Spring Boot with React and AWS

Learn to Deploy a Full Stack Spring Boot React Application to AWS

—

Ravi Kant Soni
Namrata Soni

Apress®

Spring Boot with React and AWS

Learn to Deploy a Full Stack Spring Boot React Application to AWS

Ravi Kant Soni
Namrata Soni

Apress®

Spring Boot with React and AWS: Learn to Deploy a Full Stack Spring Boot React Application to AWS

Ravi Kant Soni
s/o – Late. Ras Bihari Prasad, Sri Niwash,
Lashkariganj, Sasaram, Bihar, India

Namrata Soni
d/o – Late. Ras Bihari Prasad, Sri Niwash,
Lashkariganj, Sasaram, Bihar, India

ISBN-13 (pbk): 978-1-4842-7391-3
ISBN-13 (electronic): 978-1-4842-7392-0
https://doi.org/10.1007/978-1-4842-7392-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484273913. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my beloved father,
the late Ras Bihari Prasad
We miss you and love you, Papa. A strong and gentle soul who taught us to trust God and to believe in ourselves and our dreams.

To my beloved mother,
Smt. Manorma Devi
We love you, Maa. We could never have completed this book without your true love, warmest support, and constant encouragement.

Table of Contents

About the Authors

Ravi Kant Soni is a principal full stack engineer with more than 11 years of IT experience. He is also an AWS Certified Solutions Architect. Ravi has experience in software development, software design, systems architecture, application programming, and automation testing. He has a bachelor's degree in Information Science and Engineering from Reva University, Bangalore; and schooling from Bal Vikash Vidyalaya, Sasaram, and Bihar (India). He is the author of Build Microservices with Spring Cloud and Spring Boot (codered eccouncil, 2021), *Full Stack AngularJS for Java Developers* (Apress, 2018), *Spring: Developing Java Applications for the Enterprise* (Packt, 2017), and *Learning Spring Application Development* (Packt, 2015). He is also an esteemed member of the Board of Studies at the REVA University School of Computing and Information Technology in Bangalore. Contact Ravi at `www.linkedin.com/in/november03ravikantsoni/`.

Namrata Soni is a self-taught web application developer with a passion for beautiful and interactive UIs. She has a degree in computer science from Sagar Institute of Science & Technology, Bhopal; and schooling from Bal Vikash Vidyalaya, Sasaram, and Bihar (India). She loves clean and well-tested code, is a big fan of open source, and enjoys learning something new. Currently, she is working with React and Node.js to craft modern JavaScript applications. Contact Namrata at `www.linkedin.com/in/september-6-namrata-soni/`.

About the Technical Reviewer

Karunesh Chandra Tiwari is an IT professional with ten years of experience and has worked across distinct technologies and domains. He is a technologist and speaker and loves to provide his views on articles and blogs.

Karunesh is a BTech IT graduate from Anna University. He worked as a full stack developer for the first half of his career and currently works with BPM and CRM tools and cloud-related technologies, including developing and working with applications for some of the world's leading banks. He is a very focused and determined person and loves to learn, work in new technologies. He loves to mentor people both from a professional and a personal perspective.

Karunesh enjoys working with new technologies and loves to mentor people. Check out his LinkedIn profile at www.linkedin.com/in/karunesh-chandra-tiwari-20b9a82a/.

Acknowledgments

Writing a technical book involves fathomless research, review, and support. I wrote this book, but it wouldn't have been possible without the love and support of many people. I truly want to thank everyone listed here, from the deep bottom of my heart!

First and foremost, I need to express gratitude toward Michael Gorriz, Group Chief Information Officer, Standard Chartered Bank, for inspiring me and giving me the confidence to write this book when I anticipated my career break. All I can offer in return is a heartfelt thank you!

I want to thank my colleagues at Standard Chartered Bank. I learn something new every day and enjoy a camaraderie I've never felt in any company before. I am fortunate enough to work with such an experienced team that helped me enhance my skills. My gratitude goes to Anshu Sharma Raja, CIO, Consumer Private Business Banking at Standard Chartered Bank, and Dr. Ashish Chandra, Location Head- aXess Labs (Banking Innovation) at Standard Chartered Bank; for their guidance and strong support.

I want to thank the Apress publishing team for the utmost professionalism. The one individual who has been the roof of this shelter is Divya Modi, coordinating editor, for supporting me in the writing of this book. Also, I would like to express my special gratitude to James Markham, development editor, whose vision, commitment, and persistent efforts made publishing this book efficient.

My heartfelt thanks go to the technical reviewer, Karunesh Chandra Tiwari, for his valuable input.

My deepest gratitude and appreciation go to my dear friend **Awanish Kumar, IAS – Deputy Commissioner, Delhi;** for the intellectual stimulus from time to time, which helped me approach the book from a unique perspective.

Thanks to my dearest friend, **Dr. Meena Soni (Incharge Medical Officer, Surajpur - Basdei, and Chhattisgarh)**, for invariably motivating, encouraging, and giving me positive thoughts that worked as fuel to carry on.

Without my families' love, support, and understanding, this book would have remained a virtual commodity. My profound thanks to my beloved mother, Smt. Manorma Devi, for her love and support, which encourages my knowledge to come out on paper to ignite the imagination of others.

My special thanks go to a man who has been a rock of stability throughout my life and whose loving spirit sustains me still—my uncle Shri. Arun Kumar Soni for the great inspiration he has given me to achieve all success in life. Thanks also to my brothers, Shashi Kant and Shree Kant, and all my family members who have loved me.

My special thanks to my co-author and sister, Namrata Soni, for agreeing to co-author this book and helping me write Chapter 5, which discusses React and AWS S3. I'm still amazed that she agreed to get involved with this book, considering how enormously busy she is. Namrata, thank you!

I want to thank the goddess Maa Tara Chandi, Sasaram, Bihar, India; for giving me to such an extent.

Finally, this book is based on the innovative work of many people in our industry who have become my idols. I am thankful to everyone who supported me in one way or another in writing this book.

Welcome to *Spring Boot with React and AWS*.

—Ravi Kant Soni

CHAPTER 1

An Introduction to Amazon Web Services (AWS)

When you hear the word *amazon*, you likely first think of Amazon.com, which is one of the biggest and most successful online stores. While Amazon built its brand on developing online retail services, it has also branched out into alternative industries, among them the web services industry, where they have the eponymous Amazon Web Service (AWS), a form of cloud-computing that assists users develop software, database, and other programs that need heavy-duty computing resources.

This chapter overviews Amazon Web Services (AWS), including several AWS key services, such as Amazon Elastic Compute Cloud (Amazon EC2), AWS Elastic Beanstalk, Amazon Relational Database Service (Amazon RDS), and Amazon Route 53. It covers creating a free AWS account for developers, creating an Elastic Beanstalk server, creating a HelloWorld JSP application, building a WAR file with Maven, and uploading it to Elastic Beanstalk.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0_1

Introduction to Amazon Web Services

What precisely is Amazon Web Services (AWS)? At a really high level, AWS is a web-hosting service offered by Amazon, where you can deploy your web applications and conjointly deploy your databases. Once it's deployed, your apps are out there online. Anyone can simply enter your URL (Universal Resource Locator) in their web browser to access your application. The web connects everybody. You can deploy your application online within the cloud, so that anyone can access it. It's not only running locally; it's now running online.

AWS is a full-service cloud platform. It is more than just an application hosting platform. There are plenty of belongings you do with AWS.

- On-demand delivery of IT resources via the web
    - You can spin up servers on-demand, and you can choose your operating system.
    - You can even deploy databases within the cloud, and you get more options for the database as you wish.
- Pay-as-you-go pricing model
    - This book uses free developer accounts. You can get a free developer account for 12 months.

And, the nice thing about using the Amazon Web Services cloud is that you can be global within minutes because Amazon has worldwide data centers, as shown in Figure 1-1.

Figure 1-1. *Amazon data center*

You'll be able to deploy your application to a single data center; otherwise, you'll deploy it to multiple data centers. Also, there are no restrictions on which data center you'll be able to deploy to.

If you're based mostly within the United States, however, you can deploy applications to the data center in South America, China, or the Asia Pacific. It's completely up to you. The user can select the regions based on the application usage so that latency is low. There's no restriction as such on it.

Once you're logged in to the Amazon console, then essentially, you choose the services that you simply wish to use. You need to only deploy your applications to have a pleasant web admin console where you only configure your environment, configure your servers, then reasonably push-button deploy. Figure 1-2 shows the AWS services on AWS Management Console.

