



Linux
Professional
Institute

Web Development Essentials

Version 1.0
English

030

Table of Contents

TOPIC 031: SOFTWARE DEVELOPMENT AND WEB TECHNOLOGIES	1
 031.1 Software Development Basic	2
031.1 Lesson 1	3
Introduction	3
Source Code	3
Programming Languages	5
Guided Exercises	12
Explorational Exercises	13
Summary	14
Answers to Guided Exercises	15
Answers to Explorational Exercises	16
 031.2 Web Application Architecture	17
 031.2 Lesson 1	19
Introduction	19
Clients and Servers	19
The Client Side	20
Varieties of Web Clients	21
Languages of a Web Client	22
The Server Side	24
Handling Paths from Requests	24
Database Management Systems	25
Content Maintenance	25
Guided Exercises	27
Explorational Exercises	28
Summary	29
Answers to Guided Exercises	30
Answers to Explorational Exercises	31
 031.3 HTTP Basics	32
031.3 Lesson 1	34
Introduction	34
The Client's Request	35
The Response Header	38
Static and Dynamic Content	40
Caching	40
HTTP Sessions	41
Guided Exercises	43
Explorational Exercises	44
Summary	45

Answers to Guided Exercises	46
Answers to Explorational Exercises	47
TOPIC 032: HTML DOCUMENT MARKUP	48
032.1 HTML Document Anatomy	49
032.1 Lesson 1	50
Introduction	50
Anatomy of an HTML Document	50
Document Header	54
Guided Exercises	58
Explorational Exercises	59
Summary	60
Answers to Guided Exercises	61
Answers to Explorational Exercises	62
032.2 HTML Semantics and Document Hierarchy	64
032.2 Lesson 1	66
Introduction	66
Text	66
Headings	67
Line Breaks	69
Horizontal Lines	70
HTML Lists	71
Inline Text Formatting	77
Preformatted Text	82
Grouping Elements	83
HTML Page Structure	85
Guided Exercises	93
Explorational Exercises	94
Summary	95
Answers to Guided Exercises	97
Answers to Explorational Exercises	99
032.3 HTML References and Embedded Resources	106
032.3 Lesson 1	107
Introduction	107
Embedded Content	107
Links	111
Guided Exercises	114
Explorational Exercises	115
Summary	116
Answers to Guided Exercises	117
Answers to Explorational Exercises	118

032.4 HTML Forms	119
032.4 Lesson 1	120
Introduction	120
Simple HTML Forms	120
Input for large texts: textarea	128
Lists of Options	129
The hidden Element Type	134
The File Input Type	134
Action Buttons	135
Form Action and Methods	136
Guided Exercises	138
Explorational Exercises	139
Summary	140
Answers to Guided Exercises	141
Answers to Explorational Exercises	142
TOPIC 033: CSS CONTENT STYLING	144
033.1 CSS Basics	145
033.1 Lesson 1	146
Introduction	146
Applying Styles	147
Guided Exercises	154
Explorational Exercises	155
Summary	156
Answers to Guided Exercises	157
Answers to Explorational Exercises	158
033.2 CSS Selectors and Style Application	159
033.2 Lesson 1	160
Introduction	160
Page-Wide Styles	160
Restrictive Selectors	162
Special Selectors	168
Guided Exercises	169
Explorational Exercises	170
Summary	171
Answers to Guided Exercises	172
Answers to Explorational Exercises	173
033.3 CSS Styling	174
033.3 Lesson 1	175
Introduction	175
CSS Common Properties and Values	175

Colors	175
Background	178
Borders	180
Unit Values	180
Relative Units	181
Fonts and Text Properties	182
Guided Exercises	185
Explorational Exercises	186
Summary	187
Answers to Guided Exercises	188
Answers to Explorational Exercises	189
033.4 CSS Box Model and Layout	190
033.4 Lesson 1	191
Introduction	191
Normal Flow	191
Customizing Normal Flow	198
Responsive Design	203
Guided Exercises	204
Explorational Exercises	205
Summary	206
Answers to Guided Exercises	207
Answers to Explorational Exercises	208
TOPIC 034: JAVASCRIPT PROGRAMMING	209
034.1 JavaScript Execution and Syntax	210
034.1 Lesson 1	211
Introduction	211
Running JavaScript in the Browser	211
Browser Console	214
JavaScript Statements	215
JavaScript Commenting	216
Guided Exercises	218
Explorational Exercises	219
Summary	220
Answers to Guided Exercises	221
Answers to Explorational Exercises	222
034.2 JavaScript Data Structures	223
034.2 Lesson 1	224
Introduction	224
High-Level Languages	224
Declaration of Constants and Variables	225

Types of Values	227
Operators	231
Guided Exercises	235
Explorational Exercises	236
Summary	237
Answers to Guided Exercises	238
Answers to Explorational Exercises	239
034.3 JavaScript Control Structures and Functions	240
034.3 Lesson 1	241
Introduction	241
If Statements	241
Switch Structures	246
Loops	249
Guided Exercises	254
Explorational Exercises	255
Summary	256
Answers to Guided Exercises	257
Answers to Explorational Exercises	258
034.3 Lesson 2	260
Introduction	260
Defining a Function	260
Function Recursion	265
Guided Exercises	269
Explorational Exercises	270
Summary	271
Answers to Guided Exercises	272
Answers to Explorational Exercises	273
034.4 JavaScript Manipulation of Website Content and Styling	274
034.4 Lesson 1	275
Introduction	275
Interacting with the DOM	275
HTML Content	276
Selecting Specific Elements	278
Working with Attributes	279
Working with Classes	283
Event Handlers	285
Guided Exercises	288
Explorational Exercises	289
Summary	290
Answers to Guided Exercises	291

Answers to Explorational Exercises	292
TOPIC 035: NODEJS SERVER PROGRAMMING	293
035.1 NodeJS Basics	294
035.1 Lesson 1	295
Introduction	295
Getting Started	296
Guided Exercises	303
Explorational Exercises	304
Summary	305
Answers to Guided Exercises	306
Answers to Explorational Exercises	307
035.2 NodeJS Express Basics	308
035.2 Lesson 1	310
Introduction	310
Initial Server Script	310
Routes	313
Adjustments to the Response	317
Cookie Security	320
Guided Exercises	321
Explorational Exercises	322
Summary	323
Answers to Guided Exercises	324
Answers to Explorational Exercises	325
035.2 Lesson 2	326
Introduction	326
Static Files	327
Formatted Output	327
Templates	332
HTML Templates	334
Guided Exercises	338
Explorational Exercises	339
Summary	340
Answers to Guided Exercises	341
Answers to Explorational Exercises	342
035.3 SQL Basics	343
035.3 Lesson 1	344
Introduction	344
SQL	344
SQLite	345
Opening the Database	346

Structure of a Table	346
Data Entry	347
Queries	348
Changing the Contents of the Database	349
Closing the Database	351
Guided Exercises	353
Explorational Exercises	354
Summary	355
Answers to Guided Exercises	356
Answers to Explorational Exercises	357
Imprint	358



Topic 031: Software Development and Web Technologies



031.1 Software Development Basic

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 031.1](#)

Weight

1

Key knowledge areas

- Understand what source code is
- Understand the principles of compilers and interpreters
- Understand the concept of libraries
- Understand the concepts of functional, procedural and object-oriented programming
- Awareness of common features of source code editors and integrated development environments (IDE)
- Awareness of version control systems
- Awareness of software testing
- Awareness of important programming languages (C, C++, C#, Java, JavaScript, Python, PHP)



031.1 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	031 Software Development and Web Technologies
Objective:	031.1 Software Development Basics
Lesson:	1 of 1

Introduction

The very first computers were programmed through the grueling process of plugging cables into sockets. Computer scientists soon started a never-ending search for easy ways to tell the computer what to do. This chapter introduces the tools of programming. It discusses the key ways that text instructions—programming languages—represent the tasks a programmer wants to accomplish, and the tools that change the program into a form called *machine language* that a computer can run.

NOTE In this text, the terms *program* and *application* are used interchangeably.

Source Code

A programmer normally develops an application by writing a textual description, called *source code*, of the desired task. The source code is in a carefully defined *programming language* that represents what the computer can do in a high-level abstraction humans can understand. Tools have also been developed to let programmers as well as non-programmers express their thoughts visually, but writing source code is still the predominant way to program.

In the same way that a natural language has nouns, verbs, and constructions to express ideas in a structured way, the words and punctuation in a programming language are symbolic representations of operations that will be performed on the machine.

In this sense, source code is not very different from any other text in which the author employs the well-established rules of a natural language to communicate with the reader. In the case of source code, the “reader” is the machine, so the text cannot contain ambiguities or inconsistencies—even subtle ones.

And like any text that discusses some topic in depth, the source code also needs to be well structured and logically organized when developing complex applications. Very simple programs and didactic examples can be stored in the few lines of a single text file, which contains all the program’s source code. More complex programs can be subdivided into thousands of files, each with thousands of lines.

The source code of professional applications should be organized into different folders, usually associated with a particular purpose. A chat program, for example, can be organized into two folders: one that contains the code files that handle the transmission and reception of messages over the network, and another folder that contains the files that build the interface and react to user actions. Indeed, it is common to have many folders and subfolders with source code files dedicated to very specific tasks within the application.

Moreover, the source code is not always isolated in its own files, with everything written in a single language. In web applications, for example, an HTML document can embed JavaScript code to supplement the document with extra functionality.

Code Editors and IDE

The variety of ways in which source code can be written can be intimidating. Therefore, many developers take advantage of tools that help with writing and testing the program.

The source code file is just a plain text file. As such, it can be edited by any text editor, no matter how simple. To make it easier to distinguish between source code and plain text, each language adopts a self-explanatory filename extension: `.c` for the C language, `.py` for Python, `.js` for JavaScript, etc. General-purpose editors often understand the source code of popular languages well enough to add italics, colors, and indentation to make the code understandable.

Not every developer chooses to edit source code in a general-purpose editor. An *integrated development environment* (IDE) provides a text editor along with tools to help the programmer avoid syntactic errors and obvious inconsistencies. These editors are particularly recommended for less experienced programmers, but experienced programmers use them as well.

Popular IDEs such as Visual Studio, Eclipse, and Xcode intelligently watch what the programmer types, frequently suggesting words to use (autocompletion) and verifying code in real-time. The IDEs can even offer automated debugging and testing to identify issues whenever the source code changes.

Some more experienced programmers opt for less intuitive editors such as Vim, which offer greater flexibility and do not require installation of additional packages. These programmers use external, standalone tools to add the features that are built-in when you use an IDE.

Code Maintenance

Whether in an IDE or using standalone tools, it's important to employ some kind of *version control system* (VCS). Source code is constantly evolving because unforeseen flaws need to be fixed and enhancements need to be incorporated. An inevitable consequence of this evolution is that fixes and enhancements can interfere with other parts of applications in a large code base. Version control tools such as Git, Subversion, and Mercurial record all changes made to the code and who made the change, allowing you to trace and eventually recover from an unsuccessful modification.

Furthermore, version control tools allow each developer on the team to work on a copy of the source code files without interfering with the work of other programmers. Once the new versions of source code are ready and tested, corrections or improvements made to one copy can be incorporated by other team members.

Git, the most popular version control system nowadays, allows many independent copies of a repository to be maintained by different people, who share their changes as they desire. However, whether using a decentralized or centralized version control system, most teams maintain one trusted repository whose source code and resources can be relied on. Several online services offer storage for repositories of source code. The most popular of these services are GitHub and GitLab, but the GNU project's Savannah is also worth mentioning.

Programming Languages

A wide variety of programming languages exist; each decade sees the invention of new ones. Each programming language has its own rules and is recommended for particular purposes. Although the languages show superficial differences in syntax and keywords, what really distinguishes the languages are the deep conceptual approaches they represent, known as *paradigms*.

Paradigms

Paradigms define the premises on which a programming language is based, especially concerning how the source code should be structured.

The developer starts from the language paradigm to formulate the tasks to be performed by the

machine. These tasks, in turn, are symbolically expressed with the words and syntactic constructions offered by the language.

The programming language is *procedural* when the instructions presented in the source code are executed in sequential order, like a movie script. If the source code is segmented into functions or subroutines, a main routine takes care of calling the functions in sequence.

The following code is an example of a procedural language. Written in C, it defines variables to represent the side, area and volume of geographical shapes. The value of the `side` variable is assigned in `main()`, which is the function invoked when the program is executed. The `area` and `volume` variables are calculated in the `square()` and `cube()` subroutines that precede the main function:

```
#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;
}
```

The order of actions defined in `main()` determines the sequence of program states, characterized by the value of the `side`, `area`, and `volume` variables. The example ends after displaying the value of `volume` with the `printf` statement.

On the other hand, the paradigm of *object-oriented programming* (OOP) has as its main characteristic the separation of the program state into independent sub-states. These sub-states and associated operations are the *objects*, so called because they have a more or less independent existence within the program and because they have specific purposes.

Distinct paradigms do not necessarily restrict the type of task that can be performed by a program. The code from the previous example can be rewritten according to the OOP paradigm using the C++

language:

```
#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;
}
```

The `main()` function is still present. But now there is a new word, `class`, that introduces the definition of an object. The defined class, named `Cube`, contains its own variables and subroutines. In OOP, a variable is also called an *attribute* and a subroutine is called a *method*.

It's beyond the scope of this chapter to explain all the C++ code in the example. What's important to us here is that `Cube` contains the `side` attribute and two methods. The `volume()` method calculates the cube's volume.

It is possible to create several independent objects from the same class, and classes can be composed of other classes.

Keep in mind that these same features can be written differently and that the examples in this chapter are oversimplified. C and C++ have much more sophisticated features that allow much more complex and practical constructions.

Most programming languages do not rigorously impose one paradigm, but allow programmers to choose various aspects of one paradigm or another. JavaScript, for example, incorporates aspects of different paradigms. The programmer can decompose the entire program into functions that do not share a common state with each other:

```
function cube(side){  
  return side*side*side;  
}  
  
console.log("Volume: " + cube(2));
```

Although this example is similar to procedural programming, note that the function receives a copy of all the information necessary for its execution and always produces the same result for the same parameter, regardless of changes that happen outside the function's scope. This paradigm, called *functional*, is strongly influenced by mathematical formalism, where every operation is self-sufficient.

Another paradigm covers *declarative* languages, which describe the states you want the system to be in. A declarative language can figure out how to achieve the specified states. SQL, the universal language for querying databases, is sometimes called a declarative language, although it really occupies a unique niche in the programming pantheon.

There is no universal paradigm that can be adopted to any context. The choice of language may also be restricted by which languages are supported on the platform or execution environment where the program will be used.

A web application that will be used by the browser, for example, will need to be written in JavaScript, which is a language universally supported by browsers. (A few other languages can be used because they provide converters to create JavaScript.) So for the web browser—sometimes called the *client side* or *front end* of the web application—the developer will have to use the paradigms allowed in JavaScript. The server side or back end of the application, which handles requests from the browser, is normally programmed in a different language; PHP is most popular for this purpose.

Regardless of paradigm, every language has pre-built *libraries* of functions that can be incorporated into code. Mathematical functions—like the ones illustrated in the example code—don't need to be implemented from scratch, as the language already has the function ready to use. JavaScript, for example, provides the `Math` object with the most common math operations.

Even more specialized functions are usually available from the language vendor or third-party developers. These extra resource libraries can be in source code form; i.e., in extra files that are incorporated into the file where they will be used. In JavaScript, embedding is done with `import from`:

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

This type of import, where the embedded resource is also a source code file, is most often used in so-called *interpreted languages*. *Compiled languages* allow, among other things, the incorporation of pre-compiled features in machine language, that is, *compiled libraries*. The next section explains the differences between these types of languages.

Compilers and Interpreters

As we already know, source code is a symbolic representation of a program that needs to be translated into machine language in order to run.

Roughly speaking, there are two possible ways to do the translation: converting the source code beforehand for future execution, or converting the code at the moment of its execution. Languages of the first modality are called *compiled languages* and languages of the second modality are called *interpreted languages*. Some interpreted languages provide compilation as an option, so that the program can start faster.

In compiled languages, there is a clear distinction between the source code of the program and the program itself, which will be executed by the computer. Once compiled, the program will usually work only on the operating system and platform for which it was compiled.

In an interpreted language, the source code itself is treated as the program, and the process of converting to machine language is transparent to the programmer. For an interpreted language, it is common to call the source code a *script*. The interpreter translates the script into the machine language for the system it's running on.

Compilation and Compilers

The C programming language is one of the best-known examples of a compiled language. The C language's greatest strengths are its flexibility and performance. Both high-performance supercomputers and microcontrollers in home appliances can be programmed in the C language. Other examples of popular compiled languages are C++ and C# (C sharp). As their names suggest, these languages are inspired by C, but include features that support the object-oriented paradigm.

The same program written in C or C++ can be compiled for different platforms, requiring little or no change to the source code. It is the compiler that defines the target platform of the program. There are platform-specific compilers as well as cross-platform compilers such as GCC (which stands for *GNU Compiler Collection*) that can produce binary programs for many distinct architectures.

NOTE

There are also tools that automate the compilation process. Instead of invoking the compiler directly, the programmer creates a file indicating the different compilation steps to be performed automatically. The traditional tool used for this purpose is `make`, but a number of newer tools such as `Maven` and `Gradle` are also in widespread use. The entire build process is automated when you use an IDE.

The compilation process does not always generate a binary program in machine language. There are compiled languages that produce a program in a format generically called *bytecode*. Like a script, bytecode is not in a platform-specific language, so it requires an interpreter program that translates it into machine language. In this case, the interpreter program is simply called a *runtime*.

The Java language takes this approach, so compiled programs written in Java can be used on different operating systems. Despite its name, Java is unrelated to JavaScript.

Bytecode is closer to machine language than source code, so its execution tends to be comparatively faster. Because there is still a conversion process during the execution of the bytecode, it is difficult to obtain the same performance as an equivalent program compiled into machine language.

Interpretation and Interpreters

In interpreted languages such as JavaScript, Python, and PHP, the program does not need to be precompiled, making it easier to develop and modify it. Instead of compiling it, the script is executed by another program called an interpreter. Usually, the interpreter of a language is named after the language itself. The interpreter of a Python script, for example, is a program called `python`. The JavaScript interpreter is most often the web browser, but scripts can also be executed by the `node` program outside a browser. Because it is converted to binary instructions every time it is executed, an interpreted language program tends to be slower than a compiled language equivalent.

Nothing prevents the same application from having components written in different languages. If necessary, these components can communicate through a mutually understandable *application programming interface* (API).

The Python language, for example, has very sophisticated data mining and data tabulation capabilities. The developer can choose Python to write the parts of the program that deal with these aspects and another language, such as C++, to perform the heavier numeric processing. It is possible to adopt this strategy even when there is no API that allows direct communication between the two components. Code written in Python can generate a file in the proper format to be used by a program written in C++, for example.

Although it is possible to write almost any program in any language, the developer should adopt the one that is most in line with the application's purpose. In doing so, you benefit from the reuse of

already tested and well-documented components.

Guided Exercises

1. What kind of program can be used to edit source code?

2. What kind of tool helps to integrate the work of different developers into the same code base?

Explorational Exercises

1. Suppose you want to write a 3D game to be played in the browser. Web apps and games are programmed in JavaScript. Although it is possible to write all the graphic functions from scratch, it is more productive to use a ready-made library for this purpose. Which third-party libraries provide capabilities for 3D animation in JavaScript?

2. Besides PHP, what other languages can be used on the server side of a web application?

Summary

This lesson covers the most essential concepts of software development. The developer must be aware of important programming languages and the proper usage scenario for each. This lesson goes through the following concepts and procedures:

- What source code is.
- Source code editors and related tools.
- Procedural, object-oriented, functional, and declarative programming paradigms.
- Characteristics of compiled and interpreted languages.

Answers to Guided Exercises

1. What kind of program can be used to edit source code?

In principle, any program capable of editing plain text.

2. What kind of tool helps to integrate the work of different developers into the same code base?

A source or version control system, such as Git.

Answers to Explorational Exercises

1. Suppose you want to write a 3D game to be played in the browser. Web apps and games are programmed in JavaScript. Although it is possible to write all the graphic functions from scratch, it is more productive to use a ready-made library for this purpose. Which third-party libraries provide capabilities for 3D animation in JavaScript?

There are many options for 3D graphics libraries for JavaScript, such as threejs and BabylonJS.

2. Besides PHP, what other languages can be used on the server side of a web application?

Any language supported by the HTTP server application used on the server host. Some examples are Python, Ruby, Perl, and JavaScript itself.



031.2 Web Application Architecture

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 031.2](#)

Weight

2

Key knowledge areas

- Understand the principle of client and server computing
- Understand the role of web browsers and be aware of commonly used web browsers
- Understand the role of web servers and application servers
- Understand common web development technologies and standards
- Understand the principles of APIs
- Understand the principle of relational and non-relational (NoSQL) databases
- Awareness of commonly used open source database management systems
- Awareness of REST and GraphQL
- Awareness of single-page applications
- Awareness of web application packaging
- Awareness of WebAssembly
- Awareness of content management systems

Partial list of the used files, terms and utilities

- Chrome, Edge, Firefox, Safari, Internet Explorer

- HTML, CSS, JavaScript
- SQLite, MySQL, MariaDB, PostgreSQL
- MongoDB, CouchDB, Redis



031.2 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	031 Software Development and Web Technologies
Objective:	031.2 Web Application Architecture
Lesson:	1 of 1

Introduction

The word *application* has a broad meaning in technological jargon. When the application is a traditional program, executed locally and self-sufficient in its purpose, both the application's operating interface and the data processing components are integrated in a single "package". A *web application* is different because it adopts the client/server model and its client portion is based on HTML, which is obtained from the server and, in general, rendered by a browser.

Clients and Servers

In the client/server model, part of the work is done locally on the *client side* and part of the work is done remotely, on the *server side*. Which tasks are performed by each party varies according to the purpose of the application, but in general it's up to the client to provide an interface to the user and to layout the content in an attractive manner. It's up to the server to run the business end of the

application, processing and responding to requests made by the client. In a shopping application, for example, the client application displays an interface for the user to choose and pay for the products, but the data source and the transaction records are kept on the remote server, accessed via the network. Web applications perform this communication over the Internet, usually via the Hypertext Transfer Protocol (HTTP).

Once loaded by the browser, the client side of the application initiates interaction with the server whenever necessary or convenient. Web application servers offer an *application programming interface* (API) that defines the available requests and how those requests should be made. Thus, the client constructs a request in the format defined by the API and sends it to the server, which checks for any prerequisites for the request and sends back the appropriate response.

While the client, in the form of a mobile application or desktop browser, is a self-contained program with regard to the user interface and instructions for communicating with the server, the browser must obtain the HTML page and associated components—such as images, CSS, and JavaScript—that define the interface and instructions for communicating with the server.

The programming languages and platforms used by client and server are independent, but use a mutually understandable communication protocol. The server portion is almost always performed by a program without a graphical interface, running in highly available computing environments so that it is always ready to respond to requests. In contrast, the client portion runs on any device that is capable of rendering an HTML interface, such as smartphones.

In addition to being essential for certain purposes, the adoption of the client/server model allows an application to optimize several aspects of development and maintenance, since each part can be designed for its specific purpose. An application that displays maps and routes, for example, does not need to have all maps stored locally. Only maps relating to the location of the user's interest are required, so only those maps are requested from the central server.

The developers have direct control over the server, so they can also modify the client that is provided by it. This allows developers to improve the application, to a greater or lesser extent, without the need for the user to explicitly install new versions.

The Client Side

A web application should run the same way on any of the most popular browsers, as long as the browser is up to date. Some browsers may be incompatible with recent innovations, but only experimental applications use features not yet widely adopted.

Incompatibility issues were more common in the past, when different browsers had their own *rendering engine* and there was less cooperation in formulating and adopting standards. The

rendering engine is the main component of the browser, as it is responsible for transforming HTML and other associated components into the visual and interactive elements of the interface. Some browsers, notably Internet Explorer, needed special treatment in the code so as not to break the pages expected functioning.

Today, there are minimal differences between the main browsers, and incompatibilities are rare. In fact, the Chrome and Edge browsers use the same rendering engine (called Blink). The Safari browser, and other browsers offered on the iOS App Store, use the WebKit engine. Firefox uses an engine called Gecko. These three engines account for virtually all browsers used today. Although developed separately, the three engines are open source projects and there is cooperation between their developers, which facilitates compatibility, upkeep, and the adoption of standards.

Because browser developers have taken so much effort to stay compatible, the server is not normally linked to a single type of client. In principle, an HTTP server can communicate with any client that is also capable of communicating via HTTP. In a map application, for example, the client can be a mobile application or a browser that loads the HTML interface from the server.

Varieties of Web Clients

There are mobile and desktop applications whose interface is rendered from HTML and, like browsers, can use JavaScript as a programming language. However, unlike the client loaded in the browser, the HTML and the necessary components for the native client to function are locally present since the installation of the application. In fact, an application that works this way is virtually identical to an HTML page (both are even likely to be rendered by the same engine). There are also *progressive web apps* (PWA), a mechanism that allows you to package web application clients for offline use—limited to functions that do not require immediate communication with the server. Regarding what the application can do, there is no difference between running the browser or packaged in a PWA, except that in the latter the developer has more control over what is stored locally.

Rendering interfaces from HTML is such a recurring activity that the engine is usually a separate software component, present in the operating system. Its presence as a separate component allows different applications to incorporate it without having to embed it in the application package. This model also delegates the maintenance of the rendering engine to the operating system, facilitating updates. It is very important to keep such a crucial component up to date in order to avoid possible failures.

Regardless of their delivery method, applications written in HTML run on an abstraction layer created by the engine, which functions as an isolated execution environment. In particular, in the case of a client that runs on the browser, the application has at its disposal only those resources offered by the browser. Basic features, such as interacting with page elements and requesting files

over HTTP, are always available. Resources that may contain sensitive information, such as access to local files, geographic location, camera, and microphone, require explicit user authorization before the application is able to use them.

Languages of a Web Client

The central element of a web application client that runs on the server is the HTML document. In addition to presenting the interface elements that the browser displays in a structured way, the HTML document contains the addresses for all files required for the correct presentation and operation of the client.

HTML alone does not have much versatility to build more elaborate interfaces and does not have general-purpose programming features. For this reason, an HTML document that should function as a client application is always accompanied by one or more sets of CSS and JavaScript.

The CSS can be provided as a separate file or directly in the HTML file itself. The main purpose of CSS is to adjust the appearance and layout of the elements of the HTML interface. Although not strictly necessary, more sophisticated interfaces usually require modifications to the CSS properties of the elements to suit their needs.

JavaScript is a practically indispensable component. Procedures written in JavaScript respond to events in the browser. These events can be caused by the user or non-interactive. Without JavaScript, an HTML document is practically limited to text and images. Using JavaScript in HTML documents allows you to extend interactivity beyond hyperlinks and forms, making the page displayed by the browser like a conventional application interface.

JavaScript is a general-purpose programming language, but its main use is in web applications. The features of the browser execution environment are accessible through JavaScript keywords, used in a script to perform the desired operation. The term `document`, for example, is used in JavaScript code to refer to the HTML document associated with the JavaScript code. In the context of the JavaScript language, `document` is a *global object* with properties and methods that can be used to obtain information from any element on the HTML document. More importantly, you can use the `document` object to modify its elements and to associate them with custom actions written in JavaScript.

A client application based on web technologies is multiplatform, because it can run on any device that has a compatible web browser.

Being confined to the browser, however, imposes limitations on web applications compared to native applications. The intermediation performed by the browser allows higher level programming and increases security, but also increases the processing and memory consumption.

Developers are continually working on browsers to provide more features and improve the

performance of JavaScript applications, but there are intrinsic aspects to the execution of scripts such as JavaScript that impose a disadvantage on them compared to native programs for the same hardware.

A feature that significantly improves the performance of JavaScript applications running on the browser is *WebAssembly*. WebAssembly is a kind of compiled JavaScript that produces source code written in a more efficient, lower-level language, such as the C language. WebAssembly can accelerate mainly processor-intensive activities, because it avoids much of the translation performed by the browser when running a program written in conventional JavaScript.

Regardless of the implementation details of the application, all HTML code, CSS, JavaScript, and multimedia files must first be obtained from the server. The browser obtains these files just like an Internet page, that is, with an address accessed by the browser.

A web page that acts as an interface to a web application is like a plain HTML document, but adds additional behaviors. On conventional pages, the user is directed to another page after clicking on a link. Web applications can present their interface and respond to user events without loading new pages in the browser's window. The modification of this standard behavior in HTML pages is done via JavaScript programming.

A webmail client, for example, displays messages and switches between message folders without leaving the page. This is possible because the client uses JavaScript to react to user actions and make appropriate requests to the server. If the user clicks on the subject of a message in the inbox, a JavaScript code associated with this event requests the content of that message from the server (using the corresponding API call). As soon as the client receives the response, the browser displays the message in the appropriate portion of the same page. Different webmail clients may adopt different strategies, but they all use this same principle.

Therefore, in addition to providing the files that make up the client to the browser, the server must also be able to handle requests such as that of the webmail client, when it asks for the content of a specific message. Every request that the client can make has a predefined procedure to respond on the server, whose API can define different methods to identify which procedure the request refers to. The most common methods are:

- Addresses, through a Uniform Resource Locator (URL)
- Fields in the HTTP header
- GET/POST methods
- WebSockets

One method may be more suitable than another, depending on the purpose of the request and other

criteria taken into account by the developer. In general, web applications use a combination of methods, each in a specific circumstance.

The *Representational State Transfer* (REST) paradigm is widely used for communication in web applications, because it is based on the basic methods available in HTTP. The header of an HTTP request starts with a keyword that defines the basic operation to be performed: GET, POST, PUT, DELETE, etc., accompanied by a corresponding URL where the action will be applied. If the application requires more specific operations, with a more detailed description of the requested operation, the GraphQL protocol may be a more appropriate choice.

Applications developed using the client/server model are subject to instabilities in communication. For this reason, the client application must always adopt efficient data transfer strategies to favor its consistency and not harm the user experience.

The Server Side

Despite being the main actor in a web application, the server is the passive side of the communication, just responding to requests made by the client. In web jargon, *server* can refer to the machine that receives the requests, the program that specifically handles HTTP requests, or the recipient script that produces a response to the request. This latter definition is the most relevant in the context of web application architecture, but they are all closely related. Although they are only partially in the scope of the application server developer, the machine, operating system, and HTTP server cannot be ignored, because they are fundamental to running the application server and often intersect.

Handling Paths from Requests

HTTP servers, such as Apache and NGINX, usually require specific configuration changes to meet the needs of the application. By default, traditional HTTP servers directly associate the path indicated in the request to a file on the local file system. If a website's HTTP server keeps its HTML files in the `/srv/www` directory, for example, a request with the path `/en/about.html` will receive the content of the file `/srv/www/en/about.html` as a response, if the file exists. More sophisticated websites, and especially web applications, demand customized treatments for different types of requests. In this scenario, part of the application implementation is modifying HTTP server settings to meet application requirements.

Alternatively, there are frameworks that allow you to integrate the management of HTTP requests and the implementation of the application code in one place, allowing the developer to focus more on the application's purpose than on platform details. In Node.js Express, for example, all request mapping and corresponding programming are implemented using JavaScript. As the programming of clients is usually done in JavaScript, many developers consider it a good idea from the perspective

of code maintenance to use the same language for client and server. Other languages commonly used to implement the server side, either in frameworks or in traditional HTTP servers, are PHP, Python, Ruby, Java, and C#.

Database Management Systems

It is up to the discretion of the development team how the data received or requested by the client is stored on the server, but there are general guidelines that apply to most cases. It is convenient to keep static content—images, JavaScript and CSS code that do not change in the short term—as conventional files, either on the server’s own file system or distributed across a *content delivery network* (CDN). Other kinds of content, such as email messages in a webmail application, product details in a shopping application, and transaction logs, are more conveniently stored in a *database management system* (DBMS).

The most traditional type of database management system is the *relational database*. In it, the application designer defines data tables and the input format accepted by each table. The set of tables in the database contains all the dynamic data consumed and produced by the application. A shopping app, for example, may have a table that contains an entry with the details of each product in the store and a table that records items purchased by a user. The table of purchased items contains references to entries in the product table, creating relationships between the tables. This approach can optimize storage and access to the data, as well as allow queries in combined tables using the language adopted by the database management system. The most popular relational database language is the *Structured Query Language* (SQL, pronounced “sequel”), adopted by the open source databases SQLite, MySQL, MariaDB, and PostgreSQL.

An alternative to relational databases is a form of database that does not require a rigid structure for the data. These databases are called *non-relational databases* or simply *NoSQL*. Although they may incorporate some features similar to those found in relational databases, the focus is on allowing greater flexibility in storage and access to stored data, passing the task of processing that data to the application itself. MongoDB, CouchDB, and Redis are common non-relational database management systems.

Content Maintenance

Regardless of the database model adopted, applications have to add data and probably update it over the life span of the applications. In some applications, such as webmail, users themselves provide data to the database when using the client to send and receive messages. In other cases, such as in the shopping application, it’s important to allow the application’s maintainers to modify the database without having to resort to programming. Many organizations therefore adopt some kind of *content management system* (CMS), which allows non-technical users to administer the application. Therefore, for most web applications, it’s necessary to implement at least two types of

clients: a non-privileged client, used by ordinary users, and privileged clients, used by special users to maintain and update the information presented by the application.

Guided Exercises

1. What programming language is used together with HTML to create web applications clients?

2. How does the retrieval of a web application differ from that of a native application?

3. How does a web application differ from a native application in access to the local hardware?

4. Cite one characteristic of a web application client that makes it distinct from an ordinary web page.

Explorational Exercises

1. What feature do modern browsers offer to overcome the poor performance of CPU-intensive web application clients?

2. If a web application uses the REST paradigm for client/server communication, which HTTP method should be used when the client requests the server to erase a specific resource?

3. Cite five server scripting languages supported by the Apache HTTP server.

4. Why are non-relational databases considered easier to maintain and upgrade than relational databases?

Summary

This lesson covers the concepts and standards in web development technology and architecture. The principle is simple: the web browser runs the client application, which communicates with the core application running in the server. Albeit simple in principle, web applications must combine many technologies to adopt the principle of client and server computing over the web. The lesson goes through the following concepts:

- The role of web browsers and web servers.
- Common web development technologies and standards.
- How web clients can communicate with the server.
- Types of web servers and server databases.

Answers to Guided Exercises

1. What programming language is used together with HTML to create web applications clients?

JavaScript

2. How does the retrieval of a web application differ from that of a native application?

A web application is not installed. Instead, parts of it run on the server and the client interface runs in an ordinary web browser.

3. How does a web application differ from a native application in access to the local hardware?

All access to the local resources, such as storage, cameras, or microphones, are mediated by the browser and require explicit user authorization to work.

4. Cite one characteristic of a web application client that makes it distinct from an ordinary web page.

The interaction with traditional web pages is basically restricted to hyperlinks and sending forms, while web application clients are closer to a conventional application interface.

Answers to Explorational Exercises

1. What feature do modern browsers offer to overcome the poor performance of CPU-intensive web application clients?

The developers can use WebAssembly to implement the CPU-intensive parts of the client application. WebAssembly code generally has better performance than traditional JavaScript, because it requires less translation of instructions.

2. If a web application uses the REST paradigm for client/server communication, which HTTP method should be used when the client requests the server to erase a specific resource?

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

3. Cite five server scripting languages supported by the Apache HTTP server.

PHP, Go, Perl, Python, and Ruby.

4. Why are non-relational databases considered easier to maintain and upgrade than relational databases?

Unlike relational databases, non-relational databases do not require data to fit rigid predefined structures, making it easier to implement changes in the data structures without affecting existing data.



031.3 HTTP Basics

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 031.3](#)

Weight

3

Key knowledge areas

- Understand HTTP GET and POST methods, status codes, headers and content types
- Understand the difference between static and dynamic content
- Understand HTTP URLs
- Understand how HTTP URLs are mapped to file system paths
- Upload files to a web server's document root
- Understand caching
- Understand cookies
- Awareness of sessions and session hijacking
- Awareness of commonly used HTTP servers
- Awareness of HTTPS and TLS
- Awareness of web sockets
- Awareness of virtual hosts
- Awareness of common HTTP servers
- Awareness of network bandwidth and latency requirements and limitations

Partial list of the used files, terms and utilities

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



031.3 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	031 Software Development and Web Technologies
Objective:	031.3 HTTP Basics
Lesson:	1 of 1

Introduction

The *HyperText Transfer Protocol* (HTTP) defines how a client asks the server for a specific resource. Its working principle is quite simple: the client creates a request message identifying the resource it needs and forwards that message to the server via the network. In turn, the HTTP server evaluates where to extract the requested resource and sends a response message back to the client. The reply message contains details about the requested resource, followed by the resource itself.

