Course Guide

Developing Node.js Applications on IBM Cloud

Course code SANOD ERC 3.0

IBM Training

July 2019 edition

Contents

Trademarks

Course description

Developing Node.js Applications on IBM Cloud

Duration: 2 days

Purpose

The Cloud Application Developer career path prepares students to develop, build, deploy, and test applications that use a cloud platform to build Software as a Service (SaaS) solutions.

Module II, *Cloud Application Developer,* consists of two courses:

- Course I - Essentials of Cloud Application Development.
- Course II - Developing Node.js Applications on IBM Cloud.

This book is the course guide for *Course II - Developing Node.js Applications on IBM Cloud*. This course explains how to create various applications based on Node.js, the Express framework, ECMAScript, and React. It explains how to deploy and run these applications on IBM Cloud.

Audience

Undergraduate senior students from IT related academic programs, for example, computer science, software engineering, information systems and others.

Prerequisites

Before attending *Course II - Developing Node.js Applications on IBM Cloud*, students must meet the following prerequisites:

- Successful completion of Module I *Cloud Application Foundations* (self-study).
- Successful completion of Exercise 0, *Setting up your hands-on environment* (self-study).
- Successful completion of Module II, Course I *Essentials of Cloud Application Development (classroom)*.

Objectives

After completing this course, you should be able to:

- Describe the origin and purpose of the Node.js JavaScript framework.
- Write a simple web server with JavaScript.
- Import Node.js modules into an application.
- Explain synchronous and asynchronous calls.
- Write asynchronous calls code in Node.js applications.
- Explain request flows that are sent to Node.js applications that use the http module.

- Explain the difference between code that is written in “pure” JavaScript and code that is written with the Express framework.
- Explain what Express is and its benefits.
- Use Express as a third-party npm package.
- Explain the use of middleware functions.
- Handle routes and requests.
- Explain async patterns in ECMAScript (ES), including callbacks, promises, and async/await.
- Explain the React component lifecycle.

Agenda

Note

The following unit and exercise durations are estimates, and might not reflect every class experience.

Students in this course use an IBM Cloud Lite account to perform the exercises. This account will never expire, therefore students can continue working on IBM Cloud after the class.

Day 1

(00:30) Welcome
(01:00) Unit 1 - Introduction to server-side JavaScript
(01:30) Exercise 1 - Developing a Hello World Node.js app on Cloud
(01:00) Lunch break
(01:30) Unit 2 - Asynchronous I/O with callback programming
(01:30) Exercise 2 - Understanding asynchronous callback
(01:30) Unit 3: Express web application framework

Day 2

(01:30) Exercise 3 - Creating your first express application
(01:00) Unit 4 -. Async patterns with ECMAScript
(01:30) Exercise 4 - Building a rich UI application by using React and ES8
(01:00) Lunch break
(01:30) Practice test
(01:00) Unit 5.Building rich UI applications with React (Optional)

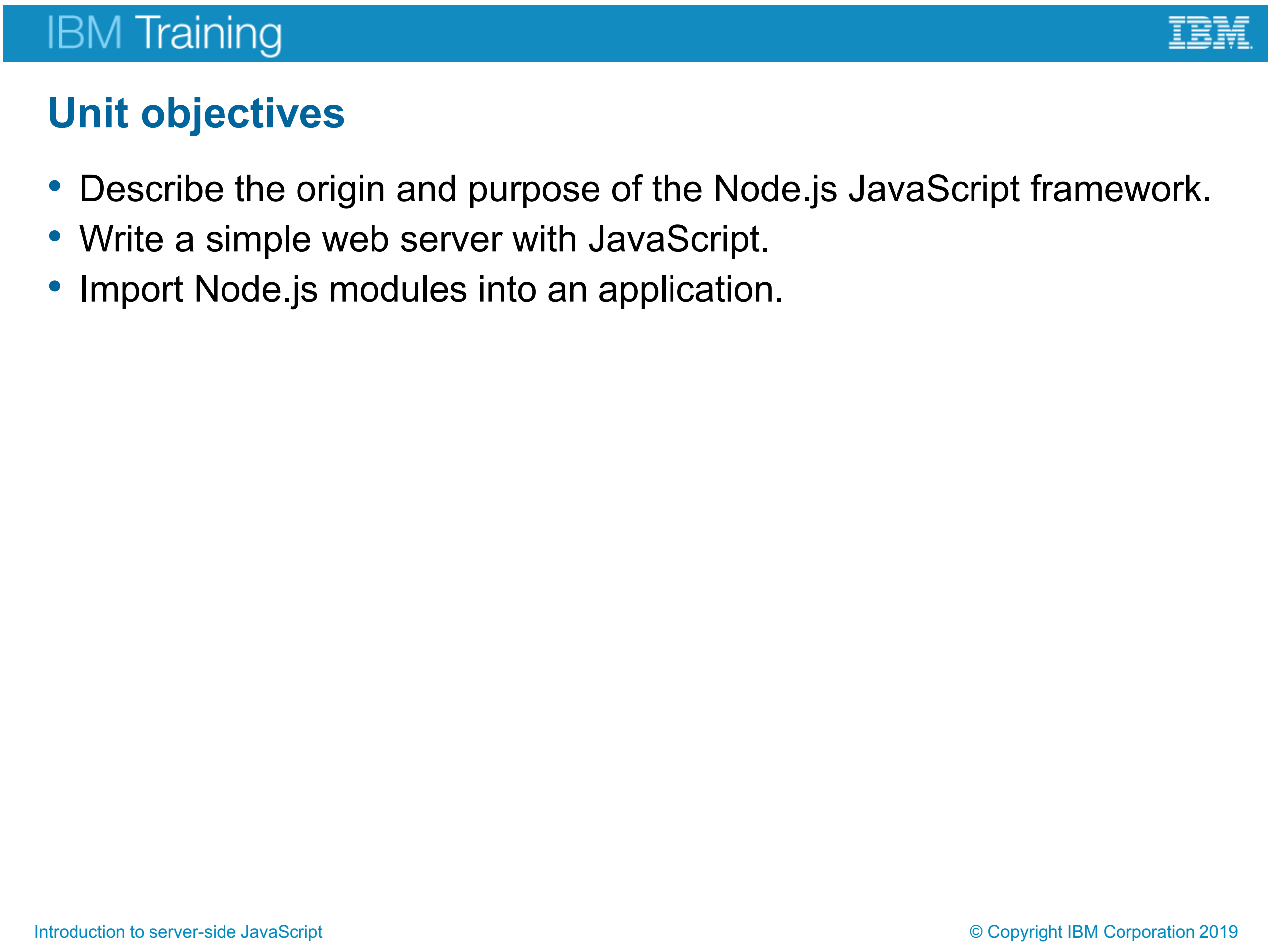
Unit 1. Introduction to server-side JavaScript

Estimated time

01:00

Overview

This unit introduces server-side JavaScript and Node.js. It describes how to create a Node.js server and implement Node.js modules.



IBM Training

Unit objectives

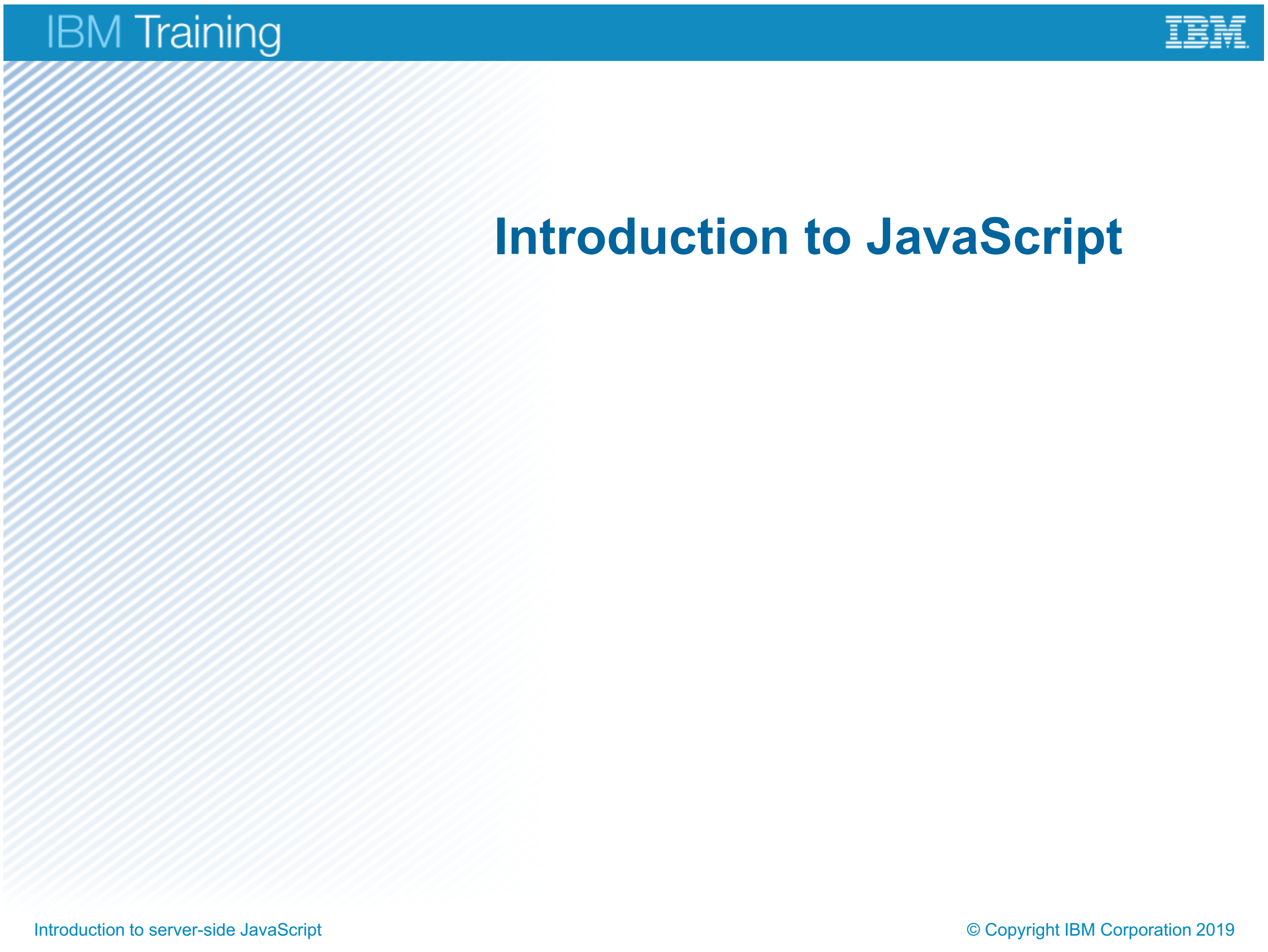
- Describe the origin and purpose of the Node.js JavaScript framework.
- Write a simple web server with JavaScript.
- Import Node.js modules into an application.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-1. Unit objectives

1.1. Introduction to JavaScript



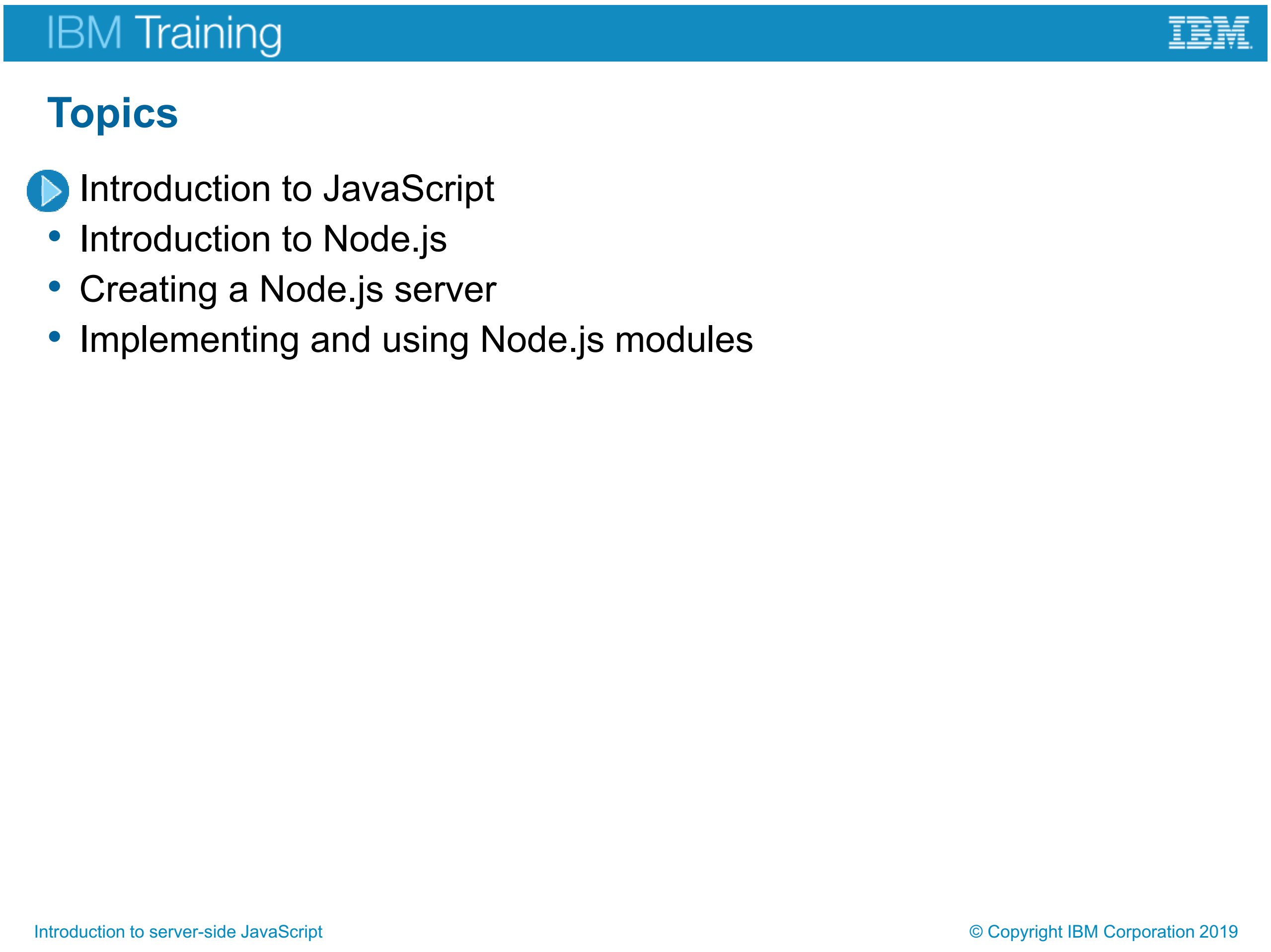
IBM Training

Introduction to JavaScript

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-2. Introduction to JavaScript


IBM Training

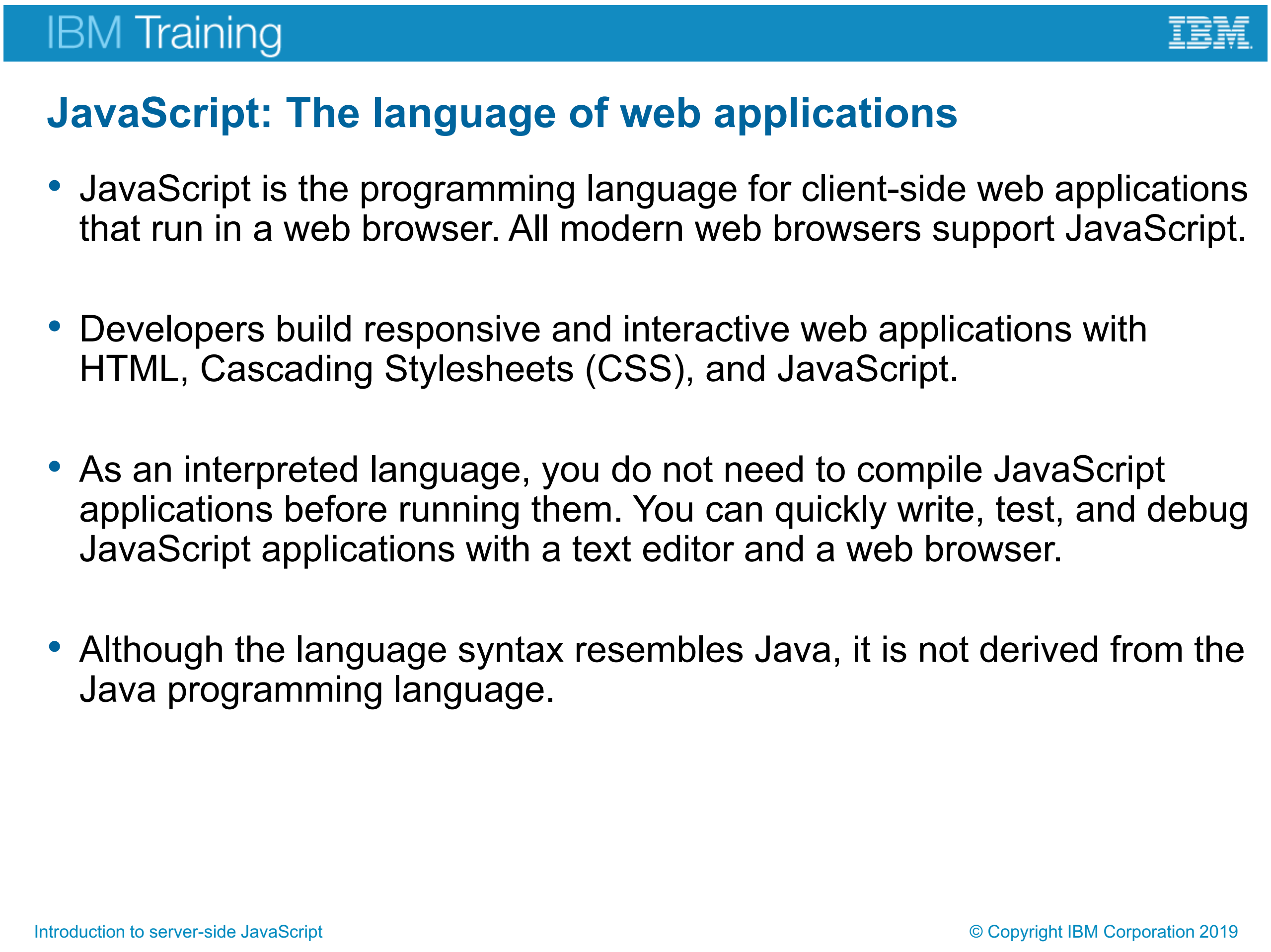
Topics

- Introduction to JavaScript
- Introduction to Node.js
- Creating a Node.js server
- Implementing and using Node.js modules

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-3. Topics



IBM Training

JavaScript: The language of web applications

- JavaScript is the programming language for client-side web applications that run in a web browser. All modern web browsers support JavaScript.
- Developers build responsive and interactive web applications with HTML, Cascading Stylesheets (CSS), and JavaScript.
- As an interpreted language, you do not need to compile JavaScript applications before running them. You can quickly write, test, and debug JavaScript applications with a text editor and a web browser.
- Although the language syntax resembles Java, it is not derived from the Java programming language.

Introduction to server-side JavaScript © Copyright IBM Corporation 2019

Figure 1-4. JavaScript: The language of web applications

JavaScript is a very popular programming language that is used for developing web applications running on web browsers. All modern web browsers support JavaScript to add dynamic parts to such applications. Web applications use:

- HTML to add elements to the application, such as labels, texts, links, and buttons.
- CSS to add styles to the elements, such as making the content of a certain label appear in red
- JavaScript to add a behavior to the web page based on certain user interactions. For example, by using JavaScript, a developer can create a drop-down menu for cities that is auto-populated based on the country that the user selects.

JavaScript is an interpreted language, which means that when you write the code, you do not a need a compiler to convert the code to certain binary files. The web browser can directly run the JavaScript code.

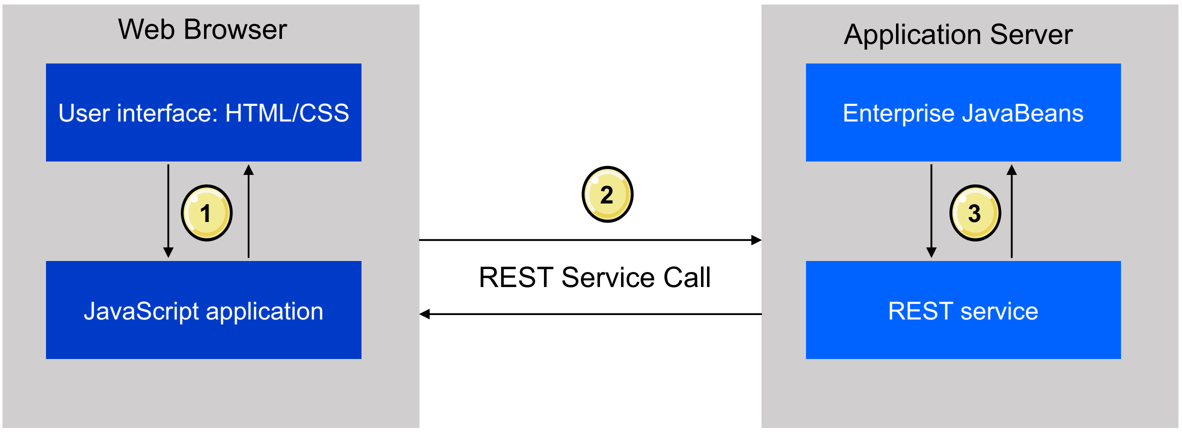
JavaScript is an easy language in which to write and test. You do not need a rich platform to develop the code. You can simply create a JavaScript file and test it by using your web browser.

Note: JavaScript is a different language than Java.

IBM Training

JavaScript applications in the web browser

With client-side JavaScript, developers create rich and interactive web applications that run in a web browser.



Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-5. JavaScript applications in the web browser

Developers can create rich and interactive web applications that run in the web browser:

1. The user interface is rendered by using HTML and CSS. When the user selects an option in the web page, it triggers a call to the JavaScript application.
2. The JavaScript application sends a web service request.
3. On the server, a REST web service intercepts the call. In the last step, the application server processes the web service request by using a server-side application. The server-side application is usually implemented by using a sever-side language like Java, so implementing the server-side application requires a different skill set than implementing the front-end application.

The next slide, shows how JavaScript can be used for server-side implementation.

IBM Training

JavaScript applications in the application server

With server-side JavaScript, applications process and route web service requests from the client.

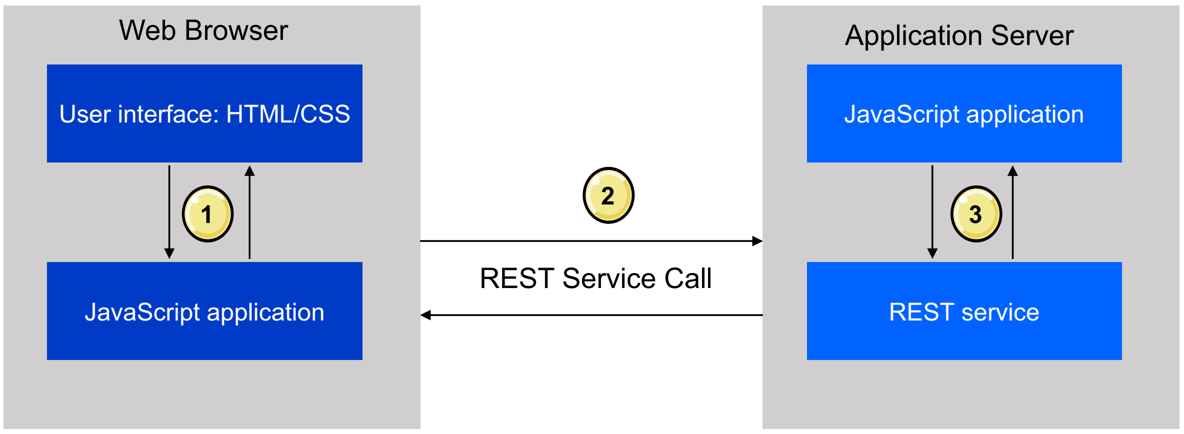


Figure 1-6. JavaScript applications in the application server

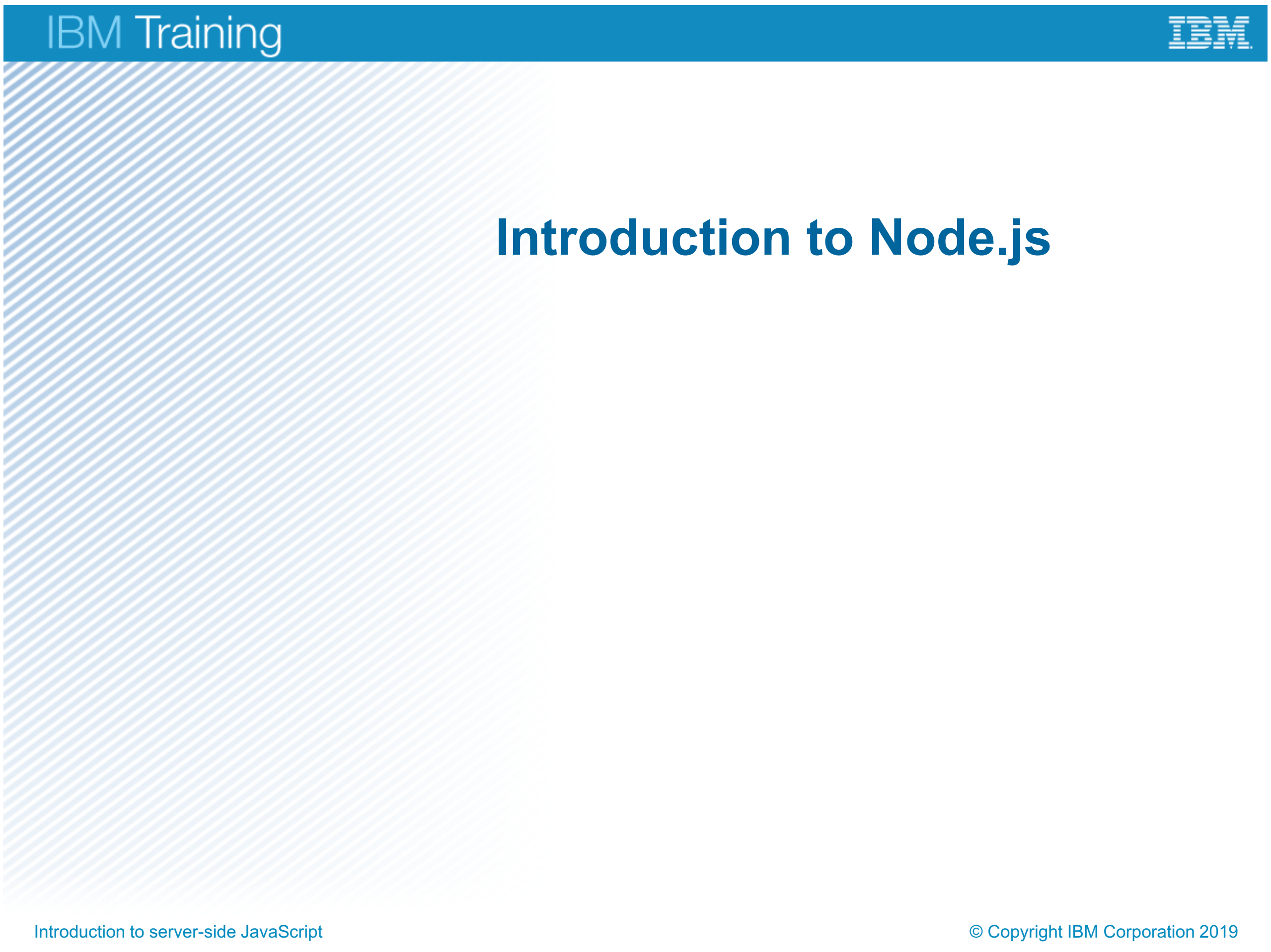
With server-side JavaScript, applications process and route web service requests from the client.

Compare this diagram with the one in the previous slide. Most of the steps are identical.

1. The user interface is rendered by using HTML and CSS. When the user selects an option in the web page, it triggers a call to the JavaScript application.
2. The JavaScript application sends a web service request.
3. On the server, a REST web service intercepts the call. In the last step, the application server processes the web service request by using a server-side application. The server-side application here is implemented by using a JavaScript platform. The well-known platform form for implementing server-side JavaScript applications is Node.js.

One of the main advantages of using Node.js is that it is based on JavaScript, which is the same language that is used for front-end development.

1.2. Introduction to Node.js



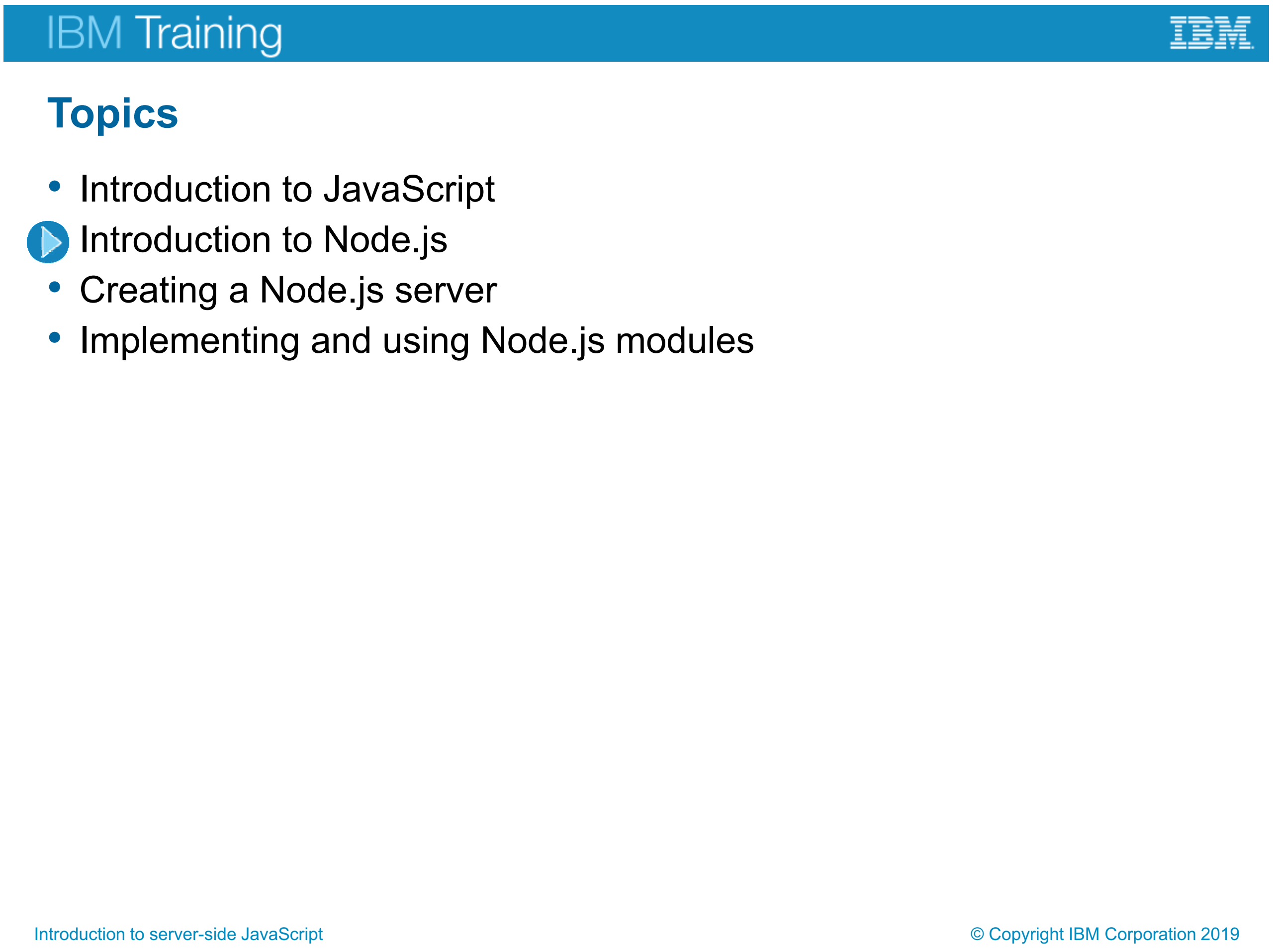
IBM Training

Introduction to Node.js

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-7. Introduction to Node.js



IBM Training

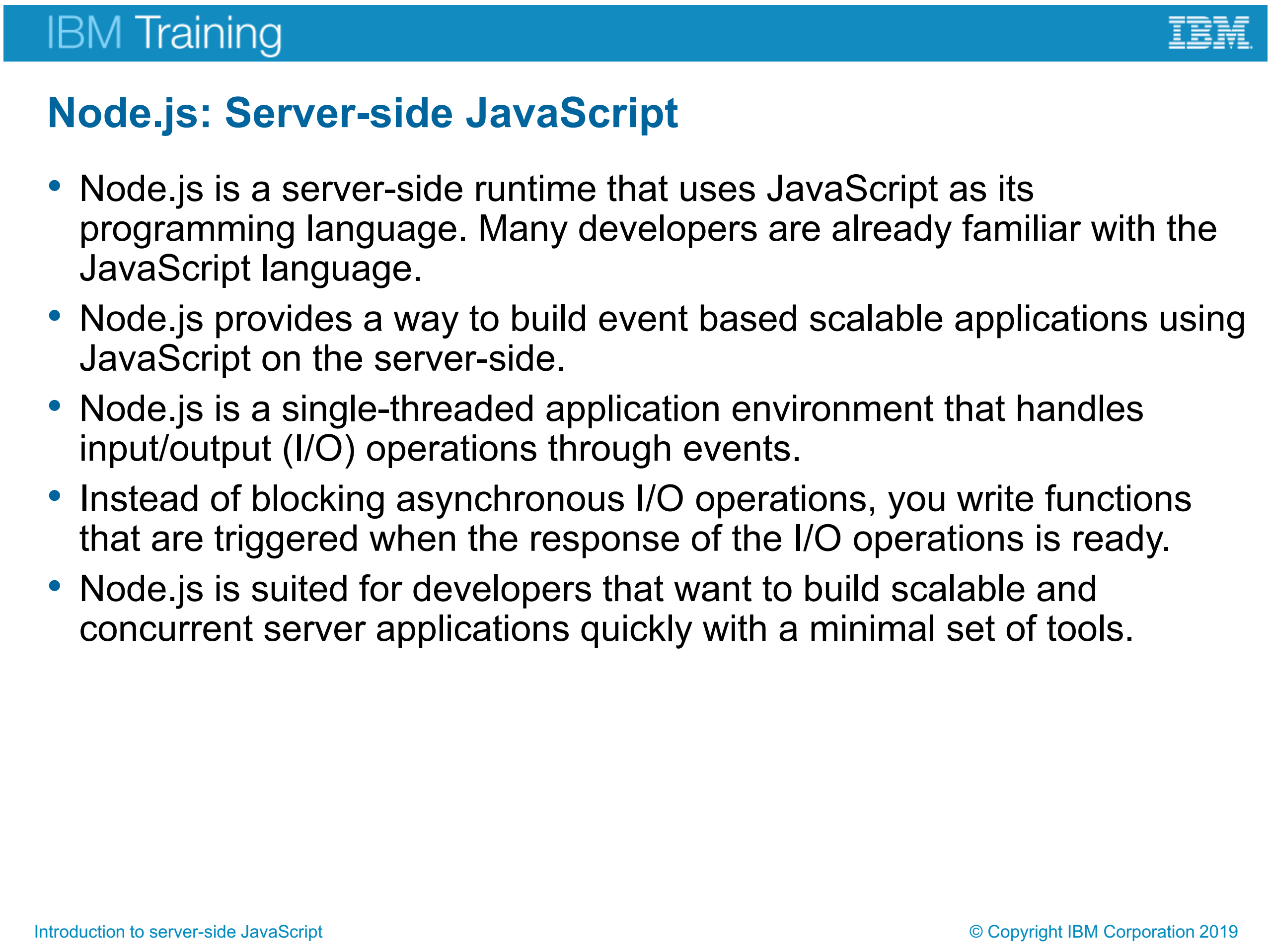
Topics

- Introduction to JavaScript
- ▶ Introduction to Node.js
- Creating a Node.js server
- Implementing and using Node.js modules

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-8. Topics



IBM Training

Node.js: Server-side JavaScript

- Node.js is a server-side runtime that uses JavaScript as its programming language. Many developers are already familiar with the JavaScript language.
- Node.js provides a way to build event based scalable applications using JavaScript on the server-side.
- Node.js is a single-threaded application environment that handles input/output (I/O) operations through events.
- Instead of blocking asynchronous I/O operations, you write functions that are triggered when the response of the I/O operations is ready.
- Node.js is suited for developers that want to build scalable and concurrent server applications quickly with a minimal set of tools.

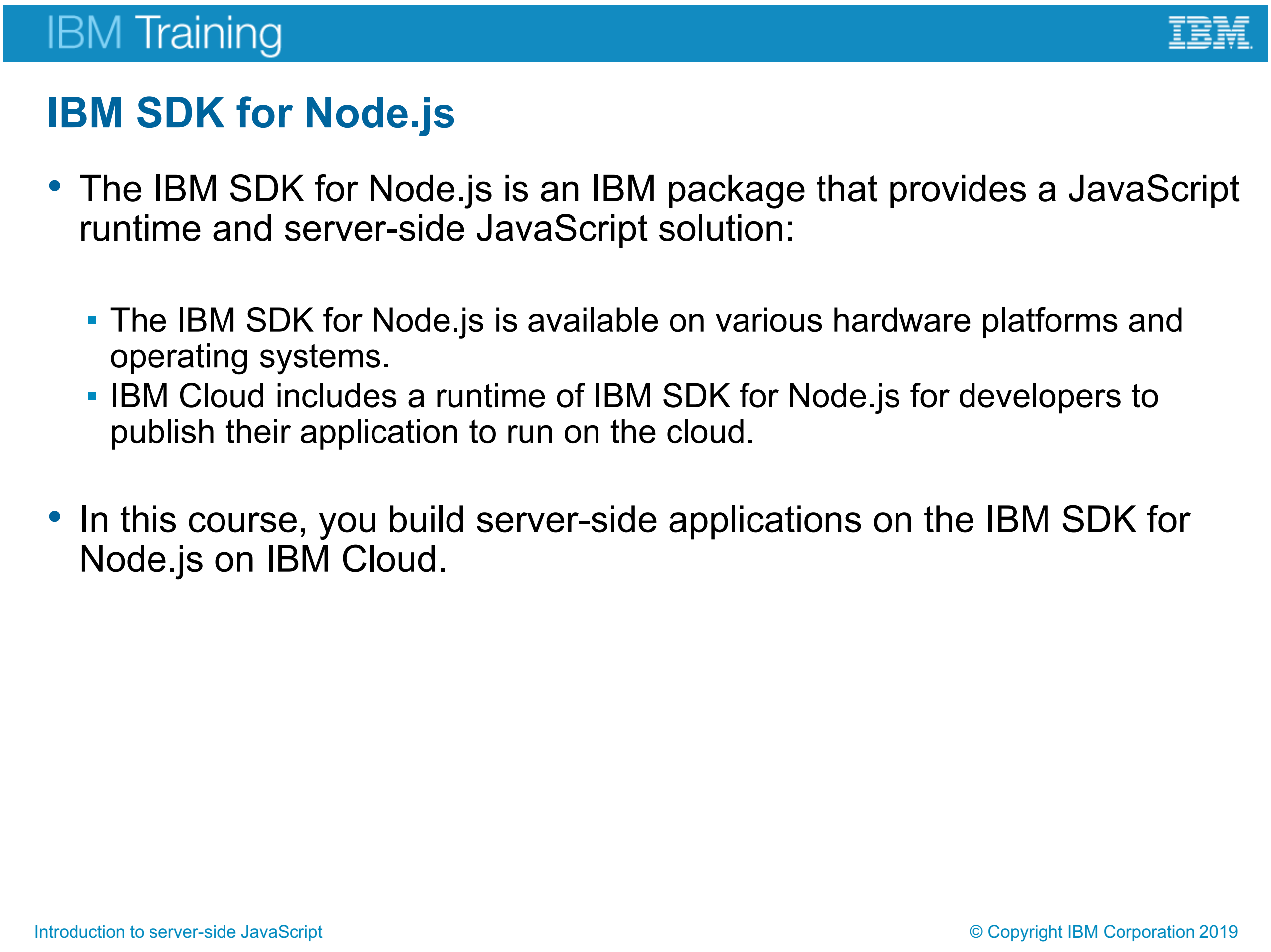
Introduction to server-side JavaScript © Copyright IBM Corporation 2019

Figure 1-9. Node.js: Server-side JavaScript

Node.js is a server-side platform that uses JavaScript as its programming language.

Node.js is a single-threaded application environment that handles input/output (I/O) operations through events.

Because Node.js is single-threaded, it is not suitable for CPU-intensive workloads.



IBM Training

IBM SDK for Node.js

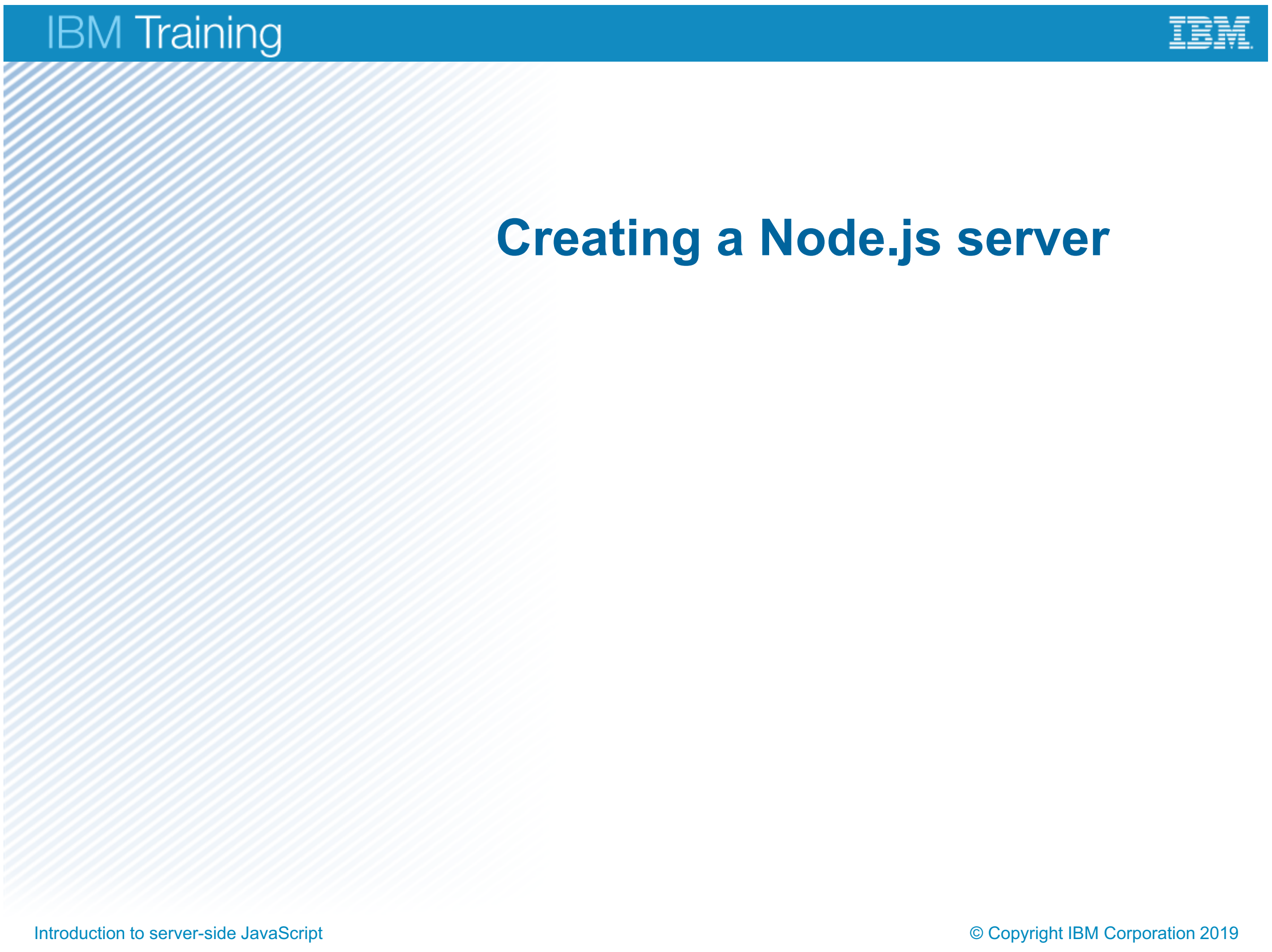
- The IBM SDK for Node.js is an IBM package that provides a JavaScript runtime and server-side JavaScript solution:
  - The IBM SDK for Node.js is available on various hardware platforms and operating systems.
  - IBM Cloud includes a runtime of IBM SDK for Node.js for developers to publish their application to run on the cloud.
- In this course, you build server-side applications on the IBM SDK for Node.js on IBM Cloud.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-10. IBM SDK for Node.js

1.3. Creating a Node.js server



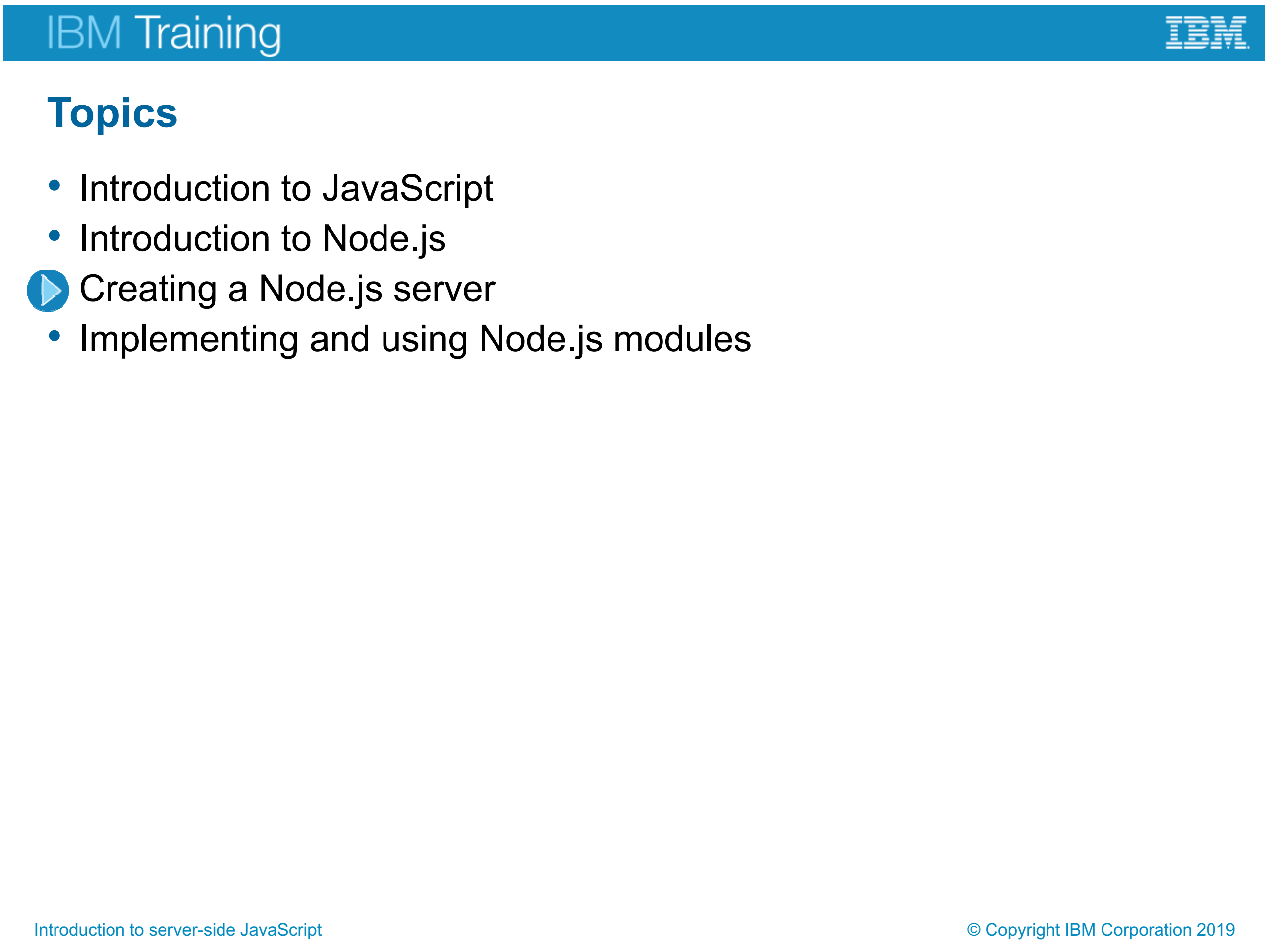
IBM Training

Creating a Node.js server

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-11. Creating a Node.js server



IBM Training

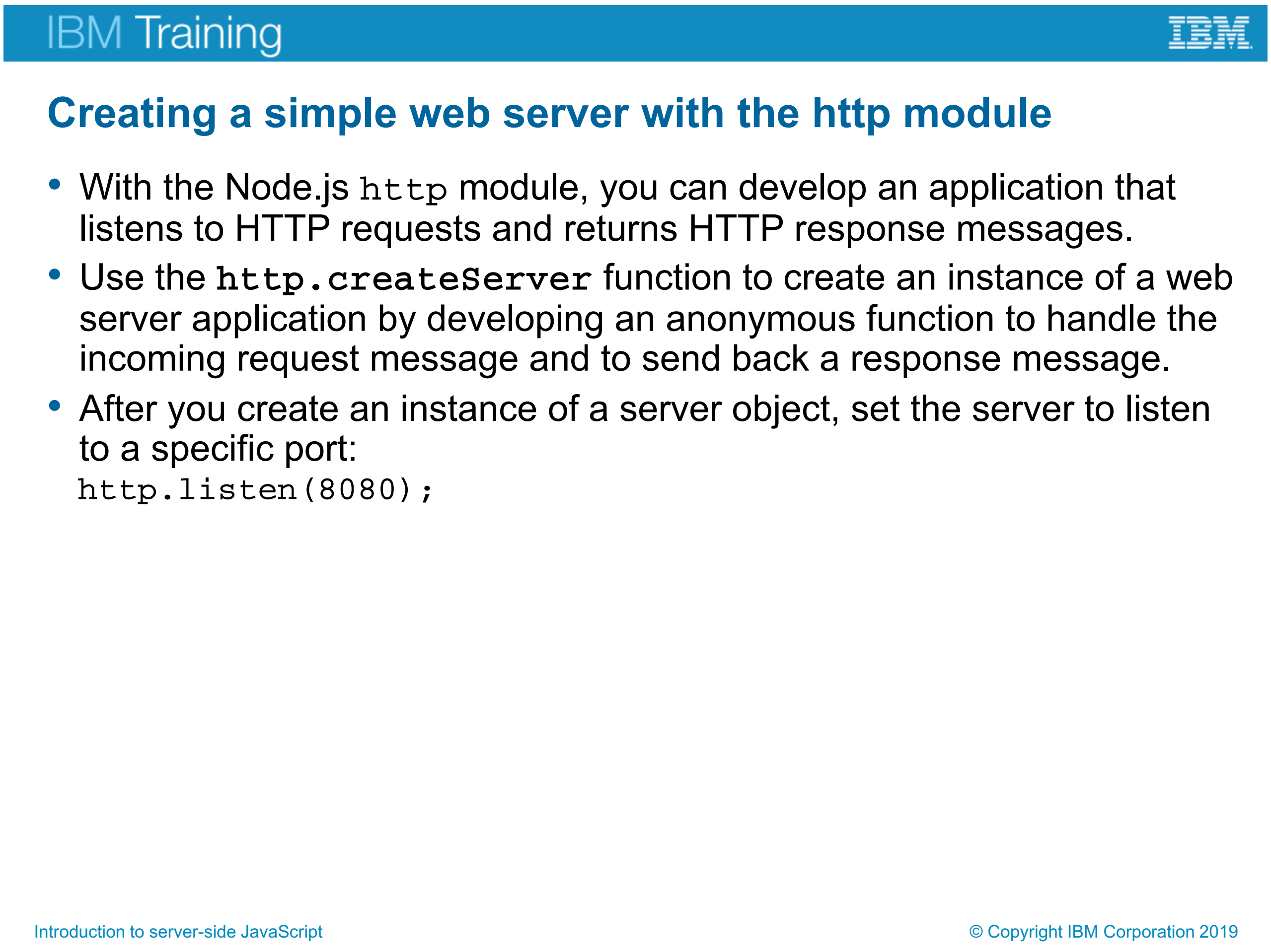
Topics

- Introduction to JavaScript
- Introduction to Node.js
- ▶ Creating a Node.js server
- Implementing and using Node.js modules

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-12. Topics



IBM Training

Creating a simple web server with the http module

- With the Node.js `http` module, you can develop an application that listens to HTTP requests and returns HTTP response messages.
- Use the **`http.createServer`** function to create an instance of a web server application by developing an anonymous function to handle the incoming request message and to send back a response message.
- After you create an instance of a server object, set the server to listen to a specific port:
  ```
  http.listen(8080);
  ```

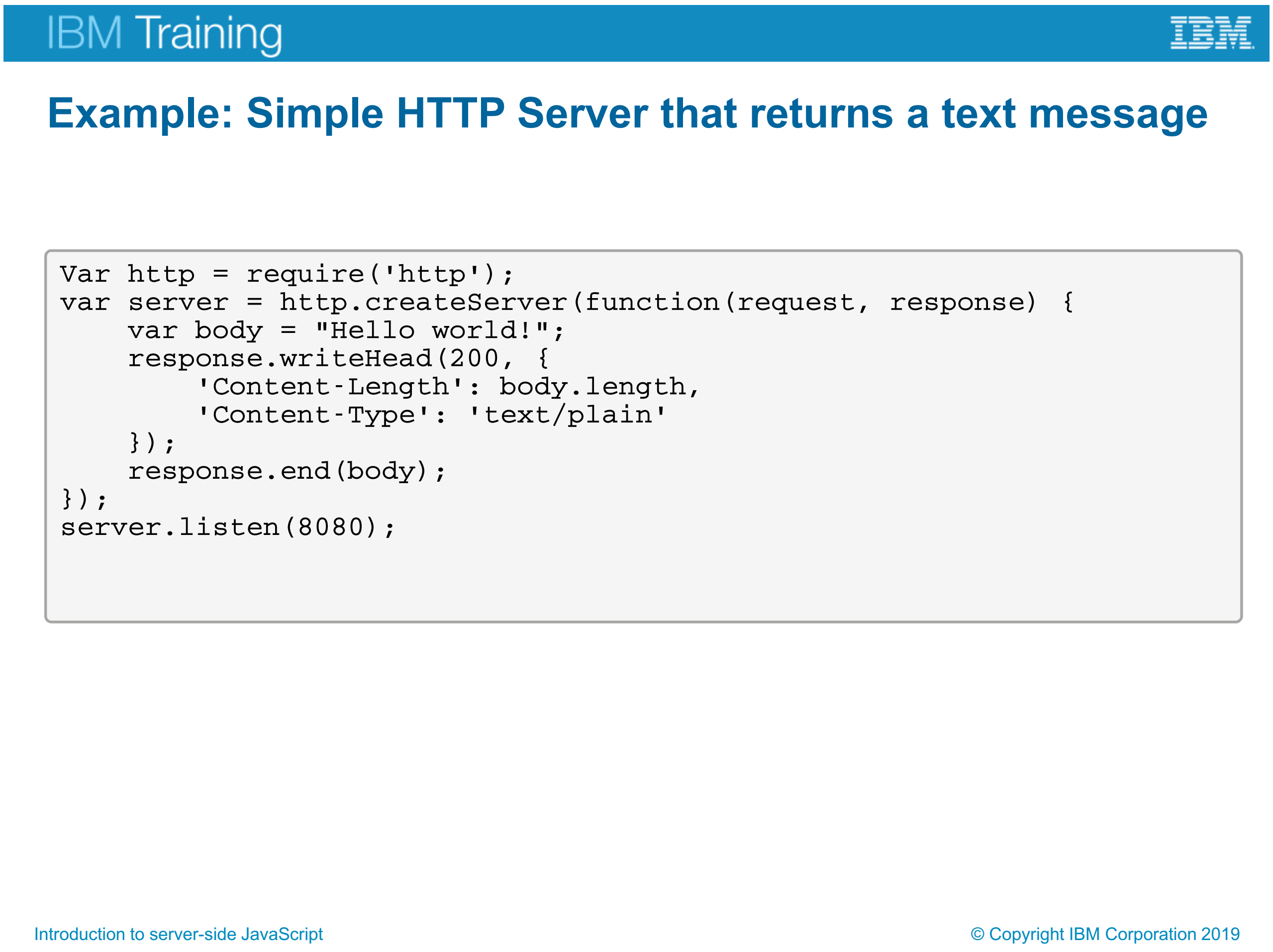
Figure 1-13. Creating a simple web server with the http module

Node.js has several useful modules that help create server-side applications (Node.js modules are covered in the next topic).

The Node.js “`http`” module is used to handle HTTP operations. You can develop an application that listens to HTTP requests and returns HTTP response messages.

The **http.createServer** function is used to create an instance of a web server application by developing an anonymous function to handle the incoming request message and to send back a response message.

Anonymous functions are functions without a name. They are usually used when they are needed only for a special purpose and there is no plan for them to be reused in the application.



IBM Training

Example: Simple HTTP Server that returns a text message

```
Var http = require('http');
var server = http.createServer(function(request, response) {
    var body = "Hello world!";
    response.writeHead(200, {
        'Content-Length': body.length,
        'Content-Type': 'text/plain'
    });
    response.end(body);
});
server.listen(8080);
```

Introduction to server-side JavaScript © Copyright IBM Corporation 2019

Figure 1-14. Example: Simple HTTP Server that returns a text message

To create an HTTP Server by using Node.js, use the “http” Node.js module.

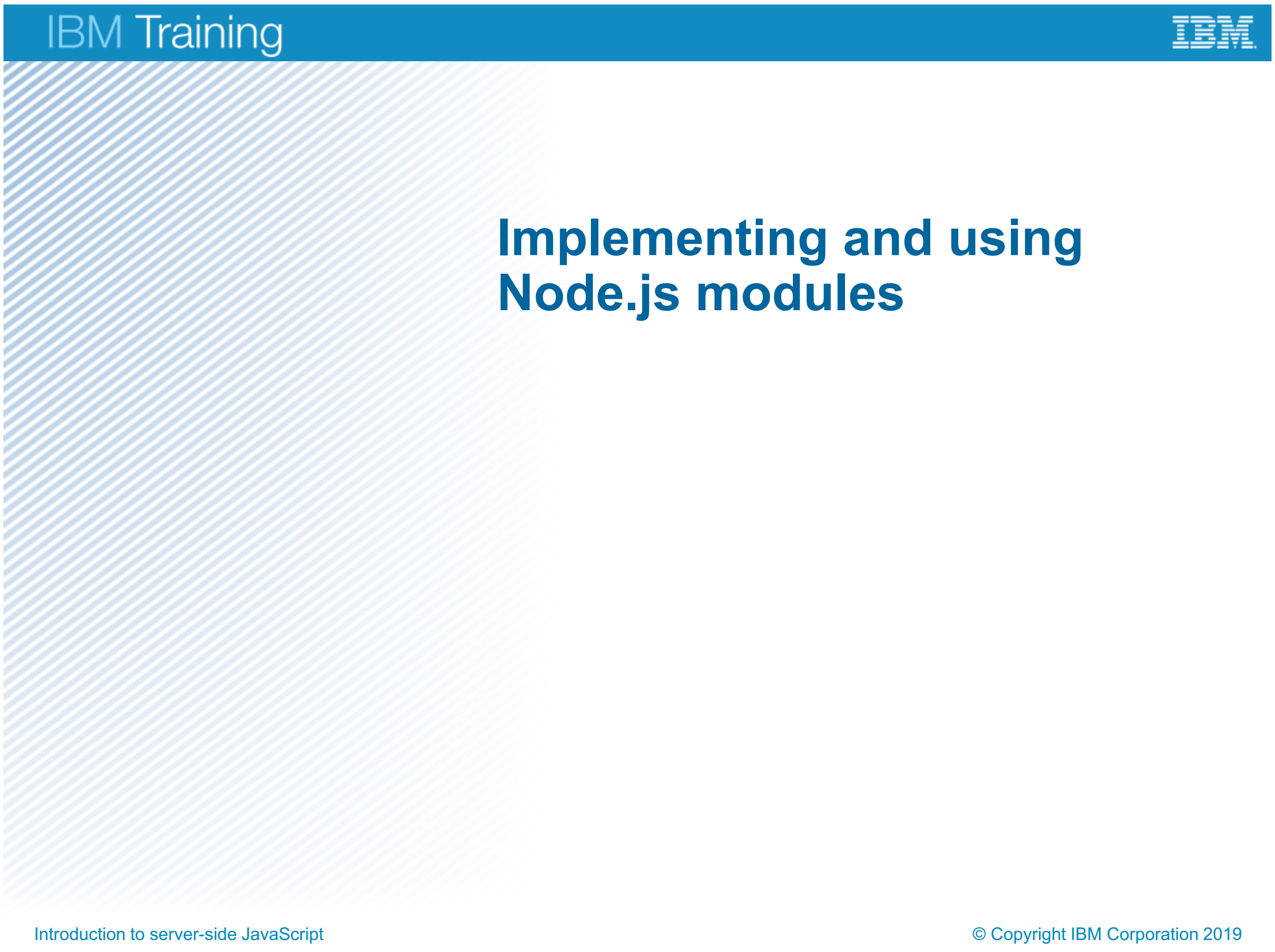
You use the **createServer** function to create an instance of the web server. The **createServer** function itself contains a function as an argument. This function is called when a request is received from a user. The “request” object contains the request details that come from the user, and the “response” object is used to complete the response that returns to the user.

In this example, the code sets the response code to 200 (a success response). The content in the body is the following text:

Hello world!

The HTTP server in this example listens on port 8080, which means that the requests coming to the server on port 8080 are processed by this Node.js HTTP server.

1.4. Implementing and using Node.js modules



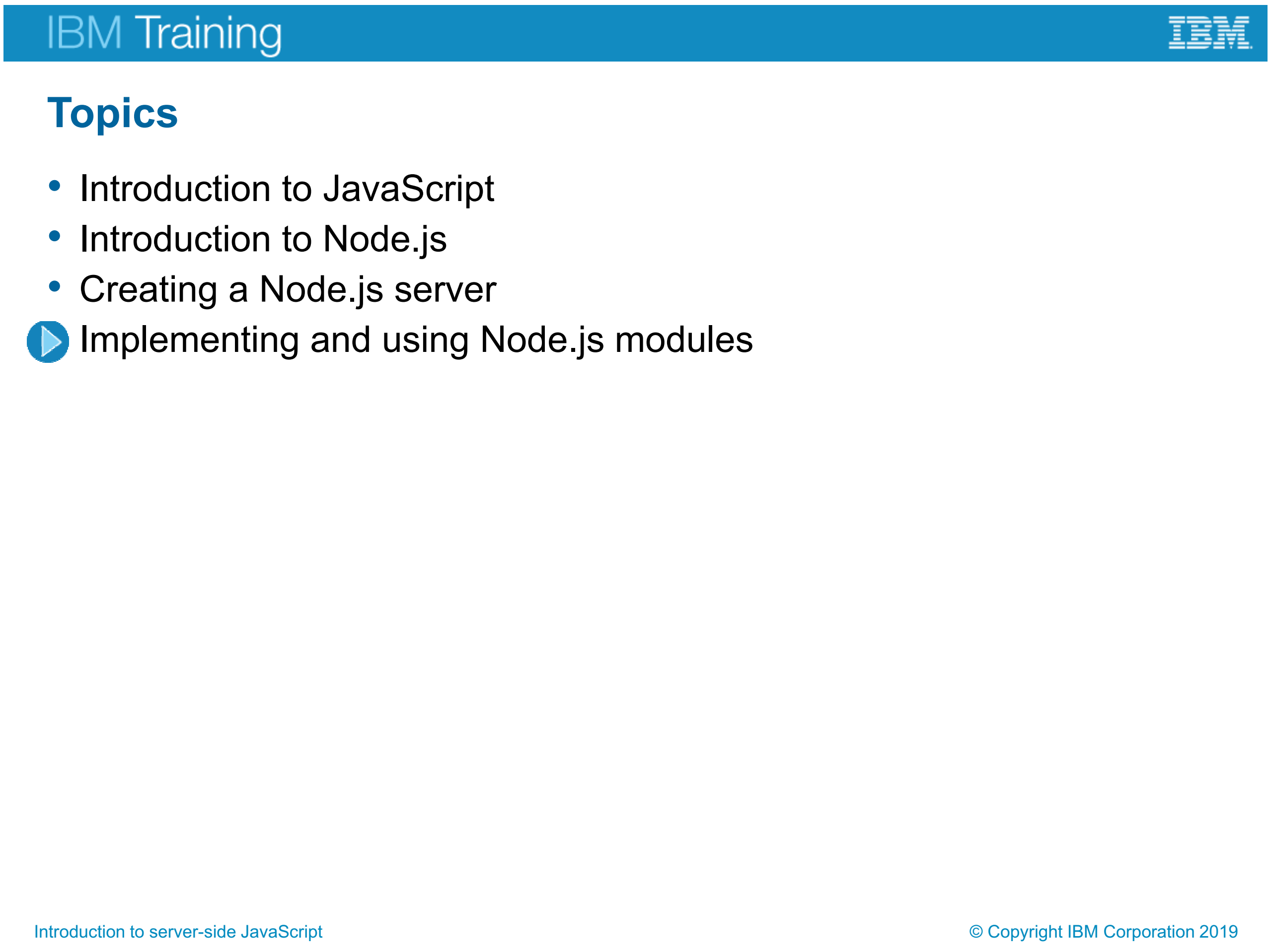
IBM Training

Implementing and using Node.js modules

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-15. Implementing and using Node.js modules



IBM Training

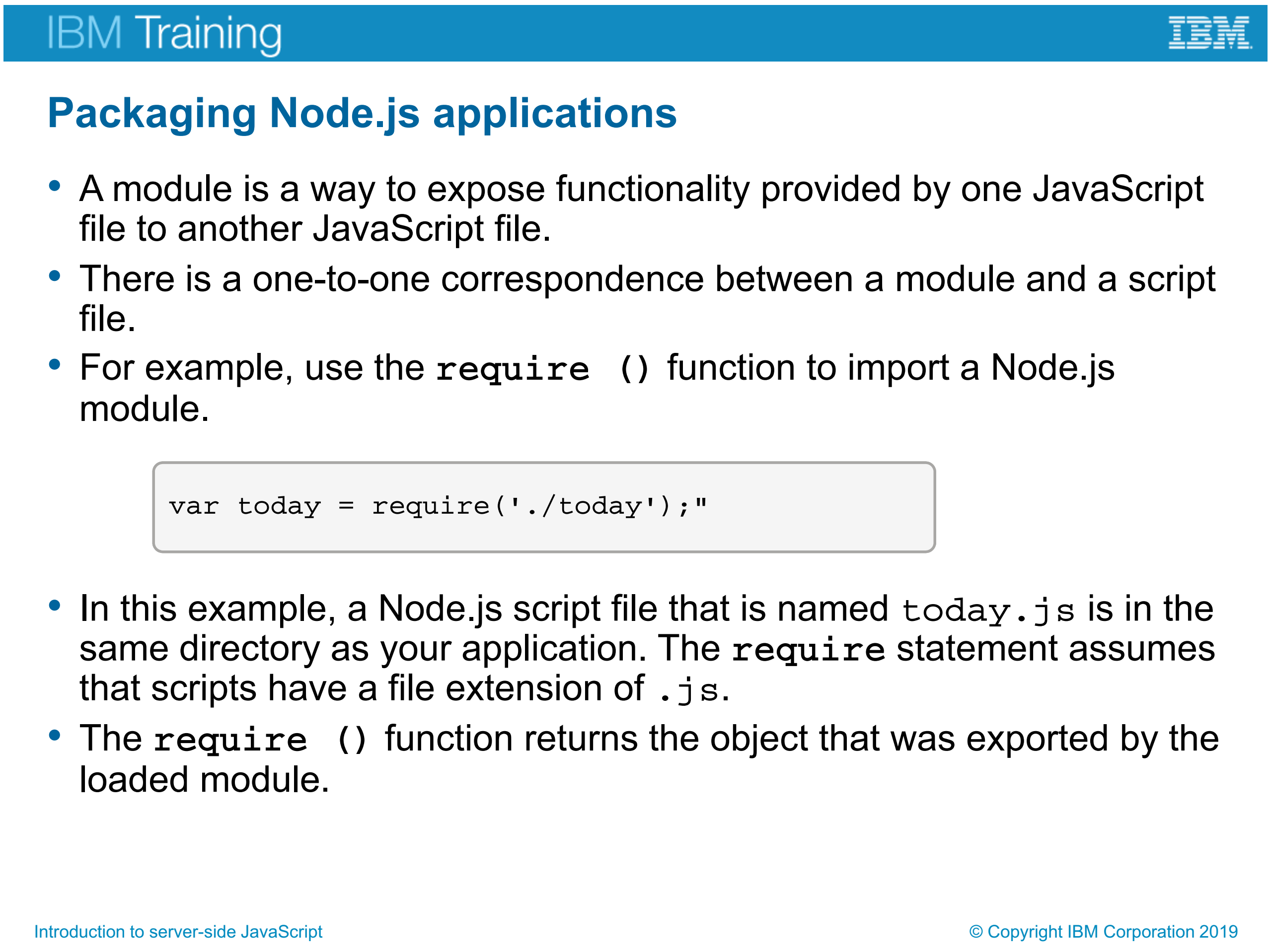
Topics

- Introduction to JavaScript
- Introduction to Node.js
- Creating a Node.js server
- Implementing and using Node.js modules

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-16. Topics



IBM Training

Packaging Node.js applications

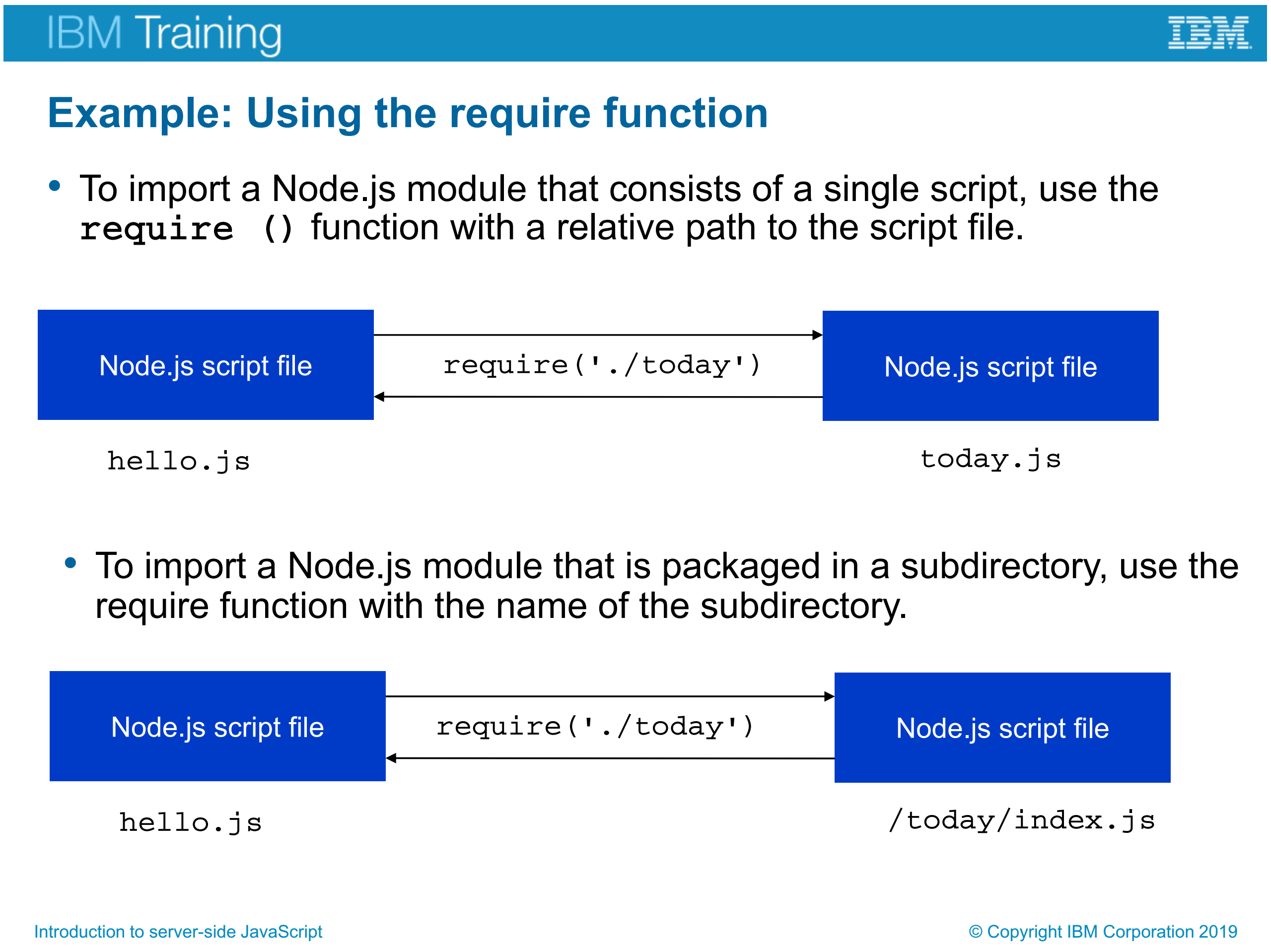
- A module is a way to expose functionality provided by one JavaScript file to another JavaScript file.
- There is a one-to-one correspondence between a module and a script file.
- For example, use the **`require ()`** function to import a Node.js module.

```
var today = require('./today');"
```

- In this example, a Node.js script file that is named `today.js` is in the same directory as your application. The **`require`** statement assumes that scripts have a file extension of `.js`.
- The **`require ()`** function returns the object that was exported by the loaded module.

Introduction to server-side JavaScript

Figure 1-17. Packaging Node.js applications



IBM Training

Example: Using the require function

- To import a Node.js module that consists of a single script, use the **`require ()`** function with a relative path to the script file.

Node.js script file → `require('./today')` → Node.js script file

`hello.js` `today.js`

- To import a Node.js module that is packaged in a subdirectory, use the require function with the name of the subdirectory.

Node.js script file → `require('./today')` → Node.js script file

`hello.js` `/today/index.js`

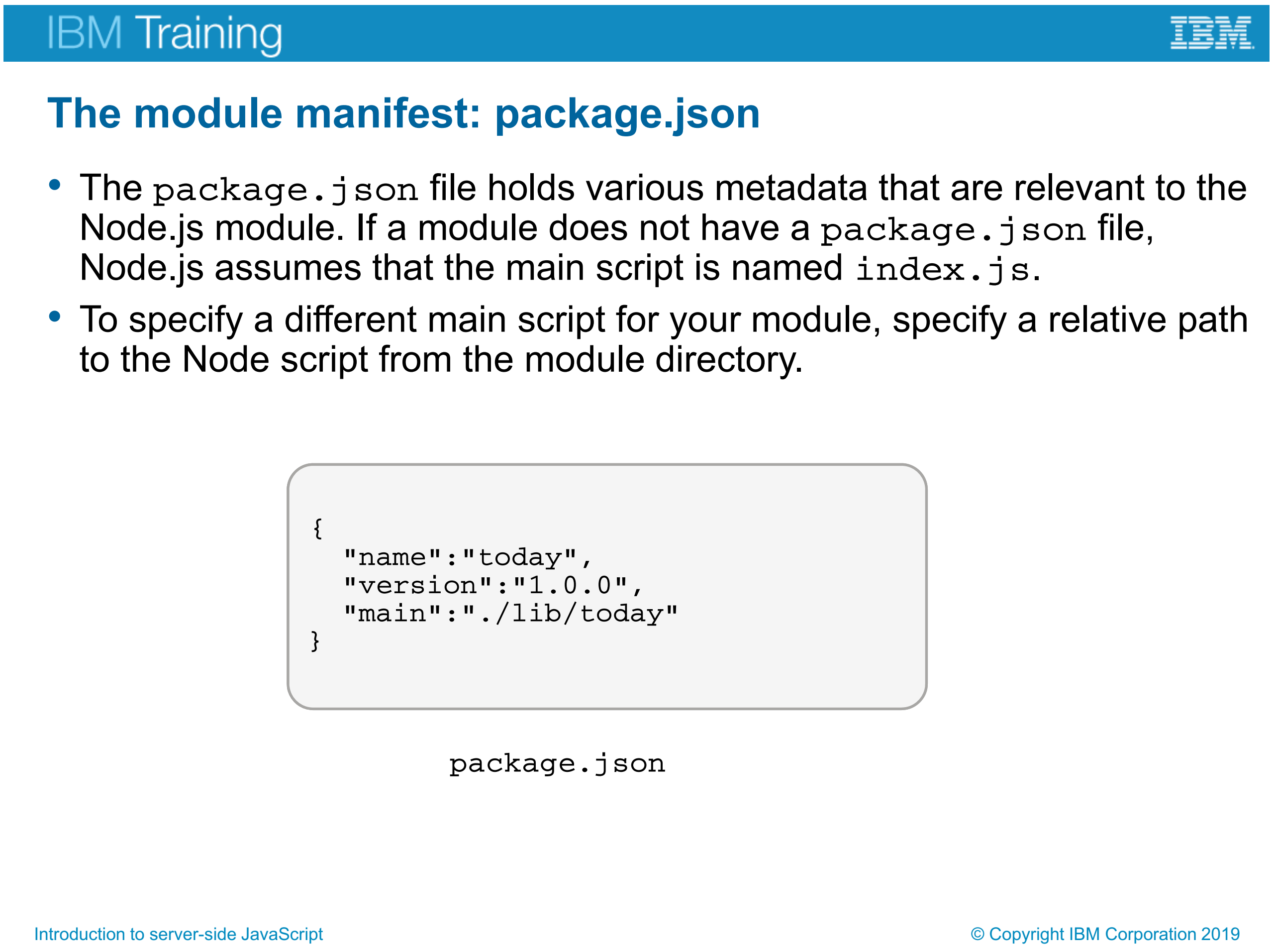
Introduction to server-side JavaScript © Copyright IBM Corporation 2019

Figure 1-18. Example: Using the require function

When creating a module in your Node.js application, you can either create it as a file (for example, “today.js”) or you can create a folder and add an “index.js” file inside the folder (for example, “/today/index.js”).

Note

If both, the “today.js” and the “/today/index.js” files are in the root directory, and **require(‘./today’)** is used in the code, the “today.js” file has priority over the “/today/index.js” file for being loaded.



IBM Training

The module manifest: package.json

- The `package.json` file holds various metadata that are relevant to the Node.js module. If a module does not have a `package.json` file, Node.js assumes that the main script is named `index.js`.
- To specify a different main script for your module, specify a relative path to the Node script from the module directory.

```
{
  "name":"today",
  "version":"1.0.0",
  "main":"./lib/today"
}
```

`package.json`

Introduction to server-side JavaScript

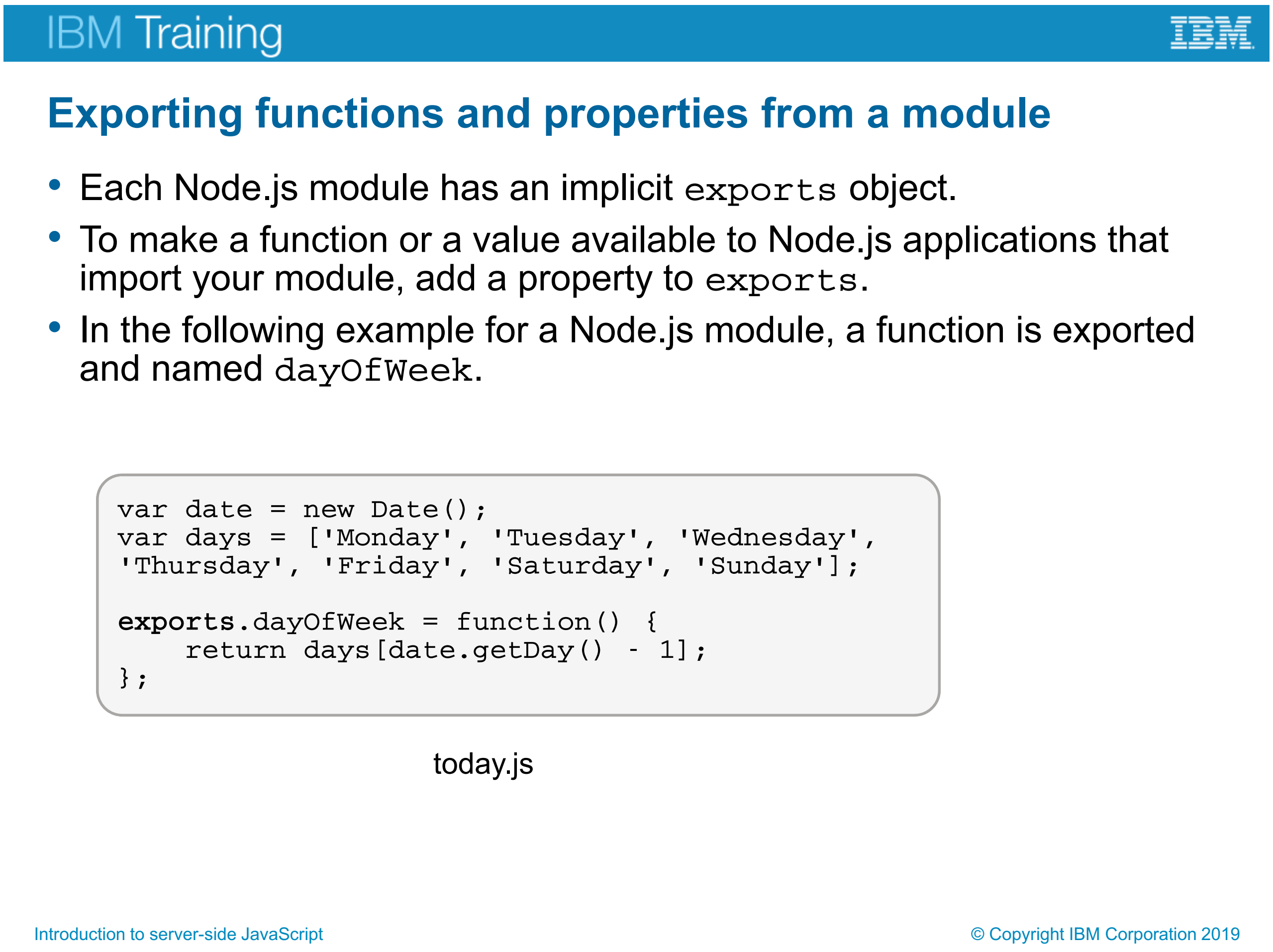
© Copyright IBM Corporation 2019

Figure 1-19. The module manifest: package.json

The slide shows a sample of the “package.json” file.

The *name* and *version* fields form a unique identifier for the module. For example, today-1.0.0.

The *main* field lists a path to the main node script. In this example, the “today.js” script is in the **lib** subdirectory.



IBM Training

Exporting functions and properties from a module

- Each Node.js module has an implicit `exports` object.
- To make a function or a value available to Node.js applications that import your module, add a property to `exports`.
- In the following example for a Node.js module, a function is exported and named `dayOfWeek`.

```
var date = new Date();
var days = ['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday'];

exports.dayOfWeek = function() {
    return days[date.getDay() - 1];
};
```

today.js

Introduction to server-side JavaScript

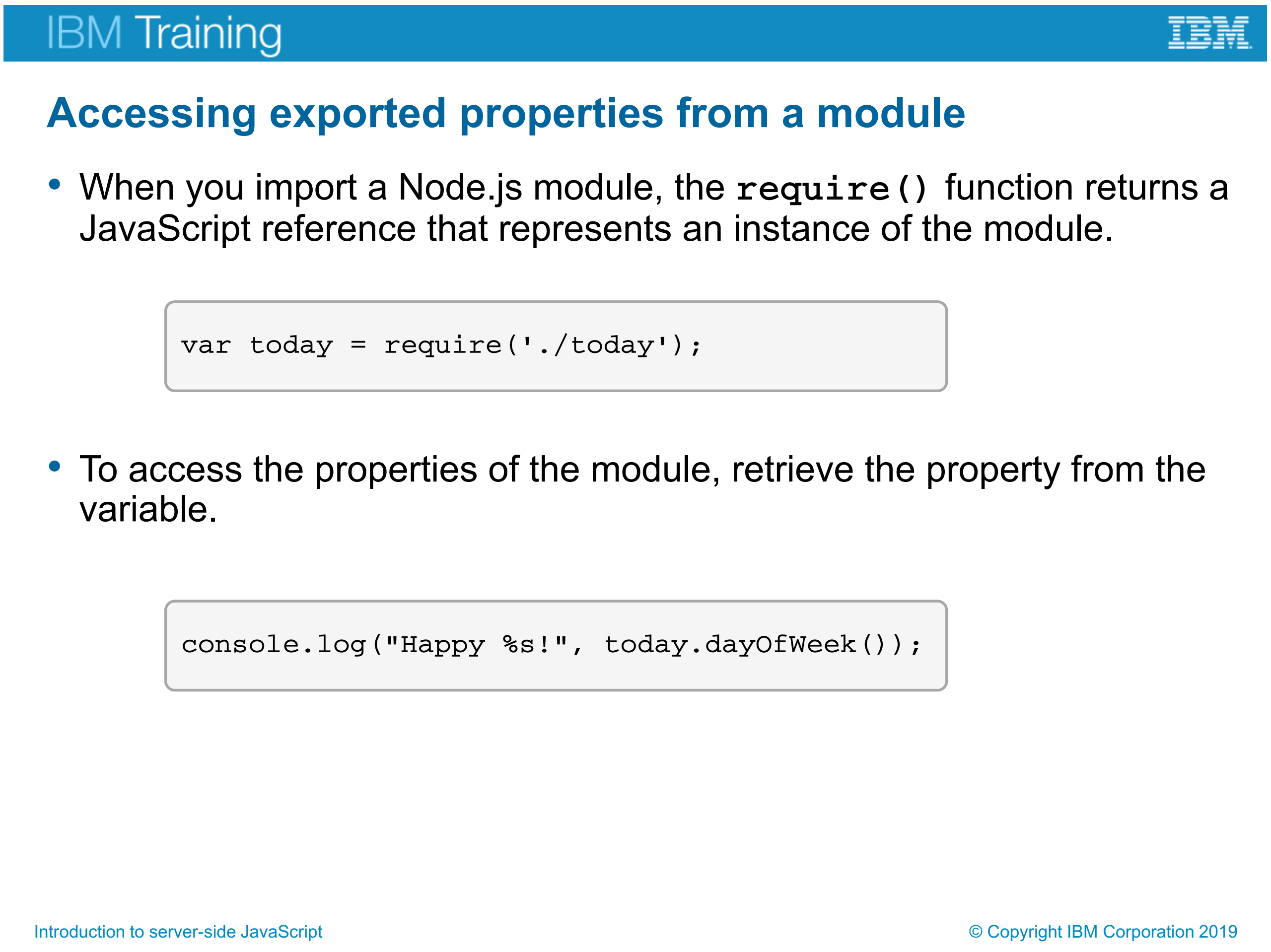
© Copyright IBM Corporation 2019

Figure 1-20. Exporting functions and properties from a module

The *exports* object is a special object that is used to expose properties or functions to be used by the caller of this Node.js module.

