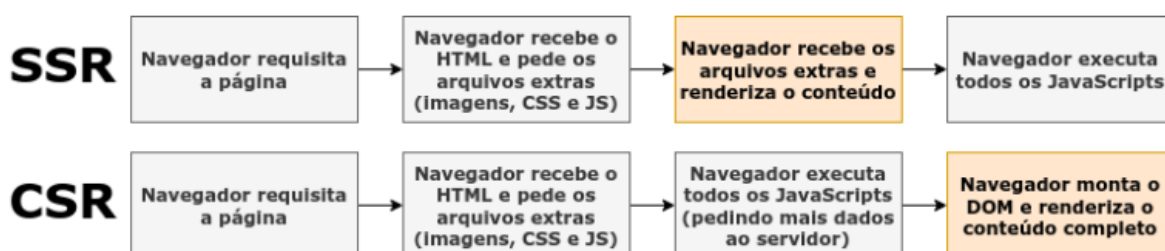
Técnicas de Renderização Dinâmica

Em termos de renderização dinâmica temos duas técnicas dominantes: renderização no servidor (SSR - Server Side Rendering) e renderização no cliente (CSR - Client Side Rendering). A diferença entre ambos é qual dos atores do modelo cliente-servidor vai ser o responsável por montar o conteúdo a ser mostrado para o usuário. Em ambos os casos temos que montar páginas com HTML, CSS e JavaScript pois são essas as tecnologias que o navegador utiliza para mostrar as páginas, e isso ocorre juntando dados provenientes de fontes dinâmicas (APIs, banco de dados, arquivos, etc) com modelos do que deve ser montado na tela (chamamos de templates).

A técnica mais clássica é a do servidor (SSR), utilizado por quase todas as aplicações legadas de antes dos anos 2000. No começo era usada a própria linguagem de programação (Java, Python, C#, etc.) para montar diretamente o HTML a ser transferido para o navegador, usando puramente manipulação de strings. Posteriormente, cada uma dessas linguagens (e seus frameworks) foram criando sublinguagens de templates, de modo a facilitar o desenvolvimento do HTML. A ideia dessas sublinguagens é usar anotações especiais dentro de arquivos HTML para que a linguagem principal fosse capaz de montar o HTML final (visualização) usando esse modelo como base, além dos dados passados ao template (ROCHA, 2018). Os exemplos mais conhecidos dessas tecnologias são o JSP (do Java), ASP.NET (C#) e o PHP.

A partir dos anos 2000, com a popularização do JavaScript, a computação em nuvem e as aplicações de página única (Single Page Application - SPA), surgiu a necessidade de deixar a responsabilidade de montar os conteúdos HTML com o navegador. Além de evitar grandes gastos com processamento de linguagens de template no lado do servidor (o que é uma preocupação na computação em nuvem), as SPAs precisam montar seu conteúdo conforme ele vai sendo requisitado (uma vez que elas nunca mudam de página efetivamente), então é necessário que o JavaScript seja responsável pela renderização deste conteúdo (VEGA, 2017).

A Figura 10.1 mostra a diferença em como a renderização funciona em ambos os casos. Podemos ver que o SSR consegue entregar uma página visualizável antes que o CSR, o que pode ser importante em alguns casos de desempenho e necessidade de conteúdo renderizado para os buscadores.



Atualmente é comum misturar as duas técnicas: entregar uma parte da página já renderizada no servidor, para que o usuário consiga visualizá-la sem executar o JavaScript (importante para o SEO - Search Engine Optimization do Google) e o resto ser dinâmico no cliente.

Existe ainda uma terceira técnica mais interessante, principalmente para páginas que possuam pouco conteúdo dinâmico, que é a geração de sites estáticos (Static Site Generator - SSG) (OLIVEIRA, 2020). A ideia é rodar algum programa que junte os dados dinâmicos com um template da mesma maneira que os anteriores, mas esse programa irá gerar e salvar o conteúdo em arquivos HTML para serem enviados ao cliente. Dessa forma a renderização será a mais rápida possível (pois o conteúdo já chega pronto no cliente) e sem custo adicional por usuário (pois só precisamos gerar uma vez). Sites muito simples como blogs, currículos pessoais e até institucionais usam bastante essa técnica, pois são conteúdos que mudam muito pouco ao longo dos dias.

Você quer ler?

A discussão entre as técnicas de renderização é quase interminável. Os dois pontos mais relevantes na comparação entre eles são o SEO (Search Engine Optimization) e o tempo de renderização no navegador. Nem toda aplicação precisa se preocupar com SEO, pois nem todas estão disponíveis para serem buscadas pelo Google, mas todas elas precisam se preocupar com o desempenho de renderização. Para saber mais sobre como as técnicas impactam nesse desempenho, leia aqui: <https://ichi.pro/pt/renderizando-na-web-214973935184806>

Renderização no Servidor com o Flask

O foco desta disciplina vai ser na renderização no servidor (SSR) usando a linguagem de templates Jinja2, que já vem junto com o micro framework Flask. O Jinja é uma linguagem de template muito utilizada no mundo Python, baseado na sintaxe do Django (outro framework Python usado em desenvolvimento web), com diversas otimizações e melhorias para uso nas nossas aplicações (Jinja — Jinja Documentation, s.d.).

Como usar no Flask

O Jinja2 já vem por padrão na instalação do Flask e sua integração é automática. Basta configurar na aplicação Flask qual a pasta que ela deve procurar os arquivos templates, que por padrão é a pasta templates junto com o arquivo da aplicação (app.py).

Os arquivos HTML já estão prontos para serem usados e lidos pela função `render_template()`, mas o Jinja2 introduz as template tags específicas para tornar os arquivos dinâmicos, permitindo uma fácil integração dos templates com os dados da aplicação. As template tags são divididas em três tipos:

- `{% TEMPLATE TAG %}`: usada para expressões de código ou lógica;
- `{{ VALOR }}`: usada para expressões de impressão de valores no HTML;
- `{# COMENTARIO #}`: usado para comentários. Detalhe importante: diferente dos comentários do HTML, comentários feitos com a template tag `{# #}` serão interpretados pelo Jinja e não serão enviados junto com a requisição.

