Linux Professional Institute

Web Development Essentials

Versão 1.0
Português

030

Table of Contents

Linux
Professional
Institute

Tópico 031: Desenvolvimento de software e tecnologias web

031.1 Noções básicas de desenvolvimento de software

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 031.1

Peso

1

Áreas chave de conhecimento

- Entender o que é código-fonte
- Entender os princípios dos compiladores e intérpretes
- Entender o conceito de bibliotecas
- Entender os conceitos de programação funcional, procedural e orientada a objetos
- Conhecimento dos recursos comuns dos editores de código-fonte e ambientes de desenvolvimento integrado (IDE)
- Conhecimentos sobre os sistemas de controle de versão
- Noções de testagem de software
- Conhecimento das linguagens de programação mais importantes (C, C++, C#, Java, JavaScript, Python, PHP)

Linux Professional Institute

031.1 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	031 Desenvolvimento de software e tecnologias web
Objetivo:	031.1 Noções básicas de desenvolvimento de software
Lição:	1 de 1

Introdução

Os primeiros computadores eram programados através de um cansativo processo de conectar cabos em soquetes. Os cientistas da computação logo iniciaram uma busca incessante por maneiras mais fáceis de dizer ao computador o que fazer. Este capítulo é uma introdução às ferramentas de programação. Ele trata das principais maneiras pelas quais uma série de instruções em forma de texto — as linguagens de programação — representam as tarefas que um programador deseja realizar e as ferramentas que transformam o programa em uma forma chamada *linguagem de máquina* que pode ser executada por um computador.

NOTE Neste texto, os termos *programa* e *aplicativo* são usados de forma intercambiável.

Código fonte

Um programador normalmente desenvolve um aplicativo escrevendo uma descrição textual, chamada *código-fonte*, da tarefa desejada. O código-fonte está em uma *linguagem de programação* cuidadosamente definida que representa o que o computador pode fazer em uma abstração de alto

nível compreensível pelos humanos. Também foram desenvolvidas ferramentas para permitir que os programadores, bem como os não-programadores, expressem suas ideias visualmente, mas o código-fonte continua sendo a forma predominante de se programar.

Da mesma forma que uma linguagem natural inclui substantivos, verbos e construções para expressar ideias de forma estruturada, as palavras e a pontuação em uma linguagem de programação são representações simbólicas de operações que serão realizadas na máquina.

Nesse sentido, o código-fonte não difere muito de qualquer outro texto em que o autor emprega as regras bem estabelecidas de uma linguagem natural para se comunicar com o leitor. No caso do código-fonte, o "leitor" é a própria máquina e, portanto, o texto não pode conter ambigüidades ou inconsistências — nem mesmo as mais sutis.

E como qualquer texto que discuta algum tópico em profundidade, o código-fonte também precisa ser bem estruturado e organizado de forma lógica ao se desenvolver aplicativos complexos. Programas muito simples e exemplos didáticos podem ser armazenados em poucas linhas de um único arquivo de texto contendo todo o código-fonte do programa. Programas mais complexos podem ser subdivididos em milhares de arquivos, cada um com milhares de linhas.

O código-fonte dos aplicativos profissionais deve ser organizado em pastas diferentes, geralmente associadas a uma finalidade específica. Um programa de chat, por exemplo, pode ser organizado em duas pastas: uma que contém os arquivos de código que tratam da transmissão e recepção de mensagens pela rede e outra com os arquivos que constituem a interface e reagem às ações do usuário. De fato, é comum que haja muitas pastas e subpastas com arquivos de código-fonte dedicados a tarefas extremamente específicas dentro do aplicativo.

Além disso, o código-fonte nem sempre está isolado em seus próprios arquivos, com tudo escrito em uma única linguagem. No caso dos aplicativos web, por exemplo, um documento HTML pode incorporar código em JavaScript para complementar o documento com funcionalidades extras.

Editores de código e IDE

Existe uma variedade estonteante de maneiras de escrever um código-fonte. Assim, muitos desenvolvedores lançam mão de ferramentas que ajudam a escrever e testar o programa.

O arquivo de código-fonte não é mais que um arquivo de texto simples e, como tal, pode ser editado em qualquer editor de texto, por mais despojado que seja. Para facilitar a distinção entre código-fonte e texto simples, cada linguagem adota uma extensão de nome de arquivo autoexplicativa: `.c` para a linguagem C, `.py` para Python, `.js` para JavaScript etc. Os editores de texto de uso geral costumam reconhecer o código-fonte das linguagens mais populares e adicionam automaticamente itálico, cores e indentação para tornar o código mais legível.

Nem todo desenvolvedor escolhe editar o código-fonte em um editor de uso geral. Um *ambiente de desenvolvimento integrado* (em inglês, *integrated development environment* ou IDE) inclui um editor de texto junto com ferramentas que ajudam o programador a evitar erros de sintaxe e inconsistências óbvias. Esses editores são particularmente recomendados para os programadores menos experientes, mas os programadores experientes também podem usá-los.

Os IDEs mais populares, como o Visual Studio, o Eclipse e o Xcode, observam de forma inteligente o que o programador digita, muitas vezes sugerindo palavras (preenchimento automático) e verificando o código em tempo real. Os IDEs podem inclusive oferecer depuração e testes automatizados para identificar problemas sempre que o código-fonte é alterado.

Alguns programadores mais experientes optam por editores menos intuitivos, como o Vim, que oferece maior flexibilidade e não requer a instalação de pacotes adicionais. Esses programadores usam ferramentas externas independentes para adicionar os recursos que são integrados quando você usa um IDE.

Manutenção de código

Seja em um IDE ou usando ferramentas autônomas, é importante empregar algum tipo de *sistema de controle de versão* (VCS). O código-fonte está em constante evolução porque as falhas imprevistas precisam ser corrigidas e aprimoramentos devem ser incorporados. Uma consequência inevitável dessa evolução é que as correções e aprimoramentos podem interferir em outras partes dos aplicativos em uma grande base de código. Ferramentas de controle de versão como Git, Subversion e Mercurial mantêm um registro de todas as alterações feitas no código e a pessoa que as fez, permitindo rastrear e, eventualmente, voltar atrás em uma modificação malsucedida.

Além disso, as ferramentas de controle de versão permitem que cada desenvolvedor da equipe trabalhe em uma cópia dos arquivos do código-fonte sem interferir no trabalho dos outros programadores. Assim que as novas versões do código-fonte estiverem prontas e testadas, as correções ou melhorias feitas em uma cópia podem ser incorporadas pelos outros membros da equipe.

O Git, o sistema de controle de versão mais popular hoje em dia, permite que muitas cópias independentes de um repositório sejam mantidas por diferentes pessoas, que compartilham suas alterações como desejarem. No entanto, quer o sistema de controle de versão seja centralizado ou descentralizado, a maioria das equipes mantém um único repositório confiável, com um código-fonte e recursos sólidos. Muitos serviços online oferecem espaços de armazenamento para repositórios de código-fonte. Os mais populares dentre eles são o GitHub e o GitLab, mas o Savannah, do projeto GNU, também vale a menção.

Linguagens de programação

Existe uma grande variedade de linguagens de programação; novas linguagens são criadas a cada década. Cada linguagem de programação tem suas próprias regras e é recomendada para fins específicos. Embora as linguagens apresentem diferenças superficiais de sintaxe e palavras-chave, o que realmente as distingue são as profundas abordagens conceituais que representam, conhecidas como *paradigmas*.

Paradigmas

Os paradigmas definem as premissas nas quais se baseia uma linguagem de programação, especialmente no que diz respeito à maneira como o código-fonte deve ser estruturado.

O desenvolvedor parte do paradigma da linguagem para formular as tarefas a serem executadas pela máquina. Essas tarefas, por sua vez, são expressas simbolicamente com as palavras e construções sintáticas oferecidas pela linguagem.

A linguagem de programação é *procedural* quando as instruções apresentadas no código-fonte são executadas em ordem sequencial, como um roteiro de filme. Se o código-fonte for segmentado em funções ou sub-rotinas, uma rotina principal se encarrega de chamar as funções em sequência.

O código a seguir é um exemplo de linguagem procedural. Escrito em C, ele define variáveis que representam as faces, a área e o volume de formas geográficas. O valor da variável `side` é atribuído em `main()`, que é a função invocada quando o programa é executado. As variáveis `area` e `volume` são calculadas nas sub-rotinas `square()` e `cube()` que precedem a função principal:

```c
#include <stdio.h>

float side;
float area;
float volume;

void square(){ area = side * side; }

void cube(){ volume = area * side; }

int main(){
  side = 2;
  square();
  cube();
  printf("Volume: %f\n", volume);
  return 0;
}
```

A ordem das ações definidas em `main()` determina a sequência de estados do programa, caracterizada pelo valor das variáveis `side`, `area` e `volume`. O exemplo se encerra após exibir o valor de `volume` com a instrução `printf`.

Por outro lado, o paradigma da *programação orientada a objetos* (OOP) tem como principal característica a separação do estado do programa em subestados independentes. Esses subestados e as operações associadas são os *objetos*, assim chamados porque têm uma existência mais ou menos independente dentro do programa e porque têm finalidades específicas.

Os diferentes paradigmas não restringem necessariamente o tipo de tarefa que pode ser executada por um programa. O código do exemplo anterior pode ser reescrito de acordo com o paradigma OOP usando a linguagem C++:

```
#include <iostream>

class Cube {
  float side;
  public:
  Cube(float s){ side = s; }
  float volume() { return side * side * side; }
};

int main(){
  float side = 2;
  Cube cube(side);
  std::cout << "Volume: " << cube.volume() << std::endl;
  return 0;
}
```

A função `main()` ainda está presente. Mas agora temos uma nova palavra, `class` (classe), que introduz a definição de um objeto. A classe definida, chamada `Cube`, contém suas próprias variáveis e sub-rotinas. Na OOP, uma variável também é chamada de *atributo* e uma sub-rotina é chamada de *método*.

Nesta lição, não há interesse em explicar todo o código C++ do exemplo. O que é importante para nós neste caso é saber que `Cube` contém o atributo `side` e dois métodos. O método `volume()` calcula o volume do cubo.

É possível criar vários objetos independentes pertencentes à mesma classe; as classes também podem ser compostas por outras classes.

Lembre-se de que esses mesmos recursos podem ser escritos de maneira diferente e que os exemplos deste capítulo são bem simplificados. C e C++ possuem recursos muito mais sofisticados, possibilitando construções bem mais complexas e práticas.

A maioria das linguagens de programação não impõe rigorosamente um paradigma, mas permite que os programadores escolham diversos aspectos de um paradigma ou de outro. O JavaScript, por exemplo, incorpora aspectos de diferentes paradigmas. O programador pode decompor o programa inteiro em funções que não compartilham um estado comum entre si:

```
function cube(side){
  return side*side*side;
}

console.log("Volume: " + cube(2));
```

Embora este exemplo seja semelhante à programação procedural, note que a função recebe uma cópia de todas as informações necessárias para sua execução e sempre produz o mesmo resultado para o mesmo parâmetro, independentemente das alterações que ocorram fora do escopo da função. Este paradigma, denominado *funcional*, é fortemente influenciado pelo formalismo matemático, no qual toda operação é autossuficiente.

Outro paradigma cobre as linguagens *declarativas*, que descrevem os estados em que se deseja que o sistema esteja. Uma linguagem declarativa pode descobrir como atingir os estados especificados. O SQL, a linguagem universal para consulta de bancos de dados, às vezes é chamada de linguagem declarativa, embora na verdade ocupe um nicho único no panteão da programação.

Não existe um paradigma universal que possa ser adotado em qualquer contexto. A escolha da linguagem também pode ser restrita pelos idiomas suportados na plataforma ou ambiente de execução em que o programa será usado.

Um aplicativo web que será executado pelo navegador, por exemplo, precisa ser escrito em JavaScript, que é uma linguagem universalmente suportada pelos navegadores (na verdade, algumas outras linguagens podem ser usadas porque fornecem conversores para criar JavaScript). Assim, para o navegador web—às vezes chamado de *lado do cliente* ou *front end* do aplicativo web—o desenvolvedor terá de usar os paradigmas permitidos em JavaScript. O lado do servidor ou back end do aplicativo, que lida com as solicitações do navegador, normalmente é programado em uma linguagem diferente; o PHP é mais popular para essa finalidade.

Independentemente do paradigma, toda linguagem possui *bibliotecas* de funções pré-construídas que podem ser incorporadas ao código. Funções matemáticas—como as ilustradas no código de exemplo—não precisam ser implementadas do zero, pois a linguagem já tem a função pronta para uso. O JavaScript, por exemplo, fornece o objeto `Math`, com as operações matemáticas mais comuns.

Geralmente há funções ainda mais especializadas disponibilizadas pelo fornecedor da linguagem ou por outros desenvolvedores. Essas bibliotecas de recursos extras podem estar na forma de código-fonte, ou seja, em arquivos extras que são incorporados ao arquivo em que serão usados. Em JavaScript, a incorporação é feita com `import from`:

```
import { OrbitControls } from 'modules/OrbitControls.js';
```

Esse tipo de importação, em que o recurso incorporado também é um arquivo de código-fonte, é mais frequentemente usado nas chamadas *linguagens interpretadas*. As *linguagens compiladas* permitem, entre outras coisas, a incorporação de funcionalidades pré-compiladas em linguagem de máquina, ou seja, bibliotecas *compiladas*. A próxima seção explica as diferenças entre esses tipos de linguagem.

Compiladores e interpretadores

Como já sabemos, o código-fonte é uma representação simbólica de um programa que precisa ser traduzido em linguagem de máquina para ser executado.

Grosso modo, existem duas maneiras possíveis de se fazer a tradução: converter com antecedência o código-fonte para execução futura ou converter o código no momento de sua execução. As linguagens da primeira modalidade são chamadas de *linguagens compiladas* e as da segunda, *linguagens interpretadas*. Algumas linguagens interpretadas oferecem a compilação como opção para que o programa possa iniciar mais rápido.

Nas linguagens compiladas, existe uma distinção clara entre o código-fonte do programa e o programa em si, que será executado pelo computador. Depois de compilado, o programa normalmente rodará apenas no sistema operacional e na plataforma para os quais foi compilado.

Em uma linguagem interpretada, o próprio código-fonte é tratado como o programa, e o processo de conversão para linguagem de máquina é transparente para o programador. Nas linguagens interpretadas, é comum chamar o código-fonte de *script*. O interpretador traduz o script para a linguagem de máquina do sistema em que ele está sendo executado.

Compilação e compiladores

A linguagem de programação C é um dos exemplos mais conhecidos de linguagem compilada. Os principais pontos fortes da linguagem C são sua flexibilidade e desempenho. Tanto supercomputadores e quando os microcontroladores de alto desempenho de eletrodomésticos podem ser programados na linguagem C. Outros exemplos de linguagens compiladas populares são C++ e C# (C sharp). Como seus nomes sugerem, essas linguagens são inspiradas em C, mas incluem recursos que suportam o paradigma orientado a objetos.

O mesmo programa escrito em C ou C++ pode ser compilado para diferentes plataformas, exigindo pouca ou nenhuma alteração no código-fonte. É o compilador que define a plataforma de destino do programa. Existem compiladores específicos a plataformas, bem como compiladores que atendem a diversas plataformas, como o GCC (que significa *GNU Compiler Collection*) , capazes de produzir programas binários para muitas arquiteturas distintas.

NOTE

Existem também ferramentas que automatizam o processo de compilação. Em vez de invocar o compilador diretamente, o programador cria um arquivo indicando as diferentes etapas de compilação a serem executadas automaticamente. A ferramenta tradicional usada para esse propósito é `make`, mas certas ferramentas mais recentes, como o Maven e o Gradle, também são bastante usadas. Todo o processo de build (a compilação final ou geração do binário) é automatizado quando se usa um IDE.

O processo de compilação nem sempre gera um programa binário em linguagem de máquina. Existem linguagens compiladas que produzem um programa em um formato genericamente denominado *bytecode*. Como um script, o bytecode não está em uma linguagem específica à plataforma e, por isso, requer um programa interpretador, que o traduz para a linguagem de máquina. Neste caso, o programa interpretador é simplesmente chamado de *runtime*.

A linguagem Java adota essa abordagem; assim, os programas compilados escritos em Java podem ser usados em diferentes sistemas operacionais. Apesar do nome, o Java não tem relação com o JavaScript.

O bytecode está mais próximo da linguagem de máquina do que do código-fonte, de forma que sua execução tende a ser comparativamente mais rápida. Como ainda há um processo de conversão durante a execução do bytecode, é difícil obter o mesmo desempenho de um programa equivalente compilado em linguagem de máquina.

Interpretação e interpretadores

Nas linguagens interpretadas como JavaScript, Python e PHP, o programa não precisa ser pré-compilado, o que facilita seu desenvolvimento e modificação. Em vez de compilá-lo, o script é executado por outro programa chamado interpretador. Normalmente, o interpretador de uma linguagem recebe o nome da própria linguagem. O interpretador de um script Python, por exemplo, é um programa chamado `python`. O interpretador de JavaScript é geralmente o navegador web, mas os scripts também podem ser executados pelo programa `node` fora de um navegador. Por ser convertido em instruções binárias toda vez que é executado, um programa em linguagem interpretada tende a ser mais lento do que um equivalente em linguagem compilada.

Nada impede que o mesmo aplicativo tenha componentes escritos em linguagens diferentes. Se necessário, esses componentes podem se comunicar por meio de uma *interface de programação de aplicativos* (API) mutuamente compreensível.

A linguagem Python, por exemplo, possui recursos muito sofisticados de mineração e tabulação de dados. O desenvolvedor pode escolher o Python para escrever as partes do programa que lidam com esses aspectos e outra linguagem, como o C++, para realizar o processamento numérico mais pesado.

É possível adotar essa estratégia mesmo quando não existe uma API que permita a comunicação direta entre os dois componentes. O código escrito em Python pode gerar um arquivo no formato adequado para ser usado por um programa escrito em C++, por exemplo.

Embora seja possível escrever quase qualquer programa em qualquer linguagem, o desenvolvedor deve adotar aquela que estiver mais de acordo com o propósito do aplicativo. Ao fazer isso, você se beneficia da reutilização de componentes já testados e bem documentados.

Exercícios Guiados

1. Que tipo de programa pode ser usado para editar um código-fonte?

2. Que tipo de ferramenta ajuda a integrar o trabalho de diferentes desenvolvedores na mesma base de código?

Exercícios Exploratórios

1. Suponha que você queira escrever um jogo em 3D para ser jogado no navegador. Os apps e games para web são programados em JavaScript. Embora seja possível escrever todas as funções gráficas do zero, é mais produtivo usar uma biblioteca pronta para esse fim. Quais bibliotecas de terceiros fornecem recursos para a animação 3D em JavaScript?

2. Além do PHP, quais outras linguagens podem ser usadas no lado do servidor de um aplicativo web?

Resumo

Esta lição trata dos conceitos mais essenciais de desenvolvimento de software. O desenvolvedor deve conhecer as linguagens de programação mais importantes e o contexto de uso adequado para cada uma delas. Esta lição aborda os seguintes conceitos e procedimentos:

- O que é código-fonte.
- Editores de código-fonte e ferramentas relacionadas.
- Paradigmas de programação procedural, orientada a objetos, funcional e declarativa.
- Características das linguagens compiladas e interpretadas.

Respostas aos Exercícios Guiados

1. Que tipo de programa pode ser usado para editar um código-fonte?

   Em princípio, qualquer programa capaz de editar texto simples.

2. Que tipo de ferramenta ajuda a integrar o trabalho de diferentes desenvolvedores na mesma base de código?

   Um sistema de controle de origem ou versão, como o Git.

Respostas aos Exercícios Exploratórios

1. Suponha que você queira escrever um jogo em 3D para ser jogado no navegador. Os apps e games para web são programados em JavaScript. Embora seja possível escrever todas as funções gráficas do zero, é mais produtivo usar uma biblioteca pronta para esse fim. Quais bibliotecas de terceiros fornecem recursos para a animação 3D em JavaScript?

   Existem muitas opções de bibliotecas de gráficos 3D para JavaScript, como threejs e BabylonJS.

2. Além do PHP, quais outras linguagens podem ser usadas no lado do servidor de um aplicativo web?

   Qualquer linguagem suportada pelo aplicativo de servidor HTTP usado no host do servidor. Alguns exemplos são Python, Ruby, Perl e o próprio JavaScript.

031.2 Arquitetura de aplicativos web

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 031.2

Peso

2

Áreas chave de conhecimento

- Entender os princípios da computação cliente/servidor
- Entender o papel dos navegadores web e conhecer os navegadores mais usados
- Entender o papel dos servidores web e dos servidores de aplicativos
- Entender as tecnologias e padrões comuns de desenvolvimento web
- Entender os princípios das APIs
- Entender o princípio dos bancos de dados relacionais e não-relacionais (NoSQL)
- Conhecimento dos sistemas de código aberto mais usados para gerenciamento de bancos de dados
- Noções de REST e GraphQL
- Noções de aplicativos de página única
- Noções de empacotamento de aplicativos web
- Noções de WebAssembly
- Conhecimentos sobre sistemas de gerenciamento de conteúdo

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- Chrome, Edge, Firefox, Safari, Internet Explorer
- HTML, CSS, JavaScript
- SQLite, MySQL, MariaDB, PostgreSQL
- MongoDB, CouchDB, Redis

031.2 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	031 Desenvolvimento de software e tecnologias web
Objetivo:	031.2 Arquitetura de aplicativos web
Lição:	1 de 1

Introdução

A palavra *aplicativo* tem um amplo significado no jargão tecnológico. Quando o aplicativo é um programa tradicional, executado localmente e auto-suficiente em sua finalidade, tanto a interface operacional do aplicativo quanto os componentes de processamento de dados são integrados em um único "pacote". Um *aplicativo web* é diferente porque adota o modelo cliente/servidor e sua parte cliente é baseada em HTML, que é obtido do servidor e, em geral, processado por um navegador.

Clientes e servidores

No modelo cliente/servidor, parte do trabalho é feito localmente no *lado do cliente* e parte do trabalho é feito remotamente, no *lado do servidor*. As tarefas realizadas por cada parte variam de acordo com a finalidade do aplicativo, mas em geral cabe ao cliente fornecer uma interface para o usuário e exibir o conteúdo de forma atraente. Cabe ao servidor executar a parte operacional do aplicativo,

processando e respondendo às solicitações feitas pelo cliente. Em um aplicativo de compras, por exemplo, o aplicativo cliente apresenta uma interface para o usuário escolher e pagar pelos produtos, mas a fonte de dados e os registros da transação são mantidos no servidor remoto, acessado pela rede. Os aplicativos web realizam essa comunicação pela internet, geralmente por meio do Protocolo de Transferência de Hipertexto (HTTP).

Depois de carregado pelo navegador, o lado do cliente do aplicativo inicia a interação com o servidor sempre que necessário ou conveniente. Os servidores de aplicativos web oferecem uma *interface de programação de aplicativos* (API) que define as solicitações disponíveis e como devem ser feitas. Assim, o cliente constrói uma solicitação no formato definido pela API e a envia ao servidor, que verifica os pré-requisitos da solicitação e envia de volta a resposta apropriada.

Enquanto o cliente, na forma de um aplicativo móvel ou navegador de desktop, é um programa independente em relação à interface do usuário e às instruções para se comunicar com o servidor, o navegador deve obter a página HTML e os componentes associados — como imagens, CSS e JavaScript — que definem a interface e as instruções para comunicação com o servidor.

As linguagens de programação e as plataformas usadas por cliente e servidor são independentes, mas empregam um protocolo de comunicação mutuamente compreensível. A parte do servidor é quase sempre realizada por um programa sem interface gráfica, rodando em ambientes de computação de alta disponibilidade, de forma a estar sempre pronto para responder às solicitações. Já a parte do cliente é executada em qualquer dispositivo capaz de renderizar uma interface HTML, como os smartphones.

Além de ser imprescindível para determinadas finalidades, a adoção do modelo cliente/servidor permite que um aplicativo otimize diversos aspectos de desenvolvimento e manutenção, já que cada parte pode ser projetada para sua função específica. Um aplicativo que exibe mapas e rotas, por exemplo, não precisa ter todos os mapas armazenados localmente. São necessários apenas os mapas relativos à localização que interessa ao usuário e, portanto, somente esses mapas são solicitados ao servidor central.

Os desenvolvedores têm controle direto sobre o servidor; assim, eles também podem modificar o cliente fornecido por ele. Isso permite que os desenvolvedores aprimorem o aplicativo, em maior ou menor grau, sem que o usuário precise formalmente instalar novas versões.

O lado do cliente

Um aplicativo web deve ser executado da mesma maneira em todos os navegadores mais populares, desde que o navegador esteja atualizado. Alguns navegadores podem ser incompatíveis com as inovações recentes, mas apenas os aplicativos experimentais usam recursos ainda não amplamente adotados.

Os problemas de incompatibilidade eram mais comuns no passado, quando cada navegador tinha seu próprio *motor de renderização* e havia menos cooperação na formulação e adoção de padrões. O motor (ou mecanismo) de renderização é o principal componente do navegador, pois é responsável por transformar o HTML e outros componentes associados nos elementos visuais e interativos da interface. Alguns navegadores, sobretudo o Internet Explorer, necessitavam de um tratamento especial no código para não quebrar o funcionamento esperado das páginas.

Hoje, as diferenças entre os navegadores principais são mínimas e as incompatibilidades, raras. Na verdade, os navegadores Chrome e Edge usam o mesmo mecanismo de renderização (chamado Blink). O navegador Safari e outros oferecidos na iOS App Store usam o mecanismo WebKit. O Firefox usa um mecanismo chamado Gecko. Esses três mecanismos são responsáveis por praticamente todos os navegadores usados hoje. Embora desenvolvidos separadamente, os três motores são projetos de código aberto e há cooperação entre seus desenvolvedores, o que facilita a compatibilidade, a manutenção e a adoção de padrões.

Como os desenvolvedores de navegadores se esforçaram muito para preservar a compatibilidade, o servidor normalmente não está vinculado a um único tipo de cliente. Em princípio, um servidor HTTP pode se comunicar com qualquer cliente que também seja capaz de se comunicar via HTTP. Em um aplicativo de mapa, por exemplo, o cliente pode ser um aplicativo móvel ou um navegador que carrega a interface HTML do servidor.

Variedades de clientes web

Existem aplicativos móveis e de desktop cuja interface é renderizada a partir de HTML e, como os navegadores, podem usar JavaScript como linguagem de programação. Porém, ao contrário do cliente carregado no navegador, o HTML e os componentes necessários para o funcionamento de um cliente nativo estão presentes localmente desde a instalação do aplicativo. Na verdade, um aplicativo que funciona dessa maneira é praticamente idêntico a uma página HTML (é provável que ambos sejam renderizados pelo mesmo mecanismo). Existem também os *aplicativos web progressivos* (Progressive Web Apps ou PWA), um mecanismo que permite empacotar clientes de aplicativos web para uso offline—limitado a funções que não requerem comunicação imediata com o servidor. Em relação às capabilidades do aplicativo, não há diferença entre rodá-lo no navegador ou empacotado em um PWA; porém, neste último, o desenvolvedor tem mais controle sobre o que é armazenado localmente.

Renderizar interfaces HTML é uma atividade tão recorrente que o mecanismo é geralmente um componente de software separado, presente no sistema operacional. Sua presença independente permite que diferentes aplicativos o incorporem sem precisar incluí-lo no pacote do aplicativo. Esse modelo também delega a manutenção do motor de renderização ao sistema operacional, facilitando as atualizações. É muito importante manter esse componente crucial sempre atualizado para evitar possíveis falhas.

Independentemente do método de fornecimento, os aplicativos escritos em HTML são executados em uma camada de abstração criada pelo mecanismo, que funciona como um ambiente de execução isolado. Em particular, no caso de um cliente que roda no navegador, o aplicativo tem à sua disposição apenas os recursos oferecidos pelo navegador. Recursos básicos, como a interação com elementos de página e solicitação de arquivos por HTTP, estão sempre disponíveis. Os recursos que podem conter informações confidenciais, como o acesso a arquivos locais, a localização geográfica, câmera e microfone, requerem uma autorização explícita do usuário antes que o aplicativo possa usá-los.

As linguagens de um cliente web

O elemento central de um cliente de aplicativo web executado no servidor é o documento HTML. Além de apresentar de forma estruturada os elementos da interface exibidos pelo navegador, o documento HTML contém os endereços de todos os arquivos necessários para a apresentação e funcionamento corretos do cliente.

O HTML sozinho não tem muita versatilidade para construir interfaces mais elaboradas, nem recursos de programação de finalidade geral. Por esse motivo, um documento HTML que deve funcionar como um aplicativo cliente vem sempre acompanhado por um ou mais conjuntos de CSS e JavaScript.

O CSS pode ser fornecido como um arquivo separado ou diretamente no próprio arquivo HTML. O principal objetivo do CSS é refinar a aparência e o layout dos elementos da interface HTML. Embora isso não seja estritamente necessário, as interfaces mais sofisticadas geralmente requerem modificações nas propriedades CSS dos elementos para atender às suas necessidades.

O JavaScript é um componente praticamente indispensável. Os procedimentos escritos em JavaScript respondem a eventos no navegador. Esses eventos podem ser causados pelo usuário ou ser não-interativos. Sem o JavaScript, um documento HTML fica praticamente limitado a texto e imagens. O uso do JavaScript em documentos HTML permite estender a interatividade muito além de hiperlinks e formulários, transformando a página exibida pelo navegador em uma interface de aplicativo convencional.

O JavaScript é uma linguagem de programação de propósito geral, mas seu principal uso é em aplicativos web. Os recursos do ambiente de execução do navegador são acessíveis por meio de palavras-chave em JavaScript, utilizadas em um script para realizar a operação desejada. O termo `document`, por exemplo, é usado no código JavaScript para se referir ao documento HTML associado a ele. No contexto da linguagem JavaScript, `document` é um *objeto global* com propriedades e métodos que podem ser usados para obter informações de qualquer elemento no documento HTML. Além disso, podemos usar o objeto `document` para modificar seus elementos e associá-los a ações personalizadas escritas em JavaScript.

Um aplicativo cliente baseado em tecnologias web é multiplataforma, pois pode ser executado em qualquer dispositivo que possua um navegador compatível.

Porém, o fato de estarem confinados ao navegador impõe limitações aos aplicativos web em comparação com os aplicativos nativos. A intermediação realizada pelo navegador permite uma programação de alto nível e aumenta a segurança, mas também aumenta as exigências de processamento e o consumo de memória.

Os desenvolvedores estão continuamente aprimorando os navegadores para fornecer mais recursos e melhorar o desempenho dos aplicativos em JavaScript, mas existem aspectos intrínsecos à execução de scripts como o JavaScript que os deixam em desvantagem na comparação com programas nativos para o mesmo hardware.

Um recurso que melhora bastante o desempenho dos aplicativos JavaScript em execução no navegador é o *WebAssembly*. O WebAssembly é um tipo de JavaScript compilado que produz código-fonte escrito em uma linguagem mais eficiente de nível inferior, como a linguagem C. O WebAssembly pode acelerar principalmente as atividades de uso intensivo do processador, pois evita grande parte da tradução realizada pelo navegador ao executar um programa escrito em JavaScript convencional.

Independentemente dos detalhes de implementação do aplicativo, todos os códigos HTML, CSS, JavaScript e arquivos multimídia devem primeiro ser obtidos do servidor. O navegador obtém esses arquivos como se fosse uma página da internet, ou seja, por meio de um endereço acessado pelo navegador.

Uma página web que atua como uma interface para um aplicativo web é como um documento HTML simples, mas com comportamentos adicionais. Nas páginas convencionais, o usuário é direcionado para outra página ao clicar em um link. Os aplicativos web podem apresentar sua interface e responder aos eventos do usuário sem carregar novas páginas na janela do navegador. A modificação desse comportamento padrão das páginas HTML é feita por meio da programação em JavaScript.

Um cliente de webmail, por exemplo, exibe as mensagens e alterna entre as pastas sem sair da página. Isso é possível porque o cliente usa JavaScript para reagir às ações do usuário e fazer solicitações apropriadas ao servidor. Se o usuário clicar no assunto de uma mensagem na caixa de entrada, um código JavaScript associado a esse evento solicitará o conteúdo dessa mensagem ao servidor (usando a chamada API correspondente). Assim que o cliente recebe a resposta, o navegador exibe a mensagem na parte apropriada da mesma página. Clientes de webmail diferentes podem adotar estratégias diferentes, mas todos empregam o mesmo princípio.

Portanto, além de fornecer ao navegador os arquivos que compõem o cliente, o servidor também deve ser capaz de atender a solicitações como a do cliente de webmail quando solicita o conteúdo de

uma determinada mensagem. Cada requisição que o cliente pode fazer está vinculada a um procedimento predefinido de resposta no servidor, cuja API pode definir diferentes métodos para identificar o procedimento ao qual a requisição se refere. Os métodos mais comuns são:

- Endereços, através de um Uniform Resource Locator (URL)
- Campos no cabeçalho HTTP
- Métodos GET/POST
- WebSockets

Um método pode ser mais adequado do que outro, dependendo da finalidade da solicitação e de outros critérios levados em consideração pelo desenvolvedor. Em geral, os aplicativos web usam uma combinação de métodos, cada um em uma circunstância específica.

O paradigma *Representational State Transfer* (REST) é amplamente utilizado para a comunicação nos aplicativos web, pois se baseia nos métodos básicos disponíveis em HTTP. O cabeçalho de uma solicitação HTTP começa com uma palavra-chave que define a operação básica a ser realizada: `GET`, `POST`, `PUT`, `DELETE`, etc., acompanhada por uma URL correspondente na qual a ação será aplicada. Se o aplicativo requer operações mais específicas, com uma descrição mais detalhada da operação solicitada, o protocolo GraphQL pode ser uma escolha mais adequada.

Os aplicativos desenvolvidos no modelo cliente/servidor estão sujeitos a instabilidades de comunicação. Por isso, o aplicativo cliente deve sempre adotar estratégias eficientes de transferência de dados para favorecer sua consistência e não prejudicar a experiência do usuário.

O lado do servidor

Apesar de ser o ator principal em um aplicativo web, o servidor é o lado passivo da comunicação, respondendo apenas às solicitações feitas pelo cliente. No jargão da web, *servidor* pode se referir à máquina que recebe as solicitações, ao programa que trata especificamente as solicitações HTTP ou ao script destinatário que produz uma resposta à solicitação. Esta última definição é a mais relevante no contexto da arquitetura de aplicativos web, mas todas estão intimamente relacionadas. Embora pertençam apenas parcialmente ao escopo do desenvolvedor do servidor de aplicativos, a máquina, o sistema operacional e o servidor HTTP não podem ser ignorados, pois são fundamentais para a execução do servidor de aplicativos e freqüentemente se cruzam.

Controlando os caminhos de solicitações

Os servidores HTTP, como o Apache e o NGINX, costumam precisar de alterações específicas de configuração para atender às necessidades do aplicativo. Por padrão, os servidores HTTP tradicionais associam diretamente o caminho indicado na solicitação a um arquivo no sistema de arquivos local.

Se o servidor HTTP de um website mantiver seus arquivos HTML no diretório `/srv/www`, por exemplo, uma solicitação com o caminho `/en/about.html` receberá o conteúdo do arquivo `/srv/www/en/about.html` como resposta, se o arquivo existir. Os sites mais sofisticados, e os aplicativos web em especial, exigem tratamentos personalizados para diferentes tipos de solicitações. Nesse cenário, parte da implementação do aplicativo consiste em modificar as configurações do servidor HTTP para atender aos requisitos do aplicativo.

Como alternativa, existem frameworks (estruturas) que permitem integrar o gerenciamento das solicitações HTTP e a implementação do código do aplicativo em um só lugar, permitindo que o desenvolvedor se concentre mais na finalidade do aplicativo do que nos detalhes da plataforma. No Node.js Express, por exemplo, todo o mapeamento de solicitações e a programação correspondente são implementados usando JavaScript. Como a programação dos clientes geralmente é feita em JavaScript, muitos desenvolvedores consideram uma boa ideia, do ponto de vista da manutenção do código, usar a mesma linguagem para o cliente e o servidor. Outras linguagens comumente usadas para implementar o lado do servidor, seja em frameworks ou em servidores HTTP tradicionais, são PHP, Python, Ruby, Java e C#.

Sistemas de gerenciamento de banco de dados

Fica a critério da equipe de desenvolvimento a forma como os dados recebidos ou solicitados pelo cliente são armazenados no servidor, mas existem diretrizes gerais que se aplicam à maioria dos casos. É conveniente manter o conteúdo estático — imagens, código JavaScript e CSS que não mudam no curto prazo — em arquivos convencionais, seja no próprio sistema de arquivos do servidor ou distribuídos em uma *rede de fornecimento de conteúdo* (CDN). Outros tipos de conteúdo, como mensagens de email em um aplicativo de webmail, detalhes do produto em um aplicativo de compras e logs de transações, são armazenados de forma mais conveniente em um *sistema de gerenciamento de banco de dados* (DBMS).

O tipo mais tradicional de sistema de gerenciamento de banco de dados é o *banco de dados relacional*. Nele, o criador do aplicativo define tabelas de dados e o formato de entrada aceito por cada tabela. O conjunto de tabelas do banco de dados contém todos os dados dinâmicos consumidos e produzidos pelo aplicativo. Um aplicativo de compras, por exemplo, pode ter uma tabela com os detalhes de cada produto da loja e uma tabela que registra os itens comprados por um usuário. A tabela de itens comprados contém referências às entradas na tabela de produtos, criando relações entre as tabelas. Essa abordagem ajuda a otimizar o armazenamento e o acesso aos dados, além de permitir consultas em tabelas combinadas utilizando a linguagem adotada pelo sistema de gerenciamento do banco de dados. A linguagem de banco de dados relacional mais popular é a *Structured Query Language* (SQL, pronuncia-se “sequel”), adotada pelos bancos de dados de código aberto SQLite, MySQL, MariaDB e PostgreSQL.

Uma alternativa aos bancos de dados relacionais é uma forma de banco de dados que não requer uma

estrutura rígida para os dados. Esses bancos de dados são chamados de *bancos de dados não-relacionais* ou simplesmente *NoSQL*. Embora possam incorporar alguns recursos semelhantes aos encontrados nos bancos de dados relacionais, o foco está em permitir maior flexibilidade no armazenamento e acesso aos dados armazenados, entregando a tarefa de processamento desses dados para o próprio aplicativo. MongoDB, CouchDB e Redis são sistemas comuns de gerenciamento de bancos de dados não-relacionais.

Manutenção de conteúdo

Qualquer que seja o modelo de banco de dados adotado, os aplicativos precisam adicionar dados e provavelmente atualizá-los ao longo da vida útil dos aplicativos. Em alguns aplicativos, como o webmail, os próprios usuários fornecem dados ao banco de dados ao usar o cliente para enviar e receber mensagens. Em outros casos, como em um aplicativo de compras, é importante permitir que os mantenedores do aplicativo modifiquem o banco de dados sem ter de recorrer à programação. Muitas empresas, portanto, adotam algum tipo de *sistema de gerenciamento de conteúdo* (CMS), que permite que usuários não-técnicos administrem o aplicativo. Portanto, para a maioria dos aplicativos web, é necessário implementar pelo menos dois tipos de clientes: o cliente não-privilegiado, empregado por usuários comuns, e o cliente privilegiado, empregado por usuários especiais para manter e atualizar as informações apresentadas pelo aplicativo.

Exercícios Guiados

1. Qual linguagem de programação é usada junto com o HTML para criar clientes de aplicativos web?

2. Como a obtenção de um aplicativo web difere daquela de um aplicativo nativo?

3. Como um aplicativo web difere de um aplicativo nativo no acesso ao hardware local?

4. Cite uma característica de um cliente de aplicativo web que o diferencia de uma página web comum.

Exercícios Exploratórios

1. Qual recurso os navegadores modernos oferecem para mitigar o baixo desempenho dos clientes de aplicativos web que usam muita CPU?

2. Se um aplicativo web usa o paradigma REST para a comunicação cliente/servidor, qual método HTTP deve ser usado quando o cliente solicita que o servidor apague um recurso específico?

3. Cite cinco linguagens de script de servidor suportadas pelo servidor Apache HTTP.

4. Por que os bancos de dados não relacionais são considerados mais fáceis de manter e atualizar do que os bancos de dados relacionais?

Resumo

Esta lição trata dos conceitos e padrões em tecnologia e arquitetura de desenvolvimento web. O princípio é simples: o navegador executa o aplicativo cliente, que se comunica com o aplicativo principal que está em execução no servidor. Embora simples em princípio, os aplicativos web precisam combinar diversas tecnologias para adotar o princípio da computação de cliente e servidor pela web. A lição abrange os seguintes conceitos:

- A função dos navegadores e servidores web.
- Tecnologias e padrões comuns de desenvolvimento para a web.
- Como os clientes web podem se comunicar com o servidor.
- Tipos de servidores web e bancos de dados de servidores.

Respostas aos Exercícios Guiados

1. Qual linguagem de programação é usada junto com o HTML para criar clientes de aplicativos web?

   JavaScript

2. Como a obtenção de um aplicativo web difere daquela de um aplicativo nativo?

   Um aplicativo web não está instalado. Em vez disso, partes dele são executadas no servidor e a interface do cliente é executada em um navegador web comum.

3. Como um aplicativo web difere de um aplicativo nativo no acesso ao hardware local?

   Todos os acessos aos recursos locais, como armazenamento, câmeras ou microfones, são mediados pelo navegador e requerem autorização explícita do usuário para funcionar.

4. Cite uma característica de um cliente de aplicativo web que o diferencia de uma página web comum.

   A interação com as páginas web tradicionais restringe-se basicamente a hiperlinks e envio de formulários, ao passo que os clientes de aplicativos web estão mais próximos de uma interface de aplicativo convencional.

Respostas aos Exercícios Exploratórios

1. Qual recurso os navegadores modernos oferecem para mitigar o baixo desempenho dos clientes de aplicativos web que usam muita CPU?

   Os desenvolvedores podem usar o WebAssembly para implementar as partes do aplicativo cliente que exigem um uso intensivo da CPU. O código do WebAssembly costuma ter melhor desempenho do que o JavaScript tradicional, pois requer menos tradução de instruções.

2. Se um aplicativo web usa o paradigma REST para a comunicação cliente/servidor, qual método HTTP deve ser usado quando o cliente solicita que o servidor apague um recurso específico?

   REST relies on standard HTTP methods, so it should use the standard `DELETE` method in this case.

3. Cite cinco linguagens de script de servidor suportadas pelo servidor Apache HTTP.

   PHP, Go, Perl, Python e Ruby.

4. Por que os bancos de dados não relacionais são considerados mais fáceis de manter e atualizar do que os bancos de dados relacionais?

   Ao contrário dos bancos de dados relacionais, os bancos de dados não relacionais não requerem que os dados se adaptem a estruturas predefinidas rígidas, sendo assim mais fácil implementar mudanças nas estruturas de dados sem afetar os dados existentes.

031.3 Noções básicas de HTTP

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 031.3

Peso

3

Áreas chave de conhecimento

- Entender os métodos HTTP GET e POST, códigos de status, cabeçalhos e tipos de conteúdo
- Entender a diferença entre conteúdo estático e dinâmico
- Entender URLs HTTP
- Entender como as URLs HTTP são mapeadas para os caminhos do sistema de arquivos
- Fazer upload de arquivos para a raiz de documentos de um servidor web
- Entender o cache
- Entender os cookies
- Conhecimentos sobre sessões e sequestro de sessões
- Conhecimento dos servidores HTTP mais usados
- Noções de HTTPS e TLS
- Noções de WebSockets
- Noções de hosts virtuais
- Conhecimento dos servidores HTTP mais comuns
- Conhecimentos sobre os requisitos e limitações de largura de banda e latência da rede

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- GET, POST
- 200, 301, 302, 401, 403, 404, 500
- Apache HTTP Server (httpd), NGINX

031.3 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	031 Desenvolvimento de software e tecnologias web
Objetivo:	031.3 Noções básicas de HTTP
Lição:	1 de 1

Introdução

O protocolo de transferência de hipertexto (HyperText Transfer Protocol ou HTTP) define a forma como um cliente solicita um recurso específico ao servidor. O princípio de funcionamento é bastante simples: o cliente cria uma mensagem de solicitação identificando o recurso de que necessita e encaminha essa mensagem para o servidor através da rede. Por sua vez, o servidor HTTP avalia de onde extrair o recurso solicitado e envia uma mensagem de resposta de volta ao cliente. A mensagem de resposta contém detalhes sobre o recurso solicitado, seguidos do recurso em si.

Mais especificamente, HTTP é o conjunto de regras que definem como o aplicativo cliente deve formatar as mensagens de *solicitação* que serão enviadas ao servidor. O servidor então segue as regras do HTTP para interpretar a solicitação e formatar mensagens de *resposta*. Além de solicitar ou transferir o conteúdo solicitado, as mensagens HTTP contêm informações extras sobre o cliente e o servidor envolvidos, sobre o conteúdo em si e até mesmo sobre sua indisponibilidade. Se um recurso não puder ser enviado, um código na resposta explica o motivo da indisponibilidade e, se possível, indica para onde o recurso foi movido.

A parte da mensagem que define os detalhes do recurso e outras informações de contexto é chamada

de *cabeçalho* da mensagem. A parte após o cabeçalho, que contém o conteúdo do recurso correspondente, é chamada de *corpo de dados* (ou carga) da mensagem. Tanto as mensagens de solicitação quanto as mensagens de resposta podem ter um corpo de dados, mas na maioria dos casos ele está presente apenas na mensagem de resposta.

A solicitação do cliente

A primeira etapa de uma troca de dados HTTP entre o cliente e o servidor é iniciada pelo cliente, quando ele escreve uma mensagem de solicitação ao servidor. Vejamos, por exemplo, uma tarefa comum de um navegador: carregar uma página HTML de um servidor que hospeda um site, como `https://learning.lpi.org/pt/`. O endereço, ou URL, fornece diversas informações relevantes. Três informações aparecem neste exemplo específico:

- O protocolo: HyperText Transfer Protocol Secure (`https`), uma versão criptografada do HTTP.
- O nome da rede do host (`learning.lpi.org`)
- A localização do recurso solicitado no servidor (o diretório `/pt/`-- neste caso, a versão em português da página inicial).

NOTE

A URL (*Uniform Resource Locator*) é um endereço que aponta para um recurso na internet. Esse recurso geralmente é um arquivo que pode ser copiado de um servidor remoto, mas as URLs também podem indicar conteúdos gerados dinamicamente e fluxos de dados.

Como o cliente lida com a URL

Antes de contatar o servidor, o cliente precisa converter `learning.lpi.org_` para o endereço IP correspondente. O cliente usa outro serviço de Internet, o *Sistema de Nomes de Domínio* (Domain Name System ou DNS), para solicitar o endereço IP de um nome de host a um ou mais servidores DNS predefinidos (em geral, os servidores DNS são definidos automaticamente pelo provedor).

Com o endereço IP do servidor, o cliente tenta se conectar à porta HTTP ou HTTPS. As portas de rede são números de identificação definidos pelo *Protocolo de Controle de Transmissão* (Transmission Control Protocol ou TCP) para entrelaçar e identificar canais de comunicação distintos em uma conexão cliente/servidor. Por padrão, os servidores HTTP recebem solicitações nas portas TCP 80 (HTTP) e 443 (HTTPS).

NOTE

Existem outros protocolos usados por aplicativos web para implementar a comunicação cliente/servidor. Para chamadas de áudio e vídeo, por exemplo, é mais apropriado usar WebSockets, um protocolo de nível inferior que é mais eficiente do que o HTTP para transferir fluxos de dados em ambas as direções.

O formato da mensagem de solicitação que o cliente envia ao servidor é o mesmo no HTTP e no HTTPS. O HTTPS já é mais utilizado do que o HTTP, pois nele todas as trocas de dados entre cliente e servidor são criptografadas, um recurso indispensável para garantir privacidade e segurança em redes públicas. A conexão criptografada é estabelecida entre o cliente e o servidor antes mesmo de qualquer mensagem HTTP ser trocada, usando o protocolo criptográfico *Transport Layer Security* (TLS). Dessa forma, toda a comunicação HTTPS é encapsulada pela TLS. Depois de descriptografada, a solicitação ou resposta transmitida por HTTPS não é diferente de uma solicitação ou resposta feita exclusivamente por HTTP.

O terceiro elemento da nossa URL, `/pt/`, será interpretado pelo servidor como a localização ou o caminho para o recurso que está sendo solicitado. Se o caminho não for fornecido na URL, o local padrão `/` será usado. A implementação mais simples de um servidor HTTP associa os caminhos nas URLs a arquivos no sistema de arquivos em que o servidor está sendo executado, mas esta é apenas uma das muitas opções disponíveis em servidores HTTP mais sofisticados.

A mensagem de solicitação

O HTTP opera através de uma conexão já estabelecida entre cliente e servidor, geralmente implementada em TCP e criptografada com TLS. Na verdade, uma vez que uma conexão que atenda aos requisitos impostos pelo servidor esteja pronta, uma solicitação HTTP digitada à mão em texto simples pode gerar a resposta do servidor. Na prática, porém, os programadores raramente precisam implementar rotinas para compor mensagens HTTP, pois a maioria das linguagens de programação fornece mecanismos que automatizam a criação dessas mensagens. No caso da URL de exemplo, `https://learning.lpi.org/pt/`, a mensagem de solicitação mais simples possível teria o seguinte conteúdo:

```
GET /pt/ HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: text/html
```

A primeira palavra da primeira linha identifica o *método* HTTP. Ele define qual operação o cliente deseja executar no servidor. O método GET informa ao servidor que o cliente solicita o recurso que o segue: `/pt/`. Tanto o cliente quanto o servidor podem suportar mais de uma versão do protocolo

HTTP, portanto a versão a ser adotada na troca de dados também é fornecida na primeira linha: `HTTP/1.1`.

NOTE

A versão mais recente do protocolo HTTP é HTTP/2. Entre outras diferenças, as mensagens escritas em HTTP/2 são codificadas em uma estrutura binária, ao passo que as mensagens escritas em HTTP/1.1 são enviadas em texto simples. Essa mudança otimiza as taxas de transmissão de dados, mas o conteúdo das mensagens permanece basicamente o mesmo.

O cabeçalho pode conter mais linhas após a primeira, para contextualizar e ajudar a identificar a solicitação ao servidor. O campo de cabeçalho `Host`, por exemplo, pode parecer redundante, porque o host do servidor foi obviamente identificado pelo cliente para estabelecer a conexão e é razoável supor que o servidor conheça a sua própria identidade. No entanto, é importante informar ao host o nome de host esperado no cabeçalho da solicitação, pois é prática comum usar o mesmo servidor HTTP para hospedar mais de um site (nesse cenário, cada host específico é chamado de *host virtual*). Portanto, o campo `Host` é usado pelo servidor HTTP para identificar a qual deles a solicitação se refere.

O campo de cabeçalho `User-Agent` contém detalhes sobre o programa cliente que está fazendo a solicitação. Este campo pode ser usado pelo servidor para adaptar a resposta às necessidades de um cliente específico, mas é mais frequentemente empregado para produzir estatísticas sobre os clientes que usam o servidor.

O campo `Accept` tem um valor mais imediato, pois informa ao servidor o formato do recurso solicitado. Se o formato do recurso for indiferente para o cliente, o campo `Accept` pode especificar `*/*` como formato.

Existem muitos outros campos de cabeçalho que podem ser usados em uma mensagem HTTP, mas os campos mostrados no exemplo já bastam para solicitar um recurso do servidor.

Além dos campos no cabeçalho da solicitação, o cliente pode incluir outros dados complementares na solicitação HTTP que será enviada ao servidor. Se esses dados consistirem apenas em parâmetros de texto simples, no formato `name=value`, eles podem ser adicionados ao caminho do método GET. Os parâmetros são incorporados ao caminho após um ponto de interrogação e são separados por "&":

```
GET /cgi-bin/receive.cgi?name=LPI&email=info@lpi.org HTTP/1.1
```

Neste exemplo, `/cgi-bin/receive.cgi` é o caminho até o script no servidor que vai processar e, possivelmente, usar os parâmetros `name` e `email` obtidos no caminho da solicitação. A string que corresponde aos campos, no formato `name=LPI&email=info@lpi.org`, é chamada *string de solicitação*

e é fornecida ao script `receive.cgi` pelo servidor HTTP que recebe a solicitação.

Quando os dados são compostos por mais do que campos curtos de texto, é mais apropriado enviá-los no corpo de dados da mensagem. Neste caso, deve-se utilizar o método HTTP POST para que o servidor receba e processe o corpo de dados da mensagem, de acordo com as especificações indicadas no cabeçalho da solicitação. Quando o método POST é usado, o cabeçalho da solicitação deve fornecer o tamanho da carga que será enviada e a maneira como o corpo é formatado:

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
Content-Length: 1503
Content-Type: multipart/form-data; boundary=------------------------
405f7edfd646a37d
```

O campo `Content-Length` indica o tamanho em bytes da carga (corpo de dados) e o campo `Content-Type` indica seu formato. O formato `multipart/form-data` é o mais comumente usado em formulários HTML tradicionais que empregam o método POST. Nesse formato, cada campo inserido no corpo de dados da solicitação é separado pelo código indicado pela palavra-chave `boundary`. O método POST deve ser usado apenas quando apropriado, pois ele usa uma quantidade de dados um pouco maior do que uma solicitação equivalente feita com o método GET. Como o método GET envia os parâmetros diretamente no cabeçalho da mensagem de solicitação, a troca de dados total tem uma latência menor, pois não é necessária uma etapa de conexão adicional para transmitir o corpo da mensagem.

O cabeçalho de resposta

Depois que o servidor HTTP recebe o cabeçalho da mensagem de solicitação, o servidor retorna uma mensagem de resposta ao cliente. Uma solicitação de arquivo HTML normalmente tem um cabeçalho de resposta semelhante a este:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 18170
Content-Type: text/html
Date: Mon, 05 Apr 2021 13:44:25 GMT
Etag: "606adcd4-46fa"
Last-Modified: Mon, 05 Apr 2021 09:48:04 GMT
Server: nginx/1.17.10
```

A primeira linha fornece a versão do protocolo HTTP usado na mensagem de resposta, que deve corresponder à versão usada no cabeçalho da solicitação. Em seguida, ainda na primeira linha,

aparece o código de status da resposta, indicando como o servidor interpretou e gerou a resposta para a solicitação.