In the example in this slide, the “today.js” module contains a function (dayOfWeek) that checks the current date and then uses it to detect the current day of the week.

The dayOfWeek function is added to the **exports** object, which exposes the dayOfWeek function to be used by other Node.js code outside the today.js Node.js module, as shown in the next slide.



IBM Training

Accessing exported properties from a module

- When you import a Node.js module, the **`require()`** function returns a JavaScript reference that represents an instance of the module.

```
var today = require('./today');
```

- To access the properties of the module, retrieve the property from the variable.

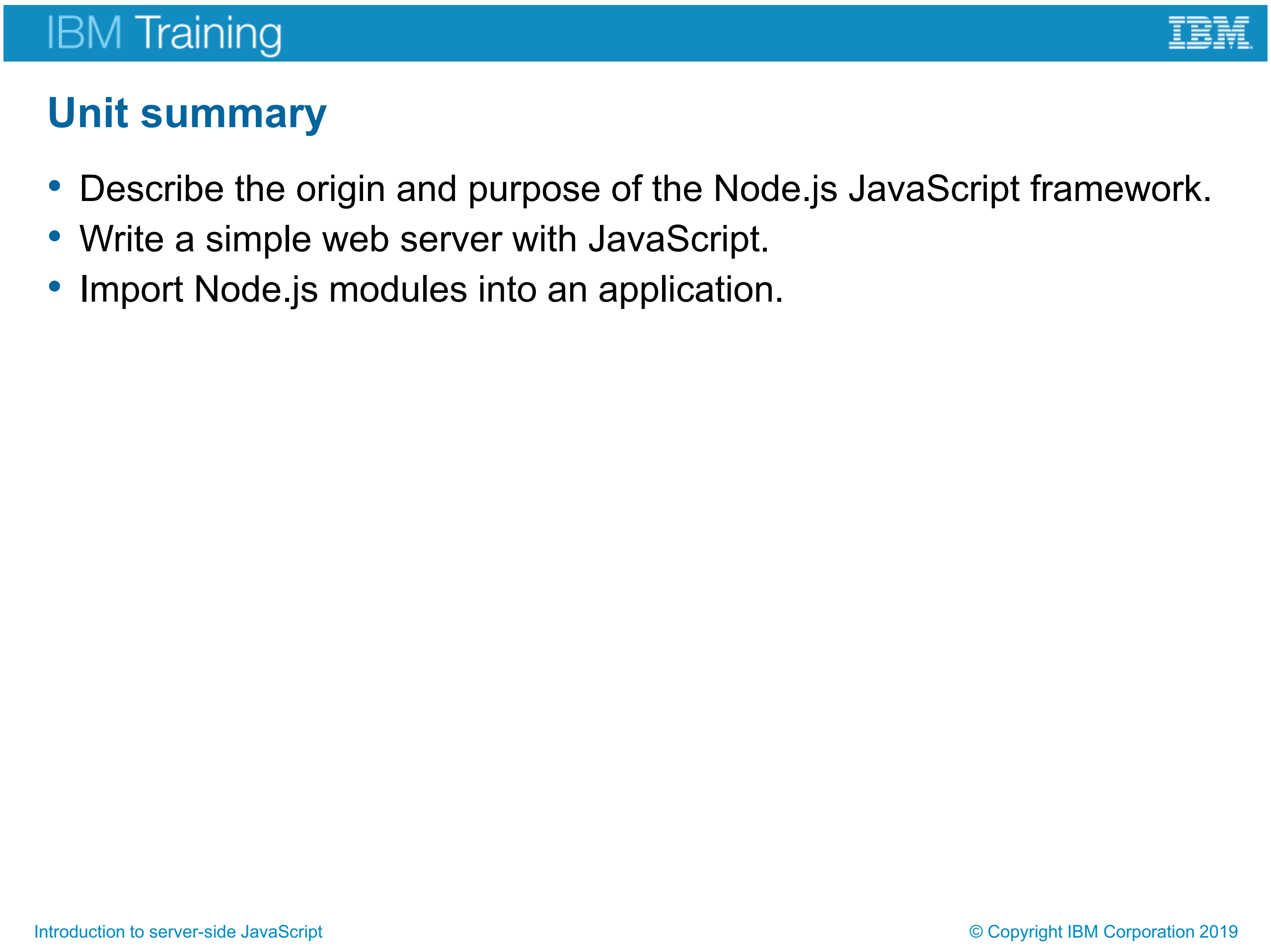
```
console.log("Happy %s!", today.dayOfWeek());
```

Introduction to server-side JavaScript

Figure 1-21. Accessing exported properties from a module

When you import a Node.js module, the **require()** function returns a JavaScript object that represents an instance of the module. In the example in the slide, the “today” variable refers to the “today” Node.js module.

To access the properties of the module, retrieve the property from the variable. In the same example, “today.dayOfWeek” represents the current exported property from the “today” Node.js module.



IBM Training

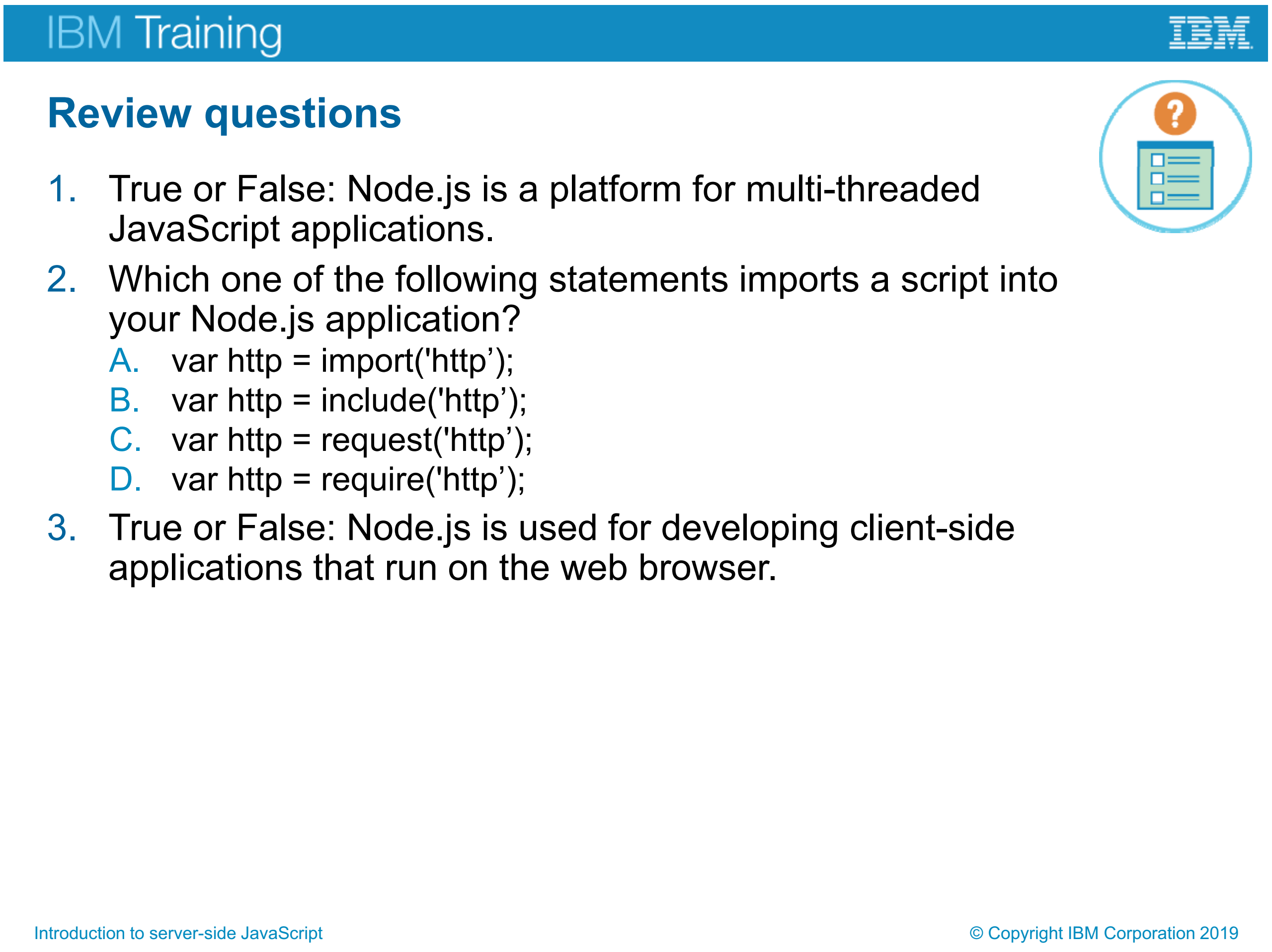
Unit summary

- Describe the origin and purpose of the Node.js JavaScript framework.
- Write a simple web server with JavaScript.
- Import Node.js modules into an application.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-22. Unit summary



IBM Training

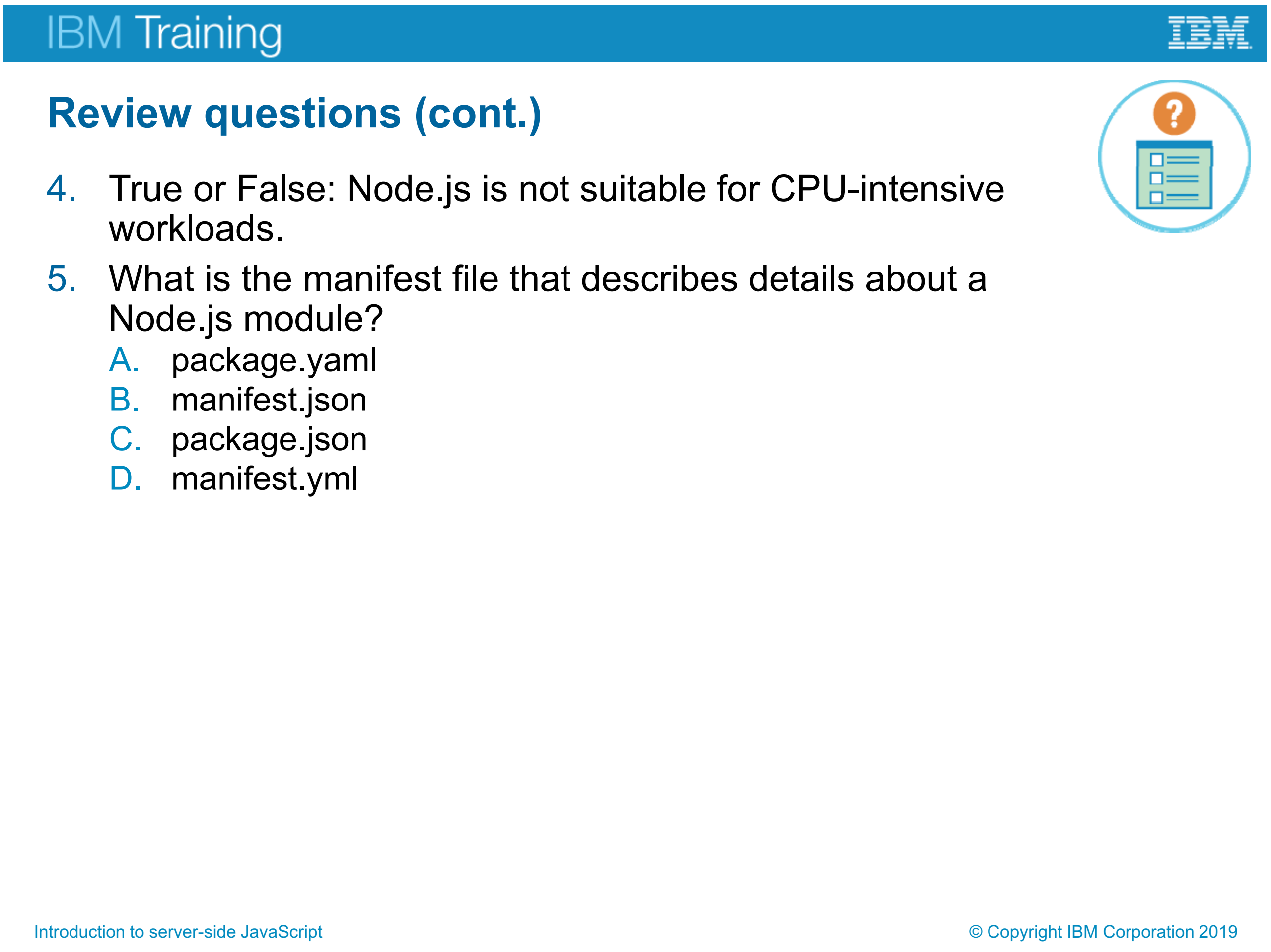
Review questions

1. True or False: Node.js is a platform for multi-threaded JavaScript applications.
2. Which one of the following statements imports a script into your Node.js application?
   A. var http = import('http’);
   B. var http = include('http’);
   C. var http = request('http’);
   D. var http = require('http’);
3. True or False: Node.js is used for developing client-side applications that run on the web browser.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-23. Review questions



IBM Training

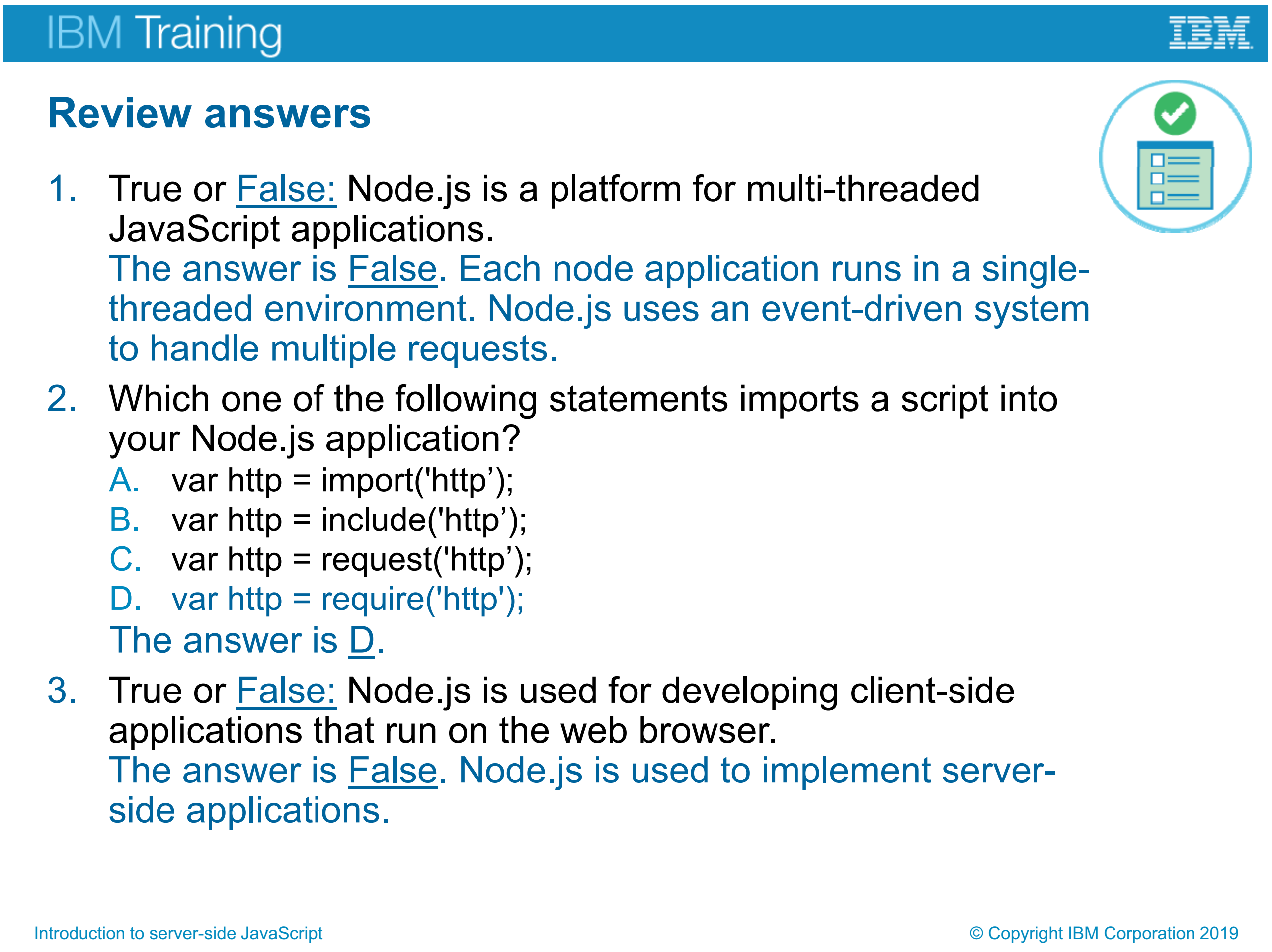
Review questions (cont.)

4. True or False: Node.js is not suitable for CPU-intensive workloads.
5. What is the manifest file that describes details about a Node.js module?
   A. package.yaml
   B. manifest.json
   C. package.json
   D. manifest.yml

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-24. Review questions (cont.)



IBM Training

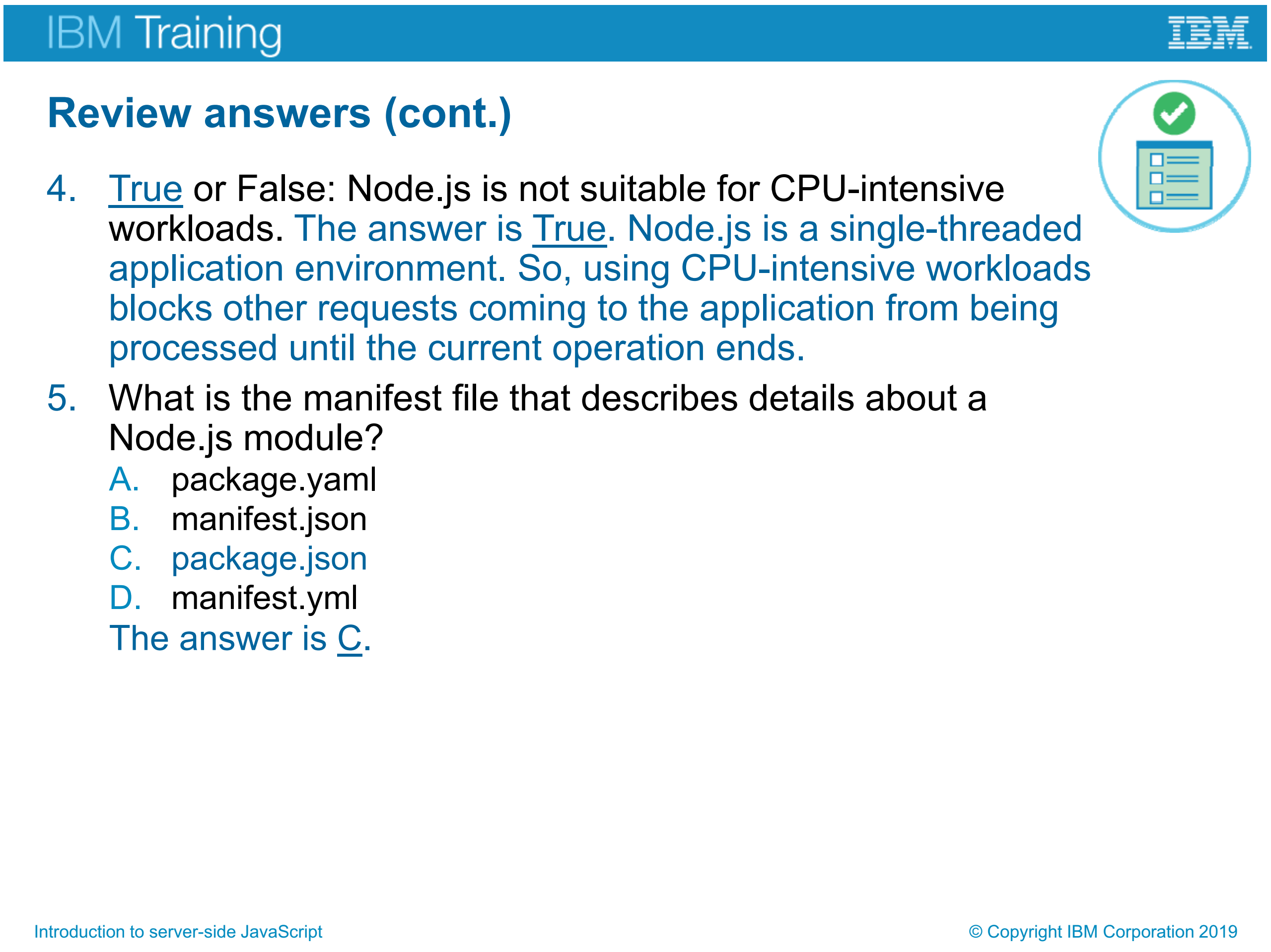
Review answers

1. True or False: Node.js is a platform for multi-threaded JavaScript applications.
   The answer is False. Each node application runs in a single-threaded environment. Node.js uses an event-driven system to handle multiple requests.
2. Which one of the following statements imports a script into your Node.js application?
   A. var http = import('http’);
   B. var http = include('http’);
   C. var http = request('http’);
   D. var http = require('http');
   The answer is D.
3. True or False: Node.js is used for developing client-side applications that run on the web browser.
   The answer is False. Node.js is used to implement server-side applications.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-25. Review answers



IBM Training

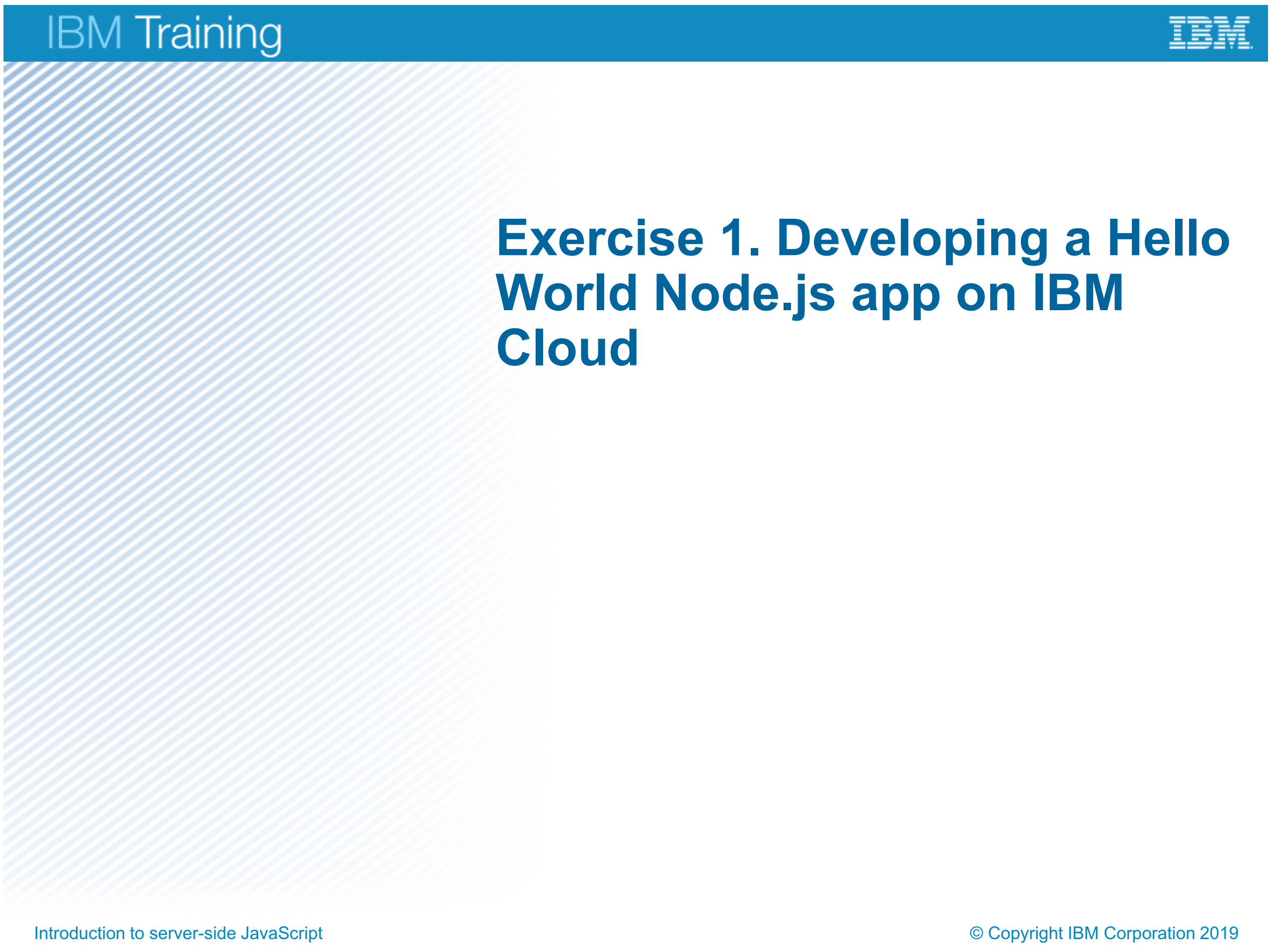
Review answers (cont.)

4. True or False: Node.js is not suitable for CPU-intensive workloads. The answer is True. Node.js is a single-threaded application environment. So, using CPU-intensive workloads blocks other requests coming to the application from being processed until the current operation ends.
5. What is the manifest file that describes details about a Node.js module?
   A. package.yaml
   B. manifest.json
   C. package.json
   D. manifest.yml

   The answer is C.

Introduction to server-side JavaScript © Copyright IBM Corporation 2019

Figure 1-26. Review answers (cont.)



IBM Training

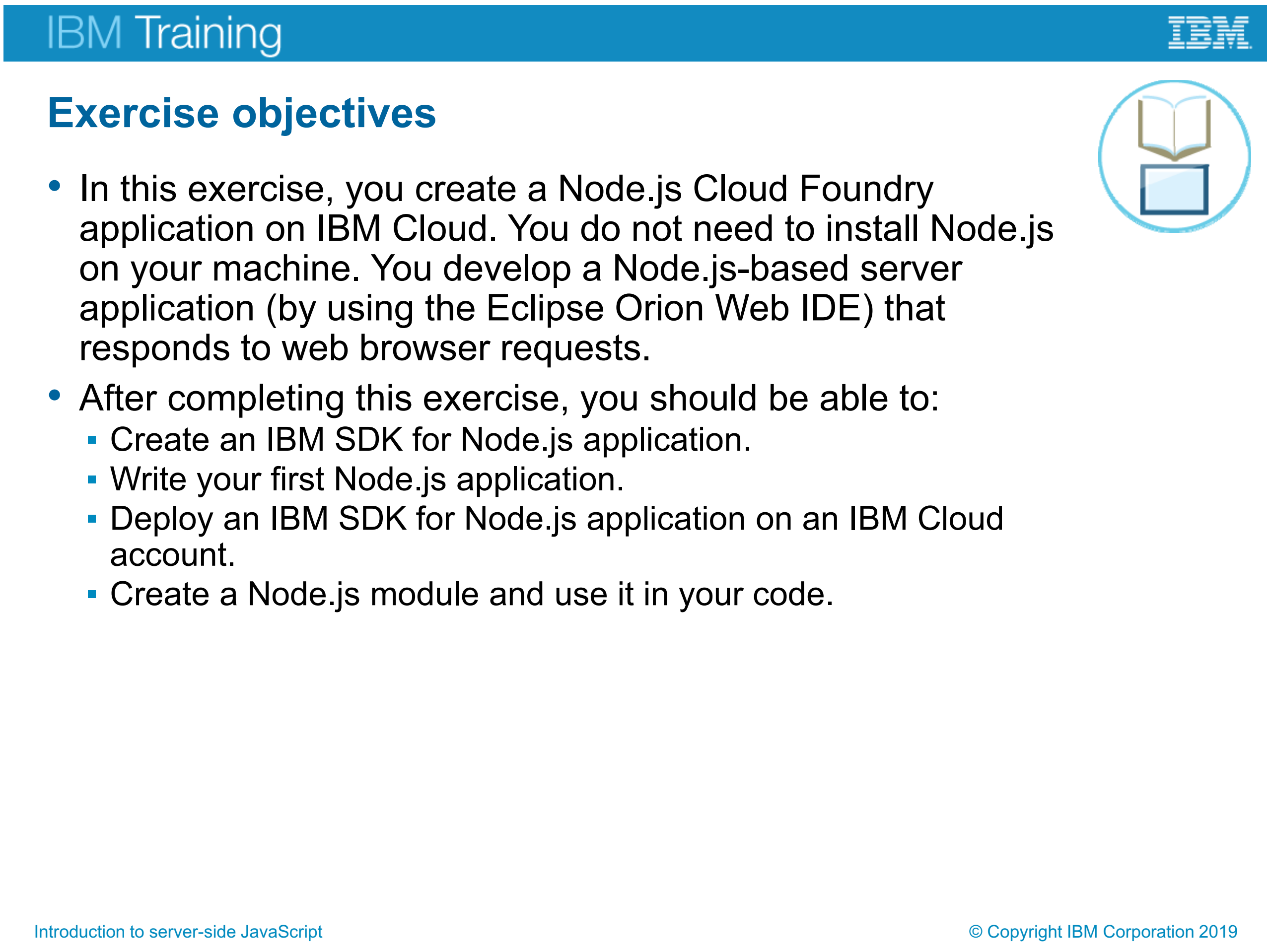
IBM

Exercise 1. Developing a Hello World Node.js app on IBM Cloud

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-27. Exercise 1. Developing a Hello World Node.js app on IBM Cloud



IBM Training

Exercise objectives

- In this exercise, you create a Node.js Cloud Foundry application on IBM Cloud. You do not need to install Node.js on your machine. You develop a Node.js-based server application (by using the Eclipse Orion Web IDE) that responds to web browser requests.
- After completing this exercise, you should be able to:
  - Create an IBM SDK for Node.js application.
  - Write your first Node.js application.
  - Deploy an IBM SDK for Node.js application on an IBM Cloud account.
  - Create a Node.js module and use it in your code.

Introduction to server-side JavaScript

© Copyright IBM Corporation 2019

Figure 1-28. Exercise objectives

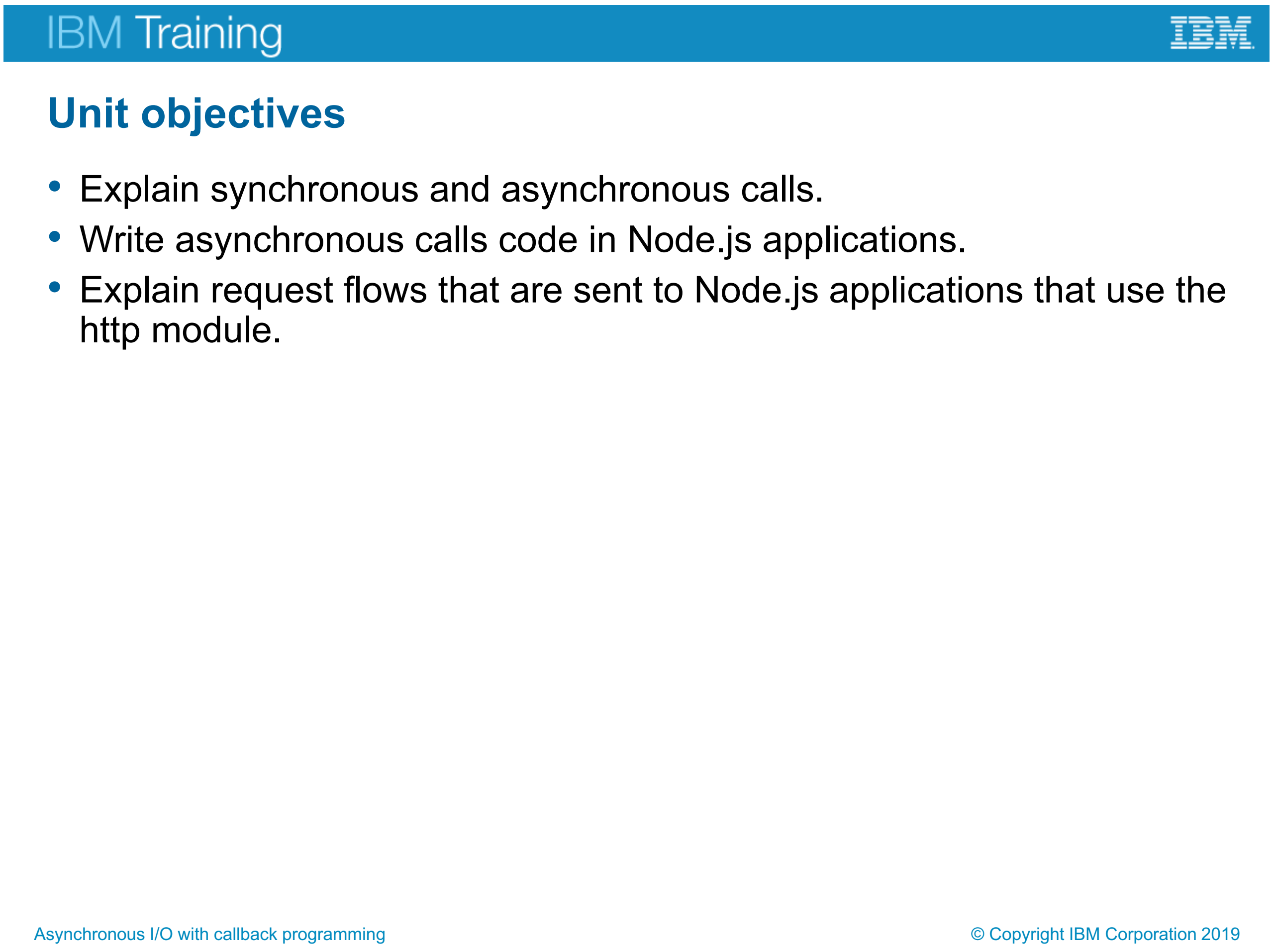
Unit 2. Asynchronous I/O with callback programming

Estimated time

01:30

Overview

The Node.js SDK relies on callback functions to handle network calls in an asynchronous manner. In this unit, you will learn how to write anonymous callback functions to act upon network events and listen, and intercept network traffic.



IBM Training

Unit objectives

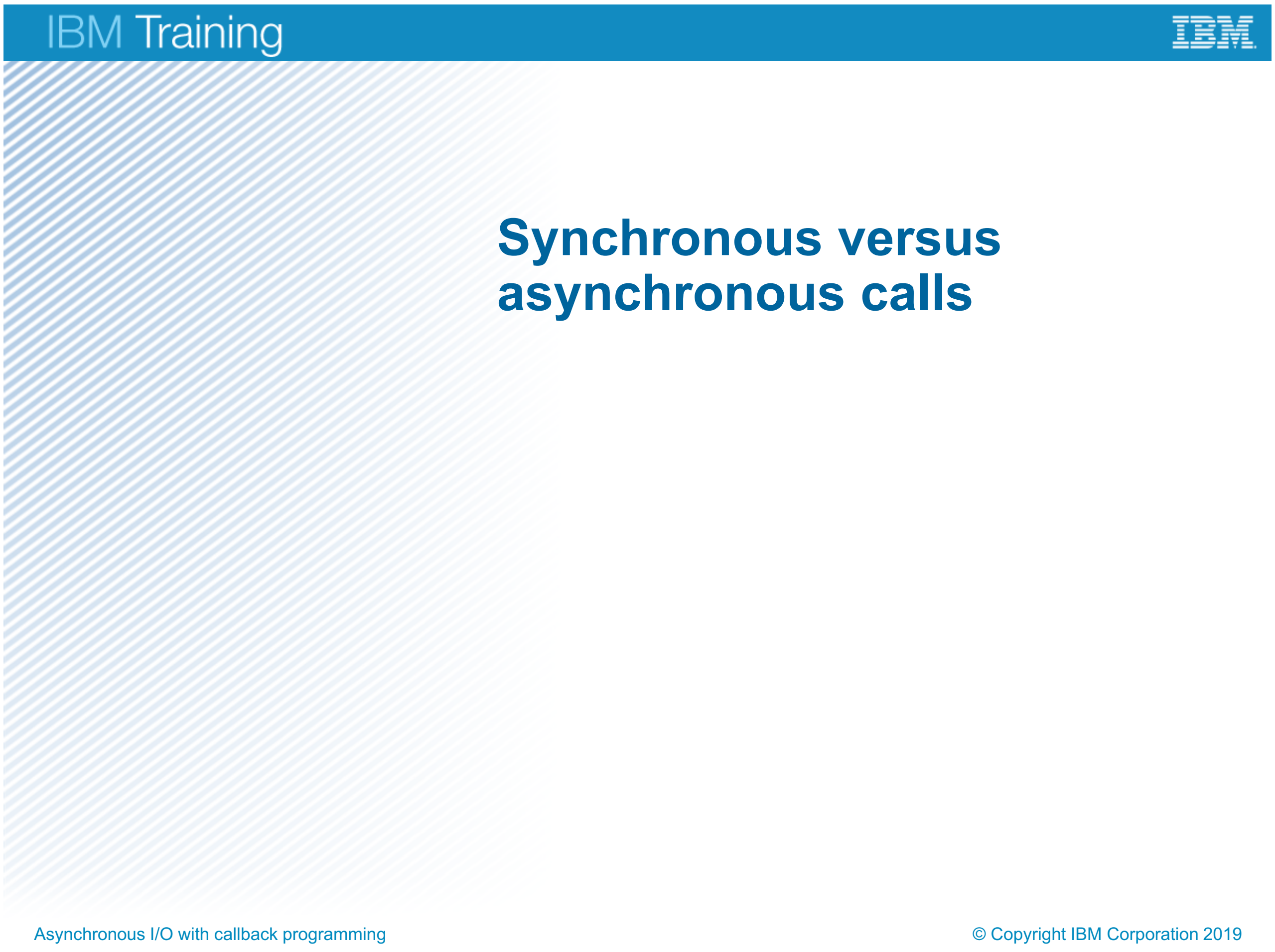
- Explain synchronous and asynchronous calls.
- Write asynchronous calls code in Node.js applications.
- Explain request flows that are sent to Node.js applications that use the http module.

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-1. Unit objectives

2.1. Synchronous versus asynchronous calls



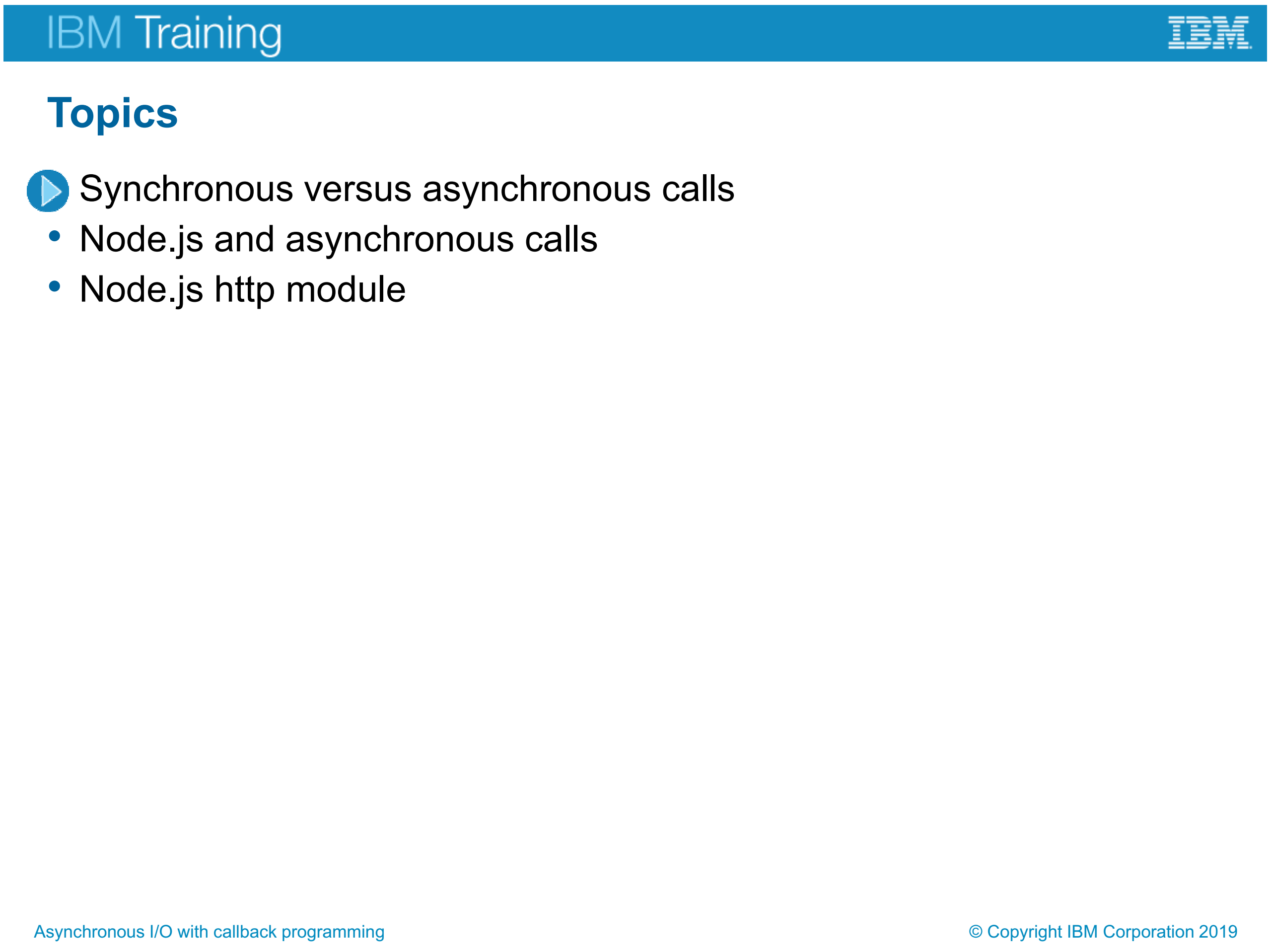
IBM Training

Synchronous versus asynchronous calls

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-2. Synchronous versus asynchronous calls


IBM Training

Topics

- Synchronous versus asynchronous calls
- Node.js and asynchronous calls
- Node.js http module

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-3. Topics

IBM Training

Synchronous calls

- With *synchronous calls*, the caller waits for the response before proceeding to other operations.
- When an application blocks (or waits) for an operation to complete, the application wastes processing time on the server.

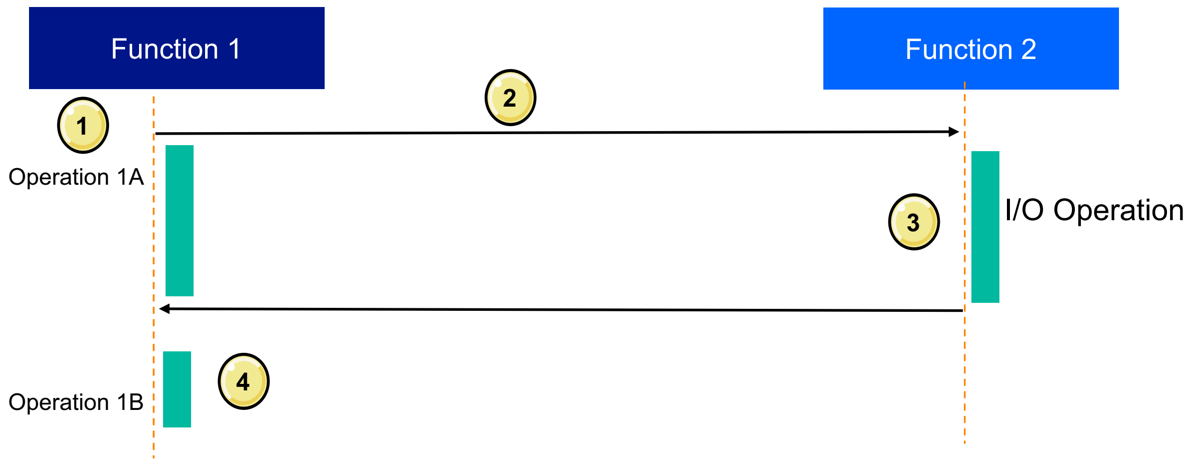


Figure 2-4. Synchronous calls

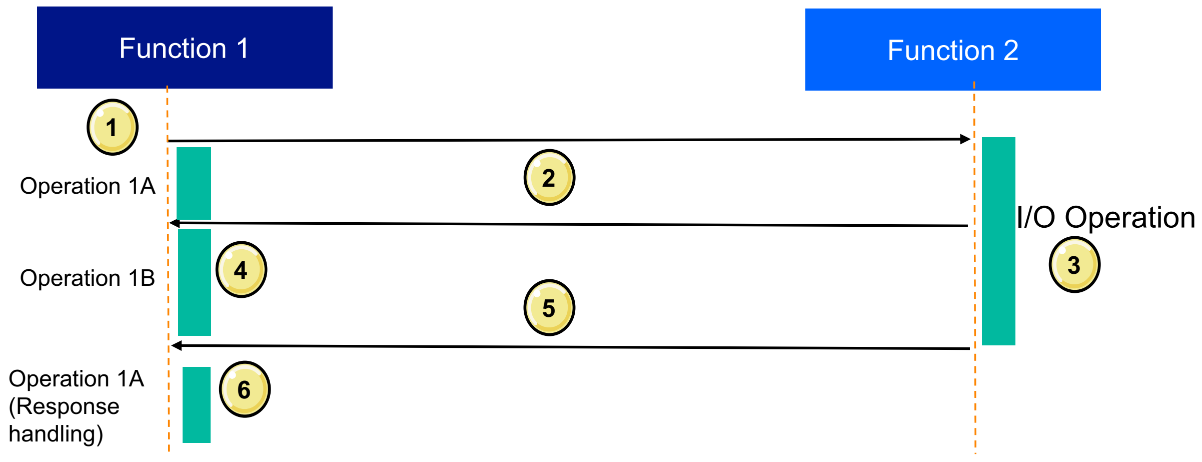
Consider the following example where Function 1 is performing two operations (Operation 1A and Operation 1B):

1. Function 1 is performing Operation 1 A.
2. Operation 1A calls Function 2.
3. Function 2 is performing an I/O operation that takes some time.
4. Function 1 is blocked from proceeding to Operation 1B until Function 2 is complete and returns the response to Function 1.

IBM Training

Asynchronous calls

- With *asynchronous calls*, the caller does not wait for the response, and can proceed to other operations.
- When the response is ready, the caller is triggered to handle the logic that was waiting for this response.



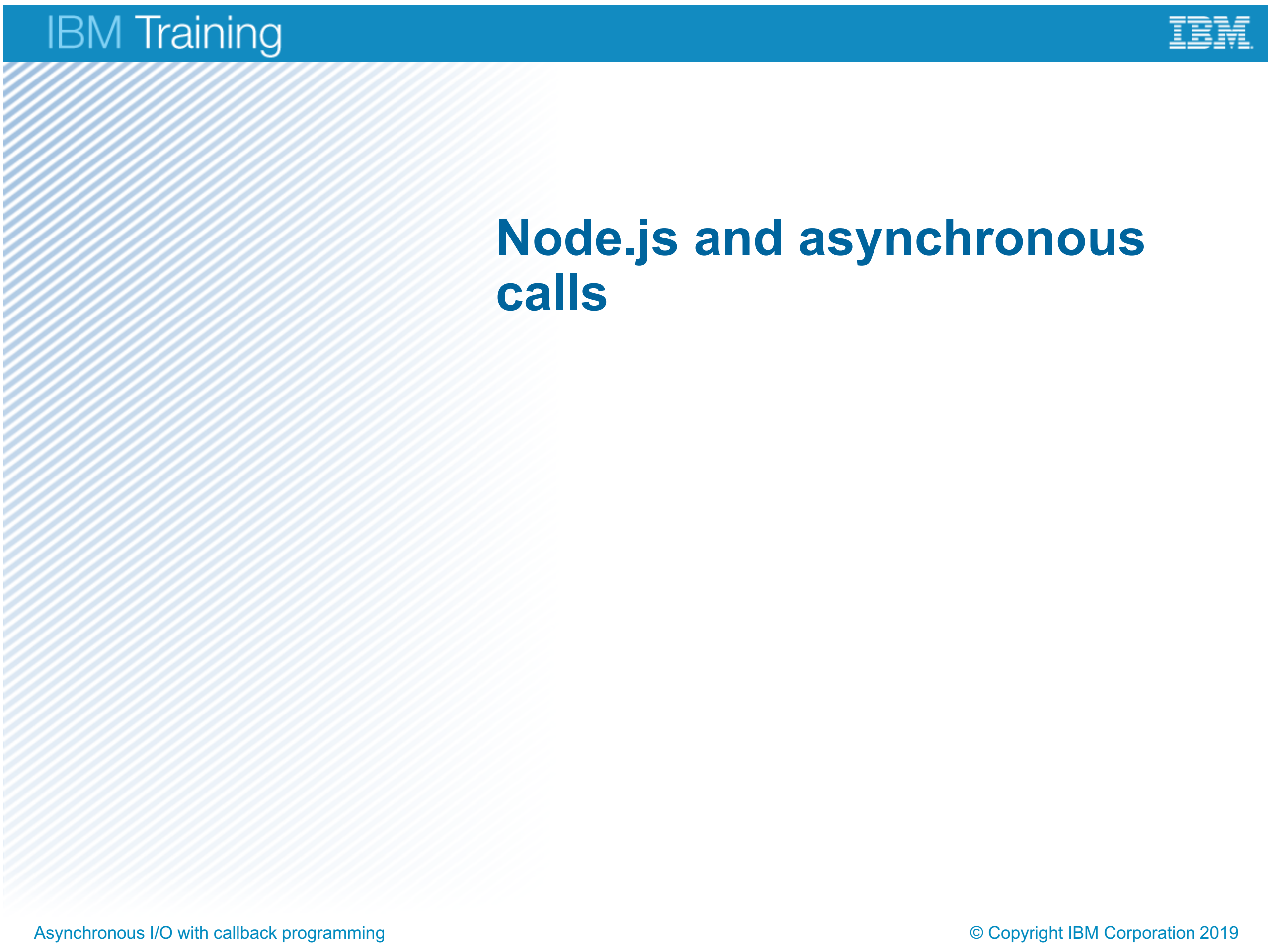
Asynchronous I/O with callback programming

Figure 2-5. Asynchronous calls

Consider the following example where Function 1 is performing two operations (Operation 1A and Operation 1B):

1. Function 1 is performing Operation 1 A.
2. Operation 1A calls Function 2.
3. Function 2 is performing an I/O operation that takes some time.
4. Function 1 is not blocked, and proceeds to Operation 1B.
5. After Function 2 is done running the operation, Function 1 is triggered, Operation 1A is complete, and the response is ready to be received.
6. Function 1 runs the required logic that was waiting for the response from Function 2.

2.2. Node.js and asynchronous calls



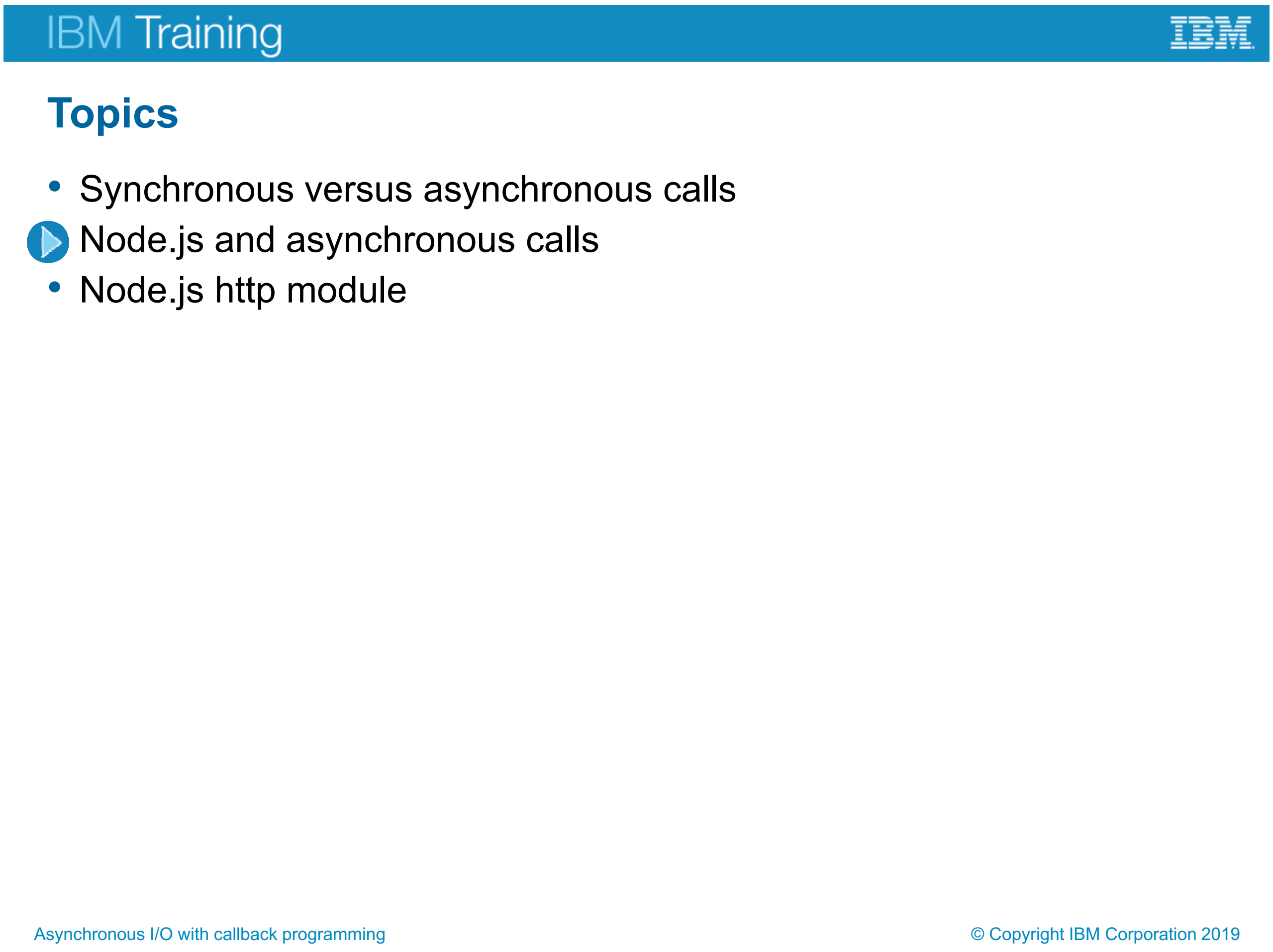
IBM Training

Node.js and asynchronous calls

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-6. Node.js and asynchronous calls



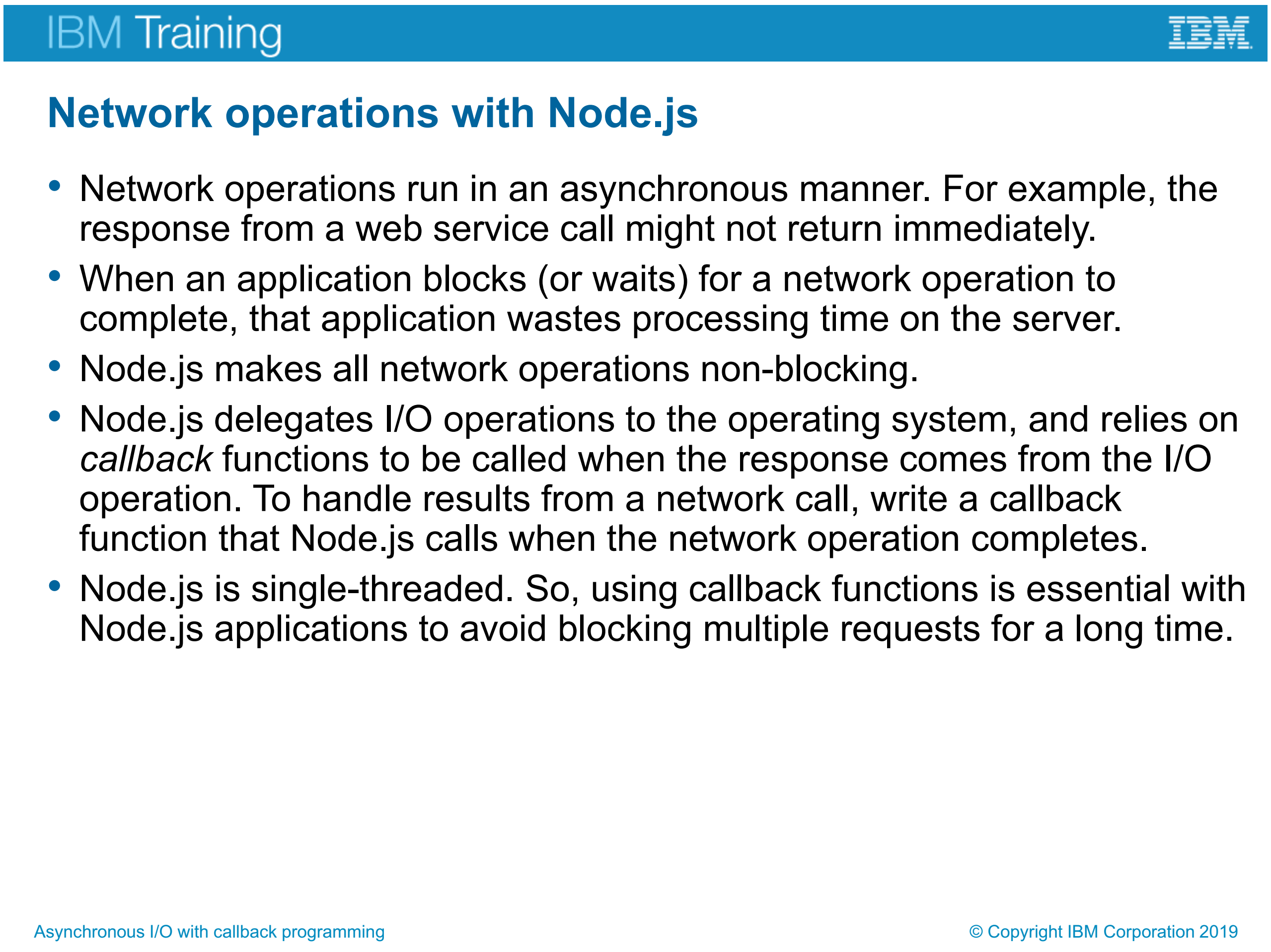
IBM Training

Topics

- Synchronous versus asynchronous calls
- ▶ Node.js and asynchronous calls
- Node.js http module

Asynchronous I/O with callback programming

Figure 2-7. Topics



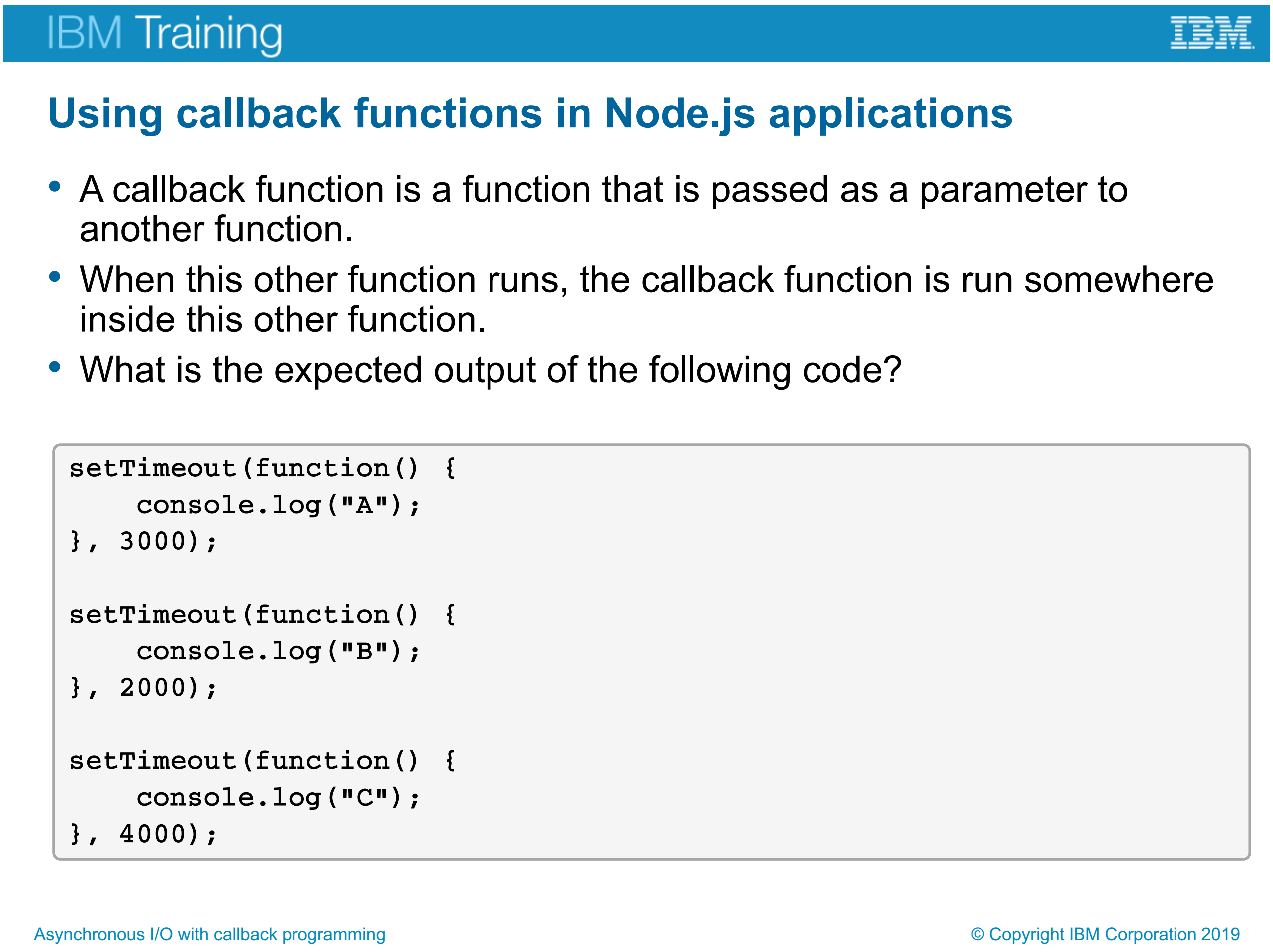
IBM Training

Network operations with Node.js

- Network operations run in an asynchronous manner. For example, the response from a web service call might not return immediately.
- When an application blocks (or waits) for a network operation to complete, that application wastes processing time on the server.
- Node.js makes all network operations non-blocking.
- Node.js delegates I/O operations to the operating system, and relies on *callback* functions to be called when the response comes from the I/O operation. To handle results from a network call, write a callback function that Node.js calls when the network operation completes.
- Node.js is single-threaded. So, using callback functions is essential with Node.js applications to avoid blocking multiple requests for a long time.

Asynchronous I/O with callback programming

Figure 2-8. Network operations with Node.js



IBM Training

Using callback functions in Node.js applications

- A callback function is a function that is passed as a parameter to another function.
- When this other function runs, the callback function is run somewhere inside this other function.
- What is the expected output of the following code?

```
setTimeout(function() {
    console.log("A");
}, 3000);

setTimeout(function() {
    console.log("B");
}, 2000);

setTimeout(function() {
    console.log("C");
}, 4000);
```

Asynchronous I/O with callback programming

Figure 2-9. Using callback functions in Node.js applications

The **`setTimeout`** function in this slide is a function that receives the following parameters:

1. A callback function that should be run after a certain time interval.

2. The time interval (in milliseconds) that should elapse before the callback function can run.

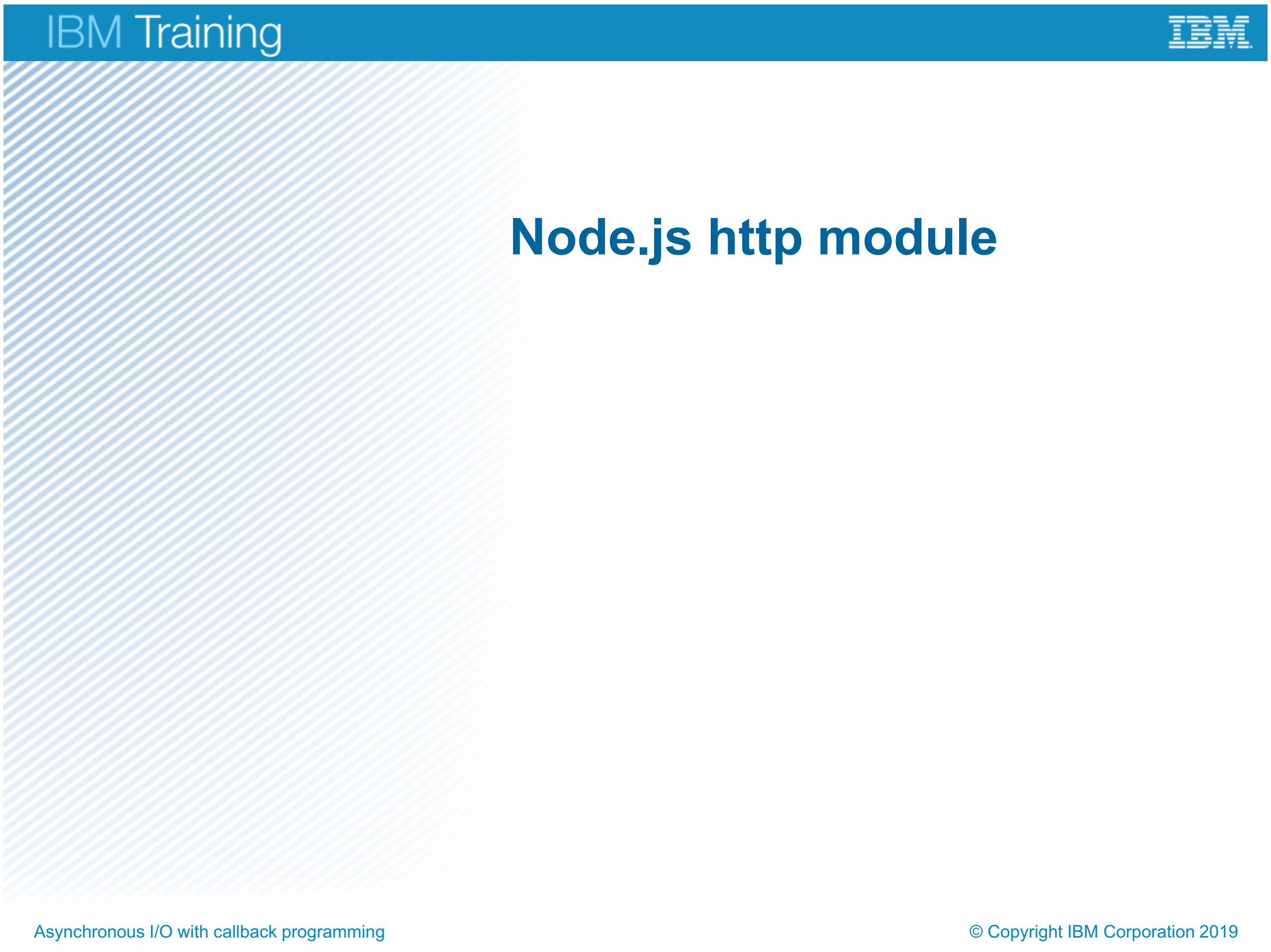
When calling the **`setTimeout`** function, Node.js is not waiting for the processing to finish. Instead, Node.js registers the callback function for it to be run after the response is returned.

When the code in the example runs, the following output is returned in the console:

```
B

A

C
```

The result depends on the sequence of the **`setTimeout`** functions finishing their run and not on the sequence of their declaration in the JavaScript file.

2.3. Node.js http module



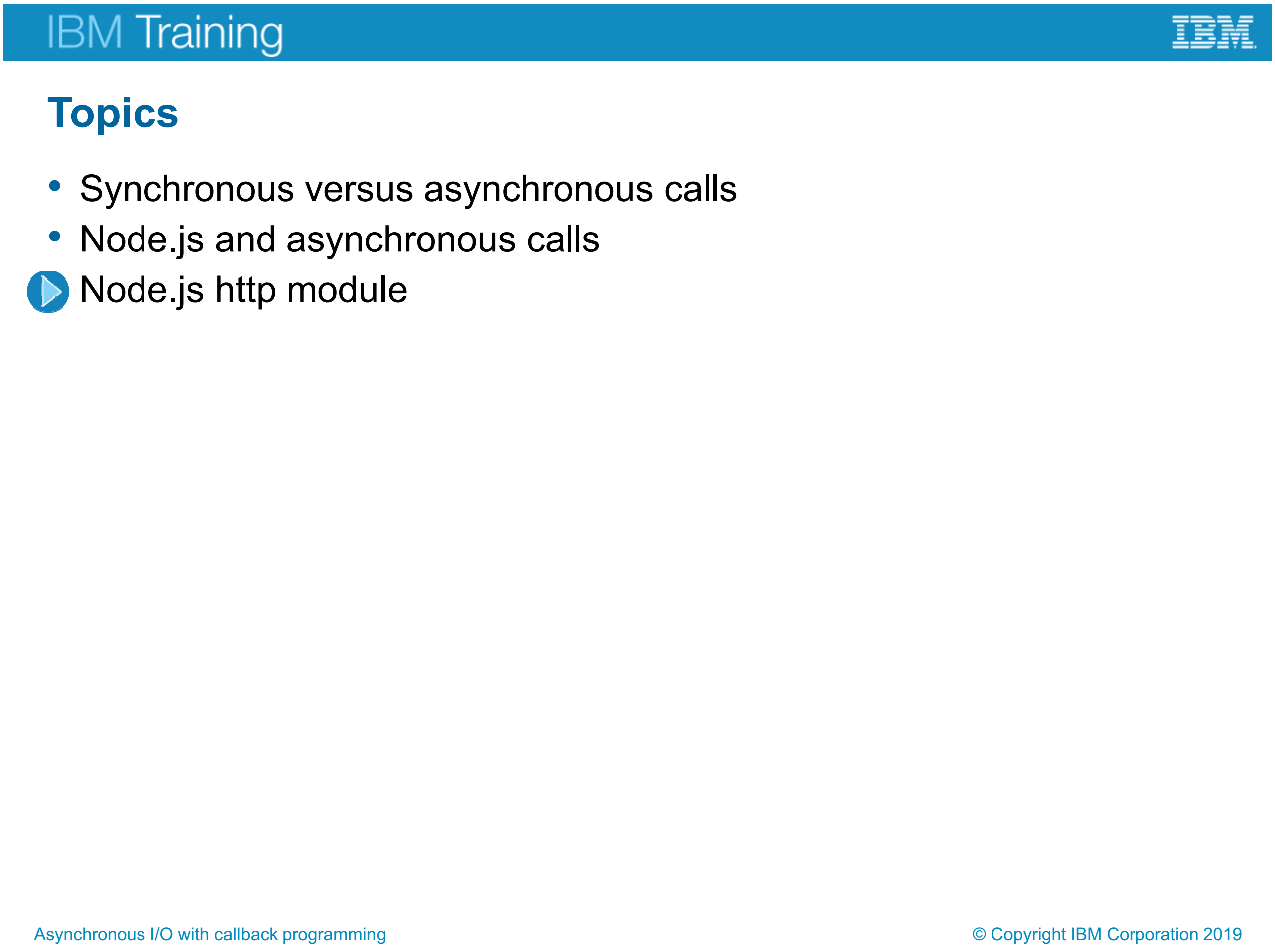
IBM Training

Node.js http module

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-10. Node.js http module



IBM Training

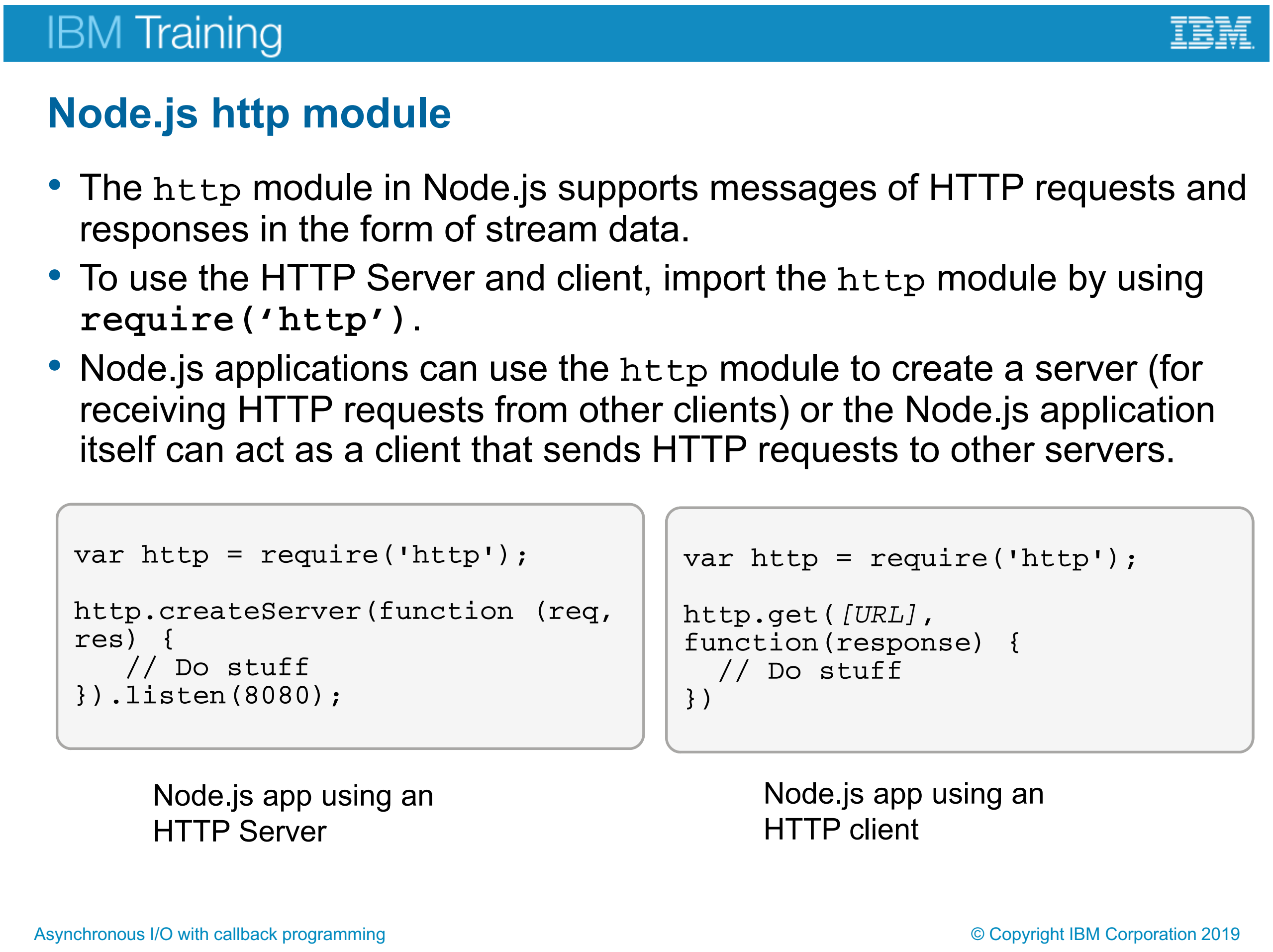
Topics

- Synchronous versus asynchronous calls
- Node.js and asynchronous calls
- Node.js http module

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-11. Topics



IBM Training

Node.js http module

- The `http` module in Node.js supports messages of HTTP requests and responses in the form of stream data.
- To use the HTTP Server and client, import the `http` module by using **`require('http')`**.
- Node.js applications can use the `http` module to create a server (for receiving HTTP requests from other clients) or the Node.js application itself can act as a client that sends HTTP requests to other servers.

```
var http = require('http');

http.createServer(function (req,
res) {
    // Do stuff
}).listen(8080);
```

Node.js app using an HTTP Server

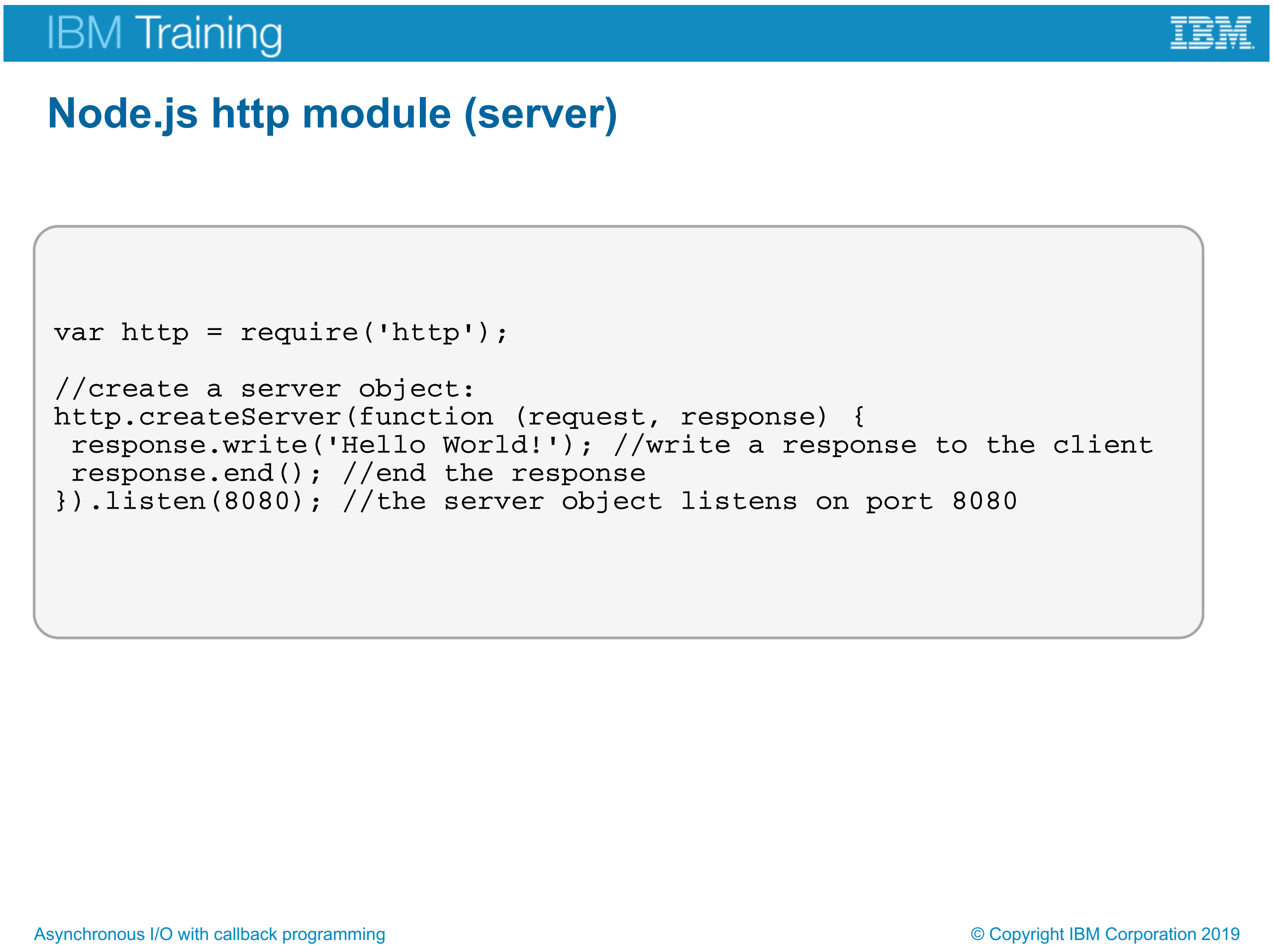
```
var http = require('http');

http.get([URL],
function(response) {
   // Do stuff
})
```

Node.js app using an HTTP client

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-12. Node.js http module



IBM Training

Node.js http module (server)

```
var http = require('http');

//create a server object:
http.createServer(function (request, response) {
 response.write('Hello World!'); //write a response to the client
 response.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-13. Node.js http module (server)

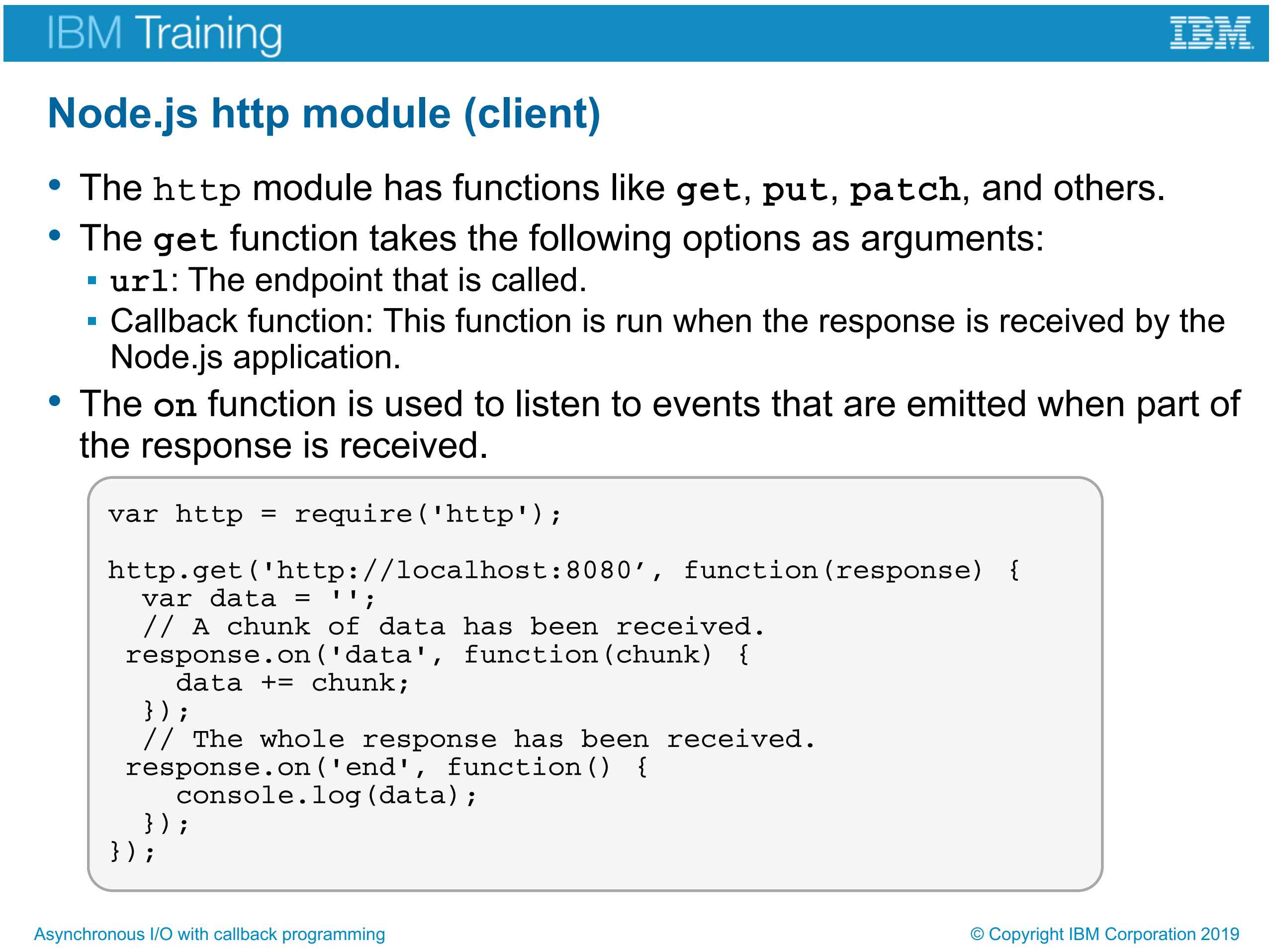
The http module contains the **createServer** function that is used to create a web server in the Node.js application.

The created HTTP server listens on a certain port to receive the HTTP requests from the callers.

The HTTP server has a callback function that is run when a request is received.

This callback function has the following arguments:

- **request**: This object contains the request details that are sent by the caller.
- **response**: This object is used to construct the response that is to be sent back to the caller.



IBM Training

Node.js http module (client)

- The `http` module has functions like **`get`**, **`put`**, **`patch`**, and others.
- The **`get`** function takes the following options as arguments:
  - **`url`**: The endpoint that is called.
  - Callback function: This function is run when the response is received by the Node.js application.
- The **`on`** function is used to listen to events that are emitted when part of the response is received.