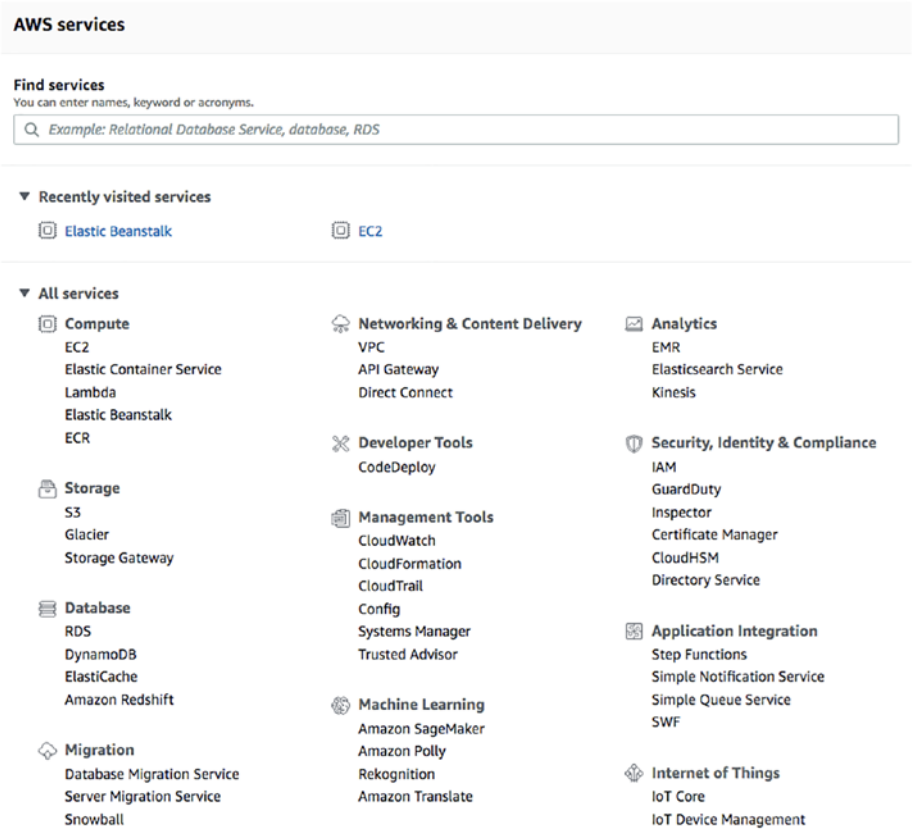


Figure 1-2. *AWS*

This was all about an introduction to Amazon Web Services. Let's dig into some of the AWS key Services.

AWS Key Services

AWS offers a wide range of services underneath different categories. This section explores several AWS key services (see Figure 1-3). First, let's look at Amazon Elastic Compute Cloud (Amazon EC2), which may include remote VMs (virtual machines). Next, you briefly look at AWS Elastic

Beanstalk, which allows developers to deploy web applications. Then, you move on to the Amazon Relational Database Service (Amazon RDS), which is a database within the cloud. Finally, you look at Amazon Route 53, which routes custom domain names to your application.

Figure 1-3. *AWS key services*

Elastic Cloud Compute (EC2)

Elastic Cloud Compute (EC2) is one of the first web service interfaces when AWS was released, allowing users to create and configure compute machines within the cloud. EC2 allows users to create VM (virtual machine) on the Amazon cloud for running applications that can be accessed via the Internet.

The software can be configured on cloud servers based on your specifications. You select the operating system (i.e., Microsoft Windows or Linux) best suited to your requirements or applications, and you get the operating system pre-installed. EC2 provides the actual host server and operating system. Figure 1-4 shows how it is set up.

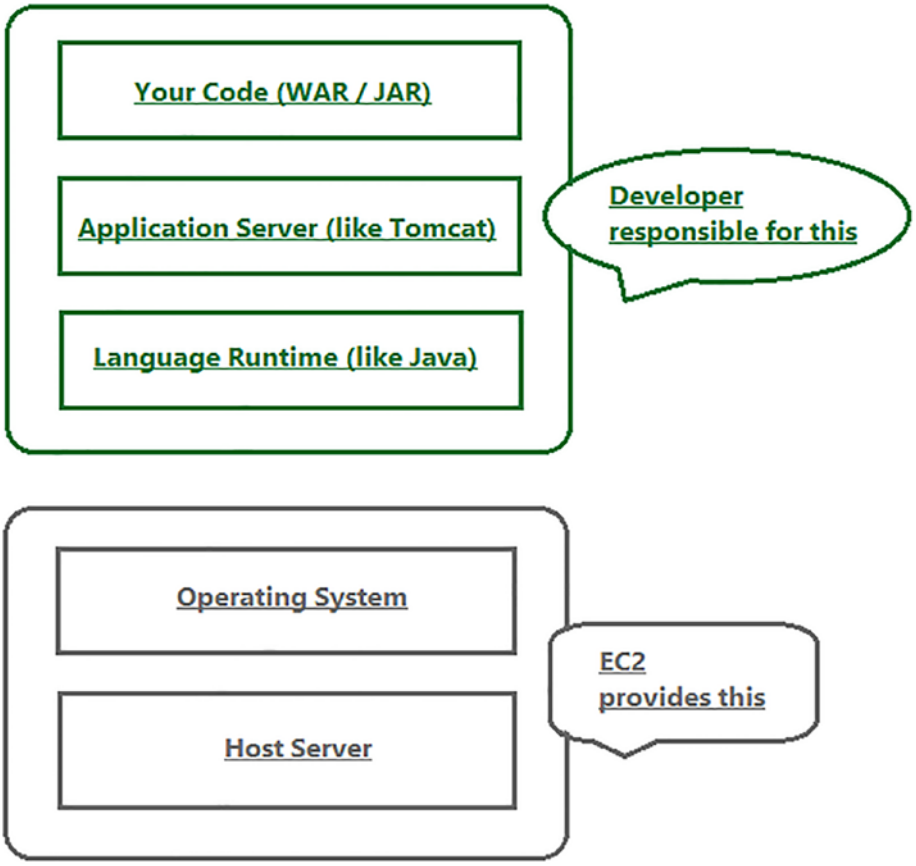


Figure 1-4. *How EC2 is set up*

If you want any additional software, you must manually install it on top of the OS as a developer. So, if you want JDK (Java Development Kit), you can install Java. You can also install Tomcat, a database, and so on. It's almost like getting a brand-new laptop that only has the operating system, and you need to install your tools on top of it.

Elastic Beanstalk

Elastic Beanstalk is a pre-packaged platform, allowing you to quickly deploy and handle your web applications without worrying about the infrastructure. You select a pre-configured virtual machine for your given web stack, like Java and Tomcat. And, there is no need to install any additional software's on the virtual machine. You simply upload the application's deployable file, and then you are out there and ready to go. Elastic Beanstalk automatically provides the application server, language runtime, operating server, and the host server, as shown in Figure 1-5.

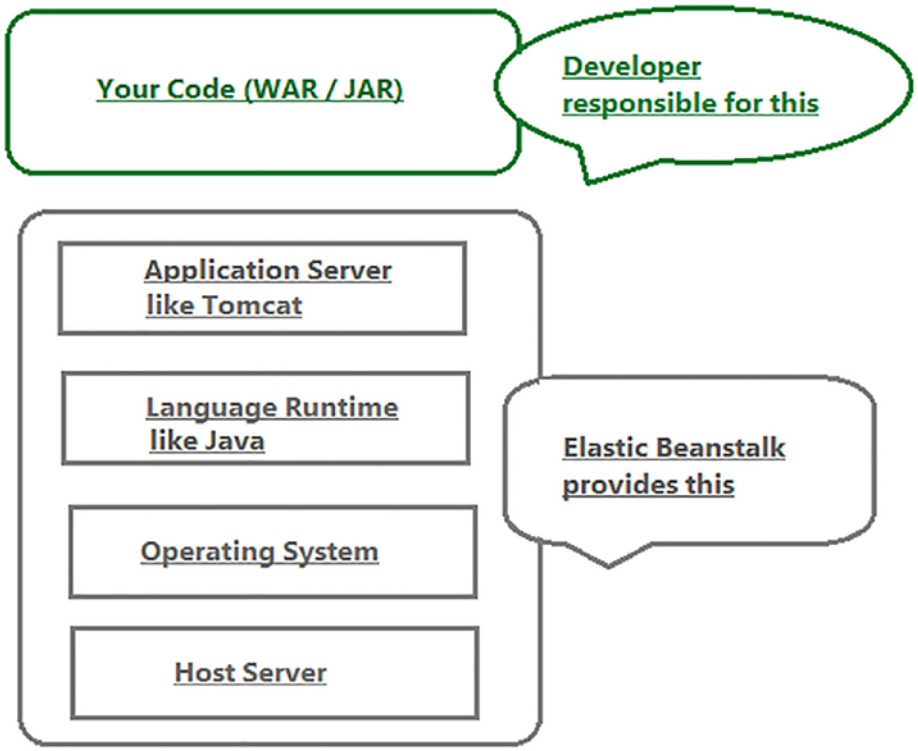


Figure 1-5. *Elastic Beanstalk*

It also has support for .NET, Node.js, PHP, Docker, and so on. You can choose the web stack that gives you all the software's pre-installed, pre-configured, and you simply deploy your code.