O código de status é um número de três dígitos, no qual o dígito mais à esquerda define a classe da resposta. Existem cinco classes de códigos de status, numeradas de 1 a 5, cada uma indicando um tipo de ação realizado pelo servidor:

1xx (Informativo)

A solicitação foi recebida, o processo está sendo continuado.

2xx (Sucesso)

A solicitação foi recebida, entendida e aceita com sucesso.

3xx (Redirecionamento)

São necessárias ações adicionais para concluir a solicitação.

4xx (Erro do cliente)

A solicitação contém sintaxe incorreta ou não pode ser atendida.

5xx (Erro do servidor)

O servidor não atendeu a uma solicitação aparentemente válida.

O segundo e o terceiro dígitos são usados para indicar detalhes adicionais. O código 200, por exemplo, indica que a solicitação pode ser atendida sem problemas. Como mostrado no exemplo, também é possível fornecer uma breve descrição após o código de resposta (`OK`). Alguns códigos específicos servem para garantir que o cliente HTTP possa acessar o recurso em situações adversas ou para ajudar a identificar o motivo da falha no caso de uma solicitação malsucedida:

`301 Moved Permanently`

O recurso de destino recebeu uma nova URL permanente, fornecida pelo campo de cabeçalho `Location` na resposta.

`302 Found`

O recurso de destino reside temporariamente em uma URL diferente.

`401 Unauthorized`

A solicitação não foi aplicada porque não possui credenciais de autenticação válidas para o recurso de destino.

`403 Forbidden`

A resposta `Forbidden` indica que, embora a solicitação seja válida, o servidor está configurado para não fornecê-la.

`404 Not Found`

O servidor de origem não encontrou uma representação atual do recurso de destino ou não está disposto a divulgar uma representação existente.

`500 Internal Server Error`

O servidor encontrou uma condição inesperada que o impediu de atender à solicitação.

`502 Bad Gateway`

O servidor, ao atuar como um gateway ou proxy, recebeu uma resposta inválida de um servidor de entrada que ele acessou ao tentar atender a solicitação.

Embora indiquem que não foi possível atender à solicitação, os códigos de status `4xx` e `5xx` pelo menos informam que o servidor HTTP está rodando e é capaz de receber solicitações. Os códigos `4xx` requerem que uma ação seja realizada no lado do cliente, pois sua URL ou credenciais estão incorretos. Por sua vez, os códigos `5xx` indicam algo errado no lado do servidor. Portanto, no contexto dos aplicativos web, essas duas classes de códigos de status indicam que a origem do erro está no próprio aplicativo, seja no cliente ou no servidor, e não na infraestrutura subjacente.

Conteúdo estático e dinâmico

Os servidores HTTP usam dois mecanismos básicos para atender ao conteúdo solicitado pelo cliente. O primeiro mecanismo fornece *conteúdo estático*: ou seja, o caminho indicado na mensagem de solicitação corresponde a um arquivo no sistema de arquivos local do servidor. O segundo mecanismo fornece *conteúdo dinâmico*: ou seja, o servidor HTTP encaminha a solicitação para outro programa — normalmente um script — para construir a resposta a partir de diversas fontes, como bancos de dados e outros arquivos.

Embora existam diferentes servidores HTTP, todos eles usam o mesmo protocolo de comunicação HTTP e adotam mais ou menos as mesmas convenções. Um aplicativo que não tem uma necessidade específica pode ser implementado com qualquer servidor tradicional, como Apache ou NGINX. Ambos são capazes de gerar conteúdo dinâmico e fornecer conteúdo estático, mas existem diferenças sutis na configuração de cada um.

A localização dos arquivos estáticos a serem servidos, por exemplo, é definida de maneiras diferentes no Apache e no NGINX. A convenção é manter esses arquivos em um diretório específico para esse fim, tendo um nome associado ao host, por exemplo `/var/www/learning.lpi.org/`. No Apache, esse caminho é definido pela diretiva de configuração `DocumentRoot /var/www/learning.lpi.org`, em

uma seção que define um host virtual. No NGINX, a diretiva usada é `root /var/www/learning.lpi.org` em uma seção `server` do arquivo de configuração.

Qualquer que seja o servidor escolhido, os arquivos em `/var/www/learning.lpi.org/` serão servidos via HTTP de maneira muito parecida. Alguns campos no cabeçalho da resposta e seus conteúdos podem variar entre os dois servidores, mas campos como `Content-Type` precisam estar presentes no cabeçalho da resposta e ser consistentes entre todos os servidores.

Cache

O HTTP foi criado para funcionar em qualquer tipo de conexão à internet, seja ela rápida ou lenta. Além disso, a maioria das trocas HTTP tem de atravessar muitos nós de rede devido à arquitetura distribuída da internet. Como resultado, é importante adotar alguma estratégia de cache de conteúdo para evitar a transferência redundante de conteúdo baixado anteriormente. As transferências HTTP trabalham com dois tipos básicos de cache: *compartilhada* e *privada*.

Uma cache compartilhada é usada por mais de um cliente. Por exemplo, um grande provedor de conteúdo pode usar caches em servidores distribuídos geograficamente para que os clientes obtenham os dados do servidor mais próximo. Quando um cliente faz uma solicitação, sua resposta é armazenada em uma cache compartilhada, e outros clientes que fizerem a mesma solicitação na mesma região receberão a resposta que está na cache.

A cache privada é criada pelo próprio cliente para seu uso exclusivo. É o tipo de cache que o navegador web cria para imagens, arquivos CSS, JavaScript ou o próprio documento HTML, para que não seja necessário fazer download novamente se esses elementos forem solicitados em um futuro próximo.

NOTE

Nem todas as solicitações HTTP precisam ser armazenadas em cache. Uma solicitação usando o método POST, por exemplo, implica em uma resposta associada exclusivamente a essa solicitação específica, de modo que o conteúdo dessa resposta não será reutilizado. Por padrão, apenas as respostas às solicitações feitas usando o método GET são armazenadas em cache. Além disso, só as respostas com códigos de status conclusivos, como 200 (OK), 206 (Partial Content), 301 (Moved Permanently) e 404 (Not Found) são adequadas para armazenamento em cache.

Tanto a estratégia de cache compartilhada quanto a privada usam cabeçalhos HTTP para controlar como o conteúdo baixado deve ser armazenado em cache. No caso da cache privada, o cliente consulta o cabeçalho da resposta e verifica se o conteúdo da cache local ainda corresponde ao conteúdo remoto atual. Em caso afirmativo, o cliente dispensa a transferência da carga da resposta e usa a versão local.

A validade do recurso em cache pode ser avaliada de várias maneiras. O servidor pode fornecer uma data de expiração no cabeçalho da resposta para a primeira solicitação, para que o cliente descarte o recurso armazenado em cache no final do prazo e solicite-o novamente para obter a versão atualizada. No entanto, o servidor nem sempre consegue determinar a data de expiração de um recurso, de modo que é comum usar o campo `ETag` no cabeçalho de resposta para identificar a versão do recurso, por exemplo `Etag: "606adcd4-46fa"`.

Para verificar se um recurso armazenado em cache precisa ser atualizado, o cliente solicita apenas o cabeçalho de resposta do servidor. Se o campo `ETag` corresponder ao da versão armazenada localmente, o cliente reutiliza o conteúdo armazenado em cache. Caso contrário, o conteúdo atualizado do recurso é baixado do servidor.

Sessões HTTP

Em um site convencional ou aplicativo web, os recursos responsáveis pelo controle da sessão baseiam-se em cabeçalhos HTTP. O servidor não pode pressupor, por exemplo, que todas as solicitações provenientes do mesmo endereço IP vêm do mesmo cliente. O método mais tradicional que permite ao servidor associar diferentes solicitações a um único cliente é o uso de *cookies*, uma etiqueta de identificação fornecida ao cliente pelo servidor e incluída no cabeçalho HTTP.

Os cookies permitem que o servidor preserve informações sobre um cliente específico, mesmo que a pessoa que está executando o cliente não se identifique explicitamente. Com os cookies, é possível implementar sessões em que os logins, cartões de compras, preferências, etc., são preservados entre diferentes solicitações feitas ao mesmo servidor que os forneceu. Os cookies também são usados para rastrear a navegação do usuário, por isso é importante ter a permissão dele antes de enviá-los.

O servidor define o cookie no cabeçalho da resposta usando o campo `Set-Cookie`. O valor do campo é um par `name=value` escolhido de forma a representar algum atributo associado a um cliente específico. O servidor pode, por exemplo, criar um número de identificação para um cliente que solicita um recurso pela primeira vez e repassá-lo ao cliente no cabeçalho da resposta:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Set-Cookie: client_id=62b5b719-fcbf
```

Se o cliente permitir o uso de cookies, as novas solicitações para este mesmo servidor terão o campo do cookie no cabeçalho:

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
Cookie: client_id=62b5b719-fcbf
```

Com esse número de identificação, o servidor pode recuperar definições específicas ao cliente e gerar uma resposta personalizada. Também é possível usar mais de um campo `Set-Cookie` para entregar cookies diferentes ao mesmo cliente. Dessa forma, mais de uma definição pode ser preservada no lado do cliente.

Os cookies suscitam problemas de privacidade e potenciais falhas de segurança, já que existe a possibilidade de serem transferidos para outro cliente, que será identificado pelo servidor como sendo o cliente original. Os cookies usados para preservar sessões podem dar acesso a informações confidenciais do cliente original. Portanto, é imprescindível que os clientes adotem mecanismos de proteção local para evitar que seus cookies sejam extraídos e reutilizados sem autorização.

Exercícios Guiados

1. Qual o método HTTP utilizado pela mensagem de solicitação a seguir?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

2. Quando um servidor HTTP hospeda muitos sites, como ele identifica para qual deles é feita uma solicitação?

3. Qual parâmetro é fornecido pela string de solicitação da URL `https://www.google.com/search?q=LPI`?

4. Por que a solicitação HTTP a seguir não é adequada para armazenamento em cache?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Exercícios Exploratórios

1. Como seria possível usar o navegador web para monitorar as solicitações e respostas feitas por uma página HTML?

2. Os servidores HTTP que fornecem conteúdo estático geralmente mapeiam o caminho solicitado para um arquivo no sistema de arquivos do servidor. O que acontece quando o caminho da solicitação aponta para um diretório?

3. O conteúdo dos arquivos enviados por HTTPS é protegido por criptografia e, portanto, não podem ser lidos por computadores entre o cliente e o servidor. Apesar disso, esses computadores intermediários podem identificar qual recurso o cliente solicitou do servidor?

Resumo

Esta lição cobre os fundamentos do HTTP, o principal protocolo usado por aplicativos clientes para solicitar recursos de servidores web. A lição abrange os seguintes conceitos:

- Mensagens de solicitação, campos de cabeçalho e métodos.
- Códigos de status das respostas.
- Como os servidores HTTP geram respostas.
- Recursos de HTTP úteis para armazenamento em cache e gerenciamento de sessão.

Respostas aos Exercícios Guiados

1. Qual o método HTTP utilizado pela mensagem de solicitação a seguir?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

   O método POST.

2. Quando um servidor HTTP hospeda muitos sites, como ele identifica para qual deles é feita uma solicitação?

   O campo `Host` no cabeçalho da solicitação informa o website de destino.

3. Qual parâmetro é fornecido pela string de solicitação da URL `https://www.google.com/search?q=LPI`?

   O parâmetro denominado `q` com o valor `LPI`.

4. Por que a solicitação HTTP a seguir não é adequada para armazenamento em cache?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

   Como as solicitações feitas com o método POST implicam em uma operação de escrita no servidor, elas não devem ser armazenadas em cache.

Respostas aos Exercícios Exploratórios

1. Como seria possível usar o navegador web para monitorar as solicitações e respostas feitas por uma página HTML?

   Todos os navegadores mais comuns oferecem *ferramentas de desenvolvimento* que, entre outras coisas, podem mostrar todas as transações de rede que foram realizadas pela página atual.

2. Os servidores HTTP que fornecem conteúdo estático geralmente mapeiam o caminho solicitado para um arquivo no sistema de arquivos do servidor. O que acontece quando o caminho da solicitação aponta para um diretório?

   Depende de como o servidor está configurado. Por padrão, a maioria dos servidores HTTP procura um arquivo chamado `index.html` (ou outro nome predefinido) nesse mesmo diretório e o envia como resposta. Se o arquivo não estiver lá, o servidor emitirá uma resposta `404 Not Found`.

3. O conteúdo dos arquivos enviados por HTTPS é protegido por criptografia e, portanto, não pode ser lido por computadores entre o cliente e o servidor. Apesar disso, esses computadores intermediários podem identificar qual recurso o cliente solicitou do servidor?

   Não, porque os próprios cabeçalhos HTTP de solicitação e resposta também são criptografados por TLS.

Linux Professional Institute

Tópico 032: Marcação de documentos HTML

032.1 A anatomia do documento HTML

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 032.1

Peso

2

Áreas chave de conhecimento

- Criar um documento HTML simples
- CEntender o papel do HTML
- CEntender o esqueleto do HTML
- CEntender a sintaxe HTML (tags, atributos, comentários)
- CEntender o cabeçalho (head) do HTML
- CEntender as tags de metadados (meta)
- CEntender a codificação de caracteres

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`
- `<meta>`, incluindo os atributos `charset` (UTF-8), `name` e `content`

032.1 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	032 Marcação de documentos HTML
Objetivo:	032.1 A anatomia do documento HTML
Lição:	1 de 1

Introdução

O HTML (*HyperText Markup Language*) é uma linguagem de marcação que informa aos navegadores como estruturar e exibir as páginas web. A versão atual é a 5.0, lançada em 2012. A sintaxe HTML é definida pelo *World Wide Web Consortium* (W3C).

O HTML é uma habilidade fundamental para o desenvolvimento web, pois é a linguagem que define a estrutura e boa parte da aparência de um website. Se você deseja seguir uma carreira em desenvolvimento web, o HTML é um excelente ponto de partida.

Anatomia de um documento HTML

Uma página HTML básica tem a seguinte estrutura:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My HTML Page</title>
    <!-- This is the Document Header -->
  </head>

  <body>
    <!-- This is the Document Body -->
  </body>
</html>
```

Vamos analisá-la em detalhes.

Tags HTML

O HTML usa *elementos* e *tags* para descrever e formatar o conteúdo. As tags consistem em parênteses angulares ao redor do nome da tag, por exemplo `<title>`. O nome da tag não faz distinção entre maiúsculas e minúsculas, embora o World Wide Web Consortium (W3C) recomende o uso de minúsculas nas versões atuais do HTML. Essas tags HTML são usadas para construir elementos HTML. A tag `<title>` é um exemplo de *tag de abertura* de um elemento HTML que define o título de um documento HTML. No entanto, um elemento possui dois outros componentes. Um elemento `<title>` completo seria parecido com este:

```
<title>My HTML Page</title>
```

Aqui, `My HTML Page` representa o *conteúdo* do elemento, ao passo que `</title>` é a tag de *fechamento* que declara que este elemento está completo.

NOTE Nem todos os elementos HTML precisam ser fechados; nesses casos, falamos de elementos vazios, elementos de auto-fechamento ou elementos nulos.

Aqui estão os outros elementos HTML do exemplo anterior:

`<html>`

Abrange todo o documento HTML. Contém todas as tags que compõem a página. Também indica que o conteúdo deste arquivo está em linguagem HTML. A tag de fechamento correspondente é `</html>`.

`<head>`

Um receptáculo para todas as metainformações relacionadas à página. A tag de fechamento correspondente a este elemento é `</head>`.

`<body>`

Um receptáculo para o conteúdo da página e sua representação estrutural. A tag de fechamento correspondente é `</body>`.

As tags `<html>`, `<head>`, `<body>` e `<title>` são as chamadas *tags estruturais*, que fornecem o esqueleto básico de um documento HTML. Em particular, elas informam o navegador web de que ele está lendo uma página HTML.

NOTE Dentre esses elementos HTML, o único que é necessário para um documento HTML ser validado é a tag `<title>`.

Como se vê, cada página HTML é um documento bem estruturado que poderia ser comparado a uma árvore, na qual o elemento `<html>` representa a raiz e os elementos `<head>` e `<body>` são os primeiros galhos. O exemplo mostra que é possível aninhar elementos. Assim, o elemento `<title>` é aninhado dentro de `<head>`, que por sua vez é aninhado dentro de `<html>`.

Para garantir que seu código HTML possa ser lido e mantido, todos os elementos HTML devem estar fechados corretamente e em ordem. Os navegadores web ainda serão capazes de exibir seu site conforme o esperado, mas o aninhamento incorreto de elementos e suas tags é uma prática que pode causar erros.

Finalmente, uma menção especial à declaração *doctype* que aparece no alto da estrutura do documento de nosso exemplo. `<!DOCTYPE>` não é uma tag HTML, mas uma instrução para o navegador web que especifica a versão do HTML usada no documento. Na estrutura básica do documento HTML vista anteriormente, usamos `<!DOCTYPE html>`, especificando que o HTML5 foi usado no documento.

Comentários em HTML

Ao criar uma página HTML, é recomendável inserir comentários no código para melhorar sua legibilidade e descrever a finalidade dos blocos de código maiores. As tags `<!--` e `-->` indicam os comentários, como mostrado no exemplo a seguir:

```
<!-- This is a comment. -->

<!--
    This is a
    multiline
    comment.
-->
```

O exemplo demonstra que os comentários, no HTML, podem ser postos em uma única linha, mas também podem se estender por várias linhas. De qualquer maneira, o resultado é que o texto entre `<!--` e `-->` é ignorado pelo navegador web e, portanto, não é exibido na página HTML. Com base nessas considerações, podemos deduzir que a página HTML básica mostrada no parágrafo "Anatomia de um documento HTML" não exibe nenhum texto, porque as linhas `<!-- This is the Document Header -->` e `<!-- This is the Document Body -->` são apenas dois comentários.

WARNING Os comentários não podem ser aninhados.

Atributos HTML

As tags HTML podem incluir um ou mais *atributos* para especificar detalhes do elemento HTML. Uma tag simples com dois atributos tem o seguinte formato:

```
<tag attribute-a="value-a" attribute-b="value-b">
```

Os atributos devem sempre ser definidos na tag de abertura.

Um atributo consiste em um nome, que indica a propriedade a ser definida, um sinal de igual, mais o valor desejado entre aspas. As aspas podem ser simples ou duplas, mas recomenda-se manter o uso de aspas simples ou duplas de forma consistente em todo o projeto. É importante não misturar aspas simples e duplas em um mesmo valor de atributo, pois o navegador web não reconhecerá aspas mistas como uma unidade.

NOTE É possível incluir um tipo de aspas dentro do outro tipo sem problemas. Por exemplo, se for preciso usar `'` no valor de um atributo, podemos envolver esse valor com `"`. Todavia, se você quiser usar o mesmo tipo de aspas dentro do valor e em torno dele, será necessário usar `&quot;` para `"` e `&apos;` para `'`.

Os atributos podem ser categorizados em *atributos genéricos* e *atributos específicos*, conforme explicado nas seções a seguir.

Atributos genéricos

Os atributos genéricos podem ser usados em qualquer elemento HTML. Dentre eles podemos citar:

`title`

Descreve o conteúdo do elemento. Seu valor é geralmente exibido como como uma dica de contexto quando o usuário passa o cursor sobre o elemento.

`id`

Associa um identificador único a um elemento. Esse identificador deve ser único dentro do documento; este não será validado se vários elementos compartilharem o mesmo `id`.

`style`

Atribui propriedades gráficas (estilos CSS) ao elemento.

`class`

Especifica uma ou várias classes para o elemento em uma lista de nomes de classes separados por espaços. Essas classes podem ser referenciadas em folhas de estilo CSS.

`lang`

Especifica o idioma do conteúdo do elemento usando códigos de idioma de dois caracteres no padrão ISO-639.

NOTE

O desenvolvedor pode armazenar informações personalizadas sobre um elemento definindo um atributo `data-`, que é indicado prefixando o nome desejado com `data-`, como em `data-additionalinfo`. Este atributo pode receber um valor, como qualquer outro atributo.

Atributos específicos

Outros atributos são específicos a cada elemento HTML. Por exemplo, o atributo `src` de um elemento HTML `<img>` especifica a URL de uma imagem. Existem muitos outros atributos específicos, que serão abordados nas próximas lições.

Cabeçalho do documento

O cabeçalho do documento define as metainformações sobre a página e é descrito pelo elemento `<head>`. Por padrão, as informações no cabeçalho do documento não são processadas pelo navegador web. Embora seja possível incluir, dentro do elemento `<head>`, elementos HTML que podem ser exibidos na página, a prática não é recomendada.

Título

O título do documento é especificado com o elemento `<title>`. O título definido entre as tags aparece na barra de título do navegador e é o nome sugerido quando a página é salva nos Favoritos. Ele também é exibido como título da página nos resultados do mecanismo de busca.

Eis um exemplo do uso desse elemento:

```
<title>My test page</title>
```

A tag `<title>` é obrigatória em todos os documentos HTML e deve aparecer somente uma vez por documento.

NOTE Não confunda o título do documento com o cabeçalho da página, que é definido no corpo.

Metadados

O elemento `<meta>` é usado para especificar metainformações, de maneira a descrever melhor o conteúdo de um documento HTML. Trata-se de um elemento de auto-fechamento, ou seja, ele não tem uma tag de fechamento. Além dos atributos genéricos, que são válidos para todos os elementos HTML, o elemento `<meta>` também usa os seguintes atributos:

`name`
: Define quais metadados serão descritos neste elemento. Pode ser definido para qualquer valor personalizado, mas os valores comumente usados são `author`, `description` e `keywords`.

`http-equiv`
: Fornece um cabeçalho HTTP para o valor do atributo `content`. Um valor comum é `refresh`, que será explicado mais tarde. Se este atributo for definido, o atributo `name` não deve ser definido.

`content`
: Fornece o valor associado ao atributo `name` ou `http-equiv`.

`charset`
: Especifica a codificação de caracteres para o documento HTML, por exemplo `utf-8` para configurá-lo para o Unicode Transformation Format — 8-bit.

Adicionar um autor, descrição e palavras-chave

Usando a tag `<meta>`, podemos incluir informações adicionais sobre o autor da página HTML e descrever o conteúdo da página, desta forma:

```
<meta name="author" content="Name Surname">
<meta name="description" content="A short summary of the page content.">
```

Procure incluir uma série de palavras-chave relacionadas ao conteúdo da página na descrição. Essa descrição costuma ser a primeira coisa que um usuário vê ao navegar com um mecanismo de busca.

Se quiser incluir palavras-chave adicionais relacionadas à página para os mecanismos de busca, adicione o seguinte elemento:

```
<meta name="keywords" content="keyword1, keyword2, keyword3, keyword4, keyword5">
```

NOTE

No passado, os spammers inseriam centenas de palavras-chave e descrições que não tinham nada a ver com o conteúdo real da página, de forma que elas também apareciam em pesquisas não relacionadas aos termos que as pessoas procuravam. Hoje em dia, as tags `<meta>` foram relegadas a uma posição de importância secundária e são usadas apenas para consolidar os tópicos tratados na página em si; portanto, não é mais possível enganar os novos e mais sofisticados algoritmos dos mecanismos de busca.

Redirecionar uma página HTML e definir um intervalo de tempo para que o documento se atualize

Usando a tag `<meta>`, é possível atualizar automaticamente uma página HTML depois de um período determinado (por exemplo, após 30 segundos), desta forma:

```
<meta http-equiv="refresh" content="30">
```

Outra alternativa é redirecionar uma página web para outra página web após o mesmo período de tempo com o seguinte código:

```
<meta http-equiv="refresh" content="30; url=http://www.lpi.org">
```

Neste exemplo, o usuário é redirecionado da página atual para `http://www.lpi.org` após 30

segundos. Os valores podem ser os que você preferir. Se especificarmos, por exemplo, `content="0; url=http://www.lpi.org"`, a página será redirecionada imediatamente.

Especificar a codificação de caracteres

O atributo `charset` especifica a codificação de caracteres para o documento HTML. Um exemplo comum é:

```
<meta charset="utf-8">
```

Este elemento especifica que a codificação de caracteres do documento é `utf-8`, um conjunto de caracteres universal que inclui praticamente qualquer caractere de qualquer idioma humano. Portanto, ao usá-lo, você evita problemas de exibição que poderiam surgir se escolhesse outros conjuntos de caracteres, como o ISO-8859-1 (o alfabeto latino).

Outros exemplos úteis

Eis duas outras aplicações úteis da tag `<meta>`:

- Configurar cookies para rastrear um visitante do site.
- Assumir o controle da janela de visualização (a área visível de uma página web dentro de uma janela do navegador), que depende do tamanho da tela do dispositivo do usuário (por exemplo, um telefone celular ou um computador).

No entanto, esses dois exemplos estão além do escopo do exame e são citados aqui como mera curiosidade.

Exercícios Guiados

1. Para cada uma das seguintes tags, indique a tag de fechamento correspondente:

`<body>`	
`<head>`	
`<html>`	
`<meta>`	
`<title>`	

2. Qual é a diferença entre uma tag e um elemento? Use esta entrada como referência:

```
<title>HTML Page Title</title>
```

3. Quais são as tags que indicam um comentário?

4. Explique o que é um atributo e dê alguns exemplos para a tag `<meta>`.

Exercícios Exploratórios

1. Crie um documento simples em HTML versão 5 com o título `My first HTML document` e um único parágrafo no corpo, contendo o texto `Hello World`. Use a tag de parágrafo `<p>` no corpo da página.

2. Adicione o autor (`Kevin Author`) e a descrição (`This is my first HTML page.`) ao documento HTML.

3. Adicione as seguintes palavras-chave relacionadas ao documento HTML: `HTML`, `Example`, `Test` e `Metadata`.

4. Adicione o elemento `<meta charset="ISO-8859-1">` ao cabeçalho do documento e mude o texto de `Hello World` para japonês (`□□□□□□□□`). O que acontece? Como seria possível resolver o problema?

5. Depois de alterar o texto do parágrafo de volta para `Hello World`, redirecione a página HTML para `https://www.google.com` após 30 segundos e inclua um comentário explicando essa operação no cabeçalho do documento.

Resumo

Nesta lição, você aprendeu:

- O papel do HTML
- A estrutura do HTML
- A sintaxe do HTML (tags, atributos, comentários)
- O cabeçalho do HTML
- As meta tags
- Como criar um documento HTML simples

Os seguintes termos foram discutidos nesta lição:

`<!DOCTYPE html>`
: A tag de declaração.

`<html>`
: O envoltório de todas as tags que constituem a página HTML.

`<head>`
: O envoltório de todos os elementos do cabeçalho.

`<body>`
: O envoltório de todos os elementos do corpo da página.

`<meta>`
: A tag dos metadados, usada para especificar informações adicionais sobre a página HTML (como o autor, a descrição e a codificação dos caracteres).

Respostas aos Exercícios Guiados

1. Para cada uma das seguintes tags, indique a tag de fechamento correspondente:

`<body>`	`</body>`
`<head>`	`</head>`
`<html>`	`</html>`
`<meta>`	None
`<title>`	`</title>`

2. Qual é a diferença entre uma tag e um elemento? Use esta entrada como referência:

   `<title>Título da página HTML</title>`.

   Um elemento HTML consiste em uma tag de abertura, uma tag de fechamento e tudo o que está entre elas. As tags HTML são usadas para demarcar o início ou o fim de um elemento. Portanto, `<title>Título da página HTML</title>` é um elemento HTML, ao passo que `<title>` e `</title>` são, respectivamente, as tags de abertura e fechamento.

3. Quais são as tags que indicam um comentário?

   Os comentários são inseridos entre as tags `<!--` e `-->` e podem estar em uma única linha ou estender-se por múltiplas linhas.

4. Explique o que é um atributo e dê alguns exemplos para a tag `<meta>`.

   Os atributos são usados para especificar mais precisamente um elemento HTML. Por exemplo, a tag `<meta>` emprega o par de atributos `name` e `content` para informar o autor e a descrição de uma página HTML. Já o atributo `charset` permite especificar a codificação de caracteres para o documento HTML.

Respostas aos Exercícios Exploratórios

1. Crie um documento simples em HTML versão 5 com o título `My first HTML document` e um único parágrafo no corpo, contendo o texto `Hello World`. Use a tag de parágrafo `<p>` no corpo da página.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

2. Adicione o autor (`Kevin Author`) e a descrição (`This is my first HTML page.`) ao documento HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

3. Adicione as seguintes palavras-chave relacionadas ao documento HTML: `HTML`, `Example`, `Test` e `Metadata`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

4. Adicione o elemento `<meta charset="ISO-8859-1">` ao cabeçalho do documento e mude o texto de `Hello World` para japonês (`こんにちは世界`). O que acontece? Como seria possível resolver o problema?

   Se o exemplo for executado conforme descrito, o texto em japonês não será exibido corretamente. Isso ocorre porque ISO-8859-1 representa a codificação de caracteres para o alfabeto latino. Para visualizar o texto, é necessário alterar a codificação de caracteres, usando, por exemplo, UTF-8 (`<meta charset="utf-8">`).

5. Depois de alterar o texto do parágrafo de volta para `Hello World`, redirecione a página HTML para `https://www.google.com` após 30 segundos e inclua um comentário explicando essa operação no cabeçalho do documento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
    <meta charset="utf-8">
    <!-- The page is redirected to Google after 30 seconds -->
    <meta http-equiv="refresh" content="30; url=https://www.google.com">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

032.2 A sema□ntica do HTML e a hierarquia de documentos

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 032.2

Peso

2

Áreas chave de conhecimento

- Criar marcação para conteúdo em um documento HTML
- Entender a estrutura hierárquica do texto no HTML
- Diferenciar entre elementos HTML de bloco e de linha
- Entender os elementos estruturais semânticos importantes no HTML

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
- `<p>`
- `<ul>`, `<ol>`, `<li>`
- `<dl>`, `<dt>`, `<dd>`
- `<pre>`
- `<blockquote>`
- `<strong>`, `<em>`, `<code>`
- `<b>`, `<i>`, `<u>`
- `<span>`

- `<div>`
- `<main>`, `<header>`, `<nav>`, `<section>`, `<footer>`

032.2 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	032 Marcação de documentos HTML
Objetivo:	032.2 A semântica do HTML e a hierarquia de documentos
Lição:	1 de 1

Introdução

Na lição anterior, aprendemos que o HTML é uma linguagem de marcação capaz de descrever semanticamente o conteúdo de um site. Um documento HTML contém uma estrutura, que consiste nos elementos HTML `<html>`, `<head>` e `<body>`. Enquanto o elemento `<head>` descreve um bloco de metainformações sobre o documento HTML que será invisível para o visitante do site, o elemento `<body>` abriga muitos outros elementos que definem a estrutura e o conteúdo do documento HTML.

Nesta lição, falaremos da formatação de texto, dos elementos semânticos fundamentais do HTML e sua finalidade e de como estruturar um documento HTML. Usaremos como exemplo uma lista de compras.

NOTE

Todos os exemplos de código apresentados estão dentro do elemento `<body>` de um documento HTML contendo a estrutura completa. Para facilitar a leitura, não incluiremos a estrutura HTML em todos os exemplos desta lição.

Texto

Em HTML, nenhum bloco de texto deve estar nu, fora de um elemento. Até mesmo um parágrafo curto deve ser rodeado pelas tags HTML `<p>`, que representam um *parágrafo*.

```
<p>Short text element spanning only one line.</p>
<p>A text element containing much longer text that may span across multiple lines,
depending on the size of the web browser window.</p>
```

Quando aberto em um navegador, esse código HTML produz o resultado mostrado na Figure 1.

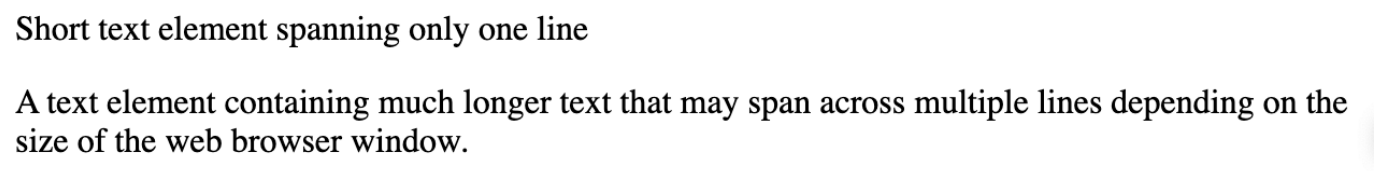

Figure 1. Representação do código HTML acima em um navegador, exibindo dois parágrafos de texto. O primeiro parágrafo é bem curto. O segundo é um pouco mais longo e se estende em uma segunda linha.

Por padrão, os navegadores web adicionam espaçamento antes e depois dos elementos `<p>` para melhorar a legibilidade. Por essa razão, `<p>` é considerado um *elemento de bloco*.

Títulos

O HTML define seis níveis de títulos e subtítulos para descrever e estruturar o conteúdo de um documento HTML. Esses títulos são designados pelas tags `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` e `<h6>`.

```
<h1>Heading level 1 to uniquely identify the page</h1>
<h2>Heading level 2</h2>
<h3>Heading level 3</h3>
<h4>Heading level 4</h4>
<h5>Heading level 5</h5>
<h6>Heading level 6</h6>
```

Um navegador web exibiria este código HTML como mostrado na Figure 2.

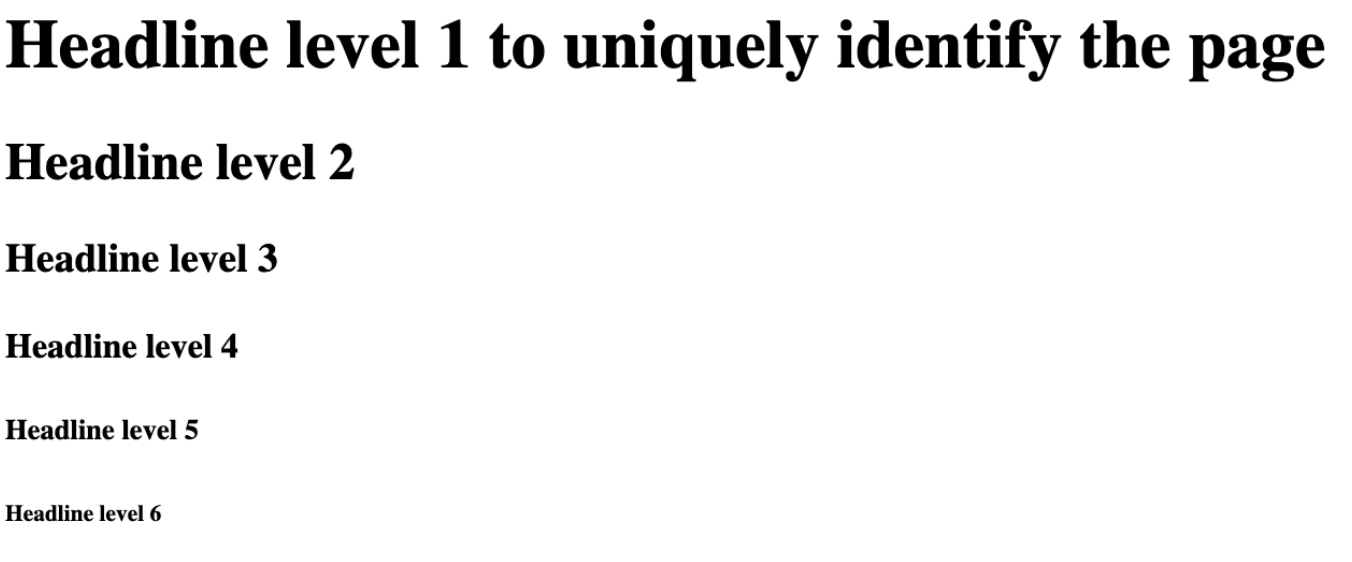

Figure 2. Representação do código HTML acima em um navegador, mostrando diferentes níveis de títulos em um documento HTML. A hierarquia dos títulos é indicada pelo tamanho do texto.

Se você está familiarizado com processadores de texto como o LibreOffice ou o Microsoft Word, provavelmente vai notar algumas semelhanças na utilização dos diferentes níveis de títulos e como eles são processados no navegador. Por padrão, o HTML usa o tamanho para indicar a hierarquia e a importância dos títulos e adiciona espaços antes e depois deles para separá-los visualmente do conteúdo.

O título que usa o elemento `<h1>` fica no topo da hierarquia e, assim, é considerado o mais importante, o que identifica o conteúdo da página. Ele é comparável ao elemento `<title>` discutido na lição anterior, mas dentro do conteúdo do documento HTML. Os elementos de título subsequentes podem ser usados para estruturar mais detalhadamente o conteúdo. Procure não saltar níveis. A hierarquia de um documento deve começar com `<h1>`, continuar com `<h2>`, em seguida `<h3>` e assim por diante. Não é obrigatório usar todos os elementos de título até o `<h6>` se o conteúdo não o exigir.

NOTE

Os títulos são ferramentas importantes para estruturar um documento HTML, tanto semântica quanto visualmente. No entanto, eles nunca devem ser usados para aumentar o tamanho de um texto estruturalmente sem importância. Pelo mesmo princípio, não se deve colocar um parágrafo curto em negrito ou itálico para parecer um título; use tags de título para marcar títulos.

Vamos começar a criação de nossa lista de compras em HTML definindo sua estrutura. Criamos primeiro um elemento `<h1>` para conter o título da página, neste caso `Garden Party`, seguido por um pequeno texto inserido em um elemento `<p>`. Em seguida, usamos dois elementos `<h2>` para

introduzir as duas seções do conteúdo: `Agenda` e `Please bring`.

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<h2>Please bring</h2>
```

Quando aberto em um navegador web, este código produz o resultado mostrado na Figure 3.

Figure 3. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples com um convite para uma festa ao ar livre, com dois títulos para a programação e uma lista de coisas a levar.

Quebras de linha

Às vezes, pode ser necessário fazer uma *quebra de linha* sem inserir outro elemento `<p>` ou qualquer elemento de bloco semelhante. Nesses casos, você pode usar o elemento de auto-fechamento `<br>`. Esse elemento deve ser usado somente para inserir quebras de linha inerentes ao conteúdo, como no caso de poemas, letras de música ou endereços. Para separar mudanças de conteúdo, é preferível usar um elemento `<p>`.

Por exemplo, poderíamos dividir o texto do parágrafo informativo de nosso exemplo anterior da seguinte maneira:

```
<p>
  Invitation to John's garden party.<br>
  Saturday, next week.
</p>
```

No navegador, esse código HTML teria o resultado mostrado na Figure 4.

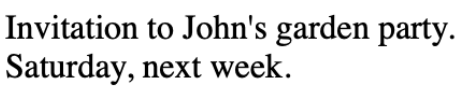

Figure 4. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples com uma quebra de linha forçada.

Linhas Horizontais

O elemento `<hr>` define uma linha horizontal, também chamada de *separador horizontal*. Por padrão, ela se estende por toda a largura do elemento pai. O elemento `<hr>` ajuda a definir uma mudança temática no conteúdo ou separar as seções do documento. Esse é um elemento vazio e, portanto, não tem tag de fechamento.

Em nosso exemplo, poderíamos separar os dois títulos:

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<hr>
<h2>Please bring</h2>
```

A Figure 5 mostra o resultado desse código.

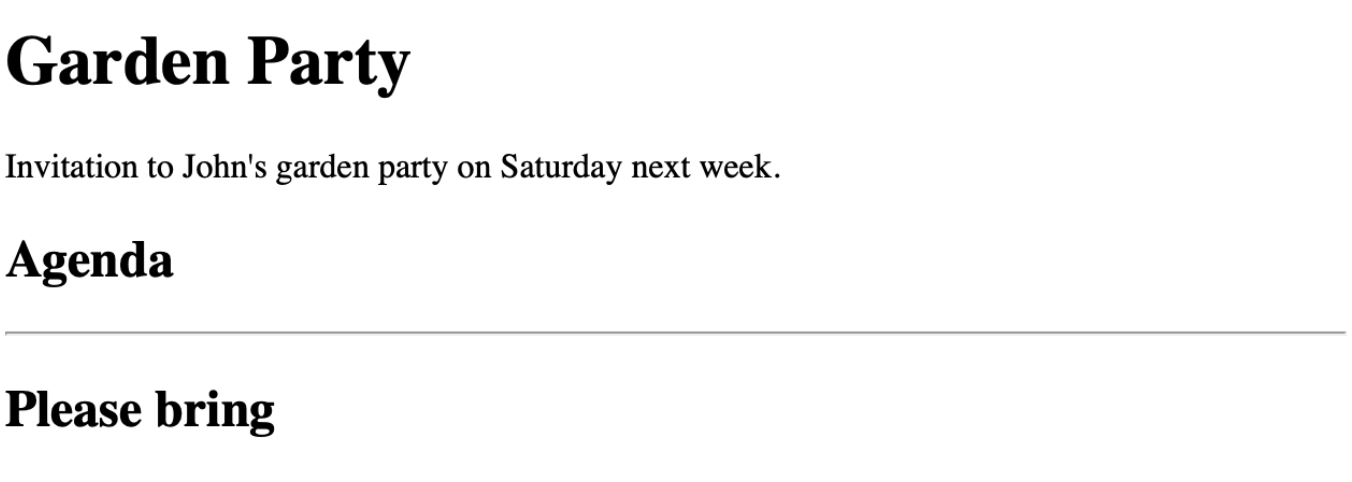

Figure 5. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples com uma lista de compras em duas seções separadas por uma linha horizontal.

Listas em HTML

Em HTML, podemos definir três tipos de listas:

Listas ordenadas
: nas quais a ordem dos elementos listados é importante

Listas não ordenadas
: nas quais a ordem dos elementos listados não é particularmente importante

Listas de definição
: para descrever mais detalhadamente certos termos

Cada uma delas contém um certo número de *itens de lista*. Vamos conhecer melhor esses tipos.

Listas ordenadas

Uma *lista ordenada* em HTML, definida pelo elemento `<ol>`, é uma coleção organizada de *itens de lista*. O que torna este elemento especial é que a ordem dos itens é relevante. Para enfatizar o fato, os navegadores web exibem números por padrão antes dos elementos filho da lista.

NOTE Os elementos `<li>` são os únicos elementos filho válidos dentro de um elemento `<ol>`.

Em nosso exemplo, podemos preencher a programação da festa ao ar livre usando um elemento `<ol>` com o seguinte código:

```
<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Em um navegador web, esse código HTML produz o resultado mostrado na Figure 6.

Figure 6. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples contendo um título de segundo nível seguido por uma lista ordenada com quatro itens referentes à programação de uma festa ao ar livre.

Opções

Como vemos neste exemplo, os itens da lista são organizados com algarismos arábicos começando em 1 por padrão. No entanto, é possível alterar esse comportamento especificando o atributo `type` da tag `<ol>`. Os valores válidos para este atributo são `1` para algarismos arábicos, `A` para letras maiúsculas, `a` para letras minúsculas, `I` para algarismos romanos maiúsculos e `i` para algarismos romanos minúsculos.

Se quiser, você também pode definir o valor inicial usando o atributo `start` da tag `<ol>`. O atributo `start` sempre é acompanhado por um valor numérico decimal, mesmo que o atributo `type` defina um tipo diferente de numeração.

Por exemplo, poderíamos ajustar a lista ordenada do exemplo anterior para que os itens da lista sejam prefixados com letras maiúsculas, começando com a letra C, como mostrado no exemplo a seguir:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Em um navegador web, esse código HTML é exibido como na Figure 7.

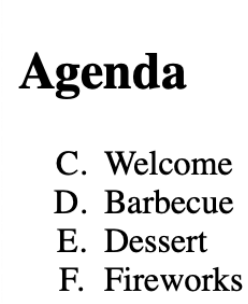

Figure 7. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples contendo um título de segundo nível seguido por uma lista ordenada de itens prefixados por letras maiúsculas começando com a letra C.

A ordem dos itens da lista também pode ser invertida usando o atributo `reversed` sem um valor.

NOTE

Em uma lista ordenada, também é possível definir o valor inicial de um item específico usando o atributo `value` da tag `<li>`. Os itens da lista serão incrementados a partir desse número. O atributo `value` sempre leva um valor numérico decimal.

Listas não ordenadas

Uma *lista não ordenada* contém uma série de itens de lista que, ao contrário daqueles em uma lista ordenada, não têm uma ordem ou sequência especial. O elemento HTML para esse tipo de lista é `<ul>`. Também neste caso, `<li>` é o elemento HTML que demarca os itens de lista.

NOTE

Os elementos `<li>` são os únicos elementos filho válidos dentro de um elemento `<ul>`.

Em nosso site de exemplo, podemos usar uma lista não ordenada para sugerir os itens que os convidados devem trazer para a festa. Usamos para isso o seguinte código HTML:

```
<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

Em um navegador da web, esse código HTML produz o resultado mostrado na Figure 8.

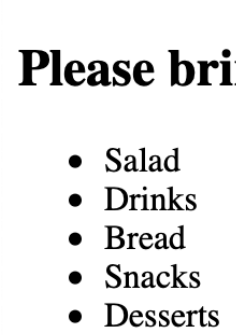

Figure 8. Representação do código HTML acima em um navegador, mostrando um exemplo de documento simples contendo um título de segundo nível seguido por uma lista não ordenada de itens com sugestões de alimentos que os convidados devem trazer para a festa.

Por padrão, cada item da lista é representado por um marcador circular. É possível alterar a aparência do marcador usando CSS, o que será discutido em lições posteriores.

Listas aninhadas

As listas podem ser aninhadas em outras listas, como listas ordenadas em listas não ordenadas e vice-versa. Para isso, a lista aninhada deve fazer parte de um elemento de lista `<li>`, já que `<li>` é o único elemento filho válido nas listas ordenadas e não ordenadas. Ao aninhar, cuidado para não sobrepor as tags HTML.

Em nosso exemplo, poderíamos adicionar algumas informações à programação que criamos antes, como mostrado no exemplo a seguir:

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li>Vegetables</li>
      <li>Meat</li>
      <li>Burgers, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Um navegador web exibiria esse código da maneira mostrada na Figure 9.

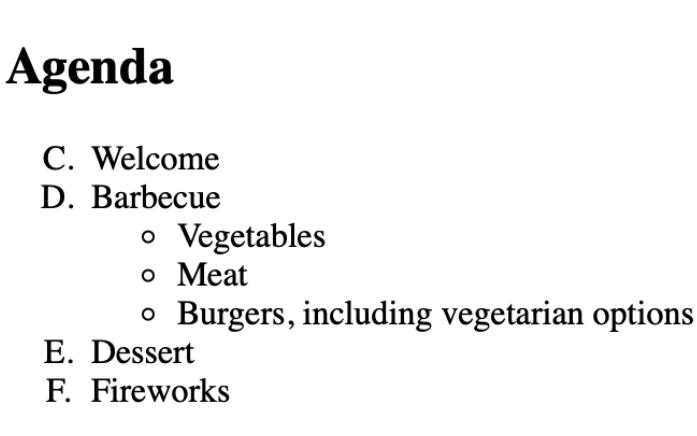

Figure 9. Representação do código HTML acima em um navegador, mostrando uma lista não ordenada aninhada em uma lista ordenada, representando a programação de uma festa ao ar livre.

Podemos ainda mais longe e aninhar múltiplos níveis de profundidade. Teoricamente, não há limite de quantidade para as listas aninhadas. Porém, ao fazer isso, leve em conta a legibilidade da página.

Listas de definição

Uma *lista de definição* é criada com o elemento `<dl>` e representa um dicionário de *termos* e *descrições*. O termo é uma palavra ou nome a descrever e a descrição é a explicação. As listas de definição vão desde pares simples de termo-descrição até definições mais extensas.

Os termos (ou *termos de definição*) são designados com o elemeno `<dt>`; já a descrição usa o elemento `<dd>`.

Um exemplo de lista de definição seria uma lista de frutas exóticas explicando a aparência delas.

```
<h3>Exotic Fruits</h3>
<dl>
  <dt>Banana</dt>
  <dd>
    A long, curved fruit that is yellow-skinned when ripe. The fruit's skin
    may also have a soft green color when underripe and get brown spots when
    overripe.
  </dd>

  <dt>Kiwi</dt>
  <dd>
    A small, oval fruit with green flesh, black seeds, and a brown, hairy
    skin.
  </dd>

  <dt>Mango</dt>
  <dd>
    A fruit larger than a fist, with a green skin, orange flesh, and one big
    seed. The skin may have spots ranging from green to yellow or red.
  </dd>
</dl>
```

Em um navegador web, isso produziria o resultado mostrado na Figure 10.

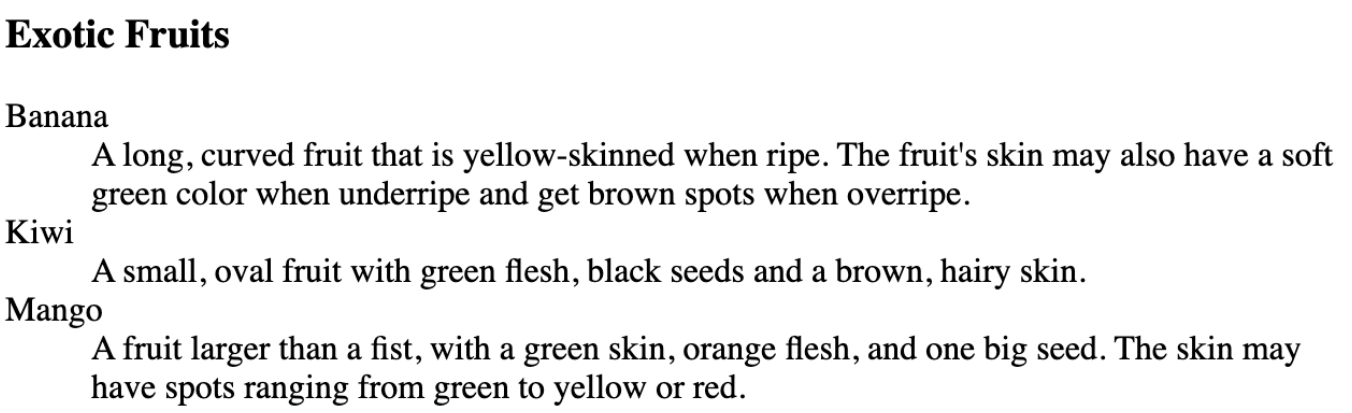

Figure 10. Exemplo de lista de definição usando frutas exóticas. A lista descreve a aparência de três frutas diferentes.

NOTE

Ao contrário das listas ordenadas e não ordenadas, em uma lista de definição qualquer elemento HTML pode ser usado como filho. Isso permite agrupar elementos e estilizá-los usando CSS.

Formatação de texto na linha

Em HTML, podemos usar elementos de formatação para alterar a aparência do texto. Esses elementos podem ser categorizados como *elementos de apresentação* ou *elementos de expressão* (ou de frase).

Elementos de apresentação

Os elementos básicos de apresentação alteram a fonte ou a aparência do texto; eles são `<b>`, `<i>`, `<u>` e `<tt>`. Esses elementos originalmente eram definidos antes que o CSS passasse a permitir o uso de negrito, itálico, etc. Atualmente, existem maneiras melhores de alterar a aparência do texto, mas ainda encontramos esses elementos ocasionalmente.

Texto em negrito

Para deixar o texto em negrito, usamos o elemento `<b>` como ilustrado no exemplo a seguir. O resultado aparece na Figure 11.

```
This <b>word</b> is bold.
```

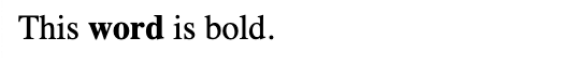

Figure 11. A tag `<b>` é usada para deixar o texto em negrito.

De acordo com a especificação do HTML5, o elemento `<b>` deve ser usado apenas quando não houver tags mais apropriadas. O elemento que produz a mesma aparência visual, aumentando ao mesmo tempo a importância semântica do texto marcado, é `<strong>`.

Texto em itálico

Para colocar o texto em itálico, usamos o elemento `<i>`, como ilustrado no exemplo a seguir. O resultado aparece na Figure 12.

```
This <i>word</i> is in italics.
```

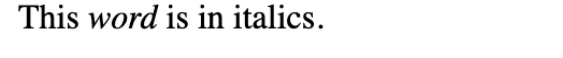

Figure 12. A tag `<i>` é usada para colocar o texto em itálico.

De acordo com a especificação do HTML5, o elemento `<i>` deve ser usado apenas quando não houver

tags mais apropriadas.

Texto sublinhado

Para sublinhar o texto, usamos o elemento `<u>`, como ilustrado no exemplo a seguir. O resultado aparece na Figure 13.

```
This <u>word</u> is underlined.
```

This word is underlined.

Figure 13. A tag `<u>` é usada para sublinhar um texto.

De acordo com a especificação do HTML 5, o elemento `<u>` deve ser usado apenas quando não houver maneiras melhores de sublinhar o texto. O CSS oferece uma alternativa mais moderna.

Largura fixa ou fonte monoespaçada

Para exibir texto em fonte monoespaçada (largura fixa), freqüentemente usada para exibir código de computador, usamos o elemento `<tt>`, como ilustrado no exemplo a seguir. O resultado aparece na Figure 14.

