

Course Guide

Developing REST APIs with Node.js

Course code VY102 ERC 6.0



May 2019 edition

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

© Copyright International Business Machines Corporation 2015, 2019.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Trademarks	vii
Course description	viii
Agenda	ix
Unit 1. Introduction to Node.js	1-1
How to check online for course material updates	1-2
Unit objectives	1-3
JavaScript: The language of web applications	1-4
JavaScript applications in the web browser	1-5
JavaScript applications in the application server	1-6
Node.js: Server-side JavaScript	1-7
IBM SDK for Node.js	1-8
Packaging Node applications	1-9
Example: Using the require function	1-10
Package.json: The module manifest	1-11
Exporting functions and properties from a module	1-12
Accessing exported properties from a module	1-13
Create a simple web server with the http package	1-14
Example: Simple HTTP server that returns a text message	1-15
1.1. What is a callback handler?	1-16
What is a callback handler?	1-17
Example: Main application calls http.request()	1-18
Unit summary	1-19
Review questions	1-20
Review answers	1-21
Exercise: Installing, verifying, and developing a Node application	1-22
Exercise objectives	1-23
Unit 2. Developing a REST API in Node	2-1
Unit objectives	2-2
2.1. Handle HTTP method calls with the Express framework	2-3
Handle HTTP method calls with the Express framework	2-4
Topics	2-5
Extending Node with packages	2-6
What is npm?	2-7
Express: A web application framework for Node	2-8
Step 1: Install the Express Node package	2-9
Example: Install and add express as a dependency	2-10
Step 2: Review dependencies in the package.json file	2-11
Example: Updated package manifest file	2-12
Step 3: Create a web application in your code	2-13
Step 4: Create an instance of the server from the app	2-14
Figure: Handle API operations with the express framework	2-15
2.2. Invoke remote services with the Request package	2-16
Invoke remote services with the Request package	2-17
Topics	2-18
Calling a remote service	2-19
Example: Calling a remote HTTP service	2-20

Figure: Call a remote service with the request package	2-21
Request: Options and callback parameters	2-22
Example: Handling error events	2-23
Propagating errors to callback functions	2-24
Build a REST API that calls other web services	2-25
Example: Define an API operation that calls a service	2-26
Figure: Define an API operation that calls remote services	2-27
2.3. Parse XML data with the xml2js package	2-29
Parse XML data with the xml2js package	2-30
Topics	2-31
Parse XML elements to JavaScript with xml2js	2-32
Advantages and disadvantages of treating XML data as a string	2-33
Step 1: Install third-party packages with npm	2-34
Step 2: Call the XML to JavaScript parser function	2-35
Unit summary	2-36
Review questions	2-37
Review answers	2-38
Exercise: Developing a REST API with Node.js	2-39
Exercise objectives	2-40

Unit 3. Static code analysis and unit testing 3-1

Unit objectives	3-2
3.1. Static code analysis with linting utilities	3-3
Static code analysis with linting utilities	3-4
Topics	3-5
JavaScript code validator	3-6
JavaScript lint utilities	3-7
JSHint (1 of 2)	3-8
JSHint (2 of 2)	3-9
ESLint installation	3-10
ESLint initial configuration	3-11
ESLint configuration file	3-12
ESLint command line	3-13
ESLint sample output	3-14
ESLint edit rule	3-15
3.2. Testing with Mocha	3-16
Testing with Mocha	3-17
Topics	3-18
Mocha	3-19
Mocha installation	3-20
Mocha test directory	3-21
Example source code that is used for a Mocha test	3-22
Chai assertion library	3-23
Mocha BDD testing	3-24
Mocha functions	3-25
Using Mocha functions	3-26
Mocha BDD code	3-27
Mocha test example with expect.js	3-28
Mocha test on invalid source code	3-29
3.3. Supertest	3-30
Supertest	3-31
Topics	3-32
Supertest	3-33
Supertest installation	3-34
API: Super-agent methods	3-35
Test case: /api/weather/KSFO	3-36

Create a unit test with Mocha and supertest	3-37
Unit test source code	3-38
Output from the unit test	3-39
Create a unit test for the error condition	3-40
Unit test error source code	3-41
Output from the error unit test	3-42
Unit summary	3-43
Review questions	3-44
Review answers	3-45
Exercise: Static code analysis and unit testing	3-46
Exercise objectives	3-47

Unit 4. Debugging and building Node applications	4-1
Unit objectives	4-2
4.1. REPL	4-3
REPL	4-4
Topics	4-5
Read, Evaluate, Print, Loop (1 of 2)	4-6
Read, Evaluate, Print, Loop (2 of 2)	4-7
4.2. Using the console	4-8
Using the console	4-9
Topics	4-10
Using the console to log messages (1 of 4)	4-11
Using the console to log messages (2 of 4)	4-12
Using the console to log messages (3 of 4)	4-13
Using the console to log messages (4 of 4)	4-14
Console output for web applications	4-15
4.3. Node built-in debugger	4-16
Node built-in debugger	4-17
Topics	4-18
Node built-in debugging utility	4-19
Node built-in debugging example	4-20
Use debug with REPL	4-21
4.4. Node Inspector	4-22
Node Inspector	4-23
Topics	4-24
Using Node Inspector	4-25
Start Node Inspector	4-26
Node Inspector in the Chrome browser	4-27
Node Inspector features	4-28
Node Inspector hover feature	4-29
Node Inspector with multiple files open	4-30
Manage debug execution with controls	4-31
4.5. Shrinkwrap	4-32
Shrinkwrap	4-33
Topics	4-34
The purpose of npm shrinkwrap	4-35
Building shrinkwrapped packages	4-37
Updating shrinkwrap packages	4-38
Shrinkwrap example	4-39
4.6. Build	4-40
Build	4-41
Topics	4-42
Use npm as a build tool	4-43
How npm manages scripts	4-44
Run scripts	4-45

Chain scripts	4-46
Unit summary	4-47
Review questions	4-48
Review answers	4-49
Exercise: Debugging and building node applications	4-50
Exercise objectives	4-51
Unit 5. Deploying and testing Node applications on IBM Cloud	5-1
Unit objectives	5-2
Topics	5-3
What is IBM Cloud?	5-4
What kind of infrastructure can you build on IBM Cloud?	5-5
What is Cloud Foundry?	5-6
Getting started: Create an IBM Cloud account	5-7
Topics	5-8
What are the IBM Cloud Developer Tools?	5-9
Steps to deploy your application to IBM Cloud	5-10
Step 1: Install the IBM Cloud Developer Tools CLI	5-11
Step 2: Enable the application to make it Cloud ready	5-12
Step 3: Review your application files	5-13
Review the generated manifest file	5-14
Example: Cloud Foundry manifest file	5-15
Step 4: Log in to your Cloud account	5-16
Step 5: Deploy your application to your Cloud account	5-17
Examine the deployment health and status	5-18
Step 6: Test your Cloud application	5-19
Topics	5-20
IBM Cloud dashboard	5-21
IBM Cloud dashboard: Application status	5-22
IBM Cloud dashboard: Application logs	5-23
IBM Cloud dashboard: Overview	5-24
Unit summary	5-25
Review questions	5-26
Review answers	5-27
Exercise: Deploying a REST API on IBM Cloud	5-28
Exercise objectives	5-29
Unit 6. Course summary	6-1
Unit objectives	6-2
Course learning objectives	6-3
Course learning objectives	6-4
Other learning resources (1 of 4)	6-5
Other learning resources (2 of 4)	6-6
Other learning resources (3 of 4)	6-7
Other learning resources (4 of 4)	6-8
Appendix A. List of abbreviations	A-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

Bluemix®

DB™

Express®

Notes®

z/OS®

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

LoopBack® is a trademark or registered trademark of StrongLoop, Inc., an IBM Company.

Other product and service names might be trademarks of IBM or other companies.

Course description

Developing REST APIs with Node.js

Duration: 2 days

Purpose

This course teaches developers how to create, test, and deploy applications with Node.js. The Node.js runtime environment is a highly scalable server-side application platform. In this course, you learn how to develop REST applications with Express, a popular web application framework for Node. You design callback functions to handle asynchronous network events. You also install and manage Node features with npm, the packaging manager for Node modules. You build, test, and deploy the lab exercises in your own workstation. You optionally deploy the application on the IBM Cloud with your own IBM Cloud account.

Audience

This course is designed for API developers who want to build REST applications with the Node.js server runtime environment.

Prerequisites

- Working knowledge of JavaScript programming
- Familiarity with web application architecture and REST API concepts

Objectives

- Install, validate, and test the Node runtime environment on your local workstation
- Install Node packages with npm
- Develop REST API operations with Express
- Develop callback functions to handle asynchronous events
- Perform static code analysis of the application with ESLint
- Run Mocha and Supertest unit tests on Node applications
- Debug Node applications with the Google Chrome browser with Node inspector
- Package Node applications
- Deploy Node applications to IBM Cloud with the IBM Cloud command-line utility
- Run Node applications on IBM Cloud

Agenda

**Note**

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

Day 1

- (00:15) Course introduction
- (00:45) Unit 1. Introduction to Node.js
- (00:45) Exercise 1. Installing, verifying, and developing a Node application
- (00:45) Unit 2. Developing a REST API in Node
- (01:00) Exercise 2. Developing a REST API with Node.js
- (00:45) Unit 3. Static code analysis and unit testing

Day 2

- (01:30) Exercise 3. Static code analysis and unit testing
- (00:45) Unit 4. Debugging and building Node applications
- (01:00) Exercise 4. Debugging and building Node applications
- (01:00) Unit 5. Deploying and testing Node applications on IBM Cloud
- (00:45) Exercise 5. Deploying a REST API on IBM Cloud (optional)
- (00:10) Unit 6. Course summary

Unit 1. Introduction to Node.js

Estimated time

00:45

Overview

This unit explains the motivation and purpose of Node.js, an event-driven JavaScript web application server framework, and runtime. It explores how to easily write web applications in JavaScript, and describes how to create a simple web server in Node.js on your workstation.

How you will check your progress

- Review questions
- Lab exercise

How to check online for course material updates



Note: If your classroom does not have internet access, ask your instructor for more information.

Instructions

1. Enter this URL in your browser:
<http://ibm.biz/CloudEduCourses>
2. On the wiki page, locate and click the **Course Information** category.
3. Find your course in the list and then click the link.
4. The wiki page displays information for the course. If an errata document is available, this page is where it is found.
5. If you want to download an attachment, such as an errata document, click the **Attachments** tab at the bottom of the page.

Comments (0)	Versions (1)	Attachments (1)	About
--------------	--------------	------------------------	-------

6. To save the file to your computer, click the document link and follow the dialog box prompts.

Figure 1-1. How to check online for course material updates

Unit objectives

- Explain the origin and purpose of the Node.js JavaScript framework
- Write a simple web server with JavaScript
- Import Node.js modules into your script



Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-2. Unit objectives

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, interactive web applications with HTML, Cascading Style Sheets (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 Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-3. JavaScript: The language of web applications

JavaScript applications in the web browser

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

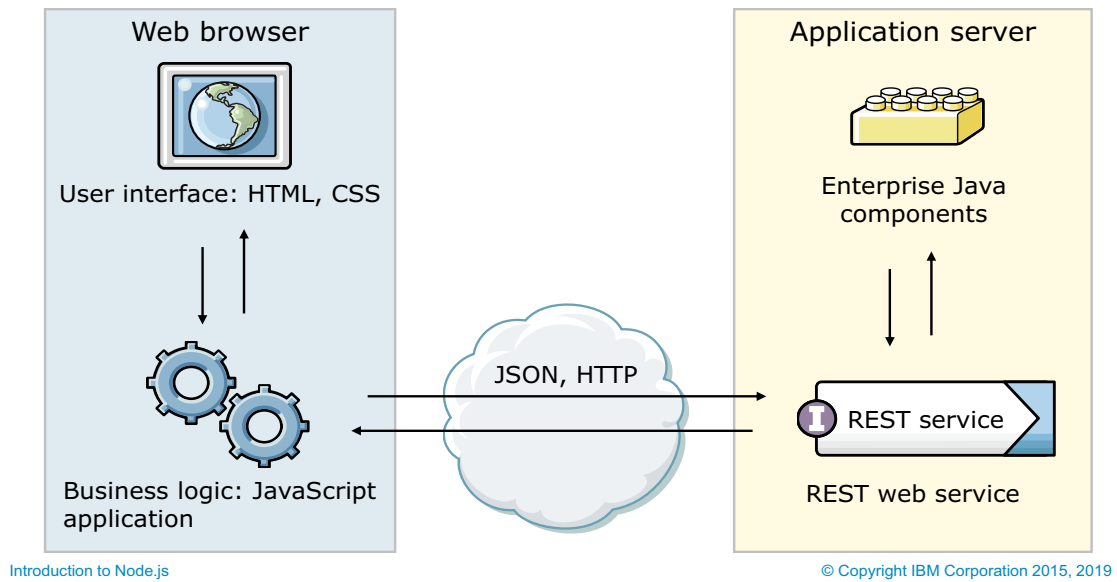


Figure 1-4. JavaScript applications in the web browser

JavaScript applications run in the web browser. With client-side JavaScript, developers create rich, interactive web applications in the web browser. In step 1, the user interface is rendered by using HTML and CSS. When the user selects an option in the web page, it triggers business logic that is written as a JavaScript application. The JavaScript application sends a web service request by using JSON over HTTP. 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, such as Enterprise Java components.

JavaScript applications in the application server

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

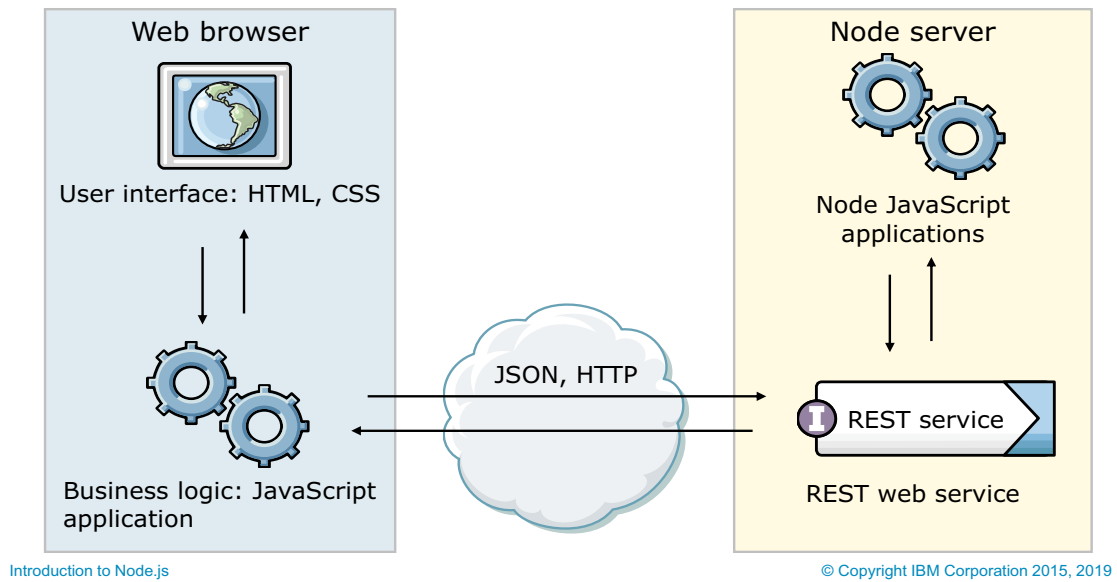


Figure 1-5. JavaScript applications in the application server

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

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

In step 1, the user selects an option in the user interface, which is written in HTML and CSS. Step 2, the option triggers a JavaScript application that implements the business logic on the client side. In step 3, the JavaScript application makes a web service call over HTTP with a data payload that is written in JSON. In the next step, a REST web service intercepts the HTTP request. In the final step, instead of invoking an Enterprise Java application, the Node server hosts an application that is written in the JavaScript language. This application runs on the server, and not in the client's web browser.

Node.js: Server-side JavaScript

- Node.js is a server-side programming framework that uses JavaScript as its programming language
 - Many developers are already familiar with the JavaScript language
- Node is built with a heavy emphasis on concurrent programming with a lightweight language
 - Node is a single-threaded application environment that handles input/output (I/O) operations through events
 - Instead of blocking on asynchronous I/O operations, you write callback functions to handle results when they complete
- Node is suited for developers who want to build scalable, concurrent server applications quickly with a minimal set of tools

Figure 1-6. Node.js: Server-side JavaScript

IBM SDK for Node.js

- The IBM SDK for Node.js is an IBM package of the Node server runtime
 - The IBM SDK for Node.js is available on various hardware platforms and operating systems
 - In addition to Microsoft Windows, Mac OS X, and Linux, IBM SDK for Node.js supports Power Systems and Linux for System z
- In this course, you build server-side applications on the IBM SDK for Node.js distribution
 - Functionally, the IBM distribution implements the same SDK as the Node.js open source project

Figure 1-7. IBM SDK for Node.js

Packaging Node applications

- A Node *module* is a package for a Node application
 - The `package.json` manifest file defines a main script for the module
- For example, use the `require` function to import a Node module

```
var today = require('./today');
```
- In this example, a Node script file that is named `today.js` in the same directory as your application
 - The `require` statement assumes that scripts have a file extension of `.js`
- The `require` function creates an object that represents the imported Node module

Figure 1-8. Packaging Node applications

A Node module is a package for a Node script file. Each module corresponds to a Node script as an entry point. The `package.json` manifest file specifies the name of the main script file. The module itself might depend on other modules.

For example, use the `require` function to import a Node module:

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

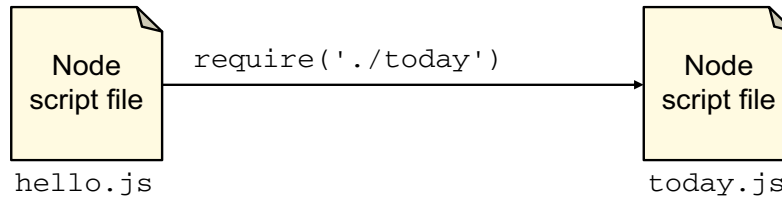
In this example, a Node script file that is named `today.js` is in the same directory as your application.

The `require` statement assumes that scripts have a file extension of `.js`.

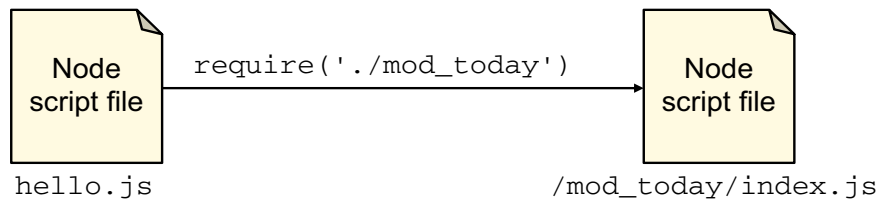
The `require` function creates an object that represents the imported Node module.

Example: Using the require function

To import a Node module that consists of a single script, use the `require` function with a relative path to the script file



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



Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-9. Example: Using the require function

When you call `require` with the name of a subdirectory, `node.js` looks for a script file with the same name as the subdirectory. For example, when you call `require('mod_today')`, the function looks for a script file named `mod_today.js`. If `mod_today.js` does not exist, it assumes that `mod_today` is the name of a directory. `Node.js` looks for a script that is named `index.js` within the `mod_today` directory.

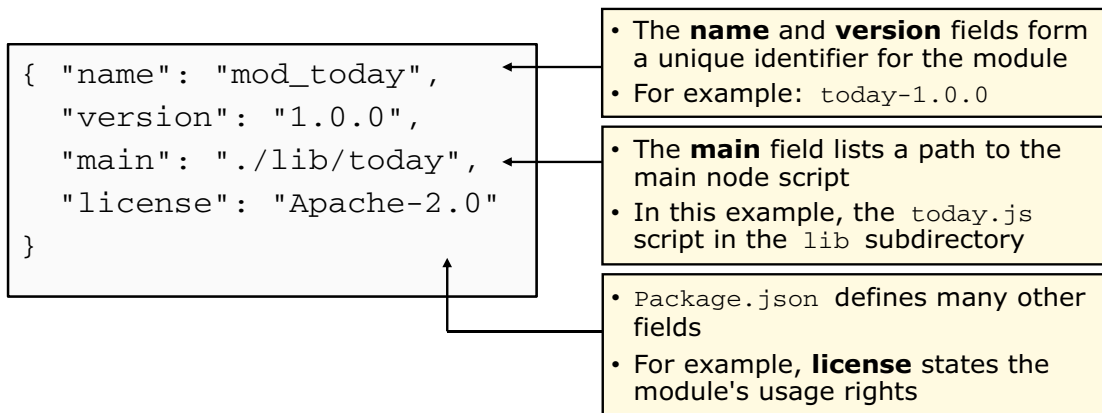
To import a Node module that consists of a single script, use the `require` function with a relative path to the script file.

In this example, the main application is in the Node script file `hello.js`. The `hello.js` file makes a `require` function call to the `today.js` script file.

In this example again, it is the same “`hello.js`” Node.js file. The Node module is saved in a directory named “`mod_today`”. The actual script file is saved in `index.js`. When “`hello.js`” calls the `require` function in the “`mod_today`” directory, the script file checks whether a file that is named `index.js` exists. This name is the default name for a script in a Node module.

Package.json: The module manifest

- The `package.json` file describes details about a Node module
 - If a module does not have a `package.json` file, `node.js` assumes that the main class 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



Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-10. `Package.json`: The module manifest

Exporting functions and properties from a module

- Each Node module has an implicit `exports` object
- To make a function or a value available to Node applications that import your module, add a property to `exports`

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

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

Figure 1-11. Exporting functions and properties from a module

In this example, the `dayOfWeek` property is added to the `exports` object. The “`dayOfWeek`” property is assigned an anonymous function that returns the day of the week. For example, if the `dayOfWeek` function returns 1, this value maps to “Monday”.