```
var http = require('http');

http.get('http://localhost:8080', function(response) {
  var data = '';
  // A chunk of data has been received.
 response.on('data', function(chunk) {
    data += chunk;
  });
  // The whole response has been received.
 response.on('end', function() {
    console.log(data);
  });
});
```

Asynchronous I/O with callback programming © Copyright IBM Corporation 2019

Figure 2-14. Node.js http module (client)

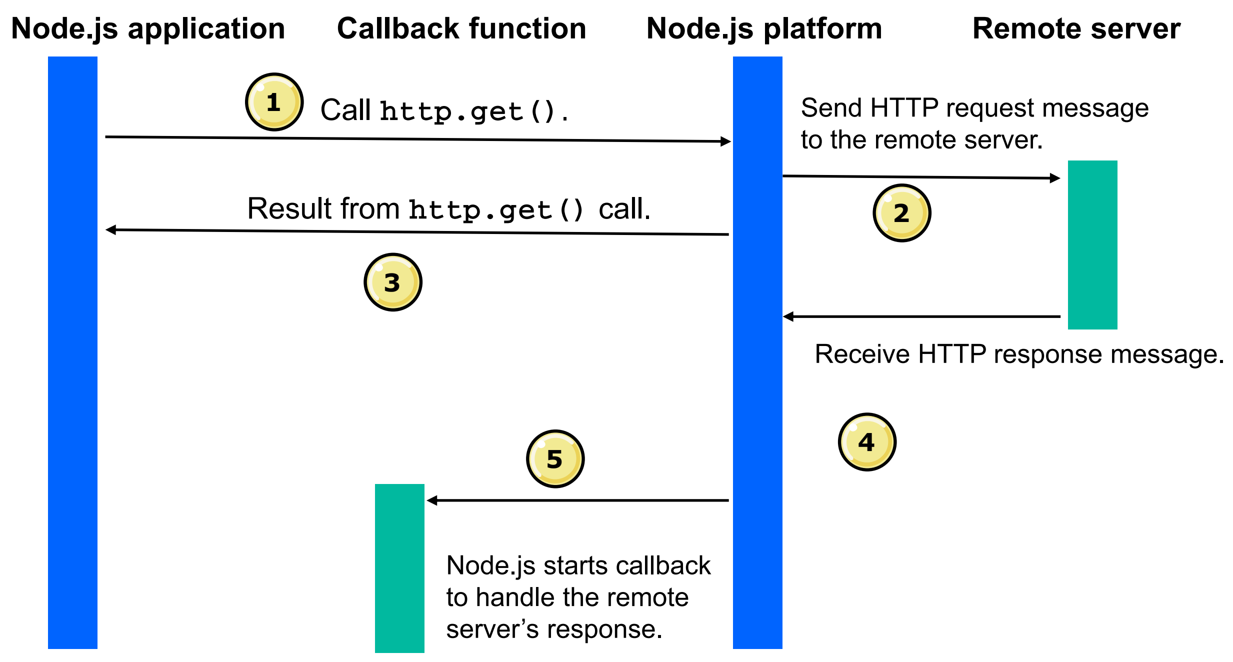
- The `http` module has functions like **`get`**, **`put`**, **`patch`**, and others.
- The **get** function takes the following options as arguments:
  - **url**: The endpoint that is called.
  - Callback function: This function is run when the response is being received by the Node.js application.
- The **on** function is used to listen to events that are emitted when part of the response is received. The following events are emitted:
  - **'data'** is emitted when a chunk of the data is received.
  - **'end' =>** is emitted when the whole response is received.

In the code that is shown in the slide, the *message* variable is initialized as an empty string. Whenever the **'data'** event is triggered, the received data is appended to the *data* variable.

When the **'end'** event it triggered, the whole response is received. So, we can log the content of the *message* variable and expect it to have the full received message from the response.

IBM Training

The http.get application interactions


Node.js application
Callback function
Node.js platform
Remote server
1 Call http.get().
Send HTTP request message to the remote server.
2
Result from http.get() call.
3
Receive HTTP response message.
4
5
Node.js starts callback to handle the remote server's response.

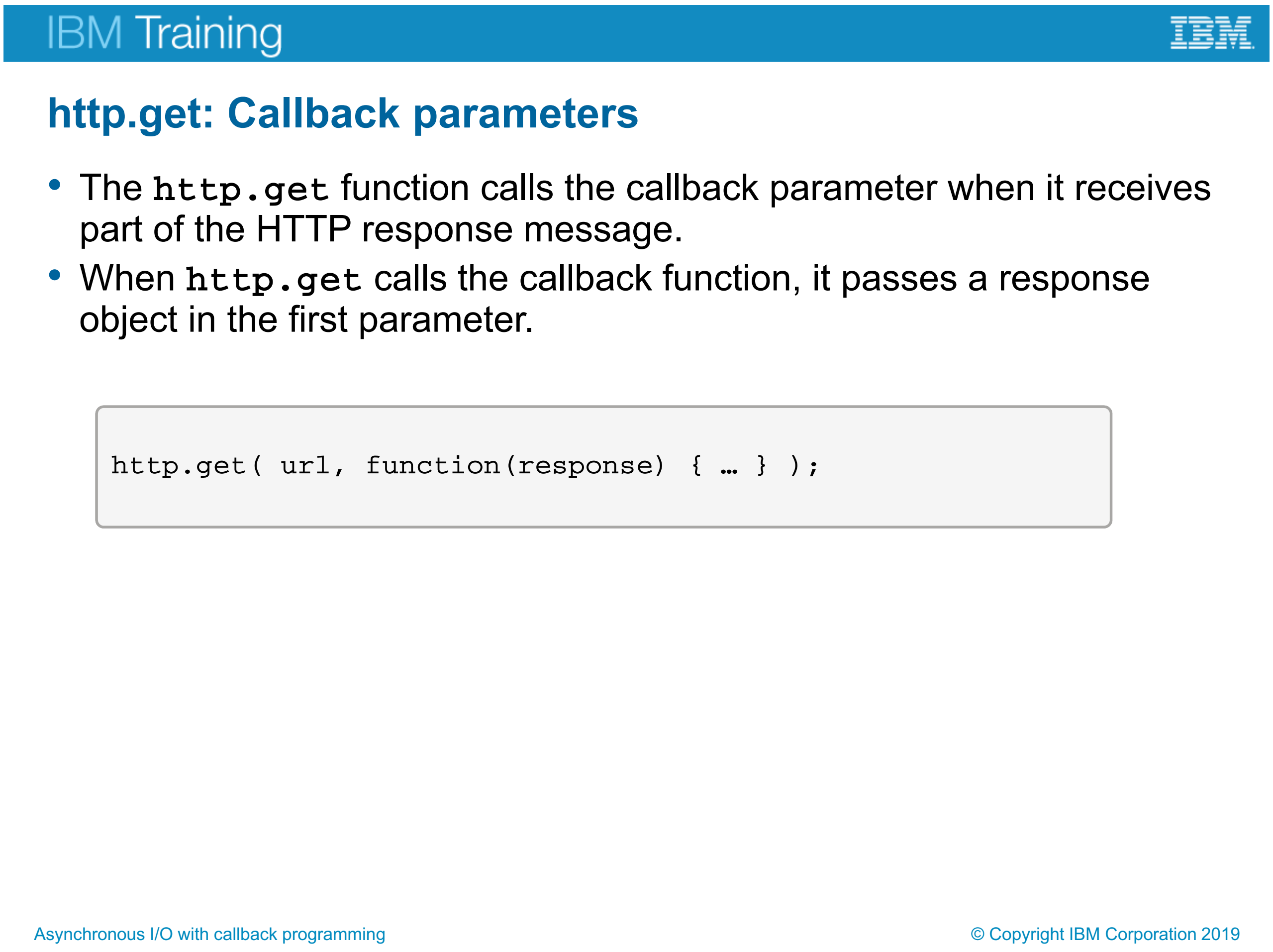
Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-15. The http.get application interactions

The main application calls **http.get**. This sequence diagram shows the interaction between the application, the Node.js, the web service call to the remote server, and the callback to the callback function.

1. In the first step, the application makes a call to **http.get()**.
2. This function makes a call to the remote web server. It makes a request to the web service.
3. Before Node.js receives the HTTP response message from the remote web server, it immediately returns a result for the **http.get** function call. This result simply indicates that the request message was sent successfully. It does not say anything about the response message.
4. The remote server replies to Node.js with the HTTP response.
5. When Node.js receives an HTTP response message from the remote server, it calls the callback function that you defined during the **http.get** function call. This function handles the HTTP response message.



IBM Training

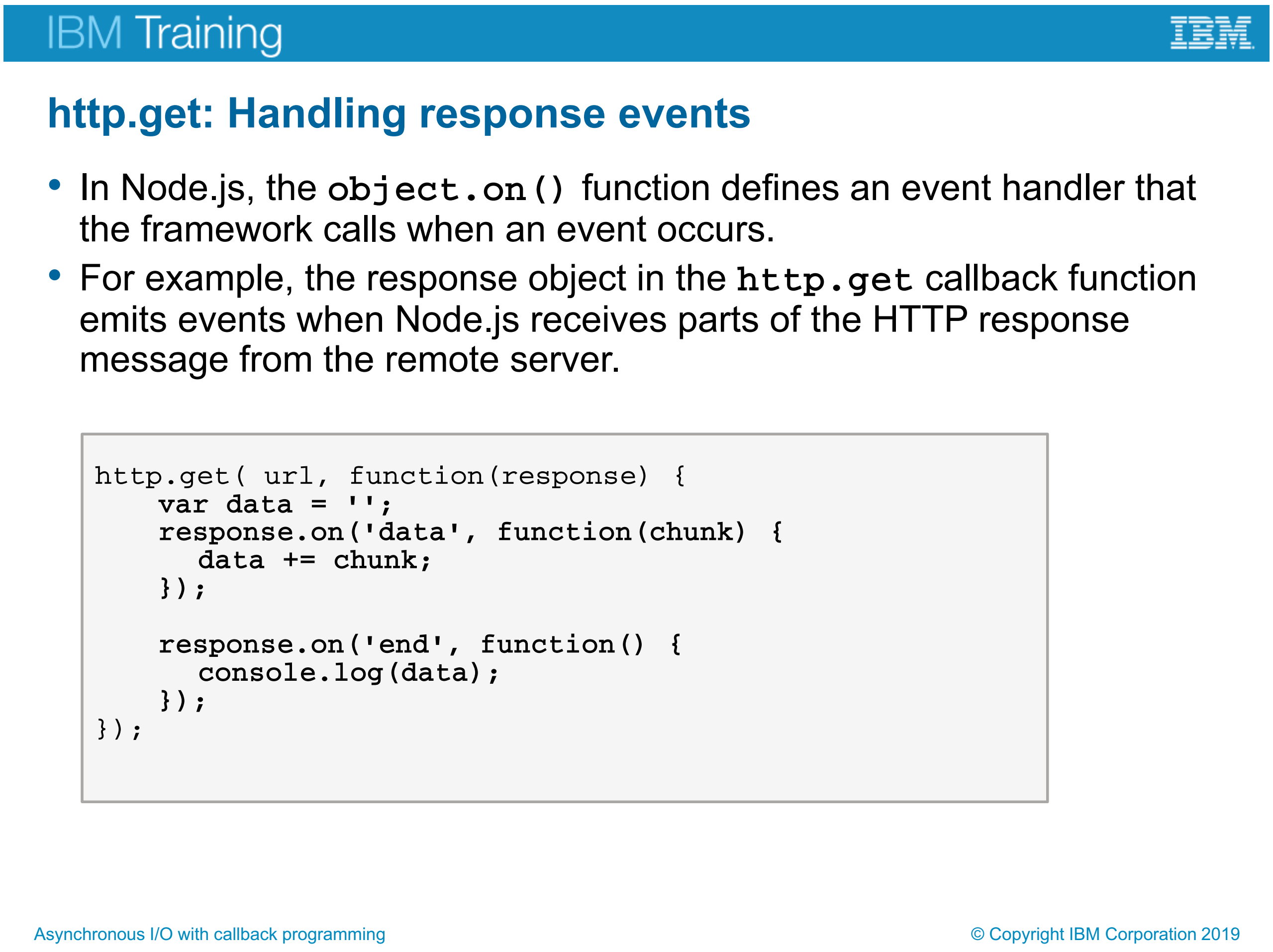
http.get: Callback parameters

- The **`http.get`** function calls the callback parameter when it receives part of the HTTP response message.
- When **`http.get`** calls the callback function, it passes a response object in the first parameter.

```
http.get( url, function(response) { … } );
```

Asynchronous I/O with callback programming

Figure 2-16. http.get: Callback parameters



IBM Training

http.get: Handling response events

- In Node.js, the `object.on()` function defines an event handler that the framework calls when an event occurs.
- For example, the response object in the `http.get` callback function emits events when Node.js receives parts of the HTTP response message from the remote server.

```
http.get( url, function(response) {
    var data = '';
    response.on('data', function(chunk) {
       data += chunk;
    });

    response.on('end', function() {
       console.log(data);
    });
});
```

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

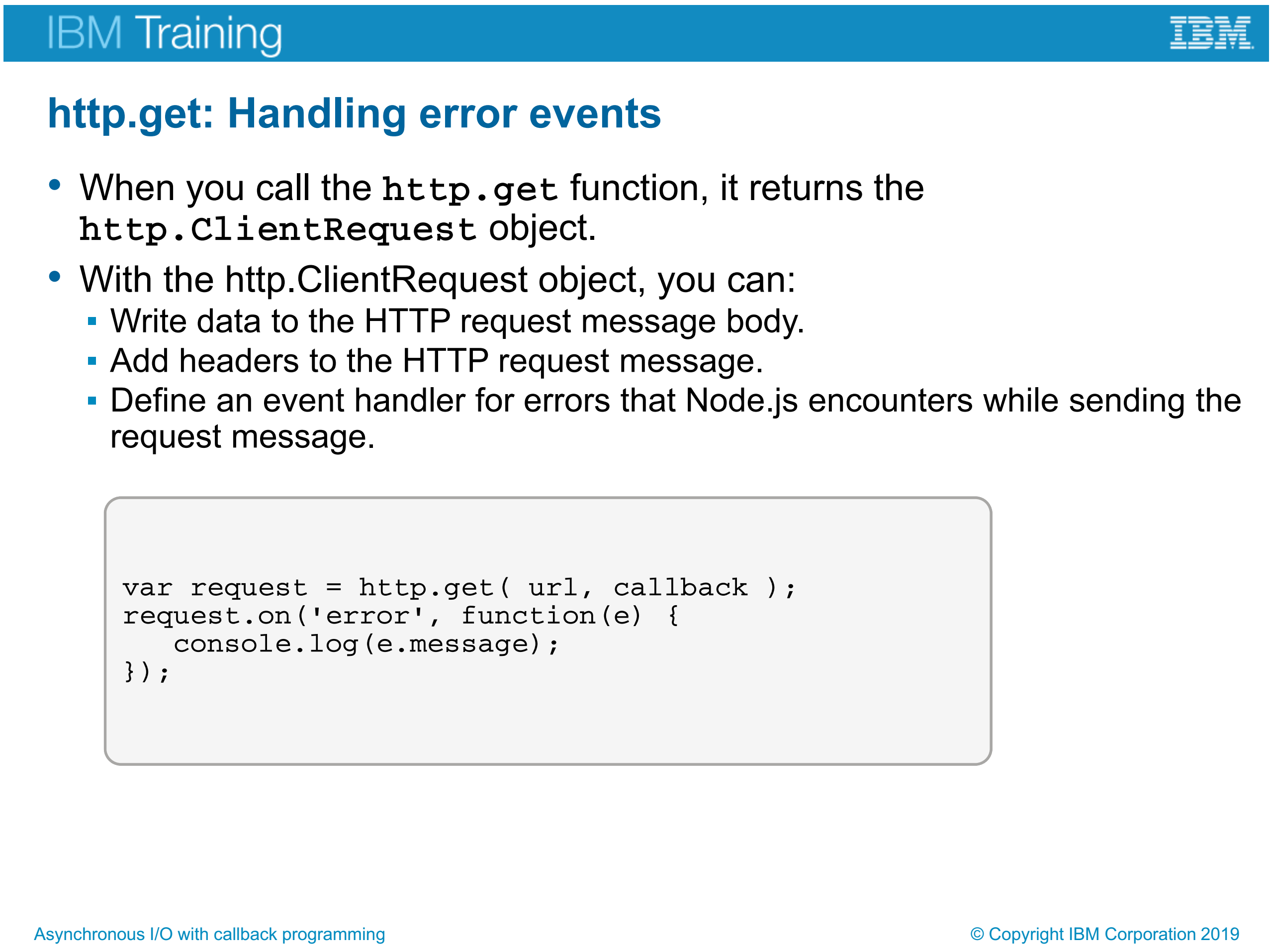
Figure 2-17. http.get: Handling response events

The response object in the **`http.get`** callback function emits events when Node.js receives parts of the HTTP response message from the remote server:

- A **'data'** event is emitted when part of the HTTP response message is received.
- An **'end'** event is emitted when the whole response is received.

The emitted events can be listened to by using the **on** function. The **on** function has the following parameters:

- An event name.
- The callback function that is called when the corresponding event is received.



IBM Training

http.get: Handling error events

- When you call the `http.get` function, it returns the `http.ClientRequest` object.
- With the http.ClientRequest object, you can:
  - Write data to the HTTP request message body.
  - Add headers to the HTTP request message.
  - Define an event handler for errors that Node.js encounters while sending the request message.

```
var request = http.get( url, callback );
request.on('error', function(e) {
   console.log(e.message);
});
```

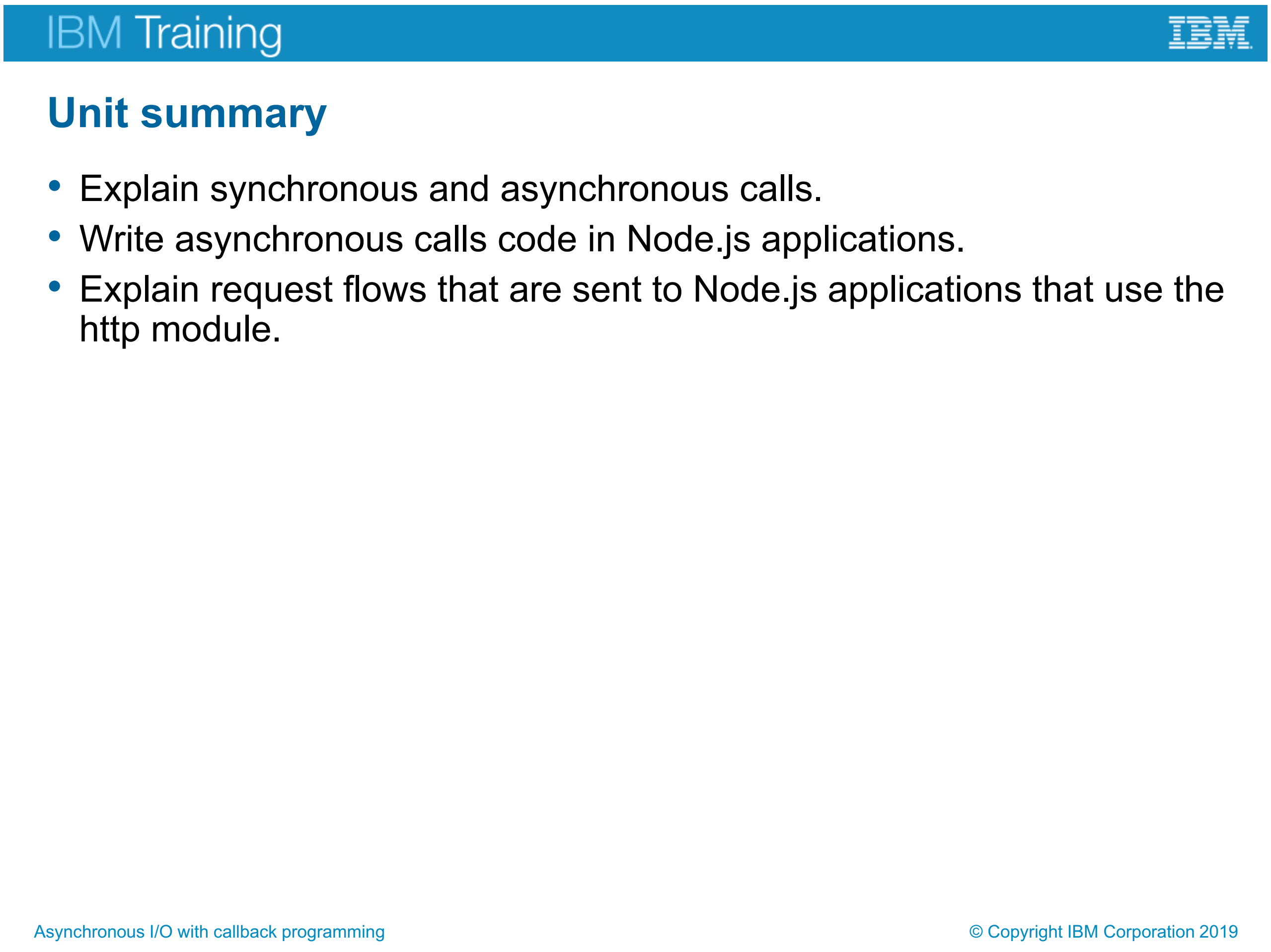
Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-18. http.get: Handling error events

The **http.get** function returns the **http.ClientRequest** object. This object is used to write data to the request body, add header attributes, and define error event handlers.

The example in this slide shows a listener that was added for the **'error'** events that might be emitted in case of errors.



IBM Training

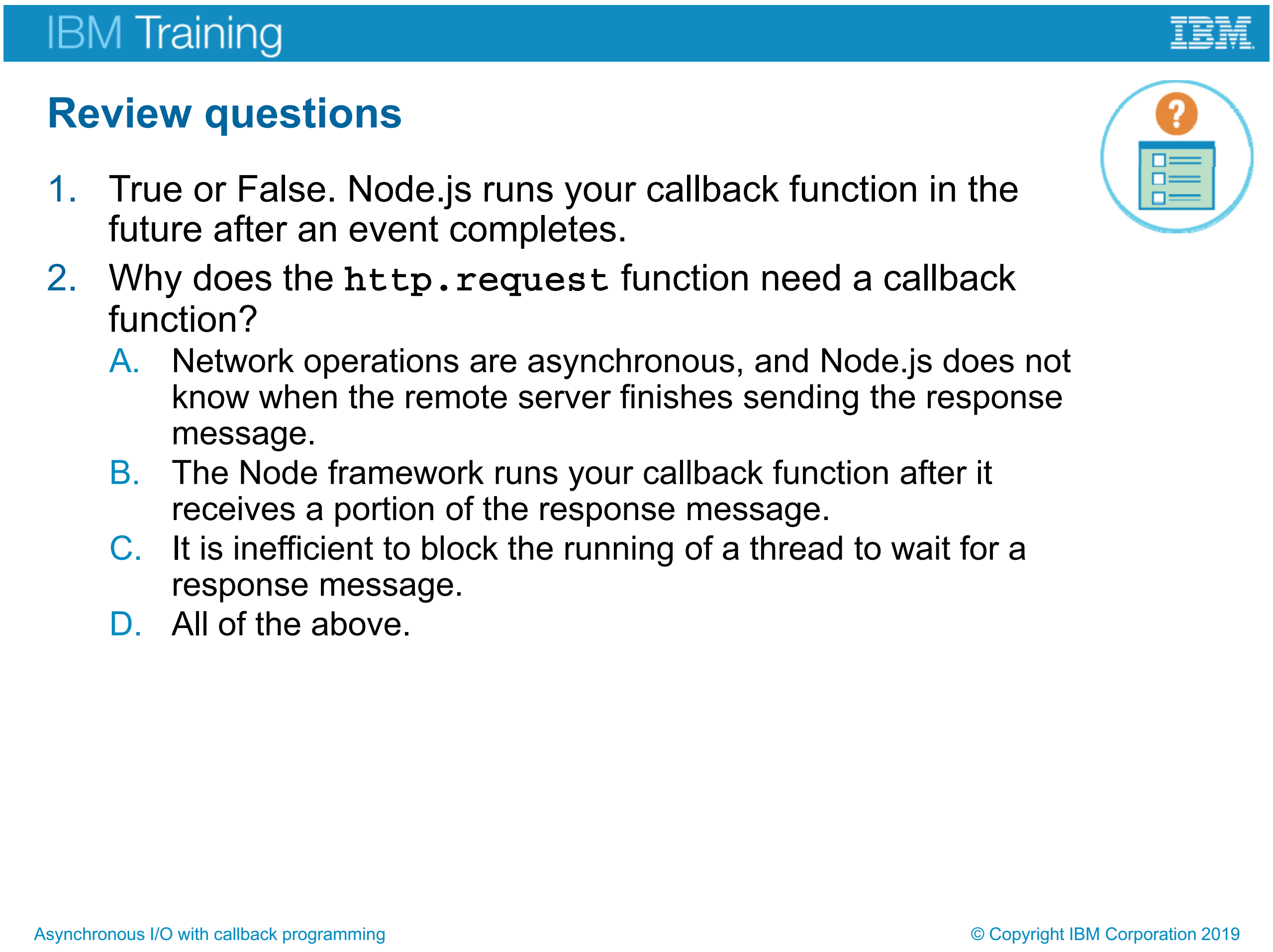
Unit summary

- Explain synchronous and asynchronous calls.
- Write asynchronous calls code in Node.js applications.
- Explain request flows that are sent to Node.js applications that use the http module.

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-19. Unit summary



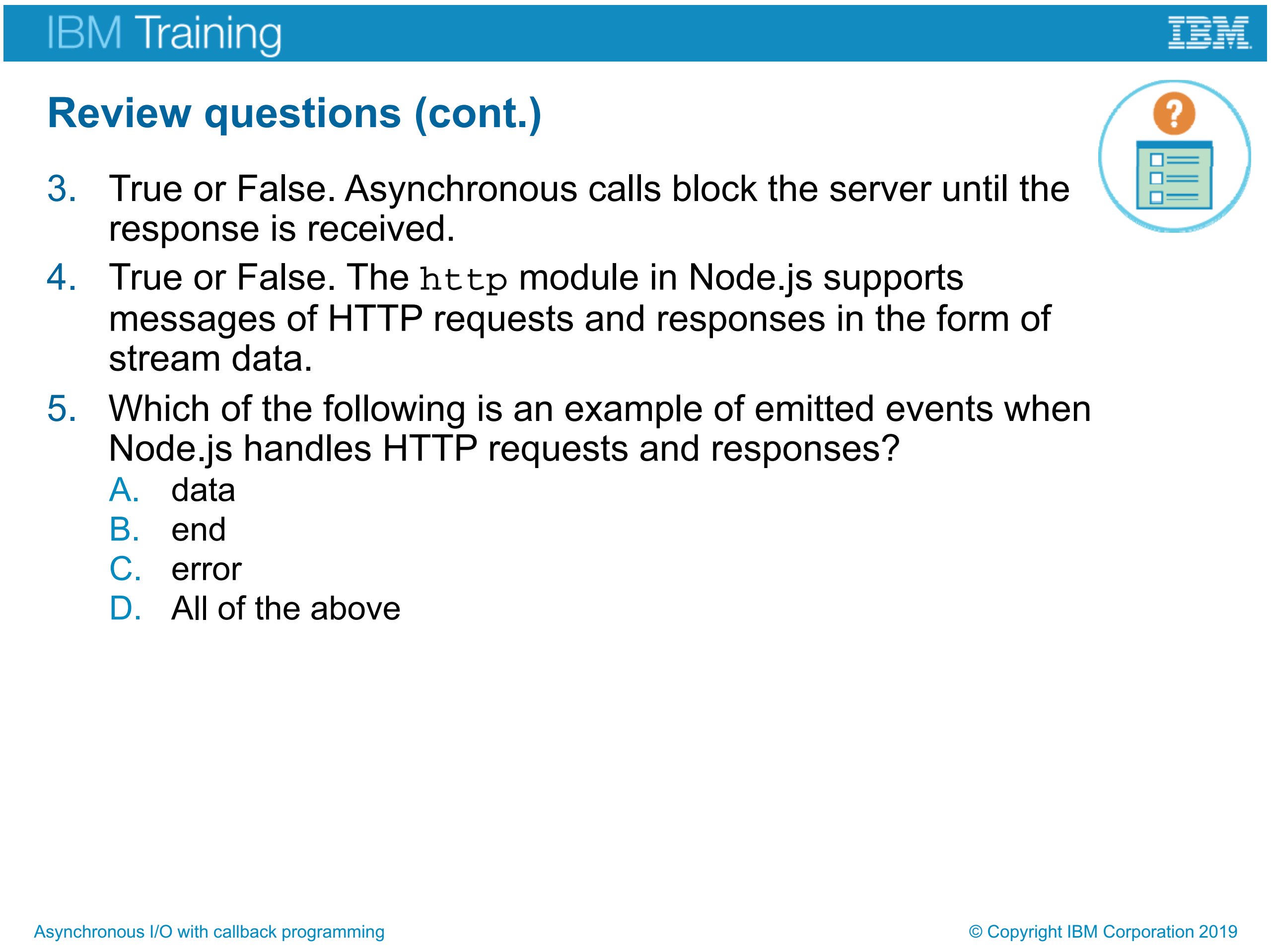
IBM Training

Review questions

1. True or False. Node.js runs your callback function in the future after an event completes.
2. Why does the `http.request` function need a callback function?
   - A. Network operations are asynchronous, and Node.js does not know when the remote server finishes sending the response message.
   - B. The Node framework runs your callback function after it receives a portion of the response message.
   - C. It is inefficient to block the running of a thread to wait for a response message.
   - D. All of the above.

Asynchronous I/O with callback programming

Figure 2-20. Review questions



IBM Training

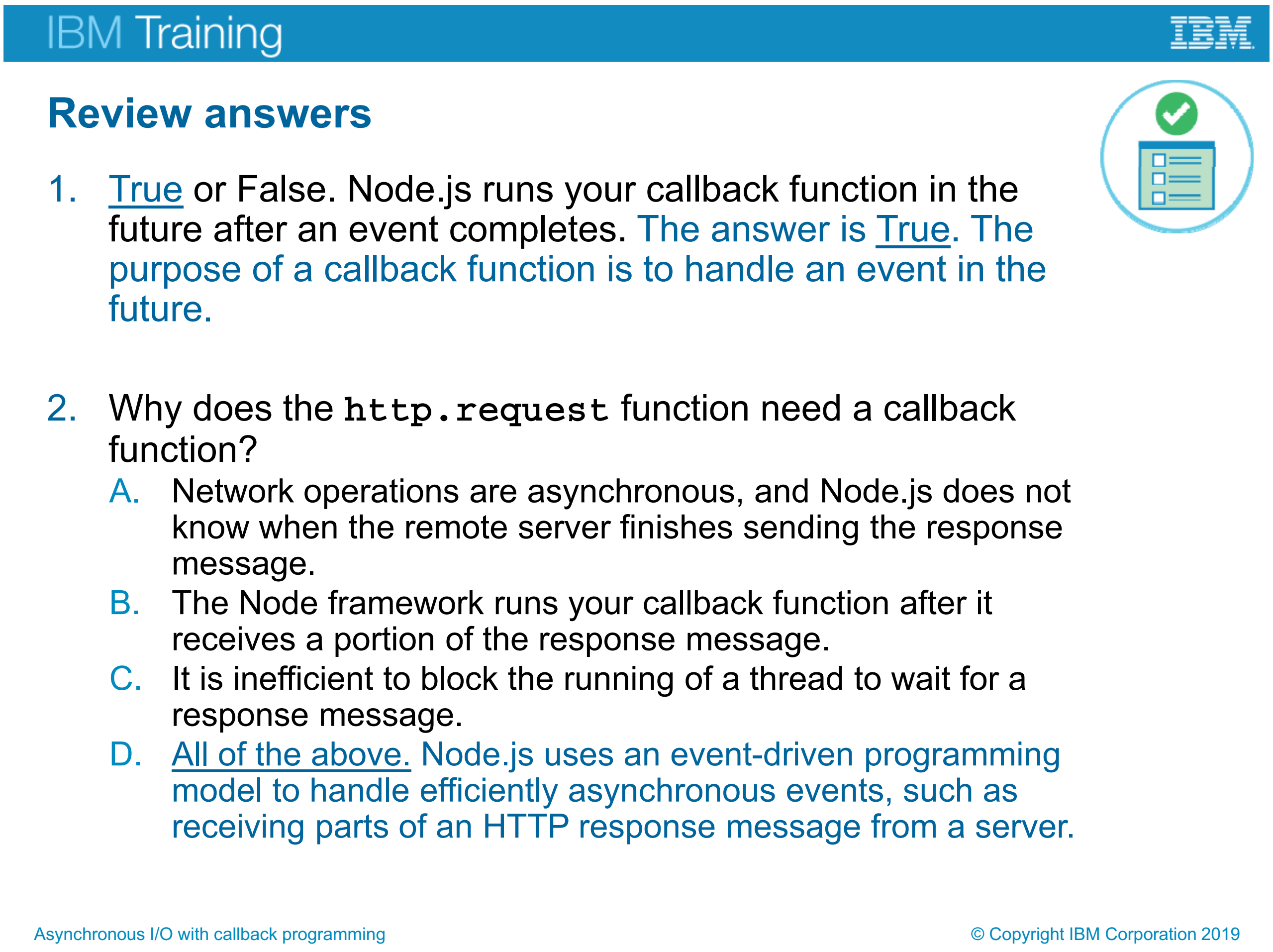
Review questions (cont.)

3. True or False. Asynchronous calls block the server until the response is received.
4. True or False. The `http` module in Node.js supports messages of HTTP requests and responses in the form of stream data.
5. Which of the following is an example of emitted events when Node.js handles HTTP requests and responses?
   A. data
   B. end
   C. error
   D. All of the above

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-21. Review questions (cont.)



IBM Training

Review answers

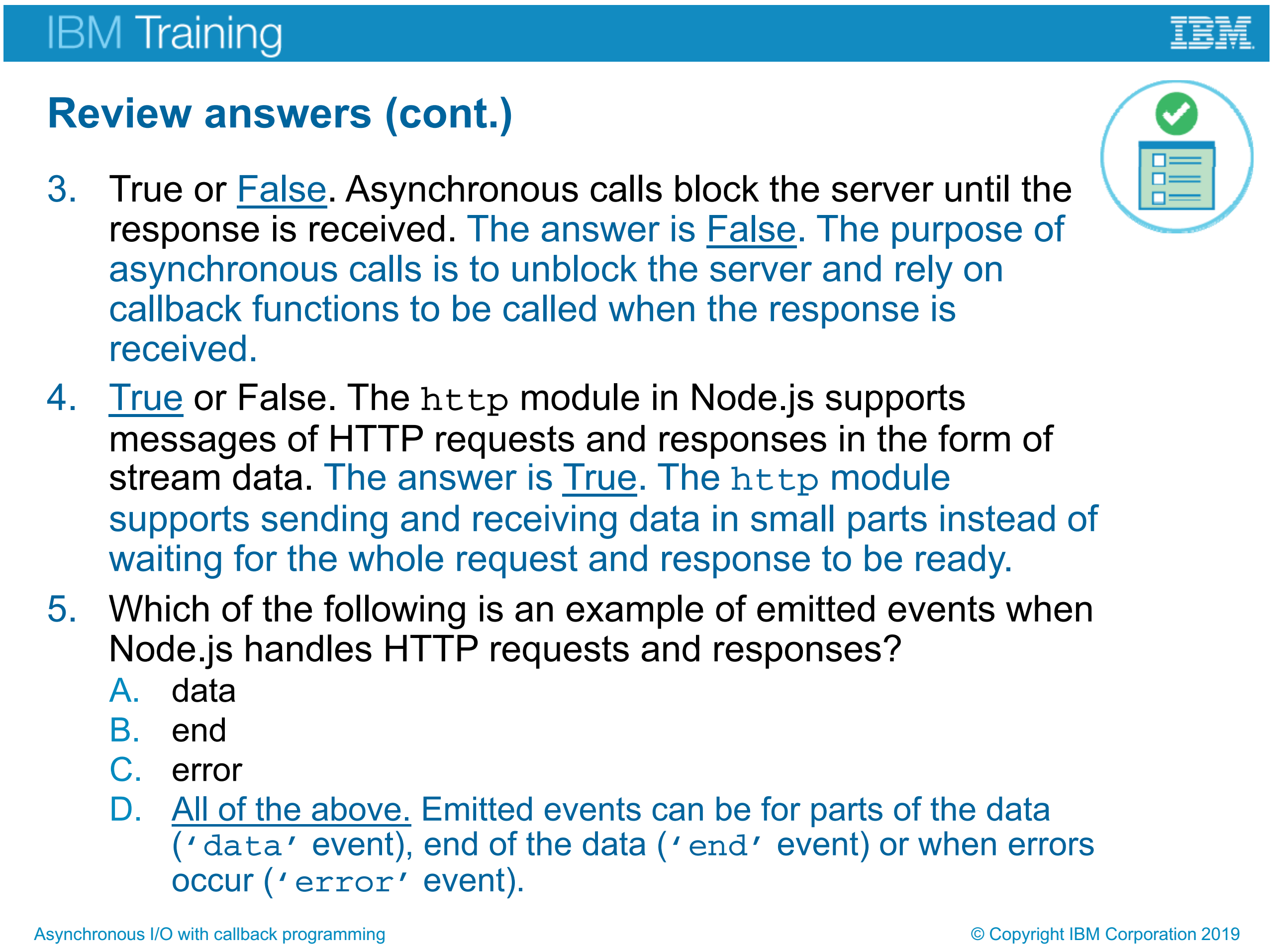
1. True or False. Node.js runs your callback function in the future after an event completes. The answer is True. The purpose of a callback function is to handle an event in the future.

2. Why does the `http.request` function need a callback function?
   - A. Network operations are asynchronous, and Node.js does not know when the remote server finishes sending the response message.
   - B. The Node framework runs your callback function after it receives a portion of the response message.
   - C. It is inefficient to block the running of a thread to wait for a response message.
   - D. All of the above. Node.js uses an event-driven programming model to handle efficiently asynchronous events, such as receiving parts of an HTTP response message from a server.

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-22. Review answers



IBM Training

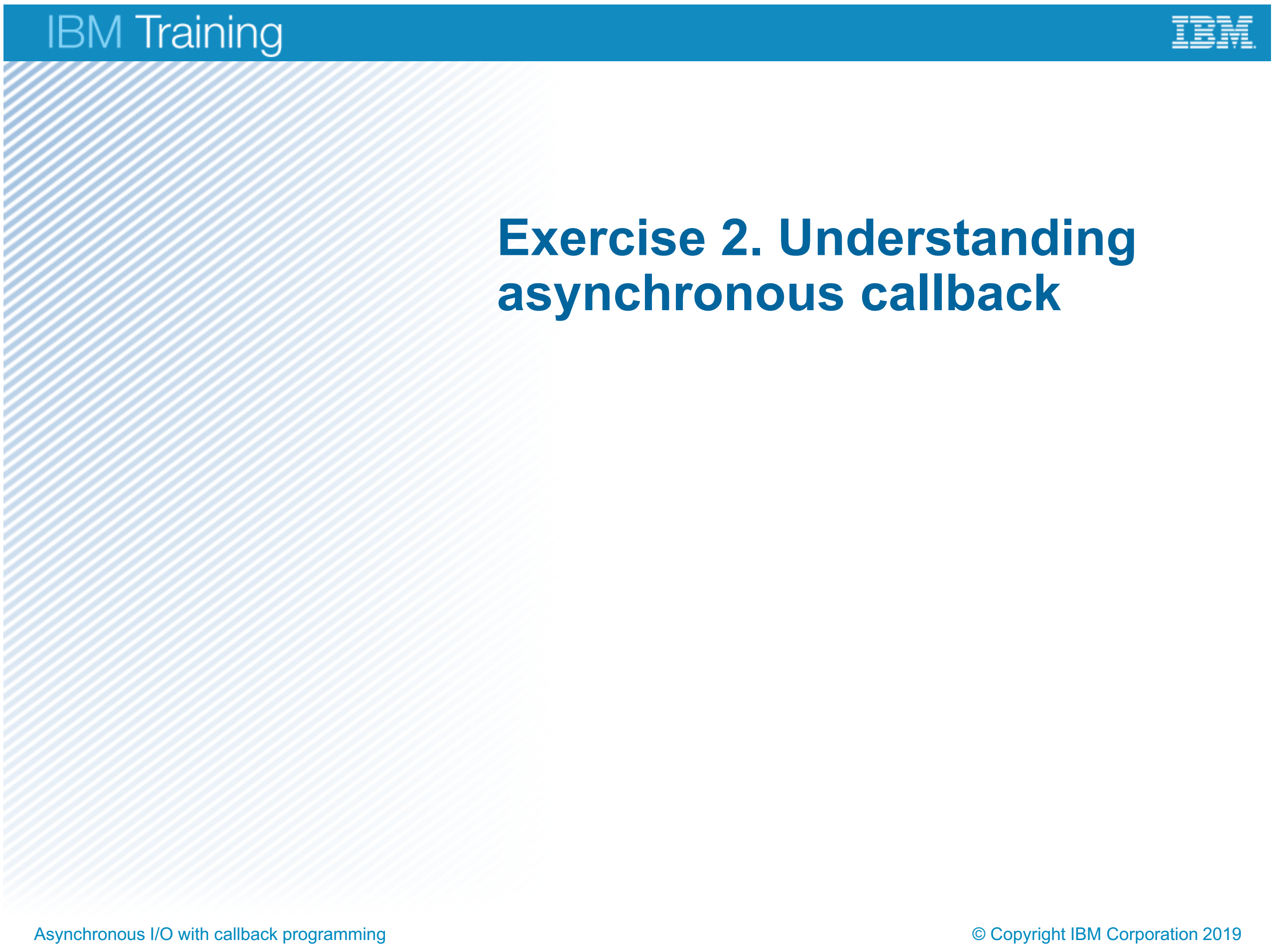
Review answers (cont.)

3. True or False. Asynchronous calls block the server until the response is received. The answer is False. The purpose of asynchronous calls is to unblock the server and rely on callback functions to be called when the response is received.
4. True or False. The `http` module in Node.js supports messages of HTTP requests and responses in the form of stream data. The answer is True. The `http` module supports sending and receiving data in small parts instead of waiting for the whole request and response to be ready.
5. Which of the following is an example of emitted events when Node.js handles HTTP requests and responses?
   A. data
   B. end
   C. error
   D. All of the above. Emitted events can be for parts of the data (`’data’` event), end of the data (`’end’` event) or when errors occur (`’error’` event).

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-23. Review answers (cont.)



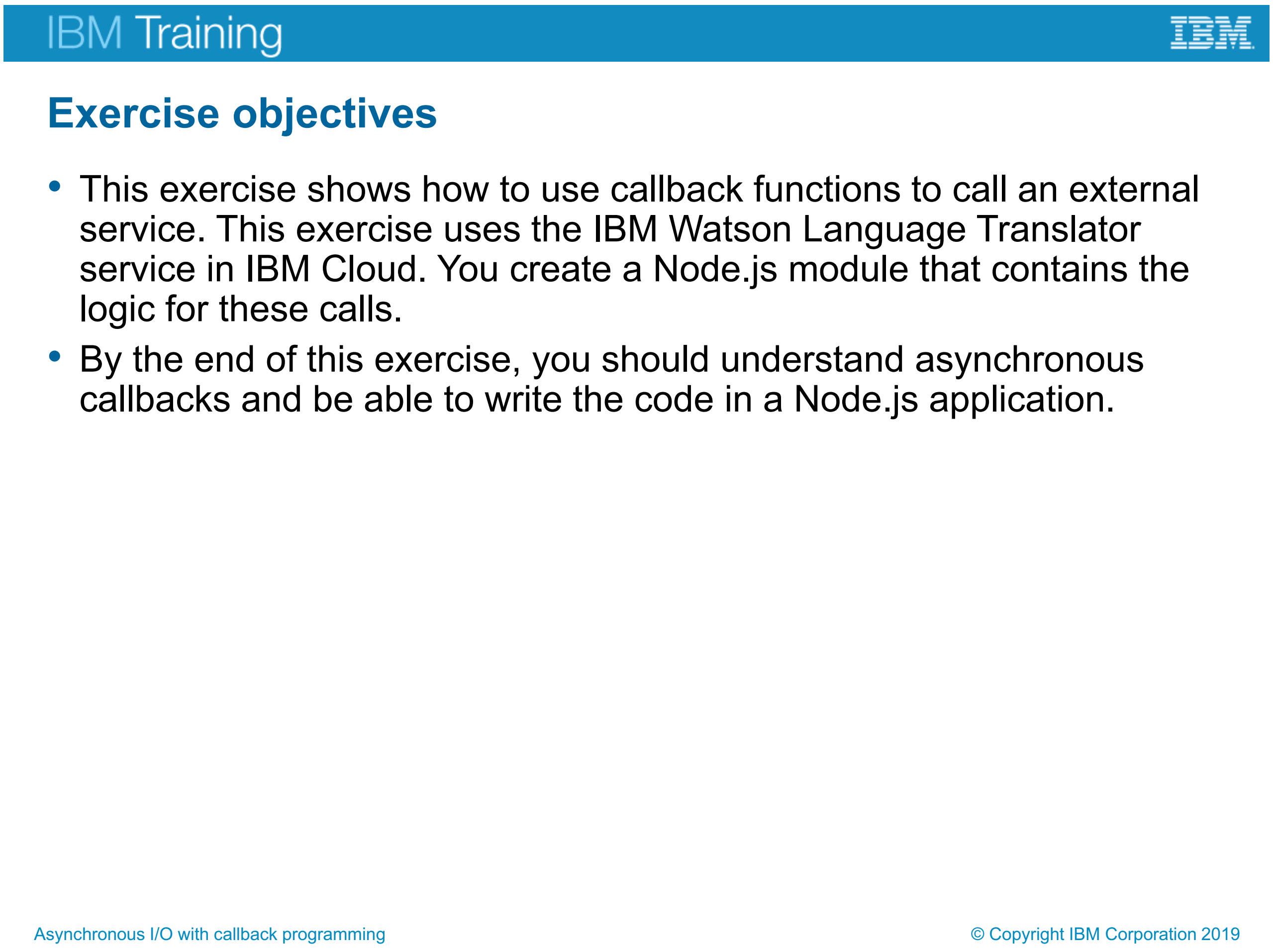
IBM Training

Exercise 2. Understanding asynchronous callback

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-24. Exercise 2. Understanding asynchronous callback



IBM Training

Exercise objectives

- This exercise shows how to use callback functions to call an external service. This exercise uses the IBM Watson Language Translator service in IBM Cloud. You create a Node.js module that contains the logic for these calls.
- By the end of this exercise, you should understand asynchronous callbacks and be able to write the code in a Node.js application.

Asynchronous I/O with callback programming

© Copyright IBM Corporation 2019

Figure 2-25. Exercise objectives

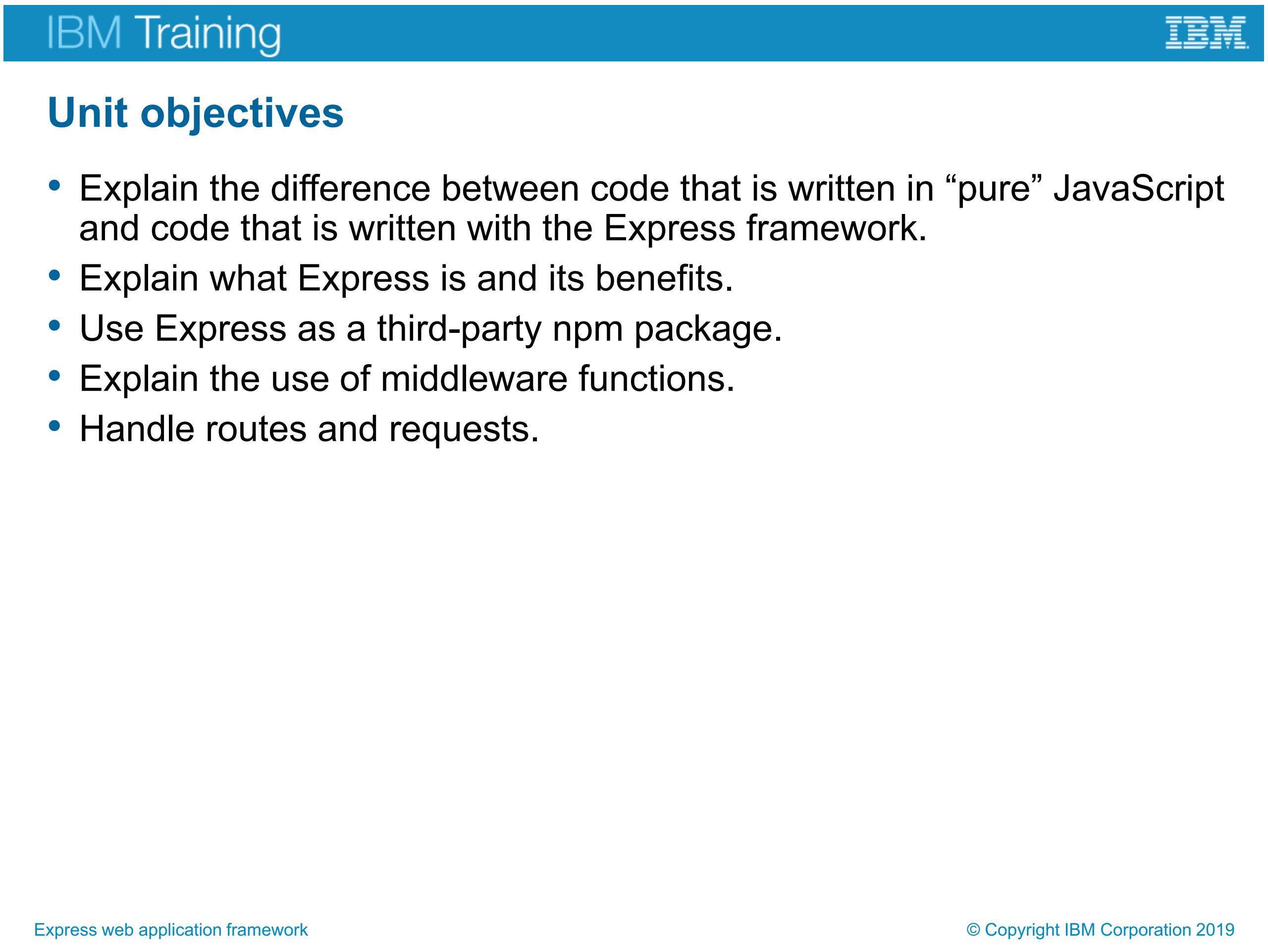
Unit 3. Express web application framework

Estimated time

01:30

Overview

This unit describes the Express web application framework, which provides a structured way to handle HTTP actions on server resources. You will learn how to write a REST service with Express and parse JSON data from an HTTP message.



IBM Training

Unit objectives

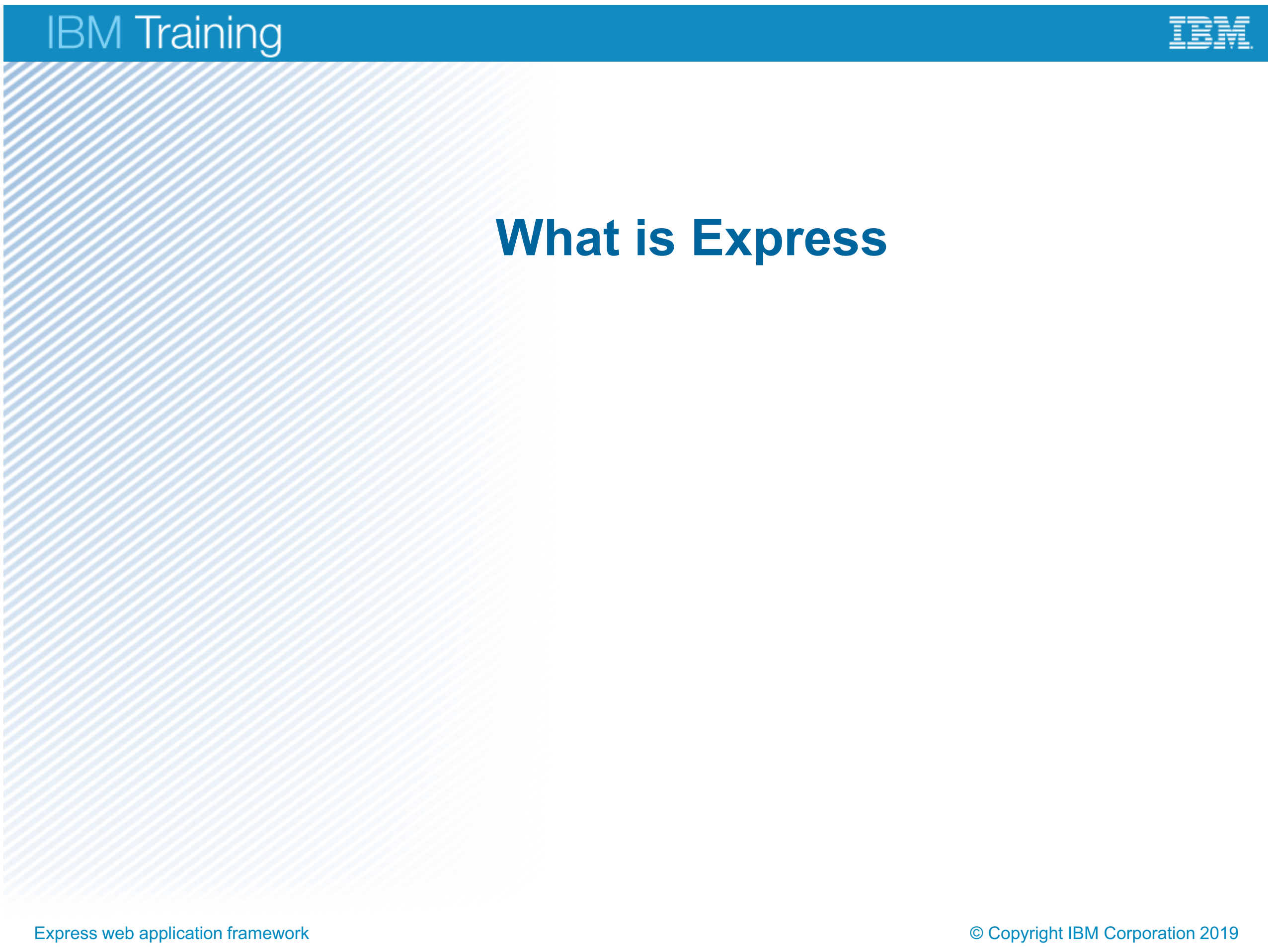
- Explain the difference between code that is written in “pure” JavaScript and code that is written with the Express framework.
- Explain what Express is and its benefits.
- Use Express as a third-party npm package.
- Explain the use of middleware functions.
- Handle routes and requests.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-1. Unit objectives

3.1. What is Express



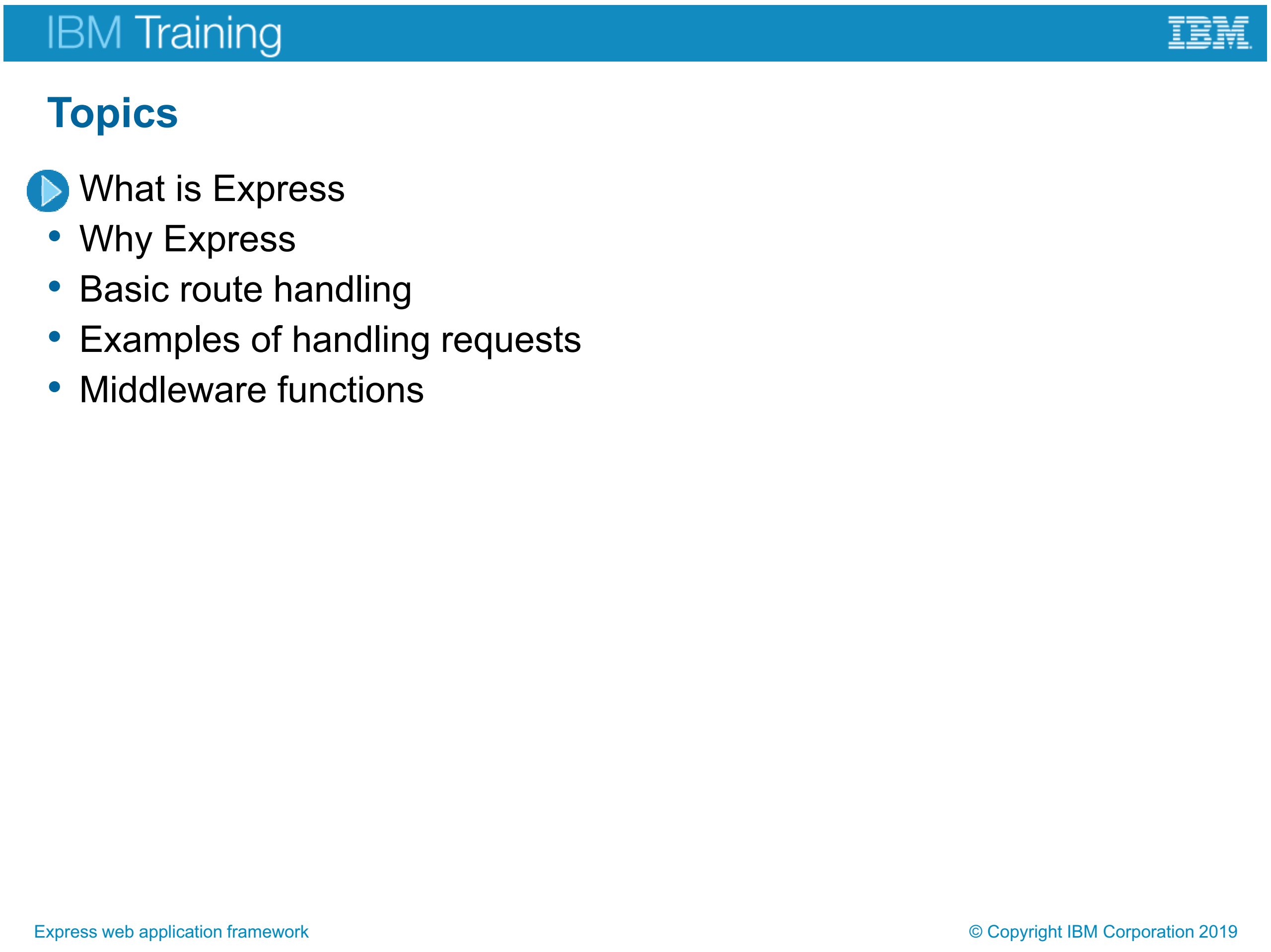
IBM Training

What is Express

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-2. What is Express



IBM Training

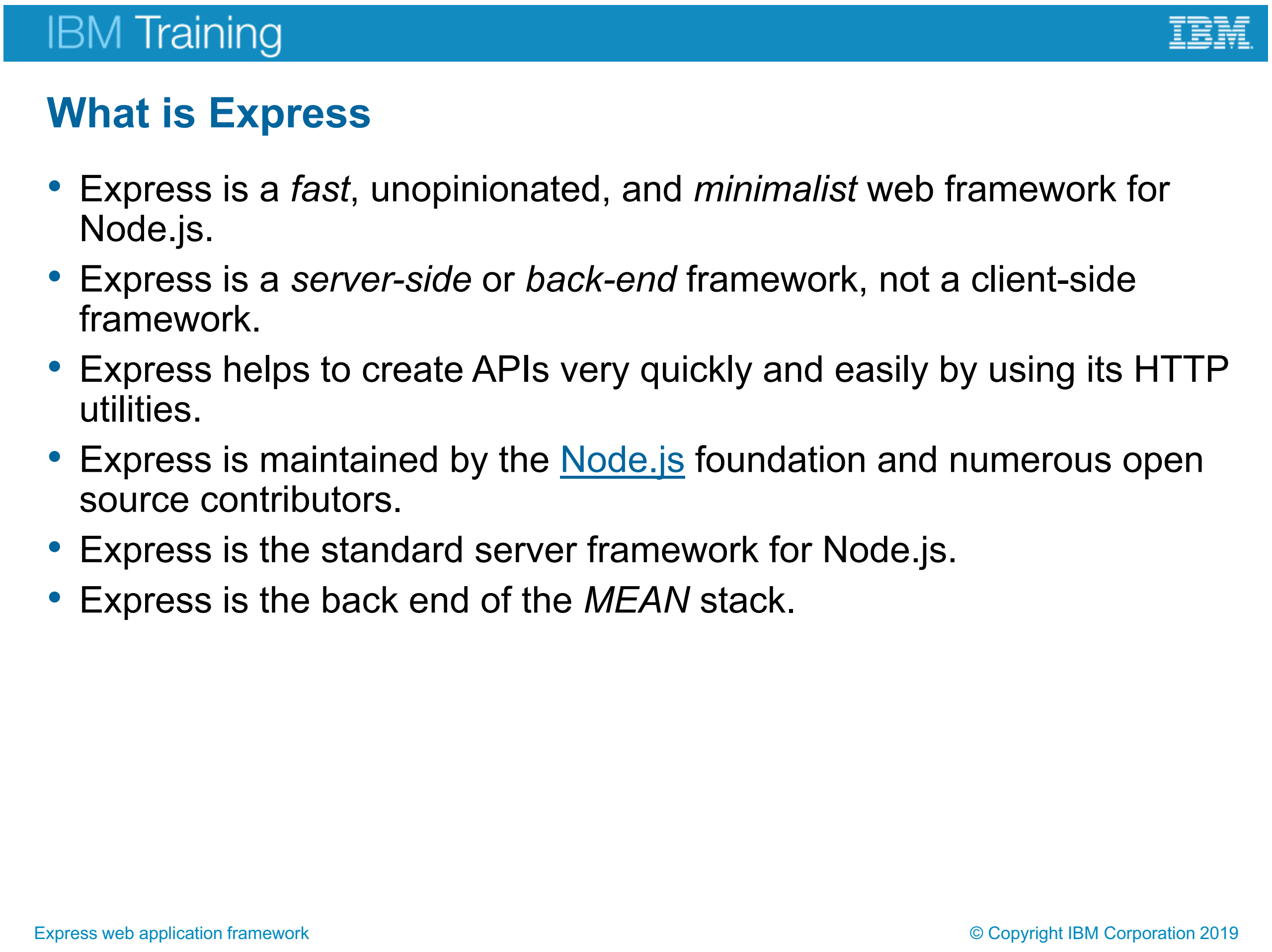
Topics

- What is Express
- Why Express
- Basic route handling
- Examples of handling requests
- Middleware functions

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-3. Topics


IBM Training

IBM

What is Express

- Express is a *fast*, unopinionated, and *minimalist* web framework for Node.js.
- Express is a *server-side* or *back-end* framework, not a client-side framework.
- Express helps to create APIs very quickly and easily by using its HTTP utilities.
- Express is maintained by the Node.js foundation and numerous open source contributors.
- Express is the standard server framework for Node.js.
- Express is the back end of the *MEAN* stack.

Express web application framework

© Copyright IBM Corporation 2019

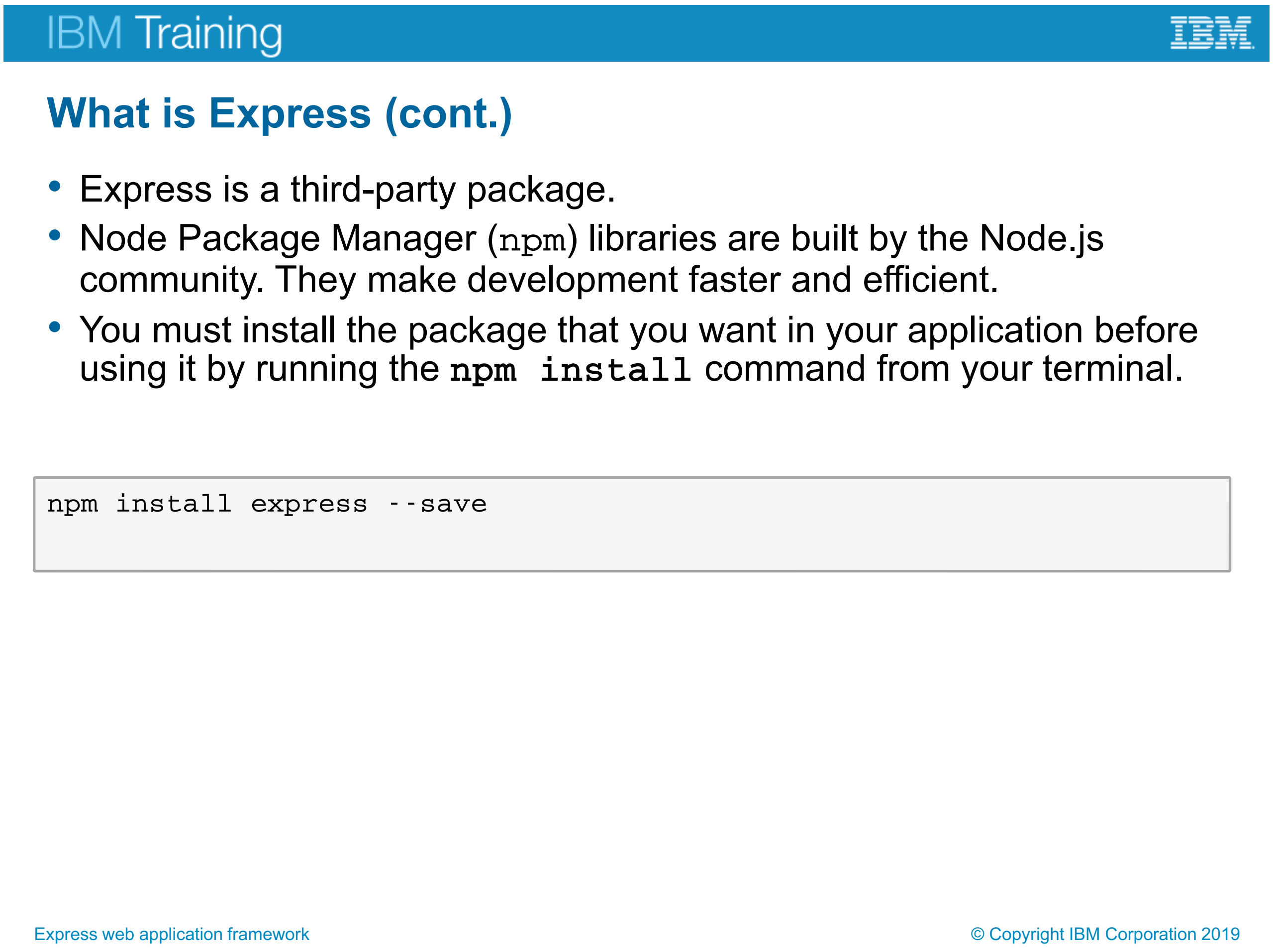
Figure 3-4. What is Express

- Express is a *fast*, unopinionated, and *minimalist* web framework for Node.js.
  - Fast: It is lightweight and highly performant.
  - Unopinionated: It is not a high-level framework. It does not assume that you build your app in a certain way or use a certain design pattern. You have full control of how you handle your requests and how the server responds.
  - Minimalist: Your job requires less code.
- Express is a *server-side* or *back-end* framework, not a client-side framework.
- Express is maintained by the Node.js foundation and numerous open source contributors.
- It is the standard server framework for Node.js. Express is the back end of the *MEAN* stack.

MEAN: MongoDB, Express.js, AngularJS, and Node.js.

Reference:

https://nodejs.org/en/



IBM Training

What is Express (cont.)

- Express is a third-party package.
- Node Package Manager (`npm`) libraries are built by the Node.js community. They make development faster and efficient.
- You must install the package that you want in your application before using it by running the **`npm install`** command from your terminal.

```
npm install express --save
```

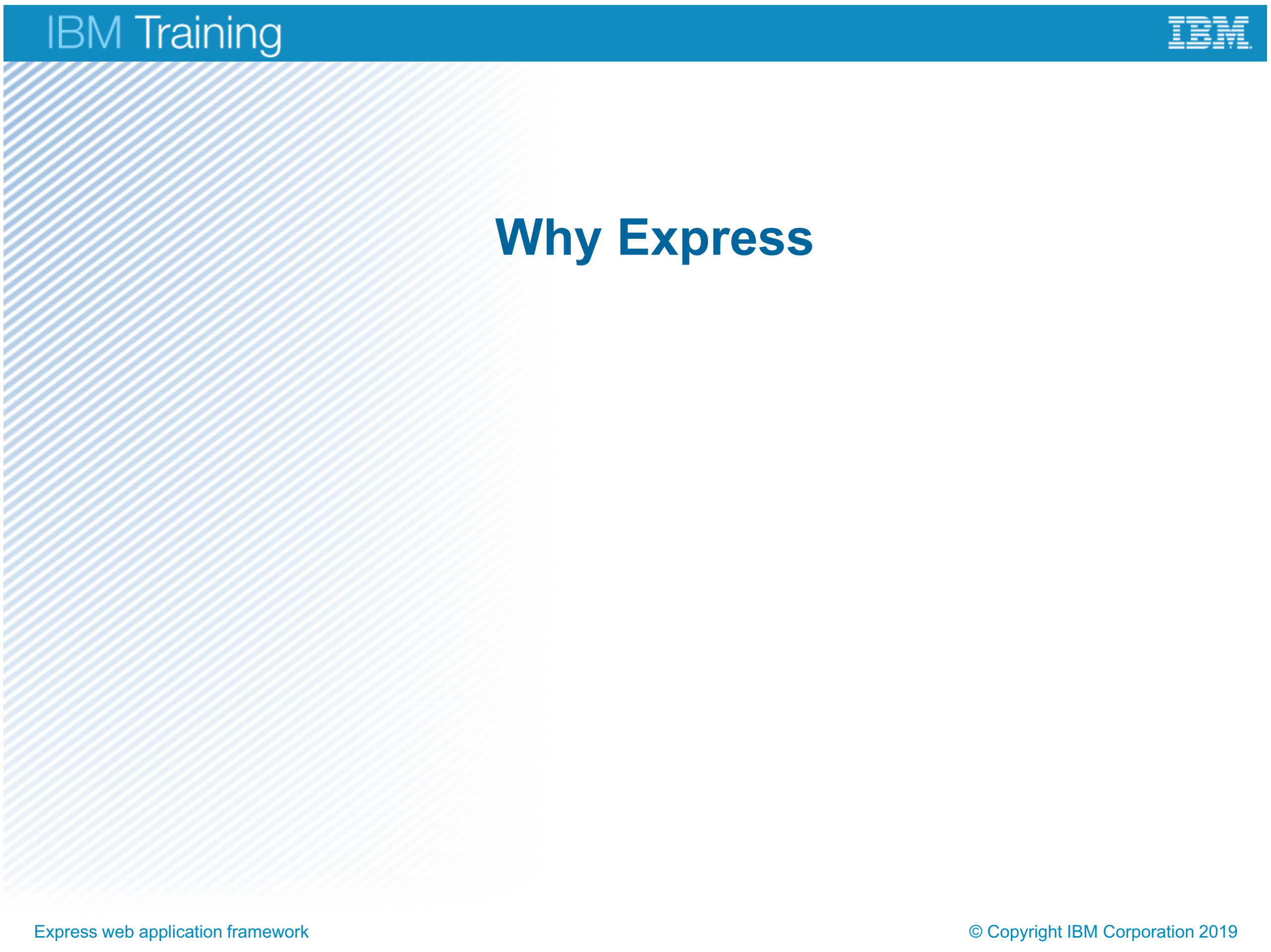
Express web application framework

© Copyright IBM Corporation 2019

Figure 3-5. What is Express (cont.)

- Express is a third-party package, which means that it should be installed before importing it into your app.
- There are over 200,000 npm libraries that were developed by the Node.js. community.
- To install an npm library, run the **npm install** command. You can set the version while installing the library or let the latest version be installed by default. Your terminal must be open in the app main directory where the package.json file is.

3.2. Why Express



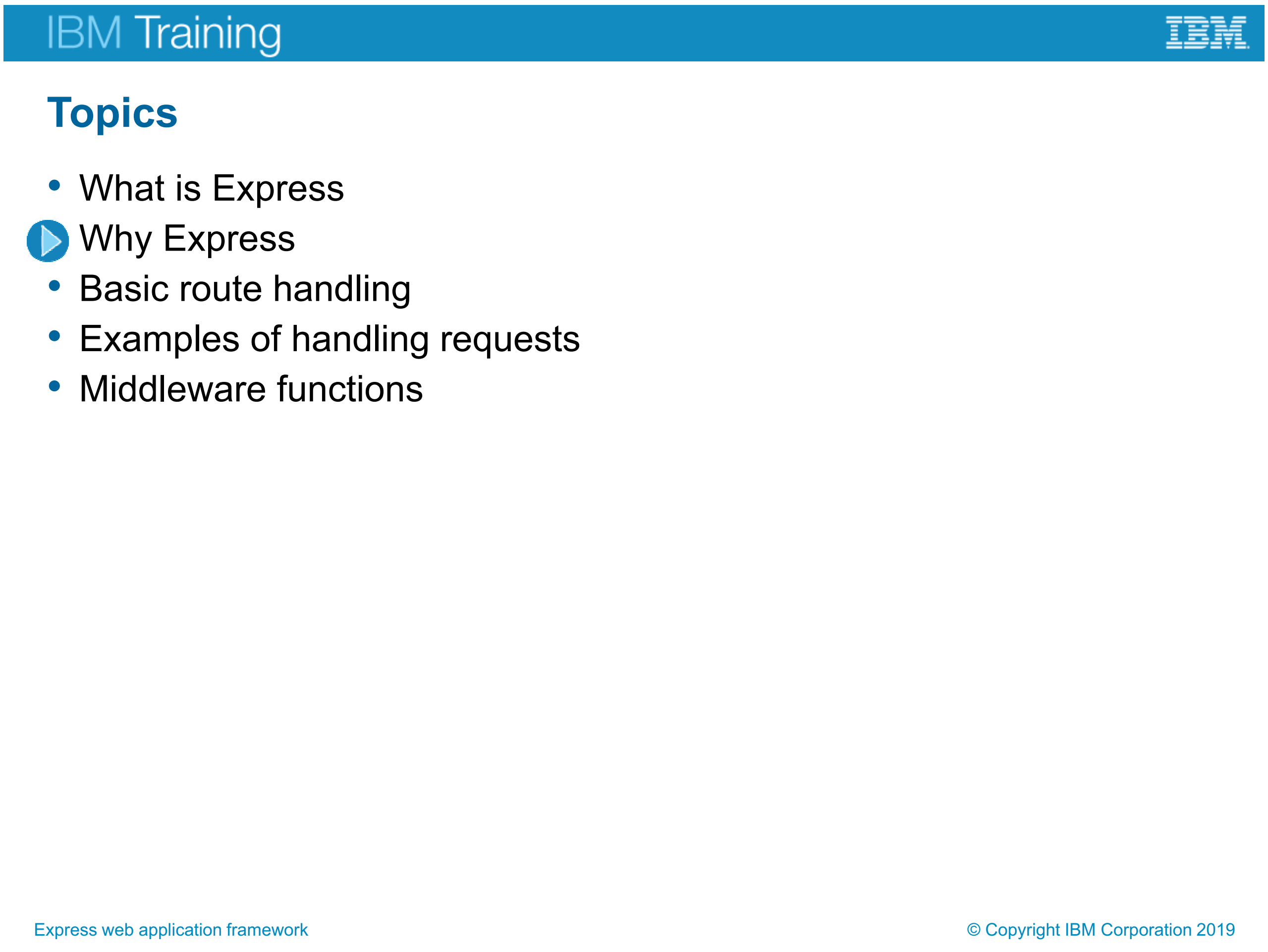
IBM Training

Why Express

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-6. Why Express



IBM Training

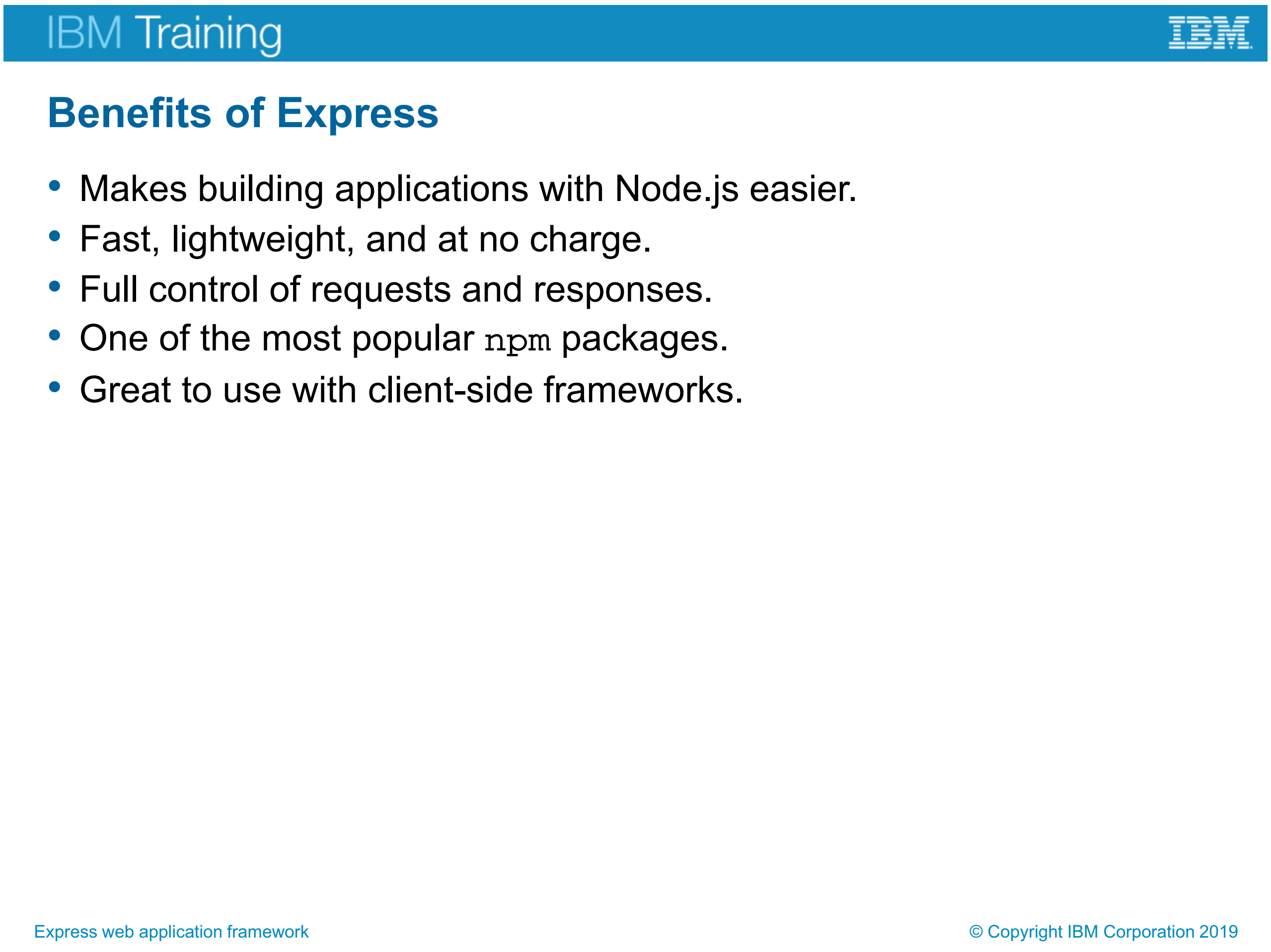
Topics

- What is Express
- ▶ Why Express
- Basic route handling
- Examples of handling requests
- Middleware functions

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-7. Topics



IBM Training

Benefits of Express

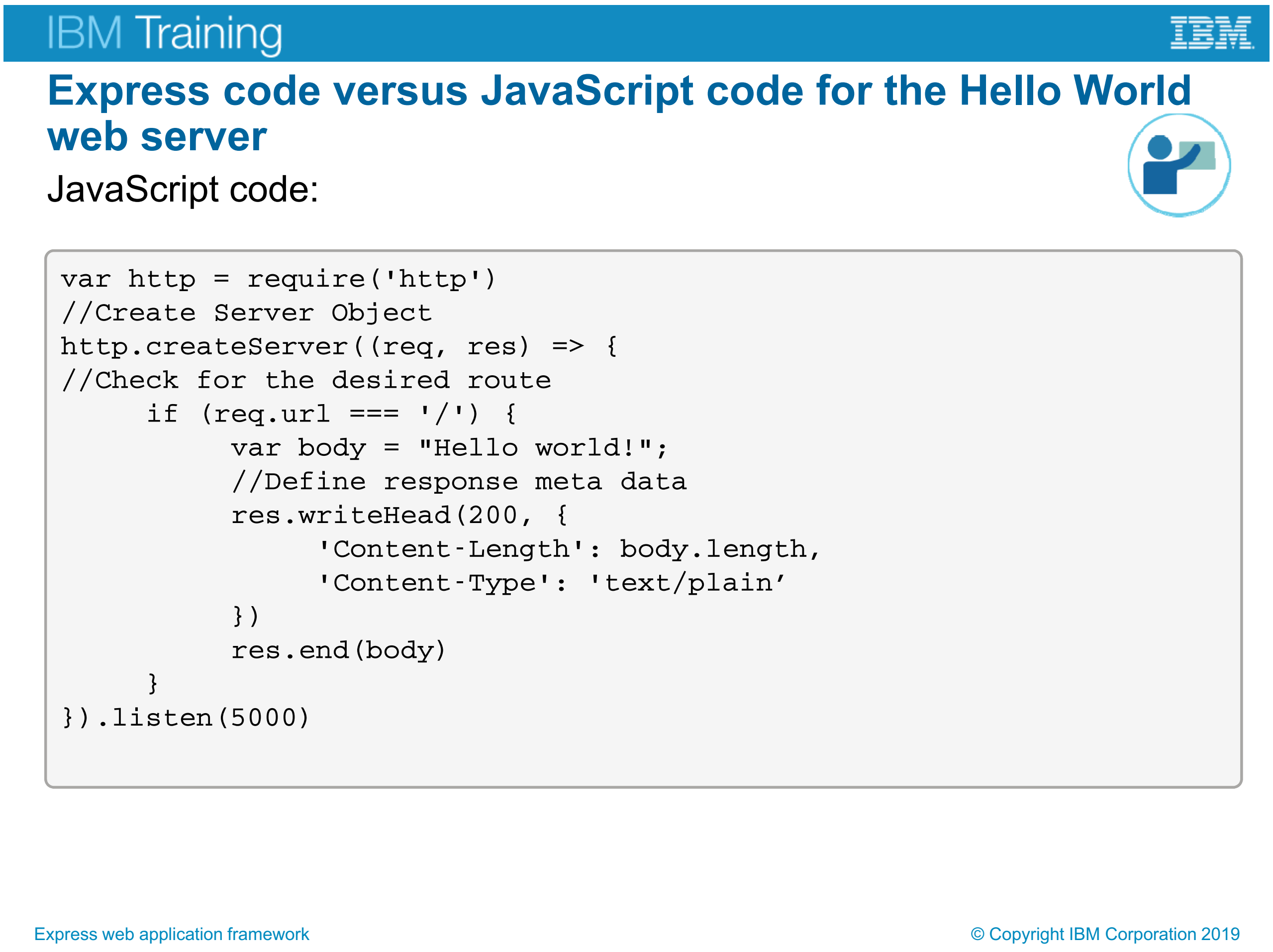
- Makes building applications with Node.js easier.
- Fast, lightweight, and at no charge.
- Full control of requests and responses.
- One of the most popular `npm` packages.
- Great to use with client-side frameworks.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-8. Benefits of Express

- Makes building applications with Node.js easier compared to building them with only JavaScript code.
- Fast, lightweight, flexible, and at no charge compared to another Node.js frameworks.
- Gives you full control of how you handle requests and responses.
- The most popular package on the npm website.
- Great to use with any JavaScript client-side framework because you use JavaScript in both the back and front ends.



IBM Training

Express code versus JavaScript code for the Hello World web server

JavaScript code:

```
var http = require('http')
//Create Server Object
http.createServer((req, res) => {
//Check for the desired route
     if (req.url === '/') {
          var body = "Hello world!";
          //Define response meta data
          res.writeHead(200, {
               'Content-Length': body.length,
               'Content-Type': 'text/plain’
          })
          res.end(body)
     }
}).listen(5000)
```

Express web application framework

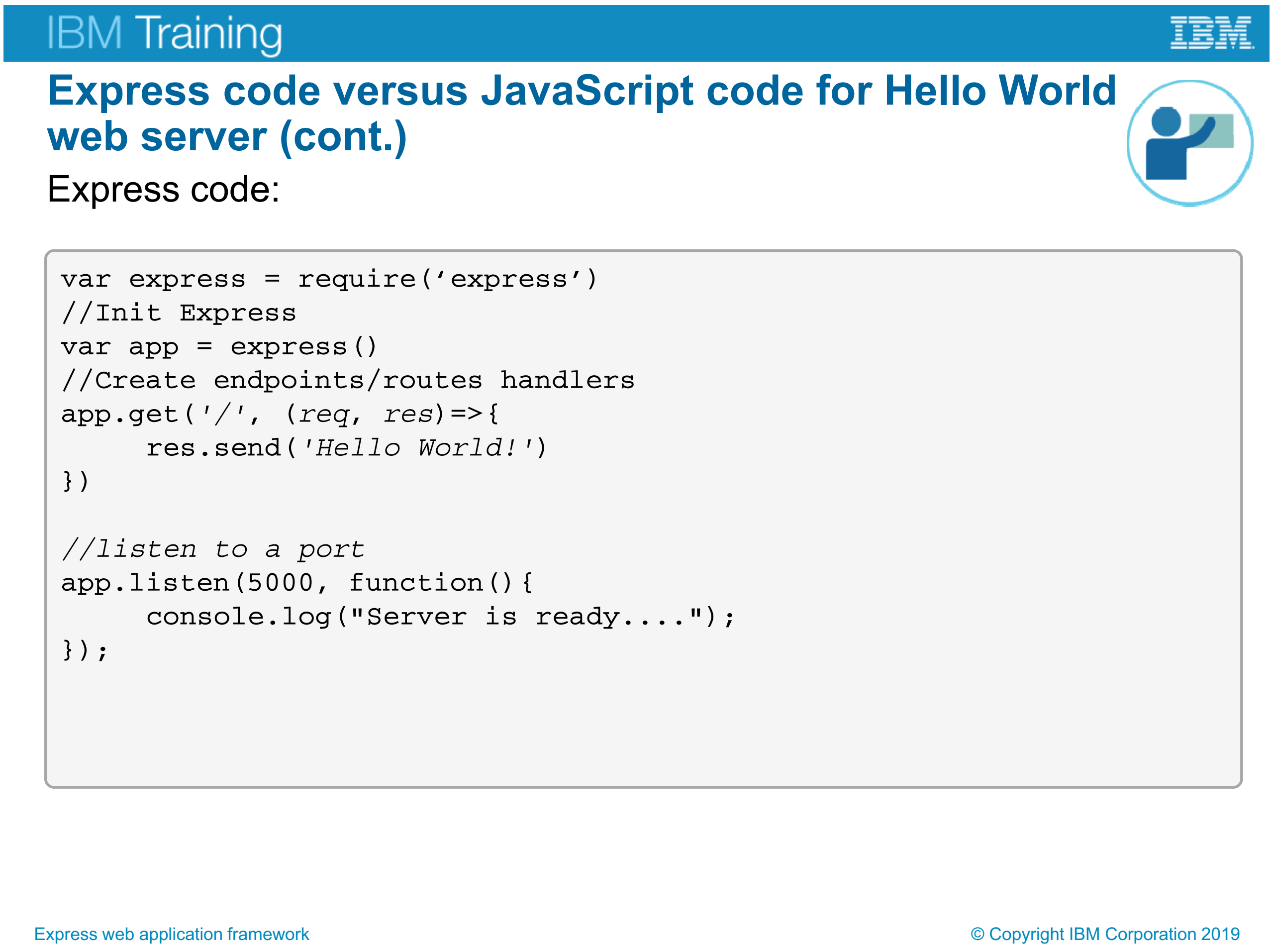
© Copyright IBM Corporation 2019

Figure 3-9. Express code versus JavaScript code for the Hello Worldweb server

To create a server by using JavaScript code:

1. Import the http module.
2. Create the server object.
3. Check whether the URL of the request is ‘/’.
4. Return the ‘Hello World!’ text and set the server to listen on port 5000.

In the next slide, create the same web server by using Express.



IBM Training

Express code versus JavaScript code for Hello World web server (cont.)

Express code:

```
var express = require(‘express’)
//Init Express
var app = express()
//Create endpoints/routes handlers
app.get('/', (req, res)=>{
     res.send('Hello World!')
})

//listen to a port
app.listen(5000, function(){
     console.log("Server is ready....");
});
```

Express web application framework © Copyright IBM Corporation 2019

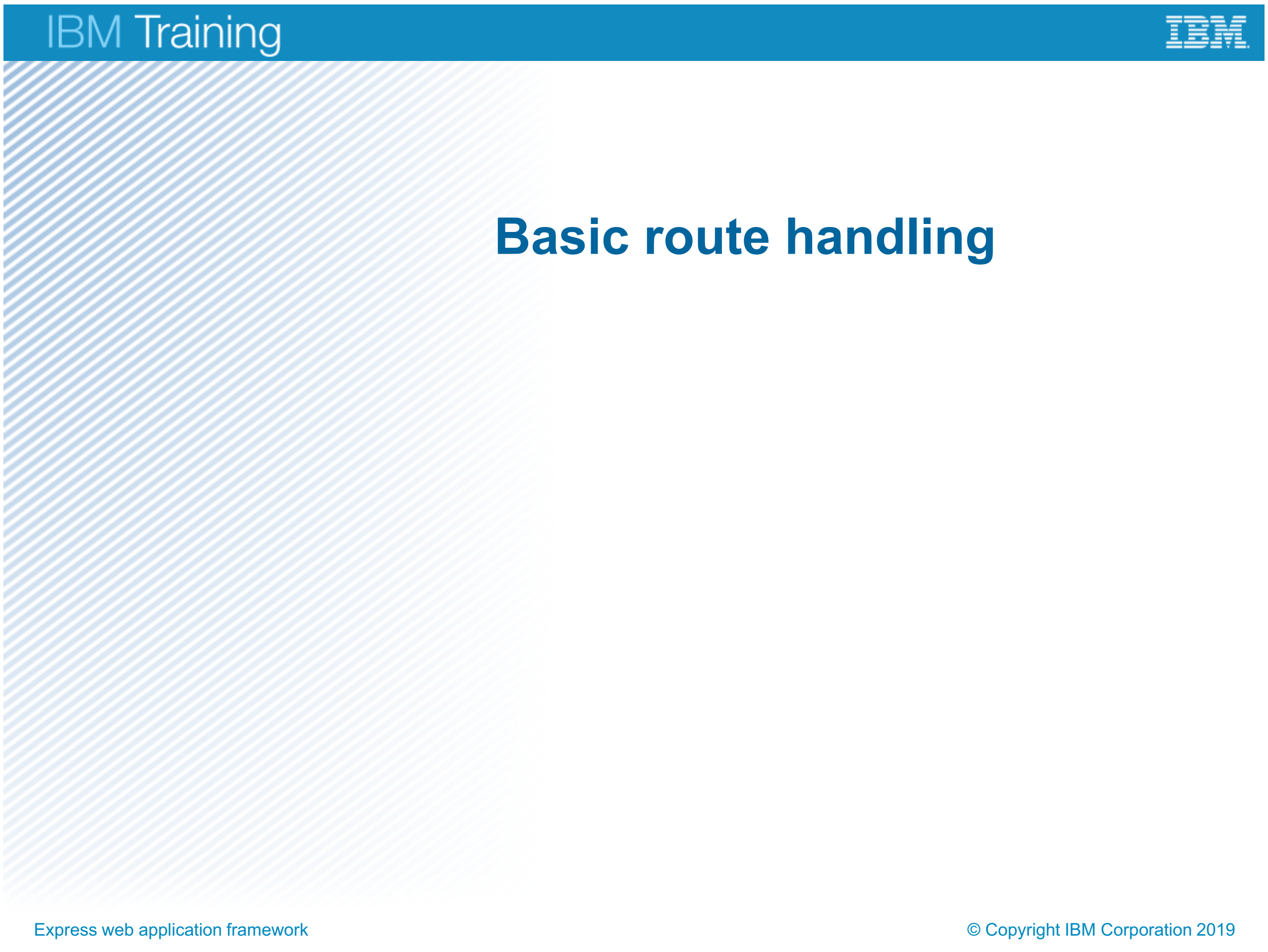
Figure 3-10. Express code versus JavaScript code for Hello Worldweb server (cont.)

To create a server by using Express:

1. Import the Express module.
2. Create an instance for the server (app).
3. Listen for the ‘/ ’ route to return ‘Hello World!’ as a response.
4. Listen on port 5000.

As you can see, creating the server by using Express is much easier.

3.3. Basic route handling



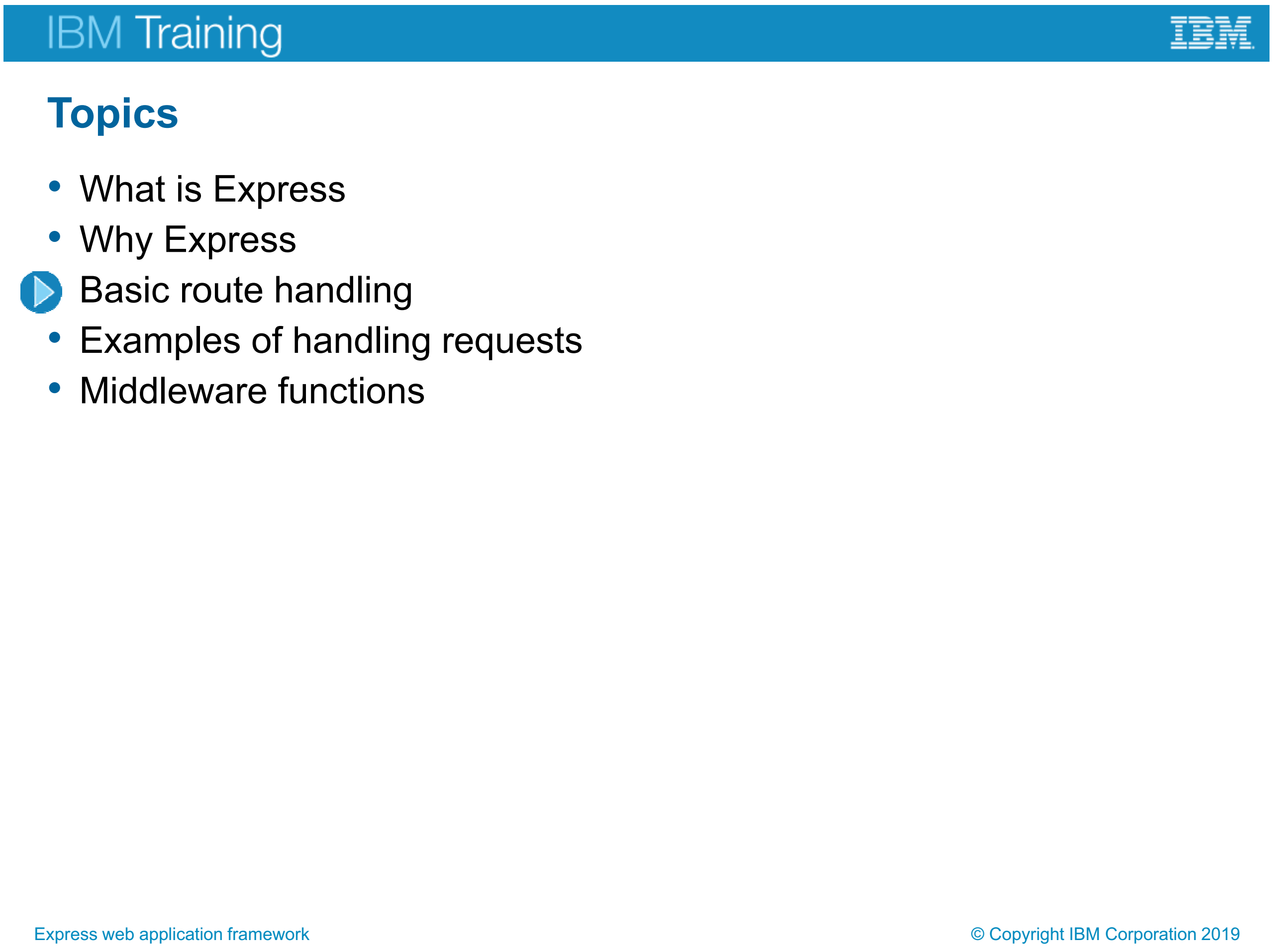
IBM Training

Basic route handling

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-11. Basic route handling



IBM Training

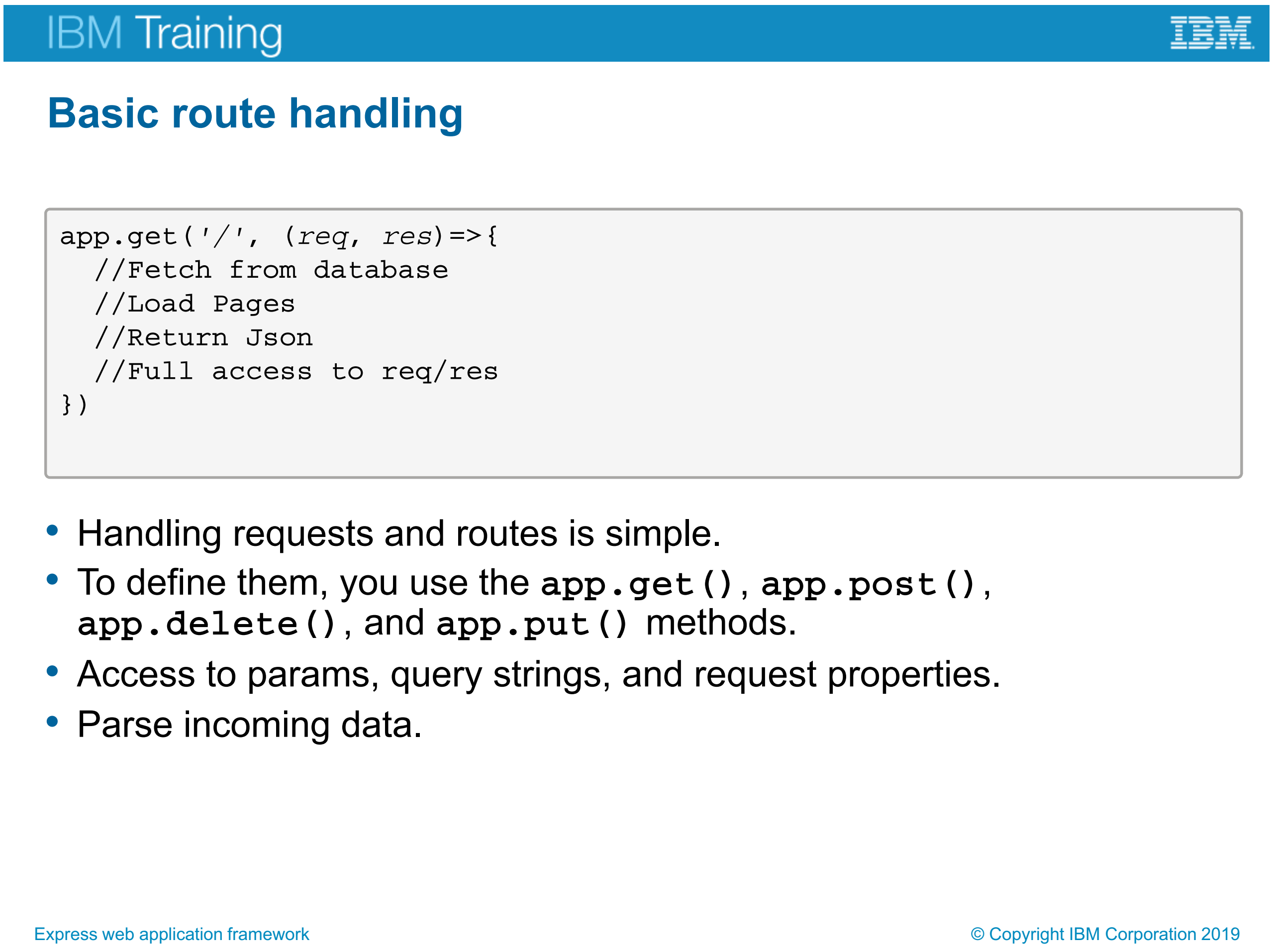
Topics

- What is Express
- Why Express
- Basic route handling
- Examples of handling requests
- Middleware functions

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-12. Topics



IBM Training

Basic route handling

```
app.get('/', (req, res)=>{
  //Fetch from database
  //Load Pages
  //Return Json
  //Full access to req/res
})
```

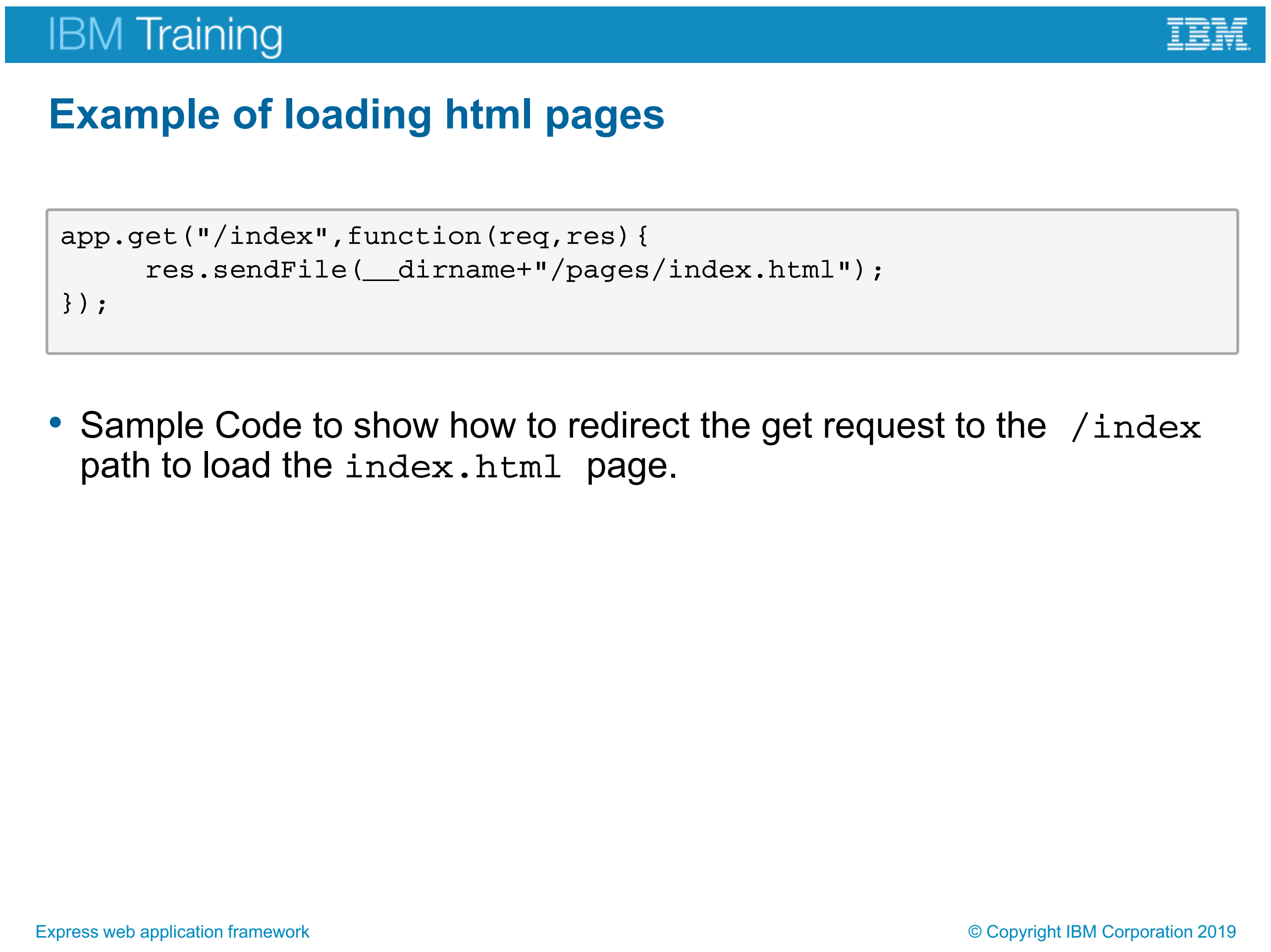
- Handling requests and routes is simple.
- To define them, you use the `app.get()`, `app.post()`, `app.delete()`, and `app.put()` methods.
- Access to params, query strings, and request properties.
- Parse incoming data.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-13. Basic route handling

- Within your route, you can fetch data from a database such as IBM Cloudant, load static pages, and load data in JSON, XML, or another format.
- The requests and routes contain important information like parameters, query strings, and any data that is sent in the body.
- You use different http methods to define your routes like **app.get()**, **app.post()**, **app.delete()**, and **app.put()**.
- The different http methods are **get** for read, **post** for create, **put** for update, and **delete** for delete.



IBM Training

Example of loading html pages

```
app.get("/index",function(req,res){
      res.sendFile(__dirname+"/pages/index.html");
});
```

- Sample Code to show how to redirect the get request to the `/index` path to load the `index.html` page.

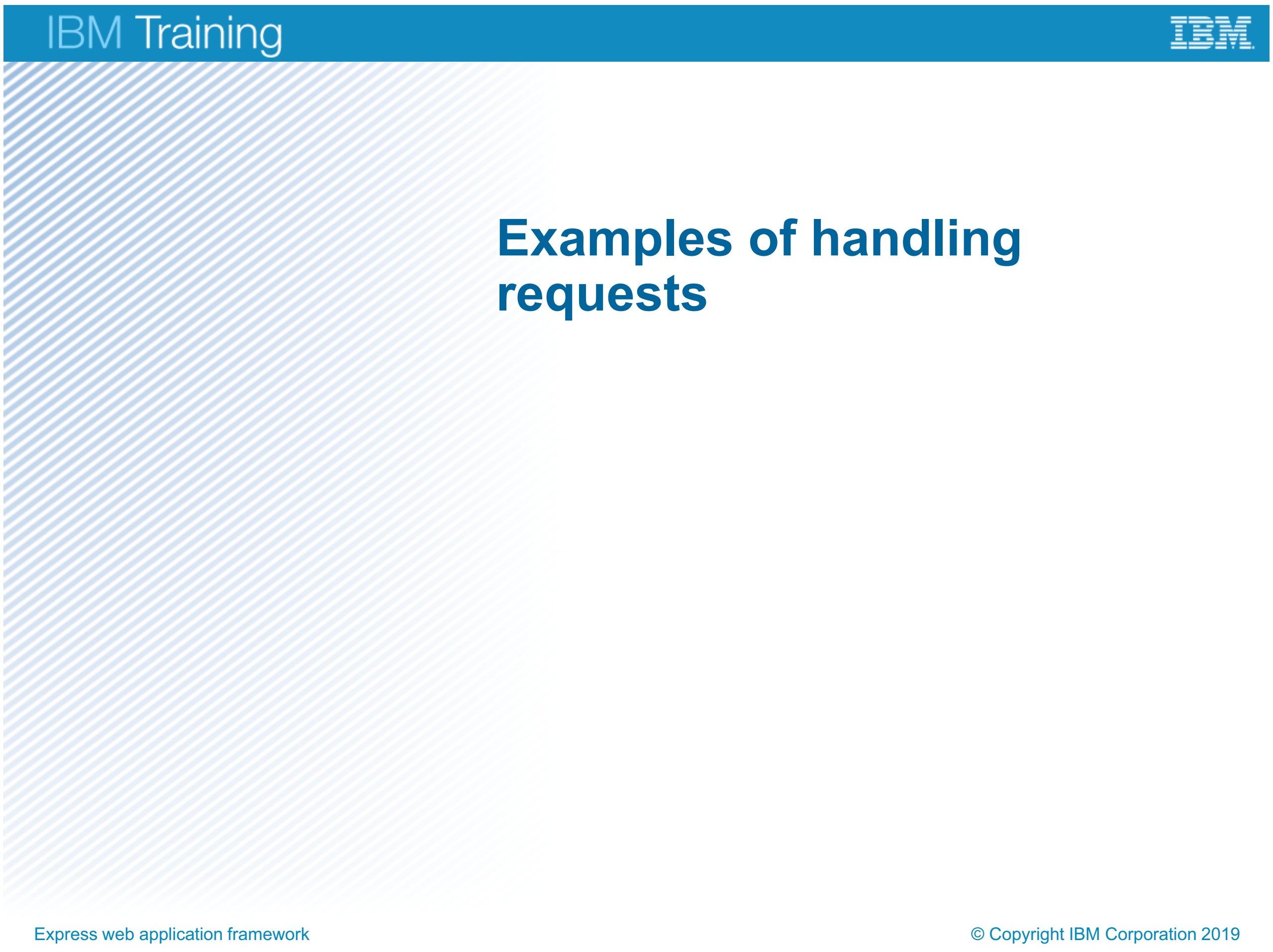
Express web application framework

© Copyright IBM Corporation 2019

Figure 3-14. Example of loading html pages

- Within your route, you can fetch data from a database such as IBM Cloudant, load static pages, and load data in JSON, XML, or another format.
- The sample code shows how to redirect the get request to the `/index` path to load the `index.html` page.

3.4. Examples of handling requests



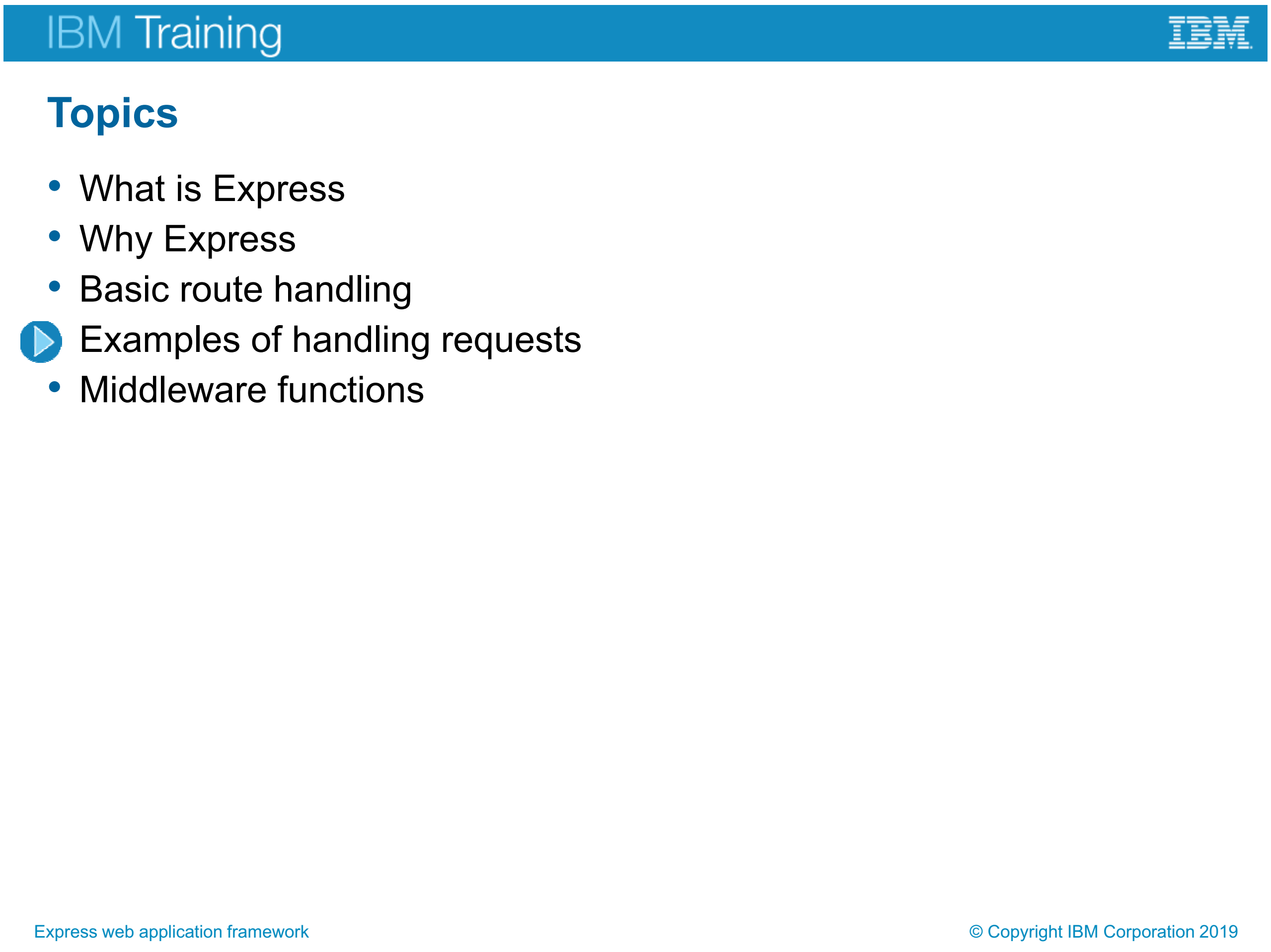
IBM Training

Examples of handling requests

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-15. Examples of handling requests



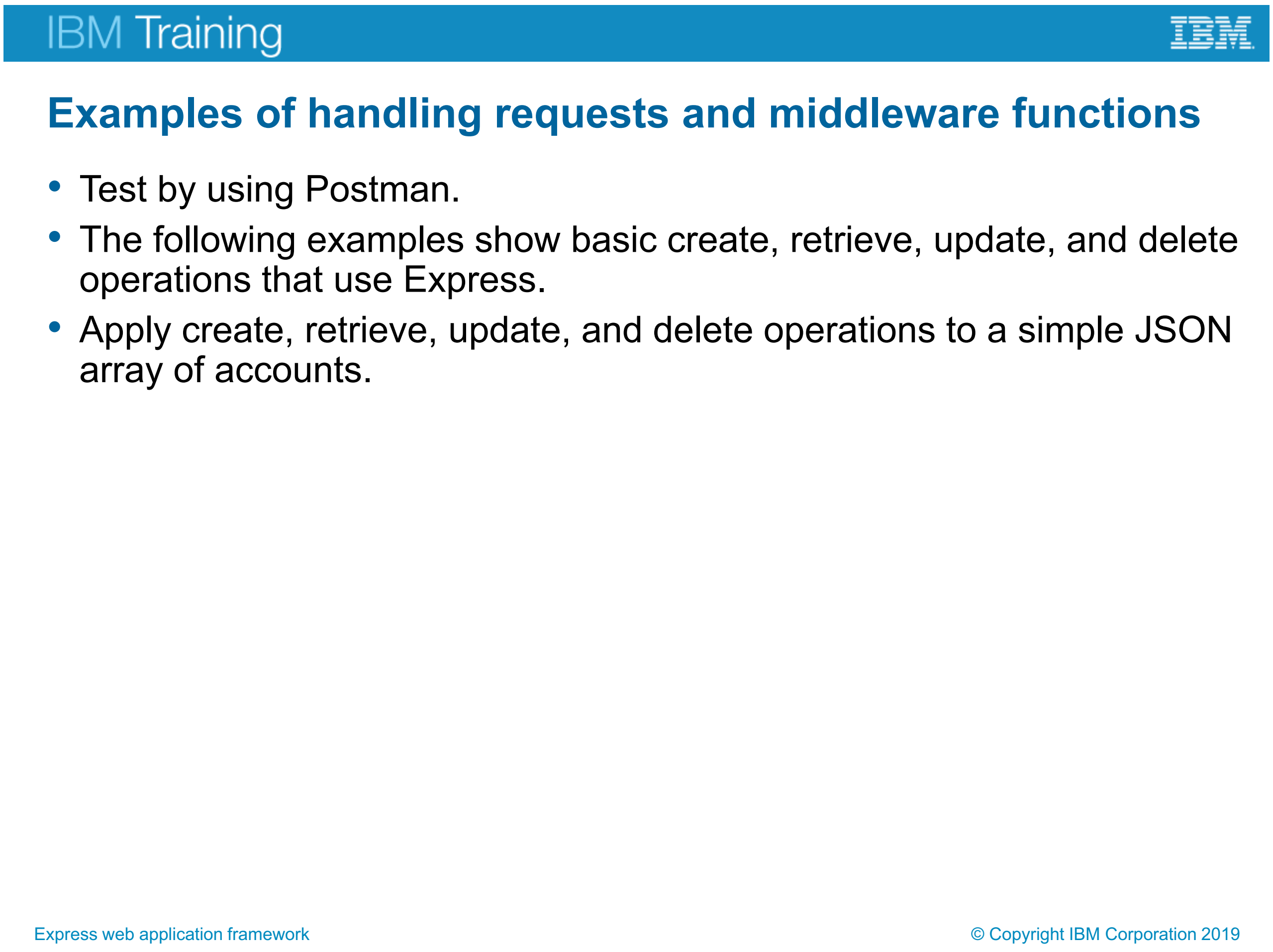
IBM Training

Topics

- What is Express
- Why Express
- Basic route handling
- ▶ Examples of handling requests
- Middleware functions

Express web application framework © Copyright IBM Corporation 2019

Figure 3-16. Topics



IBM Training

Examples of handling requests and middleware functions

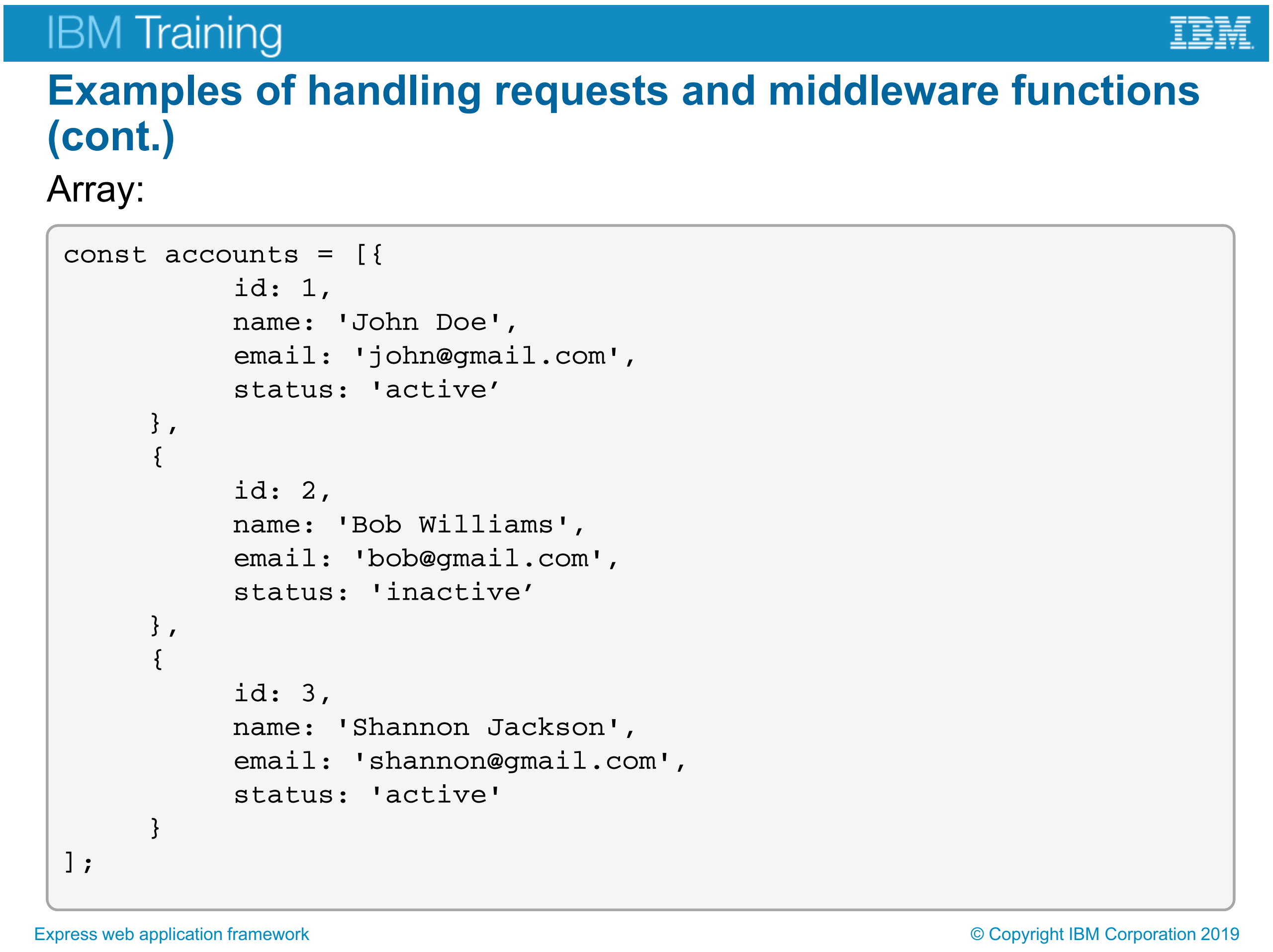
- Test by using Postman.
- The following examples show basic create, retrieve, update, and delete operations that use Express.
- Apply create, retrieve, update, and delete operations to a simple JSON array of accounts.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-17. Examples of handling requests and middleware functions

- Use Postman to test REST APIs. With Postman, you can install and start simulated REST requests and see the responses for those requests. It has many other handy utilities.
- The following slides show how you can make simple requests to do create, retrieve, update, and delete operations on a simple JSON array by using REST APIs.



IBM Training

Examples of handling requests and middleware functions (cont.)

Array:

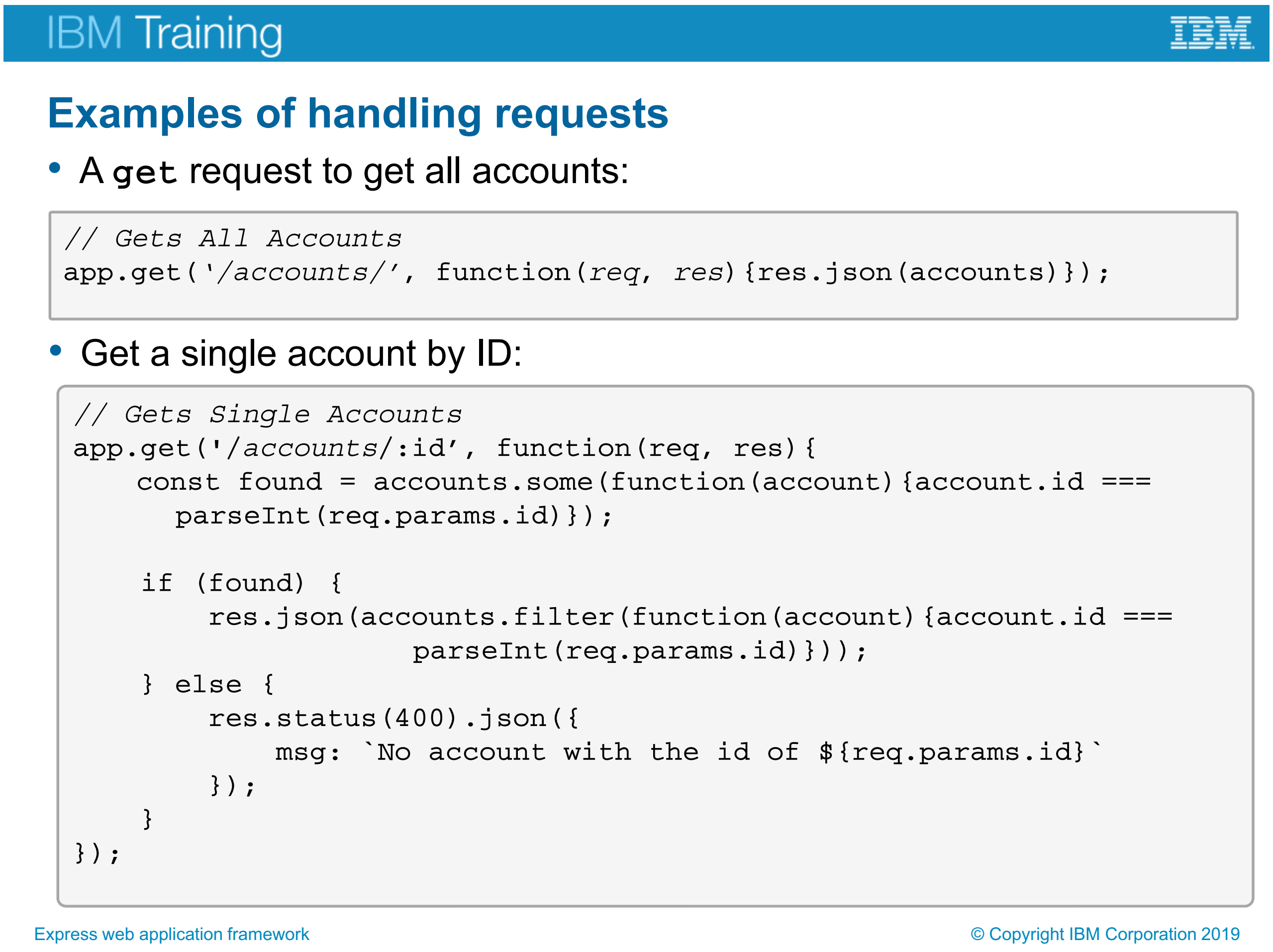
```
const accounts = [{
          id: 1,
          name: 'John Doe',
          email: 'john@gmail.com',
          status: 'active’
     },
     {
          id: 2,
          name: 'Bob Williams',
          email: 'bob@gmail.com',
          status: 'inactive’
     },
     {
          id: 3,
          name: 'Shannon Jackson',
          email: 'shannon@gmail.com',
          status: 'active'
     }
];
```

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-18. Examples of handling requests and middleware functions (cont.)

Initial JSON array of accounts.



IBM Training

Examples of handling requests

- A `get` request to get all accounts:

```
// Gets All Accounts
app.get(‘/accounts/’, function(req, res){res.json(accounts)});
```

- Get a single account by ID:

```
// Gets Single Accounts
app.get('/accounts/:id’, function(req, res){
    const found = accounts.some(function(account){account.id ===
      parseInt(req.params.id)});

    if (found) {
        res.json(accounts.filter(function(account){account.id ===
                    parseInt(req.params.id)}));
    } else {
        res.status(400).json({
            msg: `No account with the id of ${req.params.id}`
        });
    }
});
```

Express web application framework © Copyright IBM Corporation 2019

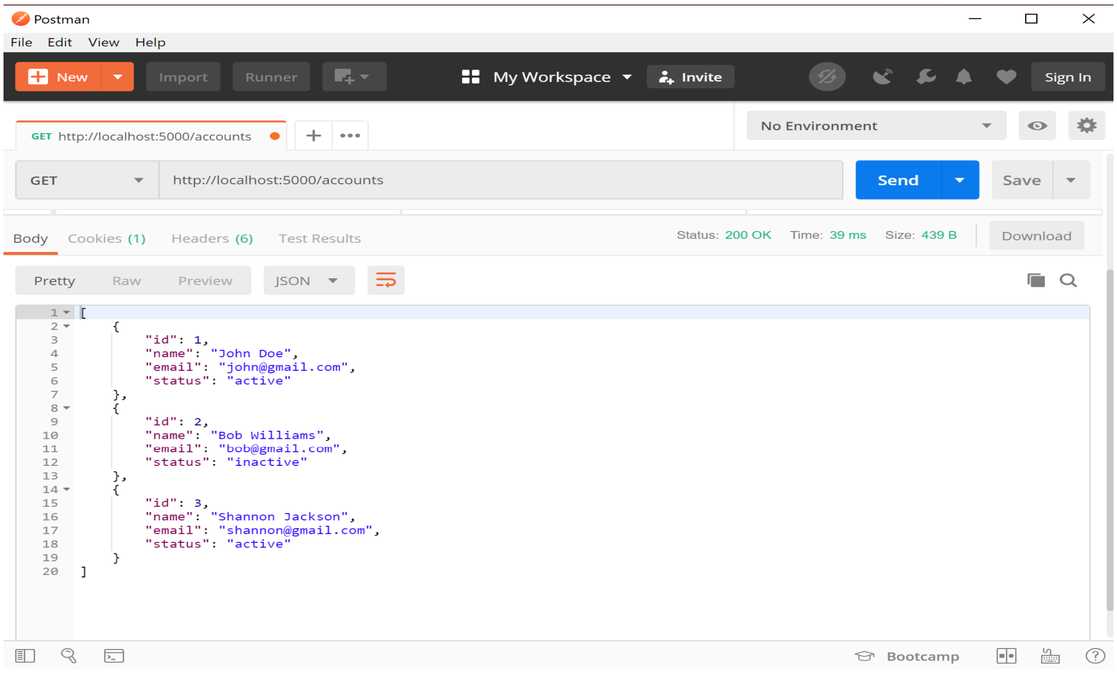
Figure 3-19. Examples of handling requests

- The first route gets the accounts array and returns the JSON array from the previous slide.
- The second route gets the account according to the ID, which is how you pass an ID as a parameter and then checks whether the ID that is provided exists. If the ID exists, the route returns the matched account. If it does not exist, the route returns a failure message to the user.

IBM Training

Testing with Postman

Get all accounts:



GET http://localhost:5000/accounts

Status: 200 OK Time: 39 ms Size: 439 B

```
[
    {
        "id": 1,
        "name": "John Doe",
        "email": "john@gmail.com",
        "status": "active"
    },
    {
        "id": 2,
        "name": "Bob Williams",
        "email": "bob@gmail.com",
        "status": "inactive"
    },
    {
        "id": 3,
        "name": "Shannon Jackson",
        "email": "shannon@gmail.com",
        "status": "active"
    }
]
```

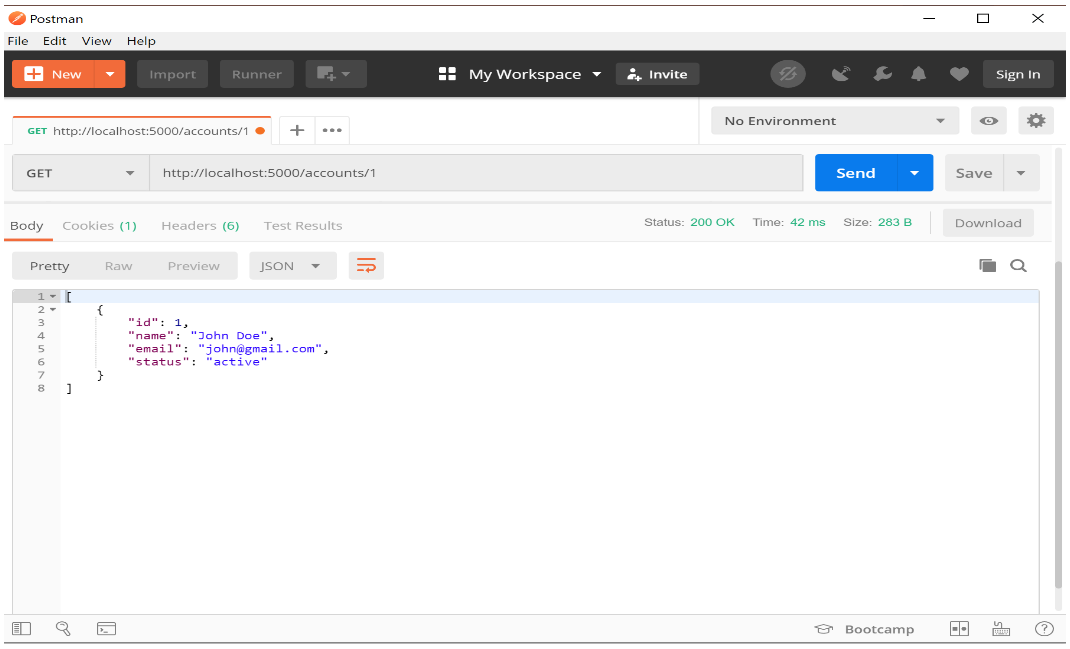
Figure 3-20. Testing with Postman

A sample request of Postman getting all the accounts.

IBM Training

Testing with Postman (cont.)

Get a single account by ID:

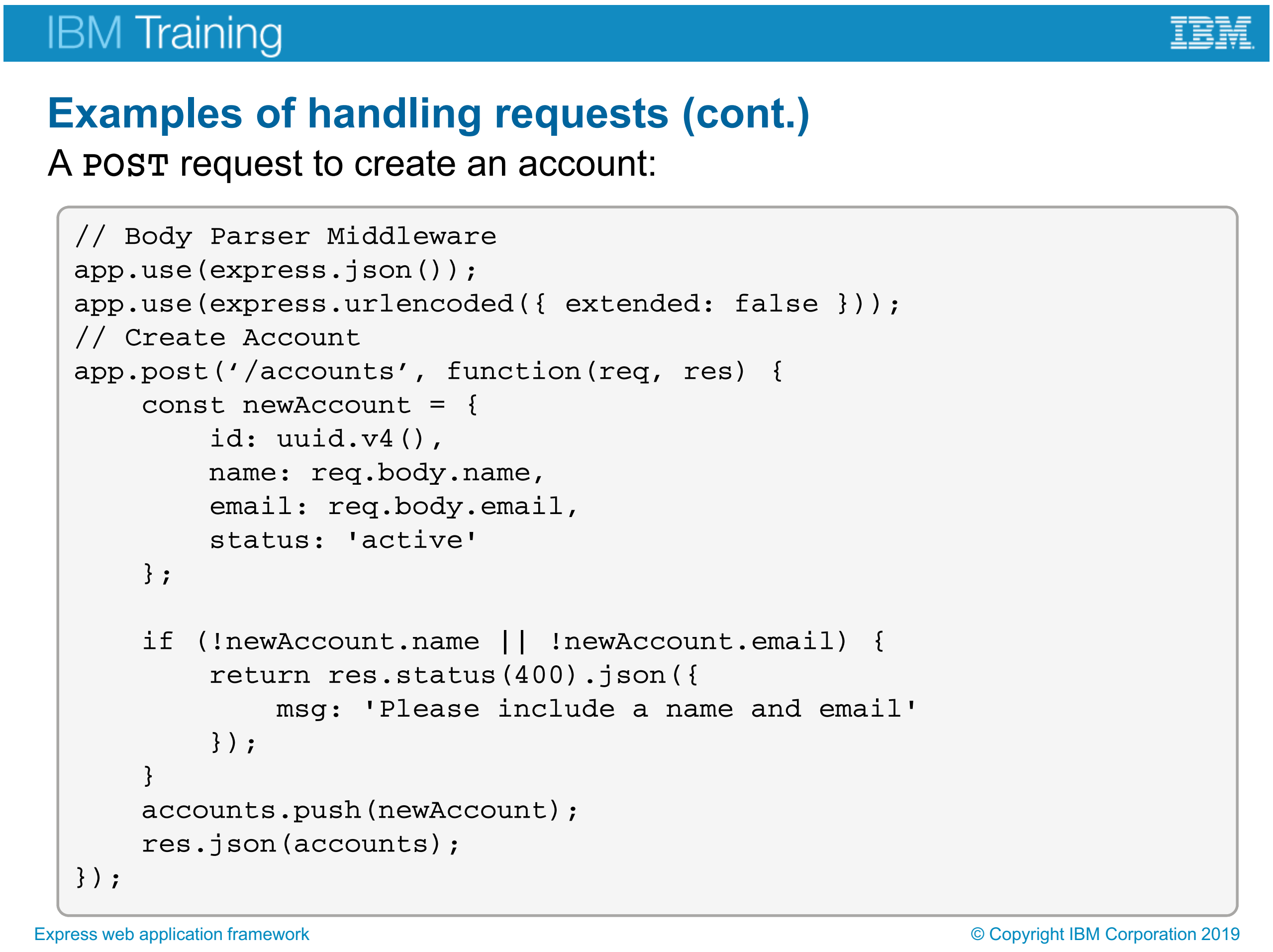


```
[
    {
        "id": 1,
        "name": "John Doe",
        "email": "john@gmail.com",
        "status": "active"
    }
]
```

Express web application framework © Copyright IBM Corporation 2019

Figure 3-21. Testing with Postman (cont.)

Getting an account by ID by using Postman.



IBM Training

Examples of handling requests (cont.)

A `POST` request to create an account:

```
// Body Parser Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
// Create Account
app.post(‘/accounts’, function(req, res) {
    const newAccount = {
        id: uuid.v4(),
        name: req.body.name,
        email: req.body.email,
        status: 'active'
    };

    if (!newAccount.name || !newAccount.email) {
        return res.status(400).json({
            msg: 'Please include a name and email'
        });
    }
    accounts.push(newAccount);
    res.json(accounts);
});
```

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-22. Examples of handling requests (cont.)

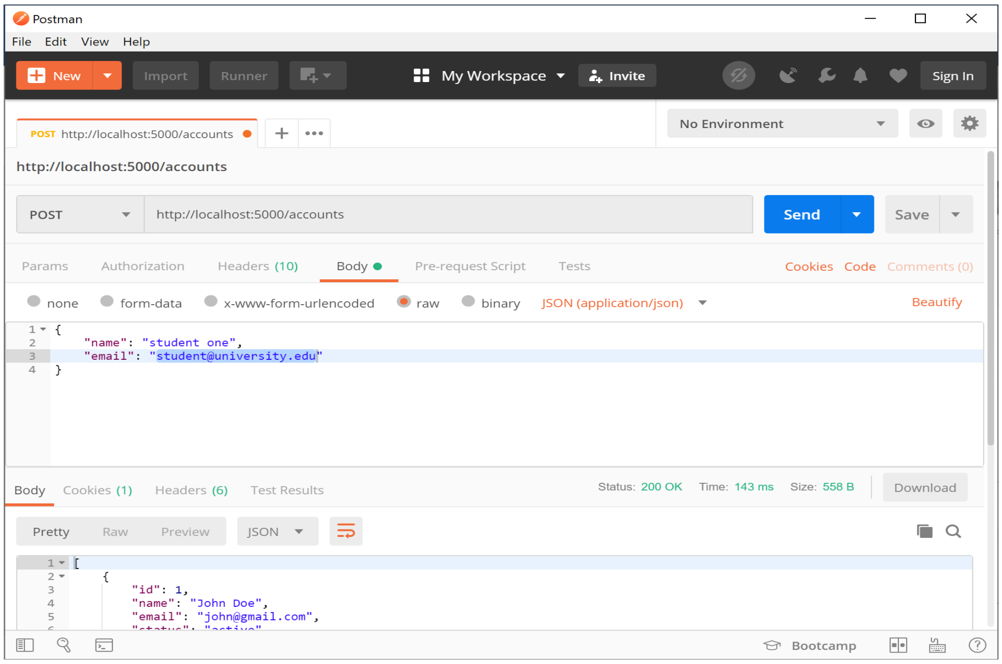
This route handles the creation of an account.

1. Use the **express.json()** middleware function to parse the request and response (or use a third-party package that is called 'body-parser’).
2. Define the route and handle it by defining an *account* object. Give it the same email and name from the body of the request; for the ID, generate it by using a third-party package to ensure randomness; and for the status, keep it as ‘active’ for any new account.
3. If a status code 400 is returned, which indicates a bad request, check whether the request contains the email and the name. The user sees a message that explains the error. If there are no errors, add it to your array and return it to the user in the response.

IBM Training

Testing with Postman (cont.)

- Create an account (part 1 of 2):


Postman
File Edit View Help
New
Import
Runner
My Workspace
Invite
Sign In
No Environment
Send
Save
Params Authorization Headers (10) Body Pre-request Script Tests
Cookies Code Comments (0)
none form-data x-www-form-urlencoded raw binary
Beautify
Body Cookies (1) Headers (6) Test Results
Download
Pretty Raw Preview JSON
Bootcamp

POST http://localhost:5000/accounts

Body: raw, JSON (application/json)

```
{
    "name": "student one",
    "email": "student@university.edu"
}
```

Status: 200 OK Time: 143 ms Size: 558 B

```
[
    {
        "id": 1,
        "name": "John Doe",
        "email": "john@gmail.com",
```

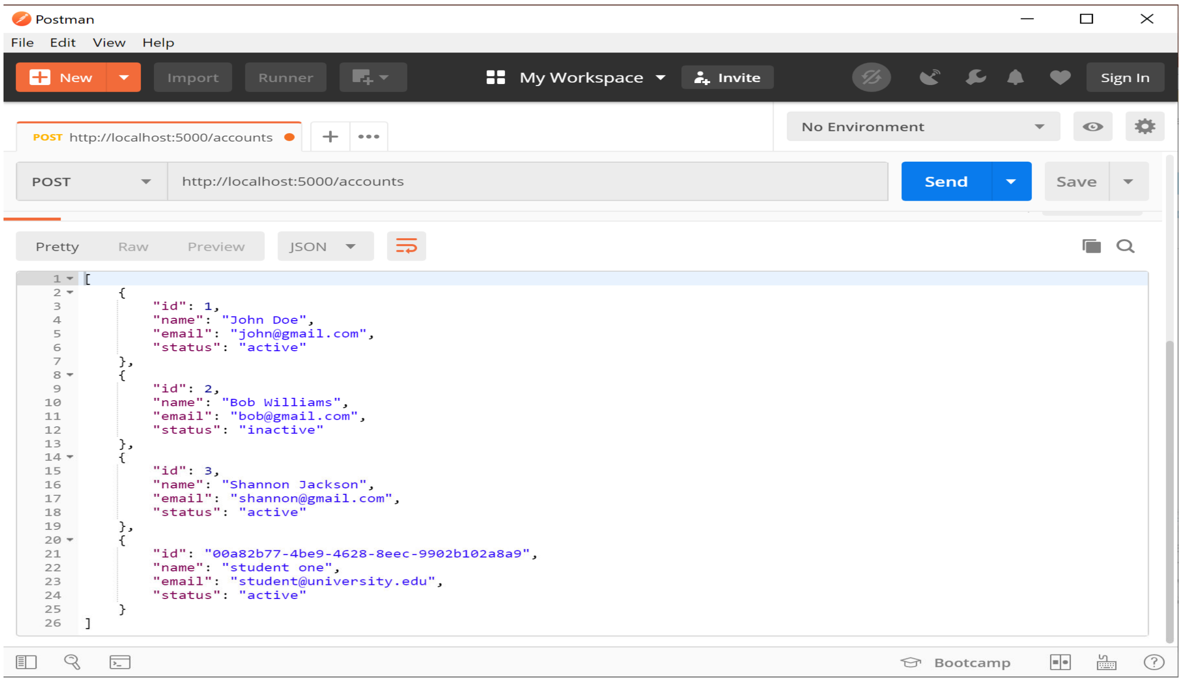
Figure 3-23. Testing with Postman (cont.)

Creating an account by using Postman (part 1 of 2).

IBM Training

Testing with Postman (cont.)

- Create an account (part 2 of 2):

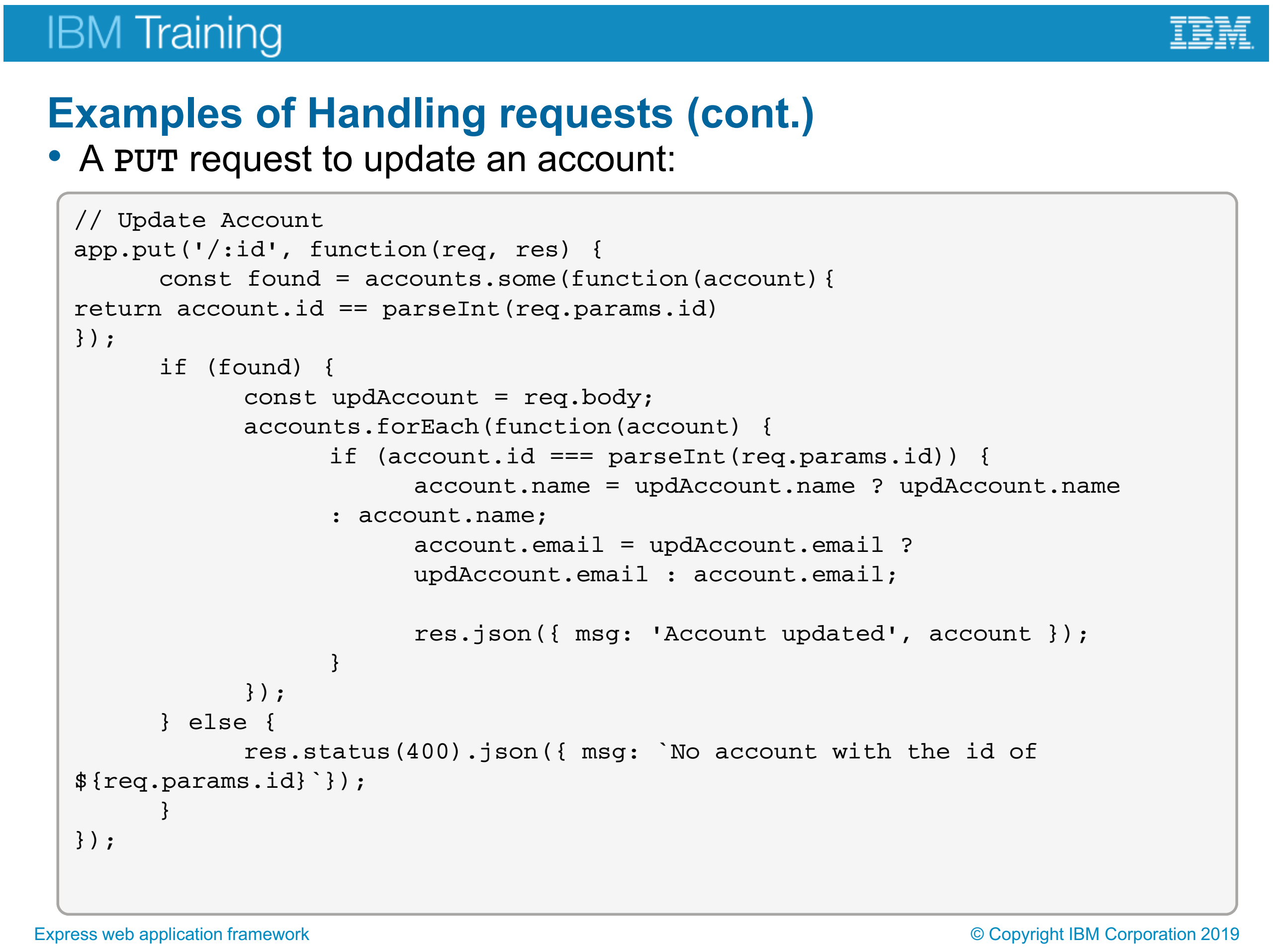


POST http://localhost:5000/accounts

```
[
    {
        "id": 1,
        "name": "John Doe",
        "email": "john@gmail.com",
        "status": "active"
    },
    {
        "id": 2,
        "name": "Bob Williams",
        "email": "bob@gmail.com",
        "status": "inactive"
    },
    {
        "id": 3,
        "name": "Shannon Jackson",
        "email": "shannon@gmail.com",
        "status": "active"
    },
    {
        "id": "00a82b77-4be9-4628-8eec-9902b102a8a9",
        "name": "student one",
        "email": "student@university.edu",
        "status": "active"
    }
]
```

Figure 3-24. Testing with Postman (cont.)

Creating an account by using Postman (part 2 of 2).



IBM Training

Examples of Handling requests (cont.)

- A **PUT** request to update an account:

```
// Update Account
app.put('/:id', function(req, res) {
      const found = accounts.some(function(account){
return account.id == parseInt(req.params.id)
});
      if (found) {
            const updAccount = req.body;
            accounts.forEach(function(account) {
                  if (account.id === parseInt(req.params.id)) {
                        account.name = updAccount.name ? updAccount.name
                  : account.name;
                        account.email = updAccount.email ?
                        updAccount.email : account.email;

                        res.json({ msg: 'Account updated', account });
                  }
            });
      } else {
            res.status(400).json({ msg: `No account with the id of
${req.params.id}`});
      }
});
```

Express web application framework © Copyright IBM Corporation 2019

Figure 3-25. Examples of Handling requests (cont.)

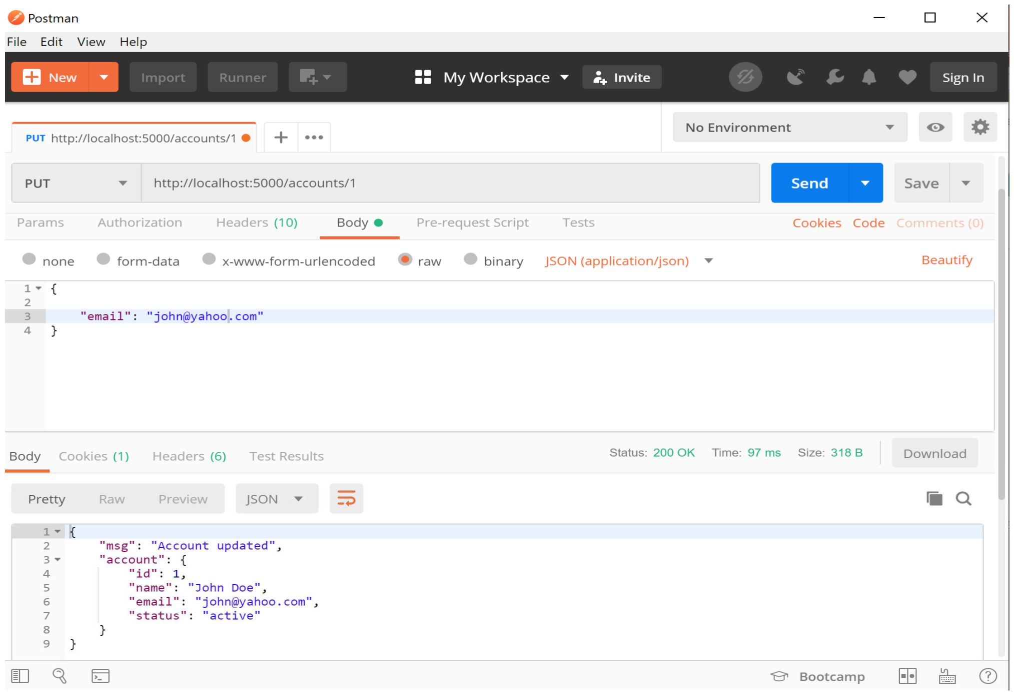
This route handles the updating of a particular account.

1. It takes the ID as a parameter and checks whether the ID is found in your list.
2. It then updates the email or the name with the new one (the caller does not have to provide both of them; only the one that needs to be updated).
3. If it succeeds, it returns the updated account.

IBM Training

Testing with Postman (cont.)

Update an account:



PUT http://localhost:5000/accounts/1

```
{

    "email": "john@yahoo.com"
}
```

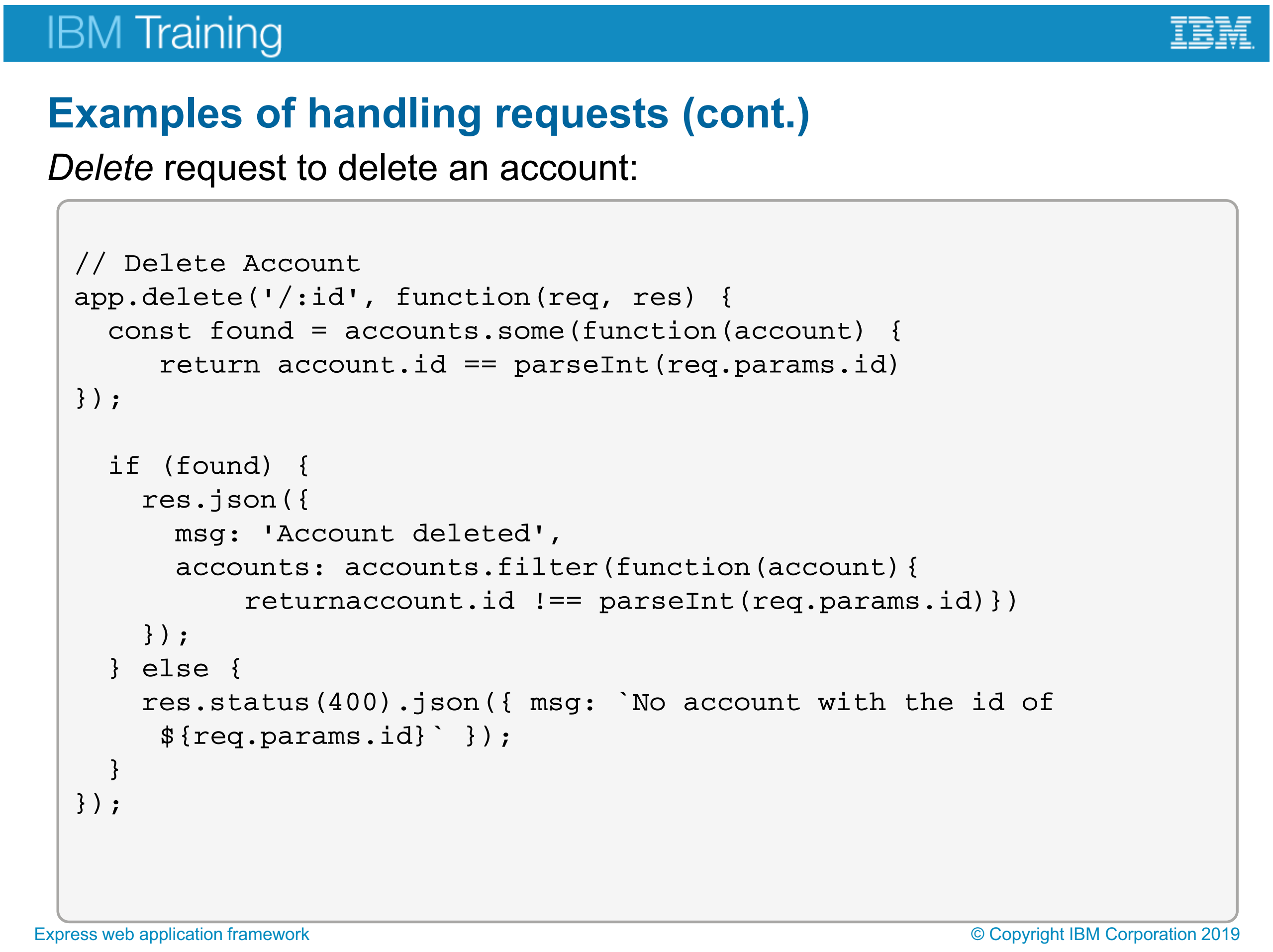
Status: 200 OK   Time: 97 ms   Size: 318 B

```
{
    "msg": "Account updated",
    "account": {
        "id": 1,
        "name": "John Doe",
        "email": "john@yahoo.com",
        "status": "active"
    }
}
```

Express web application framework

Figure 3-26. Testing with Postman (cont.)

Updating an existing account by using Postman.



IBM Training

Examples of handling requests (cont.)

Delete request to delete an account:

```
// Delete Account
app.delete('/:id', function(req, res) {
  const found = accounts.some(function(account) {
     return account.id == parseInt(req.params.id)
});

  if (found) {
    res.json({
      msg: 'Account deleted',
      accounts: accounts.filter(function(account){
          returnaccount.id !== parseInt(req.params.id)})
    });
  } else {
    res.status(400).json({ msg: `No account with the id of
     ${req.params.id}` });
  }
});
```

Express web application framework

© Copyright IBM Corporation 2019

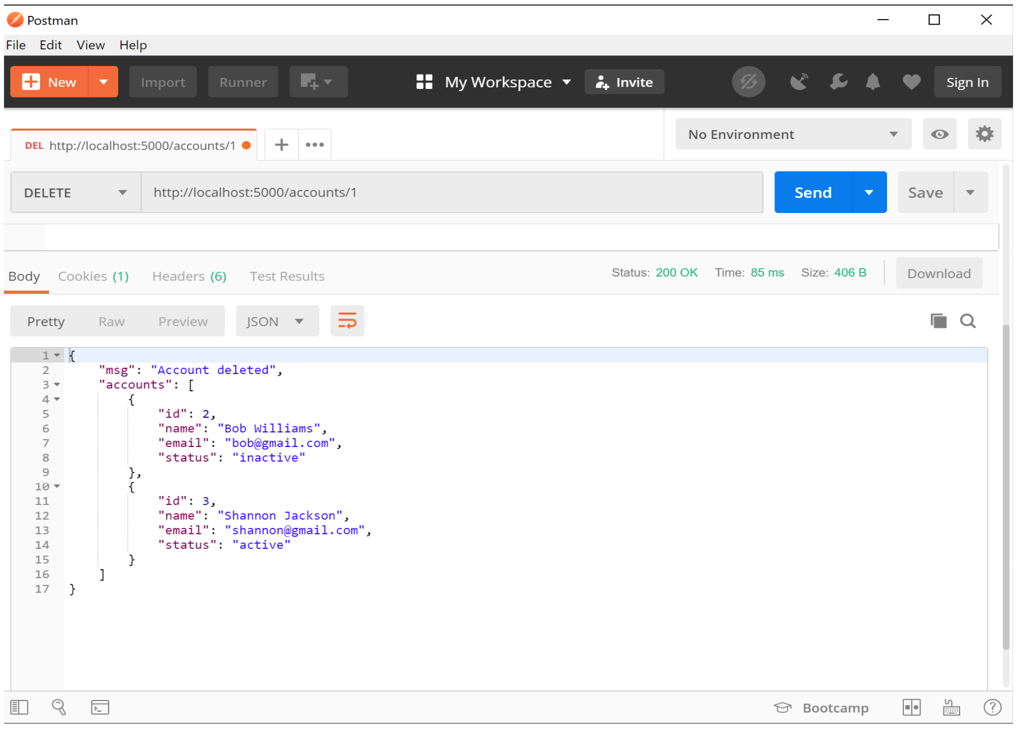
Figure 3-27. Examples of handling requests (cont.)

This route handles the deletion of an account.

IBM Training

Testing with Postman (cont.)

Deleting an account:



DEL http://localhost:5000/accounts/1

DELETE http://localhost:5000/accounts/1

Status: 200 OK Time: 85 ms Size: 406 B

```
{
    "msg": "Account deleted",
    "accounts": [
        {
            "id": 2,
            "name": "Bob Williams",
            "email": "bob@gmail.com",
            "status": "inactive"
        },
        {
            "id": 3,
            "name": "Shannon Jackson",
            "email": "shannon@gmail.com",
            "status": "active"
        }
    ]
}
```

Express web application framework © Copyright IBM Corporation 2019

Figure 3-28. Testing with Postman (cont.)

Deleting an account by using Postman.

3.5. Middleware functions

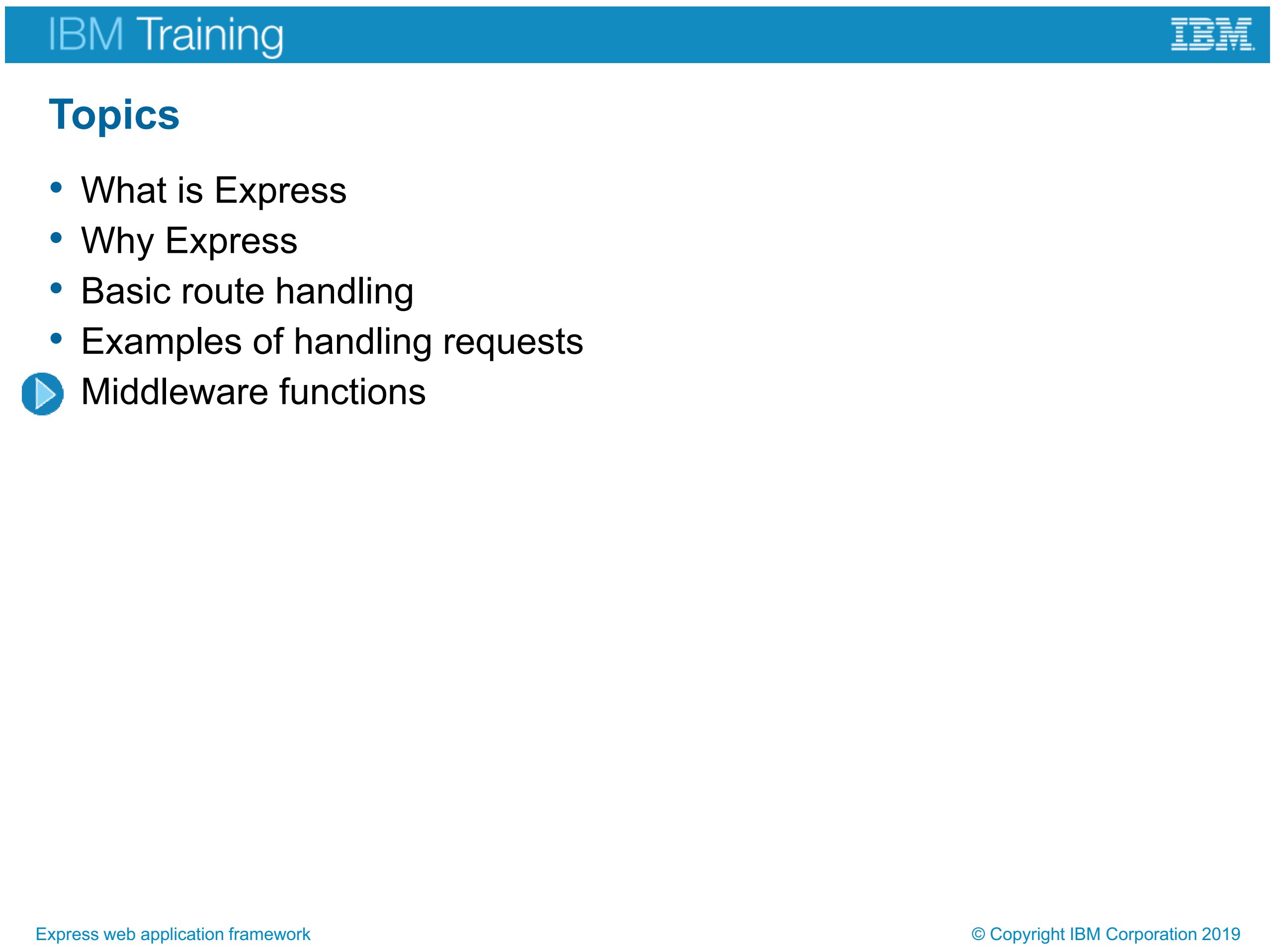
IBM Training

Middleware functions

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-29. Middleware functions



IBM Training

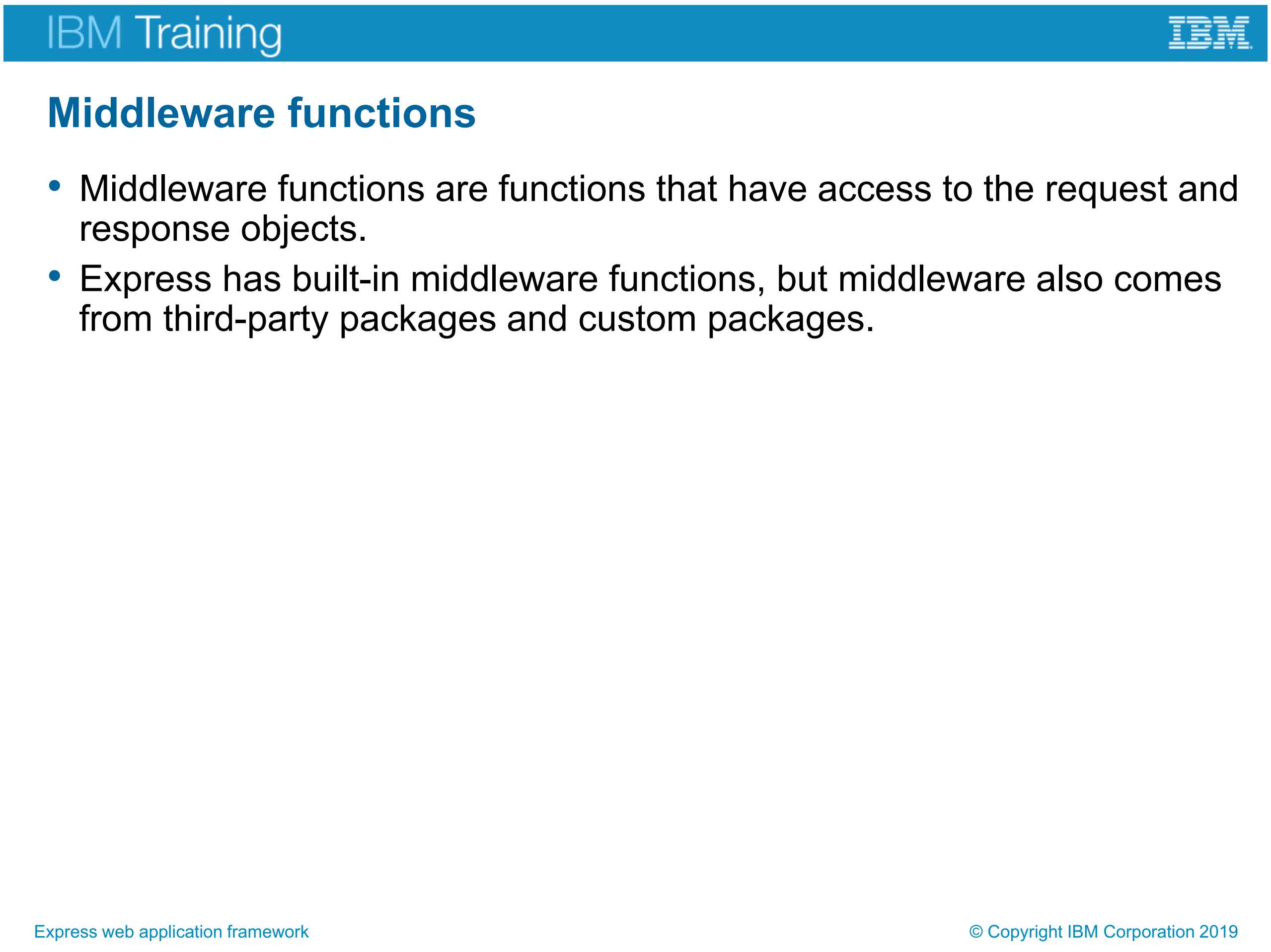
Topics

- What is Express
- Why Express
- Basic route handling
- Examples of handling requests
- Middleware functions

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-30. Topics



IBM Training

Middleware functions

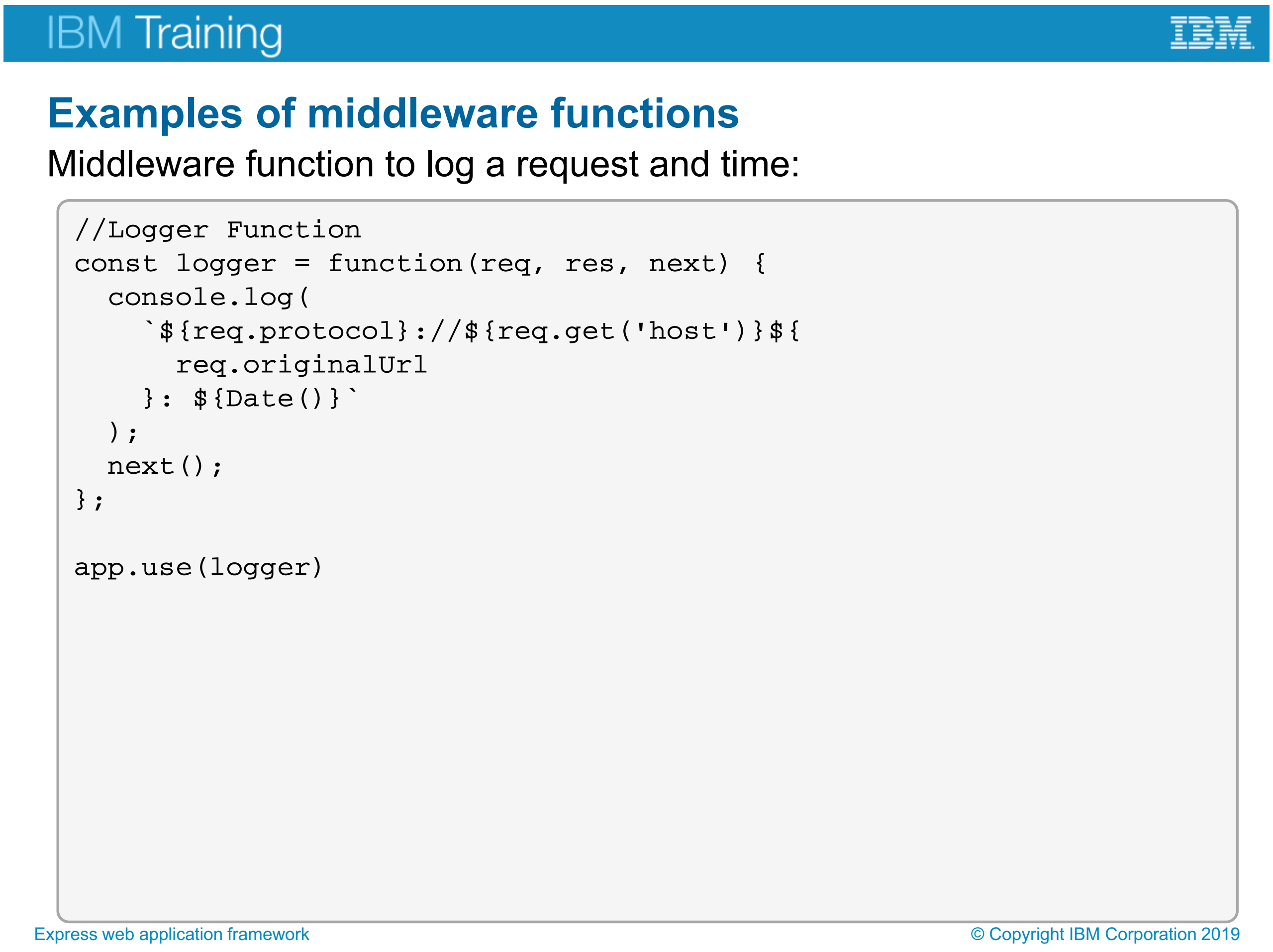
- Middleware functions are functions that have access to the request and response objects.
- Express has built-in middleware functions, but middleware also comes from third-party packages and custom packages.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-31. Middleware functions

- Middleware functions are functions that have access to the request and response objects. Express has built-in middleware functions, but middleware also comes from third-party packages and custom packages.
- Middleware functions are like a stack of functions that runs whenever a request is made to a server.
- With these functions, you can run any code, change request or response objects, end the response cycle, or call the next middleware in the stack.



IBM Training

Examples of middleware functions

Middleware function to log a request and time:

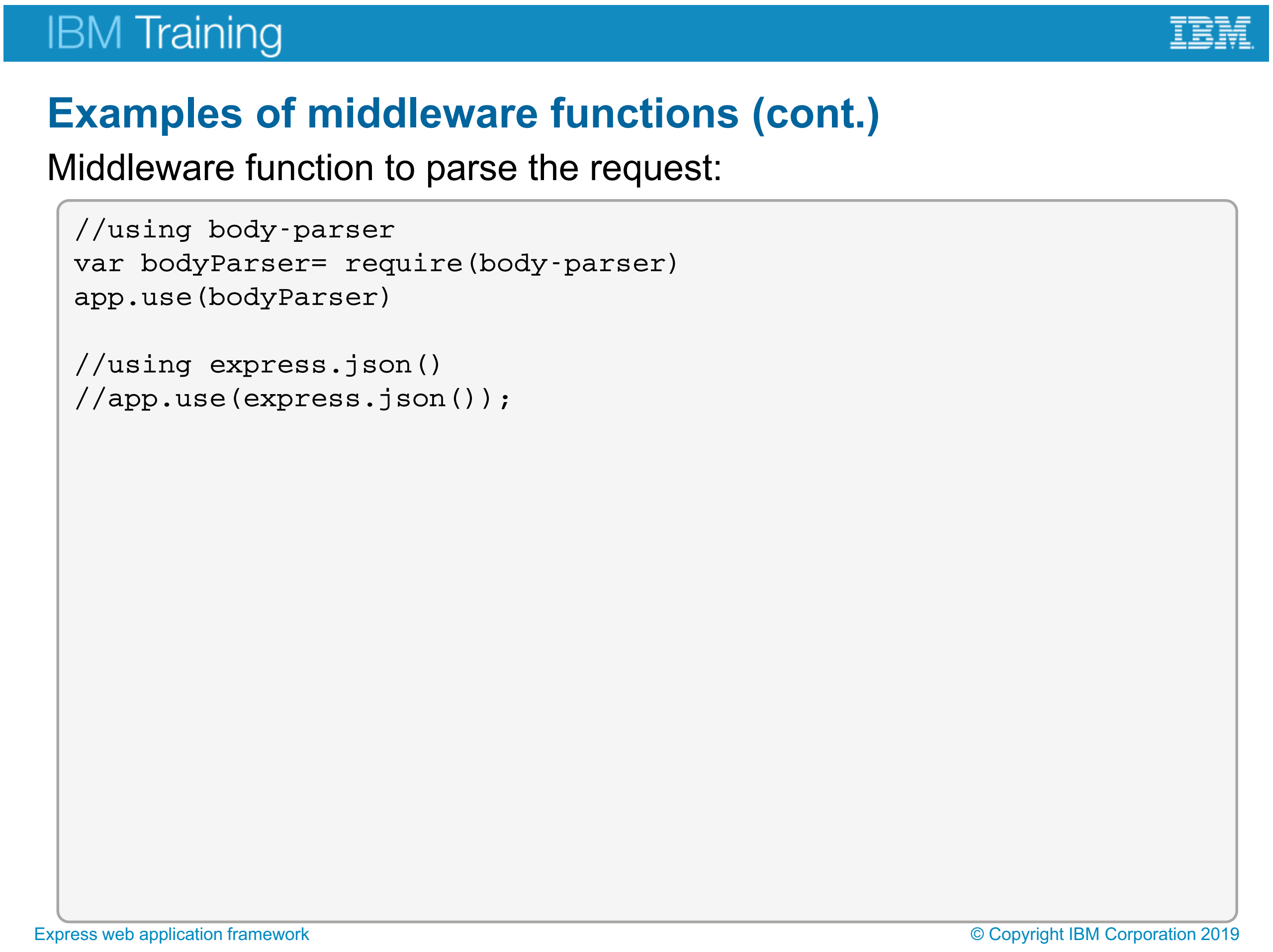
```
//Logger Function
const logger = function(req, res, next) {
  console.log(
    `${req.protocol}://${req.get('host')}${
      req.originalUrl
    }: ${Date()}`
  );
  next();
};

app.use(logger)
```

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-32. Examples of middleware functions

Here is an example of middleware functions. This simple middleware function logs to the console the URL of the request and the time that it is requested. Then, you use the **app.use()** function and pass your middleware to it to apply the function to all the requests.



IBM Training

Examples of middleware functions (cont.)

Middleware function to parse the request:

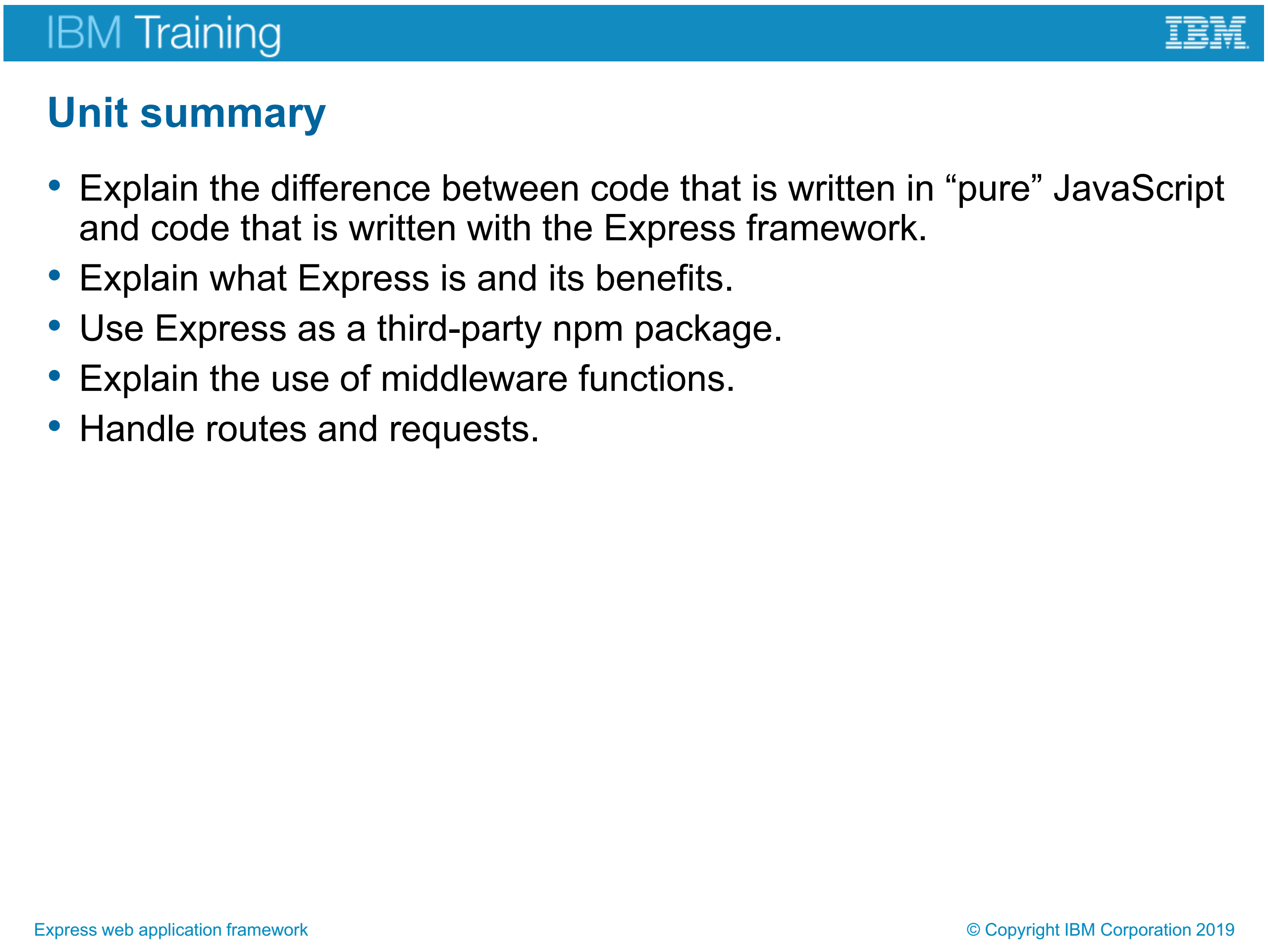
```
//using body-parser
var bodyParser= require(body-parser)
app.use(bodyParser)