```
This <tt>word</tt> is in fixed-width font.
```

This `word` is in fixed-width font.

Figure 14. A tag `<tt>` é usada para exibir texto em uma fonte de largura fixa.

A tag `<tt>` não é suportada em HTML5. Os navegadores ainda a exibem como esperado; porém, é preferível usar tags mais apropriadas, como `<code>`, `<kbd>`, `<var>` e `<samp>`.

Elementos de expressão

Os elementos de expressão ou de frase não apenas alteram a aparência do texto, como também adicionam importância semântica a uma palavra ou frase. Ao usá-los, podemos enfatizar uma palavra ou marcá-la como importante. Esses elementos, ao contrário dos elementos de apresentação, são reconhecidos pelos leitores de tela, o que torna o texto mais acessível aos visitantes com deficiência visual e permite que os mecanismos de busca leiam e avaliem melhor o conteúdo da página. Os elementos de frase que usamos ao longo desta lição são `<em>`, `<strong>` e `<code>`.

Texto enfatizado

Para enfatizar um texto, usamos o elemento `<em>`, como mostrado no exemplo a seguir:

```
This <em>word</em> is emphasized.
```

This *word* is emphasized.

Figure 15. A tag `<em>` é usada para enfatizar o texto.

Como vemos, os navegadores web exibem `<em>` com a mesma aparência de `<i>`, mas `<em>` adiciona importância semântica ao elemento de frase, o que melhora a acessibilidade para visitantes com deficiência visual.

Texto forte

Para demonstrar que um texto é importante, usamos o elemento `<strong>` como no exemplo a seguir. O resultado aparece na Figure 16.

```
This <strong>word</strong> is important.
```

This **word** is important.

Figure 16. A tag `<strong>` é usada para marcar a importância de um texto.

Como vemos, os navegadores web exibem `<strong>` da mesma forma que `<b>`, mas `<strong>` adiciona importância semântica ao elemento de frase, o que melhora a acessibilidade para visitantes com deficiência visual.

Código de Computador

Para inserir um trecho de código, podemos colocá-lo dentro do elemento `<code>` como ilustrado no exemplo a seguir. O resultado aparece na Figure 17.

```
The Markdown code <code># Heading</code> creates a heading at the highest level in
the hierarchy.
```

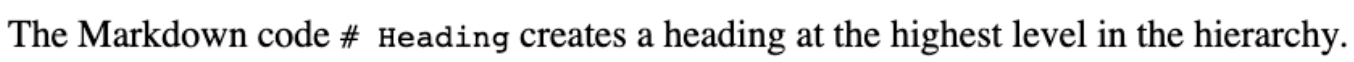

Figure 17. A tag `<code>` é usada para inserir um trecho de código de computador.

Texto destacado

Para destacar o texto com um fundo amarelo, semelhante ao estilo de um marca-texto, usamos o elemento `<mark>` como no exemplo a seguir. O resultado aparece na Figure 18.

```
This <mark>word</mark> is highlighted.
```

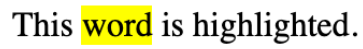

Figure 18. A tag `<mark>` é usada para realçar um texto com um fundo amarelo.

Formatando o texto de nossa lista de compras em HTML

Com base em nossos exemplos anteriores, vamos inserir alguns elementos de expressão para alterar a aparência do texto e, ao mesmo tempo, adicionar importância semântica. O resultado aparece na Figure 19.

```
<h1>Garden Party</h1>
<p>
  Invitation to <strong>John's garden party</strong>.<br>
  <strong>Saturday, next week.</strong>
</p>

<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li><em>Vegetables</em></li>
      <li><em>Meat</em></li>
      <li><em>Burgers</em>, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li><mark>Fireworks</mark></li>
</ol>

<hr>

<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

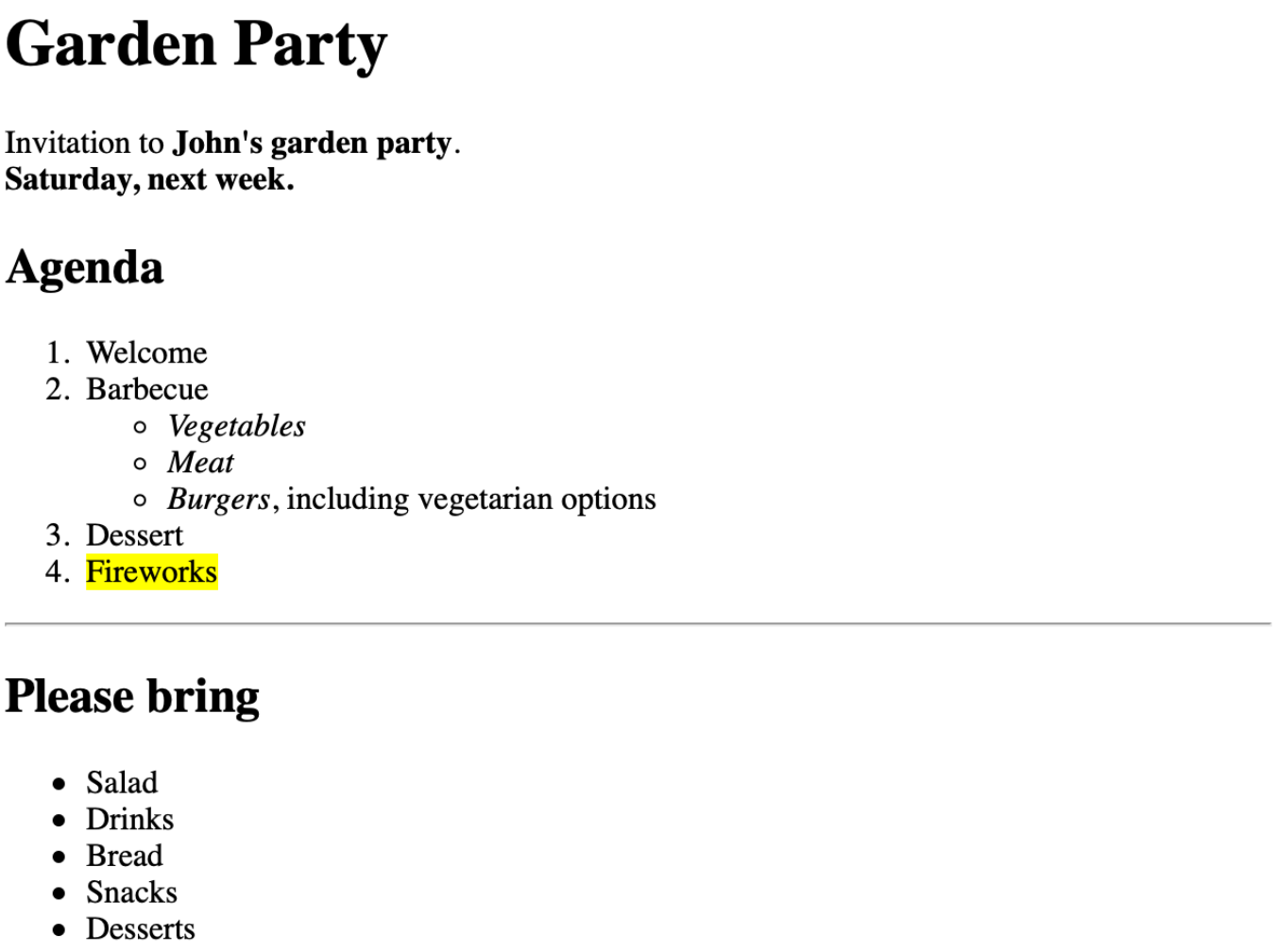

Figure 19. A página HTML com alguns elementos de formatação.

Neste exemplo de documento HTML, as informações mais importantes sobre a festa ao ar livre foram realçadas usando o elemento `<strong>`. As iguarias disponíveis para o churrasco foram enfatizadas com o elemento `<em>`. Os fogos de artifício foram destacados simplesmente com o elemento `<mark>`.

Para treinar, você pode remodelar outras partes do texto usando os outros elementos de formatação.

Texto pré-formatado

Na maioria dos elementos HTML, o espaço em branco geralmente é reduzido a um espaçamento simples ou mesmo totalmente ignorado. No entanto, existe um elemento HTML chamado `<pre>` que permite definir o chamado texto *pré-formatado*. Qualquer espaço em branco incluído no conteúdo deste elemento, incluindo espaços e quebras de linha, é preservado e exibido no navegador web. Além disso, o texto é exibido em uma fonte de largura fixa, semelhante ao elemento `<code>`.

```
<pre>
field() {
  shift $1 ; echo $1
}
</pre>
```

```
field() {
  shift $1 ; echo $1
}
```

Figure 20. Representação do código HTML em um navegador, ilustrando como o elemento `<pre>` preserva os espaços em branco.

Agrupando elementos

Por convenção, os elementos HTML são divididos em duas categorias:

Elementos de bloco

Aparecem em uma nova linha e ocupam toda a largura disponível. Alguns exemplos já discutidos são `<p>`, `<ol>` e `<h2>`.

Elementos de linha

Aparecem na mesma linha que outros elementos de texto, ocupando apenas o espaço necessário ao conteúdo. Dentre esses elementos temos `<strong>`, `<em>` e `<i>`.

NOTE

O HTML5 introduziu categorias de elementos mais precisas para evitar confusões com os blocos CSS. Para simplificar, nos limitaremos aqui à subdivisão convencional em elementos de bloco e de linha.

Os elementos fundamentais para agrupar diversos elementos juntos são `<div>` e `<span>`.

O elemento `<div>` é um contêiner em nível de bloco para outros elementos HTML e não adiciona valor semântico por si só. Este elemento serve para dividir um documento HTML em seções e estruturar seu conteúdo, tanto para melhorar a legibilidade do código quanto para aplicar estilos CSS a um grupo de elementos, como veremos em uma lição posterior.

Por padrão, os navegadores web sempre inserem uma quebra de linha antes e depois de cada elemento `<div>` para que sejam exibidos em sua própria linha.

Por sua vez, o elemento `<span>` é usado como um envoltório para texto em HTML e geralmente serve para agrupar outros elementos de linha para permitir a aplicação de estilos em uma parte menor do texto usando CSS.

O elemento `<span>` se comporta como um texto normal e não inicia uma nova linha. Portanto, trata-se de um elemento de linha.

O exemplo a seguir compara a representação visual do elemento semântico `<p>` e os elementos de agrupamento `<div>` e `<span>`:

```
<p>Text within a paragraph</p>
<p>Another paragraph of text</p>
<hr>
<div>Text wrapped within a <code>div</code> element</div>
<div>Another <code>div</code> element with more text</div>
<hr>
<span>Span content</span>
<span>and more span content</span>
```

Um navegador web exibiria este código como mostrado na Figure 21.

Figure 21. Representação do documento de teste em um navegador, ilustrando as diferenças entre os elementos parágrafo, div e span em HTML.

Já vimos que, por padrão, o navegador adiciona espaços antes e depois dos elementos `<p>`. Esses espaços não são aplicados aos elementos de agrupamento `<div>` e `<span>`. Entretanto, os elementos `<div>` são formatados como blocos independentes, ao passo que o texto nos elementos `<span>` são mostrados na mesma linha.

Estrutura da página HTML

Já falamos de como como usar elementos HTML para descrever o conteúdo de uma página web

semanticamente — em outras palavras, para transmitir significado e contexto ao texto. Outro grupo de elementos tem o propósito de descrever a *estrutura semântica* de uma página web, uma expressão ou sua estrutura. Esses são os elementos de bloco, ou seja, elementos que se comportam visualmente de forma semelhante a um elemento `<div>`. Sua finalidade é definir a estrutura semântica de uma página web especificando áreas bem definidas como cabeçalhos, rodapés e o conteúdo principal da página. Esses elementos permitem o agrupamento semântico do conteúdo de forma que ele também possa ser entendido por um computador, incluindo mecanismos de busca e leitores de tela.

O elemento `<header>`

O elemento `<header>` (cabeçalho) contém informações introdutórias ao elemento semântico circundante dentro de um documento HTML. Um cabeçalho é diferente de um título, mas um cabeçalho geralmente inclui um elemento de título (`<h1>`, ... , `<h6>`).

Na prática, esse elemento é mais frequentemente usado para representar o cabeçalho da página, como um banner com um logotipo. Também pode ser usado para introduzir o conteúdo dos seguintes elementos: `<body>`, `<section>`, `<article>`, `<nav>` ou `<aside>`.

Um mesmo documento pode ter múltiplos elementos `<header>`, mas um elemento `<header>` não pode ser aninhado dentro de outro elemento `<header>`. Um elemento `<footer>` também não pode ser usado dentro de um elemento `<header>`.

Por exemplo, para adicionar um cabeçalho ao nosso documento de exemplo, podemos fazer o seguinte:

```
<header>
  <h1>Garden Party</h1>
</header>
```

Não haverá mudanças visíveis no documento HTML, já que `<h1>` (como todos os outros elementos de título) é um elemento de nível de bloco sem outras propriedades visuais.

O elemento de conteúdo `<main>`

O elemento `<main>` é o envoltório para o conteúdo central de uma página web. Não é possível haver mais de um elemento `<main>` em um documento HTML.

Em nosso documento de exemplo, todo o código HTML que escrevemos até agora teria sido posto dentro do elemento `<main>`.

```
<main>
  <header>
    <h1>Garden Party</h1>
  </header>
  <p>
    Invitation to <strong>John's garden party</strong>.<br>
    <strong>Saturday, next week.</strong>
  </p>

  <h2>Agenda</h2>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>

  <hr>

  <h2>Please bring</h2>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</main>
```

Como no caso do elemento `<header>`, o elemento `<main>` não causa nenhuma mudança visual em nosso exemplo.

O elemento `<footer>`

O elemento `<footer>` contém notas de rodapé, como informações sobre a autoria, informações de contato ou documentos relacionados ao elemento semântico circundante, por exemplo `<section>`,

`<nav>` ou `<aside>`. Um documento pode incluir diversos elementos `<footer>` que permitem descrever melhor os elementos semânticos. Todavia, um elemento `<footer>` não pode ser aninhado dentro de outro elemento `<footer>`, nem um elemento `<header>` pode ser usado dentro de um `<footer>`.

Em nosso exemplo, podemos adicionar as informações de contato do host (John), como mostrado no exemplo a seguir:

```
<footer>
  <p>John Doe</p>
  <p>john.doe@example.com</p>
</footer>
```

O elemento `<nav>`

O elemento `<nav>` descreve uma unidade de navegação importante, como um menu, que contém uma série de hiperlinks.

NOTE Nem todos os hiperlinks devem ser postos dentro de um elemento `<nav>`. Ele é útil para listar um grupo de links.

Como os hiperlinks ainda não foram abordados, o elemento de navegação não será incluído nos exemplos desta lição.

O elemento `<aside>`

O elemento `<aside>` é um envoltório para conteúdos que não são necessários na organização do conteúdo da página principal, mas que geralmente estão indiretamente relacionados ou são suplementares. Este elemento é freqüentemente usado para barras laterais que exibem informações secundárias, como um glossário.

Em nosso exemplo, podemos adicionar informações de endereço e rota, que são apenas indiretamente relacionados ao resto do conteúdo, usando o elemento `<aside>`.

```
<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>
```

O elemento `<section>`

O elemento `<section>` define uma seção lógica de um documento que faz parte do elemento semântico circundante, mas que não funcionaria como conteúdo autônomo, como um capítulo.

Em nosso documento de exemplo, podemos agrupar as seções de conteúdo da programação e incluir seções de listagem, como mostrado no exemplo a seguir:

```
<section>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</section>

<hr>

<section>
  <header>
    <h2>Please bring</h2>
  </header>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</section>
```

Este exemplo também acrescenta mais elementos `<header>` dentro das seções, de modo que cada

seção está inserida em seu próprio elemento `<header>`.

O elemento `<article>`

O elemento `<article>` define um conteúdo independente e autônomo que faz sentido por si só, sem o resto da página. Seu conteúdo é potencialmente redistribuível ou reutilizável em outro contexto. Exemplos típicos ou materiais apropriados para um elemento `<article>` seriam uma postagem de blog, uma lista de produtos de uma loja ou o anúncio de um produto. O anúncio poderia então existir tanto por conta própria quanto em uma página maior.

Em nosso exemplo, podemos substituir a primeira `<section>` que envolve a programação por um elemento `<article>`.

```
<article>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</article>
```

O elemento `<header>` que adicionamos no exemplo anterior também pode persistir aqui, uma vez que os elementos `<article>` também podem ter seus próprios elementos `<header>`.

O exemplo final

Se combinarmos todos os exemplos anteriores, o documento HTML final de nosso convite fica assim:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <title>Garden Party</title>
  </head>

  <body>
    <main>
      <h1>Garden Party</h1>
      <p>
        Invitation to <strong>John's garden party</strong>.<br>
        <strong>Saturday, next week.</strong>
      </p>

      <article>
        <h2>Agenda</h2>
        <ol>
          <li>Welcome</li>
          <li>
            Barbecue
            <ul>
              <li><em>Vegetables</em></li>
              <li><em>Meat</em></li>
              <li><em>Burgers</em>, including vegetarian options</li>
            </ul>
          </li>
          <li>Dessert</li>
          <li><mark>Fireworks</mark></li>
        </ol>
      </article>

      <hr>

      <section>
        <h2>Please bring</h2>
        <ul>
          <li>Salad</li>
          <li>Drinks</li>
          <li>Bread</li>
          <li>Snacks</li>
          <li>Desserts</li>
        </ul>
      </section>
    </main>

    <aside>
      <p>
        10, Main Street<br>
```

```
        Newville
      </p>
      <p>Parking spaces available.</p>
    </aside>

    <footer>
      <p>John Doe</p>
      <p>john.doe@example.com</p>
    </footer>
  </body>
</html>
```

Em um navegador web, a página inteira é exibida como na Figure 22.

Garden Party

Invitation to **John's garden party**.
Saturday, next week.

Agenda

1. Welcome
2. Barbecue
    - *Vegetables*
    - *Meat*
    - *Burgers*, including vegetarian options
3. Dessert
4. Fireworks

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

10, Main Street
Newville

Parking spaces available.

John Doe

john.doe@example.com

Figure 22. Representação do documento HTML resultante em um navegador web, combinando todos os exemplos anteriores. A página representa um convite para uma festa ao ar livre, com a programação do evento e uma lista de coisas para os convidados levarem.

Exercícios Guiados

1. Para cada uma das seguintes tags, indique a tag de fechamento correspondente:

`<h5>`	
` `	
`<ol>`	
`<dd>`	
`<hr>`	
`<strong>`	
`<tt>`	
`<main>`	

2. Para cada uma das seguintes tags, indique se ela marca o início de um elemento de bloco ou de linha:

`<h3>`	
`<span>`	
`<b>`	
`<div>`	
`<em>`	
`<dl>`	
`<li>`	
`<nav>`	
`<code>`	
`<pre>`	

3. Que tipo de lista podemos criar em HTML? Quais tags devemos usar para cada um deles?

4. Quais tags encerram os elementos de bloco que podem ser usados para estruturar uma página HTML?

Exercícios Exploratórios

1. Crie uma página HTML básica com o título “Regras do Formulário”. Você usará essa página HTML para todos os exercícios exploratórios, cada um baseado nos anteriores. Em seguida, adicione um título de nível 1 com o texto “Como preencher o formulário de solicitação”, um parágrafo com o texto “Para receber o documento PDF com o curso de HTML completo, é necessário preencher os seguintes campos:” e uma lista não ordenada com os seguintes campos: “Nome”, “Sobrenome”, “Endereço de email”, “País”, “Estado” e “CEP/Código Postal”.

2. Coloque os três primeiros campos (“Nome”, “Sobrenome” e “Endereço de email”) em negrito, adicionando também uma importância semântica. Depois inclua um título de nível 2 com o texto “Campos obrigatórios” e um parágrafo com o texto “Os campos em negrito são obrigatórios.”

3. Adicione outro título de nível 2 com o texto “Passos a seguir”, um parágrafo com o texto “Há quatro etapas a seguir:” e uma lista ordenada com os seguintes itens: “Preencha os campos”, “Clique no botão Enviar”, “Verifique seu email e confirme sua solicitação clicando no link recebido” e “Verifique seu email - Você receberá o curso completo de HTML em alguns minutos”.

4. Usando `<div>`, crie um bloco para cada seção que começa com um título de nível 2.

5. Usando `<div>`, crie outro bloco para a seção começando com o título de nível 1. Em seguida, separe esta seção das outras duas com uma linha horizontal.

6. Adicione o elemento de cabeçalho com o texto “Regras do formulário - 2021” e o elemento de rodapé com o texto “Copyright - 2021”. Finalmente, adicione o elemento principal, que deve conter os três blocos `<div>`.

Resumo

Nesta lição, você aprendeu:

- Como criar marcações para o conteúdo de um documento HTML
- A estrutura hierárquica do texto em HTML
- A diferença entre elementos HTML de bloco e de linha
- Como criar documentos HTML com uma estrutura semântica

Os seguintes termos foram discutidos nesta lição:

`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
: As tags de título.

`<p>`
: A tag de parágrafo.

`<ol>`
: A tag de lista ordenada.

`<ul>`
: A tag de lista não ordenada.

`<li>`
: A tag de item da lista.

`<dl>`
: A tag da lista de definição.

`<dt>`, `<dd>`
: As tags de termo e descrição para uma lista de definição.

`<pre>`
: A tag de preservação da formatação.

`<b>`, `<i>`, `<u>`, `<tt>`, `<em>`, `<strong>`, `<code>`, `<mark>`
: As tags de formatação.

`<div>`, `<span>`

As tags de agrupamento.

`<header>`, `<main>`, `<nav>`, `<aside>`, `<footer>`

As tags usadas para criar uma estrutura e layout simples para uma página.

Respostas aos Exercícios Guiados

1. Para cada uma das seguintes tags, indique a tag de fechamento correspondente:

`<h5>`	`</h5>`
` `	Não existe
`<ol>`	`</ol>`
`<dd>`	`</dd>`
`<hr>`	Não existe
`<strong>`	`</strong>`
`<tt>`	`</tt>`
`<main>`	`</main>`

2. Para cada uma das seguintes tags, indique se ela marca o início de um elemento de bloco ou de linha:

`<h3>`	Elemento de bloco
`<span>`	Elemento de linha
`<b>`	Elemento de linha
`<div>`	Elemento de bloco
`<em>`	Elemento de linha
`<dl>`	Elemento de bloco
`<li>`	Elemento de bloco
`<nav>`	Elemento de bloco
`<code>`	Elemento de linha
`<pre>`	Elemento de bloco

3. Que tipo de lista podemos criar em HTML? Quais tags devemos usar para cada um deles?

   Em HTML, existem três tipos de listas: listas ordenadas, que consistem em uma série de itens de lista numerados, listas não ordenadas, que consistem em uma série de itens de lista que não têm uma ordem ou sequência especial, e listas de definição, que representam verbetes, como em um dicionário ou enciclopédia. As listas ordenadas são postas entre as tags `<ol>` e `</ol>`, uma lista não ordenada entre as tags `<ul>` e `</ul>` e uma lista de definição entre `<dl>` e `</dl>`. Cada item de uma lista ordenada ou não ordenada é posto entre as tags `<li>` e `</li>`. Os termos da lista de

definição ficam entre as tags `<dt>` e `</dt>` e as descrições entre `<dd>` e `</dd>`.

4. Quais tags encerram os elementos de bloco que podem ser usados para estruturar uma página HTML?

   As tags `<header>` e `</header>` encerram o cabeçalho da página, `<main>` e `</main>` contêm o conteúdo principal da página HTML, as tags `<nav>` e `</nav>` contêm o que chamamos de barra de navegação, `<aside>` e `</aside>` incluem a barra lateral e as tags `<footer>` e `</footer>` encerram o rodapé da página.

Respostas aos Exercícios Exploratórios

1. Crie uma página HTML básica com o título “Regras do Formulário”. Você usará essa página HTML para todos os exercícios exploratórios, cada um baseado nos anteriores. Em seguida, adicione um título de nível 1 com o texto “Como preencher o formulário de solicitação”, um parágrafo com o texto “Para receber o documento PDF com o curso de HTML completo, é necessário preencher os seguintes campos:” e uma lista não ordenada com os seguintes campos: “Nome”, “Sobrenome”, “Endereço de email”, “País”, “Estado” e “CEP/Código Postal”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>Como preencher o formulário de solicitação</h1>
    <p>
      Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
    </p>
    <ul>
      <li>Nome</li>
      <li>Sobrenome</li>
      <li>Endereço de email</li>
      <li>País</li>
      <li>Estado</li>
      <li>CEP/Código Postal</li>
    </ul>
  </body>
</html>

. Coloque os três primeiros campos ("`Nome`", "`Sobrenome`" e "`Endereço de
email`") em negrito, adicionando também uma importância semântica. Depois
inclua um título de nível 2 com o texto "`Campos obrigatórios`" e um parágrafo
com o texto "`Os campos em negrito são obrigatórios.`"
+
[source,html]
```

<!DOCTYPE html> <html> <head> <title>Form Rules</title> </head>

```
<body>
  <h1>Como preencher o formulário de solicitação</h1>
  <p>
    Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
  </p>
  <ul>
    <li><strong> Nome </strong></li>
    <li><strong> Sobrenome </strong></li>
    <li><strong> Endereço de email </strong></li>
   <li>País</li>
    <li>Estado</li>
    <li>CEP/Código Postal</li>
  </ul>
```

```
    <h2>Campos obrigatórios</h2>
    <p>Os campos em negrito são obrigatórios.</p>
  </body>
</html>
```

2. AAdicione outro título de nível 2 com o texto “Passos a seguir”, um parágrafo com o texto “Há quatro etapas a seguir:” e uma lista ordenada com os seguintes itens: “Preencha os campos”, “Clique no botão enviar”, “Verifique seu email e confirme sua solicitação clicando no link recebido” e “Verifique seu email - Você receberá o curso completo de HTML em alguns minutos”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>Como preencher o formulário de solicitação</h1>
    <p>
      Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
    </p>
    <ul>
      <li><strong> Nome </strong></li>
      <li><strong> Sobrenome </strong></li>
      <li><strong> Endereço de email </strong></li>
     <li>País</li>
      <li>Estado</li>
      <li>CEP/Código Postal</li>
    </ul>

    <h2>Campos obrigatórios</h2>
    <p>Os campos em negrito são obrigatórios.</p>

    <h2>Passos a seguir</h2>
    <p>Há quatro etapas a seguir:</p>
    <ol>
      <li>Preencha os campos</li>
      <li>Clique no botão Enviar</li>
      <li>
        Verifique seu email e confirme sua solicitação clicando no link
recebido
      </li>
      <li>
        Verifique seu email – Você receberá o curso completo de HTML em alguns
minutos
      </li>
    </ol>
  </body>
</html>
```

3. Usando `<div>`, crie um bloco para cada seção que começa com um título de nível 2.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>Como preencher o formulário de solicitação</h1>
    <p>
      Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
    </p>
    <ul>
      <li><strong> Nome </strong></li>
      <li><strong> Sobrenome </strong></li>
      <li><strong> Endereço de email </strong></li>
     <li>País</li>
      <li>Estado</li>
      <li>CEP/Código Postal</li>
    </ul>

    <div>
      <h2>Campos obrigatórios</h2>
      <p>Os campos em negrito são obrigatórios.</p>
    </div>

    <div>
      <h2>Passos a seguir</h2>
    <p>Há quatro etapas a seguir:</p>
      <ol>
        <li>Preencha os campos</li>
        <li>Clique no botão Enviar</li>
        <li>
          Verifique seu email e confirme sua solicitação clicando no link
recebido
        </li>
        <li>
          Verifique seu email – Você receberá o curso completo de HTML em
alguns minutos
        </li>
      </ol>
    </div>
  </body>
</html>
```

4. Usando `<div>`, crie outro bloco para a seção começando com o título de nível 1. Em seguida, separe esta seção das outras duas com uma linha horizontal.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <div>
      <h1>Como preencher o formulário de solicitação</h1>
    <p>
      Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
      </p>
      <ul>
      <li><strong> Nome </strong></li>
      <li><strong> Sobrenome </strong></li>
      <li><strong> Endereço de email </strong></li>
     <li>País</li>
      <li>Estado</li>
      <li>CEP/Código Postal</li>
      </ul>
    </div>

    <hr>

    <div>
      <h2>Campos obrigatórios</h2>
      <p>Os campos em negrito são obrigatórios.</p>
    </div>

    <div>
      <h2>Passos a seguir</h2>
    <p>Há quatro etapas a seguir:</p>
      <ol>
        <li>Preencha os campos</li>
        <li>Clique no botão Enviar</li>
        <li>
          Verifique seu email e confirme sua solicitação clicando no link
recebido
        </li>
        <li>
```

```
          Verifique seu email – Você receberá o curso completo de HTML em
alguns minutos
        </li>
      </ol>
    </div>
  </body>
</html>
```

5. Adicione o elemento de cabeçalho com o texto "Regras do formulário - 2021" e o elemento de rodapé com o texto "Copyright - 2021". Finalmente, adicione o elemento principal, que deve conter os três blocos `<div>`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <header>
      <h1>Regras do Formulário – 2021</h1>
    </header>

    <main>
      <div>
        <h1>Como preencher o formulário de solicitação</h1>
    <p>
      Para receber o documento PDF com o curso de HTML completo, é necessário
preencher os seguintes campos:
        </p>
        <ul>
      <li><strong> Nome </strong></li>
      <li><strong> Sobrenome </strong></li>
      <li><strong> Endereço de email </strong></li>
     <li>País</li>
      <li>Estado</li>
      <li>CEP/Código Postal</li>
        </ul>
      </div>

      <hr>

      <div>
        <h2>Campos obrigatórios</h2>
```

```
    <p>Os campos em negrito são obrigatórios.</p>
      </div>

      <div>
        <h2>Passos a seguir</h2>
    <p>Há quatro etapas a seguir:</p>
      <ol>
        <li>Preencha os campos</li>
      <li>Clique no botão Enviar</li>
        <li>
          Verifique seu email e confirme sua solicitação clicando no link
recebido
        </li>
        <li>
          Verifique seu email – Você receberá o curso completo de HTML em
alguns minutos
        </li>
      </ol>
      </div>
    </main>

    <footer>
      <p>Copyright – 2021</p>
    </footer>
  </body>
</html>
```

032.3 Referências e recursos incorporados do HTML

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 032.3

Peso

2

Áreas chave de conhecimento

- Criar links para recursos externos e âncoras de página
- Adicionar imagens a documentos HTML
- Entender as principais propriedades dos formatos de arquivo de mídia comuns, incluindo PNG, JPG e SVG
- Noções de iframes

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- Atributo `id`
- `<a>`, incluindo os atributos `href` e `target` (`_blank`, `_self`, `_parent`, `_top`)
- `<img>`, incluindo os atributos `src` e `alt`

032.3 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	032 Marcação de documentos HTML
Objetivo:	032.3 Referências e recursos incorporados do HTML
Lição:	1 de 1

Introdução

As páginas web modernas raramente são constituídas somente de texto. Elas incluem muitos outros tipos de conteúdo, como imagens, áudio, vídeo e até outros documentos HTML. Junto com o conteúdo externo, os documentos HTML podem conter links para outros documentos, o que torna a experiência de navegação na Internet muito mais simples.

Conteúdo incorporado

A troca de arquivos pela Internet é possível sem páginas web escritas em HTML. Nesse caso, por que o HTML é o formato de preferência para documentos web, e não o PDF ou qualquer outro formato de processamento de texto? Um motivo importante é que o HTML mantém seus recursos multimídia em arquivos separados. Em um ambiente como a Internet, onde as informações geralmente são redundantes e distribuídas em locais diferentes, é importante evitar transferências de dados desnecessárias. Na maioria das vezes, as novas versões de uma página web puxam as mesmas imagens e outros arquivos de suporte das versões anteriores, de modo que o navegador pode usar os arquivos reunidos anteriormente em vez de copiar tudo outra vez. Além disso, a separação dos arquivos facilita a personalização do conteúdo multimídia de acordo com as características do

cliente, como localidade, tamanho da tela e velocidade de conexão.

Imagens

O tipo mais comum de conteúdo incorporado são as imagens que acompanham o texto. As imagens são salvas separadamente e referenciadas dentro do arquivo HTML com a tag `<img>`:

```
<img src="logo.png">
```

A tag `<img>` não requer uma tag de fechamento. A propriedade `src` indica o local de origem do arquivo de imagem. Neste exemplo, o arquivo de imagem `logo.png` deve estar localizado no mesmo diretório do arquivo HTML, caso contrário o navegador não poderá exibi-lo. A propriedade de localização de origem aceita caminhos relativos, de forma que a *notação de ponto* pode ser usada para indicar o caminho até a imagem:

```
<img src="../logo.png">
```

Os dois pontos indicam que a imagem está localizada dentro do diretório pai em relação ao diretório onde está o arquivo HTML. Se o nome do arquivo `../logo.png` for usado dentro de um arquivo HTML cuja URL é `http://example.com/library/periodicals/index.html`, o navegador solicitará o arquivo de imagem no endereço `http://example.com/library/logo.png`.

A notação de ponto também se aplica se o arquivo HTML não for um arquivo real no sistema de arquivos; o navegador HTML interpreta a URL como se fosse um caminho para um arquivo, mas é função do servidor HTTP decidir se esse caminho se refere a um arquivo ou a um conteúdo gerado dinamicamente. O domínio e o caminho correto são adicionados automaticamente a todas as solicitações ao servidor, caso o arquivo HTML venha de uma solicitação HTTP. Da mesma forma, o navegador abrirá a imagem adequada se o arquivo HTML vier diretamente do sistema de arquivos local.

Os locais de origem que começam com uma barra `/` são tratados como caminhos absolutos. Os caminhos absolutos incluem informações completas sobre os locais da imagem e, portanto, funcionam independentemente da localização do documento HTML. Se o arquivo de imagem estiver localizado em outro servidor, o que será o caso quando uma *rede de distribuição de conteúdo* (Content Delivery Network ou CDN) for usada, o nome de domínio também deve ser incluído.

NOTE

As redes de distribuição de conteúdo são compostas por servidores distribuídos geograficamente que armazenam conteúdo estático para outros sites. Eles ajudam a melhorar o desempenho e a disponibilidade dos sites com um grande número de acessos.

Se a imagem não puder ser carregada, o navegador HTML mostrará o texto fornecido no atributo `alt` em vez da imagem. Por exemplo:

```
<img src="logo.png" alt="The Company logo">
```

O atributo `alt` também é importante para a acessibilidade. Os navegadores em modo texto e os leitores de tela usam-no como uma descrição para a imagem correspondente.

Tipos de imagem

Os navegadores podem exibir todos os tipos mais populares de imagem, como JPEG, PNG, GIF e SVG. As dimensões das imagens são detectadas no momento em que elas são carregadas, mas também podem ser predefinidas com os atributos `width` e `height` (largura e altura):

```
<img src="logo.png" alt="The Company logo" width="300" height="100">
```

A única razão para incluir atributos de dimensão à tag `<img>` é evitar quebrar o layout quando a imagem demorar muito para carregar ou quando não puder ser carregada. O uso dos atributos `width` e `height` para alterar as dimensões originais da imagem pode ter resultados indesejáveis:

- As imagens serão distorcidas se o tamanho original for menor do que as novas dimensões ou se a proporção definida for diferente da original.
- Ao se reduzir o tamanho de imagens grandes, é necessária uma maior largura de banda, o que resulta em tempos de carregamento mais longos.

O SVG é o único formato que não sofre com esses efeitos, porque todas as suas informações gráficas são armazenadas em coordenadas numéricas adequadas para o redimensionamento e suas dimensões não afetam o tamanho do arquivo (daí o nome *Scalable Vector Graphics*). Por exemplo, para desenhar um retângulo em SVG, são necessárias apenas a posição, as dimensões das arestas e as informações de cor. O valor específico de cada pixel será renderizado dinamicamente posteriormente. Na verdade, as imagens SVG são semelhantes aos arquivos HTML, no sentido de que seus elementos gráficos também são definidos por tags em um arquivo de texto. Os arquivos SVG destinam-se a representar desenhos com arestas bem definidas, como gráficos ou diagramas.

As imagens que não se enquadram nesses critérios devem ser armazenadas como *bitmaps*. Ao contrário dos formatos vetoriais de imagem, os bitmaps armazenam informações de cor de antemão para cada pixel da imagem. O armazenamento do valor cromático de cada pixel da imagem gera uma grande quantidade de dados e, por isso, os bitmaps geralmente são armazenados em formatos compactados, como JPEG, PNG ou GIF.

O formato JPEG é recomendado para fotografias, porque seu algoritmo de compactação produz bons resultados para as sombras e fundos desfocados. Para as imagens em que prevalecem as cores sólidas, o formato PNG é o mais apropriado. Portanto, o formato PNG deve ser escolhido quando for necessário converter uma imagem vetorial em bitmap.

O formato GIF oferece a qualidade de imagem mais baixa dentre todos os formatos de bitmap populares. No entanto, ele ainda é amplamente utilizado devido ao seu suporte para animações. De fato, muitos sites empregam arquivos GIF para exibir vídeos curtos, mas existem maneiras melhores de exibir conteúdo de vídeo.

Áudio e vídeo

Podemos adicionar conteúdos de áudio e vídeo a um documento HTML mais ou menos da mesma maneira que as imagens. Sem surpresa, a tag para adicionar áudio é `<audio>` e a tag para adicionar vídeo é `<video>`. Obviamente, os navegadores em modo texto não são capazes de reproduzir conteúdo multimídia, de forma que as tags `<audio>` e `<video>` empregam uma tag de fechamento para conter o texto usado como explicação para o elemento que não pôde ser mostrado. Por exemplo:

```
<audio controls src="/media/recording.mp3">
<p>Unable to play <em>recording.mp3</em></p>
</audio>
```

Se o navegador não suportar a tag `<audio>`, a linha "Unable to play recording.mp3" será mostrada em seu lugar. O uso das tags de fechamento `</audio>` ou `</video>` permitem que a página web inclua conteúdos alternativos mais elaborados do que a simples linha de texto permitida pelo atributo `alt` da tag `<img>`.

O atributo `src` das tags `<audio>` e `<video>` funcionam da mesma forma que em `<img>`, mas ele também aceita URLs apontando para uma transmissão ao vivo. O navegador cuida de armazenar em buffer, decodificar e exibir o conteúdo conforme ele é recebido. O atributo `controls` exibe os controles de reprodução. Sem ele, o visitante não poderá pausar, retroceder ou controlar a reprodução de qualquer outra maneira.

Conteúdo genérico

Um documento HTML pode ser aninhado em outro documento HTML, de forma semelhante à inserção de uma imagem em um documento HTML, mas usando a tag `<iframe>`:

```
<iframe name="viewer" src="gallery.html">
<p>Unsupported browser</p>
</iframe>
```

Os navegadores em modo texto mais simples não suportam a tag `<iframe>`, exibindo em seu lugar o texto incluído. Como acontece com as tags de multimídia, o atributo `src` define a localização de origem do documento aninhado. É possível incluir atributos `width` e `height` (largura e altura) para alterar as dimensões padrão do elemento `iframe`.

O atributo `name` permite fazer referência ao iframe e trocar o documento aninhado. Sem este atributo, o documento aninhado não pode ser trocado. Um elemento `anchor` pode ser usado para carregar um documento de outro local dentro de um iframe em vez da janela atual do navegador.

Links

O elemento de página comumente chamado de *link* da web também é conhecido pelo termo técnico *âncora* (anchor), o que explica o uso da tag `<a>`. A âncora leva a outro local, que pode ser qualquer endereço compatível com o navegador. A localização é indicada pelo atributo `href` (*hyperlink reference*):

```
<a href="contact.html">Contact Information</a>
```

A localização pode ser informada como um caminho relativo ou absoluto, como acontece com o conteúdo incorporado de que falamos anteriormente. Somente o conteúdo de texto entre as tags (por exemplo, `Contact Information`) fica visível para o visitante, geralmente na forma de um texto em azul sublinhado e clicável, mas o item que contém o link também pode ser qualquer outro conteúdo visível, como imagens:

```
<a href="contact.html"><img src="contact.png" alt="Contact Information"></a>
```

Prefixos especiais podem ser adicionados ao caminho para informar ao navegador como abri-lo. Se a âncora apontar para um endereço de email, por exemplo, seu atributo `href` deve incluir o prefixo `mailto:`:

```
<a href="mailto:info@lpi.org">Contact by email</a>
```

O prefixo `tel:` indica um número de telefone. Essa indicação é particularmente útil para visitantes que visualizam a página em dispositivos móveis:

```
<a href="tel:+123456789">Contact by phone</a>
```

Quando o link é clicado, o navegador abre o conteúdo do caminho com o aplicativo associado.

O uso mais comum das âncoras é carregar outros documentos da web. Por padrão, o navegador substitui o documento HTML atual pelo conteúdo do novo local. Esse comportamento pode ser modificado usando o atributo `target`. O destino `_blank`, por exemplo, diz ao navegador para abrir o local fornecido em uma nova janela ou nova guia do navegador, dependendo das preferências do visitante:

```
<a href="contact.html" target="_blank">Contact Information</a>
```

O destino `_self` é o padrão quando o atributo `target` não é fornecido. Ele faz com que o documento referenciado substitua o documento atual.

Outros tipos de destinos estão relacionados ao elemento `<iframe>`. Para carregar um documento referenciado dentro de um elemento `<iframe>`, o atributo `target` deve apontar para o nome do elemento iframe:

```
<p><a href="gallery.html" target="viewer">Photo Gallery</a></p>

<iframe name="viewer" width="800" height="600">
<p>Unsupported browser</p>
</iframe>
```

O elemento iframe funciona como uma janela distinta do navegador e, portanto, quaisquer links carregados a partir do documento dentro do iframe substituirão apenas o conteúdo do quadro. Para mudar esse comportamento, os elementos âncora dentro do documento aninhado também podem usar o atributo `target`. O destino `_parent`, quando usado dentro de um documento aninhado, fará com que o local referenciado substitua o documento pai que contém a tag `<iframe>`. Por exemplo, o documento `gallery.html` incorporado pode conter uma âncora que carrega a si mesma e substitui o documento pai:

```
<p><a href="gallery.html" target="_parent">Open as parent document</a></p>
```

Os documentos HTML suportam vários níveis de aninhamento com a tag `<iframe>`. O destino `_top`, quando usado em uma âncora dentro de um documento aninhado, fará com que o local referenciado substitua o documento principal na janela do navegador, independentemente de ele ser o pai imediato do `<iframe>` ou um ancestral mais distante.

Locais dentro de documentos

O endereço de um documento HTML pode conter opcionalmente um *fragmento* que pode ser usado para identificar um recurso dentro do documento. Este fragmento, também conhecido como *âncora de URL*, é uma string iniciada por uma cerquilha `#` no final da URL. Por exemplo, a palavra `History` é a âncora da URL `https://en.wikipedia.org/wiki/Internet#History`.

Quando a URL tem uma âncora, o navegador rola a página até o elemento correspondente no documento: ou seja, o elemento cujo atributo `id` é idêntico à âncora da URL. No caso da URL de nosso exemplo, `https://en.wikipedia.org/wiki/Internet#History`, o navegador irá diretamente para a seção "History". Examinando o código HTML da página, descobrimos que o título da seção possui o atributo `id` correspondente:

```
<span class="mw-headline" id="History">History</span>
```

As âncoras de URL podem ser usadas no atributo `href` da tag `<a>`, tanto para apontar para páginas externas quanto para para locais dentro da página atual. Neste último caso, basta usar apenas a cerquilha com o fragmento da URL, como em `<a href="#History">History</a>`.

WARNING O atributo `id` não deve conter espaços em branco (espaços, tabulações etc.) e deve ser único no documento.

Existem maneiras de personalizar a forma como o navegador reage às âncoras de URL. É possível, por exemplo, escrever uma função JavaScript que escuta o evento da janela *hashchange* e dispara uma ação personalizada, como uma animação ou uma solicitação HTTP. Vale notar, porém, que o fragmento de URL nunca é enviado ao servidor com a URL e, portanto, não pode ser usado como um identificador pelo servidor HTTP.

Exercícios Guiados

1. O documento HTML localizado em `http://www.lpi.org/articles/linux/index.html` tem uma tag `<img>` cujo atributo `src` aponta para `../logo.png`. Qual o caminho absoluto completo até essa imagem?

2. Cite duas razões pelas quais o atributo `alt` é importante nas tags `<img>`.

3. Que formato de imagem oferece uma boa qualidade de imagem e mantém o tamanho do arquivo pequeno quando usado com fotografias com partes desfocadas e muitas cores e tonalidades?

4. Em vez de usar um provedor terceirizado como o Youtube, qual tag HTML permite incorporar um arquivo de vídeo em um documento HTML usando apenas recursos HTML padrão?

Exercícios Exploratórios

1. Imagine um documento HTML com o hiperlink `<a href="pic1.jpeg">First picture</a>` e o elemento iframe `<iframe name="gallery"></iframe>`. Como modificar a tag do hiperlink de forma que a imagem que ele aponta seja carregada dentro do elemento iframe dado quando o usuário clicar no link?

2. O que acontecerá quando o visitante clicar em um hiperlink em um documento dentro de um iframe e o hiperlink tiver o atributo target definido como `_self`?

3. Você percebe que a âncora de URL para a segunda seção de sua página HTML não está funcionando. Qual é a causa provável deste erro?

Resumo

Esta lição aborda a maneira de adicionar imagens e outros conteúdos multimídia usando as tags HTML adequadas. Além disso, mostramos as diferentes maneiras de usar hiperlinks para carregar outros documentos e apontar para locais específicos dentro de uma página. A lição demonstrou os seguintes conceitos e procedimentos:

- A tag `<img>` e seus principais atributos: `src` e `alt`.
- Caminhos de URL relativos e absolutos.
- Formatos de imagem populares para a web e suas características.
- As tags de multimídia `<audio>` e `<video>`.
- Como inserir documentos aninhados com a tag `<iframe>`.
- A tag de hiperlink `<a>`, seu atributo `href` e os destinos especiais.
- Como usar fragmentos de URL, também conhecidos como âncoras.

Respostas aos Exercícios Guiados

1. O documento HTML localizado em `http://www.lpi.org/articles/linux/index.html` tem uma tag `<img>` cujo atributo `src` aponta para `../logo.png`. Qual o caminho absoluto completo até essa imagem?

   `http://www.lpi.org/articles/logo.png`

2. Cite duas razões pelas quais o atributo `alt` é importante nas tags `<img>`.

   Os navegadores em modo texto poderão mostrar uma descrição da imagem ausente. Os leitores de tela usam o atributo `alt` para descrever a imagem.

3. Que formato de imagem oferece uma boa qualidade de imagem e mantém o tamanho do arquivo pequeno quando usado com fotografias com partes desfocadas e muitas cores e tonalidades?

   O formato JPEG.

4. Em vez de usar um provedor terceirizado como o Youtube, qual tag HTML permite incorporar um arquivo de vídeo em um documento HTML usando apenas recursos HTML padrão?

   A tag `<video>`.

Respostas aos Exercícios Exploratórios

1. Imagine um documento HTML com o hiperlink `<a href="pic1.jpeg">First picture</a>` e o elemento iframe `<iframe name="gallery"></iframe>`. Como modificar a tag do hiperlink de forma que a imagem que ele aponta seja carregada dentro do elemento iframe dado quando o usuário clicar no link?

   Usando o atributo `target` da tag `a`: `<a href="pic1.jpeg" target="gallery">First picture</a>`.

2. O que acontecerá quando o visitante clicar em um hiperlink em um documento dentro de um iframe e o hiperlink tiver o atributo target definido como `_self`?

   O documento será carregado dentro do mesmo iframe, o que é o comportamento padrão.

3. Você percebe que a âncora de URL para a segunda seção de sua página HTML não está funcionando. Qual é a causa provável deste erro?

   O fragmento de URL após a cerquilha não corresponde ao atributo `id` do elemento correspondente à segunda seção, ou o atributo `id` do elemento não está presente.

032.4 Formulários HTML

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 032.4

Peso

2

Áreas chave de conhecimento

- Criar formulários HTML simples
- Entender os métodos do formulário HTML
- Entender os elementos e tipos de entrada em HTML

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `<form>`, incluindo os atributos `method` (`get`, `post`), `action` e `enctype`
- `<input>`, incluindo o atributo `type` (`text`, `email`, `password`, `number`, `date`, `file`, `range`, `radio`, `checkbox`, `hidden`)
- `<button>`, incluindo o atributo `type` (`submit`, `reset`, `hidden`)
- `<textarea>`
- Atributos comuns dos elementos de formulário (`name`, `value`, `id`)
- `<label>`, incluindo o atributo `for`

Linux Professional Institute

032.4 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	032 Marcação de documentos HTML
Objetivo:	032.4 Formulários HTML
Lição:	1 de 1

Introdução

Os formulários web são uma maneira simples e eficiente de solicitar informações ao visitante de uma página HTML. O desenvolvedor front-end pode usar diversos componentes, como campos de texto, caixas de seleção, botões e muitos outros para construir interfaces que enviam dados ao servidor de forma estruturada.

Formulários HTML simples

Antes de falar do código de marcação específico para formulários, vamos começar com um documento HTML simples em branco, sem nenhum conteúdo no corpo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- The body content goes here -->

</body>
</html>
```

Salve esse código de exemplo como um arquivo de texto simples com a extensão `.html` (por exemplo, `form.html`) e use seu navegador favorito para abri-lo. Após alterá-lo, pressione o botão de recarregar no navegador para exibir as modificações.

A estrutura básica do formulário é dada pela própria tag `<form>` e seus elementos internos:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- Form to collect personal information -->

<form>

<h2>Personal Information</h2>

<p>Full name:</p>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>

</body>
</html>
```

As aspas duplas não são obrigatórias para atributos de palavra única como `type`, portanto `type=text` funciona tão bem quanto `type="text"`. O desenvolvedor pode escolher a convenção que prefere usar.

Salve o novo conteúdo e recarregue a página no navegador. Você deverá ver o resultado mostrado na Figure 23.

Figure 23. Um formulário bem básico.

A tag `<form>` em si não produz nenhum resultado perceptível na página. Os elementos dentro das tags `<form>…</form>` definirão os campos e outros auxílios visuais mostrados ao visitante.

O código de exemplo contém tanto as tags gerais do HDML (`<h2>` e `<p>`) quanto a tag `<input>`, que é específica aos formulários. Ao passo que as tags gerais podem aparecer em qualquer lugar do documento, as tags específicas aos formulários devem ser usadas somente dentro do elemento `<form>`; isto é, entre a tag de abertura `<form>` e a de fechamento `</form>`.

NOTE

O HTML inclui apenas as tags e propriedades básicas para modificar a aparência padrão dos formulários. O CSS fornece mecanismos elaborados para modificar a aparência do formulário e, portanto, a recomendação é escrever um código HTML incluindo apenas os aspectos funcionais do formulário, modificando sua aparência com CSS.

Como mostrado no exemplo, a tag de parágrafo `<p>` pode ser usada para descrever o campo para o visitante. No entanto, não há uma maneira óbvia de o navegador relacionar a descrição na tag `<p>` com o elemento correspondente de inserção de dados. A tag `<label>` é mais apropriada nesses casos (a partir de agora, considere todos os exemplos de código como estando dentro do corpo do documento HTML):

```
<form>

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>
```

O atributo `for` na tag `<label>` contém o `id` do elemento de inserção de dados correspondente. Isso torna a página mais acessível, pois os leitores de tela serão capazes de narrar o conteúdo do elemento de rótulo (label) quando o elemento de inserção de dados estiver selecionado. Além disso, os visitantes podem clicar no rótulo para selecionar o campo de inserção de dados correspondente.

O atributo `id` nos elementos de formulário tem o mesmo papel que em qualquer outro elemento do documento. Ele fornece um identificador exclusivo para o elemento. O atributo `name` tem uma finalidade semelhante, mas é usado para identificar o elemento de inserção de dados no contexto do formulário. O navegador usa o atributo `name` para identificar o campo de inserção de dados ao enviar os dados do formulário para o servidor, por isso é importante usar atributos `name` significativos e exclusivos dentro do formulário.

O atributo `type` é o principal atributo do elemento de entrada de dados, pois controla o tipo de dados que o elemento aceita e sua apresentação visual ao visitante. Se o atributo `type` não for fornecido, por padrão é exibida uma caixa de texto. Os seguintes tipos de campos de entrada de dados são suportados pelos navegadores modernos:

Table 1. Tipos de entrada em formulários

Atributo do tipo	Tipo de dados	Como é exibido
`hidden`	Uma string arbitrária	N/A
`text`	Texto sem quebra de linhas	Um controle de texto
`search`	Texto sem quebra de linhas	Um controle de pesquisa
`tel`	Texto sem quebra de linhas	Um controle de texto
`url`	Uma URL absoluta	Um controle de texto
`email`	Um endereço de email ou lista de endereços de email	Um controle de texto

Atributo do tipo	Tipo de dados	Como é exibido
`password`	Texto sem quebra de linhas (informação confidencial)	Um controle de texto que oculta a inserção de dados
`date`	Uma data (ano, mês, dia) sem fuso horário	Um controle de data
`month`	Uma data que consiste em um ano e um mês sem fuso horário	Um controle de mês
`week`	Uma data que consiste em um número de semana-ano e um número de semana sem fuso horário	Um controle de semana
`time`	Um horário (hora, minuto, segundos, frações de segundo) sem fuso horário	Um controle de horário
`datetime-local`	Uma data e hora (ano, mês, dia, hora, minuto, segundos, frações de segundo) sem fuso horário	Um controle de data e hora
`number`	Um valor numérico	Um controle de texto ou controle giratório
`range`	Um valor numérico, com a semântica extra de que o valor exato não é importante	Um controle deslizante ou semelhante
`color`	Uma cor sRGB com componentes vermelhos, verdes e azuis de 8 bits	Um seletor de cores
`checkbox`	Um conjunto de zero ou mais valores em uma lista predefinida	Uma caixa de seleção (oferece opções e permite a seleção de múltiplas opções)
`radio`	Um valor enumerado	Um botão de opção (oferece opções e permite que apenas uma seja selecionada)
`file`	Zero ou mais arquivos, cada um deles com um tipo MIME e nome de arquivo opcional	Um rótulo e um botão

Atributo do tipo	Tipo de dados	Como é exibido
`submit`	Um valor enumerado que encerra o processo de entrada de dados e envia o formulário	Um botão
`image`	Uma coordenada relativa ao tamanho de uma imagem em particular, que finaliza o processo e envia o formulário	Uma imagem clicável ou um botão
`button`	N/A	Um botão genérico
`reset`	N/A	Um botão cuja função é restaurar todos os outros campos aos seus valores iniciais

A aparência dos tipos de entrada `password`, `search`, `tel`, `url` e `email` é idêntica à do tipo `text` padrão. Sua finalidade é oferecer indicações ao navegador sobre o conteúdo esperado para esse campo de entrada de dados, de modo que o navegador ou o script em execução no lado do cliente possa realizar ações personalizadas para um tipo de entrada específico. A única diferença entre o tipo de entrada de texto e o tipo de campo de senha, por exemplo, é que o conteúdo do campo de senha não é exibido conforme o visitante digita sua senha. Nos dispositivos com tela de toque, nos quais o texto é digitado em um teclado virtual, o navegador pode exibir apenas o teclado numérico quando uma entrada do tipo `tel` é selecionada. Outra ação possível é sugerir uma lista de endereços de email conhecidos quando uma entrada do tipo `email` está no foco.