Accessing exported properties from a module

- When you import a Node module, the `require` function returns a JavaScript object that represents an instance of the module
- For example, the `today` variable is an instance of the `today` Node module

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

- 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 module

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

Figure 1-12. Accessing exported properties from a module

Create a simple web server with the http package

- With the `http` Node 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
 - Develop a callback handler 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(3000);
```

Figure 1-13. Create a simple web server with the http package

Example: Simple HTTP server that returns a text message

```
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(3000);
```

Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

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

The `http.createServer` function takes one parameter: an anonymous function that takes in the request and response message. In the second line, the variable `body` has a value of `"Hello world!"`. In the third line, the `response.writeHead` function adds the HTTP status code of 200. The `'Content-Length'` header field is assigned the size of the response message body. The `'Content-Type'` field states a message type of `'text/plain'`.

In the next line, the `response.end` function closes the communications and sends the contents of the `body` variable as the message body.

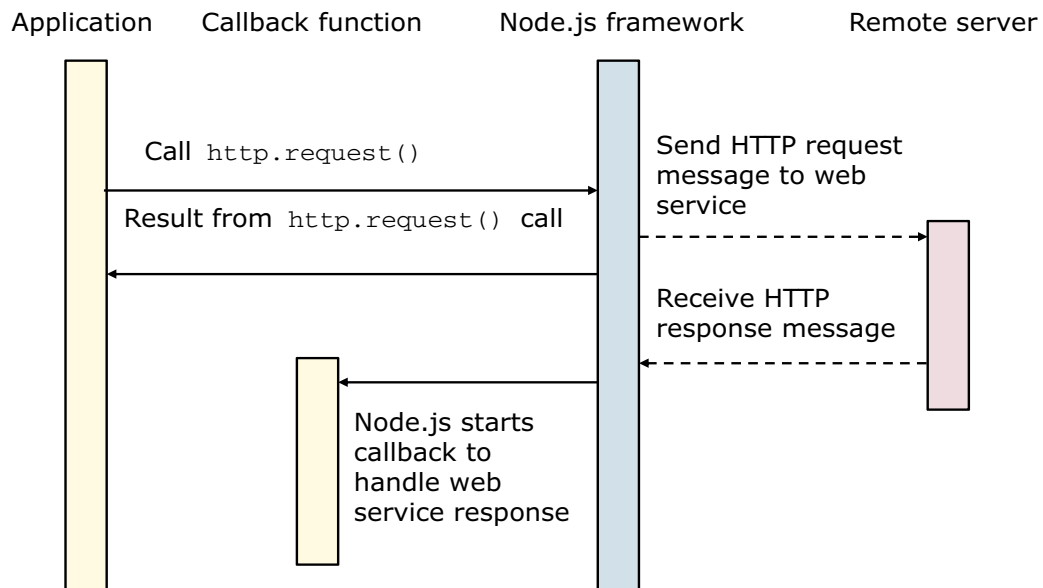
In the last line, you must call the `server.listen` function with a port number. When this server object receives an HTTP request, it calls the anonymous function that processes the message and sends back a response HTTP message.

1.1. What is a callback handler?

What is a callback handler?

- 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 in a non-blocking manner
 - Every network operation returns immediately
- To handle the result from a network call, write a **callback function** that Node.js calls when the network operation completes

Figure 1-15. What is a callback handler?

Example: Main application calls `http.request()`

Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-16. Example: Main application calls `http.request()`

Scenario 1: The main application calls `http.request`. This sequence diagram shows the interaction between the application, the Node.js framework, the web service call to the remote server, and the callback to the callback function. In the first step, the application calls `http.request`. This function calls the remote web server. It makes a request to the web service. Before the Node.js framework receives the HTTP response message from the remote web server, it immediately returns a result for the `http.request` function call. This result indicates that the request message was sent successfully. It does not say anything about the response message.

When the Node.js framework receives an HTTP response message from the remote server, it calls the callback function that you defined during the `http.request` function call. This function handles the HTTP response message.

Unit summary

- Explain the origin and purpose of the Node.js JavaScript framework
- Write a simple web server with JavaScript
- Import Node.js modules into your script



Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-17. Unit summary

Review questions



1. True or False: Node.js is a framework for multi-threaded JavaScript applications.

2. Which one of the following statements imports a script into your Node application?
 - A. `var http = import('http');`
 - B. `var http = include('http');`
 - C. `var http = request('http');`
 - D. `var http = require('http');`

Figure 1-18. Review questions

Write your answers here:

1.

Review answers



1. True or False: Node.js is a framework for multi-threaded JavaScript applications.
The answer is False. Each node application runs in a single-threaded environment. Node uses an event-driven system to handle multiple requests at the same time.

2. Which one of the following statements imports a script into your Node 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. The `require` function imports a Node module into your application.

Figure 1-19. Review answers

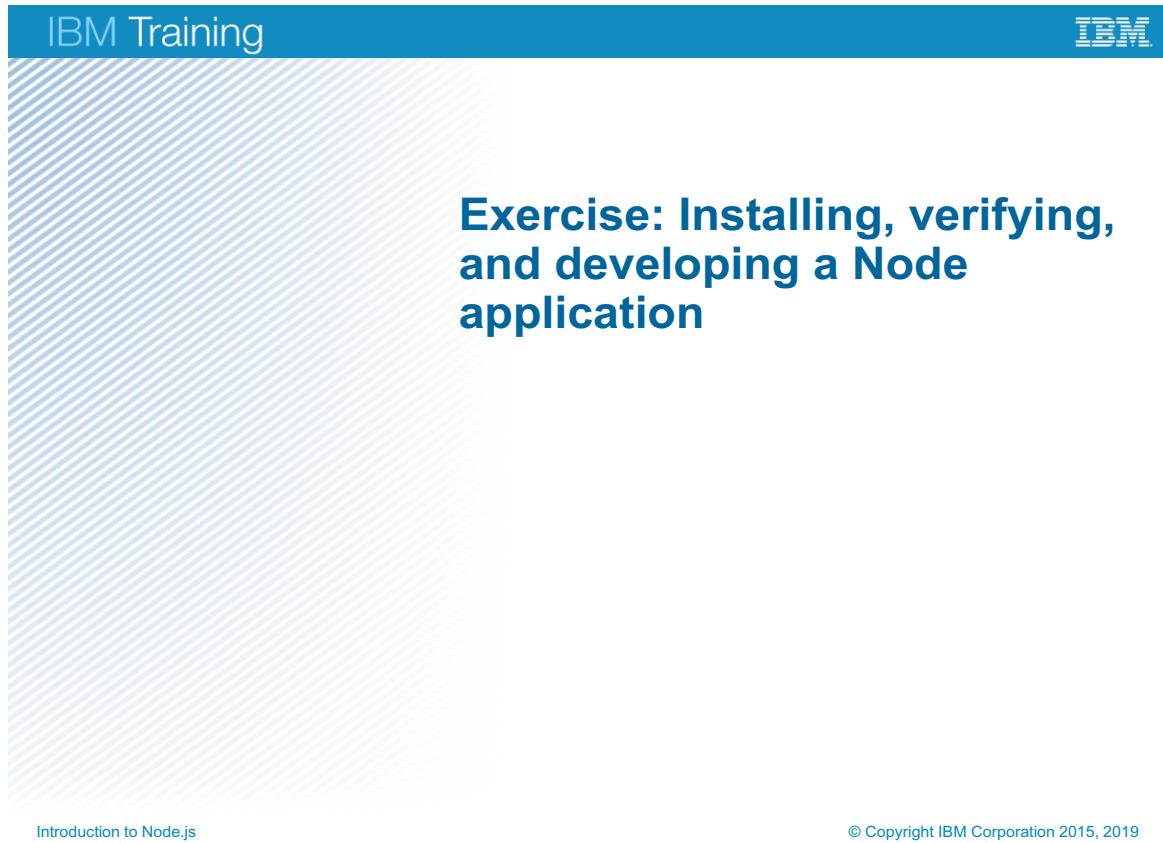


Figure 1-20. Exercise: Installing, verifying, and developing a Node application

Exercise objectives



- Install the Node.js runtime environment on a local workstation
- Verify the setup of the Node.js runtime environment
- Verify the setup of the Node package manager, npm
- Update the Node package manager on your workstation
- Define a package manifest file
- Install a third-party package in a Node application
- Start the Node interactive shell
- Run a Node application



Introduction to Node.js

© Copyright IBM Corporation 2015, 2019

Figure 1-21. Exercise objectives

Unit 2. Developing a REST API in Node

Estimated time

00:45

Overview

The Node.js framework relies on callback functions to handle network calls in an asynchronous manner. In this unit, you learn how to write anonymous callback functions to act on network events. You learn how to listen and intercept network traffic, and parse network traffic with socket programming.

How you will check your progress

- Review questions
- Lab exercise

Unit objectives

- Define a package dependency
- Explain the features of the Express Node framework
- Handle HTTP method calls with the Express framework
- Call remote services with the Request package
- Create a callback function to handle responses from remote services
- Parse XML data with the xml2js package



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-1. Unit objectives

2.1. Handle HTTP method calls with the Express framework

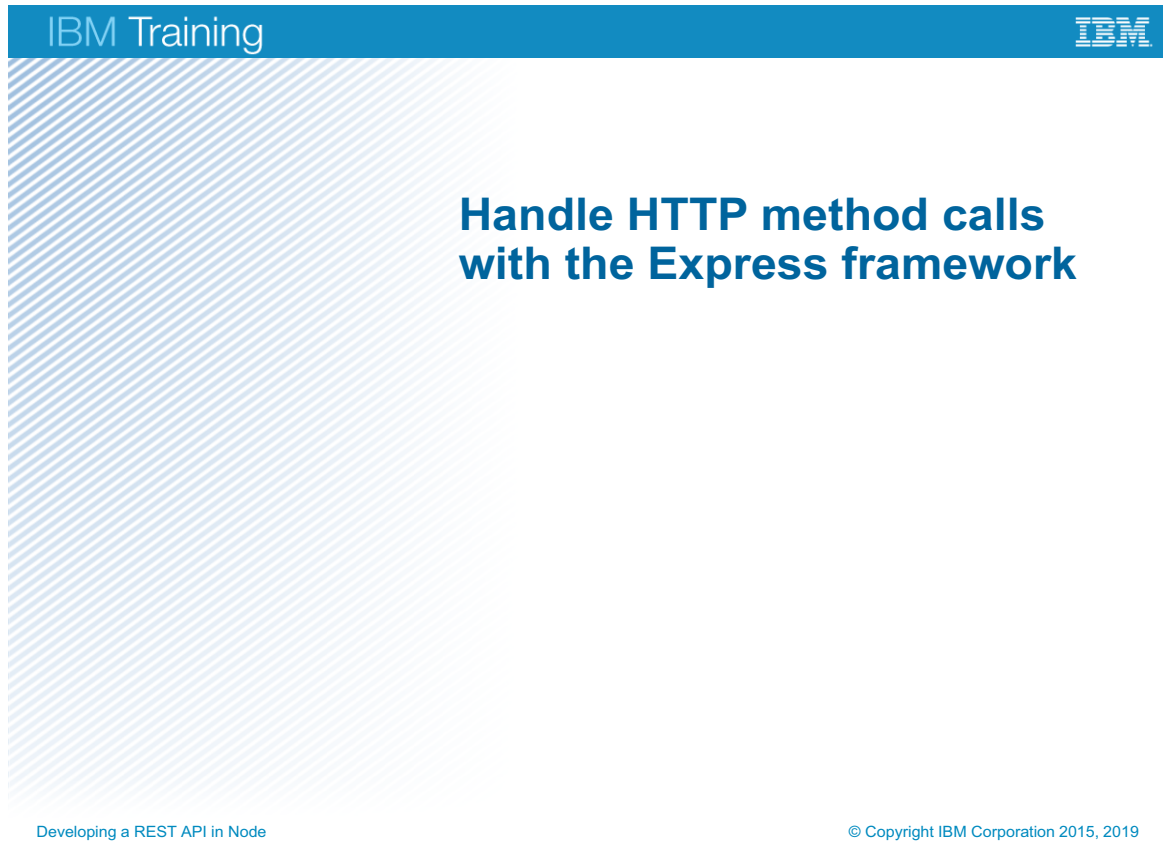


Figure 2-2. Handle HTTP method calls with the Express framework

Topics

- ▶ Handle HTTP method calls with the Express framework
 - Invoke remote services with the Request package
 - Parse XML data with the xml2js package



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-3. Topics

Extending Node with packages

- The default Node framework provides a limited set of features for building web applications
 - Developers rely on packages from the open source community to extend Node's features
- For example, Node does not provide a parsing function for XML messages
 - In simple messages, you can parse out an XML message with JavaScript string functions
 - You can also use an XML document object, but the object is not efficient in parsing a stream of XML data
- Use the `require` function to import packages into your application

```
var xml2js = require('xml2js');
```

Figure 2-4. Extending Node with packages

What is npm?

- Npm is an online repository of Node packages
- Npm is also the website that lists the Node packages in the repository
 - <https://npmjs.org>
- Npm is also a command-line utility to download, install, and manage Node packages from the online repository
 - When you install the Node runtime engine, you also install the npm utility
- The purpose of npm is to add features to the Node runtime engine
 - With the npm repository, you can reuse and share code from other developers
 - You can also manage different versions of code

Figure 2-5. What is npm?

When application developers talk about “npm”, they refer to one of three concepts. Npm is an online repository that maintains and stores Node application libraries, or Node packages. Npm also refers to the website npmjs.org, which describes the Node packages in the online repository. Finally, npm is the name of a command-line utility that application developers use to download, install, and manage Node packages from the online repository.

The purpose of the npm repository is to extend the capability of the Node runtime environment. The Node runtime takes a different philosophy in its software development kit (SDK) design. Node has a minimal set of built-in libraries to keep its memory and disk space footprint small. You use the npm repository to add more features to your Node application.

Express: A web application framework for Node

- Express is a third-party module that provides a framework for building web applications
 - It implements an “app” class that you map to a web resource path
- In contrast, the standard Node.js framework treats HTTP requests at a lower, network level
 - The `http.createServer` function relies on your custom callback function to parse through the web resource path
- The Express web application framework is one of the most popular building blocks for web applications

Figure 2-6. Express: A web application framework for Node

Step 1: Install the Express Node package

- In a terminal (Mac OS, Linux) or command prompt (Microsoft Windows), run the `npm install express --save` command
- The `npm install` command retrieves the `express` Node package from the online repository
 - The default option downloads the most recent version of the package
 - You can specify a specific version or a range of version numbers as a parameter as well
- The `--save` parameter adds the `express` package as a dependency in your `package.json` file

Figure 2-7. Step 1: Install the Express Node package

When you run the `npm install express` command, the `npm` utility searches and retrieves the newest version of the `express` Node package from the online repository.

Example: Install and add express as a dependency

```
student@cloudeu $ npm install express --save
...
npm http 200 https://registry.npmjs.org/inherits
express@4.13.2 node_modules\express
+-- merge-descriptors@0.0.2
+-- parseurl@1.0.1
+-- escape-html@1.0.1
+-- cookie@0.1.2
+-- methods@1.0.1
+-- cookie-signature@1.0.3
+-- fresh@0.2.2
+-- vary@0.1.0
+-- range-parser@1.0.0
+-- buffer-crc32@0.2.3
+-- depd@0.3.0
+-- commander@1.3.2 (keypress@0.1.0)
+-- debug@1.0.2 (ms@0.6.2)
...
```

The `npm` command downloads the Express framework, and all of its dependent modules

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-8. Example: Install and add express as a dependency

When you run the command `npm install` with the `--save` parameter, the “npm” application downloads the express Node package and all of the Node packages it requires. The log output in the console displays the names and versions of the packages that express depends on.

After the “npm” utility successfully retrieves and installs all of the dependent packages, it saves the name and version of the express Node package into your application’s package manifest file. Your application now depends on the express package to run properly.

Step 2: Review dependencies in the package.json file

- The `package.json` file stores information about the contents of a Node module
 - **Name:** A name for the Node module
 - **Version:** A string that defines the major and minor version number of the module
 - **Description:** A sentence that describes the purpose of the module
 - **Main:** Identifies the Node script as the entry point into the module
 - **Dependencies:** Lists which Node modules the current module requires

- To declare Express as a dependency, list the express module and a version number in the dependencies property

```
"dependencies": { "express": "^4.13.4" }
```

Figure 2-9. Step 2: Review dependencies in the package.json file

The caret symbol (^) in front of the version number means “greater than or equal to” the stated version.

Example: Updated package manifest file

```
{
  "name": "status-app",
  "version": "1.0.2",
  "description": "Return Node runtime status information",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "test": "node server.js"
  },
  "author": "Warren Fung <warrenf@example.com>",
  "license": "IPL-1.0",
  "dependencies": {
    "express": "^4.13.4"
  }
}
```

The `--save` option in the `npm install` command updates the dependencies section in your `package.json` file

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-10. Example: Updated package manifest file

Step 3: Create a web application in your code

- Create an instance of the `app` object from the Express web application framework
 - To handle web application requests, map an HTTP method and a web resource path to an JavaScript function

```
app.get('/api/today', function(req, res) {  
  var body =  
    "The day of the week is " + today() + ".";  
  res.type('text/plain');  
  res.set(  
    'Content-Length', Buffer.byteLength(body));  
  res.status(200).send(body);  
});
```

Listen to incoming
HTTP GET requests
to the `/api/today`
resource path

Write the HTTP
status code and
response message
with the
`res.status(code).
send(body)`
function

Figure 2-11. Step 3: Create a web application in your code

After you install a copy of the Express web application framework, you can create an instance of the `app` JavaScript object from the framework. In your application, to handle web application requests, map an HTTP method and a web resource path to the JavaScript function.

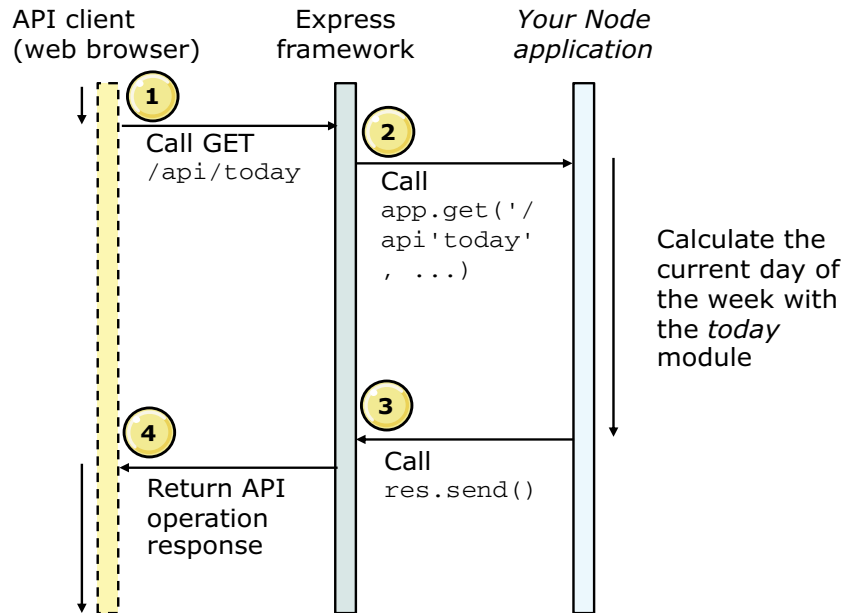
In this example, you are listening to incoming HTTP GET requests to the “temperature” resource path. You are also saving the value after the “temperature” resource path in a variable named “code”. When you run the `weather.current` function, you pass the parameter “code”, taken from the resource path.

Step 4: Create an instance of the server from the app

- Call `app.listen` to create a web server object that listens to incoming requests on the specified port
- In this example, the `app` listens to incoming requests on port `3000`
 - The second parameter defines an anonymous function that the Express framework calls when it creates an instance of the `server` object

```
var server = app.listen(3000, function() {  
  console.log('Listening on port %d', server.address().port);  
});
```

Figure 2-12. Step 4: Create an instance of the server from the app

Figure: Handle API operations with the express framework

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-13. Figure: Handle API operations with the express framework

This sequence diagram describes the method calls between the different modules in your Node application and the API client to your application.

1. An API client sends an HTTP **Get** request to the `'/api/today'` web route. The `app` object from the Express framework receives the HTTP request on port 3000.
2. The Express framework checks whether the `app` object defined a handler for a GET method call to the `'/api/today'` route. It invokes the anonymous function that you specified for the GET `'/api/today'` operation.
3. Your function reads the HTTP request from the `res` object. It returns an HTTP response with a call to the `res.send()` function.
4. The Express framework builds an HTTP response object and returns the message to the API client.

2.2. Invoke remote services with the Request package



Figure 2-14. Invoke remote services with the Request package

Topics

- Handle HTTP method calls with the Express framework
- ▶ Invoke remote services with the Request package
- Parse XML data with the xml2js package



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-15. Topics

Calling a remote service

- The `request` Node package creates an HTTP request message to a remote service
- To create an HTTP request, define two parameters:
 - Specify the HTTP **method** and the network **URI** to the remote service
 - Define a **callback function** to handle the response message from the service

Figure 2-16. Calling a remote service

The JavaScript language defines a module named `http` to handle HTTP requests and responses. To make a call from your Node application to a remote HTTP service, you can invoke the `http.request` function.

However, you must define event handlers within a callback function that handles low-level network events. For example, you must define separate event handlers for HTTP response data, response message termination, and error handlers.

The `request` Node package abstracts these details from the `http.request` function. You define one callback function that processes the HTTP response or fault message.

Example: Calling a remote HTTP service

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

var options = {
  method: 'get',
  uri: 'http://weather.gov/xml/current_obs/KSFO.xml',
  header: {
    'User-agent': 'weatherRequest/1.0'
  }
};

var callback = function(error, response, body) {
  console.log(body);
};

request(options, callback);
```

Specify the HTTP method, URI, and headers for the request message

Define a callback function to process the response from the remote service call

Make an HTTP request with the options and callback function

Developing a REST API in Node

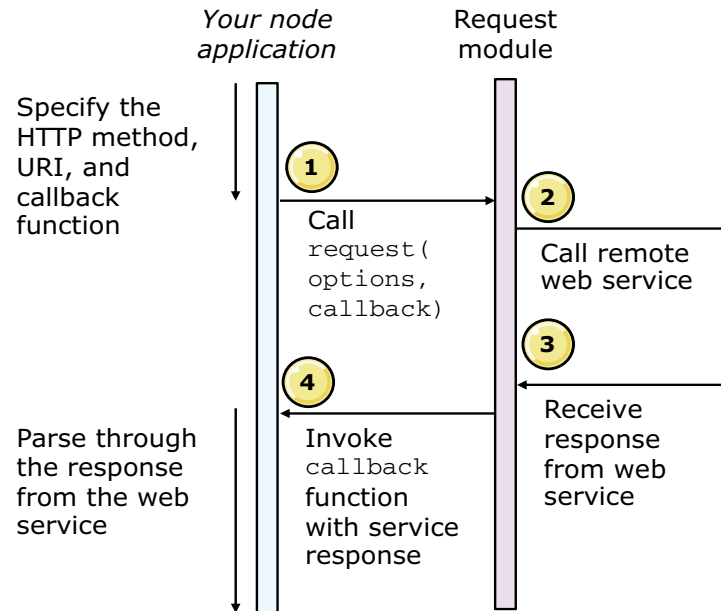
© Copyright IBM Corporation 2015, 2019

Figure 2-17. Example: Calling a remote HTTP service

This code example shows you how to make an HTTP request call from a function. First, you define a request variable to represent the request Node package. This package simplifies the steps to send an HTTP request message, and to process the HTTP response message.

The first parameter in the HTTP request function is an options variable. The options variable includes at least two variables: the HTTP method type, and the universal resource identifier (URI) or network path to the remote service.

In the example here, you make a call to the US National Weather Service to retrieve the weather observation from KSFO, or San Francisco International Airport. The second parameter of the HTTP request function is a callback function. In this case, it is an anonymous function that receives one parameter: the response object. When Node.js calls this anonymous function, events occur while receiving parts of the HTTP response object.

Figure: Call a remote service with the request package

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-18. Figure: Call a remote service with the request package

This scenario is the opposite of the one that you created with the Express framework. In this case, your Node application is the API client. You call the `request()` function to make an HTTP request to an API that is hosted on a remote server.

1. In the first step, your application defines the HTTP method, the network address, and route in the options variable. You also define a callback function to handle the response message from the remote server. You invoke the `request()` function with both parameters.
2. The request module sends an HTTP request to the endpoint that you specified in the options parameter.
3. The request module receives a response message from the remote server.
4. The request module invokes your callback handler with the contents of the response message.