It's great for deployment on a web stack, you simply select the services that you need, and it is set up for you. This is known as *platform as a service*, or PaaS. All you have to do is deploy your code.

Now, when you develop Java applications on AWS, you can use your regular Java EE APIs. You can also use third-party frameworks like Spring Boot, Hibernate, and anything in standard Java. Whatever you can run on Tomcat locally, you can run that same code on Amazon. So, there are no code changes you need to make and no special Amazon JAR files or anything.

Relational Database Service (RDS)

AWS Relational Database Service (RDS) is your relational database in the cloud. This allows you to quickly deploy a relational database in the cloud. It has support for a wide range of databases to choose from, including MySQL, Oracle, Microsoft SQL Server, and so on.

You can manage these tools using your normal admin tools. If you are using MySQL, you can use MySQL Workbench. If you are using the Oracle Database, you can use Oracle SQL Developer, and the list goes on.

AWS also has support for NoSQL databases such as MongoDB. So, all major database feature's that you need can be found in AWS with the support of the relational Database Service.

Route 53

Amazon Route 53 is a Domain Name System (DNS), which allows you to route your custom domain names to your actual application on AWS. So, you configure Route 53 to send requests from the browser to your AWS application. The AWS DNS sets up your custom domain name.

Use Case: AWS Application Architecture

For your apps, start with AWS Elastic Beanstalk because you can quickly get started with deploying your application by leveraging those pre-configured web stacks out of the box.

Use EC2 if you need some low-level control. For example, you may want to use a version of Java that Elastic Beanstalk does not support, or you may want to use a Java application server like WebLogic or make another OS-specific customization.

Figure 1-6 shows that the architecture uses Elastic Beanstalk to deploy the web application. The Java application runs on Tomcat. RDS is the database in the cloud using MySQL. Route 53 routes your custom domain name to your application hosted on AWS.

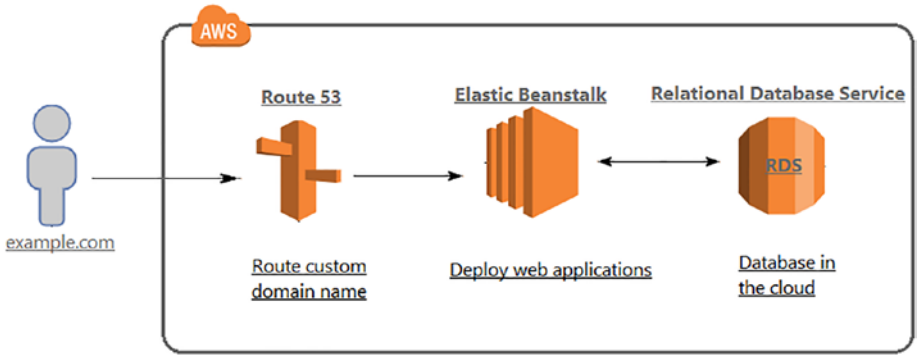


Figure 1-6. *AWS application architecture*

Create a Free AWS Account for Developer

To access Amazon Web Services, you need to create an AWS account. First, let's talk about the AWS free tier, where developers get a free 12 months trial period and enough resources to deploy your application and database for free. There is also a smaller version of AWS servers that you can use for free.

If you need to have some more advanced features, then you must pay and get access. This book uses the free tier. If you would like more information on the free tier, go to `https://aws.amazon.com/free/`.

In your web browser, go to `https://aws.amazon.com` to open the Amazon Web Services home page (see Figure 1-7; this screenshot may be different on your screen due to any updates by Amazon).

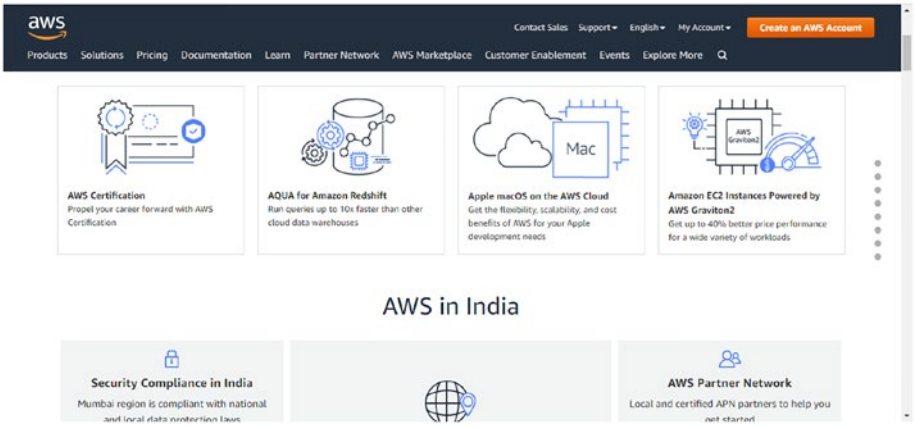


Figure 1-7. *AWS main page*

To create an AWS account, you need to provide your contact information, including your address, and a valid debit or credit card. Even though you are using a free account, Amazon needs your credit or debit card information. So, have it handy when creating your AWS account.

On the top right of the main page, click the **Create an AWS Account** button. You are redirected to the sign up for the AWS page, as shown in Figure 1-8.

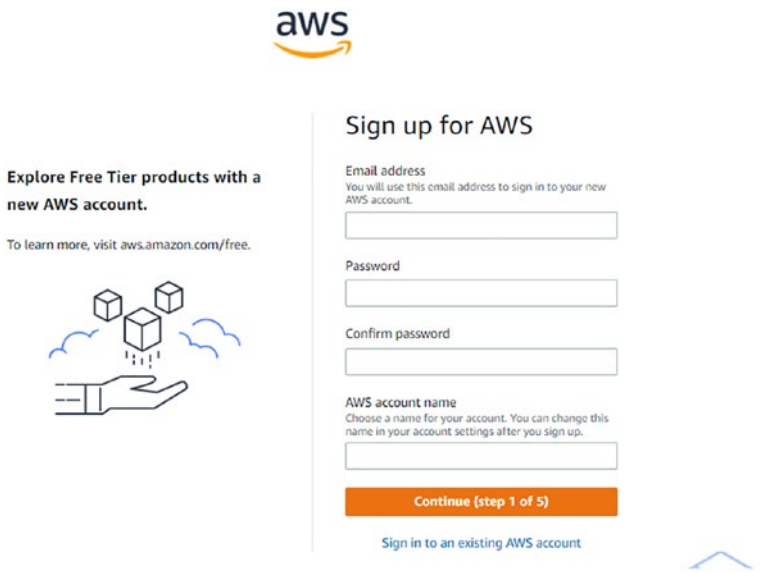


Figure 1-8. *Sign up for AWS*

Enter your email address, password (choose a strong password to prevent getting hacked), and the AWS account name that you want for this account. You must be sure that the account information you enter is correct, especially your email address. If you enter an incorrect email address, you can't access your account.

Click the Continue button to enter your contact information, as shown in Figure 1-9.