O tipo `number` também aparece como uma entrada de texto simples, mas com setas de aumentar/diminuir ao lado. Seu uso fará com que o teclado numérico apareça nos dispositivos com tela de toque quando o campo estiver selecionado.

Os outros elementos de inserção de dados têm sua própria aparência e comportamento. O tipo `date`, por exemplo, é representado de acordo com as configurações locais de formato de data e um calendário é exibido quando o campo está no foco:

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date">
</p>

</form>
```

A Figure 24 mostra como a versão para desktop do Firefox apresenta esse campo atualmente.

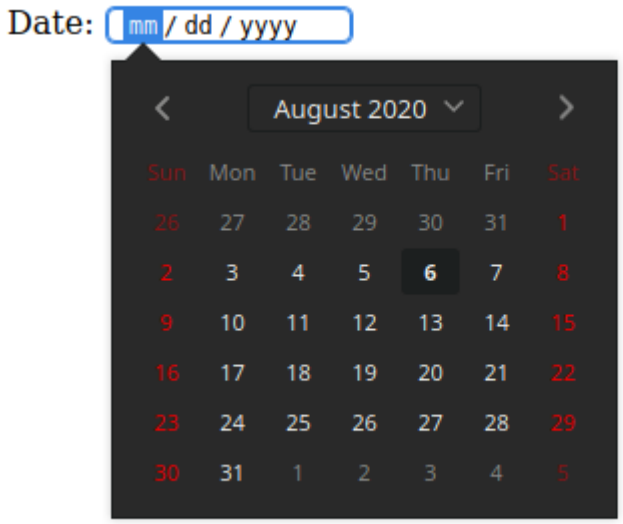

Figure 24. O tipo de entrada de data.

NOTE Os elementos podem ter uma aparência ligeiramente diferente em navegadores ou sistemas operacionais distintos, mas seu funcionamento e uso são sempre os mesmos.

Este é um recurso padrão em todos os navegadores modernos e não requer opções extras ou programação.

Independentemente do tipo de entrada, o conteúdo de um campo de inserção de dados é chamado de *valor*. Todos os valores do campo aparecem vazios por padrão, mas o atributo `value` pode ser usado para definir um valor padrão para o campo. O valor do tipo de data deve usar o formato *AAAA-MM-DD*. O valor padrão do campo de data a seguir foi definido como 6 de agosto de 2020:

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date" value="2020-08-06">
</p>

</form>
```

Os tipos de entrada específicos ajudam o visitante a preencher os campos, mas não evitam que ele

ignore as restrições e insira valores arbitrários em qualquer campo. Por isso, é importante que os valores dos campos sejam validados ao chegarem ao servidor.

Os elementos do formulário cujos valores devem ser digitados pelo visitante podem ter atributos especiais que auxiliam no preenchimento. O atributo `placeholder` (marcador de posição) insere um valor de exemplo no elemento de inserção de dados:

```
<p>Address: <input type="text" name="address" id="address" placeholder="e.g. 41
John St., Upper Suite 1"></p>
```

O marcador de posição aparece dentro do elemento de inserção de dados, conforme mostrado na Figure 25.

Figure 25. Exemplo do atributo placeholder.

Assim que o visitante começa a digitar no campo, o texto do marcador de posição desaparece. O texto do marcador de posição não é enviado como o valor do campo caso o visitante deixe o campo vazio.

O atributo `required` obriga o visitante a preencher um valor para o campo correspondente antes de enviar o formulário:

```
<p>Address: <input type="text" name="address" id="address" required
placeholder="e.g. 41 John St., Upper Suite 1"></p>
```

O atributo `required` é um atributo booleano; portanto, pode ser posto sozinho (sem o sinal de igual). É importante assinalar os campos que são obrigatórios, caso contrário o visitante não saberá quais campos estão faltando, impedindo o envio do formulário.

O atributo `autocomplete` indica se o valor do elemento de entrada pode ser preenchido automaticamente pelo navegador. Se definido como `autocomplete="off"`, o navegador não sugere valores anteriores para preencher o campo. Os elementos de inserção de dados para informações confidenciais, como números de cartão de crédito, devem ter o atributo `autocomplete` definido como `off`.

Campo de entrada para textos grandes: textarea

Ao contrário do campo de texto, onde apenas uma linha de texto pode ser inserida, o elemento `textarea` permite que o visitante insira mais de uma linha de texto. O textarea é um elemento separado, mas não é baseado no elemento input:

```
<p> <label for="comment">Type your comment here:</label> <br>

<textarea id="comment" name="comment" rows="10" cols="50">
My multi-line, plain-text comment.
</textarea>

</p>
```

A aparência típica de um textarea é mostrado na Figure 26.

Figure 26. O elemento `textarea`.

Outra diferença em relação ao elemento input é que o elemento `textarea` tem uma tag de fechamento (`</textarea>`), e assim seu conteúdo (ou seja, seu valor) fica entre as duas tags. Os atributos `rows` e `cols` (linhas e colunas) não limitam a quantidade de texto; eles são usados apenas para definir o layout. O textarea também inclui uma alça no canto inferior direito, que permite ao visitante redimensioná-lo.

Listas de opções

Diversos tipos de controles de formulário podem ser usados para apresentar uma lista de opções ao visitante: o elemento `<select>` e os tipos de entrada `radio` e `checkbox`.

O elemento `<select>` é um controle suspenso com uma lista de entradas predefinidas:

```
<p><label for="browser">Favorite Browser:</label>
<select name="browser" id="browser">
<option value="firefox">Mozilla Firefox</option>
<option value="chrome">Google Chrome</option>
<option value="opera">Opera</option>
<option value="edge">Microsoft Edge</option>
</select>
</p>
```

A tag `<option>` representa uma entrada individual no controle `<select>` correspondente. A lista completa aparece quando o visitante toca ou clica no controle, como mostrado na Figure 27.

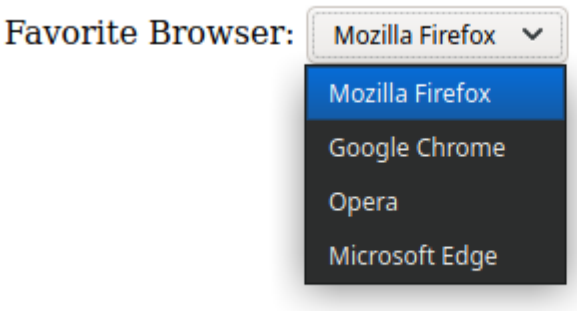

Figure 27. O elemento de formulário `select`.

A primeira entrada da lista é selecionada por padrão. Para alterar esse comportamento, você pode adicionar o atributo `selected` a outra entrada, para que ela esteja selecionada quando a página carregar.

O tipo de entrada `radio` é semelhante ao controle `<select>`, mas em vez de uma lista suspensa ele mostra todas as entradas para que o visitante selecione uma delas. Os resultados do código a seguir são mostrados na Figure 28.

```
<p>Favorite Browser:</p>

<p>
    <input type="radio" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="radio" id="browser-chrome" name="browser" value="chrome">
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="radio" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="radio" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>
```

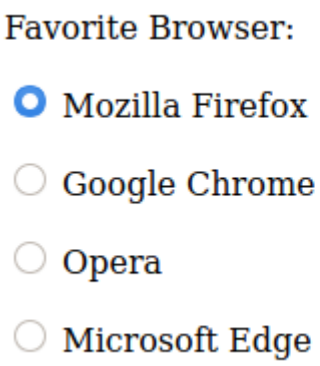

Figure 28. Elementos de entrada do tipo `radio`.

Observe que todos os tipos de entrada `radio` no mesmo grupo têm o mesmo atributo `name`. Cada um deles é exclusivo, de modo que o atributo `value` correspondente à entrada escolhida será associado ao atributo `name` compartilhado. O atributo `checked` funciona como o atributo `selected` do controle `<select>`. Ele marca a entrada correspondente quando a página é carregada pela primeira vez. A tag `<label>` é especialmente útil para as entradas de radio, porque permite ao visitante marcar uma opção clicando ou tocando no texto correspondente, além do próprio controle.

Enquanto os controles `radio` se destinam a selecionar uma única opção em uma lista, o tipo de entrada `checkbox` (caixa de seleção) permite que o visitante marque diversas opções:

```
<p>Favorite Browser:</p>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
</p>
```

As caixas de seleção também podem usar o atributo `checked` para pré-selecionar opções por padrão. Em vez dos controles redondos da entrada radio, as caixas de seleção são apresentadas como controles quadrados, como mostrado na Figure 29.

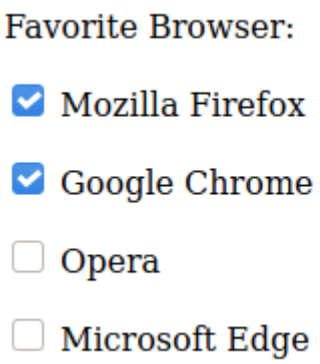

Figure 29. O tipo de entrada `checkbox`.

Se mais de uma opção for selecionada, o navegador as enviará com o mesmo nome, exigindo que o desenvolvedor de back-end escreva um código específico para ler corretamente os dados do formulário contendo caixas de seleção.

Para melhorar a usabilidade, os campos de entrada podem ser agrupados dentro de uma tag `<fieldset>`:

```
<fieldset>
<legend>Favorite Browser</legend>

<p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
</p>

<p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
</p>

<p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
</p>
</fieldset>
```

A tag `<legend>` contém o texto que é posto no alto do quadro que a tag `<fieldset>` desenha em torno dos controles (Figure 30).

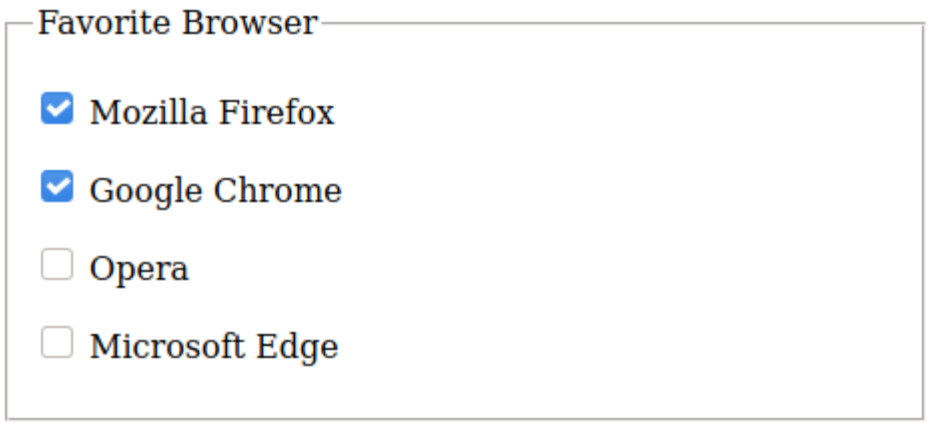

Figure 30. Agrupando elementos com a tag `fieldset`.

A tag `<fieldset>` não altera a forma como os valores dos campos são enviados ao servidor, mas permite que o desenvolvedor de front-end gerencie os controles aninhados com mais facilidade. Por exemplo, definir o atributo `disabled` em um atributo `<fieldset>` torna todos os seus elementos internos indisponíveis para o visitante.

O tipo de elemento `hidden`

Em certas situações, o desenvolvedor precisa incluir no formulário informações que não podem ser manipuladas pelo visitante, ou seja, enviar um valor escolhido pelo desenvolvedor sem que haja um campo no formulário onde o visitante possa digitar ou alterar o valor. Isso serviria, por exemplo, para incluir um token de identificação, que não precisa ser visto pelo visitante, para aquele formulário específico. Um elemento oculto de formulário seria escrito como no exemplo a seguir:

```
<input type="hidden" id="form-token" name="form-token" value="e730a375-b953-4393-
847d-2dab065bbc92">
```

O valor de um campo de entrada oculto é geralmente adicionado ao documento no lado do servidor, no momento de renderizar o documento. As entradas ocultas são tratadas como campos comuns quando o navegador as envia ao servidor, que também as lê como campos de entrada comuns.

O tipo de entrada de arquivo

Além dos dados textuais, que são digitados ou selecionados em uma lista, os formulários HTML também podem enviar arquivos arbitrários ao servidor. O tipo de entrada `file` permite ao visitante escolher um arquivo em seu sistema de arquivos local e enviá-lo diretamente pela página

```
<p>
<label for="attachment">Attachment:</label><br>
<input type="file" id="attachment" name="attachment">
</p>
```

Em vez de um campo de formulário para escrever ou selecionar um valor, o tipo de entrada `file` mostra um botão `browse` que abre uma caixa de diálogo de arquivo. Qualquer tipo de arquivo é aceito pelo tipo de entrada `file`, mas normalmente o desenvolvedor back-end restringe os tipos de arquivo permitidos e seu tamanho máximo. A verificação do tipo de arquivo também pode ser realizada no front-end, adicionando-se o atributo `accept`. Para aceitar apenas imagens JPEG e PNG, por exemplo, o atributo `accept` deve ser `accept="image/jpeg, image/png"`.

Botões de ação

Por padrão, o formulário é enviado quando o visitante pressiona a tecla Enter em qualquer campo de entrada. Para tornar as coisas mais intuitivas, um botão de envio deve ser adicionado com o tipo de entrada `submit`:

```
<input type="submit" value="Submit form">
```

O texto do atributo `value` é mostrado no botão, como visto na Figure 31.

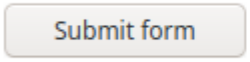

Figure 31. Um botão de envio padrão.

Outro botão útil a se incluir em formulários complexos é o botão `reset`, que limpa o formulário e o devolve ao seu estado original:

```
<input type="reset" value="Clear form">
```

Como no caso do botão de enviar, o texto do atributo `value` é usado para rotular o botão. Alternativamente, a tag `<button>` pode ser usada para adicionar botões em formulários ou em qualquer outro lugar da página. Ao contrário dos botões feitos com a tag `<input>`, o elemento de botão tem uma tag de fechamento e o rótulo do botão é seu conteúdo interno:

```
<button>Submit form</button>
```

Quando dentro de um formulário, a ação padrão do elemento `button` é enviar o formulário. Como no caso dos botões de input, o atributo do tipo button pode ser alterado para `reset`.

Ações e métodos do formulário

A última etapa ao se escrever um formulário HTML é definir como e para onde os dados devem ser enviados. Esses aspectos dependem dos detalhes do cliente e do servidor.

No lado do servidor, a abordagem mais comum é configurar um arquivo de script para analisar, validar e processar os dados do formulário de acordo com a finalidade do aplicativo. Por exemplo, o desenvolvedor back-end pode escrever um script chamado `receive_form.php` para receber os dados enviados do formulário. No lado do cliente, o script é indicado no atributo `action` da tag do formulário:

```
<form action="receive_form.php">
```

O atributo `action` segue as mesmas convenções de todos os endereços HTTP. Se o script estiver no mesmo nível de hierarquia da página que contém o formulário, ele pode ser escrito sem o caminho completo. Caso contrário, o caminho absoluto ou relativo deve ser fornecido. O script também deve gerar a resposta que servirá como página de destino, carregada pelo navegador após o visitante enviar o formulário.

O HTTP proporciona métodos distintos para enviar dados de formulário por meio de uma conexão com o servidor. Os métodos mais comuns são `get` e `post`, que devem ser indicados no atributo `method` da tag `form`:

```
<form action="receive_form.php" method="get">
```

Ou:

```
<form action="receive_form.php" method="post">
```

No método `get`, os dados do formulário são codificados diretamente na URL de solicitação. Quando o visitante envia o formulário, o navegador carrega a URL definida no atributo `action` com os campos do formulário anexados a ela.

O método `get` é preferível para pequenas quantidades de dados, como formulários de contato simples. No entanto, a URL não pode exceder alguns milhares de caracteres, de modo que o método `post` deve ser usado quando os formulários contêm campos grandes ou não textuais, como imagens.

O método `post` faz com que o navegador envie os dados do formulário na seção do corpo da solicitação HTTP. Embora necessário para dados binários que excedem o limite de tamanho de uma URL, o método `post` adiciona uma sobrecarga desnecessária à conexão quando usado com formas textuais mais simples, por isso o método `get` é preferível nesses casos.

O método escolhido não afeta a forma como o visitante interage com o formulário. Os métodos `get` e `post` são processados de forma diferente pelo script do lado do servidor que recebe o formulário.

Ao usar o método `post`, também é possível alterar o tipo MIME do conteúdo do formulário com o atributo de formulário `enctype`. Ele afeta a maneira como os campos e valores do formulário serão empilhados na comunicação HTTP com o servidor. O valor padrão para `enctype` é `application/x-www-form-urlencoded`, semelhante ao formato usado no método `get`. Se o formulário contiver campos de entrada do tipo `file`, é necessário usar o enctype `multipart/form-data`.

Exercícios Guiados

1. Qual a maneira correta de associar uma tag `<label>` a uma tag `<input>`?

2. Que tipo de elemento de entrada permite criar um controle deslizante para escolher um valor numérico?

3. Qual a finalidade do atributo de formulário `placeholder`?

4. Como selecionar por padrão a segunda opção de um controle de seleção?

Exercícios Exploratórios

1. Como alterar uma entrada de arquivo para que sejam aceitos somente arquivos PDF?

2. Como informar o visitante sobre os campos obrigatórios de um formulário?

3. Reúna três fragmentos de código desta lição em um mesmo formulário. Atenção para não usar o mesmo atributo `name` ou `id` em múltiplos controles do formulário.

Resumo

Esta lição aborda a criação de formulários HTML simples para enviar dados ao servidor. No lado do cliente, os formulários HTML consistem em elementos HTML padrão que são combinados de forma a construir interfaces personalizadas. Além disso, os formulários devem ser configurados para se comunicarem corretamente com o servidor. A lição trata dos seguintes conceitos e procedimentos:

- A tag `<form>` e a estrutura básica do formulário.
- Elementos de entrada básicos e especiais.
- O papel de tags especiais como `<label>`, `<fieldset>` e `<legend>`.
- Botões e ações do formulário.

Respostas aos Exercícios Guiados

1. Qual a maneira correta de associar uma tag `<label>` a uma tag `<input>`?

   O atributo `for` da tag `<label>` deve conter o id da tag `<input>` correspondente.

2. Que tipo de elemento de entrada permite criar um controle deslizante para escolher um valor numérico?

   O tipo de entrada `range`.

3. Qual a finalidade do atributo de formulário `placeholder`?

   O atributo `placeholder` contém um exemplo de inserção de dados pelo visitante que fica visível quando o campo de entrada correspondente está vazio.

4. Como selecionar por padrão a segunda opção de um controle de seleção?

   O segundo elemento `option` deve trazer o atributo `selected`.

Respostas aos Exercícios Exploratórios

1. Como alterar uma entrada de arquivo para que sejam aceitos somente arquivos PDF?

   O atributo `accept` do elemento de entrada deve ser configurado para `application/pdf`.

2. Como informar o visitante sobre os campos obrigatórios de um formulário?

   Os campos obrigatórios são geralmente assinalados com um asterisco (`*`), e uma breve explicação como "Os campos marcados com um * são obrigatórios" é incluída no início do formulário.

3. Reúna três fragmentos de código desta lição em um mesmo formulário. Atenção para não usar o mesmo atributo `name` ou `id` em múltiplos controles do formulário.

```
  <form action="receive_form.php" method="get">

  <h2>Personal Information</h2>

  <label for="fullname">Full name:</label>
  <p><input type="text" name="fullname" id="fullname"></p>

  <p>
    <label for="date">Date:</label>
    <input type="date" name="date" id="date">
  </p>

  <p>Favorite Browser:</p>

  <p>
    <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
    <label for="browser-firefox">Mozilla Firefox</label>
  </p>

  <p>
    <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
    <label for="browser-chrome">Google Chrome</label>
  </p>

  <p>
    <input type="checkbox" id="browser-opera" name="browser" value="opera">
    <label for="browser-opera">Opera</label>
  </p>

  <p>
    <input type="checkbox" id="browser-edge" name="browser" value="edge">
    <label for="browser-edge">Microsoft Edge</label>
  </p>

  <p><input type="reset" value="Clear form"></p>
  <p><input type="submit" value="Submit form"></p>

  </form>
```

Linux Professional Institute

Tópico 033: Estilização de conteúdo com CSS

033.1 Noções básicas de CSS

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 033.1

Peso

1

Áreas chave de conhecimento

- Incorporar CSS em um documento HTML
- Entender a sintaxe do CSS
- Adicionar comentários ao CSS
- Conhecimento dos recursos e requisitos de acessibilidade

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- Atributos HTML `style` e `type` (text/css)
- `<style>`
- `<link>`, incluindo os atributos `rel` (stylesheet), `type` (text/css) e `src`
- `;`
- `/,/`

033.1 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	033 Estilização de conteúdo com CSS
Objetivo:	033.1 Noções básicas de CSS
Lição:	1 de 1

Introdução

Todos os navegadores web exibem as páginas HTML de acordo com regras de apresentação padrão que são práticas e diretas, mas não visualmente atraentes. O HTML, por si só, oferece poucos recursos para produzir páginas elaboradas que correspondam aos conceitos modernos de experiência do usuário. Depois de criar algumas páginas HTML simples, você provavelmente percebeu que elas são feiosas quando comparadas às páginas bem projetadas que vemos comumente na Internet. Isso ocorre porque, no HTML moderno, o código de marcação destinado à estrutura e função dos elementos no documento (ou seja, o *conteúdo semântico*) é separado das regras que definem a aparência dos elementos (a *apresentação*). As regras de apresentação são escritas em uma linguagem diferente, chamada *Cascading Style Sheets* (CSS). Ela permite alterar quase todos os aspectos visuais do documento, como fontes, cores e o posicionamento dos elementos ao longo da página.

Na maioria dos casos, os documentos HTML não são pensados para ser exibidos em um layout fixo, como um arquivo PDF. Em vez disso, as páginas web baseadas em HTML devem se adaptar a uma ampla variedade de tamanhos de tela, ou até mesmo a formatos impressos. O CSS oferece opções ajustáveis para garantir que a página seja exibida conforme o esperado, adaptada ao dispositivo ou aplicativo que a abre.

Aplicação de estilos

Existem várias maneiras de incluir CSS em um documento HTML: escrever diretamente na tag do elemento, em uma seção separada do código-fonte da página ou em um arquivo externo a ser referenciado pela página HTML.

O atributo `style`

A maneira mais simples de modificar o estilo de um elemento específico é escrevê-lo diretamente na tag do elemento usando o atributo `style`. Todos os elementos HTML visíveis aceitam um atributo `style`, cujo valor pode ser uma ou mais regras de CSS, também conhecidas como *propriedades*. Vamos começar com um exemplo simples, um elemento de parágrafo:

```
<p>My stylized paragraph</p>
```

A sintaxe básica de uma propriedade CSS personalizada é `property: value`, onde `property` é o aspecto particular que você deseja alterar no elemento e `value` define o que substituirá a opção padrão feita pelo navegador. Algumas propriedades se aplicam a todos os elementos e outras se aplicam apenas a elementos específicos. Da mesma forma, existem valores adequados a serem utilizados em cada propriedade.

Para mudar a cor do nosso parágrafo simples, por exemplo, usamos a propriedade `color`. A propriedade `color` refere-se à cor do *primeiro plano*, ou seja, a cor das letras do parágrafo. A cor em si é indicada na parte do valor da regra e pode ser especificada em vários formatos diferentes, incluindo nomes simples como `red`, `green`, `blue`, `yellow` etc. Assim, para deixar a letra do parágrafo roxa, adicione a propriedade personalizada `color: purple` ao atributo `style` do elemento:

```
<p style="color: purple">My first stylized paragraph</p>
```

Outras propriedades personalizadas podem entrar na mesma propriedade `style`, mas devem ser separadas por ponto e vírgula. Se você quiser aumentar o tamanho da fonte, por exemplo, adicione `font-size: larger` à propriedade `style`:

```
<p style="color: purple; font-size: larger">My first stylized paragraph</p>
```

NOTE Não é necessário adicionar espaços ao redor dos dois pontos e ponto e vírgula, mas eles ajudam a facilitar a leitura do código CSS.

Para ver o resultado dessas alterações, salve o HTML a seguir em um arquivo e abra-o em um navegador web (os resultados são mostrados na Figure 32):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
</head>
<body>

<p style="color: purple; font-size: larger">My first stylized paragraph</p>

<p style="color: green; font-size: larger">My second stylized paragraph</p>

</body>
</html>
```

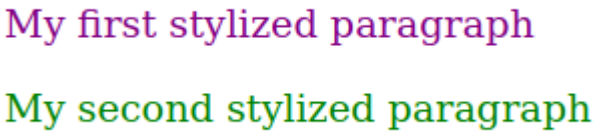

Figure 32. Uma alteração visual simples usando CSS.

Podemos imaginar cada elemento da página como um retângulo ou caixa cujas propriedades geométricas e decorações podem ser manipuladas com CSS. O mecanismo de renderização emprega dois conceitos básicos padrão para o posicionamento do elemento: posicionamento *em bloco* e posicionamento *em linha*. Os elementos de bloco ocupam todo o espaço horizontal do elemento pai e são posicionados sequencialmente, de cima para baixo. Sua altura (sua *dimensão vertical*, que não deve ser confundida com sua posição *no alto* da página) geralmente depende da quantidade de conteúdo que incluem. Os elementos em linha são posicionados horizontalmente, da esquerda para a direita, até que não haja mais espaço no lado direito; depois disso, o elemento continua em uma nova linha logo abaixo da atual. Elementos como `p`, `div` e `section` são posicionados como blocos por padrão, ao passo que elementos como `span`, `em`, `a` e letras sozinhas são posicionados em linha. Esses métodos básicos de posicionamento podem ser modificados fundamentalmente pelas regras do CSS.

O retângulo correspondente ao elemento `p` no corpo do documento HTML de amostra ficará visível se adicionarmos a propriedade `background-color` à regra (Figure 33):

```
<p style="color: purple; font-size: larger; background-color: silver">My first
stylized paragraph</p>

<p style="color: green; font-size: larger; background-color: silver">My second
stylized paragraph</p>
```

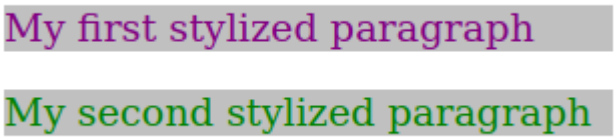

Figure 33. Retângulos correspondentes aos parágrafos.

Conforme adicionamos novas propriedades personalizadas de CSS ao atributo `style`, o código como um todo vai começando a ficar confuso. A inclusão de muitas regras de CSS no atributo `style` prejudica a separação entre a estrutura (HTML) e a apresentação (CSS). Além disso, você logo perceberá que muitos estilos são compartilhados entre diferentes elementos e não vale a pena repeti-los em cada um deles.

Regras de CSS

Em vez de estilizar os elementos diretamente nos atributos `style` de cada um deles, é muito mais prático informar ao navegador sobre a coleção inteira de elementos aos quais o estilo personalizado se aplica. Para isso, adicionamos um *seletor* às propriedades personalizadas, combinando elementos por tipo, classe, ID único, posição relativa etc. A combinação de um *seletor* com as propriedades personalizadas correspondentes — o que também chamamos de *declarações* — constitui uma *regra de CSS*. A sintaxe básica de uma regra de CSS é `selector { property: value }`. Como no caso do atributo `style`, propriedades separadas por ponto e vírgula podem ser agrupadas, como em `p { color: purple; font-size: larger }`. Essa regra pega todos os elementos `p` da página e aplica as propriedades personalizadas `color` e `font-size`.

Uma regra de CSS para um elemento pai abarcará automaticamente todos os seus elementos filhos. Assim, podemos aplicar propriedades personalizadas a todo o texto da página, independentemente de ele estar dentro ou fora de uma tag `<p>`, usando o seletor `body`: `body { color: purple; font-size: larger }`. Essa estratégia nos livra de precisar escrever a mesma regra novamente para todos os elementos filhos, embora possa ser necessário escrever regras adicionais para "desfazer" ou modificar as regras herdadas.

A tag `style`

A tag `<style>` permite escrever regras de CSS dentro da página HTML que queremos estilizar. Ela permite ao navegador diferenciar o código CSS do código HTML. A tag `<style>` entra na seção `head` do documento:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <style type="text/css">

  p { color: purple; font-size: larger }

  </style>

</head>
<body>

<p>My first stylized paragraph</p>

<p>My second stylized paragraph</p>

</body>
</html>
```

O atributo `type` informa ao navegador qual tipo de conteúdo está dentro da tag `<style>`, ou seja, seu *tipo MIME*. Como todo navegador que suporta CSS pressupõe que o tipo da tag `<style>` é `text/css` por padrão, a inclusão do atributo `type` é opcional. Há também um atributo `media` que indica o tipo de mídia — monitores ou impressão, por exemplo — a que se aplicam as regras de CSS na tag `<style>`. Por padrão, as regras de CSS se aplicam a qualquer mídia em que o documento for exibido.

Como no código HTML, as quebras de linha e o recuo de código não alteram a forma como as regras de CSS são interpretadas pelo navegador. Se escrevermos:

```
<style type="text/css">

p { color: purple; font-size: larger }

</style>
```

o resultado será exatamente igual ao de:

```
<style type="text/css">

p {
  color: purple;
  font-size: larger;
}

</style>
```

Os bytes extras ocupados pelos espaços e quebras de linha fazem pouca diferença no tamanho final do documento e não têm um impacto significativo no tempo de carregamento da página, portanto a escolha do layout é questão de gosto. Observe o ponto e vírgula após a última declaração (`font-size: larger;`) da regra de CSS. Esse ponto e vírgula não é obrigatório, mas sua presença facilita a inclusão de outra declaração na linha seguinte sem se preocupar com a falta de um ponto e vírgula.

As declarações em linhas separadas também ajudam quando precisamos comentar uma declaração. Sempre que você quiser desabilitar temporariamente uma declaração para, por exemplo, resolver um problema na página, basta colocar a linha correspondente entre `/*` e `*/`:

```
p {
  color: purple;
  /*
  font-size: larger;
  */
}
```

ou:

```
p {
  color: purple;
  /* font-size: larger; */
}
```

Escrita assim, a declaração `font-size: larger` será ignorada pelo navegador. Tenha o cuidado de abrir e fechar os comentários corretamente; caso contrário, o navegador pode não ser capaz de interpretar as regras.

Os comentários também são úteis para escrever lembretes sobre as regras:

```
/* Texts inside <p> are purple and larger */
p {
  color: purple;
  font-size: larger;
}
```

Lembretes como o do exemplo são opcionais em folhas de estilo contendo um pequeno número de regras, mas tornam-se essenciais para ajudar a navegar por folhas de estilo com centenas de regras (ou mais).

Embora o atributo `style` e a tag `<style>` sejam adequados para testar estilos personalizados e úteis em situações específicas, eles não são usados comumente. Em vez disso, as regras de CSS normalmente são mantidas em um arquivo separado que pode ser consultado a partir do documento HTML.

CSS em arquivos externos

O método mais usado para associar definições CSS a um documento HTML é armazenar o CSS em um arquivo separado. Este método oferece duas vantagens principais sobre os anteriores:

- As mesmas regras de estilo podem ser compartilhadas entre documentos distintos.
- O arquivo CSS geralmente é armazenado em cache pelo navegador, acelerando os tempos de carregamento futuros. Os arquivos CSS têm a extensão `.css` e, como os arquivos HTML, podem ser editados em qualquer editor de texto simples. Ao contrário dos arquivos HTML, os arquivos CSS não têm uma estrutura em níveis construída com tags hierárquicas, como `<head>` ou `<body>`. Em vez disso, o arquivo CSS é apenas uma lista de regras processadas em ordem sequencial pelo navegador. As mesmas regras escritas dentro de uma tag `<style>` poderiam constar em um arquivo CSS.

A associação entre o documento HTML e as regras de CSS armazenadas em um arquivo é definida apenas no documento HTML. Para o arquivo CSS, não importa se existem elementos que correspondem às suas regras e, portanto, não há necessidade de enumerar no arquivo CSS os documentos HTML aos quais ele está vinculado. No lado do HTML, todas as folhas de estilo vinculadas serão aplicadas ao documento, como se as regras estivessem escritas em uma tag `<style>`.

A tag HTML `<link>` define uma folha de estilo externa a ser usada no documento atual e deve constar da seção `head` do documento HTML:

```
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <link href="style.css" rel="stylesheet">

</head>
```

Podemos colocar a regra para o elemento `p` que usamos anteriormente no arquivo `style.css` e o resultado visto pelo visitante da página web será o mesmo. Se o arquivo CSS não estiver no mesmo diretório do documento HTML, especifique o caminho relativo ou completo até ele no atributo `href`. A tag `<link>` pode fazer referência a diferentes tipos de recursos externos, por isso é importante estabelecer a relação que o recurso externo tem com o documento atual. Para os arquivos CSS externos, a relação é definida em `rel="stylesheet"`.

O atributo `media` pode ser usado da mesma forma que na tag `<style>`: ele indica as mídias, como monitores de computador ou impressão, às quais as regras do arquivo externo devem se aplicar.

O CSS pode alterar completamente um elemento, mas ainda assim é importante empregar o elemento apropriado para os componentes da página. Caso contrário, as tecnologias de acessibilidade, como os leitores de tela, podem não ser capazes de identificar as seções corretas da página.

Exercícios Guiados

1. Que métodos podem ser usados para alterar a aparência de elementos HTML usando CSS?

2. Por que não é recomendado usar o atributo `style` na tag `<p>` se houver parágrafos irmãos que devem ter a mesma aparência?

3. Qual é a política de posicionamento padrão para o elemento `div`?

4. Qual atributo da tag `<link>` indica a localização de um arquivo CSS externo?

5. Qual a seção correta para a inclusão do elemento `link` dentro de um documento HTML?

Exercícios Exploratórios

1. Por que não é recomendado usar uma tag `<div>` para identificar uma palavra em uma frase comum?

2. O que acontece se você iniciar um comentário com `/*` no meio de um arquivo CSS, mas esquecer de fechá-lo com `*/`?

3. Escreva uma regra CSS para incluir um sublinhado em todos os elementos `em` do documento.

4. Como podemos indicar que uma folha de estilo de uma tag `<style>` ou `<link>` deve ser usada somente por sintetizadores de fala?

Resumo

Esta lição cobre os conceitos básicos de CSS e como integrá-lo ao HTML. Regras escritas em folhas de estilo CSS são o método padrão para alterar a aparência dos documentos HTML. O CSS nos permite manter o conteúdo semântico separado das políticas de apresentação. Esta lição aborda os seguintes conceitos e procedimentos:

- Como alterar as propriedades CSS usando o atributo `style`.
- A sintaxe básica das regras de CSS.
- Uso da tag `<style>` para incorporar regras de CSS ao documento.
- Uso da tag `<link>` para incorporar folhas de estilo externas ao documento.

Respostas aos Exercícios Guiados

1. Que métodos podem ser usados para alterar a aparência de elementos HTML usando CSS?

   Três métodos básicos: Escrever diretamente na tag do elemento, em uma seção separada do código-fonte da página ou em um arquivo externo a ser consultado pela página HTML.

2. Por que não é recomendado usar o atributo `style` na tag `<p>` se houver parágrafos irmãos que devem ter a mesma aparência?

   A declaração de CSS precisará ser reproduzida nas outras tags `<p>`, o que consome tempo, aumenta o tamanho do arquivo e dá margem a erros.

3. Qual é a política de posicionamento padrão para o elemento `div`?

   O elemento `div` é tratado como um elemento de bloco por padrão, então ele ocupará todo o espaço horizontal de seu elemento pai e sua altura dependerá do conteúdo.

4. Qual atributo da tag `<link>` indica a localização de um arquivo CSS externo?

   O atributo `href`.

5. Qual a seção correta para a inclusão do elemento `link` dentro de um documento HTML?

   O documento `link` deve estar na seção `head` do documento HTML.

Respostas aos Exercícios Exploratórios

1. Por que não é recomendado usar uma tag `<div>` para identificar uma palavra em uma frase comum?

   A tag `<div>` cria uma separação semântica entre duas seções distintas do documento e interfere nas ferramentas de acessibilidade quando é usada para elementos de texto em linha.

2. O que acontece se você iniciar um comentário com `/*` no meio de um arquivo CSS, mas esquecer de fechá-lo com `*/`?

   Todas as regras após o comentário serão ignoradas pelo navegador.

Escreva uma regra CSS para incluir um sublinhado em todos os elementos `em` *do documento.*

+ `em { text-decoration: underline }`

+ ou

+ `em { text-decoration-line: underline }`

1. Como podemos indicar que uma folha de estilo de uma tag `<style>` ou `<link>` deve ser usada somente por sintetizadores de fala?

   O valor de seu atributo `media` deve ser `speech`.

033.2 Seletores de CSS e aplicação de estilo

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 033.2

Peso

3

Áreas chave de conhecimento

- Usar seletores para aplicar regras de CSS aos elementos
- Entender as pseudoclasses CSS
- Entender a ordem e a precedência das regras de CSS
- Entender a herança CSS

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `element`; `.class`; `#id`
- `a, b`; `a.class`; `a b`;
- `:hover`, `:focus`
- `!important`

033.2 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	033 Estilização de conteúdo com CSS
Objetivo:	033.2 Seletores de CSS e aplicação de estilo
Lesson:	1 of 1

Introdução

Ao escrever uma regra CSS, devemos informar ao navegador a quais elementos a regra se aplica. Para isso, especificamos um *seletor*: um padrão que pode corresponder a um elemento ou grupo de elementos. Os seletores existem em muitas formas diferentes, que podem ser baseadas no nome do elemento, seus atributos, seu posicionamento relativo na estrutura do documento ou uma combinação dessas características.

Estilos de página inteira

Uma das vantagens de se usar CSS é não precisar escrever regras individuais para elementos que compartilham o mesmo estilo. Um asterisco aplica a regra a todos os elementos da página web, como mostrado no exemplo a seguir:

```
* {
  color: purple;
  font-size: large;
}
```

O seletor `*` não é o único método para se aplicar uma regra de estilo a todos os elementos da página. Um seletor que usa um nome de tag para aplicar estilos aos elementos contidos nessa tag é chamado de *seletor de tipo*; assim, qualquer nome de tag HTML, como `body`, `p`, `table`, `em`, etc., pode ser usado como seletor. No CSS, o estilo do pai é *herdado* pelos elementos filhos. Portanto, na prática, usar o seletor `*` tem o mesmo efeito que aplicar uma regra ao elemento `body`:

```
body {
  color: purple;
  font-size: large;
}
```

Além disso, o recurso de cascata do CSS permite ajustar as propriedades herdadas de um elemento. Você pode escrever uma regra CSS geral que se aplica a todos os elementos da página e, em seguida, escrever regras para elementos ou conjuntos de elementos específicos.

Se o mesmo elemento corresponder a duas ou mais regras conflitantes, o navegador aplicará a regra do seletor mais específico. Veja por exemplo as seguintes regras de CSS:

```
body {
  color: purple;
  font-size: large;
}

li {
  font-size: small;
}
```

O navegador aplicará os estilos `color: purple` e `font-size: large` a todos os elementos dentro do elemento `body`. No entanto, se houver elementos `li` na página, o navegador substituirá o estilo `font-size: large` pelo estilo `font-size: small`, porque o seletor `li` tem uma relação mais forte com o elemento `li` do que o seletor `body`.

O CSS não limita o número de seletores equivalentes na mesma folha de estilo, de forma que é possível definir duas ou mais regras usando o mesmo seletor:

```
li {
  font-size: small;
}

li {
  font-size: x-small;
}
```

Nesse caso, ambas as regras são igualmente específicas para o elemento `li`. O navegador não pode aplicar regras conflitantes, então ele escolherá a regra que aparece em último lugar no arquivo de CSS. Para evitar confusão, a recomendação é agrupar todas as propriedades que usam o mesmo seletor.

A ordem em que as regras aparecem na folha de estilo afeta a forma como são aplicadas no documento, mas podemos contornar esse comportamento usando uma regra `important`. Se, por algum motivo, você quiser manter as duas regras `li` separadas, mas forçar a aplicação da primeira em vez da segunda, marque a primeira regra como importante:

```
li {
  font-size: small !important;
}

li {
  font-size: x-small;
}
```

É preciso usar com cautela as regras `!Important`, pois elas quebram a cascata natural da folha de estilo e tornam mais difícil encontrar e corrigir problemas no arquivo CSS.

Seletores restritivos

Vimos que é possível alterar certas propriedades herdadas usando seletores correspondentes a tags específicas. No entanto, é comum precisarmos usar estilos distintos em elementos do mesmo tipo.

Podemos incorporar atributos de tags HTML aos seletores para restringir o conjunto de elementos aos quais eles se referem. Suponha que a página HTML em que você está trabalhando tem dois tipos de listas não ordenadas (`<ul>`): um deles é usado no topo da página como um menu para as seções do site e o outro em listas convencionais no corpo do texto:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

<div id="content">

<p>The three rocky planets of the solar system are:</p>

<ul>
  <li>Mercury</li>
  <li>Venus</li>
  <li>Earth</li>
  <li>Mars</li>
</ul>

<p>The outer giant planets made most of gas are:</p>

<ul>
  <li>Jupiter</li>
  <li>Saturn</li>
  <li>Uranus</li>
  <li>Neptune</li>
</ul>

</div><!-- #content -->


<div id="footer">

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

```
</div><!-- #footer -->

</body>
</html>
```

Por padrão, aparece um círculo preto à esquerda de cada item da lista. Queremos remover os círculos da lista do menu superior, mas preservar os círculos na outra lista. No entanto, se simplesmente usarmos o seletor `li`, os círculos na lista que está na seção do corpo do texto também serão removidos. É necessário pedir ao navegador para modificar apenas os itens de uma lista, mas não da outra.

Existem diversas maneiras de escrever seletores que correspondam a elementos específicos na página. Como mencionado anteriormente, veremos primeiro como usar os atributos dos elementos para isso. Neste exemplo em particular, podemos usar o atributo `id` para especificar apenas a lista superior.

O atributo `id` atribui um identificador único ao elemento correspondente, que podemos usar como parte do seletor da regra de CSS. Antes de escrever a regra, edite o arquivo HTML de exemplo e adicione `id="topmenu"` ao elemento `ul` usado para o menu superior:

```
<ul id="topmenu">
  <li>Home</li>
  <li>Articles</li>
  <li>About</li>
</ul>
```

Já existe um elemento `link` na seção `head` do documento HTML apontando para o arquivo de folha de estilo `style.css` na mesma pasta, por isso podemos adicionar as seguintes regras CSS a ele:

```
ul#topmenu {
  list-style-type: none
}
```

A cerquilha é usada em um seletor, após um elemento, para designar um ID (sem espaço de separação). O nome da tag à esquerda da cerquilha é opcional, pois não haverá outro elemento com o mesmo ID. Portanto, o seletor pode ser escrito apenas como `#topmenu`.

Embora `list-style-type` não seja uma propriedade direta do elemento `ul`, as propriedades CSS do elemento pai são herdadas por seus filhos, de modo que o estilo atribuído ao elemento `ul` será herdado por seus elementos filhos `li`.

Nem todos os elementos têm um ID pelo qual podem ser selecionados. Na verdade, é preferível que apenas alguns elementos-chave de layout em uma página tenham IDs. Veja as listas de planetas usadas no nosso código, por exemplo. Embora seja possível atribuir IDs exclusivos para cada elemento repetido individual, esse método não seria prático para páginas mais longas, com muito conteúdo. Em vez disso, podemos usar o ID do elemento `div` pai como seletor para alterar as propriedades de seus elementos internos.

No entanto, o elemento `div` não está diretamente relacionado às listas em HTML; portanto, adicionar a propriedade `list-style-type` a ele não terá efeito em seus filhos. Assim, para trocar o círculo preto nas listas dentro do conteúdo `div` por um círculo vazado, precisamos usar um seletor *descendente*:

```
#topmenu {
  list-style-type: none
}

#content ul {
  list-style-type: circle
}
```

O seletor `#content ul` é chamado de seletor descendente porque corresponde apenas aos elementos `ul` que são filhos do elemento cujo ID é `content`. Podemos usar tantos níveis de descendência quantos forem necessários. Por exemplo, `#content ul li` corresponderia apenas aos elementos `li` descendentes dos elementos `ul`, que por sua vez são descendentes do elemento cujo ID é `content`. Mas, neste exemplo, o seletor mais longo terá o mesmo efeito que `#content ul`, já que os elementos `li` herdam as propriedades CSS definidas para seu `ul` pai. Os seletores descendentes se tornam uma técnica essencial à medida que o layout da página cresce em complexidade.

Digamos que agora você deseja alterar a propriedade `font-style` dos itens da lista `topmenu` e da lista no *div de rodapé* para torná-los oblíquos. Não é possível simplesmente escrever uma regra de CSS usando `ul` como seletor, porque isso também vai alterar os itens da lista no *div de conteúdo*. Até agora, modificamos as propriedades CSS usando um seletor por vez, e esse método também pode ser usado para esta tarefa:

```
#topmenu {
  font-style: oblique
}

#footer ul {
  font-style: oblique
}
```

No entanto, os seletores separados não são a única maneira de fazer isso. O CSS nos permite agrupar seletores que compartilham um ou mais estilos, usando uma lista de seletores separados por vírgulas:

```
#topmenu, #footer ul {
  font-style: oblique
}
```

O agrupamento de seletores elimina o trabalho extra de escrever estilos duplicados. Além disso, você pode querer alterar a propriedade novamente no futuro e talvez não se lembre de alterá-la em todos os lugares diferentes.

Classes

Se não quiser se preocupar muito com a hierarquia de elementos, você pode simplesmente adicionar uma `class` ao conjunto de elementos que deseja personalizar. As classes são semelhantes aos IDs, mas em vez de identificar um único elemento na página, elas identificam um grupo de elementos que compartilham as mesmas características.