More specifically, HTTP is the set of rules that define how the client application should format *request* messages that will be sent to the server. The server then follows HTTP rules to interpret the request and format *reply* messages. In addition to requesting or transferring requested content, HTTP messages contain extra information about the client and server involved, about the content itself, and even about its unavailability. If a resource cannot be sent, a code in the response explains the reason for the unavailability and, if possible, indicates where the resource was moved.

The part of the message that defines the resource details and other context information is called the *header* of the message. The part following the header, which contains the content of the corresponding resource, is called the *payload* of the message. Both request messages and response

messages can have a payload, but in most cases, only the response message has one.

The Client's Request

The first stage of an HTTP data exchange between the client and the server is initiated by the client, when it writes a request message to the server. Take, for example, a common browser task: to load an HTML page from a server hosting a website, such as `https://learning.lpi.org/en/`. The address, or URL, provides several pieces of relevant information. Three pieces of information appear in this particular example:

- The protocol: HyperText Transfer Protocol Secure (`https`), an encrypted version of HTTP.
- The web host's network name (`learning.lpi.org`)
- The location of the requested resource on the server (the `/en/` directory—in this case, the English version of the home page).

NOTE

A *Uniform Resource Locator* (URL) is an address that points to a resource on the Internet. This resource is usually a file that can be copied from a remote server, but URLs can also indicate dynamically generated content and data streams.

How the Client Handles the URL

Before contacting the server, the client needs to convert `learning.lpi.org` to its corresponding IP address. The client uses another Internet service, the *Domain Name System* (DNS), to request the IP address of a host name from one or more predefined DNS servers (DNS servers are usually automatically defined by the Internet Service Provider, ISP).

With the server's IP address, the client tries to connect to the HTTP or HTTPS port. Network ports are identification numbers defined by the *Transmission Control Protocol* (TCP) to intertwine and identify distinct communication channels within a client/server connection. By default, HTTP servers receive requests on TCP ports 80 (HTTP) and 443 (HTTPS).

NOTE

There are other protocols used by web applications to implement client/server communication. For audio and video calls, for example, it is more appropriate to use WebSockets, a lower level protocol that is more efficient than HTTP for transferring data streams in both directions.

The format of the request message that the client sends to the server is the same in HTTP and HTTPS. HTTPS is already more widely used than HTTP, because all data exchanges between client and server are encrypted, which is an indispensable feature to promote privacy and security on public networks. The encrypted connection is established between client and server even before any

HTTP message is exchanged, using the *Transport Layer Security* (TLS) cryptographic protocol. By doing this, all HTTPS communication is encapsulated by TLS. Once decrypted, the request or response transmitted over HTTPS is no different from a request or response made exclusively over HTTP.

The third element of our URL, `/en/`, will be interpreted by the server as the location or path for the resource being requested. If the path is not provided in the URL, the default location `/` will be used. The simplest implementation of an HTTP server associates paths in URLs with files on the file system where the server is running, but this is just one of the many options available on more sophisticated HTTP servers.

The Request Message

HTTP operates through a connection already established between client and server, usually implemented in TCP and encrypted with TLS. In fact, once a connection meeting the requirements imposed by the server is ready, an HTTP request typed by hand in plain text could generate the response from the server. In practice, however, programmers rarely need to implement routines to compose HTTP messages, as most programming languages provide mechanisms that automate the making of the HTTP message. In the case of the example URL, `https://learning.lpi.org/en/`, the simplest possible request message would have the following content:

```
GET /en/ 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
```

The first word of the first line identifies the HTTP *method*. It defines which operation the client wants to perform on the server. The GET method informs the server that the client requests the resource that follows it: `/en/`. Both client and server may support more than one version of the HTTP protocol, so the version to be adopted in the data exchange is also provided in the first line: HTTP/1.1.

NOTE

The most recent version of the HTTP protocol is HTTP/2. Among other differences, messages written in HTTP/2 are encoded in a binary structure, whereas messages written in HTTP/1.1 are sent in plain text. This change optimizes data transmission rates, but the content of the messages is basically the same.

The header can contain more lines after the first one to contextualize and help identify the request to

the server. The `Host` header field, for example, may appear redundant, because the server's host has obviously been identified by the client in order to establish the connection and it's reasonable to assume that the server knows its own identity. Nonetheless, it's important to inform the host of the expected host name in the request header, because it is common practice to use the same HTTP server to host more than one website. (In this scenario, each specific host is called *virtual host*.) Therefore, the `Host` field is used by the HTTP server to identify which one the request refers to.

The `User-Agent` header field contains details about the client program making the request. This field can be used by the server to adapt the response to the needs of a specific client, but it is more often used to produce statistics about the clients using the server.

The `Accept` field is of more immediate value, because it informs the server about the format for the requested resource. If the client is indifferent about the resource format, the `Accept` field can specify `*/*` as the format in.

There are many other header fields that can be used in an HTTP message, but the fields shown in the example are enough to request a resource from the server.

In addition to the fields in the request header, the client can include other complementary data in the HTTP request that will be sent to the server. If this data consists only of simple text parameters, in the format `name=value`, they can be added to the path of the GET method. The parameters are embedded in the path after a question mark and are separated by ampersand (&) characters:

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

In this example, `/cgi-bin/receive.cgi` is the path to the script on the server that will process and possibly use the parameters `name` and `email`, obtained from the request path. The string that corresponds to the fields, in the format `name=LPI&email=info@lpi.org`, is called *query string* and is supplied to the `receive.cgi` script by the HTTP server that receives the request.

When the data is made up of more than short text fields, it's more appropriate to send it in the payload of the message. In this case, the HTTP POST method must be used so that the server receives and processes the message's payload, according to the specifications indicated in the request header. When the POST method is used, the request header must provide the size of the payload that will be sent next and how the body is formatted:

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

The `Content-Length` field indicates the size in bytes of the payload and the `Content-Type` field indicates its format. The `multipart/form-data` format is the one most commonly used in traditional HTML forms that use the POST method. In this format, each field inserted in the request's payload is separated by the code indicated by the `boundary` keyword. The POST method should be used only when appropriate, as it uses a slightly larger amount of data than an equivalent request made with the GET method. Because the GET method sends the parameters directly in the request's message header, the total data exchange has a lower latency, because an additional connection stage to transmit the message body will not be necessary.

The Response Header

After the HTTP server receives the request message header, the server returns a response message back to the client. An HTML file request typically has a response header like this:

```
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
```

The first line provides the version of the HTTP protocol used in the response message, which must correspond to the version used in the request header. Then, still in the first line, the status code of the response appears, indicating how the server interpreted and generated the response for the request.

The status code is a three-digit number, where the left-most digit defines the response class. There are five classes of status codes, numbered from 1 to 5, each indicating a type of action taken by the server:

1xx (Informational)

The request was received, continuing the process.

2xx (Successful)

The request was successfully received, understood, and accepted.

3xx (Redirection)

Further action needs to be taken in order to complete the request.

4xx (Client Error)

The request contains bad syntax or cannot be fulfilled.

5xx (Server Error)

The server failed to fulfill an apparently valid request.

The second and third digits are used to indicate additional details. Code 200, for example, indicates that the request could be answered without any problems. As shown in the example, a brief text description following the response code (OK) can also be provided. Some specific codes are of particular interest to ensure that the HTTP client can access the resource in adverse situations or to help to identify the reason for failure in the event of an unsuccessful request:

301 Moved Permanently

The target resource has been assigned a new permanent URL, provided by the `Location` header field in the response.

302 Found

The target resource resides temporarily under a different URL.

401 Unauthorized

The request has not been applied because it lacks valid authentication credentials for the target resource.

403 Forbidden

The `Forbidden` response indicates that, although the request is valid, the server is configured to not provide it.

404 Not Found

The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

500 Internal Server Error

The server encountered an unexpected condition that prevented it from fulfilling the request.

502 Bad Gateway

The server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.

Although they indicate that it was not possible to fulfill the request, status codes 4xx and 5xx at least indicate that the HTTP server is running and is capable of receiving requests. The 4xx codes require an action to be taken on the client, because its URL or credentials are wrong. In contrast, 5xx codes indicate something wrong on the server side. Therefore, in the context of web applications, these two classes of status codes indicate that the source of the error lies in the application itself, either client or server, not in the underlying infrastructure.

Static and Dynamic Content

HTTP servers use two basic mechanisms to fulfill the content requested by the client. The first mechanism provides *static content*: that is, the path indicated in the request message corresponds to a file on the server's local file system. The second mechanism provides *dynamic content*: that is, the HTTP server forwards the request to another program—probably a script—to build the response from different sources, such as databases and other files.

Although there are different HTTP servers, they all use the same HTTP communication protocol and adopt more or less the same conventions. An application that does not have a specific need can be implemented with any traditional server, such as Apache or NGINX. Both are capable of generating dynamic content and providing static content, but there are subtle differences in the configuration of each.

The location of static files to be served up, for example, is defined in different ways in Apache and NGINX. The convention is to keep these files in a specific directory for this purpose, having a name associated with the host, for example `/var/www/learning.lpi.org/`. In Apache, this path is defined by the configuration directive `DocumentRoot /var/www/learning.lpi.org`, in a section that defines a virtual host. In NGINX, the directive used is `root /var/www/learning.lpi.org` in a `server` section of the configuration file.

Whichever server you choose, the files at `/var/www/learning.lpi.org/` will be served via HTTP in almost the same way. Some fields in the response header and their contents may vary between the two servers, but fields like `Content-Type` must be present in the response header and must be consistent across any server.

Caching

HTTP was designed to work on any type of Internet connection, fast or slow. Furthermore, most HTTP exchanges have to traverse many network nodes due to the distributed architecture of the

Internet. As a result, it is important to adopt some content caching strategy to avoid the redundant transfer of previously downloaded content. HTTP transfers can work with two basic types of cache: *shared* and *private*.

A shared cache is used by more than a single client. For example, a large content provider might use caches on geographically distributed servers, so that clients get the data from their nearest server. Once a client has made a request and its response was stored in a shared cache, other clients making that same request in that same area will receive the cached response.

A private cache is created by the client itself for its exclusive use. It is the type of caching the web browser does for images, CSS files, JavaScript, or the HTML document itself, so they don't need to be downloaded again if requested in the near future.

NOTE

Not all HTTP requests must be cached. A request using the POST method, for example, implies a response associated exclusively with that particular request, so its response content should not be reused. By default, only responses to requests made using the GET method are cached. Furthermore, only responses with conclusive status codes such as 200 (OK), 206 (Partial Content), 301 (Moved Permanently), and 404 (Not Found) are suitable for caching.

Both the shared and private cache strategy use HTTP headers to control how the downloaded content should be cached. For the private cache, the client consults the response header and verifies whether the content in the local cache still corresponds to the current remote content. If it does, the client waives the transfer of the response payload and uses the local version.

The validity of the cached resource can be assessed in several ways. The server can provide an expiration date in the response header for the first request, so that the client discards the cached resource at the end of the term and requests it again to obtain the updated version. However, the server is not always able to determine the expiration date of a resource, so it is common to use the ETag response header field to identify the version of the resource, for example Etag: "606adcd4-46fa".

To verify that a cached resource needs updating, the client requests only its response header from the server. If the ETag field matches the one in the locally stored version, the client reuses the cached content. Otherwise, the updated content of the resource is downloaded from the server.

HTTP Sessions

In a conventional website or web application, the features that handle session control are based on HTTP headers. The server cannot assume, for example, that all requests coming from the same IP address are from the same client. The most traditional method that allows the server to associate

different requests to a single client is the use of *cookies*, an identification tag that is given to the client by the server and that is provided in the HTTP header.

Cookies allow the server to preserve information about a specific client, even if the person running the client does not identify himself or herself explicitly. With cookies, it is possible to implement sessions where logins, shopping carts, preferences, etc., are preserved in between different requests made to the same server that provided them. Cookies are also used to track user browsing, so it is important to ask for consent before sending them.

The server sets the cookie in the response header using the `Set-Cookie` field. The field value is a `name=value` pair chosen to represent some attribute associated with a specific client. The server can, for example, create an identification number for a client that requests a resource for the first time and pass it on to the client in the response header:

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

If the client allows the use of cookies, new requests to this same server have the cookie field in the header:

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

With this identification number, the server can retrieve specific definitions for the client and generate a customized response. It is also possible to use more than one `Set-Cookie` field to deliver different cookies to the same customer. In this way, more than one definition can be preserved on the client side.

Cookies raise both privacy issues and potential security holes, because there is a possibility that they can be transferred to another client, who will be identified by the server as the original client. Cookies used to preserve sessions can give access to sensitive information from the original client. Therefore, it's very important for clients to adopt local protection mechanisms to prevent their cookies from being extracted and reused without authorization.

Guided Exercises

1. What HTTP method does the following request message use?

```
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. When an HTTP server hosts many websites, how is it able to identify which one a request is for?

3. What parameter is provided by the query string of the URL <https://www.google.com/search?q=LPI>?

4. Why is the following HTTP request not suitable for caching?

```
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
```

Explorational Exercises

1. How could you use the web browser to monitor the requests and responses made by an HTML page?

2. HTTP servers that provide static content usually map the requested path to a file in the server's filesystem. What happens when the path in the request points to a directory?

3. The contents of files sent over HTTPS are protected by encryption, so they cannot be read by computers between the client and the server. Despite this, can these computers in the middle identify which resource the client has requested from the server?

Summary

This lesson covers the basics of HTTP, the main protocol used by client applications to request resources from web servers. The lesson goes through the following concepts:

- Request messages, header fields, and methods.
- Response status codes.
- How HTTP servers generate responses.
- HTTP features useful for caching and session management.

Answers to Guided Exercises

1. What HTTP method does the following request message use?

```
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
```

The POST method.

2. When an HTTP server hosts many websites, how is it able to identify which one a request is for?

The Host field in the request header provides the targeted website.

3. What parameter is provided by the query string of the URL <https://www.google.com/search?q=LPI>?

The parameter named q with a value of LPI.

4. Why is the following HTTP request not suitable for caching?

```
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
```

Because requests made with the POST method imply a write operation on the server, they should not be cached.

Answers to Explorational Exercises

1. How could you use the web browser to monitor the requests and responses made by an HTML page?

All popular browsers offer *development tools* that, among other things, can show all network transactions that have been carried out by the current page.

2. HTTP servers that provide static content usually map the requested path to a file in the server's filesystem. What happens when the path in the request points to a directory?

It depends on how the server is configured. By default, most HTTP servers look for a file named `index.html` (or another predefined name) in that same directory and send it as the response. If the file isn't there, the server issues a `404 Not Found` response.

3. The contents of files sent over HTTPS are protected by encryption, so they cannot be read by computers between the client and the server. Despite this, can these computers in the middle identify which resource the client has requested from the server?

No, because the request and response HTTP headers themselves are also encrypted by TLS.



Topic 032: HTML Document Markup



032.1 HTML Document Anatomy

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 032.1

Weight

2

Key knowledge areas

- Create a simple HTML document
- Understand the role of HTML
- Understand the HTML skeleton
- Understand the HTML syntax (tags, attributes, comments)
- Understand the HTML head
- Understand meta tags
- Understand character encoding

Partial list of the used files, terms and utilities

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`
- `<meta>`, including the `charset (UTF-8)`, `name` and `content` attributes



032.1 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	032 HTML Document Markup
Objective:	032.1 HTML Document Anatomy
Lesson:	1 of 1

Introduction

HTML (*HyperText Markup Language*) is a markup language that tells web browsers how to structure and display web pages. The current version is 5.0, which was released in 2012. The HTML syntax is defined by the *World Wide Web Consortium* (W3C).

HTML is a fundamental skill in web development, as it defines the structure and a good deal of the appearance of a website. If you want a career in web development, HTML is definitely a good starting point.

Anatomy of an HTML Document

A basic HTML page has the following structure:

```

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

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

```

Now, let's analyze it in detail.

HTML Tags

HTML uses *elements* and *tags* to describe and format content. Tags consist of angle brackets around a tag name, for example `<title>`. The tag name is not case-sensitive, although the World Wide Web Consortium (W3C) recommends using lowercase letters in current versions of HTML. These HTML tags are used to build HTML elements. The tag `<title>` is an example for an *opening tag* of an HTML element that defines the title of an HTML document. However, an element has two further components. A full `<title>` element looking like this:

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

Here, `My HTML Page` serves as the element *content*, while `</title>` serves as the *closing tag* that declares that this element is complete.

NOTE

Not all HTML elements need to be closed; in such cases, we speak of empty elements, self-closing elements, or void elements.

Here are the other HTML elements from the previous example:

`<html>`

Encloses the entire HTML document. This contains all the tags that make up the page. It also indicates that the content of this file is in HTML language. Its corresponding closing tag is `</html>`.

<head>

A container for all meta information regarding the page. The corresponding closing tag of this element is `</head>`.

<body>

A container for the page content and its structural representation. Its corresponding closing tag is `</body>`.

The `<html>`, `<head>`, `<body>` and `<title>` tags are so-called *skeleton tags*, which provide the basic structure of an HTML document. In particular, they tell the web browser that it is reading an HTML page.

NOTE

Of these HTML elements, the only one that is required for an HTML document to be validated is the `<title>` tag.

As you can see, each HTML page is a well-structured document and could even be referred to as a tree, where the `<html>` element represents the document root and the `<head>` and `<body>` elements are the first branches. The example shows that it is possible to nest elements: For example, the `<title>` element is nested inside the `<head>` element, which is in turn nested inside the `<html>` element.

To ensure that your HTML code is readable and maintainable, make sure that all HTML elements are closed properly and in order. Web browsers may still render your web site as expected, but incorrect nesting of elements and their tags is an error-prone practice.

Finally, a special mention goes to the *doctype* declaration at the very top of the example document structure. `<!DOCTYPE>` is not an HTML tag, but an instruction for the web browser that specifies the HTML version used in the document. In the basic HTML document structure shown earlier, `<!DOCTYPE html>` was used, specifying that HTML5 is used in this document.

HTML Comments

When creating an HTML page, it is good practice to insert comments into the code to improve its readability and describe the purpose of larger code blocks. A comment is inserted between the `<!--` and `-->` tags, as shown in the following example:

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

The example demonstrates that HTML comments can be placed in a single line, but may also span over multiple lines. In any case, the result is that the text between `<!--` and `-->` is ignored by the web browser and therefore not displayed in the HTML page. Based on these considerations, you can deduce that the basic HTML page shown in the previous section does not display any text, because the lines `<!-- This is the Document Header -->` and `<!-- This is the Document Body -->` are just two comments.

WARNING Comments cannot be nested.

HTML Attributes

HTML tags may include one or more *attributes* to specify details of the HTML element. A simple tag with two attributes has the following form:

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

Attributes must always be set on the opening tag.

An attribute consists of a name, which indicates the property that should be set, an equal sign, and the desired value within quotes. Both single quotes and double quotes are acceptable, but it is recommended to use of single quotes or double quotes consistently throughout a project. It is important not to mix single and double quotes for a single attribute value, as the web browser will not recognize mixed quotes as one unit.

NOTE

You can include one type of quotation marks within the other type without any problems. For example, if you need to use `'` in an attribute value, you can wrap that value within `"`. However, if you want to use the same type of quotation mark inside the value as you are using to wrap the value, you need to use `"` for `"` and `'` for `'`.

The attributes can be categorized into *core attributes* and *specific attributes* as explained in the

following sections.

Core Attributes

Core attributes are attributes that can be used on any HTML element. They include:

title

Describes the content of the element. Its value is often displayed as a tooltip that is shown when the user moves their cursor over the element.

id

Associates a unique identifier with an element. This identifier must be unique within the document, and the document will not validate when multiple elements share the same `id`.

style

Assigns graphic properties (CSS styles) to the element.

class

Specifies one or multiple classes for the element in a space-separated list of class names. These classes can be referenced in CSS stylesheets.

lang

Specifies the language of the element content using ISO-639 standard two-character language codes.

NOTE

The developer can store custom information about an element by defining a so-called `data-` attribute, which is indicated by prefixing the desired name with `data-` as in `data-additionalinfo`. You can assign this attribute a value just like any other attribute.

Specific Attributes

Other attributes are specific to each HTML element. For example, the `src` attribute of an HTML `` element specifies the URL of an image. There are many more specific attributes, which will be covered in the following lessons.

Document Header

The document header defines meta information regarding the page and is described by the `<head>` element. By default, the information within the document header is not rendered by the web

browser. While it is possible to use the `<head>` element to contain HTML elements that could be displayed on the page, doing so is not recommended.

Title

The document title is specified using the `<title>` element. The title defined between the tags appears in the web browser title bar and is the suggested name for the bookmark when you try to bookmark the page. It is also displayed in search engine results as the title of the page.

An example of this element is the following:

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

The `<title>` tag is required in all HTML documents and should appear only once in each document.

NOTE

Do not confuse the title of the document with the heading of the page, which is set in the body.

Metadata

The `<meta>` element is used to specify meta information to further describe the content of an HTML document. It is a so-called self-closing element, which means that it does not have a closing tag. Aside from the core attributes that are valid for every HTML element, the `<meta>` element also uses the following attributes:

name

Defines what metadata will be described in this element. It can be set to any custom defined value, but commonly used values are `author`, `description`, and `keywords`.

http-equiv

Provides an HTTP header for the value of the `content` attribute. A common value is `refresh`, which will be explained later. If this attribute is set, the `name` attribute should not be set.

content

Provides the value associated with the `name` or `http-equiv` attribute.

charset

Specifies the character encoding for the HTML document, for example `utf-8` to set it to Unicode Transformation Format—8-bit.

Add an Author, Description, and Keywords

Using the `<meta>` tag, you can specify additional information about the author of the HTML page and describe the page content like this:

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

Try to include a series of keywords related to the content of the page in the description. This description is often the first thing a user sees when navigating with a search engine.

If you also want to provide additional keywords related to the web page to search engines, you can add this element:

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

NOTE

In the past, spammers entered hundreds of keywords and descriptions unrelated to the actual content of the page so that it also appeared in searches unrelated to the terms people searched for. Nowadays, `<meta>` tags are relegated to a position of secondary importance and are used only to consolidate the topics covered in the web page, so that it is no longer possible to mislead the new and more sophisticated search engine algorithms.

Redirect an HTML Page and Define a Time Interval for the Document to Refresh Itself

Using the `<meta>` tag, you can automatically refresh an HTML page after a certain period (for example after 30 seconds) in this way:

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

Alternatively, you can redirect a web page to another web page after the same amount of time with the following code:

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

In this example, the user is redirected from the current page to `http://www.lpi.org` after 30 seconds. The values can be anything you like. For example, if you specify `content="0"`,

url=http://www.lpi.org", the page is redirected immediately.

Specify the Character Encoding

The `charset` attribute specifies the character encoding for the HTML document. A common example is:

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

This element specifies that the document's character encoding is `utf-8`, which is a universal character set that includes practically any character of any human language. Therefore, by using it, you will avoid problems in displaying some characters that you may have using other character sets such as ISO-8859-1 (the Latin alphabet).

Other Useful Examples

Two other useful applications of the `<meta>` tag are:

- Set cookies to keep track of a site visitor.
- Take control over the viewport (the visible area of a web page inside a web browser window), which depends on the screen size of the user device (for example, a mobile phone or a computer).

However, these two examples are beyond the scope of the exam and their study is left to the curious reader to explore elsewhere.

Guided Exercises

1. For each of the following tags, indicate the corresponding closing tag:

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

2. What is the difference between a tag and an element? Use this entry as a reference:

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

3. What are the tags between which a comment should be placed?

4. Explain what an attribute is and provide some examples for the <meta> tag.

Explorational Exercises

1. Create a simple HTML version 5 document with the title `My first HTML document` and only one paragraph in the body, containing the text `Hello World`. Use the paragraph tag `<p>` in the body.

2. Add the author (`Kevin Author`) and description (`This is my first HTML page.`) of the HTML document.

3. Add the following keywords related to the HTML document: `HTML`, `Example`, `Test`, and `Metadata`.

4. Add the `<meta charset="ISO-8859-1">` element to the document header and change the `Hello World` text to Japanese (おはようございます). What happens? How can you solve the problem?

5. After changing the paragraph text back to `Hello World`, redirect the HTML page to `https://www.google.com` after 30 seconds and add a comment explaining this in the document header.

Summary

In this lesson you learned:

- The role of HTML
- The HTML skeleton
- The HTML syntax (tags, attributes, comments)
- The HTML head
- The meta tags
- How to create a simple HTML document

The following terms were discussed in this lesson:

<!DOCTYPE html>

The declaration tag.

<html>

The container for all the tags that make up the HTML page.

<head>

The container for all head elements.

<body>

The container for all body elements.

<meta>

The tag for metadata, used to specify additional information for the HTML page (such as author, description, and character encoding).

Answers to Guided Exercises

1. For each of the following tags, indicate the corresponding closing tag:

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

2. What is the difference between a tag and an element? Use this entry as a reference:

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

An HTML element consists of a start tag, a closing tag, and everything between them. An HTML tag is used to mark the beginning or end of an element. Therefore, `<title>HTML Page Title</title>` is an HTML element, while `<title>` and `</title>` are the start and closing tag respectively.

3. What are the tags between which a comment should be placed?

A comment is inserted between the `<!--` and `-->` tags and can be put on a single line or can span multiple lines.

4. Explain what an attribute is and provide some examples for the `<meta>` tag.

An attribute is used to more precisely specify an HTML element. For example, the `<meta>` tag uses the `name` and `content` attribute pair to add the author and description of an HTML page. Instead, using the `charset` attribute you can specify the character encoding for the HTML document.

Answers to Explorational Exercises

1. Create a simple HTML version 5 document with the title My first HTML document and only one paragraph in the body containing the text Hello World. Use the paragraph tag `<p>` in the body.

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

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

2. Add the author (Kevin Author) and description (This is my first HTML page.) of the HTML document.

```
<!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. Add the following keywords related to the HTML document: HTML, Example, Test and 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. Add the `<meta charset="ISO-8859-1">` element to the document header and change the Hello World text to Japanese (世界). What happens? How can you solve this situation?

If the example is carried out as described, the Japanese text is not displayed correctly. This is because ISO-8859-1 represents character encoding for the Latin alphabet. To view the text, you need to change the character encoding, using for example UTF-8 (`<meta charset="utf-8">`).

5. After changing the paragraph text back to Hello World, redirect the HTML page to <https://www.google.com> after 30 seconds and add a comment explaining this in the document header.

```

<!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 HTML Semantics and Document Hierarchy

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 032.2](#)

Weight

2

Key knowledge areas

- Create markup for contents in an HTML document
- Understand the hierarchical HTML text structure
- Differentiate between block and inline HTML elements
- Understand important semantic structural HTML elements

Partial list of the used files, terms and utilities

- `<h1>, <h2>, <h3>, <h4>, <h5>, <h6>`
- `<p>`
- `, , `
- `<dl>, <dt>, <dd>`
- `<pre>`
- `<blockquote>`
- `, , <code>`
- `, <i>, <u>`
- ``

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



032.2 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	032 HTML Document Markup
Objective:	032.2 HTML Semantics and Document Hierarchy
Lesson:	1 of 1

Introduction

In the previous lesson, we learned that HTML is a markup language that can semantically describe the content of a website. An HTML document contains a so-called skeleton that consists of the HTML elements `<html>`, `<head>`, and `<body>`. While the `<head>` element describes a block of meta information for the HTML document that will be invisible to the visitor to the website, the `<body>` element may contain many other elements to define the structure and content of the HTML document.

In this lesson, we will walk through text formatting, fundamental semantic HTML elements and their purpose, and how to structure an HTML document. We'll use a shopping list as our example.

NOTE

All subsequent code examples lie within the `<body>` element of an HTML document containing the complete skeleton. For readability, we will not show the HTML skeleton in every example in this lesson.

Text

In HTML, no block of text should be bare, standing outside an element. Even a short paragraph

should be wrapped in the `<p>` HTML tags, which is short name for *paragraph*.

`<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>`

Opened in a web browser, this HTML code produces the result shown in [Figure 1](#).

Short text element spanning only one line

A text element containing much longer text that may span across multiple lines depending on the size of the web browser window.

Figure 1. Web browser representation of HTML code showing two paragraphs of text. The first paragraph is very short. The second paragraph is a bit longer and wraps into a second line.

By default, web browsers add spacing before and after `<p>` elements for improved readability. For this reason, `<p>` is called a *block element*.

Headings

HTML defines six levels of headings to describe and structure the content of an HTML document. These headings are marked by the HTML tags `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` and `<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>`

A web browser renders this HTML code as shown in [Figure 2](#).

Headline level 1 to uniquely identify the page

Headline level 2

Headline level 3

Headline level 4

Headline level 5

Headline level 6

Figure 2. Web browser representation of HTML code showing different levels of headings in an HTML document. The hierarchy of headings is indicated through the size of the text.

If you are familiar with word processors such as LibreOffice or Microsoft Word, you may notice some similarities in how an HTML document uses different levels of headings and how they are rendered in the web browser. By default, HTML uses size to indicate the hierarchy and importance of headings and adds space before and after every heading to visually separate it from content.

A heading using the element `<h1>` is at the top of the hierarchy and thus is considered the most important heading that identifies the content of the page. It's comparable to the `<title>` element discussed in the previous lesson, but within the content of the HTML document. Subsequent heading elements can be used to further structure the content. Make sure not to skip heading levels in between. The document hierarchy should begin with `<h1>`, continue with `<h2>`, then `<h3>` and so on. You don't need to use every heading element down to `<h6>` if your content does not demand it.

NOTE

Headings are important tools to structure an HTML document, both semantically and visually. However, headings should never be used to increase the size of otherwise structurally unimportant text. By the same principle, one should not make a short paragraph bold or italic to make it look like a heading; use heading tags to mark headings.

Let's begin creating the shopping list HTML document by defining its outline. We will create an `<h1>` element to contain the page title, in this case `Garden Party`, followed by short information wrapped in a `<p>` element. Additionally, we use two `<h2>` elements to introduce the two content sections `Agenda` and `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>
```

Opened in a web browser, this HTML code produces the result shown in [Figure 3](#).

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 3. Web browser representation of HTML code showing a simple example document describing an invitation to a garden party, with two headings for agenda and list of things to bring.

Line Breaks

Sometimes it may be necessary to cause a *line break* without inserting another `<p>` element or any similar block element. In such cases, you can use the self-closing `
` element. Note that it should be used only to insert line breaks that belong to the content, as is the case for poems, song lyrics, or addresses. If the content is separated by meaning, it is better to use a `<p>` element instead.

For example, we could split up the text in the information paragraph from our previous example as follows:

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

In a web browser, this HTML code produces the result shown in [Figure 4](#).

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

Figure 4. Web browser representation of HTML code showing a simple example document with a forced line break.

Horizontal Lines

The `<hr>` element defines a horizontal line, also called a *horizontal rule*. By default, it spans the whole width of its parent element. The `<hr>` element can help you define a thematic change in the content or separate the sections of the document. The element is self-closing and therefore has no closing tag.

For our example, we could separate the two headings:

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

Figure 5 shows the result of this code.

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 5. Web browser representation a simple example document describing a shopping list with two sections separated by a horizontal line.

HTML Lists

In HTML, you can define three types of lists:

Ordered lists

where the order of the list elements matters

Unordered lists

where the order of the list elements is not particularly important

Description lists

to more closely describe some terms

Each contains any number of *list items*. We'll describe each type of list.

Ordered Lists

An *ordered list* in HTML, denoted using the HTML element ``, is a collection of any number of *list items*. What makes this element special is that the order of its list elements is relevant. To emphasize this, web browsers display numerals before the child list items by default.

NOTE

`` elements are the only valid child elements within an `` element.

For our example, we can fill in the agenda for the garden party using an `` element with the following code:

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

In a web browser, this HTML code produces the result shown in [Figure 6](#).

Agenda

1. Welcome
2. Barbecue
3. Dessert
4. Fireworks

Figure 6. Web browser representation of a simple example document containing a second-level heading followed by an ordered list with four items describing the agenda for a garden party.

Options

As you can see in this example, the list items are numbered with decimal numerals beginning at 1 by default. However, you can change this behavior by specifying the `type` attribute of the `` tag. Valid values for this attribute are `1` for decimal numerals, `A` for uppercase letters, `a` for lowercase letters, `I` for Roman uppercase numerals, and `i` for Roman lowercase numerals.

If you want, you can also define the starting value by using the `start` attribute of the `` tag. The `start` attribute always takes a decimal numerical value, even if the `type` attribute sets a different type of numbering.

For example, we could adjust the ordered list from the previous example so that the list items will be prefixed with capital letters, beginning with the letter C, as shown in the following example:

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

Within a web browser, this HTML code is rendered like [Figure 7](#).

Agenda

- C. Welcome
- D. Barbecue
- E. Dessert
- F. Fireworks

Figure 7. Web browser representation a simple example document containing a second-level heading followed by an ordered list with list items that are prefixed by capital letters beginning with the letter C.

The order of the list items can also be reversed using the `reversed` attribute without a value.

NOTE In an ordered list, you can also set the initial value of a specific list item using the `value` attribute of the `` tag. List items that follow will increment from that number. The `value` attribute always takes a decimal numerical value.

Unordered Lists

An *unordered list* contains a series of list items that, unlike those in an ordered list, do not have a special order or sequence. The HTML element for this list is ``. Once again, `` is the HTML element to mark its list items.

NOTE `` elements are the only valid child elements within a `` element.

For our example web site, we can use the unordered list to list items for guests to bring to the party. We can achieve this with the following HTML code:

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

Within a web browser, this HTML code produces the display shown in [Figure 8](#).

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 8. Web browser representation of a simple document containing a second-level heading followed by an unordered list with list items regarding foodstuffs that guests are asked to bring to the garden party.

By default, each list item is represented using a disc bullet. You may change its appearance using CSS, which will be discussed in later lessons.

Nesting Lists

Lists can be nested within other lists, such as ordered lists within unordered lists and vice versa. To achieve this, the nested list must be part of a list element ``, because `` is the only valid child element of unordered and ordered lists. When nesting, be careful not to overlap the HTML tags.

For our example, we could add some information of the agenda we created before, as shown in the following example:

```
<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>
```

A web browser renders the code as shown in [Figure 9](#).

Agenda

- C. Welcome
- D. Barbecue
 - Vegetables
 - Meat
 - Burgers, including vegetarian options
- E. Dessert
- F. Fireworks

Figure 9. Web browser representation of HTML code showing an unordered lists nested within an ordered list, to represent the agenda for a garden party.

You could go even further and nest multiple levels deep. Theoretically, there is no limit to how many lists you can nest. When doing this however, consider readability for your visitors.

Description Lists

A *description list* is defined using the `<dl>` element and represents a dictionary of *keys* and *values*. The key is a term or name that you want to describe, and the value is the description. Description lists can range from simple key-value pairs to extensive definitions.

A key (or *term*) is defined using the `<dt>` element, while its description is defined using the `<dd>` element.

An example for such a description list could be a list of exotic fruits that explains what they look like.

```
<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>
```

In a web browser, this produces the result shown in [Figure 10](#).

Exotic Fruits

Banana

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.

Kiwi

A small, oval fruit with green flesh, black seeds and a brown, hairy skin.

Mango

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.

Figure 10. An example of an HTML description list using exotic fruits. The list describes the appearance of three different exotic fruits.

NOTE

As opposed to ordered lists and unordered lists, in a description list, any HTML element is valid as a direct child. This allows you to group elements and style them elsewhere using CSS.

Inline Text Formatting

In HTML, you can use formatting elements to change the appearance of the text. These elements can be categorized as *presentation elements* or *phrase elements*.

Presentation Elements

Basic presentation elements change the font or look of text; these are ``, `<i>`, `<u>` and `<tt>`. These elements were originally defined before CSS let make text bold, italic, etc. There are now usually better ways to alter the look of text, but you still see these elements sometimes.

Bold Text

To make text bold, you can wrap it within the `` element as illustrated in the following example. The result appears in [Figure 11](#).

This ``word`` is bold.

This **word** is bold.

Figure 11. The `` tag is used to make text bold.

According to the HTML5 specification, the `` element should be used only when there are no more appropriate tags. The element that produces the same visual output, but additionally adds semantic importance to the marked text, is ``.

Italic Text

To italicize text, you can wrap it within the `<i>` element as illustrated in the following example. The result appears in [Figure 12](#).

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

This *word* is in italics.

Figure 12. The `<i>` tag is used to italicize text.

According to the HTML 5 specification, the `<i>` element should be used only when there are no more appropriate tags.