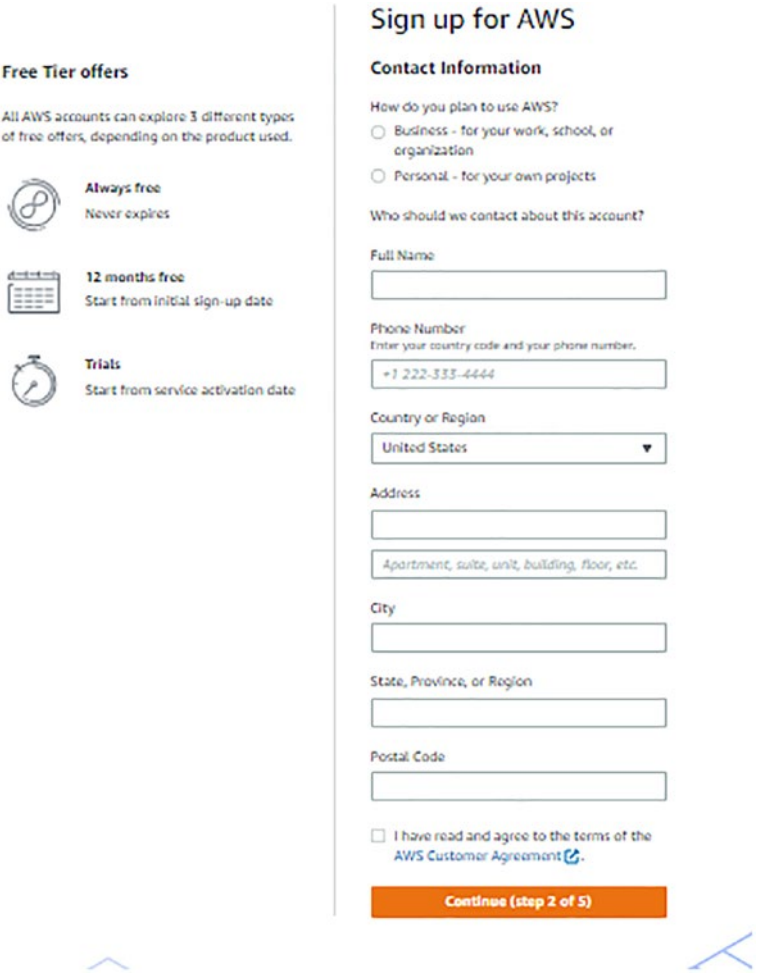


Figure 1-9. *Contact information*

First, select the Personal account type. (A business account is associated with an organization, and a personal account is associated with an individual.) Enter your full name, phone number, country, address, city, state, and postal code.

Finally, select the little check box at the bottom to show that you have read and agree to the terms of the AWS Customer Agreement, and then click the Continue button.

You receive an email from AWS to confirm that your AWS account has been created. You can sign in to your new account using the email address and password you registered with. However, you can't use AWS services until you finish your account activation.

Billing information is where you must enter your credit or debit card number and so forth, as shown in Figure 1-10. It is used for verification purposes.

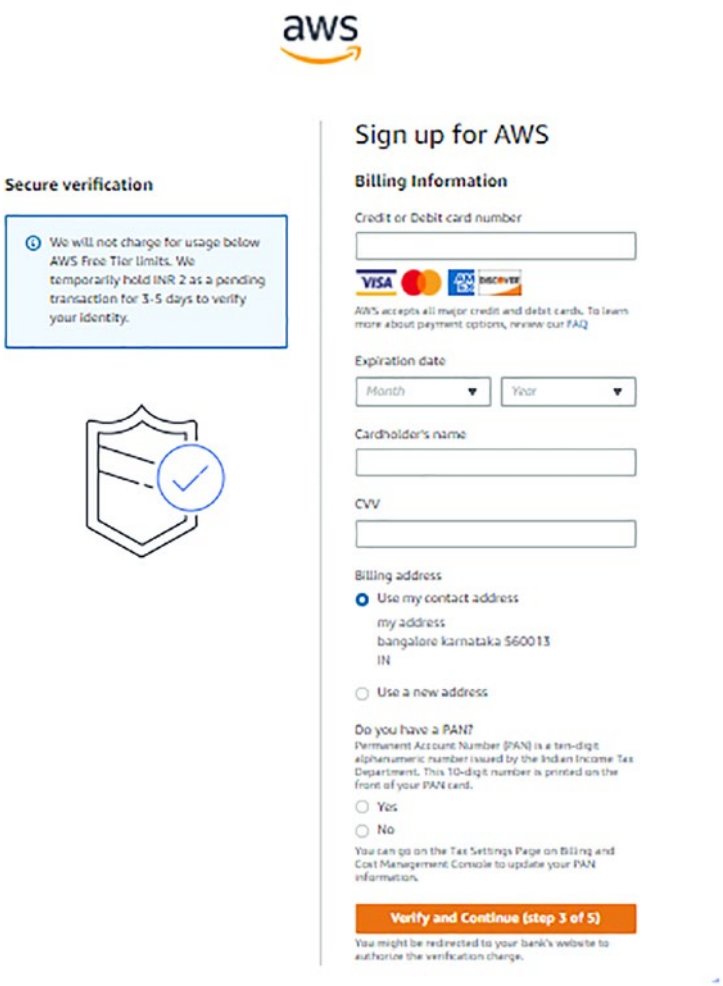


Figure 1-10. *Billing information*

Amazon does not charge your card unless your usage exceeds AWS Free Tier limits. In this book, everything that we show you is within the Free Tier limits.

AWS requires phone number verification, as shown in Figure 1-11. Choose your country or region code from the list, enter a phone number where you can be immediately reached, and enter the characters displayed in Security Check.

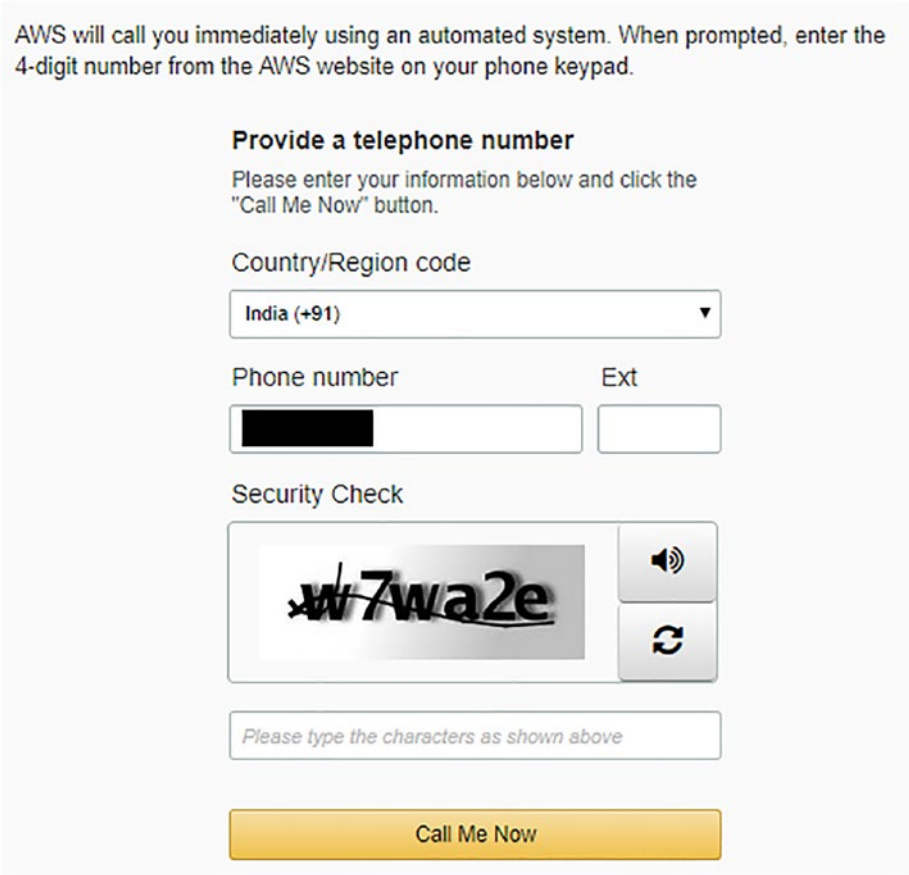


Figure 1-11. *Phone number verification*

Once you type the security check characters, click the Call Me Now button. A verification code is displayed on the screen, and at the same time, you get a call from Amazon to verify your registered phone number. You must enter the PIN you received and choose to continue. Once your identity has been successfully verified, you can see on the window that your phone is verified, and you are redirected to the next screen to choose your support plan, as shown in Figure 1-12.