//using express.json()
//app.use(express.json());
```

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-33. Examples of middleware functions (cont.)

You can use the third-party package 'body-parser’ to parse a request body or use the one that is provided by Express, **express.json()**.



IBM Training

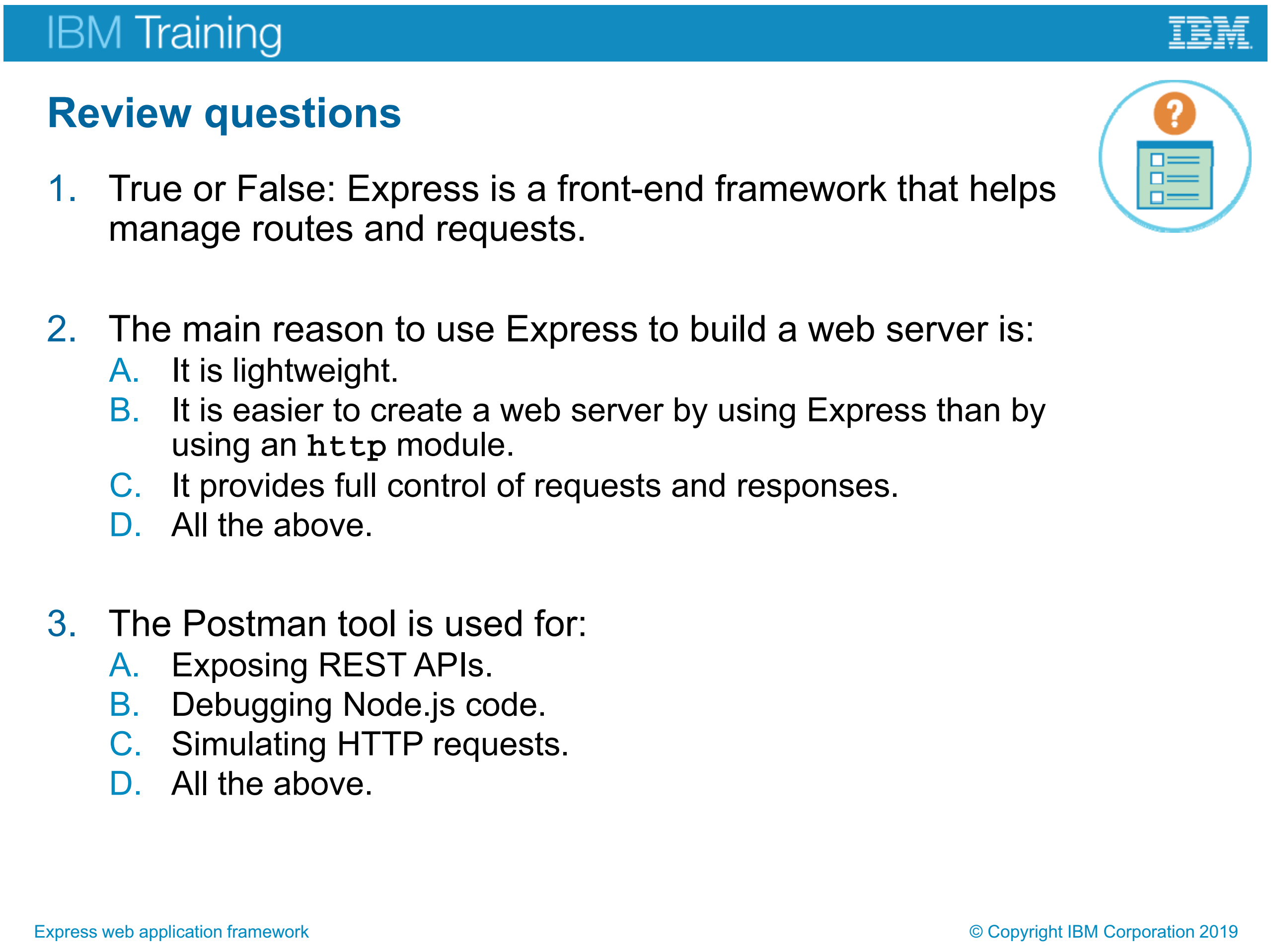
Unit summary

- Explain the difference between code that is written in “pure” JavaScript and code that is written with the Express framework.
- Explain what Express is and its benefits.
- Use Express as a third-party npm package.
- Explain the use of middleware functions.
- Handle routes and requests.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-34. Unit summary



IBM Training

Review questions

1. True or False: Express is a front-end framework that helps manage routes and requests.

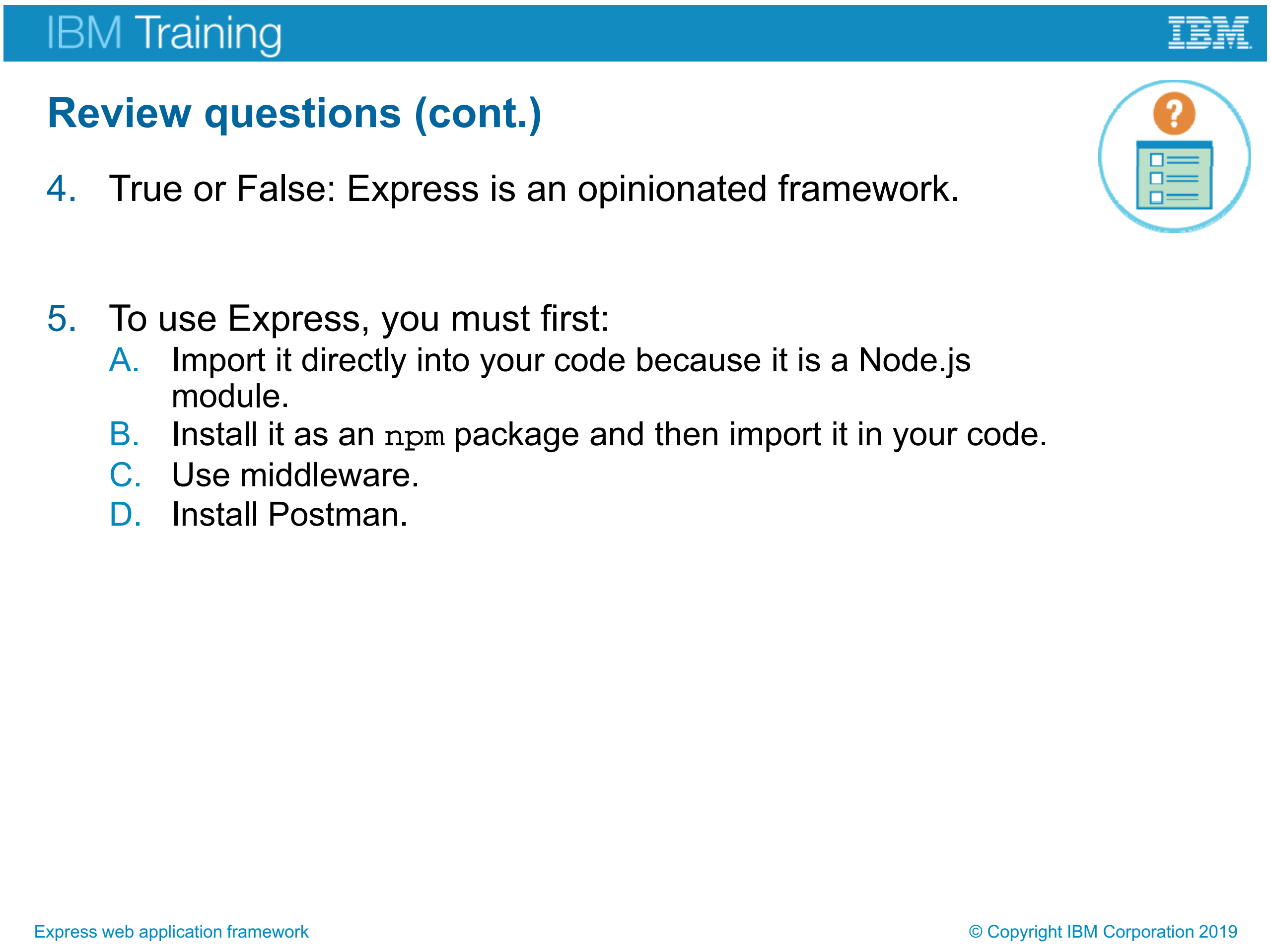
2. The main reason to use Express to build a web server is:
   A. It is lightweight.
   B. It is easier to create a web server by using Express than by using an `http` module.
   C. It provides full control of requests and responses.
   D. All the above.

3. The Postman tool is used for:
   A. Exposing REST APIs.
   B. Debugging Node.js code.
   C. Simulating HTTP requests.
   D. All the above.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-35. Review questions



IBM Training

IBM

Review questions (cont.)

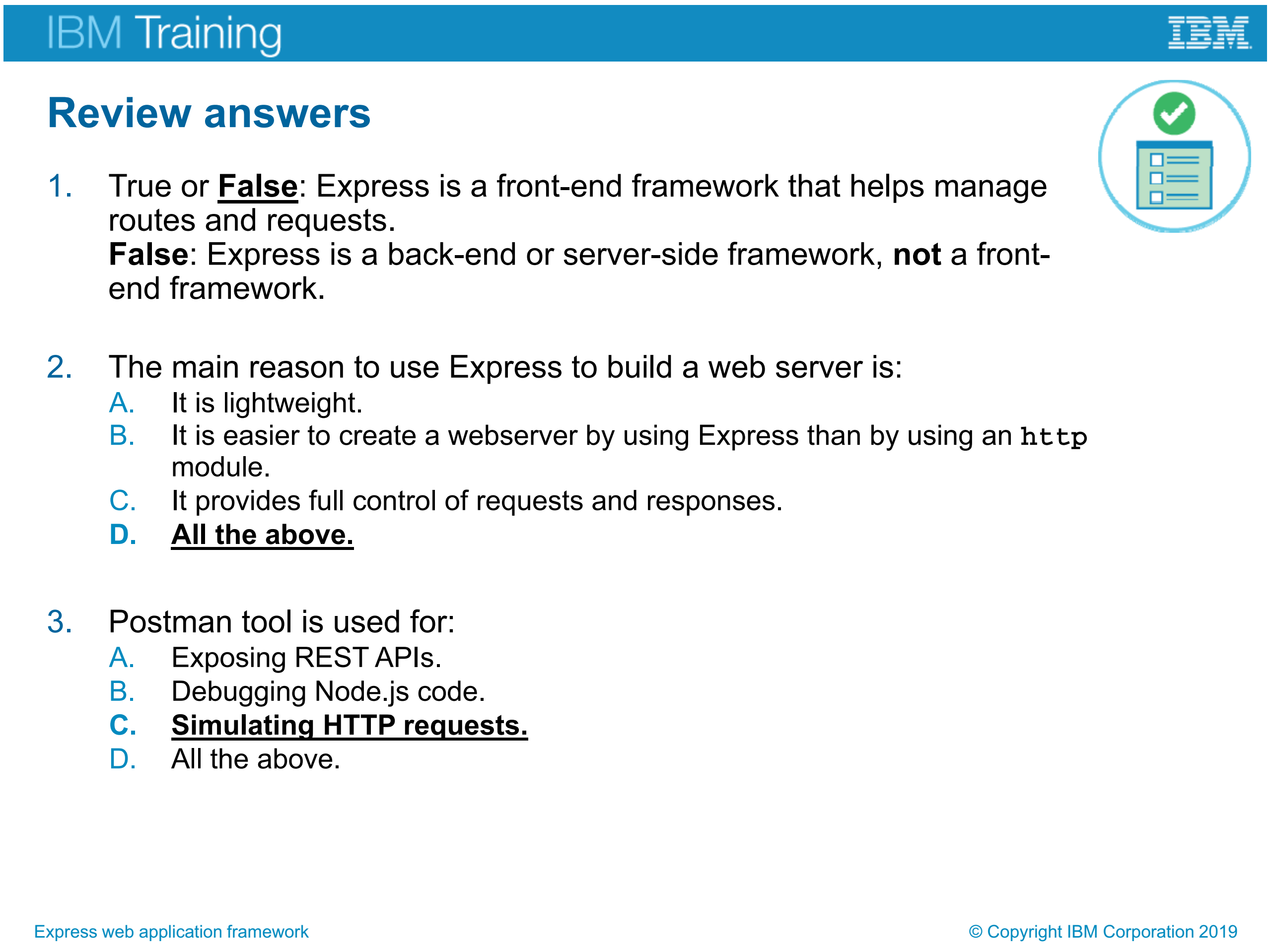
4. True or False: Express is an opinionated framework.

5. To use Express, you must first:
   A. Import it directly into your code because it is a Node.js module.
   B. Install it as an `npm` package and then import it in your code.
   C. Use middleware.
   D. Install Postman.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-36. Review questions (cont.)



IBM Training

Review answers

1. True or **<u>False</u>**: Express is a front-end framework that helps manage routes and requests.
   False: Express is a back-end or server-side framework, **not** a front-end framework.

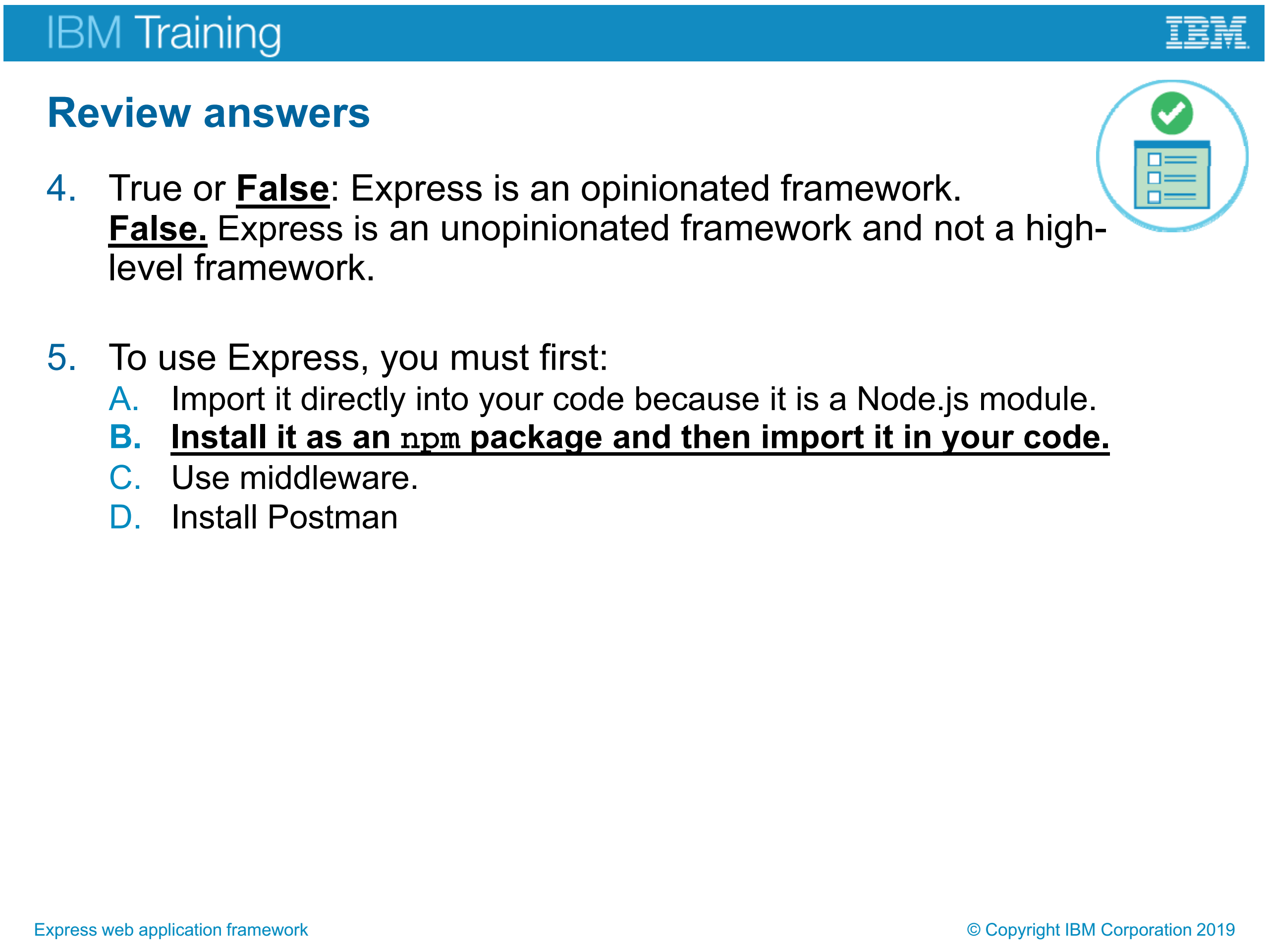
2. The main reason to use Express to build a web server is:
   A. It is lightweight.
   B. It is easier to create a webserver by using Express than by using an `http` module.
   C. It provides full control of requests and responses.
   D. <u>All the above.</u>

3. Postman tool is used for:
   A. Exposing REST APIs.
   B. Debugging Node.js code.
   C. <u>Simulating HTTP requests.</u>
   D. All the above.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-37. Review answers



IBM Training

Review answers

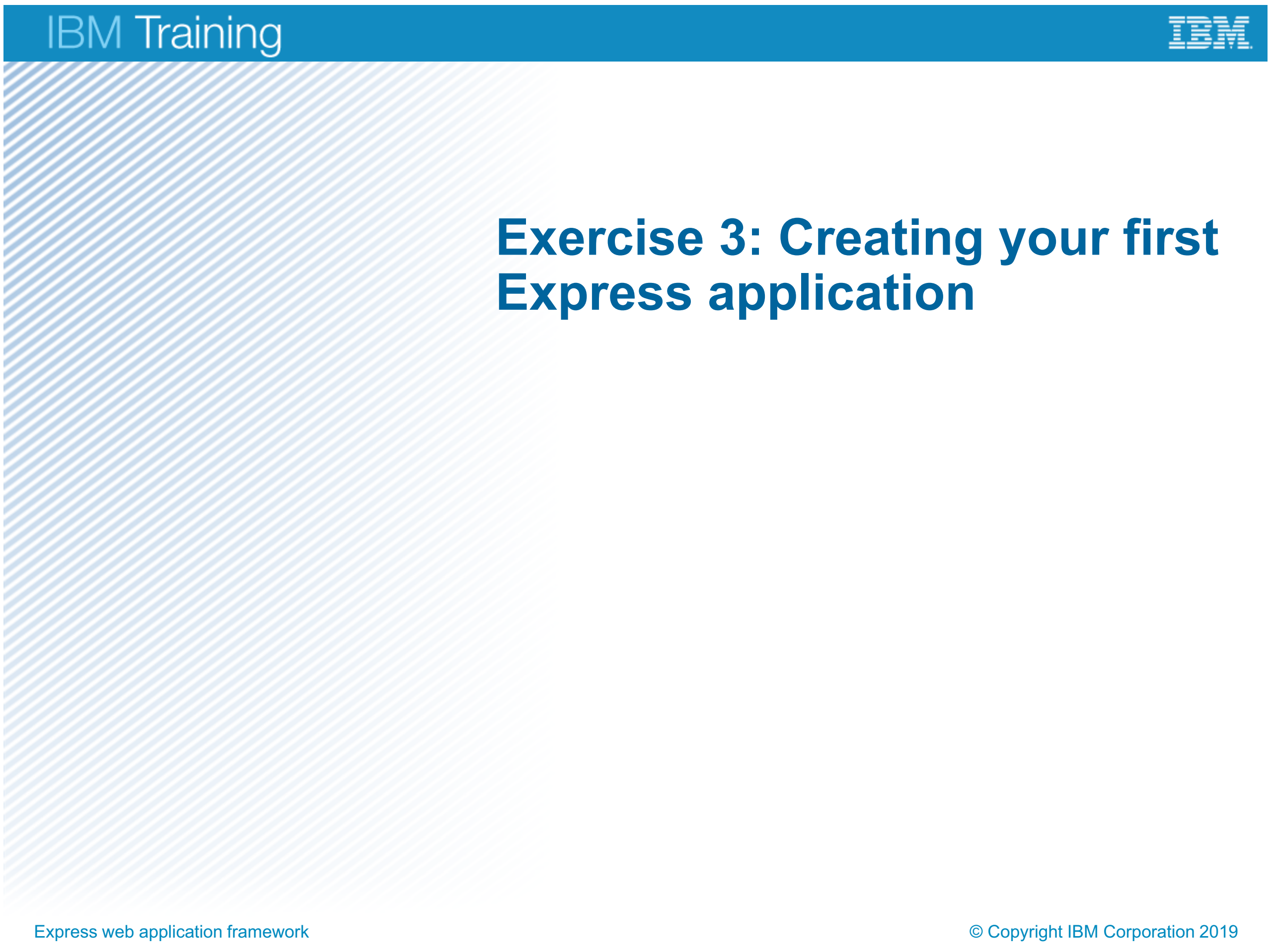
4. True or **<u>False</u>**: Express is an opinionated framework.
   <u>False.</u> Express is an unopinionated framework and not a high-level framework.

5. To use Express, you must first:
   A. Import it directly into your code because it is a Node.js module.
   B. <u>Install it as an `npm` package and then import it in your code.</u>
   C. Use middleware.
   D. Install Postman

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-38. Review answers



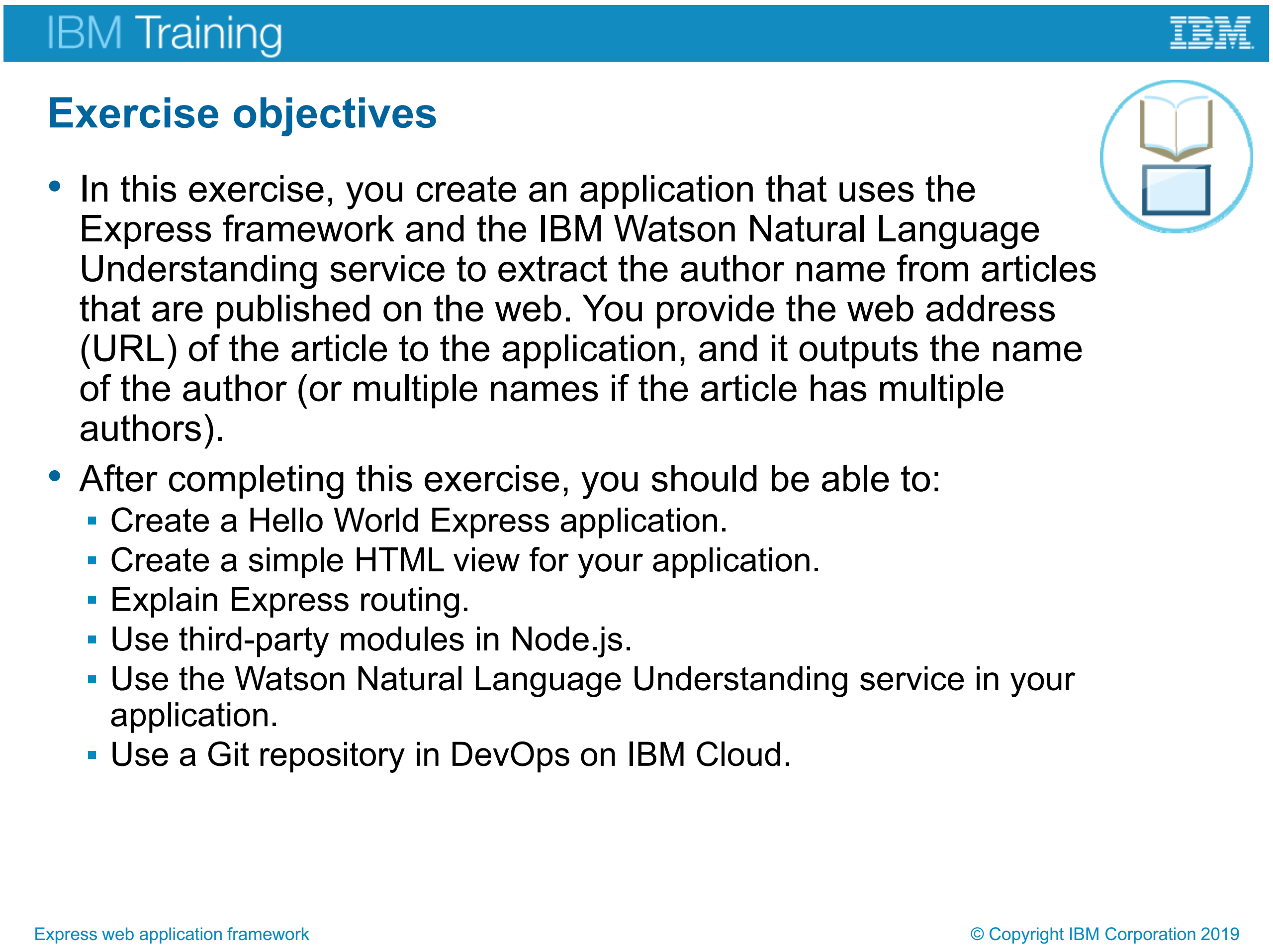
IBM Training

Exercise 3: Creating your first Express application

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-39. Exercise 3: Creating your first Express application



IBM Training

Exercise objectives

- In this exercise, you create an application that uses the Express framework and the IBM Watson Natural Language Understanding service to extract the author name from articles that are published on the web. You provide the web address (URL) of the article to the application, and it outputs the name of the author (or multiple names if the article has multiple authors).
- After completing this exercise, you should be able to:
  - Create a Hello World Express application.
  - Create a simple HTML view for your application.
  - Explain Express routing.
  - Use third-party modules in Node.js.
  - Use the Watson Natural Language Understanding service in your application.
  - Use a Git repository in DevOps on IBM Cloud.

Express web application framework

© Copyright IBM Corporation 2019

Figure 3-40. Exercise objectives

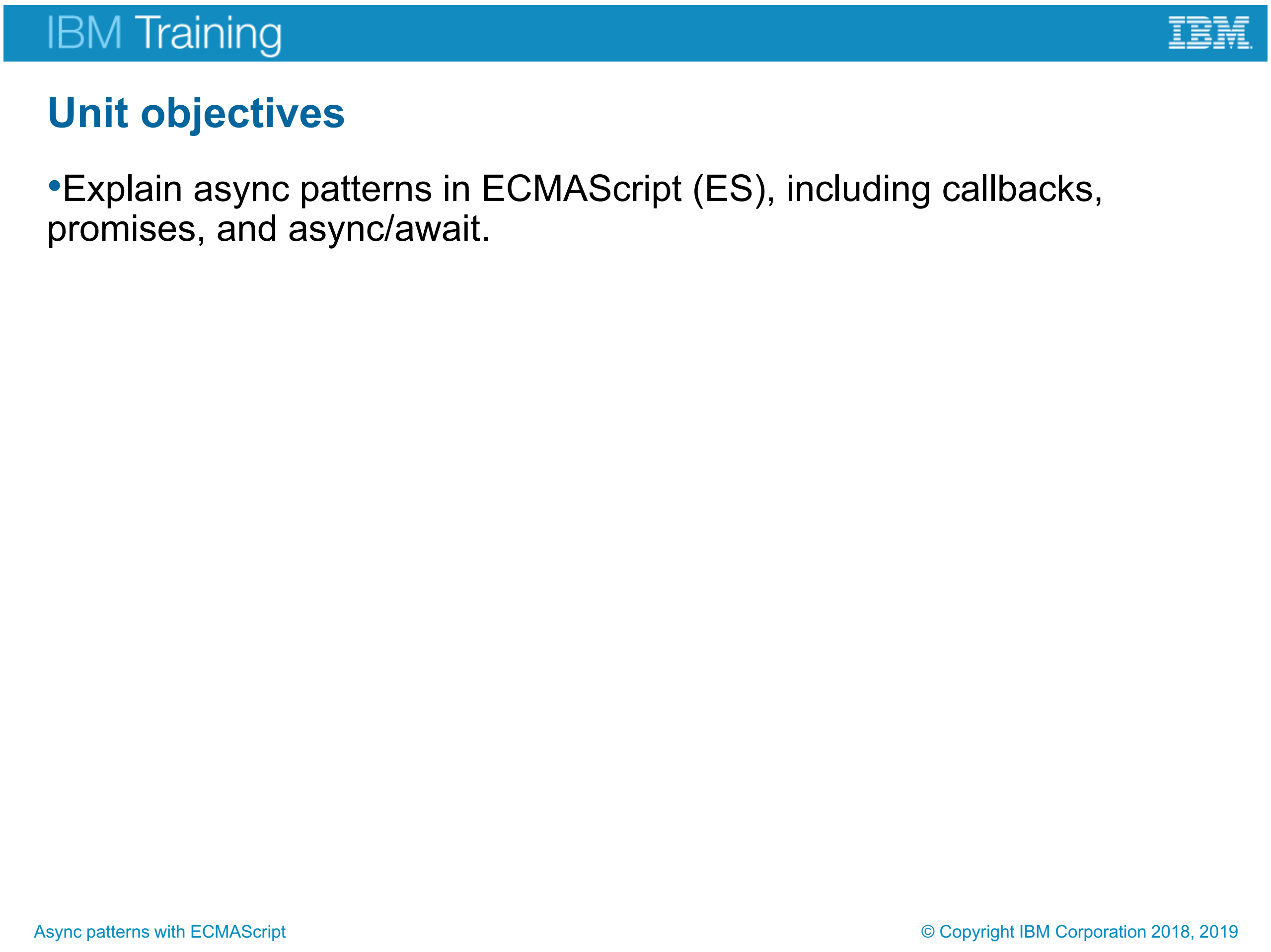
Unit 4. Async patterns with ECMAScript

Estimated time

01:00

Overview

This unit describes async patterns in ECMAScript including callbacks, promises, and async/await.



IBM Training

Unit objectives

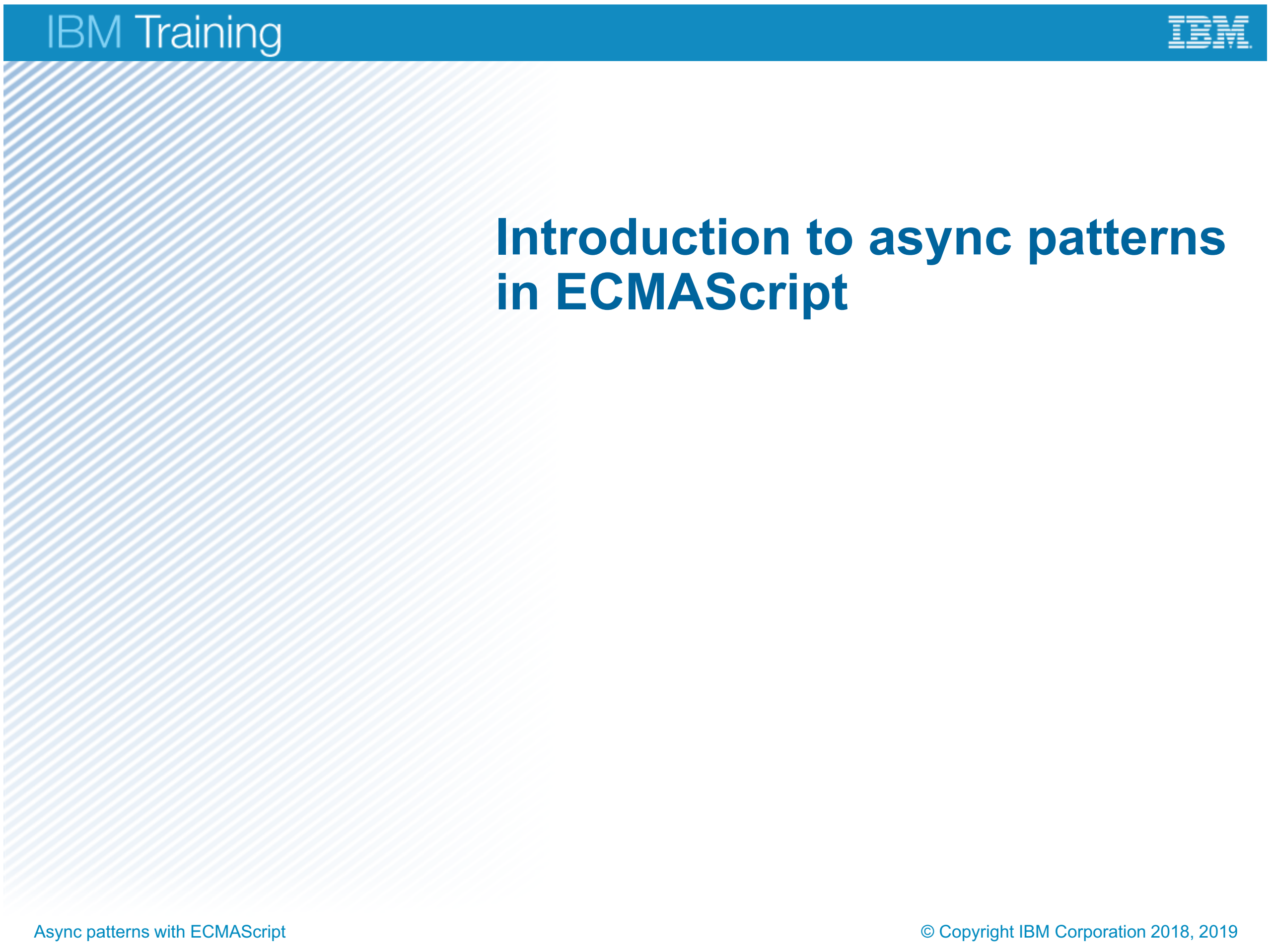
- Explain async patterns in ECMAScript (ES), including callbacks, promises, and async/await.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-1. Unit objectives

4.1. Introduction to async patterns in ECMAScript



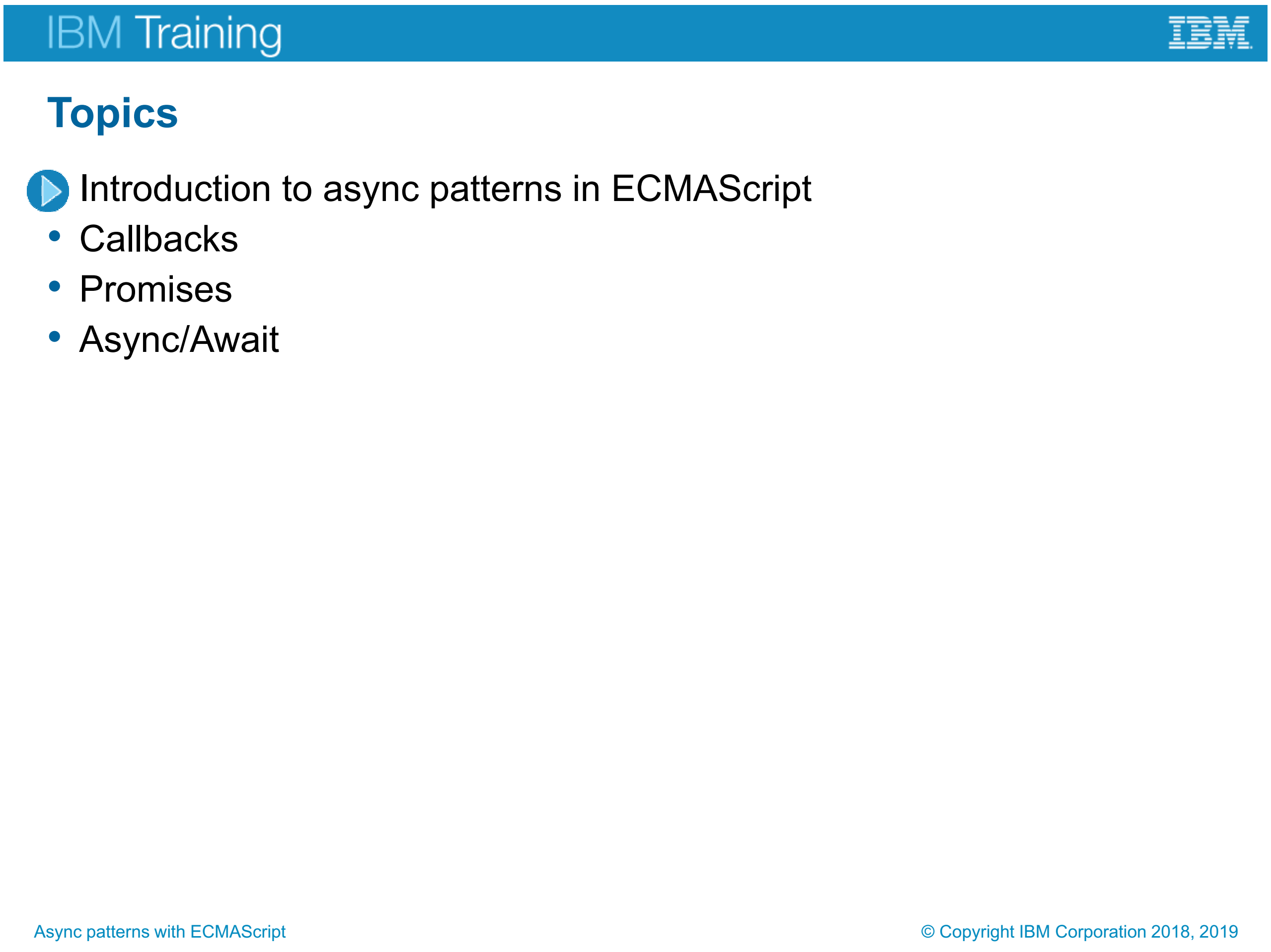
IBM Training

Introduction to async patterns in ECMAScript

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-2. Introduction to async patterns in ECMAScript



IBM Training

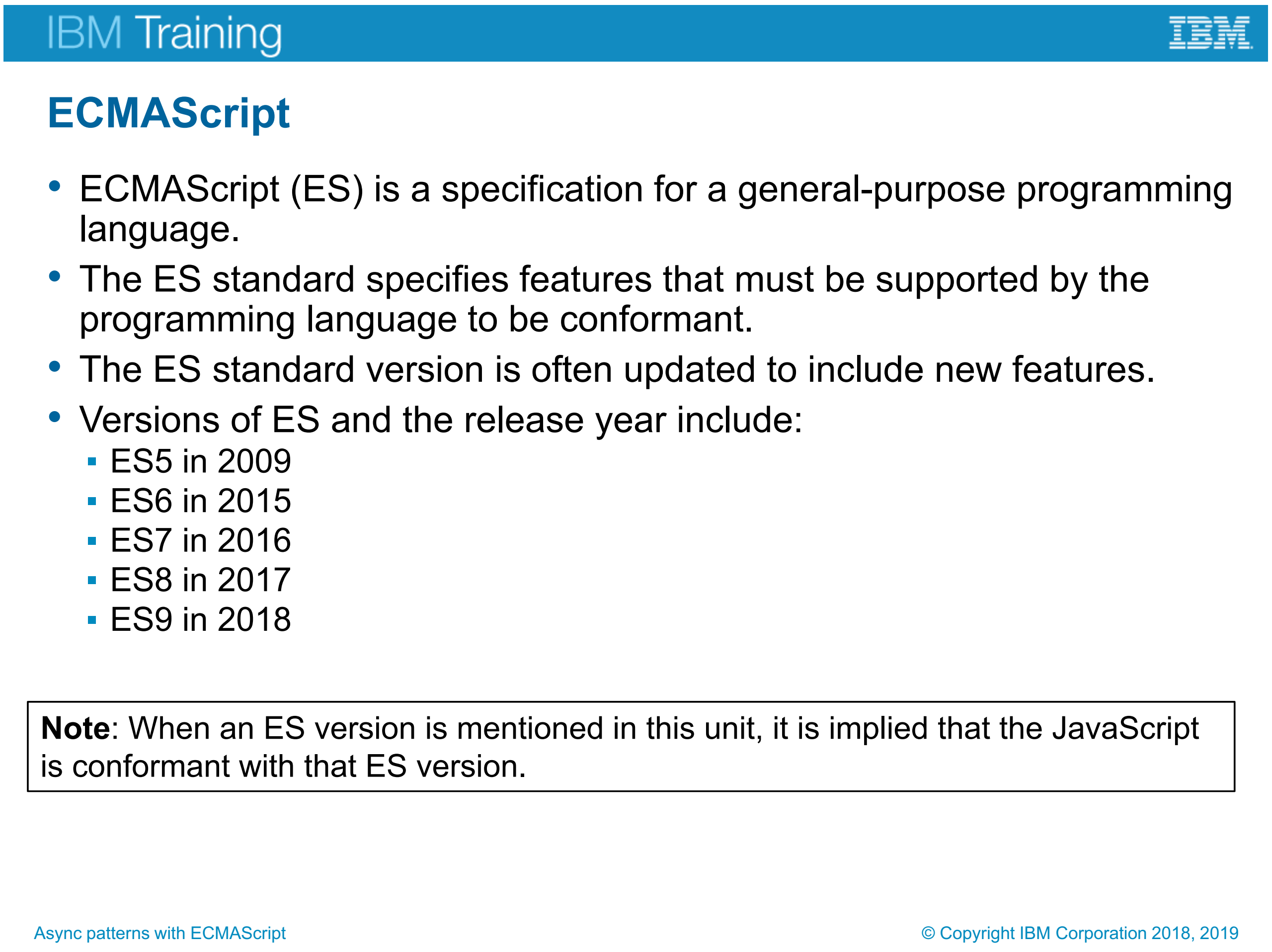
Topics

- Introduction to async patterns in ECMAScript
- Callbacks
- Promises
- Async/Await

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-3. Topics


IBM Training

ECMAScript

- ECMAScript (ES) is a specification for a general-purpose programming language.
- The ES standard specifies features that must be supported by the programming language to be conformant.
- The ES standard version is often updated to include new features.
- Versions of ES and the release year include:
  - ES5 in 2009
  - ES6 in 2015
  - ES7 in 2016
  - ES8 in 2017
  - ES9 in 2018

Note: When an ES version is mentioned in this unit, it is implied that the JavaScript is conformant with that ES version.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-4. ECMAScript

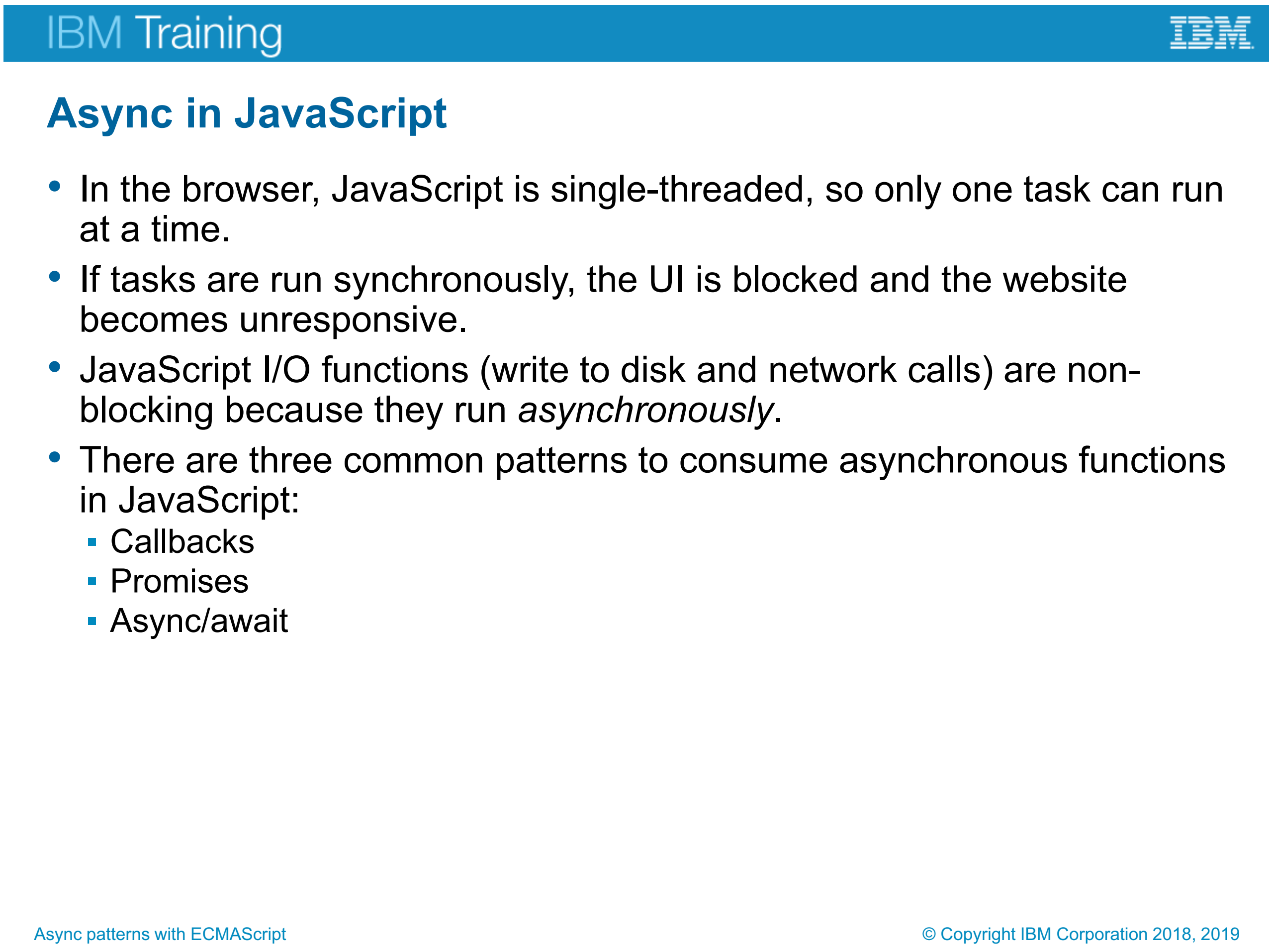
- ES is a specification for a general-purpose programming language.
- The ES standard specifies features that must be supported by the programming language to be conformant.
- The ES standard version is often updated to include new features.
- Versions of ES and the release year include:
  - ES5 in 2009
  - ES6 in 2015
  - ES7 in 2016
  - ES8 in 2017
  - ES9 in 2018

Note

Moving forward, when an ES version is mentioned in this unit, it is implied that the JavaScript is conformant with that ES version.

The ES standards allow programmers to be confident that the features that they use are supported if the programming language is compliant with a version of ES that supports those features.
There are many more features that are defined by the ES standards. For example, ES6 introduced **let** versus **const**, classes, template strings, destructuring, anonymous/arrow functions and much more.
For a full list of the language features that are supported in the versions of ES, go to the following website:

https://www.ecma-international.org/publications/standards/Ecma-262.htm


IBM Training

Async in JavaScript

- In the browser, JavaScript is single-threaded, so only one task can run at a time.
- If tasks are run synchronously, the UI is blocked and the website becomes unresponsive.
- JavaScript I/O functions (write to disk and network calls) are non-blocking because they run *asynchronously*.
- There are three common patterns to consume asynchronous functions in JavaScript:
  - Callbacks
  - Promises
  - Async/await

Async patterns with ECMAScript

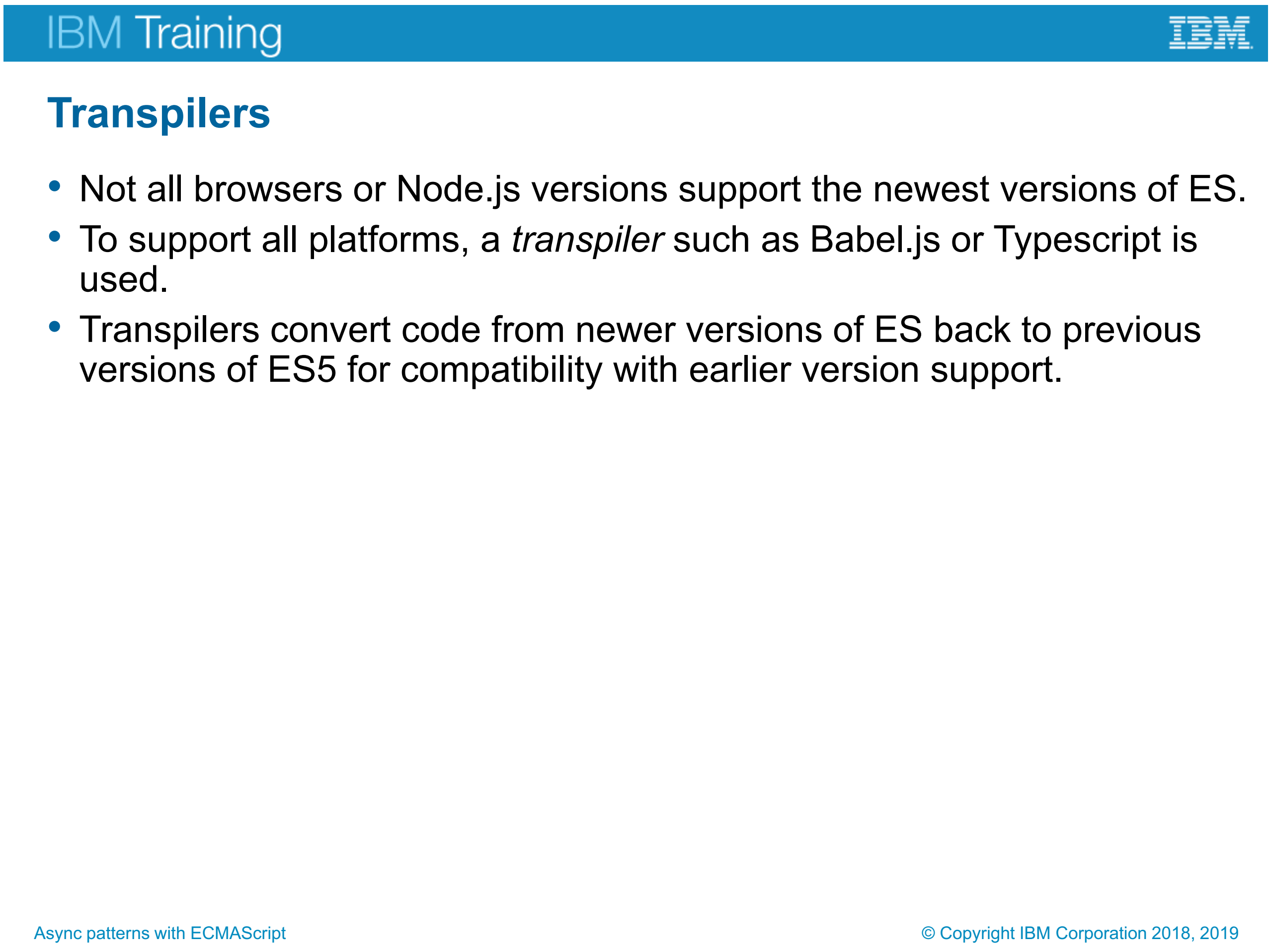
© Copyright IBM Corporation 2018, 2019

Figure 4-5. Async in JavaScript

- In the browser, JavaScript is single-threaded, so only one task can run at a time.
- If tasks are run synchronously, the UI is blocked and the website becomes unresponsive.
- JavaScript I/O functions (write to disk and network calls) are non-blocking because they run *asynchronously*.
- There are three common patterns to consume asynchronous functions in JavaScript:
  - Callbacks
  - Promises
  - Async/await

These common patterns allow asynchronous functions to run in a deterministic manner.

For more information, see the JavaScript Event Loop at the following website: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop



IBM Training

Transpilers

- Not all browsers or Node.js versions support the newest versions of ES.
- To support all platforms, a *transpiler* such as Babel.js or Typescript is used.
- Transpilers convert code from newer versions of ES back to previous versions of ES5 for compatibility with earlier version support.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-6. Transpilers

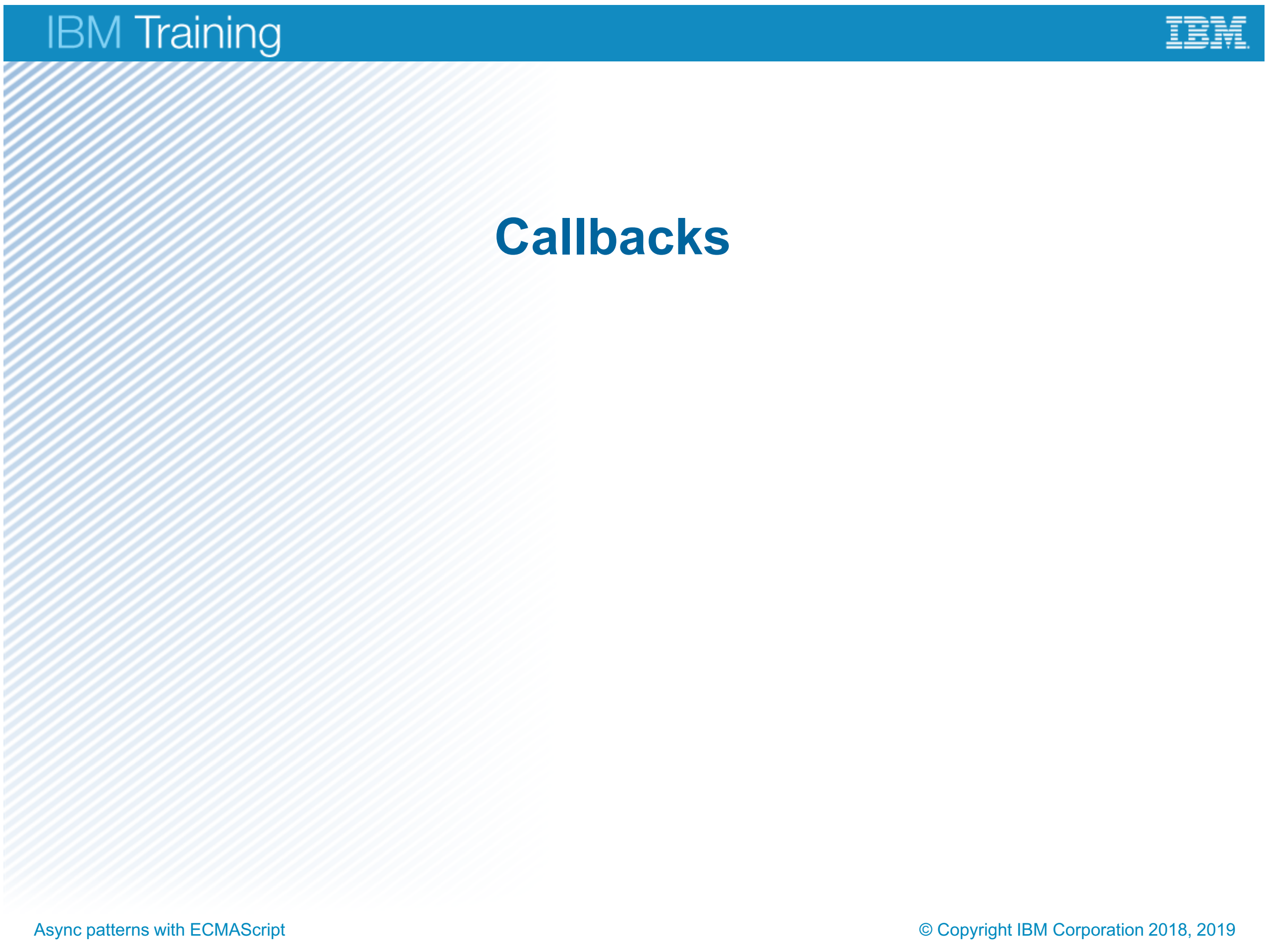
- Not all browsers or Node.js versions support the newest versions of ES.
- To support all platforms, a *transpiler* such as Babel.js or Typescript is used.
- Transpilers convert code from newer versions of ES back to previous versions of ES5 for compatibility with earlier version support.

If you are using an ES5 engine, likely it will not support the features of ES6. Transpilers allow programmers to write code in a newer version of ES and then the transpiler converts the code to be conformant with an older, supported version of ES.

To learn more about ES compatibility with earlier version, see the following resource:

https://developer.ibm.com/articles/wa-ecmascript6-neward-p1/

4.2. Callbacks



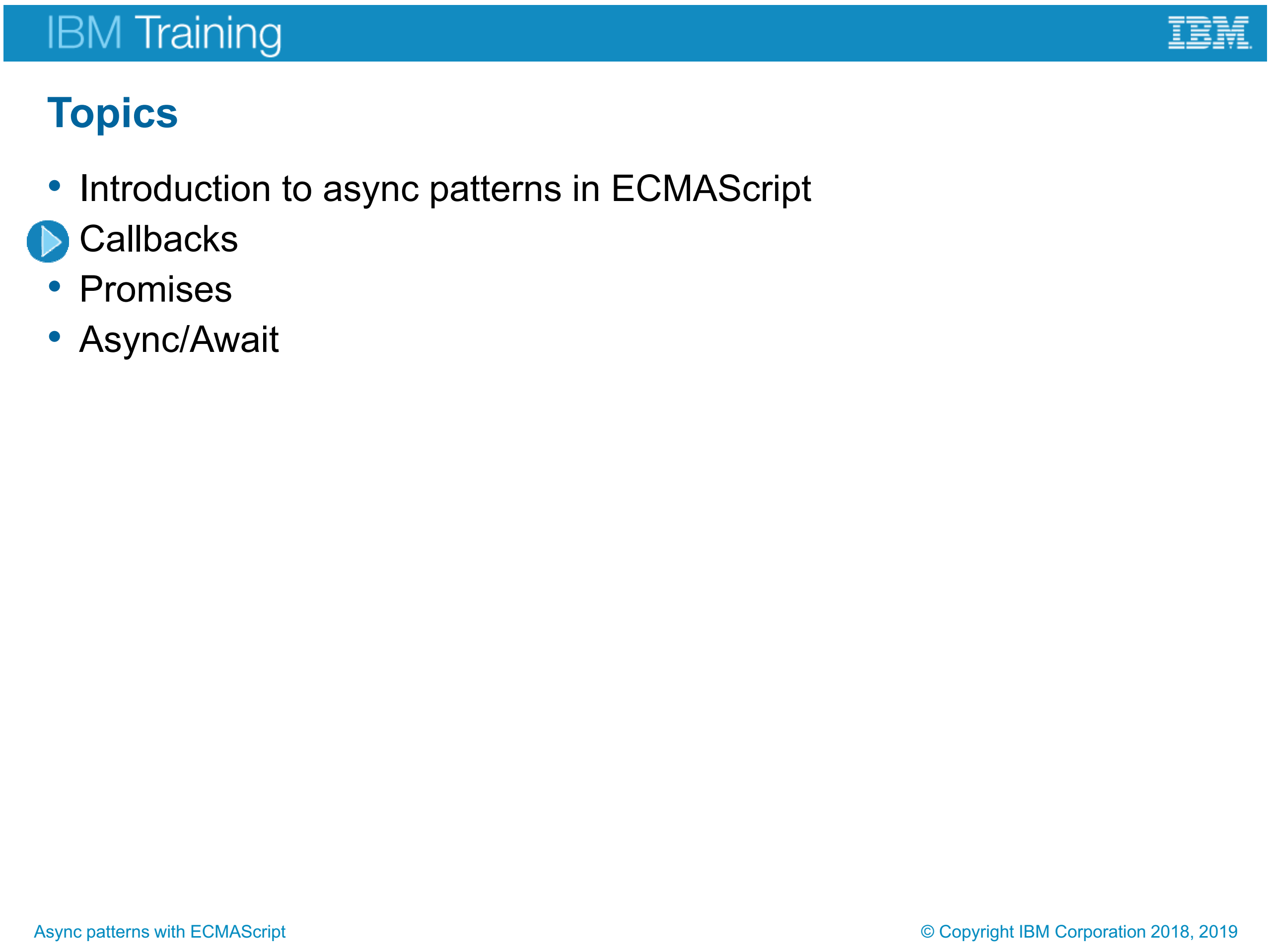
IBM Training

Callbacks

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-7. Callbacks



IBM Training

Topics

- Introduction to async patterns in ECMAScript
- ▶ Callbacks
- Promises
- Async/Await

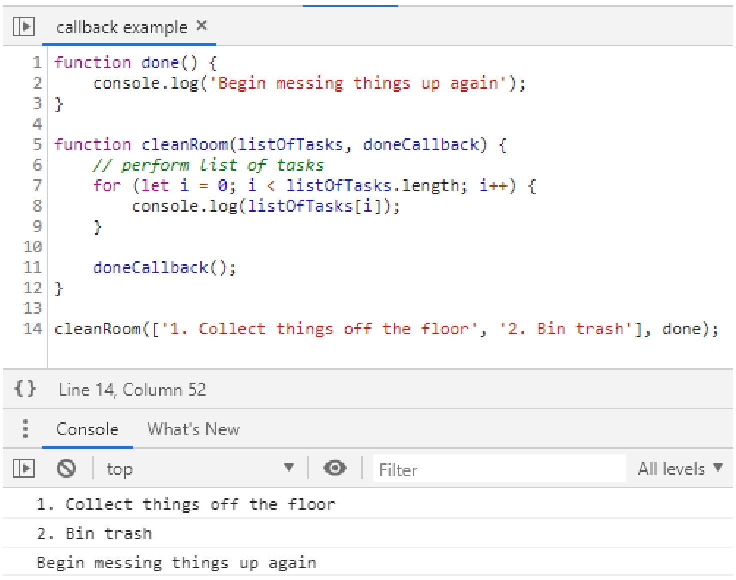
Async patterns with ECMAScript

Figure 4-8. Topics

IBM Training

Callbacks

- A *callback* function is passed as an argument into another function, which is invoked inside the other function to complete an action.
- A callback is a language feature of ES5 (2009).



callback example

```javascript
function done() {
    console.log('Begin messing things up again');
}

function cleanRoom(listOfTasks, doneCallback) {
    // perform list of tasks
    for (let i = 0; i < listOfTasks.length; i++) {
        console.log(listOfTasks[i]);
    }

    doneCallback();
}

cleanRoom(['1. Collect things off the floor', '2. Bin trash'], done);
```

Line 14, Column 52

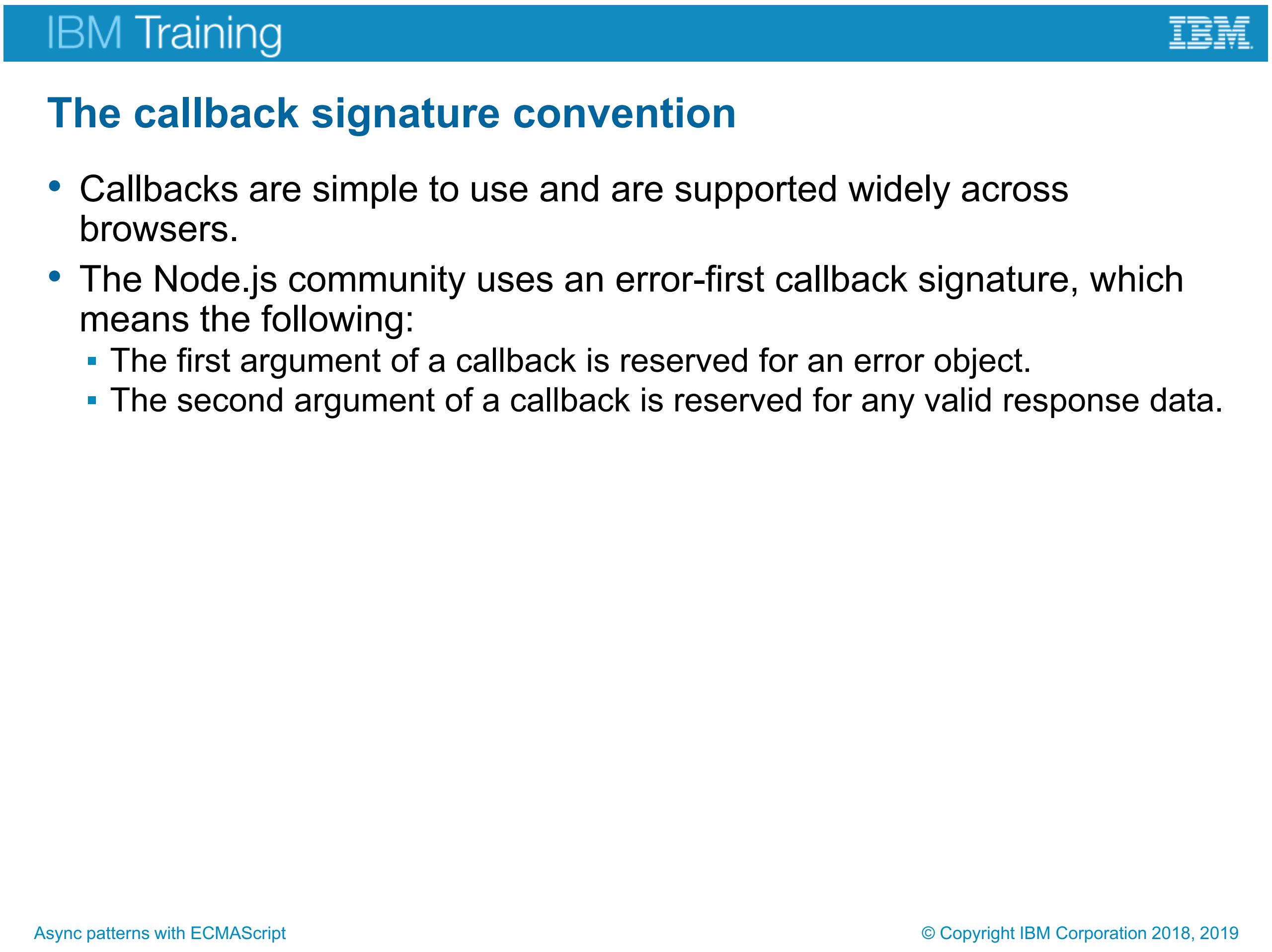
```
1. Collect things off the floor
2. Bin trash
Begin messing things up again
```

Async patterns with ECMAScript

Figure 4-9. Callbacks

- A *callback* function is passed as an argument into another function, which is invoked inside the other function to complete an action.
- A callback is a language feature of ES5 (2009).

The code example shows the function **cleanRoom** receiving an array of strings as the first argument and a function as the second argument. The **cleanRoom** function returns only after the **done** function returns.


IBM Training

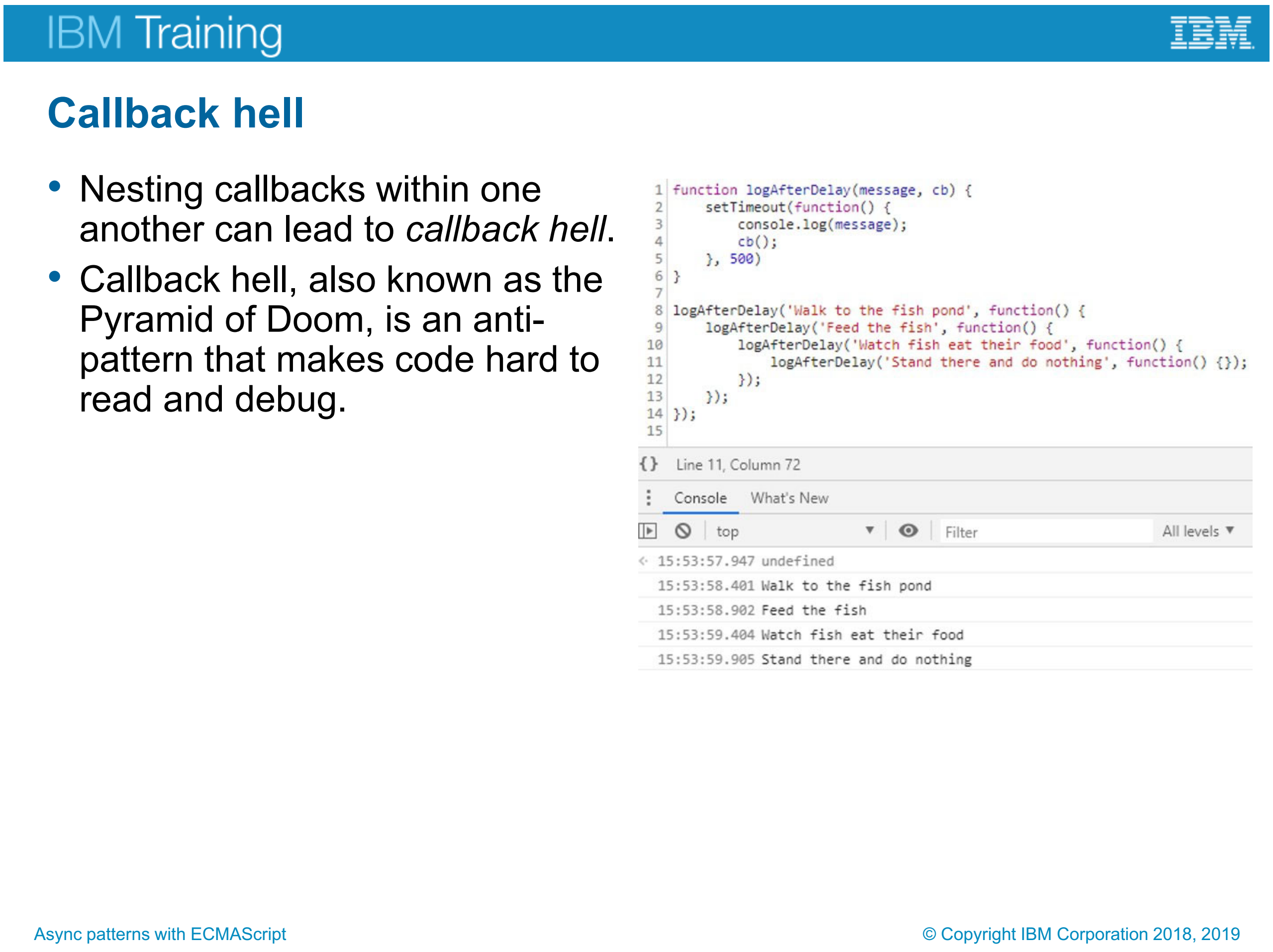
The callback signature convention

- Callbacks are simple to use and are supported widely across browsers.
- The Node.js community uses an error-first callback signature, which means the following:
  - The first argument of a callback is reserved for an error object.
  - The second argument of a callback is reserved for any valid response data.

Async patterns with ECMAScript © Copyright IBM Corporation 2018, 2019

Figure 4-10. The callback signature convention

- Callbacks are simple to use and are supported widely across browsers.
- The Node.js community uses an error-first callback signature, which means the following:
  - The first argument of a callback is reserved for an error object.
  - The second argument of a callback is reserved for any valid response data.



IBM Training

Callback hell

- Nesting callbacks within one another can lead to *callback hell*.
- Callback hell, also known as the Pyramid of Doom, is an anti-pattern that makes code hard to read and debug.