Underlined Text

To underline text, you can wrap it within the `<u>` element as illustrated in the following example. The result appears in [Figure 13](#).

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

This word is underlined.

Figure 13. The `<u>` tag is used to underline text.

According to the HTML 5 specification, the `<u>` element should be used only when there are no better ways to underline text. CSS provides a modern alternative.

Fixed-Width or Monospaced Font

To display text in a monospaced (fixed-width) font, often used to display computer code, you can use the `<tt>` element as illustrated in the following example. The result appears in [Figure 14](#).

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

This word is in fixed-width font.

Figure 14. The `<tt>` tag is used to display text in a fixed-width font.

The `<tt>` tag is not supported in HTML5. Web browsers still render it as expected. However, you should use more appropriate tags, which include `<code>`, `<kbd>`, `<var>`, and `<samp>`.

Phrase Elements

Phrase elements not only change the appearance of text, but also add semantic importance to a word or phrase. Using them, you can emphasize a word or mark it as important. These elements, as opposed to presentation elements, are recognized by screen readers, which makes the text more accessible to visually impaired visitors and allows search engines to better read and evaluate the page content. The phrase elements we use throughout this lesson are ``, ``, and `<code>`.

Emphasized Text

To emphasize text, you can wrap it within the `` element as illustrated in the following example:

This `word` is emphasized.

This **word** is emphasized.

Figure 15. The `` tag is used to emphasize text.

As you can see, web browsers display `` in the same way as `<i>`, but `` adds semantic importance as a phrase element, which improves accessibility for visually impaired visitors.

Strong Text

To mark text as important, you can wrap it within the `` element as illustrated in the following example. The result appears in Figure 16.

This `word` is important.

This **word** is important.

Figure 16. The `` tag is used to mark text as important.

As you can see, web browsers display `` in the same way as ``, but `` adds semantic importance as a phrase element, which improves accessibility for visually impaired visitors.

Computer Code

To insert a piece of computer code, you can wrap it within the `<code>` element as illustrated in the following example. The result appears in Figure 17.

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

The **Markdown code # Heading** creates a heading at the highest level in the hierarchy.

Figure 17. The `<code>` tag is used to insert a piece of computer code.

Marked Text

To highlight text with a yellow background, similar to the style of a highlighter, you can use the

`<mark>` element as illustrated in the following example. The result appears in [Figure 18](#).

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

This **word** is highlighted.

Figure 18. The `<mark>` tag is used to highlight text with a yellow background.

Formatting the Text of our HTML Shopping List

Drawing on our previous examples, let's insert some phrase elements to change the appearance of the text while also adding semantic importance. The result appears in [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>
```

Garden Party

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

Agenda

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

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 19. The HTML page with some formatting elements.

In this sample HTML document, the most important information regarding the garden party itself is marked as important using the `` element. Foods that are available for the barbecue are emphasized using the `` element. The fireworks are simply highlighted using the `<mark>` element.

As an exercise, you can try formatting other pieces of text using the other formatting elements as well.

Preformatted Text

In most HTML elements, white space is usually reduced to a single space or even ignored entirely. However, there is an HTML element named `<pre>` that lets you define so-called *preformatted* text. White space in the content of this element, including spaces and line breaks, is preserved and rendered in the web browser. Additionally, the text is displayed in a fixed-width font, similar to the

<code> element.

```

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

```

```

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

```

Figure 20. Web browser representation of HTML code illustrating how the <pre> HTML element preserves white space.

Grouping Elements

By convention, HTML elements are divided into two categories:

Block-Level Elements

These appear on a new line and take up the entire available width. Examples of block-level elements that we already discussed are <p>, , and <h2>.

Inline-Level Elements

These appear in the same line as other elements and text, taking up only as much space as their content requires. Examples of inline-level elements are , , and <i>.

NOTE

HTML5 has introduced more accurate and precise element categories, trying to avoid confusion with CSS block and inline boxes. For simplicity, we will stick here to the conventional subdivision into block and inline elements.

The fundamental elements for grouping multiple elements together are the <div> and elements.

The <div> element is a block-level container for other HTML elements and does not add semantic value by itself. You can use this element to divide an HTML document into sections and structure your content, both for code readability and to apply CSS styles to a group of elements, as you will learn in a later lesson.

By default, web browsers always insert a line break before and after each `<div>` element so that each is displayed on its own line.

In contrast, the `` element is used as a container for HTML text and is generally used to group other inline elements in order to apply styles using CSS to a smaller portion of text.

The `` element behaves just like regular text and does not start on a new line. It is therefore an inline element.

The following example compares the visual representation of the semantic `<p>` element and the grouping elements `<div>` and ``:

```
<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>
```

A web browser renders this code as shown in [Figure 21](#).

Text within a paragraph

Another paragraph of text

Text wrapped within a `div` element
Another `div` element with more text

Span content and more span content

Figure 21. Web browser representation of a test document illustrating the differences between paragraph, div and span elements in HTML.

We already saw that by default, the web browser adds spacing before and after `<p>` elements. This spacing is not applied to either of the grouping elements `<div>` and ``. However, `<div>` elements are formatted as their own blocks, while the text in `` elements is shown in the same line.

HTML Page Structure

We have discussed how to use HTML elements to describe the content of a web page semantically—in other words, to convey meaning and context to the text. Another group of elements are designed for the purpose of describing the *semantic structure* of a web page, an expression or its structure. These elements are block elements, i.e., they visually behave similarly to a `<div>` element. Their purpose is to define the semantic structure of a web page by specifying well-defined areas such as headers, footers and the page's main content. These elements allow the semantic grouping of content so that it may be understood by a computer as well, including search engines and screen readers.

The `<header>` Element

The `<header>` element contains introductory information to the surrounding semantic element within an HTML document. A header is different from a heading, but a header often includes a heading element (`<h1>`, ..., `<h6>`).

In practice, this element is most often used to represent the page header, such as a banner with a logo. It can also be used to introduce the content for any of the following elements: `<body>`, `<section>`, `<article>`, `<nav>`, or `<aside>`.

A document may have multiple `<header>` elements, but a `<header>` element cannot be nested within another `<header>` element. Neither can a `<footer>` element be used within a `<header>`.

For example, to add a page header to our example document, we can do the following:

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

There will be no visible changes to the HTML document, as `<h1>` (like all other heading elements) is a block-level element with no further visual properties.

The `<main>` Content Element

The `<main>` element is a container for the central content of a web page. There must be no more than one `<main>` element in an HTML document.

In our example document, all the HTML code we have written so far would be placed inside the `<main>` element.

```
<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>
```

Like the `<header>` element, the `<main>` element does not cause any visual changes in our example.

The `<footer>` Element

The `<footer>` element contains footnotes, for example authorship information, contact information, or related documents, for its surrounding semantic element, e.g. `<section>`, `<nav>`, or `<aside>`. A document can have multiple `<footer>` elements that allow you to better describe semantic elements.

However, a `<footer>` element cannot be nested within another `<footer>` element, nor can a `<header>` element be used within a `<footer>`.

For our example, we can add contact information for the host (John) as shown in the following example:

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

The `<nav>` Element

The `<nav>` element describes a major navigational unit, such as a menu, that contains a series of hyperlinks.

NOTE

Not all hyperlinks must be wrapped within a `<nav>` element. It's useful when listing a group of links.

Because hyperlinks have not yet been covered, the navigation element will not be included in this lesson's example.

The `<aside>` Element

The `<aside>` element is a container for content that is not necessary within the ordering of the main page content, but is usually indirectly related or supplementary. This element is often used for sidebars that display secondary information, such as a glossary.

For our example, we can add address and journey information, which are only indirectly related to the remaining content, using the `<aside>` element.

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

The `<section>` Element

The `<section>` element defines a logical section in a document that is part of the surrounding semantic element, but would not work as stand-alone content, such as a chapter.

In our example document, we can wrap the content sections for the agenda and bring in list sections as shown in the following example:

```
<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>
```

This example also adds further `<header>` elements within the sections, so that each section is within

its own `<header>` element.

The `<article>` Element

The `<article>` element defines independent and standalone content that makes sense on its own without the rest of the page. Its content is potentially redistributable or reusable in another context. Typical examples or material appropriate for an `<article>` element are a blog posting, a product listing for a shop, and an advertisement for a product. The advertisement could then exist both on its own and within a larger page.

In our example, we can replace the first `<section>` that wraps the agenda with an `<article>` element.

```
<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>
```

The `<header>` element we added in the previous example may persist here as well, because `<article>` elements may have their own `<header>` elements.

The Final Example

Combining all previous examples, the final HTML document for our invitation looks as follows:

```
<!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>
    </p>
  </aside>

```

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

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

In a web browser, the whole page is rendered as shown in [Figure 22](#).

Garden Party

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

Agenda

1. Welcome
 2. Barbecue
 - o *Vegetables*
 - o *Meat*
 - o *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. Web browser representation of the resulting HTML document combining the previous examples. The page represents an invitation to a garden party and describes the agenda for the evening and a list of food for the guests to bring.

Guided Exercises

1. For each of the following tags, indicate the corresponding closing tag:

<h5>	
	
<dd>	
<hr>	
	
<tt>	
<main>	

2. For each of the following tags, indicate whether it marks the beginning of a block or inline element:

<h3>	
	
	
<div>	
	
<dl>	
	
<nav>	
<code>	
<pre>	

3. What kind of lists can you create in HTML? Which tags should you use for each of them?

4. What tags enclose the block elements that you can use to structure an HTML page?

Explorational Exercises

1. Create a basic HTML page with the title “Form Rules”. You will use this HTML page for all explorational exercises, each of which is based on the previous ones. Then add a level 1 heading with the text “How to fill in the request form”, a paragraph with the text “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” and an unordered list with the following list items: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, and “Zip/Postal Code”.

2. Put the first three fields (“Name”, “Surname”, and “Email Address”) in bold, while also adding semantic importance. Then add a level 2 heading with the text “Required fields” and a paragraph with the text “Bold fields are mandatory.”

3. Add another level 2 heading with the text “Steps to follow”, a paragraph with the text “There are four steps to follow:”, and an ordered list with the following list items: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, and “Check your e-mail - You will receive the full HTML course in minutes”.

4. Using `<div>`, create a block for each section that starts with a level 2 heading.

5. Using `<div>`, create another block for the section starting with the level 1 heading. Then divide this section from the other two with an horizontal line.

6. Add the header element with the text “Form Rules - 2021” and the footer element with the text “Copyright Note - 2021”. Finally, add the main element that must contain the three `<div>` blocks.

Summary

In this lesson you learned:

- How to create markup for contents in an HTML document
- The hierarchical HTML text structure
- The difference between block and inline HTML elements
- How to create HTML documents with a semantic structure

The following terms were discussed in this lesson:

<h1>, <h2>, <h3>, <h4>, <h5>, <h6>

The heading tags.

<p>

The paragraph tag.

The ordered list tag.

The unordered list tag.

The list item tag.

<dl>

The description list tag.

<dt>, <dd>

The tags of each term and description for a description list.

<pre>

The preserve formatting tag.

, <i>, <u>, <tt>, , , <code>, <mark>

The formatting tags.

**<div>, **

The grouping tags.

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

The tags used to provide a simple structure and layout to an HTML page.

Answers to Guided Exercises

1. For each of the following tags, indicate the corresponding closing tag:

<h5>	</h5>
 	Does not exist
	
<dd>	</dd>
<hr>	Does not exist
	
<tt>	</tt>
<main>	</main>

2. For each of the following tags, indicate whether it marks the beginning of a block or inline element:

<h3>	Block Element
	Inline Element
	Inline Element
<div>	Block Element
	Inline Element
<dl>	Block Element
	Block Element
<nav>	Block Element
<code>	Inline Element
<pre>	Block Element

3. What kind of lists can you create in HTML? Which tags should you use for each of them?

In HTML, you can create three types of lists: ordered lists consisting of a series of numbered list items, unordered lists consisting of a series of list items that have no special order or sequence, and description lists representing entries as in a dictionary or encyclopedia. An ordered list is enclosed between the `` and `` tags, an unordered list is enclosed between the `` and `` tags, and a description list is enclosed between the `<dl>` and `</dl>` tags. Each item in an ordered or unordered list is enclosed between the `` and `` tags, while each term in a

description list is enclosed between the `<dt>` and `</dt>` tags and its description is enclosed between the `<dd>` and `</dd>` tags.

4. What tags enclose the block elements that you can use to structure an HTML page?

The `<header>` and `</header>` tags enclose the page header, the `<main>` and `</main>` tags enclose the main content of the HTML page, the `<nav>` and `</nav>` tags enclose the so-called navigation bar, the `<aside>` and `</aside>` tags enclose the sidebar, and the `<footer>` and `</footer>` tags enclose the page footer.

Answers to Explorational Exercises

1. Create a basic HTML page with the title “Form Rules”. You will use this HTML page for all explorational exercises, each of which is based on the previous ones. Then add a level 1 heading with the text “How to fill in the request form”, a paragraph with the text “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” and an unordered list with the following list items: “Name”, “Surname”, “Email Address”, “Nation”, “Country”, and “Zip/Postal Code”.

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

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li>Name</li>
      <li>Surname</li>
      <li>Email Address</li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>
  </body>
</html>
```

2. Put the first three fields (“Name”, “Surname”, and “Email Address”) in bold, while also adding semantic importance. Then add a level 2 heading with the text “Required fields” and a paragraph with the text “Bold fields are mandatory.”

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

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>
  </body>
</html>
```

3. Add another level 2 heading with the text “Steps to follow”, a paragraph with the text “There are four steps to follow:”, and an ordered list with the following list items: “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, and “Check your e-mail - You will receive the full HTML course in minutes”.

```

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

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>

    <h2>Steps to follow</h2>
    <p>There are four steps to follow:</p>
    <ol>
      <li>Fill in the fields</li>
      <li>Click the Submit button</li>
      <li>
        Check your e-mail and confirm your request by clicking on the link you
        receive
      </li>
      <li>
        Check your e-mail – You will receive the full HTML course in minutes
      </li>
    </ol>
  </body>
</html>

```

4. Using `<div>`, create a block for each section that starts with a level 2 heading.

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

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
      <li><strong> Email Address </strong></li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
          you
          receive
        </li>
        <li>
          Check your e-mail – You will receive the full HTML course in minutes
        </li>
      </ol>
    </div>
  </body>
</html>
```

5. Using `<div>`, create another block for the section starting with the level 1 heading. Then divide this section from the other two with an horizontal line.

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

  <body>
    <div>
      <h1>How to fill in the request form</h1>
      <p>
        To receive the PDF document with the complete HTML course, it is
        necessary to fill in the following fields:
      </p>
      <ul>
        <li><strong> Name </strong></li>
        <li><strong> Surname </strong></li>
        <li><strong> Email Address </strong></li>
        <li>Nation</li>
        <li>Country</li>
        <li>Zip/Postal Code</li>
      </ul>
    </div>

    <hr>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
          you
          receive
        </li>
      </ol>
    </div>
  </body>
</html>
```

```
<li>
  Check your e-mail – You will receive the full HTML course in minutes
</li>
</ol>
</div>
</body>
</html>
```

6. Add the header element with the text “Form Rules - 2021” and the footer element with the text “Copyright Note - 2021”. Finally, add the main element that must contain the three `<div>` blocks.

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

  <body>
    <header>
      <h1>Form Rules – 2021</h1>
    </header>

    <main>
      <div>
        <h1>How to fill in the request form</h1>
        <p>
          To receive the PDF document with the complete HTML course, it is
          necessary to fill in the following fields:
        </p>
        <ul>
          <li><strong> Name </strong></li>
          <li><strong> Surname </strong></li>
          <li><strong> Email Address </strong></li>
          <li>Nation</li>
          <li>Country</li>
          <li>Zip/Postal Code</li>
        </ul>
      </div>

      <hr>

      <div>
        <h2>Required fields</h2>
        <p>Bold fields are mandatory.</p>
      </div>
    </main>
  </body>
</html>
```

```
</div>

<div>
  <h2>Steps to follow</h2>
  <p>There are four steps to follow:</p>
  <ol>
    <li>Fill in the fields</li>
    <li>Click the Submit button</li>
    <li>
      Check your e-mail and confirm your request by clicking on the link
      you receive
    </li>
    <li>
      Check your e-mail – You will receive the full HTML course in
      minutes
    </li>
  </ol>
</div>
</main>

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



032.3 HTML References and Embedded Resources

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 032.3

Weight

2

Key knowledge areas

- Create links to external resources and page anchors
- Add images to HTML documents
- Understand key properties of common media file formats, including PNG, JPG and SVG
- Awareness of iframes

Partial list of the used files, terms and utilities

- `id` attribute
- `<a>`, including the `href` and `target` (`_blank`, `_self`, `_parent`, `_top`) attributes
- ``, including the `src` and `alt` attributes



032.3 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	032 HTML Document Markup
Objective:	032.3 HTML References and Embedded Resources
Lesson:	1 of 1

Introduction

Any modern web page is rarely made of just text. It comprises many other types of contents, such as images, audio, video and even other HTML documents. Along with external content, HTML documents can contain links to other documents, which makes the experience of browsing the Internet much simpler.

Embedded Content

File exchange is possible over the Internet without web pages written in HTML, so why is HTML the chosen format for web documents and not PDF or any other word processing format? One important reason is that HTML keeps its multimedia resources in separate files. In an environment such as the Internet, where information is often redundant and distributed at different locations, it is important to avoid unnecessary data transfers. Most of the time, new versions of a web page pull in the same images and other support files as previous versions, so the web browser can use the previously fetched files instead of copying everything over again. Furthermore, keeping the files separate facilitates the customization of multimedia content according to the client's characteristics, such as their location, screen size, and connection speed.

Images

The most common type of embedded content are images that accompany the text. Images are kept separately and are referenced inside the HTML file with the `` tag:

```

```

The `` tag does not require a closing tag. The `src` property indicates the source location for the image file. In this example, the image file `logo.png` must be located in the same directory as the HTML file, otherwise the browser will not be able to display it. The source location property accepts relative paths, so the *dot notation* can be used to indicate the path to the image:

```

```

The two dots indicate that the image is located inside the parent directory relative to the directory where the HTML file is. If the filename `../logo.png` is used inside an HTML file whose URL is `http://example.com/library/periodicals/index.html`, the browser will request the image file at the address `http://example.com/library/logo.png`.

The dot notation also applies if the HTML file isn't an actual file in the filesystem; the HTML browser interprets the URL as if it is a path to a file, but it is the HTTP server's job to decide whether that path refers to a file or to dynamically generated content. The domain and the proper path are automatically added to all requests to the server, in case the HTML file came from a HTTP request. Likewise, the browser will open the proper image if the HTML file was opened directly from the local filesystem.

Source locations beginning with a forward slash `/` are treated as absolute paths. Absolute paths have complete information for the image's location, so they work regardless of the HTML document's location. If the image file is located at another server, which will be the case when a *Content Delivery Network* (CDN) is used, the domain name must also be included.

NOTE

Content Delivery Networks are composed of geographically distributed servers that store static content for other websites. They help to improve performance and availability for heavily accessed sites.

If the image can't be loaded, the HTML browser will show the text provided by the `alt` attribute instead of the image. For example:

```

```

The `alt` attribute is also important for accessibility. Text-only browsers and screen readers use it as a description for the corresponding image.

Image Types

Web browsers can display all the popular image types, such as JPEG, PNG, GIF, and SVG. The dimensions of the images are detected as soon as the images are loaded, but they can be predefined with the `width` and `height` attributes:

```

```

The only reason for including dimension attributes to the `` tag is to avoid breaking the layout when the image takes too long to load or when it can not be loaded at all. Using the `width` and `height` attributes to change the image's original dimensions may result in undesirable results:

- Images will be distorted when the original size is smaller than the new dimensions or when the new proportion ratio differs from the original.
- Downsizing large images uses extra bandwidth that will result in longer loading timings.

SVG is the only format that doesn't suffer from these effects, because all of its graphical information is stored in numerical coordinates well suited for scaling and its dimensions don't affect the file size (hence the name *Scalable Vector Graphics*). For example, only the position, side dimensions, and color information are necessary to draw a rectangle in SVG. The particular value for every single pixel will be dynamically rendered afterwards. In fact, SVG images are similar to HTML files, in the sense that their graphic elements are also defined by tags in a text file. SVG files are intended for representing sharp-edged drawings, such as charts or diagrams.

Images that don't fit these criteria should be stored as *bitmaps*. Unlike vector based image formats, bitmaps store color information for every pixel in the image beforehand. Storing the color value for each pixel in the image generates a very large amount of data, so bitmaps are usually stored in compressed formats, such as JPEG, PNG, or GIF.

The JPEG format is recommended for photographs, because its compression algorithm produces good results for shades and blurry backgrounds. For images where solid colors prevail, the PNG format is more appropriate. Therefore, the PNG format should be chosen when it is necessary to convert a vector image to a bitmap.

The GIF format offers the lowest image quality of all the popular bitmap formats. Nevertheless, it is still widely used because of its support for animations. Indeed, many websites employ GIF files to display short videos, but there are better ways to display video content.

Audio and Video

Audio and video contents may be added to an HTML document in more or less the same way as images. Unsurprisingly, the tag to add audio is `<audio>` and the tag to add video is `<video>`. Obviously, text-only browsers aren't able to play multimedia content, so the `<audio>` and `<video>` tags employ the closing tag to hold the text used as a fallback for the element that could not be shown. For example:

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

If the browser doesn't support the `<audio>` tag, the line "Unable to play recording.mp3" will be shown instead. The use of closing `</audio>` or `</video>` tags allows a web page to include more elaborate alternative content than the simple line of text permitted by the `alt` attribute of the `` tag.

The `src` attribute for `<audio>` and `<video>` tags works in the same way as for the `` tag, but also accepts URLs pointing to a live stream. The browser takes care of buffering, decoding, and displaying the content as it is received. The `controls` attribute displays playback controls. Without it, the visitor will not be able to pause, rewind, or otherwise control the playback.

Generic Content

An HTML document can be nested into another HTML document, similarly to the insertion of an image into an HTML document, but using the tag `<iframe>`:

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

Simpler text-only browsers don't support the `<iframe>` tag and will display the enclosed text instead. As with the multimedia tags, the `src` attribute sets the source location of the nested document. The `width` and `height` attributes can be added to change the default dimensions of the `iframe` element.

The `name` attribute allows to refer to the `iframe` and change the nested document. Without this

attribute, the nested document cannot be changed. An `anchor` element can be used to load a document from another location inside an `iframe` instead of the current browser window.

Links

The page element commonly referred to as a web *link* is also known by the technical term *anchor*, hence the use of tag `<a>`. The anchor leads to another location, which can be any address supported by the browser. The location is indicated by the `href` (*hyperlink reference*) attribute:

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

The location can be written as a relative or absolute path, as with the embedded contents discussed earlier. Only the enclosed text content (e.g., `Contact Information`) is visible to the visitor, usually as clickable underlined blue text by default, but the item displayed over the link can also be any other visible content, such as images:

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

Special prefixes can be added to the location to tell the browser how to open it. If the anchor points to an email address, for example, its `href` attribute should include the `mailto:` prefix:

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

The `tel:` prefix indicates a phone number. It's particularly useful for visitors viewing the page on mobile devices:

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

When the link is clicked, the browser opens the location's contents with the associated application.

The most common use of anchors is to load other web documents. By default, the browser will replace the current HTML document with content at the new location. This behavior can be modified by using the `target` attribute. The `_blank` target, for example, tells the browser to open the given location in a new window or new browser tab, depending on the visitor's preferences:

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

The `_self` target is the default when the `target` attribute is not provided. It causes the referenced document to replace the current document.

Other types of targets are related to the `<iframe>` element. To load a referenced document inside an `<iframe>` element, the `target` attribute should point to the name of the `iframe` element:

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

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

The `iframe` element works as a distinct browser window, so any links loaded from the document inside the `iframe` will replace only the contents of the `iframe`. To change that behaviour, the anchor elements inside the framed document can also use the `target` attribute. The `_parent` target, when used inside a framed document, will cause the referenced location to replace the parent document containing the `<iframe>` tag. For example, the embedded `gallery.html` document could contain an anchor that loads itself while replacing the parent document:

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

HTML documents support multiple levels of nesting with the `<iframe>` tag. The `_top` target, when used in an anchor inside a framed document, will cause the referenced location to replace the main document in the browser window, regardless if it is the immediate parent of the corresponding `<iframe>` or an ancestor further back in the chain.

Locations Inside Documents

The address of an HTML document may optionally contain a *fragment* that can be used to identify a resource inside the document. This fragment, also known as the *URL anchor*, is a string following a hash sign `#` at the end of the URL. For example, the word `History` is the anchor in the URL `https://en.wikipedia.org/wiki/Internet#History`.

When the URL has an anchor, the browser will scroll to the corresponding element in the document: that is, the element whose `id` attribute is equal to the anchor in the URL. In the case of the given URL, `https://en.wikipedia.org/wiki/Internet#History`, the browser will jump straight to the “History” section. Examining the HTML code of the page, we find out that the title of the section has the corresponding `id` attribute:

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

URL anchors can be used in the `href` attribute of the `<a>` tag, either when they are pointing to external pages or when they are pointing to locations inside the current page. In the latter case, it is enough to put start with just the hash sign with the URL fragment, as in `History`.

WARNING

The `id` attribute must not contain whitespace (spaces, tabs, etc.) and must be unique within the document.

There are ways to customize how the browser will react to URL anchors. It is possible, for example, to write a JavaScript function that listens to the `hashchange` window event and triggers a customized action, such as an animation or an HTTP request. It is worth noting, however, that the URL fragment is never sent to the server with the URL, so it cannot be used as an identifier by the HTTP server.

Guided Exercises

1. The HTML document located at <http://www.lpi.org/articles/linux/index.html> has an `` tag whose `src` attribute points to `../logo.png`. What is the complete absolute path to this image?

2. Name two reasons why the `alt` attribute is important in `` tags.

3. What image format gives good image quality and keeps the file size small when it is used for photographs with blurry points and with many colors and shades?

4. Instead of using a third-party provider such as Youtube, what HTML tag lets you embed a video file in an HTML document using only standard HTML features?

Explorational Exercises

1. Assume that an HTML document has the hyperlink `First picture` and the iframe element `<iframe name="gallery"></iframe>`. How could you modify the hyperlink tag so the image it points to will load inside the given iframe element after the user clicks on the link?

2. What will happen when the visitor clicks a hyperlink in a document inside an iframe and the hyperlink has the target attribute set to `_self`?

3. You notice that the URL anchor for the second section of your HTML page is not working. What is the probable cause of this error?

Summary

This lesson covers how to add images and other multimedia content using the proper HTML tags. Moreover, the reader learns the different ways hyperlinks can be used to load other documents and point to specific locations inside a page. The lesson goes through the following concepts and procedures:

- The `` tag and its main attributes: `src` and `alt`.
- Relative and absolute URL paths.
- Popular image formats for the Web and their characteristics.
- The `<audio>` and `<video>` multimedia tags.
- How to insert nested documents with the `<iframe>` tag.
- The hyperlink tag `<a>`, its `href` attribute, and special targets.
- How to use URL fragments, also known as hash anchors.

Answers to Guided Exercises

1. The HTML document located at <http://www.lpi.org/articles/linux/index.html> has an `` tag whose `src` attribute points to `../logo.png`. What is the complete absolute path to this image?

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

2. Name two reasons why the `alt` attribute is important in `` tags.

Text-only browsers will be able to show a description of the missing image. Screen readers use the `alt` attribute to describe the image.

3. What image format gives good image quality and keeps the file size small when it is used for photographs with blurry points and with many colors and shades?

The JPEG format.

4. Instead of using a third-party provider such as Youtube, what HTML tag lets you embed a video file in an HTML document using only standard HTML features?

The `<video>` tag.

Answers to Explorational Exercises

1. Assume that an HTML document has the hyperlink `First picture` and the iframe element `<iframe name="gallery"></iframe>`. How could you modify the hyperlink tag so the image it points to will load inside the given iframe element after the user clicks on the link?

Using the `target` attribute of the `a` tag: `First picture`.

2. What will happen when the visitor clicks a hyperlink in a document inside an iframe and the hyperlink has the `target` attribute set to `_self`?

The document will be loaded inside the same iframe, which is the default behaviour.

3. You notice that the URL anchor for the second section of your HTML page is not working. What is the probable cause of this error?

The URL fragment after the hash sign doesn't match the `id` attribute in the element corresponding to the second section, or the `id` attribute of the element is not present.



032.4 HTML Forms

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 032.4

Weight

2

Key knowledge areas

- Create simple HTML forms
- Understand HTML form methods
- Understand HTML input elements and types

Partial list of the used files, terms and utilities

- `<form>`, including the `method` (get, post), `action`, and `enctype` attributes
- `<input>`, including the `'type` (text, email, password, number, date, file, range, radio, checkbox, hidden) attribute
- `<button>`, including the `type` (submit, reset, hidden, button) attribute
- `<textarea>`
- Common form element attributes (`name`, `value`, `id`)
- `<label>`, including the `'for` attribute



032.4 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	032 HTML Document Markup
Objective:	032.4 HTML Forms
Lesson:	1 of 1

Introduction

Web forms provide a simple and efficient way to request visitor information from an HTML page. The front end developer can use various components such as text fields, checkboxes, buttons, and many others to build interfaces that will send data to the server in a structured way.

Simple HTML Forms

Before jumping into markup code specific to forms, let us first start with a simple blank HTML document, without any body content:

```
<!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>
```

Save the code sample as a raw text file with a `.html` extension (as in `form.html`) and use your favorite browser to open it. After changing it, press the reload button in the browser to show the modifications.

The basic form structure is given by the `<form>` tag itself and its internal elements:

```
<!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>
```

The double quotes are not required for single word attributes like `type`, so `type=text` works as well as `type="text"`. The developer can choose which convention to use.

Save the new content and reload the page in the browser. You should see the result shown in [Figure 23](#).

Personal Information

Full name:

[Clear form](#)

[Submit form](#)

Figure 23. A very basic form.

The `<form>` tag by itself does not produce any noticeable result on the page. The elements inside the `<form>...</form>` tags will define the fields and other visual aids shown to the visitor.

The example code contains both general HTML tags (`<h2>` and `<p>`) and the `<input>` tag, which is a form-specific tag. Whereas general tags can appear anywhere in the document, form-specific tags should be used only within the `<form>` element; that is, between the opening `<form>` and closing `</form>` tags.

NOTE

HTML provides only basic tags and properties to modify the standard appearance of forms. CSS provides elaborate mechanisms to modify the look and feel of the form, so the recommendation is to write HTML code that deals only with the functional aspects of the form and modify its appearance with CSS.

As shown in the example, the paragraph tag `<p>` can be used to describe the field to the visitor. However, there is no obvious way the browser could relate the description in the `<p>` tag with the corresponding input element. The `<label>` tag is more appropriate in these cases (from now on, consider all the code samples as being inside the body of the HTML document):

```

<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>

```

The `for` attribute in the `<label>` tag contains the `id` of the corresponding input element. It makes the page more accessible, as screenreaders will be able to speak the contents of the label element when the input element is in focus. Moreover, visitors can click the label to place the focus in its corresponding input field.

The `id` attribute works for form elements like it does for any other element in the document. It provides an identifier for the element that is unique within the entire document. The `name` attribute has a similar purpose, but it is used to identify the input element in the form's context. The browser uses the `name` attribute to identify the input field when sending the form data to the server, so it is important to use meaningful and unique `name` attributes inside the form.

The `type` attribute is the main attribute of the `input` element, because it controls the data type the element accepts and its visual presentation to the visitor. If the `type` attribute is not provided, by default the input shows a text box. The following input types are supported by modern browsers:

Table 1. Form input types

Type attribute	Data type	How it is displayed
<code>hidden</code>	An arbitrary string	N/A
<code>text</code>	Text with no line breaks	A text control
<code>search</code>	Text with no line breaks	A search control
<code>tel</code>	Text with no line breaks	A text control
<code>url</code>	An absolute URL	A text control
<code>email</code>	An email address or list of email addresses	A text control
<code>password</code>	Text with no line breaks (sensitive information)	A text control that obscures data entry

Type attribute	Data type	How it is displayed
date	A date (year, month, day) with no time zone	A date control
month	A date consisting of a year and a month with no time zone	A month control
week	A date consisting of a week-year number and a week number with no time zone	A week control
time	A time (hour, minute, seconds, fractional seconds) with no time zone	A time control
datetime-local	A date and time (year, month, day, hour, minute, second, fraction of a second) with no time zone	A date and time control
number	A numerical value	A text control or spinner control
range	A numerical value, with the extra semantic that the exact value is not important	A slider control or similar
color	An sRGB color with 8-bit red, green and blue components	A color picker
checkbox	A set of zero or more values from a predefined list	A checkbox (offers choices and allows multiple choices to be selected)
radio	An enumerated value	A radio button (offers choices and allows only one choice to be selected)
file	Zero or more files each with a MIME type and optional file name	A label and a button
submit	An enumerated value, which ends the input process and causes the form to be submitted	A button

Type attribute	Data type	How it is displayed
image	A coordinate, relative to a particular image's size, which ends the input process and causes the form to be submitted	Either a clickable image or a button
button	N/A	A generic button
reset	N/A	A button whose function is to reset all other fields to their initial values

The appearance of the `password`, `search`, `tel`, `url`, and `email` input types do not differ from the standard `text` type. Their purpose is to offer hints to the browser about the intended content for that input field, so the browser or the script running on the client side can take custom actions for a specific input type. The only difference between the `text` input type and the `password` field type, for example, is that the contents of the `password` field are not displayed as the visitor types them in. In touch screen devices, where the text is typed with a on-screen keyboard, the browser can pop up only the numerical keyboard when an input of type `tel` gain focus. Another possible action is to suggest a list of known email addresses when an input of type `email` gain focus.

The `number` type also appears as a simple text input, but with increment/decrement arrows at its side. Its use will cause the numerical keyboard to show up in touchscreen devices when it has the focus.

The other input elements have their own appearance and behavior. The `date` type, for example, is rendered according to the local date format settings and a calendar is displayed when the field gains focus:

```

<form>

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

</form>

```

Figure 24 shows how the desktop version of Firefox currently renders this field.

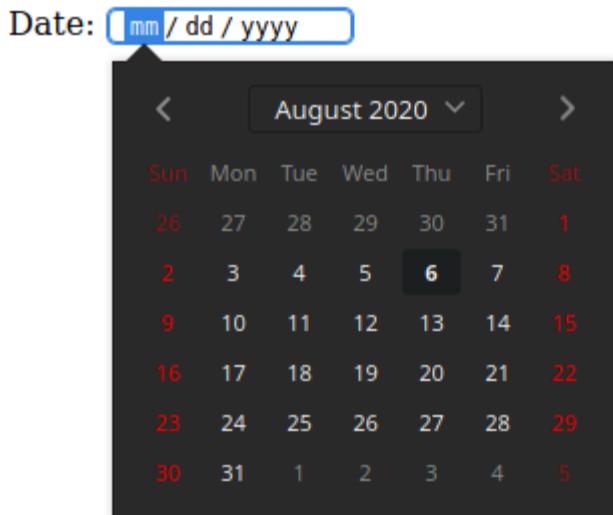


Figure 24. The date input type.

NOTE

For elements may appears slightly different in different browsers or operating systems, but their functioning and usage are always the same.

This is a standard feature in all modern browsers and does not require extra options or programming.

Regardless of the input type, the content of an input field is called its *value*. All field values are empty by default, but the `value` attribute can be used to set a default value for the field. The value for the date type must use the `YYYY-MM-DD` format. The default value in the following date field is 6 August 2020:

```

<form>

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

</form>

```

The specific input types assist the visitor in filling in the fields, but do not prevent the visitor from bypassing the restrictions and entering arbitrary values in any field. That is why it is important that the field values are validated when they arrive at the server.

Form elements whose values must be typed by the visitor may have special attributes that assist in filling them out. The `placeholder` attribute inserts an example value in the input element:

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

The placeholder appears inside the input element, as shown in [Figure 25](#).

Address:

Figure 25. Placeholder attribute example.

Once the visitor starts typing in the field, the placeholder text disappears. The placeholder text is not sent as the field value if the visitor leaves the field empty.

The `required` attribute requires the visitor to fill in a value for the corresponding field before submitting the form:

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

The `required` attribute is a Boolean attribute, so it can be placed alone (without the equal sign). It is important to mark the fields that are required, otherwise the visitor will not be able to tell which fields are missing and preventing the form submission.

The `autocomplete` attribute indicates whether the value of the input element can be automatically completed by the browser. If set to `autocomplete="off"`, then the browser does not suggest past values to fill the entry. Input elements for sensitive information, such as credit card numbers, should have the `autocomplete` attribute set to `off`.

Input for large texts: `textarea`

Unlike the text field, where only one line of text can be inserted, the `textarea` element allows the visitor to enter more than one line of text. The `textarea` is a separate element, but it is not based on the `input` element:

```
<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>
```

The typical appearance of a textarea is [Figure 26](#).

Type your comment here:

My multi-line, plain-text comment.

Figure 26. The textarea element.

Another difference from the input element is that the textarea element has a closing tag (`</textarea>`), so its content (i.e. its value) goes in between them. The `rows` and `cols` attribute do not limit the amount of text; they are used only to define the layout. The textarea also has a handle in the bottom-right corner, which allows the visitor to resize it.

Lists of Options

Several types of form controls can be used to present a list of options to the visitor: the `<select>` element and the `radio` and `checkbox` input types.

The `<select>` element is a dropdown control with a list of predefined entries:

```
<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>
```

The `<option>` tag represents a single entry in the corresponding `<select>` control. The entire list appears when the visitor taps or clicks over the control, as shown in [Figure 27](#).

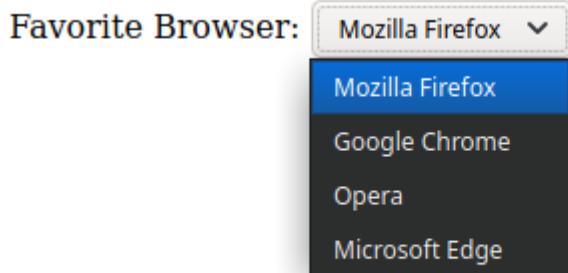


Figure 27. The `select` form element.

The first entry in the list is selected by default. To change this behavior, you can add the `selected` attribute to another entry so it will be selected when the page loads.

The `radio` input type is similar to the `<select>` control, but instead of a dropdown list, it shows all the entries so the visitor can mark one of them. Results of the following code are shown in [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>

```

Favorite Browser:

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 28. Input elements of type radio.