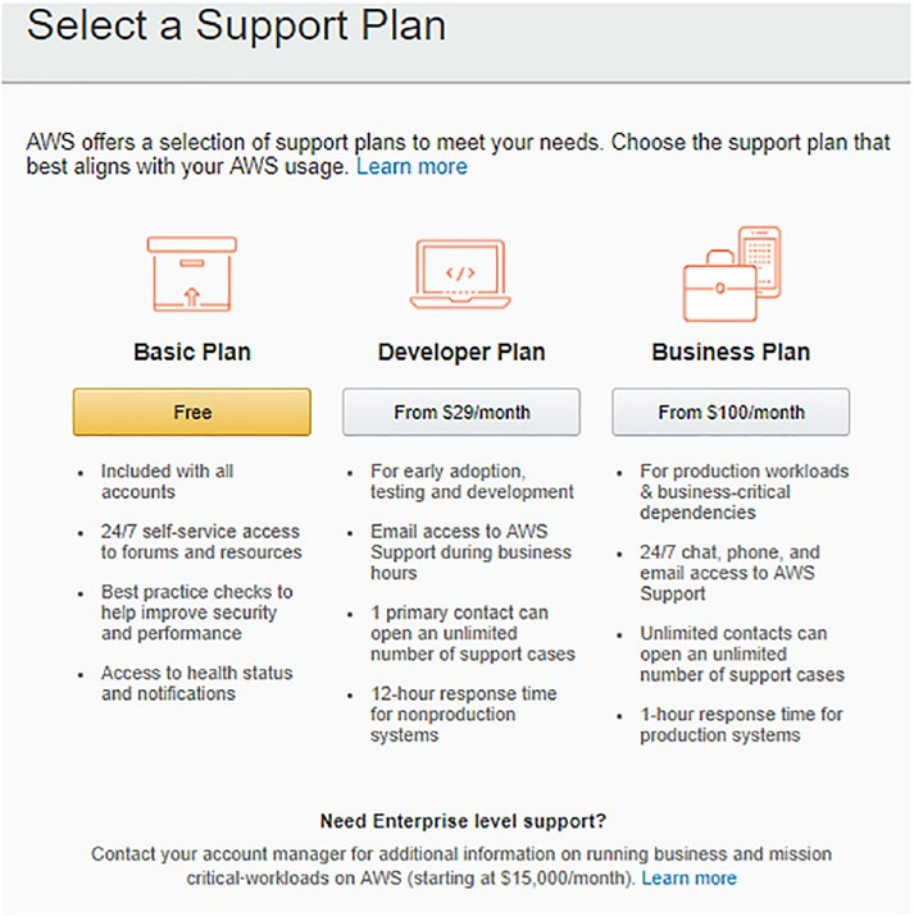


Figure 1-12. *Support plan*

Choose the support plan that meets your needs. Select the Basic Plan for free support. Click the Free button, and you are redirected to the AWS Registration Confirmation page.

Now you can *sign in to the AWS Management Console*. Go to `https://console.aws.amazon.com` to start using AWS.

Select **Root user**, enter your AWS account email address, and click the Next button, as shown in Figure 1-13.

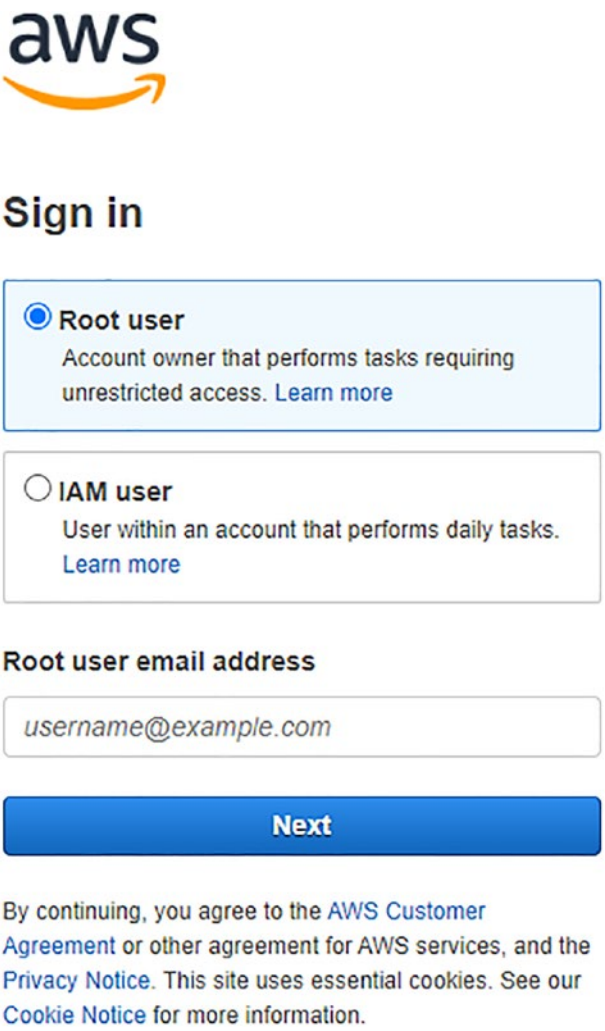


Figure 1-13. *Sign in to the console*

Next, enter your AWS account password, and click **Sign in**, as shown in Figure 1-14.

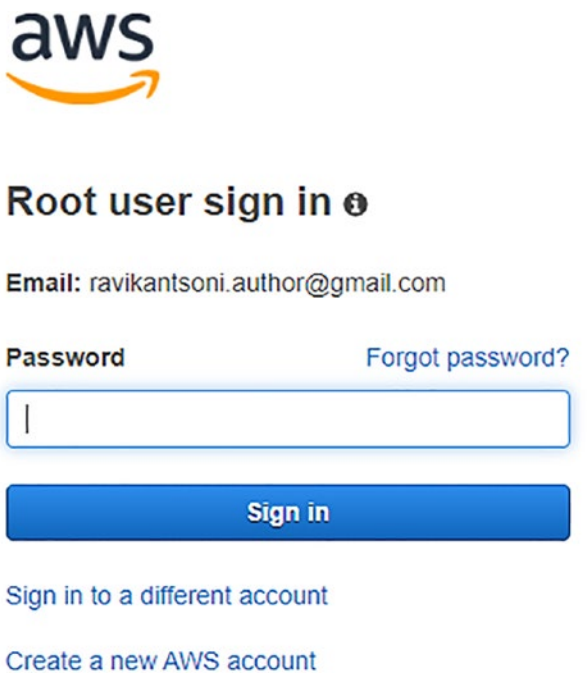


Figure 1-14. *Sign-in password*

The AWS Management Console is shown in Figure 1-15.

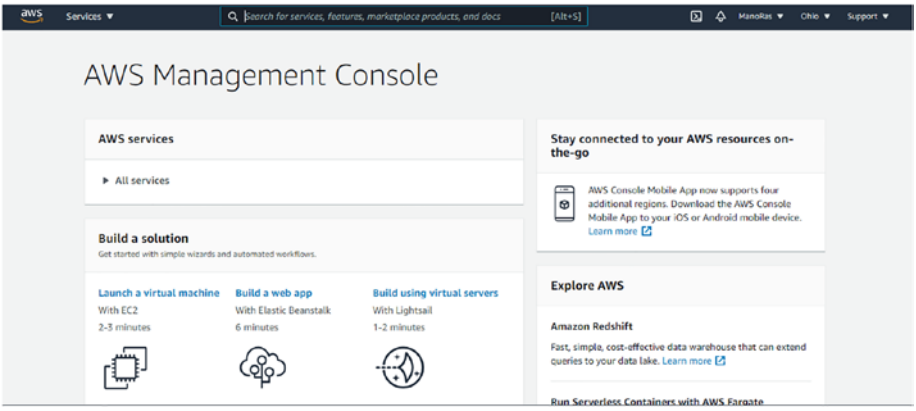


Figure 1-15. *AWS Management Console*

This is where you can find all the different services that are available and provided, but they are grouped by category, as shown in Figure 1-16.

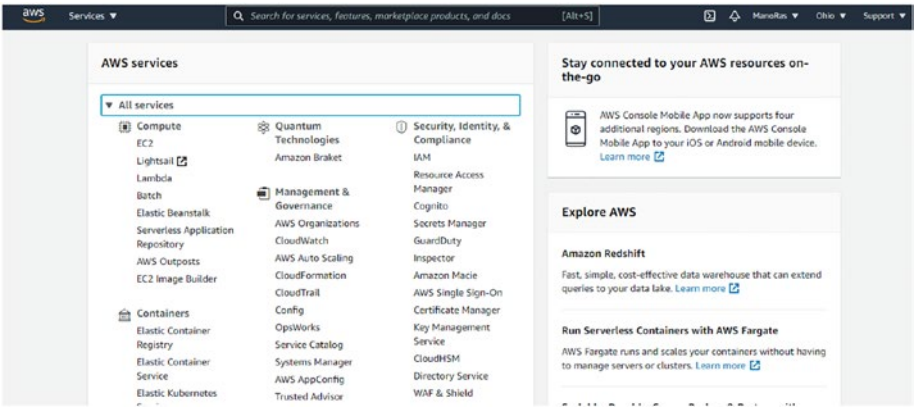


Figure 1-16. *AWS services by category*

The next section uses the Elastic Beanstalk service to begin building a web application. Tomcat is running in the AWS cloud.