Veja por exemplo a página HTML na qual estamos trabalhando nesta lição. É improvável que nas páginas do mundo real encontremos estruturas simples assim, então seria mais prático selecionar um elemento usando apenas as classes ou uma combinação de classes e descendência. Para aplicar a propriedade `font-style: oblique` às listas de menu usando classes, primeiro precisamos adicionar a propriedade `class` aos elementos no arquivo HTML. Faremos isso primeiro no menu superior:

```
<ul id="topmenu" class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

E depois no menu do rodapé:

```
<div id="footer">

<ul class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

</div><!-- #footer -->
```

Com isso, podemos substituir o grupo de seletores `#topmenu, #footer ul` pelo seletor baseado em classe `.menu`:

```
.menu {
  font-style: oblique
}
```

Como no caso dos seletores baseados em ID, adicionar o nome da tag à esquerda do ponto nos seletores baseados em classe é opcional. No entanto, ao contrário dos IDs, a mesma classe pode e deve ser usada em mais de um elemento, e eles sequer precisam ser do mesmo tipo. Portanto, se a classe `menu` é compartilhada entre diferentes tipos de elementos, o uso do seletor `ul.menu` afetaria apenas os elementos `ul` que possuem a classe `menu`. Mas se usarmos `.menu` como seletor, qualquer elemento que tenha a classe `menu` será afetado, independentemente do seu tipo.

Além disso, os elementos podem ser associados a mais de uma classe. Poderíamos diferenciar entre o menu superior e o inferior adicionando uma classe extra a cada um deles:

```
<ul id="topmenu" class="menu top">
```

E no menu do rodapé:

```
<ul class="menu footer">
```

Quando o atributo `class` possui mais de uma classe, elas devem ser separadas por espaços. Agora, além da regra CSS compartilhada entre os elementos da classe `menu`, podemos abordar o menu superior e o rodapé usando suas classes correspondentes:

```
.menu {
  font-style: oblique
}

.menu.top {
  font-size: large
}

.menu.footer {
  font-size: small
}
```

Esteja ciente de que `.menu.top` é diferente de `.menu .top` (com um espaço entre as palavras). O primeiro seletor afetará os elementos que têm ambas as classes `menu` e `top`, enquanto o segundo afeta os elementos que têm a classe `top` e um elemento pai com a classe `menu`.

Seletores especiais

Os seletores CSS também podem fazer referência a estados dinâmicos de elementos. Esses seletores são particularmente úteis para elementos interativos, como hiperlinks. Podemos querer alterar a aparência dos hiperlinks quando o ponteiro do mouse se move sobre eles, para chamar a atenção do visitante.

De volta à página de exemplo, poderíamos remover os sublinhados dos links no menu superior e exibir uma linha apenas quando o ponteiro do mouse se movesse sobre o link correspondente. Para fazer isso, primeiro escrevemos uma regra para remover o sublinhado dos links no menu superior:

```
.menu.top a {
  text-decoration: none
}
```

Em seguida, usamos a pseudoclasse `:hover` nesses mesmos elementos para criar uma regra de CSS a ser aplicada somente quando o ponteiro do mouse estiver sobre o elemento correspondente:

```
.menu.top a:hover {
  text-decoration: underline
}
```

O seletor de pseudoclasse `:hover` aceita todas as propriedades CSS das regras de CSS convencionais. Outras pseudoclasses são `:visited`, que corresponde aos hiperlinks já visitados, e `:focus`, que

corresponde aos elementos do formulário que estão em foco.

Exercícios Guiados

1. Suponha que uma página HTML tenha uma folha de estilo associada contendo as duas regras a seguir:

```
p {
  color: green;
}

p {
  color: red;
}
```

   Que cor o navegador aplicará ao texto dentro dos elementos `p`?

2. Qual é a diferença entre usar o seletor de ID `div#main` e `#main`?

3. Qual seletor corresponde a todos os elementos `p` usados dentro de um `div` com o atributo de ID `#main`?

4. Qual é a diferença entre usar o seletor de classe `p.highlight` e `.highlight`?

Exercícios Exploratórios

1. Escreva uma única regra CSS que altere a propriedade `font-style` para `oblique`. A regra deve corresponder apenas aos elementos `a` que estão dentro de `<div id="sidebar"></div>` ou `<ul class="links"></ul>`.

2. Suponha que você queira alterar o estilo dos elementos cujo atributo `class` está definido como `article reference`, como em `<p class="article reference">`. Entretanto, o seletor `.article .reference` não parece alterar sua aparência. Por que o seletor não está alterando os elementos conforme o esperado?

3. Escreva uma regra de CSS para alterar a propriedade `color` de todos os links visitados na página para `red`.

Resumo

Esta lição abordou a maneira de usar seletores CSS e como o navegador decide quais estilos aplicar a cada elemento. Por ser separado da marcação HTML, o CSS fornece muitos seletores para alterar elementos individuais ou grupos de elementos na página. A lição tratou dos seguintes conceitos e procedimentos:

- Estilos de página inteira e herança de estilo.
- Aplicação de estilos por tipo de elemento.
- Uso do ID e da classe do elemento como seletor.
- Seletores compostos.
- Uso de pseudoclasses para aplicar estilos aos elementos dinamicamente.

Respostas aos Exercícios Guiados

1. Suponha que uma página HTML tenha uma folha de estilo associada contendo as duas regras a seguir:

```
p {
  color: green;
}

p {
  color: red;
}
```

   Que cor o navegador aplicará ao texto dentro dos elementos `p`?

   A cor `red` (vermelha). Quando dois ou mais seletores equivalentes têm propriedades conflitantes, o navegador dará preferência ao último.

2. Qual é a diferença entre usar o seletor de ID `div#main` e `#main`?

   O seletor `div#main` afetará um elemento `div` com o ID `main`, ao passo que o seletor `#main` afetará o elemento que tenha o ID `main`, qualquer que seja o seu tipo.

3. Qual seletor corresponde a todos os elementos `p` usados dentro de um `div` com o atributo de ID `#main`?

   O seletor `#main p` ou `div#main p`.

4. Qual é a diferença entre usar o seletor de classe `p.highlight` e `.highlight`?

   O seletor `p.highlight` afeta somente os elementos de tipo `p` que têm a classe `highlight`, enquanto o seletor `.highlight` afeta todos os elementos que têm a classe `highlight`, independentemente de seu tipo.

Respostas aos Exercícios Exploratórios

1. Escreva uma única regra CSS que altere a propriedade `font-style` para `oblique`. A regra deve corresponder apenas aos elementos `a` que estão dentro de `<div id="sidebar"></div>` ou `<ul class="links"></ul>`.

   ```
   #sidebar a, ul.links a {
     font-style: oblique
   }
   ```

2. Suponha que você queira alterar o estilo dos elementos cujo atributo `class` está definido como `article reference`, como em `<p class="article reference">`. Entretanto, o seletor `.article .reference` não parece alterar sua aparência. Por que o seletor não está alterando os elementos conforme o esperado?

   O seletor `.article .reference` afetará os elementos com a classe `reference` que forem descendentes dos elementos com a classe `article`. Para afetar os elementos pertencentes às classes `article` e `reference`, o seletor deve ser `.article.reference` (sem o espaço entre as palavras).

3. Escreva uma regra de CSS para alterar a propriedade `color` de todos os links visitados na página para `red`.

   ```
   a:visited {
     color: red;
   }
   ```

033.3 Estilização com CSS

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 033.3

Peso

2

Áreas chave de conhecimento

- Entender as propriedades fundamentais do CSS
- Entender as unidades comumente usadas em CSS

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `px`, `%`, `em`, `rem`, `vw`, `vh`
- `color`, `background`, `background-*`, `font`, `font-*`, `text-*`, `list-style`, `line-height`

033.3 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	033 Estilização de conteúdo com CSS
Objetivo:	033.3 Estilização com CSS
Lição:	1 de 1

Introdução

O CSS inclui centenas de propriedades que podem ser usadas para modificar a aparência dos elementos HTML. Somente os designers experientes conseguem memorizar a maioria deles. No entanto, vale a pena conhecer as propriedades básicas que são aplicáveis a qualquer elemento, bem como algumas propriedades específicas de determinados elementos. Este capítulo trata de algumas propriedades populares que você certamente encontrará em seu caminho.

Propriedades e valores comuns do CSS

Muitas propriedades CSS têm o mesmo tipo de valor. As cores, por exemplo, têm o mesmo formato numérico, quer estejamos mexendo na cor da fonte ou na cor do fundo. Da mesma forma, as unidades disponíveis para mudar o tamanho da fonte também são usadas para alterar a espessura de uma borda. No entanto, o formato do valor nem sempre é único. As cores, por exemplo, podem ser inseridas em vários formatos diferentes, como veremos a seguir.

Cores

Alterar a cor de um elemento é provavelmente uma das primeiras coisas que os designers aprendem

a fazer em CSS. É possível trocar a cor do texto, a cor do plano de fundo, a cor da borda dos elementos e muito mais.

O valor de uma cor em CSS pode ser escrito como uma *palavra-chave da cor* (ou seja, um nome de cor) ou como um valor numérico que lista cada componente da cor. Todos os nomes de cores comuns (em inglês), como `black`, `white`, `red`, `purple`, `green`, `yellow` e `blue`, são aceitos como palavras-chave de cores. A lista completa de palavras-chave de cores aceitas pelo CSS está disponível na página web da W3C. Mas para ter um controle mais preciso sobre a cor, é melhor usar o valor numérico.

Palavras-chave de cor

Mostraremos primeiro o uso da palavra-chave de cor, por ser mais simples. A propriedade `color` altera a cor do texto no elemento correspondente. Portanto, para colocar todo o texto da página em roxo, você pode escrever a seguinte regra CSS:

```
* {
  color: purple;
}
```

Valores Numéricos de Cor

Embora intuitivas, as palavras-chave de cor não oferecem a gama completa de cores possíveis nos monitores modernos. Os webdesigners geralmente desenvolvem uma paleta de cores personalizada, atribuindo valores específicos aos componentes vermelho, verde e azul.

Cada componente de uma cor é representado por um número binário de oito bits, variando de 0 a 255. Todos os três componentes devem ser especificados na mistura de cores, sempre na ordem vermelho, verde, azul. Portanto, para trocar a cor de todo o texto da página para vermelho usando a notação RGB, usamos `rgb(255,0,0)`:

```
* {
  color: rgb(255,0,0);
}
```

Um componente definido como `0` indica que a cor básica correspondente não é usada na mistura de cores. Também é possível usar porcentagens em vez de números:

```
* {
  color: rgb(100%,0%,0%);
}
```

A notação RGB raramente é vista nos aplicativos de desenho para a criação de layouts ou apenas para a escolha de cores. Em vez disso, é mais comum ver as cores CSS expressas em valores *hexadecimais*. Os componentes de uma cor em notação hexadecimal também variam de 0 a 255, mas de uma forma mais sucinta, começando com um símbolo de hash (cerquilha) `#` e limitando-se a um comprimento fixo de dois dígitos para todos os componentes. O valor mínimo `0` é `00` e o valor máximo `255` é `FF`, de modo que a cor `red` (vermelho) seria `#FF0000`.

TIP

Se os dígitos de cada componente de uma cor hexadecimal se repetirem, o segundo dígito pode ser omitido. A cor `red`, por exemplo, pode ser escrita com um único dígito para cada componente: `#F00`.

A lista a seguir mostra a notação RGB e hexadecimal para algumas cores básicas:

Palavra-chave da cor	Notação RGB	Valor Hexadecimal
`black`	`rgb(0,0,0)`	`#000000`
`white`	`rgb(255,255,255)`	`#FFFFFF`
`red`	`rgb(255,0,0)`	`#FF0000`
`purple`	`rgb(128,0,128)`	`#800080`
`green`	`rgb(0,128,0)`	`#008000`
`yellow`	`rgb(255,255,0`	`#FFFF00`
`blue`	`rgb(0,0,255)`	`#0000FF`

As palavras-chave de cores e as letras nos valores de cor hexadecimais não diferenciam maiúsculas de minúsculas.

Opacidade da cor

Além de cores opacas, é possível preencher os elementos da página com cores semitransparentes. A opacidade de uma cor é definida com um quarto componente no valor da cor. Ao contrário dos outros componentes, cujos valores são números inteiros entre 0 a 255, a opacidade é uma fração que varia de `0` a `1`.

O valor mais baixo, `0`, resulta em uma cor completamente transparente, tornando-a indistinguível de qualquer outra cor totalmente transparente. O valor mais alto, `1`, resulta em uma cor totalmente

opaca, idêntica à cor original, sem nenhuma transparência.

Ao usar a notação RGB, especifique as cores com um componente de opacidade por meio do prefixo `rgba` em vez de `rgb`. Portanto, para tornar a cor vermelha semitransparente, use `rgba(255,0,0,0.5)`. O caractere `a` indica o *canal alpha*, um termo comumente usado para especificar o componente de opacidade no jargão gráfico digital.

A opacidade também pode ser definida na notação hexadecimal. Nesse caso, como os outros componentes de cor, a opacidade varia de `00` a `FF`. Portanto, para tornar a cor `red` semitransparente usando a notação hexadecimal, usaríamos `#FF000080`.

Plano de fundo

A cor de fundo de elementos individuais ou de toda a página é definido com a propriedade `background-color`. Ela aceita os mesmos valores da propriedade `color`, seja na forma de palavras-chave ou em notação RGB/hexadecimal.

No entanto, o plano de fundo dos elementos HTML não se restringe às cores. Com a propriedade `background-image`, é possível usar uma imagem como fundo. Todos os formatos convencionais aceitos pelo navegador, como JPEG e PNG, podem ser usados.

O caminho para a imagem deve ser especificado usando um designador `url()`. Se a imagem que você deseja usar estiver na mesma pasta do arquivo HTML, basta incluir o nome do arquivo:

```
body {
  background-image: url("background.jpg");
}
```

Neste exemplo, o arquivo de imagem `background.jpg` será usado como imagem de fundo para todo o corpo da página. Por padrão, a imagem de fundo é repetida se seu tamanho não bastar para cobrir a página inteira, começando no canto superior esquerdo da área correspondente ao seletor da regra. Esse comportamento pode ser modificado com a propriedade `background-repeat`. Se você quiser que a imagem de fundo seja posta na área do elemento sem repeti-la, use o valor `no-repeat`:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
}
```

Também podeMOS fazer a imagem se repetir apenas na direção horizontal (`background-repeat:`

`repeat-x`) ou apenas na vertical (`background-repeat: repeat-y`).

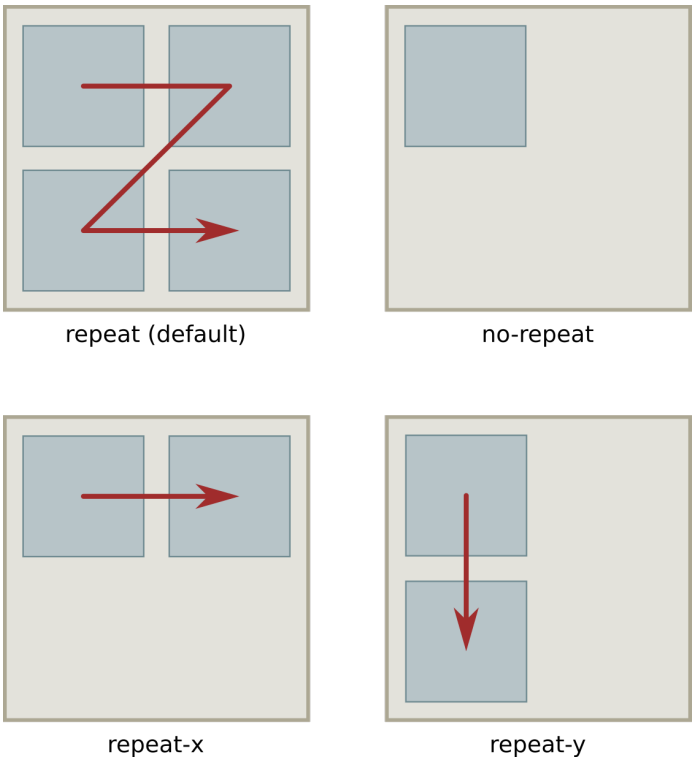

Figure 34. Posicionamento do fundo usando a propriedade `background-repeat`.

TIP

Duas ou mais propriedades CSS podem ser combinadas em uma só, que seria a propriedade *abreviada* (ou shorthand) do plano de fundo. As propriedades `background-image` e `background-repeat`, por exemplo, podem ser reunidas em uma única propriedade de plano de fundo com `background: no-repeat url("background.jpg")`.

Uma imagem de fundo também pode ser posta em uma posição específica na área do elemento graças à propriedade `background-position`. As cinco posições básicas são `top`, `bottom`, `left`, `right` e `center`, mas a posição superior esquerda da imagem também pode ser definida com porcentagens:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
  background-position: 30% 10%;
}
```

A porcentagem de cada posição é relativa ao tamanho correspondente do elemento. No exemplo, o lado esquerdo da imagem de fundo estará a 30% da largura do corpo (normalmente o corpo é todo o

documento visível) e o lado superior da imagem estará a 10% da altura do corpo.

Bordas

Outra personalização de layout comum feita com CSS é alterar a borda de um elemento. A borda se refere à linha que forma um retângulo ao redor do elemento e tem três propriedades básicas: `color`, `style` e `width`.

A cor da borda, definida com `border-color`, segue o mesmo formato que vimos para as outras propriedades de cor.

As bordas podem ser traçadas em um estilo diferente de uma linha sólida. A propriedade `border-style: dashed`, por exemplo, criaria uma borda tracejada. Os outros valores de estilo possíveis são:

`dotted`
: Uma sequência de pontos redondos

`double`
: Duas linhas retas

`groove`
: Uma linha com aparência sulcada

`ridge`
: Uma linha com aparência tridimensional

`inset`
: Um elemento em baixo relevo

`outset`
: Um elemento em alto relevo

A propriedade `border-width` define a espessura da borda. Seu valor pode ser uma palavra-chave em inglês (`thin`, `medium` ou `thick`) ou um valor numérico específico. Se você preferir usar um valor numérico, também precisará especificar a unidade correspondente. Falaremos disso a seguir.

Valores de unidade

Os tamanhos e distâncias em CSS podem ser definidos de diferentes maneiras. As unidades absolutas são baseadas em um número fixo de pixels da tela e, portanto, não são tão diferentes dos tamanhos e dimensões fixos usados na mídia impressa. Existem também unidades relativas, que são calculadas

dinamicamente a partir de alguma medida fornecida pela mídia onde a página está sendo exibida, como o espaço disponível ou outro tamanho escrito em unidades absolutas.

Unidades absolutas

As unidades absolutas são equivalentes às suas correspondentes físicas, como centímetros ou polegadas. Nos monitores convencionais, uma polegada tem 96 pixels de largura. As seguintes unidades absolutas são comumente usadas:

`in` (polegada)

1 in equivale a a 2,54 cm ou 96 px.

`cm` (centímetro)

1 cm equivale a 96 px / 2,54.

`mm` (milímetro)

1 mm equivale a 1 cm / 10.

`px` (pixel)

1 px equivale a 1 / 96 de polegada.

`pt` (ponto)

1pt equivale a 1 / 72 de polegada.

Lembre-se de que a proporção de pixels por polegada pode variar. Nas telas de alta resolução, onde os pixels são compactados com mais densidade, uma polegada corresponderá a mais de 96 pixels.

Unidades relativas

As unidades relativas variam de acordo com as outras medidas ou com as dimensões da janela de visualização. A janela de visualização é a área do documento atualmente visível em sua janela. No modo de tela inteira, a janela de visualização corresponde à tela do dispositivo. As seguintes unidades relativas são comumente usadas:

`%`

Porcentagem — é relativa ao elemento pai.

`em`

O tamanho da fonte usada no elemento.

`rem`

O tamanho da fonte usada no elemento raiz.

`vw`

1% da largura da janela de visualização.

`vh`

1% da altura da janela de visualização.

A vantagem de se usar unidades relativas é poder criar layouts ajustáveis alterando apenas alguns tamanhos determinados. Por exemplo, podemos usar a unidade `pt` para definir o tamanho da fonte no elemento do corpo e a unidade `rem` para as fontes dos outros elementos. Depois de alterar o tamanho da fonte do corpo, todos os outros tamanhos de fonte serão ajustados a partir dali. Além disso, o uso de `vw` e `vh` para definir as dimensões das seções da página as torna ajustáveis para telas de tamanhos diferentes.

Propriedades das fontes e do texto

A tipografia, ou o estudo dos tipos de fonte, é um assunto muito amplo dentro do design, e a tipografia CSS não fica para trás. No entanto, existem algumas propriedades básicas de fonte que atenderão às necessidades da maioria dos usuários que estão aprendendo CSS.

A propriedade `font-family` define o nome da fonte a ser usada. Não há garantia de que a fonte escolhida estará disponível no sistema onde a página será visualizada, portanto esta propriedade talvez não tenha efeito no documento. Embora seja possível fazer o navegador baixar e usar o arquivo de fonte especificado, a maioria dos webdesigners prefere usar uma família de fontes genérica em seus documentos.

As três famílias de fontes genéricas mais comuns são `serif`, `sans-serif` e `monospace`. Serif é a família de fontes padrão da maioria dos navegadores. Se você preferir usar `sans-serif` para a página inteira, adicione a seguinte regra à sua folha de estilo:

```
* {
  font-family: sans-serif;
}
```

Opcionalmente, podemos primeiro definir um nome específico de família de fonte, seguido pelo nome de família genérico:

```
* {
  font-family: "DejaVu Sans", sans-serif;
}
```

Se o dispositivo que exibe a página tiver essa família de fontes específica, o navegador a usará. Caso contrário, ele usará a fonte padrão correspondente ao nome de família genérico. Os navegadores procuram pelas fontes na ordem em que elas estão especificadas na propriedade.

Qualquer pessoa que tenha usado um aplicativo de processamento de texto também estará familiarizado com três outros ajustes de fonte: tamanho, peso e estilo. Esses três ajustes têm seus equivalentes nas propriedades CSS: `font-size`, `font-weight` e `font-style`.

A propriedade `font-size` aceita palavras-chave de tamanho como `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx-large`. Essas palavras-chave são relativas ao tamanho de fonte padrão usado pelo navegador. As palavras-chave `larger` e `smaller` alteram o tamanho da fonte com relação ao tamanho da fonte do elemento pai. Também é possível expressar o tamanho da fonte com valores numéricos, incluindo a unidade após o valor, ou com porcentagens.

Se você não deseja alterar o tamanho da fonte, mas sim a distância entre as linhas, use a propriedade `line-height`. Uma `line-height` de `1` torna a altura da linha do mesmo tamanho da fonte do elemento, o que pode deixar as linhas de texto próximas demais umas das outras. Portanto, um valor maior que 1 é mais apropriado para textos. Assim como na propriedade `font-size`, outras unidades podem ser usadas junto com o valor numérico.

`font-weight` define a espessura da fonte, usando as conhecidas palavras-chave `normal` ou `bold` (negrito). As palavras-chave `lighter` e `bolder` alteram o peso da fonte do elemento em relação ao peso da fonte de seu elemento pai.

A propriedade `font-style` pode ser definida como `italic` para selecionar a versão em itálico da família de fontes atual. O valor `oblique` seleciona a versão oblíqua da fonte. Essas duas opções são quase idênticas, mas a versão em itálico de uma fonte geralmente é um pouco mais estreita do que a versão oblíqua. Se não houver versões em itálico ou oblíquo da fonte, a fonte será inclinada artificialmente pelo navegador.

Existem outras propriedades que alteram a forma como o texto é processado no documento. Você pode, por exemplo, adicionar um sublinhado a algumas partes do texto que deseja enfatizar. Primeiro, use uma tag `<span>` para delimitar o texto:

```
<p>CSS is the <span class="under">proper mechanism</span> to style HTML
documents.</p>
```

Use então o seletor `.under` para alterar a propriedade `text-decoration`:

```
.under {
  text-decoration: underline;
}
```

Por padrão, todos os elementos `a` (link) são sublinhados. Para remover o sublinhado, use o valor `none` em `text-decoration` de todos os elementos `a`:

```
a {
  text-decoration: none;
}
```

Ao revisar o conteúdo, alguns autores gostam de riscar as partes incorretas ou desatualizadas do texto, para que o leitor saiba que o texto foi atualizado e o que foi removido. Para fazer isso, use o valor `line-through` da propriedade `text-decoration`:

```
.disregard {
  text-decoration: line-through;
}
```

Mais uma vez, uma tag `<span>` pode ser usada para aplicar o estilo:

```
<p>Netscape Navigator <span class="disregard">is</span> was one of the most popular
Web browsers.</p>
```

Existem decorações específicas a um elemento. Os círculos nas listas de marcadores podem ser personalizados usando a propriedade `list-style-type`. Para transformá-los em quadrados, por exemplo, usamos `list-style-type: square`. Para simplesmente removê-los, definimos o valor de `list-style-type` como `none`.

Exercícios Guiados

1. Como adicionar uma semitransparência à cor azul (notação RGB `rgb(0,0,255)`) para usá-la na propriedade CSS `color`?

2. Que cor corresponde ao valor hexadecimal `#000`?

3. Dado que `Times New Roman` é uma fonte `serif` e que não está disponível em todos os dispositivos, como seria possível escrever uma regra CSS para solicitar o uso de `Times New Roman`, tendo a família de fontes genérica `serif` como substituto ?

4. Como usar uma palavra-chave de tamanho relativo para definir uma fonte menor para o elemento `<p class="disclaimer">` em relação ao seu elemento pai?

Exercícios Exploratórios

1. A propriedade `background` é uma abreviatura para definir mais de uma propriedade `background-*` de uma vez. Reescreva a regra CSS a seguir como uma única propriedade abreviada `background`.

```
body {
  background-image: url("background.jpg");
  background-repeat: repeat-x;
}
```

2. Escreva uma regra de CSS para o elemento `<div id="header"></div>` de maneira a alterar *somente* a borda inferior para `4px`.

3. Escreva uma propriedade `font-size` que dobre o tamanho da fonte usada no elemento raiz da página.

4. O *duplo espaçamento* é um recurso comum de formatação nos processadores de texto. Como definir um formato semelhante usando CSS?

Resumo

Esta lição trata da aplicação de estilos simples aos elementos de um documento HTML. O CSS inclui centenas de propriedades e a maioria dos web designers precisará recorrer a manuais de referência para lembrar de todas elas. No entanto, um conjunto relativamente pequeno de propriedades e valores é usado na maioria dos casos e é importante sabê-los de cor. A lição abordou os seguintes conceitos e procedimentos:

- Propriedades CSS fundamentais para alterar cores, planos de fundo e fontes.
- As unidades absolutas e relativas que o CSS pode usar ara definir tamanhos e distâncias, como `px`, `em`, `rem`, `vw`, `vh`, etc.

Respostas aos Exercícios Guiados

1. Como adicionar uma semitransparência à cor azul (notação RGB `rgb(0,0,255)`) para usá-la na propriedade CSS `color`?

   Use o prefixo `rgba` e inclua `0.5` no quarto valor: `rgba(0,0,0,0.5)`.

2. Que cor corresponde ao valor hexadecimal `#000`?

   A cor `black` (preto). O valor hexadecimal `#000` é a abreviatura de `#000000`.

3. Dado que `Times New Roman` é uma fonte `serif` e que não está disponível em todos os dispositivos, como seria possível escrever uma regra CSS para solicitar o uso de `Times New Roman`, tendo a família de fontes genérica `serif` como substituto ?

   ```
   * {
     font-family: "Times New Roman", serif;
   }
   ```

4. Como usar uma palavra-chave de tamanho relativo para definir uma fonte menor para o elemento `<p class="disclaimer">` em relação ao seu elemento pai?

   Using the `smaller` keyword:

   ```
   p.disclaimer {
     font-size: smaller;
   }
   ```

Respostas aos Exercícios Exploratórios

1. A propriedade `background` é uma abreviatura para definir mais de uma propriedade `background-*` de uma vez. Reescreva a regra CSS a seguir como uma única propriedade abreviada `background`.

```
body {
  background-image: url("background.jpg");
  background-repeat: repeat-x;
}
```

```
body {
  background: repeat-x url("background.jpg");
}
```

2. Escreva uma regra de CSS para o elemento `<div id="header"></div>` de maneira a alterar *somente* a borda inferior para `4px`.

```
#header {
  border-bottom-width: 4px;
}
```

3. Escreva uma propriedade `font-size` que dobre o tamanho da fonte usada no elemento raiz da página.

   A unidade `rem` corresponde ao tamanho da fonte usado no elemento raiz, de modo que a propriedade seria `font-size: 2rem`.

4. O *duplo espaçamento* é um recurso comum de formatação nos processadores de texto. Como definir um formato semelhante usando CSS?

   Definindo a propriedade `line-height` para o valor `2em`, pois a unidade `em` corresponde ao tamanho da fonte do elemento atual.

033.4 Layout e modelo de caixa CSS

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 033.4

Peso

2

Áreas chave de conhecimento

- Definir a dimensão, posição e alinhamento dos elementos em um layout CSS
- Especificar como o texto flui em torno de outros elementos
- Entender o fluxo do documento
- Noções de grade CSS
- Noções de design web responsivo
- Noções de consultas de mídia CSS

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `width`, `height`, `padding`, `padding-*`, `margin`, `margin-*`, `border`, `border-*`
- `top`, `left`, `right`, `bottom`
- `display: block | inline | flex | inline-flex | none`
- `position: static | relative | absolute | fixed | sticky`
- `float: left | right | none`
- `clear: left | right | both | none`

Linux Professional Institute

033.4 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	033 Estilização de conteúdo com CSS
Objetivo:	033.4 Layout e modelo de caixa CSS
Lição:	1 de 1

Introdução

Cada um dos elementos visíveis em um documento HTML é renderizado como uma caixa retangular. Assim, o termo *modelo de caixa* descreve o método usado pelo CSS para modificar as propriedades visuais dos elementos. Como caixas de tamanhos diferentes, os elementos HTML podem ser aninhados dentro de elementos *contêiner*—normalmente o elemento `div`—e em seguida separados em seções.

Com o CSS, é possível modificar a posição das caixas, desde pequenos ajustes até mudanças drásticas na disposição dos elementos na página. Além do fluxo normal, a posição de cada caixa pode se basear nos elementos que a rodeiam, seja em seu relacionamento com o contêiner pai ou com a *janela de visualização*, que é a área da página visível para o usuário. Nenhum mecanismo atende a todos os requisitos de layout possíveis, então costuma ser necessário usar uma combinação deles.

Fluxo normal

A forma padrão como o navegador renderiza a árvore do documento é chamada de *fluxo normal*. Os retângulos correspondentes aos elementos são postos mais ou menos na mesma ordem em que aparecem na árvore do documento em relação aos seus elementos pais. No entanto, dependendo do

tipo de elemento, a caixa correspondente pode seguir regras de posicionamento distintas.

Uma boa maneira de entender a lógica do fluxo normal é tornar as caixas visíveis. Podemos começar com uma página bem básica contendo apenas três elementos `div` separados, cada um com um parágrafo de texto aleatório:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CSS Box Model and Layout</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
  <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->

</body>
</html>
```

Cada palavra está em um elemento `span` para que possamos estilizá-las e constatar que são tratadas como caixas também. Para tornar as caixas visíveis, precisamos editar o arquivo de folha de estilo `style.css` referenciado pelo documento HTML. As seguintes regras fazem esse trabalho:

```
* {
  font-family: sans;
  font-size: 14pt;
}

div {
  border: 2px solid #00000044;
}

#first {
  background-color: #c4a000ff;
}

#second {
  background-color: #4e9a06ff;
}

#third {
  background-color: #5c3566da;
}

h2 {
  background-color: #ffffff66;
}

p {
  background-color: #ffffff66;
}

span {
  background-color: #ffffffaa;
}
```

O resultado aparece na Figure 35.

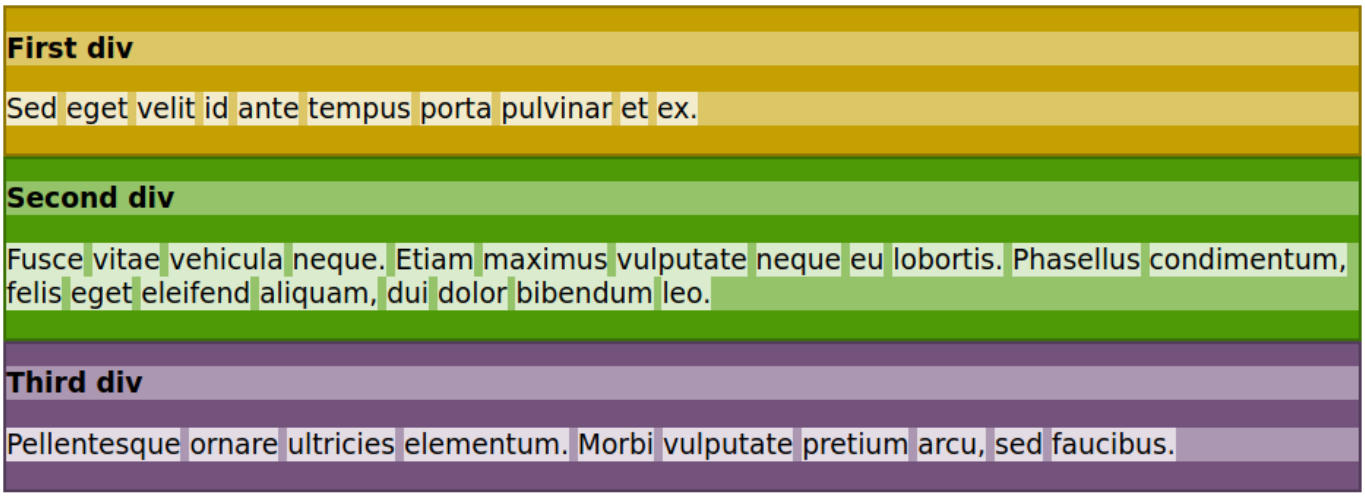

Figure 35. O fluxo de elementos básico é de cima para baixo e da esquerda para a direita.

A Figure 35 mostra que cada tag HTML tem uma caixa correspondente no layout. Os elementos `div`, `h2` e `p` se estendem até a largura do corpo, ao passo que o pai de cada elemento `h2` e `p` é o `div` correspondente. As caixas que se estendem pela largura de seu elemento pai são chamadas elementos de *bloco*. Algumas das tags HTML mais comuns representadas como blocos são `h1`, `h2`, `h3`, `p`, `ul`, `ol`, `table`, `li`, `div`, `section`, `form` e `aside`. Elementos de bloco irmãos — ou seja, que compartilham o mesmo elemento pai imediato — são empilhados dentro do pai, de cima para baixo.

NOTE

Alguns elementos de bloco não se destinam a ser usados como contêineres para outros elementos de bloco. É possível, por exemplo, inserir um elemento de bloco dentro de um elemento `h1` ou `p`, mas isso não é muito aconselhável. Em vez disso, é melhor usar uma tag apropriada como contêiner. As tags de contêiner mais comuns são `div`, `section` e `aside`.

Além do texto em si, elementos como `h1`, `p` e `li` esperam conter apenas elementos *de linha* como filhos. Como a maioria dos modos de escrita ocidentais, os elementos de linha seguem o fluxo de texto da esquerda para a direita. Quando não há espaço restante no lado direito, o fluxo de elementos de linha continua na linha seguinte, assim como o texto. Algumas tags HTML comuns tratadas como caixas de linha são `span`, `a`, `em`, `strong`, `img`, `input` e `label`.

Em nossa página HTML de exemplo, todas as palavras dentro dos parágrafos foram circundadas por uma tag `span`, para que pudessem ser destacadas com uma regra CSS correspondente. Como mostrado na imagem, cada elemento `span` é posicionado horizontalmente da esquerda para a direita até que não haja mais espaço no elemento pai.

A altura do elemento depende de seu conteúdo. Portanto, o navegador ajusta a altura de um elemento de contêiner de forma a acomodar os elementos de bloco aninhados ou as linhas dos

elementos de linha. No entanto, algumas propriedades CSS afetam a forma de uma caixa, sua posição e o posicionamento de seus elementos internos.

As propriedades `margin` e `padding` afetam todos os tipos de caixa. Se você não definir essas propriedades explicitamente, o navegador definirá algumas delas usando valores padrão. Conforme visto na Figure 35, os elementos `h2` e `p` foram exibidos com um espacinho entre eles. Essas lacunas são as margens superior e inferior que o navegador adiciona por padrão a esses elementos. Podemos removê-las modificando as regras de CSS para os seletores `h2` e `p`:

```
h2 {
  background-color: #ffffff66;
  margin: 0;
}

p {
  background-color: #ffffff66;
  margin: 0;
}
```

O resultado aparece na Figure 36.

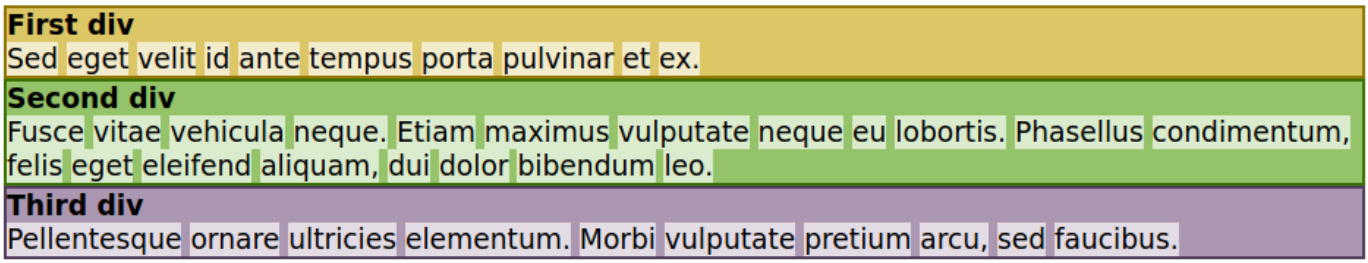

Figure 36. A propriedade `margin` permite alterar ou remover margens de elementos.

O elemento `body`, por padrão, também inclui uma pequena margem que cria um espacinho ao redor dele. Essa lacuna também pode ser removida usando a propriedade `margin`.

Enquanto a propriedade `margin` (margem) define o espaço entre o elemento e seus arredores, a propriedade `padding` (preenchimento) do elemento define o espaço interno entre os limites do contêiner e seus elementos filhos. Considere os elementos `h2` e `p` dentro de cada `div` em nosso código, por exemplo. Poderíamos usar a propriedade de margem deles para criar um espaço nas bordas do `div` correspondente, mas é mais simples alterar a propriedade `padding` do contêiner:

```
#second {
  background-color: #4e9a06ff;
  padding: 1em;
}
```

Apenas a regra para o segundo `div` foi modificada, por isso os resultados (Figure 37) mostram a diferença entre o segundo `div` e os outros contêineres `div`.

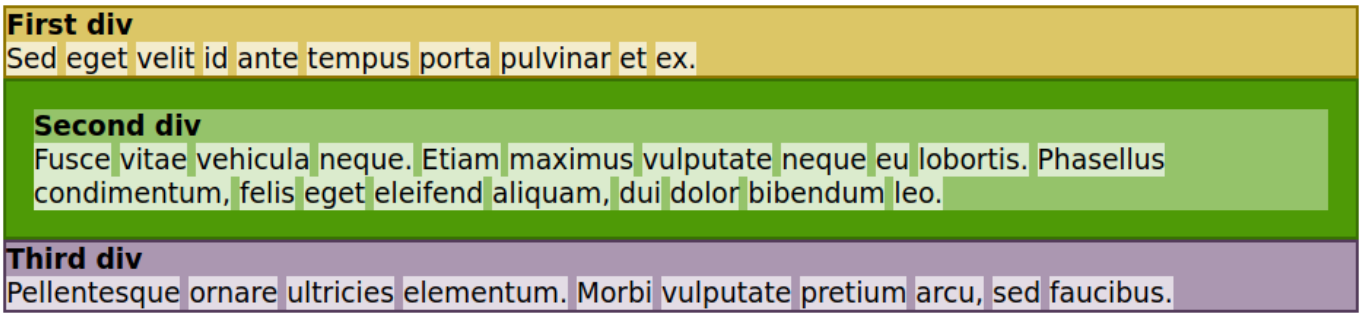

Figure 37. Contêiners `div` diferentes podem ter diferentes preenchimentos.

A propriedade `margin` é uma abreviatura para quatro propriedades que controlam os quatro lados da caixa: `margin-top`, `margin-right`, `margin-bottom` e `margin-left`. Quando um único valor é atribuído a `margin`, como nos exemplos que vimos até agora, ele é aplicado nas quatro margens da caixa. Quando dois valores são escritos, o primeiro define as margens superior e inferior e o segundo, as margens direita e esquerda. A instrução `margin: 1em 2em`, por exemplo, define um espaço de 1 em para as margens superior e inferior e um espaço de 2 em para as margens direita e esquerda. Escrevemos quatro valores para definir as margens dos quatro lados em sentido horário, começando no topo. Os diferentes valores na propriedade abreviada não precisam usar as mesmas unidades.

A propriedade `padding` também é uma abreviatura e segue os mesmos princípios da propriedade `margin`.

Em seu comportamento padrão, os elementos de bloco são esticados para se ajustar à largura disponível. Mas isso não é obrigatório. A propriedade `width` define um tamanho horizontal fixo para a caixa:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
}
```

O acréscimo de `width: 6em` à regra de CSS encolhe o primeiro `div` horizontalmente, deixando um

espaço em branco à direita dele (Figure 38).

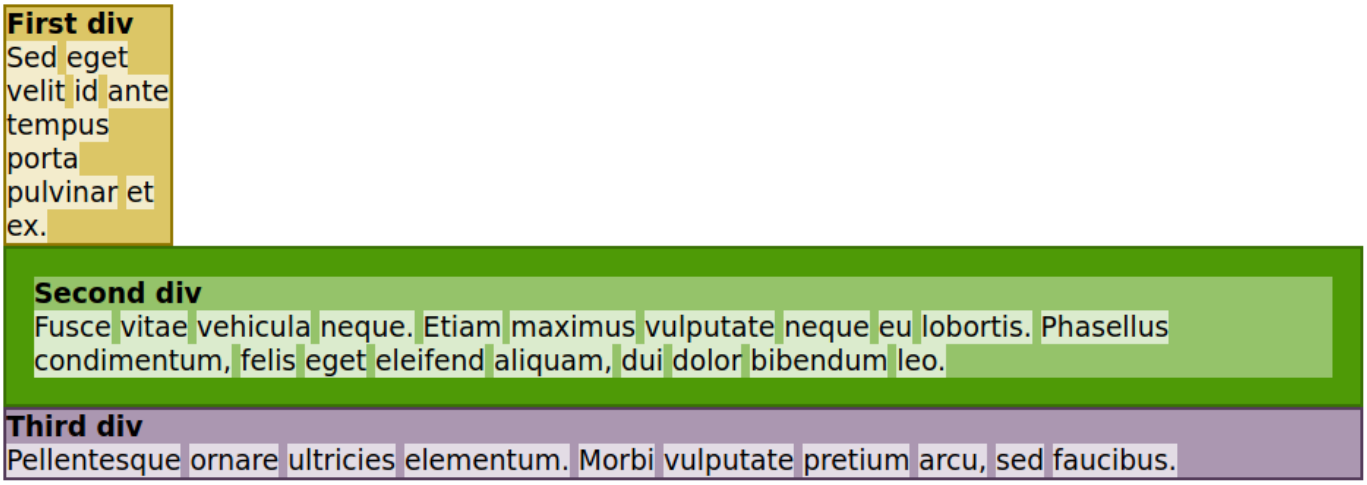

Figure 38. A propriedade `width` altera a largura horizontal do primeiro `div`.

Em vez de deixar o primeiro `div` alinhado à esquerda, vamos centralizá-lo. Centralizar uma caixa é equivalente a definir margens do mesmo tamanho em ambos os lados; assim, podemos usar a propriedade de margem para centralizá-la. O tamanho do espaço disponível costuma variar, por isso usamos o valor `auto` para as margens esquerda e direita:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  margin: 0 auto;
}
```

As margens esquerda e direita são calculadas automaticamente pelo navegador e a caixa é centralizada (Figure 39).

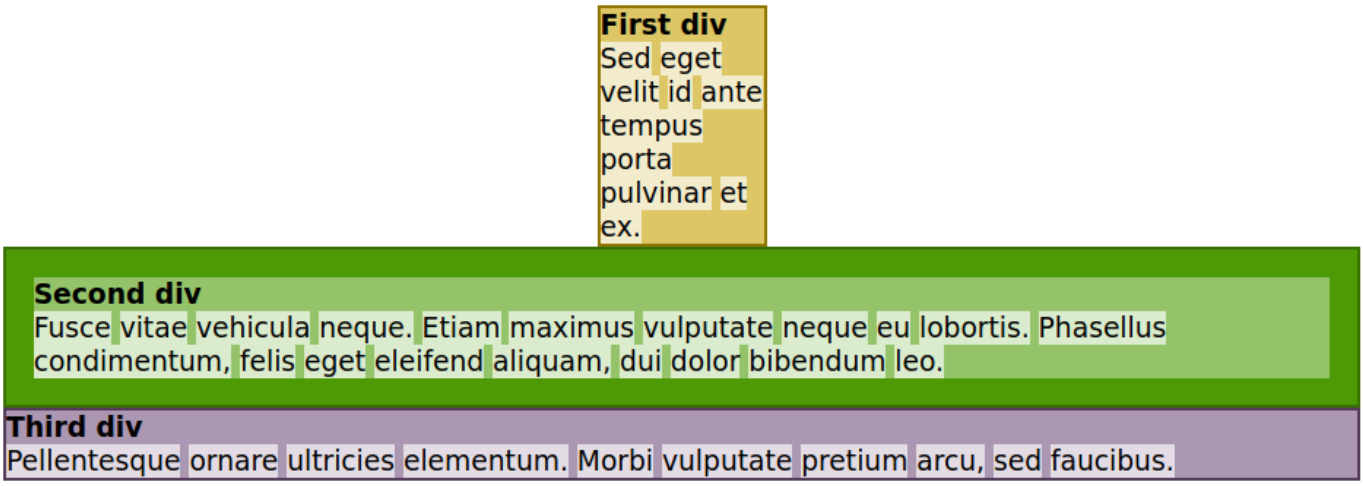

Figure 39. A propriedade `margin` é usada para centralizar o primeiro `div`.

Como é fácil constatar, ao tornar um elemento de bloco mais estreito, não abrimos mais espaço para o elemento seguinte. O fluxo natural é preservado, como se o elemento mais estreito ainda ocupasse toda a largura disponível.

Personalizando o fluxo normal

O fluxo normal é simples e sequencial. O CSS também permite interromper o fluxo normal e posicionar elementos de maneiras bastante específicas, até mesmo neutralizando a rolagem da página, se desejado. Veremos várias maneiras de controlar o posicionamento dos elementos nesta seção.

Elementos flutuantes

É possível fazer com que elementos de bloco irmãos compartilhem o mesmo espaço horizontal. Uma das maneiras é usar a propriedade `float`, que remove o elemento do fluxo normal. Como o próprio nome sugere, a propriedade `float` faz a caixa flutuar sobre os elementos do bloco que vêm depois, que são renderizados como se estivessem sob a caixa flutuante. Para fazer o primeiro `div` flutuar à direita, adicione `float: right` à regra CSS correspondente:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  float: right;
}
```

As margens automáticas são ignoradas em uma caixa flutuante, de modo que a propriedade `margin` pode ser removida. A Figure 40 mostra o resultado da flutuação à direita do primeiro `div`.

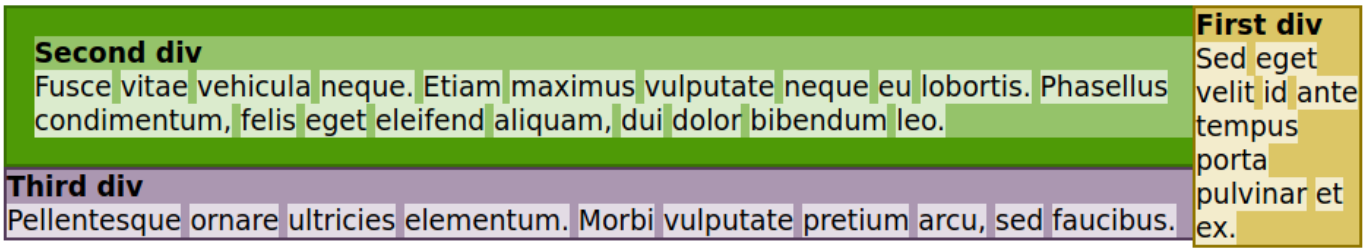

Figure 40. O primeiro `div` está flutuando e não faz parte do fluxo normal.

Por padrão, todos os elementos de bloco que vêm após o elemento flutuante ficarão abaixo dele. Portanto, se a altura for suficiente, a caixa flutuante cobrirá todos os elementos de bloco restantes.

Embora um elemento flutuante fique acima dos outros elementos de bloco, o conteúdo de linha dentro do contêiner do elemento flutuante se ajusta em torno do elemento flutuante. A inspiração para isso vem do layout de revistas e jornais, que muitas vezes ajustam o texto em torno de uma imagem, por exemplo.

A imagem anterior mostra como o primeiro `div` cobre o segundo `div` e parte do terceiro `div`. Vamos supor que queremos que o primeiro `div` flutue sobre o segundo `div`, mas não sobre o terceiro. A solução é incluir a propriedade `clear` na regra CSS correspondente ao terceiro `div`:

```
#third {
  background-color: #5c3566da;
  clear: right;
}
```

Ao definir a propriedade `clear` como `right`, o elemento correspondente pula os elementos anteriores que flutuam à direita, retomando o fluxo normal (Figure 41).

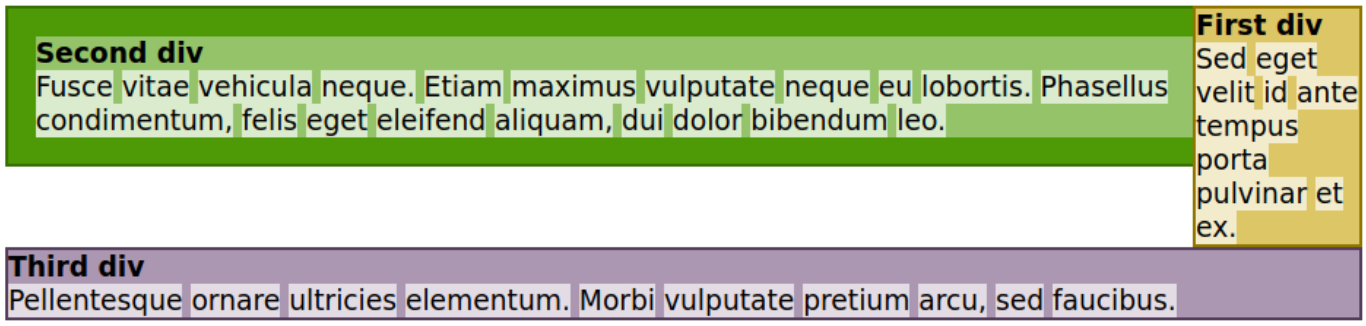

Figure 41. Usamos a propriedade `clear` para retornar ao fluxo normal.

Da mesma forma, se um elemento anterior estiver flutuando à esquerda, usamos `clear: left` para retomar o fluxo normal. Se for preciso pular elementos flutuantes à esquerda e à direita, usamos

`clear: both`.

Posicionamento das caixas

No fluxo normal, cada caixa segue as caixas anteriores na árvore do documento. Os elementos irmãos anteriores "empurram" os elementos que vêm depois deles, movendo-os para a direita e para baixo dentro de seu elemento pai. O elemento pai pode ter seus próprios irmãos fazendo o mesmo com ele. É como colocar azulejos lado a lado em uma parede, começando pelo topo.

Esse método de posicionamento das caixas é denominado *estático* e é o valor padrão da propriedade `position` do CSS. Além de definir margens e preenchimento, não há como reposicionar uma caixa estática na página.

Como os azulejos na analogia da parede, o posicionamento estático não é obrigatório. Como no caso dos azulejos, as caixas podem ser postas em qualquer lugar, inclusive cobrindo outras caixas. Para isso, atribua a propriedade `position` a um dos seguintes valores:

`relative`

O elemento segue o fluxo normal do documento, mas pode usar as propriedades `top`, `right`, `bottom` e `left` para definir deslocamentos relativos à sua posição estática original. Os deslocamentos também podem ser negativos. Os outros elementos permanecem em seus lugares originais, como se o elemento relativo ainda fosse estático.

`absolute`

O elemento ignora o fluxo normal dos outros elementos e se posiciona na página de acordo com as propriedades `top`, `right`, `bottom` e `left`. Seus valores são relativos ao corpo do documento ou a um contêiner pai não estático.

`fixed`

O elemento ignora o fluxo normal dos outros elementos e se posiciona de acordo com as propriedades `top`, `right`, `bottom` e `left`. Seus valores são relativos à janela de visualização (ou seja, a área da tela na qual o documento é exibido). Os elementos fixos não se movem quando o visitante rola o documento, mas agem como um adesivo fixado na tela.

`sticky`

O elemento segue o fluxo normal do documento. No entanto, em vez de sair da janela de visualização quando o documento é rolado, ele se fixa na posição definida pelas propriedades `top`, `right`, `bottom` e `left`. Se o valor `top` for `10px`, por exemplo, o elemento para de rolar quando atinge 10 pixels a partir do limite superior da janela de visualização. Quando isso acontece, o resto da página continua a rolar, mas o elemento aderente se comporta como um elemento fixo nessa posição. Ele volta à sua posição original quando o documento for rolado de volta na janela

de exibição. Os elementos aderentes são comumente usados hoje em dia para criar menus principais sempre visíveis.

Nos posicionamentos que aceitam as propriedades `top`, `right`, `bottom` e `left`, não é necessário usar todas elas. Se você definir as propriedades `top` e `height` de um elemento absoluto, por exemplo, o navegador calcula implicitamente a propriedade `bottom` (posição superior + altura = posição inferior).

A propriedade `display`

Se a ordem fornecida pelo fluxo normal não for um problema em seu design, mas você quiser alterar a forma como as caixas se alinham na página, modifique a propriedade `display` do elemento. A propriedade `display` pode inclusive fazer o elemento desaparecer completamente do documento renderizado, quando definida como `display: none`. Isso é útil quando desejamos exibir o elemento posteriormente usando JavaScript.

A propriedade `display` também pode, por exemplo, fazer com que um elemento de bloco se comporte como um elemento de linha (`display: inline`). No entanto, não é aconselhável fazer isso. Existem métodos melhores para colocar os elementos do contêiner lado a lado, como o *modelo flexbox*.

O modelo flexbox foi inventado para vencer as limitações dos elementos flutuantes e eliminar o uso inadequado de tabelas para estruturar o layout da página. Quando definimos a propriedade `display` de um elemento de contêiner como `flex` para transformá-lo em um contêiner flexbox, seus filhos imediatos se comportam mais ou menos como as células de uma linha de tabela.

TIP

Se quiser ter ainda mais controle sobre o posicionamento dos elementos na página, dê uma olhada no recurso de *CSS grid*. As grades são um sistema poderoso que emprega linhas e colunas para criar layouts elaborados.

Para testar a exibição flex, adicione um novo elemento `div` à página de exemplo e torne-o contêiner dos três elementos `div` existentes:

```
<div id="container">

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
  <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultricies</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->


</div><!-- #container -->
```

Adicione a seguinte regra de CSS à folha de estilo para transformar o contêiner `div` em um contêiner flexbox:

```
#container {
  display: flex;
}
```

O resultado são os três elementos `div` internos renderizados lado a lado (Figure 42).

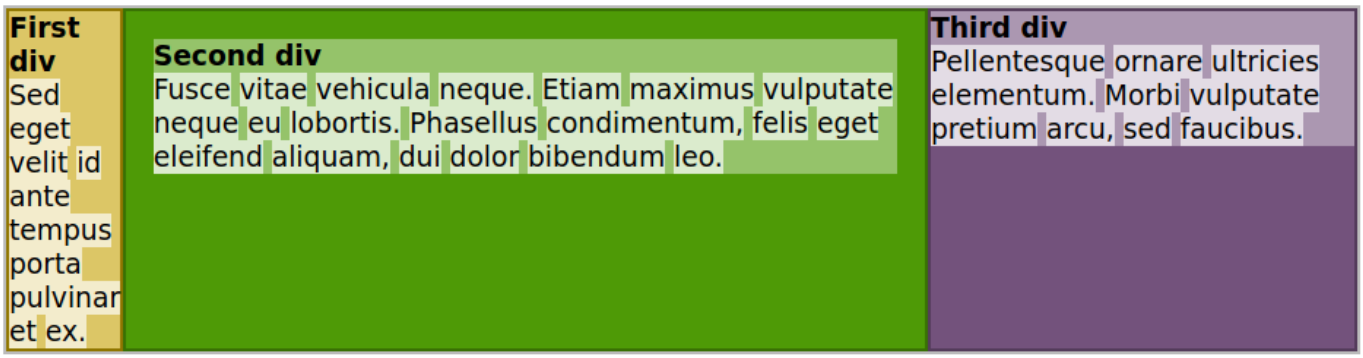

Figure 42. O modelo flexbox cria uma grade.

O valor `inline-flex` em vez de `flex` daria basicamente o mesmo resultado, mas faria com que os filhos se comportassem mais como elementos de linha.

Design responsivo

Sabemos que o CSS oferece propriedades para ajustar o tamanho dos elementos e fontes em relação à área de tela disponível. No entanto, você pode querer ir mais longe e usar um design diferente para dispositivos diferentes: por exemplo, sistemas de desktop versus dispositivos com dimensões de tela abaixo de um determinado tamanho. Essa abordagem é chamada de *responsive web design* (web design responsivo), e o CSS fornece métodos chamados *consultas de mídia* (media queries) para torná-la possível.

No exemplo anterior, modificamos o layout da página para colocar os elementos `div` lado a lado nas colunas. Esse layout é adequado para telas maiores, mas ficaria muito confuso em telas menores. Para resolver esse problema, podemos adicionar à folha de estilo uma consulta de mídia correspondente apenas a telas com pelo menos `600px` de largura:

```
@media (min-width: 600px){
  #container {
    display: flex;
  }
}
```

As regras de CSS dentro da diretiva `@media` serão usadas somente se os critérios entre parênteses forem satisfeitos. Neste exemplo, se a largura da janela de visualização for menor que `600px`, a regra não será aplicada ao contêiner `div` e seus filhos serão exibidos como elementos `div` convencionais. O navegador reavalia as consultas de mídia sempre que a dimensão da janela de visualização muda, de modo que o layout é alterado em tempo real ao se redimensionar a janela do navegador ou girar o smartphone.

Exercícios Guiados

1. Se a propriedade `position` não for modificada, qual método de posicionamento será usado pelo navegador?

2. Como ter certeza de que a caixa de um elemento será renderizada depois de qualquer elemento flutuante anterior?

3. Como usar a propriedade abreviada `margin` para definir as margens superior/inferior como `4px` e as margens direita/esquerda como `6em`?

4. Como centralizar horizontalmente um elemento de contêiner estático com largura fixa na página?

Exercícios Exploratórios

1. Escreva uma regra de CSS que corresponda ao elemento `<div class="picture">` de forma que o texto dentro dos elementos de bloco seguintes se ajustem a seu lado direito.

2. Como a propriedade `top` afeta um elemento estático em relação ao seu elemento pai?

3. De que maneira o posicionamento de um elemento é afetado no fluxo normal quando sua propriedade `display` é alterada para `flex`?

4. Qual recurso do CSS permite usar um conjunto separado de regras dependendo das dimensões da tela?

Resumo

Esta lição cobre o modelo de caixa do CSS e a maneira de personalizá-lo. Além do fluxo normal do documento, o designer pode lançar mão de diferentes mecanismos de posicionamento para implementar um layout personalizado. A lição aborda os seguintes conceitos e procedimentos:

- O fluxo normal do documento.
- Ajustes na margem e preenchimento da caixa de um elemento.
- Uso das as propriedades float e clear.
- Mecanismos de posicionamento: estático, relativo, absoluto, fixo e aderente.
- Valores alternativos para a propriedade `display`.
- Noções básicas de design responsivo.

Respostas aos Exercícios Guiados

1. Se a propriedade `position` não for modificada, qual método de posicionamento será usado pelo navegador?

   O método `static`.

2. Como ter certeza de que a caixa de um elemento será renderizada depois de qualquer elemento flutuante anterior?

   A propriedade `clear` do elemento deve estar definida como `both`.

3. Como usar a propriedade abreviada `margin` para definir as margens superior/inferior como `4px` e as margens direita/esquerda como `6em`?

   + Poderia ser `margin: 4px 6em` ou `margin: 4px 6em 4px 6em`.

4. Como centralizar horizontalmente um elemento de contêiner estático com largura fixa na página?

   Usando o valor `auto` nas propriedades `margin-left` e `margin-right` do elemento.

Respostas aos Exercícios Exploratórios

1. Escreva uma regra de CSS que corresponda ao elemento `<div class="picture">` de forma que o texto dentro dos elementos de bloco seguintes se ajustem a seu lado direito.

   ```
   .picture { float: left; }
   ```

2. Como a propriedade `top` afeta um elemento estático em relação ao seu elemento pai?

   A propriedade `top` não se aplica a elementos estáticos.

3. De que maneira o posicionamento de um elemento é afetado no fluxo normal quando sua propriedade `display` é alterada para `flex`?

   O posicionamento do elemento em si não muda, mas seus elementos filho imediatos serão renderizados lado a lado horizontalmente.

4. Qual recurso do CSS permite usar um conjunto separado de regras dependendo das dimensões da tela?

   As *consultas de mídia* (media queries) permitem ao navegador verificar as dimensões da janela de visualização antes de aplicar uma regra de CSS.

Linux
Professional
Institute

Tópico 034: Programação em JavaScript

034.1 Execução e sintaxe de JavaScript

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 034.1

Peso

1

Áreas chave de conhecimento

- Executar JavaScript em um documento HTML
- Entender a sintaxe do JavaScript
- Adicionar comentários ao código JavaScript
- Acessar o console do JavaScript
- Escrever para o console do JavaScript

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `<script>`, incluindo os atributos `type` (text/javascript) e `src`
- `//`, `/* */`
- `console.log`

034.1 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	034 Programação em JavaScript
Objetivo:	034.1 Execução e sintaxe de JavaScript
Lição:	1 de 1

Introdução

As páginas web são desenvolvidas usando três tecnologias padrão: HTML, CSS e JavaScript. O JavaScript é uma linguagem de programação que permite ao navegador atualizar dinamicamente o conteúdo do site. Ele normalmente é executado pelo mesmo navegador usado para visualizar uma página web. Dessa forma, como ocorre com o CSS e o HTML, o comportamento exato de um código pode ser diferente de acordo com o navegador. Mas os navegadores mais comuns aderem à especificação ECMAScript. Este é um padrão que unifica o uso do JavaScript na web e será a base desta lição, junto com a especificação HTML5, que especifica como o JavaScript precisa ser posto em uma página web para que um navegador possa executá-lo.

Executando o JavaScript no navegador

Para executar o JavaScript, o navegador precisa obter o código diretamente, como parte do HTML que compõe a página, ou ainda na forma de uma URL que indica a localização de um script a ser executado.

O exemplo a seguir mostra como incluir o código diretamente no arquivo HTML:

```
<html>
  <head>
  </head>
  <body>
    <h1>Website Headline</h1>
    <p>Content</p>

    <script>
      console.log('test');
    </script>

  </body>
</html>
```

O código é empacotado entre as tags `<script>` e `</script>`. Tudo o que for incluído nessas tags será executado pelo navegador diretamente ao carregar a página.

A posição do elemento `<script>` dentro da página determina quando ele será executado. Um documento HTML é analisado de cima para baixo, e a partir disso o navegador decide quando exibir os elementos na tela. No exemplo acima, as tags `<h1>` e `<p>` do website são analisadas, e provavelmente exibidas, antes de o script ser executado. Se o código JavaScript dentro da tag `<script>` for muito demorado de executar, ainda assim a página seria exibida sem problemas. Se, porém, o script tiver sido posto acima das outras tags, o visitante da página teria de esperar até que o script termine de ser executado para poder visualizar a página. Por essa razão, as tags `<script>` costumam ser posicionadas em um destes locais:

- No finalzinho do corpo do HTML, para que o script seja a última coisa a ser executada. Fazemos isso quando o código tem uma função que não seria útil sem o resto do conteúdo da página. Um exemplo seria adicionar funcionalidade a um botão, já que o botão precisa existir para que a funcionalidade faça sentido.
- Dentro do elemento `<head>` do HTML. Esse posicionamento garante que o script seja executado antes de o corpo do HTML ser analisado. Se você quiser alterar o comportamento de carregamento da página, ou tiver algo que precisa ser executado enquanto a página ainda não está totalmente carregada, coloque o script aqui. Além disso, se houver vários scripts que dependem de um script específico, coloque esse script compartilhado dentro de head para fazer com que ele seja executado antes dos outros.

Por diversos motivos, incluindo uma maior facilidade de gerenciamento, vale a pena colocar o código JavaScript em arquivos separados externos ao código HTML. Os arquivos JavaScript externos são incluídos usando uma tag `<script>` com um atributo `src`, da seguinte maneira:

```
<html>
  <head>
    <script src="/button-interaction.js"></script>
  </head>
  <body>
  </body>
</html>
```

A tag `src` informa ao navegador a localização da fonte, ou seja, o arquivo que contém o código JavaScript. O local pode ser um arquivo no mesmo servidor, como no exemplo acima, ou qualquer URL acessível pela web, como `https://www.lpi.org/example.js`. O valor do atributo `src` segue a mesma convenção da importação de arquivos CSS ou de imagem, podendo ser relativo ou absoluto. Ao encontrar uma tag de script com o atributo `src`, o navegador tentará obter o arquivo de origem usando uma solicitação HTTP `GET`. Portanto, os arquivos externos precisam estar acessíveis.

Quando usamos o atributo `src`, qualquer código ou texto posto entre as tags `<script>…</script>` é ignorado, de acordo com a especificação do HTML.