Note that all the `radio` input types in the same group have the same `name` attribute. Each of them is exclusive, so the corresponding `value` attribute for the chosen entry will be the one associated with the shared `name` attribute. The `checked` attribute works like the `selected` attribute of the `<select>` control. It marks the corresponding entry when the page loads for the first time. The `<label>` tag is especially useful for radio entries, because it allows the visitor to check an entry by clicking or tapping on the corresponding text in addition to the control itself.

Whereas `radio` controls are intended for selecting only a single entry of a list, the `checkbox` input type lets the visitor check multiple entries:

```
<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>
```

Checkboxes also can use the `checked` attribute to make entries selected by default. Instead of the round controls of the `radio` input, checkbox are rendered as square controls, as shown in [Figure 29](#).

Favorite Browser:

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 29. The `checkbox` input type.

If more than one entry is selected, the browser will submit them with the same name, requiring the backend developer to write specific code to properly read form data containing checkboxes.

To improve usability, input fields can be grouped together inside a `<fieldset>` tag:

```
<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>
```

The `<legend>` tag contains the text that is placed at the top of the frame the `<fieldset>` tag draws around the controls ([Figure 30](#)).

Favorite Browser

- Mozilla Firefox
- Google Chrome
- Opera
- Microsoft Edge

Figure 30. Grouping elements with the `fieldset` tag.

The `<fieldset>` tag does not change how the field values are submitted to the server, but it lets the frontend developer control the nested controls more easily. For example, setting the `disabled` attribute in a `<fieldset>` attribute will make all its inner elements unavailable to the visitor.

The `hidden` Element Type

There are situations where the developer wants to include information in the form that cannot be manipulated by the visitor, that is, to submit a value chosen by the developer without presenting a form field where the visitor can type or change the value. The developer may want, for example, to include an identification token for that particular form that does not need to be seen by the visitor. A hidden form element is coded as in the following example:

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

The value of a hidden input field is usually added to the document at the server side, when rendering the document. Hidden inputs are treated like ordinary fields when the browser sends them to the server, which also reads them as ordinary input fields.

The File Input Type

In addition to textual data, either typed or selected from a list, HTML forms can also submit arbitrary files to the server. The `file` input type lets the visitor pick a file from the local file system and send it directly from the web page:

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

Instead of a form field to write in or select a value from, the `file` input type shows a `browse` button that will open a file dialog. Any file type is accepted by the `file` input type, but the backend developer will probably restrict the allowed file types and their maximum size. The file type verification can also be performed in the frontend by adding the `accept` attribute. To accept only JPEG and PNG images, for example, the `accept` attribute should be `accept="image/jpeg, image/png"`.

Action Buttons

By default, the form is submitted when the visitor presses the Enter key at any input field. To make things more intuitive, a submit button should be added with the `submit` input type:

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

The text in the `value` attribute is displayed on the button, as shown in [Figure 31](#).

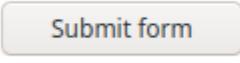


Figure 31. A standard submit button.

Another useful button to include in complex forms is the `reset` button, which clears the form and returns it to its original state:

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

Like the submit button, the text in the `value` attribute is used to label the button. Alternatively, the `<button>` tag can be used to add buttons to forms or anywhere else in the page. Unlike buttons made with the `<input>` tag, the button element has a closing tag and the button label is their inner content:

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

When inside a form, the default action for the `button` element is to submit the form. Like the `input`

buttons, the button's type attribute can be switched to `reset`.

Form Action and Methods

The last step in writing an HTML form is to define how and to where the data should be sent. These aspects depend on details in both the client and the server.

On the server side, the most common approach is to have a script file that will parse, validate, and process the form data according to the application's purpose. For example, the backend developer could write a script called `receive_form.php` to receive the data sent from the form. On the client side, the script is indicated in the `action` attribute of the `form` tag:

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

The `action` attribute follows the same conventions as all HTTP addresses. If the script is in the same hierarchy level of the page containing the form, it can be written without its path. Otherwise, the absolute or the relative path must be supplied. The script should also generate the response to serve as a landing page, loaded by the browser after the visitor submits the form.

HTTP provides distinct methods for sending form data through a connection with the server. The most common methods are `get` and `post`, which should be indicated in the `method` attribute of the `form` tag:

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

Or:

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

In the `get` method, the form data is encoded directly in the request URL. When the visitor submits the form, the browser will load the URL defined in the `action` attribute with the form fields appended to it.

The `get` method is preferred for small amounts of data, such as simple contact forms. However, the URL cannot exceed a few thousands characters, so the `post` method should be used when forms contain large or non-textual fields, like images.

The `post` method makes the browser send the form data in the body section of the HTTP request. While necessary for binary data that exceeds the size limit of a URL, the `post` method adds

unnecessary overhead to the connection when used in simpler textual forms, so the `get` method is preferred in such cases.

The chosen method does not affect how the visitor interacts with the form. The `get` and `post` methods are processed differently by the server-side script that receives the form.

When using the `post` method, it is also possible to change the MIME type of the form contents with `enctype` form attribute. This affects how the form fields and values will be stacked together in the HTTP communication with the server. The default value for `enctype` is `application/x-www-form-urlencoded`, which is similar to the format used in the `get` method. If the form contains input fields of type `file`, the `enctype multipart/form-data` should be used instead.

Guided Exercises

1. What is the correct way to associate a `<label>` tag to an `<input>` tag?

2. What input element type provides a slider control to pick a numerical value?

3. What is the purpose of the `placeholder` form attribute?

4. How could you make the second option in a select control selected by default?

Explorational Exercises

1. How could you change a file input to make it accept only PDF files?

2. How could you inform the visitor about which fields in a form are required?

3. Put together three code snippets in this lesson in a single form. Make sure not to use the same name or `id` attribute in multiple form controls.

Summary

This lesson covers how to create simple HTML forms to send data back to the server. At the client side, HTML forms consist of standard HTML elements that are combined to build custom interfaces. Moreover, forms must be configured to properly communicate with the server. The lesson goes through the following concepts and procedures:

- The `<form>` tag and basic form structure.
- Basic and special input elements.
- The role of special tags like `<label>`, `<fieldset>` and `<legend>`.
- Form buttons and actions.

Answers to Guided Exercises

1. What is the correct way to associate a `<label>` tag to an `<input>` tag?

The `for` attribute of the `<label>` tag should contain the `id` of the corresponding `<input>` tag.

2. What input element type provides a slider control to pick a numerical value?

The `range` input type.

3. What is the purpose of the `placeholder` form attribute?

The `placeholder` attribute contains an example of visitor input that is visible when the corresponding input field is empty.

4. How could you make the second option in a select control selected by default?

The second `option` element should have the `selected` attribute.

Answers to Explorational Exercises

1. How could you change a file input to make it accept only PDF files?

The `accept` attribute of the `input` element should be set to `application/pdf`.

2. How could you inform the visitor about which fields in a form are required?

Usually, the required fields are marked with an asterisk (*), and a brief note like “Fields marked with * are required” is placed close to the form.

3. Put together three code snippets in this lesson in a single form. Make sure not to use the same `name` or `id` attribute in multiple form controls.

```
<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>
```



Topic 033: CSS Content Styling



033.1 CSS Basics

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 033.1

Weight

1

Key knowledge areas

- Embedding CSS within an HTML document
- Understand the CSS syntax
- Add comments to CSS
- Awareness of accessibility features and requirements

Partial list of the used files, terms and utilities

- HTML style and type (text/css) attributes
- `<style>`
- `<link>`, including the `rel` (stylesheet), `type` (text/css) and `src` attributes
- `;`
- `/, /`



033.1 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	033 CSS Content Styling
Objective:	033.1 CSS Basics
Lesson:	1 of 1

Introduction

All web browsers render HTML pages using default presentation rules that are practical and straightforward, but not visually appealing. HTML by itself offers few features to write elaborate pages based on modern user experience concepts. After writing simple HTML pages, you have probably realized they are unsightly when compared to well-designed pages found on the Internet. This is because, in modern HTML, the markup code intended for the structure and function of the elements in the document (i.e., the *semantic content*) is separate from the rules that define how the elements should look (the *presentation*). The rules for presentation are written in a different language called *Cascading Style Sheets* (CSS). It lets you change almost all the document's visual aspects, such as fonts, colors, and the placement of the elements along the page.

HTML documents are not, in most cases, intended to be displayed in a fixed layout like a PDF file. Rather, HTML-based web pages will probably be rendered on a wide variety of screen sizes or even be printed. CSS provides tunable options to ensure that the web page will be rendered as intended, adjusted to the device or application that opens it.

Applying Styles

There are several ways to including CSS in an HTML document: write it directly in the element's tag, in a separate section of the page's source code, or in an external file to be referenced by the HTML page.

The `style` Attribute

The simplest way to modify the style of a specific element is to write it directly in the element tag using the `style` attribute. All visible HTML elements allow a `style` attribute, whose value may be one or more CSS rules, also known as *properties*. Let's start with a simple example, a paragraph element:

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

The basic syntax of a customized CSS property is `property: value`, where `property` is the particular aspect you want to change in the element and `value` defines the replacement for the default option made by the browser. Some properties apply to all elements and some properties apply only to specific elements. Likewise, there are appropriate values to be used in each property.

To change the color of our plain paragraph, for example, we use the `color` property. The `color` property refers to the *foreground* color, that is, the color of the letters in the paragraph. The color itself goes in the `value` part of the rule and it can be specified in many different formats, including simple names like `red`, `green`, `blue`, `yellow`, etc. Thus, to make the letter of the paragraph `purple`, add the customized property `color: purple` to the `style` attribute of the element:

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

Other customized properties can go in the same `style` property, but they must be separated by semi-colons. If you want to make the font size larger, for example, add `font-size: larger` to the `style` property:

```
<p style="color: purple; font-size: larger">My first stylized paragraph</p>
```

NOTE

It's not necessary to add spaces around the colons and semi-colons, but they can make it easier to read the CSS code.

To see the result of these changes, save the following HTML in a file and then open it in a web

browser (results are shown in [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>
```

My first stylized paragraph

My second stylized paragraph

Figure 32. A very simple visual change using CSS.

You can imagine every element in the page as a rectangle or a box whose geometric properties and decorations you can manipulate with CSS. The rendering mechanism uses two basic standard concepts for element placement: *block* placement and *inline* placement. Block elements occupy all the horizontal space of their parent element and are placed sequentially, from top to bottom. Their height (their *vertical dimension*, not to be confused with their *top* position in the page) generally depends on the amount of content they have. Inline elements follow the left-to-right method similar to Western written languages: elements are placed horizontally, from left to right, until there is no more space in the right side, whereupon the element continues on a new line just under the current one. Elements such as p, div, and section are placed as blocks by default, whereas elements such as span, em, a, and single letters are placed inline. These basic placement methods can be fundamentally modified by CSS rules.

The rectangle corresponding to the p element in the body of the sample HTML document will be visible if you add the `background-color` property to the rule ([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>
```

My first stylized paragraph

My second stylized paragraph

Figure 33. Rectangles corresponding to the paragraphs.

As you add new CSS customized properties to the `style` attribute, you'll note that the overall code starts to get cluttered. Writing too many CSS rules in the `style` attribute undermines the separation of structure (HTML) and presentation (CSS). Moreover, you'll soon realize that many styles are shared among different elements and it is not wise to repeat them in every element.

CSS Rules

Rather than styling the elements directly in their `style` attributes, it is much more practical to tell the browser about the entire collection of elements to which the custom style applies. We do it by adding a *selector* to the customized properties, matching elements by type, class, unique ID, relative position, etc. The combination of a *selector* and corresponding customized properties—also known as *declarations*—is called a *CSS rule*. The basic syntax of a CSS rule is `selector { property: value }`. As in the `style` attribute, properties separated by semi-colons can be grouped together, as in `p { color: purple; font-size: larger }`. This rule matches every `p` element in the page and applies the customized `color` and `font-size` properties.

A CSS rule for a parent element will automatically match all its children elements. This means that we could apply the customized properties to all text in the page, regardless of whether it is inside or outside a `<p>` tag, by using the `body` selector instead: `body { color: purple; font-size: larger }`. This strategy frees us from writing the same rule again for all its children, but it may be necessary to write additional rules to “undo” or to modify the inherited rules.

The `style` Tag

The `<style>` tag lets us write CSS rules inside the HTML page we want to style. It allows the browser to differentiate the CSS code from the HTML code. The `<style>` tag goes in the `head` section of the document:

```
<!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>
```

The `type` attribute tells the browser what kind of content is inside the `<style>` tag, i.e. its *MIME type*. Because every browser that supports CSS assumes that the type of the `<style>` tag is `text/css` by default, including the `type` attribute is optional. There is also a `media` attribute that indicates the media—computer screens or printing, for example—to which the CSS rules in the `<style>` tag apply. By default, the CSS rules apply to any medium where the document is rendered.

As in HTML code, line breaks and code indentation do not change how the CSS rules are interpreted by the browser. Writing:

```
<style type="text/css">

  p { color: purple; font-size: larger }

</style>
```

has the exact same result as writing:

```
<style type="text/css">  
  
p {  
  color: purple;  
  font-size: larger;  
}  
  
</style>
```

The extra bytes used by the spaces and line-breaks make little difference in the final size of the document and do not have a significant impact on page's loading time, so the choice of layout is a matter of taste. Note the semi-colon after the last declaration (`font-size: larger;`) of the CSS rule. That semi-colon is not mandatory, but having it there makes it easier to add another declaration in the next line without worrying about missing semi-colons.

Having the declarations in separate lines also helps when you need to comment out a declaration. Whenever you want to temporarily disable a declaration for troubleshooting reasons, for example, you can enclose the corresponding line with `/*` and `*/`:

```
p {  
  color: purple;  
  /*  
  font-size: larger;  
  */  
}
```

or:

```
p {  
  color: purple;  
  /* font-size: larger; */  
}
```

Written like this, the `font-size: larger` declaration will be ignored by the browser. Be careful to open and close the comments properly, otherwise the browser may be not able to interpret the rules.

Comments are also useful to write reminders about the rules:

```
/* Texts inside <p> are purple and larger */
p {
  color: purple;
  font-size: larger;
}
```

Reminders like the one in the example are expendable in stylesheets containing a small number of rules, but they are essential to help navigate stylesheets with hundreds (or more) rules.

Even though the `style` attribute and the `<style>` tag are adequate for testing custom styles and useful for specific situations, they are not commonly used. Instead, CSS rules are usually kept in a separate file that can be referenced from the HTML document.

CSS in External Files

The preferred method to associate CSS definitions with an HTML document is to store the CSS in a separate file. This method offers two main advantages over the previous ones:

- The same styling rules can be shared among distinct documents.
- The CSS file is usually cached by the browser, improving future loading times.

CSS files have the `.css` extension and, like HTML files, they can be edited by any plain text editor. Unlike HTML files, CSS files have no top level structure built with hierarchical tags such as `<head>` or `<body>`. Rather, the CSS file is just a list of rules processed in sequential order by the browser. The same rules written inside a `<style>` tag could instead go in a CSS file.

The association between the HTML document and the CSS rules stored in a file is defined in the HTML document only. To the CSS file, it does not matter whether elements matching its rules exist, so there is no need to enumerate in the CSS file the HTML documents it is linked to. On the HTML side, every linked stylesheet will be applied to the document, just as if the rules were written in a `<style>` tag.

The `<link>` HTML tag defines an external stylesheet to be used in the current document and should go in the `head` section of the HTML document:

```
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <link href="style.css" rel="stylesheet">

</head>
```

Now you can place the rule for the `p` element we used before into the `style.css` file, and the results seen by the visitor to the web page will be the same. If the CSS file is not in the same directory as the HTML document, specify its relative or full path in the `href` attribute. The `<link>` tag can refer to different types of external resources, so it is important to establish what relationship the external resource has with the current document. For external CSS files, the relationship is defined in `rel="stylesheet"`.

The `media` attribute can be used in the same way as for the `<style>` tag: it indicates the media, such as computer screens or printing, to which the rules in the external file should apply.

CSS can completely change an element, but it is still important to use the appropriate element for the components of the page. Otherwise, assistive technologies such as screen-readers may not be able to identify the correct sections of the page.

Guided Exercises

1. What methods can be used to change the appearance of HTML elements using CSS?

2. Why is it not recommended to use the `style` attribute of the `<p>` tag if there are sibling paragraphs that should look the same?

3. What is the default placement policy for placing a `div` element?

4. What attribute of the `<link>` tag indicates the location of an external CSS file?

5. What is the correct section to insert the `link` element inside an HTML document?

Explorational Exercises

1. Why is it not recommended to use a `<div>` tag to identify a word in an ordinary sentence?

2. What happens if you start a comment with `/*` in the middle of a CSS file, but forget to close it with `*/`?

3. Write a CSS rule to draw an underline in all `em` elements of the document.

4. How can you indicate that a stylesheet from a `<style>` or `<link>` tag should be used only by speech synthesizers?

Summary

This lesson covers the basic concepts of CSS and how to integrate it with HTML. Rules written in CSS stylesheets are the standard method to change the appearance of HTML documents. CSS allows us to keep the semantic content separate from the presentation policies. This lesson goes through the following concepts and procedures:

- How to change CSS properties using the `style` attribute.
- The basic CSS rule syntax.
- Using the `<style>` tag to embed CSS rules in the document.
- Using the `<link>` tag to incorporate external stylesheets to the document.

Answers to Guided Exercises

1. What methods can be used to change the appearance of HTML elements using CSS?

Three basic methods: Write it directly in the element's tag, in a separate section of the page's source code, or in an external file to be referenced by the HTML page.

2. Why is it not recommended to use the `style` attribute of the `<p>` tag if there are sibling paragraphs that should look the same?

The CSS declaration will need to be replicated in the other `<p>` tags, which is time consuming, increases the file size, and it is prone to errors.

3. What is the default placement policy for placing a `div` element?

The `div` element is treated as a block element by default, so it will occupy all the horizontal space of its parent element and its height will depend on its contents.

4. What attribute of the `<link>` tag indicates the location of an external CSS file?

The `href` attribute.

5. What is the correct section to insert the `link` element inside an HTML document?

The `link` element should be in the `head` section of the HTML document.

Answers to Explorational Exercises

1. Why it is not recommended to use a `<div>` tag to identify a word in an ordinary sentence?

The `<div>` tag provides a semantic separation between two distinct sections of the document and interferes with accessibility tools when it is used for inline text elements.

2. What happens if you start a comment with `/*` in the middle of a CSS file, but forget to close it with `*/`?

All the rules after the comment will be ignored by the browser.

3. Write a CSS rule to draw an underline in all `em` elements of the document.

```
em { text-decoration: underline }
```

or

```
em { text-decoration-line: underline }
```

4. How can you indicate that a stylesheet from a `<style>` or `<link>` tag should be used only by speech synthesizers?

The value of its `media` attribute must be `speech`.



033.2 CSS Selectors and Style Application

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 033.2

Weight

3

Key knowledge areas

- Use selectors to apply CSS rules to elements
- Understand CSS pseudo-classes
- Understand rule order and precedence in CSS
- Understand inheritance in CSS

Partial list of the used files, terms and utilities

- `element; .class; #id`
- `a, b; a.class; a b;`
- `:hover, :focus`
- `!important`



033.2 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	033 CSS Content Styling
Objective:	033.2 CSS Selectors and Style Application
Lesson:	1 of 1

Introduction

When writing a CSS rule, we must tell the browser to which elements the rule applies. We do so by specifying a *selector*: a pattern that can match an element or group of elements. Selectors come in many different forms, which can be based on the element's name, its attributes, its relative placement in the document structure, or a combination of these characteristics.

Page-Wide Styles

One of the advantages of using CSS is that you do not need to write individual rules to elements sharing the same style. An asterisk applies the rule to all elements in the web page, as shown in the following example:

```
* {  
  color: purple;  
  font-size: large;  
}
```

The `*` selector is not the only method to apply a style rule to all elements in the page. A selector that

simply matches an element by its tag name is called a *type selector*, so any HTML tag name such as `body`, `p`, `table`, `em`, etc., can be used as selectors. In CSS, the parent's style is *inherited* by its children elements. So, in practice, using the `*` selector has the same effect as applying a rule to the `body` element:

```
body {  
  color: purple;  
  font-size: large;  
}
```

Furthermore, the cascading feature of CSS allows you to fine tune the inherited properties of an element. You can write a general CSS rule that applies to all elements in the page, and then write rules for specific elements or sets of elements.

If the same element matches two or more conflicting rules, the browser applies the rule from the most specific selector. Take the following CSS rules as an example:

```
body {  
  color: purple;  
  font-size: large;  
}  
  
li {  
  font-size: small;  
}
```

The browser will apply the `color: purple` and `font-size: large` styles to all elements inside the `body` element. However, if there are `li` elements in the page, the browser will replace the `font-size: large` style by the `font-size: small` style, because the `li` selector has a stronger relationship with the `li` element than the `body` selector does.

CSS does not limit the number of equivalent selectors in the same stylesheet, so you can have two or more rules using the same selector:

```
li {  
    font-size: small;  
}
```

```
li {  
    font-size: x-small;  
}
```

In this case, both rules are equally specific to the `li` element. The browser cannot apply conflicting rules, so it will choose the rule that comes later in the CSS file. To avoid confusion, the recommendation is to group together all properties that use the same selector.

The order in which the rules appear in the stylesheet affect how they are applied in the document, but you can override this behavior by using an `important` rule. If, for any reason, you want to keep the two separate `li` rules, but force the application of the first one instead of the second one, mark the first rule as important:

```
li {  
    font-size: small !important;  
}  
  
li {  
    font-size: x-small;  
}
```

Rules marked with `!important` should be used with caution, because they break the natural stylesheet cascading and make it harder to find and correct problems within the CSS file.

Restrictive Selectors

We saw that we can change certain inherited properties by using selectors matching specific tags. However, we usually need to use distinct styles for elements of the same type.

Attributes of HTML tags can be incorporated into selectors to restrict the set of elements they refer to. Suppose the HTML page you are working on has two types of unordered lists (``): one is used at the top of the page as a menu to the sections of the website and the other type is used for conventional lists in the text body:

```
<!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>
```

By default, each list item has a black circle to its left. You may want to remove the circles from the top menu list while leaving the circles in the other list. However, you cannot simply use the `li` selector because doing so will also remove the circles in the list inside the text body section. You will need a way to tell the browser to modify only the list items used in one list, but not the other.

There are several ways to write selectors matching specific elements in the page. As mentioned earlier, we will first see how to use the elements' attributes to do so. For this example in particular, we can use the `id` attribute to specify the top list only.

The `id` attribute assigns a unique identifier to the corresponding element, which we can use as the selector part of the CSS rule. Before writing the CSS rule, edit the sample HTML file and add `id="topmenu"` to the `ul` element used for the top menu:

```
<ul id="topmenu">
  <li>Home</li>
  <li>Articles</li>
  <li>About</li>
</ul>
```

There is already a `link` element in the `head` section of the HTML document pointing to the `style.css` stylesheet file in the same folder, so we can add the following CSS rules to it:

```
ul#topmenu {
  list-style-type: none
}
```

The hash character is used in a selector, following an element, to designate an ID (without spaces separating them). The tag name to the left of the hash is optional, as there will be no other element with the same ID. Therefore, the selector could be written just as `#topmenu`.

Even though the `list-style-type` property is not a direct property of the `ul` element, CSS properties of the parent element are inherited by its children, so the style assigned to the `ul` element will be inherited by its child `li` elements.

Not all elements have an ID by which they can be selected. Indeed, only a few key layout elements in a page are expected to have IDs. Take the lists of planets used in the sample code, for instance.

Although it is possible to assign unique IDs for each individual repeated element like these, this method is not practical for longer pages with lots of contents. Rather, we can use the parent `div` element's ID as the selector to change the properties of its inner elements.

However, the `div` element is not directly related to HTML lists, so adding the `list-style-type` property to it will have no effect on its children. Thus, to change the black circle in the lists inside the content `div` to a hollow circle, we should use a *descendant* selector:

```
#topmenu {
  list-style-type: none
}

#content ul {
  list-style-type: circle
}
```

The `#content ul` selector is called a descendant selector because it matches only the `ul` elements that are children of the element whose ID is `content`. We can use as many levels of descendancy as necessary. For instance, using `#content ul li` would match only the `li` elements that are descendants of `ul` elements that are descendants of the element whose ID is `content`. But in this example, the longer selector will have the same effect as using `#content ul`, because the `li` elements inherit the CSS properties set to their parent `ul`. Descendant selectors are an essential technique as the page layout grows in complexity.

Let's say that now you want to change the `font-style` property of the list items in the `topmenu` list and in the list in the *footer* `div` to make them look oblique. You can't simply write a CSS rule using `ul` as the selector, because it will also change the list items in the `content div`. So far, we have changed CSS properties using one selector at a time, and this method can also be used for this task:

```
#topmenu {
  font-style: oblique
}

#footer ul {
  font-style: oblique
}
```

Separate selectors are not the only way to do it, though. CSS allow us to group together selectors that share one or more styles, using a list of selectors separated by commas:

```
#topmenu, #footer ul {
  font-style: oblique
}
```

Grouping selectors eliminates the extra work of writing duplicate styles. Furthermore, you may want to change the property again in the future and may not remember to change it in all the different places.

Classes

If you do not want to worry too much about the element hierarchy, you can simply add a `class` to the set of elements you want to customize. Classes are similar to IDs, but instead of identifying only a single element in the page, they can identify a group of elements sharing the same characteristics.

Take the sample HTML page we are working on, for instance. It's unlikely that in real-world pages we will find structures simple as that, so it would be more practical to select an element using classes only, or a combination of classes and descendancy. To apply the `font-style: oblique` property to the menu lists using classes, first we need to add the `class` property to the elements in the HTML file. We'll do it first in the top menu:

```
<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>
```

And then in the footer's menu:

```
<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 --&gt;</pre>

```

With that, we can replace the selector group `#topmenu, #footer ul` by the class-based selector `.menu`:

```
.menu {
  font-style: oblique
}
```

As with the ID-based selectors, adding the tag name to the left of the dot in class-based selectors is optional. However, unlike IDs, the same class is supposed to be used in more than one element and they do not even need to be of the same type. Therefore, if the `menu` class is shared among different element types, using the `ul.menu` selector would match only the `ul` elements having the `menu` class. Instead, using `.menu` as the selector will match any element having the `menu` class, regardless of its type.

Furthermore, elements can be associated to more than one class. We could differentiate between the top and the bottom menu by adding an extra class to each one of them:

```
<ul id="topmenu" class="menu top">
```

And in the footer's menu:

```
<ul class="menu footer">
```

When the `class` attribute has more than one class, they must be separated by spaces. Now, in addition to the CSS rule shared between elements of the `menu` class, we can address the top and footer menu using their corresponding classes:

```
.menu {
  font-style: oblique
}

.menu.top {
  font-size: large
}

.menu.footer {
  font-size: small
}
```

Be aware that writing `.menu.top` differs from `.menu .top` (with a space between the words). The first selector will match elements that have both `menu` and `top` classes, whereas the second will match elements that have the `top` class and a parent element with the `menu` class.

Special Selectors

CSS selectors can also match dynamic states of elements. These selectors are particularly useful for interactive elements, such as hyperlinks. You may want the appearance of hyperlinks when the mouse pointer moves over them, to draw the visitor's attention.

Back to our sample page, we could remove the underlines from the links in the top menu and show a line only when the mouse pointer moves over the corresponding link. To do this, we first write a rule to remove the underline from the links in the top menu:

```
.menu.top a {  
    text-decoration: none  
}
```

Then we use the `:hover` pseudo-class on those same elements to create a CSS rule that will apply only when the mouse pointer is over the corresponding element:

```
.menu.top a:hover {  
    text-decoration: underline  
}
```

The `:hover` pseudo-class selector accepts all the CSS properties of conventional CSS rules. Other pseudo-classes include `:visited`, which matches hyperlinks that have already been visited, and `:focus`, which matches form elements that have received focus.

Guided Exercises

1. Suppose an HTML page has a stylesheet associated to it, containing the two following rules:

```
p {  
  color: green;  
}
```

```
p {  
  color: red;  
}
```

What color will the browser apply to the text inside the `p` elements?

2. What is the difference between using the ID selector `div#main` and `#main`?

3. Which selector matches all `p` elements used inside a `div` with ID attribute `#main`?

4. What is the difference between using the class selector `p.highlight` and `.highlight`?

Explorational Exercises

1. Write a single CSS rule that changes the `font-style` property to `oblique`. The rule must match only `a` elements that are inside `<div id="sidebar"></div>` or `<ul class="links">`.

2. Suppose you want to change the style of the elements whose `class` attribute is set to `article reference`, as in `<p class="article reference">`. However, the `.article .reference` selector does not appear to alter their appearance. Why is the selector not matching the elements as expected?

3. Write a CSS rule to change the `color` property of all visited links in the page to `red`.

Summary

This lesson covers how to use CSS selectors and how the browser decides what styles to apply to each element. Being separate from the HTML markup, CSS provides many selectors to match individual elements or groups of elements in the page. The lesson goes through the following concepts and procedures:

- Page wide styles and style inheritance.
- Styling elements by type.
- Using the element ID and class as the selector.
- Compound selectors.
- Using pseudo-classes to style elements dynamically.

Answers to Guided Exercises

1. Suppose an HTML page has a stylesheet associated to it, containing the two following rules:

```
p {  
  color: green;  
}  
  
p {  
  color: red;  
}
```

What color will the browser apply to the text inside the `p` elements?

The color `red`. When two or more equivalent selectors have conflicting properties, the browser will choose the last one.

2. What is the difference between using the ID selector `div#main` and `#main`?

The selector `div#main` matches a `div` element having the ID `main`, whereas the `#main` selector matches the element having the ID `main`, regardless of its type.

3. Which selector matches all `p` elements used inside a `div` with ID attribute `#main`?

The selector `#main p` or `div#main p`.

4. What is the difference between using the class selector `p.highlight` and `.highlight`?

The selector `p.highlight` matches only the elements of type `p` having the class `highlight`, whereas the `.highlight` selector matches all elements having the class `highlight`, regardless of their type.

Answers to Explorational Exercises

1. Write a single CSS rule that changes the `font-style` property to `oblique`. The rule must match only `a` elements that are inside `<div id="sidebar"></div>` or `<ul class="links">`.

```
#sidebar a, ul.links a {  
    font-style: oblique  
}
```

2. Suppose you want to change the style of the elements whose `class` attribute is set to `article reference`, as in `<p class="article reference">`. However, the `.article .reference` selector does not appear to alter their appearance. Why is the selector not matching the elements as expected?

The `.article .reference` selector will match the elements having the class `reference` that are descendants of elements having the class `article`. To match elements having both `article` and `reference` classes, the selector should be `.article.reference` (without the space between them).

3. Write a CSS rule to change the `color` property of all visited links in the page to `red`.

```
a:visited {  
    color: red;  
}
```



033.3 CSS Styling

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 033.3

Weight

2

Key knowledge areas

- Understand fundamental CSS properties
- Understand units commonly used in CSS

Partial list of the used files, terms and utilities

- px, %, em, rem, vw, vh
- color, background, background-*, font, font-*, text-*, list-style, line-height



**Linux
Professional
Institute**

033.3 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	033 CSS Content Styling
Objective:	033.3 CSS Styling
Lesson:	1 of 1

Introduction

CSS provides hundreds of properties that can be used to modify the appearance of HTML elements. Only experienced designers manage to remember most of them. Nevertheless, it is practical to know the basic properties that are applicable to any element, as well as some element-specific properties. This chapter covers some popular properties you are likely to use.

CSS Common Properties and Values

Many CSS properties have the same value type. Colors, for example, have the same numerical format whether you are changing the font color or the background color. Similarly, the units available to change the size of the font are also used to change the thickness of a border. However, the format of the value is not always unique. Colors, for example, can be entered in various different formats, as we will see next.

Colors

Changing the color of an element is probably one of the first things designers learn to do with CSS. You can change the color of the text, the background color, the color of the border of elements, and

more.

The value of a color in CSS can be written as a *color keyword* (i.e. a color name) or as a numerical value listing each color component. All common color names, such as `black`, `white`, `red`, `purple`, `green`, `yellow`, and `blue` are accepted as color keywords. The entire list of color keywords accepted by CSS is available at a [W3C web page](#). But to have finer control over the color, you can use the numerical value.

Color Keywords

We'll use the color keyword first, because it's simpler. The `color` property changes the color of the text in the corresponding element. So to put all the text on the page in purple, you could write the following CSS rule:

```
* {  
  color: purple;  
}
```

Numerical Color Values

Although intuitive, color keywords do not offer the complete range of possible colors in modern displays. Web designers usually develop a color palette that employs custom colors, by assigning specific values to the red, green, and blue components.

Each color component is an eight-bit binary number, ranging from 0 to 255. All three components must be specified in the color mixture, and their order is always red, green, blue. Therefore, to change the color for all the text in the page to red using RGB notation, use `rgb(255, 0, 0)`:

```
* {  
  color: rgb(255, 0, 0);  
}
```

A component set to 0 means that the corresponding basic color is not used in the color mixture. Percentages can also be used instead of numbers:

```
* {  
  color: rgb(100%, 0%, 0%);  
}
```

The RGB notation is rarely seen if you use a drawing application to create layouts or just to pick its colors. Rather, it is more common to see colors in CSS expressed as *hexadecimal* values. The color components in hexadecimal also range from 0 to 255, but in a more succinct way, starting with a hash symbol # and using a fixed two-digit length for all components. The minimal value 0 is 00 and the maximum value 255 is FF, so the color red is FF0000.

TIP

If the digits in each component of a hexadecimal color repeat, the second digit can be omitted. The color red, for example, can be written with a single digit for each component: #F00.

The following list shows the RGB and hexadecimal notation for some basic colors:

Color Keyword	RGB Notation	Hexadecimal Value
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

Color keywords and the letters in hexadecimal color values are case-insensitive.

Color Opacity

In addition to opaque colors, it is possible to fill page elements with semi-transparent colors. The opacity of a color can be set using a fourth component in the color value. Unlike the other color components, where the values are integer numbers ranging from 0 to 255, the opacity is a fraction ranging from 0 to 1.

The lowest value, 0, results in a completely transparent color, making it undistinguishable from any other fully transparent color. The highest value, 1, results in a fully opaque color, which is the same as the original color with no transparency at all.

When you use the RGB notation, specify colors with an opacity component through the `rgba` prefix instead of `rgb`. Thus, to make the color red half-transparent, use `rgba(255, 0, 0, 0.5)`. The `a` character stands for *alpha channel*, a term commonly used to specify the opacity component in digital graphics jargon.

The opacity can also be set in the hexadecimal notation. Here, like the other color components, the opacity ranges from `00` to `FF`. Therefore, to make the color `red` half-transparent using hexadecimal notation, use `#FF000080`.