Request: Options and callback parameters

- The `request` function calls the *callback* parameter when it receives part of the HTTP response message

- The *callback* parameter is optional
- You can send an HTTP request and disregard the response message

```
request( options, [callback] );
```

- When `http.request` calls the callback function, it passes a *response* object in the second parameter

```
request( options, function(error, response, body) { ... } );
```

Figure 2-19. Request: Options and callback parameters

Example: Handling error events

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

var options = {
  method: 'get',
  uri: 'http://weather.gov/xml/current_obs/KSFO.xml',
  header: {
    'User-agent': 'weatherRequest/1.0'
  }
};

var callback = function(error, response, body) {
  if (error) {
    console.err(error.message);
    return;
  }
  console.log(body);
};

request(options, callback);
```

Check for an error message before processing the response message

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-20. Example: Handling error events

An example for handling error events is provided here. If the call to the weather service returns an error, display the contents of the HTTP fault message to the console error log. The rest of the invocation code remains the same.

Propagating errors to callback functions

- As an asynchronous framework, Node.js makes extensive use of callback functions to return the result to the calling function
- Node.js modules in the SDK pass an error object as the first parameter in a callback function

```
function ( error, parameter1, parameter2, ... ) { ... }
```

- With this convention, the callback function checks if the first parameter holds an error object
 - If `error` is defined, the callback function handles the error and cleans up any open network or database connections
 - If `error` is not defined, then the callback function examines the result from the call

Figure 2-21. Propagating errors to callback functions

In the example, the function is defined with an error as the first parameter, followed by parameter 1, parameter 2, and so on.

Build a REST API that calls other web services

- In the previous examples, you built two parts of a REST API:
 - With the `Express` Node package, you can build a web application that handles requests on a specific web route
 - With the `Request` package, you can call remote services and process the response
- Combine your code from the `Express` and `Request` packages to build a REST API operation that calls other web services

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-22. Build a REST API that calls other web services

At this point, you learned how to build a REST API for the “`/api/today`” endpoint. You also learned how to make calls to remote web services with the `Request` package.

In the next step, combine both of these techniques to build an API operation that invokes an external web service.

Example: Define an API operation that calls a service

```
var request = require('request');
api.get('/api/weather', function(req, res) {
  var options = {
    method: 'get',
    uri:
    'http://weather.gov/xml/current_obs/KSFO.xml',
    header: {
      'User-agent': 'weatherRequest/1.0'
    }
  };
  var callback = function(error, response, body) {
    if (error) {
      res.status(500).send(error.message);
      return;
    }
    res.status(response.statusCode).send(body);
  };
  request(options, callback);
}
```

Define an API GET operation for the '/api/weather' route

Instead of printing the result to the console log, write the remote service result to the API operation response message

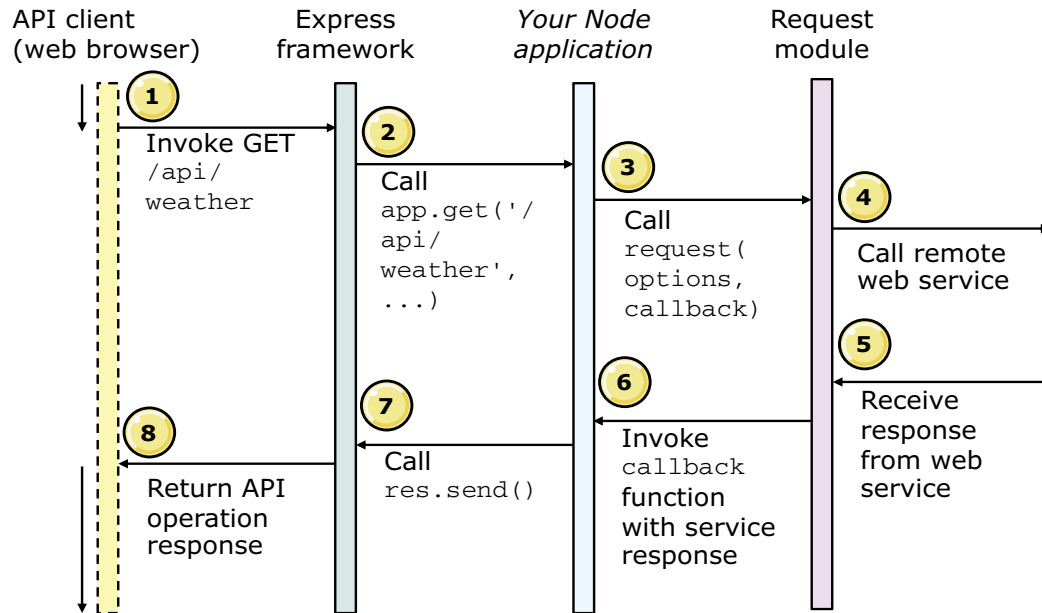
Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-23. Example: Define an API operation that calls a service

In this example, define an API operation for the '/api/weather' route with the Express framework. In the implementation of the '/api/weather' operation, use the request module to invoke the weather service. Write the result of the remote service call to the response message for the '/api/weather' API operation.

As a recommendation, document the order of HTTP calls in a sequence diagram. The code example has two sets of response messages. The `res` object represents the response message for the '/api/weather' operation in your API. The `response` object represents the response from the remote weather service call.

Figure: Define an API operation that calls remote services

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-24. Figure: Define an API operation that calls remote services

In the most recent version of the “`api/weather`”, you build an API operation with the Express framework. To retrieve the most recent weather observation, call the weather web service with the Request package.

1. The API client makes an HTTP GET request to the “`/api/weather`” endpoint on your Node application server, on port **3000**.
2. The Express app object listens to HTTP requests on port **3000**. It invokes the anonymous function that you specified for the `app.get('/api/weather')` route.
3. The function in your Node application processes the HTTP request message for `/api/weather`. Within the function, it calls the `request(options, callback)` function to make a second HTTP request to the weather web service.
4. The Request package sends an HTTP request to the endpoint that you specified in the **options** parameter.
5. The Request package receives an HTTP response from the weather service.
6. The Request package calls your **callback** handler function. Your callback function checks whether an error occurred during the call.

7. If the weather web service call completed successfully, your callback function processes the result and sends an HTTP response message to the original API client with the `res.send()` function.
8. The API client receives a response from the API operation call.

2.3. Parse XML data with the xml2js package

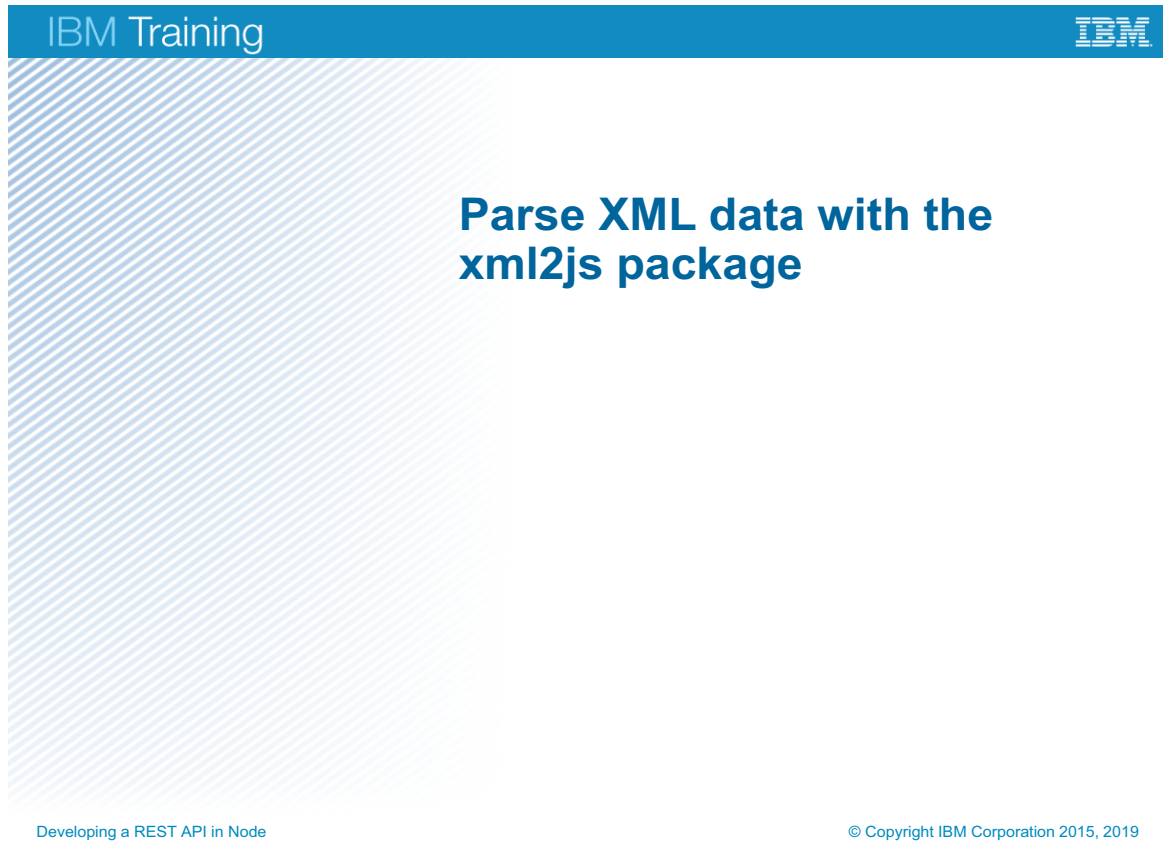


Figure 2-25. Parse XML data with the xml2js package

Topics

- Handle HTTP method calls with the Express framework
- Invoke remote services with the Request package
- ▶ Parse XML data with the xml2js package



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-26. Topics

Parse XML elements to JavaScript with xml2js

- The `xml2js` Node package parses a string of XML elements into a JavaScript object
- Unlike other XML parsing Node packages, `xml2js` is used only in JavaScript
 - It does not require an XML parsing library that is written in another language
- Third-party packages might have a software license that is different from the Node.js framework
 - Confirm that the licensing terms work with your company and your application before installing the package

Figure 2-27. Parse XML elements to JavaScript with xml2js

Advantages and disadvantages of treating XML data as a string

- String matching ignores the structure of XML data
 - The message body might contain malformed XML data
- Depending on the complexity of the XML data, string matching might be more efficient than building an XML tree of the data
- String matching is less tolerant to changes in the XML data structure
 - If the message adds or removes any XML elements, you must change the regular expression on the string match function

Figure 2-28. Advantages and disadvantages of treating XML data as a string

Step 1: Install third-party packages with npm

- The `npm` application manages Node packages in your Node framework installation
- Run the `npm install` command to retrieve and set up a Node package and any package dependencies

```
# npm install xml2js --save
npm http GET https://registry.npmjs.org/xml2js
npm http 304 https://registry.npmjs.org/xml2js
npm http GET https://registry.npmjs.org/sax
npm http GET https://registry.npmjs.org/xmlbuilder
npm http 304 https://registry.npmjs.org/sax
npm http 304 https://registry.npmjs.org/xmlbuilder
npm http GET https://registry.npmjs.org/lodash-node
npm http 304 https://registry.npmjs.org/lodash-node
xml2js@0.4.2 node_modules\xml2js
+-- sax@0.5.8
+-- xmlbuilder@2.2.1 (lodash-node@2.4.1)
```

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-29. Step 1: Install third-party packages with npm

In this example, you call `npm install xml2js` from the command line. The `npm` application retrieves the Node module “xml2js” and any dependencies that it requires from the internet.

Step 2: Call the XML to JavaScript parser function

```
var parse = require('xml2js').parseString;
```

```
var callback = function(error, response, body) {
  if (!error) {
    res.status(500).send(error.message);
  };
  parse(body, function(err, result) {
    var message =
      'The current temperature is ' +
      result.current_observation.temp_f[0] +
      ' degrees Fahrenheit.';
    res.type('text/plain');
    res.set('Content-Length', Buffer.byteLength(message));
    res.send(message);
  });
};
```

The `parseString` function runs the callback function when it finishes processing the XML tree in `body`

- The `result` JavaScript variable represents the contents of the XML fragment in `body`
- The property `current_observation.temp_f[0]` maps to the first child element in the `<temp_f>` XML element
- The `<temp_f>` element is a child of `<current_observation>`

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-30. Step 2: Call the XML to JavaScript parser function

In the second step, you import the package into your application.

In the first line, the `parseString` function calls the callback function when it finishes processing the XML tree in `body`.

In the second line, the `result` JavaScript variable represents the contents of the XML fragment in `body`. In other words, the `result` JavaScript variable represents the XML element, but in JavaScript form. The property `current_observation.temp_f[0]` maps to the first child element in the `<temp_f>` XML element. The `<temp_f>` element is a child of the `<current_observation>` element.

Unit summary

- Define a package dependency
- Explain the features of the Express Node framework
- Handle HTTP method calls with the Express framework
- Call remote services with the Request package
- Create a callback function to handle responses from remote services
- Parse XML data with the xml2js package



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-31. Unit summary

Review questions



1. True or False: The Node.js API (application programming interface) provides an extensive set of libraries to parse web service data types.
2. True or False: To install a Node module to your application, run the npm install command within your own module's directory.
3. True or False: The convention is to pass an error object as the first parameter in a callback function.
4. True or False: The app object from the Express module provides a finer-grained programming model for HTTP requests and responses.

Figure 2-32. Review questions

Write your answers here:

- 1.
- 2.
- 3.

Review answers



1. True or False: The Node.js API (application programming interface) provides an extensive set of libraries to parse web service data types.
The answer is False. Node.js is a minimalist, extensible framework. Developers can install third-party modules to parse web service data.
2. True or False: To install a Node module to your application, run the `npm install` command within your own module's directory.
The answer is True. The `npm` command downloads and installs third-party Node modules from the internet.
3. True or False: The convention is to pass an error object as the first parameter in a callback function.
The answer is True. By placing the error object as the first parameter of a callback function, applications can quickly check whether the response is an error condition or a normal one.
4. True or False: The `app` object from the Express module provides a finer-grained programming model for HTTP requests and responses.
The answer is True. The `http.server` object handles all HTTP traffic on the specified port. The `express.app` object handles HTTP traffic that matches a resource path pattern on a specified port.

Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-33. Review answers

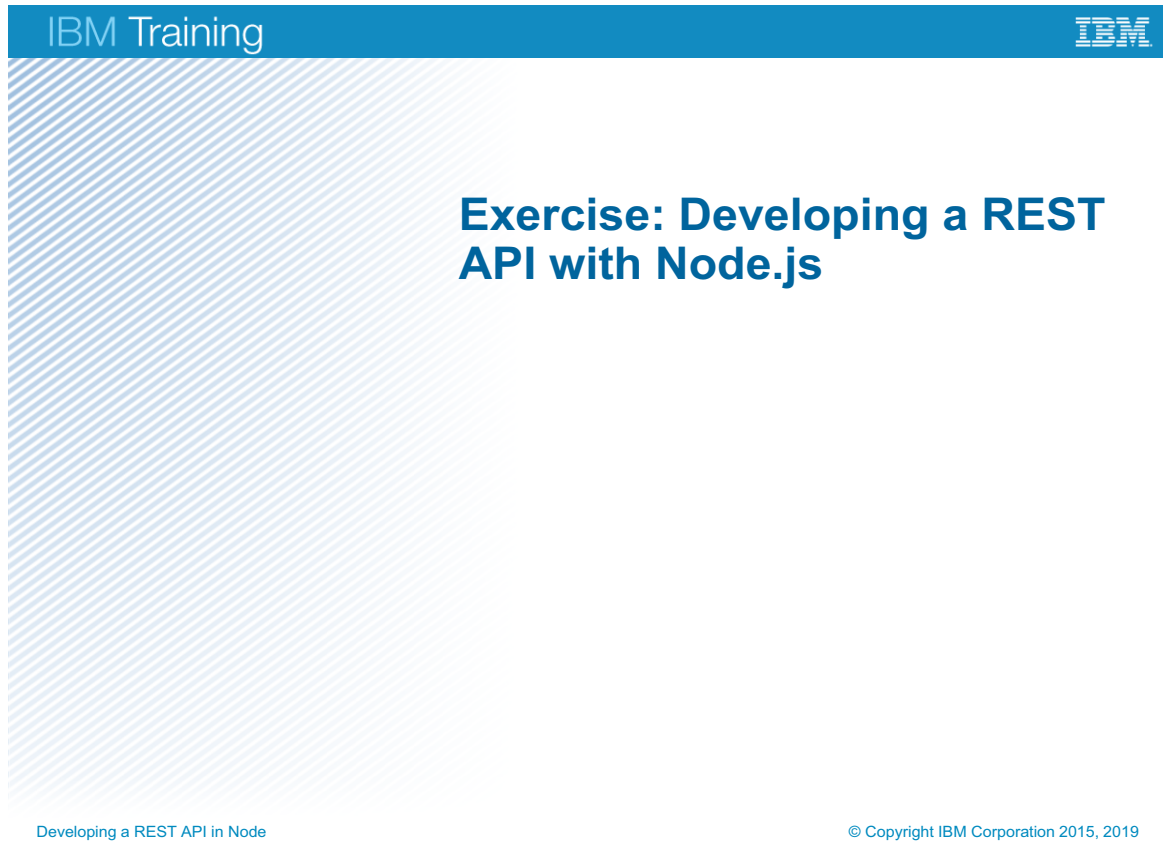


Figure 2-34. Exercise: Developing a REST API with Node.js

Exercise objectives



- Install the Express Node package
- Define an Express web application
- Handle requests to web resources with Express
- Call remote services with the Request package
- Create a callback function to handle responses from remote calls
- Handle errors with a callback return parameter
- Test a callback function in a Node application



Developing a REST API in Node

© Copyright IBM Corporation 2015, 2019

Figure 2-35. Exercise objectives

Unit 3. Static code analysis and unit testing

Estimated time

00:45

Overview

This unit describes how to validate JavaScript code with static code analysis tools. The unit also describes some of the testing tools that are used to test Node applications.

How you will check your progress

- Review questions
- Lab exercise

Unit objectives

- Describe how to validate JavaScript code with ESLint
- Describe how to unit test your application with Mocha
- Explain how to create HTTP unit tests with Supertest



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-1. Unit objectives

3.1. Static code analysis with linting utilities

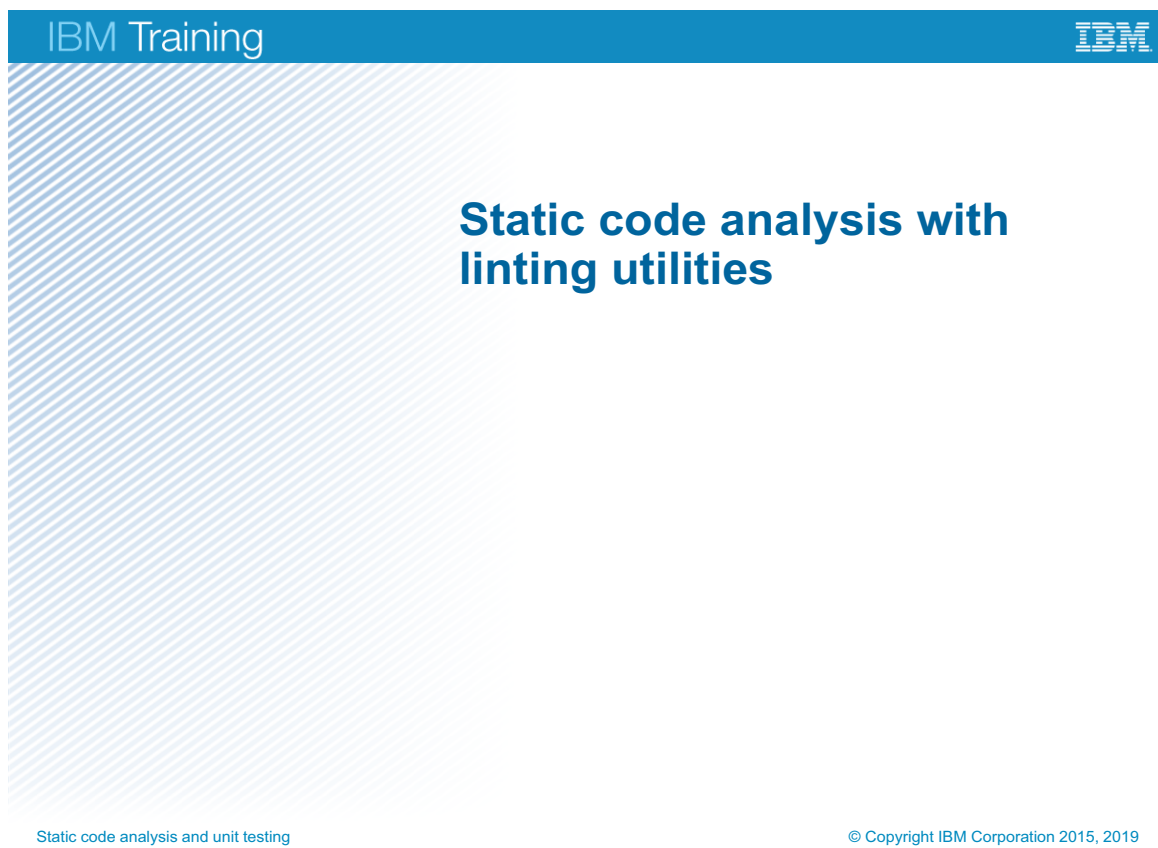


Figure 3-2. Static code analysis with linting utilities

Topics

- ▶ Static code analysis with linting utilities
 - Testing with Mocha
 - Supertest



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-3. Topics

JavaScript code validator

- *Linting* is the process of running a program that can analyze code for potential errors
 - The name *lint* is derived from a UNIX utility that flags potential issues in C language source code
 - Code linting is a type of static analysis that is used to find problematic patterns or code that does not conform to certain style guidelines
- JavaScript is an interpreted language
 - Cannot rely on a compiler to flag syntax errors
 - Instead, use one of the JavaScript lint utilities to analyze the source code
 - A good linting tool helps to ensure that JavaScript projects adhere to coding standards
 - Linting tools allow developers to discover problems with their JavaScript code without executing it

Figure 3-4. JavaScript code validator

You can use one of a number of lint utilities to find common errors in JavaScript code.

JavaScript lint utilities

- JSLint
 - www.javascriptlint.com
 - Earliest of the linting tools
 - Not configurable or extensible
- JSHint
 - jshint.com
 - Forked version of JSLint with good documentation
 - More configurable than JSLint
 - Ready to go
 - Supports ECMAScript 2015 (ES6)
- ESLint
 - Most recent version
 - eslint.org
 - Extensible
 - Comes with over 200 custom rules
 - Best ES6 support
 - Requires a little more configuration setup than JSHint
 - Documentation good, but unclear in places

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-5. JavaScript lint utilities

The slide lists three open source linting tools for JavaScript. The earliest tool, created in 2002, is JSLint. It is easy to install and use, but it is not configurable like many of the later tools.

JSHint is a forked version of JSLint. JSHint supports a configuration file, and most settings can be configured. It supports ECMAScript 2015 (ES6). The disadvantage of JSHint is that it does not support custom rules.

ESLint is an open source JavaScript linting utility that was originally created in 2013. ESLint is easily extensible and comes with many custom rules that can be toggled.

JSHint (1 of 2)

- Install with npm
 - Install JSHint in the root folder of the package
 - `npm install jshint --save-dev`
 - JSHint is installed in the `node_modules` folder of the package
 - Creates `devDependencies` in the `package.json` file

```
{
  "name": "sample_app",
  ...
  "main": "hello.js",
},
  "devDependencies": {
    "jshint": "^2.9.1"
  }
}
```

- Modules that are listed in `devDependencies` are not included in an `npm shrinkwrap` command

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-6. JSHint (1 of 2)

JSHint is simple to install and set up. In the example, JSHint is installed locally with the `npm install jshint --save-dev` command.

Installing the utility locally installs the utility in the `node_modules` directory under the package root directory.

The package is added to `package.json` as a development dependency.

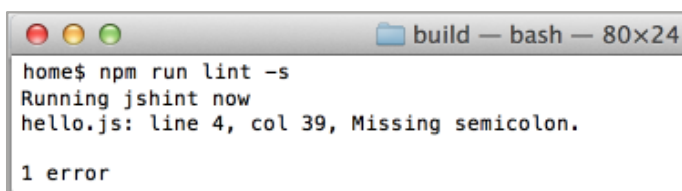
Developers can install their preferred linting utility inside their packages. The modules can be automatically excluded from the dependency tree in the production version.

JSHint (2 of 2)

Source for `package.json`

```
{
  "name": "sample_app",
  "version": "1.0.0",
  "description": "Sample",
  "main": "hello.js",
  "scripts": {
    "lint": "echo 'Running jshint now' && jshint hello.js"
  }
  ...
}
```

- Run the script to perform lint
 - `npm run lint` or
 - `npm run lint -s`



```
build — bash — 80x24
home$ npm run lint -s
Running jshint now
hello.js: line 4, col 39, Missing semicolon.

1 error
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-7. JSHint (2 of 2)

The example `package.json` file includes a script that is named `lint`, which is used to run the JSHint utility on the `hello.js` source.