```
<html>
  <head>
    <script src="/button-interaction.js">
      console.log("test"); // <-- This is ignored
    </script>
  </head>
  <body>
  </body>
</html>
```

Existem outros atributos que podem ser adicionados à tag `script` para especificar ainda mais como o navegador deve obter os arquivos e tratá-los posteriormente. A lista a seguir detalha os atributos mais importantes:

`async`

Pode ser usado em tags `script` e instrui o navegador a buscar o script em segundo plano, de forma a não bloquear o processo de carregamento da página. O carregamento da página ainda será interrompido para o navegador analisar o script depois de obtê-lo (o que é feito imediatamente). Este atributo é booleano e, portanto, escrever a tag como `<script async src="/script.js"></script>` já é suficiente, não sendo necessário fornecer nenhum valor.

`defer`

Semelhante ao `async`, instrui o navegador a não bloquear o processo de carregamento da página enquanto busca o script. Mas, em vez disso, o navegador posterga a análise do script e espera até que todo o documento HTML tenha sido analisado e somente então analisa o script, e só então anuncia que o documento foi completamente carregado. Como no caso de `async`, `defer` é um atributo booleano e é usado da mesma maneira. Uma vez que `defer` implica em `async`, não há razão para especificar as duas tags ao mesmo tempo.

NOTE

Depois de analisar completamente uma página, o navegador indica que ela está pronta para ser exibida disparando um evento `DOMContentLoaded`, permitindo que o visitante veja o documento. Assim, o JavaScript incluído em um evento `<head>` sempre terá a chance de agir na página antes de ser exibido, mesmo com o atributo `defer`.

`type`

Indica o tipo de script que o navegador deve esperar dentro da tag. O padrão é JavaScript (`type="application/javascript"`), por isso este atributo não é necessário ao se incluir um código JavaScript ou ao se apontar para um recurso JavaScript com a tag `src`. Geralmente, todos os tipos de MIME podem ser especificados, mas apenas os scripts denotados como JavaScript serão executados pelo navegador. Existem dois casos de uso realistas para este atributo: dizer ao navegador para não executar o script definindo `type` para um valor arbitrário como `template` ou `other`, ou dizer ao navegador que o script é um módulo ES6. Não falaremos dos módulos ES6 nesta lição.

WARNING

Quando múltiplos scripts têm o atributo `async`, eles serão executados na ordem em que o download for concluído, e *não* na ordem das tags `script` do documento. O atributo `defer`, por outro lado, preserva a ordem das tags `script`.

Console do navegador

Embora ele geralmente seja executado como parte de um site, existe outra maneira de executar o JavaScript: usando o console do navegador. Todos os navegadores modernos para desktop incluem um menu que permite executar o código JavaScript no mecanismo JavaScript do navegador. O recurso permite testar um novo código ou depurar sites existentes.

Existem várias maneiras de acessar o console, dependendo do navegador. A maneira mais fácil é usar atalhos de teclado. Estes são os atalhos de teclado para alguns dos navegadores mais comuns:

Chrome

`Ctrl` + `Shift` + `J` (`Cmd` + `Option` + `J` no Mac)

Firefox

`Ctrl` + `Shift` + `K` (`Cmd` + `Option` + `K` no Mac)

Safari

`Ctrl` + `Shift` + `?` (`Cmd` + `Option` + `?` no Mac)

Você também pode clicar com o botão direito em uma página web e selecionar a opção "Inspecionar" ou "Inspecionar Elemento" para abrir o inspetor, que é outra ferramenta do navegador. Quando o inspetor é aberto, aparece um novo painel. Selecione a guia "Console" para abrir o console do navegador.

Depois de acessar o console, você pode executar o JavaScript da página inserindo o código diretamente no campo de entrada. O resultado de qualquer código executado será mostrado em uma linha separada.

Declarações de JavaScript

Agora que sabemos como começar a executar um script, vamos tratar dos fundamentos dessa execução. Um script JavaScript é uma coleção de declarações e blocos. Um exemplo de declaração é `console.log('test')`. Esta instrução diz ao navegador para enviar a palavra `test` para o console do navegador.

Cada declaração em JavaScript é encerrada por um ponto e vírgula (;). Graças a isso, o navegador sabe que a declaração foi concluída e uma nova pode ser iniciada. Veja o seguinte script:

```
var message = "test"; console.log(message);
```

Escrevemos duas declarações. Cada uma delas é encerrada por um ponto e vírgula ou pelo final do script. Para fins de legibilidade, podemos colocar as declarações em linhas separadas. Dessa forma, o script também pode ser escrito assim:

```
var message = "test";
console.log(message);
```

Isso é possível porque todos os espaços em branco entre as declarações, como um espaço, uma nova linha ou uma tabulação, são ignorados. Também é possível incluir espaços em branco entre palavras-

chave individuais dentro das declarações, mas isso será explicado em uma lição posterior. As declarações também podem estar vazias ou ser compostas apenas por espaços em branco.

Se uma declaração for inválida por não ter sido encerrada por um ponto e vírgula, o ECMAScript tenta inserir automaticamente os pontos e vírgulas adequados com base em um conjunto complexo de regras. A regra mais importante é: Se uma instrução inválida for composta de duas declarações válidas separadas por uma nova linha, insira um ponto e vírgula na nova linha. Por exemplo, o código a seguir não forma uma declaração válida:

```
console.log("hello")
console.log("world")
```

Mas um navegador moderno irá executá-lo automaticamente como se tivesse sido escrito com os pontos e vírgulas adequados:

```
console.log("hello");
console.log("world");
```

Assim, é possível omitir o ponto e vírgula em certos casos. No entanto, como as regras para a inserção automática de ponto e vírgula são complexas, recomendamos sempre encerrar adequadamente suas declarações para evitar erros indesejados.

Comentários em JavaScript

Scripts extensos podem se tornar bastante intricados. É aconselhável comentar o código conforme ele é escrito, tornando-o mais fácil de ler para outras pessoas ou para você mesmo no futuro. Como alternativa, você pode incluir metainformações no script, como dados de copyright ou informações sobre quando o script foi escrito e por quê.

Para possibilitar a inclusão dessas metainformações, o JavaScript suporta *comentários*. Um desenvolvedor pode incluir caracteres especiais em um script, indicando certas partes como comentários que serão ignorado na execução. O exemplo a seguir traz uma versão bem comentada do script que vimos anteriormente.

```
/*
 This script was written by the author of this lesson in May, 2020.
 It has exactly the same effect as the previous script, but includes comments.
*/

// First, we define a message.
var message = "test";

console.log(message); // Then, we output the message to the console.
```

Os comentários não são declarações e não precisam ser encerrados com ponto e vírgula. Eles seguem suas próprias regras de encerramento, dependendo da forma como o comentário é escrito. Existem duas maneiras de escrever comentários em JavaScript:

Comentário multilinhas

Use `/*` e `*/` para indicar o início e o fim de um comentário multilinhas. Tudo o que vier após `/*`, até a primeira ocorrência de `*/`, é ignorado. Esse tipo de comentário geralmente abrange mais de uma linha, mas também pode ser usado para linhas únicas ou mesmo no meio de uma linha, desta forma:

```
console.log(/* what we want to log: */ "hello world")
```

Como o objetivo dos comentários geralmente é aumentar a legibilidade de um script, evite usar esse estilo de comentário dentro de uma linha.

Comentário de uma linha

Use `//` (duas barras) para *comentar* uma linha. Tudo o que vem após a barra dupla na mesma linha é ignorado. No exemplo mostrado anteriormente, esse padrão é usado primeiro para comentar uma linha inteira. Após a declaração `console.log(message);`, ele é usado para escrever um comentário sobre o resto da linha

Em geral, comentários de uma linha devem ser usados para linhas únicas e comentários multilinha para linhas múltiplas, mesmo sendo possível usá-los de outras maneiras. Evite incluir comentários dentro das declarações.

Os comentários também podem ser usados para remover temporariamente as linhas do código em si, da seguinte maneira:

```
// We temporarily want to use a different message
// var message = "test";
var message = "something else";
```

Exercícios Guiados

1. Crie uma variável chamada `ColorName` e atribua a ela o valor `RED`.

2. Quais dos scripts a seguir são válidos?

`console.log("hello")` `console.log("world");`	
`console.log("hello");` `console.log("world");`	
`// console.log("hello")` `console.log("world");`	
`console.log("hello");` `console.log("world") //;`	
`console.log("hello"); /*` `console.log("world") */`	

Exercícios Exploratórios

1. Quantas instruções JavaScript podem ser escritas em uma única linha sem usar um ponto e vírgula?

2. Crie duas variáveis chamadas `x` e `y` e imprima a soma delas no console.

Resumo

Nesta lição, aprendemos diferentes maneiras de executar códigos em JavaScript e como modificar o comportamento de carregamento do script. Também vimos os conceitos básicos de composição de script e comentários e aprendemos a usar o comando `console.log()`.

HTML usado nesta lição:

<script>

A tag `script` pode ser usada para incluir JavaScript diretamente na página. Também é possível especificar um arquivo externo com o atributo `src`. Modifique a forma como o script é carregado com os atributos `async` e `defer`.

Conceitos de JavaScript apresentados nesta lição:

`;`

O ponto e vírgula é usado para separar declarações. O ponto e vírgula às vezes pode — mas não deve — ser omitido.

`//`, `/*…*/`

Comentários podem ser usados para adicionar explicações ou metainformações a um arquivo de script, ou para evitar que certas declarações sejam executadas.

`console.log("text")`

O comando `console.log()` pode ser usado para enviar texto para o console do navegador.

Respostas aos Exercícios Guiados

1. Crie uma variável chamada `ColorName` e atribua a ela o valor `RED`.

```
var ColorName = "RED";
```

2. Quais dos scripts a seguir são válidos?

`console.log("hello")` `console.log("world");`	Inválido: O primeiro `console.log()` não foi encerrado apropriadamente e a linha como um todo não forma uma declaração válida.
`console.log("hello");` `console.log("world");`	Válido: As declarações são todas encerradas corretamente.
`// console.log("hello")` `console.log("world");`	Válido: O código inteiro é ignorado por se tratar de um comentário.
`console.log("hello");` `console.log("world") //;`	Inválido: Falta um ponto e vírgula na última declaração. O ponto e vírgula no finalzinho é ignorado porque está comentado.
`console.log("hello"); /*` `console.log("world") */`	Válido: Uma declaração válida é seguida por um código comentado, que é ignorado.

Respostas aos Exercícios Exploratórios

1. Quantas instruções JavaScript podem ser escritas em uma única linha sem usar um ponto e vírgula?

   Se estivermos no final de um script, podemos escrever uma declaração e ela será encerrada no final do arquivo. Caso contrário, não se pode escrever uma declaração sem um ponto e vírgula com a sintaxe que vimos até aqui.

2. Crie duas variáveis chamadas `x` e `y` e imprima a soma delas no console.

   ```
   var x = 5;
   var y = 10;
   console.log(x+y);
   ```

034.2 Estruturas de dados em JavaScript

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 034.2

Peso

3

Áreas chave de conhecimento

- Definir e usar variáveis e constantes
- Entender tipos de dados
- Entender a conversão/coerção de tipo
- Entender matrizes (arrays) e objetos
- Noções de escopo de variáveis

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `=`, `+`, `-`, `*`, `/`, `%`, `--`, `++`, `+=`, `-=`, `*=`, `/=`
- `var`, `let`, `const`
- `boolean`, `number`, `string`, `symbol`
- `array`, `object`
- `undefined`, `null`, `NaN`

034.2 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	034 Programação em JavaScript
Objetivo:	034.2 Estruturas de dados em JavaScript
Lição:	1 de 1

Introdução

As linguagens de programação, como as linguagens naturais, representam a realidade por meio de símbolos que são combinados em declarações imbuídas de significado. A realidade representada por uma linguagem de programação são os recursos da máquina, como as operações do processador, os dispositivos e a memória.

Cada uma das muitas linguagens de programação adota um paradigma para representar as informações. O JavaScript utiliza as convenções típicas das linguagens de *alto nível*, nas quais a maioria dos detalhes, como a alocação de memória, está implícita, permitindo que o programador se concentre na finalidade do script no contexto do aplicativo.

Linguagens de alto nível

Linguagens de alto nível fornecem regras abstratas que permitem ao programador escrever menos código para expressar uma ideia. O JavaScript oferece maneiras convenientes para aproveitar a memória do computador, com conceitos de programação que simplificam a escrita de práticas recorrentes e que geralmente bastam para as finalidades de um desenvolvedor web.

NOTE Embora seja possível empregar mecanismos especializados para um acesso meticuloso à memória, os tipos mais simples de dados que veremos são de uso mais geral.

As operações típicas de um aplicativo web consistem em solicitar dados por meio de alguma instrução JavaScript e armazená-los para que sejam processados e, em seguida, apresentados ao usuário. Esse armazenamento é bastante flexível em JavaScript, com formatos adequados a cada finalidade.

Declaração de constantes e variáveis

A declaração de constantes e variáveis para armazenar dados é a pedra angular de qualquer linguagem de programação. O JavaScript adota a convenção da maioria das linguagens de programação, atribuindo valores a constantes ou variáveis com a sintaxe `name = value`. A constante ou variável à esquerda assume o valor mostrado à direita. O nome da constante ou variável deve começar com uma letra ou sublinhado (underscore).

O tipo de dados armazenados na variável não precisa ser indicado, pois o JavaScript é uma linguagem de *tipagem dinâmica*. O tipo da variável é inferido a partir do valor atribuído a ela. Porém, é aconselhável designar certos atributos na declaração para garantir o resultado esperado.

NOTE O TypeScript é uma linguagem inspirada no JavaScript que, como as linguagens de baixo nível, permite declarar variáveis para tipos específicos de dados.

Constantes

Uma constante é um símbolo atribuído uma única vez, quando o programa é iniciado, mantendo-se sempre igual. As constantes são úteis para especificar valores fixos, por exemplo a constante `PI` como 3,14159265 ou `COMPANY_NAME` como o nome de sua empresa.

Assim, por exemplo, em um aplicativo web poderíamos ter um cliente que recebe informações meteorológicas de um servidor remoto. O programador pode decidir que o endereço do servidor deve ser constante, pois ele não mudará durante a execução da aplicação. As informações de temperatura, no entanto, podem ser diferentes a cada chegada de novos dados do servidor.

O intervalo entre as consultas feitas ao servidor também pode ser definido como uma constante, que pode ser consultada de qualquer parte do programa:

```
const update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Quando invocada, a função `setup_app()` exibe a mensagem `Update every 10 minutes` no console. O termo `const`, colocado antes do nome `update_interval`, garante que seu valor permanecerá o mesmo durante toda a execução do script. Se for feita uma tentativa de redefinir o valor de uma constante, é emitido um erro `TypeError: Assignment to constant variable`.

Variáveis

Sem o termo `const`, o JavaScript pressupõe automaticamente que `update_interval` é uma variável e que seu valor pode ser modificado. Isso equivale a declarar a variável explicitamente com `var`:

```
var update_interval;
update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Observe que, embora a variável `update_interval` tenha sido definida fora da função, ela foi acessada de dentro da função. Qualquer constante ou variável declarada fora de funções ou blocos de código definidos por chaves ({}) tem *escopo global*; ou seja, pode ser acessada de qualquer parte do código. O oposto não é verdadeiro: uma constante ou variável declarada dentro de uma função tem *escopo local*, sendo acessível apenas de dentro da própria função. Os blocos de código delimitados por chaves, como aqueles colocados em estruturas de decisão `if` ou loops `for`, delimitam o escopo das constantes, mas não das variáveis declaradas como `var`. O código a seguir, por exemplo, é válido:

```
var success = true;
if ( success == true )
{
  var message = "Transaction succeeded";
  var retry = 0;
}
else
{
  var message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

A declaração `console.log(message)` é capaz de acessar a variável `message`, mesmo que ela tenha sido declarada dentro do bloco de código da estrutura `if`. O mesmo não aconteceria se `message` fosse uma constante, conforme exemplificado a seguir:

```
var success = true;
if ( success == true )
{
  const message = "Transaction succeeded";
  var retry = 0;
}
else
{
  const message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

Nesse caso, apareceria uma mensagem de erro do tipo `ReferenceError: message is not defined` e o script seria interrompido. Embora isso pareça uma limitação, restringir o escopo de variáveis e constantes ajuda a evitar confusão entre as informações processadas no corpo do script e em seus diferentes blocos de código. Por esse motivo, as variáveis declaradas com `let` em vez de `var` também têm escopo restrito aos blocos delimitados por chaves. Há outras diferenças sutis entre as variáveis declaradas com `var` ou `let`, mas a mais significativa diz respeito ao escopo da variável, como mostrado aqui.

Tipos de valores

Na maioria das vezes, o programador não precisa se preocupar com o tipo de dados armazenados em uma variável, pois o JavaScript os identifica automaticamente com um de seus tipos *primitivos* durante a primeira atribuição de um valor à variável. Algumas operações, no entanto, podem ser específicas para um certo tipo de dados e podem resultar em erros quando usadas sem critério. Além disso, o JavaScript oferece tipos *estruturados*, que permitem combinar mais de um tipo primitivo em um único objeto.

Tipos primitivos

Os tipos primitivos correspondem às variáveis tradicionais, que armazenam apenas um valor. Os tipos são definidos implicitamente, de modo que o operador `typeof` pode ser usado para identificar o tipo de valor que é armazenado em uma variável:

```
console.log("Undefined variables are of type", typeof variable);

{
  let variable = true;
  console.log("Value `true` is of type " + typeof variable);
}

{
  let variable = 3.14159265;
  console.log("Value `3.14159265` is of type " + typeof variable);
}

{
  let variable = "Text content";
  console.log("Value `Text content` is of type " + typeof variable);
}

{
  let variable = Symbol();
  console.log("A symbol is of type " + typeof variable);
}
```

Este script exibirá no console o tipo de variável usada em cada caso:

```
undefined variables are of type undefined
Value `true` is of type boolean
Value `3.114159265` is of type number
Value `Text content` is of type string
A symbol is of type symbol
```

Note que a primeira linha tenta encontrar o tipo de uma variável não declarada. Isso faz com que a variável dada seja identificada como `undefined`. O tipo `symbol` é o primitivo menos intuitivo. Sua finalidade é fornecer um nome de atributo único dentro de um objeto quando não existir a necessidade de definir um nome de atributo específico. Um objeto é uma das estruturas de dados que veremos a seguir.

Tipos estruturados

Embora os tipos primitivos bastem para escrever rotinas simples, seu uso exclusivo nem sempre é indicado em aplicativos mais complexos. Um aplicativo de comércio eletrônico, por exemplo, seria muito mais difícil de escrever, porque o programador precisaria encontrar maneiras de armazenar listas de itens e seus valores correspondentes usando apenas variáveis com tipos primitivos.

Os tipos estruturados simplificam a tarefa de agrupar informações da mesma natureza em uma única variável. Uma lista de itens em um carrinho de compras, por exemplo, pode ser armazenada em uma única variável do tipo *matriz* (array):

```
let cart = ['Milk', 'Bread', 'Eggs'];
```

Conforme demonstrado no exemplo, uma matriz de itens é designada entre colchetes. O exemplo preencheu a matriz com três valores de string literais, por isso o uso de aspas simples. As variáveis também podem ser usadas como itens em uma matriz, mas nesse caso devem ser designadas sem aspas. O número de itens em uma matriz pode ser consultado com a propriedade `length`:

```
let cart = ['Milk', 'Bread', 'Eggs'];
console.log(cart.length);
```

O número `3` será exibido na saída do console. Novos itens podem ser adicionados à matriz com o método `push()`:

```
cart.push('Candy');
console.log(cart.length);
```

Desta vez, a quantidade exibida será `4`. Cada item da lista pode ser acessado por seu índice numérico, começando com `0`:

```
console.log(cart[0]);
console.log(cart[3]);
```

A saída exibida no console será:

```
Milk
Candy
```

Assim como se pode usar `push()` para adicionar um elemento, usamos `pop()` para remover o último elemento de uma matriz.

Os valores armazenados em uma matriz não precisam ser do mesmo tipo. É possível, por exemplo, armazenar a quantidade de cada item ao lado dele. Uma lista de compras como a do exemplo anterior pode ser construída da seguinte forma:

```
let cart = ['Milk', 1, 'Bread', 4, 'Eggs', 12, 'Candy', 2];

// Item indexes are even
let item = 2;

// Quantities indexes are odd
let quantity = 3;

console.log("Item: " + cart[item]);
console.log("Quantity: " + cart[quantity]);
```

A saída exibida no console após a execução deste código é:

```
Item: Bread
Quantity: 4
```

Como você já deve ter notado, combinar os nomes dos itens com suas respectivas quantidades em uma única matriz pode não ser uma boa ideia, porque a relação entre eles não é explícita na estrutura de dados e é muito suscetível a erros (humanos). Mesmo se usássemos uma matriz para os nomes e outra para as quantidades, manter a integridade da lista exigiria o mesmo cuidado e não seria muito produtivo. Nessas situações, a melhor alternativa é usar uma estrutura de dados mais apropriada: um

objeto.

No JavaScript, uma estrutura de dados de tipo objeto permite vincular propriedades a uma variável. Além disso, ao contrário de uma matriz, os elementos que constituem um objeto não têm uma ordem fixa. Um item de lista de compras, por exemplo, pode ser um objeto com as propriedades `name` e `quantity` (nome e quantidade):

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);
```

Este exemplo mostra que um objeto pode ser definido usando chaves ({}), onde cada par de propriedade/valor é separado por dois pontos e as propriedades são separadas por vírgulas. A propriedade está acessível no formato *variable.property*, como em `item.name`, tanto para leitura quanto para atribuição de novos valores. A saída exibida no console após a execução deste código é:

```
Item: Milk
Quantity: 1
```

Finalmente, cada objeto que representa um item pode ser incluído na matriz da lista de compras. Isso pode ser feito diretamente ao se criar a lista:

```
let cart = [{ name: 'Milk', quantity: 1 }, { name: 'Bread', quantity: 4 }];
```

Como anteriormente, um novo objeto que representa um item pode ser adicionado posteriormente à matriz:

```
cart.push({ name: 'Eggs', quantity: 12 });
```

Os itens na lista agora podem ser acessados por seu índice e seu nome de propriedade:

```
console.log("Third item: " + cart[2].name);
console.log(cart[2].name + " quantity: " + cart[2].quantity);
```

A saída exibida no console após a execução deste código é:

```
third item: eggs
Eggs quantity: 12
```

As estruturas de dados permitem que o programador mantenha o código muito mais organizado e fácil de manter, seja pelo autor original ou por outros programadores da equipe. Além disso, muitas saídas de funções JavaScript estão em tipos estruturados, que precisam ser manipulados adequadamente pelo programador.

Operadores

Até aqui, praticamente apenas vimos como atribuir valores a variáveis recém-criadas. Qualquer programa, por mais simples que seja, realizará diversas outras manipulações nos valores das variáveis. O JavaScript oferece muitos tipos de *operadores* capazes de atuar diretamente no valor de uma variável ou armazenar o resultado da operação em uma nova variável.

A maioria dos operadores é orientada para operações aritméticas. Para aumentar a quantidade de um item na lista de compras, por exemplo, basta usar o operador de adição `+`:

```
item.quantity = item.quantity + 1;
```

O trecho de código a seguir imprime o valor de `item.quantity` antes e depois da adição. Não confunda as funções do sinal de mais no trecho. As declarações `console.log` usam um sinal de mais para combinar duas strings.

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);

item.quantity = item.quantity + 1;
console.log("New quantity: " + item.quantity);
```

A saída exibida no console após a execução deste código é:

```
Item: Milk
Quantity: 1
New quantity: 2
```

Observe que o valor anteriormente armazenado em `item.quantity` é usado como operando da

operação de adição: `item.quantity = item.quantity + 1`. Somente após a conclusão da operação, o valor em `item.quantity` é atualizado com o resultado. Esse tipo de operação aritmética envolvendo o valor atual da variável de destino é bastante comum, e assim existem operadores abreviados que permitem escrever a mesma operação em um formato reduzido:

```
item.quantity += 1;
```

As outras operações básicas também têm operadores abreviados equivalentes:

- `a = a - b` equivale a `a -= b`.
- `a = a * b` equivale a `a *= b`.
- `a = a / b` equivale a `a /= b`.

Para adição e subtração, existe um terceiro formato disponível quando o segundo operando é apenas uma unidade:

- `a = a + 1` equivale a `a++`.
- `a = a - 1` equivale a `a--`.

É possível combinar mais de um operador na mesma operação e armazenar o resultado em uma nova variável. Por exemplo, a seguinte declaração calcula o preço total de um item mais o custo de envio:

```
let total = item.quantity * 9.99 + 3.15;
```

A ordem em que as operações são realizadas segue a ordem tradicional de precedência: primeiro são feitas as operações de multiplicação e divisão e em seguida as operações de adição e subtração. Os operadores com a mesma precedência são executados na ordem em que aparecem na expressão, da esquerda para a direita. Para substituir a ordem de precedência padrão, podemos usar parênteses, como em `a * (b + c)`.

Em algumas situações, o resultado de uma operação nem precisa ser armazenado em uma variável. Esse é o caso quando queremos avaliar o resultado de uma expressão dentro de uma instrução `if`:

```
if ( item.quantiy % 2 == 0 )
{
  console.log("Quantity for the item is even");
}
else
{
  console.log("Quantity for the item is odd");
}
```

O operador `%` (módulo) retorna o restante da divisão do primeiro operando pelo segundo operando. No exemplo, a instrução `if` verifica se o resto da divisão de `item.quantity` por `2` é igual a zero, ou seja, se `item.quantity` é um múltiplo de 2.

Quando um dos operandos do operador `+` é uma string, os outros operadores sofrem uma *coerção* em strings e o resultado é uma concatenação de strings. Nos exemplos anteriores, esse tipo de operação foi usado para concatenar strings e variáveis no argumento da declaração `console.log`.

Essa conversão automática pode não ser o comportamento desejado. Um valor fornecido pelo usuário em um campo de formulário, por exemplo, pode ser identificado como uma string, mas na verdade é um valor numérico. Em casos como esse, a variável deve primeiro ser convertida em um número com a função `Number()`:

```
sum = Number(value1) + value2;
```

Além disso, é importante verificar se o usuário forneceu um valor válido antes de prosseguir com a operação. Em JavaScript, uma variável sem um valor atribuído contém o valor `null`. Isso permite que o programador use uma declaração de decisão como `if (value1 == null)` para verificar se uma variável teve um valor atribuído, independentemente do tipo de valor atribuído à variável.

Exercícios Guiados

1. Um array (matriz) é uma estrutura de dados presente em várias linguagens de programação, das quais algumas permitem apenas matrizes com itens do mesmo tipo. No caso do JavaScript, é possível definir uma matriz com itens de tipos diferentes?

2. Com base no exemplo `let item = { name: 'Milk', quantity: 1 }` para um objeto em uma lista de compras, como esse objeto poderia ser declarado de forma a incluir o preço do item?

3. Em uma única linha de código, quais são as maneiras de atualizar o valor de uma variável para a metade de seu valor atual?

Exercícios Exploratórios

1. No código a seguir, qual valor será exibido na saída do console?

```
var value = "Global";

{
  value = "Location";
}

console.log(value);
```

2. O que acontece quando um ou mais operandos envolvidos em uma operação de multiplicação é uma string?

3. Como é possível remover o item `Eggs` da matriz `cart` declarada com `let cart = ['Milk', 'Bread', 'Eggs']`?

Resumo

Esta lição cobre o uso básico de constantes e variáveis em JavaScript. O JavaScript é uma *linguagem de tipagem dinâmica*; assim, o programador não precisa especificar o tipo de variável antes de definilo. No entanto, é importante conhecer os tipos primitivos da linguagem para garantir o resultado correto das operações básicas. Além disso, estruturas de dados como matrizes e objetos combinam tipos primitivos e permitem ao programador construir variáveis compostas mais complexas. Esta lição aborda os seguintes conceitos e procedimentos:

- Compreender constantes e variáveis
- Escopo de uma variável
- Declaração de variáveis com `var` e `let`
- Tipos primitivos
- Operadores aritméticos
- Matrizes e objetos
- Coerção e conversão de tipo

Respostas aos Exercícios Guiados

1. Um array (matriz) é uma estrutura de dados presente em várias linguagens de programação, das quais algumas permitem apenas matrizes com itens do mesmo tipo. No caso do JavaScript, é possível definir uma matriz com itens de tipos diferentes?

   Sim, em JavaScript é possível definir matrizes com itens de diferentes tipos primitivos, como strings e números.

2. Com base no exemplo `let item = { name: 'Milk', quantity: 1 }` para um objeto em uma lista de compras, como esse objeto poderia ser declarado de forma a incluir o preço do item?

   ```
   let item = { name: 'Milk', quantity: 1, price: 4.99 };
   ```

3. Em uma única linha de código, quais são as maneiras de atualizar o valor de uma variável para a metade de seu valor atual?

   Pode-se usar a própria variável como um operando, `value = value / 2`, ou o operador abreviado `/=`: `value /= 2`.

Respostas aos Exercícios Exploratórios

1. No código a seguir, qual valor será exibido na saída do console?

```
var value = "Global";

{
  value = "Location";
}

console.log(value);
```

   `Location`

2. O que acontece quando um ou mais operandos envolvidos em uma operação de multiplicação é uma string?

   O JavaScript atribuirá o valor `NaN` (Not a Number) ao resultado, indicando que a operação é inválida.

3. Como é possível remover o item `Eggs` da matriz `cart` declarada com `let cart = ['Milk', 'Bread', 'Eggs']`?

   As matrizes em Javascript incluem o método `pop()`, que remove o último item da lista: `cart.pop()`.

034.3 Estrututras de controle e funções do JavaScript

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 034.3

Peso

4

Áreas chave de conhecimento

- Entender valores verdadeiros e falsos
- Compreender os operadores de comparação
- Entender a diferença entre comparações comuns e estritas
- Usar condicionais
- Usar loops
- Definir funções personalizadas

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `if`, `else if`, `else`
- `switch`, `case`, `break`
- `for`, `while`, `break`, `continue`
- `function`, `return`
- `==`, `!=`, `<`, `<=`, `>`, `>=`
- `===`, `!==`

034.3 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	034 Programação em JavaScript
Objetivo:	034.3 Estruturas de controle e funções do JavaScript
Lição:	1 de 2

Introdução

Como qualquer outra linguagem de programação, o código JavaScript é uma coleção de declarações que informam a um interpretador de instruções o que fazer, em ordem sequencial. No entanto, isso não significa que todas as declarações devam ser executadas apenas uma vez ou que devam ser executadas obrigatoriamente. A maioria das declarações deve ser executada apenas quando determinadas condições específicas forem atendidas. Mesmo quando um script é disparado de forma assíncrona por eventos independentes, muitas vezes ele precisa verificar uma série de variáveis de controle para encontrar a parte certa do código a ser executada.

Declarações If

A estrutura de controle mais simples é fornecida pela instrução `if`, que executará a instrução imediatamente posterior a ela se a condição especificada for verdadeira. O JavaScript considera as condições como verdadeiras se o valor avaliado for diferente de zero. Qualquer coisa entre parênteses após a palavra `if` (os espaços são ignorados) será interpretado como uma condição. No exemplo a seguir, o número literal `1` é a condição:

```
if ( 1 ) console.log("1 is always true");
```

O número `1` foi explicitamente escrito nesta condição de exemplo, então ele é tratado como um valor constante (permanece o mesmo durante a execução do script) e sempre resultará em verdadeiro quando usado como uma expressão condicional. A palavra `true` (sem as aspas duplas) também pode ser usada no lugar de `1`, pois também é tratada como um valor verdadeiro literal pela linguagem. A instrução `console.log` imprime seus argumentos na *janela do console* do navegador.

TIP

O console do navegador exibe erros, avisos e mensagens informativas enviadas com a instrução `console.log` do JavaScript. No Chrome, a combinação de teclas `Ctrl` + `Shift` + `J` (`Cmd` + `Option` + `J` no Mac) abre o console. No Firefox, a combinação `Ctrl` + `Shift` + `K` (`Cmd` + `Option` + `K` no Mac) abre a guia do console nas ferramentas do desenvolvedor.

Embora sintaticamente correto, o uso de expressões constantes em condições não é muito útil. Em uma aplicação real, é aconselhável testar a veracidade de uma variável:

```
let my_number = 3;
if ( my_number ) console.log("The value of my_number is", my_number, "and it yields
true");
```

O valor atribuído à variável `my_number` (3) é diferente de zero e, portanto, resulta verdadeiro. Mas este exemplo não é de uso comum, porque é raro ser preciso testar se um número é igual a zero. É muito mais comum comparar um valor com outro e testar se o resultado é verdadeiro:

```
let my_number = 3;
if ( my_number == 3 ) console.log("The value of my_number is", my_number, "indeed
");
```

O operador de comparação duplo igual é usado aqui porque o operador igual já está definido como operador de atribuição. O valor em cada lado do operador é chamado de *operando*. A ordem dos operandos não importa e qualquer expressão que retorne um valor pode ser um operando. Eis uma lista de outros operadores de comparação disponíveis:

`value1 == value2`

: True if `value1` is equal to `value2`.

`value1 != value2`

True if `value1` is not equal to `value2`.

`value1 < value2`

True if `value1` is less than `value2`.

`value1 > value2`

True if `value1` is greater than `value2`.

`value1 <= value2`

True if `value1` is less than or equal to `value2`.

`value1 >= value2`

True if `value1` is grater than or equal to `value2`.

Normalmente, não importa se o operando à esquerda do operador é uma string e o operando à direita é um número, desde que o JavaScript seja capaz de converter a expressão em uma comparação significativa. Portanto, a string que contém o caractere `1` será tratada como o número 1 quando comparada a uma variável numérica. Para garantir que a expressão seja verdadeira apenas se ambos os operandos forem exatamente do mesmo tipo e valor, o operador de igualdade estrita `===` deve ser usado em vez de `==`. Da mesma forma, o operador de desigualdade estrita `!==` é avaliado como verdadeiro se o primeiro operando não for exatamente do mesmo tipo e valor do segundo operador.

Opcionalmente, a estrutura de controle `if` pode executar uma instrução alternativa quando a expressão for avaliada como falsa:

```
let my_number = 4;
if ( my_number == 3 ) console.log("The value of my_number is 3");
else console.log("The value of my_number is not 3");
```

A instrução `else` deve seguir imediatamente a instrução `if`. Até agora, estamos executando apenas uma declaração quando a condição é atendida. Para executar mais de uma declaração, devemos colocá-las entre chaves:

```
let my_number = 4;
if ( my_number == 3 )
{
  console.log("The value of my_number is 3");
  console.log("and this is the second statement in the block");
}
else
{
  console.log("The value of my_number is not 3");
  console.log("and this is the second statement in the block");
}
```

Um grupo de uma ou mais declarações delimitadas por um par de chaves é conhecido como *declaração de bloco*. É comum usar declarações de bloco mesmo quando há apenas uma instrução a executar, a fim de tornar o estilo de codificação consistente em todo o script. Além disso, o JavaScript não exige que as chaves ou quaisquer declarações estejam em linhas separadas, mas esse procedimento melhora a legibilidade e facilita a manutenção do código.

As estruturas de controle podem ser aninhadas umas nas outras, mas é importante não misturar as chaves de abertura e fechamento de cada declaração de bloco:

```
let my_number = 4;

if ( my_number > 0 )
{
  console.log("The value of my_number is positive");

  if ( my_number % 2 == 0 )
  {
    console.log("and it is an even number");
  }
  else
  {
    console.log("and it is an odd number");
  }
} // end of if ( my_number > 0 )
else
{
  console.log("The value of my_number is less than or equal to 0");
  console.log("and I decided to ignore it");
}
```

As expressões avaliadas pela instrução `if` podem ser mais elaboradas do que simples comparações. Este é o caso do exemplo anterior, no qual a expressão aritmética `my_number % 2` foi empregada entre parênteses no `if` aninhado. O operador `%` retorna o resto após dividir o número à sua esquerda pelo número à sua direita. Os operadores aritméticos, como `%`, têm precedência sobre os operadores de comparação como `==`, de forma que a comparação usará o resultado da expressão aritmética como operando esquerdo.

Em muitas situações, é possível combinar estruturas condicionais aninhadas em uma única estrutura usando *operadores lógicos*. Se estivéssemos interessados apenas em números pares positivos, por exemplo, uma única estrutura `if` poderia ser usada:

```
let my_number = 4;

if ( my_number > 0 && my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("and it is an even number");
}
else
{
  console.log("The value of my_number either 0, negative");
  console.log("or it is a negative number");
}
```

O operador "e" comercial duplo `&&` na expressão avaliada é o operador lógico *AND*. Ela é avaliada como verdadeira apenas se a expressão à sua esquerda e a expressão à sua direita forem avaliadas como verdadeiras. Se você quiser combinar números que sejam ou positivos ou pares, o operador `||` deve ser usado, representando o operador lógico *OR*:

```
let my_number = -4;

if ( my_number > 0 || my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("or it is a even negative number");
}
```

Neste exemplo, apenas os números ímpares negativos não corresponderão aos critérios impostos pela expressão composta. Se você tiver a intenção oposta, ou seja, encontrar correspondências apenas dentre os números ímpares negativos, adicione o operador lógico *NOT* `!` ao início da expressão:

```
let my_number = -5;

if ( ! ( my_number > 0 || my_number % 2 == 0 ) )
{
  console.log("The value of my_number is an odd negative number");
}
```

O acréscimo do parêntese na expressão composta força a expressão que está entre eles a ser avaliada primeiro. Sem esses parênteses, o operador NOT se aplicaria apenas a `my_number > 0` e, em seguida, a expressão OR seria avaliada. Os operadores `&&` e `||` são conhecidos como operadores lógicos *binários*, porque requerem dois operandos. `!` é conhecido como um operador lógico *unário*, porque requer apenas um operando.

Estruturas Switch

Embora a estrutura `if` seja bastante versátil e suficiente para controlar o fluxo do programa, a estrutura de controle `switch` pode ser mais adequada quando se trata de avaliar resultados diferentes de verdadeiro ou falso. Por exemplo, se quiséssemos realizar uma ação distinta para cada item escolhido em uma lista, seria necessário escrever uma estrutura `if` para cada avaliação:

```
// Available languages: en (English), es (Spanish), pt (Portuguese)
let language = "pt";

// Variable to register whether the language was found in the list
let found = 0;

if ( language == "en" )
{
  found = 1;
  console.log("English");
}

if ( found == 0 && language == "es" )
{
  found = 1;
  console.log("Spanish");
}

if ( found == 0 && language == "pt" )
{
  found = 1;
  console.log("Portuguese");
}

if ( found == 0 )
{
  console.log(language, " is unknown to me");
}
```

Neste exemplo, uma variável auxiliar `found` é usada por todas as estruturas `if` para descobrir se uma correspondência foi encontrada. Em um caso como este, a estrutura `switch` realizaria a mesma tarefa, mas de forma mais sucinta:

```
switch ( language )
{
  case "en":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

Cada `case` aninhado é chamado de *cláusula*. Quando uma cláusula corresponde à expressão avaliada, ela executa as instruções que estão após os dois pontos até a instrução `break`. A última cláusula não precisa de uma instrução `break` e é freqüentemente usada para definir a ação padrão quando nenhuma outra correspondência é encontrada. Como visto no exemplo, a variável auxiliar não é necessária na estrutura `switch`.

WARNING

O `switch` usa a comparação estrita para comparar as expressões com as cláusulas de `case`.

Se mais de uma cláusula disparar a mesma ação, é possível combinar duas ou mais condições de `case`:

```
switch ( language )
{
  case "en":
  case "en_US":
  case "en_GB":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
  case "pt_BR":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

Laços

Nos exemplos anteriores, as estruturas `if` e `switch` eram adequadas para tarefas que precisam ser executadas apenas uma vez após passar por um ou mais testes condicionais. No entanto, existem situações em que uma tarefa deve ser executada repetidamente—no que se chama um *laço*, ou loop—enquanto a expressão condicional for avaliada como verdadeira. Se você precisa saber se um número é primo, por exemplo, será preciso verificar se a divisão desse número por qualquer inteiro maior que 1 e menor do que ele mesmo tem um resto igual a 0. Se for o caso, o número tem um fator inteiro e não é primo (este não é um método rigoroso ou eficiente para encontrar números primos, mas funciona como um exemplo simples). As estruturas de controle de laço são mais adequadas para esses casos, em particular a instrução `while`:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// The first factor to try
let factor = 2;

// Execute the block statement if factor is
// less than candidate and keep doing it
// while factor is less than candidate
while ( factor < candidate )
{

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next factor to try. Simply
  // increment the current factor by one
  factor++;
}

// Display the result in the console window.
// If candidate has no integer factor, then
// the auxiliary variable is_prime still true
if ( is_prime )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

A declaração de bloco após a instrução `while` será executada repetidamente enquanto a condição `factor < candidate` for verdadeira. Ela será executada pelo menos uma vez, desde que inicializemos

a variável `factor` com um valor inferior a `candidate`. A estrutura `if` aninhada na estrutura `while` avalia se o resto de `candidate` dividido por `factor` é zero. Se for o caso, o número candidato não é primo e o loop pode terminar. A declaração `break` encerra o laço e a execução pula para a primeira instrução após o bloco `while`.

Note que o resultado da condição usada pela declaração `while` deve mudar a cada loop, caso contrário a declaração de bloco vai rodar "para sempre". No exemplo, incrementamos a variável `factor` — o próximo divisor que queremos testar — garantindo que o laço terminará em algum momento.

Esta implementação simples de um código para testar números primos funciona conforme o esperado. No entanto, sabemos que um número que não é divisível por dois não será divisível por nenhum outro número par. Portanto, poderíamos simplesmente pular os números pares adicionando outra instrução `if`:

```
while ( factor < candidate )
{

  // Skip even factors bigger than two
  if ( factor > 2 && factor % 2 == 0 )
  {
    factor++;
    continue;
  }

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next number that will divide the candidate
  factor++;
}
```

A declaração `continue` é semelhante à declaração `break`, mas ao invés de encerrar esta iteração do laço, ela ignora o resto do bloco do laço e inicia uma nova iteração. Observe que a variável `factor` foi modificada antes da declaração `continue`; caso contrário, o loop teria o mesmo resultado novamente na iteração seguinte. Este exemplo é demasiado simples e pular parte do loop não vai melhorar muito o desempenho, mas a possibilidade de pular instruções redundantes é importantíssima para se escrever aplicativos eficientes.

Os loops são tão comumente usados que eles existem em muitas variantes diferentes. O loop `for` é especialmente adequado para iterar valores sequenciais, pois ele permite definir as regras do laço em uma única linha:

```
for ( let factor = 2; factor < candidate; factor++ )
{
  // Skip even factors bigger than two
  if ( factor > 2 && factor % 2 == 0 )
  {
    continue;
  }

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }
}
```

Este exemplo produz exatamente o mesmo resultado que o exemplo anterior com `while`, mas a expressão entre parênteses inclui três partes, separadas por ponto e vírgula: a inicialização (`let factor = 2`), a condição de loop (`factor < candidate`) e a expressão final a ser avaliada no final de cada iteração do loop (`factor++`). As instruções `continue` e `break` também se aplicam aos loops `for`. A expressão final entre parênteses (`factor++`) será avaliada após a declaração `continue` e, portanto, não deve estar dentro da declaração de bloco, ou será incrementada duas vezes antes da próxima iteração.

O JavaScript tem tipos especiais de loops `for` para trabalhar com objetos do tipo array (matriz). Poderíamos, por exemplo, verificar uma matriz de variáveis candidatas em vez de apenas uma:

```
// A naive prime number tester

// The array of numbers we want to evaluate
let candidates = [111, 139, 293, 327];

// Evaluates every candidate in the array
for (candidate of candidates)
{
  // Auxiliary variable
  let is_prime = true;

  for ( let factor = 2; factor < candidate; factor++ )
  {
    // Skip even factors bigger than two
    if ( factor > 2 && factor % 2 == 0 )
    {
      continue;
    }

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }
  }

  // Display the result in the console window
  if ( is_prime )
  {
    console.log(candidate, "is prime");
  }
  else
  {
    console.log(candidate, "is not prime");
  }
}
```

A declaração `for (candidate of candidates)` atribui um elemento da matriz `candidates` à variável `candidate` e o usa na declaração de bloco, repetindo o processo para cada elemento da matriz. Não é necessário declarar `candidate` separadamente, porque o loop `for` o define. Finalmente, o mesmo código do exemplo anterior foi aninhado nesta nova declaração de bloco, mas desta vez testando cada candidato da matriz por sua vez.

Exercícios Guiados

1. Quais valores da variável `my_var` correspondem à condição `my_var > 0 && my_var < 9`?

2. Quais valores da variável `my_var` correspondem à condição `my_var > 0 || my_var < 9`?

3. How many times does the following `while` loop execute its block statement?

```
let i = 0;
while ( 1 )
{
  if ( i == 10 )
  {
    continue;
  }
  i++;
}
```

Exercícios Exploratórios

1. O que acontece se o operador de atribuição igual `=` for usado em vez do operador de comparação igual `==`?

2. Escreva um fragmento de código usando a estrutura de controle `if`, no qual uma comparação de igualdade comum retorne verdadeiro, mas uma comparação de igualdade estrita não.

3. Reescreva a seguinte declaração `for` usando o operador lógico unário NOT na condição de loop. O resultado da condição deve ser o mesmo.

```
for ( let factor = 2; factor < candidate; factor++ )
```

4. Com base nos exemplos desta lição, escreva uma estrutura de controle de loop que imprima todos os fatores inteiros de um determinado número.

Resumo

Esta lição aborda o uso de estruturas de controle no código JavaScript. Estruturas condicionais e de laço (loop) são elementos essenciais de qualquer paradigma de programação, e o desenvolvimento web em JavaScript não é exceção. A lição trata dos seguintes conceitos e procedimentos:

- A declaração `if` e os operadores de comparação.
- Como usar a estrutura `switch` com `case`, `default` e `break`.
- A diferença entre comparações comuns e estritas.
- Estruturas de controle de laço: `while` e `for`.

Respostas aos Exercícios Guiados

1. Quais valores da variável `my_var` correspondem à condição `my_var > 0 && my_var < 9`?

   Somente números maiores que 0 e menores que 9. O operador lógico && (AND) requer que ambas as comparações correspondam.

2. Quais valores da variável `my_var` correspondem à condição `my_var > 0 || my_var < 9`?

   O operador lógico `||` (OR) fará com que qualquer número corresponda à condição, já que qualquer número será maior que 0 ou menor que 9.

3. Quantas vezes o loop `while` a seguir executa sua declaração de bloco?

```
let i = 0;
while ( 1 )
{
  if ( i == 10 )
  {
    continue;
  }
  i++;
}
```

   A declaração de bloco será repetida para sempre, pois não foi fornecida nenhuma condição de interrupção.

Respostas aos Exercícios Exploratórios

1. O que acontece se o operador de atribuição igual `=` for usado em vez do operador de comparação igual `==`?

   O valor à direita do operador é atribuído à variável à esquerda e o resultado é passado para a comparação, o que pode não ser o comportamento desejado.

2. Escreva um fragmento de código usando a estrutura de controle `if`, no qual uma comparação de igualdade comum retorne verdadeiro, mas uma comparação de igualdade estrita não.

   ```
   let a = "1";
   let b = 1;

   if ( a == b )
   {
     console.log("An ordinary comparison will match.");
   }

   if ( a === b )
   {
     console.log("A strict comparison will not match.");
   }
   ```

3. Reescreva a seguinte declaração `for` usando o operador lógico unário NOT na condição de loop. O resultado da condição deve ser o mesmo.

   ```
   for ( let factor = 2; factor < candidate; factor++ )
   ```

   Answer:

   ```
   for ( let factor = 2; ! (factor >= candidate); factor++ )
   ```

4. Com base nos exemplos desta lição, escreva uma estrutura de controle de loop que imprima todos os fatores inteiros de um determinado número.