Exemplos simples da utilização dessas tags podem ser vistos no projeto disponível em: <https://github.com/rwill-impacta/flask-template-tags-jinja>.

Imprimindo valores no template

Para passar valores do Python para os templates no Jinja2, usamos a função `render_template()` passando o nome do arquivo de template e quantas variáveis de contexto quisermos. Essas variáveis são passadas como parâmetros nomeados do Python (técnica conhecida como `kwargs`) (TARDIVO, 2018), podendo ser de quaisquer tipos e em qualquer quantidade. Por exemplo, se quisermos passar uma string em Python que representa um título que será mostrado na tag `<h1>` do index do site, devemos usar a função: `render_template('index.html', titulo='Título Modificado pelo Python')`.

Após passar as variáveis de contexto para um template, é possível acessá-las dentro do template usando a expressão de impressão das template tags. Para isso basta usar o nome da variável passada como parâmetro dentro da template tag: `{{ titulo }}`. Ao executar o servidor e acessar a página no navegador, vamos ver o título 'Título modificado pelo Python' dentro da tag `h1`. Se trocarmos o valor da string da variável de contexto, ao atualizar a página o servidor vai recriá-la com o novo valor (Expressions, s.d.).

O Flask permite passar qualquer tipo de objeto e valor no contexto. Se o for uma lista, podemos acessar seus elementos pelo índice (Exemplo: `{{ lista[0] }}`), se for um objeto ou dicionário, podemos acessar seus atributos (objeto) ou valores associados a uma chave (dicionário) pelo operador ponto (Exemplo: `{{ objeto.propriedade }}`).

Não podemos executar funções Python dentro dos templates, mas o Jinja2 possibilita usarmos filtros (filters) que servem para alterar de alguma maneira o que irá ser exibido na impressão. Para usar um filtro, devemos colocá-lo após a variável de contexto, separando-o através do caractere pipe (`|`). Por exemplo, na expressão `{{ titulo | upper }}` o título irá aparecer com todas as letras em maiúsculo (Built In Filters, s.d.). Também é possível construir filtros personalizados.

Estruturas de Controle

Além da impressão de valores presentes no contexto, podemos usar estruturas de controle similares às que existem no Python para ter maiores possibilidades na construção das páginas. Vamos focar nas duas mais utilizadas: o `for` e o `if`. Ambas têm uma estrutura quase idêntica àquelas que existem no Python.

Para usar a template tag `for`, usamos ela dentro dos marcadores `{% ... %}`, com uma sintaxe igual a do Python, ou seja, se tivermos no contexto da aplicação uma lista, podemos escrever `{% for item in lista %}` (Template Tag `for`, s.d.). Para marcar o fim do bloco `for`, usamos o marcador `{% endfor %}`. Note que é importante marcar o fim do bloco, pois diferente da linguagem Python, o Jinja não considera as indentações e espaços, portanto essa é a única forma de informar ao Jinja onde termina o laço `for`. Tudo que estiver entre a abertura e o fechamento do bloco `for` será repetido para cada item a lista. Além disso, a variável `item` estará disponível dentro do bloco para ser utilizada em expressões como a impressão: `{{ item }}`.

A template tag `if` também funciona de maneira análoga ao `if-elif-else` do Python. Usamos o marcador `{% if CONDICAÇÃO %}` para abrir o bloco [Template Tag `if`, [S.d.]]. A condição do bloco `if` respeita as mesmas regras que a do Python, com os mesmos operadores lógicos (`and`, `or` e `not`). Assim como o `for`, precisamos fechar o bloco com o marcador `{% endif %}`. Entre o marcador de abertura e o de fechamento, podemos ter vários marcadores `{% elif CONDICAÇÃO %}` com condições diferentes e um último marcador `{% else %}`. Outras estrutura de controle podem ser estudadas em (List of Control Structure, s.d.).

Herança de Templates

Uma das grandes vantagens do Jinja2 é a herança de templates. É possível criarmos um template de modelo, com todas as estruturas básicas de uma página, como o cabeçalho, rodapé e navegação, bem como as tags de configuração dentro da `head` do HTML. Depois todos os outros templates que criarmos poderão herdar essa

estrutura toda e se concentrar apenas no seu conteúdo específico (Template Inheritance, s.d.). Um exemplo do uso de herança de templates que utiliza como base o projeto da seção 10.3.1 está disponível em: <https://github.com/rwill-impacta/flask-heranca-templates>.

Para isso, criamos um outro arquivo HTML chamado base.html (poderia ser qualquer outro nome) com essa estrutura toda em comum. Depois, devemos marcar todos os pontos onde os templates que herdarem o modelo deverão introduzir seus conteúdos. Usamos a template tag `{% block NOME %}`. Assim como outras template tags há necessidade de fechar o bloco com `{% endblock %}`. Essa tag serve para marcar uma região que terá conteúdo introduzido por outro template, através da herança de templates. Após a construção do modelo, todos os templates que queiram herdar a estrutura devem fazer duas coisas: indicar a herança usando a template tag `{% extends "MODELO" %}` (no Exemplo do nosso projeto: `{% extends "base.html" %}`) e definir quais blocos vão ser inseridos no modelo, usando a mesma template tag `{% block NOME %}` com o mesmo nome definido no modelo.

A Figura 10.2 mostra um exemplo completo, retirado do projeto que disponibilizamos no github. O primeiro código representa o arquivo base.html onde é definida uma estrutura básica do HTML e dois blocks: um no título e outro dentro do body chamado conteúdo. O segundo código representa o arquivo index.html, que usa a template tag `{% extends "base.html" %}` para indicar que o index.html usará a estrutura do base.html. No index.html temos os dois blocks definidos para a inserção de conteúdo: o título e o conteúdo. O resultado é que o template index.html terá todo o código HTML presente no arquivo base.html, mas os trechos definidos pelo block título será substituído por `<title>Página inicial</title>`, enquanto que o trecho definido pelo block conteúdo será substituído pelas tags h1 e p definidas dentro desse bloco.