ESLint installation

- Uses many custom rules
 - Concise output that includes the rule name
 - Helps you identify which rules are causing the errors
 - Every rule is a plug-in
- Installation
 - `npm install -g eslint`
- Verification
 - `eslint -v`
 - `V2.4.0`

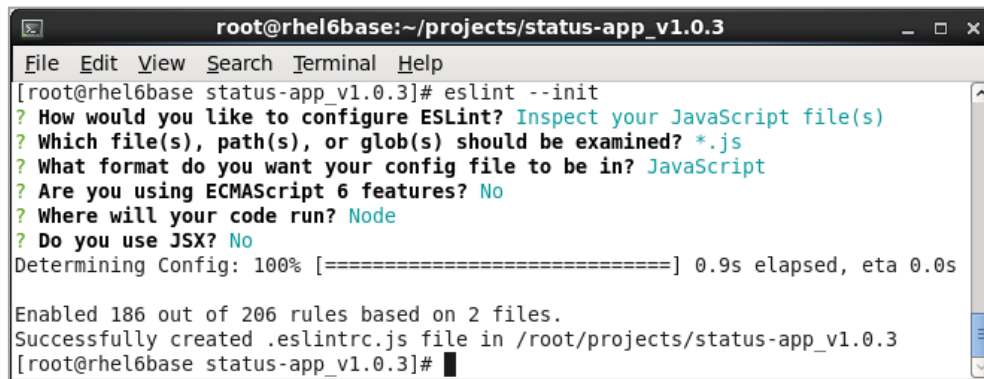
Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-8. ESLint installation

ESLint initial configuration

- Requires the initial setup of a configuration file
 - `eslint --init`
 - Creates an `.eslintrc` file in your directory
 - You can set up different configurations for different projects

A terminal window titled 'root@rhel6base:~/projects/status-app_v1.0.3' showing the execution of 'eslint --init'. The terminal displays a series of prompts: 'How would you like to configure ESLint? Inspect your JavaScript file(s)', 'Which file(s), path(s), or glob(s) should be examined? *.js', 'What format do you want your config file to be in? JavaScript', 'Are you using ECMAScript 6 features? No', 'Where will your code run? Node', and 'Do you use JSX? No'. It then shows a progress bar for 'Determining Config: 100%' and a message: 'Enabled 186 out of 206 rules based on 2 files. Successfully created .eslintrc.js file in /root/projects/status-app_v1.0.3'. The prompt returns to the root user at the terminal.

```
root@rhel6base:~/projects/status-app_v1.0.3
File Edit View Search Terminal Help
[root@rhel6base status-app_v1.0.3]# eslint --init
? How would you like to configure ESLint? Inspect your JavaScript file(s)
? Which file(s), path(s), or glob(s) should be examined? *.js
? What format do you want your config file to be in? JavaScript
? Are you using ECMAScript 6 features? No
? Where will your code run? Node
? Do you use JSX? No
Determining Config: 100% [=====] 0.9s elapsed, eta 0.0s
Enabled 186 out of 206 rules based on 2 files.
Successfully created .eslintrc.js file in /root/projects/status-app_v1.0.3
[root@rhel6base status-app_v1.0.3]#
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

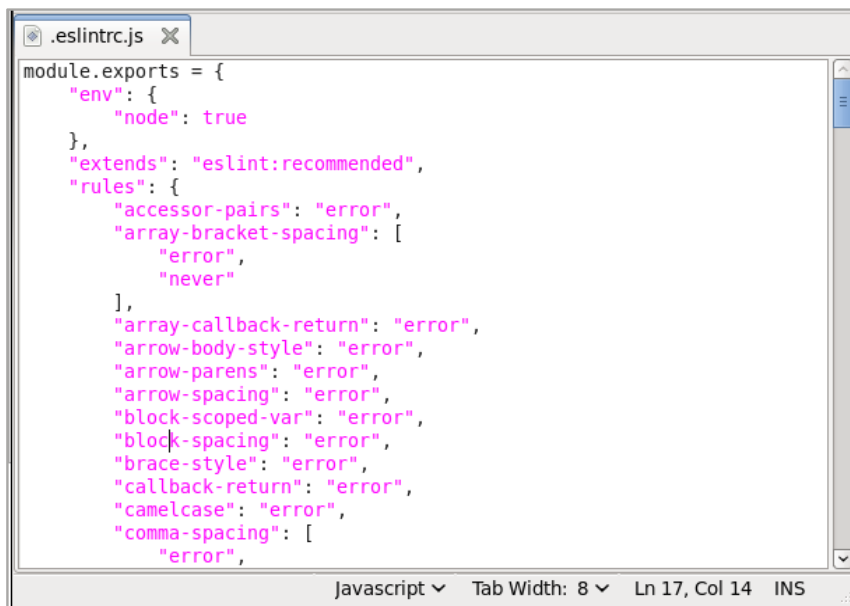
Figure 3-9. ESLint initial configuration

The figure shows an example of setting up the `.eslintrc.js` file after typing `eslint --init`. In this example, the `.eslintrc.js` file sets up the rules based on the inspection of the code in existing JavaScript files.

According to the response in the terminal window, the ESLint initial configuration enabled 186 out of 206 rules based on parsing two files.

ESLint configuration file

- Edit the rules in the `.eslintrc.js` file, if required



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-10. ESLint configuration file

Since the ESLint configuration file is specific to a package, it is written to the root directory of the package.

To change the configuration file, you can either edit the file, or re-create it, choosing different options when prompted during the initial creation of the file.

In the configuration file, you see the rules that are configured, under the `rules` property.

Rules can be configured with three property values:

- `"off"` (0): This value turns off the rule.
- `"warn"` (1): This value turns on the rule as a warning.
- `"error"` (2): This value turns on the rule as an error (exit code is 1).

ESLint command line

```
eslint [options] [file | dir | glob]*
```

Options:

<code>-c, --config path</code>	Use configuration from this file
<code>--no-eslintrc</code>	Disable the use of configuration from <code>.eslintrc</code>
<code>--env [String]</code>	Specify environments
<code>--ext [String]</code>	Specify JavaScript extensions, default <code>.js</code>
<code>--parser String</code>	Specify parser to be used, default espre
<code>--parser-options</code>	Parser options, for example <code>ecmaVersion:6</code>
<code>--rule</code>	Option specifies a rule to be used, example <code>eslint --rule 'quotes: [2, double]'</code>

For more options, run: `eslint -h`

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

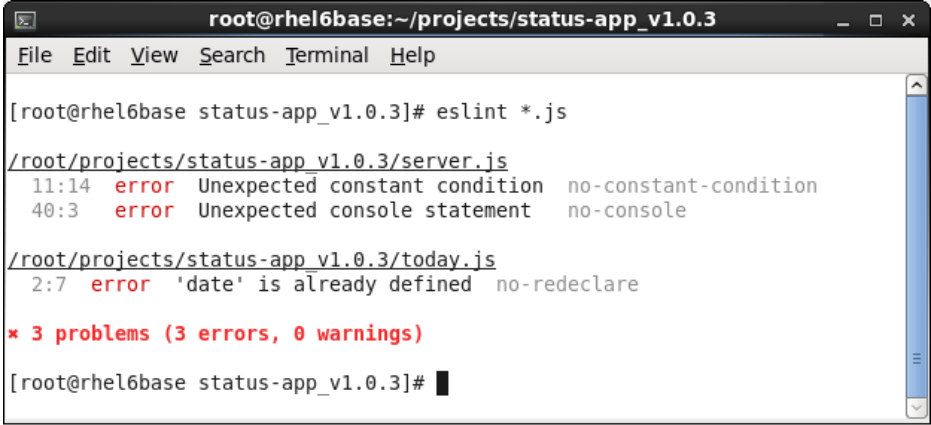
Figure 3-11. ESLint command line

ESLint is written to be used primarily on the command line.

ESLint uses a parser that is named `espre` to parse the JavaScript code. ESLint does not use the Node runtime engine because the code that you are checking is never run, an important distinction in static (as opposed to dynamically run) code analysis.

You can view the options by running `eslint -h`.

ESLint sample output



```
root@rhel6base:~/projects/status-app_v1.0.3
File Edit View Search Terminal Help

[root@rhel6base status-app_v1.0.3]# eslint *.js

/root/projects/status-app_v1.0.3/server.js
 11:14  error  Unexpected constant condition  no-constant-condition
 40:3   error  Unexpected console statement   no-console

/root/projects/status-app_v1.0.3/today.js
  2:7   error  'date' is already defined      no-redeclare

* 3 problems (3 errors, 0 warnings)

[root@rhel6base status-app_v1.0.3]#
```

- These rules displayed errors when running ESLint:
 - `no-constant-condition`: Disallow use of constant expressions in conditions
 - `no-console`: Disallow use of console
 - `no-redeclare`: Disallow declaring the same variable more than once

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-12. ESLint sample output

ESLint identifies the line number and position in the JavaScript file where the error condition occurred.

For more information about ESLint rules, see: eslint.org/docs/rules/

ESLint edit rule

- Edit `.eslintrc.js`
 - Example: Set no-console to off
 - You can add a rule or modify an existing rule

```
"rules": {  
  "no-console": "off",  
  ...  
}
```

- Property values
 - "off"
 - "warn"
 - "error"
- When the rule no-console is set to off, you can use these statements in the source code without ESLint flagging them:
 - `console.log("Some string");`
 - `console.warn("Some warning");`
 - `console.error(err);`

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-13. ESLint edit rule

Change the rules by editing the rules in the `.eslintrc.js` file.

3.2. Testing with Mocha

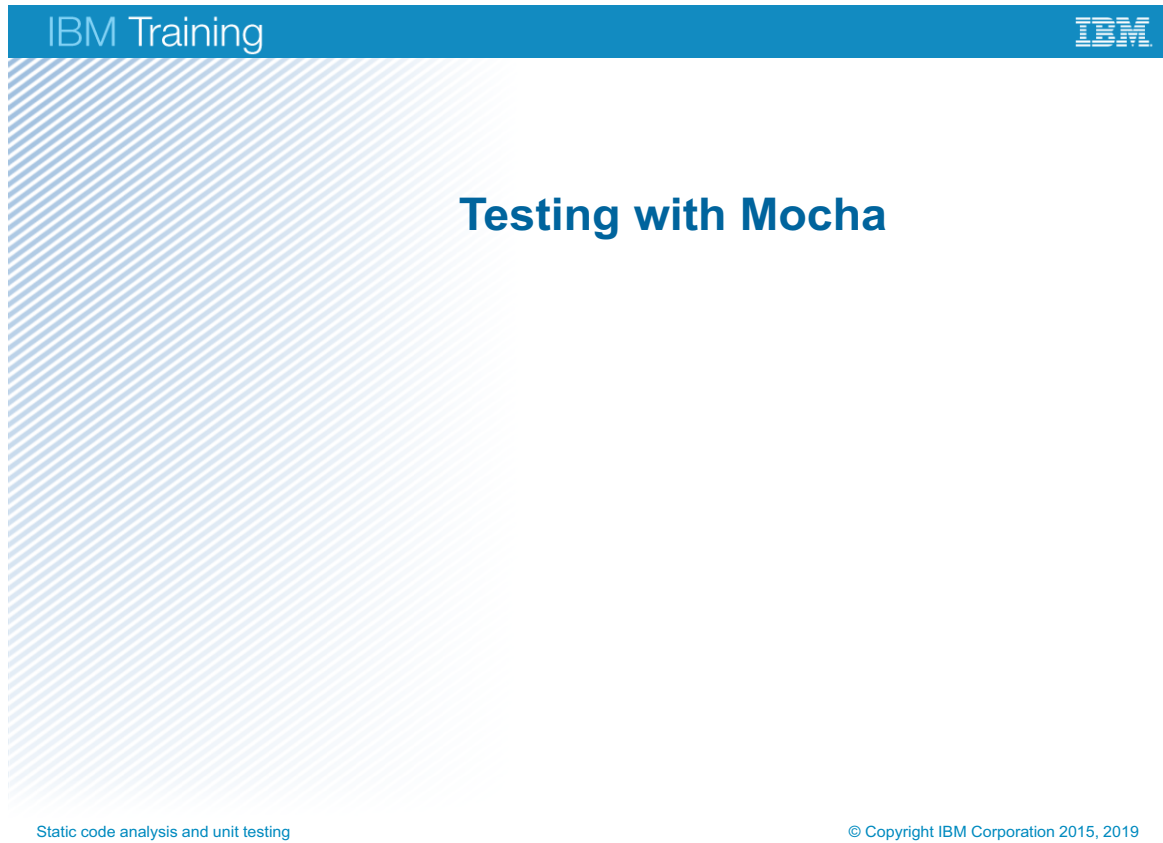


Figure 3-14. Testing with Mocha

Topics

- Static code analysis with linting utilities
- ▶ Testing with Mocha
- Supertest



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-15. Topics

Mocha

- Mocha is a testing module for Node.js
 - Simple, extensible, and fast
 - Includes browser support
 - Synchronous or asynchronous testing
 - Test coverage reports
 - Pluggable assertion libraries

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-16. Mocha

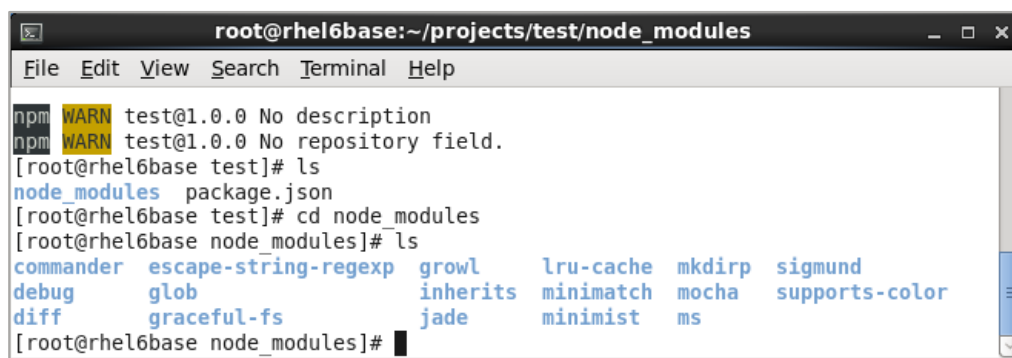
Mocha is a JavaScript test framework for Node.js and the browser.

The framework provides a standard way to define test cases and a convention for how a test case signals success or failure.

For more information, see: mochajs.org

Mocha installation

- Install with npm
 - Install Mocha in the root folder of the package
 - `npm install mocha --save-dev`
 - Mocha is installed in the `node_modules` folder of the package
 - Creates devDependencies in the `package.json` file



```
root@rhel6base:~/projects/test/node_modules
File Edit View Search Terminal Help
npm WARN test@1.0.0 No description
npm WARN test@1.0.0 No repository field.
[root@rhel6base test]# ls
node_modules package.json
[root@rhel6base test]# cd node_modules
[root@rhel6base node_modules]# ls
commander  escape-string-regexp  growl      lru-cache  mkdirp  sigmund
debug      glob                  inherits   minimatch  mocha    supports-color
diff       graceful-fs           jade       minimist   ms
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

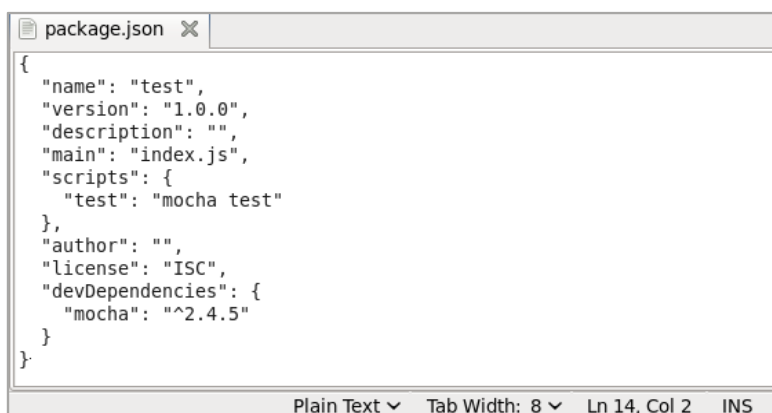
Figure 3-17. Mocha installation

Here you see the modules that are installed in the `node_modules` directory after typing the command `npm install mocha --save-dev` from the root folder of the package.

Alternatively, you can install Mocha globally with npm to make it available to all projects.

Mocha test directory

- The `test/` directory
 - By default, Mocha looks for the files to test in `./test/*.js`
 - Recommends putting tests in a `test/` folder
- Running the test script of the `package.json` file runs all the JavaScript files in the `/test` directory
 - `npm run test`



```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "mocha test"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "mocha": "^2.4.5"
  }
}
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-18. Mocha test directory

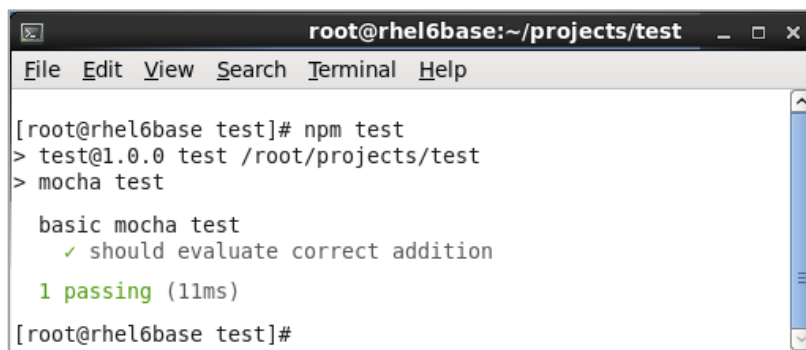
By default, Mocha looks for all the `*.js` files to test in the `./test` subdirectory of the project. These test cases themselves are functions that are defined in JavaScript files.

Example source code that is used for a Mocha test

- `./test/basic-test.js` source code:

```
var assert = require('assert');
describe('basic mocha test', function() {
  it('should evaluate correct addition', function(){
    assert(2 + 3 === 5);
  });
});
```

- Run npm test:

A screenshot of a terminal window titled 'root@rhel6base:~/projects/test'. The terminal shows the command 'npm test' being executed, which runs 'mocha test'. The output shows a single test 'basic mocha test' passing with the message '✓ should evaluate correct addition' and '1 passing (11ms)'.

```
root@rhel6base:~/projects/test
File Edit View Search Terminal Help

[root@rhel6base test]# npm test
> test@1.0.0 test /root/projects/test
> mocha test

  basic mocha test
    ✓ should evaluate correct addition

  1 passing (11ms)

[root@rhel6base test]#
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-19. Example source code that is used for a Mocha test

The example source code shows a simple example of a Mocha test and its output. The code in the example runs the Mocha test without needing any additional assertion libraries.

The `assert` module is part of Node.js itself. You can write the assertion code with the Node.js `assert()` function alone.

Other libraries, such as Chai, include their own `assert`, `expect`, and `should` functions. The `chai.assert` library provides its own `assert` interface that is a full-function `assert` library, which provides more options than the `assert` functions that are found within Node.js.

Mocha can use other libraries, such as `expect.js`, to support `expect()` style assertions. Libraries such as `expect()` make the code more readable.

Chai assertion library

- Full assert API
- Uses the assert-dot notation

```
const assert = require('chai').assert
var label = 'bar';

//without optional message
assert.typeOf(label, 'string');

//with optional message
assert.typeOf(label, 'string', 'label is a string');

assert.equal(label, 'bar', 'label equals \'bar\'');

assert.lengthOf(label, 3,
  'label\'s value has a length of 3')
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-20. Chai assertion library

The Chai assertion library has an assert interface that exposes the assert statements in a classic assert-dot notation. This library can easily be integrated into Mocha tests by using the `require('chai')` statement at the start of the test code.

You can use the Chai assert style to include an optional message as the last argument in the assert statement.

These messages are included in the error message when the assertion fails.

Mocha BDD testing

- Behavior-driven development
 - Best practices for writing great tests
 - Used together with test-driven development and unit testing
 - Interfaces (API) for Mocha tests
- BDD interfaces for Mocha
 - `describe()`, `context()`, `it()`, `before()`, `after()`, `beforeEach()`, and `afterEach()`
- `describe`
 - Describes a test suite, test case, or action
 - Can be nested
 - `describe('a name for the test unit', function() { ...`
- `it`
 - An expected outcome
 - Followed by contained assertions within the closure scope
 - `it('should return status code 200', function() {
 assert.equal(res.statusCode, 200, 'statusCode is not 200')
 }`

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-21. Mocha BDD testing

Mocha behavior-driven development is a suggested practice for writing good tests. BDD is a refinement of practices that come from test-driven development (TDD).

The Mocha BDD interface provides the `describe()`, `context()`, `it()`, `before()`, `after()`, `beforeEach()`, and `afterEach()` functions.

The `mocha` command starts a test harness for a series of test cases that the `describe` function defines. The `it` function declares an assertion about the application.

For more information, see: <https://mochajs.org/#interfaces>

Mocha can use other libraries such as `expect.js` to support `expect()` style assertions, or `chai` that supports `expect()`, `assert()`, and `should` style assertions.

Mocha functions

- `describe`
 - A test suite that might contain many tests
- `it`
 - A test with an expected outcome
- `before`
 - Executes before any tests in a test suite
- `after`
 - Executes after all tests in a test suite
- `beforeEach`
 - Executes before each test
- `afterEach`
 - Executes after each test

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-22. Mocha functions

Using Mocha functions

```
describe('BDD test', function() {  
  before(function() {console.log('before function')}});  
  after(function() {console.log('after function')}});  
  beforeEach(function() {console.log('beforeEach function')}});  
  afterEach(function() {console.log('afterEach function')}});  
  it('unit T1', function() {});  
  it('unit T2', function() {});  
});
```

A terminal window with a menu bar (File, Edit, View, Search, Terminal, Help) and a title bar. The terminal shows the command 'npm test' being run, which executes 'mocha test'. The output displays the BDD test structure: 'BDD test', 'before function', 'beforeEach function', 'unit T1' (with a green checkmark), 'afterEach function', 'beforeEach function', 'unit T2' (with a green checkmark), 'afterEach function', and 'after function'. At the bottom, it shows '2 passing (14ms)' in green text.

```
File Edit View Search Terminal Help  
[root@rhel6base test]# npm test  
> test@1.0.0 test /root/projects/test  
> mocha test  
  BDD test  
  before function  
  beforeEach function  
    ✓ unit T1  
  afterEach function  
  beforeEach function  
    ✓ unit T2  
  afterEach function  
  after function  
  
  2 passing (14ms)
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-23. Using Mocha functions

The example shows the output when running a Mocha test by using the `describe`, `before`, `after`, `beforeEach`, `afterEach`, and `it` functions.

Mocha BDD code

```
describe('a name for the unit test', function() {  
  it('should return a value x given the input y', function()  
  {  
    expect(x).to.equal(y);  
  });  
});
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-24. Mocha BDD code

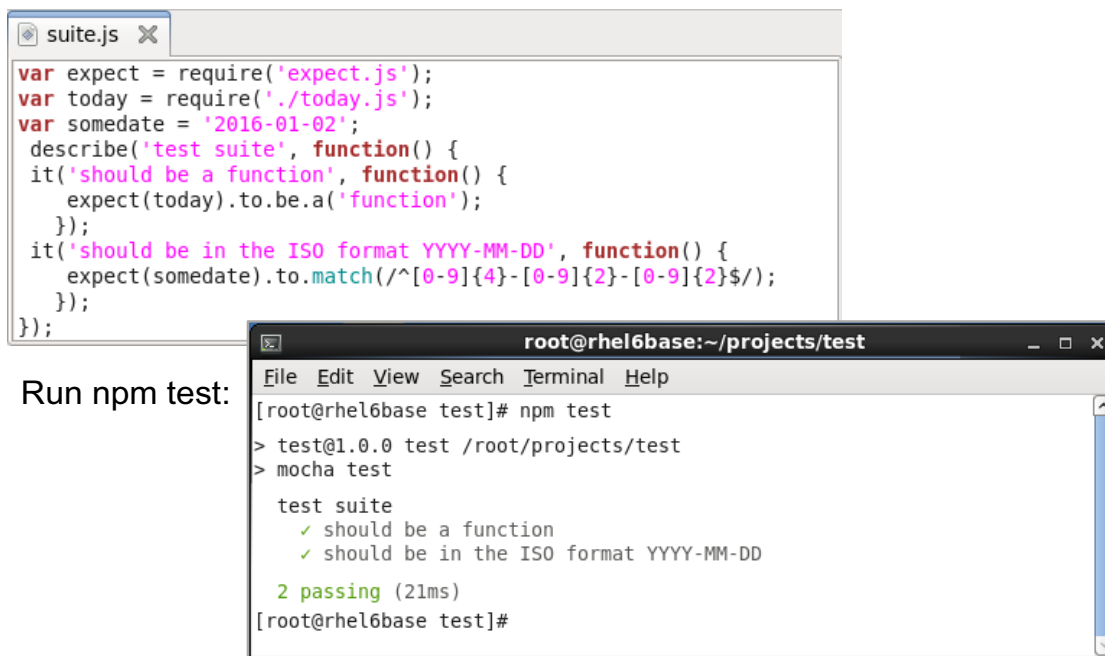
In this example, the `describe` function declares the name of the function or the application that you want to test.

The `it` function contains an assertion statement that is true if the application works properly.

The code within the second anonymous function checks whether the assertion holds true. The `expect(x).to.equal(y)` function takes two parameters. If the two parameters are equal, the function completes without issue. If the two parameters do not match, the `expect(...)` function throws a runtime error.

The Mocha framework catches the runtime error, and marks the test case as failed.

Mocha test example with expect.js



Run npm test:

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-25. Mocha test example with expect.js

Mocha can use other libraries such as `expect.js` to support `expect()` style assertions.

The `expect.js` library can be installed with npm or added to the `package.json` file.

In the example, a grouping that is named 'test suite' is defined with the `describe` option.

Each `it` statement is a test case that includes an expected result.

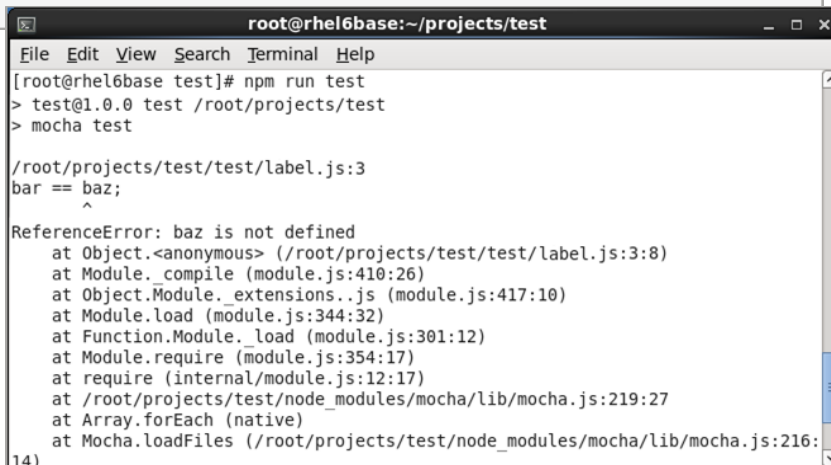
The first test case tests whether `today()` is a function. The second test case checks some input date string against a regular expression.

Mocha test on invalid source code

- `./test/label.js` source code:

```
var label = "hello";
var bar = "world";
bar == baz;
console.log(label + " " + bar);
```

- Run npm test:



```
root@rhel6base:~/projects/test
File Edit View Search Terminal Help
[root@rhel6base test]# npm run test
> test@1.0.0 test /root/projects/test
> mocha test

/root/projects/test/test/label.js:3
bar == baz;
      ^
ReferenceError: baz is not defined
    at Object.<anonymous> (/root/projects/test/test/label.js:3:8)
    at Module._compile (module.js:410:26)
    at Object.Module._extensions..js (module.js:417:10)
    at Module.load (module.js:344:32)
    at Function.Module._load (module.js:301:12)
    at Module.require (module.js:354:17)
    at require (internal/module.js:12:17)
    at /root/projects/test/node_modules/mocha/lib/mocha.js:219:27
    at Array.forEach (native)
    at Mocha.loadFiles (/root/projects/test/node_modules/mocha/lib/mocha.js:216:
14)
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-26. Mocha test on invalid source code

In this example, the Mocha test displays a reference error that the variable that is named `baz` is not defined.