```
for ( let factor = 2; factor <= my_number; factor++ )
{
  if ( my_number % factor == 0 )
  {
    console.log(factor, " is an integer factor of ", my_number);
  }
}
```

034.3 Lição 2

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	034 Programação em JavaScript
Objetivo:	034.3 Estruturas de controle e funções do JavaScript
Lição:	2 de 2

Introdução

Além do conjunto padrão de funções integradas fornecido pela linguagem JavaScript, os desenvolvedores podem escrever suas próprias funções personalizadas, de forma a conduzir uma entrada de dados até uma saída adequada às necessidades do aplicativo. As funções personalizadas são, basicamente, um conjunto de declarações encapsuladas e empregadas como parte de uma expressão.

O uso de funções é uma boa maneira de evitar a duplicação do código, já que elas podem ser chamadas de locais diferentes ao longo de todo o programa. Além disso, o agrupamento de declarações em funções facilita a vinculação de ações personalizadas a eventos, o que é um aspecto essencial da programação em JavaScript.

Definição de função

Conforme um programa cresce, vai ficando mais difícil organizar o que ele faz sem usar funções. Cada função tem seu próprio escopo privado de variáveis, de forma que as variáveis definidas dentro de uma função estarão disponíveis apenas dentro dessa mesma função. Assim, elas não se

confundem com variáveis de outras funções. As variáveis globais continuam acessíveis de dentro das funções, mas a maneira preferível de enviar valores de entrada para uma função é por meio de *parâmetros de função*. Como exemplo, vamos incrementar o validador de número primo da lição anterior:

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// Start with the lowest prime number after 1
let factor = 2;

// Keeps evaluating while factor is less than the candidate
while ( factor < candidate )
{

  if ( candidate % factor == 0 )
  {
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
  }

  // The next number that will divide the candidate
  factor++;
}

// Display the result in the console window
if ( is_prime )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

Se em um ponto mais adiante no programa for preciso verificar se um número é primo, será necessário repetir o código que já foi escrito. Essa prática não é recomendada, já que quaisquer

correções ou melhorias no código original precisariam ser replicadas manualmente em todos os lugares nos quais o código foi copiado. Além disso, a repetição do código sobrecarrega o navegador e a rede, tornando mais lenta a exibição da página web. Em vez disso, mova as declarações apropriadas para uma função:

```
// A naive prime number tester function
function test_prime(candidate)
{
  // Auxiliary variable
  let is_prime = true;

  // Start with the lowest prime number after 1
  let factor = 2;

  // Keeps evaluating while factor is less than the candidate
  while ( factor < candidate )
  {

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }

    // The next number that will divide the candidate
    factor++;
  }

  // Send the answer back
  return is_prime;
}
```

A declaração da função começa com uma declaração `function`, seguida pelo nome da função e seus parâmetros. O nome da função deve seguir as mesmas regras que os nomes das variáveis. Os parâmetros da função, também conhecidos como *argumentos* da função, são separados por vírgulas e postos entre parênteses.

TIP

Não é obrigatório listar os argumentos na declaração da função. Os argumentos passados para uma função podem ser obtidos de um objeto `arguments`, semelhante a uma matriz, dentro daquela função. O indice de argumentos começa em 0, de forma que o primeiro argumento é `arguments[0]`, o segundo é `arguments[1]`, e assim por diante.

No exemplo, a função `test_prime` tem apenas um argumento: o argumento `candidate`, que é o candidato a número primo a ser testado. Os argumentos da função funcionam como variáveis, mas seus valores são atribuídos pela declaração que chama a função. Por exemplo, a declaração `test_prime(231)` chama a função `test_prime` e atribui o valor 231 ao argumento `candidate`, que então fica disponível dentro do corpo da função como uma variável comum.

Se a declaração de chamada usar variáveis simples para os parâmetros da função, seus valores serão copiados para os argumentos da função. Este procedimento — copiar os valores dos parâmetros usados na declaração de chamada para os parâmetros usados dentro da função — é chamado de *passagem de argumentos por valor*. Quaisquer modificações feitas no argumento pela função não afetam a variável original usada na declaração de chamada. No entanto, se a declaração inicial usar objetos complexos como argumentos (ou seja, objetos com propriedades e métodos vinculados) para os parâmetros da função, eles serão *passados como referência* e a função será capaz de modificar o objeto original usado na declaração de chamada.

Os argumentos que são passados por valor, assim como as variáveis declaradas dentro da função, não são visíveis fora dela. Ou seja, seu escopo é restrito ao corpo da função em que foram declarados. Ainda assim, as funções geralmente são empregadas para criar alguma saída visível fora da função. Para compartilhar um valor com sua função de chamada, uma função define uma declaração `return`.

Por exemplo, a função `test_prime` do exemplo anterior retorna o valor da variável `is_prime`. Portanto, a função pode substituir a variável em qualquer lugar em que ela teria sido usada no exemplo original:

```
// The number we want to evaluate
let candidate = 231;

// Display the result in the console window
if ( test_prime(candidate) )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

A declaração `return`, como seu nome indica, devolve o controle para a função de chamada. Portanto, onde quer que a declaração `return` seja colocada na função, nada que venha depois dela é executado. Uma função pode conter múltiplas instruções `return`. Esta prática pode ser útil se algumas delas estiverem dentro de blocos condicionais de declarações, de modo que a função possa ou não executar uma instrução `return` determinada a cada execução.

Nem sempre as funções retornam um valor, por isso a instrução `return` não é obrigatória. As declarações internas da função são executadas independentemente de sua presença, de modo que as funções também podem ser utilizadas, por exemplo, para alterar os valores das variáveis globais ou o conteúdo dos objetos passados por referência. Não obstante, se a função não tiver uma declaração `return`, seu valor de retorno por padrão será `undefined`: uma variável reservada que não tem um valor e não pode ser escrita.

Expressões das funções

Em JavaScript, as funções são apenas mais um tipo de *objeto*. Assim, elas podem ser empregadas no script como variáveis. Essa característica se torna explícita quando a função é declarada usando uma sintaxe alternativa, chamada *expressões de função*:

```
let test_prime = function(candidate)
{
  // Auxiliary variable
  let is_prime = true;

  // Start with the lowest prime number after 1
  let factor = 2;

  // Keeps evaluating while factor is less than the candidate
  while ( factor < candidate )
  {

    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime
      is_prime = false;
      break;
    }

    // The next number that will divide the candidate
    factor++;
  }

  // Send the answer back
  return is_prime;
}
```

A única diferença entre este exemplo e a declaração da função no exemplo anterior está na primeira linha: `let test_prime = function(candidate)` em vez de `function test_prime(candidate)`. Em uma expressão de função, o nome `test_prime` é usado para o objeto que contém a função e não para nomear a função em si. As funções definidas em expressões de função são chamadas da mesma maneira que as funções definidas usando a sintaxe de declaração. No entanto, enquanto as funções declaradas podem ser chamadas antes ou depois de sua declaração, as expressões de função só podem ser chamadas após sua inicialização. Como ocorre com as variáveis, chamar uma função definida em uma expressão antes de sua inicialização causará um erro de referência.

Recursão de função

Além de executar declarações e chamar funções integradas, as funções personalizadas também podem chamar outras funções personalizadas, incluindo elas mesmas. Nesse caso, falamos em *recursão de função* (ou recursividade de função). Dependendo do tipo de problema que se está tentando resolver, pode ser mais simples usar funções recursivas do que loops aninhados para

executar tarefas repetitivas.

Até agora, aprendemos como usar uma função para testar se um determinado número é primo. Agora, vamos supor que queiramos encontrar o próximo primo após um determinado número. Poderíamos empregar um loop `while` para incrementar o número do candidato e escrever um loop aninhado que buscaria fatores inteiros para aquele candidato:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
  // We are only interested in the positive primes,
  // so we will consider the number 2 as the next
  // prime after any number less than two.
  if ( from < 2 )
  {
    return 2;
  }

  // The number 2 is the only even positive prime,
  // so it will be easier to treat it separately.
  if ( from == 2 )
  {
    return 3;
  }

  // Decrement "from" if it is an even number
  if ( from % 2 == 0 )
  {
    from--;
  }

  // Start searching for primes greater then 3.

  // The prime candidate is the next odd number
  let candidate = from + 2;

  // "true" keeps the loop going until a prime is found
  while ( true )
  {
    // Auxiliary control variable
    let is_prime = true;

    // "candidate" is an odd number, so the loop will
```

```
    // try only the odd factors, starting with 3
    for ( let factor = 3; factor < candidate; factor = factor + 2 )
    {
      if ( candidate % factor == 0 )
      {
        // The remainder is zero, so the candidate is not prime.
        // Test the next candidate
        is_prime = false;
        break;
      }
    }
    // End loop and return candidate if it is prime
    if ( is_prime )
    {
      return candidate;
    }
    // If prime not found yet, try the next odd number
    candidate = candidate + 2;
  }
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```

Note que precisamos usar uma condição constante para o loop `while` (a expressão `true` dentro do parêntese) e a variável auxiliar `is_prime` para saber quando parar o loop. Embora essa solução esteja correta, usar loops aninhados não é tão elegante quanto usar a recursão para executar a mesma tarefa:

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
  // We are only interested in the positive primes,
  // so we will consider the number 2 as the next
  // prime after any number less than two.
  if ( from < 2 )
  {
    return 2;
  }

  // The number 2 is the only even positive prime,
  // so it will be easier to treat it separately.
```

```
  if ( from == 2 )
  {
    return 3;
  }

  // Decrement "from" if it is an even number
  if ( from % 2 == 0 )
  {
    from--;
  }

  // Start searching for primes greater then 3.

  // The prime candidate is the next odd number
  let candidate = from + 2;

  // "candidate" is an odd number, so the loop will
  // try only the odd factors, starting with 3
  for ( let factor = 3; factor < candidate; factor = factor + 2 )
  {
    if ( candidate % factor == 0 )
    {
      // The remainder is zero, so the candidate is not prime.
      // Call the next_prime function recursively, this time
      // using the failed candidate as the argument.
      return next_prime(candidate);
    }
  }

  // "candidate" is not divisible by any integer factor other
  // than 1 and itself, therefore it is a prime number.
  return candidate;
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```

Ambas as versões de `next_prime` retornam o próximo número primo após o número dado como seu único argumento (`from`). A versão recursiva, como a versão anterior, começa verificando os casos especiais (ou seja, números menores ou iguais a dois). Em seguida, incrementa o candidato e começa a procurar por quaisquer fatores inteiros com o loop `for` (observe que o loop `while` não está mais presente). Nesse ponto, o único número primo par já foi testado, então o candidato e seus possíveis fatores são incrementados de dois em dois (um número ímpar mais dois é o número ímpar seguinte).

Existem apenas duas maneiras de sair do loop `for` no exemplo. Se todos os fatores possíveis forem testados e nenhum deles tiver um resto igual a zero ao dividir o candidato, o loop `for` se completa e a função retorna o candidato como o número primo seguinte depois de `from`. Caso contrário, se `factor` for um fator inteiro de `candidate` (`candidate % factor == 0`), o valor retornado virá da função `next_prime` chamada recursivamente, desta vez com o `candidate` incrementado como parâmetro `from`. As chamadas por `next_prime` serão empilhadas umas sobre as outras, até que um candidato finalmente não retorne fatores inteiros. Nesse momento, a última instância de `next_prime` contendo o número primo o retornará para a instância `next_prime` anterior, e assim sucessivamente até chegar à primeira instância de `next_prime`. Embora cada invocação da função use os mesmos nomes como variáveis, as invocações são isoladas entre si, de modo que suas variáveis são mantidas separadas na memória.

Exercícios Guiados

1. Que tipo de sobrecarga os desenvolvedores podem mitigar usando funções?

2. Qual a diferença entre argumentos de função passados por valor e argumentos de função passados por referência?

3. Qual valor será usado como saída de uma função personalizada se não houver uma declaração de retorno?

Exercícios Exploratórios

1. Qual é a causa provável de um *Uncaught Reference Error* (Erro de referência não detectado) emitido ao chamar uma função declarada com a sintaxe *expression*?

2. Escreva uma função chamada `multiples_of` que recebe três argumentos: `factor`, `from` e `to`. Dentro da função, use a instrução `console.log()` para imprimir todos os múltiplos de `factor` que estejam entre `from` e `to`.

Resumo

Esta lição ensina como escrever funções personalizadas em código JavaScript. As funções personalizadas permitem que o desenvolvedor divida a aplicação em "pedaços" de código reutilizáveis, facilitando a criação e manutenção de programas mais extensos. A lição aborda os seguintes conceitos e procedimentos:

- Como definir uma função personalizada: declarações de função e expressões de função.
- Uso de parâmetros como entrada da função.
- Uso da declaração `return` para definir a saída da função.
- Recursão de função.

Respostas aos Exercícios Guiados

1. Que tipo de sobrecarga os desenvolvedores podem mitigar usando funções?

   As funções nos permitem reutilizar trechos de código, o que facilita a manutenção do programa. Um arquivo de script menor também economiza memória e tempo de download.

2. Qual a diferença entre argumentos de função passados por valor e argumentos de função passados por referência?

   Quando passado por valor, o argumento é copiado para a função e a função não é capaz de modificar a variável original na declaração de chamada. Quando passado por referência, a função é capaz de manipular a variável original usada na declaração de chamada.

3. Qual valor será usado como saída de uma função personalizada se não houver uma declaração de retorno?

   O valor retornado será definido como `undefined`.

Respostas aos Exercícios Exploratórios

1. Qual é a causa provável de um *Uncaught Reference Error* (Erro de referência não detectado) emitido ao chamar uma função declarada com a sintaxe *expression*?

   A função foi chamada antes de sua declaração no arquivo de script.

2. Escreva uma função chamada `multiples_of` que recebe três argumentos: `factor`, `from` e `to`. Dentro da função, use a instrução `console.log()` para imprimir todos os múltiplos de `factor` que estejam entre `from` e `to`.

```
function multiples_of(factor, from, to)
{
  for ( let number = from; number <= to; number++ )
  {
    if ( number % factor == 0 )
    {
      console.log(factor, "*", number / factor, "=", number);
    }
  }
}
```

034.4 Manipulação de conteúdo e estilo de websites com JavaScript

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 034.4

Peso

4

Áreas chave de conhecimento

- Compreender o conceito e a estrutura do DOM
- Alterar o conteúdo e as propriedades dos elementos HTML por meio do DOM
- Alterar o estilo CSS dos elementos HTML por meio do DOM
- Acionar funções em JavaScript a partir de elementos HTML

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()`, `document.querySelectorAll()`
- `innerHTML`, `setAttribute()`, `removeAttribute()` propriedades e métodos dos elementos DOM
- `classList`, `classList.add()`, `classList.remove()`, `classList.toggle()` propriedades e métodos dos elementos DOM
- Atributos dos elementos HTML `onClick`, `onMouseOver`, `onMouseOut`

034.4 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	034 Programação em JavaScript
Objetivo:	034.4 Manipulação de conteúdo e estilo de websites com JavaScript
Lição:	1 de 1

Introdução

HTML, CSS e JavaScript são três tecnologias distintas que andam de mãos dadas na web. Para criar páginas que sejam de fato dinâmicas e interativas, o programador de JavaScript deve saber combinar componentes de HTML e CSS em tempo de execução, uma tarefa muito facilitada pelo uso do *Document Object Model* (DOM).

Interagindo com o DOM

O DOM é uma estrutura de dados que funciona como uma interface de programação para o documento, na qual cada aspecto do documento é representado como um nó no DOM e cada alteração feita no DOM reverbera imediatamente no documento. Para mostrar como usar o DOM em JavaScript, salve o seguinte código HTML em um arquivo chamado `example.html`:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first">
<p>The dynamic content goes here</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section</p>
</div><!-- #content_second -->

</body>
</html>
```

O DOM estará disponível somente depois que o HTML for carregado. Assim, escreva o seguinte código JavaScript no final do corpo da página (antes da tag `</body>` final):

```
<script>
let body = document.getElementsByTagName("body")[0];
console.log(body.innerHTML);
</script>
```

O objeto `document` é o elemento DOM superior; todos os outros se ramificam a partir dele. O método `getElementsByTagName()` lista todos os elementos descendentes de `document` que possuem o nome de tag dado. Mesmo que a tag `body` seja usada apenas uma vez no documento, o método `getElementsByTagName()` sempre retorna uma coleção semelhante a uma matriz com os elementos encontrados, por isso usamos o índice `[0]` para retornar o primeiro (e único) elemento encontrado.

Conteúdo em HTML

Como mostrado no exemplo anterior, o elemento DOM retornado por `document.getElementsByTagName("body")[0]` foi atribuído à variável `body`. A variável `body` pode então ser usada para manipular o elemento do corpo da página, porque ela herda todos os métodos e atributos DOM daquele elemento. Por exemplo, a propriedade `innerHTML` contém todo o código de marcação HTML escrito dentro do elemento correspondente, podendo assim ser usada para ler a marcação interna. Nossa chamada `console.log(body.innerHTML)` imprime o conteúdo dentro de

`<body></body>` no console web. A variável também pode ser usada para substituir esse conteúdo, como em `body.innerHTML = "<p>Content erased</p>`.

Em vez de alterar partes inteiras da marcação HTML, é mais prático manter a estrutura do documento inalterada e apenas interagir com seus elementos. Depois que o documento é renderizado pelo navegador, todos os elementos são acessíveis por métodos DOM. É possível, por exemplo, listar e acessar todos os elementos HTML usando a string especial `*` no método `getElementsByTagName()` do objeto `document`:

```
let elements = document.getElementsByTagName("*");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

Este código coloca todos os elementos encontrados em `document` na variável `elements`. A variável `elements` é um objeto semelhante a uma matriz (array), de modo que podemos iterar por cada um de seus itens com um loop `for`. Se a página HTML onde este código é executado tiver um elemento com um atributo `id` definido como `content_first` (veja a página HTML de exemplo no início da lição), a instrução `if` corresponderá a esse elemento e seu conteúdo de marcação será alterado para `<p>New content</p>`. Observe que os atributos de um elemento HTML no DOM são acessíveis usando a *notação de pontos* das propriedades do objeto JavaScript: portanto, `element.id` se refere ao atributo `id` do elemento atual do loop `for`. O método `getAttribute()` também pode ser usado, como em `element.getAttribute("id")`.

Não é necessário iterar por todos os elementos se a ideia for inspecionar apenas um subconjunto deles. Por exemplo, o método `document.getElementsByClassName()` limita os elementos correspondentes aos que possuem uma classe específica:

```
let elements = document.getElementsByClassName("content");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

No entanto, iterar por meio de muitos elementos do documento usando um loop não é a melhor estratégia quando é preciso alterar um elemento específico na página.

Seleção de elementos específicos

O JavaScript oferece métodos otimizados para selecionar o elemento exato no qual você deseja trabalhar. O loop anterior pode ser totalmente substituído pelo método `document.getElementById()`:

```
let element = document.getElementById("content_first");
element.innerHTML = "<p>New content</p>";
```

Cada atributo `id` no documento deve ser único, então o método `document.getElementById()` retorna apenas um único objeto DOM. Até mesmo a declaração da variável `element` pode ser omitida, porque o JavaScript nos permite encadear métodos diretamente:

```
document.getElementById("content_first").innerHTML = "<p>New content</p>";
```

O método `getElementById()` é preferível para localizar elementos no DOM, porque seu desempenho é muito melhor do que métodos iterativos ao se trabalhar com documentos complexos. No entanto, nem todos os elementos têm um ID explícito, e o método retorna um valor *null* se nenhum elemento corresponder ao ID fornecido (isso também impede o uso de atributos ou funções encadeadas, como o `innerHTML` usado no exemplo acima). Além disso, é mais prático conceder atributos de ID apenas aos componentes principais da página e, em seguida, usar seletores CSS para localizar seus elementos filhos. Os seletores, apresentados em uma lição anterior sobre CSS, são padrões que correspondem a elementos no DOM. O método `querySelector()` retorna o primeiro elemento correspondente na árvore do DOM, ao passo que `querySelectorAll()` retorna todos os elementos que correspondem ao seletor especificado.

No exemplo anterior, o método `getElementById()` recupera o elemento que tem o ID `content_first`. O método `querySelector()` pode realizar a mesma tarefa:

```
document.querySelector("#content_first").innerHTML = "<p>New content</p>";
```

Como o método `querySelector()` usa a sintaxe do seletor, o ID fornecido deve iniciar com um caractere de hash. Se nenhum elemento correspondente for encontrado, o método `querySelector()` retornas *null*.

No exemplo anterior, todo o conteúdo do div `content_first` é substituído pela string de texto fornecida. A string contém código HTML, o que não é considerado uma prática recomendada. É

preciso ter cuidado ao adicionar marcação HTML embutida (hard-coded) ao código JavaScript, porque fica mais difícil rastrear elementos quando é necessário fazer alterações na estrutura geral do documento.

Os seletores não se restringem ao ID do elemento. O elemento interno `p` pode ser endereçado diretamente:

```
document.querySelector("#content_first p").innerHTML = "New content";
```

O seletor `#content_first p` encontrará apenas o primeiro elemento `p` dentro do div `#content_first`. Isso funciona bem quando queremos manipular o primeiro elemento. Mas pode ser que seja preciso alterar o segundo parágrafo:

```
<div class="content" id="content_first">
<p>Don't change this paragraph.</p>
<p>The dynamic content goes here.</p>
</div><!-- #content_first -->
```

Nesse caso, usamos a pseudoclasse `:nth-child(2)` para encontrar o segundo elemento `p`:

```
document.querySelector("#content_first p:nth-child(2)").innerHTML = "New content";
```

O número `2` em `p:nth-child(2)` indica o segundo parágrafo que corresponde ao seletor. Veja a lição sobre seletores CSS para saber mais sobre seletores e como usá-los.

Trabalhando com atributos

A capacidade do JavaScript de interagir com o DOM não se restringe à manipulação de conteúdo. Na verdade, o uso mais difundido do JavaScript no navegador é modificar os atributos dos elementos HTML existentes.

Digamos que nossa página HTML de exemplo original tenha agora três seções de conteúdo:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first" hidden>
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third" hidden>
<p>Third section.</p>
</div><!-- #content_third -->

</body>
</html>
```

Podemos querer tornar apenas um deles visível por vez, daí o atributo `hidden` em todas as tags `div`. Isso é útil, por exemplo, para exibir apenas uma imagem de uma galeria de imagens. Para tornar um deles visível quando a página for carregada, adicionamos o seguinte código JavaScript à página:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}

document.querySelector(content_visible).removeAttribute("hidden");
```

A expressão avaliada pela instrução `switch` retorna aleatoriamente o número 0, 1 ou 2. O seletor de ID correspondente é então atribuído à variável `content_visible`, que é usada pelo método `querySelector(content_visible)`. A chamada `removeAttribute("hidden")` encadeada remove o atributo `hidden` do elemento.

Também é possível fazer o contrário: todas as seções podem estar inicialmente visíveis (sem o atributo `hidden`) e o programa em JavaScript conferir o atributo `hidden` a cada seção, exceto a que está em `content_visible`. Para fazer isso, você deve iterar por todos os elementos div de conteúdo que forem diferentes do escolhido, o que pode ser feito usando o método `querySelectorAll()`:

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}

// Hide all content divs, except content_visible
for ( element of document.querySelectorAll(".content:not("+content_visible+")") )
{
  // Hidden is a boolean attribute, so any value will enable it
  element.setAttribute("hidden", "");
}
```

Se a variável `content_visible` for definida como `#content_first`, o seletor será `.content:not(#content_first)`, o que indica todos os elementos da classe `content`, exceto os que têm o ID `content_first`. O método `setAttribute()` adiciona ou altera os atributos dos elementos HTML. Seu primeiro parâmetro é o nome do atributo e o segundo é o valor do atributo.

Todavia, a maneira correta de alterar a aparência dos elementos é com CSS. Nesse caso, podemos definir a propriedade CSS `display` como `hidden` e, em seguida, alterá-la para `block` usando JavaScript:

```
<style>
div.content { display: none }
</style>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->

<script>
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
  break;
  case 1:
    content_visible = "#content_second";
  break;
  case 2:
    content_visible = "#content_third";
  break;
}
document.querySelector(content_visible).style.display = "block";
</script>
```

As mesmas boas práticas que se aplicam à combinação de tags HTML com JavaScript também se aplicam ao CSS. Portanto, não é recomendado escrever propriedades CSS diretamente no código JavaScript. Em vez disso, as regras CSS devem ser escritas separadamente do código JavaScript. A maneira adequada de alternar estilos visuais é selecionar uma classe CSS predefinida para o elemento.

Trabalhando com classes

Os elementos podem ter mais de uma classe associada, sendo mais fácil escrever estilos que podem ser adicionados ou removidos quando necessário. Seria cansativo alterar muitos atributos CSS diretamente em JavaScript, então é melhor criar uma nova classe CSS com esses atributos e, em seguida, adicionar a classe ao elemento. Os elementos DOM têm a propriedade `classList`, que pode ser usada para visualizar e manipular as classes atribuídas ao elemento correspondente.

Por exemplo, em vez de alterar a visibilidade do elemento, podemos criar uma classe CSS adicional para destacar nosso div `content`:

```
div.content {
  border: 1px solid black;
  opacity: 0.25;
}
div.content.highlight {
  border: 1px solid red;
  opacity: 1;
}
```

Esta folha de estilo adiciona uma borda preta fina e uma semitransparência a todos os elementos da classe `content`. Apenas os elementos que também pertencem à classe `highlight` serão totalmente opacos e terão a borda vermelha fina. Então, em vez de alterar as propriedades CSS diretamente como fizemos antes, podemos usar o método `classList.add("highlight")` no elemento selecionado:

```
// Which content to highlight
let content_highlight;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_highlight = "#content_first";
  break;
  case 1:
    content_highlight = "#content_second";
  break;
  case 2:
    content_highlight = "#content_third";
  break;
}

// Highlight the selected div
document.querySelector(content_highlight).classList.add("highlight");
```

Todas as técnicas e exemplos que vimos até agora foram realizados no final do processo de carregamento da página, mas eles não se limitam a essa etapa. Na verdade, o que torna o JavaScript tão útil para os desenvolvedores web é sua capacidade de reagir aos eventos da página, como veremos a seguir.

Manipuladores de eventos

Todos os elementos visíveis da página são suscetíveis a eventos interativos, como os cliques ou o próprio movimento do mouse. Podemos associar ações personalizadas a esses eventos, o que expande muito as capacidades de um documento HTML.

O elemento HTML que mais obviamente se beneficia de uma ação associada é o elemento `button`. Para entender como funciona, adicione três botões acima do primeiro elemento `div` da página de exemplo:

```
<p>
<button>First</button>
<button>Second</button>
<button>Third</button>
</p>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->
```

Os botões não fazem nada por conta própria, mas suponha que você queira destacar o `div` correspondente ao botão pressionado. Podemos usar o atributo `onClick` para associar uma ação a cada botão:

```
<p>
<button
onClick="document.getElementById('content_first').classList.toggle('highlight')">Fi
rst</button>
<button
onClick="document.getElementById('content_second').classList.toggle('highlight')">S
econd</button>
<button
onClick="document.getElementById('content_third').classList.toggle('highlight')">Th
ird</button>
</p>
```

O método `classList.toggle()` adiciona a classe especificada ao elemento se ela não estiver presente e remove essa classe se ela já estiver presente. Se você executar o exemplo, notará que mais de um `div` pode ser destacado ao mesmo tempo. Para destacar apenas o `div` correspondente ao botão pressionado, é necessário remover a classe `highlight` dos outros elementos `div`. No entanto, se a ação personalizada for muito longa ou envolver mais de uma linha de código, é mais prático escrever uma função separada da tag do elemento:

```
function highlight(id)
{
  // Remove the "highlight" class from all content elements
  for ( element of document.querySelectorAll(".content") )
  {
    element.classList.remove('highlight');
  }

  // Add the "highlight" class to the corresponding element
  document.getElementById(id).classList.add('highlight');
}
```

Como nos exemplos anteriores, esta função pode ser posta dentro de uma tag `<script>` ou em um arquivo JavaScript externo associado ao documento. A função `highlight` remove primeiro a classe `highlight` de todos os elementos `div` associados à classe `content`, adicionando em seguida a classe `highlight` ao elemento escolhido. Cada botão deve então chamar esta função a partir de seu atributo `onClick`, usando o ID correspondente como argumento da função:

```
<p>
<button onClick="highlight('content_first')">First</button>
<button onClick="highlight('content_second')">Second</button>
<button onClick="highlight('content_third')">Third</button>
</p>
```

Além do atributo `onClick`, poderíamos usar o atributo `onMouseOver` (disparado quando o dispositivo apontador é usado para mover o cursor sobre o elemento), o atributo `onMouseOut` (disparado quando o dispositivo apontador não está mais sobre o elemento), etc. Além disso, os manipuladores de eventos não se restringem aos botões; portanto, podemos atribuir ações personalizadas a esses manipuladores de eventos para todos os elementos HTML visíveis.

Exercícios Guiados

1. Usando o método `document.getElementById()`, como seria possível inserir a frase "Dynamic content" no conteúdo interno do elemento cujo ID é `message`?

2. Qual é a diferença entre referenciar um elemento por seu ID usando o método `document.querySelector()` ou fazê-lo com o método `document.getElementById()`?

3. Qual é a finalidade do método `classList.remove()`?

4. Qual o resultado do uso do método `myelement.classList.toggle("active")` se a classe `active` não tiver sido atribuída a `myelement`?

Exercícios Exploratórios

1. Qual argumento atribuído ao método `document.querySelectorAll()` fará com que ele imite o método `document.getElementsByTagName("input")`?

2. Como usar a propriedade `classList` para listar todas as classes associadas a um elemento determinado?

Resumo

Esta lição ensina como usar o JavaScript para alterar o conteúdo em HTML e suas propriedades CSS usando o DOM (Document Object Model). Essas mudanças podem ser disparadas por eventos do usuário, o que é útil para criar interfaces dinâmicas. A lição aborda os seguintes conceitos e procedimentos:

- Como inspecionar a estrutura do documento usando métodos como `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()` e `document.querySelectorAll()`.
- Como alterar o conteúdo do documento com a propriedade `innerHTML`.
- Como adicionar e modificar os atributos dos elementos da página com os métodos `setAttribute()` e `removeAttribute()`.
- A maneira correta de manipular classs de elementos usando a propriedade `classList` e sua relação com os estilos CSS.
- Como vincular funções a eventos de mouse em elementos específicos.

Respostas aos Exercícios Guiados

1. Usando o método `document.getElementById()`, como seria possível inserir a frase "Dynamic content" no conteúdo interno do elemento cujo ID é `message`?

   It can be done with the `innerHTML` property:

   ```
   document.getElementById("message").innerHTML = "Dynamic content"
   ```

2. Qual é a diferença entre referenciar um elemento por seu ID usando o método `document.querySelector()` ou fazê-lo com o método `document.getElementById()`?

   O ID deve ser acompanhado pelo caractere de hash (cerquilha) nas funções que usam seletores, como `document.querySelector()`.

3. Qual é a finalidade do método `classList.remove()`?

   Ele remove a classe (cujo nome é dado como argumento da função) do atributo `class` do elemento correspondente.

4. Qual o resultado do uso do método `myelement.classList.toggle("active")` se a classe `active` não tiver sido atribuída a `myelement`?

   O método atribuirá a classe `active` a `myelement`.

Respostas aos Exercícios Exploratórios

1. Qual argumento atribuído ao método `document.querySelectorAll()` fará com que ele imite o método `document.getElementsByTagName("input")`?

   O uso de `document.querySelectorAll("input")` encontrará todos os elementos `input` na página, bem como `document.getElementsByTagName("input")`.

2. Como usar a propriedade `classList` para listar todas as classes associadas a um elemento determinado?

   A propriedade `classList` é um objeto semelhante a um array (matriz), de modo que um loop `for` pode ser usado para iterar por todas as classes que ele contém.

Linux Professional Institute

Tópico 035: Programação do servidor Node.js

035.1 Noções básicas de Node.js

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 035.1

Peso

1

Áreas chave de conhecimento

- Entender os conceitos de Node.js
- Executar um aplicativo NodeJS
- Instalar pacotes NPM

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `node [file.js]`
- `npm init`
- `npm install [module_name]`
- `package.json`
- `node_modules`

035.1 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	035 Programação do servidor Node.js
Objetivo:	035.i Noções básicas de Node.js
Lição:	1 de 1

Introdução

O Node.js é um ambiente de tempo de execução que executa código JavaScript em servidores web — o chamado *back-end* (lado do servidor) — em vez de usar uma segunda linguagem, como Python ou Ruby, para os programas do lado do servidor. A linguagem JavaScript já é utilizada no front-end moderno dos aplicativos web, interagindo com o HTML e CSS da interface de usuário do navegador web. O uso do Node.js em conjunto com o JavaScript no navegador oferece a possibilidade de se empregar apenas uma linguagem de programação para todo o aplicativo.

A principal razão para a existência do Node.js é a maneira como ele lida com diversas conexões simultâneas no back-end. Uma das maneiras mais comuns de um servidor de aplicativos web manipular conexões é por meio da execução de vários processos. Quando abrimos um aplicativo instalado no computador, inicia-se um processo que consome muitos recursos. Imagine o que acontece quando milhares de usuários estão fazendo a mesma coisa em um grande aplicativo web.

O Node.js evita esse problema usando um design chamado *laço de eventos* (event loop), que é um loop interno que verifica continuamente as tarefas de entrada a serem computadas. Graças ao uso generalizado do JavaScript e à onipresença das tecnologias web, o Node.js foi largamente adotado em aplicativos pequenos e grandes. Existem outras características que também ajudaram no crescimento

do Node.js, como o processamento de entrada/saída (I/O) assíncrono e sem bloqueio, explicado posteriormente nesta lição.

O ambiente do Node.js usa um mecanismo em JavaScript para interpretar e executar o código JavaScript no lado do servidor ou na área de trabalho. Nessas condições, o código JavaScript que o programador escreve é analisado e compilado dinamicamente (just-in-time) para executar as instruções de máquina geradas pelo código JavaScript original.

NOTE Conforme você progride nestas lições sobre o Node.js, você vai notar que o JavaScript do Node.js não é exatamente o mesmo que roda no navegador (que segue a especificação ECMAScript), mas é bastante semelhante.

Para começar

Esta seção e os exemplos a seguir presumem que o Node.js já está instalado em seu sistema operacional Linux e que você já domina habilidades básicas, como executar comandos no terminal.

Para executar os exemplos a seguir, crie um diretório de trabalho chamado `node_examples`. Abra um prompt de terminal e digite `node`. Se o Node.js estiver instalado corretamente, ele apresentará um prompt `>` onde será possível testar os comandos JavaScript interativamente. Este tipo de ambiente é chamado REPL, que em inglês é a sigla para "ler, avaliar, imprimir e fazer loop". Digite a seguinte entrada (ou alguma outra instrução JavaScript) nos prompts `>`. Pressione a tecla Enter após cada linha e o ambiente REPL retornará os resultados de suas ações:

```
> let array = ['a', 'b', 'c', 'd'];
undefined
> array.map( (element, index) => (`Element: ${element} at index: ${index}`));
[
  'Element: a at index: 0',
  'Element: b at index: 1',
  'Element: c at index: 2',
  'Element: d at index: 3'
]
>
```

O trecho de código foi escrito usando a sintaxe ES6, que oferece uma função de mapa para iterar sobre a matriz e imprimir os resultados usando modelos de string. É possível escrever praticamente qualquer comando que seja válido. Para sair do terminal Node.js, digite `.exit`, sem esquecer o ponto inicial.

Para scripts e módulos mais longos, é mais prático usar um editor de texto como o VS Code, Emacs

ou Vim. Você pode salvar as duas linhas de código que acabamos de mostrar (com uma pequena modificação) em um arquivo chamado `start.js`:

```
let array = ['a', 'b', 'c', 'd'];
array.map( (element, index) => ( console.log(`Element: ${element} at index: ${
index}`)));
```

Em seguida, execute o script do shell de maneira a produzir os mesmos resultados de antes:

```
$ node ./start.js
Element: a at index: 0
Element: b at index: 1
Element: c at index: 2
Element: d at index: 3
```

Antes de mergulhar em mais código, vamos dar uma visão geral do funcionamento do Node.js, usando seu ambiente de execução de thread único e o laço de eventos.

Laço de eventos e thread único

É difícil dizer quanto tempo seu programa Node.js levará para lidar com uma solicitação. Algumas solicitações podem ser curtas—por exemplo, percorrer variáveis na memória e retorná-las—ao passo que outras podem exigir atividades mais demoradas, como abrir um arquivo no sistema ou emitir uma consulta a um banco de dados e aguardar os resultados. Como o Node.js lida com essa incerteza? O laço de eventos é a resposta.

Imagine um chef de cozinha realizando várias tarefas. Assar um bolo é algo requer um certo tempo do forno. O chef não fica sentado esperando o bolo ficar pronto e só depois começa a coar um café. Em vez disso, enquanto o forno assa o bolo, o chef prepara o café e faz outras coisas em paralelo. Mas ele está sempre verificando se está na hora de mudar o foco para uma tarefa específica (como fazer café) ou de tirar o bolo do forno.

O laço de eventos é como o chef que está constantemente ciente do que se passa ao redor. No Node.js, um "verificador de eventos" está sempre verificando as operações que foram concluídas ou que estão esperando para serem processadas pelo mecanismo JavaScript.

Graças a esse método, uma operação assíncrona e longa não bloqueia as outras operações mais rápidas que vêm depois. Isso ocorre porque o mecanismo de laço de evento está sempre verificando se aquela tarefa longa, como uma operação de I/O, já foi realizada. Caso contrário, o Node.js pode continuar a processar outras tarefas. Quando a tarefa em segundo plano é concluída, os resultados

são retornados e o aplicativo que está rodando sobre o Node.js pode usar uma função de gatilho (retorno de chamada) para processar a saída.

Como o Node.js evita o uso de múltiplos encadeamentos (threads), como fazem outros ambientes, ele é chamado de *ambiente de thread único* e, portanto, é importantíssimo que a execução ocorra sem bloqueios. É por isso que o Node.js usa um laço de eventos. Para as tarefas de computação intensiva, o Node.js não está entre as melhores ferramentas: existem outras linguagens de programação e ambientes que fazem isso de forma mais eficiente.

Nas seções a seguir, examinaremos mais de perto as funções de retorno de chamada. Por enquanto, entenda que as funções de retorno de chamada são gatilhos executados após a conclusão de uma operação predefinida.

Módulos

É sempre recomendável dividir as funcionalidades complexas e os trechos extensos de código em partes menores. Essa modularização ajuda a organizar melhor a base de código, abstrair as implementações e evitar problemas de engenharia complicados. Para atender a essas necessidades, os programadores empacotam blocos de código-fonte para serem consumidos por outras partes internas ou externas do código.

Considere o exemplo de um programa que calcula o volume de uma esfera. Abra seu editor de texto e crie um arquivo chamado `volumeCalculator.js` contendo o seguinte código JavaScript:

```
const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * radius
}

console.log(`A sphere with radius 3 has a ${sphereVol(3)} volume.`);
console.log(`A sphere with radius 6 has a ${sphereVol(6)} volume.`);
```

A seguir, execute o arquivo usando o Node:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A sphere with radius 6 has a 904.7786842338603 volume.
```

Aqui, uma função simples foi usada para calcular o volume de uma esfera com base em seu raio. Imagine que também precisamos calcular o volume de um cilindro, um cone e assim por diante: percebemos rapidamente que essas funções específicas devem ser adicionadas ao arquivo

`volumeCalculator.js`, que pode acabar se tornando uma enorme coleção de funções. Para organizar melhor a estrutura, podemos usar a ideia por trás dos módulos como pacotes de código separado.

Para isso, crie um arquivo separado chamado `polyhedrons.js`:

```
const coneVol = (radius, height) => {
  return 1 / 3 * Math.PI * Math.pow(radius, 2) * height;
}

const cylinderVol = (radius, height) => {
  return Math.PI * Math.pow(radius, 2) * height;
}

const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * Math.pow(radius, 3);
}

module.exports = {
  coneVol,
  cylinderVol,
  sphereVol
}
```

A seguir, no arquivo `volumeCalculator.js`, exclua o código antigo e substitua-o por este trecho:

```
const polyhedrons = require('./polyhedrons.js');

console.log(`A sphere with radius 3 has a ${polyhedrons.sphereVol(3)} volume.`);
console.log(`A cylinder with radius 3 and height 5 has a ${polyhedrons.cylinderVol
(3, 5)} volume.`);
console.log(`A cone with radius 3 and height 5 has a ${polyhedrons.coneVol(3, 5)}
volume.`);
```

Depois execute o nome do arquivo no ambiente Node.js:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A cylinder with radius 3 and height 5 has a 141.3716694115407 volume.
A cone with radius 3 and height 5 has a 47.12388980384689 volume.
```

No ambiente Node.js, todo arquivo de código-fonte é considerado um módulo, mas a palavra

"module" em Node.js indica um pacote de código embrulhado como no exemplo anterior. Graças aos módulos, abstraímos as funções de volume do arquivo principal, `volumeCalculator.js`, reduzindo assim seu tamanho e facilitando a aplicação de testes de unidade, o que é uma boa prática ao desenvolver aplicativos no mundo real.

Agora que sabemos como os módulos são usados no Node.js, podemos usar uma das ferramentas mais importantes: o *Node Package Manager* (NPM).

Uma das principais tarefas do NPM é gerenciar, baixar e instalar módulos externos no projeto ou no sistema operacional. O comando `npm init` permite inicializar um repositório de nó.

O NPM fará as perguntas padrão sobre o nome do seu repositório, versão, descrição e assim por diante. Você pode ignorar essas etapas usando `npm init --yes`, e o comando irá gerar automaticamente um arquivo `package.json` descrevendo as propriedades do seu projeto/módulo.

Abra o arquivo `package.json` em seu editor de texto favorito e você verá um arquivo JSON contendo propriedades como palavras-chave, comandos de script para usar com NPM, um nome etc.

Uma dessas propriedades são as dependências instaladas em seu repositório local. O NPM adicionará o nome e a versão dessas dependências em `package.json`, junto com `package-lock.json`, outro arquivo usado como reserva pelo NPM no caso de `package.json` falhar.

Digite o seguinte em seu terminal:

```
$ npm i dayjs
```

O sinalizador `i` é um atalho para o argumento `install`. Se você estiver conectado à internet, o NPM vai procurar um módulo chamado `dayjs` no repositório remoto do Node.js, baixar o módulo e instalá-lo localmente. O NPM também adicionará essa dependência aos arquivos `package.json` e `package-lock.json`. Você perceberá a presença de uma pasta chamada `node_modules`, que contém o módulo instalado junto com outros módulos, se eles forem necessários. O diretório `node_modules` contém o código real que será usado quando a biblioteca for importada e chamada. Porém, esta pasta não é salva nos sistemas de versionamento que usam Git, já que o arquivo `package.json` fornece todas as dependências usadas. Outro usuário pode pegar o arquivo `package.json` e simplesmente executar `npm install` em sua própria máquina, onde o NPM criará uma pasta `node_modules` com todas as dependências de `package.json`, evitando assim o controle de versão para os milhares de arquivos disponíveis no repositório NPM.

Agora que o módulo `dayjs` está instalado no diretório local, abra o console Node.js e digite as seguintes linhas:

```
const dayjs = require('dayjs');
dayjs().format('YYYY MM-DDTHH:mm:ss')
```

O módulo `dayjs` é carregado com a palavra-chave `require`. Quando um método do módulo é chamado, a biblioteca pega a data e hora atuais do sistema e as exibe no formato especificado:

```
2020 11-22T11:04:36
```

Este é o mesmo mecanismo usado no exemplo anterior, em que o tempo de execução do Node.js carrega a função de terceiros no código.

Funcionalidade do servidor

Como o Node.js controla o back-end de aplicativos web, uma de suas principais tarefas é lidar com solicitações HTTP.

Eis um resumo de como os servidores web lidam com as solicitações HTTP de entrada. A funcionalidade do servidor é escutar solicitações, determinar o mais rápido possível qual a resposta de que cada um precisa e retornar essa resposta ao remetente da solicitação. Esse aplicativo deve receber uma solicitação HTTP de entrada acionada pelo usuário, analisar a solicitação, realizar o cálculo, gerar a resposta e enviá-la de volta. Um módulo HTTP, como o Node.js, é usado porque simplifica essas etapas, permitindo que o programador se concentre no próprio aplicativo.

Considere o seguinte exemplo, que implementa essa funcionalidade básica:

```
const http = require('http');
const url = require('url');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  const queryObject = url.parse(req.url,true).query;
  let result = parseInt(queryObject.a) + parseInt(queryObject.b);

  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Result: ${result}\n`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Salve esse conteúdo em um arquivo chamado `basic_server.js` e execute-o através de um comando `node`. O terminal que executa o Node.js exibirá a seguinte mensagem:

```
Server running at http://127.0.0.1:3000/
```

Em seguida, visite a URL a seguir em seu navegador web: `http://127.0.0.1:3000/numbers?a=2&b=17`

O Node.js está rodando um servidor web em seu computador e usando dois módulos: `http` e `url`. O módulo `http` configura um servidor HTTP básico, processa as solicitações da web de entrada e as entrega ao nosso aplicativo simples. O módulo URL analisa os argumentos passados na URL, converte-os em um formato inteiro e executa a operação de adição. O módulo `http` então envia a resposta em forma de texto para o navegador web.

Em um aplicativo web real, o Node.js é comumente usado para processar e obter dados, geralmente de um banco de dados, e retornar as informações processadas ao front-end para exibição. Mas o aplicativo básico nesta lição mostra de forma concisa como o Node.js usa módulos para lidar com solicitações da web como um servidor web.

Exercícios Guiados

1. Quais são as razões para se usar módulos em vez de escrever funções simples?

2. Por que o ambiente Node.js se tornou tão popular? Cite uma característica.

3. Qual é a finalidade do arquivo `package.json`?

4. Por que não é recomendado salvar e compartilhar a pasta `node_modules`?

Exercícios Exploratórios

1. Como você pode executar aplicativos Node.js em seu computador?

2. Como delimitar parâmetros na URL ao se fazer análises dentro do servidor?

3. Descreva um cenário em que uma tarefa específica pode ser um gargalo para um aplicativo Node.js.

4. Como seria possível implementar um parâmetro para multiplicar ou somar os dois números no exemplo do servidor?

Resumo

Esta lição trouxe uma visão geral do ambiente do Node.js, suas características e como ele pode ser usado para implementar programas simples. A lição inclui os seguintes conceitos:

- O que é o Node.js e por que é usado.
- Como executar programas do Node.js usando a linha de comando.
- Os laços de eventos e o thread único.
- Módulos.
- Node Package Manager (NPM).
- Funcionalidade de servidor.

Respostas aos Exercícios Guiados

1. Quais são as razões para se usar módulos em vez de escrever funções simples?

   Ao optar por módulos em vez de funções convencionais, o programador cria uma base de código mais simples de ler e manter, bem como para escrever testes automatizados.

2. Por que o ambiente Node.js se tornou tão popular? Cite uma característica.

   Um dos motivos é a flexibilidade da linguagem JavaScript, que já era amplamente utilizada no front-end dos aplicativos web. O Node.js permite o uso de apenas uma linguagem de programação em todo o sistema.

3. Qual é a finalidade do arquivo `package.json`?

   Este arquivo contém metadados para o projeto, como nome, versão, dependências (bibliotecas) e assim por diante. Com um arquivo `package.json`, outras pessoas podem baixar e instalar as mesmas bibliotecas e executar testes da mesma forma que o criador original.

4. Por que não é recomendado salvar e compartilhar a pasta `node_modules`?

   A pasta `node_modules` contém as implementações de bibliotecas disponíveis em repositórios remotos. Portanto, a melhor maneira de compartilhar essas bibliotecas é indicá-las no arquivo `package.json` e então usar o NPM para baixá-las. Esse método é mais simples e menos sujeito a erros, já que não é necessário rastrear e manter as bibliotecas localmente.

Respostas aos Exercícios Exploratórios

1. Como você pode executar aplicativos Node.js em seu computador?

   É possível fazer isso digitando `node PATH/FILE_NAME.js` na linha de comando do terminal, alterando `PATH` para o caminho do arquivo Node.js e trocando `FILE_NAME.js` pelo nome de arquivo escolhido.

2. Como delimitar parâmetros na URL ao se fazer análises dentro do servidor?

   O caractere "e comercial" (&) é usado para delimitar esses parâmetros, de forma que eles possam ser extraídos e analisados no código JavaScript.

3. Descreva um cenário em que uma tarefa específica pode ser um gargalo para um aplicativo Node.js.

   O Node.js não é um bom ambiente para executar processos intensivos de CPU, já que usa um único thread. Um cenário de computação numérica poderia desacelerar e bloquear todo o aplicativo. Se uma simulação numérica for necessária, é melhor usar outras ferramentas.

4. Como seria possível implementar um parâmetro para multiplicar ou somar os dois números no exemplo do servidor?

   Use um operador ternário ou uma condição if-else para verificar um parâmetro adicional. Se o parâmetro é a string `mult` ele retorna o produto dos números, do contrário retorna a soma. Substitua o código antigo pelo trecho abaixo. Reinicie o servidor na linha de comando pressionando `Ctrl` + `C` e execute novamente o comando para reiniciar o servidor. A seguir, teste o novo aplicativo visitando a URL `http://127.0.0.1:3000/numbers?a=2&b=17&operation=mult` em seu navegador. Se você omitir ou alterar o último parâmetro, os resultados devem ser a soma dos números.

```
let result = queryObject.operation == 'mult' ? parseInt(queryObject.a) *
parseInt(queryObject.b) : parseInt(queryObject.a) + parseInt(queryObject.b);
```

035.2 Noções básicas de NodeJS Express

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 035.2

Peso

4

Áreas chave de conhecimento

- Definir rotas para arquivos estáticos e modelos EJS
- Servir arquivos estáticos através do Express
- Servir modelos EJS através do Express
- Criar modelos EJS simples e não aninhados
- Usar o objeto da solicitação para acessar os parâmetros HTTP GET e POST e processar dados enviados por meio de formulários HTML
- Conhecimentos sobre a validação de entrada de dados do usuário
- Noções de cross-site scripting (XSS)
- Noções de cross-site request forgery (CSRF)

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- Módulo `express` e `body-parser`
- Objeto app do Express
- `app.get()`, `app.post()`
- `res.query`, `res.body`
- Módulo `node` do ejs

- `res.render()`
- `<% … %>`, `<%= … %>`, `<%# … %>`, `<%– … %>`
- `views/`

035.2 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	035 Programação do servidor NodeJS
Objetivo:	035.3 Noções básicas de NodeJS Express
Lição:	1 de 2

Introdução

O Express.js, ou simplesmente Express, é um framework popular que roda em Node.js e é usado para escrever servidores HTTP que lidam com solicitações de clientes de aplicativos web. O Express oferece suporte a diversas maneiras de ler parâmetros enviados por HTTP.

Script inicial do servidor

Para demonstrar os recursos básicos do Express ao receber e tratar solicitações, vamos simular um aplicativo que solicita algumas informações ao servidor. Em particular, o servidor de exemplo:

- Fornece uma função `echo`, que simplesmente retorna a mensagem enviada pelo cliente.
- Informa ao cliente seu endereço IP mediante solicitação.
- Usa cookies para identificar clientes conhecidos.

O primeiro passo é criar o arquivo JavaScript que funcionará como servidor. Usando `npm`, crie um diretório chamado `myserver` com o arquivo JavaScript:

```
$ mkdir myserver
$ cd myserver/
$ npm init
```

No ponto de entrada, qualquer nome de arquivo pode ser usado. Aqui, usaremos o nome de arquivo padrão: `index.js`. A lista a seguir mostra um arquivo `index.js` básico que será usado como ponto de entrada para o nosso servidor:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.get('/', (req, res) => {
  res.send('Request received')
})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Algumas constantes importantes para a configuração do servidor são definidas nas primeiras linhas do script. As duas primeiras, `express` e `app`, correspondem ao módulo `express` incluído e a uma instância desse módulo que executa nosso aplicativo. Adicionaremos as ações a serem realizadas pelo servidor ao objeto `app`.

As outras duas constantes, `host` e `port`, definem o host e a porta de comunicação associada ao servidor.

Caso você tenha um host acessível publicamente, use o nome dele ao invés de `myserver` como o valor de `host`. Se o nome do host não for informado, o Express usará por padrão `localhost`, o computador em que o aplicativo é executado. Nesse caso, nenhum cliente externo poderá acessar o programa, o que pode ser bom para testes, mas oferece pouco valor em um contexto de produção.

É imprescindível informar a porta, ou o servidor não poderá ser iniciado.

Este script anexa apenas dois procedimentos ao objeto `app`: a ação `app.get()`, que responde às solicitações feitas por clientes através de HTTP `GET`, e a chamada `app.listen()`, necessária para ativar o servidor e que atribui a ele um host e uma porta.

Para iniciar o servidor, basta executar o comando `node`, fornecendo o nome do script como

argumento:

```
$ node index.js
```

Assim que a mensagem `Server ready at http://myserver:8080` aparecer, o servidor estará pronto para receber as solicitações de um cliente HTTP. As solicitações podem ser feitas a partir de um navegador no mesmo computador em que o servidor está sendo executado ou a partir de outra máquina que possa acessar o servidor.

Todos os detalhes da transação que veremos aqui são mostrados no navegador, bastando abrir a janela do console do desenvolvedor. Alternativamente, o comando `curl` pode ser usado para comunicação HTTP, permitindo inspecionar mais facilmente os detalhes da conexão. Caso não esteja familiarizado com a linha de comando do shell, você pode criar um formulário HTML para enviar solicitações a um servidor.

O exemplo a seguir mostra como usar o comando `curl` na linha de comando para fazer uma solicitação HTTP ao servidor recém-implantado:

```
$ curl http://myserver:8080 -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 16
< ETag: W/"10-1WVvDtVyAF0vX9evlsFlfiJTT5c"
< Date: Fri, 02 Jul 2021 14:35:11 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Request received
```

A opção `-v` do comando `curl` exibe todos os cabeçalhos de solicitação e resposta, bem como outras informações de depuração. As linhas que começam com `>` indicam os cabeçalhos de solicitação

enviados pelo cliente e as linhas que começam com `<` indicam os cabeçalhos de resposta enviados pelo servidor. As linhas começando com `*` são informações geradas pelo próprio `curl`. O conteúdo da resposta é mostrado apenas no final, que neste caso é a linha `Request received`.

A URL do serviço, que aqui contém o nome do host do servidor e a porta (`http://myserver:8080`), foram dados como argumentos para o comando `curl`. Como nenhum diretório ou nome de arquivo é fornecido, o padrão é o diretório raiz `/`. A barra aparece como o arquivo de solicitação na linha `> GET / HTTP/1.1`, que é seguido, na saída, pelo nome do host e porta.

Além de exibir cabeçalhos de conexão HTTP, o comando `curl` auxilia no desenvolvimento de aplicativos, permitindo enviar dados para o servidor usando diferentes métodos HTTP e em diferentes formatos. Essa flexibilidade torna mais fácil depurar quaisquer problemas e implementar novos recursos no servidor.

Rotas

As solicitações que o cliente pode fazer ao servidor dependem de quais *rotas* foram definidas no arquivo `index.js`. Uma rota especifica um método HTTP e define um *caminho* (mais precisamente, um URI) que pode ser solicitado pelo cliente.

Até aqui, o servidor tem apenas uma rota configurada:

```
app.get('/', (req, res) => {
  res.send('Request received')
})
```

Embora seja uma rota muito simples, que retorna apenas uma mensagem de texto puro ao cliente, ela basta para identificar os componentes mais importantes usados para estruturar a maioria das rotas:

- O método HTTP servido pela rota. No exemplo, o método HTTP `GET` é indicado pela propriedade `get` do objeto `app`.
- O caminho servido pela rota. Quando o cliente não especifica um caminho para a solicitação, o servidor usa o diretório raiz, que é o diretório base reservado para uso do servidor web. Um exemplo posterior neste capítulo usa o caminho `/echo`, que corresponde a uma solicitação feita a `myserver:8080/echo`.
- A função executada quando o servidor recebe uma solicitação nesta rota, geralmente escrita de forma abreviada como uma *função de seta* porque a sintaxe `=>` aponta para a definição da função sem nome. O parâmetro `req` (abreviação de "request") e o parâmetro `res` (abreviação de "response") fornecem detalhes sobre a conexão, passada para a função pela própria instância do

aplicativo.

Método POST

Para estender a funcionalidade de nosso servidor de teste, vamos ver como definir uma rota para o método HTTP `POST`. Ele é usado pelos clientes que precisam enviar ao servidor dados extras, além dos que estão incluídos no cabeçalho da solicitação. A opção `--data` do comando `curl` invoca automaticamente o método `POST` e inclui o conteúdo que será enviado ao servidor via `POST`. A linha `POST / HTTP/1.1` na saída a seguir mostra que o método `POST` foi empregado. No entanto, nosso servidor definiu apenas um método `GET`, por isso ocorre um erro quando usamos `curl` para enviar uma solicitação via `POST`:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> POST / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
> Content-Length: 37
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 37 out of 37 bytes
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< X-Powered-By: Express
< Content-Security-Policy: default-src 'none'
< X-Content-Type-Options: nosniff
< Content-Type: text/html; charset=utf-8
< Content-Length: 140
< Date: Sat, 03 Jul 2021 02:22:45 GMT
< Connection: keep-alive
<
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot POST /</pre>
</body>
</html>
* Connection #0 to host myserver left intact
```

No exemplo anterior, executar `curl` com o parâmetro `--data message="This is the POST request body"` equivale a enviar um formulário contendo um campo de texto chamado `message` preenchido com `This is the POST request body`.