Explore and Create an AWS Elastic Beanstalk Server

On the **AWS services** page, scroll down to the Compute section and select Elastic Beanstalk, as shown in Figure 1-17. It allows you to run and manage your web application.

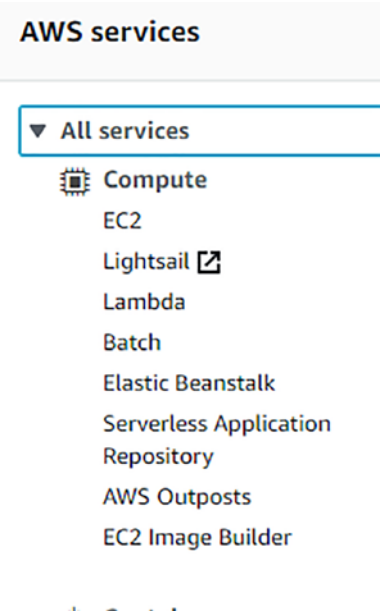


Figure 1-17. *Elastic Beanstalk under Compute section*

The AWS Elastic Beanstalk page is shown in Figure 1-18.

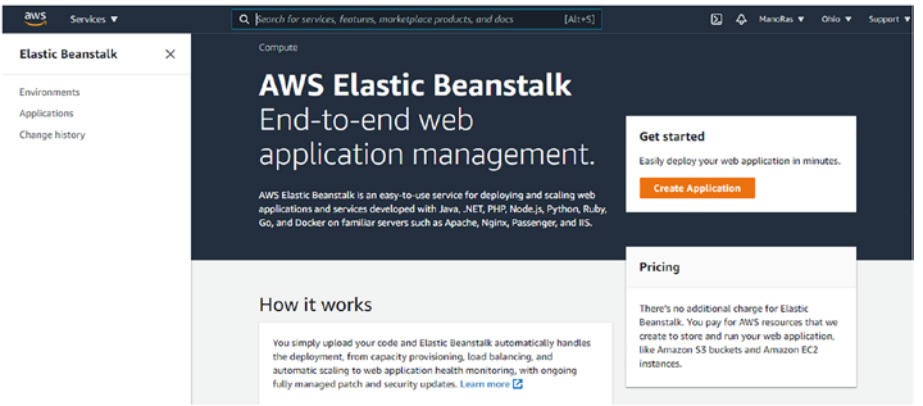


Figure 1-18. *AWS Elastic Beanstalk*

Elastic Beanstalk is the simplest way to deploy and run your web application on AWS. Elastic Beanstalk automatically handles the deployment details of capacity provisioning, load balancing, automatic scaling, and web application health monitoring.

Here, you select a platform, upload an application, or use a sample, and then run it. This chapter used a sample, and Tomcat is the platform for deploying the application code.

Click the Create Application button. This takes you to the **Create a web app** page shown in Figure 1-19.

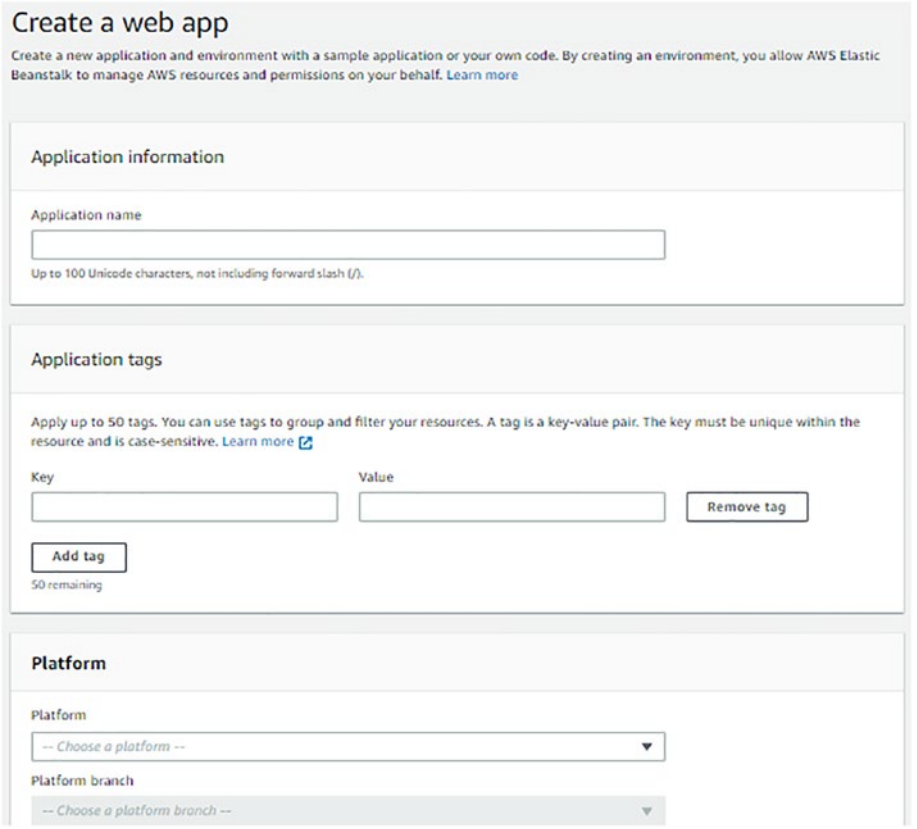


Figure 1-19. *Create a web app*

Name the application **My First Elastic Beanstalk Application**, as shown in Figure 1-20.

Figure 1-20. *Application name under Application information*

Next, select the platform from the drop-down list. Choose Tomcat, as shown in Figure 1-21.

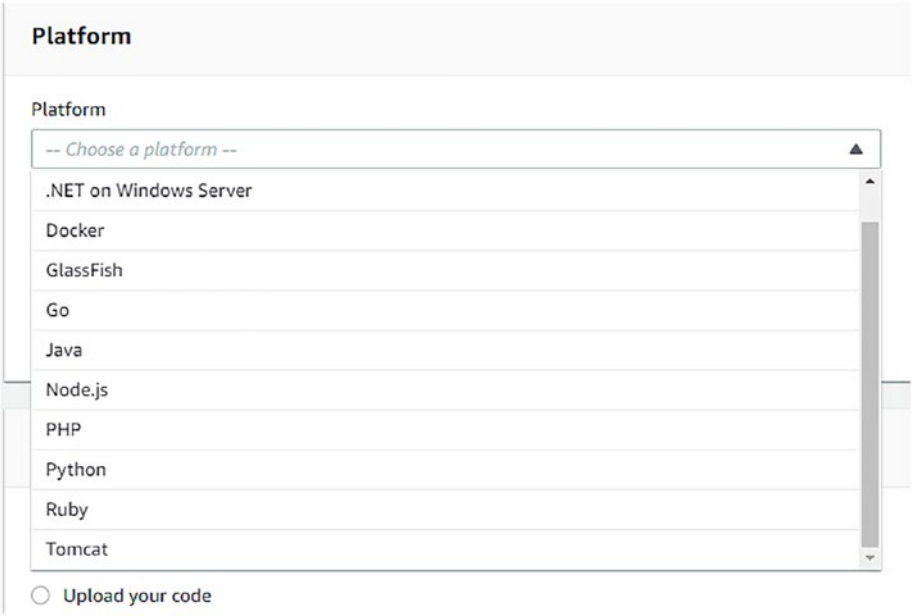


Figure 1-21. *Platform under Application information*

Select the default Tomcat branch and version, as shown in Figure 1-22.

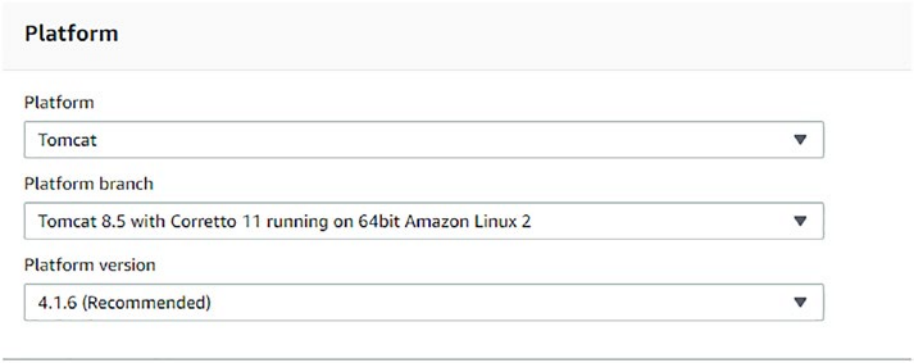


Figure 1-22. *Platform details on selecting platform under Application information*

Under Application code**,** select **Sample** application, as shown in Figure 1-23, and then click the **Create** application button.