```
function logAfterDelay(message, cb) {
    setTimeout(function() {
        console.log(message);
        cb();
    }, 500)
}

logAfterDelay('Walk to the fish pond', function() {
    logAfterDelay('Feed the fish', function() {
        logAfterDelay('Watch fish eat their food', function() {
            logAfterDelay('Stand there and do nothing', function() {});
        });
    });
});
```

Line 11, Column 72

Console   What's New

top   Filter   All levels

```
< 15:53:57.947 undefined
  15:53:58.401 Walk to the fish pond
  15:53:58.902 Feed the fish
  15:53:59.404 Watch fish eat their food
  15:53:59.905 Stand there and do nothing
```

Async patterns with ECMAScript   © Copyright IBM Corporation 2018, 2019

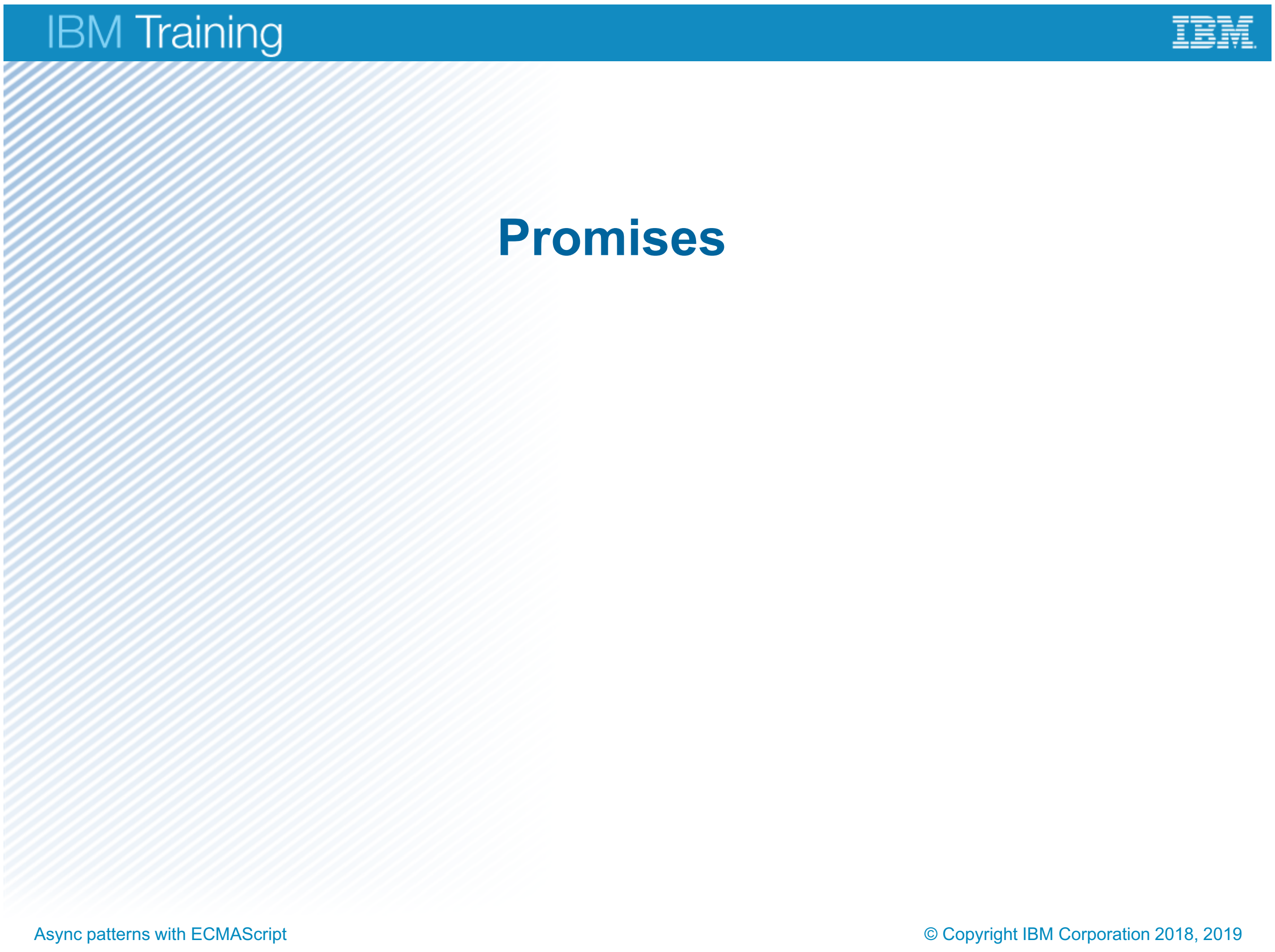
Figure 4-11. Callback hell

- Nesting callbacks within one another can lead to *callback hell*.
- Callback hell is required in cases where you want to ensure a specific sequence.
- Callback hell, also known as the Pyramid of Doom, is an anti-pattern that makes code hard to read and debug.

The example shows the **logAfterDelay** function, which receives a string of “Walk to the fish pond” as the first argument and a new instance of the **logAfterDelay** function as the second argument. As the number of nested functions grows, the code becomes harder to read.

The programmer is faced with the challenge of splitting code into separate functions to obey the single responsibility principle and for reusability, and balancing these items against the readability of the code.

4.3. Promises



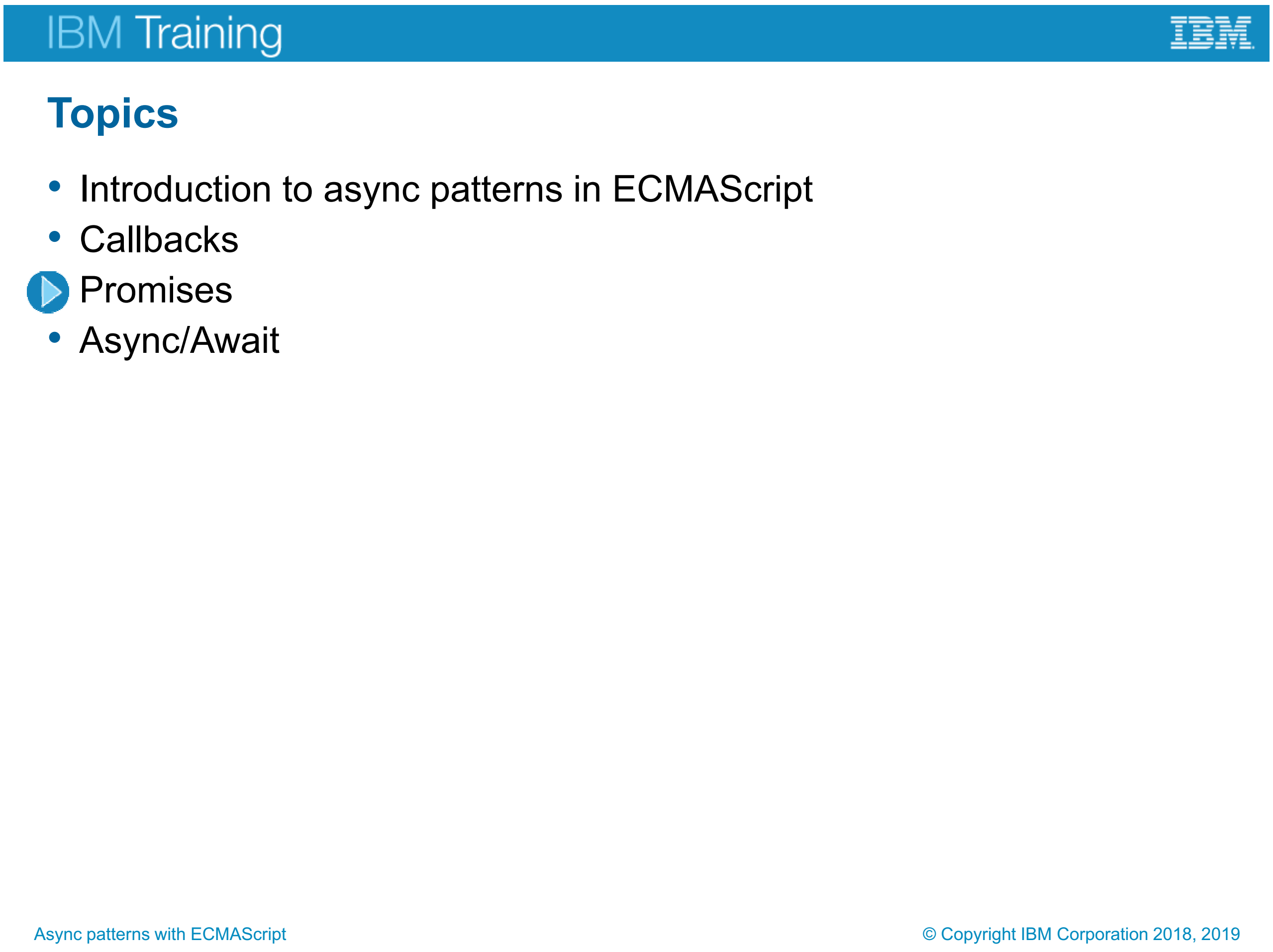
IBM Training

Promises

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-12. Promises



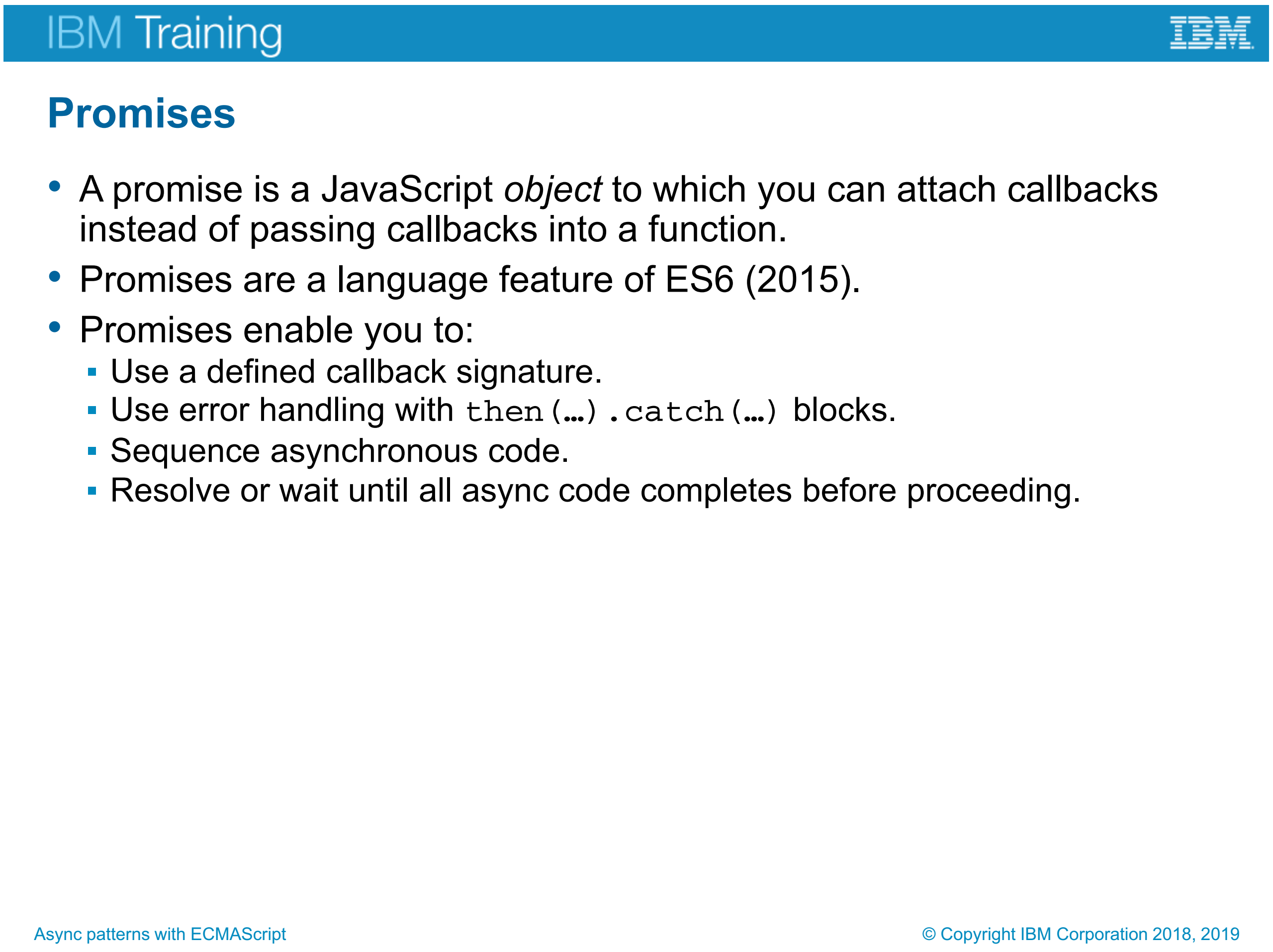
IBM Training

Topics

- Introduction to async patterns in ECMAScript
- Callbacks
- Promises
- Async/Await

Async patterns with ECMAScript

Figure 4-13. Topics


IBM Training

Promises

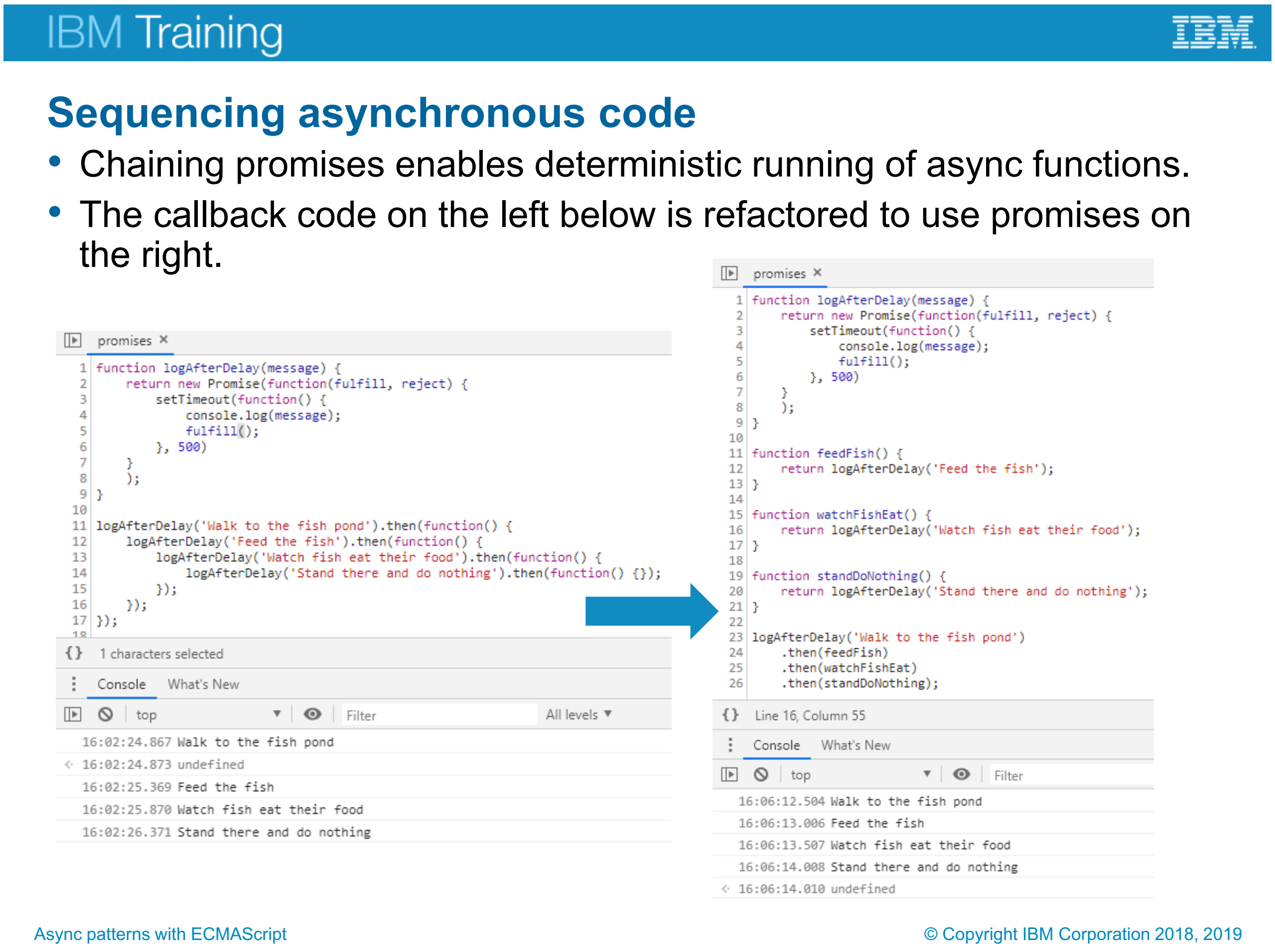
- A promise is a JavaScript *object* to which you can attach callbacks instead of passing callbacks into a function.
- Promises are a language feature of ES6 (2015).
- Promises enable you to:
  - Use a defined callback signature.
  - Use error handling with `then(…).catch(…)` blocks.
  - Sequence asynchronous code.
  - Resolve or wait until all async code completes before proceeding.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-14. Promises

- A promise is a JavaScript *object* to which you can attach callbacks instead of passing callbacks into a function.
- Promises are a language feature of ES6 (2015).
- Promises enable you to:
  - Use a defined callback signature.
  - Use error handling with "`then(…).catch(…)`" blocks.
  - Sequence asynchronous code.
  - Resolve or wait until all async code completes before proceeding.



IBM Training

Sequencing asynchronous code

- Chaining promises enables deterministic running of async functions.
- The callback code on the left below is refactored to use promises on the right.

promises ×

```
function logAfterDelay(message) {
    return new Promise(function(fulfill, reject) {
        setTimeout(function() {
            console.log(message);
            fulfill();
        }, 500)
    }
    );
}

logAfterDelay('Walk to the fish pond').then(function() {
    logAfterDelay('Feed the fish').then(function() {
        logAfterDelay('Watch fish eat their food').then(function() {
            logAfterDelay('Stand there and do nothing').then(function() {});
        });
    });
});
```

1 characters selected

Console | What's New

```
16:02:24.867 Walk to the fish pond
16:02:24.873 undefined
16:02:25.369 Feed the fish
16:02:25.870 Watch fish eat their food
16:02:26.371 Stand there and do nothing
```

promises ×

```
function logAfterDelay(message) {
    return new Promise(function(fulfill, reject) {
        setTimeout(function() {
            console.log(message);
            fulfill();
        }, 500)
    }
    );
}

function feedFish() {
    return logAfterDelay('Feed the fish');
}

function watchFishEat() {
    return logAfterDelay('Watch fish eat their food');
}

function standDoNothing() {
    return logAfterDelay('Stand there and do nothing');
}

logAfterDelay('Walk to the fish pond')
    .then(feedFish)
    .then(watchFishEat)
    .then(standDoNothing);
```

Line 16, Column 55

Console | What's New

```
16:06:12.504 Walk to the fish pond
16:06:13.006 Feed the fish
16:06:13.507 Watch fish eat their food
16:06:14.008 Stand there and do nothing
16:06:14.010 undefined
```

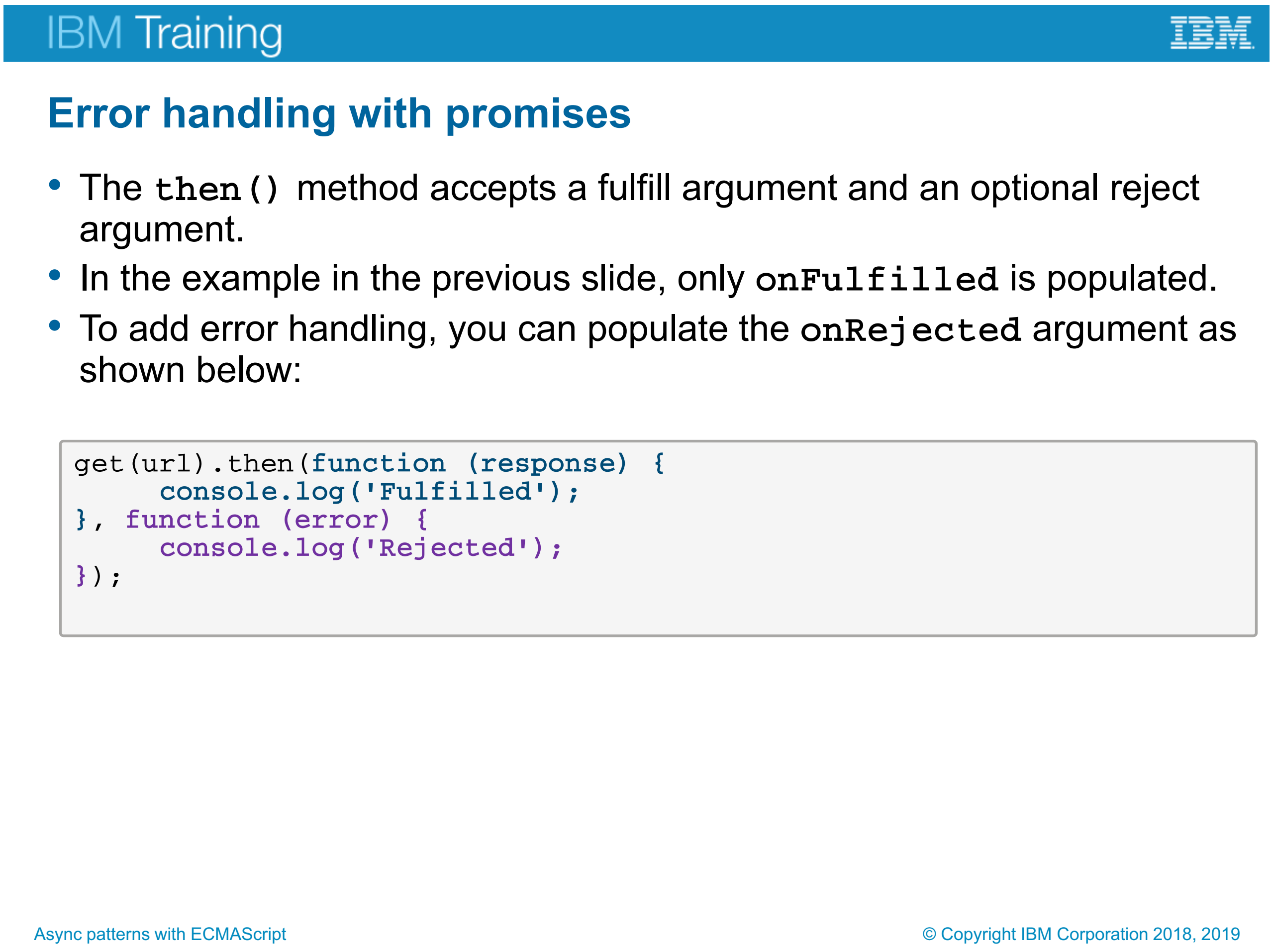
Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-15. Sequencing asynchronous code

- Chaining promises enables deterministic running of async functions. Deterministic running means that the results are ordered.
- The callback code on the left below is refactored to use promises on the right.
- The **then()** method returns a promise. Its formal signature is ".then(onFulfilled [, onRejected])", where "[, onRejected]" is optional.
- In the example on the right, only **onFulfilled** runs.

Using the power of promises, which were introduced in ES6, you can refactor your code from callback hell to a much easier to understand sequence of ".then()" methods to ensure that each callback is resolved in a deterministic manner.
The code on the left does not run in a deterministic manner (observe the output). Each time the code is run, the output might not be the same given the same input.
The code on the right calls each function in order by using **then()** to wait for the promise to be resolved before calling the next chained function (running from top to bottom). The chaining of promises ensures that the output is always the same for the same given input.



IBM Training

Error handling with promises

- The `then()` method accepts a fulfill argument and an optional reject argument.
- In the example in the previous slide, only `onFulfilled` is populated.
- To add error handling, you can populate the `onRejected` argument as shown below:

```
get(url).then(function (response) {
     console.log('Fulfilled');
}, function (error) {
     console.log('Rejected');
});
```

Async patterns with ECMAScript

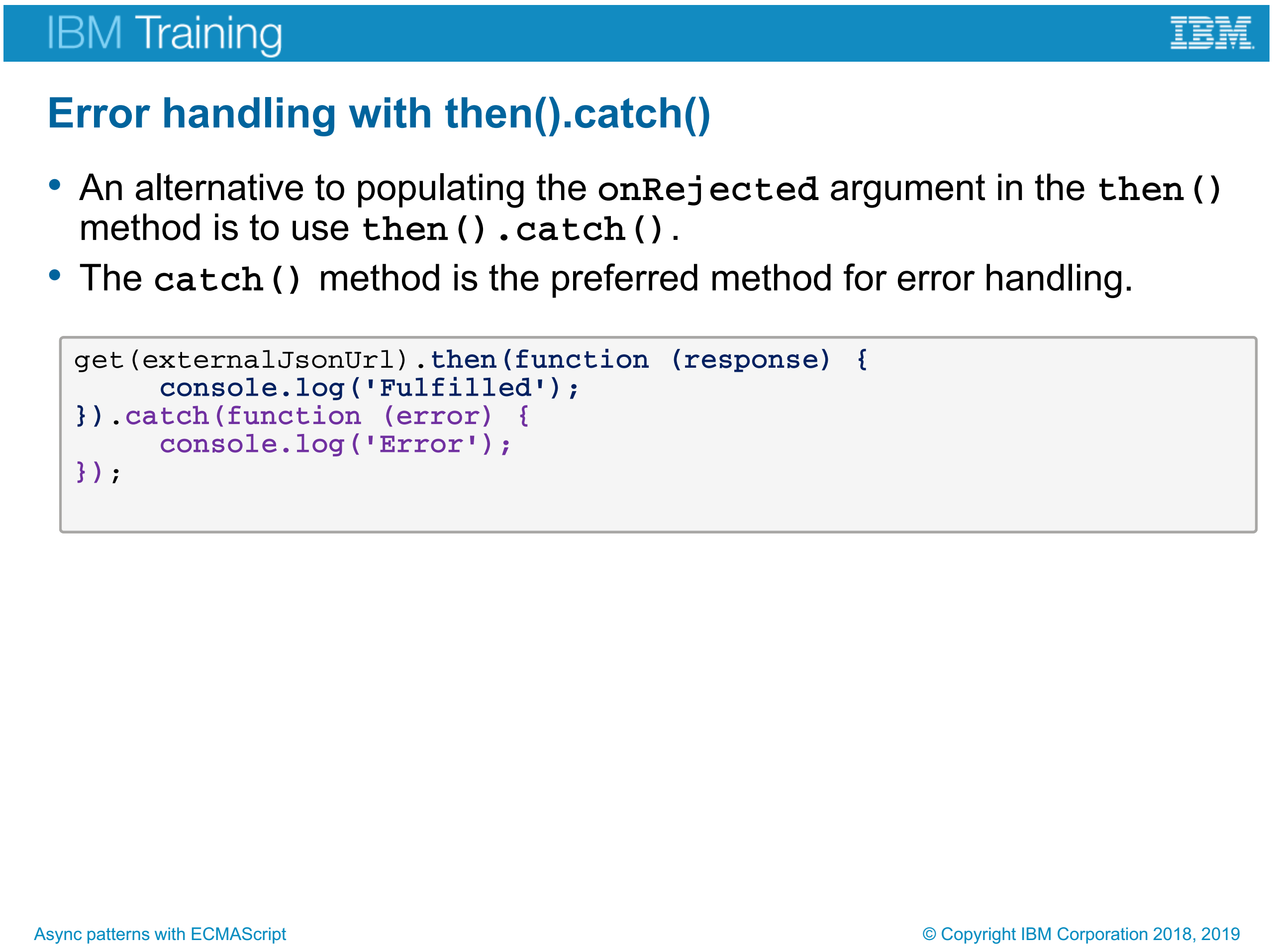
© Copyright IBM Corporation 2018, 2019

Figure 4-16. Error handling with promises

- The **then()** method accepts a fulfill argument and an optional reject argument.
- In the example in the previous slide, only **onFulfilled** is populated.
- To add error handling, you can populate the **onRejected** argument.

For more information, go to the following website:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then



IBM Training

Error handling with then().catch()

- An alternative to populating the `onRejected` argument in the `then()` method is to use `then().catch()`.
- The `catch()` method is the preferred method for error handling.

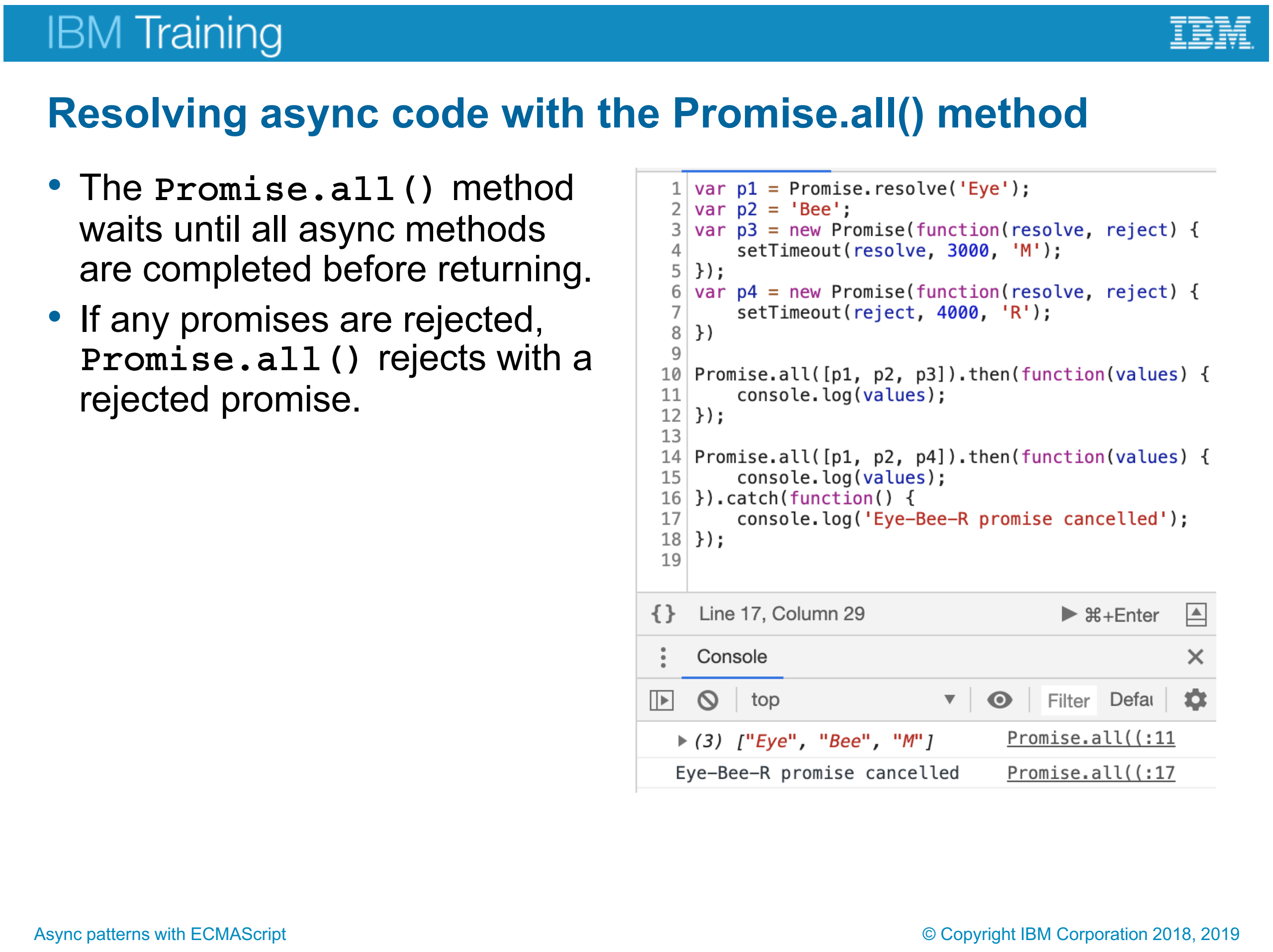
```
get(externalJsonUrl).then(function (response) {
      console.log('Fulfilled');
}).catch(function (error) {
      console.log('Error');
});
```

Async patterns with ECMAScript © Copyright IBM Corporation 2018, 2019

Figure 4-17. Error handling with then().catch()

- An alternative to populating the **onRejected** argument in the **then()** method is to use **then().catch()**.
- The **catch()** method is the preferred method for error handling.

Using the catch() method is preferred because the function is the same as using an **onRejected** argument. However, the code readability is much better when you use **catch()**.



IBM Training

Resolving async code with the Promise.all() method

- The **`Promise.all()`** method waits until all async methods are completed before returning.
- If any promises are rejected, **`Promise.all()`** rejects with a rejected promise.

```
var p1 = Promise.resolve('Eye');
var p2 = 'Bee';
var p3 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 3000, 'M');
});
var p4 = new Promise(function(resolve, reject) {
    setTimeout(reject, 4000, 'R');
})

Promise.all([p1, p2, p3]).then(function(values) {
    console.log(values);
});

Promise.all([p1, p2, p4]).then(function(values) {
    console.log(values);
}).catch(function() {
    console.log('Eye-Bee-R promise cancelled');
});
```

Line 17, Column 29 ⌘+Enter

Console

top Filter Defau

```
▸ (3) ["Eye", "Bee", "M"]            Promise.all((:11
Eye-Bee-R promise cancelled          Promise.all((:17
```

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-18. Resolving async code with the Promise.all() method

- The **`Promise.all()`** method waits until all async methods are completed before returning.
- If any promises are rejected, **`Promise.all()`** rejects with a rejected promise.

Using **`Promise.all()`** allows efficient async functions to run in parallel but then wait at certain run points to allow deterministic running and run an atomic transaction.
An atomic transaction means the transaction (composed of smaller transactions) either fully runs or does not run at all. It looks like a single transaction, and there is never a partial run.

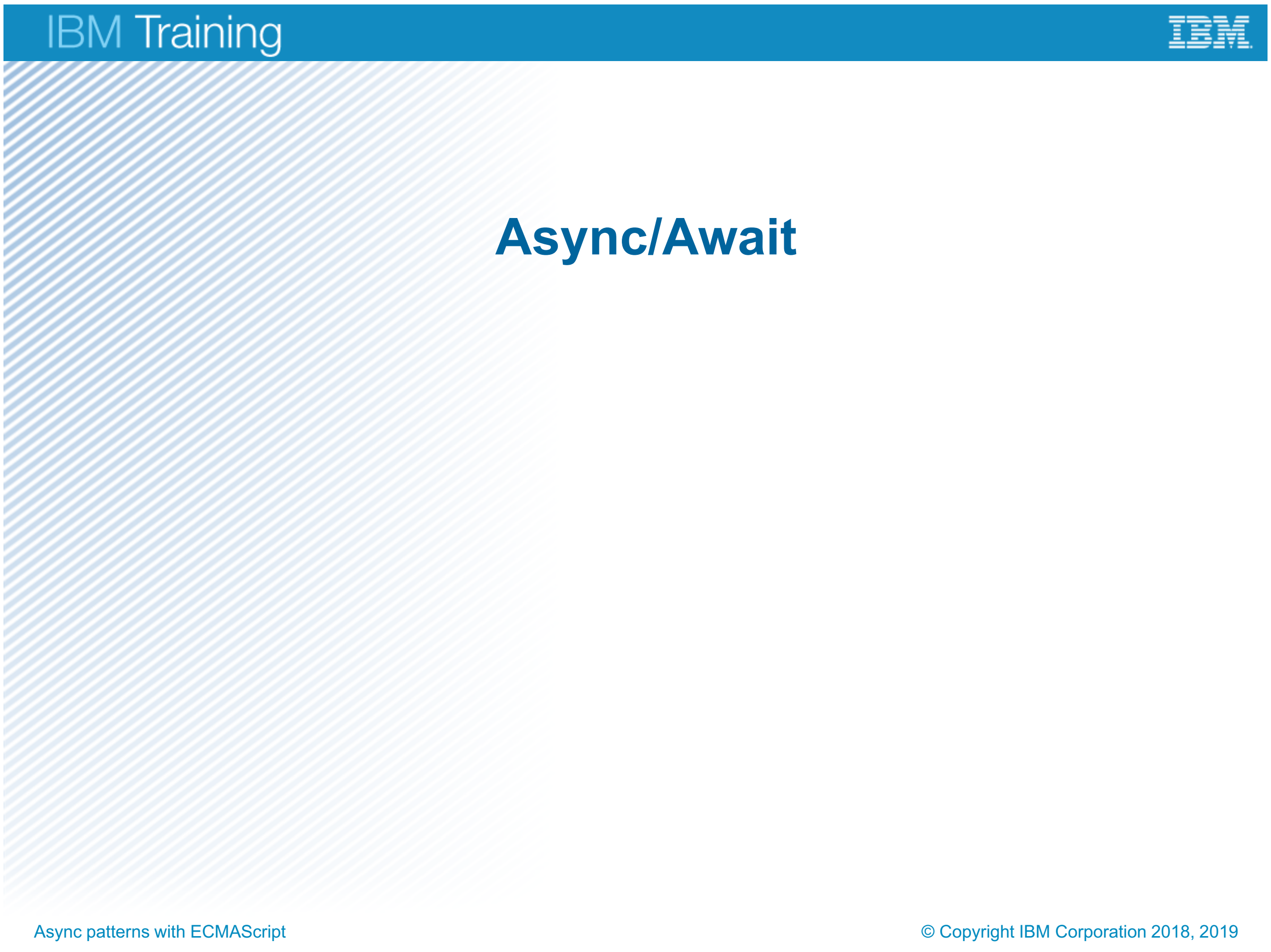
The code creates four vars that are labeled p1, p2, p3, and p4:

- p1 resolves a promise to return the string ‘Eye’, which is then assigned to this variable.
- p2 is assigned the string ‘Bee’.
- p3 resolves a new instance of a promise after 3000 milliseconds to assign the value of ‘R’.
- p4 rejects a new instance of a promise after 4000 milliseconds to assign the return value of the error, ‘promise cancelled’.

To learn more about Promises, go to the following website:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

4.4. Async/Await



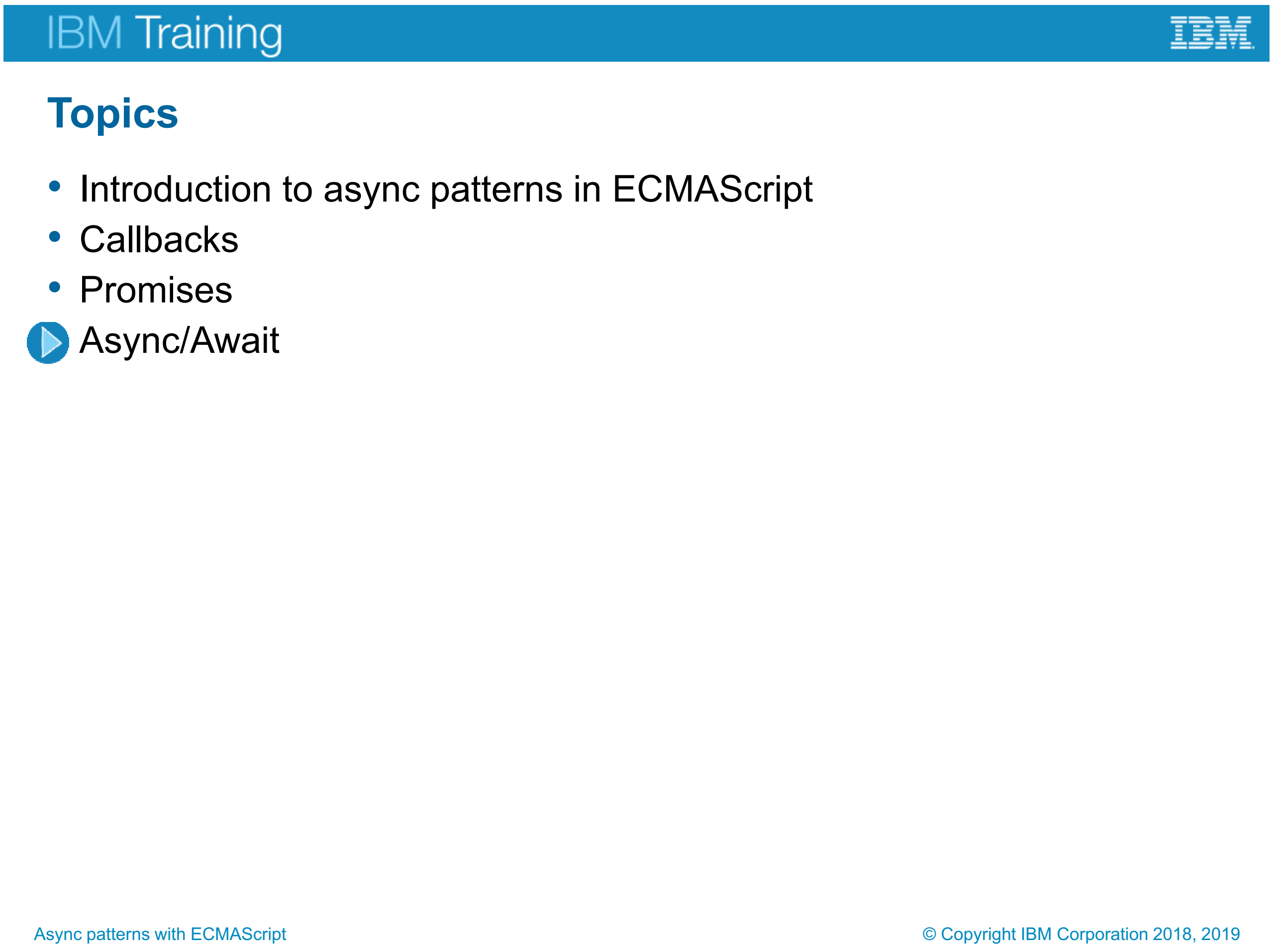
IBM Training

Async/Await

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-19. Async/Await



IBM Training

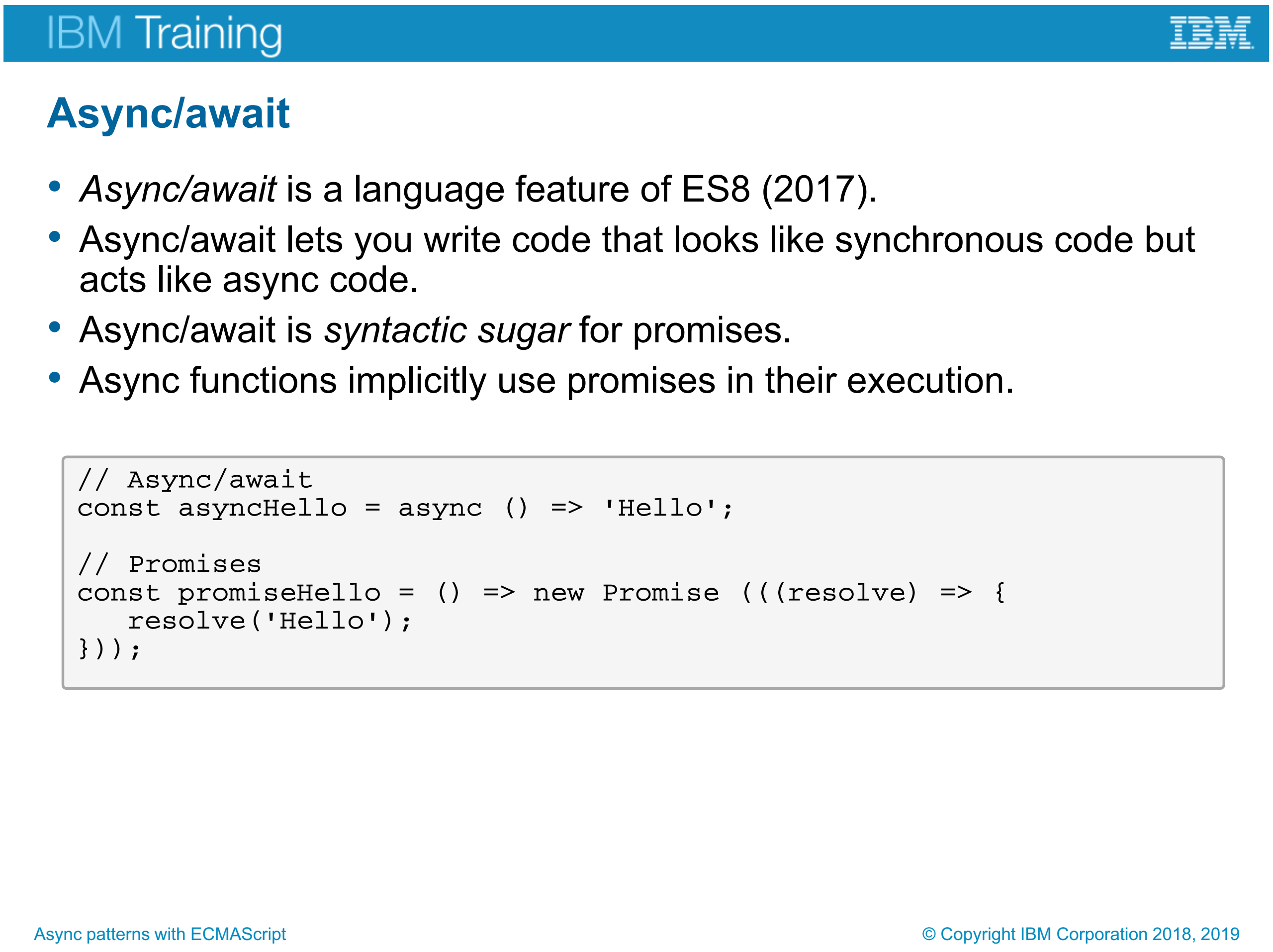
Topics

- Introduction to async patterns in ECMAScript
- Callbacks
- Promises
- Async/Await

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-20. Topics



IBM Training

Async/await

- *Async/await* is a language feature of ES8 (2017).
- Async/await lets you write code that looks like synchronous code but acts like async code.
- Async/await is *syntactic sugar* for promises.
- Async functions implicitly use promises in their execution.

```
// Async/await
const asyncHello = async () => 'Hello';

// Promises
const promiseHello = () => new Promise (((resolve) => {
   resolve('Hello');
}));
```

Async patterns with ECMAScript

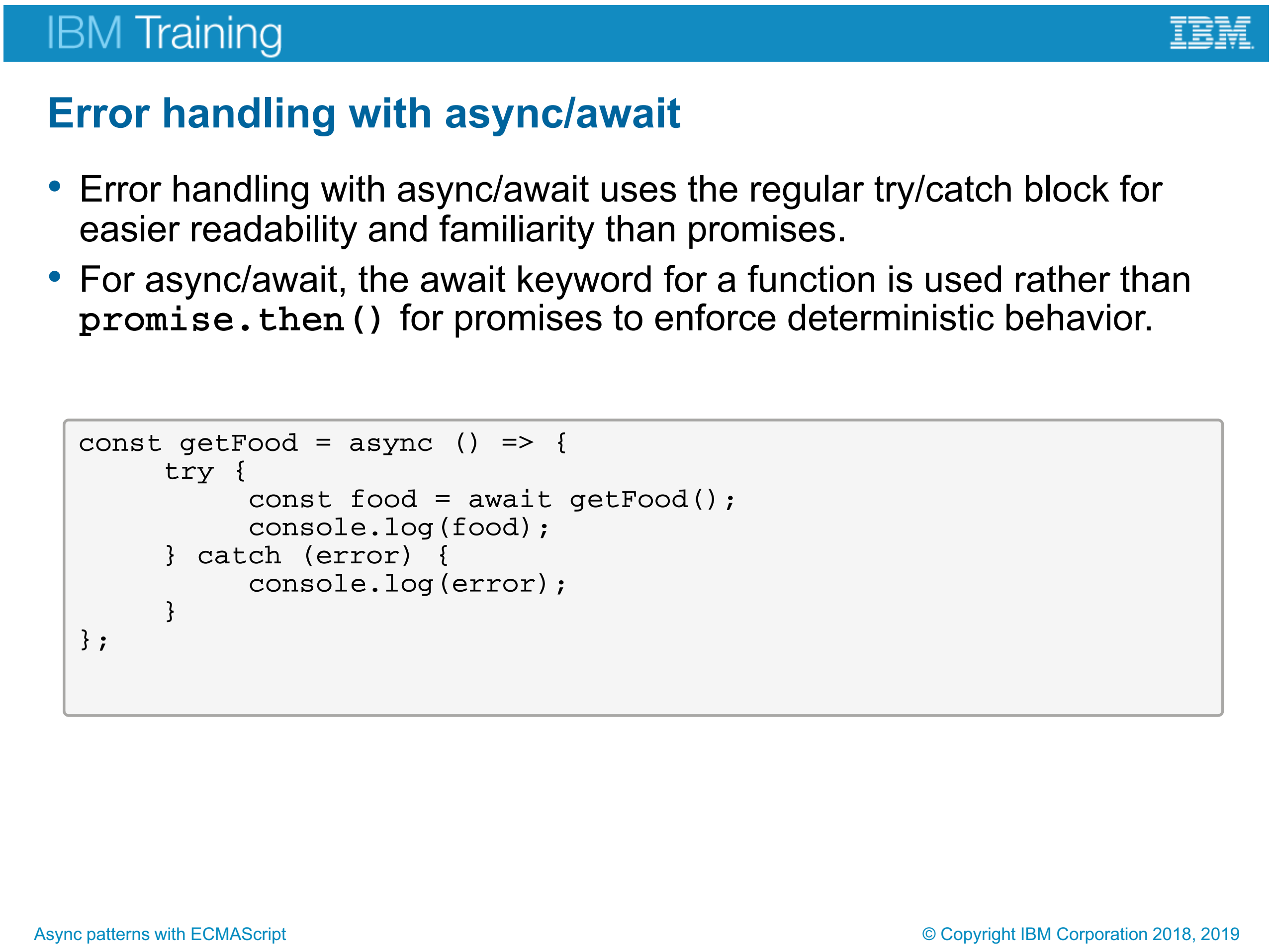
© Copyright IBM Corporation 2018, 2019

Figure 4-21. Async/await

- *Async/await* is a language feature of ES8 (2017).
- Async/await lets you write code that looks like synchronous code but acts like async code.
- Async/await is *syntactic sugar* for promises.
- Async functions implicitly use promises in their execution.

Syntactic sugar is a term to indicate that a different syntax is used to make code easier to read and write for the programmer. Importantly, syntactic sugar does not add new features.

The example shows the asynchronous pattern for async/await versus promises to return the string *hello*.



IBM Training

Error handling with async/await

- Error handling with async/await uses the regular try/catch block for easier readability and familiarity than promises.
- For async/await, the await keyword for a function is used rather than **`promise.then()`** for promises to enforce deterministic behavior.

```
const getFood = async () => {
      try {
            const food = await getFood();
            console.log(food);
      } catch (error) {
            console.log(error);
      }
};
```

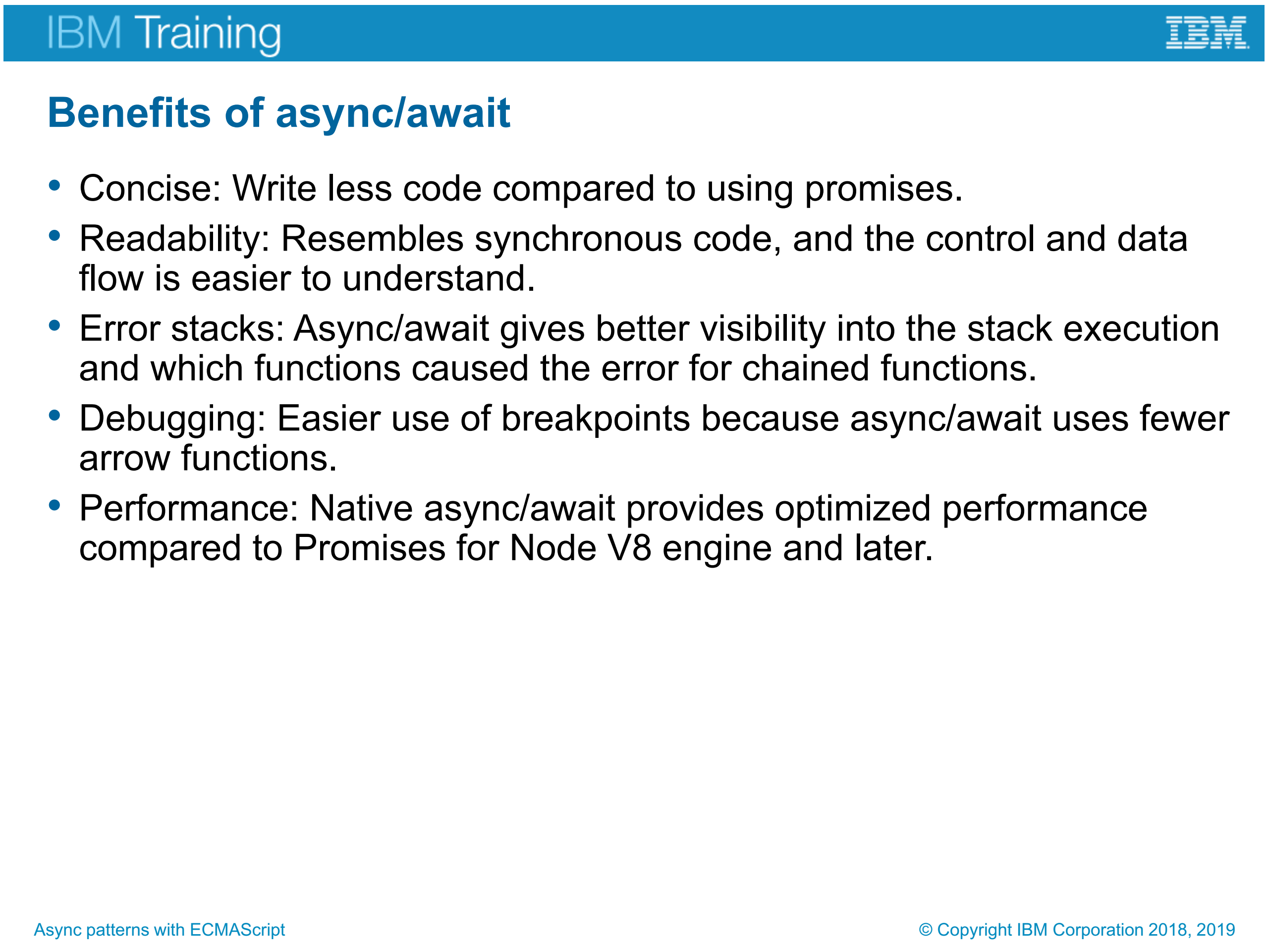
Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-22. Error handling with async/await

- Error handling with async/await uses the regular try/catch block for easier readability and familiarity than promises.
- For async/await, the **`await`** keyword for a function is used rather than **`promise.then()`** for promises to enforce deterministic behavior.

The await **myFunction()** suspends the running of the current function, but **promise.then(myFunction)** continues running the current function after **myFunction** is resolved. This is a significant difference leading to differences in performance and execution tracing.



IBM Training

Benefits of async/await

- Concise: Write less code compared to using promises.
- Readability: Resembles synchronous code, and the control and data flow is easier to understand.
- Error stacks: Async/await gives better visibility into the stack execution and which functions caused the error for chained functions.
- Debugging: Easier use of breakpoints because async/await uses fewer arrow functions.
- Performance: Native async/await provides optimized performance compared to Promises for Node V8 engine and later.

Async patterns with ECMAScript

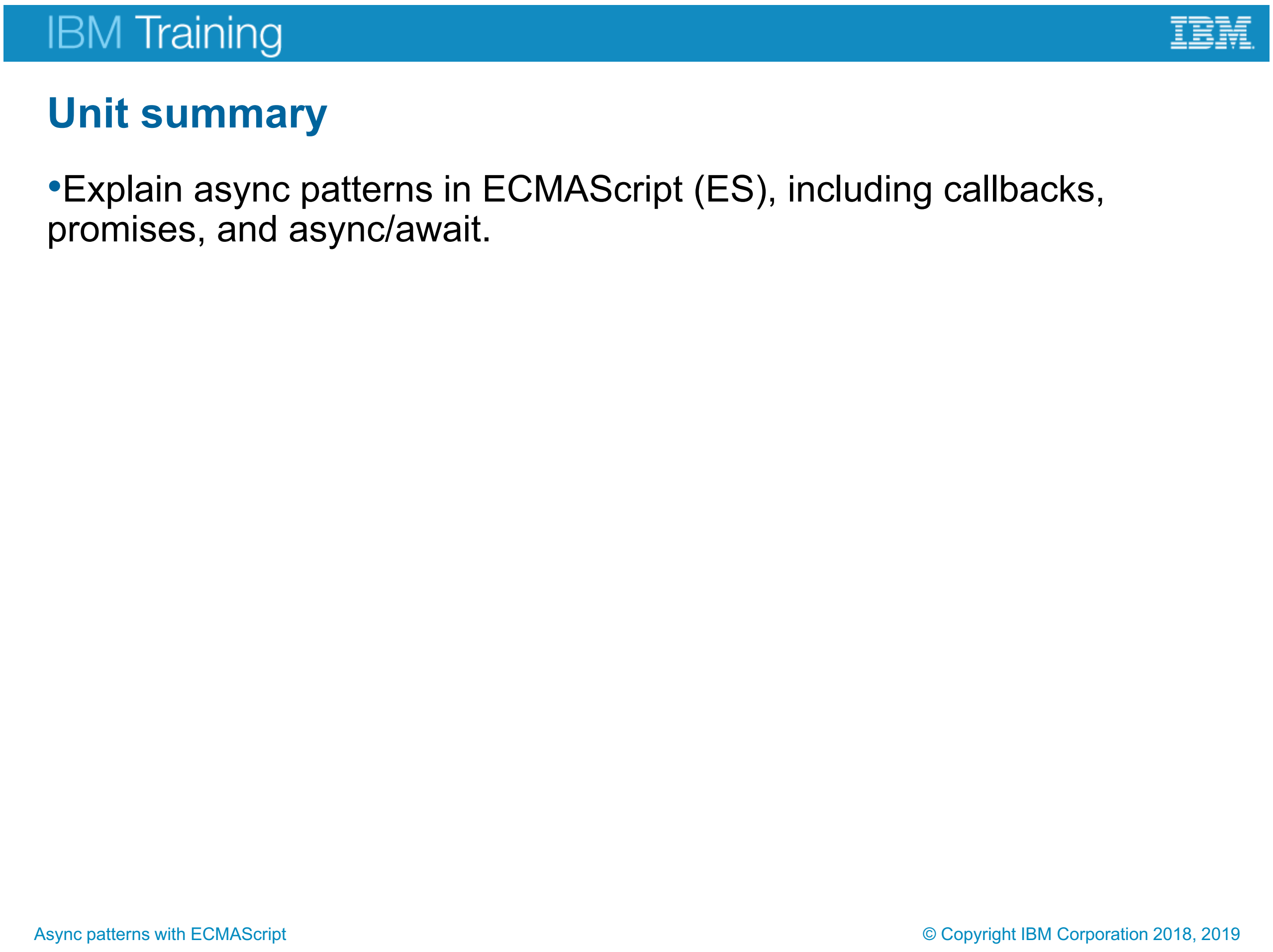
© Copyright IBM Corporation 2018, 2019

Figure 4-23. Benefits of async/await

- Concise: Write less code compared to using promises.
- Readability: Resembles synchronous code, and the control and data flow is easier to understand.
- Error Stacks: Async/await gives better visibility into the stack execution and which functions caused the error for chained functions.
- Debugging: Easier use of breakpoints because async/await uses fewer arrow functions.
- Performance: Native async/await provides optimized performance compared to Promises for Node V8 engine and later.

Native async/await indicates that a transpiler was not used to convert the ES8 objects into a previous version of ES, and that the JavaScript engine supported and ran the ES8 objects directly. To learn more about async/await, go to the following website:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function



IBM Training

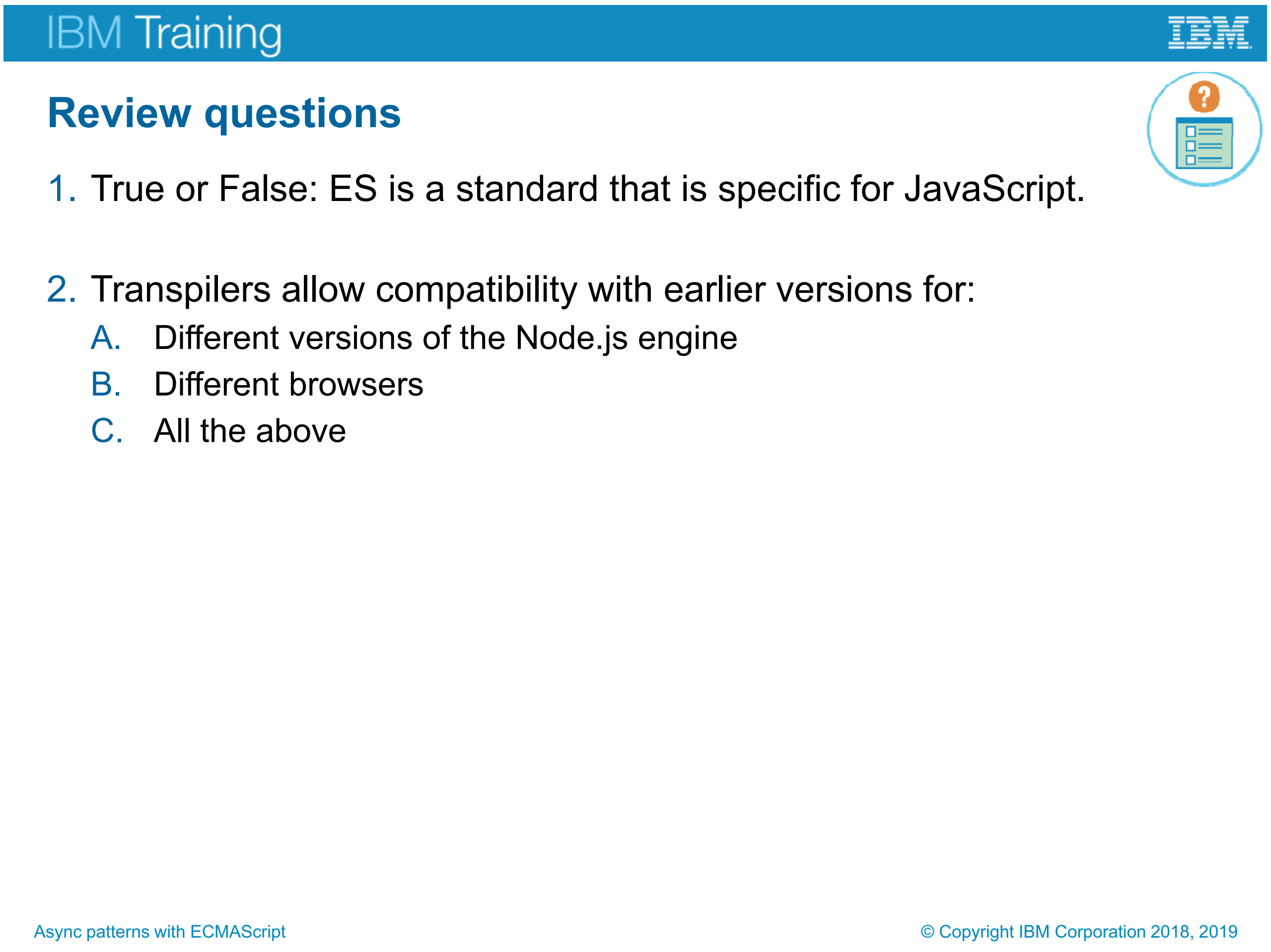
Unit summary

- Explain async patterns in ECMAScript (ES), including callbacks, promises, and async/await.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-24. Unit summary



IBM Training

Review questions

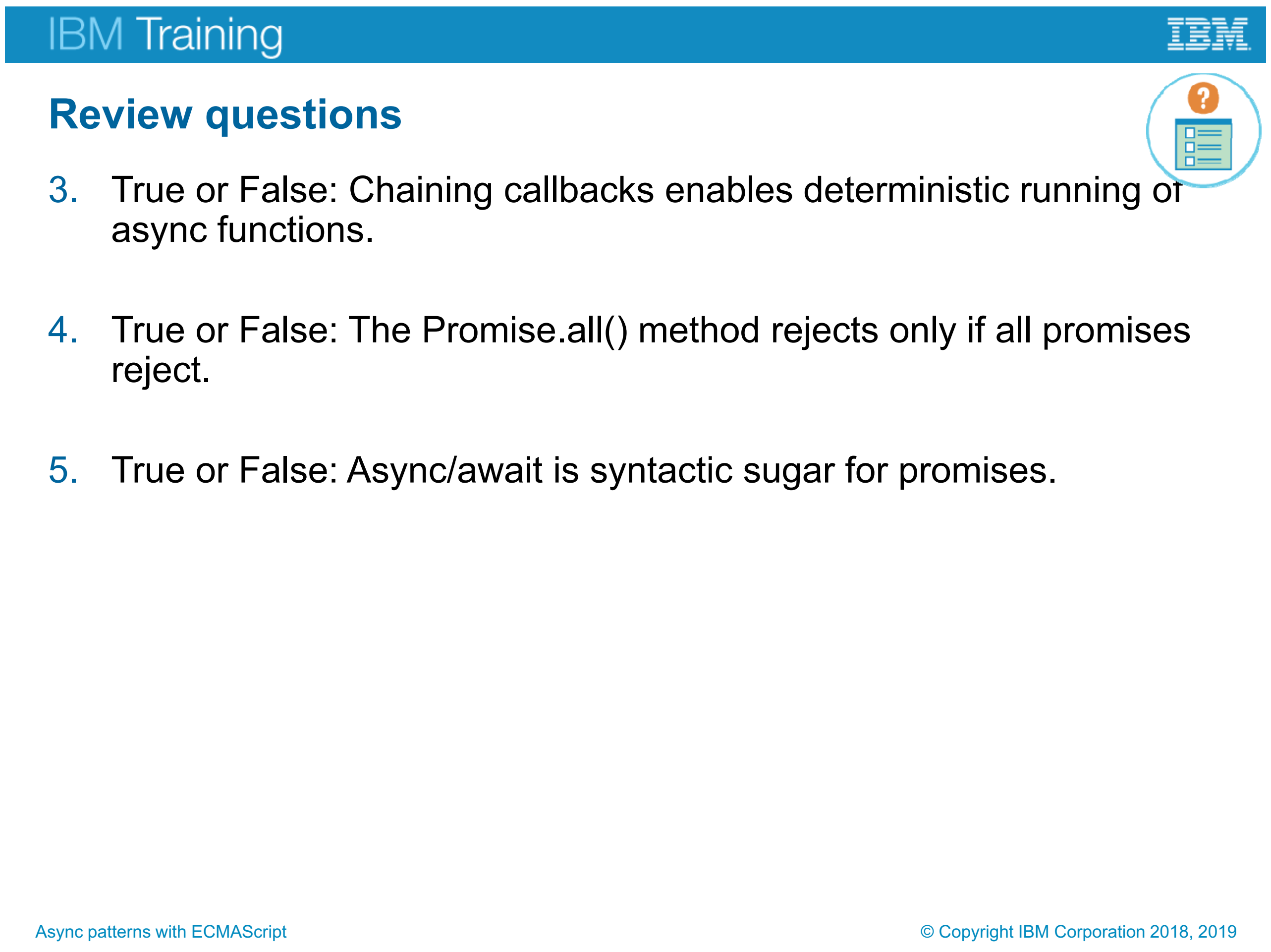
1. True or False: ES is a standard that is specific for JavaScript.

2. Transpilers allow compatibility with earlier versions for:
   A. Different versions of the Node.js engine
   B. Different browsers
   C. All the above

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-25. Review questions



IBM Training

Review questions

3. True or False: Chaining callbacks enables deterministic running of async functions.

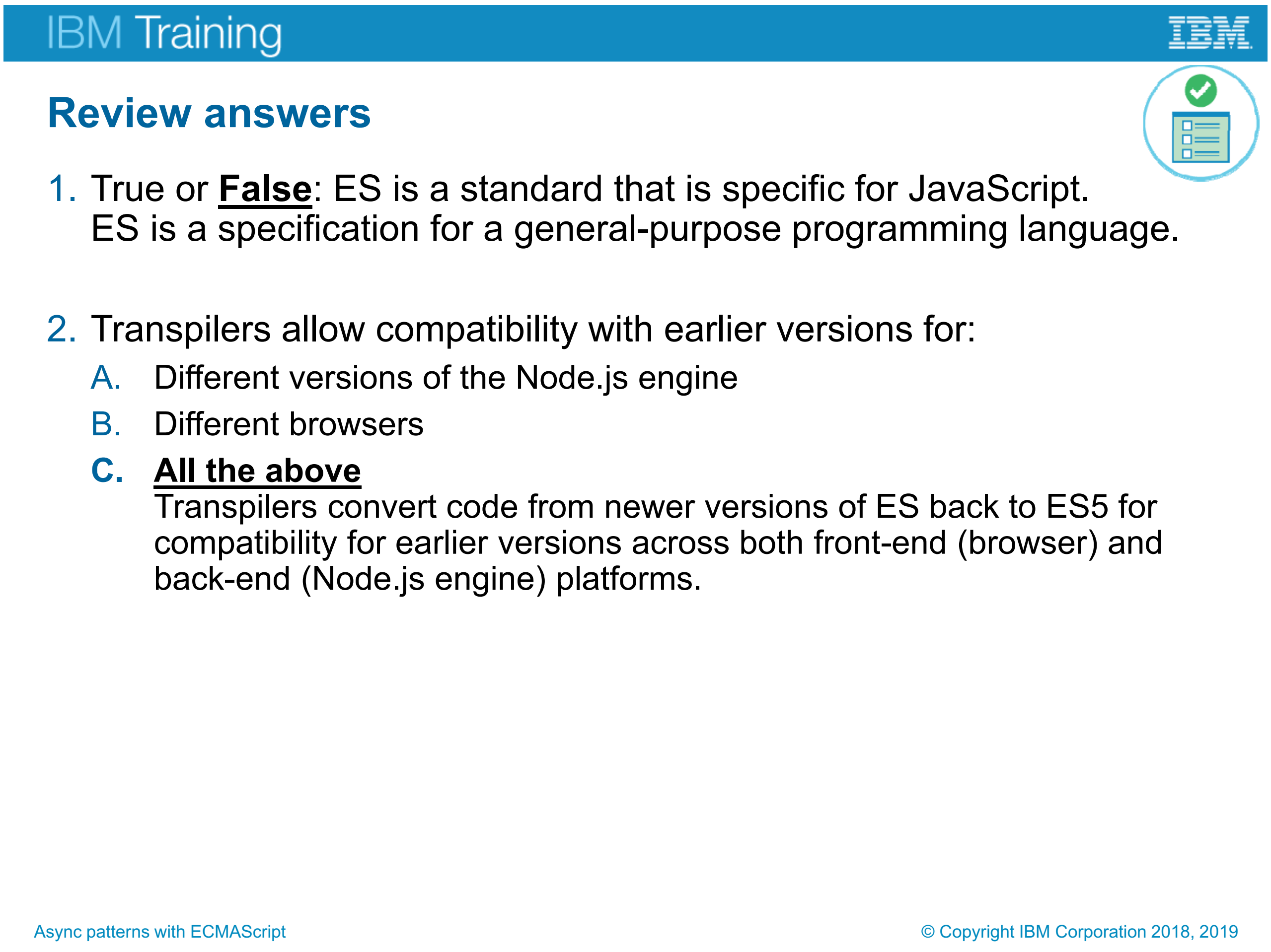
4. True or False: The Promise.all() method rejects only if all promises reject.

5. True or False: Async/await is syntactic sugar for promises.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-26. Review questions



IBM Training

Review answers

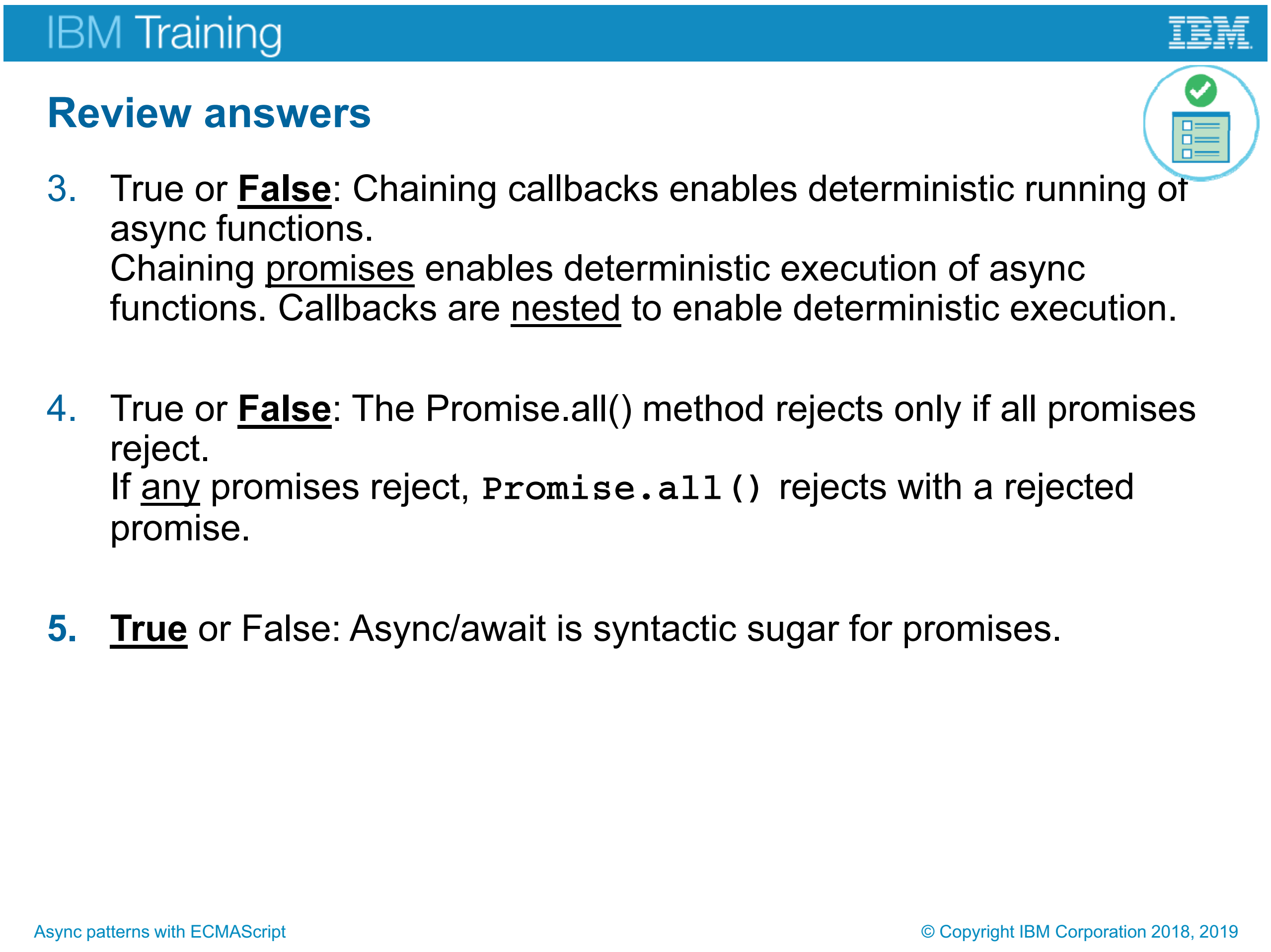
1. True or **<u>False</u>**: ES is a standard that is specific for JavaScript.
   ES is a specification for a general-purpose programming language.

2. Transpilers allow compatibility with earlier versions for:
   A. Different versions of the Node.js engine
   B. Different browsers
   C. <u>All the above</u>
   Transpilers convert code from newer versions of ES back to ES5 for compatibility for earlier versions across both front-end (browser) and back-end (Node.js engine) platforms.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-27. Review answers



IBM Training

Review answers

3. True or **<u>False</u>**: Chaining callbacks enables deterministic running of async functions.
   Chaining <u>promises</u> enables deterministic execution of async functions. Callbacks are <u>nested</u> to enable deterministic execution.

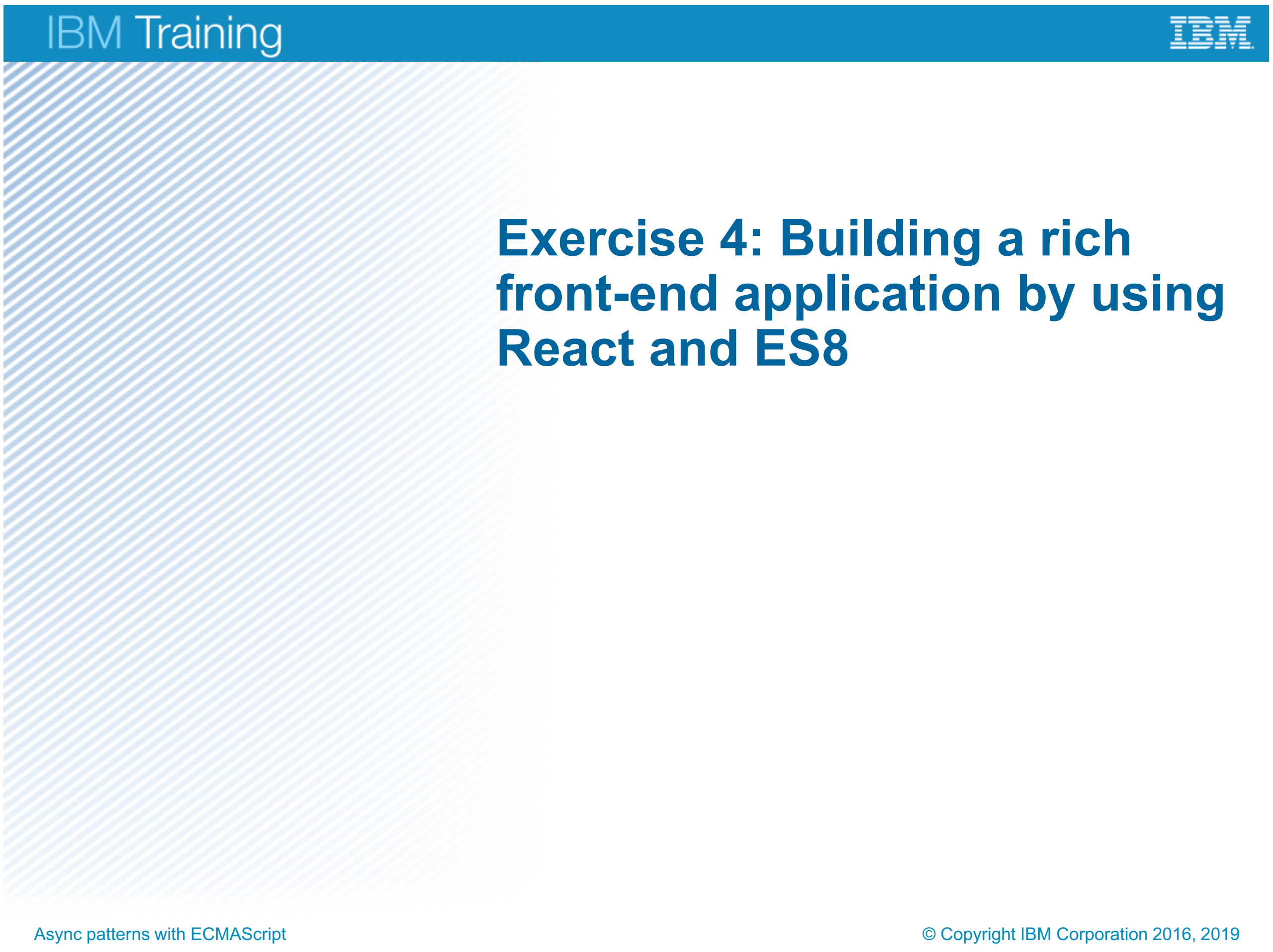
4. True or **<u>False</u>**: The Promise.all() method rejects only if all promises reject.
   If <u>any</u> promises reject, `Promise.all()` rejects with a rejected promise.

5. **<u>True</u>** or False: Async/await is syntactic sugar for promises.

Async patterns with ECMAScript

© Copyright IBM Corporation 2018, 2019

Figure 4-28. Review answers



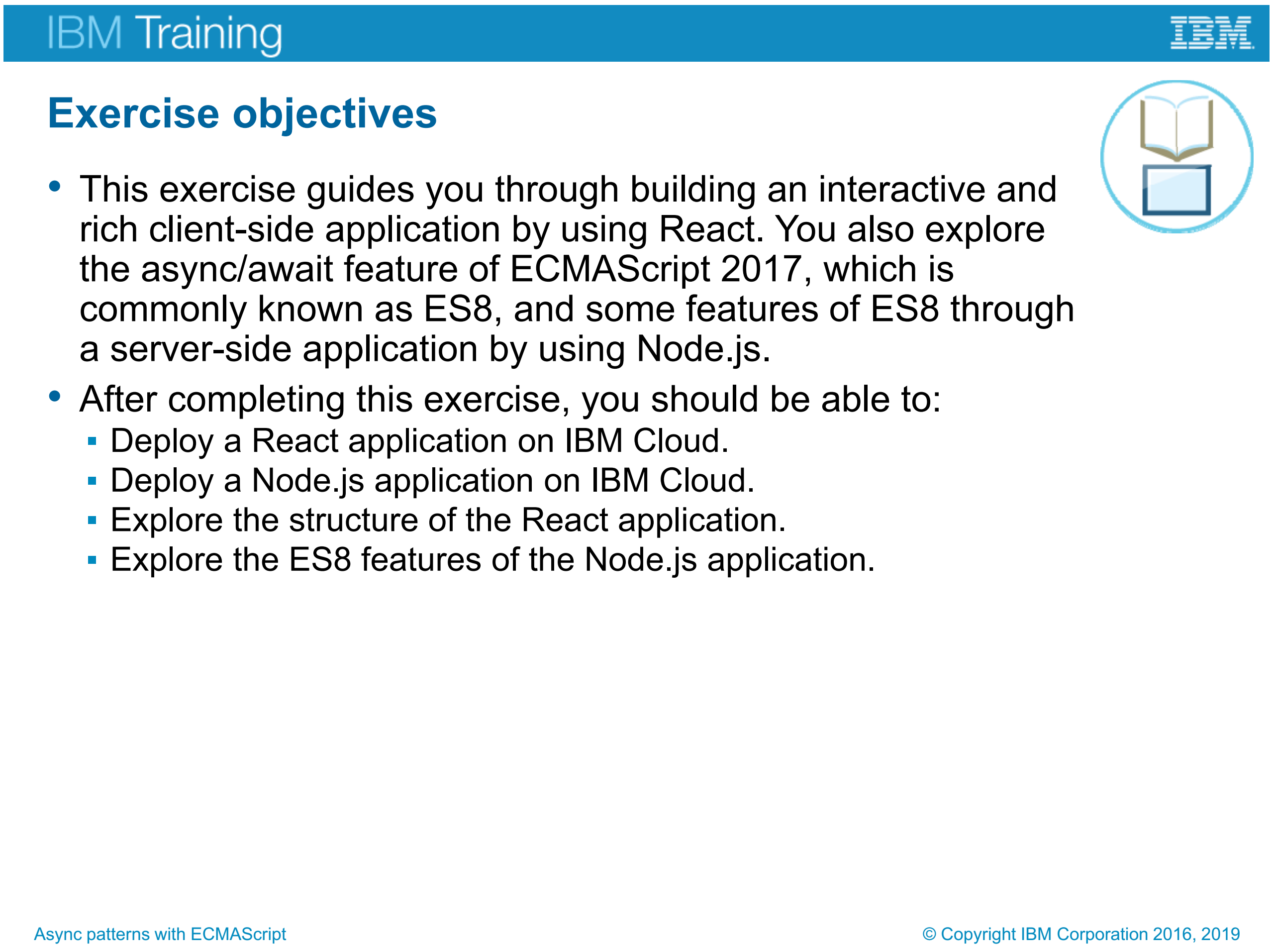
IBM Training

Exercise 4: Building a rich front-end application by using React and ES8

Async patterns with ECMAScript

© Copyright IBM Corporation 2016, 2019

Figure 4-29. Exercise 4: Building a rich front-end application by using React and ES8



IBM Training

Exercise objectives

- This exercise guides you through building an interactive and rich client-side application by using React. You also explore the async/await feature of ECMAScript 2017, which is commonly known as ES8, and some features of ES8 through a server-side application by using Node.js.
- After completing this exercise, you should be able to:
  - Deploy a React application on IBM Cloud.
  - Deploy a Node.js application on IBM Cloud.
  - Explore the structure of the React application.
  - Explore the ES8 features of the Node.js application.

Async patterns with ECMAScript

© Copyright IBM Corporation 2016, 2019

Figure 4-30. Exercise objectives

Unit 5. Building rich UI applications with React (Optional)

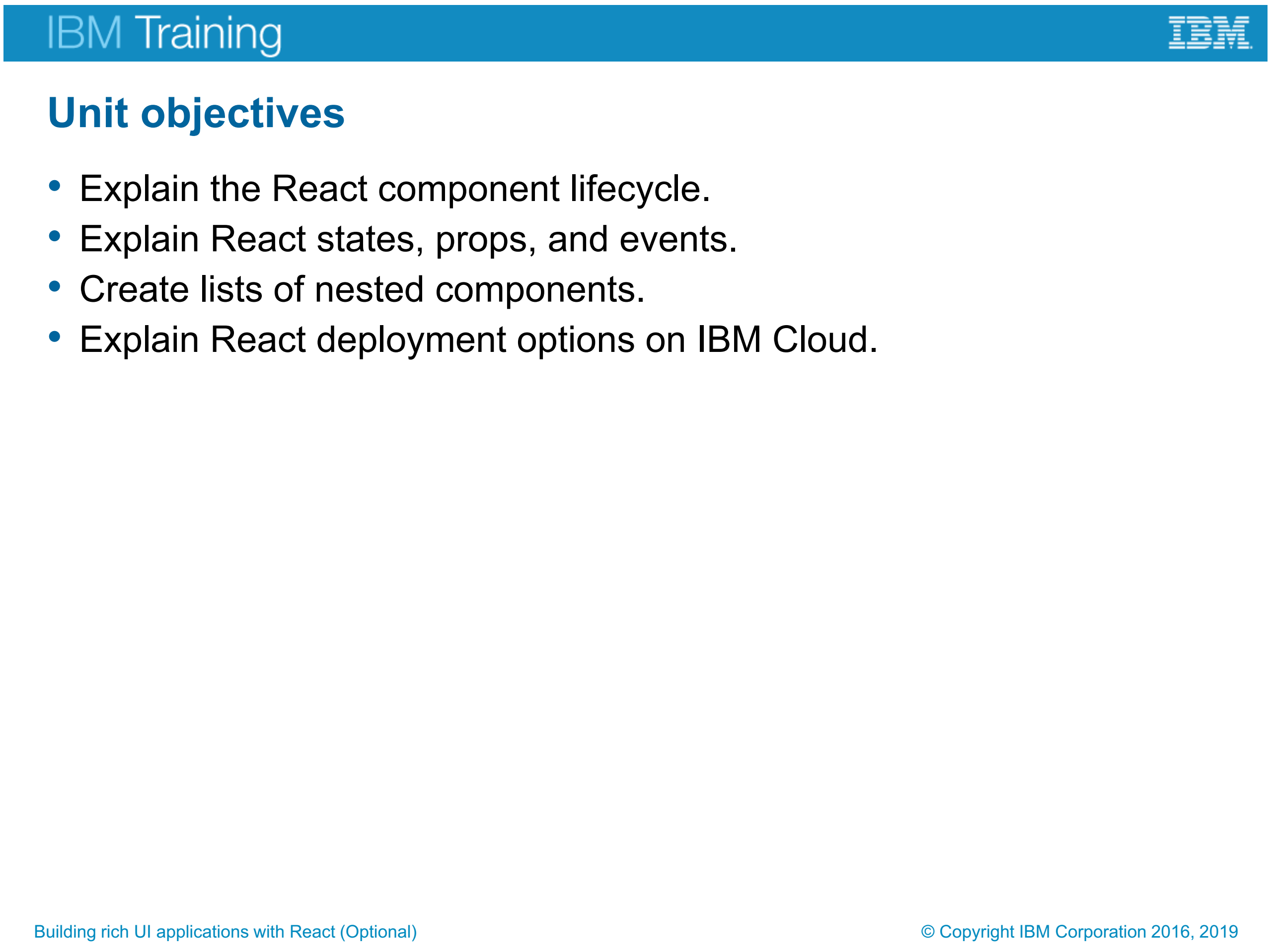
Estimated time

01:00

Overview

This unit describes async patterns in JavaScript. It introduces React and basic React concepts such as:

- Components
- Props
- State
- Events
- Component lifecycle
- Nested components
- Lists and keys



IBM Training

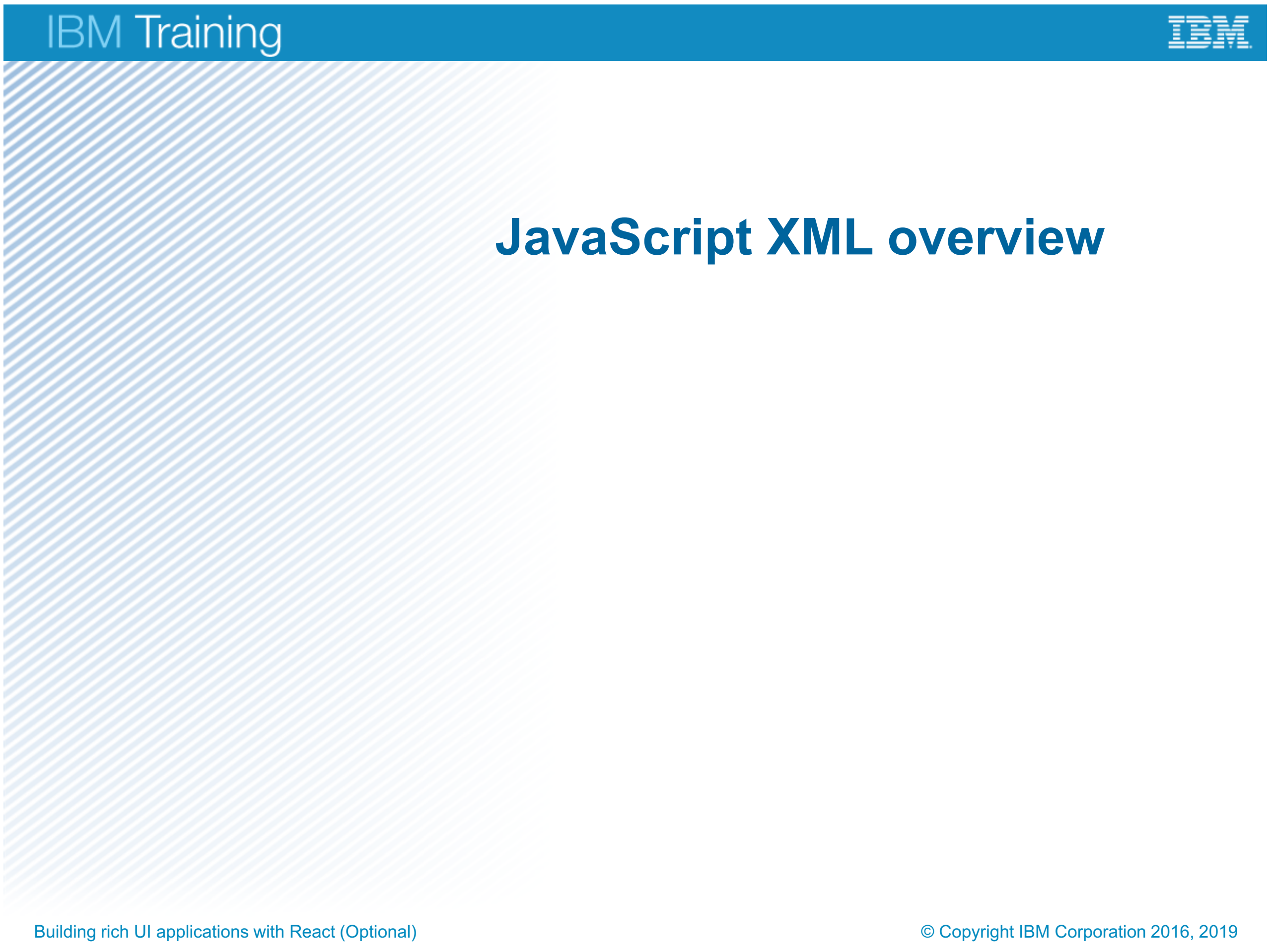
Unit objectives

- Explain the React component lifecycle.
- Explain React states, props, and events.
- Create lists of nested components.
- Explain React deployment options on IBM Cloud.

Building rich UI applications with React (Optional)

Figure 5-1. Unit objectives

5.1. JavaScript XML overview

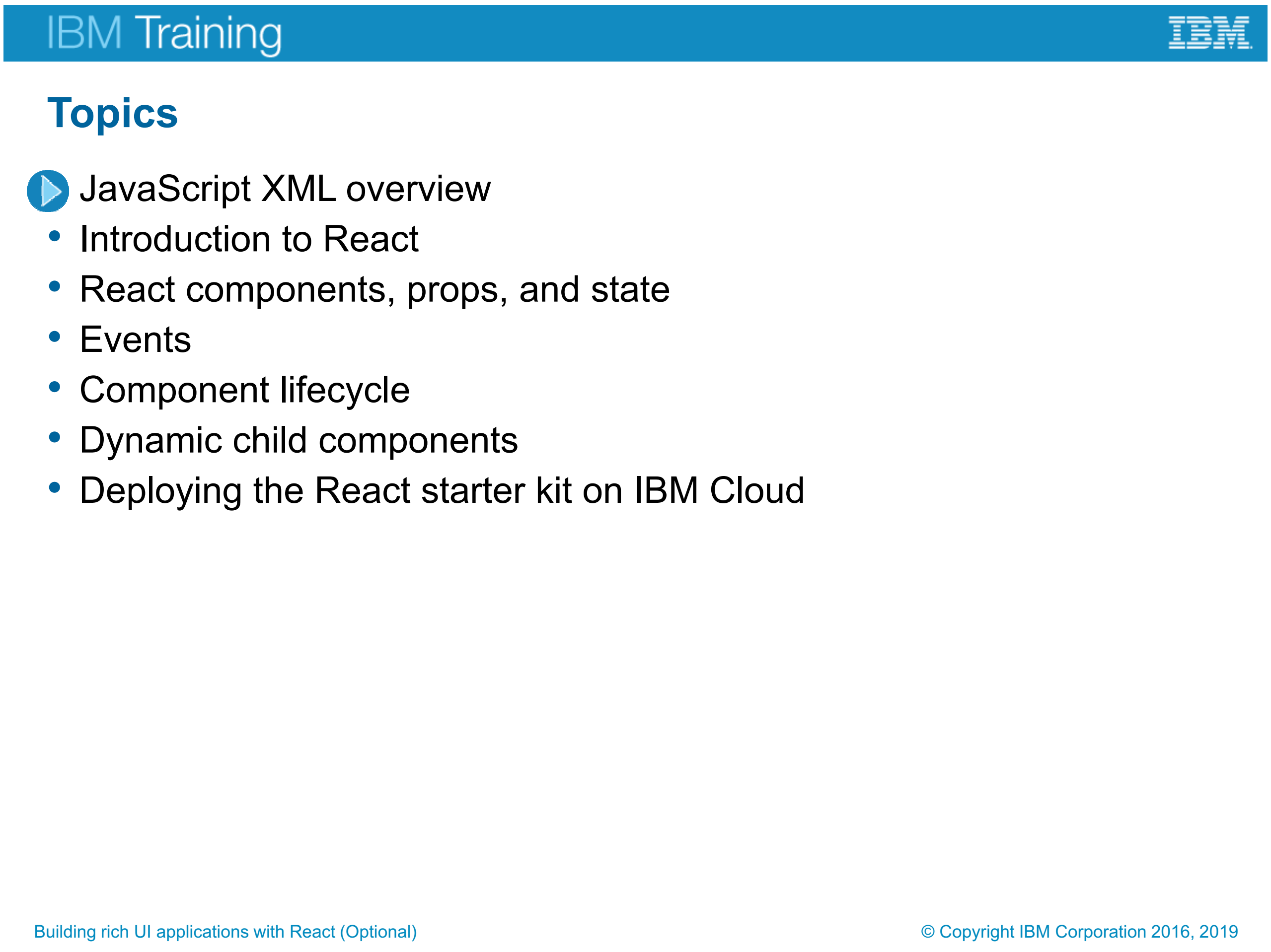

IBM Training

JavaScript XML overview

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-2. JavaScript XML overview



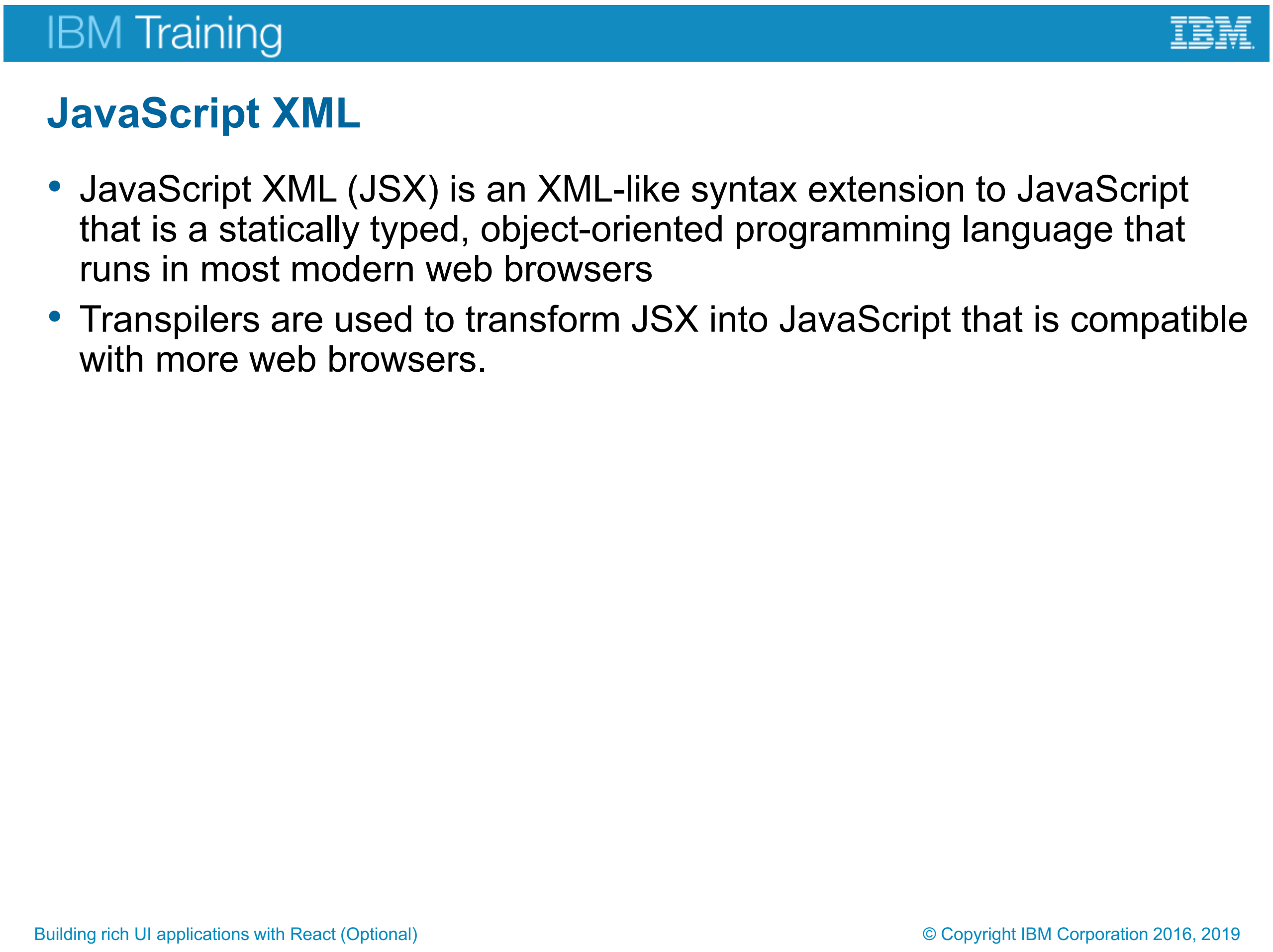
IBM Training

Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-3. Topics



IBM Training

JavaScript XML

- JavaScript XML (JSX) is an XML-like syntax extension to JavaScript that is a statically typed, object-oriented programming language that runs in most modern web browsers
- Transpilers are used to transform JSX into JavaScript that is compatible with more web browsers.

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-4. JavaScript XML

- JavaScript XML (JSX) is an XML-like syntax extension to JavaScript that is a statically typed, object-oriented programming language that runs in most modern web browsers
- Transpilers are used to transform JSX into JavaScript that is compatible with more web browsers.

Babel is a common transpiler that transforms JavaScript from JSX or ES2015+ into ES5.

For more information about Babel, see the following website:

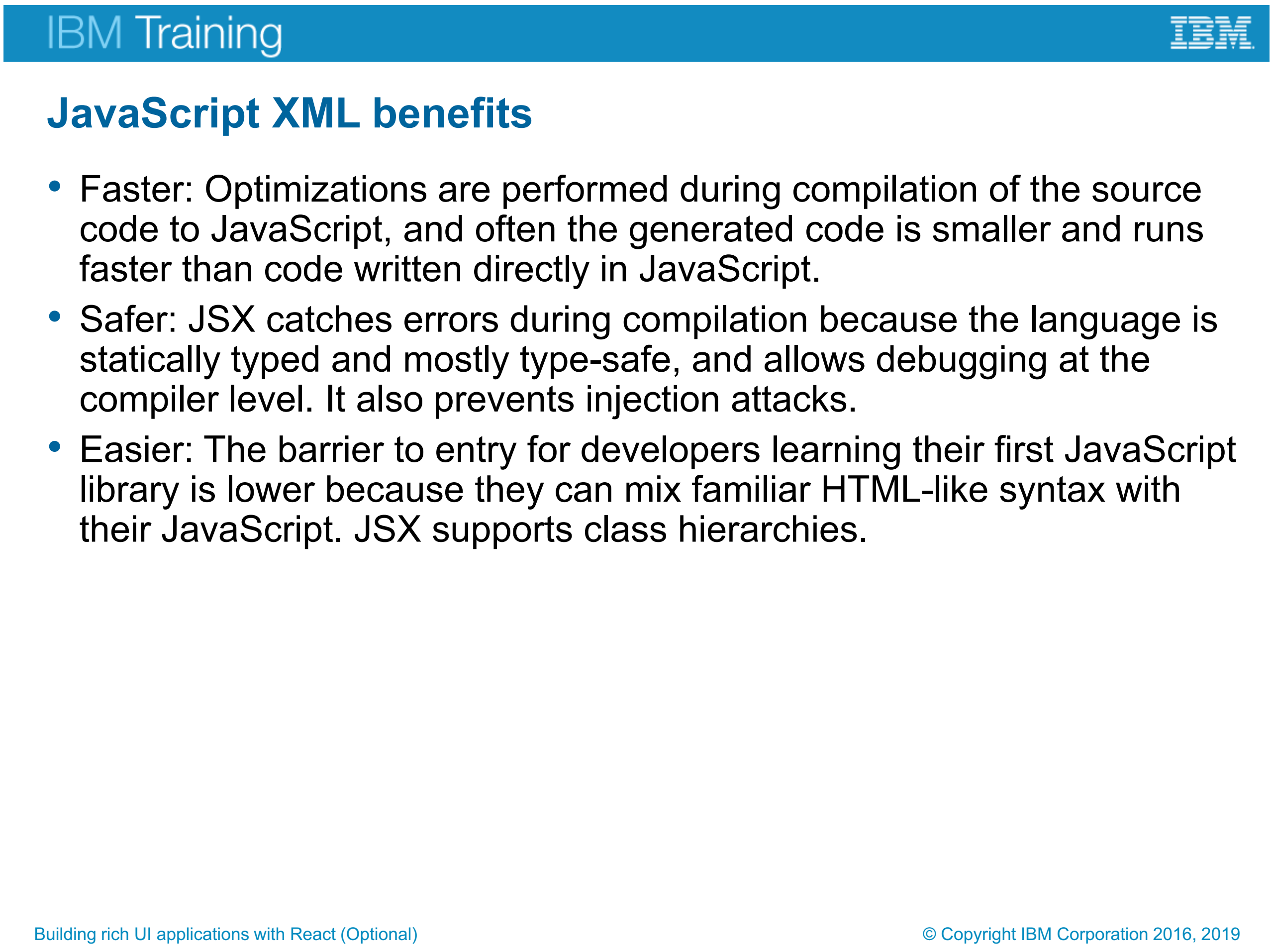
https://babeljs.io/docs/en/

For more information about JSX, see the following website:

https://facebook.github.io/jsx/

For more information about the compatibility of ES5 with web browsers, see the following website:

http://kangax.github.io/compat-table/es5/



IBM Training

JavaScript XML benefits

- Faster: Optimizations are performed during compilation of the source code to JavaScript, and often the generated code is smaller and runs faster than code written directly in JavaScript.
- Safer: JSX catches errors during compilation because the language is statically typed and mostly type-safe, and allows debugging at the compiler level. It also prevents injection attacks.
- Easier: The barrier to entry for developers learning their first JavaScript library is lower because they can mix familiar HTML-like syntax with their JavaScript. JSX supports class hierarchies.

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-5. JavaScript XML benefits

- Faster: Optimizations are performed during compilation of the source code to JavaScript, and often the generated code is smaller and runs faster than code written directly in JavaScript.
- Safer: JSX catches errors during compilation because the language is statically typed and mostly type-safe, and allows debugging at the compiler level. It also prevents injection attacks.
- Easier: The barrier to entry for developers learning their first JavaScript library is lower because they can mix familiar HTML-like syntax with their JavaScript. JSX supports class hierarchies.

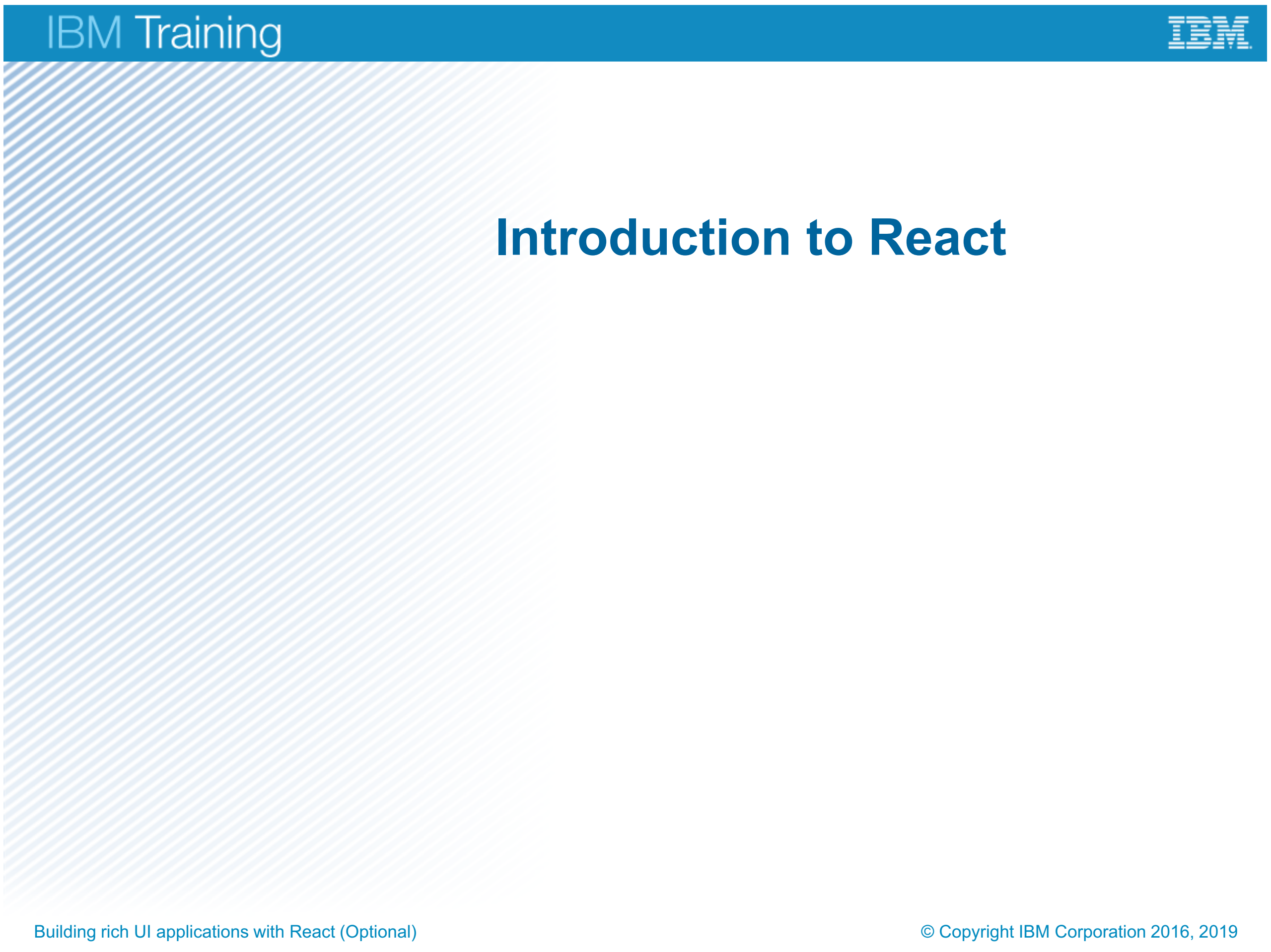
For more information about JSX benefits, see the following website:

https://jsx.github.io/

For more information about JSX preventing injection attacks, see the following website:

https://reactjs.org/docs/introducing-jsx.html

5.2. Introduction to React



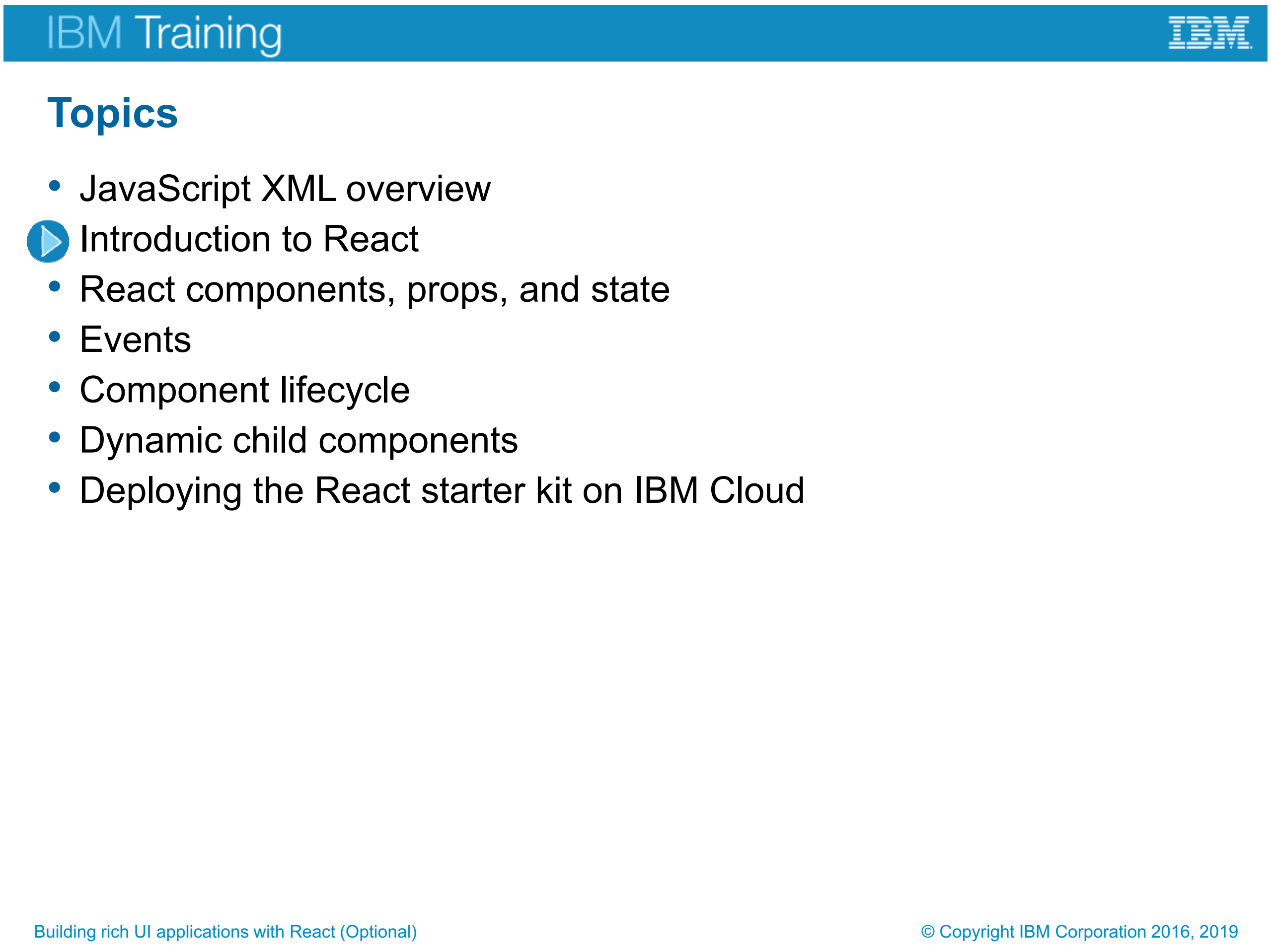
IBM Training

Introduction to React

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-6. Introduction to React



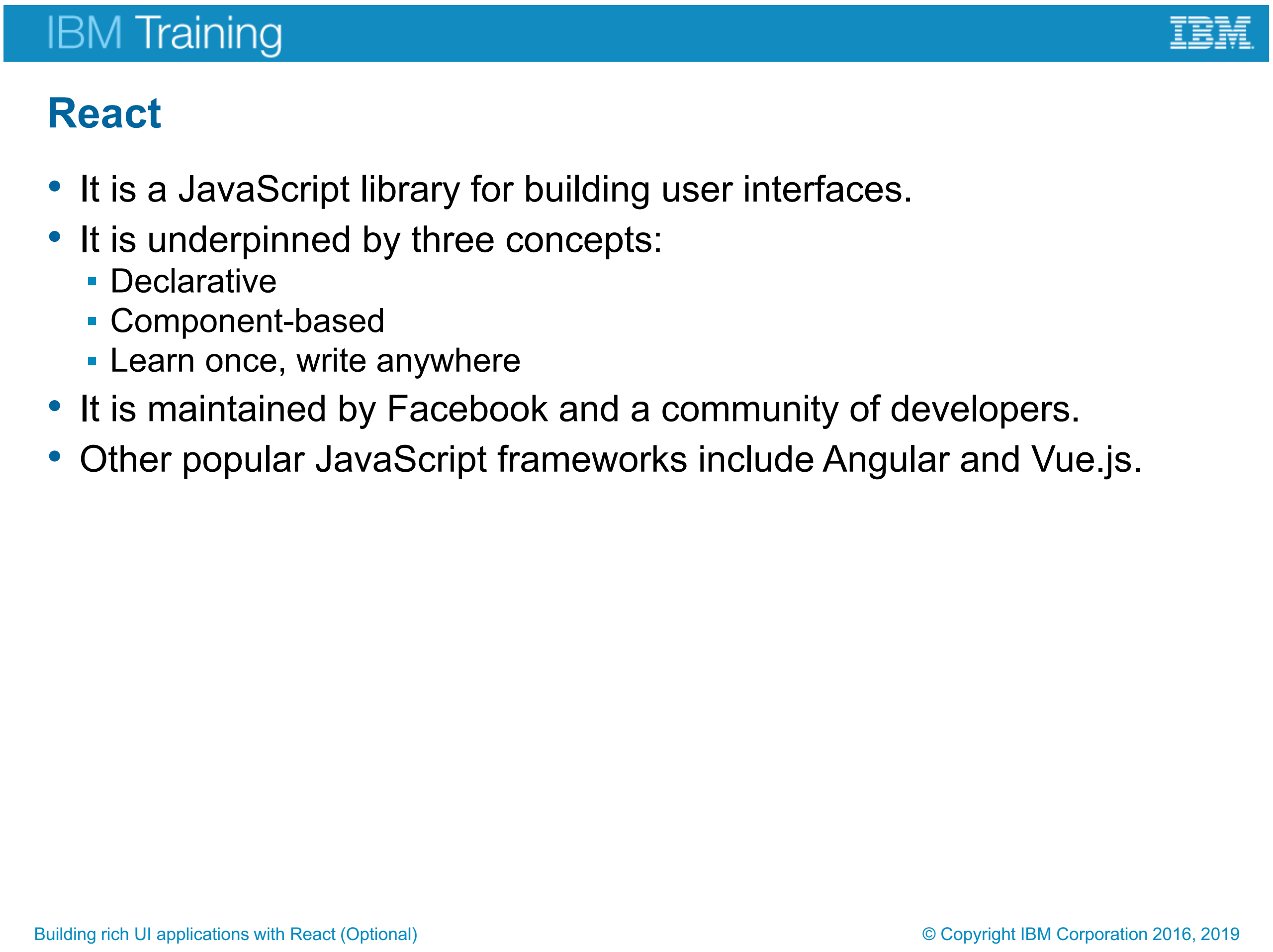
IBM Training

Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-7. Topics



IBM Training

React

- It is a JavaScript library for building user interfaces.
- It is underpinned by three concepts:
  - Declarative
  - Component-based
  - Learn once, write anywhere
- It is maintained by Facebook and a community of developers.
- Other popular JavaScript frameworks include Angular and Vue.js.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-8. React

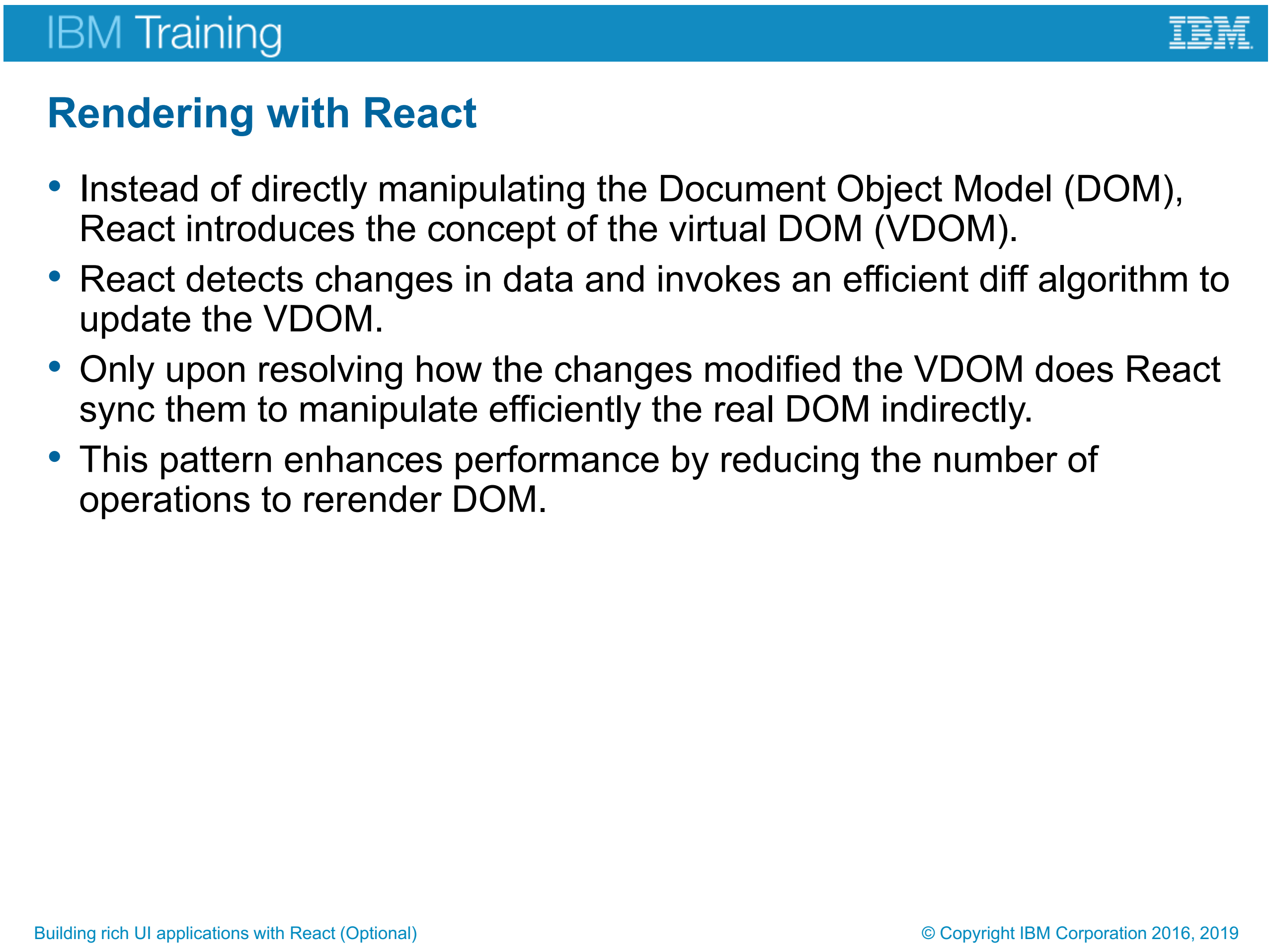
- It is a JavaScript library for building user interfaces.
- It is underpinned by three concepts:
  - Declarative
  - Component-based
  - Learn once, write anywhere
- It is maintained by Facebook and a community of developers.
- Other popular JavaScript frameworks include Angular and Vue.js.

- Declarative: Tell the system what you want to do but not how it is done (the opposite of imperative). This approach allows the programmer to build the UI without directly interacting with the Document Object Model (DOM) or DOM events directly.

- Component-based: A component should obey the single responsivity principle with a hierarchy for easier reuse and readability.
- Learn once, write anywhere: Developers want the write-once, run-anywhere paradigm so that they can create functions across systems, but there were issues with debugging and optimization. React Native uses the *learn once, write anywhere* approach where developers can reuse their React web skills to develop on mobile iOS and Android instead of learning an entirely new framework.

For more information about React concepts of declarative, component-based' and "learn once, write anywhere", go to the following website:

https://reactjs.org/



IBM Training

Rendering with React

- Instead of directly manipulating the Document Object Model (DOM), React introduces the concept of the virtual DOM (VDOM).
- React detects changes in data and invokes an efficient diff algorithm to update the VDOM.
- Only upon resolving how the changes modified the VDOM does React sync them to manipulate efficiently the real DOM indirectly.
- This pattern enhances performance by reducing the number of operations to rerender DOM.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

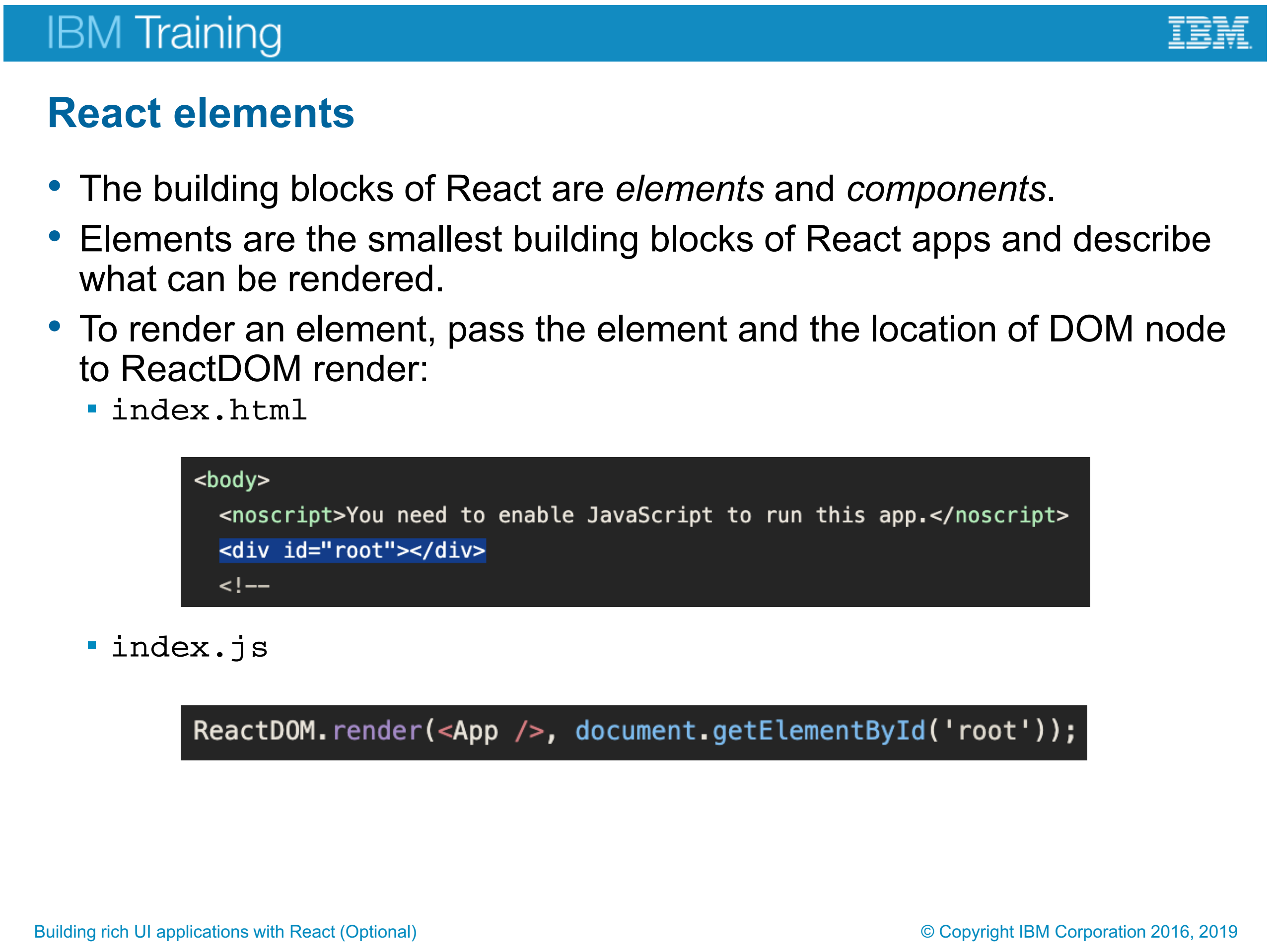
Figure 5-9. Rendering with React

- Instead of directly manipulating the Document Object Model (DOM), React introduces the concept of the virtual DOM (VDOM).
- React detects changes in data and invokes an efficient diff algorithm to update the VDOM.
- Only upon resolving how the changes modified the VDOM does React sync them to manipulate efficiently the real DOM indirectly.
- This pattern enhances performance by reducing the number of operations to rerender DOM.

The VDOM is a programming concept where a virtual representation of a UI is cached in memory and reconciled with the real DOM by using the ReactDOM library, which results in performance gains for UI updates.

For more information about DOM, see the following website:
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

For more information about the VDOM, see the following website:
https://reactjs.org/docs/faq-internals.html



IBM Training

React elements

- The building blocks of React are *elements* and *components*.
- Elements are the smallest building blocks of React apps and describe what can be rendered.
- To render an element, pass the element and the location of DOM node to ReactDOM render:
  - `index.html`

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
```