Arquivo: base.html

```
<!DOCTYPE html>
<html>
<head>
  {% block titulo %}{% endblock %}
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/static/estilos.css">
</head>

<body>
  <header>
    <nav>
      <a href="/">Página inicial</a> |
      <a href="/cursos">Todos os cursos</a> |
      <a href="/curso/devweb">Desenvolvimento Web</a> |
      <a href="/curso/poo">Programação Orientada a Objetos</a> |
      <a href="/alunos">Média dos alunos</a>
    </nav>
  </header>

  <main>
    {% block conteudo %}{% endblock %}
  </main>

  <footer>
```

```
<p>Rodapé do projeto de demonstração.</p>
</footer>
</body>
</html>
```

Arquivo: index.html

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

{% block titulo %}
    <title>Página inicial</title>
{% endblock %}

{% block conteudo %}
    <h1>{{ titulo }}</h1>
    <p>Página inicial do projeto que demonstra o uso de template tags.</p>
{% endblock %}
```

Arquivos Estáticos

Nem todo conteúdo na web é dinâmico. Alguns arquivos já estão prontos e só devem ser transferidos para o navegador poder completar a renderização junto com o HTML. É o caso de imagens, vídeos, arquivos CSS e JavaScript. Idealmente esses arquivos devem ser servidos por serviços e servidores especializados, como CDNs (Content Delivery Networks), pois eles são mais bem preparados para lidar com questões de entrega e cache de navegadores. Mas podemos servir esses arquivos de dentro do Flask também.

Por padrão, esses arquivos precisam estar na pasta static na raiz do projeto, mas pode ser modificado passando a opção `static_folder=PASTA` no construtor da classe Flask. Com isso, todos os arquivos estáticos que estejam na pasta static podem ser acessados pelo caminho `/static/arquivo`. O prefixo do path pode ser alterado pela opção `static_url_path=CAMINHO` no construtor da classe Flask. (Static Files, s.d.).

Controle de Sessão - Autenticação e Autorização

Nem tudo que existe na Internet deve ser visto por todos. A sua caixa de e-mail é apenas sua para ver, a área de um professor no site de uma universidade é para apenas ele mexer e assim por diante. Existem conteúdos que não são públicos e endereçados a pessoas específicas, e para garantir isso existem os sistemas de autenticação (garantem que usuário é ele mesmo) e autorização (garantem que o usuário tem permissão para isso) nas aplicações, de forma a proteger nosso conteúdo.

Com o crescimento do uso da internet para praticamente todos os tipos de negócios e usos, cresce também a preocupação com os dados que estão trafegando na rede. Além de segurança na parte física (roteadores, redes, cabeamentos, etc), precisamos nos preocupar com a segurança nas nossas aplicações. Os dados dos usuários devem estar protegidos de modo a garantir que apenas eles consigam ver esses dados (ou alguém com sua permissão).

Nesse sentido surgem dois sistemas comuns em aplicações da web: o sistema de autenticação e o sistema de autorização. Ambos são complementares um do outro, sendo necessários para termos certeza de que as funcionalidades e dados estejam sendo acessadas pelas pessoas corretas na nossa aplicação. Existem diversos

mecanismos para garantir a segurança e modelagem destes sistemas. Em nossa disciplina iremos cobrir o mais clássico e utilizado: o sistema de autenticação por usuário e senha.

Fluxo de Autenticação

Existem diversos mecanismos de autenticação de usuários (sejam pessoas ou outros computadores): certificados digitais, biometria facial ou dos dedos, captchas, etc (AMOROSO, 2009). Mas até os dias atuais, o mais comum encontrado nos sites da Internet é o sistema de usuário e senha.

Dependendo do mecanismo de autenticação utilizado, o fluxo do sistema de autenticação pode variar um pouco, mas a estrutura é muito similar entre eles. A Figura 11.1 mostra a sequência de passos referente ao sistema de usuário e senha. Em seguida vamos detalhar um pouco mais cada uma dessas cinco etapas.



Envio de dados de autenticação

Essa fase é a que envolve a interação direta do usuário. No nosso exemplo, o usuário deverá digitar seu nome de usuário, sua senha e submeter o formulário ao servidor. Em alguns casos, temos a utilização de um captcha, um mecanismo de identificação de robôs na navegação dos sites.

Algumas outras técnicas permitem que o nosso site delegue essa função para outros provedores de identidade. Esse é o caso do OAuth2, que permite que a nossa aplicação delegue para um outro serviço ou aplicação (como o Google) a verificação de autenticidade do usuário. Nesse caso, pulamos direto para a etapa 4 (da Figura 11.1), pois a 1, 2 e 3 seriam feitas por esse serviço OAuth2 (ANDREY, 2017).

Recebendo valores por GET e POST

Para construir uma aplicação que permita o usuário autenticar-se através de usuário e senha, primeiro precisamos saber receber esses dados em uma rota. Para entender o mecanismo necessário para o recebimento de valores de formulários em uma rota, usaremos o projeto Flask disponível em: <https://github.com/rwill-impacta/flask-sessao-cliente>

Todos os controles no Flask, por padrão, respondem apenas às requisições com o método GET. Para responder às requisições com o método POST (método HTTP que permite enviar os dados do formulário no corpo da requisição), é necessário mudar uma configuração na definição de rota, como mostrado no decorador da função exemplo_post(). No projeto de exemplo temos dois controles similares que exemplificam como receber valores por GET (rota 'exemplo_get') e POST (rota 'exemplo_post'). Em ambos os casos, configuramos templates

(exemplo_get.html e exemplo_post.html, respectivamente) que carregam um formulário simples com os campos de usuário (com o nome `usuario`) e senha (com o nome `senha`). Para receber os valores associados a esses campos dentro dos controles, usamos um objeto chamado `request`, que deve ser importado a partir do módulo `Flask`.

O objeto `request` disponibiliza dois atributos que funcionam como dicionários: `request.args`, que contém os valores enviados via GET, e `request.form`, que contém os valores enviados por POST. As chaves desses dicionários são os nomes enviados pelo formulário, enquanto que os valores representam os valores associados a cada um desses nomes. Ambas as rotas (`/exemplo_get` e `/exemplo_post`) verificam se os valores de usuário e senha recebidos são strings vazias, e se não forem, mostram esses valores.

Os controles no Flask permitem configurar também os métodos HTTP que eles aceitarão no formato de uma lista do Python, usando o parâmetro `methods` (Exemplo: `@app.route('/exemplo_post', methods=['GET', 'POST'])`). Ou seja, podemos ter rotas que aceitam somente requisições por GET, ou somente por POST, ou ambas. No caso de rotas que aceitam múltiplos métodos HTTP, podemos identificar se a requisição foi feita usando GET ou POST através do atributo `request.method`, cujo valor é uma string com o nome do método usado na requisição (Exemplo: `if request.method == 'POST'`, testa se a requisição foi feita usando o método POST).

Definindo a sessão do usuário

De posse dos dados de autenticação, o servidor precisa verificar a autenticidade deles. Primeiro o servidor procura em seu catálogo de usuários, que pode ser um banco de dados, arquivos ou serviço externo. Caso não exista, a requisição já retorna com alguma mensagem de erro. Um exemplo simplista que demonstra a autenticação de usuário está disponível no projeto (o mesmo que usamos na seção 11.2.2): <https://github.com/rwill-impacta/flask-sessao-cliente>

Em nosso exemplo, a rota `/login` carregará um formulário quando acessada diretamente pelo navegador (através do método GET). Uma vez que este formulário é submetido ao servidor (através do método POST), utilizaremos essa mesma rota para acessar as credenciais enviadas (nome de usuário e senha), e comparamos com dois usuários hipotéticos: `rafael`, com a senha `1234`, e `maria`, com a senha `4321`. Mais adiante veremos como isso pode ser feito com um catálogo de usuários em um banco de dados.