Background

The background color of single elements or of the entire page is set with the `background-color` property. It takes the same values as the `color` property, either as keywords or using the RGB/hexadecimal notation.

The background of HTML elements is not restricted to colors, though. With the `background-image` property it is possible to use an image as the background. The accepted image formats are all the conventional ones accepted by the browser, such as JPEG and PNG.

The path to the image should be specified using an `url()` designator. If the image you want to use is in the same folder as the HTML file, it is enough to use only its filename:

```
body {  
    background-image: url("background.jpg");  
}
```

In this example, the `background.jpg` image file will be used as the background image for the entire body of the page. By default, the background image is repeated if its size does not cover the entire page, starting from the top-left corner of the area corresponding to the rule's selector. This behavior can be modified with the `background-repeat` property. If you want the background image to be placed in the element's area without repeating it, use the `no-repeat` value:

```
body {  
    background-image: url("background.jpg");  
    background-repeat: no-repeat;  
}
```

You can also make the image repeat only in the horizontal direction (`background-repeat: repeat-x`) or only in the vertical direction (`background-repeat: repeat-y`).

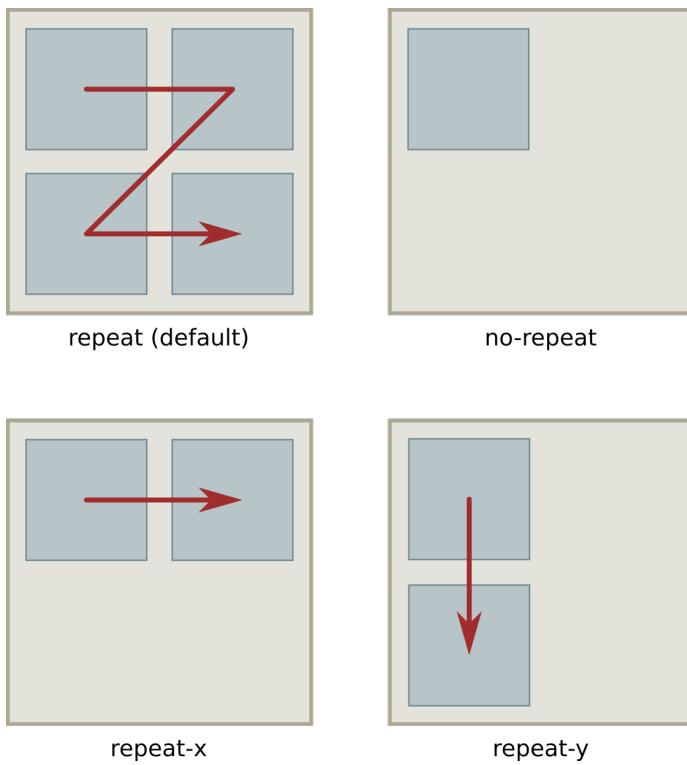


Figure 34. Background placement using the `background-repeat` property.

TIP

Two or more CSS properties can be combined in a single property, called the `background shorthand` property. The `background-image` and `background-repeat` properties, for example, can be combined in a single `background` property with `background: no-repeat url("background.jpg")`.

A background image can also be placed in a specific position in the element's area using the `background-position` property. The five basic positions are `top`, `bottom`, `left`, `right` and `center`, but the top-left position of the image can also be adjusted with percents:

```
body {
  background-image: url("background.jpg");
  background-repeat: no-repeat;
  background-position: 30% 10%;
}
```

The percent for each position is relative to the corresponding size of the element. In the example, the background image left side will be at 30% of the body's width (usually the body is the entire visible document) and the top side of the image will be at 10% of the body's height.

Borders

Changing an element's border is also a common layout customization made with CSS. The border refers to the line forming a rectangle around the element and has three basic properties: `color`, `style` and `width`.

The color of the border, defined with `border-color`, follows the same format we saw for the other color properties.

Borders can be traced in a style other than a solid line. You could, for example, use dashes for the border with the property `border-style: dashed`. The other possible style values are:

dotted

A sequence of rounded dots

double

Two straight lines

groove

A line with a carved appearance

ridge

A line with an extruded appearance

inset

An element that appears embedded

outset

An element that appears embossed

The `border-width` property set the thickness of the border. Its value can be a keyword (`thin`, `medium` or `thick`) or a specific numerical value. If you prefer to use a numerical value, you will also need to specify its corresponding unit. This is described next.

Unit Values

Sizes and distances in CSS can be defined in various ways. Absolute units are based on a fixed number of screen pixels, so they are not so different from the fixed sizes and dimensions used in printed media. There are also relative units, which are dynamically calculated from some measurement given by the media where the page is being rendered, like the available space or another size written in absolute units.

Absolute Units

Absolute units are equivalent to their physical counterparts, like centimeters or inches. On conventional computer screens, one inch will be 96 pixels wide. The following absolute units are commonly used:

in (inch)

1 in is equivalent to 2.54 cm or 96 px.

cm (centimeter)

1 cm is equivalent to 96 px / 2.54.

mm (millimeter)

1 mm is equivalent to 1 cm / 10.

px (pixel)

1 px is equivalent to 1 / 96th of an inch.

pt (point)

1pt is equivalent to 1 / 72th of an inch.

Keep in mind that the ratio of pixels to inch may vary. In high resolution screens, where pixels are more densely packed, an inch will correspond to more than 96 pixels.

Relative Units

Relative units vary according to other measurements or to the viewport dimensions. The viewport is the area of the document currently visible in its window. In full screen mode, the viewport corresponds to the device screen. The following relative units are commonly used:

%

Percentage—it is relative to the parent element.

em

The size of the font used in the element.

rem

The size of the font used in the root element.

vw

1% of the viewport's width.

vh

1% of the viewport's height.

The advantage of using relative units is that you can create layouts that are adjustable by changing only a few key sizes. For example, you can use the `pt` unit to set the font size in the `body` element and the `rem` unit for the fonts in other elements. Once you change the size of font for the `body`, all the other font sizes will adjust accordingly. Furthermore, using `vw` and `vh` to set the dimensions of page sections makes them adjustable to screens with different sizes.

Fonts and Text Properties

Typography, or the study of font types, is a very broad design subject, and CSS typography does not lag behind. However, there are a few basic font properties that will meet the needs of most users learning CSS.

The `font-family` property sets the name of the font to be used. There is no guarantee that the chosen font will be available in the system where the page will be viewed, so this property may have no effect in the document. Although it is possible to make the browser download and use the specified font file, most web designers are happy using a generic font family in their documents.

The three most common generic font families are `serif`, `sans-serif` and `monospace`. `Serif` is the default font family in most browsers. If you prefer to use `sans-serif` for the entire page, add the following rule to your stylesheet:

```
* {  
  font-family: sans-serif;  
}
```

Optionally, you can first set a specific font family name, followed by the generic family name:

```
* {  
  font-family: "DejaVu Sans", sans-serif;  
}
```

If the device rendering the page has that specific font family, the browser will use it. If not, it will use its default font matching the generic family name. Browsers search for fonts in the order they are specified in the property.

Anyone who has used a word processing application will also be familiar with three other font adjustments: size, weight, and style. These three adjustments have CSS properties counterparts: `font-size`, `font-weight`, and `font-style`.

The `font-size` property accepts keyword sizes such as `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx-large`. These keywords are relative to the default font size used by the browser. The keywords `larger` and `smaller` change the font size relative to the parent's element font size. You can also express the font size with numerical values, including the unit after the value, or with percentages.

If you do not want to change the font size, but the distance between lines, use the `line-height` property. A `line-height` of 1 will make the line height the same size of the font of the element, which can make the text lines too close together. Therefore, a value greater than 1 is more appropriate for texts. Like the `font-size` property, other units can be used together with the numerical value.

The `font-weight` sets the thickness of the font with the familiar keywords `normal` or `bold`. The keywords `lighter` and `bolder` change the font weight of the element relative to the font weight of its parent element.

The `font-style` property can be set to `italic` to select the italic version of the current font family. The `oblique` value selects the oblique version of the font. These two options are almost identical, but the italic version of a font is usually a little more narrow than the oblique version. If neither italic nor oblique versions of the font exist, the font will be artificially tilted by the browser.

There are other properties that change the way text is rendered in the document. You can, for example, add an underline to some parts of the text you want to emphasize. First, use a `` tag to delimit the text:

```
<p>CSS is the <span class="under">proper mechanism</span> to style HTML documents.</p>
```

Then you can use the `.under` selector to change the `text-decoration` property:

```
.under {
  text-decoration: underline;
}
```

By default, all `a` (link) elements are underlined. If you want to remove it, use the `none` value for the `text-decoration` of all `a` elements:

```
a {  
  text-decoration: none;  
}
```

When reviewing content, some authors like to cross out incorrect or outdated parts of the text, so that the reader will know the text has been updated and what has been removed. To do so, use the `line-through` value of the `text-decoration` property:

```
.disregard {  
  text-decoration: line-through;  
}
```

Again, a `` tag can be used to apply the style:

```
<p>Netscape Navigator <span class="disregard">is</span> was one of the most popular  
Web browsers.</p>
```

Other decorations may be specific to an element. The circles in bullet point lists can be customized using the `list-style-type` property. To change them to squares, for example, use `list-style-type: square`. To simply remove them, set the value of `list-style-type` to `none`.

Guided Exercises

1. How could you add half-transparency to the color blue (RGB notation `rgb(0,0,255)`) in order to use it in the CSS `color` property?

2. What color corresponds to the `#000` hexadecimal value?

3. Given that `Times New Roman` is a `serif` font and that it will not be available in all devices, how could you write a CSS rule to request `Times New Roman` while setting the `serif` generic font family as a fallback?

4. How could you use a relative size keyword to set the font size of the element `<p class="disclaimer">` smaller in relation to its parent element?

Explorational Exercises

1. The `background` property is a shorthand to set more than one `background-*` property at once. Rewrite the following CSS rule as a single `background` shorthand property.

```
body {  
  background-image: url("background.jpg");  
  background-repeat: repeat-x;  
}
```

2. Write a CSS rule for the element `<div id="header"></div>` that changes *only* its bottom border width to `4px`.

3. Write a `font-size` property that doubles the font size used in the page's root element.

4. *Double-spacing* is a common text formatting feature in word processors. How could you set a similar format using CSS?

Summary

This lesson covers the application of simple styles to elements of an HTML document. CSS provides hundreds of properties, and most web designers will need to resort to reference manuals to remember them all. Nonetheless, a relatively small set of properties and values are used most of the time, and it is important to know them by heart. The lesson goes through the following concepts and procedures:

- Fundamental CSS properties dealing with colors, backgrounds, and fonts.
- The absolute and relative units CSS can use to set sizes and distances, such as `px`, `em`, `rem`, `vw`, `vh`, etc.

Answers to Guided Exercises

1. How could you add half-transparency to the color blue (RGB notation `rgb(0,0,255)`) in order to use it in the CSS `color` property?

Use the `rgba` prefix and add `0.5` as the fourth value: `rgba(0,0,0,0.5)`.

2. What color corresponds to the `#000` hexadecimal value?

The color `black`. The `#000` hexadecimal value is a shorthand for `#000000`.

3. Given that `Times New Roman` is a `serif` font and that it will not be available in all devices, how could you write a CSS rule to request `Times New Roman` while setting the `serif` generic font family as a fallback?

```
* {  
  font-family: "Times New Roman", serif;  
}
```

4. How could you use a relative size keyword to set the font size of the element `<p class="disclaimer">` smaller in relation to its parent element?

Using the `smaller` keyword:

```
p.disclaimer {  
  font-size: smaller;  
}
```

Answers to Explorational Exercises

1. The `background` property is a shorthand to set more than one `background-*` property at once. Rewrite the following CSS rule as a single `background` shorthand property.

```
body {  
  background-image: url("background.jpg");  
  background-repeat: repeat-x;  
}
```

```
body {  
  background: repeat-x url("background.jpg");  
}
```

2. Write a CSS rule for the element `<div id="header"></div>` that changes *only* its bottom border width to `4px`.

```
#header {  
  border-bottom-width: 4px;  
}
```

3. Write a `font-size` property that doubles the font size used in the page's root element.

The `rem` unit corresponds to the font size used in the root element, so the property should be `font-size: 2rem`.

4. *Double-spacing* is a common text formatting feature in word processors. How could you set a similar format using CSS?

Set the `line-height` property to the value `2em`, because the `em` unit corresponds to the font size of the current element.



033.4 CSS Box Model and Layout

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 033.4

Weight

2

Key knowledge areas

- Define the dimension, position and alignment of elements in a CSS layout
- Specify how text flows around other elements
- Understand the document flow
- Awareness of the CSS grid
- Awareness of responsive web design
- Awareness of CSS media queries

Partial list of the used files, terms and utilities

- `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`



033.4 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	033 CSS Content Styling
Objective:	033.4 CSS Box Model and Layout
Lesson:	1 of 1

Introduction

Each visible element in an HTML document is rendered as a rectangular box. Thus, the term *box model* describes the approach CSS takes to modifying the visual properties of the elements. Like boxes of different sizes, HTML elements can be nested inside *container* elements—usually the `div` element—so they can be segregated in sections.

We can use CSS to modify the position of the boxes, from minor adjustments to drastic changes in the disposition of the elements on the page. Besides the normal flow, the position for each box can be based on the elements around it, either its relationship to its parent container or its relationship to the *viewport*, which is the area of the page visible to the user. No single mechanism meets all possible layout requirements, so you may require a combination of them.

Normal Flow

The default way the browser renders the document tree is called *normal flow*. The rectangles corresponding to the elements are placed more or less in the same order they appear in the document tree, relative to their parent elements. Nevertheless, depending on the element type, the corresponding box may follow distinct positioning rules.

A good way to understand the logic of the normal flow is to make the boxes visible. We can start with a very basic page, having only three separate `div` elements, each one having a paragraph with random text:

```
<!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>premium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->

</body>
</html>
```

Every word is in a `span` element so we can style the words and see how they are treated as boxes as well. To make the boxes visible, we must edit the stylesheet file `style.css` referenced by the HTML document. The following rules will do what we need:

```
* {
  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: #fffff66;
}

p {
  background-color: #fffff66;
}

span {
  background-color: #fffffaa;
}
```

The result appears in [Figure 35](#).

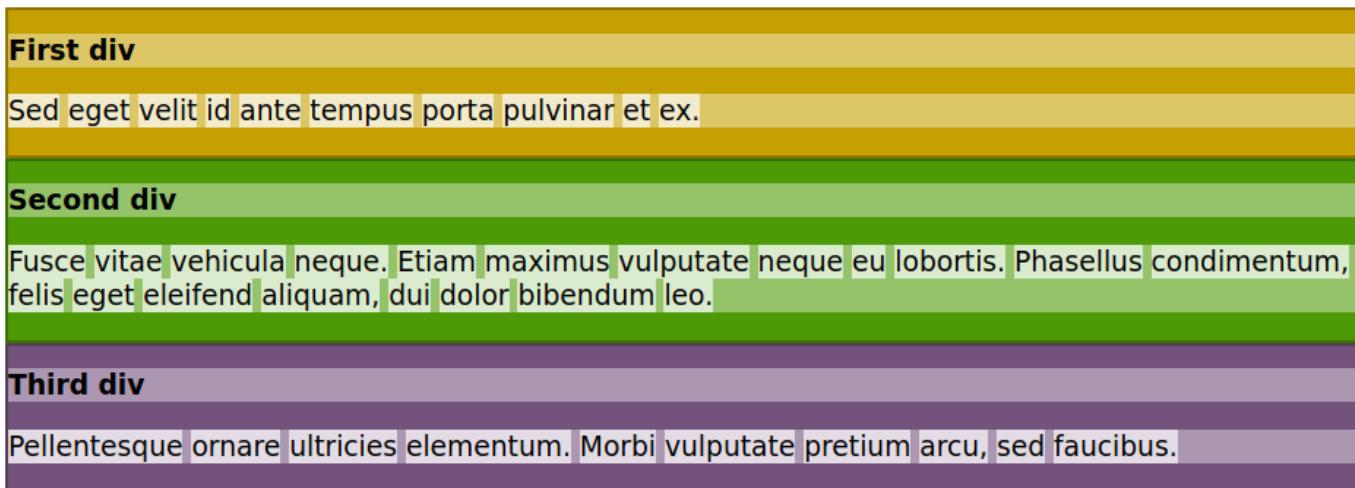


Figure 35. The basic element flow is from top to bottom and left to right.

Figure 35 shows that each HTML tag has a corresponding box in the layout. The `div`, `h2`, and `p` elements extend to the width of their parent element. For instance, the parent element of the `div` elements is the `body` element, so they extend to the width of the `body`, while the parent of each `h2` and `p` element is its corresponding `div`. The boxes that extend to the width of their parent element are called *block* elements. Some of the most common HTML tags rendered as blocks are `h1`, `h2`, `h3`, `p`, `ul`, `ol`, `table`, `li`, `div`, `section`, `form`, and `aside`. Sibling block elements—block elements sharing the same immediate parent element—are stacked inside their parent from top to bottom.

NOTE

Some block elements are not intended to be used as containers for other block elements. It is possible, for example, to insert a block element inside a `h1` or `p` element, but it is not considered best practice. Rather, the designer should use an appropriate tag as a container. Common container tags are `div`, `section`, and `aside`.

Besides text itself, elements such as `h1`, `p`, and `li` expect only *inline* elements as children. Like most Western scripts, inline elements follow the left-to-right flow of text. When there is no remaining room in the right side, the flow of inline elements continues in the next line, just like text. Some common HTML tags treated as inline boxes are `span`, `a`, `em`, `strong`, `img`, `input`, and `label`.

In our sample HTML page, every word inside the paragraphs was surrounded by a `span` tag, so they could be highlighted with a corresponding CSS rule. As shown in the image, each `span` element is placed horizontally, from left to right, until there is no more room in the parent element.

The height of the element depends on its contents, so the browser adjusts the height of a container element to accommodate its nested block elements or lines of inline elements. However, some CSS properties affect a box's shape, its position, and the placement of its inner elements.

The `margin` and `padding` properties affect all box types. If you do not set these properties explicitly, the browser sets some of them using standard values. As seen in [Figure 35](#), the `h2` and `p` elements were rendered with a gap between them. These gaps are the top and bottom margins the browser adds to these elements by default. We can remove them by modifying the CSS rules for the `h2` and `p` selectors:

```
h2 {
  background-color: #fffff66;
  margin: 0;
}

p {
  background-color: #fffff66;
  margin: 0;
}
```

The result appears in [Figure 36](#).

Figure 36. The margin property can change or remove margins from elements.

The `body` element also, by default, has a small margin that creates a gap surrounding. This gap can also be removed using the `margin` property.

While the `margin` property defines the gap between the element and its surroundings, the `padding` property of the element defines the internal gap between the container's limits and its child elements. Consider the `h2` and `p` elements inside each `div` in the sample code, for example. We could use their `margin` property to create a gap to the borders of the corresponding `div`, but it is simpler to change the `padding` property of the container:

```
#second {
  background-color: #4e9a06ff;
  padding: 1em;
}
```

Only the rule for the second `div` was modified, so the results (Figure 37) show the difference between the second `div` and the other `div` containers.

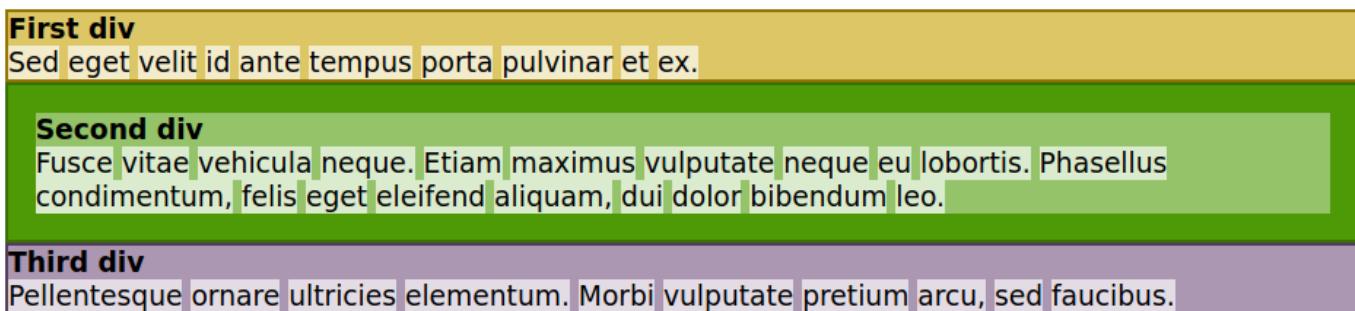


Figure 37. Different `div` containers can have different paddings.

The `margin` property is a shorthand for four properties controlling the four sides of the box: `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. When `margin` is assigned a single value, as in the examples so far, all four margins of the box use it. When two values are written, the first defines the top and bottom margins, while the second defines the right and left margins. Using `margin: 1em 2em`, for example, defines a gap of 1 em for the top and bottom margins and a gap of 2 em for the right and left margins. Writing four values sets the margins for the four sides in a clockwise direction, beginning at the top. The different values in the shorthand property are not required to use the same units.

The `padding` property is a shorthand as well, following the same principles as the `margin` property.

In their default behaviour, block elements stretch to fit the available width. But this is not mandatory. The `width` property can set a fixed horizontal size to the box:

```
#first {  
  background-color: #c4a000ff;  
  width: 6em;  
}
```

The addition of `width: 6em` to the CSS rule shrinks the first `div` horizontally, leaving a blank space to its right side (Figure 38).

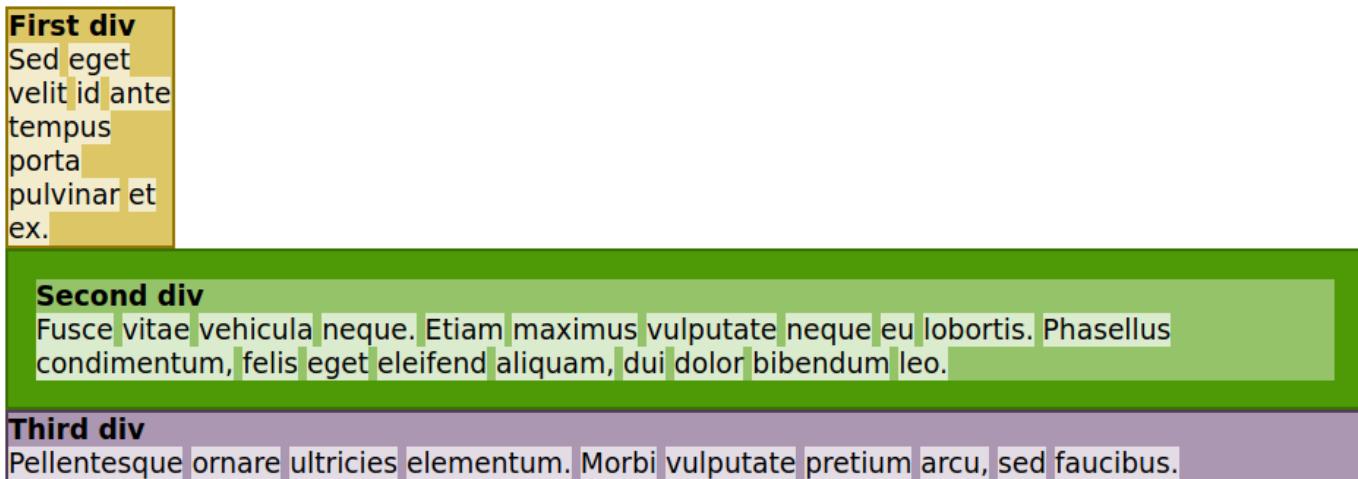


Figure 38. The `width` property changes the horizontal width of the first div.

Instead of leaving the first `div` aligned to the left, we may want to center it. Centering a box is equivalent to setting margins of the same size on both sides, so we can use the `margin` property to center it. The size of the available space may vary, so we use the `auto` value to the left and right margins:

```
#first {  
  background-color: #c4a000ff;  
  width: 6em;  
  margin: 0 auto;  
}
```

The left and right margins are automatically computed by the browser and the box will be centered ([Figure 39](#)).

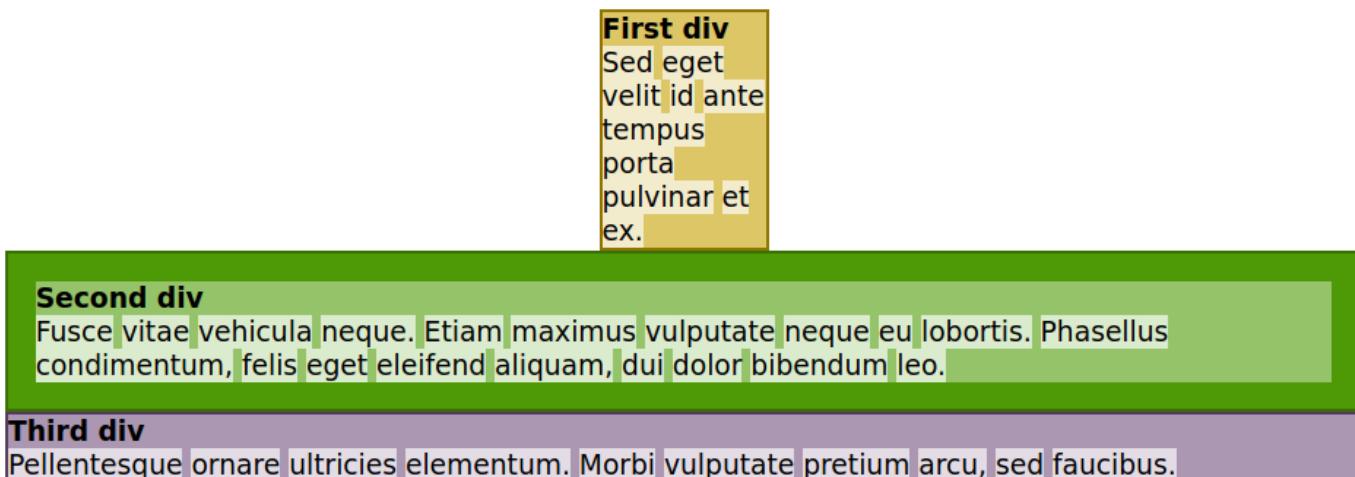


Figure 39. The `margin` property is used to center the first div.

As shown, making a block element narrower does not make the remaining space available to the next element. The natural flow is still preserved, as if the narrower element still occupies all the available width.

Customizing Normal Flow

Normal flow is simple and sequential. CSS also lets you break normal flow and position elements in very specific ways, even overriding the scrolling of the page if you want. We'll look at several ways to control the positioning of elements in this section.

Floating Elements

It is possible to make sibling block elements share the same horizontal space. One way to do so is through the `float` property, which removes the element from the normal flow. As its name suggests, the `float` property makes the box float over the block elements coming after, so they will be rendered as if they were under the floated box. To make the first `div` float to the right, add `float: right` to the corresponding CSS rule:

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  float: right;
}
```

The automatic margins are ignored in a floated box, so the `margin` property can be removed. [Figure 40](#) shows the result of the floating the first `div` to the right.

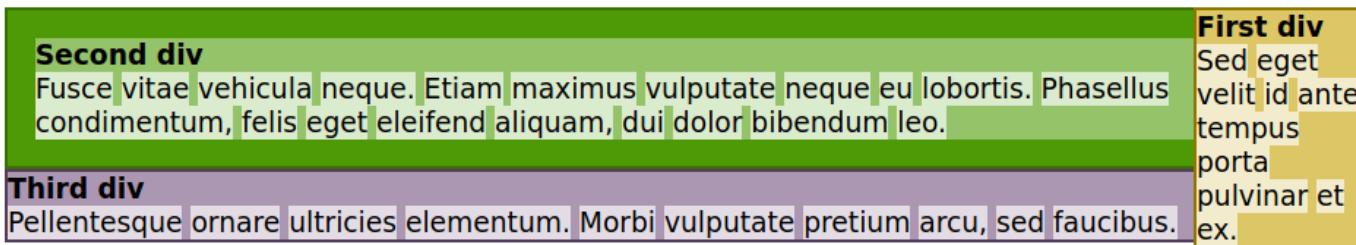


Figure 40. The first div is floating and is not part of the normal flow.

By default, all block elements coming after the floated element will go under it. Therefore, given enough height, the floated box will cover all remaining block elements.

Although a floating element goes above other block elements, the inline contents inside the floating element's container wrap around the floating element. The inspiration for this comes from magazine and newspaper layouts, which often wrap text around an image, for example.

The previous image shows how the first div covers the second div and part of the third div. Assume we want the first div to float over the second div, but not the third. The solution is to include the clear property in the CSS rule corresponding to the third div:

```
#third {
  background-color: #5c3566da;
  clear: right;
}
```

Setting the clear property to right makes the corresponding element skip any previous elements floated to the right, resuming normal flow (Figure 41).

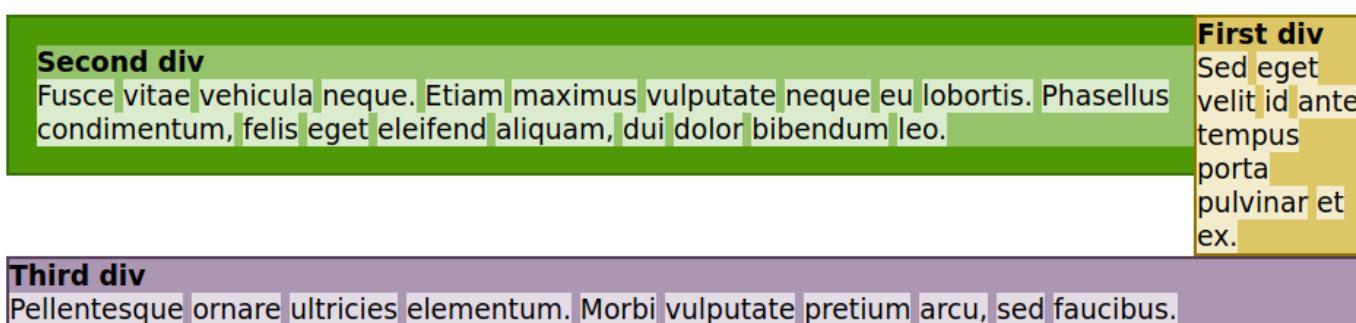


Figure 41. The clear property returns to normal flow.

Likewise, if a previous element floated to the left, you can use clear: left to resume normal flow. When you have to skip floated elements on both the left and the right, use clear: both.

Positioning Boxes

In normal flow, each box goes after the boxes coming before it in the document tree. The previous sibling elements “push” the elements coming after them, moving them to the right and downwards inside their parent element. The parent element may have its own siblings doing the same thing to it. It is like placing tiles side by side in a wall, beginning at the top.

This method of positioning the boxes is called *static*, and is the default value for the CSS `position` property. Other than defining margins and padding, there is no way to reposition a static box in the page.

Like the tiles in the wall analogy, static placement is not mandatory. As with tiles, you can place the boxes anywhere you want, even covering other boxes. To do so, assign the `position` property to one of the following values:

relative

The element follows the normal flow of the document, but it can use the `top`, `right`, `bottom`, and `left` properties to set offsets relative to its original static position. The offsets can also be negative. The other elements remain in their original places, as if the relative element is still static.

absolute

The element ignores the normal flow of the other elements and positions itself on the page by the `top`, `right`, `bottom`, and `left` properties. Their values are relative to the document’s body or to a non-static parent container.

fixed

The element ignores the normal flow of the other elements and positions itself by the `top`, `right`, `bottom`, and `left` properties. Their values are relative to the viewport (i.e., the screen area where the document is shown). Fixed elements do not move as the visitor scrolls through the document, but resemble a sticker fixed on the screen.

sticky

The element follows the normal flow of the document. However, instead of going off the viewport when the document scrolls, it will stop at the position set by the `top`, `right`, `bottom`, and `left` properties. If the `top` value is `10px`, for example, the element will stop scrolling under the top part of the viewport when it reaches 10 pixels from the top limit of the viewport. When that happens, the rest of the page continues to scroll, but the sticky element behaves like a fixed element in that position. It will go back to its original position when the document scroll back to its position in the viewport. Sticky elements are commonly used nowadays to create top menus that always be visible.

Positions that can use the `top`, `right`, `bottom`, and `left` properties are not required to use them all. If you set both the `top` and `height` properties of an absolute element, for example, the browser implicitly calculates its `bottom` property (`top + height = bottom`).

The `display` Property

If the order given by the normal flow is not an issue in your design, but you want to change how the boxes align themselves in the page, modify the `display` property of the element. The `display` property can even make the element completely disappear from the rendered document, by setting `display: none`. This is useful when you want to show the element later using JavaScript.

The `display` property can also, for example, make a block element behave like an inline element (`display: inline`). Doing so is not considered good practice, though. Better methods exist to place container elements side by side, such as the *flexbox model*.

The flexbox model was invented to overcome the limitations of floats and to eliminate the inappropriate use of tables to structure the page layout. When you set the `display` property of a container element to `flex` to turn it into a flexbox container, its immediate children will behave more or less like cells in a table row.

TIP

If you want even more control over the placement of the elements on the page, take a look at the *CSS grid* feature. The grid is a powerful system based on rows and columns to create elaborate layouts.

To test the flex display, add a new `div` element to the example page and make it the container for the three existing `div` elements:

```
<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 -->
```

Add the following CSS rule to the stylesheet to turn the container `div` into a flexbox container:

```
#container {
  display: flex;
}
```

The result is the three inner `div` elements rendered side by side (Figure 42).

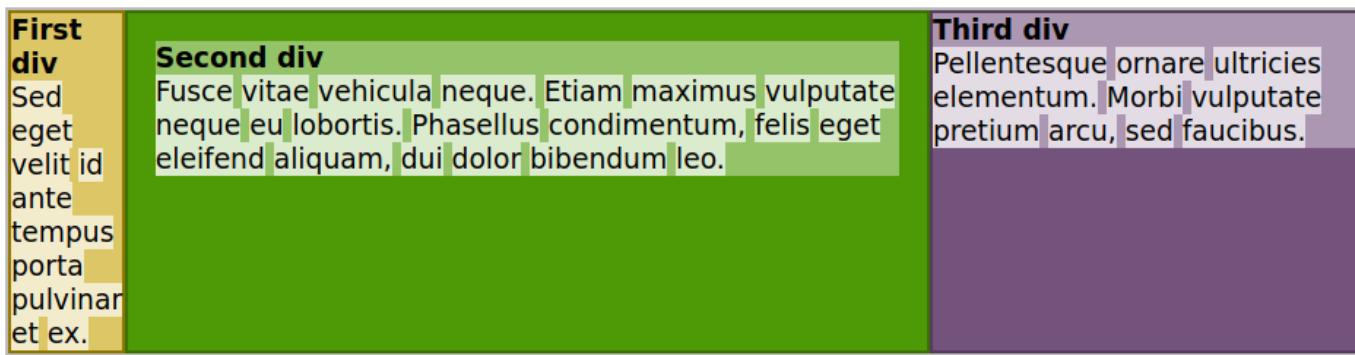


Figure 42. The flexbox model creates a grid.

Using the value `inline-flex` instead of `flex` has basically the same result, but makes the children behave more like inline elements.

Responsive Design

We know that CSS provides properties that adjust the sizes of elements and fonts relative to the available screen area. However, you may want to go further and use a different design for different devices: for example, desktop systems versus devices with screens dimensions under a certain size. This approach is called *responsive web design*, and CSS provides methods called *media queries* to make it possible.

In the previous example, we modified the page layout to place the `div` elements side by side in columns. That layout is suitable for larger screens, but it will be too cluttered in smaller screens. To solve this problem, we can add a media query to the stylesheet matching only screens with at least 600px in width:

```
@media (min-width: 600px){
  #container {
    display: flex;
  }
}
```

The CSS rules inside the `@media` directive will be used only if the criteria in parenthesis is satisfied. In this example, if the viewport width is less than 600px, the rule will not be applied to the container `div` and its children will be rendered as conventional `div` elements. The browser re-evaluates the media queries every time the viewport dimension changes, so the layout can be changed in real time while resizing the browser window or rotating the smartphone.

Guided Exercises

1. If the `position` property is not modified, what positioning method will be used by the browser?

2. How can you make sure an element's box will be rendered after any previously floated elements?

3. How can you use the `margin` shorthand property to set the top/bottom margins to `4px` and the right/left margins to `6em`?

4. How can you horizontally center a static container element with fixed width on the page?

Explorational Exercises

1. Write a CSS rule matching the element `<div class="picture">` so the text inside its following block elements settles toward its right side.

2. How does the `top` property affect a static element relative to its parent element?

3. How does changing the `display` property of an element to `flex` affect its placement in the normal flow?

4. What CSS feature allows you to use a separate set of rules depending on the dimensions of the screen?

Summary

This lesson covers the CSS box model and how we can customize it. In addition to the normal flow of the document, the designer can make use of different positioning mechanisms to implement a custom layout. The lesson goes through the following concepts and procedures:

- The normal flow of the document.
- Adjustments to the margin and padding of an element's box.
- Using the float and clear properties.
- Positioning mechanisms: static, relative, absolute, fixed and sticky.
- Alternative values for the `display` property.
- Responsive design basics.

Answers to Guided Exercises

1. If the `position` property is not modified, what positioning method will be used by the browser?

The `static` method.

2. How can you make sure an element's box will be rendered after any previously floated elements?

The `clear` property of the element should be set to `both`.

3. How can you use the `margin` shorthand property to set the top/bottom margins to `4px` and the right/left margins to `6em`?

It can be either `margin: 4px 6em` or `margin: 4px 6em 4px 6em`.

4. How can you horizontally center a static container element with fixed width on the page?

Using the `auto` value in its `margin-left` and `margin-right` properties.

Answers to Explorational Exercises

1. Write a CSS rule matching the element `<div class="picture">` so the text inside its following block elements settles toward its right side.