Although Mocha can catch these types of errors, a static code analysis tool such as ESLint should be run as a pretest to the test suite to catch errors in the source code.

Mocha is designed as a test tool and not necessarily a code validation tool.

3.3. Supertest



Figure 3-27. Supertest

Topics

- Static code analysis with linting utilities
- Testing with Mocha
- ▶ Supertest



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-28. Topics

Supertest

- High-level abstraction for testing HTTP
 - Can drop down to lower-level API provided by super-agent
 - HTTP assertions are made easy with super-agent
- Simplifies the task of making HTTP requests
- Compare the response against asserted values
- Works with any test framework, such as Mocha, or without using any test framework at all

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-29. Supertest

Supertest simplifies the task of HTTP requests because you can make HTTP requests from your test code rather than typing them in a browser.

Supertest installation

- Install with npm
 - Install Supertest as an npm module
 - `npm install supertest --save-dev`
 - Save it to your `package.json` file as a development dependency
- When installed, supertest can be referenced with
 - `require('supertest');`

Figure 3-30. Supertest installation

Install Supertest with npm in the root folder of the package and save it as a development dependency.

API: Super-agent methods

- Assert response status code
 - `.expect(status [,fn])`
- Assert response status code and body
 - `.expect(status, body [,fn])`
- Assert response body text
 - `.expect(body [,fn])`
- Assert header field value
 - `.expect(field, value [,fn])`
- Perform the request and invoke `fn(err, res)`
 - `.end(fn)`

Figure 3-31. API: Super-agent methods

The page shows some of the commonly used super-agent methods.

Test case: /api/weather/KSFO



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-32. Test case: /api/weather/KSFO

The figure shows the output in the browser when the `http://localhost:3000/api/weather/KSFO` application is run.

Next, you create the Mocha and supertest unit test for testing this application.

Create a unit test with Mocha and supertest

- Create a file that is named `super-weather.js` in the `/test` folder
- Code the following unit test case:
 - Include `require('server.js')`
 - Access the `/api/weather/KSFO` page
 - Should return a response code `200`
- Provide a description of the unit test by using the “describe” function
- Use the “it” function to describe one or several unit test cases
- The “done” function is used to indicate the end of the test case and provide the callback

Figure 3-33. Create a unit test with Mocha and supertest

Unit test source code

```
var server = require('../server.js');
var expect = require('expect.js');
var supertest = require('supertest');
var request = supertest.agent('http://localhost:3000');
describe('/api/weather/KSFO', function() {
  it('should return temperature in San Francisco',
    function(done) {
      request
        .get('/api/weather/KSFO')
        .expect(200)
        .end(function(err, res) {
          expect(res.status).to.equal(200);
          done();
        });
    });
});
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-34. Unit test source code

In the example, you see the code for the unit test that was described previously.

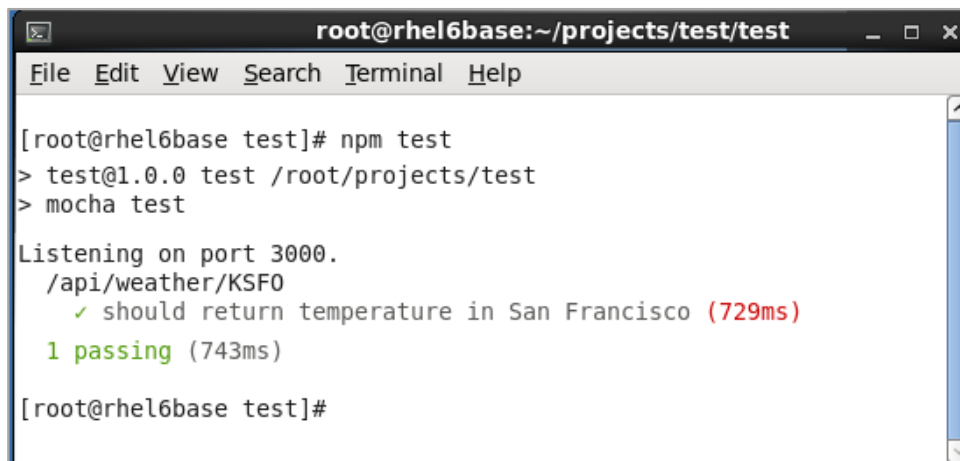
You can use the `supertest` agent to test the HTTP call without needing to type it in the browser.

The `describe` function gives a description of the test case.

The `it` function describes each unit test, what to run, and the expected result.

Output from the unit test

- Run npm test
 - Runs the Mocha test
 - Starts the server application
 - Displays the result

A terminal window titled 'root@rhel6base:~/projects/test/test' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of 'npm test', which runs 'mocha test'. The output indicates the server is listening on port 3000, a test for '/api/weather/KSFO' passed with a message '✓ should return temperature in San Francisco (729ms)', and a final summary '1 passing (743ms)'.

```
root@rhel6base:~/projects/test/test
File Edit View Search Terminal Help

[root@rhel6base test]# npm test
> test@1.0.0 test /root/projects/test
> mocha test

Listening on port 3000.
  /api/weather/KSFO
    ✓ should return temperature in San Francisco (729ms)
  1 passing (743ms)

[root@rhel6base test]#
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-35. Output from the unit test

Create a unit test for the error condition

- Create a file that is named `super-error.js` in the `/test` folder

Code the following unit test case:

- Include `require('server.js')`
 - Invoke an invalid URL
 - Should return a response code `404`
-
- Provide a description of the unit test by using the “describe” function
 - Use the “it” function to describe one or several unit test cases
 - The “done” function is used to indicate the end of the test case and provide the callback

Figure 3-36. Create a unit test for the error condition

Unit test error source code

```
var server = require('../server.js');
var expect = require('expect.js');
var supertest = require('supertest');
var request = supertest.agent('http://localhost:3000');
describe('invalid URL', function() {
  it('should return 404 not found', function(done) {
    request
      .get('/random')
      .expect(404)
      .end(function(err, res) {
        expect(res.status).to.equal(404);
        done();
      });
  });
});
```

Static code analysis and unit testing

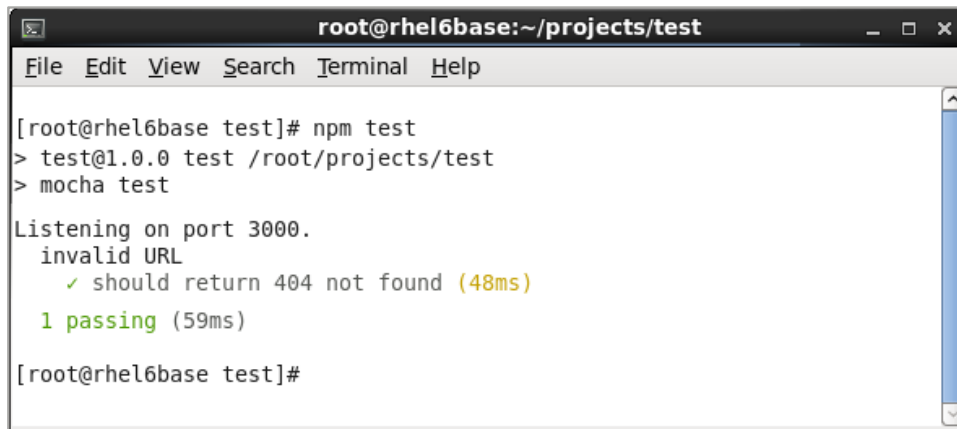
© Copyright IBM Corporation 2015, 2019

Figure 3-37. Unit test error source code

In the example, you see the code for the unit test for the invalid URL that was described previously. You can use the `supertest` agent to test the HTTP call without needing to type it in the browser. The `describe` function gives a description of the test case. The `it` function describes each unit test, what to run, and the expected result.

Output from the error unit test

- Run npm test
 - Run the Mocha test
 - Start the server application
 - Display the result



```
root@rhel6base:~/projects/test
File Edit View Search Terminal Help

[root@rhel6base test]# npm test
> test@1.0.0 test /root/projects/test
> mocha test

Listening on port 3000.
  invalid URL
    ✓ should return 404 not found (48ms)
  1 passing (59ms)

[root@rhel6base test]#
```

Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-38. Output from the error unit test

Unit summary

- Describe how to validate JavaScript code with ESLint
- Describe how to unit test your application with Mocha
- Explain how to create HTTP unit tests with Supertest



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-39. Unit summary

Review questions



1. True or False: Setting no-console to “off” in `.eslintrc.js` disallows the use of the console.
2. True or False: When using expect style assertions in Mocha, you must load one of the additional assertion libraries.
3. Which of these utilities uses many customizable rules?
 - A. JSHint
 - B. JSLint
 - C. ESLint
 - D. Lint

Figure 3-40. Review questions

Write your answers here:

- 1.
- 2.
- 3.

Review answers



1. True or False: Setting `no-console` to “off” in `.eslintrc.js` disallows the use of the console.
The answer is False. When you set the `no-console` rule to “off”, the ESLint static code analysis engine does not flag the use of the `console` object.
2. True or False: When using expect style assertions in Mocha, you must load one of the additional assertion libraries.
The answer is True.
3. Which of these utilities uses many customizable rules?
 - A. JSHint
 - B. JSLint
 - C. ESLint
 - D. LintThe answer is C.

Figure 3-41. Review answers

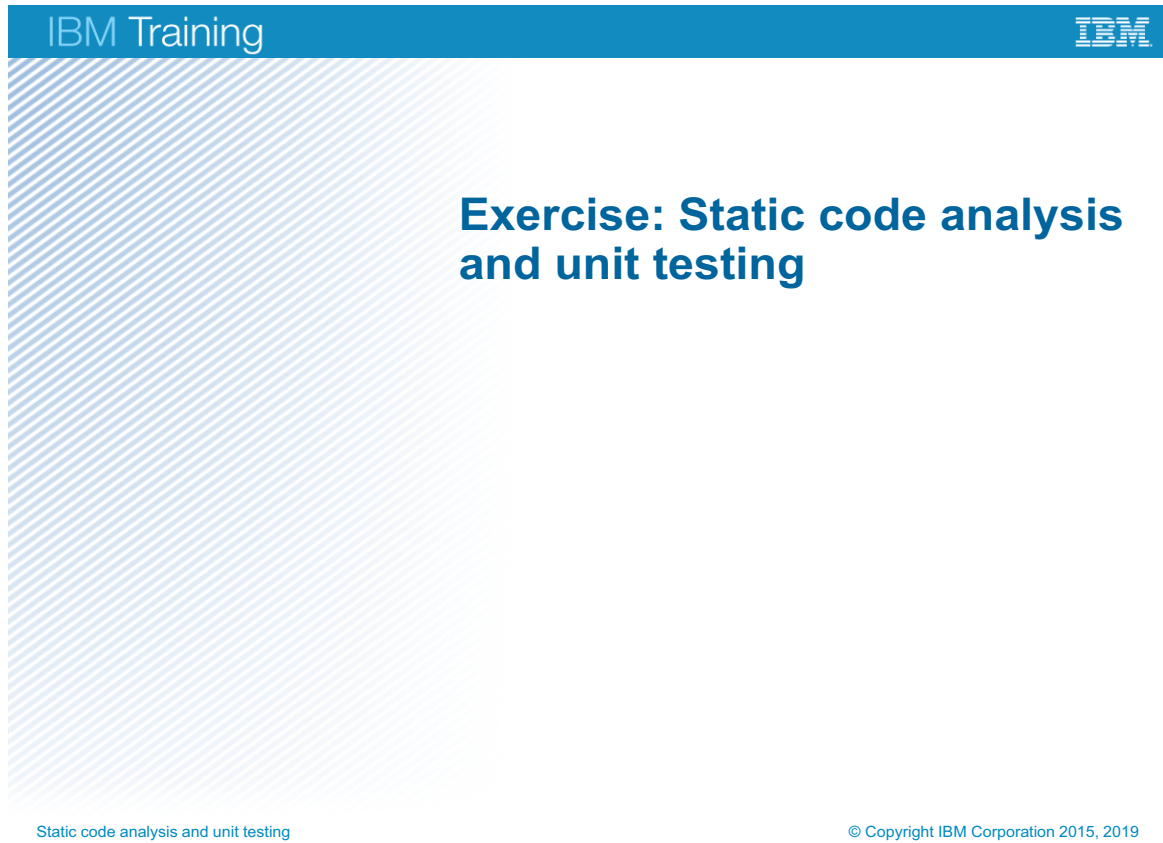


Figure 3-42. Exercise: Static code analysis and unit testing

Exercise objectives



- Explain the purpose of static code analysis
- Explain the purpose of unit testing
- Perform static code analysis with *ESLint*
- Create and run a function test suite in *Mocha*
- Create and run a web application test suite in *Supertest*



Static code analysis and unit testing

© Copyright IBM Corporation 2015, 2019

Figure 3-43. Exercise objectives

Unit 4. Debugging and building Node applications

Estimated time

00:45

Overview

This unit describes some of the utilities and techniques that are used to debug and package Node applications.

How you will check your progress

- Review questions
- Lab exercise

Unit objectives

- Describe the Read, Evaluate, Print, and Loop (REPL) capability of Node
- Describe the use of the console to log diagnostic information
- Describe the standard command-line debug feature of Node
- Examine the Node Inspector graphical debug tool
- Describe the purpose of the npm shrinkwrap utility
- Describe the use of npm as a build tool



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-1. Unit objectives

4.1. REPL



Figure 4-2. REPL

Topics

- ▶ REPL
 - Using the console
 - Node built-in debugger
 - Node Inspector
 - Shrinkwrap
 - Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-3. Topics

Read, Evaluate, Print, Loop (1 of 2)

- Command shell for interactive evaluation of node code
- Valid JavaScript can be typed into REPL
- Useful for experimenting or debugging Node.js code

```
$ node
>

$ node
> function label() { return "bar" }
> Undefined
> label()
> 'bar'
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-4. Read, Evaluate, Print, Loop (1 of 2)

The Node.js REPL terminal stands for Read, Evaluate, Print, Loop.

REPL is a command-line environment like a Windows command prompt or a Linux shell where a command is typed and the system responds with output in an interactive mode.

Node.js comes bundled with a REPL environment.

REPL is useful for experimenting or for troubleshooting Node.js code.

Read, Evaluate, Print, Loop (2 of 2)

- Identifiers can be used with or without the `var` keyword

```
$ node
> var date = new Date();
undefined
> date
Thu Mar 31 2016 13:36:50 GMT-0400 (EDT)
> var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
undefined
> days[date.getDay()];
'Thu'
```

Figure 4-5. Read, Evaluate, Print, Loop (2 of 2)

When the `var` keyword is used, the value of the expression is stored, but not returned. When an identifier is used without the `var` keyword, the value is stored and also returned.

4.2. Using the console

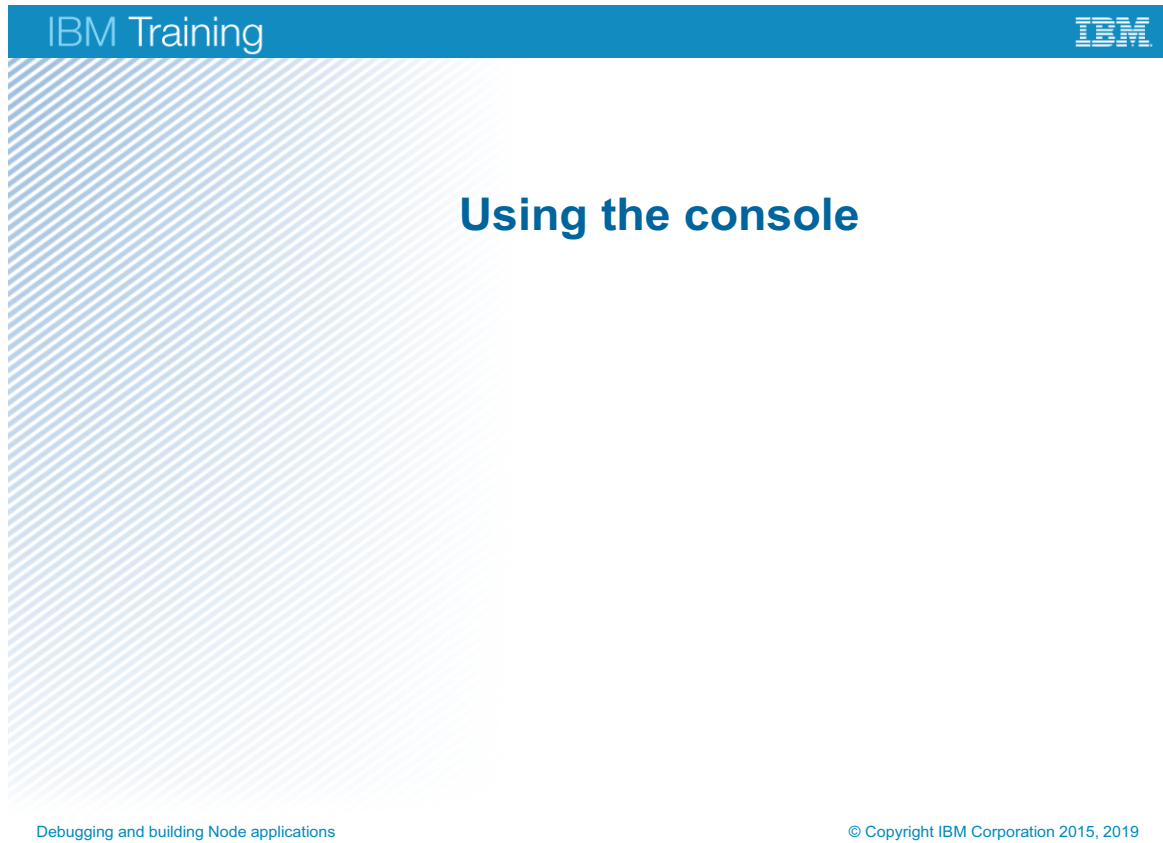


Figure 4-6. Using the console

Topics

- REPL
- ▶ Using the console
 - Node built-in debugger
 - Node Inspector
 - Shrinkwrap
 - Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-7. Topics

Using the console to log messages (1 of 4)

- Log diagnostic information to help debug a web page or application
 - Log messages are written to the console in Node.js
 - Output is displayed in a terminal window for a non-web application
 - For web applications, output can be displayed in the Developer Tools Console of the Chrome browser, when using a debug plug-in like Node Inspector
- Sending output to the console requires adding JavaScript statements to the source code

```
console.log('Some message');  
console.info('Information message..');  
console.warn('This might be a problem..');  
console.err('Error: %s (%i)', 'Server is not responding', 500);  
console.log('The current time is: ', Date.now());
```

Figure 4-8. Using the console to log messages (1 of 4)

The console log is an operating system log file. If you have a terminal window open while you run a node application, the console log output appears in the terminal.

In a browser such as Google Chrome, the Developer Tools Console can read the console log if it is attached to the debug process with the Node Inspector plug-in.

The output from the console is then displayed in the console window of the Developer tools for web pages.

The JavaScript console is available from the Chrome stacked (three lines) menu. Click **More tools** > Developer tools.

Using the console to log messages (2 of 4)

- The command `console.error()` does not automatically print a stack trace

```
var err = new Error('Oops...');  
...  
console.error(err);
```

- The terminal displays:

```
[Error: Oops...]
```

Figure 4-9. Using the console to log messages (2 of 4)

The `console.error()` function prints only the error and does not automatically display a stack trace.

Using the console to log messages (3 of 4)

- Use the stack property of the error object to display the stack trace

```
var err = new Error('Oops...');  
...  
console.error(err.stack);
```

- The terminal displays:

```
Error: Oops...  
  at doGet (/path/to/some-module.js:25:19)  
  at Server.handleRequests (/path/to/server.js:7:5)  
  ...
```

Figure 4-10. Using the console to log messages (3 of 4)

Include the stack property of the error object to display the stack trace.

Using the console to log messages (4 of 4)

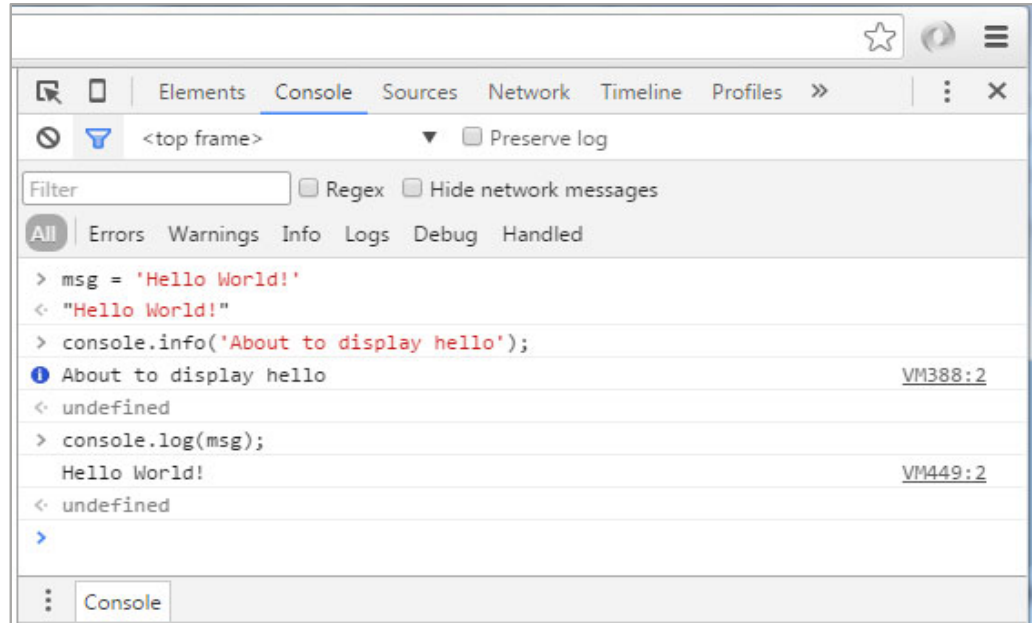
- Output is sent to `stdout` or `stderr`