Quando a aplicação tem certeza que o usuário é ele mesmo, definimos um objeto que representa a sessão do usuário. A sessão é uma vinculação da utilização da aplicação por um usuário em um contexto específico. Esse contexto, em geral, abrange: qual computador o usuário está usando para acessar (especificamente o endereço de IP), qual navegador ele está usando (no caso da web) e dados de geolocalização, entre outros que o servidor tenha acesso e ache pertinente. Na estrutura de sessão podemos guardar informações específicas da interação do usuário com a aplicação, como um carrinho de compras em uma aplicação de e-commerce, por exemplo. A sessão pode estar representada por um objeto em uma estrutura de dados na memória do servidor, uma linha de tabela de banco de dados ou um arquivo de texto.

No Flask, existe um objeto global que pode ser importado a partir do módulo, chamado de `session` (para importá-lo, usamos o comando: `from flask import session`). Esse objeto se comporta como um dicionário Python (uma estrutura de dados), que permite guardar os dados referentes à sessão de usuário. Em nosso exemplo, vamos guardar o nome do usuário da seguinte forma: `session['usuario'] = 'NOME_DO_USUARIO'` (obviamente, substituindo essa string pelo nome do usuário em questão). A única restrição é que os dados guardados no objeto `session` sejam possíveis de serializar e desserializar em strings. Note que, por ser um dicionário, poderíamos guardar outros valores úteis, como por exemplo o perfil do usuário (Exemplo: `session['perfil'] = 'aluno'`), e assim autorizar (ou não) o usuário a visualizar certas páginas da aplicação. Na rota `/login`, após

guardar o(s) valor(es) na sessão, basta retornar um redirecionamento para a página `/area_logada`, usando a função `redirect()` (que também deve ser importada do pacote `flask`). A rota `/area_logada` representa uma página privada que só pode ser visualizada por usuários devidamente autenticados pelo sistema.

Controle de Sessão

Vimos que a sessão de usuário é uma referência do acesso do usuário em um contexto (navegador, ip, computador, etc) com a nossa aplicação. Essa referência é feita para que qualquer outra requisição do usuário à nossa aplicação não precise efetuar a autenticação novamente, pois já sabemos que nesse contexto as requisições são feitas pelo nosso usuário autenticado.

Como as requisições HTTP são stateless (sem estado), ou seja, uma requisição não guarda informações para a outra, precisamos de outros mecanismos para guardar a sessão. A maneira mais clássica é a utilização dos cookies. Resumidamente, um cookie é composto por, no mínimo, um nome (identificador para o navegador) e um valor (nesse caso um identificador de sessão) (Cookies HTTP, s.d.). Esses valores são armazenados pelo navegador, e sempre estão disponíveis em futuras requisições para o servidor que definiu aquele cookie até que ele expire (isso pode ser definido por tempo de duração, ou ao fechar o navegador, etc).

Há técnicas mais atuais para a transmissão da informação da sessão em formato de tokens criptografados, que usam a linguagem JavaScript na tradução, armazenamento e transmissão. A técnica mais utilizada atualmente é a do JWT (JSON Web Token) (JSON Web Tokens, s.d.), mas em nossa disciplina usaremos os cookies, pois o Flask já vem com a lógica simplificada para usar essa técnica.

Voltando ao exemplo definido no nosso projeto em Flask, se olharmos a resposta após a manipulação do `session`, veremos que um cookie foi criado no navegador com o nome `session` (você pode visualizar esse cookie usando as Ferramentas de Desenvolvedor no Chrome ou Firefox, por exemplo). O Flask pega os dados presentes no objeto `session` e serializa (traduz para string) no cookie usando criptografia, e envia essa informação ao cliente. A criptografia é um processo de codificar uma certa quantidade de dados de forma que eles fiquem impossíveis de serem interpretados sem que o leitor conheça o segredo para decodificar esses dados (KASPERSKY, s.d.). É um mecanismo muito importante em todas as comunicações de rede, não só no HTTP (inclusive o HTTPS usa uma forma de criptografia).

Após esse processo, toda requisição em que esse cookie esteja presente, ele será desserializado (traduzido de volta para objeto) e o objeto `session` fica povoado no Flask. Para que esse mecanismo funcione, temos que configurar uma chave secreta na aplicação Flask, que será utilizada nesse processo de criptografia. No objeto que representa a nossa aplicação (variável `app`), devemos definir o atributo `secret_key` com uma chave (string) secreta, que só o servidor deve saber: `app.secret_key='SENHA-MUITO-SECRETA'` (na vida real, utilize uma string muito grande, com vários caracteres especiais, letras maiúsculas e minúsculas, números, etc).

Em resumo, essa chave secreta será usada internamente pelo Flask para criptografar o objeto de sessão e enviá-lo ao navegador do usuário, que armazenará essa informação como um cookie. Note que apesar do usuário ter acesso à string criptografada (pois é fácil visualizar os cookies armazenados), não é possível que o usuário mude o valor do cookie para um valor válido (que faça sentido para o servidor), pois isso implicaria na quebra da criptografia utilizada. Ou seja, o servidor envia o objeto criptografado ao usuário e pede que ele o armazene (assim o objeto ficará disponível em futuras requisições), mas só o servidor conseguirá decifrar o que há dentro daquele objeto. É assim que as sessões dos usuários são mantidas!

Exibindo uma página privada ao usuário

Em nosso projeto de exemplo, a rota `/area_logada` mostra uma página que o usuário só pode ver caso tenha realizado o processo de autenticação corretamente. Para verificar se o usuário está logado, o servidor deve verificar se o objeto de sessão está povoado com informações que só o servidor poderia ter colocado lá. No nosso caso, verificamos se a chave `'usuario'` do objeto `session` (que funciona como um dicionário do Python) está definida, e se está, quem é o usuário logado. Note que um usuário mal intencionado não conseguiria montar esse objeto, uma vez que do lado do cliente (no navegador) essa informação está armazenada criptografada. Somente o servidor consegue ter acesso a esses valores, pois só ele conhece a chave secreta usada na criptografia.

Adicionalmente, poderíamos ter adicionado mais informações no objeto da sessão. Suponha que essa rota só deve ser exibida para usuários do perfil "aluno". Ao autenticar o usuário, seria possível adicionar duas chaves no dicionário: `session['usuario'] = 'NOME_DO_USUARIO'` e também `session['perfil'] = 'aluno'`. Assim, seria possível conferir essas duas informações a cada nova requisição à rota `/area_logada`, e caso o usuário não tenha o perfil adequado para visualizá-la, o controle associado a essa rota retornaria um template de erro, informando que o usuário não possui a permissão adequada.

Expiração do cookie de sessão

Por padrão, o cookie de sessão do Flask expira quando o navegador é fechado, fazendo com que a sessão do usuário expire (ou seja, o usuário terá que refazer o processo de autenticação). Podemos mudar essa configuração para que ele expire depois de algum tempo (Exemplo: 1 minuto, 5 minutos, 1 hora e meia, etc), mesmo que o usuário não tenha fechado o navegador. Para isso, precisamos criar uma sessão permanente no Flask, incluindo o seguinte controle em nossa aplicação. A Figura 11.2 ilustra o código adicional necessário para que todas as sessões definidas pelo servidor expirem em 1 minuto e meio.

Arquivo: `app.py`