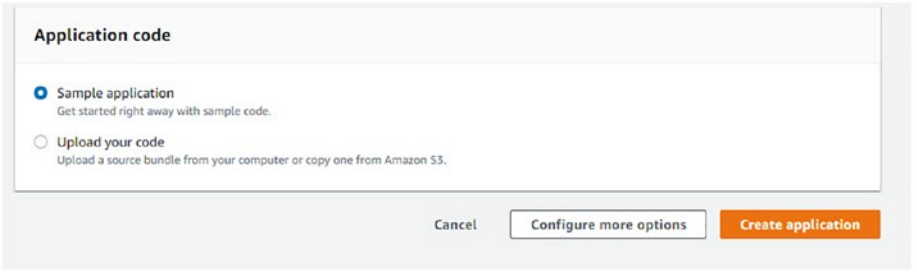


Figure 1-23. *Application code*

At this point, AWS provisions a server for you, as shown in Figure 1-24. It has Java installed, running on Linux, and Tomcat is already pre-installed.

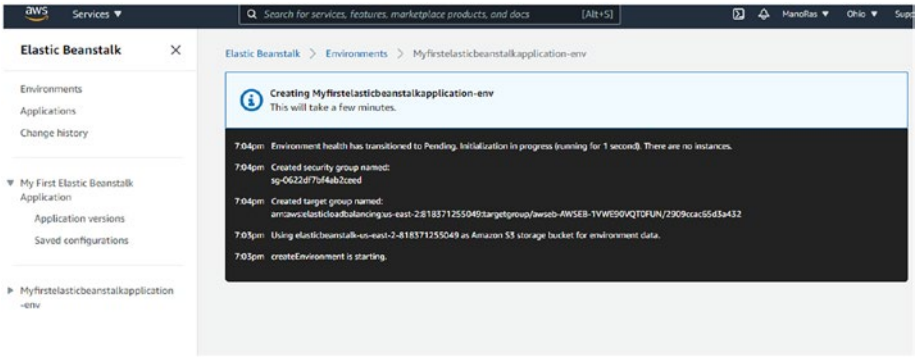


Figure 1-24. *Environment details*

You see diagnostics on the screen while the work is going on in the background.

Eventually, your application is deployed successfully, and the health status is OK, as shown in Figure 1-25. The link to your application appears in the top-left of the window. So, if you click the link, you see your application up and running.

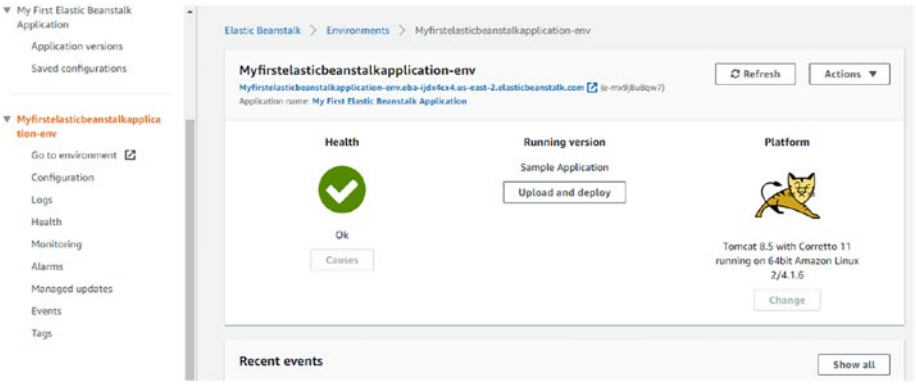


Figure 1-25. *Health OK*

This will start the sample application and configuring Linux, Tomcat, and Java. The logs shown in the following Figure 1-26 inform that the environment launched successfully.

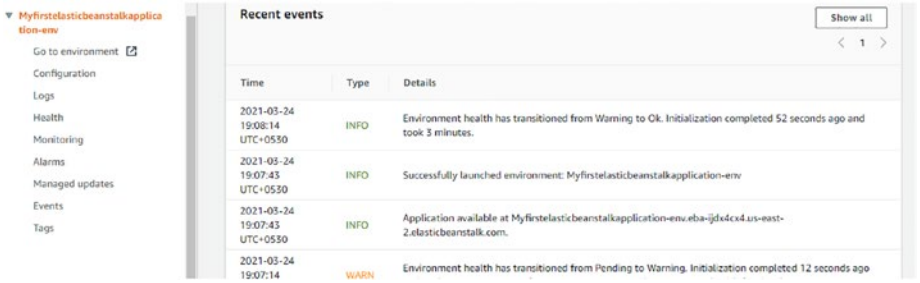


Figure 1-26. *Logs*

Figure 1-27 shows the Congratulations screen.

Figure 1-27. *Congratulations screen*

Your app is now running on the AWS cloud, and its URL is live on the Internet. Tomcat is running on your dedicated environment in the AWS cloud.

Right now, you are simply using the sample application, but later, you upload your applications and run them in the AWS cloud. You can add a custom domain name to the URL.

Create a HelloWorld JSP Application, Build WAR with Maven, and Upload WAR to Elastic Beanstalk

As a proof of concept, let's deploy the HelloWorld JSP application on Elastic Beanstalk. It's just a simple application on the Java side, which allows you to focus on the Elastic Beanstalk deployment process. Advanced Spring Boot and database CRUD operations are covered later.

To understand the mechanics of how to deploy, let's look at the step-by-step development process.

1. Create the HelloWorld JSP application in Spring Tool Suite (STS).
2. Package the WAR file using Maven.
3. Create a new application in Elastic Beanstalk.
4. Upload the WAR file to Elastic Beanstalk.

Create a HelloWorld JSP Application

Create a Maven web application project using STS or any IDE of your choice.

First, open Spring Tool Suite, select File menu ➤ New ➤ Maven Project, as shown in Figure 1-28.

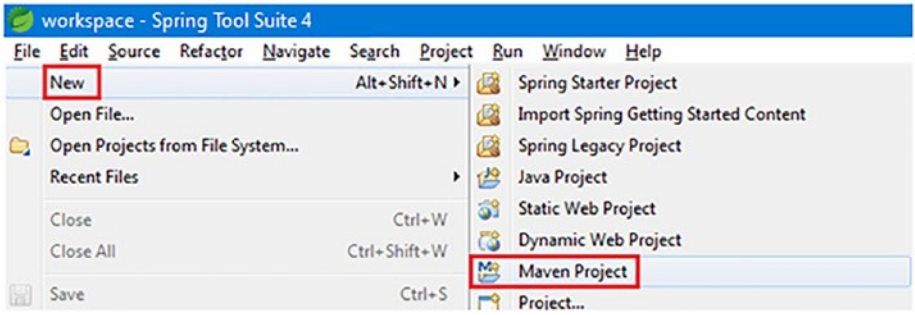


Figure 1-28. *Select Maven Project*

Figure 1-29 shows the New Maven Project wizard. Select the default location, and click Next.

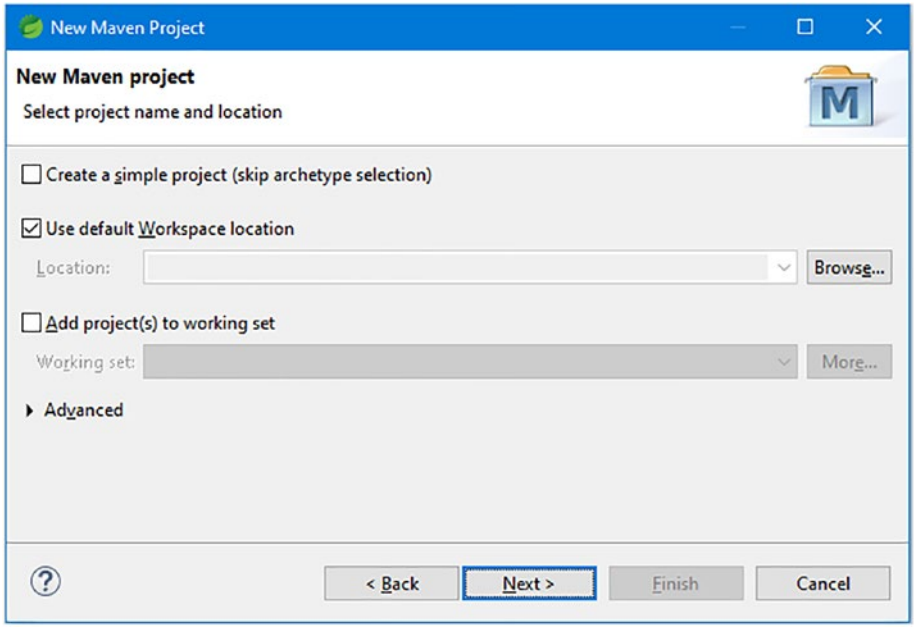


Figure 1-29. *New Maven Project wizard*

Then, select `maven-archetype-webapp` and click Next, as shown in Figure 1-30.