```
console.log('message', data);    //stdout
console.info('message');         //stdout

console.warn('warning');         //stderr
console.error(err);              //stderr
```

Figure 4-11. Using the console to log messages (4 of 4)

Console output for web applications



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-12. Console output for web applications

The figure shows the **Console** tab of the Developer tools in the Chrome browser. Console messages can be filtered by type, such as errors, warnings, information, logs, debug, and handled. For more information about using the console on the Chrome browser, see: <https://developer.chrome.com/devtools/docs/console>

Notice that the web browser interacts with the Node runtime with a plug-in such as Node Inspector. The Node Inspector plug-in for Node.js piggybacks on the Google Chrome Developer tools, which were meant to monitor and debug client-side JavaScript. Node Inspector is discussed later in this unit.

4.3. Node built-in debugger

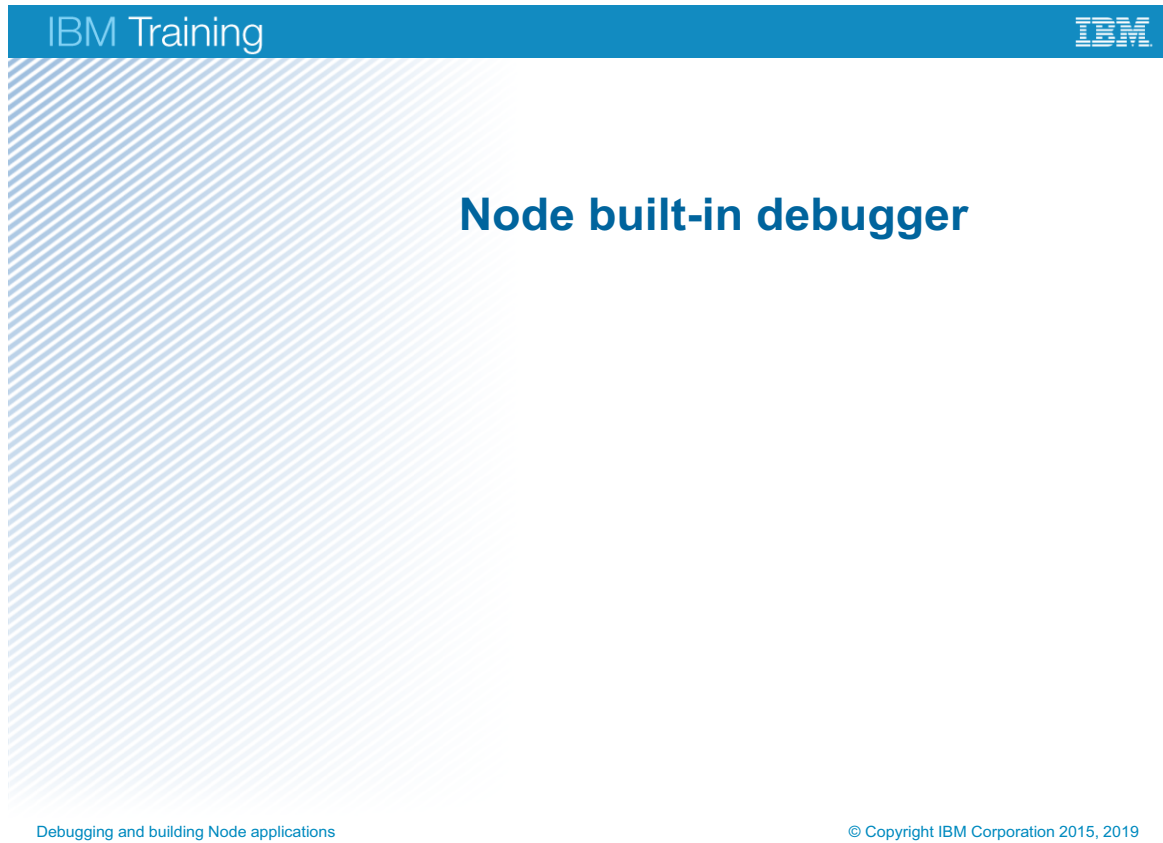


Figure 4-13. Node built-in debugger

Topics

- REPL
- Using the console
- ▶ Node built-in debugger
 - Node Inspector
 - Shrinkwrap
 - Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-14. Topics

Node built-in debugging utility

- Standard command-line debug utility for Node

```
node debug server.js
< Debugger listening on port 5858
debug> . Ok
break in server.js:1
> 1 var today = require('./today');
    2 var http = require('http');
    3 var port = process.env.VCAP_APP_PORT || 3000
debug>
(To exit, press ^C again or type .exit)
debug>
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-15. Node built-in debugging utility

To use the standard debugger that comes with Node, type the `debug` keyword when running the `node` command. The debug utility starts listening on port 5858 and breaks on the first line of the program. You can enter commands from the debug prompt. Simple step and inspection commands are available.

Inserting the statement `debugger;` into the source code of a script enables a breakpoint at that position in the code. You can also use the `setBreakpoint(line)` from the debug prompt to set a breakpoint at a particular line in the source code.

The following lists the stepping command reference:

- `cont`, `c`: Continue execution
- `next`, `n`: Step next
- `step`, `s`: Step in
- `out`, `o`: Step out
- `pause`: Pause running code

For more information about the commands, see: <https://nodejs.org/api/debugger.html>

Node built-in debugging example

```
status-app_v1.0.3 — node — 80x24
Kevins-MacBook-Pro:status-app_v1.0.3 kevin_home$ node debug server.js
< Debugger listening on port 5858
connecting to 127.0.0.1:5858 ... ok
break in server.js:1
> 1 var port = (process.env.VCAP_APP_PORT || 3000);
  2
  3 var today = require('./today');
debug> setBreakpoint('today.js',2)
Warning: script 'today.js' was not loaded yet.
> 1 var port = (process.env.VCAP_APP_PORT || 3000);
  2
  3 var today = require('./today');
  4
  5 var request = require('request');
  6 var express = require('express');
debug> c
< Listening on port 3000.
break in today.js:2
  1 module.exports = function() {
> 2   var date = new Date();
  3   var days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Satur
day', 'Sunday'];
  4   return days[ date.getDay() - 1 ];
debug>
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-16. Node built-in debugging example

This figure shows the output from running the command `node debug <application>`.

The figure also shows an example of using the `setBreakpoint` command in the debug prompt.

Use debug with REPL

- When stepping through the code in the debugger, type: `repl`

```
debug> setBreakpoint(3)
debug> c
break in today.js:3
    1 module exports = function() {
    2   var date = new Date();
>   3   var days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'];
    4   return days[ date.getDay() - 1];
    5 }
debug> repl
Press Ctrl + C to leave debug repl
> date
> Wed Mar 09 2106 14:43:32 GMT-0800 (PST)
debug>
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-17. Use debug with REPL

You can use REPL to evaluate JavaScript expressions when running the standard debugger.

Type `repl` at the debug prompt, and then you can type REPL expressions.

Pressing Ctrl+C returns you to the debug prompt.

A useful function of the REPL feature when debugging is that you can write arbitrary code to test the logic of your application, without changing the code inside the application.

In this example, you retrieved the value of the `date` variable without writing a line of `console.log` code.

4.4. Node Inspector

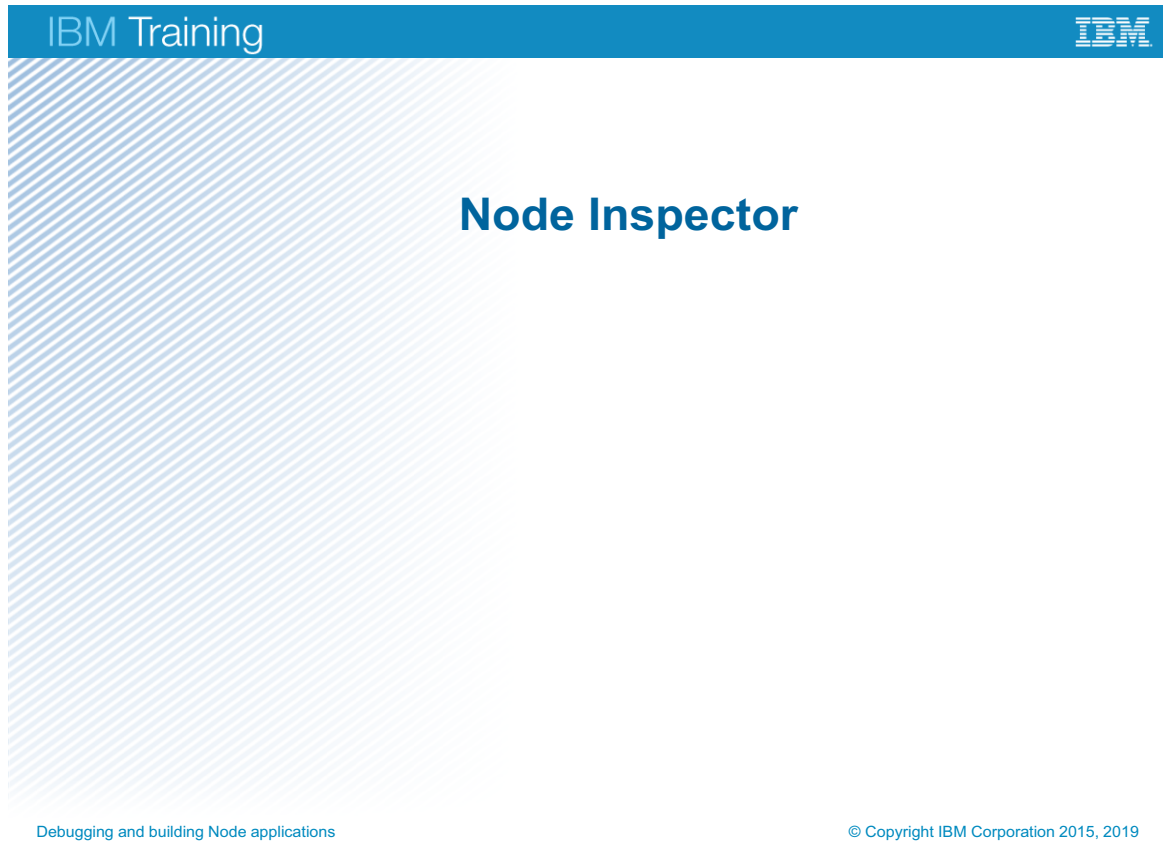


Figure 4-18. Node Inspector

Topics

- REPL
- Using the console
- Node built-in debugger
- ▶ Node Inspector
 - Shrinkwrap
 - Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-19. Topics

Using Node Inspector

- Graphical debug environment for Node.js when used with the Google Chrome browser
- For older versions of Node (before V6) you could install the node-inspector to connect with the Chrome Development tools with:
`npm install -g node-inspector`
- With later versions of Node, you do not have to install the node-inspector utility
- As of May 2016, you can perform advanced debugging by using the `--inspect` option when you run your Node application

```
$ node --inspect server.js
```

- Go to your Chrome browser and type `chrome://inspect` in the address area
- The Chrome Developer Tools page is displayed
 - Click the `inspect` link to open the file in the graphical debugger

Figure 4-20. Using Node Inspector

Node Inspector can be used with the Chrome browser to provide a graphical debug environment for Node.js.

For older versions of Node, use npm to install Node Inspector.

After Node Inspector is installed, you can type `node-inspector` in the terminal to start Node Inspector. The Node Inspector opens in a browser tab. Then, type `node --debug <application-name>`. Refresh the browser session where Node Inspector is running to see the application source code.

For newer versions of Node, you do not need to install the node-inspector utility.

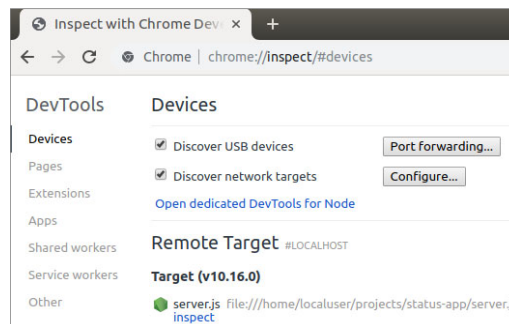
You can perform advanced debugging by using the `--inspect` option when you run your Node application.

Start Node Inspector

- Debug an application with Node Inspector

```
$ node --inspect-brk server.js  
Debugger listening on ws://127.0.0.1:9229/bbd335fb-f9e2  
For help, see: https://nodejs.org/en/docs/inspector
```

- The `--inspect-brk` flag is used to break on the first statement in the script file
- Type `chrome://inspect` in the Chrome browser
 - Then click the Open dedicated DevTools for Node or inspect link



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-21. Start Node Inspector

To perform advanced graphical debugging with the Chrome Development Tools, download the current version of Node (v6.3.0 or later).

Run the node application with the `--inspect` flag.

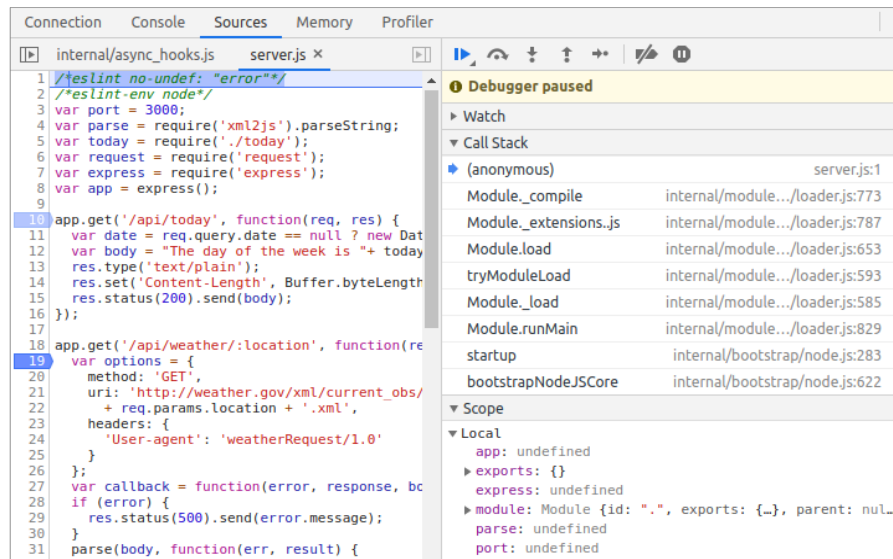
When the debugger starts, open the Chrome browser and type `chrome://inspect` in the address area.

Then, click the Open dedicated DevTools for Node or the inspect link on the page.

The advanced graphical debug inspector page opens in the browser.

Node Inspector in the Chrome browser

- Powerful JavaScript debugging interface



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-22. Node Inspector in the Chrome browser

After you start the node-inspector utility, it launches the Google Chrome browser and the Developer tools. The tools listen to the default Node.js debug port at 9229.

The Developer tools now interact and control the Node.js debugger through port 9229.

The figure shows the Node Inspector graphical debugger in the browser.

The application is paused at the first line of code that is indicated by the blue highlighted line in the middle pane of the browser.

The Node Inspector graphical interface works only with the Chrome browser.

Node Inspector features

- Browse in the source files
- Set or deactivate breakpoints
- Step over, step in, step out, resume, pause on exceptions
- Watch expressions
- Monitor the call stack
- Display scope variables
- Hover over an expression in the source to display its value
- Edit variables and object properties
- Continue to location
- Profile CPU and heap
- Inspect console output

Debugging and building Node applications

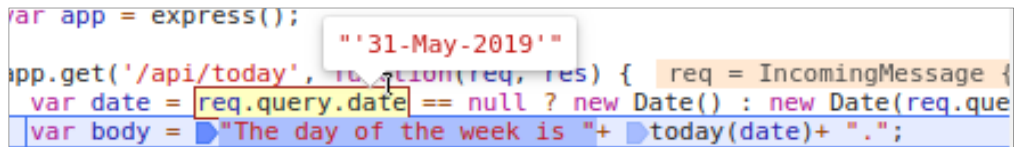
© Copyright IBM Corporation 2015, 2019

Figure 4-23. Node Inspector features

Node Inspector supports almost all of the debugging features of the Chrome Developer tools.

Node Inspector hover feature

- Node Inspector works similarly to the Chrome Developer Tools
- Hover over a keyword or expression in the source to display its value in a tooltip

A screenshot of a code editor showing JavaScript code. A yellow tooltip is displayed over the expression 'req.query.date', showing the value "'31-May-2019'". The code includes a function definition for 'app.get' and a conditional statement for 'var date'.

```
var app = express();  
app.get('/api/today', function(req, res) { req = IncomingMessage {  
  var date = req.query.date == null ? new Date() : new Date(req.que  
  var body = "The day of the week is " + today(date) + ".";
```

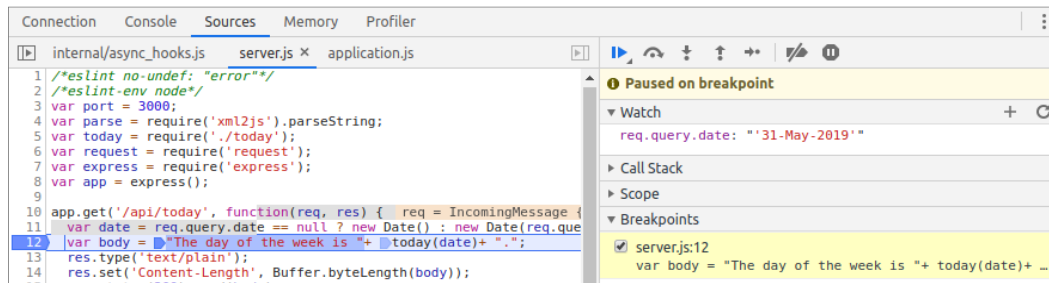
Figure 4-24. Node Inspector hover feature

The figure shows the hover feature of the Node Inspector in the Chrome browser.

When the cursor is hovered over a JavaScript keyword, a pop-up window opens with details about the keyword.

Node Inspector with multiple files open

- Click a file in the Sources pane to open it
- Click a line number to set a breakpoint
 - A breakpoint is set at line 12 of server.js



Debugging and building Node applications

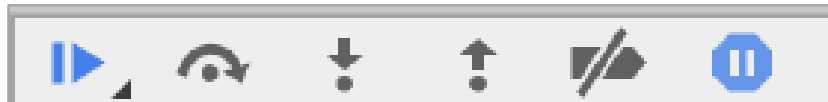
© Copyright IBM Corporation 2015, 2019

Figure 4-25. Node Inspector with multiple files open

You can display the source code for any of the files by clicking the file in the Sources pane. Breakpoints can be set by clicking the line number vertical column to the left of the source code. In the figure, a breakpoint is set on line 12 of `server.js`.

Manage debug execution with controls

- Resume script execution (F8)
- Step over next function call (F10)
- Step into next function call (F11)
- Step out of current function (Shift+F11)



Also:

- Deactivate breakpoints
- Pause on exceptions

Figure 4-26. Manage debug execution with controls

You manage processing through the source file by using the debug execution control icons or keyboard shortcuts.

4.5. Shrinkwrap

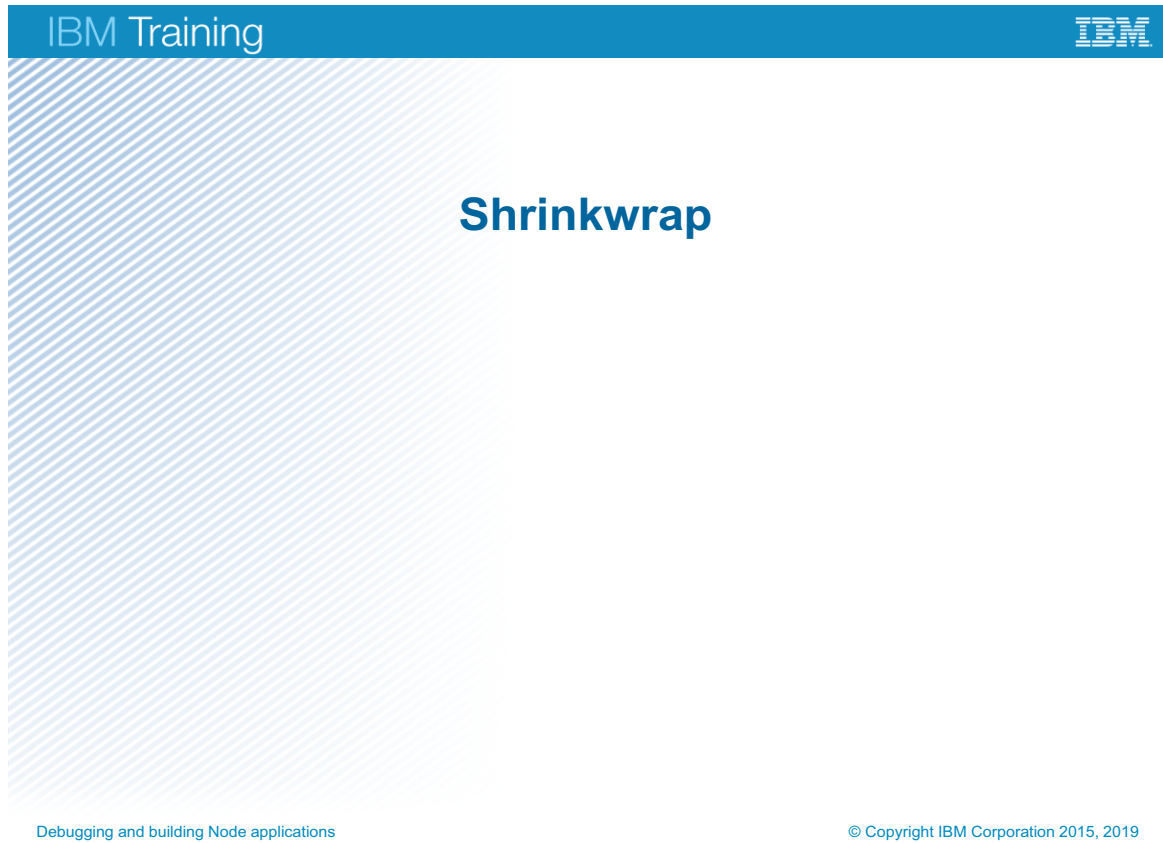


Figure 4-27. Shrinkwrap

Topics

- REPL
- Using the console
- Node built-in debugger
- Node Inspector
- ▶ Shrinkwrap
- Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-28. Topics

The purpose of npm shrinkwrap

- By default, `npm install` recursively installs all the target dependencies as specified in the `package.json` file
 - Chooses the current available version that satisfies the dependency's semantic versioning (semver) pattern in `package.json`
- For production or application cloning, it is desirable to fully specify the version of each dependency so that subsequent builds use the same versions and do not inadvertently pick newer versions
- The `npm shrinkwrap` command is used to lock down the versions of all the dependent modules of an application
 - Controls exactly which versions of each dependency is used when your package is installed

Figure 4-29. The purpose of npm shrinkwrap

Managing dependencies is a fundamental problem in building complex software. The problem of managing module dependencies is especially relevant when building Node.js applications.

The packages that make up node modules do not exist in isolation but rather as nodes in a large graph. The software is constantly changing (releasing new versions), and each package has its own constraints about what other packages it requires to run (dependencies).

Semantic versioning allows application authors to specify which versions of packages and dependencies can be used for the application to work correctly.

By default, `npm install` recursively installs all the target dependencies as specified in `package.json`, choosing the most recent available version that satisfies the dependency's semantic versioning (semver) pattern in `package.json`.

If you always install the most recent version of the software that satisfies the semantic versioning, you might have the following problem: if you run `npm install` again at a different time, you cannot guarantee that you get the same set of code dependencies.

By running the `npm shrinkwrap` command immediately after running `npm install`, you can lock down the versions of a package's dependencies to match the versions that are chosen by running `npm install`.

**Information**

Instead of using npm shrinkwrap to lock the module versions, you can use the built-in package lock file. This approach is described in the course exercise that accompanies this unit.

Building shrinkwrapped packages

- To shrinkwrap an existing package:
 1. Run `npm install` in the package root to install the current versions of all dependencies
 2. Validate that the package works as expected with these versions
 3. Run `npm shrinkwrap` to generate the `npm-shrinkwrap.json` file
 4. Publish your package
- Subsequent `npm install` commands that are run in the package root use the `npm-shrinkwrap.json` file rather than the `package.json` file to recursively install all the package dependencies

Figure 4-30. Building shrinkwrapped packages

The steps that are used to build a shrinkwrapped package for a node.js application are provided on the slide.