```
# Além de importar as classes, funções e objetos do Flask, você deve importar a seguinte função do módulo
datetime:
from datetime import timedelta
@app.before_request
def before_request():
    session.permanent = True
    app.permanent_session_lifetime = timedelta(minutes=1, seconds=30)
```

Encerramento da sessão

Como a sessão está intimamente ligada com o cookie, um só pode existir em conjunto com o outro. Dessa forma o encerramento de sessão deve ser acompanhado com o encerramento do cookie, e isso ocorre de duas formas possíveis: encerramento por ação do usuário (logout) e expiração do cookie.

Caso o usuário queira encerrar sua sessão por conta própria, o servidor irá limpar o objeto de sessão e avisar o navegador de que o cookie associado a sessão deve ser removido. Isso é feito criando-se um controle no Flask, onde dentro dele chamamos o método `session.clear()`. A rota `/sair` do nosso projeto de exemplo faz exatamente isso, e logo em seguida redireciona o usuário para a rota `/login`.

Você quer ler?

Existem diversos sistemas de autenticação e autorização utilizados nas aplicações web mais modernas. OAuth, SAML, OpenID são alguns desses sistemas. Associado a eles, existe um mecanismo de controle de sessão

moderno que permite o controle ser feito a partir do cliente, além do servidor. Esse mecanismo é o JWT, ou JSON Web Token. O JWT transmite informações da sessão de maneira criptografada, permitindo que ele seja controlado e acessado por todos que sabem como lê-lo. Veja mais detalhes em <https://imasters.com.br/back-end/entendendo-o-jwt>.

Aprimorando o nosso projeto com banco de dados

Agora que você entendeu como funciona o mecanismo de sessão de usuário no Flask, vamos aprimorar o nosso exemplo com um banco de dados contendo alguns usuários. Para facilitar a criação e manipulação do banco de dados, usaremos o SGBD (Sistema Gerenciador de Banco de Dados) relacional SQLite (disponível em <https://www.sqlite.org/>), que já vem embutido na linguagem de programação Python, facilitando a instalação, e possui uma sintaxe baseada no PostgreSQL. O projeto completo está disponível no seguinte link: <https://github.com/rwill-impacta/flask-autenticacao-bd>.

Antes de rodar esse projeto, será necessário instalar o pacote SQL Alchemy (disponível em <https://www.sqlalchemy.org/>), um módulo Python que facilita a manipulação e integração do Python com SGBDs. Neste novo projeto nós importamos esse pacote, portanto não será possível executá-lo sem fazer essa instalação. Para isso, utilize o comando mostrado na Figura 11.3 no CMD do Windows ou terminal do Linux/MAC (você deve usar a conta de administrador do computador).

Figura 11.3. Instalação do pacote SQL Alchemy