```
.picture { float: left; }
```

2. How does the `top` property affect a static element relative to its parent element?

The `top` property does not apply to static elements.

3. How does changing the `display` property of an element to `flex` affect its placement in the normal flow?

The placement of the element itself does not change, but its immediate child elements will be rendered side by side horizontally.

4. What CSS feature allows you to use a separate set of rules depending on the dimensions of the screen?

Media queries allow the browser to verify the viewport dimensions before applying a CSS rule.



Topic 034: JavaScript Programming



034.1 JavaScript Execution and Syntax

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 034.1](#)

Weight

1

Key knowledge areas

- Run JavaScript within an HTML document
- Understand the JavaScript syntax
- Add comments to JavaScript code
- Access the JavaScript console
- Write to the JavaScript console

Partial list of the used files, terms and utilities

- `<script>`, including the `type ('text/javascript')` and `src` attributes
- `;`
- `//, /* */`
- `console.log`



034.1 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	034 JavaScript Programming
Objective:	034.1 JavaScript Execution and Syntax
Lesson:	1 of 1

Introduction

Web pages are developed using three standard technologies: HTML, CSS, and JavaScript. JavaScript is a programming language that lets the browser dynamically update website content. The JavaScript is usually executed by the same browser used to view a webpage. This means that, similar to CSS and HTML, the exact behavior of any code you write might differ between browsers. But most common browsers adhere to the [ECMAScript specification](#). This is a standard that unifies JavaScript usage in the web, and will be the basis for the lesson, together with the [HTML5 specification](#), which specifies how JavaScript needs to be put into a webpage for a browser to execute it.

Running JavaScript in the Browser

To execute JavaScript, the browser needs to obtain the code either directly, as part of the HTML that makes up the web page, or as a URL that indicates a location for a script to be executed.

The following example shows how to include code directly in the HTML file:

```
<html>
  <head>
  </head>
  <body>
    <h1>Website Headline</h1>
    <p>Content</p>

    <script>
      console.log('test');
    </script>

  </body>
</html>
```

The code is wrapped between `<script>` and `</script>` tags. Everything included in these tags will be executed by the browser directly when loading the page.

The position of the `<script>` element within the page dictates when it will be executed. An HTML document is parsed from top to bottom, with the browser deciding when to display elements on the screen. In the example just shown, the `<h1>` and `<p>` tags of the website are parsed, and likely displayed, before the script runs. If the JavaScript code within the `<script>` tag would take very long to execute, the page would still display without any problem. If, however, the script had been placed above the other tags, the web page visitor would have to wait until the script finishes executing before seeing the page. For this reason, `<script>` tags are usually placed in one of two places:

- At the very end of the HTML body, so that the script is the last thing that gets executed. Do this when the code adds something to the page that would not be useful without the content. An example would be adding functionality to a button, as the button has to exist for the functionality to make sense.
- Inside the `<head>` element of the HTML. This ensures that the script is executed before the HTML body is parsed. If you want to change the loading behavior of the page, or have something that needs to be executed while the page is still not fully loaded, you can put the script here. Also, if you have multiple scripts that depend on a particular script, you can put that shared script in the head to make sure it's executed before other scripts.

For a variety of reasons, including manageability, it is useful to put your JavaScript code into separate files that exist outside of your HTML code. External JavaScript files are included using a `<script>` tag with a `src` attribute, as follows:

```
<html>
  <head>
    <script src="/button-interaction.js"></script>
  </head>
  <body>
    </body>
</html>
```

The `src` tag tells the browser the location of the source, meaning the file containing the JavaScript code. The location can be a file on the same server, as in the example above, or any web-accessible URL such as <https://www.lpi.org/example.js>. The value of the `src` attribute follows the same convention as the import of CSS or image files, in that it can be relative or absolute. Upon encountering a script tag with the `src` attribute, the browser will then attempt to obtain the source file by using an HTTP GET request, so external files need to be accessible.

When you use the `src` attribute, any code or text that is placed between the `<script>...</script>` tags is ignored, according to the HTML specification.

```
<html>
  <head>
    <script src="/button-interaction.js">
      console.log("test"); // <-- This is ignored
    </script>
  </head>
  <body>
    </body>
</html>
```

There are other attributes you can add to the `script` tag to further specify how the browser should get the files, and how it should handle them afterwards. The following list goes into detail about the more important attributes:

async

Can be used on `script` tags and instructs the browser to fetch the script in the background, so as not to block the page load process. The page load will still be interrupted while after the browser obtains the script, because the browser has to parse it, which is done immediately once the script has been fetched completely. This attribute is Boolean, so writing the tag like `<script async src="/script.js"></script>` is sufficient and no value has to be provided.

defer

Similar to `async`, this instructs the browser not to block the page load process while fetching the script. Rather than that, the browser will defer parsing the script. The browser will wait until the entire HTML document has been parsed and only then it will parse the script, before announcing that the document has been to be completely loaded. Like `async`, `defer` is a Boolean attribute and is used in the same way. Since `defer` implies `async`, it is not useful to specify both tags together.

NOTE

When a page is completely parsed, the browser indicates that it is ready to display by triggering a `DOMContentLoaded` event, when the visitor will be able to see the document. Thus, the JavaScript included in a `<head>` event always has a chance to act on the page before it is displayed, even if you include the `defer` attribute.

type

Denotes what type of script the browser should expect within the tag. The default is JavaScript (`type="application/javascript"`), so this attribute isn't necessary when including JavaScript code or pointing to a JavaScript resource with the `src` tag. Generally, all MIME types can be specified, but only scripts that are denoted as JavaScript will be executed by the browser. There are two realistic use cases for this attribute: telling the browser not to execute the script by setting `type` to an arbitrary value like `template` or `other`, or telling the browser that the script is an ES6 module. We don't cover ES6 modules in this lesson.

WARNING

When multiple scripts have the `async` attribute, they will be executed in the order they finish downloading, *not* in the order of the `script` tags in the document. The `defer` attribute, on the other hand, maintains the order of the `script` tags.

Browser Console

While usually executed as part of a website, there is another way of executing JavaScript: through the browser console. All modern desktop browsers provide a menu through which you can execute JavaScript code in the browser's JavaScript engine. This is usually done for testing new code or to debug existing websites.

There are multiple ways to access the browser console, depending on the browser. The easiest way is through keyboard shortcuts. The following are the keyboard shorcuts for some of the browsers currently in use:

Chrome

`Ctrl + Shift + J` (`Cmd + Option + J` on Mac)

Firefox

`Ctrl + Shift + K` (`Cmd + Option + K` on Mac)

Safari

`Ctrl + Shift + ?` (`Cmd + Option + ?` on Mac)

You can also right-click on a webpage and select the “Inspect” or “Inspect Element” option to open the inspector, which is another browser tool. When the inspector opens, a new panel will appear. In this panel, you can select the “Console” tab, which will bring up the browser console.

Once you pull up the console, you can execute JavaScript on the page by typing the JavaScript directly into the input field. The result of any executed code will be shown in a separate line.

JavaScript Statements

Now that we know how to start executing a script, we will cover the basics of how a script is actually executed. A JavaScript script is a collection of statements and blocks. An example statement is `console.log('test')`. This instruction tells the browser to output the word `test` to the browser console.

Every statement in JavaScript is terminated by a semicolon (`;`). This tells the browser that the statement is done and a new one can be started. Consider the following script:

```
var message = "test"; console.log(message);
```

We have written two statements. Every statement is terminated either by a semicolon or by the end of the script. For readability purposes, we can put the statements on separate lines. This way, the script could also be written as:

```
var message = "test";
console.log(message);
```

This is possible because all whitespace between statements, such as a space, a newline, or a tab, is ignored. Whitespace can also often be put between individual keywords within statements, but this will be further explained in an upcoming lesson. Statements can also be empty, or comprised only of whitespace.

If a statement is invalid because it has not been terminated by a semicolon, ECMAScript makes an attempt to automatically insert the proper semicolons, based on a complex set of rules. The most important rule is: If an invalid statement is composed of two valid statements separated by a newline, insert a semicolon at the newline. For instance, the following code does not form a valid statement:

```
console.log("hello")
console.log("world")
```

But a modern browser will automatically execute it as if it were written with the proper semicolons:

```
console.log("hello");
console.log("world");
```

Thus, it is possible to omit semicolons in certain cases. However, as the rules for automatic semicolon insertion are complex, we advise you always to properly terminate your statements to avoid unwanted errors.

JavaScript Commenting

Big scripts can get quite complicated. You might want to comment on what you are writing, to make the script easier to read for other people, or for yourself in the future. Alternatively, you might want to include meta information in the script, such as copyright information, or information about when the script was written and why.

To make it possible to include such meta information, JavaScript supports *comments*. A developer can include special characters in a script that denote certain parts of it as a comment, which will be skipped on execution. The following is a heavily commented version of the script we saw earlier.

```
/*
 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.
```

Comments are not statements and do not need to be terminated by a semicolon. Instead, they follow their own rules for termination, depending on the way the comment is written. There are two ways to write comments in JavaScript:

Multi-line comment

Use `/*` and `*/` to signal the start and end of a multi-line comment. Everything after `/*`, until the first occurrence of `*/` is ignored. This kind of comment is generally used to span multiple lines, but it can also be used for single lines, or even within a line, like so:

```
console.log(/* what we want to log: */ "hello world")
```

Because the goal of comments is generally to increase the readability of a script, you should avoid using this comment style within a line.

Single-line comment

Use `//` (two forward slashes) to *comment out* a line. Everything after the double-slash on the same line is ignored. In the example shown earlier, this pattern is first used to comment an entire line. After the `console.log(message);` statement, it is used to write a comment on the rest of the line.

In general, single-line comments should be used for single lines, and multi-line comments for multiple lines, even if it possible to use them in other ways. Comments within a statement should be avoided.

Comments can also be used to temporarily remove lines of actual code, as follows:

```
// We temporarily want to use a different message
// var message = "test";
var message = "something else";
```

Guided Exercises

1. Create a variable called ColorName and assign the value RED to it.

2. Which of the following scripts are valid?

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") */	

Explorational Exercises

1. How many JavaScript statements can be written on a single line without using a semicolon?

2. Create two variables named `x` and `y`, then print their sum to the console.

Summary

In this lesson we learned different ways of executing JavaScript, and how to modify the script loading behavior. We also learned the basic concepts of script composition and commenting, and have learned to use the `console.log()` command.

HTML used in this lesson:

`<script>`

The `script` tag can be used to include JavaScript directly or by specifying a file with the `src` attribute. Modify how the script is loaded with the `async` and `defer` attributes.

JavaScript concepts introduced in this lesson:

;

A semicolon is used to separate statements. Semicolons can sometimes—but should not be—omitted.

`//, /*...*/`

Commenting can be used to add comments or meta information to a script file, or to prevent statements from being executed.

`console.log("text")`

The `console.log()` command can be used to output text to the browser console.

Answers to Guided Exercises

1. Create a variable called ColorName and assign the value RED to it.

```
var ColorName = "RED";
```

2. Which of the following scripts are valid?

console.log("hello") console.log("world");	Invalid: The first <code>console.log()</code> command isn't properly terminated, and the line as a whole doesn't form a valid statement.
console.log("hello"); console.log("world");	Valid: Each statement is properly terminated.
// console.log("hello") console.log("world");	Valid: The entire code is ignored because it is a comment.
console.log("hello"); console.log("world") //;	Invalid: The last statement is missing a semicolon. The semicolon at the very end is ignored because it is commented.
console.log("hello"); /* console.log("world") */	Valid: A valid statement is followed by commented code, which gets ignored.

Answers to Explorational Exercises

1. How many JavaScript statements can be written on a single line without using a semicolon?

If we are at the end of a script, we can write one statement and it will be terminated by the end of the file. Otherwise, you cannot write a statement without a semicolon with the syntax that you learned so far.

2. Create two variables named `x` and `y`, then print their sum to the console.

```
var x = 5;  
var y = 10;  
console.log(x+y);
```



034.2 JavaScript Data Structures

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 034.2](#)

Weight

3

Key knowledge areas

- Define and use variables and constants
- Understand data types
- Understand type conversion/coercion
- Understand arrays and objects
- Awareness of the variable scope

Partial list of the used files, terms and utilities

- `=, +, -, *, /, %, --, ++, +=, -=, *=, /=`
- `var, let, const`
- `boolean, number, string, symbol`
- `array, object`
- `undefined, null, NaN`



034.2 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	034 JavaScript Programming
Objective:	034.2 JavaScript Data Structures
Lesson:	1 of 1

Introduction

Programming languages, like natural languages, represent reality through symbols that are combined into meaningful statements. The reality represented by a programming language is the machine's resources, such as processor operations, devices, and memory.

Each of the myriad programming languages adopts a paradigm for representing information. JavaScript adopts conventions typical of *high-level* languages, where most of details such as memory allocation are implicit, allowing the programmer to focus on the script's purpose in the context of the application.

High-Level Languages

High-level languages provide abstract rules so that the programmer needs to write less code to express an idea. JavaScript offers convenient ways to make use of computer memory, using programming concepts that simplify the writing of recurring practices and that are generally sufficient for the web developer's purpose.

NOTE

Although it's possible to use specialized mechanisms for meticulous memory access, the simpler types of data we'll look at are more general in use.

Typical operations in a web application consist of requesting data through some JavaScript instruction and storing it to be processed and eventually presented to the user. This storage is quite flexible in JavaScript, with suitable storage formats for each purpose.

Declaration of Constants and Variables

The declaration of constants and variables to hold data is the cornerstone of any programming language. JavaScript adopts the convention of most programming languages, assigning values to constants or variables with the `name = value` syntax. The constant or variable on the left takes the value on the right. The constant or variable name must start with a letter or underscore.

The type of data stored in the variable does not need to be indicated, because JavaScript is a *dynamic typing* language. The type of the variable is inferred from the value assigned to it. However, it is convenient to designate certain attributes in the declaration to guarantee the expected result.

NOTE

TypeScript is a JavaScript-inspired language that, like lower-level languages, allows you to declare variables for specific types of data.

Constants

A constant is a symbol that is assigned once when the program starts and never changes. Constants are useful to specify fixed values such as defining the constant PI to be 3.14159265, or COMPANY_NAME to hold the name of your company.

In a web application, for example, take a client that receives weather information from a remote server. The programmer may decide that the address to the server must be constant, because it will not change during the application's execution. The temperature information, however, can change with every new data arrival from the server.

The interval between queries made to the server can also be defined as a constant, which can be queried from any part of the program:

```
const update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

When invoked, the `setup_app()` function displays the message `Update every 10 minutes` message on the console. The term `const` placed before the name `update_interval` makes sure that its value will remain the same throughout the entire execution of the script. If an attempt is made to reset the value of a constant, a `TypeError: Assignment to constant variable` error is issued.

Variables

Without the term `const`, JavaScript automatically assumes that `update_interval` is a variable and that its value can be modified. This is equivalent to declaring the variable explicitly with `var`:

```
var update_interval;
update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Note that although the `update_interval` variable was defined outside the function, it was accessed from within the function. Any constant or variable declared outside of functions or code blocks defined by braces (`{}`) has *global scope*; that is, it can be accessed from any part of the code. The opposite is not true: a constant or variable declared inside a function has *local scope*, so it is accessible only from inside the function itself. Bracket-delimited code blocks, such as those placed in `if` decision structures or `for` loops, delimit the scope of constants, but not variables declared as `var`. The following code, for example, is valid:

```
var success = true;
if ( success == true )
{
  var message = "Transaction succeeded";
  var retry = 0;
}
else
{
  var message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

The `console.log(message)` statement is able to access the `message` variable, even though it has been declared within the code block of the `if` structure. The same would not happen if `message` were

constant, as exemplified in the following example:

```
var success = true;
if ( success == true )
{
  const message = "Transaction succeeded";
  var retry = 0;
}
else
{
  const message = "Transaction failed";
  var retry = 1;
}

console.log(message);
```

In this case, an error message of type `ReferenceError: message is not defined` would be issued and the script would be stopped. While it may seem like a limitation, restricting the scope of variables and constants helps to avoid confusion between the information processed in the script body and in its different code blocks. For this reason, variables declared with `let` instead of `var` are also restricted in scope by the blocks delimited by braces. There are other subtle differences between declaring a variable with `var` or with `let`, but the most significant of these concerns the scope of the variable, as discussed here.

Types of Values

Most of the time, the programmer doesn't need to worry about the type of data stored in a variable, because JavaScript automatically identifies it with one of its *primitive* types during the first assignment of a value to the variable. Some operations, however, can be specific to one data type or another and can result in errors when used without discretion. In addition, JavaScript offers *structured* types that allow you to combine more than one primitive type into a single object.

Primitive Types

Primitive type instances correspond to traditional variables, which store only one value. Types are defined implicitly, so the `typeof` operator can be used to identify what type of value is stored in a variable:

```
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);
}
```

This script will display on the console what type of variable was used in each case:

```
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
```

Notice that the first line tries to find the type of an undeclared variable. This causes the given variable to be identified as `undefined`. The `symbol` type is the least intuitive primitive. Its purpose is to provide a unique attribute name within an object when there is no need to define a specific attribute name. An object is one of the data structures we'll look at next.

Structured Types

While primitive types are sufficient for writing simple routines, there are drawbacks to using them exclusively in more complex applications. An e-commerce application, for example, would be much more difficult to write, because the programmer would need to find ways to store lists of items and corresponding values using only variables with primitive types.

Structured types simplify the task of grouping information of the same nature into a single variable. A list of items in a shopping cart, for instance, can be stored in a single variable of type *array*:

```
let cart = ['Milk', 'Bread', 'Eggs'];
```

As demonstrated in the example, an array of items is designated with square brackets. The example has populated the array with three literal string values, hence the use of single quotes. Variables can also be used as items in an array, but in that case they must be designated without quotes. The number of items in an array can be queried with the `length` property:

```
let cart = ['Milk', 'Bread', 'Eggs'];
console.log(cart.length);
```

The number 3 will be displayed in the console output. New items can be added to the array with the `push()` method:

```
cart.push('Candy');
console.log(cart.length);
```

This time, the amount displayed will be 4. Each item in the list can be accessed by its numeric index, starting with 0:

```
console.log(cart[0]);
console.log(cart[3]);
```

The output displayed on the console will be:

```
Milk
Candy
```

Just as you can use `push()` to add an element, you can use `pop()` to remove the last element from an array.

Values stored in an array do not need be of the same type. It is possible, for example, to store the quantity of each item beside it. A shopping list like the one in the previous example could be constructed as follows:

```
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]);
```

The output displayed on the console after running this code is:

```
Item: Bread
Quantity: 4
```

As you may have already noticed, combining the names of items with their respective quantities in a single array may not be a good idea, because the relationship between them is not explicit in the data structure and is very susceptible to (human) errors. Even if an array were used for names and another array for quantities, maintaining the integrity of the list would require the same care and would not be very productive. In these situations, the best alternative is to use a more appropriate data structure: an *object*.

In JavaScript, an object-type data structure lets you bind properties to a variable. Also, unlike an array, the elements that make up an object do not have a fixed order. A shopping list item, for example, can be an object with the properties `name` and `quantity`:

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
console.log("Quantity: " + item.quantity);
```

This example shows that an object can be defined using braces ({}), where each property/value pair is separated by a colon and the properties are separated by commas. The property is accessible in the format `variable.property`, as in `item.name`, both for reading and for assigning new values. The output displayed on the console after running this code is:

```
Item: Milk
Quantity: 1
```

Finally, each object representing an item can be included in the shopping list array. This can be done directly when creating the list:

```
let cart = [{ name: 'Milk', quantity: 1 }, { name: 'Bread', quantity: 4 }];
```

As before, a new object representing an item can be added later to the array:

```
cart.push({ name: 'Eggs', quantity: 12 });
```

Items in the list are now accessed by their index and their property name:

```
console.log("Third item: " + cart[2].name);
console.log(cart[2].name + " quantity: " + cart[2].quantity);
```

The output displayed on the console after running this code is:

```
third item: eggs
Eggs quantity: 12
```

Data structures allow the programmer to keep their code much more organized and easier to maintain, whether by the original author or by other programmers on the team. Also, many outputs from JavaScript functions are in structured types, which need to be handled properly by the programmer.

Operators

So far, we've pretty much seen only how to assign values to newly created variables. As simple as it is, any program will perform several other manipulations on the values of variables. JavaScript offers several types of *operators* that can act directly on the value of a variable or store the result of the operation in a new variable.

Most operators are geared toward arithmetic operations. To increase the quantity of an item in the shopping list, for example, just use the `+` addition operator:

```
item.quantity = item.quantity + 1;
```

The following snippet prints the value of `item.quantity` before and after the addition. Do not mix

up the roles of the plus sign in the snippet. The `console.log` statements use a plus sign to combine two 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);
```

The output displayed on the console after running this code is:

```
Item: Milk
Quantity: 1
New quantity: 2
```

Note that the value previously stored in `item.quantity` is used as the operand of the addition operation: `item.quantity = item.quantity + 1`. Only after the operation is complete the value in `item.quantity` is updated with the result of the operation. This kind of arithmetic operation involving the current value of the target variable is quite common, so there are shorthand operators that allow you to write the same operation in a reduced format:

```
item.quantity += 1;
```

The other basic operations also have equivalent shorthand operators:

- `a = a - b` is equivalent to `a -= b`.
- `a = a * b` is equivalent to `a *= b`.
- `a = a / b` is equivalent to `a /= b`.

For addition and subtraction, there is a third format available when the second operand is only one unit:

- `a = a + 1` is equivalent to `a++`.
- `a = a - 1` is equivalent to `a--`.

More than one operator can be combined in the same operation and the result can be stored in a new variable. For example, the following statement calculates the total price of an item plus a shipping

cost:

```
let total = item.quantity * 9.99 + 3.15;
```

The order in which the operations are carried out follows the traditional order of precedence: first the multiplication and division operations are carried out, and only then are the addition and subtraction operations carried out. Operators with the same precedence are executed in the order they appear in the expression, from left to right. To override the default precedence order, you can use parentheses, as in `a * (b + c)`.

In some situations, the result of an operation doesn't even need to be stored in a variable. This is the case when you want to evaluate the result of an expression within an `if` statement:

```
if ( item.quantity % 2 == 0 )
{
  console.log("Quantity for the item is even");
}
else
{
  console.log("Quantity for the item is odd");
}
```

The `%` (modulo) operator returns the remainder of the division of the first operand by the second operand. In the example, the `if` statement checks whether the remainder of the division of `item.quantity` by 2 is equal to zero, that is, whether `item.quantity` is a multiple of 2.

When one of the operands of the `+` operator is a string, the other operators are *coerced* into strings and the result is a concatenation of strings. In the previous examples, this type of operation was used to concatenate strings and variables into the argument of the `console.log` statement.

This automatic conversion might not be the desired behavior. A user-supplied value in a form field, for example, might be identified as a string, but it is actually a numeric value. In cases like this, the variable must first be converted to a number with the `Number()` function:

```
sum = Number(value1) + value2;
```

Moreover, it is important to verify that the user has provided a valid value before proceeding with the operation. In JavaScript, a variable without an assigned value contains the value `null`. This allows the programmer to use a decision statement such as `if (value1 == null)` to check

whether a variable was assigned a value, regardless of the type of the value assigned to the variable.

Guided Exercises

1. An array is a data structure present in several programming languages, some of which allow only arrays with items of the same type. In the case of JavaScript, is it possible to define an array with items of different types?

2. Based on the example `let item = { name: 'Milk', quantity: 1 }` for an object in a shopping list, how could this object be declared to include the price of the item?

3. In a single line of code, what are the ways to update a variable's value to half its current value?

Explorational Exercises

1. In the following code, what value will be displayed in the console output?

```
var value = "Global";  
  
{  
    value = "Location";  
}  
  
console.log(value);
```

2. What will happen when one or more of the operands involved in a multiplication operation is a string?

3. How is it possible to remove the Eggs item from the `cart` array declared with `let cart = ['Milk', 'Bread', 'Eggs']` ?

Summary

This lesson covers the basic use of constants and variables in JavaScript. JavaScript is a *dynamic typing language*, so the programmer doesn't need to specify the variable type before setting it. However, it is important to know the language's primitive types to ensure the correct outcome of basic operations. Furthermore, data structures such as arrays and objects combine primitive types and allow the programmer to build more complex, composite variables. This lesson goes through the following concepts and procedures:

- Understanding constants and variables
- Variable scope
- Declaring variables with `var` and `let`
- Primitive types
- Arithmetic operators
- Arrays and objects
- Type coercion and conversion

Answers to Guided Exercises

1. An array is a data structure present in several programming languages, some of which allow only arrays with items of the same type. In the case of JavaScript, is it possible to define an array with items of different types?

Yes, in JavaScript it is possible to define arrays with items of different primitive types, such as strings and numbers.

2. Based on the example `let item = { name: 'Milk', quantity: 1 }` for an object in a shopping list, how could this object be declared to include the price of the item?

```
let item = { name: 'Milk', quantity: 1, price: 4.99 };
```

3. In a single line of code, what are the ways to update a variable's value to half its current value?

One can use the variable itself as an operand, `value = value / 2`, or the shorthand operator `/=`: `value /= 2`.

Answers to Explorational Exercises

1. In the following code, what value will be displayed in the console output?

```
var value = "Global";  
  
{  
  value = "Location";  
}  
  
console.log(value);
```

Location

2. What will happen when one or more of the operands involved in a multiplication operation is a string?

JavaScript will assign the value NaN (Not a Number) to the result, indicating that the operation is invalid.

3. How is it possible to remove the Eggs item from the `cart` array declared with `let cart = ['Milk', 'Bread', 'Eggs']`?

Arrays in javascript have the `pop()` method, which removes the last item in the list: `cart.pop()`.



034.3 JavaScript Control Structures and Functions

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 034.3](#)

Weight

4

Key knowledge areas

- Understand truthy and falsy values
- Understand comparison operators
- Understand the difference between loose and strict comparison
- Use conditionals
- Use loops
- Define custom functions

Partial list of the used files, terms and utilities

- `if, else if, else`
- `switch, case, break`
- `for, while, break, continue`
- `function, return`
- `==, !=, <, >, >=`
- `====, !===`



034.3 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	034 JavaScript Programming
Objective:	034.3 JavaScript Control Structures and Functions
Lesson:	1 of 2

Introduction

Like any other programming language, JavaScript code is a collection of statements that tells an instruction interpreter what to do in sequential order. However, this does not mean that every statement should be executed only once or that they should execute at all. Most statements should execute only when specific conditions are met. Even when a script is triggered asynchronously by independent events, it often has to check a number of control variables to find out the right portion of code to run.

If Statements

The simplest control structure is given by the `if` instruction, which will execute the statement immediately after it if the specified condition is true. JavaScript considers conditions to be true if the evaluated value is non-zero. Anything inside the parenthesis after the `if` word (spaces are ignored) will be interpreted as a condition. In the following example, the literal number `1` is the condition:

```
if ( 1 ) console.log("1 is always true");
```

The number 1 is explicitly written in this example condition, so it is treated as a constant value (it remains the same throughout the execution of the script) and it will always yield true when used as a conditional expression. The word `true` (without the double quotes) could also be used instead of 1, as it is also treated as a literal true value by the language. The `console.log` instruction prints its arguments in the browser's *console window*.

TIP

The browser console shows errors, warnings, and informational messages sent with the `console.log` JavaScript instruction. On Chrome, the `Ctrl + Shift + J` (`Cmd + Option + J` on Mac) key combination opens the console. On Firefox, the `Ctrl + Shift + K` (`Cmd + Option + K` on Mac) key combination opens the console tab in the developer's tools.

Although syntactically correct, using constant expressions in conditions is not very useful. In an actual application, you'll probably want to test the trueness of a variable:

```
let my_number = 3;
if ( my_number ) console.log("The value of my_number is", my_number, "and it yields
true");
```

The value assigned to the variable `my_number` (3) is non-zero, so it yields true. But this example is not common usage, because you rarely need to test whether a number equals zero. It is much more common to compare one value to another and test whether the result is true:

```
let my_number = 3;
if ( my_number == 3 ) console.log("The value of my_number is", my_number, "indeed
");
```

The double equal comparison operator is used because the single equal operator is already defined as the assignment operator. The value on each side of the operator is called an *operand*. The ordering of the operands does not matter and any expression that returns a value can be an operand. Here a list of other available comparison operators:

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 greater than or equal to value2.

Usually, it does not matter whether the operand to the left of the operator is a string and the operand to the right is a number, as long as JavaScript is able to convert the expression to a meaningful comparison. So the string containing the character 1 will be treated as the number 1 when compared to a number variable. To make sure the expression yields true only if both operands are of the exact same type and value, the strict identity operator `==` should be used instead of `=`. Likewise, the strict non-identity operator `!=` evaluates as true if the first operand is not of the exact same type and value as the second operator.

Optionally, the `if` control structure can execute an alternate statement when the expression evaluates as false:

```
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");
```

The `else` instruction should immediately follow the `if` instruction. So far, we are executing only one statement when the condition is met. To execute more than one statement, you must enclose them in curly braces:

```
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");
}
```

A group of one or more statements delimited by a pair of curly braces is known as a *block statement*. It is common to use block statements even when there is only one instruction to execute, in order to make the coding style consistent throughout the script. Moreover, JavaScript does not require the curly braces or any statements to be on separate lines, but to do so improves readability and makes code maintenance easier.

Control structures can be nested inside each other, but it is important not to mix up the opening and closing braces of each block statement:

```
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");
}
```

The expressions evaluated by the `if` instruction can be more elaborate than simple comparisons. This is the case in the previous example, where the arithmetical expression `my_number % 2` was employed inside the parentheses in the nested `if`. The `%` operator returns the remainder after dividing the number on its left by the number on its right. Arithmetical operators like `%` take precedence over comparison operators like `==`, so the comparison will use the result of the arithmetical expression as its left operand.

In many situations, nested conditional structures can be combined into a single structure using *logical operators*. If we were interested only in positive even numbers, for example, a single `if` structure could be used:

```
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");
}
```

The double ampersand operator `&&` in the evaluated expression is the logical *AND* operator. It evaluates as true only if the expression to its left and the expression to its right evaluate as true. If you want to match numbers that are either positive or even, the `||` operator should be used instead, which stands for the *OR* logical operator:

```
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");
}
```

In this example, only negative odd numbers will fail to match the criteria imposed by the composite expression. If you have the opposite intent, that is, to match only the negative odd numbers, add the logical *NOT* operator `!` to the beginning of the expression:

```
let my_number = -5;

if ( ! ( my_number > 0 || my_number % 2 == 0 ) )
{
    console.log("The value of my_number is an odd negative number");
}
```

Adding the parenthesis in the composite expression forces the expression enclosed by them to be evaluated first. Without these parentheses, the NOT operator would apply only to `my_number > 0` and then the OR expression would be evaluated. The `&&` and `||` operators are known as *binary* logical operators, because they require two operands. `!` is known as a *unary* logical operator, because it requires only one operand.

Switch Structures

Although the `if` structure is quite versatile and sufficient to control the flow of the program, the `switch` control structure may be more appropriate when results other than true or false need to be evaluated. For example, if we want to take a distinct action for each item chosen from a list, it will be necessary to write an `if` structure for each assessment:

```
// 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");
}
```

In this example, an auxiliary variable `found` is used by all `if` structures to find out whether a match has occurred. In a case such as this one, the `switch` structure will perform the same task, but in a more succinct manner:

```
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");
}
```

Each nested `case` is called a *clause*. When a clause matches the evaluated expression, it executes the statements following the colon until the `break` statement. The last clause does not need a `break` statement and is often used to set the default action when no other matches occur. As seen in the example, the auxiliary variable is not needed in the `switch` structure.

WARNING `switch` uses strict comparison to match expressions against its `case` clauses.

If more than one clause triggers the same action, you can combine two or more `case` conditions:

```
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");
}
```

Loops

In previous examples, the `if` and `switch` structures were well suited for tasks that need to run only once after passing one or more conditional tests. However, there are situations when a task must execute repeatedly—in a so-called *loop*—as long as its conditional expression keeps testing as true. If you need to know whether a number is prime, for example, you will need to check whether dividing this number by any integer greater than 1 and lower than itself has a remainder equal to 0. If so, the number has an integer factor and it is not prime. (This is not a rigorous or efficient method to find prime numbers, but it works as a simple example.) Loop controls structures are more suitable for such cases, in particular the `while` statement:

```
// 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");
}
```

The block statement following the `while` instruction will execute repeatedly as long as the condition `factor < candidate` is true. It will execute at least once, as long as we initialize the `factor` variable

with a value lower than `candidate`. The `if` structure nested in the `while` structure will evaluate whether the remainder of `candidate` divided by `factor` is zero. If so, the `candidate` number is not prime and the loop can finish. The `break` statement will finish the loop and the execution will jump to the first instruction after the `while` block.

Note that the outcome of the condition used by the `while` statement must change at every loop, otherwise the block statement will loop “forever”. In the example, we increment the `factor` variable—the next divisor we want to try—and it guarantees the loop will end at some point.

This simple implementation of a prime number tester works as expected. However, we know that a number that is not divisible by two will not be divisible by any other even number. Therefore, we could just skip the even numbers by adding another `if` instruction:

```
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++;
}
```

The `continue` statement is similar to the `break` statement, but instead of finishing this iteration of the loop, it will ignore the rest of the loop’s block and start a new iteration. Note that the `factor` variable was modified before the `continue` statement, otherwise the loop would have the same outcome again in the next iteration. This example is too simple and skipping part of the loop will not really improve its performance, but skipping redundant instructions is very important when writing efficient applications.

Loops are so commonly used that they exist in many different variants. The `for` loop is specially

suited to iterate through sequential values, because it lets us define the loop rules in a single line:

```
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;
  }
}
```

This example produces the exact same result as the previous `while` example, but its parenthetical expression includes three parts, separated by semicolons: the initialization (`let factor = 2`), the loop condition (`factor < candidate`), and the final expression to be evaluated at the end of each loop iteration (`factor++`). The `continue` and `break` statements also apply to `for` loops. The final expression in the parentheses (`factor++`) will be evaluated after the `continue` statement, so it should not be inside the block statement, otherwise it will be incremented two times before the next iteration.

JavaScript has special types of `for` loops to work with array-like objects. We could, for example, check an array of candidate variables instead of just one:

```

// 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");
    }
}

```

The `for (candidate of candidates)` statement assigns one element of the `candidates` array to the `candidate` variable and uses it in the block statement, repeating the process for every element in the array. You don't need to declare `candidate` separately, because the `for` loop defines it. Finally, the same code from the previous example was nested in this new block statement, but this time testing each candidate in the array.

Guided Exercises

1. What values of the `my_var` variable match the condition `my_var > 0 && my_var < 9?`

2. What values of the `my_var` variable match the condition `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++;
}
```

Explorational Exercises

1. What happens if the equal assignment operator `=` is used instead of the equal comparison operator `==`?

2. Write a code fragment using the `if` control structure where an ordinary equality comparison will return true, but a strict equality comparison will not.

3. Rewrite the following `for` statement using the unary NOT logical operator in the loop condition. The outcome of the condition should be the same.