  - `index.js`

```
ReactDOM.render(<App />, document.getElementById('root'));
```

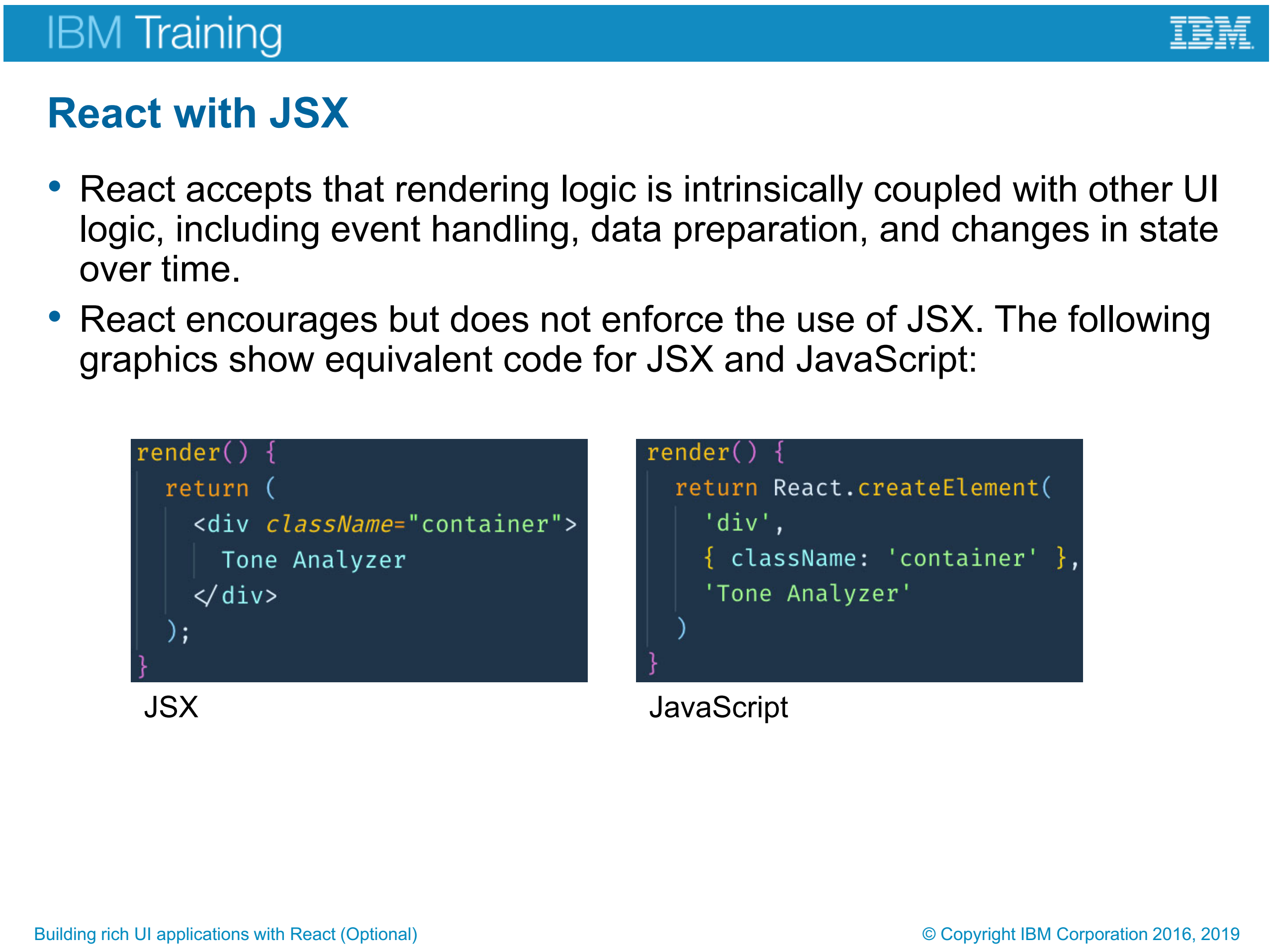
Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-10. React elements

- The building blocks of React are elements and components.
- Elements are the smallest building blocks of React apps and describe what can be rendered.
- To render an element, pass the element and the location of DOM node to ReactDOM render.

For more information about React elements, see the following website:
https://reactjs.org/docs/rendering-elements.html



IBM Training

React with JSX

- React accepts that rendering logic is intrinsically coupled with other UI logic, including event handling, data preparation, and changes in state over time.
- React encourages but does not enforce the use of JSX. The following graphics show equivalent code for JSX and JavaScript:

```
render() {
  return (
    <div className="container">
      Tone Analyzer
    </div>
  );
}
```

JSX

```
render() {
  return React.createElement(
    'div',
    { className: 'container' },
    'Tone Analyzer'
  )
}
```

JavaScript

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

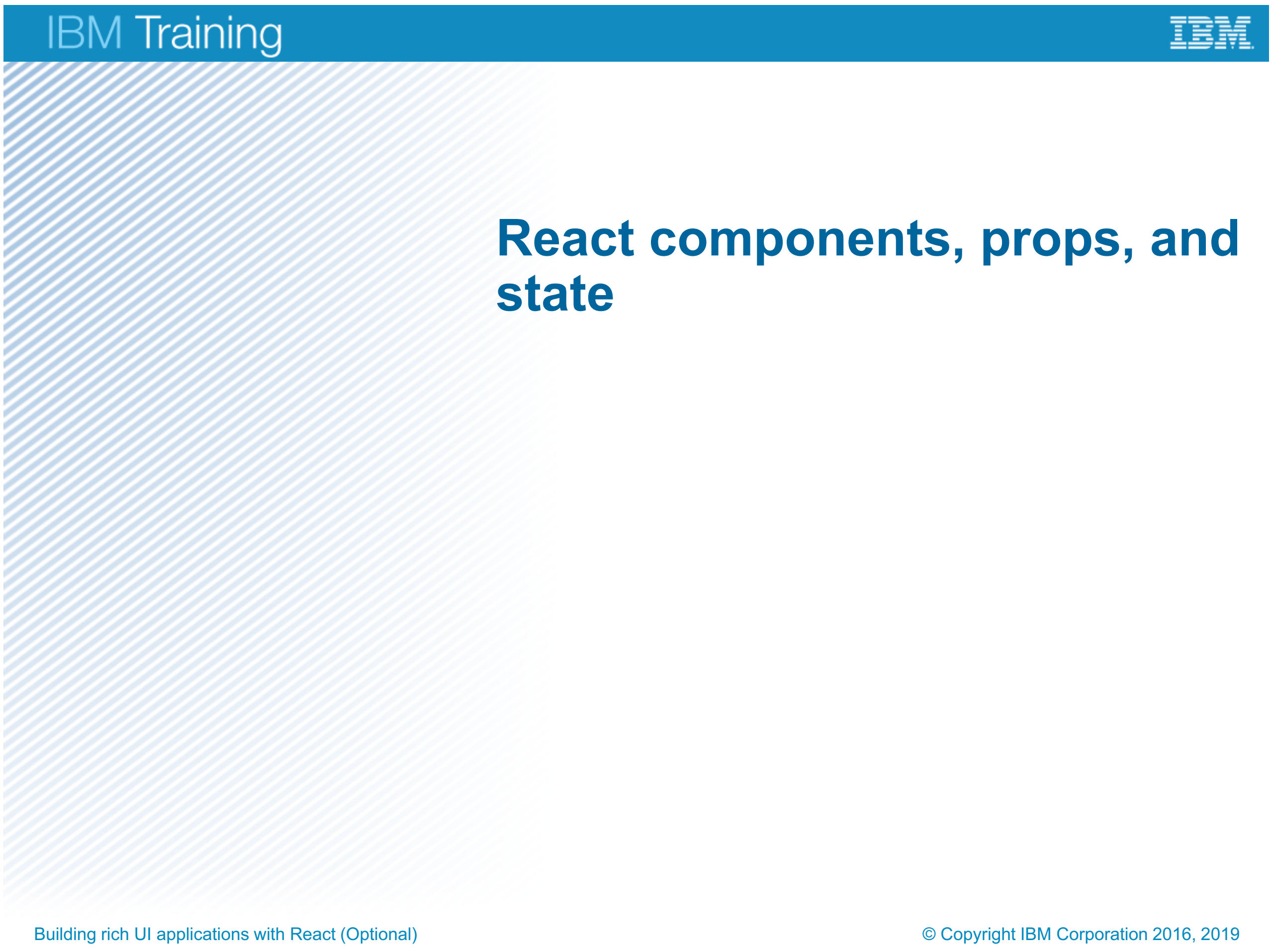
Figure 5-11. React with JSX

- React accepts that rendering logic is intrinsically coupled with other UI logic, including event handling, data preparation, and changes in state over time.
- React encourages but does not enforce the use of JSX.

The code snippets render a “div” element with a class name that is equal to the container that contains the text ‘Tone Analyzer’.
Any valid JavaScript expression can be embedded in JSX by using curly braces.

Upon compilation, JSX expressions become JavaScript functions and are evaluated to JavaScript objects.

5.3. React components, props, and state

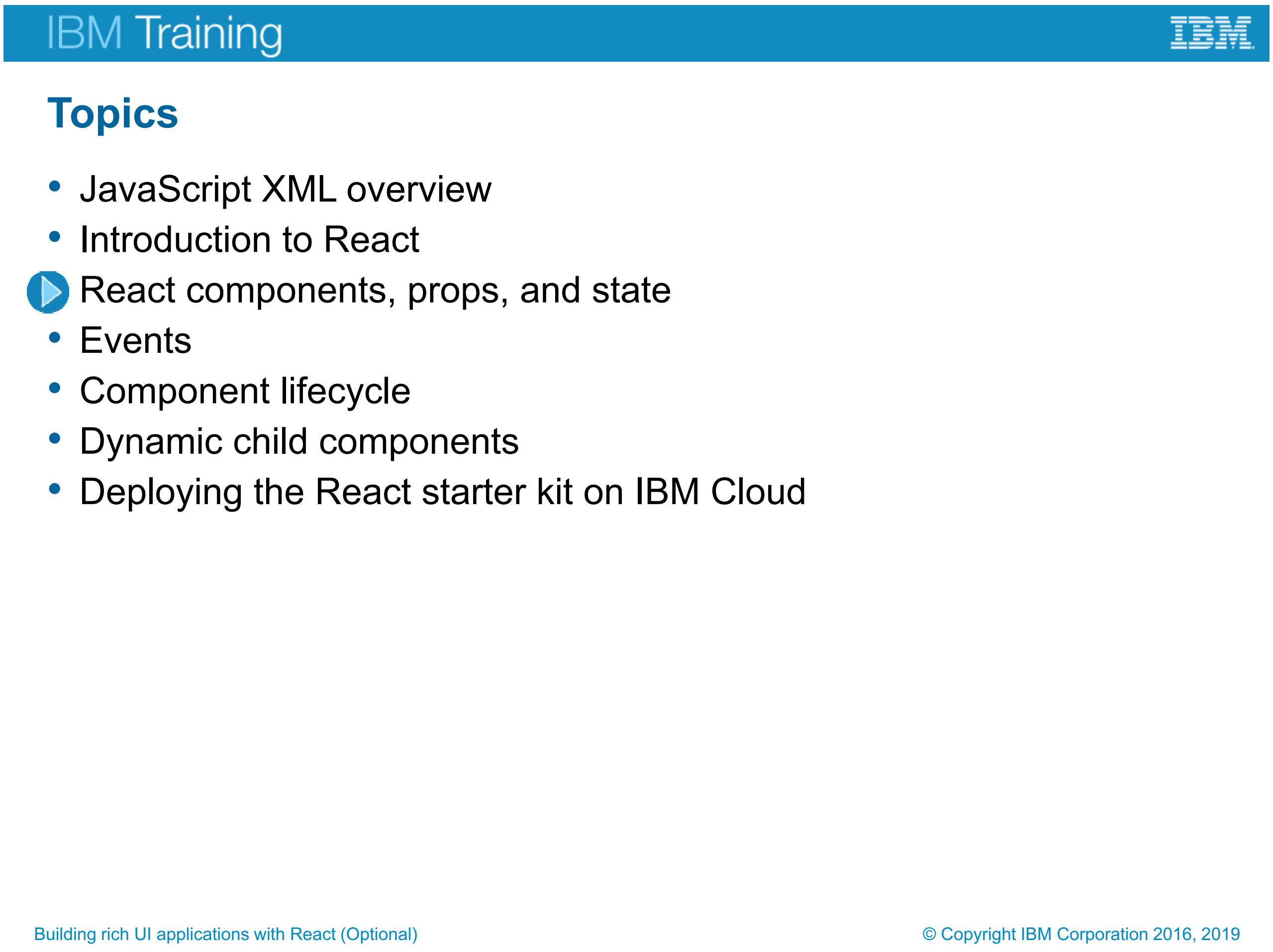

IBM Training

React components, props, and state

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-12. React components, props, and state



IBM Training

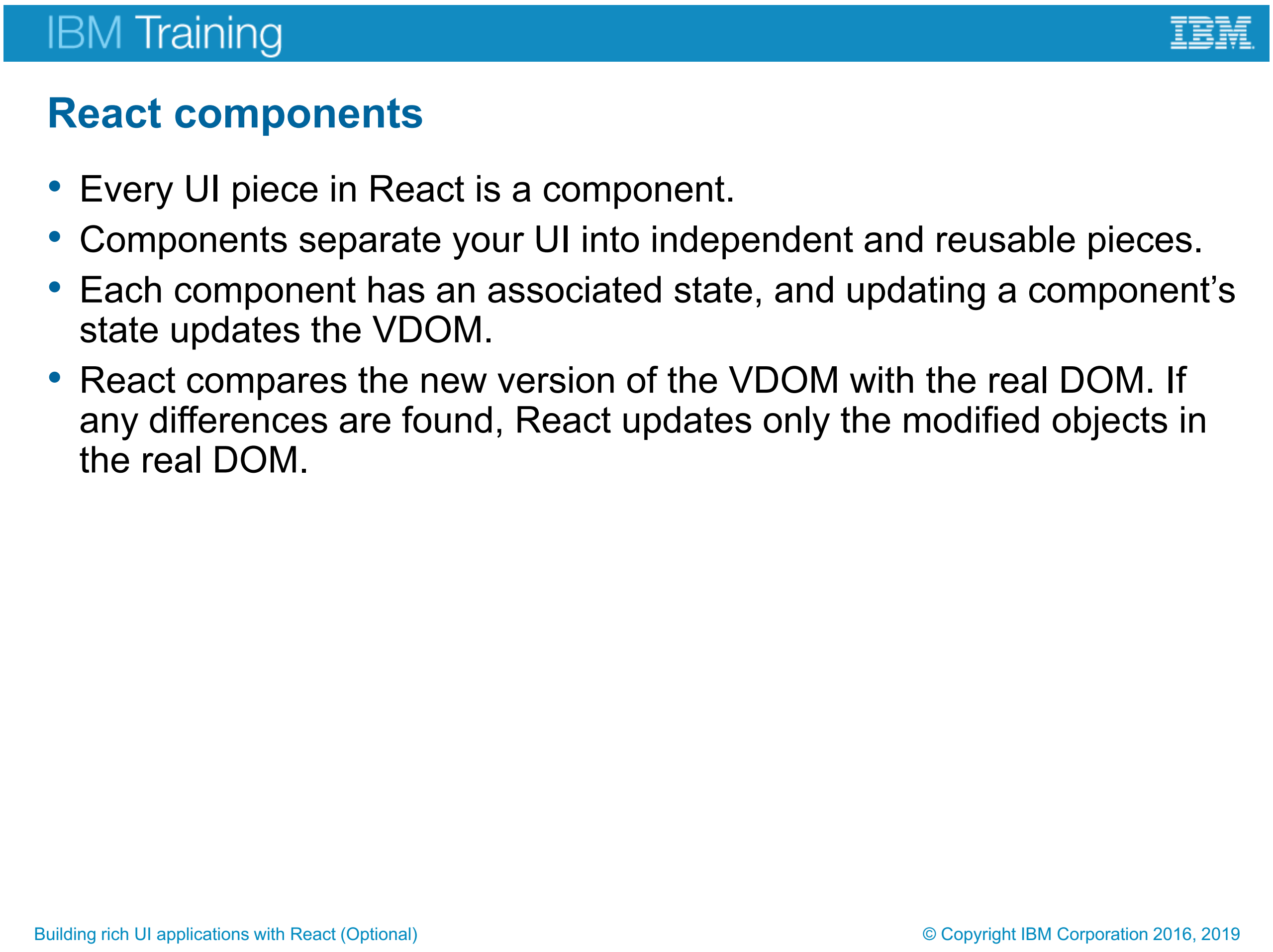
Topics

- JavaScript XML overview
- Introduction to React
- ▶ React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-13. Topics



IBM Training

React components

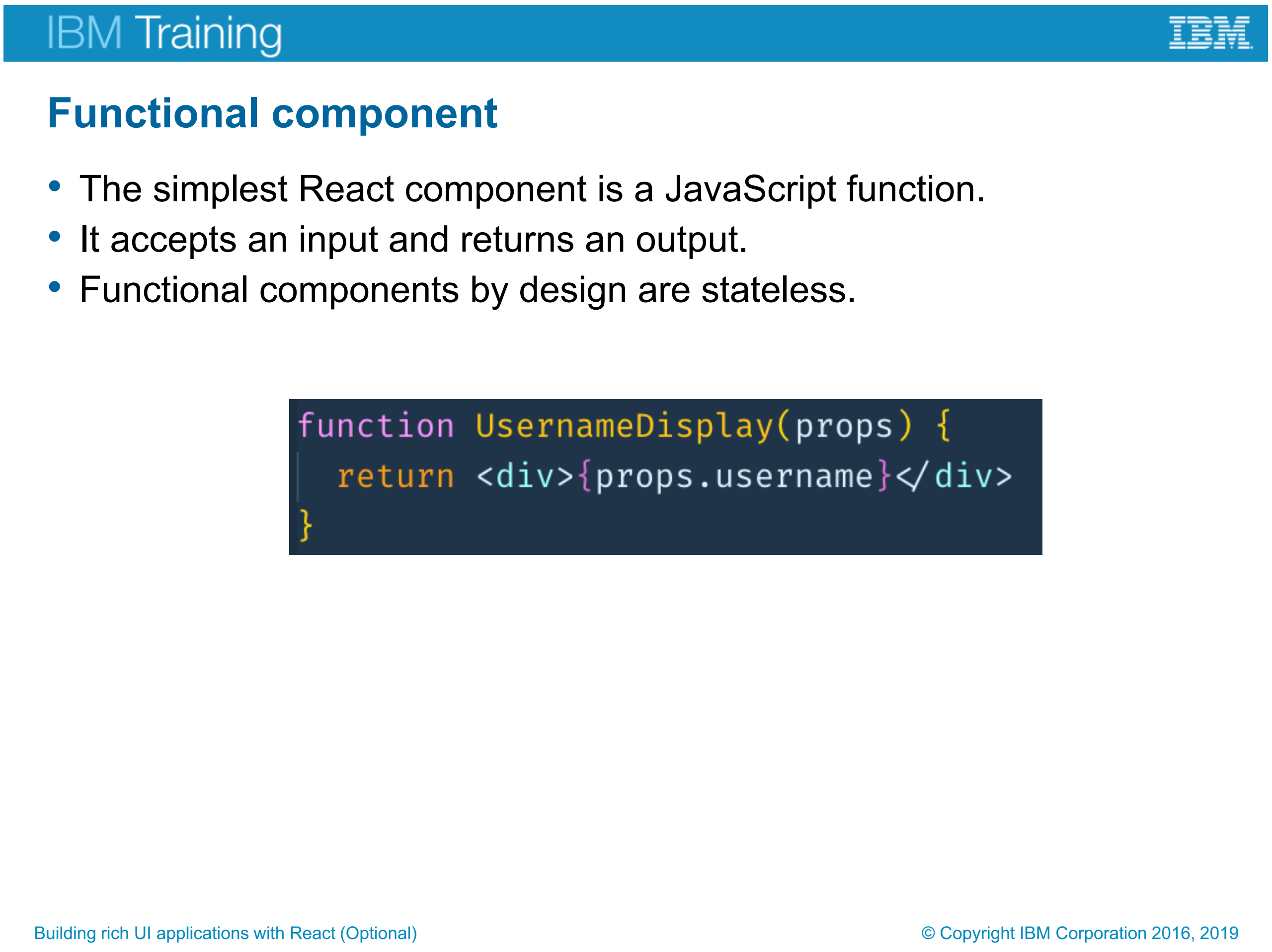
- Every UI piece in React is a component.
- Components separate your UI into independent and reusable pieces.
- Each component has an associated state, and updating a component's state updates the VDOM.
- React compares the new version of the VDOM with the real DOM. If any differences are found, React updates only the modified objects in the real DOM.

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-14. React components

- Every UI piece in React is a component.
- Components separate your UI into independent and reusable pieces.
- Each component has an associated state, and updating a component's state updates the VDOM.
- React compares the new version of the VDOM with the real DOM. If any differences are found, React updates only the modified objects in the real DOM.

React defines components to enable the UI to be split into independent and reusable pieces. Components are either a stateless functional component or a stateful class component.



IBM Training

Functional component

- The simplest React component is a JavaScript function.
- It accepts an input and returns an output.
- Functional components by design are stateless.

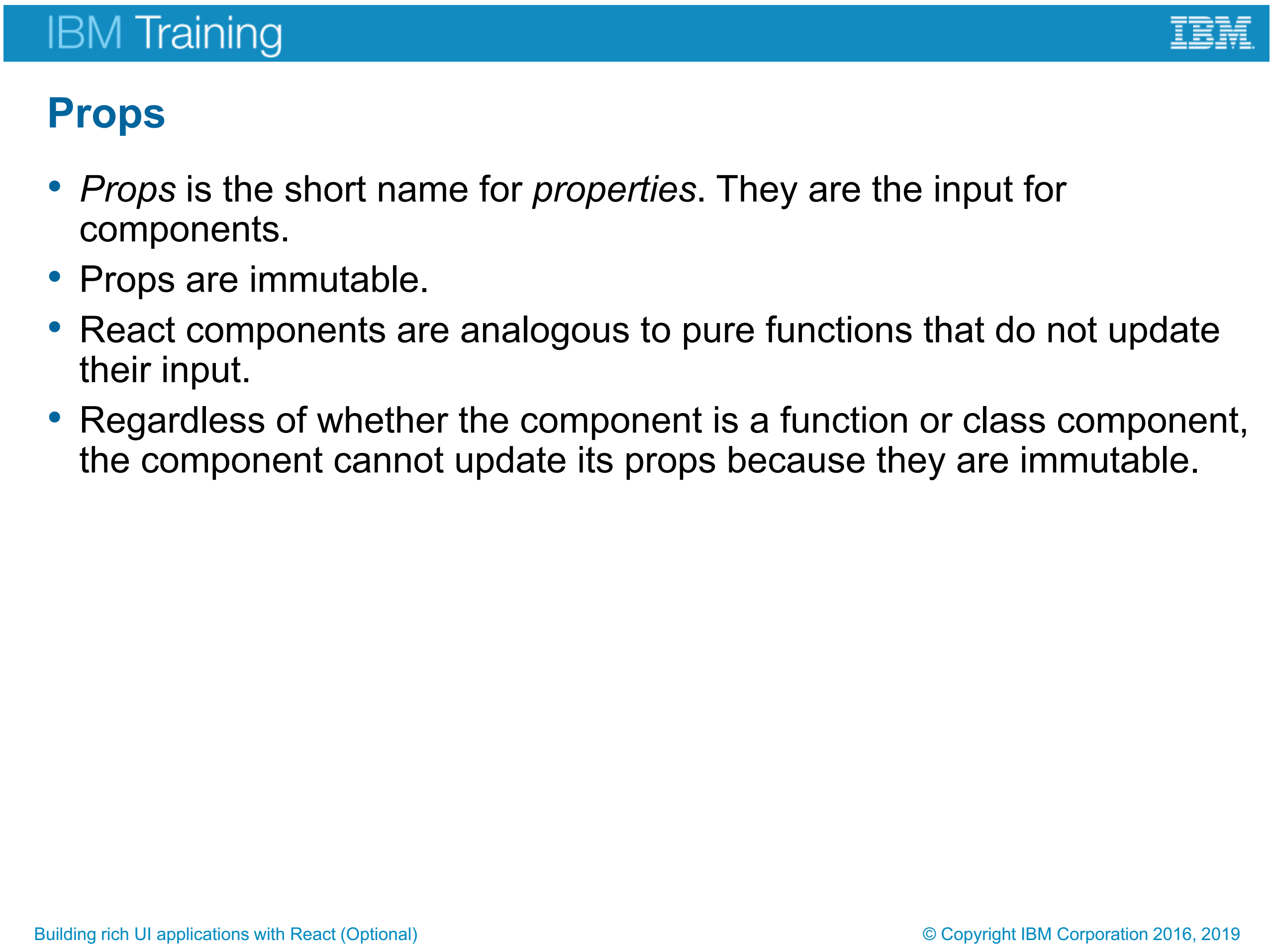
```
function UsernameDisplay(props) {
  return <div>{props.username}</div>
}
```

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-15. Functional component

- The simplest React component is a JavaScript function.
- It accepts an input and returns an output.
- Functional components by design are stateless.



IBM Training

Props

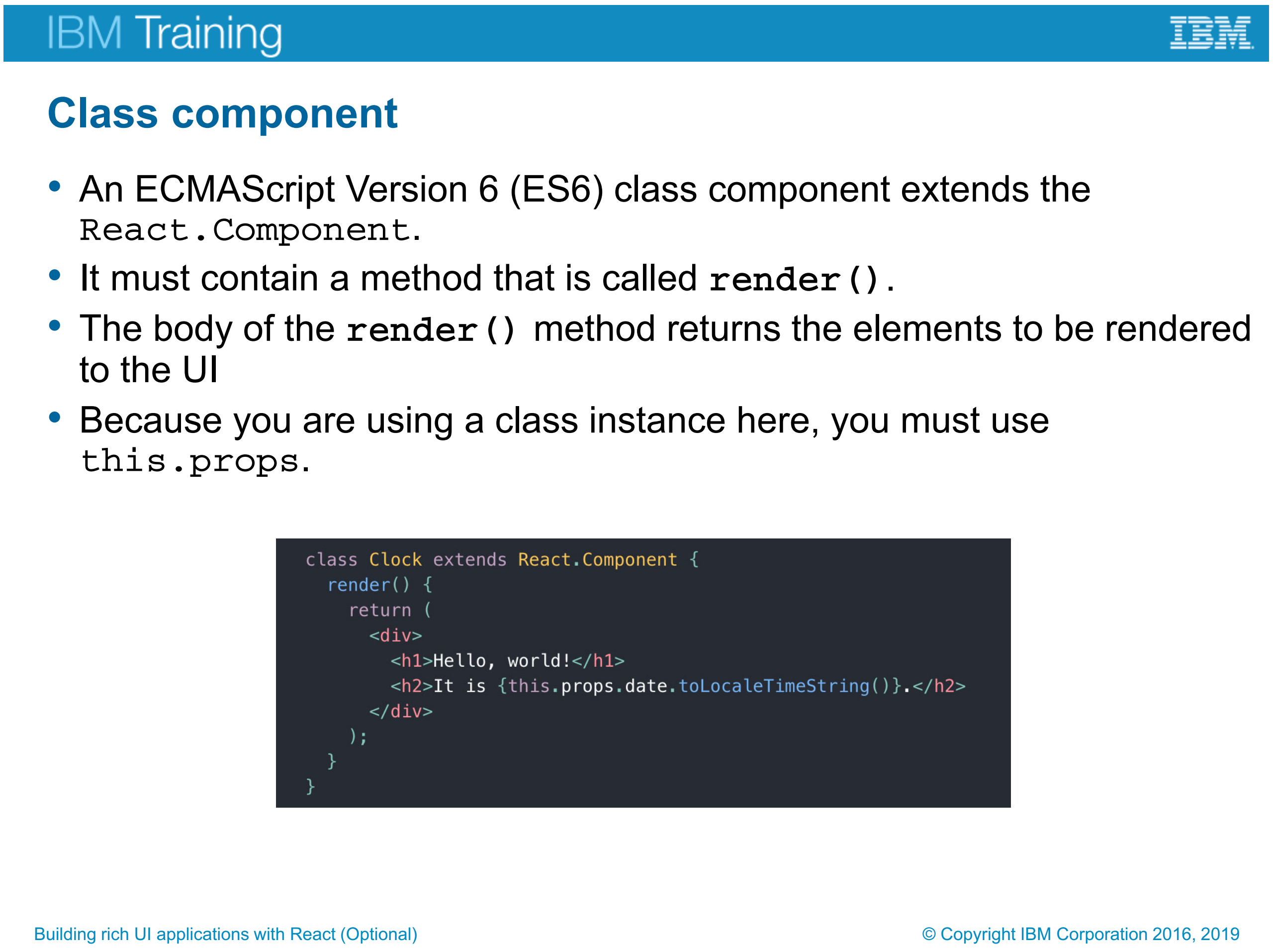
- *Props* is the short name for *properties*. They are the input for components.
- Props are immutable.
- React components are analogous to pure functions that do not update their input.
- Regardless of whether the component is a function or class component, the component cannot update its props because they are immutable.

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-16. Props

- *Props* is the short name for *properties*. They are the input for components.
- Props are immutable.
- React components are analogous to pure functions that do not update their input.
- Regardless of whether the component is a function or class component, the component cannot update its props because they are immutable.

Immutable means that the data cannot be modified after it is initially set.
With props, you can pass data to child components. For example, if you have a component that displays the results of a search, a parent component can pass down the list of results props: ”this.props.results”.



IBM Training

Class component

- An ECMAScript Version 6 (ES6) class component extends the `React.Component`.
- It must contain a method that is called **`render()`**.
- The body of the **`render()`** method returns the elements to be rendered to the UI
- Because you are using a class instance here, you must use `this.props`.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-17. Class component

- An ECMAScript Version 6 (ES6) class component extends the `React.Component`.
- It must contain a method that is called **`render()`**.
- The body of the **`render()`** method returns the elements to be rendered to the UI
- Because you are using a class instance here, you must use “`this.props`”.

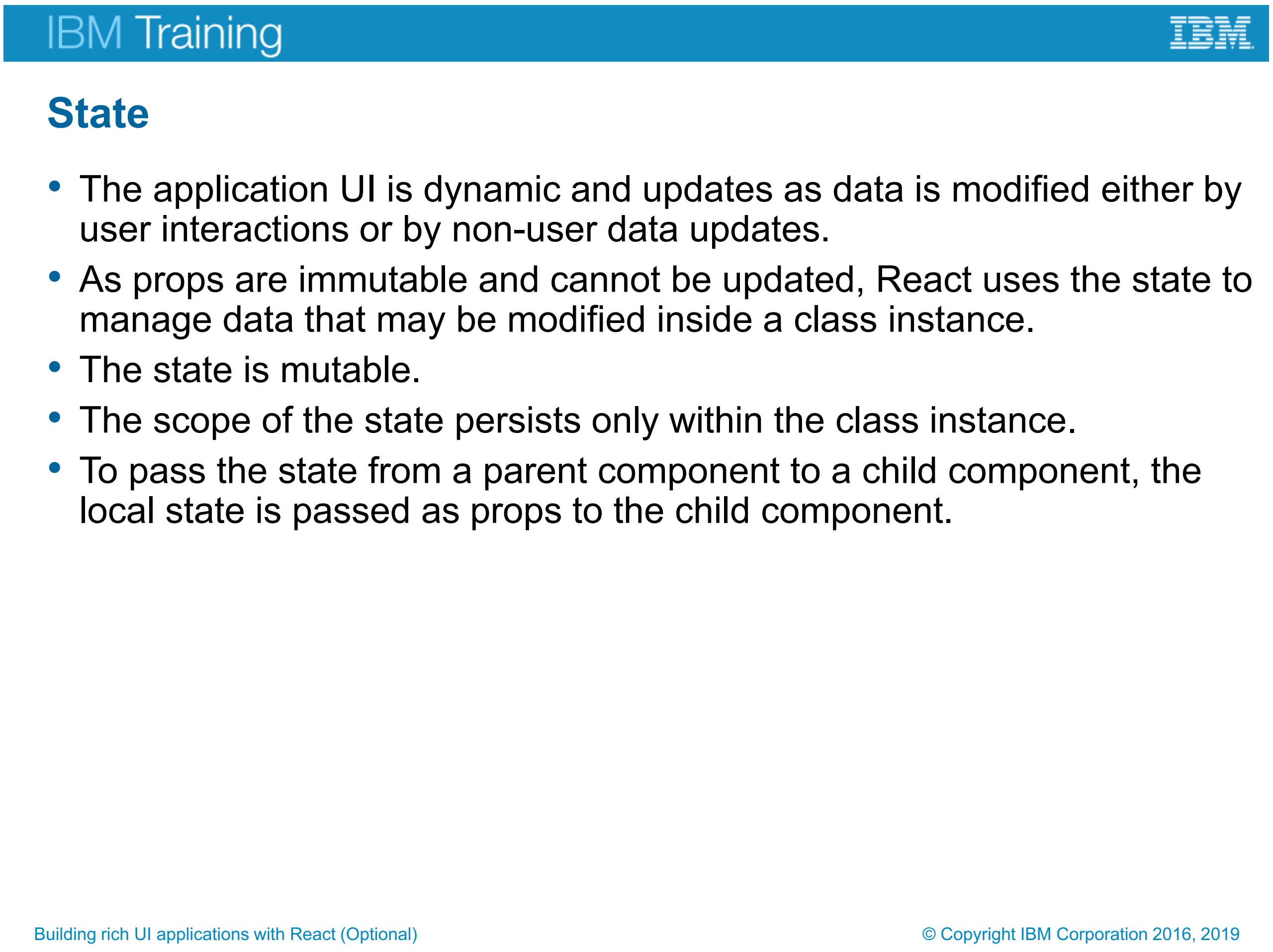
When do you use functional (stateless) components versus class (stateful) components?

Benefits of functional components:

- Less code
- Easier to understand
- Stateless by design
- Simpler to test

Functional components are always preferred when the component does not store the state. However, if your component needs to store the state, such as when the user can update the color of a component by clicking the UI and the component updates the state by using a listener event, then a class component should be used.

To learn more about class components, go to the following website: https://reactjs.org/docs/components-and-props.html



IBM Training

State

- The application UI is dynamic and updates as data is modified either by user interactions or by non-user data updates.
- As props are immutable and cannot be updated, React uses the state to manage data that may be modified inside a class instance.
- The state is mutable.
- The scope of the state persists only within the class instance.
- To pass the state from a pared component to a child component, the local state is passed as props to the child component.

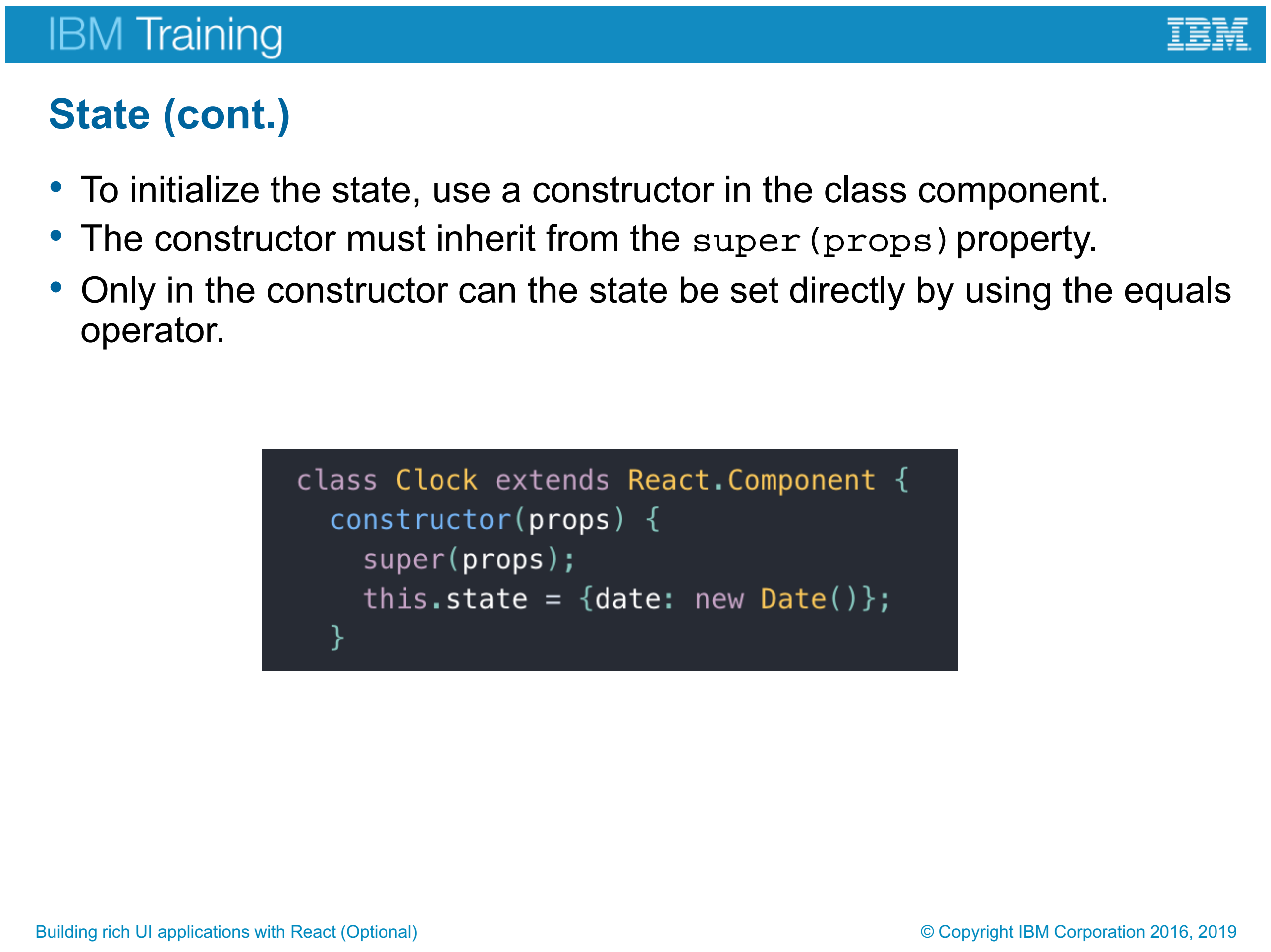
Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-18. State

- The application UI is dynamic and updates as data is modified either by user interactions or by non-user data updates.
- As props are immutable and cannot be updated, React uses the state to manage data that may be modified inside a class instance.
- The state is mutable.
- The scope of the state persists only within the class instance.
- To pass the state from a parent component to a child component, the local state is passed as props to the child component

Mutable means that the data can be modified even after it is initially set.
The scope of the state exists only within the class instance so that different instances of the same class can have different values of state. For example, one instance of a component might have a state color of blue and another instance of the same component might have a state color of red concurrently.
Passing a state from a component to another component is explained further when we explore nested components.


IBM Training

State (cont.)

- To initialize the state, use a constructor in the class component.
- The constructor must inherit from the `super(props)` property.
- Only in the constructor can the state be set directly by using the equals operator.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Building rich UI applications with React (Optional)

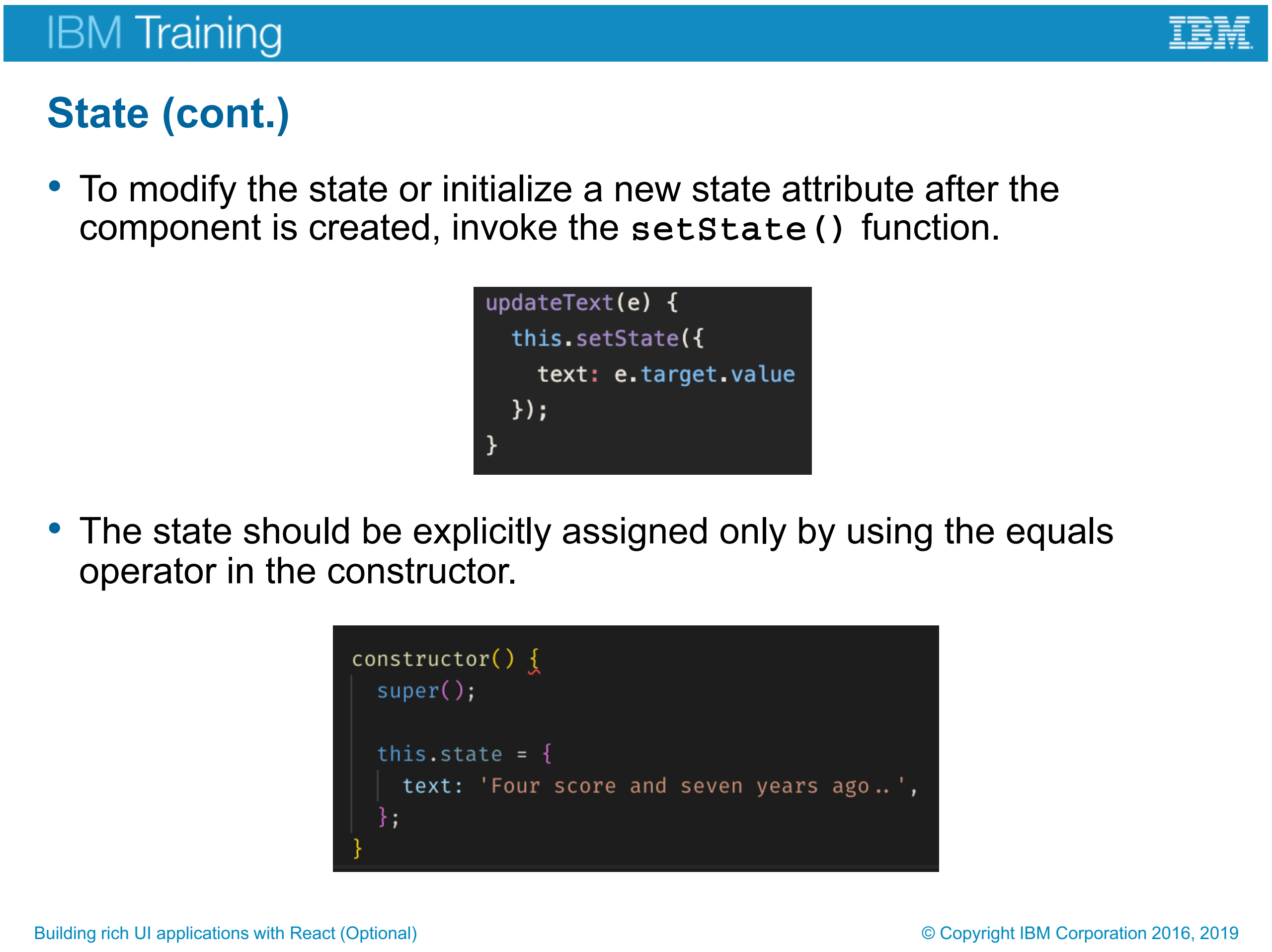
© Copyright IBM Corporation 2016, 2019

Figure 5-19. State (cont.)

- To initialize the state, use a constructor in the class component.
- The constructor must inherit from the `super(props)`property.
- Only in the constructor can the state be set directly by using the equals operator.

Passing props to the constructor provides access to “this.props” inside the constructor. For example, if you used “console.log(this.props)” without passing props into the constructor, then it would fail because “this.props” would be undefined.

Not passing props to the constructor has no effect on accessing “this.props” later on outside the constructor. The render method always has access to “this.props”.



IBM Training

State (cont.)

- To modify the state or initialize a new state attribute after the component is created, invoke the `setState()` function.

```
updateText(e) {
  this.setState({
    text: e.target.value
  });
}
```

- The state should be explicitly assigned only by using the equals operator in the constructor.

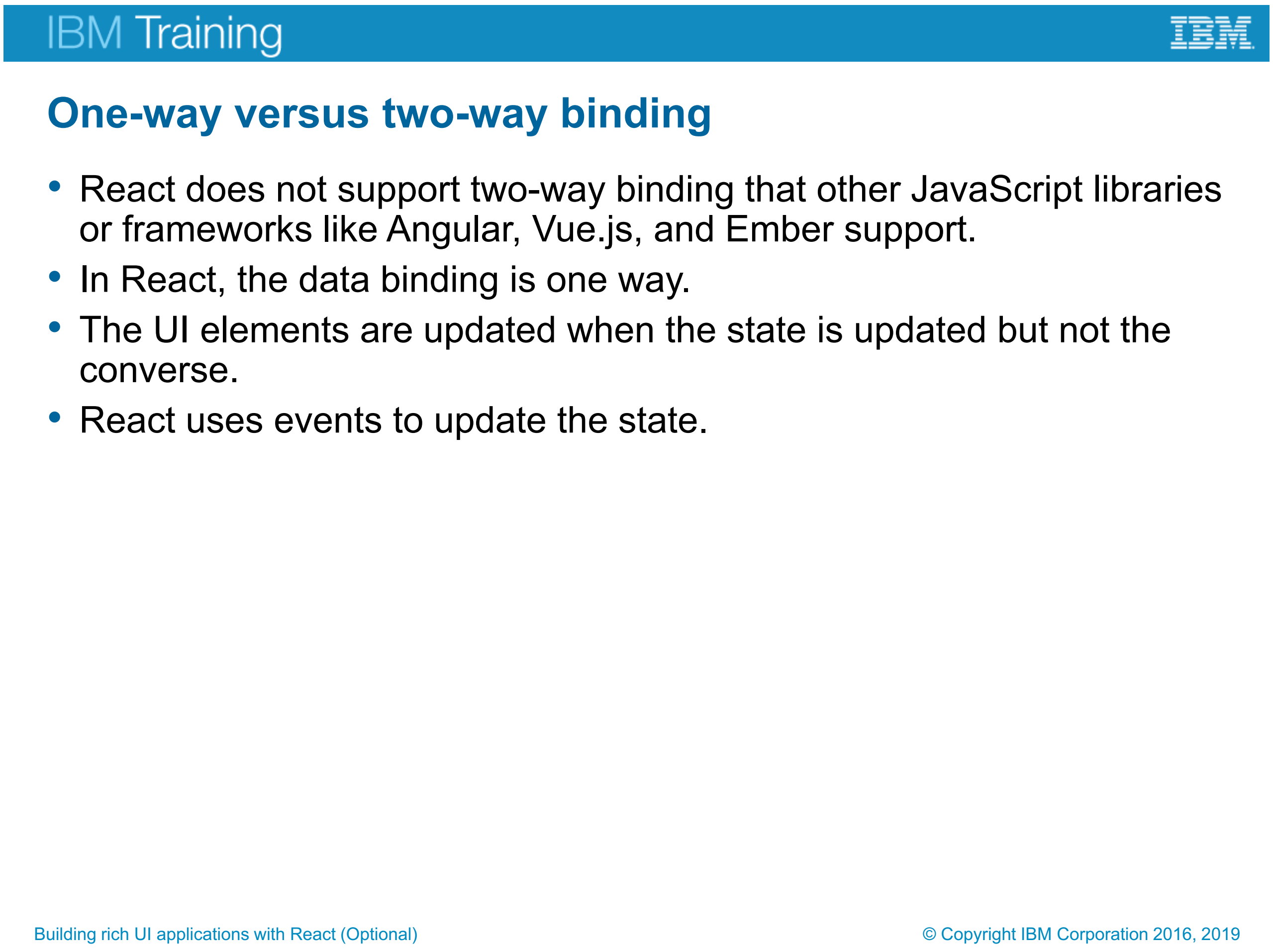
```
constructor() {
  super();

  this.state = {
    text: 'Four score and seven years ago..',
  };
}
```

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-20. State (cont.)

- To modify the state or initialize a new state attribute after the component is created, invoke the **`setState()`** function.
- The state should be explicitly assigned only by using the equals operator in the constructor.

You must use the **super()** method to grant access to “this.state” in the constructor. In this example, we did not pass props to the super constructor, so we do not have access to “this.props” in the constructor.



IBM Training

One-way versus two-way binding

- React does not support two-way binding that other JavaScript libraries or frameworks like Angular, Vue.js, and Ember support.
- In React, the data binding is one way.
- The UI elements are updated when the state is updated but not the converse.
- React uses events to update the state.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-21. One-way versus two-way binding

- React does not support two-way binding that other JavaScript libraries or frameworks like Angular, Vue.js, and Ember support.
- In React, the data binding is one way.
- The UI elements are updated when the state is updated but not the converse.
- React uses events to update the state.

Two-way binding means that the component's model data is updated when the UI field changes, and vice-versa.

The one-way design decision allows the data flow to be predictable, and simplifies the debugging and development of React applications.