```
# Instalação no Windows (através do cmd) :  
py -m pip install -U sqlalchemy  
  
# Instalação no Linux ou Mac (através do Terminal) :  
python3 -m pip install -U sqlalchemy
```

Uma vez que as configurações necessárias tenham sido efetuadas com sucesso, vamos criar as tabelas do banco de dados e adicionar alguns dados. No diretório models você encontrará o arquivo banco_de_dados.py. Execute esse arquivo diretamente e escolha a opção “1- Criar tabelas e inserir dados no Banco de Dados”. Note que um novo arquivo será criado nesse mesmo diretório, com o nome: dados.db. Esse arquivo armazena todo o nosso banco de dados que criamos com o SGBD SQLite.

Como o objetivo é ser didático, nosso banco de dados é muito simples e só possui uma única tabela chamada Aluno. A Figura 11.4 ilustra todas as 6 colunas dessa tabela e seus dados armazenados. Se desejar, você pode listar todos os dados armazenados, rodando novamente o arquivo banco_de_dados.py e escolhendo a opção “2- Listar os dados gravados no Banco de Dados”.

Table: Aluno						
	usuario	senha	nome	curso	data_inicio	media
	Filter	Filter	Filter	Filter	Filter	Filter
1	rafael	d404559f602eab6fd602ac7680dacbfaadd13630335e951f097...	Rafael Will	Ciência da ...	01/02/2022	7.5
2	maria	7e2feac95dcd7d1df803345e197369af4b156e4e7a95fcb2955...	Maria dos S...	Análise e D...	11/08/2022	8.6
3	jose	9961d468d1563f74f3b425ea9972d8d7b661838c806781156c...	José Silva	Sistemas de...	31/05/2022	6.9
4	ana	0a6f9ebaa55e21ce270b6df2e7d812c987d511ab0472d24b50...	Ana Beatriz	Engenharia ...	05/10/2022	9.2

O diretório `models` possui outro arquivo chamado `aluno.py`. Esse arquivo implementa a classe `Aluno`, que define 6 atributos, mapeando cada um dos atributos da tabela de mesmo nome contida no BD em SQLite. Já o arquivo `banco_de_dados.py` possui a classe `BancoDeDados`, que disponibiliza alguns métodos para manipulação dos dados contidos no banco de dados. Cada um dos métodos possui um comentário explicando a sua função, mas vamos focar a nossa atenção no método `obter_aluno(usuario)`, que recebe um nome de usuário como parâmetro e retorna um objeto da classe `Aluno`, caso exista um aluno com esse nome de usuário, ou o valor `None`, caso não exista nenhum aluno com o nome de usuário informado.

Armazenando senhas

É extremamente não recomendável guardar as senhas dos usuários como texto limpo (plain text), pois isso abre brechas para vazamentos e quebra de privacidade se qualquer outra pessoa obtiver acesso onde as senhas estão guardadas. O ideal é criptografar as senhas dos usuários e guardar somente esse resultado. Jamais armazene as senhas originais no sistema.

Ao observar os dados listados na Figura 11.4, você deve ter notado que a coluna senha do BD parece um pouco estranha. Na verdade, antes de gravar a senha no BD, usamos um algoritmo de criptografia de mão única (algoritmo de hashing) (SOUZA, 2020), chamado SHA-512. Essa categoria de criptografia garante que uma mesma sequência de dados sempre resulte na mesma sequência criptografada toda vez que for aplicado o mesmo algoritmo (no nosso caso, o SHA-512). Dessa forma, quando o usuário define sua senha pela primeira vez, usamos o algoritmo para guardar o valor criptografado e, toda vez que ele usar sua senha, criptografamos essa senha e comparamos com o que está guardado no banco de dados. Se ambas as versões codificadas forem iguais, sabemos que as senhas também são (mesmo sem saber a senha em específico). Veja uma explicação mais detalhada em (PALMEIRA, 2018). Caso não sejam iguais, devemos retornar uma mensagem de erro ao usuário. A Figura 11.5 ilustra como calcular o hash SHA-512 de uma string (variável `s`) em Python. Note que a variável hash sempre possui tamanho fixo (128 caracteres), independente do texto contido na string `s`.