When `npm install` installs a package with an `npm-shrinkwrap.json` file in the package root, the shrinkwrap file (rather than `package.json` files) completely drives the installation of that package and all of its dependencies.

Updating shrinkwrap packages

- To find out which packages are outdated, from the package root directory, run:
`npm outdated`
 - The command informs you of any outdated packages in the `shrinkwrap.json` file
- To update any outdated package, run:
`npm update <package_name>`
- Retest the application
- Rerun `npm shrinkwrap`
- Publish your package

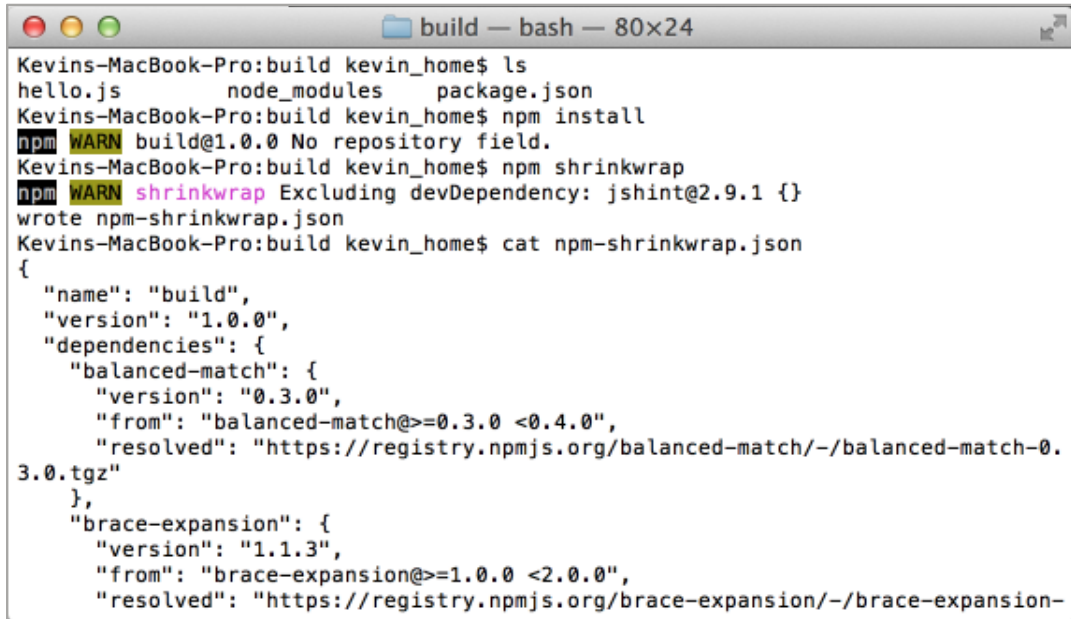
Figure 4-31. Updating shrinkwrap packages

One way to update dependency versions in shrinkwrap packages is to first run the `npm outdated` command. This command reads the repository and informs you of any outdated packages.

You can examine the list of outdated packages and decide whether you want to include them in your production software. Then, run the command `npm update <package_name>`.

Thoroughly test the application. Then, rerun `npm shrinkwrap` and publish your package.

Shrinkwrap example



```

Kevins-MacBook-Pro:build kevin_home$ ls
hello.js      node_modules  package.json
Kevins-MacBook-Pro:build kevin_home$ npm install
npm WARN build@1.0.0 No repository field.
Kevins-MacBook-Pro:build kevin_home$ npm shrinkwrap
npm WARN shrinkwrap Excluding devDependency: jshint@2.9.1 {}
wrote npm-shrinkwrap.json
Kevins-MacBook-Pro:build kevin_home$ cat npm-shrinkwrap.json
{
  "name": "build",
  "version": "1.0.0",
  "dependencies": {
    "balanced-match": {
      "version": "0.3.0",
      "from": "balanced-match@>=0.3.0 <0.4.0",
      "resolved": "https://registry.npmjs.org/balanced-match/-/balanced-match-0.3.0.tgz"
    },
    "brace-expansion": {
      "version": "1.1.3",
      "from": "brace-expansion@>=1.0.0 <2.0.0",
      "resolved": "https://registry.npmjs.org/brace-expansion/-/brace-expansion-

```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-32. Shrinkwrap example

When you run `npm install` from the package root directory, the npm program installs the current version of all dependencies for the packages that are in the `node_modules` subdirectory.

The `npm shrinkwrap` command creates a file that is named `npm-shrinkwrap.json`.

By default, the `npm shrinkwrap` command excludes development dependencies.

To include development dependencies, use the command `npm shrinkwrap -dev`.

When you review the `npm-shrinkwrap.json` file, you see that the version number for each of the dependencies is fixed. The `npm-shrinkwrap.json` file also specifies the exact file to load the dependency version.

4.6. Build

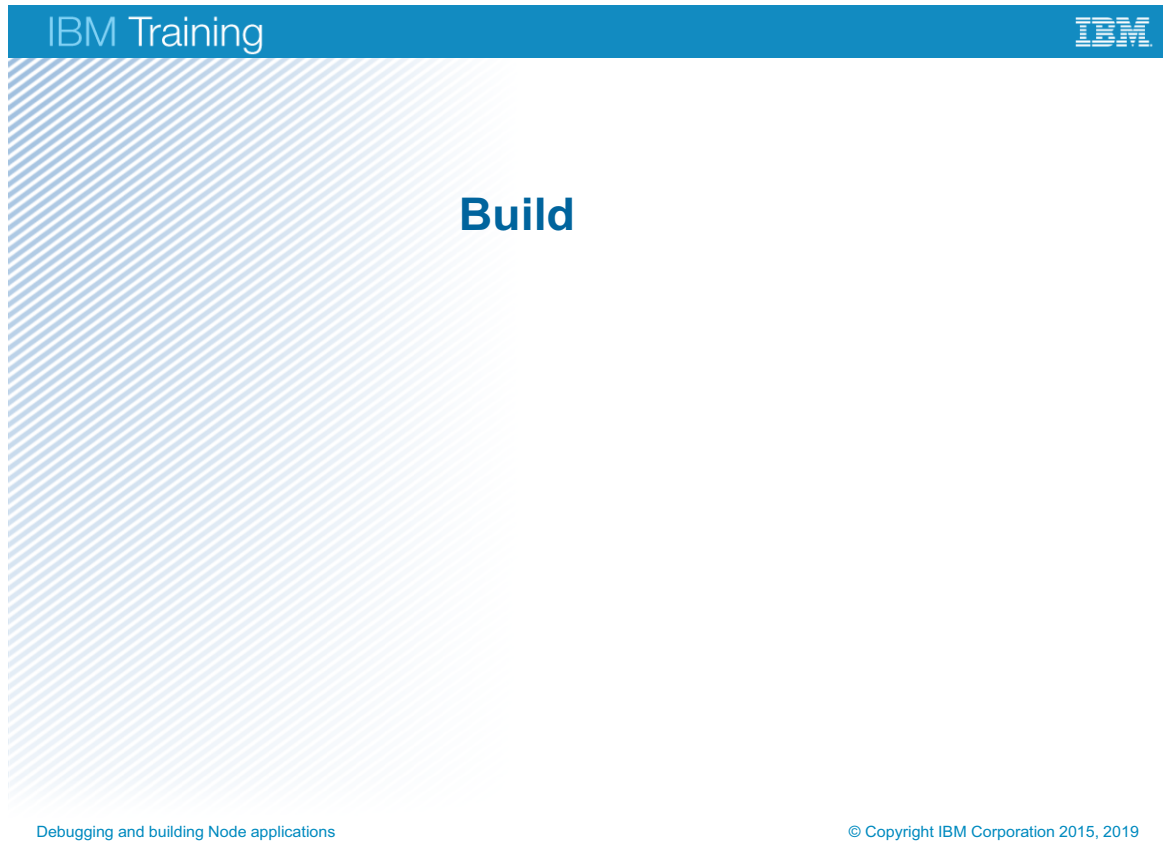


Figure 4-33. Build

Topics

- REPL
- Using the console
- Node built-in debugger
- Node Inspector
- Shrinkwrap
- ▶ Build



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-34. Topics

Use npm as a build tool

- A way to automate common tasks by using `npm` itself
- The `package.json` file contains a `scripts` object where the tasks are registered
 - Each task in the `scripts` object is a command that can be run

```
"scripts": {
  "env": "echo 'Show node environment' && env",
  "lint": "echo 'Running jshint now' && jshint hello.js",
  "test": "echo 'Running test' && node hello.js",
  "build": "echo 'Run build' && npm run lint && npm run test"
},
"devDependencies": {
  "jshint": "^2.9.1"
}
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-35. Use npm as a build tool

You can use npm to automate common tasks and create a workflow for Node packages.

The `package.json` file contains a `scripts` object where the tasks are registered, where each task in the `scripts` object is a command that can be run.

In the example, four scripts (tasks) can be run: `env`, `lint`, `test`, and `build`.

The advantage in using npm scripts is that packages do not have to be installed globally.

Running the command `npm install jshint --save-dev` installs the `jshint` package in the `node_modules` folder and adds the `devDependencies` to the `package.json` file.

After installing JSHint, you can run the command `npm run lint` to run the `lint` task of the `scripts` object that is declared in the `package.json` file.

How npm manages scripts

- If you type `npm run` without any arguments, a list of available scripts is displayed

```
$ npm run
Lifecycle scripts included in build:
  test
    echo 'Running test' && node hello.js
available via 'npm run-script':
  env
    echo 'Show node environment' && env
  lint
    echo 'Running jshint now' && jshint hello.js
  build
    echo 'Run build' && npm run lint && npm run test
```

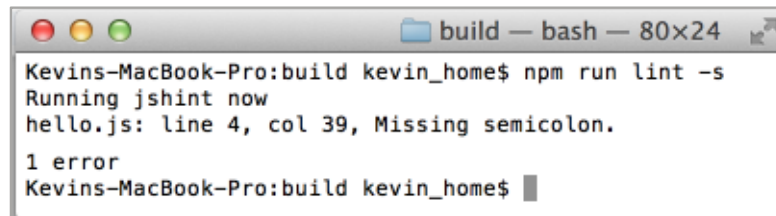
Figure 4-36. How npm manages scripts

If you type `npm run` without any arguments, a list of available tasks (properties) in the script object of the `package.json` file is displayed.

Run scripts

```
"scripts": {  
  "env": "echo 'Show node environment' && env",  
  "lint": "echo 'Running jshint now' && jshint hello.js",  
  "test": "echo 'Running test' && node hello.js",  
  "build": "echo 'Run build' && npm run lint && npm run test"  
},
```

- To run any of the tasks, type: `npm run <script-name>`
- Use the `-s` option to silently run the script without displaying all the npm output



```
build — bash — 80x24  
Kevins-MacBook-Pro:build kevin_home$ npm run lint -s  
Running jshint now  
hello.js: line 4, col 39, Missing semicolon.  
1 error  
Kevins-MacBook-Pro:build kevin_home$
```

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-37. Run scripts

To run any of the tasks that are coded in the `scripts` object of the `package.json` file, type `npm run <script-name>`. Default scripts such as `test` and `start` can also be invoked without the `run` argument, as in `npm test` or `npm start`.

Chain scripts

```
"scripts": {  
  "env": "echo 'Show node environment' && env",  
  "lint": "echo 'Running jshint now' && jshint hello.js",  
  "test": "echo 'Running test' && node hello.js",  
  "build": "echo 'Run build' && npm run lint && npm run test"  
},  
...
```

- You can chain tasks within the scripts object by using the `&&` syntax
 - The `echo` command is chained with the `env` command in the `env` script
- You can chain any number of scripts together to make a build script that can include validation, testing, and running the code
 - Run scripts inside npm scripts

Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-38. Chain scripts

You can chain tasks within the scripts object by using the `&&` syntax. The `echo` command and the `jshint` command are both run when `npm run lint` is invoked from the terminal.

You can run scripts within scripts to build a script that includes syntax validation, testing, and code execution.

Scripts can also be used to copy files, make directories, and for other development tasks.

Unit summary

- Describe the Read, Evaluate, Print, and Loop (REPL) capability of Node
- Describe the use of the console to log diagnostic information
- Describe the standard command-line debug feature of Node
- Examine the Node Inspector graphical debug tool
- Describe the purpose of the npm shrinkwrap utility
- Describe the use of npm as a build tool



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-39. Unit summary

Review questions



1. True or False: If you type the literal `x=10` into the REPL interactive command, it displays and stores the value of X.
2. True or False: You can start the REPL interactive command by typing `node` in a terminal prompt or by typing `repl` when you are at the debug prompt of the built-in debugger.
3. Which of these commands is used to start the built-in debug utility of Node?
 - A. `node-debug <application-name>`
 - B. `node debug <application-name>`
 - C. `node --debug <application-name>`
 - D. `node <application-name> --debug`

Figure 4-40. Review questions

Write your answers here:

- 1.
- 2.
- 3.

Review answers



1. True or False: If you type the literal `x=10` into the REPL interactive command, it displays and stores the value of X.
The answer is True.
2. True or False: You can start the REPL interactive command by typing `node` in a terminal prompt or by typing `repl` when you are at the debug prompt of the built-in debugger.
The answer is True.
3. Which of these commands is used to start the built-in debug utility of Node?
 - A. `node-debug <application-name>`
 - B. `node debug <application-name>`
 - C. `node --debug <application-name>`
 - D. `node <application-name> --debug`.The answer is B.

Figure 4-41. Review answers

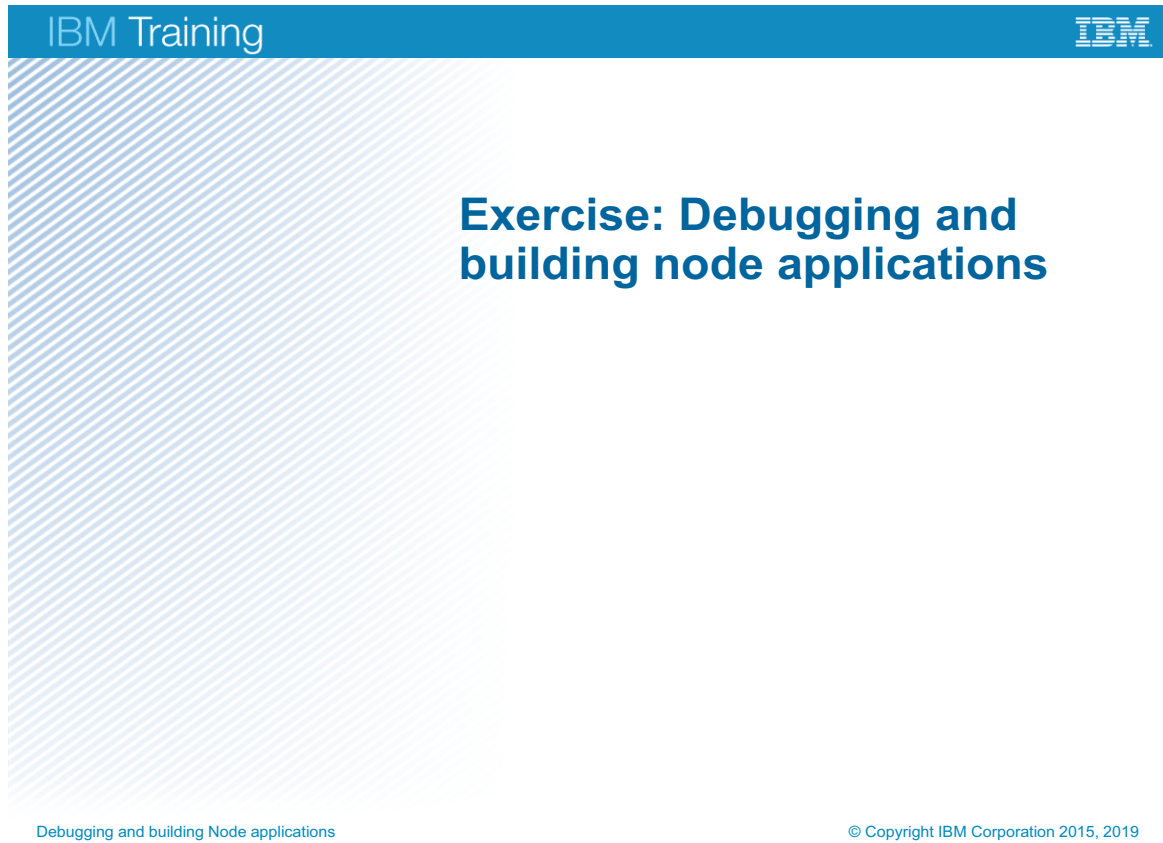


Figure 4-42. Exercise: Debugging and building node applications

Exercise objectives



- Use the standard node debug utility of Node
- Enable Node Inspector
- Work with the Node Inspector graphical debug tool
- Use package lock to set node module versions
- Use script objects and npm to build node applications



Debugging and building Node applications

© Copyright IBM Corporation 2015, 2019

Figure 4-43. Exercise objectives

Unit 5. Deploying and testing Node applications on IBM Cloud

Estimated time

01:00

Overview

This unit describes the architecture of IBM Cloud. Describe the features of the IBM Cloud Development Tools that are used to enable and deploy cloud applications. Explain how to deploy a Node.js application to IBM Cloud.

How you will check your progress

- Review questions
- Lab exercises

Unit objectives

- Describe the architecture of IBM Cloud
- Describe the features of the IBM Cloud Developer Tools
- Describe what is Cloud Foundry
- Describe how to install the IBM Cloud Developer Tools command-line interface (CLI)
- Explain how to deploy a Node application to IBM Cloud
- Describe how to view and manage the resources in your cloud account from the IBM Cloud dashboard



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-1. Unit objectives

Topics

- ▶ IBM Cloud architecture
 - Deploy a Node application to IBM Cloud
 - Manage IBM Cloud application environment settings



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-2. Topics

What is IBM Cloud?



- The IBM cloud platform combines platform as a service (PaaS) with infrastructure as a service (IaaS)
 - IBM Cloud Foundry is an open source platform as a service (PaaS) offering
 - The IBM Cloud Kubernetes Service uses open source as the foundation for advanced capabilities in security, scalability, and infrastructure management
- With Cloud, organizations and developers can quickly and easily create, deploy, and manage applications that are hosted on a Cloud platform

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-3. What is IBM Cloud?

The IBM cloud platform combines platform as a service (PaaS) with infrastructure as a service (IaaS).

Use powerful, open source technologies across runtimes, containers, and virtual machines to power your apps and services.

IBM Cloud is a platform that helps developers build and run modern apps and services. It provides developers with easy access to computing resources and services.

What kind of infrastructure can you build on IBM Cloud?



- **Cloud Foundry** technology provides a computing platform for your server runtime environments, services, and applications
 - IBM Cloud takes care of the management and maintenance the infrastructure that runs your applications



- **IBM Containers** provide more fine-grained control over the computing infrastructure that runs your application and services
 - You can use and extend public images from the IBM Cloud catalog or images from the public Docker hub



- **Virtual servers** are software implementation of hardware that runs applications like a computer
 - You can configure the operating system, server runtime environment, and application

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-4. What kind of infrastructure can you build on IBM Cloud?

The IBM Cloud service supports three infrastructure types. With the **Cloud Foundry** technology, IBM Cloud manages your application and services on your behalf. The Cloud Foundry infrastructure is the simplest way to get started with IBM Cloud.

If you want to port an image from another Cloud infrastructure provider, or if you want to use an existing public image, consider the **IBM Container** technology.

If you want to have control over the infrastructure down to the operating system level, consider **virtual machine**.

The exercises in this course introduce you to the **Cloud Foundry** infrastructure in IBM Cloud.

What is Cloud Foundry?

- The **Cloud Foundry open source project** is a platform as a service (PaaS) architecture for Cloud applications
 - With a PaaS architecture, your Cloud provider installs, manages, and hosts the operating system, application runtime server, and services for your application
- As an open source project, you can deploy your applications with any Cloud vendor that implements the Cloud Foundry platform

Figure 5-5. What is Cloud Foundry?

The Cloud Foundry open source project is a platform as a service (PaaS) architecture for Cloud applications.

With a PaaS architecture, your Cloud provider installs, manages, and hosts the operating system, application runtime server, and services for your application.

As an open source project, you can deploy your applications with any Cloud vendor that implements the Cloud Foundry platform.

Getting started: Create an IBM Cloud account

1. Open the IBM Cloud page at: <https://cloud.ibm.com/login>
2. Click **SIGN UP** to create an IBM Cloud account
 - If you do not have an IBM ID, complete the details in **register for your free account** to create one
 - You must register for an **IBM ID** before you create an IBM Cloud account
 - You can register for an IBM Cloud account for free

Figure 5-6. Getting started: Create an IBM Cloud account

Before you deploy an application to IBM Cloud, you must sign up for an IBM Cloud account. You can register for a free IBM Cloud account at cloud.ibm.com.

Topics

- IBM Cloud architecture
- ▶ Deploy a Node application to IBM Cloud
- Manage IBM Cloud application environment settings



Figure 5-7. Topics

What are the IBM Cloud Developer Tools?

- The IBM Cloud Developer Tools offer a command line approach to creating, developing, and deploying cloud applications
- Installation of the Cloud Developer Tools installs the latest stand-alone IBM Cloud CLI version available, plus the following tools:
 - IBM Cloud Developer Tools plug-in
 - IBM Cloud Foundry plug-in
 - IBM Cloud Container Registry plug-in
 - IBM Cloud Kubernetes Service plug-in
 - Docker, Git, Helm
- Review, control, and manage your Cloud environment from your terminal or command prompt window

Figure 5-8. What are the IBM Cloud Developer Tools?

The IBM Cloud Developer Tools offer a command line approach to creating, developing, and deploying cloud applications.

The installation command for the Cloud Developer Tools installs the latest stand-alone IBM Cloud CLI version available, plus the following tools:

IBM Cloud Developer Tools plug-in
IBM Cloud Foundry plug-in
IBM Cloud Container Registry plug-in
IBM Cloud Kubernetes Service plug-in
Docker, Git, Helm

Review, control, and manage your Cloud environment from your terminal or command prompt window/

The Cloud Foundry plug-in is a cross-platform

management and deployment utility for Cloud Foundry environments.

You can view and manage your user account, space, and organization settings.

You can deploy an application from your workstation to a Cloud Foundry environment.

Steps to deploy your application to IBM Cloud

1. Download and install the IBM Cloud Developer Tools command-line interface (CLI) for your operating system
2. Enable the application to make it Cloud-ready
3. Review the application files
4. Issue the `ibmcloud login` command to connect to your Cloud account
5. Issue the `ibmcloud dev deploy` command to publish and deploy your application to your Cloud account
6. Test the Cloud application

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-9. Steps to deploy your application to IBM Cloud

Steps to deploy your application to IBM Cloud.

1. Download and install the IBM Cloud Developer Tools command-line interface (CLI) for your operating system
2. Enable the application to make it Cloud-ready
3. Review the application files
4. Issue the `ibmcloud login` command to connect to your Cloud account
5. Issue the `ibmcloud dev deploy` command to publish and deploy your application to your Cloud account
6. Test the Cloud application

Step 1: Install the IBM Cloud Developer Tools CLI

- Run the `curl` command to install the Cloud Developer Tools CLI

```
$ curl -sL http://ibm.biz/idt-installer | bash
```

- Run the `ibmcloud dev --version` command to verify that you installed the application successfully

```
$ ibmcloud dev --version
ibmcloud dev version 2.2.0
```

- Review the CLI commands

```
$ ibmcloud dev help
NAME:
  ibmcloud dev - A CLI plugin to create, manage,
  and run applications on IBM Cloud
USAGE:
  ibmcloud dev command [arguments...] [command
  options]
```

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-10. Step 1: Install the IBM Cloud Developer Tools CLI

Step 1: Install the IBM Cloud Developer Tools CLI.

The `curl` command to install the Cloud Developer Tools for Mac OS and Linux is displayed on the page.

See the documentation at <https://cloud.ibm.com/docs/home/tools> for instructions for running the command for Windows 10.

After your installation completes, Run the `ibmcloud dev --version` command to verify that the application installed successfully.

The Cloud Developer Tools is a CLI plugin to create, manage, and run applications on IBM Cloud.

Type `ibmcloud dev help` to display the list of CLI commands.



Information

The installation of the IBM Cloud Developer Tools installs the `ibmcloud` command line tool for interacting with the IBM Cloud. Type `ibmcloud help` to see the list of commands.

Step 2: Enable the application to make it Cloud ready

- Run the command to enable an existing application in the current directory:

```
$ ibmcloud dev enable
```

- The enable command attempts to automatically detect the language of an existing application
 - The presence of a `package.json` file identifies a Node.js application
 - Can specify the language of the application with the `--language` option
- Use the `--no-create` parameter to prevent creating an application in IBM Cloud, and create the enablement files locally
 - Files are generated to be used for local Docker containers, Cloud Foundry deployment, or Kubernetes Container deployment
 - All deployment environments can be used through a manual `deploy` command
- The enable command creates a default `manifest.yml` file
 - Used to deploy the application as a Cloud Foundry application

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-11. Step 2: Enable the application to make it Cloud ready

Step 2: Enable the application to make it Cloud ready.

Run the `ibmcloud dev enable` command to enable an existing app in the current directory.

The enable command attempts to automatically detect the language of an existing application and then prompts for necessary additional information.

The presence of necessary files provides application language detection for a valid project structure. The presence of a `package.json` file identifies a Node.js app.

Use the `--no-create` parameter to prevent creating an application in IBM Cloud, and create the enablement files locally.

Files are generated to be used for local Docker containers, Cloud Foundry deployment, Cloud Foundry Enterprise Environment deployment, or Kubernetes Container deployment. All deployment environments can be used through a manual `deploy` command.

The enable command creates a default `manifest.yml` file that is used when you deploy the application to the Cloud as a Cloud Foundry application

Step 3: Review your application files

- Before you deploy your application to the Cloud Foundry platform, make sure that your application is Cloud-ready
- See “Considerations for Designing and Running an Application in the Cloud”:
 - <https://docs.cloudfoundry.org/devguide/deploy-apps/prepare-to-deploy.html>
- For Node.js applications, review your application settings
 - Review the generated `manifest.yml` file
 - Define a start script in the `package.json` file
 - Use the `process.env.PORT` as the application port for your application

```
$ var port = process.env.PORT || 3000
```
- See “Tips for Node.js applications”:
 - <https://docs.cloudfoundry.org/buildpacks/node/node-tips.html>

Figure 5-12. Step 3: Review your application files

Step 3: Review and update your application files.

Clients connect to applications running on Cloud Foundry in the Cloud by making requests to URLs associated with the application. Cloud Foundry allows HTTP requests to apps on ports 80 and 443. You must use the `PORT` environment variable to determine which port your app should listen on. To also run your app locally, set the default port as 3000.

Review the generated manifest file

- The application manifest file tells the Cloud Foundry CLI utility how to deploy an application
- Example settings:
 - The **name** of the application
 - The number of application **instances**
 - The **disk quota** allocation for the application
 - The **memory** allocation for the application
 - The web **route** to the application
- The `manifest.yml` file follows the YAML format:
 - The manifest file begins with three dashes (`---`)
 - The **applications** block begins with a heading followed by a colon (`:`)
 - The application **name** starts with a dash (`-`) and a space
 - Subsequent lines begin with two spaces to align with the name field

Figure 5-13. Review the generated manifest file

You should review the generated `manifest.yml` and include any custom deployment options.

The generated manifest file includes the name of the application and the number of instances that are started when the application is deployed. If the domain option is not specified, a default domain of `appdomain.cloud` is used when the application is deployed to the IBM Cloud.

Example: Cloud Foundry manifest file

- Manifest.yml file contents:

```
---
applications:
- instances: 1
  timeout: 180
  name: statusapp
  buildpack: sdk-for-nodejs
  command: npm start
  memory: 256M
  domain: not-used.net
  host: not-used
```

- If a route is not specified, the Cloud Foundry CLI utility uses default values to create the application route
 - For example, the web route is:
<https://status-app.us-south.cf.appdomain.cloud>

Figure 5-14. Example: Cloud Foundry manifest file

On the first time that you create a Cloud Foundry application, the deploy operation uses default values to create the web route. The Cloud Foundry system creates the route one time only. On the next deploy call to the same application, the utility uses the existing route.

In the example, the web route is a combination of the application name, region, Cloud Foundry short name, and domain name.

Step 4: Log in to your Cloud account

- Issue the `ibmcloud login` command to the Cloud domain

```
$ ibmcloud login -apikey @apiKey.json -r us-south
API endpoint: https://cloud.ibm.com
Region: us-south
Authenticating...
OK
...
Targeted region us-south

API endpoint: https://cloud.ibm.com
...
Tip: If you are managing Cloud Foundry applications and
services
- Use 'ibmcloud target --cf' to target Cloud Foundry org/space
interactively, or use 'ibmcloud target --cf-api ENDPOINT -o
ORG -s SPACE' to target the org/space
```

Figure 5-15. Step 4: Log in to your Cloud account

Step 4: Log in to your Cloud account.

In the example, the user signs on to the IBM Cloud account with an api key that is created and download from the IBM Cloud web interface.

In addition to being logged in, you must target an organization and a space.

Use 'ibmcloud target --cf' or 'ibmcloud target -o ORG -s SPACE'

In the second part, the user types the command:

`ibmcloud target --cf` to interactively target the Cloud Foundry organization and space.

Step 5: Deploy your application to your Cloud account

- Issue the `ibmcloud dev deploy` command to deploy your application to Cloud

```
$ ibmcloud dev deploy
...
Deploying to Cloud Foundry...

Executing ibmcloud cf push

Invoking 'cf push'...

Pushing from manifest to org kevinom_org / space dev as
kevinom@ca.ibm.com...
Using manifest file /home/localuser/projects/status-
app/manifest.yml
Getting app info...
Updating app with these attributes...
  name:                statusapp
  path:                /home/localuser/projects/status-app
  buildpacks:
    sdk-for-nodejs
```

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-16. Step 5: Deploy your application to your Cloud account

Step 5: Deploy your application to your Cloud account

The `ibmcloud dev deploy` command runs the `cf push` in the current directory to upload and deploy the files in the IBM Cloud as a Cloud Foundry application.

The deploy process uses the buildpack to build the application and deploy it on the IBM Cloud as a Node.js application.

Examine the deployment health and status

- After the server successfully deployed your application, it displays the memory, space, instances, and web address to your application

```

name:                statusapp
requested state:     started
routes:              status-app.us-south.cf.appdomain.cloud
last uploaded:      Fri 14 Jun 09:47:21 PDT 2019
stack:               cflinuxfs3
buildpacks:          sdk-for-nodejs
type:                web
instances:           1/1
memory usage:        256M
start command:       npm start
state               since                cpu    memory          disk
details
#0    running        2019-06-14T16:47:40Z    0.3%    66.2M of 256M    68.6M
of 1G
OK
Your app is hosted at http://status-app.us-
south.cf.appdomain.cloud/

```

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

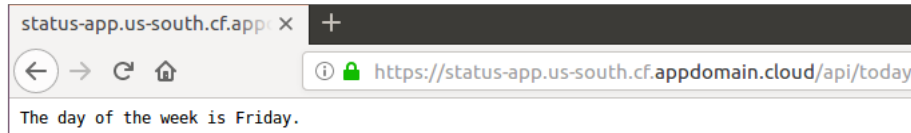
Figure 5-17. Examine the deployment health and status

Examine the deployment health and status.

After the server successfully deployed your application, it displays the details of the application status on the IBM Cloud. The console displays the memory, space, instances, and web address that is used to run the application.

Step 6: Test your Cloud application

- Test your application on the deployed host and route



- Review the application log with the `cf logs` command

```
$ ibmcloud cf logs statusapp --recent
Retrieving logs for app statusapp in org kevinom_org / space dev
as kevinom@ca.ibm.com...

2019-06-14T14:49:05.50-0700 [RTR/22] OUT status-app.us-
south.cf.appdomain.cloud - [2019-06-14T21:49:05.417+0000] "GET
/api/today HTTP/1.1" 200 0 30
```

Figure 5-18. Step 6: Test your Cloud application

Step 6: Test your Cloud application.

Test your application by specifying the application web route in a browser session.

The page displays the application that is running in the IBM Cloud environment.

You can view the logs by typing the command `ibmcloud cf logs statusapp --recent` from the CLI.

Topics

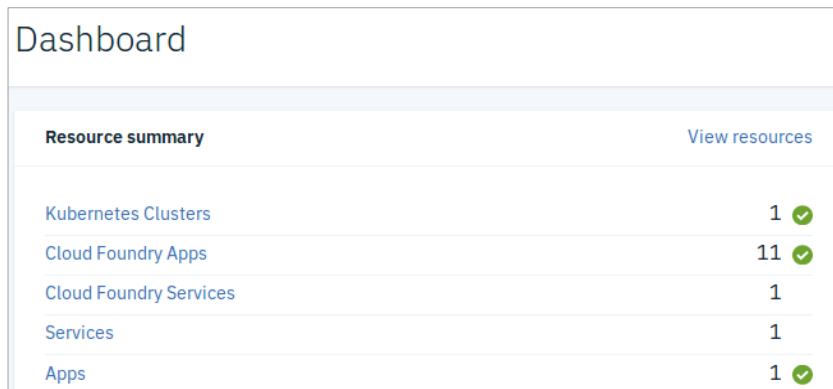
- ▶ Manage IBM Cloud application environment settings



Figure 5-19. Topics

IBM Cloud dashboard

- The IBM Cloud **dashboard** is a web application view of your account's health and status
 - Access the dashboard from the IBM Cloud site: <https://cloud.ibm.com>
 - Examine the **Cloud Foundry Apps** in your IBM Cloud account



The screenshot shows the IBM Cloud dashboard with a 'Resource summary' table. The table lists various resources and their counts, with a 'View resources' link at the top right. The resources listed are Kubernetes Clusters (1), Cloud Foundry Apps (11), Cloud Foundry Services (1), Services (1), and Apps (1). Each resource has a green checkmark icon next to its count.

Resource summary		View resources
Kubernetes Clusters	1	✓
Cloud Foundry Apps	11	✓
Cloud Foundry Services	1	
Services	1	
Apps	1	✓

Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-20. IBM Cloud dashboard

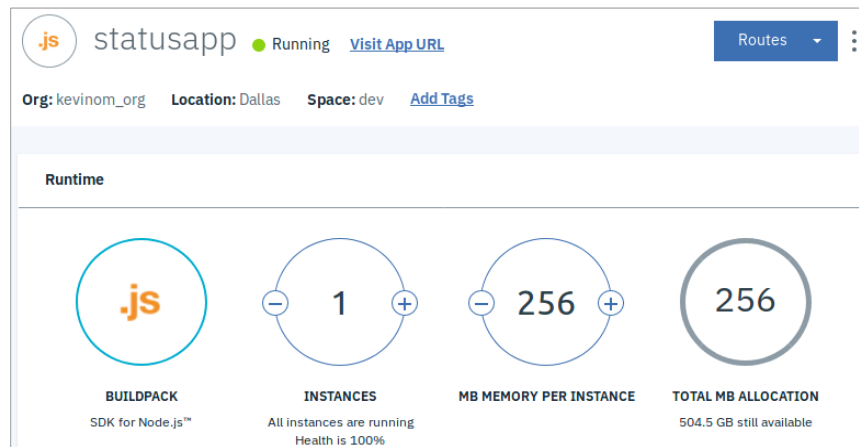
The IBM Cloud **dashboard** is a web application view of the resources in your account.

The dashboard is displayed by default when you sign on to your cloud account.

Click the Cloud Foundry Apps link to examine the Cloud Foundry Apps in your IBM Cloud account.

IBM Cloud dashboard: Application status

- Retrieve the health and status of your application
- Set the memory allocation for your application
- Create multiple instances of your application



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-21. IBM Cloud dashboard: Application status

When you click one of the applications in the list of applications, the details for the application are shown.

You see the status of the application and an option to visit the app URL to display the running application.

From the application status page, you can change the memory requirements and the number of running instances of the application.

IBM Cloud dashboard: Application logs

The screenshot shows the IBM Cloud dashboard for an application named 'statusapp'. The application is in a 'Running' state. The dashboard includes a sidebar with navigation options: Getting started, Overview, Runtime, Connections, Logs (selected), Autoscaling (Beta), Monitoring, and API Management. The main content area displays the application's status and logs. The logs table has columns for TYPE, INSTANCE, LOGS, and TIME. Two log entries are visible: one for an API instance 10, and another for an API instance 8. A yellow callout box highlights the 'Logs' tab and the log entries, with the text: 'Review the server logs for your application and the runtime environment'.

TYPE	INSTANCE	LOGS	TIME
API	10	Created app with guid 877d733d-96d2-4ca6-b066-10bb8cd90900	Jun 14, 2019 02:47:00.482 F
API	8	Uploading bits for app with guid 877d733d-96d2-4ca6-b066-10bb8cd90900	Jun 14, 2019 02:47:03.421 F

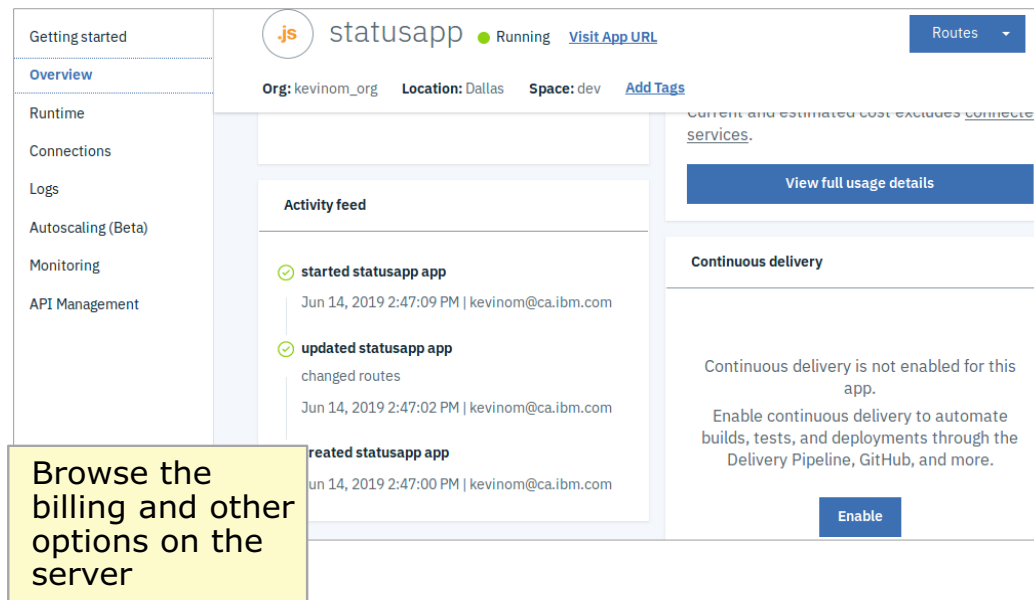
Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-22. IBM Cloud dashboard: Application logs

You can review the server logs for your application by clicking the Logs tab for the application.

IBM Cloud dashboard: Overview



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-23. IBM Cloud dashboard: Overview

The overview tab for the application gives you access to the billing and other options for the application.

Unit summary

- Describe the architecture of IBM Cloud
- Describe the features of the IBM Cloud Developer Tools
- Describe what is Cloud Foundry
- Describe how to install the IBM Cloud Developer Tools command-line interface (CLI)
- Explain how to deploy a Node application to IBM Cloud
- Describe how to view and manage the resources in your cloud account from the IBM Cloud dashboard



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-24. Unit summary

Review questions



1. True or False: The `ibmcloud dev enable` command attempts to automatically detect the language of an existing application based on files in the current directory.
2. True or False: The manifest file is in the JSON format.

Figure 5-25. Review questions

Write your answers here:

- 1.
- 2.

Review answers



1. True or False: The `ibmcloud dev enable` command attempts to automatically detect the language of an existing application based on files in the current directory.
The answer is True
2. True or False: The manifest file is in the JSON format.
The answer is False. The Cloud Foundry application manifest file is in the YAML format.

Figure 5-26. Review answers

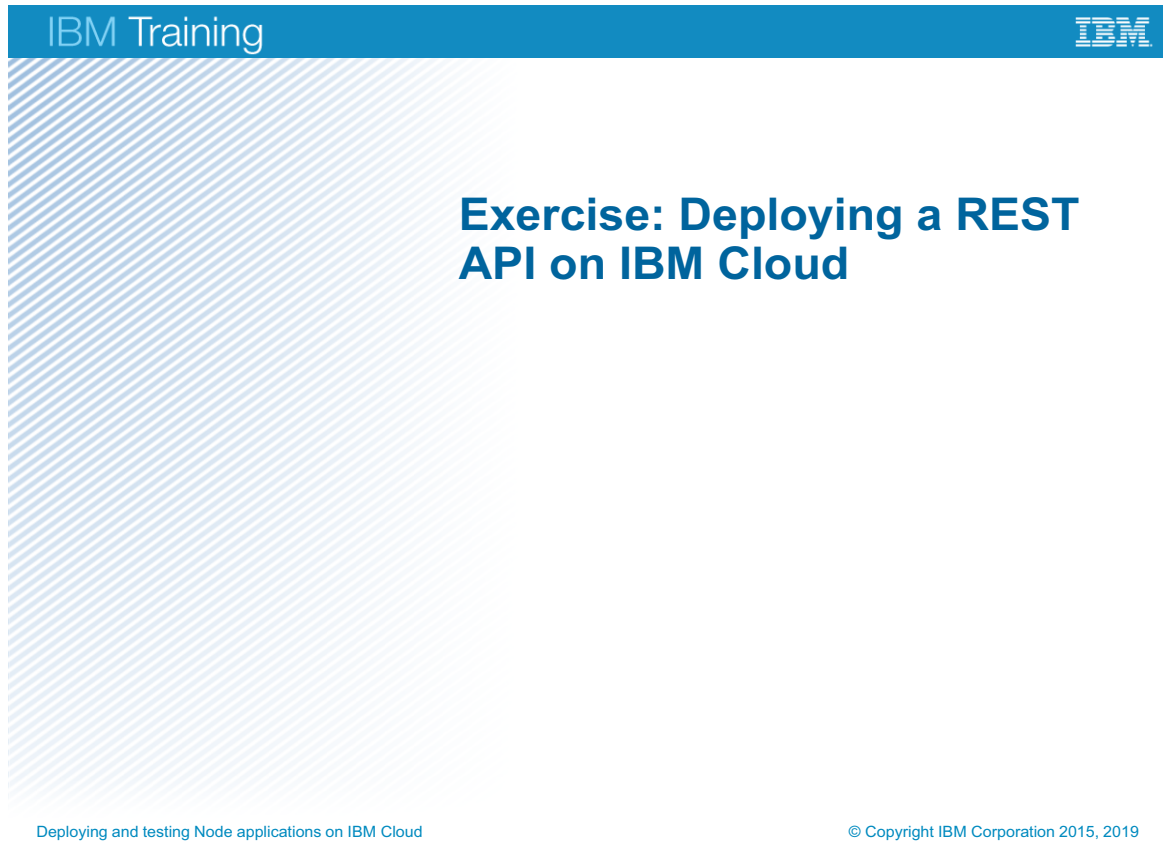


Figure 5-27. Exercise: Deploying a REST API on IBM Cloud

Exercise objectives



- Register for an IBM Cloud account
- Install the IBM Cloud command-line interface (CLI)
- Enable an existing application for IBM Cloud
- Build and run an application on a local container that is built with the IBM Cloud Developer Tools
- Deploy a node application into IBM Cloud
- Retrieve the application logs from an IBM Cloud application



Deploying and testing Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 5-28. Exercise objectives

Unit 6. Course summary

Estimated time

00:10

Overview

This unit summarizes the course and provides information for future study.

Unit objectives

- Explain how the course met its learning objectives
- Access the IBM Training website
- Identify other IBM Training courses that are related to this topic
- Locate appropriate resources for further study

© Copyright IBM Corporation 2015, 2019

Figure 6-1. Unit objectives

Course learning objectives

After completing this course, you should be able to:

- Install, validate, and test the Node runtime environment on your local workstation

- Install Node packages with npm

- Develop REST API operations with Express

- Develop callback functions to handle asynchronous events

- Perform static code analysis of the application with ESLint

- Run Mocha and Supertest unit tests on Node applications

- Debug Node applications with the Google Chrome browser with Node inspector

© Copyright IBM Corporation 2015, 2019

Figure 6-2. Course learning objectives

Course learning objectives

After completing this course, you should be able to:

- Package Node applications

- Deploy Node applications to IBM Cloud with the IBM Cloud command-line utility

- Run Node applications on IBM Cloud

© Copyright IBM Corporation 2015, 2019

Figure 6-3. Course learning objectives

Other learning resources (1 of 4)

- **IBM Skills Gateway**
 - Search the new IBM Training & Skills website (formerly IBM Authorized Training website) to find and access the content you want.
 - <https://www-03.ibm.com/services/learning/ites.wss/zz-en?pageType=page&c=a0011023>
- **IBM Cloud Education Wiki Home**
 - Go to the wiki to find course abstracts, course correction documents, and curriculum development plans for IBM Cloud offerings.
 - <https://www.ibm.com/developerworks>
- **Role-based Learning Journeys**
 - Learning Journeys describe the appropriate courses, in the recommended order, for specific products and roles.
 - <https://www-03.ibm.com/services/learning/ites.wss/zz/en?pageType=page&c=a0003096>

© Copyright IBM Corporation 2015, 2019

Figure 6-4. Other learning resources (1 of 4)

Other learning resources (2 of 4)

- **IBM Professional Certification Program**
 - IBM Professional Certification enables skilled IT professionals to demonstrate their expertise to the world. It validates skills and proficiency in the latest IBM technology and solutions.
 - <https://www.ibm.com/certify>
- **IBM Training blog, Twitter, and Facebook**
 - These official IBM Training and Skills accounts provide information about IBM course offerings, industry information, conference events, and other education-related topics.
 - <https://www.ibm.com/blogs/ibm-training>
 - <https://twitter.com/IBMTraining>
 - <https://www.facebook.com/ibmtraining>
- Node.js website:
<https://nodejs.org>

© Copyright IBM Corporation 2015, 2019

Figure 6-5. Other learning resources (2 of 4)

Other learning resources (3 of 4)

- **Business Partner Technical Enablement Portal**
 - <https://ibm.box.com/s/695khv9nyzekaorykqmsjrematz3v9xh>
 - This program provides technical training content modules to IBM software partners (via PartnerWorld) and IBM Business Partners.
- **IBM Developer**
 - IBM's official developer program offers access to software trials and downloads, how-to information, and expert practitioners.
 - <https://developer.ibm.com>
- **IBM Education Assistant**
 - These multimedia educational modules help users gain a better understanding of IBM Software products and use them more effectively to meet business requirements.
 - <https://www.ibm.com/products/software>

© Copyright IBM Corporation 2015, 2019

Figure 6-6. Other learning resources (3 of 4)

Other learning resources (4 of 4)

- **IBM Knowledge Center**

- The IBM Knowledge Center is the primary home for IBM product documentation.
- <https://www.ibm.com/support/knowledgecenter>

- **IBM Marketplace**

- IBM Marketplace is the landing page for all IBM Cloud products. Go to the Marketplace to learn about IBM offerings for Cloud, Cognitive, Data and Analytics, Mobile, Security, IT Infrastructure, and Enterprise and Business Solutions.
- <https://www.ibm.com/products>

- **IBM Redbooks**

- IBM Redbooks are developed and published by the IBM International Technical Support Organization (ITSO). Redbooks typically provide positioning and value guidance, installation and implementation experiences, typical solution scenarios, and step-by-step "how-to" guidelines.
- <http://www.redbooks.ibm.com>

© Copyright IBM Corporation 2015, 2019

Figure 6-7. Other learning resources (4 of 4)

Appendix A. List of abbreviations

API	application programming interface
BDD	behavior-driven development
CF	Cloud Foundry
CLI	command-line interface
CPU	central processing unit
DB	database
GB	gigabyte
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
ICAO	International Civil Aviation Organization
IDE	integrated development environment
I/O	input/output
JSON	JavaScript Object Notation
LTS	Long Term Support
MB	megabyte
OS	operating system
PaaS	platform as a service
RAM	random access memory
REPL	Read, Evaluate, Print, and Loop
REST	REpresentational State Transfer
SDK	software development kit
TDD	test-driven development
UNIX	Uniplexed Information and Computing System
URI	Universal Resource Identifier
VNC	virtual network computing
YAML	yet another markup language



IBM Training