```
for ( let factor = 2; factor < candidate; factor++ )
```

4. Based on the examples from this lesson, write a loop control structure that prints all the integer factors of a given number.

Summary

This lesson covers how to use control structures in JavaScript code. Conditional and loop structures are essential elements of any programming paradigm, and JavaScript web development is no exception. The lesson goes through the following concepts and procedures:

- The `if` statement and comparison operators.
- How to use the `switch` structure with `case`, `default`, and `break`.
- The difference between ordinary and strict comparison.
- Loop control structures: `while` and `for`.

Answers to Guided Exercises

1. What values of the `my_var` variable match the condition `my_var > 0 && my_var < 9?`

Only numbers that are both greater than 0 and less than 9. The `&&` (AND) logical operator requires both comparisons to match.

2. What values of the `my_var` variable match the condition `my_var > 0 || my_var < 9?`

Using the `||` (OR) logical operator will cause any number to match, as any number will either be greater than 0 or less than 9.

3. How many times does the following `while` loop execute its block statement?

```
let i = 0;
while (1)
{
  if (i == 10)
  {
    continue;
  }
  i++;
}
```

The block statement will repeat indefinitely, as no stop condition was supplied.

Answers to Explorational Exercises

1. What happens if the equal assignment operator `=` is used instead of the equal comparison operator `==`?

The value on the right side of the operator is assigned to the variable on the left and the result is passed to the comparison, which may not be the desired behavior.

2. Write a code fragment using the `if` control structure where an ordinary equality comparison will return true, but a strict equality comparison will not.

```
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. Rewrite the following `for` statement using the unary NOT logical operator in the loop condition. The outcome of the condition should be the same.

```
for ( let factor = 2; factor < candidate; factor++ )
```

Answer:

```
for ( let factor = 2; ! (factor >= candidate); factor++ )
```

4. Based on the examples from this lesson, write a loop control structure that prints all the integer factors of a given number.

```
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 Lesson 2

Certificate:	Web Development Essentials
Version:	1.0
Topic:	034 JavaScript Programming
Objective:	034.3 JavaScript Control Structures and Functions
Lesson:	2 of 2

Introduction

In addition to the standard set of builtin functions provided by the JavaScript language, developers can write their own custom functions to map an input to an output suitable to the application's needs. Custom functions are basically a set of statements encapsulated to be used elsewhere as part of an expression.

Using functions is a good way to avoid writing duplicate code, because they can be called from different locations throughout the program. Moreover, grouping statements in functions facilitates the binding of custom actions to events, which is a central aspect of JavaScript programming.

Defining a Function

As a program grows, it becomes harder to organize what it does without using functions. Each function has its own private variable scope, so the variables defined inside a function will be available only inside that same function. Thus, they won't get mixed up with variables from other functions. Global variables are still accessible from within functions, but the preferable way to send input values to a function is through *function parameters*. As an example, we are going to build on

the prime number validator from the previous lesson:

```
// 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");
}
```

If later in the code you need to check if a number is prime, it would be necessary to repeat the code that has already been written. This practice is not recommended, because any corrections or improvements to the original code would need to be replicated manually everywhere the code was copied to. Moreover, repeating code places a burden on the browser and network, possibly slowing down the display of the web page. Instead of doing this, move the appropriate statements to a

function:

```
// 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;
}
```

The function declaration starts with a `function` statement, followed by the name of the function and its parameters. The name of the function must follow the same rules as the names for variables. The parameters of the function, also known as the function *arguments*, are separated by commas and enclosed by parenthesis.

TIP

Listing the arguments in the function declaration is not mandatory. The arguments passed to a function can be retrieved from an array-like `arguments` object inside that function. The index of the arguments starts at 0, so the first argument is `arguments[0]`, the second argument is `arguments[1]`, and so on.

In the example, the `test_prime` function has only one argument: the `candidate` argument, which is the prime number candidate to be tested. Function arguments work like variables, but their values

are assigned by the statement calling the function. For example, the statement `test_prime(231)` will call the `test_prime` function and assign the value 231 to the `candidate` argument, which will then be available inside the function's body like an ordinary variable.

If the calling statement uses simple variables for the function's parameters, their values will be copied to the function arguments. This procedure—copying the values of the parameters used in the calling statement to the parameters used inside the function— is called *passing arguments by value*. Any modifications made to the argument by the function does not affect the original variable used in the calling statement. However, if the calling statement uses complex objects as arguments (that is, an object with properties and methods attached to it) for the function's parameters, they will be *passed as reference* and the function will be able to modify the original object used in the calling statement.

The arguments that are passed by value, as well as the variables declared within the function, are not visible outside it. That is, their scope is restricted to the body of the function where they were declared. Nonetheless, functions are usually employed to create some output visible outside the function. To share a value with its calling function, a function defines a `return` statement.

For instance, the `test_prime` function in the previous example returns the value of the `is_prime` variable. Therefore, the function can replace the variable anywhere it would be used in the original example:

```
// 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");
}
```

The `return` statement, as its name indicates, returns control to the calling function. Therefore, wherever the `return` statement is placed in the function, nothing following it is executed. A function can contain multiple `return` statements. This practice can be useful if some are within conditional blocks of statements, so that the function might or might not execute a particular `return` statement on each run.

Some functions may not return a value, so the `return` statement is not mandatory. The function's internal statements are executed regardless of its presence, so functions can also be used to change the values of global variables or the contents of objects passed by reference, for example. Notwithstanding, if the function does not have a `return` statement, its default return value is set to `undefined`: a reserved variable that does not have a value and cannot be written.

Function Expressions

In JavaScript, functions are just another type of *object*. Thus, functions can be employed in the script like variables. This characteristic becomes explicit when the function is declared using an alternative syntax, called *function expressions*:

```
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;
}
```

The only difference between this example and the function declaration in the previous example is in the first line: `let test_prime = function(candidate)` instead of `function test_prime(candidate)`. In a function expression, the `test_prime` name is used for the object containing the function and not to name the function itself. Functions defined in function expressions are called in the same way as

functions defined using the declaration syntax. However, whereas declared functions can be called before or after their declaration, function expressions can be called only after their initialization. Like with variables, calling a function defined in an expression before its initialization will cause a reference error.

Function Recursion

In addition to executing statements and calling builtin functions, custom functions can also call other custom functions, including themselves. To call a function from itself is called *function recursion*. Depending on the type of problem you are trying to solve, using recursive functions can be more straightforward than using nested loops to perform repetitive tasks.

So far, we know how to use a function to test whether a given number is prime. Now suppose you want to find the next prime following a given number. You could employ a `while` loop to increment the candidate number and write a nested loop which will look for integer factors for that candidate:

```
// 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 than 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 that we need to use a constant condition for the `while` loop (the `true` expression inside the parenthesis) and the auxiliary variable `is_prime` to know when to stop the loop. Although this solution is correct, using nested loops is not as elegant as using recursion to perform the same task:

```

// 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 than 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));
```

Both versions of `next_prime` return the next prime number after the number given as its only argument (`from`). The recursive version, like the previous version, starts by checking the special cases (i.e. numbers less or equal to two). Then it increments the candidate and starts looking for any integer factors with the `for` loop (notice that the `while` loop is not there anymore). At that point, the only even prime number has already been tested, so the candidate and its possible factors are incremented by two. (An odd number plus two is the next odd number.)

There are only two ways out of the `for` loop in the example. If all the possible factors are tested and none of them has a remainder equal to zero when dividing the candidate, the `for` loop completes and the function returns the candidate as the next prime number after `from`. Otherwise, if `factor` is an integer factor of `candidate` (`candidate % factor == 0`), the returned value comes from the `next_prime` function called recursively, this time with the incremented `candidate` as its `from` parameter. The calls for `next_prime` will be stacked on top of one another, until one candidate finally turns up no integer factors. Then the last `next_prime` instance containing the prime number will return it to the previous `next_prime` instance, and thus successively down to the first `next_prime` instance. Even though each invocation of the function uses the same names for variables, the invocations are isolated from each other, so their variables are kept separated in memory.

Guided Exercises

1. What kind of overhead can developers mitigate by using functions?

2. What is the difference between function arguments passed by value and function arguments passed by reference?

3. Which value will be used as the output of a custom function if it does not have a return statement?

Explorational Exercises

1. What is the probable cause of an *Uncaught Reference Error* issued when calling a function declared with the *expression* syntax?

2. Write a function called `multiples_of` that receives three arguments: `factor`, `from`, and `to`. Inside the function, use the `console.log()` instruction to print all multiples of `factor` lying between `from` and `to`.

Summary

This lesson covers how to write custom functions in JavaScript code. Custom functions let the developer divide the application in “chunks” of reusable code, making it easier to write and maintain larger programs. The lesson goes through the following concepts and procedures:

- How to define a custom function: function declarations and function expressions.
- Using parameters as the function input.
- Using the `return` statement to set the function output.
- Function recursion.

Answers to Guided Exercises

1. What kind of overhead can developers mitigate by using functions?

Functions allow us to reuse code, which facilitates code maintenance. A smaller script file also saves memory and download time.

2. What is the difference between function arguments passed by value and function arguments passed by reference?

When passed by value, the argument is copied to the function and the function is not able to modify the original variable in the calling statement. When passed by reference, the function is able to manipulate the original variable used in the calling statement.

3. Which value will be used as the output of a custom function if it does not have a return statement?

The returned value will be set to `undefined`.

Answers to Explorational Exercises

1. What is the probable cause of an *Uncaught Reference Error* issued when calling a function declared with the *expression* syntax?

The function was called before its declaration in the script file.

2. Write a function called `multiples_of` that receives three arguments: `factor`, `from`, and `to`. Inside the function, use the `console.log()` instruction to print all multiples of `factor` lying between `from` and `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 JavaScript Manipulation of Website Content and Styling

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 034.4

Weight

4

Key knowledge areas

- Understand the concept and structure of the DOM
- Change the contents and properties of HTML elements through the DOM
- Change the CSS styling of HTML elements through the DOM
- Trigger JavaScript functions from HTML elements

Partial list of the used files, terms and utilities

- `document.getElementById()`, `document.getElementsByClassName()`, `document.querySelector()`, `document.querySelectorAll()`
- `innerHTML`, `setAttribute()`, `removeAttribute()` properties and methods of DOM elements
- `classList`, `classList.add()`, `classList.remove()`, `classList.toggle()` properties and methods of DOM elements
- `onClick`, `onMouseOver`, `onMouseOut` attributes of HTML elements



034.4 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	034 JavaScript Programming
Objective:	034.4 JavaScript Manipulation of Website Content and Styling
Lesson:	1 of 1

Introduction

HTML, CSS, and JavaScript are three distinct technologies that come together on the Web. In order to make truly dynamic and interactive pages, the JavaScript programmer must combine components from HTML and CSS at run time, a task that is greatly facilitated by using the *Document Object Model* (DOM).

Interacting with the DOM

The DOM is a data structure that works as a programming interface to the document, where every aspect of the document is represented as a node in the DOM and every change made to the DOM will immediately reverberate in the document. To show how to use the DOM in JavaScript, save the following HTML code in a file called `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>
```

The DOM will be available only after the HTML is loaded, so write the following JavaScript at the end of the page body (before the ending `</body>` tag):

```
<script>
let body = document.getElementsByTagName("body")[0];
console.log(body.innerHTML);
</script>
```

The `document` object is the top DOM element, all other elements branch off from it. The `getElementsByTagName()` method lists all the elements descending from `document` that have the given tag name. Even though the `body` tag is used only once in the document, the `getElementsByTagName()` method always return an array-like collection of found elements, hence the use of the `[0]` index to return the first (and only) element found.

HTML Content

As shown in the previous example, the DOM element returned by the `document.getElementsByTagName("body")[0]` was assigned to the `body` variable. The `body` variable can then be used to manipulate the page's body element, because it inherits all the DOM methods and attributes from that element. For instance, the `innerHTML` property contains the entire HTML markup code written inside the corresponding element, so it can be used to read the inner markup. Our `console.log(body.innerHTML)` call prints the content inside `<body></body>` to the web console.

The variable can also be used to replace that content, as in `body.innerHTML = "<p>Content erased</p>"`.

Rather than changing entire portions of HTML markup, it is more practical to keep the document structure unaltered and just interact with its elements. Once the document is rendered by the browser, all the elements are accessible by DOM methods. It is possible, for example, to list and access all the HTML elements using the special string `*` in the `getElementsByTagName()` method of the `document` object:

```
let elements = document.getElementsByTagName("*");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

This code will place all the elements found in `document` in the `elements` variable. The `elements` variable is an array-like object, so we can iterate through each of its items with a `for` loop. If the HTML page where this code runs has an `id` attribute set to `content_first` (see the sample HTML page shown at the start of the lesson), the `if` statement matches that element and its markup contents will be changed to `<p>New content</p>`. Note that the attributes of an HTML element in the DOM are accessible using the *dot* notation of JavaScript object properties: therefore, `element.id` refers to the `id` attribute of the current element of the `for` loop. The `getAttribute()` method could also be used, as in `element.getAttribute("id")`.

It is not necessary to iterate through all the elements if you want to inspect only a subset of them. For example, the `document.getElementsByClassName()` method limits the matched elements to those having a specific class:

```
let elements = document.getElementsByClassName("content");
for ( element of elements )
{
  if ( element.id == "content_first" )
  {
    element.innerHTML = "<p>New content</p>";
  }
}
```

However, iterating through many document elements using a loop is not the best strategy when you

have to change a specific element in the page.

Selecting Specific Elements

JavaScript provides optimized methods to select the exact element you want to work on. The previous loop could be entirely replaced by the `document.getElementById()` method:

```
let element = document.getElementById("content_first");
element.innerHTML = "<p>New content</p>";
```

Every `id` attribute in the document must be unique, so the `document.getElementById()` method returns only a single DOM object. Even the declaration of the `element` variable can be omitted, because JavaScript lets us chain methods directly:

```
document.getElementById("content_first").innerHTML = "<p>New content</p>";
```

The `getElementById()` method is the preferable method to locate elements in the DOM, because its performance is much better than iterative methods when working with complex documents. However, not all elements have an explicit ID, and the method returns a `null` value if no element matches with the provided ID (this also prevents the use of chained attributes or functions, like the `innerHTML` used in the example above). Moreover, it is more practical to assign ID attributes only to the main components of the page and then use CSS selectors to locate their child elements.

Selectors, introduced in an earlier lesson on CSS, are patterns that match elements in the DOM. The `querySelector()` method returns the first matching element in the DOM's tree, while `querySelectorAll()` returns all elements that match the specified selector.

In the previous example, the `getElementById()` method retrieves the element bearing the `content_first` ID. The `querySelector()` method can perform the same task:

```
document.querySelector("#content_first").innerHTML = "<p>New content</p>";
```

Because the `querySelector()` method uses selector syntax, the provided ID must begin with a hash character. If no matching element is found, the `querySelector()` method returns `null`.

In the previous example, the entire content of the `content_first` div is replaced by the text string provided. The string has HTML code in it, which is not considered a best practice. You must be careful when adding hard-coded HTML markup to JavaScript code, because tracking elements can become difficult when changes to the overall document structure are required.

Selectors are not restricted to the ID of the element. The internal `p` element can be addressed directly:

```
document.querySelector("#content_first p").innerHTML = "New content";
```

The `#content_first p` selector will match only the first `p` element inside the `#content_first` div. It works fine if we want to manipulate the first element. However, we may want to change the second paragraph:

```
<div class="content" id="content_first">
<p>Don't change this paragraph.</p>
<p>The dynamic content goes here.</p>
</div><!-- #content_first -->
```

In this case, we can use the `:nth-child(2)` pseudo-class to match the second `p` element:

```
document.querySelector("#content_first p:nth-child(2)").innerHTML = "New content";
```

The number 2 in `p:nth-child(2)` indicates the second paragraph that matches the selector. See the CSS selectors lesson to know more about selectors and how to use them.

Working with Attributes

JavaScript's ability to interact with the DOM is not restricted to content manipulation. Indeed, the most pervasive use of JavaScript in the browser is to modify the attributes of the existing HTML elements.

Let's say our original HTML example page has now three sections of content:

```
<!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>
```

You may want to make only one of them visible at a time, hence the `hidden` attribute in all `div` tags. This is useful, for example, to show only one image from a gallery of images. To make one of them visible when the page loads, add the following JavaScript code to the page:

```
// 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");
```

The expression evaluated by the `switch` statement randomly returns the number 0, 1, or 2. The corresponding ID selector is then assigned to the `content_visible` variable, which is used by the `querySelector(content_visible)` method. The chained `removeAttribute("hidden")` call removes the `hidden` attribute from the element.

The opposite approach is also possible: All sections could be initially visible (without the `hidden` attribute) and the JavaScript program can then assign the `hidden` attribute to every section except the one in `content_visible`. To do so, you must iterate through all the content `div` elements that are different from the chosen one, which can be done by using the `querySelectorAll()` method:

```
// 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", "");
}
```

If the `content_visible` variable was set to `#content_first`, the selector will be `.content:not(#content_first)`, which reads as all elements having the `content` class except those having the `content_first` ID. The `setAttribute()` method adds or changes attributes of HTML elements. Its first parameter is the name of the attribute and the second is the value of the attribute.

However, the proper way to change the appearance of elements is with CSS. In this case, we can set the `display` CSS property to `hidden` and then change it to `block` using 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>

```

The same good practices that apply to mixing HTML tags with JavaScript apply also to CSS. Thus, writing CSS properties directly in JavaScript code is not recommended. Instead, CSS rules should be written apart from JavaScript code. The proper way to alternate visual styling is to select a pre-defined CSS class for the element.

Working with Classes

Elements may have more than one associated class, making it easier to write styles that can be added

or removed when necessary. It would be exhausting to change many CSS attributes directly in JavaScript, so you can create a new CSS class with those attributes and then add the class to the element. DOM elements have the `classList` property, which can be used to view and manipulate the classes assigned to the corresponding element.

For example, instead of changing the visibility of the element, we can create an additional CSS class to highlight our content div:

```
div.content {  
    border: 1px solid black;  
    opacity: 0.25;  
}  
div.content.highlight {  
    border: 1px solid red;  
    opacity: 1;  
}
```

This stylesheet will add a thin black border and semi-transparency to all elements having the `content` class. Only the elements that also have the `highlight` class will be fully opaque and have the thin red border. Then, instead of changing the CSS properties directly as we did before, we can use the `classList.add("highlight")` method in the selected element:

```
// 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");
```

All the techniques and examples we have seen so far were performed at the end of the page loading process, but they are not restricted to this stage. In fact, what makes JavaScript so useful to Web developers is its ability to react to events on the page, which we will see next.

Event Handlers

All visible page elements are susceptible to interactive events, such as the click or the movement of the mouse itself. We can associate custom actions to these events, which greatly expands what an HTML document can do.

Probably the most obvious HTML element that benefits from an associated action is the `button` element. To show how it works, add three buttons above the first `div` element of the example page:

```
<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 -->
```

The buttons do nothing on their own, but suppose you want to highlight the `div` corresponding to the pressed button. We can use the `onClick` attribute to associate an action to each button:

```
<p>
<button
  onClick="document.getElementById('content_first').classList.toggle('highlight')">First</button>
<button
  onClick="document.getElementById('content_second').classList.toggle('highlight')">Second</button>
<button
  onClick="document.getElementById('content_third').classList.toggle('highlight')">Third</button>
</p>
```

The `classList.toggle()` method adds the specified class to the element if it is not present, and removes that class if it is already present. If you run the example, you will note that more than one `div` can be highlighted at the same time. To highlight only the `div` corresponding to the pressed button, it is necessary to remove the `highlight` class from the other `div` elements. Nonetheless, if the custom action is too long or involves more than one line of code, it's more practical to write a function apart from the element tag:

```
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');
}
```

Like the previous examples, this function can be placed inside a `<script>` tag or in an external JavaScript file associated with the document. The `highlight` function first removes the `highlight` class from all the `div` elements associated with the `content` class, then adds the `highlight` class to the chosen element. Each button should then call this function from its `onClick` attribute, using the corresponding ID as the function's argument:

```
<p>
<button onClick="highlight('content_first')">First</button>
<button onClick="highlight('content_second')">Second</button>
<button onClick="highlight('content_third')">Third</button>
</p>
```

In addition to the `onClick` attribute, we could use the `onMouseOver` attribute (triggered when the pointing device is used to move the cursor onto the element), the `onMouseOut` attribute (triggered when the pointing device is no longer contained within the element), etc. Moreover, event handlers are not restricted to buttons, so you can assign custom actions to these event handlers for all visible HTML elements.

Guided Exercises

1. Using the `document.getElementById()` method, how could you insert the phrase “Dynamic content” to the inner content of the element whose ID is `message`?

2. What is the difference between referencing an element by its ID using the `document.querySelector()` method and doing so via the `document.getElementById()` method?

3. What is the purpose of the `classList.remove()` method?

4. What is the result of using the method `myElement.classList.toggle("active")` if `myElement` does not have the `active` class assigned to it?

Explorational Exercises

1. What argument to the `document.querySelectorAll()` method will make it mimic the `document.getElementsByTagName("input")` method?

2. How can you use the `classList` property to list all the classes associated with a given element?

Summary

This lesson covers how to use JavaScript to change HTML contents and their CSS properties using the DOM (Document Object Model). These changes can be triggered by user events, which is useful to create dynamic interfaces. The lesson goes through the following concepts and procedures:

- How to inspect the structure of the document using methods like `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()` and `document.querySelectorAll()`.
- How to change the document's content with the `innerHTML` property.
- How to add and modify the attributes of page elements with methods `setAttribute()` and `removeAttribute()`.
- The proper way to manipulate elements classes using the `classList` property and its relation to CSS styles.
- How to bind functions to mouse events in specific elements.

Answers to Guided Exercises

1. Using the `document.getElementById()` method, how could you insert the phrase “Dynamic content” to the inner content of the element whose ID is `message`?

It can be done with the `innerHTML` property:

```
document.getElementById("message").innerHTML = "Dynamic content"
```

2. What is the difference between referencing an element by its ID using the `document.querySelector()` method and doing so via the `document.getElementById()` method?

The ID must be accompanied by the hash character in functions that use selectors, such as `document.querySelector()`.

3. What is the purpose of the `classList.remove()` method?

It removes the class (whose name is given as the argument of the function) from the `class` attribute of the corresponding element.

4. What is the result of using the method `myelement.classList.toggle("active")` if `myelement` does not have the `active` class assigned to it?

The method will assign the `active` class to `myelement`.

Answers to Explorational Exercises

1. What argument to the `document.querySelectorAll()` method will make it mimic the `document.getElementsByTagName("input")` method?

Using `document.querySelectorAll("input")` will match all the `input` elements in the page, just like `document.getElementsByTagName("input")`.

2. How can you use the `classList` property to list all the classes associated with a given element?

The `classList` property is an array-like object, so a `for` loop can be used to iterate through all the classes it contains.



Topic 035: NodeJS Server Programming



035.1 NodeJS Basics

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 035.1

Weight

1

Key knowledge areas

- Understand the concepts of Node.js
- Run a NodeJS application
- Install NPM packages

Partial list of the used files, terms and utilities

- `node [file.js]`
- `npm init`
- `npm install [module_name]`
- `package.json`
- `node_modules`



035.1 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	035 Node.js Server Programming
Objective:	035.1 Node.js Basics
Lesson:	1 of 1

Introduction

Node.js is a JavaScript runtime environment that runs JavaScript code in web servers—the so-called web *backend* (server side)—instead of using a second language like Python or Ruby for server-side programs. The JavaScript language is already used in the modern frontend side of web applications, interacting with the HTML and CSS of the interface that the user interacts with a web browser. Using Node.js in tandem with JavaScript in the browser offers the possibility of just one programming language for the whole application.

The main reason for the existence of Node.js is the way it handles multiple concurrent connections in the backend. One of the most common ways that a web application server handles connections is through the execution of multiple processes. When you open a desktop application in your computer, a process starts and uses a lot of resources. Now think about when thousands of users are doing the same thing in a large web application.

Node.js avoids this problem using a design called the *event loop*, which is an internal loop that continuously checks for incoming tasks to be computed. Thanks to the widespread use of JavaScript and the ubiquity of web technologies, Node.js has seen a huge adoption in both small and large applications. There are other characteristics that also helped Node.js to be widely adopted, such as

asynchronous and non-blocking input/output (I/O) processing, which is explained later in this lesson.

The Node.js environment uses a JavaScript engine to interpret and execute JavaScript code on the server side or on the desktop. In these conditions, the JavaScript code that the programmer writes is parsed and compiled just-in-time to execute the machine instructions generated by the original JavaScript code.

NOTE As you progress through these lessons about Node.js, you may notice that the Node.js JavaScript is not exactly the same as the one that runs on the browser (which follows the [ECMAScript specification](#)), but is quite similar.

Getting Started

This section and the following examples assume that Node.js is already installed on your Linux operating system, and that the user already has basic skills such as how to execute commands in the terminal.

To run the following examples, create a working directory called `node_examples`. Open a terminal prompt and type `node`. If you have correctly installed Node.js, it will present a `>` prompt where you can test JavaScript commands interactively. This kind of environment is called **REPL**, for “read, evaluate, print, and loop”. Type the following input (or some other JavaScript statements) at the `>` prompts. Press the Enter key after each line, and the REPL environment will return the results of its actions:

```
> 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'
]
>
```

The snippet was written using ES6 syntax, which offers a `map` function to iterate over the array and print the results using string templates. You can write pretty much any command that is valid. To exit the Node.js terminal, type `.exit`, remembering to include the initial period.

For longer scripts and modules, it is more convenient to use a text editor such as VS Code, Emacs, or

Vim. You can save the two lines of code just shown (with a little modification) in a file called `start.js`:

```
let array = ['a', 'b', 'c', 'd'];
array.map( (element, index) => ( console.log(`Element: ${element} at index: ${index}`)));
```

Then you can run the script from the shell to produce the same results as before:

```
$ node ./start.js
Element: a at index: 0
Element: b at index: 1
Element: c at index: 2
Element: d at index: 3
```

Before diving into some more code, we are going to get an overview of how Node.js works, using its single thread execution environment and the event loop.

Event Loop and Single Thread

It is hard to tell how much time your Node.js program will take to handle a request. Some requests may be short—perhaps just looping through variables in memory and returning them—whereas others may require time-consuming activities such as opening a file on the system or issuing a query to a database and waiting for the results. How does Node.js handle this uncertainty? The event loop is the answer.

Imagine a chef doing multiple tasks. Baking a cake is one task that requires a lot of time for the oven to cook it. The chef does not stay there waiting for the cake to be ready and then set out to make some coffee. Instead, while the oven bakes the cake, the chef makes coffee and other tasks in parallel. But the cook is always checking whether it is the right time to switch focus to a specific task (making coffee), or to get the cake out of oven.

The event loop is like the chef who is constantly aware of the surrounding activities. In Node.js, an “event-checker” is always checking for operations that have completed or are waiting to be processed by the JavaScript engine.

Using this approach, an asynchronous and long operation does not block other quick operations that come after. This is because the event loop mechanism is always checking whether that long task, such as an I/O operation, is already done. If not, Node.js can continue to process other tasks. Once the background task is complete, the results are returned and the application on top of Node.js can

use a trigger function (callback) to further process the output.

Because Node.js avoids the use of multiple threads, as other environments do, it is called a *single-threaded environment*, and therefore a non-blocking approach is of the utmost importance. This is why Node.js uses an event loop. For compute-intensive tasks, Node.js is not among the best tools, however: there are other programming languages and environments that address these problems more efficiently.

In the following sections, we will take a closer look at callback functions. For now, understand that callback functions are triggers that are executed upon the completion of a predefined operation.

Modules

It is a best practice to break down complex functionality and extensive pieces of code into smaller parts. Doing this modularization helps to better organize the codebase, abstract away the implementations, and avoid complicated engineering problems. To meet those necessities, programmers package blocks of source code to be consumed by other internal or external parts of code.

Consider the example of a program that calculates the volume of a sphere. Open up your text editor and create a file named `volumeCalculator.js` containing the following 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.');
```

Now, execute the file using Node:

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A sphere with radius 6 has a 904.7786842338603 volume.
```

Here, a simple function was used to compute the volume of a sphere, based on its radius. Imagine that we also need to calculate the volume of a cylinder, cone, and so on: we quickly notice that those specific functions must be added to the `volumeCalculator.js` file, which can become a huge collection of functions. To better organize the structure, we can use the idea behind modules as packages of separated code.

In order to do that, create a separated file called `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
}
```

Now, in the `volumeCalculator.js` file, delete the old code and replace it with this snippet:

```
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.');
```

And then run the filename against the Node.js environment:

```
$ 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.
```

In the Node.js environment, every source code file is considered a module, but the word “module” in Node.js indicates a package of code wrapped up as in the previous example. By using modules, we abstracted the volume functions away from the main file, `volumeCalculator.js`, thus reducing its size and making it easier to apply unit tests, which are a good practice when developing real world

applications.

Now that we know how modules are used in Node.js, we can use one of the most important tools: the *Node Package Manager* (NPM).

One of the main jobs of NPM is to manage, download, and install external modules into the project or in the operating system. You can initialize a node repository with the command `npm init`.

NPM will ask the default questions about the name of your repository, version, description, and so on. You can bypass these steps using `npm init --yes`, and the command will automatically generate a `package.json` file that describes the properties of your project/module.

Open the `package.json` file in your favorite text editor and you will see a JSON file containing properties such as keywords, script commands to use with NPM, a name, etc.

One of those properties is the dependencies that are installed in your local repository. NPM will add the name and version of these dependencies into `package.json`, along with `package-lock.json`, another file used as fallback by NPM in case `package.json` fails.

Type the following in your terminal:

```
$ npm i dayjs
```

The `i` flag is a shortcut for the argument `install`. If you are connected to the internet, NPM will search for a module named `dayjs` in the remote repository of Node.js, download the module, and install it locally. NPM will also add this dependency to your `package.json` and `package-lock.json` files. Now you can see that there is a folder called `node_modules`, which contains the installed module along with other modules if they are needed. The `node_modules` directory contains the actual code that is going to be used when the library is imported and called. However, this folder is not saved in versioning systems using Git, since the `package.json` file provides all the dependencies used. Another user can take the `package.json` file and simply run `npm install` in their own machine, where NPM will create a `node_modules` folder with all the dependencies in the `package.json`, thus avoiding version control for the thousands of files available on the NPM repository.

Now that the `dayjs` module is installed in the local directory, open the Node.js console and type the following lines:

```
const dayjs = require('dayjs');
dayjs().format('YYYY MM-DDTHH:mm:ss')
```

The `dayjs` module is loaded with the `require` keyword. When a method from the module is called, the library takes the current system datetime and outputs it in the specified format:

```
2020 11-22T11:04:36
```

This is the same mechanism used in the previous example, where the Node.js runtime loads the third party function into the code.

Server Functionality

Because Node.js controls the back end of web applications, one of its core tasks is to handle HTTP requests.

Here is a summary of how web servers handle incoming HTTP requests. The functionality of the server is to listen for requests, determine as quickly as possible what response each needs, and return that response to the sender of the request. This application must receive an incoming HTTP request triggered by the user, parse the request, perform the calculation, generate the response, and send it back. An HTTP module such as Node.js is used because it simplifies those steps, allowing a web programmer to focus on the application itself.

Consider the following example that implements this very basic functionality:

```
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}/`);
});
```

Save these contents in a file called `basic_server.js` and run it through a `node` command. The terminal running Node.js will display the following message:

```
Server running at http://127.0.0.1:3000/
```

Then visit the following URL in your web browser: `http://127.0.0.1:3000/numbers?a=2&b=17`

Node.js is running a web server in your computer, and using two modules: `http` and `url`. The `http` module sets up a basic HTTP server, processes the incoming web requests, and hands them to our simple application code. The `URL` module parses the arguments passed in the URL, converts them into an integer format, and performs the addition operation. The `http` module then sends the response as text to the web browser.

In a real web application, Node.js is commonly used to process and retrieve data, usually from a database, and return the processed information to the front end to display. But the basic application in this lesson concisely shows how Node.js makes use of modules to handle web requests as a web server.

Guided Exercises

1. What are the reasons to use modules instead of writing simple functions?

2. Why did the Node.js environment become so popular? Cite one characteristic.

3. What is the purpose of the `package.json` file?

4. Why is it not recommended to save and share the `node_modules` folder?

Explorational Exercises

1. How can you execute Node.js applications on your computer?

2. How can you delimit parameters in the URL to parse inside the server?

3. Specify a scenario where a specific task could be a bottleneck for a Node.js application.

4. How would you implement a parameter to multiply or sum the two numbers in the server example?

Summary

This lesson provided an overview of the Node.js environment, its characteristics, and how it can be used to implement simple programs. This lesson includes the following concepts:

- What Node.js is, and why it is used.
- How to run Node.js programs using the command line.
- The event loops and the single thread.
- Modules.
- Node Package Manager (NPM).
- Server functionality.

Answers to Guided Exercises

1. What are the reasons to use modules instead of writing simple functions?

By opting for modules instead of conventional functions, the programmer creates a simpler codebase to read and maintain and for which to write automated tests.

2. Why did the Node.js environment become so popular? Cite two characteristics.

One reason is the flexibility of the JavaScript language, which was already widely used in the front end of web applications. Node.js allows the use of just one programming language in the whole system.

3. What is the purpose of the `package.json` file?

This file contains metadata for the project, such as the name, version, dependencies (libraries), and so on. Given a `package.json` file, other people can download and install the same libraries and run tests in the same way that the original creator did.

4. Why is it not recommended to save and share the `node_modules` folder?

The `node_modules` folder contains the implementations of libraries available in remote repositories. So the better way to share these libraries is to indicate them in the `package.json` file and then use NPM to download those libraries. This method is simpler and more error-free, because you do not have to track and maintain the libraries locally.

Answers to Explorational Exercises

1. How can you execute Node.js applications on your computer?

You can run them by typing `node PATH/FILE_NAME.js` at the command line in your terminal, changing `PATH` to the path of your Node.js file and changing `FILE_NAME.js` to your chosen filename.

2. How can you delimit parameters in the URL to parse inside the server?

The ampersand character `&` is used to delimit those parameters, so that they can be extracted and parsed in the JavaScript code.

3. Specify a scenario where a specific task could be a bottleneck for a Node.js application.

Node.js is not a good environment in which to run CPU intensive processes because it uses a single thread. A numerical computation scenario could slow down and lock the entire application. If a numerical simulation is needed, it is better to use other tools.

4. How would you implement a parameter to multiply or sum the two numbers in the server example?

Use a ternary operator or an if-else condition to check for an additional parameter. If the parameter is the string `mult` return the product of the numbers, else return the sum. Replace the old code with the snippet below. Restart the server in the command line by pressing `Ctrl + C` and rerunning the command to restart the server. Now test the new application by visiting the URL `http://127.0.0.1:3000/numbers?a=2&b=17&operation=mult` in your browser. If you omit or change the last parameter, the results should be the sum of the numbers.

```
let result = queryObject.operation == 'mult' ? parseInt(queryObject.a) *  
parseInt(queryObject.b) : parseInt(queryObject.a) + parseInt(queryObject.b);
```



035.2 NodeJS Express Basics

Reference to LPI objectives

[Web Development Essentials version 1.0, Exam 030, Objective 035.2](#)

Weight

4

Key knowledge areas

- Define routes to static files and EJS templates
- Serve static files through Express
- Serve EJS templates through Express
- Create simple, non-nested EJS templates
- Use the request object to access HTTP GET and POST parameters and process data submitted through HTML forms
- Awareness of user input validation
- Awareness of cross-site Scripting (XSS)
- Awareness of cross-site request forgery (CSRF)

Partial list of the used files, terms and utilities

- express and body-parser modules
- Express app object
- `app.get()`, `app.post()`
- `res.query()`, `res.body()`
- ejs node module

- `res.render()`
- `<% ... %>, <%= ... %>, <%# ... %>, <%- ... %>`
- `views/`



035.2 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	035 NodeJS Server Programming
Objective:	035.2 NodeJS Express Basics
Lesson:	1 of 2

Introduction

Express.js, or simply Express, is a popular framework that runs on Node.js and is used to write HTTP servers that handle requests from web application clients. Express supports many ways to read parameters sent over HTTP.

Initial Server Script

To demonstrate Express's basic features for receiving and handling requests, let's simulate an application that requests some information from the server. In particular, the example server:

- Provides an echo function, which simply returns the message sent by the client.
- Tells the client its IP address upon request.
- Uses cookies to identify known clients.

The first step is to create the JavaScript file that will operate as the server. Using `npm`, create a directory called `myserver` with the JavaScript file:

```
$ mkdir myserver
$ cd myserver/
$ npm init
```

For the entry point, any filename can be used. Here we will use the default filename: `index.js`. The following listing shows a basic `index.js` file that will be used as the entry point for our server:

```
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}`)
})
```

Some important constants for the server configuration are defined in the first lines of the script. The first two, `express` and `app`, correspond to the included `express` module and an instance of this module that runs our application. We will add the actions to be performed by the server to the `app` object.

The other two constants, `host` and `port`, define the host and communication port associated to the server.