```
import hashlib
s = '1234'
hash = hashlib.sha512(s.encode('UTF-8')).hexdigest()
print(hash)
print("Tamanho do hash:", len(hash))
```

Na prática, somente calcular o hash da senha e armazená-la no BD ainda não configura uma boa prática de segurança. Adicionalmente, é necessário “salgar as senhas” (técnica conhecida como “password salting”), para evitar que dois usuários que usem a mesma senha produzam hashes idênticos. Não implementamos essa técnica em nosso projeto para não adicionar mais níveis de complexidade, mas você pode ler a respeito sobre esse problema e a técnica de salting em (BRITO, 2019).

Manipulando a sessão

A lógica de autenticação está presente no controle associado à rota `/login`. Ele verifica se a requisição utiliza o método POST, e em caso afirmativo, obtém as strings de usuário e senha que foram enviadas. Em seguida, calcula-se o hash SHA-512 da senha digitada pelo usuário (isto é, a senha que foi enviada por POST), pois desejamos compará-la com o hash armazenado no banco de dados: se forem iguais, é a prova de que o usuário é de fato quem ele diz ser.

Com o nome do usuário, é fácil verificar se ele existe no BD através do método `obter_aluno(usuario)` da classe `BancoDeDados`, e se existir, trazemos todos os dados daquele usuário, através de um objeto da classe `Aluno`. A

partir deste objeto, temos acesso à senha criptografada para comparar o seu valor com o hash que acabamos de calcular.

Uma vez que o nome do usuário existe (comparando se o objeto retornado não é igual a None) e as senhas criptografadas são iguais, autenticamos o usuário, criando a sua sessão com a instrução: `session['usuario'] = obj_aluno.usuario`. Ou seja, o nome do usuário logado será guardado junto à chave 'usuario' no objeto de sessão do servidor, ao mesmo tempo em que o servidor definirá o cookie session com essa informação criptografada com a chave secreta definida no atributo `app.secret_key`, que só o servidor conhece.

Proteção dos controles

Alguns controles não deverão ser acessados por usuários não autenticados. Para isso, em cada controle que deve haver essa proteção, podemos verificar o nome do usuário logado através do objeto global session. Caso não exista usuário logado, então não devemos deixar o usuário continuar nesse controle, redirecionando-o para a tela de autenticação. É possível inclusive construir um decorador customizado para esse propósito (View Decorators, s.d.).

Em nossa aplicação de exemplo, o controle associado à rota 'area_logada' verifica se o objeto global session possui a chave 'usuario', pois se existir, é um sinal de que só o servidor poderia ter adicionado esse valor lá. Se existir, obtemos o nome do usuário logado, e buscamos os seus dados no banco de dados, novamente usando o método `obter_aluno(usuario)` da classe `BancoDeDados`. Como este método retorna um objeto da classe `Aluno`, podemos passar esse objeto carregado com os dados de um aluno diretamente para a função `render_template()`, e acessar seus atributos públicos, propriedades (properties) e métodos dentro de um template qualquer, através de template tags.

AJAX

Vimos como construir uma aplicação completamente dinâmica em relação aos dados que estão no servidor. Mas cada chamada feita no servidor é bloqueante, ou seja, enquanto as coisas acontecem no servidor, o navegador fica travado esperando a resposta. Através do JavaScript, podemos utilizar um mecanismo de requisições assíncronas, que não travam o resto do processamento do navegador para obter sua resposta.

Aprendemos a usar as tecnologias que rodam no navegador para construir páginas dinâmicas, com layout fluído e forte semântica. Também construímos uma aplicação no servidor mais complexa, capaz de responder as requisições e criar conteúdos dinâmicos baseados nos nossos dados.

A grande limitação do que vimos é o fato de que apenas uma requisição pode ser feita por vez para obter ou enviar dados. Seja a submissão de um formulário, o clicar em uma âncora ou a manipulação da barra de endereços, ao fazer a requisição para o servidor, o navegador fica bloqueado, ou seja, não podemos interagir diretamente com ele ou com a página, até que a requisição termine e o resultado seja renderizado na tela.

Outro problema é que mesmo requisições que vão obter um conjunto pequeno de dados (Exemplo: uma lista de quatro cursos), precisam retornar uma quantidade bem maior por conta do layout do site, outros scripts e estilos, e outros dados para remontar a página novamente. Veremos uma técnica em JavaScript para contornar esses problemas encontrados nas estruturas clássicas de sites.

Síncrono vs Assíncrono

Dizemos que requisições são bloqueantes no navegador quando ao fazer a requisição precisamos esperar ela responder para poder voltar a interagir com a página. Uma consequência desse processo é que não podemos

fazer requisições deste tipo simultaneamente. Uma é disparada após a outra, sem paralelismo de nenhum tipo. Nesse caso, dizemos que as requisições são síncronas, ou seja, cada instrução é executada após a anterior finalizar. Scripts Python ou JavaScripts comuns são síncronos: cada linha é executada assim que a anterior termina de executar.

Mas e se quiséssemos fazer algumas tarefas em paralelo? Muitas linguagens de programação fornecem maneiras de executarmos códigos de maneira paralela, usando múltiplas threads (processos separados no sistema operacional para a execução de um programa). Porém, o JavaScript é uma linguagem single thread, ou seja, ela possui uma única thread e não pode abrir outras durante a execução. Para contornar esse problema, o JavaScript utiliza um mecanismo para tornar algumas requisições assíncronas, ou seja, possíveis de serem executadas em paralelo (Conceitos gerais da programação assíncrona, s.d.).

AJAX - Asynchronous JavaScript and XML

A técnica desenvolvida no JavaScript para requisições assíncronas recebeu o nome de Asynchronous JavaScript and XML (AJAX), ou JavaScript Assíncrono e XML. Essa técnica define as maneiras de ser possível fazer requisições assíncronas usando o JavaScript e de como controlá-las.

Além da possibilidade de chamadas assíncronas, o AJAX permitiu que dados pudessem ser requisitados ao servidor, sem que haja necessidade de desmontar a tela inteira e recarregá-la novamente do zero, trazendo todos os dados do servidor. Basta efetuar a requisição da parte necessária via JavaScript e manipular o resultado a partir dele.

Os objetivos XMLHttpRequest e fetch

Temos duas maneiras de executar requisições assíncronas no JavaScript: com o objeto XMLHttpRequest ou o fetch API. O segundo, mais moderno, não é adotado em versões antigas dos navegadores, então nem sempre pode ser usado. Vamos ver como cada um deles funciona. Exemplos do uso desses objetos podem ser vistos em: <https://github.com/rwill-impacta/flask-ajax>

XMLHttpRequest

Para criar requisições assíncronas da maneira clássica, precisamos instanciar um objeto do tipo XMLHttpRequest (XMLHttpRequest, s.d.) no JavaScript: `let xhr = new XMLHttpRequest()`. Ou seja, todos os métodos que chamaremos a seguir usam esse objeto que chamamos de xhr (você pode chamá-lo de qualquer outro nome válido para uma variável). A partir deste objeto, vamos configurar os dados da requisição e executá-la.

O primeiro passo é abrir uma conexão com o servidor, usando a função `open`: `xhr.open(MÉTODO, URL, ASYNC)`. Nesta função, passamos os seguintes parâmetros: o método HTTP utilizado (GET ou POST, inclusive é possível usar outros métodos), a URL (absoluta ou relativa) que irá responder a requisição, e por fim um parâmetro booleano indicando se a requisição será assíncrona ou não (caso este valor seja omitido, seu valor será true, ou seja, a requisição será assíncrona).

Com a conexão aberta no objeto xhr, podemos ainda configurá-la com cabeçalhos de requisição específicos, usando o método `xhr.setRequestHeader(NOME, VALOR)`, passando o nome e o valor do cabeçalho. Nem sempre é necessário usar esse método (somente em casos específicos), e em nosso exemplo ele foi omitido. Depois basta enviar a requisição usando o método `xhr.send()`. Se for necessário enviar dados no corpo da requisição, é possível codificar esses dados em uma string e passar como parâmetro do método `send()`.

Como a requisição por padrão é assíncrona, não podemos esperar que após a execução do `send()` teremos os dados imediatamente disponíveis para uso. A requisição pode acabar demorando por algum motivo de servidor ou rede, por isso é necessário pedir para o `XMLHttpRequest`, através do objeto instanciado `xhr`, avisar quando a resposta chegar. Para tal, usamos o evento `readystatechange`. Este evento é disparado toda vez que a propriedade `readyState` do objeto `xhr` muda. O valor dessa propriedade muda toda vez que o estado da requisição é alterado (a requisição foi enviada, os dados de cabeçalho foram recebidos, os dados começaram a chegar, a requisição foi finalizada, etc). A requisição estará finalizada quando o valor dela for 4 (valores menores se referem a situações intermediárias de rede).

Portanto, devemos associar um listener ao evento usando a propriedade `onreadystatechange`. Dentro da função que responde este evento, devemos verificar se a requisição finalizou (`xhr.readyState === 4`) ou usando a constante `XMLHttpRequest.DONE`, cujo valor também é 4, e também se ela foi bem sucedida, isto é, se o código de status HTTP é igual a 200: (`xhr.status === 200`). Caso as duas condições lógicas sejam verdadeiras, podemos acessar a resposta usando a propriedade `xhr.responseText`. A Figura 12.1 mostra um exemplo de uma requisição usando o `XMLHttpRequest`.

Code:

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "/obter_dados");

xhr.onreadystatechange = function() {
  if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {
    console.log("A requisição deu certo e retornou: " + xhr.responseText);
  }
}
xhr.send();
```