Como o servidor foi configurado com apenas uma rota para o caminho `/`, e essa rota responde apenas ao método HTTP `GET`, o cabeçalho de resposta inclui a linha `HTTP/1.1 404 Not Found`. Além disso, o Express gerou automaticamente uma curta resposta HTML com o aviso `Cannot POST`.

Agora que vimos como gerar uma solicitação `POST` usando o `curl`, vamos escrever um programa

Express capaz de tratar essa solicitação com sucesso.

Primeiro, note que o campo `Content-Type` no cabeçalho da solicitação diz que os dados enviados pelo cliente estão no formato `application/x-www-form-urlencoded`. O Express não reconhece esse formato por padrão, por isso precisamos usar o módulo `express.urlencoded`. Quando incluímos esse módulo, o objeto `req` — passado como parâmetro para a função de manipulação (handler) — tem configurada a propriedade `req.body.message`, que corresponde ao campo `mensagem` enviado pelo cliente. O módulo é carregado com `app.use`, que deve ser posto antes da declaração das rotas:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.urlencoded({ extended: true }))
```

Feito isso, bastaria alterar `app.get` para `app.post` na rota existente para atender às solicitações feitas via `POST` e para recuperar o corpo da solicitação:

```
app.post('/', (req, res) => {
  res.send(req.body.message)
})
```

Em vez de substituir a rota, outra possibilidade seria simplesmente adicionar essa nova rota, já que o Express identifica o método HTTP no cabeçalho da solicitação e usa a rota apropriada. Como estamos interessados em adicionar mais de uma funcionalidade a este servidor, é conveniente separar cada uma com seu próprio caminho, como `/echo` e `/ip`.

Manipulador de caminhos e funções

Tendo definido qual método HTTP responderá à solicitação, precisamos definir um caminho específico para o recurso e uma função que processe e gere uma resposta ao cliente.

Para expandir a funcionalidade `echo` do servidor, podemos definir uma rota usando o método `POST` com o caminho `/echo`:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

O parâmetro `req` da função de manipulação (handler) contém todos os detalhes da solicitação armazenados como propriedades. O conteúdo do campo `message` no corpo da solicitação está disponível na propriedade `req.body.message`. O exemplo simplesmente envia este campo de volta ao cliente através da chamada `res.send(req.body.message)`.

Lembre-se de que as alterações feitas só entram em vigor depois que o servidor é reiniciado. Como estamos executando o servidor a partir de uma janela de terminal para os exemplos dados neste capítulo, você pode desligar o servidor pressionando `Ctrl` + `C` nesse terminal. Em seguida, execute novamente o servidor por meio do comando `node index.js`. A resposta obtida pelo cliente para a solicitação `curl` mostrada anteriormente agora é bem-sucedida:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
This is the POST request body
```

Outras maneiras de passar e devolver informações em uma solicitação GET

Pode ser excessivo usar o método HTTP `POST` para enviar apenas mensagens de texto curtas, como a usada no exemplo. Nesses casos, os dados podem ser enviados em uma *string de solicitação* iniciada por um ponto de interrogação. Assim, a string `?message=This+is+the+message` poderia ser incluída no caminho de solicitação do método HTTP `GET`. Os campos usados na string de solicitação estão disponíveis para o servidor na propriedade `req.query`. Portanto, um campo denominado `message` está disponível na propriedade `req.query.message`. Outra maneira de enviar dados através do método HTTP `GET` é usar os *parâmetros de rota* do Express:

```
app.get('/echo/:message', (req, res) => {
  res.send(req.params.message)
})
```

A rota neste exemplo corresponde às solicitações feitas com o método `GET` usando o caminho `/echo/:message`, onde `:message` é um espaço reservado para qualquer termo enviado com esse rótulo pelo cliente. Esses parâmetros estão acessíveis na propriedade `req.params`. Com esta nova rota, a função `echo` do servidor pode ser solicitada de forma mais sucinta pelo cliente:

```
$ curl http://myserver:8080/echo/hello
hello
```

Em outras situações, as informações de que o servidor precisa para processar a solicitação não precisam ser fornecidas explicitamente pelo cliente. Por exemplo, o servidor tem outra maneira de recuperar o endereço IP público do cliente. Essa informação está presente no objeto `req` por padrão,

na propriedade `req.ip`:

```
app.get('/ip', (req, res) => {
  res.send(req.ip)
})
```

Agora o cliente pode solicitar o caminho `/ip` com o método `GET` para encontrar seu próprio endereço IP público:

```
$ curl http://myserver:8080/ip
187.34.178.12
```

Outras propriedades do objeto `req` podem ser modificadas pelo cliente, especialmente os cabeçalhos de solicitação disponíveis em `req.headers`. A propriedade `req.headers.user-agent`, por exemplo, identifica qual programa está fazendo a solicitação. Embora não seja uma prática comum, o cliente pode alterar o conteúdo deste campo; assim, o servidor não deve usá-lo para identificar de forma confiável um cliente específico. É ainda mais importante validar os dados fornecidos explicitamente pelo cliente, evitando assim inconsistências nos limites e formatos que podem afetar adversamente o aplicativo.

Ajustes à resposta

Como vimos nos exemplos anteriores, o parâmetro `res` é responsável por retornar uma resposta ao cliente. Além disso, o objeto `res` pode alterar outros aspectos da resposta. Você deve ter notado que, embora as respostas que implementamos até agora sejam apenas breves mensagens de texto puro, o cabeçalho `Content-Type` das respostas está usando `text/html; charset=utf-8`. Embora isso não impeça que a resposta em texto puro seja aceita, será mais correto redefinir este campo no cabeçalho da resposta como `text/plain` com a configuração `res.type('text/plain')`.

Outros tipos de ajustes de resposta envolvem o uso de *cookies*, que permitem ao servidor identificar um cliente que já fez uma solicitação anteriormente. Os cookies são importantes para a utilização de recursos avançados, como a criação de sessões privadas que associam solicitações a um usuário específico, mas aqui veremos apenas um exemplo simples de como usar um cookie para identificar um cliente que já acessou o servidor.

Dado o design modularizado do Express, o gerenciamento de cookies deve ser instalado com o comando `npm` antes de ser usado no script:

```
$ npm install cookie-parser
```

Após a instalação, o gerenciamento de cookies precisa ser incluído no script do servidor. A seguinte definição deve ser adicionada perto do início do arquivo:

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

Para ilustrar o uso dos cookies, vamos modificar a função de manipulação da rota com o caminho raiz `/` já existente no script. O objeto `req` possui uma propriedade `req.cookies`, em que os cookies enviados no cabeçalho da solicitação são preservados. O objeto `res`, por sua vez, possui um método `res.cookie()` que cria um novo cookie a ser enviado ao cliente. A função de manipulação no exemplo a seguir verifica se um cookie com o nome `known` existe na solicitação. Se tal cookie não existir, o servidor pressupõe que se trata de um visitante que chegou ao site pela primeira vez e envia a ele um cookie com esse nome através da chamada `res.cookie('known', '1')`. Atribuímos arbitrariamente o valor `1` ao cookie porque ele precisa ter algum conteúdo, mas o servidor não consulta esse valor. Este aplicativo pressupõe que a simples presença do cookie indica que o cliente já solicitou esta rota antes:

```
app.get('/', (req, res) => {
  res.type('text/plain')
  if ( req.cookies.known === undefined ){
    res.cookie('known', '1')
    res.send('Welcome, new visitor!')
  }
  else
    res.send('Welcome back, visitor');
})
```

Por padrão, o `curl` não usa cookies nas transações. Mas ele tem opções para armazenar (`-c cookies.txt`) e enviar cookies armazenados (`-b cookies.txt`):

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
* Added cookie known="1" for domain myserver, path /, expire 0
< Set-Cookie: known=1; Path=/
< Content-Length: 21
< ETag: W/"15-l7qrxcqicl4xv6EfA5fZFWCFrgY"
< Date: Sat, 03 Jul 2021 23:45:03 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome, new visitor!
```

Como esse comando foi o primeiro acesso desde que os cookies foram implementados no servidor, o cliente não tinha cookies para incluir na solicitação. Como seria de se esperar, o servidor não identificou o cookie na solicitação e, portanto, incluiu o cookie nos cabeçalhos de resposta, conforme indicado na linha `Set-Cookie: known=1; Path=/` da saída. Como habilitamos os cookies em `curl`, uma nova solicitação incluirá o cookie `known=1` nos cabeçalhos da solicitação, permitindo que o servidor identifique a presença do cookie:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
>Accept: /
> Cookie: known=1
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
< Content-Length: 21
< ETag: W/"15-ATq2flQYtLMYIUpJwwpb5SjV9Ww"
< Date: Sat, 03 Jul 2021 23:45:47 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome back, visitor
```

Segurança dos cookies

O desenvolvedor deve estar ciente das potenciais vulnerabilidades ao usar cookies para identificar os clientes que fazem solicitações. Os invasores podem usar técnicas como *cross-site scripting* (XSS) e *cross-site request forgery* (CSRF) para roubar os cookies de um cliente e, assim, personificá-lo ao fazer uma solicitação ao servidor. De modo geral, esses tipos de ataques usam campos de comentários não validados ou URLs construídas meticulosamente para inserir código JavaScript malicioso na página. Quando executado por um cliente autêntico, esse código pode copiar cookies válidos e armazená-los, ou encaminhá-los para outro destino.

Portanto, especialmente em aplicativos profissionais, é importante instalar e empregar recursos mais especializados do Express, conhecidos como *middleware*. Os módulos `express-session` ou `cookie-session` permitem um controle mais completo e seguro sobre a sessão e o gerenciamento de cookies. Esses componentes oferecem controles extras para evitar que os cookies sejam desviados de seu emissor original.

Exercícios Guiados

1. Como o conteúdo do campo `comment`, enviado dentro de uma string de consulta do método HTTP `GET`, pode ser lido em uma função de manipulação?

2. Escreva uma rota que use o método HTTP `GET` e o caminho `/agent` para enviar de volta ao cliente o conteúdo do cabeçalho `user-agent`.

3. O Express.js tem um recurso chamado *parâmetros de rota*, onde um caminho como `/user/:name` pode ser usado para receber o parâmetro `name` enviado pelo cliente. Como o parâmetro `name` pode ser acessado de dentro da função de manipulação da rota?

Exercícios Exploratórios

1. Se o nome de host de um servidor é `myserver`, qual rota do Express receberia o envio no formulário abaixo?

```
<form action="/contact/feedback" method="post"> ... </form>
```

2. Durante o desenvolvimento do servidor, o programador não consegue ler a propriedade `req.body`, mesmo depois de verificar se o cliente está enviando o conteúdo corretamente através do método HTTP `POST`. Qual é a causa provável desse problema?

3. O que acontece quando o servidor tem uma rota definida para o caminho `/user/:name` e o cliente faz uma solicitação para `/user/`?

Resumo

Esta lição explica como escrever scripts do Express para receber e lidar com solicitações HTTP. O Express usa o conceito de *rotas* para definir os recursos disponíveis aos clientes, o que lhe dá grande flexibilidade para construir servidores para qualquer tipo de aplicativo web. Esta lição aborda os seguintes conceitos e procedimentos:

- Rotas que usam os métodos HTTP `GET` e HTTP `POST`.
- Como os dados de formulários são armazenados no objeto `request`.
- Como usar parâmetros de rota.
- Personalização de cabeçalhos de resposta.
- Gerenciamento básico de cookies.

Respostas aos Exercícios Guiados

1. Como o conteúdo do campo `comment`, enviado dentro de uma string de consulta do método HTTP `GET`, pode ser lido em uma função de manipulação?

   O campo `comment` está disponível na propriedade `req.query.comment`.

2. Escreva uma rota que use o método HTTP `GET` e o caminho `/agent` para enviar de volta ao cliente o conteúdo do cabeçalho `user-agent`.

   ```
   app.get('/agent', (req, res) => {
     res.send(req.headers.user-agent)
   })
   ```

3. O Express.js tem um recurso chamado *parâmetros de rota*, onde um caminho como `/user/:name` pode ser usado para receber o parâmetro `name` enviado pelo cliente. Como o parâmetro `name` pode ser acessado de dentro da função de manipulação da rota?

   O parâmetro `name` está acessível na propriedade `req.params.name`.

Respostas aos Exercícios Exploratórios

1. Se o nome de host de um servidor é `myserver`, qual rota do Express receberia o envio no formulário abaixo?

   ```
   <form action="/contact/feedback" method="post"> ... </form>
   ```

   ```
   app.post('/contact/feedback', (req, res) => {
     ...
   })
   ```

2. Durante o desenvolvimento do servidor, o programador não consegue ler a propriedade `req.body`, mesmo depois de verificar se o cliente está enviando o conteúdo corretamente através do método HTTP `POST`. Qual é a causa provável desse problema?

   O programador não incluiu o módulo `express.urlencoded`, que permite ao Express extrair o corpo de uma solicitação.

3. O que acontece quando o servidor tem uma rota definida para o caminho `/user/:name` e o cliente faz uma solicitação para `/user/`?

   O servidor emite uma resposta `404 Not Found`, porque a rota requer que o parâmetro `:name` seja fornecido pelo cliente.

035.2 Lição 2

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	035 Programação do servidor NodeJS
Objetivo:	035.3 Noções básicas de NodeJS Express
Lição:	2 de 2

Introdução

Os servidores web têm mecanismos muito versáteis para produzir respostas às solicitações do cliente. Para algumas solicitações, basta que o servidor web forneça uma resposta estática e não processada, porque o recurso solicitado é o mesmo para qualquer cliente. Por exemplo, quando um cliente solicita uma imagem acessível a todos, o servidor precisa apenas enviar o arquivo que contém a imagem.

Mas quando as respostas são geradas dinamicamente, elas podem precisar ser mais bem estruturadas do que simples linhas escritas no script do servidor. Nesses casos, é conveniente que o servidor web seja capaz de gerar um documento completo, que pode ser interpretado e processado pelo cliente. No contexto do desenvolvimento de aplicativos web, os documentos HTML são comumente criados como modelos e mantidos separados do script do servidor, que insere dados dinâmicos em locais predeterminados no modelo apropriado e, em seguida, envia a resposta formatada ao cliente.

Os aplicativos web geralmente consomem recursos estáticos e dinâmicos. Um documento HTML, mesmo que tenha sido gerado dinamicamente, pode ter referências a recursos estáticos, como arquivos CSS e imagens. Para demonstrar como o Express ajuda a lidar com esse tipo de demanda, vamos primeiro configurar um servidor de exemplo que entrega arquivos estáticos e, em seguida,

implementar rotas que geram respostas estruturadas baseadas em modelos.

Arquivos estáticos

A primeira etapa é criar o arquivo JavaScript que será executado como servidor. Vamos seguir o mesmo padrão mostrado nas lições anteriores para criar um aplicativo Express simples: primeiro, crie um diretório chamado `server`; em seguida, instale os componentes básicos com o comando `npm`:

```
$ mkdir server
$ cd server/
$ npm init
$ npm install express
```

Para o ponto de entrada, qualquer nome de arquivo pode ser empregado, mas aqui usaremos o nome de arquivo padrão: `index.js`. A lista a seguir mostra um arquivo `index.js` básico que será usado como ponto de partida para o nosso servidor:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Não é necessário escrever código explícito para enviar um arquivo estático. O Express possui um middleware para esse fim, chamado `express.static`. Se o seu servidor precisa enviar arquivos estáticos para o cliente, basta carregar o middleware `express.static` no início do script:

```
app.use(express.static('public'))
```

O parâmetro `public` indica o diretório onde estão armazenados os arquivos que o cliente pode solicitar. Os caminhos solicitados pelos clientes não devem incluir o diretório `public`, mas apenas o nome do arquivo, ou o caminho para o arquivo relativo ao diretório `public`. Para solicitar o arquivo `public/layout.css`, por exemplo, o cliente faz uma solicitação para `/layout.css`.

Saída formatada

Embora o envio de conteúdo estático seja descomplicado, o conteúdo gerado dinamicamente pode variar muito. A criação de respostas dinâmicas com mensagens curtas facilita o teste de aplicativos em seus estágios iniciais de desenvolvimento. Veja a seguir, por exemplo, uma rota de teste que simplesmente repassa ao cliente uma mensagem enviada pelo método HTTP `POST`. A resposta pode apenas replicar o conteúdo da mensagem em texto simples, sem nenhuma formatação:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

Uma rota como essa é um bom exemplo para aprender a usar o Express e para fins de diagnóstico, onde uma resposta bruta enviada com `res.send()` é suficiente. Mas um servidor útil deve ser capaz de produzir respostas mais complexas. Assim, vamos aprender a desenvolver esse tipo de rota mais sofisticado.

Nosso novo aplicativo, em vez de apenas mandar de volta o conteúdo da solicitação atual, mantém uma lista completa das mensagens enviadas nas solicitações anteriores de cada cliente e envia de volta a lista de cada cliente quando solicitado. Existe a opção de mandar uma resposta mesclando todas as mensagens, mas outros modos de saída formatada são mais apropriados, especialmente no caso de respostas mais elaboradas.

Para receber e armazenar mensagens de clientes enviadas durante a sessão atual, precisamos primeiro incluir módulos extras para lidar com cookies e dados enviados pelo método HTTP `POST`. O único propósito do servidor de exemplo a seguir é registrar as mensagens enviadas via `POST` e exibir as mensagens enviadas anteriormente quando o cliente emitir uma solicitação `GET`. Portanto, existem duas rotas para o caminho `/`. A primeira atende às solicitações feitas com o método HTTP `POST` e a segunda atende às solicitações feitas com o método HTTP `GET`:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.static('public'))

const cookieParser = require('cookie-parser')
app.use(cookieParser())

const { v4: uuidv4 } = require('uuid')
```

```
app.use(express.urlencoded({ extended: true }))

// Array to store messages
let messages = []

app.post('/', (req, res) => {

  // Only JSON enabled requests
  if ( req.headers.accept != "application/json" )
  {
    res.sendStatus(404)
    return
  }

  // Locate cookie in the request
  let uuid = req.cookies.uuid

  // If there is no uuid cookie, create a new one
  if ( uuid === undefined )
    uuid = uuidv4()

  // Add message first in the messages array
  messages.unshift({uuid: uuid, message: req.body.message})

  // Collect all previous messages for uuid
  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  // Update cookie expiration date
  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  // Send back JSON response
  res.json(user_entries)

})

app.get('/', (req, res) => {

  // Only JSON enabled requests
```

```
  if ( req.headers.accept != "application/json" )
  {
    res.sendStatus(404)
    return
  }

  // Locate cookie in the request
  let uuid = req.cookies.uuid

  // Client's own messages
  let user_entries = []

  // If there is no uuid cookie, create a new one
  if ( uuid === undefined ){
    uuid = uuidv4()
  }
  else {
    // Collect messages for uuid
    messages.forEach( (entry) => {
      if ( entry.uuid == req.cookies.uuid )
        user_entries.push(entry.message)
    })
  }

  // Update cookie expiration date
  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  // Send back JSON response
  res.json(user_entries)

})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Mantivemos a configuração dos arquivos estáticos no topo, porque em breve será útil fornecer arquivos estáticos como `layout.css`. Além do middleware `cookie-parser` apresentado na lição anterior, o exemplo também inclui o middleware `uuid` para gerar um número de identificação único que é passado como um cookie para cada cliente que envia uma mensagem. Se eles ainda não estiverem no diretório do servidor de exemplo, instale esses módulos com o comando `npm install cookie-parser uuid`.

A matriz global `messages` armazena as mensagens enviadas por todos os clientes. Cada item desta matriz consiste em um objeto com as propriedades `uuid` e `message`.

A novidade nesse script é o método `res.json()`, usado ao final das duas rotas para gerar uma resposta no formato JSON com a matriz contendo as mensagens já enviadas pelo cliente:

```
// Send back JSON response
res.json(user_entries)
```

JSON é um formato de texto simples que permite agrupar um conjunto de dados em uma única estrutura associativa: ou seja, o conteúdo é expresso na forma de chaves e valores. O JSON é particularmente útil quando as respostas serão processadas pelo cliente. Usando esse formato, um objeto ou matriz JavaScript pode ser facilmente reconstruído no lado do cliente com todas as propriedades e índices do objeto original no servidor.

Como estamos estruturando todas as mensagens em JSON, recusamos as solicitações que não contenham `application/json` em seu cabeçalho `accept`:

```
// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}
```

Uma solicitação feita com um comando `curl` simples para inserir uma nova mensagem não será aceita, porque `curl`, por padrão, não especifica `application/json` no cabeçalho `accept`:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt
Not Found
```

A opção `-H "accept: application/json"` altera o cabeçalho da solicitação de forma a especificar o formato da resposta, que desta vez será aceita e respondida no formato especificado:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt -H "accept: application/json"
["My first message"]
```

Para obter mensagens usando a outra rota, o processo é semelhante, mas desta vez usando o método HTTP `GET`:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -H "accept:
application/json"
["Another message","My first message"]
```

Modelos

As respostas em formatos como o JSON são convenientes para a comunicação entre programas, mas o objetivo principal da maioria dos servidores de aplicativos web é produzir conteúdo HTML para o consumo humano. Não é uma boa ideia incorporar código HTML dentro de um código JavaScript, pois a mistura de linguagens no mesmo arquivo torna o programa mais suscetível a erros e atrapalha a manutenção do código.

O Express pode trabalhar com diferentes *mecanismos de modelo* (template engines) que separam o HTML para o conteúdo dinâmico; a lista completa pode ser encontrada no site de mecanismos de modelo do Express. Um dos mecanismos de modelo mais populares é o *Embedded JavaScript* (EJS), que permite criar arquivos HTML com tags específicas para a inserção de conteúdo dinâmico.

Como outros componentes do Express, o EJS precisa ser instalado no diretório em que o servidor está sendo executado:

```
$ npm install ejs
```

Em seguida, o mecanismo EJS deve ser definido como o renderizador padrão no script do servidor (próximo ao início do arquivo `index.js`, antes das definições de rota):

```
app.set('view engine', 'ejs')
```

A resposta gerada com o modelo é enviada ao cliente com a função `res.render()`, que recebe como parâmetros o nome do arquivo do modelo e um objeto contendo valores que estarão acessíveis de dentro do modelo. As rotas usadas no exemplo anterior podem ser reescritas para gerar respostas em HTML, bem como em JSON:

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid
```

```
  if ( uuid === undefined )
    uuid = uuidv4()

  messages.unshift({uuid: uuid, message: req.body.message})

  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  if ( req.headers.accept == "application/json" )
    res.json(user_entries)
  else
    res.render('index', {title: "My messages", messages: user_entries})

})

app.get('/', (req, res) => {

  let uuid = req.cookies.uuid

  let user_entries = []

  if ( uuid === undefined ){
    uuid = uuidv4()
  }
  else {
    messages.forEach( (entry) => {
      if ( entry.uuid == req.cookies.uuid )
        user_entries.push(entry.message)
    })
  }

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  if ( req.headers.accept == "application/json" )
    res.json(user_entries)
```

```
  else
    res.render('index', {title: "My messages", messages: user_entries})

})
```

Note que o formato da resposta depende do cabeçalho `accept` encontrado na solicitação:

```
if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})
```

Uma resposta em formato JSON é enviada apenas se o cliente fizer expressamente essa solicitação. Caso contrário, a resposta é gerada a partir do modelo `index`. A mesma matriz `user_entries` alimenta a saída em JSON e o modelo, mas o objeto usado como parâmetro para este último também possui a propriedade `title: "My messages"`, que será usada como um título dentro do template.

Modelos HTML

A exemplo dos arquivos estáticos, os arquivos que contêm modelos HTML residem em seu próprio diretório. Por padrão, o EJS pressupõe que os arquivos de modelo estão no diretório `views/`. No exemplo, um modelo chamado `index` foi usado, por isso o EJS procura pelo arquivo `views/index.ejs`. A lista a seguir é o conteúdo de um modelo `views/index.ejs` simples que pode ser usado com o código de exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title><%= title %></title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>
<% messages.forEach( (message) => { %>
<li><%= message %></li>
<% }) %>
</ul>

</div>

</body>
</html>
```

A primeira tag EJS especial é o elemento `<title>` na seção `<head>`:

```
<%= title %>
```

Durante o processo de renderização, essa tag especial será substituída pelo valor da propriedade `title` do objeto passado como parâmetro para a função `res.render()`.

A maior parte do modelo é composta de código HTML convencional; assim, o modelo contém o formulário HTML para o envio de novas mensagens. O servidor de teste responde aos métodos HTTP `GET` e `POST` para o mesmo caminho `/`, por isso temos os atributos `action="/"` e `method="post"` na tag de formulário.

Outras partes do modelo são uma mistura de código HTML e tags EJS. O EJS tem tags para fins específicos dentro do modelo:

`<% … %>`

Inserts flow control. No content is directly inserted by this tag, but it can be used with JavaScript structures to choose, repeat, or suppress sections of HTML. Example starting a loop: `<% messages.forEach( (message) => { %>`

`<%# … %>`

Define um comentário cujo conteúdo é ignorado pelo analisador. Ao contrário dos comentários escritos em HTML, esses comentários não são visíveis para o cliente.

`<%= … %>`

Insere o conteúdo escapado da variável. É importante escapar o conteúdo desconhecido para evitar a execução de código em JavaScript, o que abriria brechas para ataques de Cross-Site Scripting (XSS). Exemplo: `<%= title %>`

`<%- … %>`

Insere o conteúdo da variável sem escapar.

A combinação de código HTML e tags EJS é evidente no trecho em que as mensagens do cliente são renderizadas como uma lista HTML:

```
<ul>
<% messages.forEach( (message) => { %>
<li><%= message %></li>
<% }) %>
</ul>
```

Neste trecho, a primeira tag `<% … %>` inicia uma instrução `forEach` que percorre todos os elementos da matriz `message`. Os delimitadores `<%` e `%>` permitem controlar os trechos de HTML. Um novo item de lista HTML, `<li><%= message %></li>`, será criado para cada elemento de `messages`. Com essas mudanças, o servidor enviará a resposta em HTML quando uma solicitação como a seguinte for recebida:

```
$ curl http://myserver:8080/ --data message="This time" -c cookies.txt -b
cookies.txt
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My messages</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
</p>
</form>

<ul>

<li>This time</li>

<li>in HTML</li>

</ul>

</div>

</body>
</html>
```

A separação entre o código de processamento das solicitações e o código de apresentação da resposta torna o código mais limpo e permite que uma equipe divida o desenvolvimento do aplicativo entre diferentes especialistas. Um webdesigner, por exemplo, pode se concentrar nos arquivos de modelo em `views/` e nas folhas de estilo relacionadas, que são fornecidas como arquivos estáticos armazenados no diretório `public/` no servidor de exemplo.

Exercícios Guiados

1. Como o `express.static` deve ser configurado para que os clientes possam solicitar arquivos no diretório `assets`?

2. Como o tipo de resposta, que é especificado no cabeçalho da solicitação, pode ser identificado em uma rota do Express?

3. Qual método do parâmetro de rota `res` (resposta) gera uma resposta no formato JSON a partir de uma matriz JavaScript chamada `content`?

Exercícios Exploratórios

1. Por padrão, os arquivos de modelo do Express ficam no diretório `views`. Como essa configuração pode ser modificada para que os arquivos de modelo sejam armazenados em `templates`?

2. Suponha que um cliente recebe uma resposta em HTML sem título (ou seja, `<title></title>`). Depois de verificar o modelo EJS, o desenvolvedor encontra a tag `<title><% title %></title>` na seção `head` do arquivo. Qual é a causa provável do problema?

3. Use tags de modelo EJS para escrever uma tag HTML `<h2></h2>` com o conteúdo da variável JavaScript `h2`. Essa tag deve ser renderizada somente se a variável `h2` não estiver vazia.

Resumo

Esta lição trata dos métodos básicos fornecidos pelo Express.js para gerar respostas estáticas e formatadas, mas dinâmicas. Não é difícil configurar um servidor HTTP para os arquivos estáticos, e o sistema de modelos EJS constitui uma maneira fácil de gerar conteúdo dinâmico a partir de arquivos HTML. Esta lição aborda os seguintes conceitos e procedimentos:

- Uso do `express.static` para respostas de arquivos estáticos.
- Como criar uma resposta correspondente ao tipo de conteúdo definido no cabeçalho da solicitação.
- Respostas estruturadas em JSON.
- Uso de tags EJS em modelos baseados em HTML.

Respostas aos Exercícios Guiados

1. Como o `express.static` deve ser configurado para que os clientes possam solicitar arquivos no diretório `assets`?

   É preciso adicionar uma chamada para `app.use(express.static('assets'))` no script do servidor.

2. Como o tipo de resposta, que é especificado no cabeçalho da solicitação, pode ser identificado em uma rota do Express?

   O cliente define os tipos aceitáveis no campo de cabeçalho `accept`, que é mapeado para a propriedade `req.headers.accept`.

3. Qual método do parâmetro de rota `res` (resposta) gera uma resposta no formato JSON a partir de uma matriz JavaScript chamada `content`?

   O método `res.json()`: `res.json(content)`.

Respostas aos Exercícios Exploratórios

1. Por padrão, os arquivos de modelo do Express ficam no diretório `views`. Como essa configuração pode ser modificada para que os arquivos de modelo sejam armazenados em `templates`?

   O diretório pode ser definido nas configurações iniciais do script com `app.set('views', './templates')`.

2. Suponha que um cliente recebe uma resposta em HTML sem título (ou seja, `<title></title>`). Depois de verificar o modelo EJS, o desenvolvedor encontra a tag `<title><% title %></title>` na seção `head` do arquivo. Qual é a causa provável do problema?

   A tag `<%= %>` deve ser usada para circunscrever o conteúdo de uma variável, como em `<%= title %>`.

3. Use tags de modelo EJS para escrever uma tag HTML `<h2></h2>` com o conteúdo da variável JavaScript `h2`. Essa tag deve ser renderizada somente se a variável `h2` não estiver vazia.

   ```
   <% if ( h2 != "" ) { %>
   <h2><%= h2 %></h2>
   <% } %>
   ```

035.3 Noções básicas de SQL

Referência ao LPI objectivo

Web Development Essentials version 1.0, Exam 030, Objective 035.3

Peso

3

Áreas chave de conhecimento

- Estabelecer uma conexão de banco de dados com o NodeJS
- Recuperar dados do banco de dados no NodeJS
- Executar consultas SQL com o NodeJS
- Criar consultas SQL simples, excluindo junções (joins)
- Entender as chaves primárias
- Variáveis de escape usadas em consultas SQL
- Noções de injeções de SQL

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- Módulo `sqlite3` NPM
- `Database.run()`, `Database.close()`, `Database.all()`, `Database.get()`, `Database.each()`
- `CREATE TABLE`
- `INSERT`, `SELECT`, `DELETE`, `UPDATE`

035.3 Lição 1

Certificação:	Web Development Essentials
Versão:	1.0
Tópico:	035 Programação do servidor NodeJS
Objetivo:	035.3 Noções básicas de SQL
Lição:	1 de 1

Introdução

Embora seja possível escrever suas próprias funções para implementar um armazenamento persistente, costuma ser mais conveniente usar um sistema de gerenciamento de banco de dados para acelerar o desenvolvimento e garantir melhor segurança e estabilidade aos dados formatados em tabela. A estratégia mais popular para armazenar dados organizados em tabelas inter-relacionadas, especialmente quando essas tabelas são pesadamente consultadas e atualizadas, é instalar um banco de dados relacional que suporte *Structured Query Language* (SQL), uma linguagem voltada para bancos de dados relacionais. O Node.js oferece suporte a diversos sistemas de gerenciamento de banco de dados SQL. Seguindo os princípios de portabilidade e execução do espaço de usuário adotados pelo Node.js Express, o SQLite é uma boa escolha para o armazenamento persistente dos dados usados por este tipo de servidor HTTP.

SQL

A Structured Query Language é específica aos bancos de dados. As operações de escrita e leitura são expressas em frases chamadas *declarações* e *consultas*. Tanto as declarações quanto as consultas são compostas de *cláusulas*, que definem as condições para a execução da operação.

Nomes e endereços de email, por exemplo, podem ser armazenados em uma tabela de banco de dados contendo os campos `name` e `email`. Um banco de dados pode conter múltiplas tabelas e, portanto, cada tabela deve ter um nome exclusivo. Se usarmos o nome `contacts` para a tabela de nomes e emails, um novo registro poderá ser inserido com a seguinte *declaração*:

```
INSERT INTO contacts (name, email) VALUES ("Carol", "carol@example.com");
```

Esta declaração de inserção é composta pela cláusula `INSERT INTO`, que define a tabela e os campos onde os dados serão inseridos. A segunda cláusula, `VALUES`, define os valores a inserir. Não é necessário colocar as cláusulas em maiúsculas, mas costumamos fazer isso para ser mais fácil reconhecer as palavras-chave do SQL em uma declaração ou consulta.

Uma consulta na tabela de contatos seria feita de maneira semelhante, mas usando a cláusula `SELECT`:

```
SELECT email FROM contacts;
dave@example.com
carol@example.com
```

Neste caso, a cláusula `SELECT email` seleciona um campo dentre as entradas da tabela `contacts`. A cláusula `WHERE` restringe a consulta a linhas específicas:

```
SELECT email FROM contacts WHERE name = "Dave";
dave@example.com
```

O SQL tem muitas outras cláusulas, que veremos em seções posteriores. Antes disso, é necessário entender como é possível integrar o banco de dados SQL com o Node.js.

SQLite

O SQLite é provavelmente a solução mais simples para incorporar recursos de banco de dados SQL a um aplicativo. Ao contrário de outros sistemas populares de gerenciamento de banco de dados, o SQLite não é um servidor de banco de dados ao qual um cliente se conecta. Em vez disso, o SQLite fornece um conjunto de funções que permitem ao desenvolvedor criar um banco de dados como um arquivo convencional. No caso de um servidor HTTP implementado com o Node.js Express, esse arquivo geralmente está localizado no mesmo diretório que o script do servidor.

Antes de usar o SQLite em Node.js, é necessário instalar o módulo `sqlite3`. Execute o seguinte comando no diretório de instalação do servidor, ou seja, o diretório que contém o script Node.js a ser executado.

```
$ npm install sqlite3
```

Esteja ciente de que existem diversos módulos que suportam o SQLite, como `better-sqlite3`, cujo uso é sutilmente diferente de `sqlite3`. Os exemplos nesta lição valem para o módulo `sqlite3` e podem não funcionar como esperado caso você escolha outro módulo.

Abrindo o banco de dados

Para demonstrar como um servidor Node.js Express pode trabalhar com um banco de dados SQL, vamos escrever um script que armazena e exibe as mensagens enviadas por um cliente identificado por um cookie. As mensagens são enviadas pelo cliente através do método HTTP POST e a resposta do servidor pode ser formatada em JSON ou HTML (a partir de um template), dependendo do formato solicitado pelo cliente. Esta lição não entrará em detalhes sobre o uso de métodos HTTP, cookies e modelos. Os trechos de código mostrados presumem que você já tenha um servidor Node.js Express em que esses recursos estão configurados e disponíveis.

A forma mais simples de armazenar as mensagens enviadas pelo cliente é colocá-las em uma matriz global, onde cada mensagem enviada anteriormente é associada a uma chave de identificação única para cada cliente. Essa chave pode ser enviada ao cliente na forma de um cookie, que é apresentado ao servidor em solicitações futuras para recuperar as mensagens anteriores.

Todavia, essa abordagem tem um ponto fraco: como as mensagens são armazenadas apenas em uma matriz global, todas serão perdidas quando a sessão atual do servidor for encerrada. Essa é uma das vantagens de se trabalhar com bancos de dados, pois os dados são armazenados de forma persistente e não são perdidos se o servidor for reiniciado.

Usando o arquivo `index.js` como script do servidor principal, podemos incorporar o módulo `sqlite3` e indicar o arquivo que serve de banco de dados, da seguinte forma:

```
const sqlite3 = require('sqlite3')
const db = new sqlite3.Database('messages.sqlite3');
```

Se ele ainda não existir, o arquivo `messages.sqlite3` será criado no mesmo diretório do arquivo `index.js`. Dentro deste único arquivo, serão armazenadas todas as estruturas e dados respectivos. Todas as operações de banco de dados realizadas no script serão intermediadas pela constante `db`, que é o nome dado ao novo objeto `sqlite3` que abre o arquivo `messages.sqlite3`.

Estrutura de uma tabela

Nenhum dado pode ser inserido no banco de dados até que pelo menos uma tabela seja criada. As tabelas são criadas com a declaração `CREATE TABLE`:

```
db.run('CREATE TABLE IF NOT EXISTS messages (id INTEGER PRIMARY KEY AUTOINCREMENT,
uuid CHAR(36), message TEXT)')
```

O método `db.run()` é usado para executar declarações SQL no banco de dados. A declaração em si é escrita como um parâmetro para o método. Embora as declarações SQL devam terminar com um ponto e vírgula quando inseridas em um processador de linha de comando, o ponto e vírgula é opcional nas declarações passadas como parâmetros em um programa.

Como o método `run` será executado toda vez que o script for executado com `node index.js`, a declaração SQL inclui a cláusula condicional `IF NOT EXISTS` para evitar erros em execuções futuras quando a tabela `messages` já existir.

Os campos que compõem a tabela `messages` são `id`, `uuid` e `message`. O campo `id` é um número inteiro único usado para identificar cada entrada na tabela, por isso é criado como `PRIMARY KEY`, ou chave primária. As chaves primárias não podem ser nulas e não pode haver duas chaves primárias idênticas na mesma tabela. Assim, quase toda tabela SQL possui uma chave primária para rastrear o conteúdo da tabela. Embora seja possível escolher explicitamente o valor da chave primária de um novo registro (desde que ela ainda não exista na tabela), é conveniente que a chave seja gerada automaticamente. O sinalizador `AUTOINCREMENT` no campo `id` é usado para esse fim.

NOTE

A configuração explícita de chaves primárias no SQLite é opcional, porque o próprio SQLite cria uma chave primária automaticamente. Conforme declarado na documentação do SQLite: "No SQLite, as linhas da tabela normalmente incluem um `ROWID`, um número inteiro assinado de 64 bits que é único entre todas as linhas da mesma tabela. Se uma tabela contém uma coluna do tipo `INTEGER PRIMARY KEY`, essa coluna se torna um alias para o `ROWID`. Podemos acessar o `ROWID` usando um dentre quatro nomes: os três nomes originais descritos acima ou o nome dado à coluna `INTEGER PRIMARY KEY`. Todos esses nomes são aliases uns para os outros e funcionam igualmente bem em qualquer contexto."

Os campos `uuid` e `message` armazenam, respectivamente, a identificação do cliente e o conteúdo da mensagem. Um campo do tipo `CHAR(36)` armazena uma quantidade fixa de 36 caracteres e um campo do tipo `TEXT` armazena textos de comprimento arbitrário.

Inserção de dados

A principal função do nosso servidor de exemplo é armazenar mensagens vinculadas ao cliente que as enviou. O cliente envia a mensagem no campo `message` no corpo da solicitação enviada com o método HTTP POST. A identificação do cliente está em um cookie chamado `uuid`. Munidos dessas informações, podemos escrever a rota para o Express inserir novas mensagens no banco de dados:

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if ( uuid === undefined )
    uuid = uuidv4()

  // Insert new message into the database
  db.run('INSERT INTO messages (uuid, message) VALUES (?, ?)', uuid, req.body
.message)

  // If an error occurrs, err object contains the error message.
  db.all('SELECT id, message FROM messages WHERE uuid = ?', uuid, (err, rows) => {

    let expires = new Date(Date.now());
    expires.setDate(expires.getDate() + 30);
    res.cookie('uuid', uuid, { expires: expires })

    if ( req.headers.accept == "application/json" )
      res.json(rows)
    else
      res.render('index', {title: "My messages", rows: rows})

  })

})
```

Desta vez, o método `db.run()` executa uma declaração de inserção, mas note que `uuid` e `req.body.message` não são escritos diretamente na linha da declaração. Em vez disso, os valores foram substituídos por pontos de interrogação. Cada ponto de interrogação corresponde a um parâmetro que segue a declaração SQL no método `db.run()`.

O uso de pontos de interrogação como espaços reservados na declaração que é executada no banco de dados permite ao SQLite distinguir mais facilmente entre os elementos estáticos da declaração e seus dados variáveis. Esta estratégia permite que o SQLite *escape* ou *sanitize* o conteúdo da variável que faz parte da declaração, evitando uma falha de segurança comum chamada *injeção de SQL*. Nesse

ataque, usuários mal-intencionados inserem declarações SQL nos dados variáveis, na esperança de que essas declarações sejam executadas inadvertidamente; a sanitização impede o ataque, desativando caracteres perigosos nos dados.

Consultas

Como mostrado no código de exemplo, nossa intenção é usar a mesma rota para inserir novas mensagens no banco de dados e gerar a lista de mensagens enviadas anteriormente. O método `db.all()` retorna a coleção de todas as entradas na tabela que correspondem aos critérios definidos na consulta.

Ao contrário das declarações executadas por `db.run()`, `db.all()` gera uma lista de registros que são tratados pela função de seta designada no último parâmetro:

```
(err, rows) => {}
```

Esta função, por sua vez, leva dois parâmetros: `err` e `rows`. O parâmetro `err` será usado se ocorrer um erro que impeça a execução da consulta. Se ela for bem-sucedida, todos os registros ficam disponíveis na matriz `rows`, onde cada elemento é um objeto correspondente a um único registro da tabela. As propriedades deste objeto correspondem aos nomes dos campos indicados na consulta: `uuid` e `message`.

A matriz `rows` é uma estrutura de dados JavaScript. Como tal, pode ser usada para gerar respostas com métodos fornecidos pelo Express, como `res.json()` e `res.render()`. Quando renderizado dentro de um modelo EJS, um loop convencional pode listar todos os registros:

```
<ul>
<% rows.forEach( (row) => { %>
<li><strong><%= row.id %></strong>: <%= row.message %></li>
<% }) %>
</ul>
```

Em vez de preencher a matriz `rows` com todos os registros retornados pela consulta, em alguns casos pode ser mais conveniente tratar cada registro individualmente com o método `db.each()`. A sintaxe do método `db.each()` é semelhante à do método `db.all()`, mas o parâmetro `row` em `(err, row) ▯ {}` corresponde a um único registro por vez.

Alterando o conteúdo do banco de dados

Até agora, nosso cliente só pode adicionar e consultar mensagens no servidor. Como o cliente agora

conhece o `id` das mensagens enviadas anteriormente, podemos implementar uma função para modificar um registro específico. A mensagem modificada também pode ser enviada para uma rota configurada com o método HTTP POST, mas desta vez com um parâmetro de rota para capturar o `id` fornecido pelo cliente no caminho da solicitação:

```
app.post('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    // 401 Unauthorized
    res.sendStatus(401)
  }
  else {

    // Update the stored message
    // using named parameters
    let param = {
      $message: req.body.message,
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('UPDATE messages SET message = $message WHERE id = $id AND uuid = $uuid
', param, function(err){

      if ( this.changes > 0 )
      {
        // A 204 (No Content) status code means the action has
        // been enacted and no further information is to be supplied.
        res.sendStatus(204)
      }
      else
        res.sendStatus(404)

    })
  }
})
```

Esta rota demonstra como usar as cláusulas `UPDATE` e `WHERE` para modificar um registro existente. Uma diferença importante em relação aos exemplos anteriores é o uso de *parâmetros nomeados*, nos quais os valores são agrupados em um único objeto (`param`) e passados para o método `db.run()` ao invés de especificar cada valor por si mesmo. Neste caso, os nomes dos campos (precedidos por `$`)

são as propriedades do objeto. Os parâmetros nomeados permitem o uso de nomes de campo (precedidos por `$`) como espaços reservados ao invés de pontos de interrogação.

Uma declaração como a do exemplo não causará nenhuma modificação no banco de dados se a condição imposta pela cláusula `WHERE` não encontrar uma correspondência entre os registros da tabela. Para avaliar se algum registro foi modificado pela declaração, uma função de retorno de chamada pode ser usada como o último parâmetro do método `db.run()`. Dentro da função, o número de registros alterados pode ser consultado em `this.changes`. Note que as funções de seta não podem ser usadas neste caso, porque apenas as funções regulares na forma `function(){}` definem o objeto `this`.

A remoção de um registro é muito semelhante à sua modificação. Podemos, por exemplo, continuar usando o parâmetro de rota `:id` para identificar a mensagem a ser excluída, mas desta vez em uma rota invocada pelo método HTTP DELETE do cliente:

```
app.delete('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    res.sendStatus(401)
  }
  else {
    // Named parameters
    let param = {
    $id: req.params.id,
    $uuid: uuid
    }

    db.run('DELETE FROM messages WHERE id = $id AND uuid = $uuid', param, function
(err){
      if ( this.changes > 0 )
        res.sendStatus(204)
      else
        res.sendStatus(404)
    })
  }
})
```

Os registros são excluídos de uma tabela com a cláusula `DELETE FROM`. Mais uma vez, usamos a função de retorno de chamada para avaliar quantas entradas foram removidas da tabela.

Fechando o banco de dados

Uma vez definido, o objeto `db` pode ser referenciado a qualquer momento durante a execução do script, pois o arquivo de banco de dados permanece aberto durante a sessão atual. Não é comum fechar o banco de dados durante a execução do script.

No entanto, vale a pena instaurar uma função para fechar o banco de dados, de forma a evitar o seu fechamento abrupto quando o processo do servidor é concluído. Embora improvável, o encerramento abrupto do banco de dados pode resultar em inconsistências caso os dados da memória ainda não estejam registrados no arquivo. Por exemplo, pode ocorrer perda de dados com um desligamento abrupto do banco de dados se o script for encerrado pelo usuário pressionando o atalho de teclado `Ctrl` + `C`.

Nesse caso do `Ctrl` + `C`, o método `process.on()` pode interceptar os sinais enviados pelo sistema operacional e executar um desligamento ordenado do banco de dados e do servidor:

```
process.on('SIGINT', () => {
  db.close()
  server.close()
  console.log('HTTP server closed')
})
```

O atalho `Ctrl` + `C` invoca o sinal do sistema operacional `SIGINT`, que encerra um programa em primeiro plano no terminal. Antes de encerrar o processo ao receber o sinal `SIGINT`, o sistema invoca a função de retorno de chamada (o último parâmetro no método `process.on()`). Dentro da função de retorno de chamada, podemos colocar qualquer código de limpeza, em particular o método `db.close()`, para fechar o banco de dados, e `server.close()`, que fecha graciosamente a própria instância do Express.

Exercícios Guiados

1. Qual é a finalidade de uma chave primária em uma tabela de banco de dados SQL?

2. Qual a diferença entre as consultas usando `db.all()` e `db.each()`?

3. Por que é importante usar espaços reservados e não incluir dados enviados pelo cliente diretamente em uma declaração ou consulta SQL?

Exercícios Exploratórios

1. Qual método do módulo `sqlite3` pode ser usado para retornar apenas uma entrada da tabela, mesmo se a consulta corresponder a diversas entradas?

2. Suponha que a matriz `rows` foi passada como um parâmetro para uma função de retorno de chamada e contém o resultado de uma consulta feita com `db.all()`. Como um campo chamado `price`, presente na primeira posição de `rows`, pode ser referenciado dentro da função de retorno de chamada?

3. O método `db.run()` executa declarações de modificação do banco de dados, como `INSERT INTO`. Depois de inserir um novo registro em uma tabela, como seria possível recuperar a chave primária do registro recém-inserido?

Resumo

Esta lição trata do uso básico de bancos de dados SQL em aplicativos Node.js Express. O módulo `sqlite3` oferece uma maneira simples de armazenar dados persistentes em um banco de dados SQLite, onde um único arquivo contém todo o banco de dados e não requer um servidor de banco de dados especializado. Esta lição aborda os seguintes conceitos e procedimentos:

- Como estabelecer uma conexão de banco de dados a partir do Node.js.
- Como criar uma tabela simples e o papel das chaves primárias.
- Uso da declaração SQL `INSERT INTO` para adicionar novos dados de dentro do script.
- Consultas SQL usando os métodos padrão do SQLite e as funções de retorno de chamada.
- Alteração de dados no banco de dados usando as declarações SQL `UPDATE` e `DELETE`.

Respostas aos Exercícios Guiados

1. Qual é a finalidade de uma chave primária em uma tabela de banco de dados SQL?

   A chave primária é o campo de identificação exclusivo para cada registro em uma tabela de banco de dados.

2. Qual a diferença entre as consultas usando `db.all()` e `db.each()`?

   O método `db.all()` invoca a função de retorno de chamada com uma única matriz contendo todas as entradas correspondentes à consulta. O método `db.each()` invoca a função de retorno de chamada para cada linha de resultado.

3. Por que é importante usar espaços reservados e não incluir dados enviados pelo cliente diretamente em uma declaração ou consulta SQL?

   Com espaços reservados, os dados enviados pelo usuário são escapados antes de serem incluídos na consulta ou declaração. Isso dificulta os ataques de injeção de SQL, nos quais as declarações SQL são postas dentro de dados variáveis em uma tentativa de executar operações arbitrárias no banco de dados.

Respostas aos Exercícios Exploratórios

1. Qual método do módulo `sqlite3` pode ser usado para retornar apenas uma entrada da tabela, mesmo se a consulta corresponder a diversas entradas?

   O método `db.get()` tem a mesma sintaxe de `db.all()`, mas retorna apenas a primeira entrada correspondente à consulta.

2. Suponha que a matriz `rows` foi passada como um parâmetro para uma função de retorno de chamada e contém o resultado de uma consulta feita com `db.all()`. Como um campo chamado `price`, presente na primeira posição de `rows`, pode ser referenciado dentro da função de retorno de chamada?

   Cada item em `rows` é um objeto cujas propriedades correspondem aos nomes dos campos do banco de dados. Portanto, o valor do campo `price` no primeiro resultado está em `rows[0].price`.

3. O método `db.run()` executa declarações de modificação do banco de dados, como `INSERT INTO`. Depois de inserir um novo registro em uma tabela, como seria possível recuperar a chave primária do registro recém-inserido?

   Uma função regular na forma `function(){}` pode ser usada como a função de retorno de chamada do método `db.run()`. Dentro dele, a propriedade `this.lastID` contém o valor da chave primária do último registro inserido.

Imprint

PDF gerado: 2022-04-21

Os Materiais Didáticos da LPI são uma iniciativa do Linux Professional Institute (https://lpi.org). Os Materiais Didáticos e suas traduções estão disponíveis em https://learning.lpi.org.

Para perguntas e comentários sobre esta edição, bem como sobre todo o projeto, escreva para: learning@lpi.org.