5.4. Events

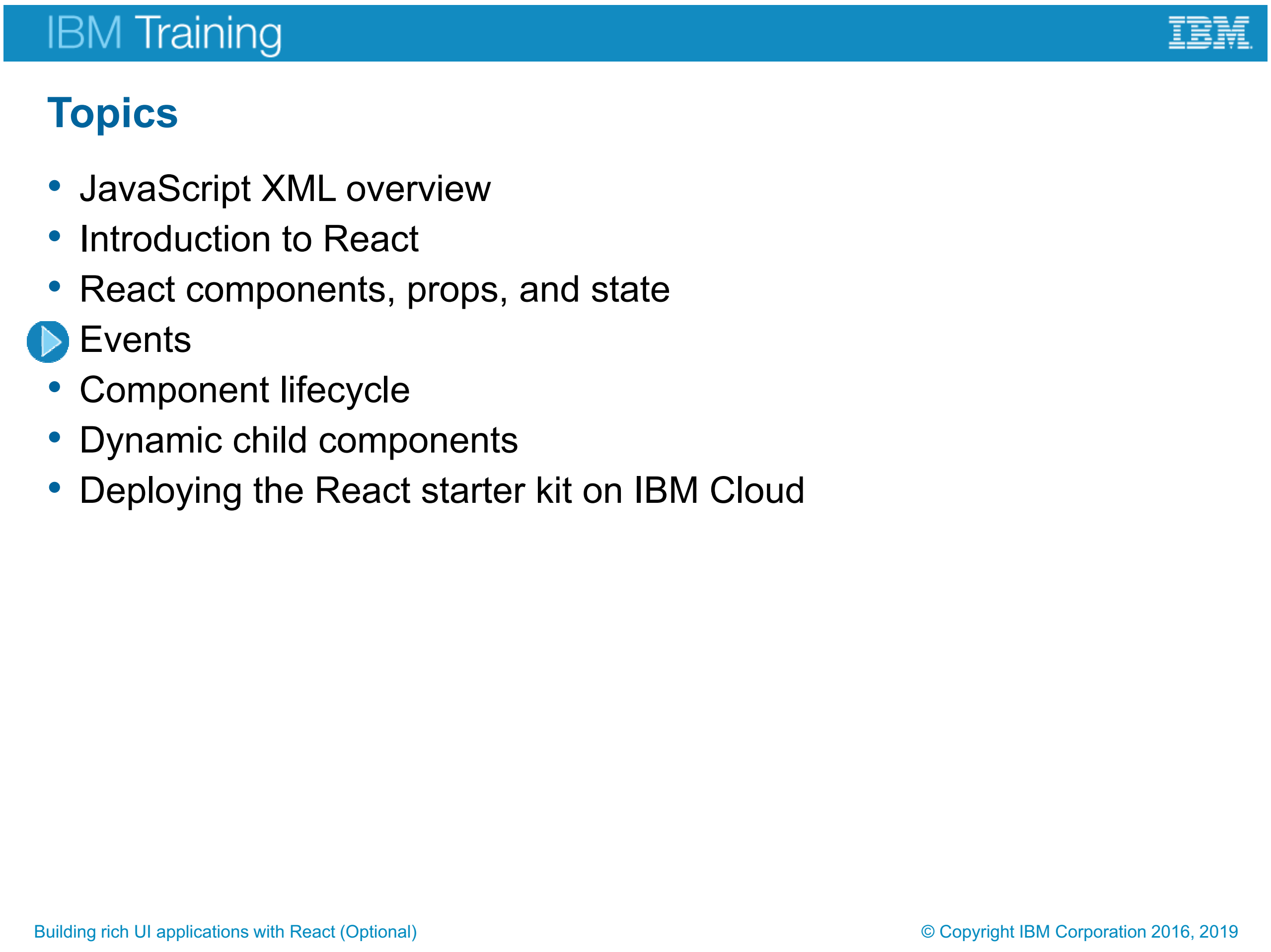
IBM Training

Events

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-22. Events



IBM Training

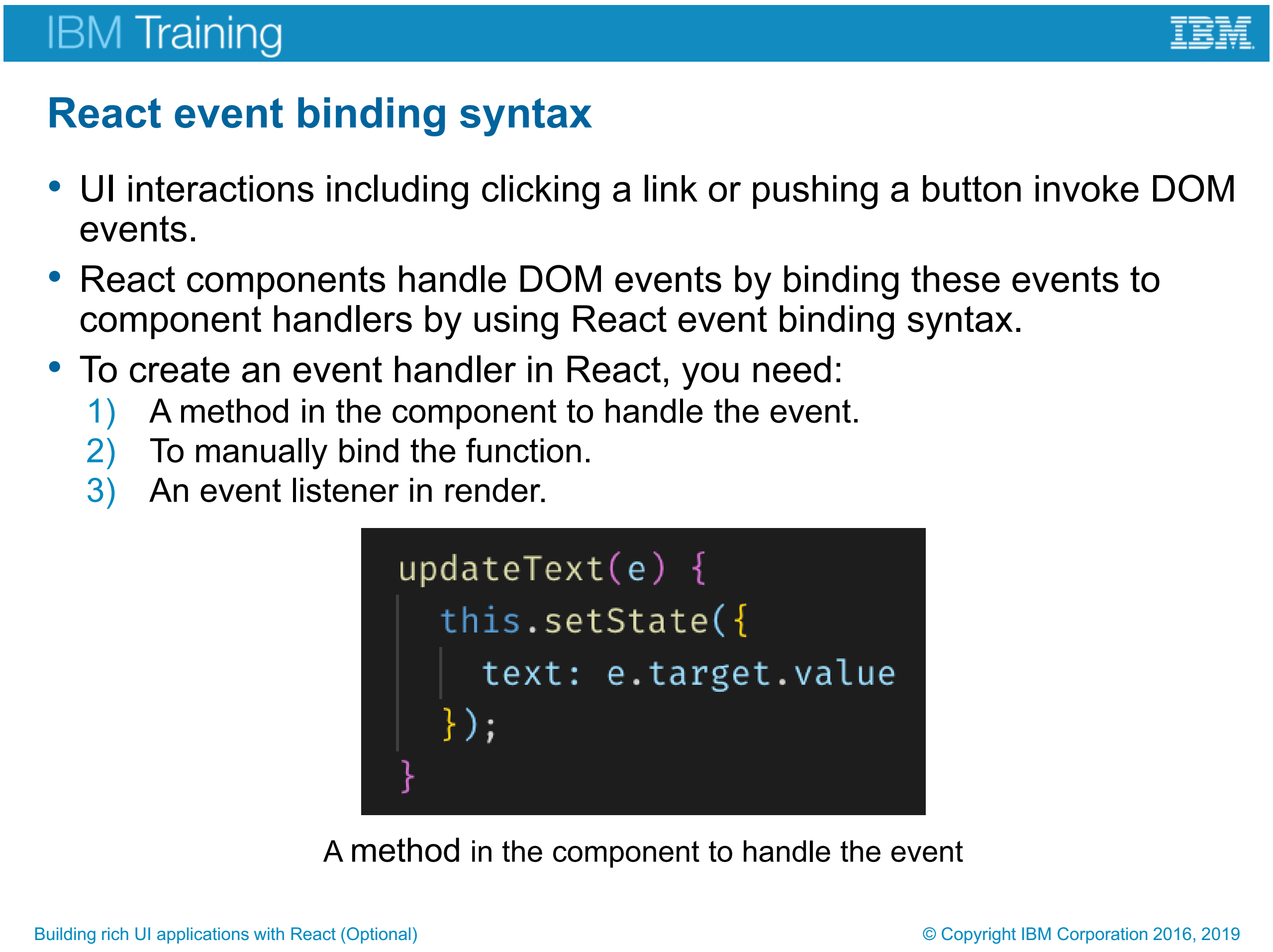
Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-23. Topics



IBM Training

React event binding syntax

- UI interactions including clicking a link or pushing a button invoke DOM events.
- React components handle DOM events by binding these events to component handlers by using React event binding syntax.
- To create an event handler in React, you need:
  1) A method in the component to handle the event.
  2) To manually bind the function.
  3) An event listener in render.

```
updateText(e) {
  this.setState({
    text: e.target.value
  });
}
```

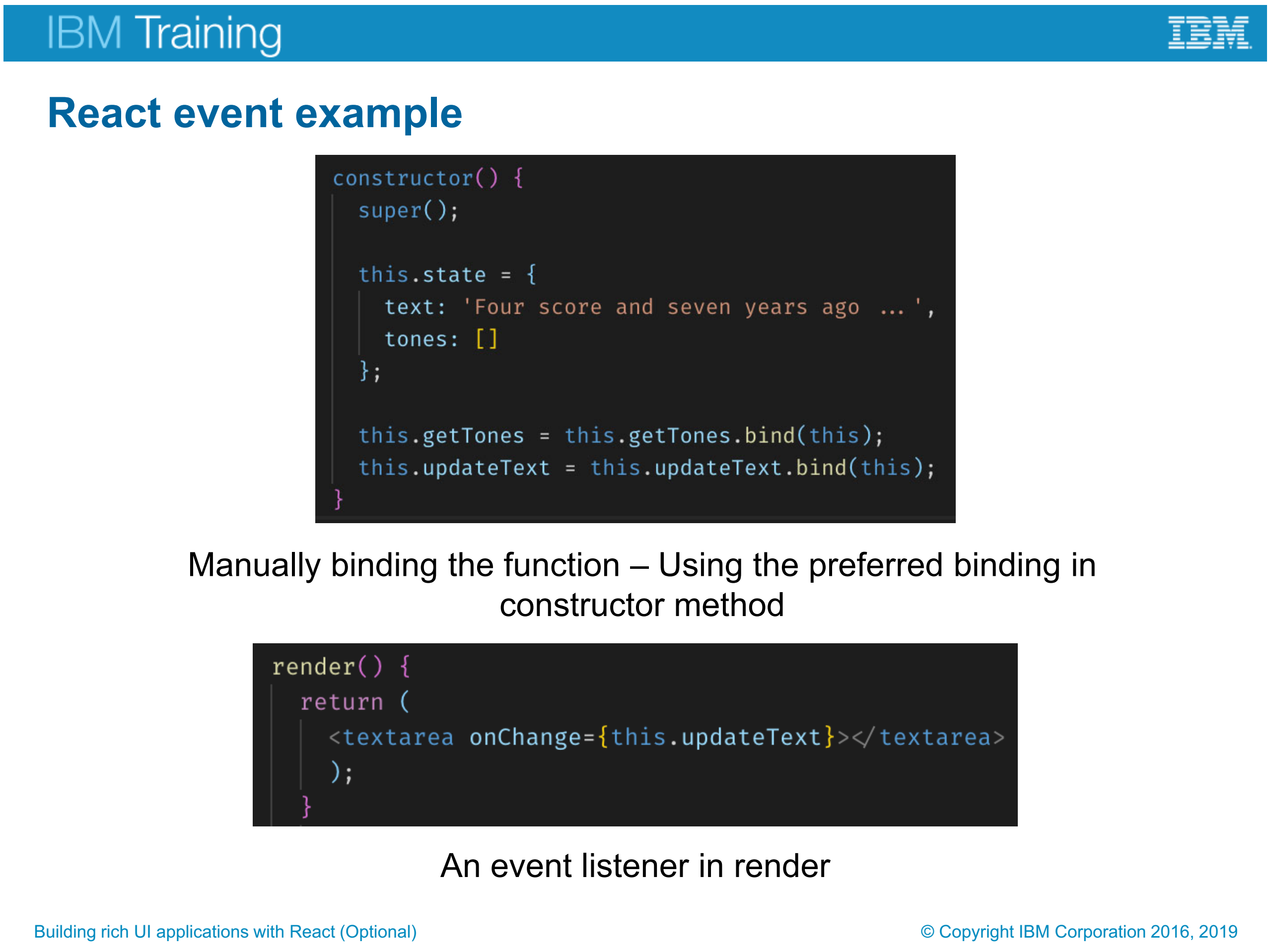
A method in the component to handle the event

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-24. React event binding syntax

- UI interactions including clicking a link or pushing a button to invoke DOM events.
- React components handle DOM events by binding these events to component handlers by using React event binding syntax.
- To create an event handler in React, you need:
  a. A method in the component to handle the event.
  b. To manually bind the function.
  c. An event listener in render.



IBM Training

React event example

```
constructor() {
  super();

  this.state = {
    text: 'Four score and seven years ago ... ',
    tones: []
  };

  this.getTones = this.getTones.bind(this);
  this.updateText = this.updateText.bind(this);
}
```

Manually binding the function – Using the preferred binding in constructor method

```
render() {
  return (
    <textarea onChange={this.updateText}></textarea>
    );
  }
```

An event listener in render

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-25. React event example

ES6 Class components do not auto-bind functions inside that component.

There are three methods to bind:

- Bind in constructor
- Arrow function in render
- Bind in render

Bind in constructor is the **preferred** method as the function is created only once upon creation of the class, however it requires “super()” to be called in the constructor.

Both arrow function in render and bind in render methods results in a new function being called every time the component re-renders, which can lead to performance issues.

5.5. Component lifecycle

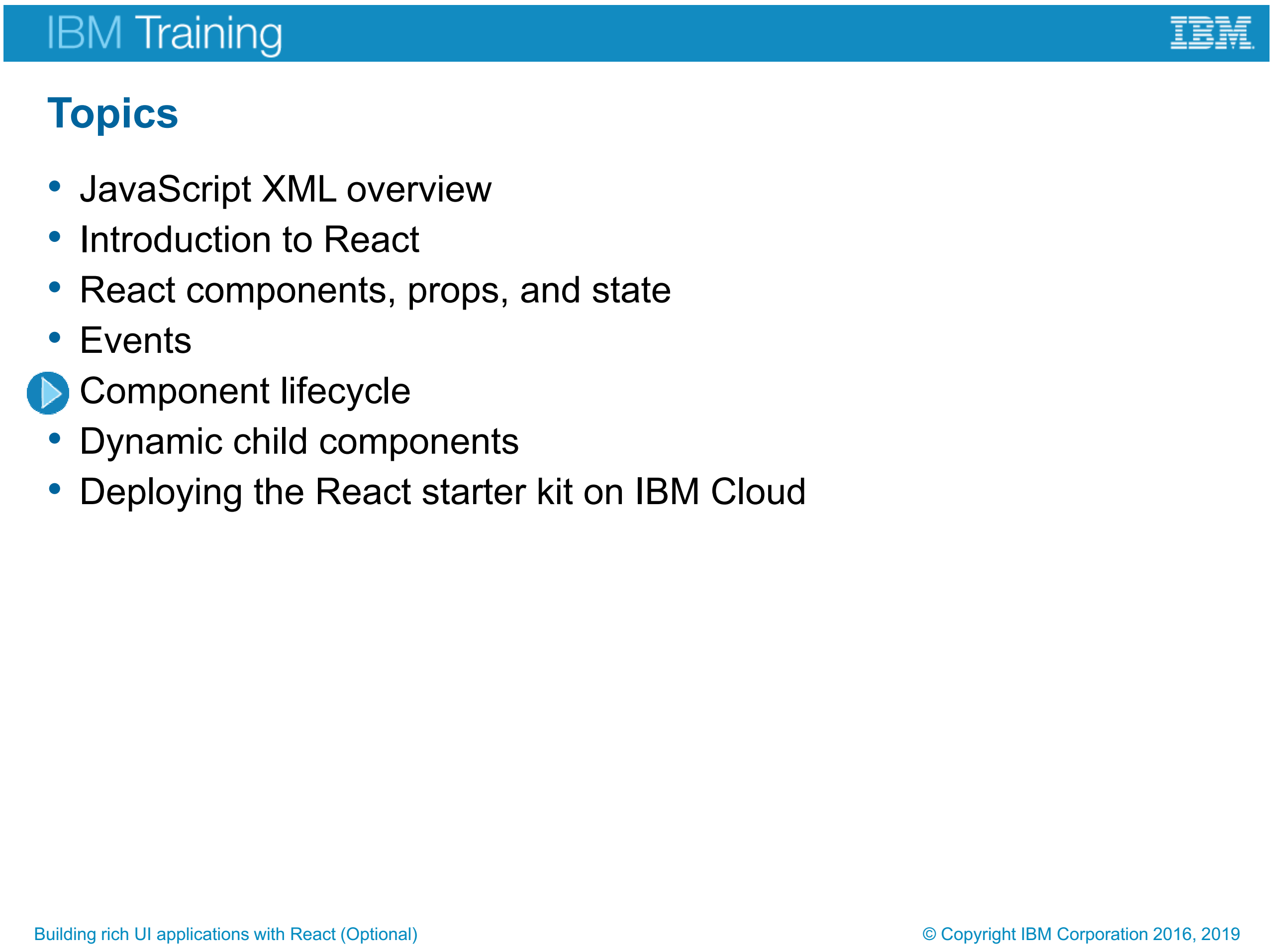
IBM Training

Component lifecycle

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-26. Component lifecycle



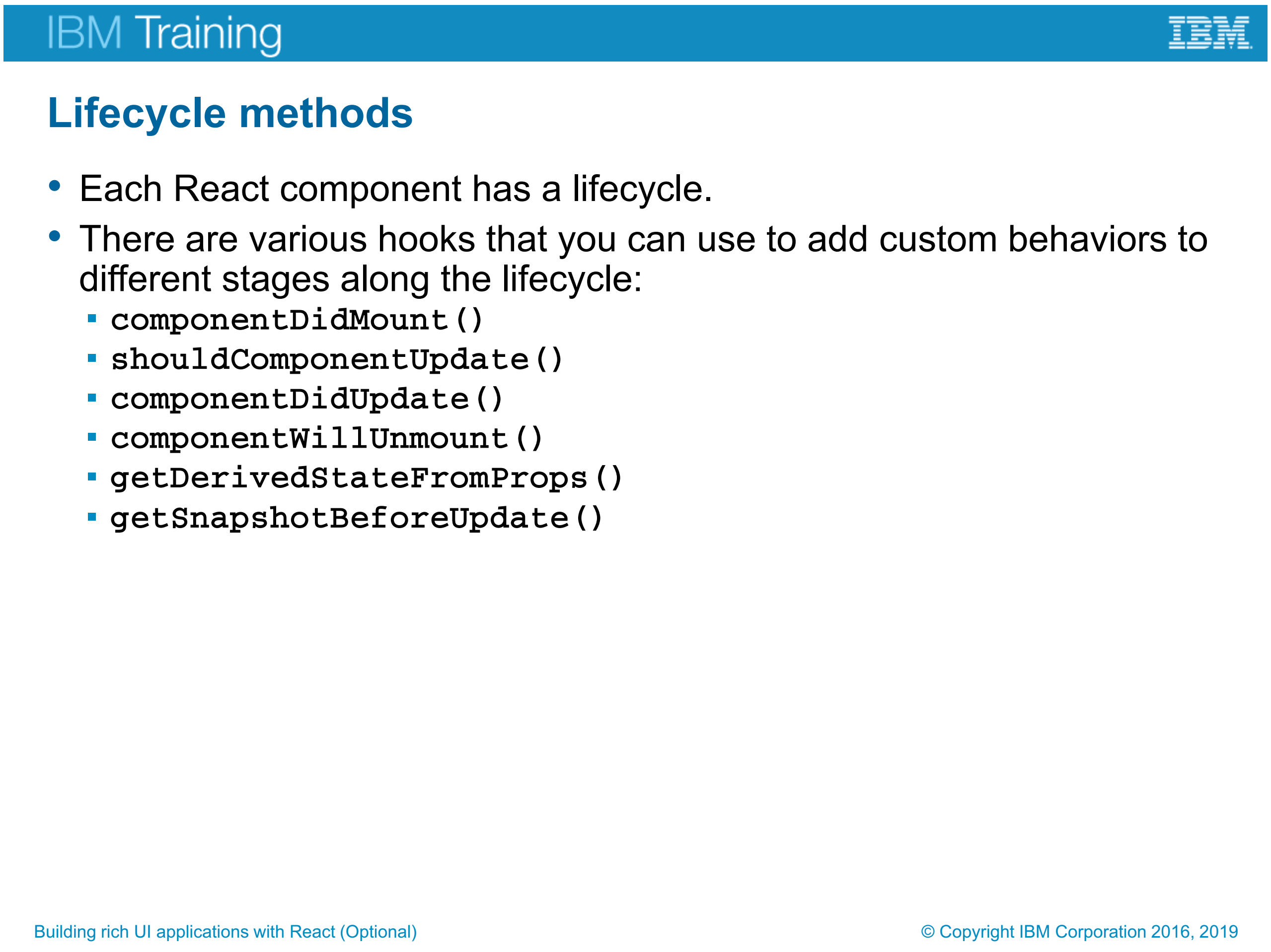
IBM Training

Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- ▶ Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

Figure 5-27. Topics


IBM Training

Lifecycle methods

- Each React component has a lifecycle.
- There are various hooks that you can use to add custom behaviors to different stages along the lifecycle:
  - `componentDidMount()`
  - `shouldComponentUpdate()`
  - `componentDidUpdate()`
  - `componentWillUnmount()`
  - `getDerivedStateFromProps()`
  - `getSnapshotBeforeUpdate()`

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-28. Lifecycle methods

- Each React component has a lifecycle.
- There are various hooks that you can use to add custom behaviors to different stages along the lifecycle:
  - **componentDidMount**: Run after initial render. The component can access the DOM.

    Can be used to call API endpoints for data.
  - **shouldComponentUpdate:** Run when props and states are being received.

    Return false to avoid unnecessary renders to improve performance.
  - **componentDidUpdate:** Run after the component is rendered to the DOM.

    Can be used to work with DOM after the component is rendered.
  - **componentWillUnmount:** Run just before component is removed from the DOM.

    Used for cleanup.
  - **getDerivedStateFromProps:** Run before calling the render method, both on the initial mount and on subsequent updates.

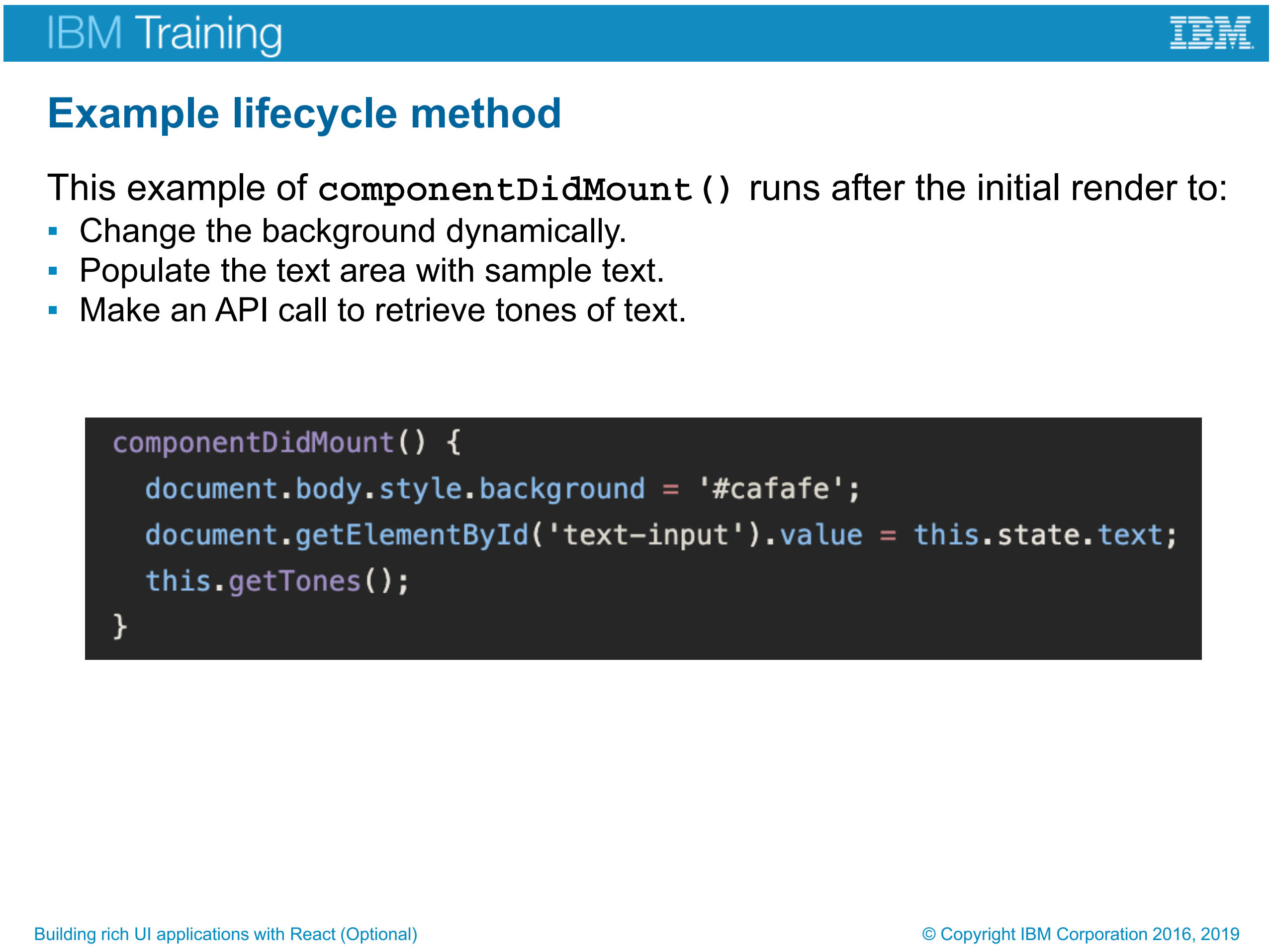
    Used to update the state before rendering.

- **getSnapshotBeforeUpdate:** Run before the most recently rendered output is committed to the DOM.

  Used for capturing information from the DOM before potential DOM changes.

For more information about React docs for components, see the following website: https://reactjs.org/docs/react-component.html

For the component lifecycle diagram, see the following website: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/



IBM Training

Example lifecycle method

This example of `componentDidMount()` runs after the initial render to:

- Change the background dynamically.
- Populate the text area with sample text.
- Make an API call to retrieve tones of text.

```
componentDidMount() {
  document.body.style.background = '#cafafe';
  document.getElementById('text-input').value = this.state.text;
  this.getTones();
}
```

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-29. Example lifecycle method

This example of **`componentDidMount()`** runs after the initial render to:

- Change the background dynamically.
- Populate the text area with sample text.
- Make an API call to retrieve tones of text.

5.6. Dynamic child components

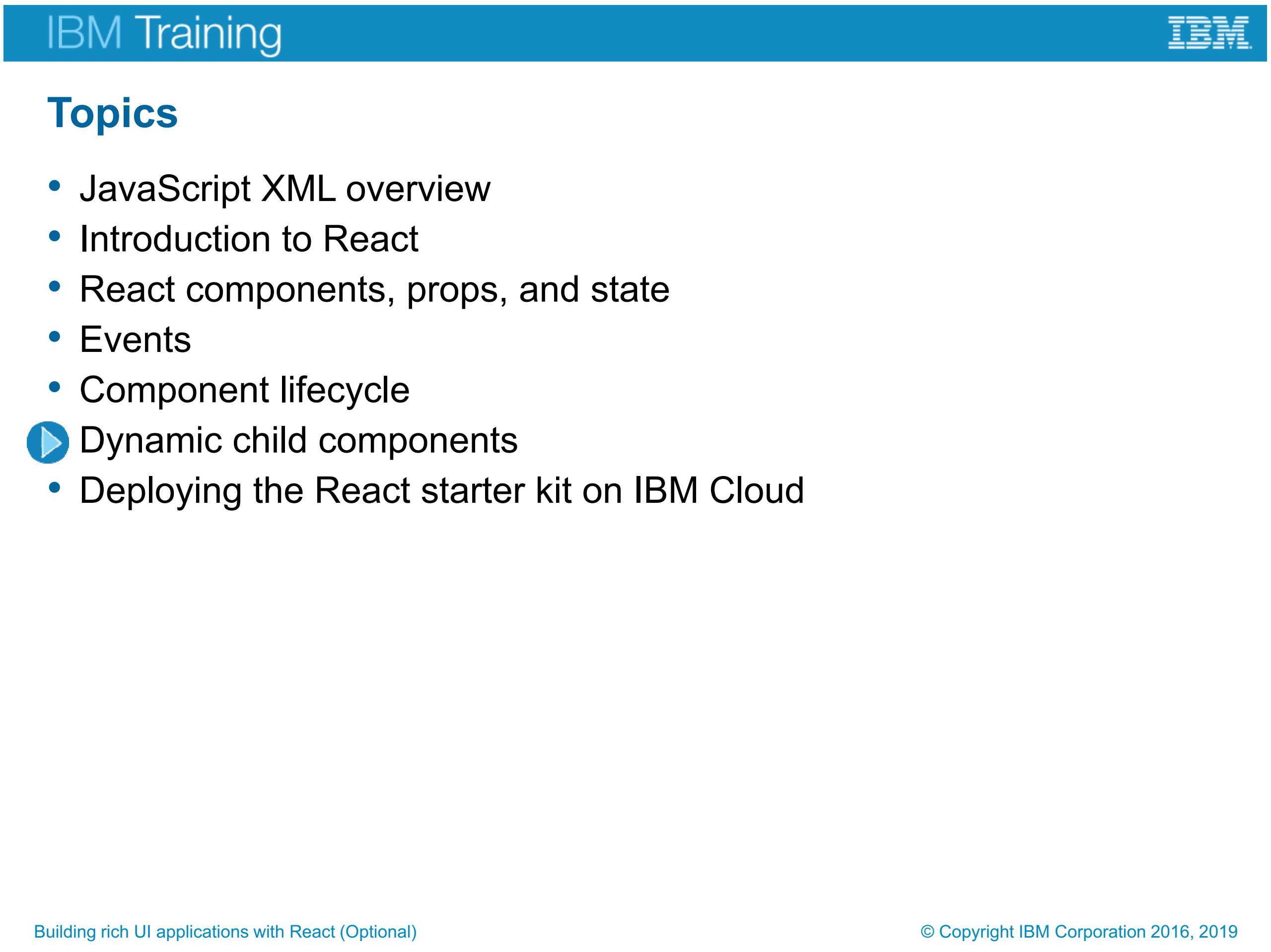
IBM Training

Dynamic child components

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-30. Dynamic child components



IBM Training

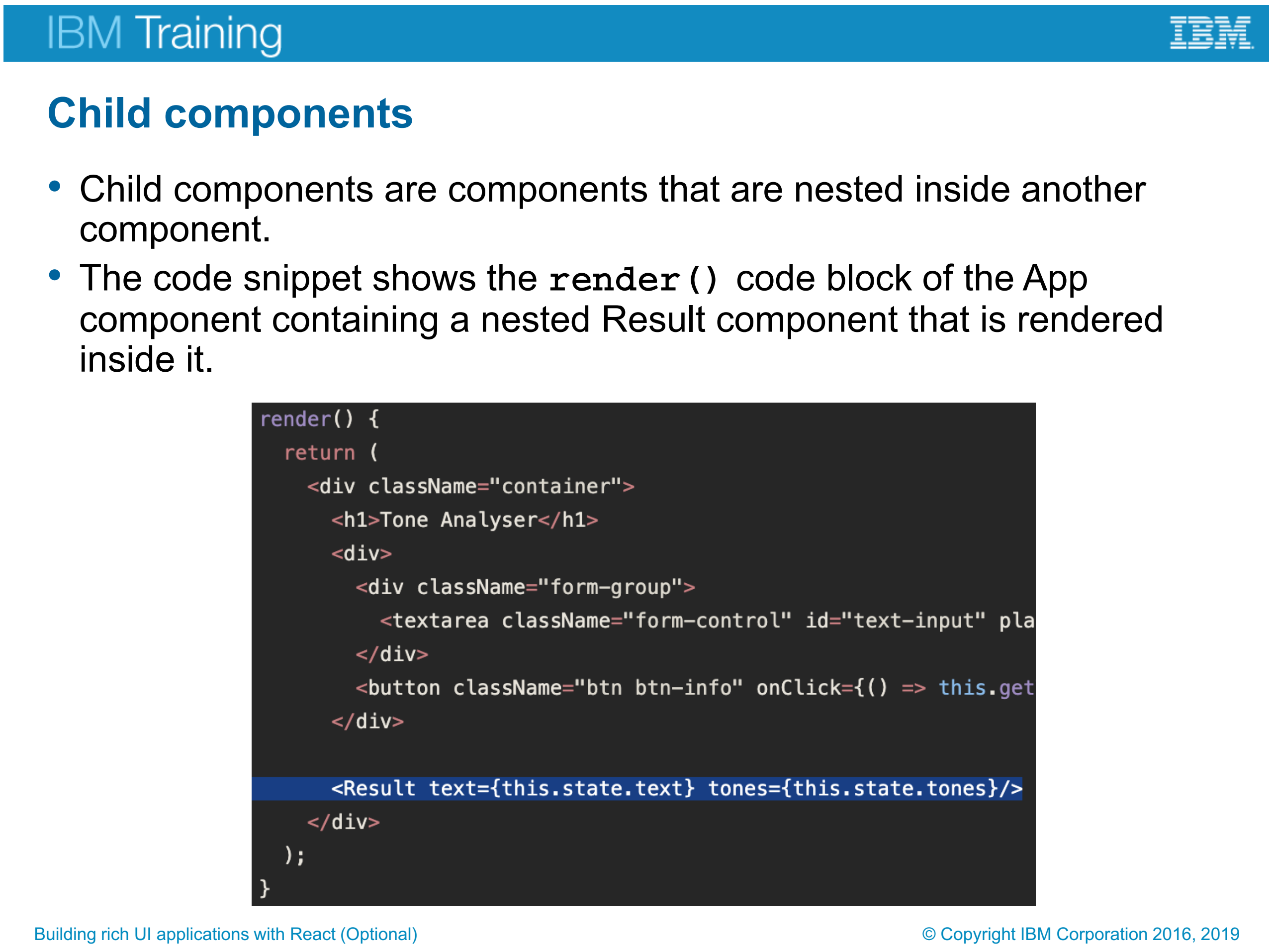
Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-31. Topics



IBM Training

Child components

- Child components are components that are nested inside another component.
- The code snippet shows the **render()** code block of the App component containing a nested Result component that is rendered inside it.

```
render() {
  return (
    <div className="container">
      <h1>Tone Analyser</h1>
      <div>
        <div className="form-group">
          <textarea className="form-control" id="text-input" pla
        </div>
        <button className="btn btn-info" onClick={() => this.get
      </div>

      <Result text={this.state.text} tones={this.state.tones}/>
    </div>
  );
}
```

Building rich UI applications with React (Optional)

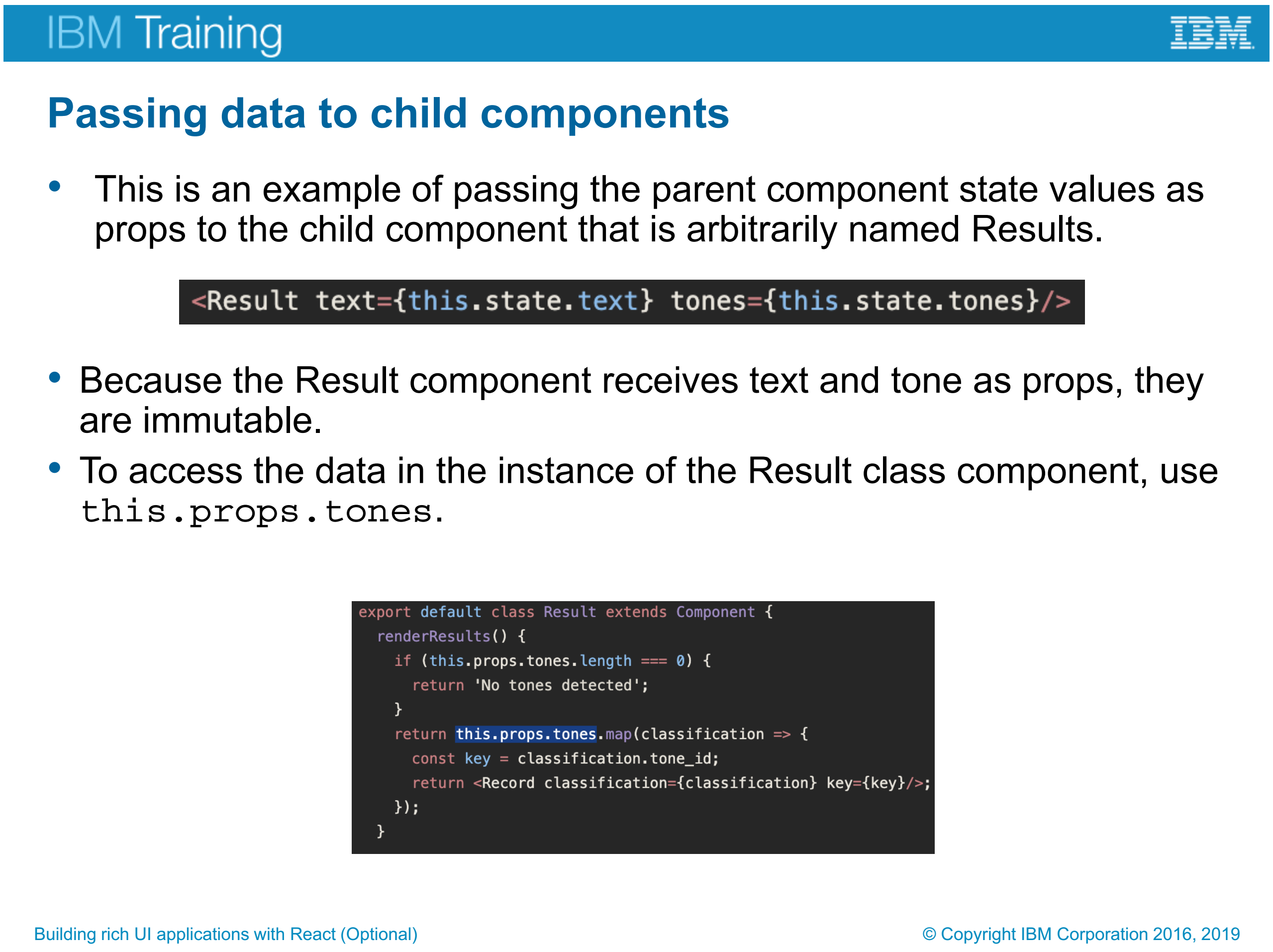
© Copyright IBM Corporation 2016, 2019

Figure 5-32. Child components

- Child components are components that are nested inside another component.
- The code snippet shows the **render()** code block of the App component containing a nested Result component that is rendered inside it.

React has a methodology and great capability to reuse code across components.

For more information about React composition and inheritance, see the following website:
https://reactjs.org/docs/composition-vs-inheritance.html



IBM Training

Passing data to child components

- This is an example of passing the parent component state values as props to the child component that is arbitrarily named Results.

```
<Result text={this.state.text} tones={this.state.tones}/>
```

- Because the Result component receives text and tone as props, they are immutable.
- To access the data in the instance of the Result class component, use `this.props.tones`.

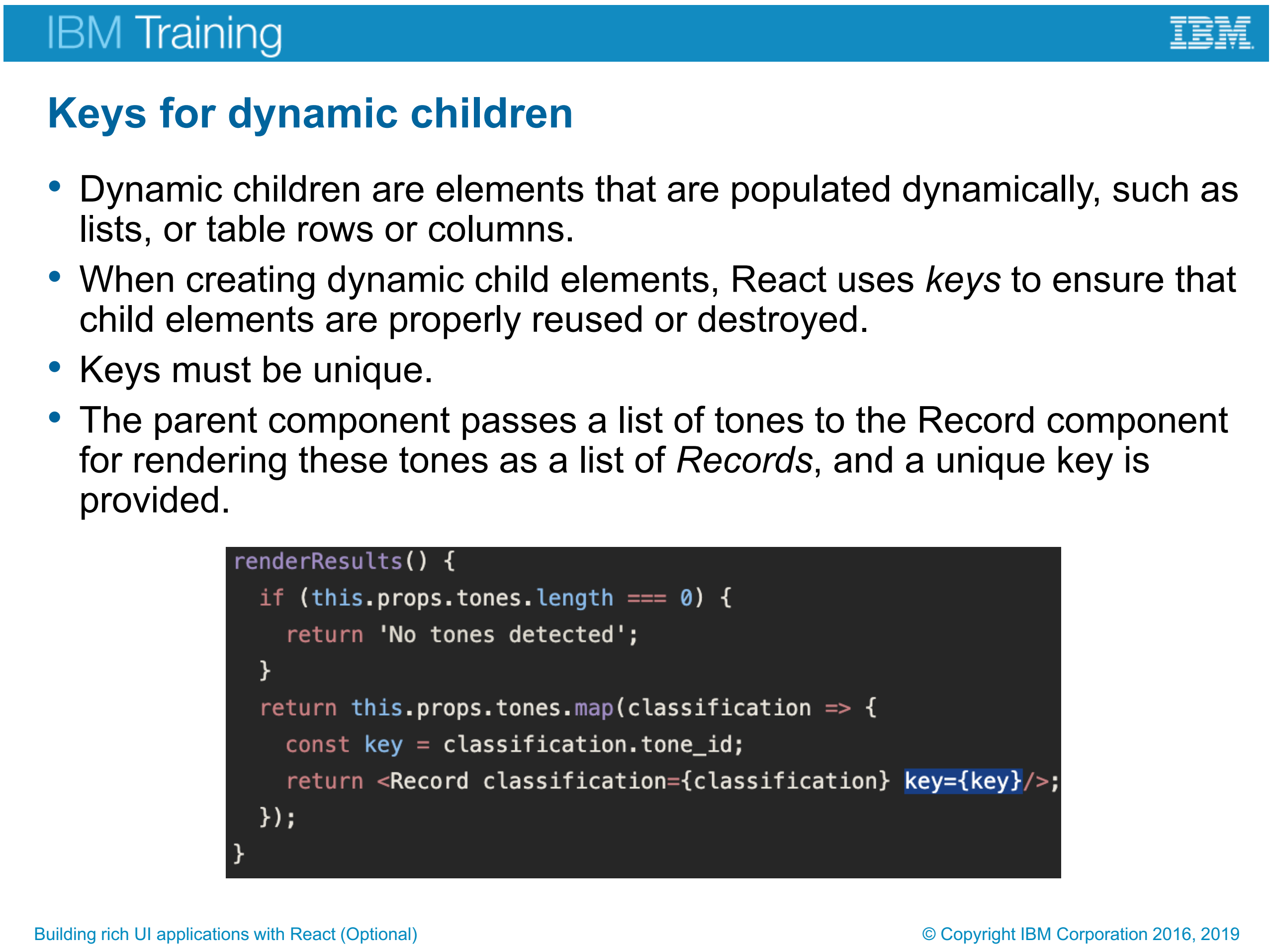
```
export default class Result extends Component {
  renderResults() {
    if (this.props.tones.length === 0) {
      return 'No tones detected';
    }
    return this.props.tones.map(classification => {
      const key = classification.tone_id;
      return <Record classification={classification} key={key}/>;
    });
  }
```

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-33. Passing data to child components

- This is an example of passing the parent component state values as props to the child component that is arbitrarily named Results.
- Because the Result component receives text and tone as props, they are immutable.
- To access the data in the instance of the Result class component, use “`this.props.tones`”.



IBM Training

Keys for dynamic children

- Dynamic children are elements that are populated dynamically, such as lists, or table rows or columns.
- When creating dynamic child elements, React uses *keys* to ensure that child elements are properly reused or destroyed.
- Keys must be unique.
- The parent component passes a list of tones to the Record component for rendering these tones as a list of *Records*, and a unique key is provided.

```
renderResults() {
  if (this.props.tones.length === 0) {
    return 'No tones detected';
  }
  return this.props.tones.map(classification => {
    const key = classification.tone_id;
    return <Record classification={classification} key={key}/>;
  });
}
```

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-34. Keys for dynamic children

- Dynamic children are elements that are populated dynamically, such as lists, or table rows or columns.
- When creating dynamic child elements, React uses keys to ensure that child elements are properly reused or destroyed.
- Keys must be unique.
- The parent component passes a list of tones to the Record component for rendering these tones as a list of Records, and a unique key is provided.

The key is provided to ensure that React can update a single element of a list, or table row or column by using the key rather than updating the entire list, or table row or column.

React does not recommend using indexes for keys because the order of items might change as data is updated. Keys are unique among siblings and do not need to be globally unique, meaning that you can use the same key identifier in two different components.

For more information about keys in React, see the following website:
https://reactjs.org/docs/lists-and-keys.html

5.7. Deploying the React starter kit on IBM Cloud

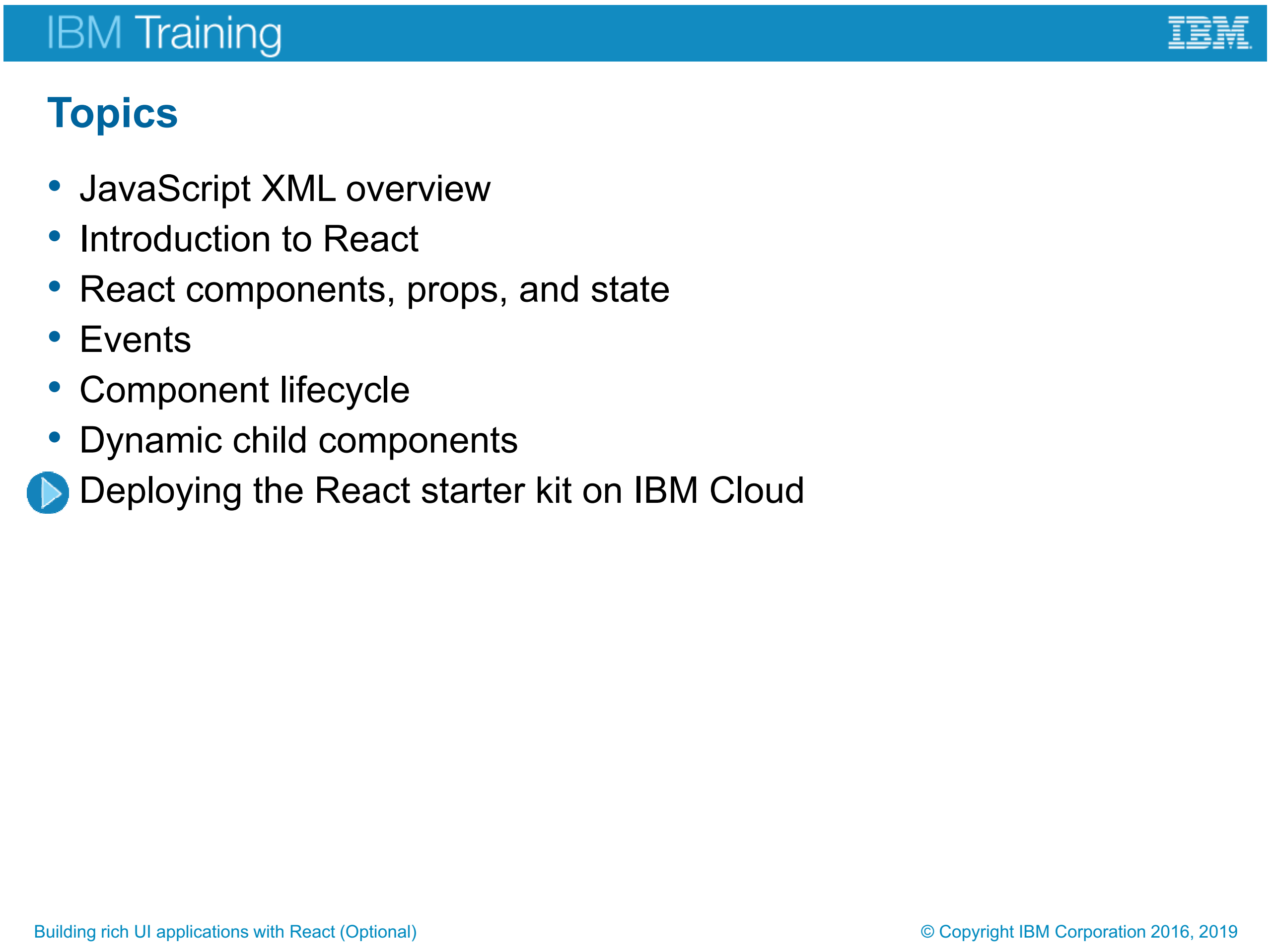
IBM Training

Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-35. Deploying the React starter kit on IBM Cloud



IBM Training

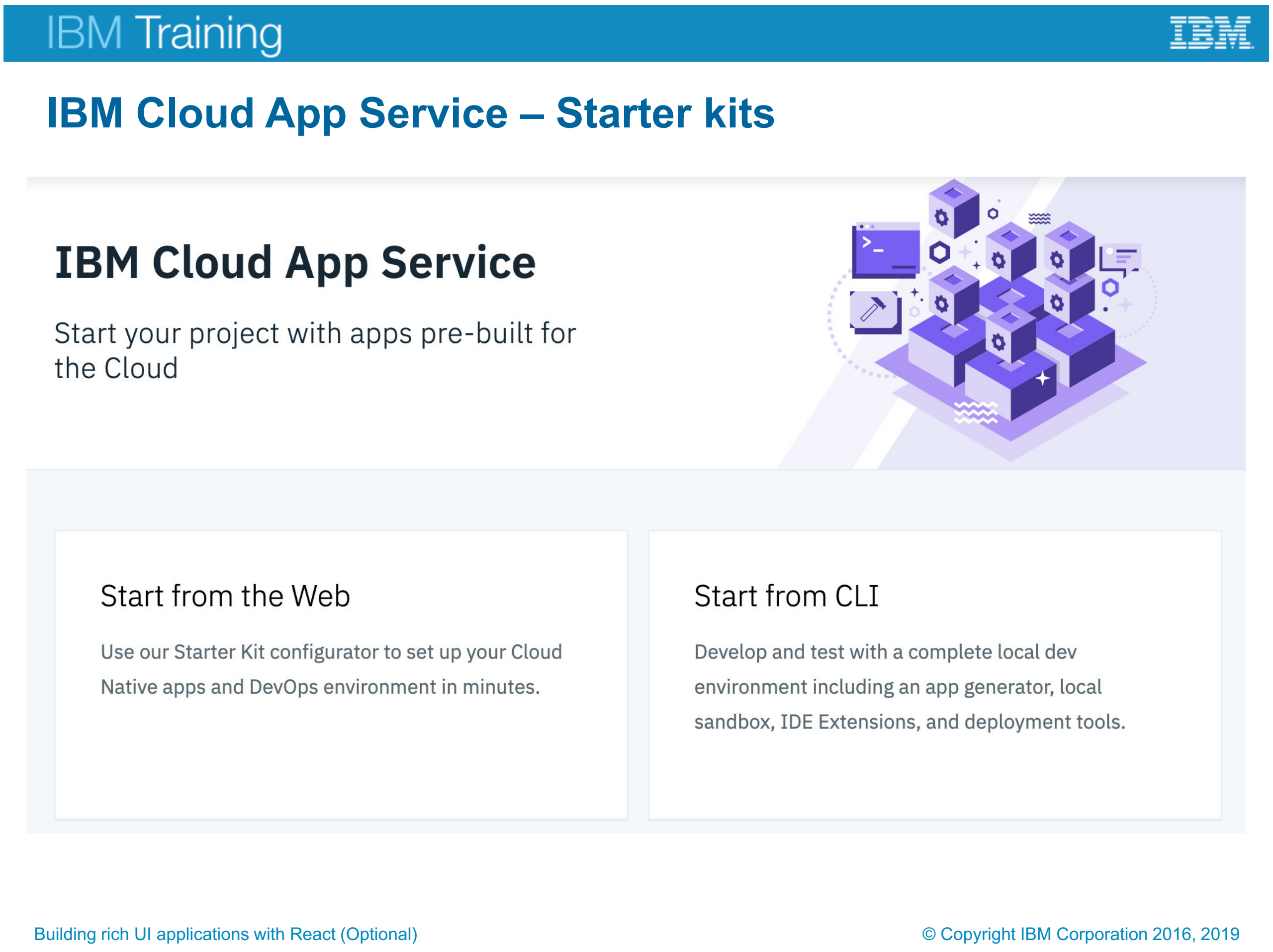
Topics

- JavaScript XML overview
- Introduction to React
- React components, props, and state
- Events
- Component lifecycle
- Dynamic child components
- ▶ Deploying the React starter kit on IBM Cloud

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-36. Topics


IBM Training

IBM Cloud App Service – Starter kits

IBM Cloud App Service

Start your project with apps pre-built for the Cloud

Start from the Web

Use our Starter Kit configurator to set up your Cloud Native apps and DevOps environment in minutes.

Start from CLI

Develop and test with a complete local dev environment including an app generator, local sandbox, IDE Extensions, and deployment tools.

Building rich UI applications with React (Optional)

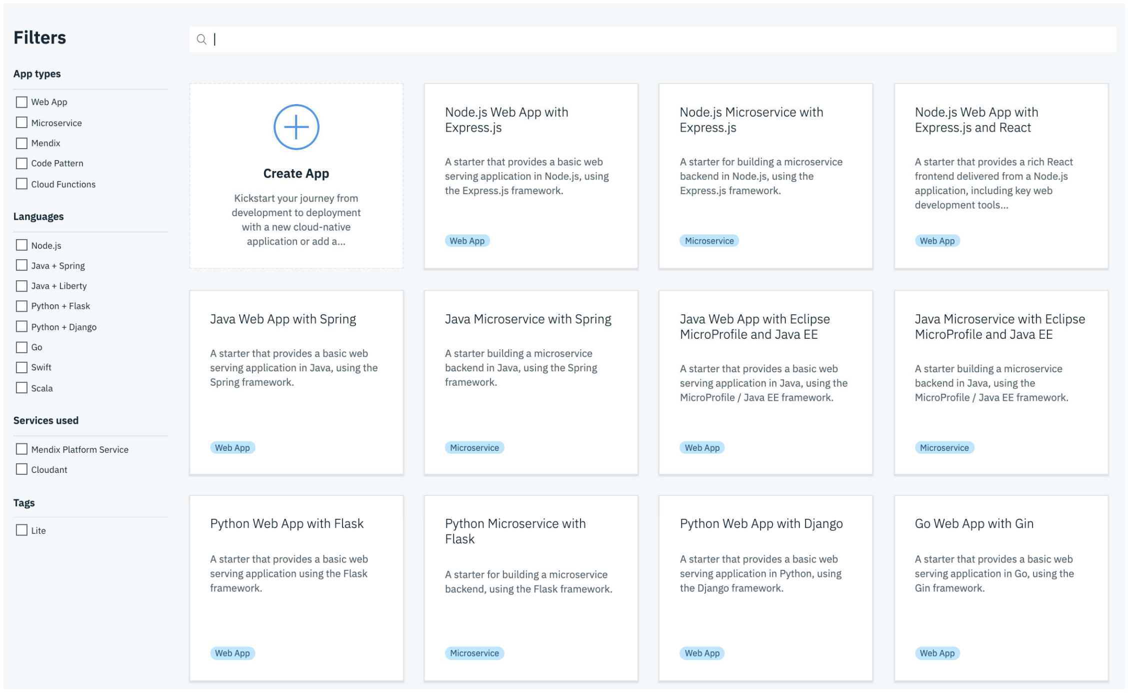
© Copyright IBM Corporation 2016, 2019

Figure 5-37. IBM Cloud App Service – Starter kits

Start your project with apps pre-built for the cloud. You can start a pre-built app through the web UI or from the command-line interface.

IBM Training

IBM Cloud App Service – Starter kits (cont.)



Filters

App types
- ☐ Web App
- ☐ Microservice
- ☐ Mendix
- ☐ Code Pattern
- ☐ Cloud Functions

Languages
- ☐ Node.js
- ☐ Java + Spring
- ☐ Java + Liberty
- ☐ Python + Flask
- ☐ Python + Django
- ☐ Go
- ☐ Swift
- ☐ Scala

Services used
- ☐ Mendix Platform Service
- ☐ Cloudant

Tags
- ☐ Lite

Create App
Kickstart your journey from development to deployment with a new cloud-native application or add a...

Node.js Web App with Express.js
A starter that provides a basic web serving application in Node.js, using the Express.js framework.
Web App

Node.js Microservice with Express.js
A starter for building a microservice backend in Node.js, using the Express.js framework.
Microservice

Node.js Web App with Express.js and React
A starter that provides a rich React frontend delivered from a Node.js application, including key web development tools...
Web App

Java Web App with Spring
A starter that provides a basic web serving application in Java, using the Spring framework.
Web App

Java Microservice with Spring
A starter building a microservice backend in Java, using the Spring framework.
Microservice

Java Web App with Eclipse MicroProfile and Java EE
A starter that provides a basic web serving application in Java, using the MicroProfile / Java EE framework.
Web App

Java Microservice with Eclipse MicroProfile and Java EE
A starter building a microservice backend in Java, using the MicroProfile / Java EE framework.
Microservice

Python Web App with Flask
A starter that provides a basic web serving application using the Flask framework.
Web App

Python Microservice with Flask
A starter for building a microservice backend, using the Flask framework.
Microservice

Python Web App with Django
A starter that provides a basic web serving application in Python, using the Django framework.
Web App

Go Web App with Gin
A starter that provides a basic web serving application in Go, using the Gin framework.
Web App

Building rich UI applications with React (Optional)

Figure 5-38. IBM Cloud App Service – Starter kits (cont.)

There are many options for the starter kits that are shown in the screen capture.

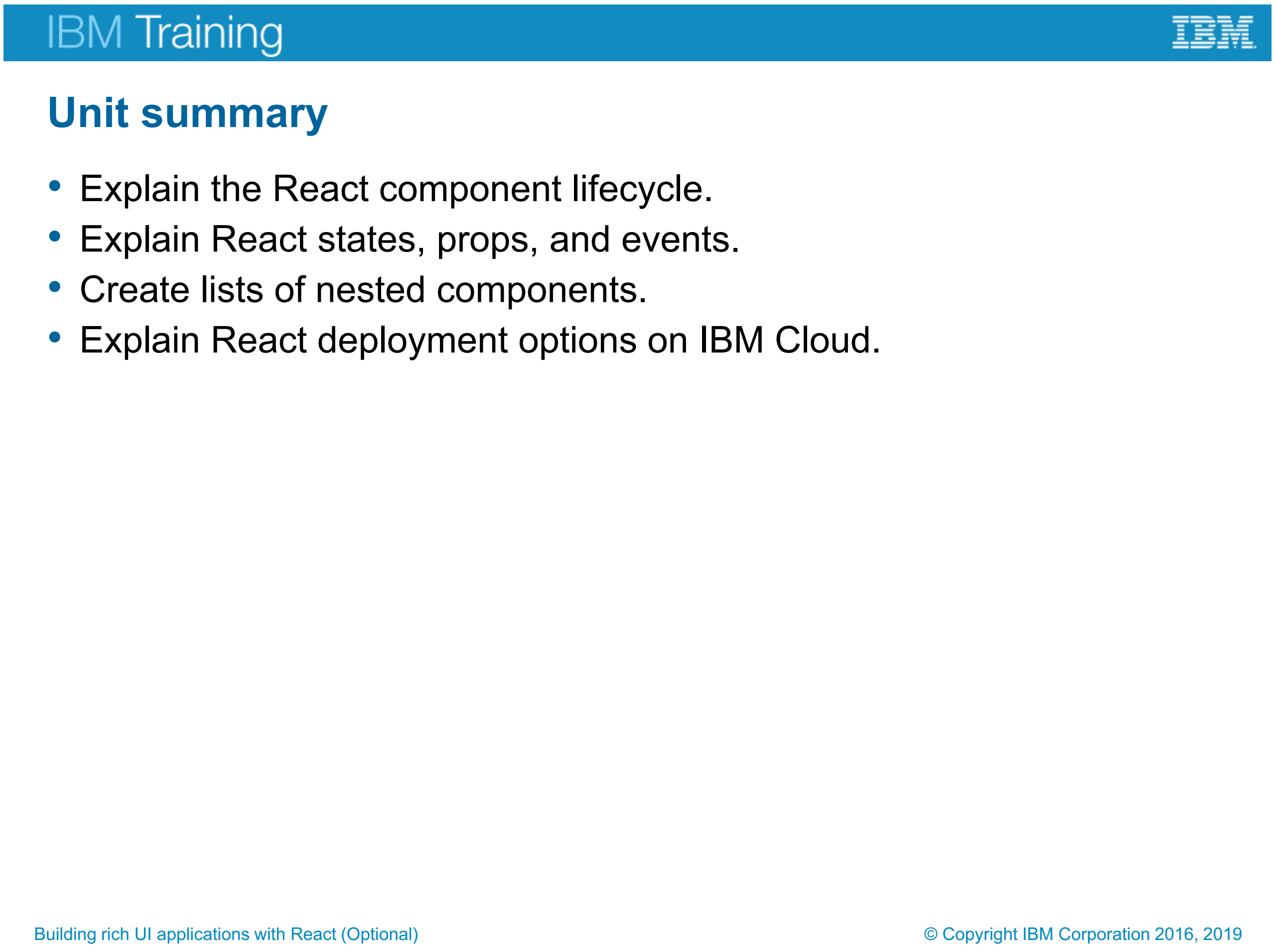
For example, you can use a pre-built application that is based on Express and React to start your development. This starter kit comes pre-configured as a Web App with Express.js product that uses the Node.js runtime. Add services; generate and download the code; use the IBM Cloud Developer Tools CLI to run and debug locally; and then deploy to Kubernetes, Cloud Foundry, or a DevOps Pipeline.

This starter kit helps you to:

- Build out the Web App architecture pattern.
- Generate an application with Express.js.
- Generate an application with files for deploying to Kubernetes, Cloud Foundry, or a DevOps Pipeline.
- Generate an application with files for monitoring and distributed trace by using Node Application Metrics.

For more information about how to use either of these technologies, see the following resources:

- Deploying applications to Cloud Foundry on IBM Cloud:
  https://www.ibm.com/cloud/cloud-foundry
- Deploying a Node.js Web App with Express.js and React starter kit on IBM Cloud:
  https://cloud.ibm.com/developer/appservice/starter-kits/nodejs-web-app-with-expressjs-and-react



IBM Training

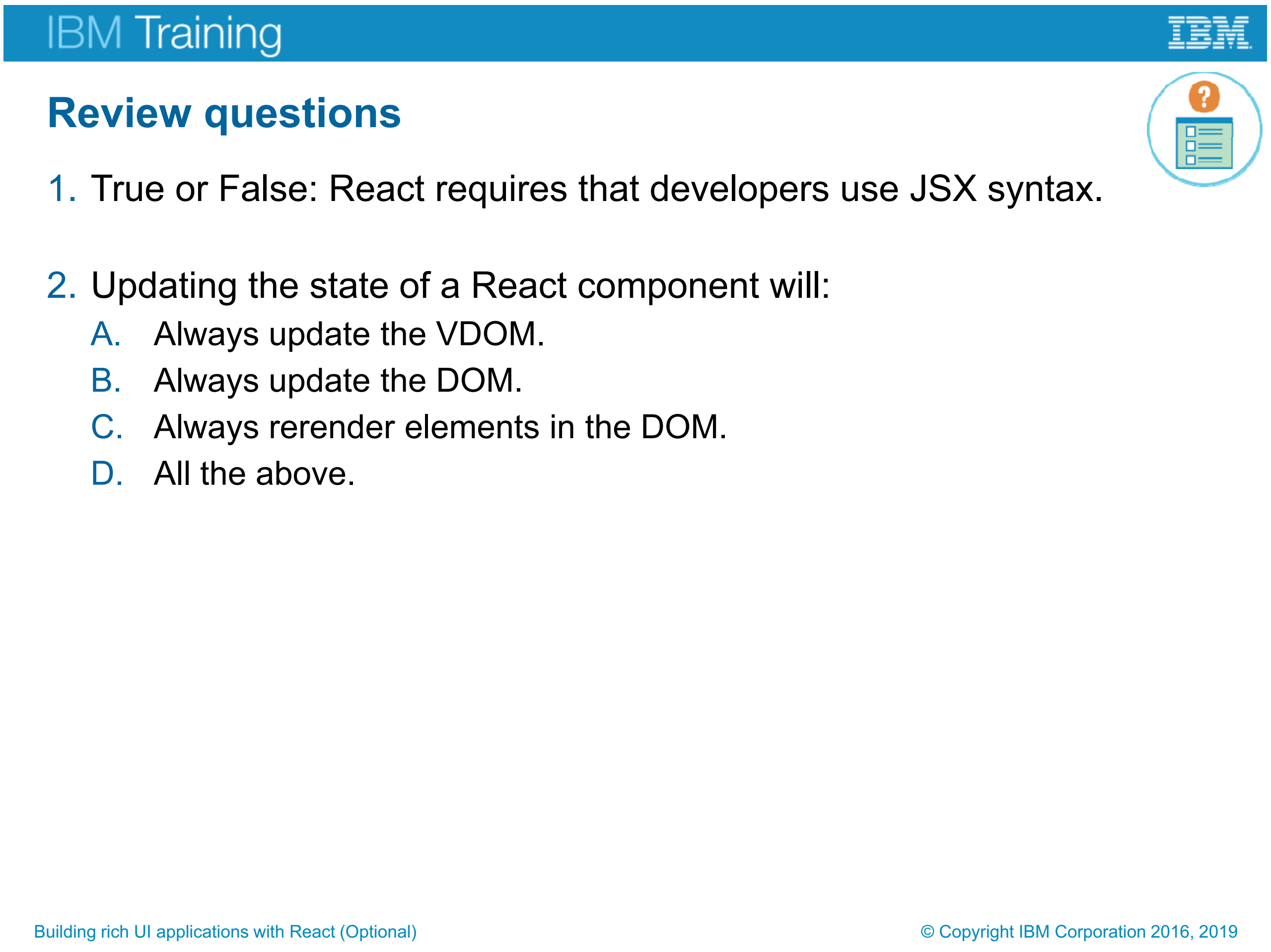
Unit summary

- Explain the React component lifecycle.
- Explain React states, props, and events.
- Create lists of nested components.
- Explain React deployment options on IBM Cloud.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-39. Unit summary



IBM Training

Review questions

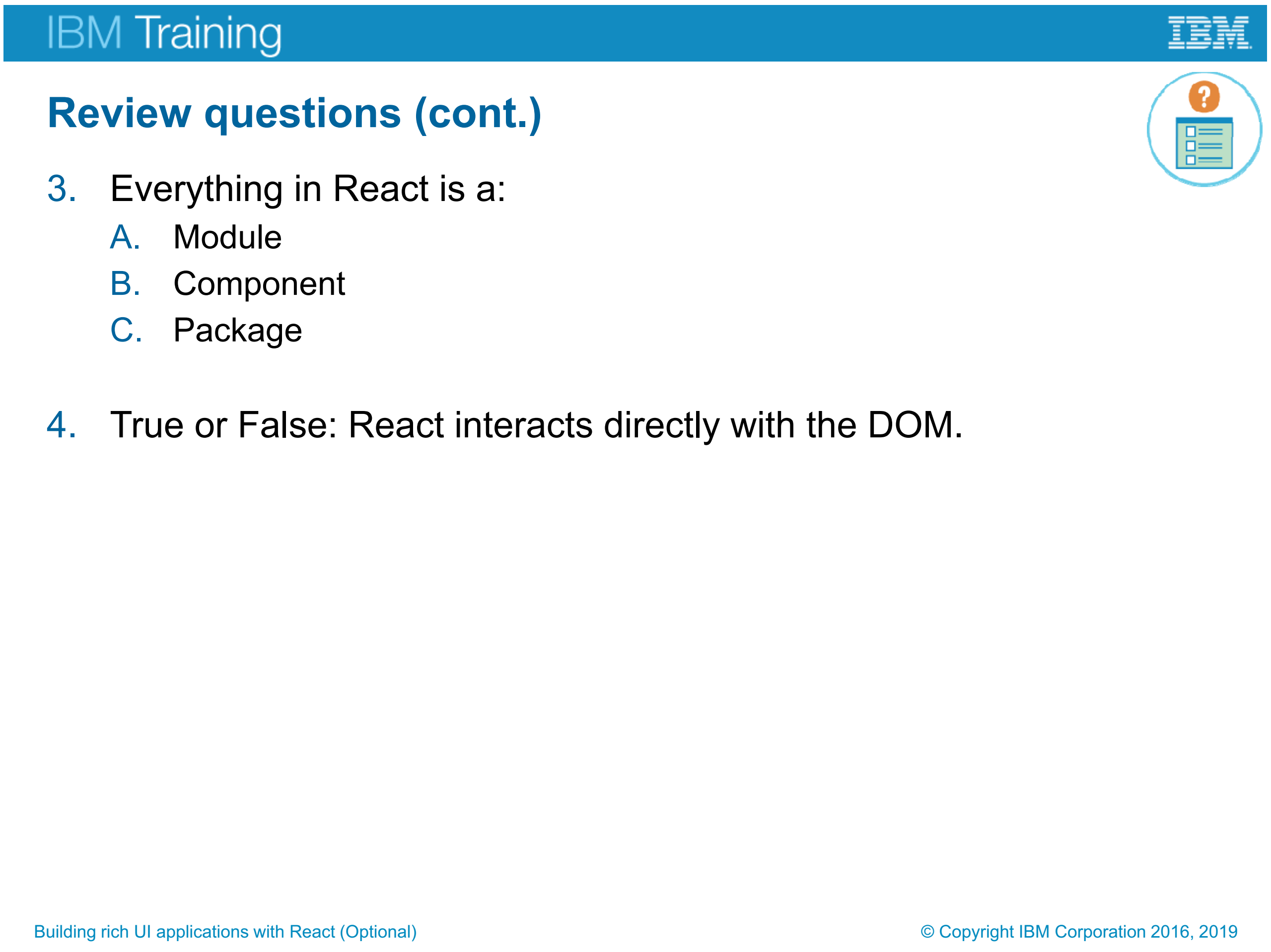
1. True or False: React requires that developers use JSX syntax.

2. Updating the state of a React component will:
   - A. Always update the VDOM.
   - B. Always update the DOM.
   - C. Always rerender elements in the DOM.
   - D. All the above.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-40. Review questions



IBM Training

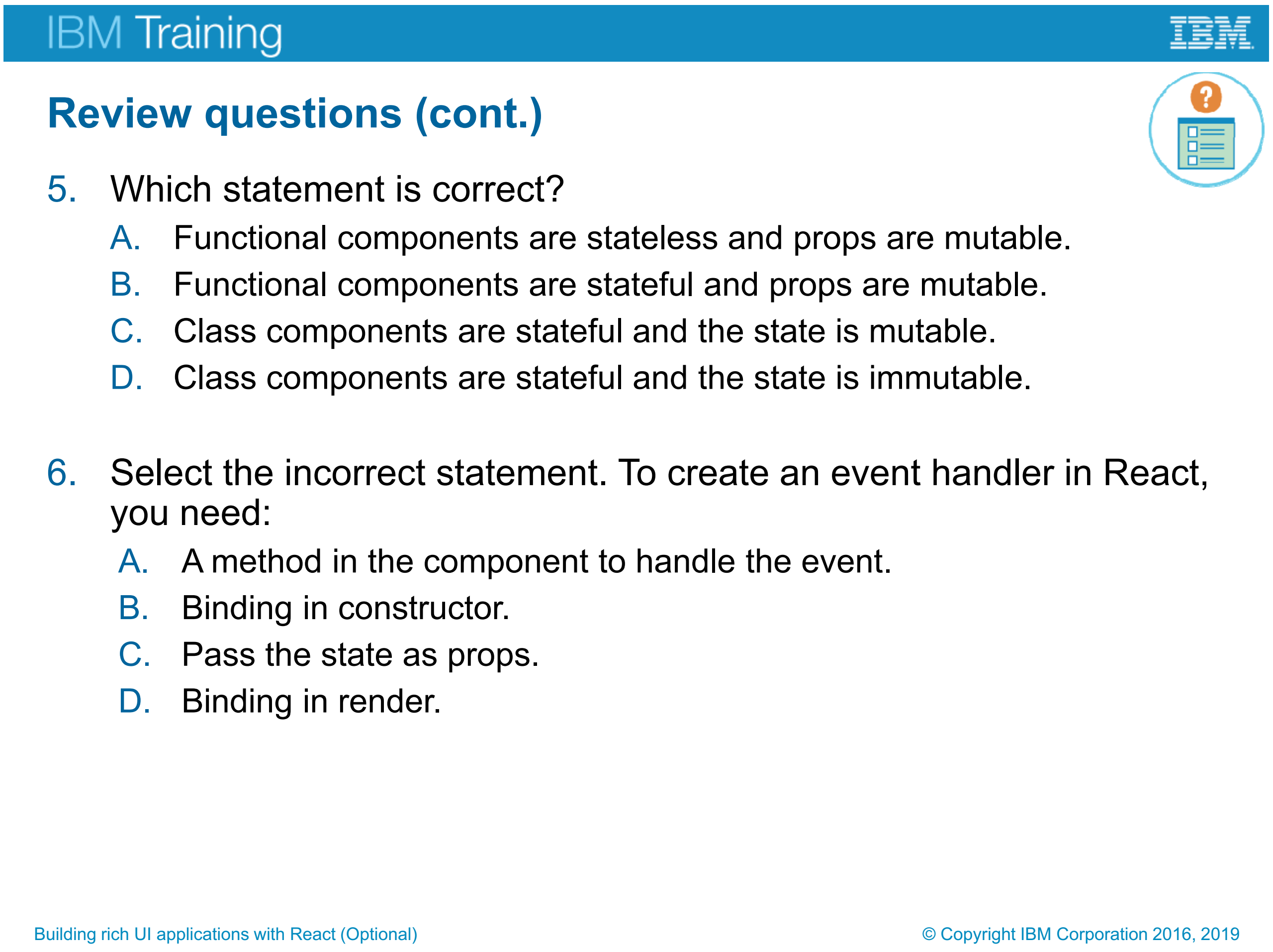
Review questions (cont.)

3. Everything in React is a:
   A. Module
   B. Component
   C. Package

4. True or False: React interacts directly with the DOM.

Building rich UI applications with React (Optional) © Copyright IBM Corporation 2016, 2019

Figure 5-41. Review questions (cont.)



IBM Training

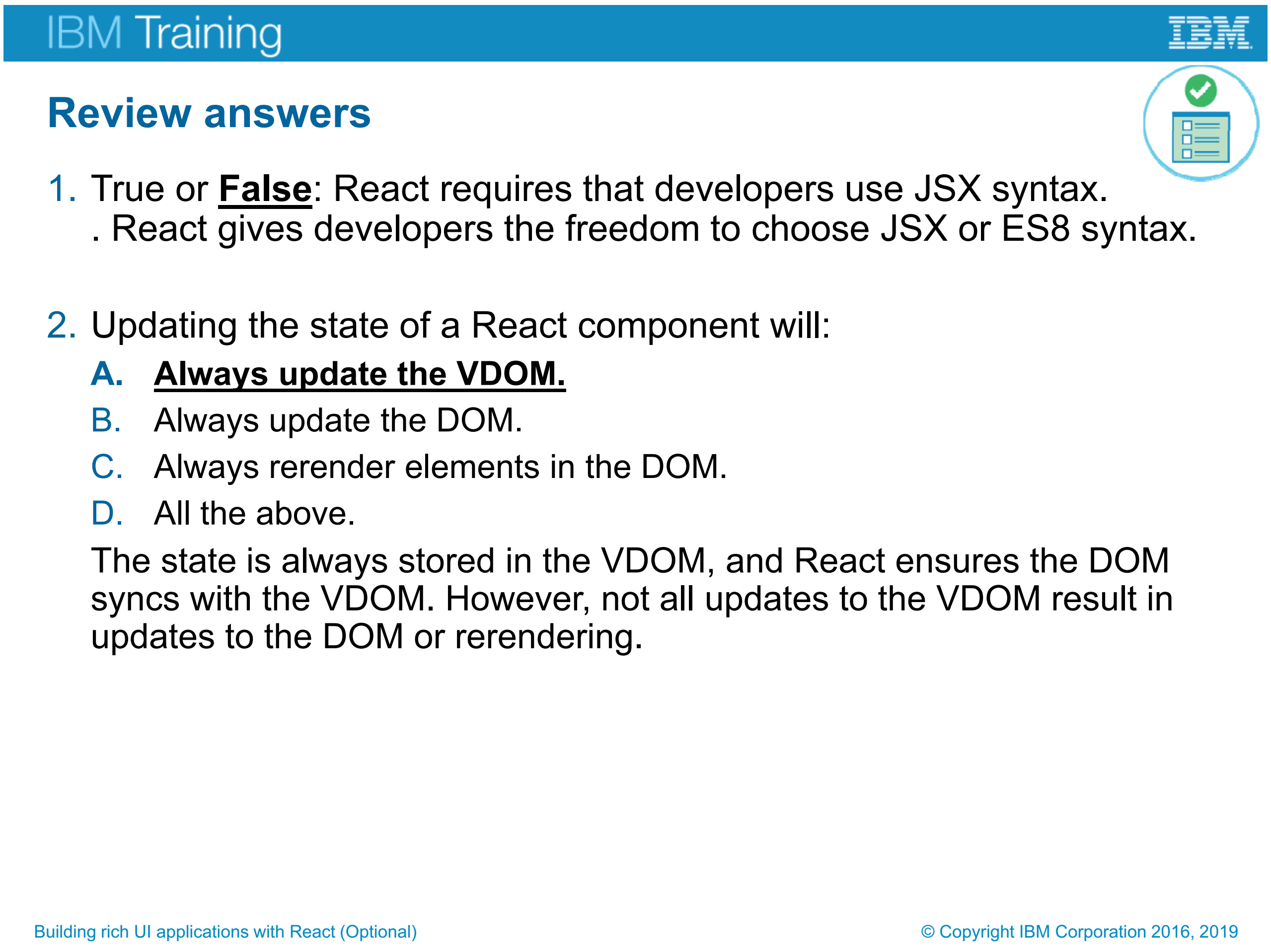
Review questions (cont.)

5. Which statement is correct?
   - A. Functional components are stateless and props are mutable.
   - B. Functional components are stateful and props are mutable.
   - C. Class components are stateful and the state is mutable.
   - D. Class components are stateful and the state is immutable.

6. Select the incorrect statement. To create an event handler in React, you need:
   - A. A method in the component to handle the event.
   - B. Binding in constructor.
   - C. Pass the state as props.
   - D. Binding in render.

Building rich UI applications with React (Optional)

Figure 5-42. Review questions (cont.)



IBM Training

Review answers

1. True or **<u>False</u>**: React requires that developers use JSX syntax.
   . React gives developers the freedom to choose JSX or ES8 syntax.

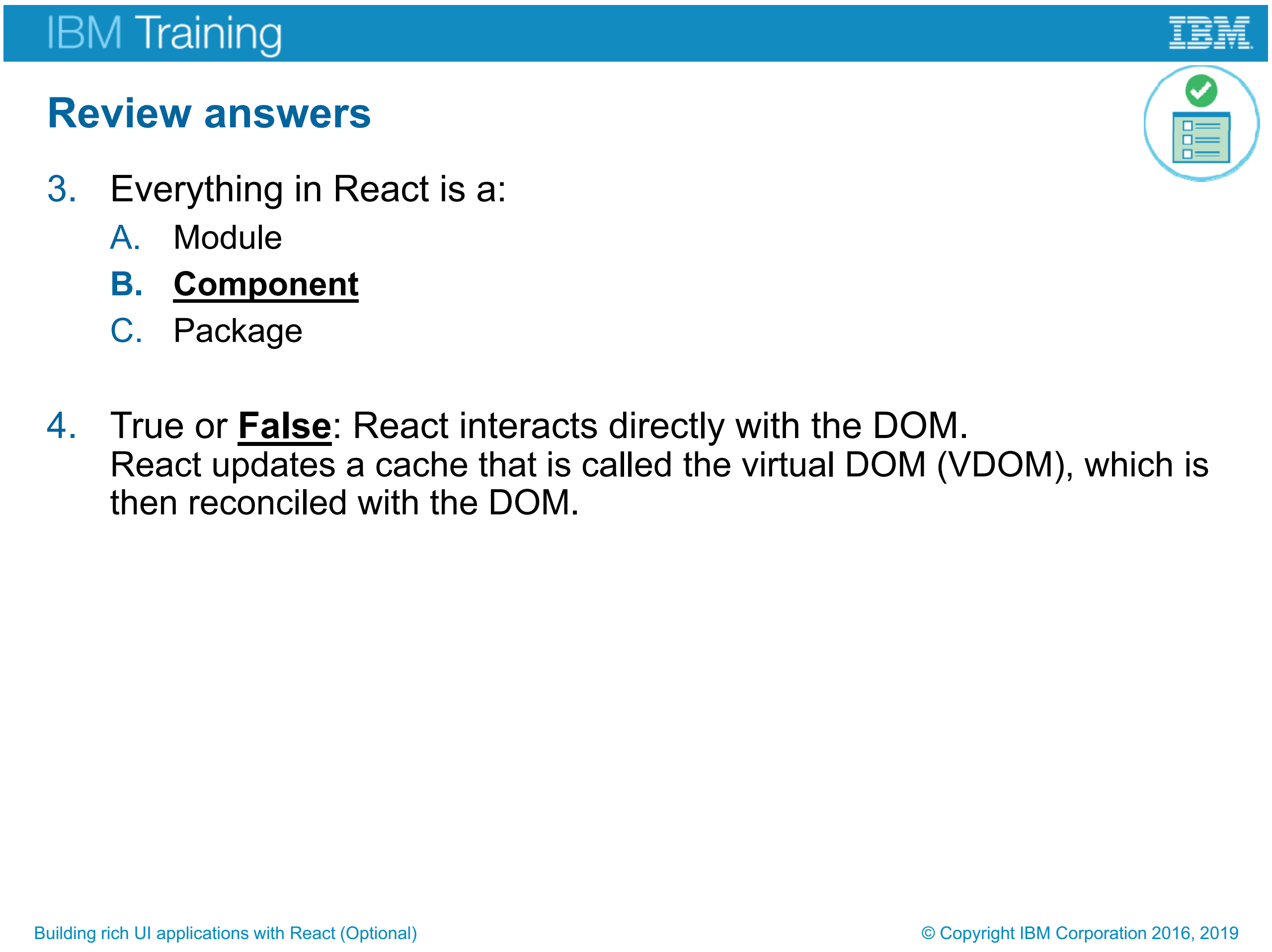
2. Updating the state of a React component will:
   A. **<u>Always update the VDOM.</u>**
   B. Always update the DOM.
   C. Always rerender elements in the DOM.
   D. All the above.

   The state is always stored in the VDOM, and React ensures the DOM syncs with the VDOM. However, not all updates to the VDOM result in updates to the DOM or rerendering.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-43. Review answers



IBM Training

Review answers

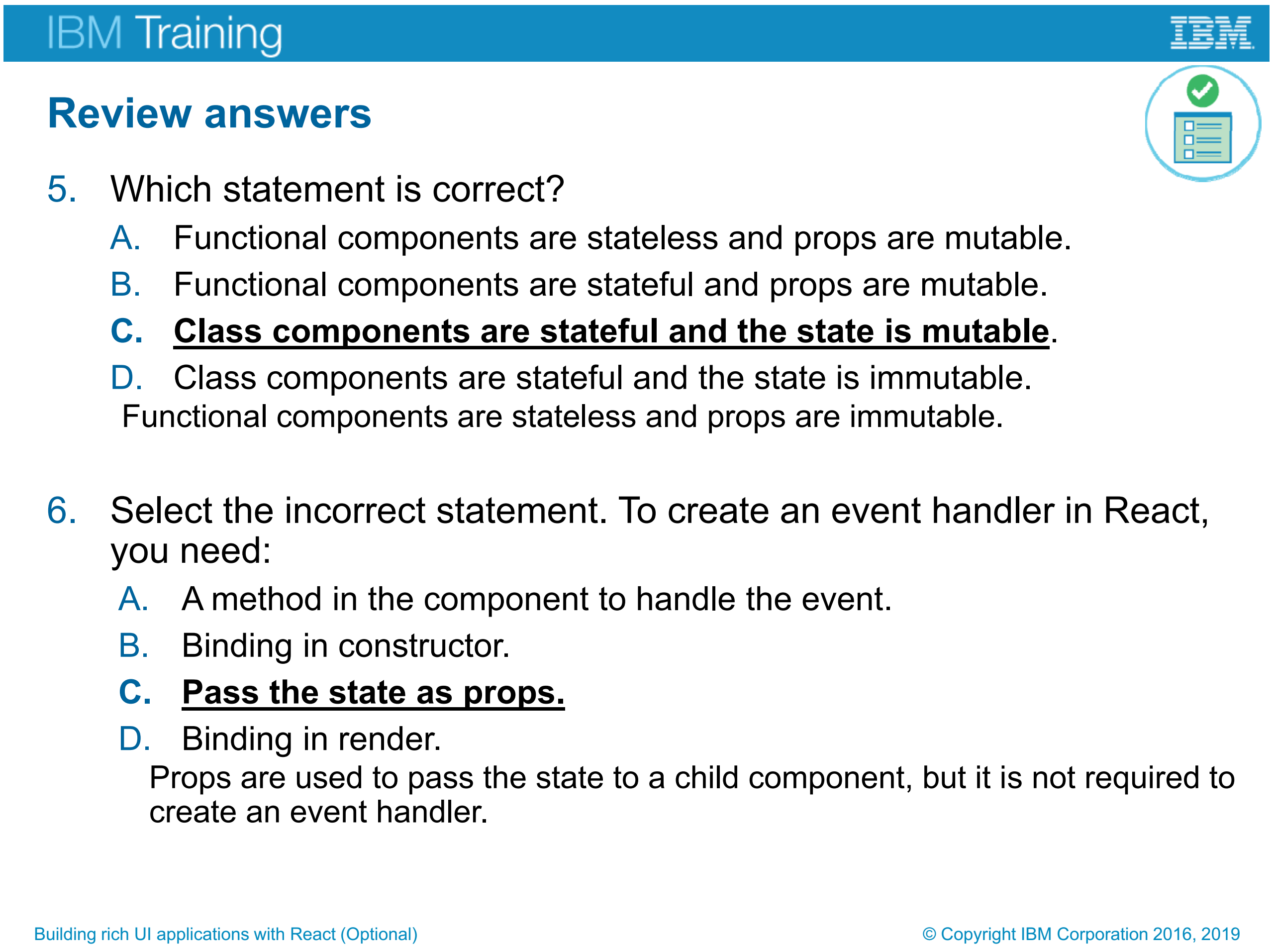
3. Everything in React is a:
   A. Module
   B. <u>Component</u>
   C. Package

4. True or **<u>False</u>**: React interacts directly with the DOM.
   React updates a cache that is called the virtual DOM (VDOM), which is then reconciled with the DOM.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-44. Review answers



IBM Training

Review answers

5. Which statement is correct?
   - A. Functional components are stateless and props are mutable.
   - B. Functional components are stateful and props are mutable.
   - **C. <u>Class components are stateful and the state is mutable</u>.**
   - D. Class components are stateful and the state is immutable.

   Functional components are stateless and props are immutable.

6. Select the incorrect statement. To create an event handler in React, you need:
   - A. A method in the component to handle the event.
   - B. Binding in constructor.
   - **C. <u>Pass the state as props.</u>**
   - D. Binding in render.

   Props are used to pass the state to a child component, but it is not required to create an event handler.

Building rich UI applications with React (Optional)

© Copyright IBM Corporation 2016, 2019

Figure 5-45. Review answers

IBM Training