If you have a publicly accessible host, use its name instead of `myserver` as the value of `host`. If you don't provide the host name, Express will default to `localhost`, the computer where the application runs. In that case, no outside clients will be able to reach the program, which may be fine for testing but offers little value in production.

The port needs to be provided, or the server will not start.

This script attaches only two procedures to the `app` object: the `app.get()` action that answers requests made by clients through HTTP GET, and the `app.listen()` call, which is required to activate the server and assigns it a host and port.

To start the server, just run the `node` command, providing the script name as an argument:

```
$ node index.js
```

As soon as the message `Server ready at http://myserver:8080` appears, the server is ready to receive requests from an HTTP client. Requests can be made from a browser on the same computer where the server is running, or from another machine that can access the server.

All transaction details we'll see here are shown in the browser if you open a window for the developer console. Alternatively, the `curl` command can be used for HTTP communication and allows you to inspect connection details more easily. If you are not familiar with the shell command line, you can create an HTML form to submit requests to a server.

The following example shows how to use the `curl` command on the command line to make an HTTP request to the newly deployed server:

```
$ 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
```

The `-v` option of the `curl` command displays all the request and response headers, as well as other debugging information. The lines starting with `>` indicate the request headers sent by the client and the lines starting with `<` indicate the response headers sent by the server. Lines starting with `*` are information generated by `curl` itself. The content of the response is displayed only at the end, which in this case is the line `Request received`.

The service's URL, which in this case contains the server's hostname and port (`http://myserver:8080`), were given as arguments to the `curl` command. Because no directory or filename is given, these default to the root directory `/`. The slash turns up as the request file in the `> GET / HTTP/1.1` line, which is followed in the output by the hostname and port.

In addition to displaying HTTP connection headers, the `curl` command assists application development by allowing you to send data to the server using different HTTP methods and in different formats. This flexibility makes it easier to debug any problems and implement new features on the server.

Routes

The requests the client can make to the server depend on what *routes* have been defined in the `index.js` file. A route specifies an HTTP method and defines a *path* (more precisely, a URI) that can be requested by the client.

So far, the server has only one route configured:

```
app.get('/', (req, res) => {
  res.send('Request received')
})
```

Even though it is a very simple route, simply returning a plain text message to the client, it is enough to identify the most important components that are used to structure most routes:

- The HTTP method served by the route. In the example, the HTTP GET method is indicated by the `get` property of the `app` object.
- The path served by the route. When the client does not specify a path for the request, the server uses the root directory, which is the base directory set aside for use by the web server. A later example in this chapter uses the path `/echo`, which corresponds to a request made to `myserver:8080/echo`.
- The function executed when the server receives a request on this route, usually written in abbreviated form as an *arrow function* because the syntax `=>` points to the definition of the nameless function. The `req` parameter (short for “request”) and `res` parameter (short for “response”) give details about the connection, passed to the function by the `app` instance itself.

POST Method

To extend the functionality of our test server, let's see how to define a route for the HTTP POST method. It's used by clients when they need to send extra data to the server beyond those included

in the request header. The `--data` option of the `curl` command automatically invokes the POST method, and includes content that will be sent to the server via POST. The POST / HTTP/1.1 line in the following output shows that the POST method was used. However, our server defined only a GET method, so an error occurs when we use `curl` to send a request 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
```

In the previous example, running `curl` with the parameter `--data message="This is the POST request body"` is equivalent to submitting a form containing the text field named `message`, filled with `This is the POST request body`.

Because the server is configured with only one route for the `/` path, and that route only responds to the HTTP GET method, so the response header has the line `HTTP/1.1 404 Not Found`. In addition, Express automatically generated a short HTML response with the warning `Cannot POST`.

Having seen how to generate a POST request through `curl`, let's write an Express program that can successfully handle the request.

First, note that the `Content-Type` field in the request header says that the data sent by the client is in the `application/x-www-form-urlencoded` format. Express does not recognize that format by default, so we need to use the `express.urlencoded` module. When we include this module, the `req` object—passed as a parameter to the handler function—has the `req.body.message` property set, which corresponds to the `message` field sent by the client. The module is loaded with `app.use`, which should be placed before the declaration of routes:

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.urlencoded({ extended: true }))
```

Once this is done, it would be enough to change `app.get` to `app.post` in the existing route to fulfill requests made via POST and to recover the request body:

```
app.post('/', (req, res) => {
  res.send(req.body.message)
})
```

Instead of replacing the route, another possibility would be to simply add this new route, because Express identifies the HTTP method in the request header and uses the appropriate route. Because we are interested in adding more than one functionality to this server, it is convenient to separate each one with its own path, such as `/echo` and `/ip`.

Path and Function Handler

Having defined which HTTP method will respond to the request, we now need to define a specific path for the resource and a function that processes and generates a response to the client.

To expand the `echo` functionality of the server, we can define a route using the POST method with the path `/echo`:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

The `req` parameter of the handler function contains all the request details stored as properties. The content of the `message` field in the request body is available in the `req.body.message` property. The example simply sends this field is sent back to the client through the `res.send(req.body.message)` call.

Remember that the changes you make take effect only after the server is restarted. Because you are running the server from a terminal window during the examples in this chapter, you can shut down the server by pressing `Ctrl+C` on that terminal. Then rerun the server through the `node index.js` command. The response obtained by the client to the `curl` request we showed before is now successful:

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
This is the POST request body
```

Other Ways to Pass and Return Information in a GET Request

It might be excessive to use the HTTP POST method if only short text messages like the one used in the example will be sent. In such cases, data can be sent in a *query string* that starts with a question mark. Thus, the string `?message=This+is+the+message` could be included within the request path of the HTTP GET method. The fields used in the query string are available to the server in the `req.query` property. Therefore, a field named `message` is available in the `req.query.message` property.

Another way to send data via the HTTP GET method is to use Express's *route parameters*:

```
app.get('/echo/:message', (req, res) => {
  res.send(req.params.message)
})
```

The route in this example matches requests made with the GET method using the path `/echo/:message`, where `:message` is a placeholder for any term sent with that label by the client. These parameters are accessible in the `req.params` property. With this new route, the server's `echo` function can be requested more succinctly by the client:

```
$ curl http://myserver:8080/echo/hello
hello
```

In other situations, the information the server needs to process the request do not need to be explicitly provided by the client. For instance, the server has another way to retrieve the client's public IP address. That information is present in the `req` object by default, in the `req.ip` property:

```
app.get('/ip', (req, res) => {
  res.send(req.ip)
})
```

Now the client can request the `/ip` path with the GET method to find its own public IP address:

```
$ curl http://myserver:8080/ip
187.34.178.12
```

Other properties of the `req` object can be modified by the client, especially the request headers available in `req.headers`. The `req.headers.user-agent` property, for example, identifies which program is making the request. Although it is not common practice, the client can change the contents of this field, so the server should not use it to reliably identify a particular client. It is even more important to validate the data explicitly provided by the client, to avoid inconsistencies in boundaries and formats that could adversely affect the application.

Adjustments to the Response

As we've seen in previous examples, the `res` parameter is responsible for returning a response to the client. Furthermore, the `res` object can change other aspects of the response. You may have noticed that, although the responses we've implemented so far are just brief plain text messages, the `Content-Type` header of the responses is using `text/html; charset=utf-8`. Although this does not prevent the plain text response from being accepted, it will be more correct if we redefine this field in the response header to `text/plain` with the setting `res.type('text/plain')`.

Other types of response adjustments involve using *cookies*, which allow the server to identify a client that has previously made a request. Cookies are important for advanced features, such as creating private sessions that associate requests to a specific user, but here we'll just look at a simple example of how to use a cookie to identify a client that has previously accessed the server.

Given the modularized design of Express, cookie management must be installed with the `npm` command before being used in the script:

```
$ npm install cookie-parser
```

After installation, cookie management must be included in the server script. The following definition should be included near the beginning of the file:

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

To illustrate the use of cookies, let's modify the route's handler function with the `/` root path that already exists in the script. The `req` object has a `req.cookies` property, where cookies sent in the request header are kept. The `res` object, on the other hand, has a `res.cookie()` method that creates a new cookie to be sent to the client. The handler function in the following example checks whether a cookie with the name `known` exists in the request. If such a cookie does not exist, the server assumes that this is a first-time visitor and sends it a cookie with that name through the `res.cookie('known', '1')` call. We arbitrarily assign the value `1` to the cookie because it is supposed to have some content, but the server doesn't consult that value. This application just assumes that the simple presence of the cookie indicates that the client has already requested this route before:

```
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!');
})
```

By default, `curl` does not use cookies in transactions. But it has options to store (`-c cookies.txt`) and send stored cookies (`-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-l7qrxcqic14xv6EfA5fZFWCFrgY"
< Date: Sat, 03 Jul 2021 23:45:03 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome, new visitor!
```

Because this command was the first access since cookies were implemented on the server, the client did not have any cookies to include in the request. As expected, the server did not identify the cookie in the request and therefore included the cookie in the response headers, as indicated in the `Set-Cookie: known=1; Path=/` line of the output. Since we have enabled cookies in `curl`, a new request will include the cookie `known=1` in the request headers, allowing the server to identify the cookie's presence:

```
$ 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-ATq2fIQYtLMYIUpJwwpb5SjV9lw"
< Date: Sat, 03 Jul 2021 23:45:47 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome back, visitor
```

Cookie Security

The developer should be aware of potential vulnerabilities when using cookies to identify clients making requests. Attackers can use techniques such as *cross-site scripting* (XSS) and *cross-site request forgery* (CSRF) to steal cookies from a client and thereby impersonate them when making a request to the server. Generally speaking, these types of attacks use non-validated comment fields or meticulously constructed URLs to insert malicious JavaScript code into the page. When executed by an authentic client, this code can copy valid cookies and store them or forward them to another destination.

Therefore, especially in professional applications, it is important to install and use more specialized Express features, known as *middleware*. The `express-session` or `cookie-session` module provide more complete and secure control over session and cookie management. These components enable extra controls to prevent cookies from being diverted from their original issuer.

Guided Exercises

1. How can the content of the `comment` field, sent within a query string of the HTTP GET method, be read in a handler function?

2. Write a route that uses the HTTP GET method and the `/agent` path to send back to the client the contents of the `user-agent` header.

3. Express.js has a feature called *route parameters*, where a path such as `/user/:name` can be used to receive the `name` parameter sent by the client. How can the `name` parameter be accessed within the handler function of the route?

Explorational Exercises

1. If a server's host name is `myserver`, which Express route would receive the submission in the form below?

```
<form action="/contact/feedback" method="post"> ... </form>
```

2. During server development, the programmer is not able to read the `req.body` property, even after verifying that the client is correctly sending the content via the HTTP `POST` method. What is a likely cause for this problem?

3. What happens when the server has a route set to the path `/user/:name` and the client makes a request to `/user/?`

Summary

This lesson explains how to write Express scripts to receive and handle HTTP requests. Express uses the concept of *routes* to define the resources available to clients, which gives you great flexibility to build servers for any kind of web application. This lesson goes through the following concepts and procedures:

- Routes that use the HTTP GET and HTTP POST methods.
- How form data is stored in the `request` object.
- How to use route parameters.
- Customizing response headers.
- Basic cookie management.

Answers to Guided Exercises

1. How can the content of the `comment` field, sent within a query string of the HTTP GET method, be read in a handler function?

The `comment` field is available in the `req.query.comment` property.

2. Write a route that uses the HTTP GET method and the `/agent` path to send back to the client the contents of the `user-agent` header.

```
app.get('/agent', (req, res) => {
  res.send(req.headers.user-agent)
})
```

3. Express.js has a feature called *route parameters*, where a path such as `/user/:name` can be used to receive the `name` parameter sent by the client. How can the `name` parameter be accessed within the handler function of the route?

The `name` parameter is accessible in the `req.params.name` property.

Answers to Explorational Exercises

1. If a server's host name is `myserver`, which Express route would receive the submission in the form below?

```
<form action="/contact/feedback" method="post"> ... </form>
```

```
app.post('/contact/feedback', (req, res) => {  
  ...  
})
```

2. During server development, the programmer is not able to read the `req.body` property, even after verifying that the client is correctly sending the content via the HTTP POST method. What is a likely cause for this problem?

The programmer did not include the `express.urlencoded` module, which lets Express extract the body of a request.

3. What happens when the server has a route set to the path `/user/:name` and the client makes a request to `/user/?`

The server will issue a `404 Not Found` response, because the route requires the `:name` parameter to be provided by the client.



035.2 Lesson 2

Certificate:	Web Development Essentials
Version:	1.0
Topic:	035 NodeJS Server Programming
Objective:	035.2 NodeJS Express Basics
Lesson:	2 of 2

Introduction

Web servers have very versatile mechanisms to produce responses to client requests. For some requests, it's enough for the web server to provide a static, unprocessed response, because the requested resource is the same for any client. For instance, when a client requests an image that is accessible to everyone, it is enough for the server to send the file containing the image.

But when responses are dynamically generated, they may need to be better structured than simple lines written in the server script. In such cases, it is convenient for the web server to be able to generate a complete document, which can be interpreted and rendered by the client. In the context of web application development, HTML documents are commonly created as templates and kept separate from the server script, which inserts dynamic data in predetermined places in the appropriate template and then sends the formatted response to the client.

Web applications often consume both static and dynamic resources. An HTML document, even if it was dynamically generated, may have references to static resources such as CSS files and images. To demonstrate how Express helps handle this kind of demand, we'll first set up an example server that delivers static files and then implement routes that generate structured, template-based responses.

Static Files

The first step is to create the JavaScript file that will run as a server. Let's follow the same pattern covered in previous lessons to create a simple Express application: first create a directory called `server` and then install the base components with the `npm` command:

```
$ mkdir server
$ cd server/
$ npm init
$ npm install express
```

For the entry point, any filename can be used, but here we will use the default filename: `index.js`. The following listing shows a basic `index.js` file that will be used as a starting point for our server:

```
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}`)
})
```

You don't have to write explicit code to send a static file. Express has middleware for this purpose, called `express.static`. If your server needs to send static files to the client, just load the `express.static` middleware at the beginning of the script:

```
app.use(express.static('public'))
```

The `public` parameter indicates the directory that stores files the client can request. Paths requested by clients must not include the `public` directory, but only the filename, or the path to the file relative to the `public` directory. To request the `public/layout.css` file, for example, the client makes a request to `/layout.css`.

Formatted Output

While sending static content is straightforward, dynamically generated content can vary widely. Creating dynamic responses with short messages makes it easy to test applications in their initial stages of development. For example, the following is a test route that just send back to the client a

message that it sent by the HTTP POST method. The response can just replicate the message content in plain text, without any formatting:

```
app.post('/echo', (req, res) => {
  res.send(req.body.message)
})
```

A route like this is a good example to use when learning Express and for diagnostics purposes, where a raw response sent with `res.send()` is enough. But a useful server must be able to produce more complex responses. We'll move on now to develop that more sophisticated type of route.

Our new application, instead of just sending back the contents of the current request, maintains a complete list of the messages sent in previous requests by each client and sends back each client's list when requested. A response merging all messages is an option, but other formatted output modes are more appropriate, especially as responses become more elaborate.

To receive and store client messages sent during the current session, first we need to include extra modules for handling cookies and data sent via the HTTP POST method. The only purpose of the following example server is to log messages sent via `POST` and display previously sent messages when the client issues a `GET` request. So there are two routes for the `/` path. The first route fulfills requests made with the HTTP `POST` method and the second fulfills requests made with the HTTP `GET` method:

```
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}`)
})

```

We kept the static files configuration at the top, because it will soon be useful to provide static files such as `layout.css`. In addition to the `cookie-parser` middleware introduced in the previous chapter, the example also includes the `uuid` middleware to generate a unique identification number passed as a cookie to each client that sends a message. If not already installed in the example server directory, these modules can be installed with the command `npm install cookie-parser uuid`.

The global array called `messages` stores the messages sent by all clients. Each item in this array consists of an object with the properties `uuid` and `message`.

What's really new in this script is the `res.json()` method, used at the end of the two routes to generate a response in JSON format with the array containing the messages already sent by the client:

```
// Send back JSON response
res.json(user_entries)
```

JSON is a plain text format that allows you to group a set of data into a single structure that is associative: that is, content is expressed as keys and values. JSON is particularly useful when responses are going to be processed by the client. Using this format, a JavaScript object or array can be easily reconstructed on the client side with all the properties and indexes of the original object on the server.

Because we are structuring each message in JSON, we refuse requests that do not contain `application/json` in their `accept` header:

```
// Only JSON enabled requests
if ( req.headers.accept != "application/json" )
{
  res.sendStatus(404)
  return
}
```

A request made with a plain `curl` command to insert a new message will not be accepted, because `curl` by default does not specify `application/json` in the `accept` header:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt
Not Found
```

The `-H "accept: application/json"` option changes the request header to specify the response's format, which this time will be accepted and answered in the specified format:

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt -H "accept: application/json"
["My first message"]
```

Getting messages using the other route is done in a similar way, but this time using the HTTP GET method:

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -H "accept: application/json"
["Another message", "My first message"]
```

Templates

Responses in formats such as JSON are convenient for communicating between programs, but the main purpose of most web application servers is to produce HTML content for human consumption. Embedding HTML code within JavaScript code is not a good idea, because mixing languages in the same file makes the program more susceptible to errors and harms the maintenance of the code.

Express can work with different *template engines* that separate out the HTML for dynamic content; the full list can be found at the [Express template engines](#) site. One of the most popular template engines is *Embedded JavaScript* (EJS), which allows you to create HTML files with specific tags for dynamic content insertion.

Like other Express components, EJS needs to be installed in the directory where the server is running:

```
$ npm install ejs
```

Next, the EJS engine must be set as the default renderer in the server script (near the beginning of the `index.js` file, before the route definitions):

```
app.set('view engine', 'ejs')
```

The response generated with the template is sent to the client with the `res.render()` function, which receives as parameters the template file name and an object containing values that will be accessible from within the template. The routes used in the previous example can be rewritten to generate HTML responses as well as 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 that the format of the response depends on the `accept` header found in the request:

```
if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})
```

A response in JSON format is sent only if the client explicitly requests it. Otherwise, the response is generated from the `index` template. The same `user_entries` array feeds both the JSON output and the template, but the object used as a parameter for the latter also has the `title: "My messages"` property, which will be used as a title inside the template.

HTML Templates

Like static files, the files containing HTML templates reside in their own directory. By default, EJS assumes the template files are in the `views/` directory. In the example, a template named `index` was used, so EJS looks for the `views/index.ejs` file. The following listing is the content of a simple `views/index.ejs` template that can be used with the example code:

```

<!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>

```

The first special EJS tag is the `<title>` element in the `<head>` section:

```
<%= title %>
```

During the rendering process, this special tag will be replaced by the value of the `title` property of the object passed as a parameter to the `res.render()` function.

Most of the template is made up of conventional HTML code, so the template contains the HTML form for sending new messages. The test server responds to the HTTP GET and POST methods for the same path `/`, hence the `action="/"` and `method="post"` attributes in the form tag.

Other parts of the template are a mixture of HTML code and EJS tags. EJS has tags for specific

purposes within the template:

`<% ... %>`

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) => { %>`

`<%# ... %>`

Defines a comment, whose content is ignored by the parser. Unlike comments written in HTML, these comments are not visible to the client.

`<%= ... %>`

Inserts the escaped content of the variable. It is important to escape unknown content to avoid JavaScript code execution, which can open loopholes for cross-site Scripting (XSS) attacks. Example: `<%= title %>`

`<%- ... %>`

Inserts the content of the variable without escaping.

The mix of HTML code and EJS tags is evident in the snippet where client messages are rendered as an HTML list:

```
<ul>
<% messages.forEach( message ) => { %>
<li><%= message %></li>
<% } %>
</ul>
```

In this snippet, the first `<% ... %>` tag starts a `forEach` statement that loops through all the elements of the `messages` array. The `<%` and `%>` delimiters let you control the snippets of HTML. A new HTML list item, `<%= message %>`, will be produced for each element of `messages`. With these changes, the server will send the response in HTML when a request like the following is received:

```
$ 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>
```

The separation between the code for processing the requests and the code for presenting the response makes the code cleaner and allows a team to divide application development between people with distinct specialties. A web designer, for example, can focus on the template files in `views/` and related stylesheets, which are provided as static files stored in the `public/` directory on the example server.

Guided Exercises

1. How should `express.static` be configured so that clients can request files in the `assets` directory?

2. How can the response's type, which is specified in the request's header, be identified within an Express route?

3. Which method of the `res` (response) route parameter generates a response in JSON format from a JavaScript array called `content`?

Explorational Exercises

1. By default, Express template files are in the `views` directory. How can this setting be modified so that template files are stored in `templates`?

2. Suppose a client receives an HTML response with no title (i.e. `<title></title>`). After verifying the EJS template, the developer finds the `<title><% title %></title>` tag in the head section of the file. What is the likely cause of the problem?

3. Use EJS template tags to write a `<h2></h2>` HTML tag with the contents of the JavaScript variable `h2`. This tag should be rendered only if the `h2` variable is not empty.

Summary

This lesson covers the basic methods Express.js provides to generate static and formated yet dynamic responses. Little effort is required to set up an HTTP server for static files and the EJS templating system provides an easy way for generating dynamic content from HTML files. This lesson goes through the following concepts and procedures:

- Using `express.static` for static file responses.
- How to create a response to match the content type field in the request header.
- JSON-structured responses.
- Using EJS tags in HTML based templates.

Answers to Guided Exercises

1. How should `express.static` be configured so that clients can request files in the `assets` directory?

A call to `app.use(express.static('assets'))` should be added to the server script.

2. How can the response's type, which is specified in the request's header, be identified within an Express route?

The client sets acceptable types in the `accept` header field, which is mapped to the `req.headers.accept` property.

3. Which method of the `res` (response) route parameter generates a response in JSON format from a JavaScript array called `content`?

The `res.json()` method: `res.json(content)`.

Answers to Explorational Exercises

1. By default, Express template files are in the `views` directory. How can this setting be modified so that template files are stored in `templates`?

The directory can be defined in the initial script settings with `app.set('views', './templates')`.

2. Suppose a client receives an HTML response with no title (i.e. `<title></title>`). After verifying the EJS template, the developer finds the `<title><% title %></title>` tag in the `head` section of the file. What is the likely cause of the problem?

The `<%= %>` tag should be used to enclose the contents of a variable, as in `<%= title %>`.

3. Use EJS template tags to write a `<h2></h2>` HTML tag with the contents of the JavaScript variable `h2`. This tag should be rendered only if the `h2` variable is not empty.

```
<% if ( h2 != "" ) { %>
<h2><%= h2 %></h2>
<% } %>
```



035.3 SQL Basics

Reference to LPI objectives

Web Development Essentials version 1.0, Exam 030, Objective 035.3

Weight

3

Key knowledge areas

- Establish a database connection from NodeJS
- Retrieve data from the database in NodeJS
- Execute SQL queries from NodeJS
- Create simple SQL queries excluding joins
- Understand primary keys
- Escape variables used in SQL queries
- Awareness of SQL injections

Partial list of the used files, terms and utilities

- `sqlite3` NPM module
- `Database.run()`, `Database.close()`, `Database.all()`, `Database.get()`, `Database.each()`
- `CREATE TABLE`
- `INSERT`, `SELECT`, `DELETE`, `UPDATE`



035.3 Lesson 1

Certificate:	Web Development Essentials
Version:	1.0
Topic:	035 NodeJS Server Programming
Objective:	035.3 SQL Basics
Lesson:	1 of 1

Introduction

Although you can write your own functions to implement persistent storage, it may be more convenient to use a database management system to speed up development and ensure better security and stability for table-formatted data. The most popular strategy for storing data organized in interrelated tables, especially when those tables are heavily queried and updated, is to install a relational database that supports *Structured Query Language* (SQL), a language geared towards relational databases. Node.js supports various SQL database management systems. Following the principles of portability and user space execution adopted by Node.js Express, SQLite is an appropriate choice for persistent storage of data used by this type of HTTP server.

SQL

The Structured Query Language is specific to databases. Write and read operations are expressed in sentences called *statements* and *queries*. Both statements and queries are made up of *clauses*, which define the conditions for executing the operation.

Names and email addresses, for example, can be stored in a database table that contains name and email fields. A database can contain several tables, so each table must have a unique name. If we use

the name `contacts` for the names and emails table, a new record can be inserted with the following *statement*:

```
INSERT INTO contacts (name, email) VALUES ("Carol", "carol@example.com");
```

This insertion statement is composed of the `INSERT INTO` clause, which defines the table and fields where the data will be inserted. The second clause, `VALUES`, sets the values that will be inserted. It is not necessary to capitalize clauses, but it is common practice, so as to better recognize SQL keywords within a statement or query.

A query on the `contacts` table is done in a similar way, but using the `SELECT` clause:

```
SELECT email FROM contacts;
dave@example.com
carol@example.com
```

In this case, the `SELECT email` clause selects one field from entries in the `contacts` table. The `WHERE` clause restricts the query to specific rows:

```
SELECT email FROM contacts WHERE name = "Dave";
dave@example.com
```

SQL has many other clauses, and we'll look at some of them in later sections. But first it is necessary to see how to integrate the SQL database with Node.js.

SQLite

SQLite is probably the simplest solution for incorporating SQL database features into an application. Unlike other popular database management systems, SQLite is not a database server to which a client connects. Instead, SQLite provides a set of functions that allow the developer to create a database like a conventional file. In the case of an HTTP server implemented with Node.js Express, this file is usually located in the same directory as the server script.

Before using SQLite in Node.js, you need to install the `sqlite3` module. Run the following command in the server's installation directory; i.e., the directory containing the Node.js script you will run.

```
$ npm install sqlite3
```

Be aware that there are several modules that support SQLite, such as `better-sqlite3`, whose usage is subtly different from `sqlite3`. The examples in this lesson are for the `sqlite3` module, so they might not work as expected if you choose another module.

Opening the Database

To demonstrate how a Node.js Express server can work with an SQL database, let's write a script that stores and displays messages sent by a client identified by a cookie. Messages are sent by the client via the HTTP POST method and the server response can be formatted as JSON or HTML (from a template), depending on the format requested by the client. This lesson won't go into detail about using HTTP methods, cookies, and templates. The code snippets shown here assume that you already have a Node.js Express server where these features are configured and available.

The simplest way to store the messages sent by the client is to store them in a global array, where each message previously sent is associated with a unique identification key for each client. This key can be sent to the client as a cookie, which is presented to the server on future requests to retrieve its previous messages.

However, this approach has a weakness: because messages are stored only in a global array, all messages will be lost when the current server session is terminated. This is one of the advantages of working with databases, because the data is persistently stored and is not lost if the server is restarted.

Using the `index.js` file as the main server script, we can incorporate the `sqlite3` module and indicate the file that serves as the database, as follows:

```
const sqlite3 = require('sqlite3')
const db = new sqlite3.Database('messages.sqlite3');
```

If it doesn't already exist, the `messages.sqlite3` file will be created in the same directory as the `index.js` file. Inside this single file, all structures and respective data will be stored. All database operations performed in the script will be intermediated by the `db` constant, which is the name given to the new `sqlite3` object that opens the `messages.sqlite3` file.

Structure of a Table

No data can be inserted into the database until at least one table is created. Tables are created with the statement `CREATE TABLE`:

```
db.run('CREATE TABLE IF NOT EXISTS messages (id INTEGER PRIMARY KEY AUTOINCREMENT,  
uuid CHAR(36), message TEXT)')
```

The `db.run()` method is used to execute SQL statements in the database. The statement itself is written as a parameter for the method. Although SQL statements must end with a semicolon when entered in a command-line processor, the semicolon is optional in statements passed as parameters in a program.

Because the `run` method will be performed every time the script is executed with `node index.js`, the SQL statement includes the conditional clause `IF NOT EXISTS` to avoid errors in future executions when the `messages` table already exists.

The fields that make up the `messages` table are `id`, `uuid`, and `message`. The `id` field is a unique integer used to identify each entry in the table, so it is created as `PRIMARY KEY`. Primary keys cannot be null and there cannot be two identical primary keys in the same table. Therefore, almost every SQL table has a primary key in order to track the table's contents. Although it is possible to explicitly choose the value for the primary key of a new record (provided it does not yet exist in the table), it is convenient for the key to be generated automatically. The `AUTOINCREMENT` flag in the `id` field is used for this purpose.

Explicit setting of primary keys in SQLite is optional, because SQLite itself creates a primary key automatically. As stated in the SQLite documentation: “In SQLite, table rows normally have a 64-bit signed integer `ROWID` which is unique among all rows in the same table. If a table contains a column of type `INTEGER PRIMARY KEY`, then that column becomes an alias for the `ROWID`. You can then access the `ROWID` using any of four different names, the original three names described above, or the name given to the `INTEGER PRIMARY KEY` column. All these names are aliases for one another and work equally well in any context.”

NOTE

The `uuid` and `message` fields store the client identification and message content, respectively. A field of type `CHAR(36)` stores a fixed amount of 36 characters, and a field of type `TEXT` stores texts of arbitrary length.

Data Entry

The main function of our example server is to store messages that are linked to the client that sent them. The client sends the message in the `message` field in the body of the request sent with the HTTP POST method. The client's identification is in a cookie called `uuid`. With this information, we can write the Express route to insert new messages in the database:

```

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 occurs, 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})

  })
}

})

```

This time, the `db.run()` method executes an insert statement, but note that the `uuid` and `req.body.message` are not written directly into the statement line. Instead, question marks were substituted for the values. Each question mark corresponds to a parameter that follows the SQL statement in the `db.run()` method.

Using question marks as placeholders in the statement that is executed in the database makes it easier for SQLite to distinguish between the static elements of the statement and its variable data. This strategy allows SQLite to *escape* or *sanitize* the variable contents that are part of the statement, preventing a common security breach called *SQL injection*. In that attack, malicious users insert SQL statements into the variable data in the hope that the statements will be executed inadvertently; sanitizing foils the attack by disabling dangerous characters in the data.

Queries

As shown in the sample code, our intent is to use the same route to insert new messages into the

database and to generate the list of previously sent messages. The `db.all()` method returns the collection of all entries in the table that match the criteria defined in the query.

Unlike the statements performed by `db.run()`, `db.all()` generates a list of records that are handled by the arrow function designated in the last parameter:

```
(err, rows) => {}
```

This function, in turn, takes two parameters: `err` and `rows`. The `err` parameter will be used if an error occurs that prevents the execution of the query. Upon success, all records are available in the `rows` array, where each element is an object corresponding to a single record from the table. The properties of this object correspond to the field names indicated in the query: `uuid` and `message`.

The `rows` array is a JavaScript data structure. As such, it can be used to generate responses with methods provided by Express, such as `res.json()` and `res.render()`. When rendered inside an EJS template, a conventional loop can list all records:

```
<ul>
<% rows.forEach( (row) => { %>
<li><strong><%= row.id %></strong>: <%= row.message %></li>
<% }) %>
</ul>
```

Instead of filling the `rows` array with all the records returned by the query, in some cases it might be more convenient to treat each record individually with the `db.each()` method. The `db.each()` method syntax is similar to the `db.all()` method, but the `row` parameter in `(err, row) => {}` matches a single record at a time.

Changing the Contents of the Database

So far, our client can only add and query messages on the server. Since the client now knows the `id` of the previously sent messages, we can implement a function to modify a specific record. The modified message can also be sent to an HTTP POST method route, but this time with a route parameter to catch the `id` given by the client in the request path:

```
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)

      })
    }
  })
})
```

This route demonstrates how to use the `UPDATE` and `WHERE` clauses to modify an existing record. An important difference from the previous examples is the use of *named parameters*, where values are bundled into a single object (`param`) and passed to the `db.run()` method instead of specifying each value by itself. In this case, the field names (preceded by `$`) are the object's properties. Named parameters allow the use of field names (preceded by `$`) as placeholders instead of question marks.

A statement like the one in the example will not cause any modification to the database if the condition imposed by the WHERE clause fails to match some record in the table. To evaluate whether any records were modified by the statement, a callback function can be used as the last parameter of

the `db.run()` method. Inside the function, the number of changed records can be queried from `this.changes`. Note that arrow functions cannot be used in this case, because only regular functions of the form `function(){}` define the `this` object.

Removing a record is very similar to modifying it. We can, for example, continue using the `:id` route parameter to identify the message to be deleted, but this time in a route invoked by the client's HTTP DELETE method:

```
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)
    })
  }
})
```

Records are deleted from a table with the `DELETE FROM` clause. We again used the callback function to evaluate how many entries have been removed from the table.

Closing the Database

Once defined, the `db` object can be referenced at any time during script execution, because the database file remains open throughout the current session. It is not common to close the database while the script is running.

A function to close the database is useful, however, to avoid abruptly closing the database when the

server process finishes. Although unlikely, abruptly shutting down the database can result in inconsistencies if in-memory data is not yet committed to the file. For instance, an abrupt database shutdown with data loss can occur if the script is terminated by the user by pressing the ***Ctrl* + *C*** keyboard shortcut.

In the ***Ctrl* + *C*** scenario just described, the `process.on()` method can intercept signals sent by the operating system and execute an orderly shutdown of both the database and the server:

```
process.on('SIGINT', () => {
  db.close()
  server.close()
  console.log('HTTP server closed')
})
```

The ***Ctrl* + *C*** shortcut invokes the SIGINT operating system signal, which terminates a foreground program in the terminal. Before ending the process when receiving the SIGINT signal, the system invokes the callback function (the last parameter in the `process.on()` method). Inside the callback function, you can put any cleanup code, in particular the `db.close()` method to close the database and `server.close()`, which gracefully closes the Express instance itself.

Guided Exercises

1. What is the purpose of a primary key in a SQL database table?

2. What is the difference between querying using `db.all()` and `db.each()`?

3. Why is it important to use placeholders and not include data sent by the client directly in an SQL statement or query?

Explorational Exercises

1. Which method in the `sqlite3` module can be used to return only one table entry, even if the query matches multiple entries?

2. Suppose the `rows` array was passed as a parameter to a callback function and contains the result of a query made with `db.all()`. How can a field called `price`, which is present in the first position of `rows`, be referenced inside the callback function?

3. The `db.run()` method executes database modification statements, such as `INSERT INTO`. After inserting a new record into a table, how could you retrieve the primary key of the newly inserted record?

Summary

This lesson covers the basic use of SQL databases within Node.js Express applications. The `sqlite3` module offers a simple way of storing persistent data in a SQLite database, where a single file contains the entire database and does not require a specialized database server. This lesson goes through the following concepts and procedures:

- How to establish a database connection from Node.js.
- How to create a simple table and the role of primary keys.
- Using the `INSERT INTO` SQL statement to add new data from within the script.
- SQL queries using standard SQLite methods and callback functions.
- Changing data in the database using `UPDATE` and `DELETE` SQL statements.

Answers to Guided Exercises

1. What is the purpose of a primary key in a SQL database table?

The primary key is the unique identification field for each record within a database table.

2. What is the difference between querying using `db.all()` and `db.each()`?

The `db.all()` method invokes the callback function with a single array containing all entries corresponding to the query. The `db.each()` method invokes the callback function for each result row.

3. Why is it important to use placeholders and not include data sent by the client directly in an SQL statement or query?

With placeholders, user-submitted data is escaped before being included in the query or statement. This hampers SQL injection attacks, where SQL statements are placed inside variable data in an attempt to perform arbitrary operations on the database.

Answers to Explorational Exercises

1. Which method in the `sqlite3` module can be used to return only one table entry, even if the query matches multiple entries?

The `db.get()` method has the same syntax as `db.all()`, but returns only the first entry corresponding to the query.

2. Suppose the `rows` array was passed as a parameter to a callback function and contains the result of a query made with `db.all()`. How can a field called `price`, which is present in the first position of `rows`, be referenced inside the callback function?

Each item in `rows` is an object whose properties correspond to database field names. So the value of the `price` field in the first result is in `rows[0].price`.

3. The `db.run()` method executes database modification statements, such as `INSERT INTO`. After inserting a new record into a table, how could you retrieve the primary key of the newly inserted record?

A regular function of the form `function(){}()` can be used as the callback function of the `db.run()` method. Inside it, the `this.lastID` property contains the primary key value of the last inserted record.

Imprint

© 2022 by Linux Professional Institute: Learning Materials, “Web Development Essentials (030) (Version 1.0)”.

PDF generated: 2022-04-21

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0). To view a copy of this license, visit

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

While Linux Professional Institute has used good faith efforts to ensure that the information and instructions contained in this work are accurate, Linux Professional Institute disclaims all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

The LPI Learning Materials are an initiative of Linux Professional Institute (<https://lpi.org>). Learning Materials and their translations can be found at <https://learning.lpi.org>.

For questions and comments on this edition as well as on the entire project write an email to: learning@lpi.org.