fetch API

O JavaScript possui mais uma interface para execução de requisições assíncronas, o `fetch API`. Esse objeto possui uma maior simplicidade na criação das chamadas e no tratamento de respostas.

Ele está disponível através da função global `fetch`. Executamos ela passando a URL da requisição e um objeto de opções (Exemplo: `fetch(URL, CONFIG)`) (Usando `Fetch s.d.`). Nesse objeto podemos passar o método HTTP, cabeçalhos de requisição customizados, dados do corpo, entre outros dados. Ao executar essa função, é retornado um objeto de `Promesa (Promise)` (`Promise, s.d.`). Esse objeto representa uma execução assíncrona qualquer que terá seu valor disponibilizado no futuro (ou seja, quando o servidor terminar de responder a requisição).

Um objeto do tipo `Promise` permite conectar dois tipos de retorno: um para o sucesso e outro para tratar um erro na requisição. Ambos devem ser feitos passando uma função a ser executada quando cada um dos eventos ocorrer (sucesso ou erro). Para isso, usamos o método `then()` para passar uma chamada a ser executada no sucesso, recebendo os dados de resposta, ou o método `catch()` para tratar o erro da chamada, recebendo um objeto com os dados do erro.

O retorno do bloco `then()` é um objeto de resposta com os dados sendo processados ainda, então precisamos retornar qual o formato ele será interpretado. Na resposta retornada podemos usar o método `.json()` para formatar em um object genérico do JavaScript. No exemplo mostrado na Figura 12.2 a requisição é feita para a

URL /obter_dados, e registramos duas funções de retorno ao seu promise: a primeira função recebe a resposta a ser codificada e retorna ela em formato texto (string), enquanto que a segunda função recebe a string com a resposta e a imprime no console. Há ainda uma terceira função associada ao método catch(), que só será executada caso ocorra algum erro ao fazer a requisição assíncrona.

Code:

```
fetch("/obter_dados").then(function(resposta) {  
    return resposta.json()  
}).then(function (texto){  
    console.log("A requisição deu certo e retornou: " + texto);  
}).catch(function (){  
    console.log("Ocorreu algum erro ao fazer a requisição!");  
});
```

Vamos praticar?

Uma das coisas mais comuns a se fazer usando o AJAX é o consumo de APIs Públicas. As APIs são serviços web que ficam disponíveis para qualquer um utilizar para consumir seus dados. Elas podem abraçar qualquer assunto, de cinema a carros. Veja uma lista grande de APIs públicas para consumo livre para testar o uso do AJAX: <https://github.com/public-apis/public-apis>.

O formato JSON

Mesmo não sendo obrigatório utilizá-lo, o formato de dados JSON (JavaScript Object Notation) é o mais associado ao AJAX. Originalmente o AJAX era usado para transferir dados usando o formato XML (eXtensible Markup Language), sendo a origem da letra X na sigla do AJAX, inclusive. Mas a simplicidade e leveza do formato JSON acabou fazendo com que esse formato fosse o mais utilizado na troca de informações no AJAX.

O formato JSON é similar à declaração de objects genéricos do JavaScript ou dicionários do Python. Os dados sempre devem ficar entre chaves ({}) e possui várias entradas de chave e valor. Uma diferença em relação ao tipo dicionário do Python, é que no JSON todas as chaves devem ser strings (Introdução ao JSON, s.d.). Já os valores podem ser strings, números, booleanos, null, vetores e inclusive outros JSON. É possível definir valores como vetores (lista de valores), através de colchetes ([]), conforme o exemplo da Figura 12.3.

Code:

```
{  
  "nome": "João da Silva",  
  "idade": 32,  
  "telefones": [  
    "(99) 1234-5678",  
    "(99) 98765-4321"  
  ],  
  "matriculado": true,  
  "curso": {  
    "sigla": "ADS",  
    "nome": "Análise e Desenvolvimento de Sistemas"  
  }  
}
```

É possível também definir vetores (listas) contendo vários objetos JSON dentro do vetor, conforme mostra o exemplo da Figura 12.4.

```
{
  "dados_alunos": [
    {
      "nome": "João da Silva",
      "idade": 32
    },
    {
      "nome": "Maria dos Santos",
      "idade": 29
    },
    {
      "nome": "Guilherme Souza",
      "idade": 45
    }
  ]
}
```

O Flask permite representar dados no formato JSON através da função `jsonify()`, presente no pacote flask (a função deve ser importada: `from flask import jsonify`). Essa função pode ser útil, por exemplo, em rotas que serão consumidas por requisições AJAX, onde desejamos retornar texto no formato JSON ao invés de renderizar um template com a função `render_template()`. Essa função recebe vários objetos do Python (strings, dicionários, listas, etc) e converte esses dados para uma string que representa JSON válido. Cada chave do JSON deve ser passada para a função como um parâmetro nomeado, e o valor do parâmetro nomeado é o valor associado àquela chave. (Flask API - `jsonify`, s.d.).