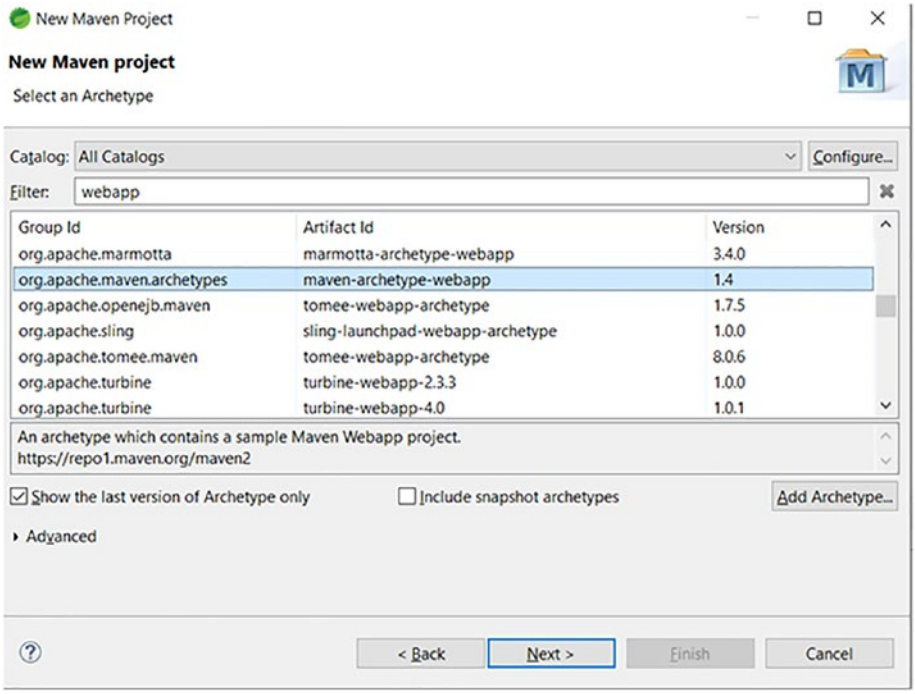


Figure 1-30. *Select an archetype*

Next, provide the group ID, artifact ID, and package information, and then hit the Finish button, as shown in Figure 1-31.

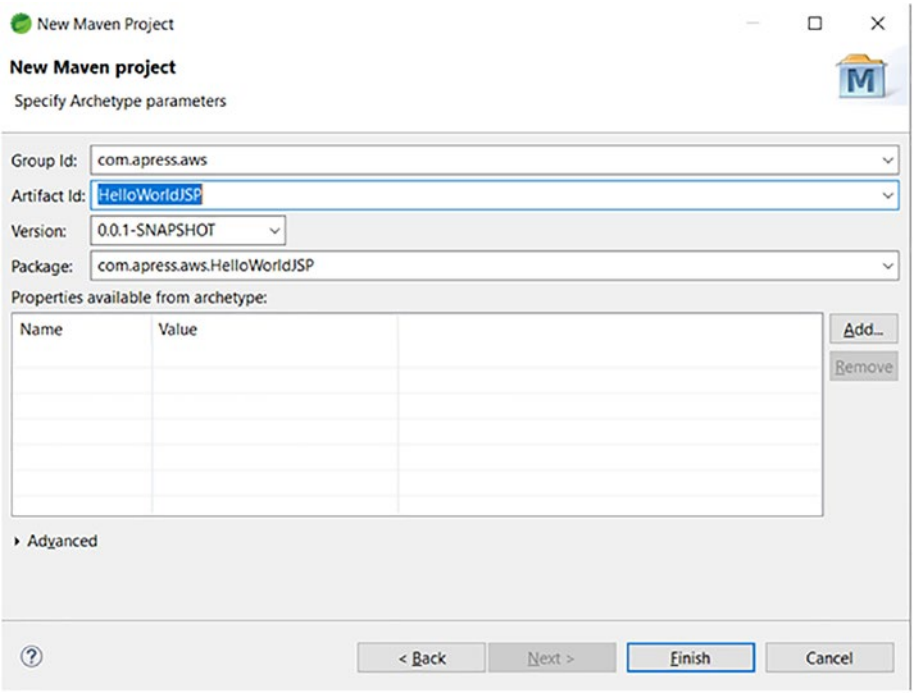


Figure 1-31. *Specify archetype parameters*

A project directory is created, as shown in Figure 1-32.

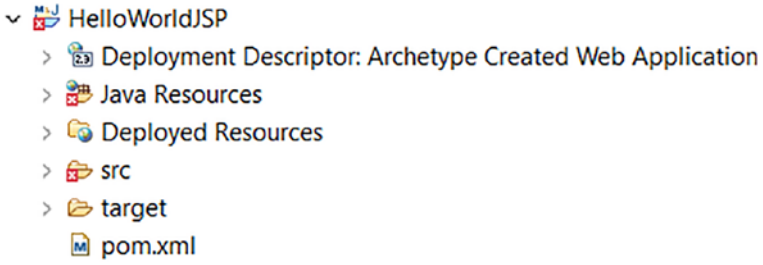


Figure 1-32. *HelloWorldJSP project directory*

If you look in the problem's view in IDE, the error shown is "The superclass javax.servlet.http.HttpServlet was not found on the Java Build Path". This error indicates that an HTTP servlet is not available in the project classpath.

Once you add a target runtime to the project, an HTTP servlet is available in the project classpath. Errors are resolved after configuring the runtime server, such as the Tomcat server.

To configure the Tomcat server, right-click the project and select Properties. Select Targeted Runtimes, and then select Apache Tomcat v8.5, as shown in Figure 1-33. Then, click the Apply and Close button.

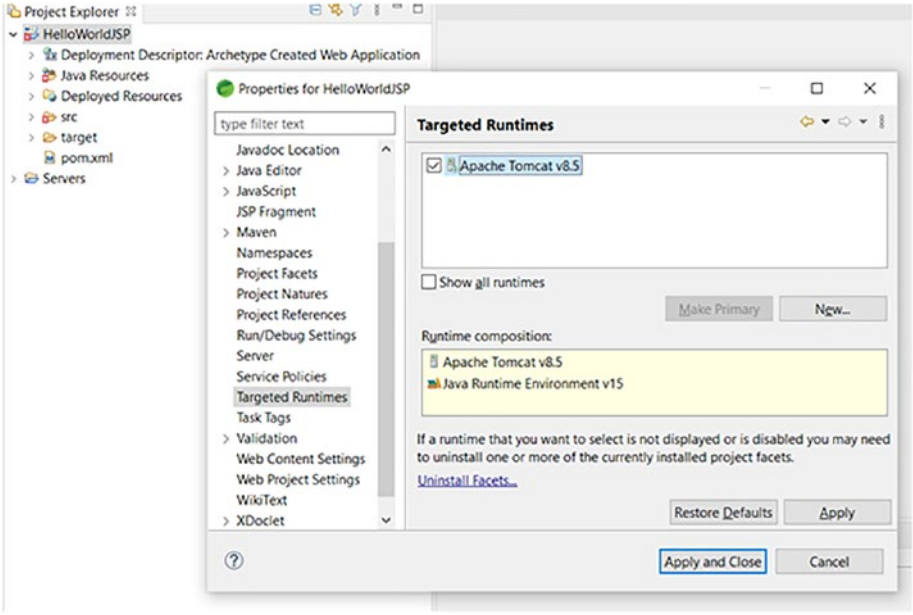


Figure 1-33. *Targeted runtimes*

To run the application on the local Tomcat server, right-click the project, select Run As and Run On Server. Select the Tomcat server in the window, and click the Finish button (see Figure 1-34).

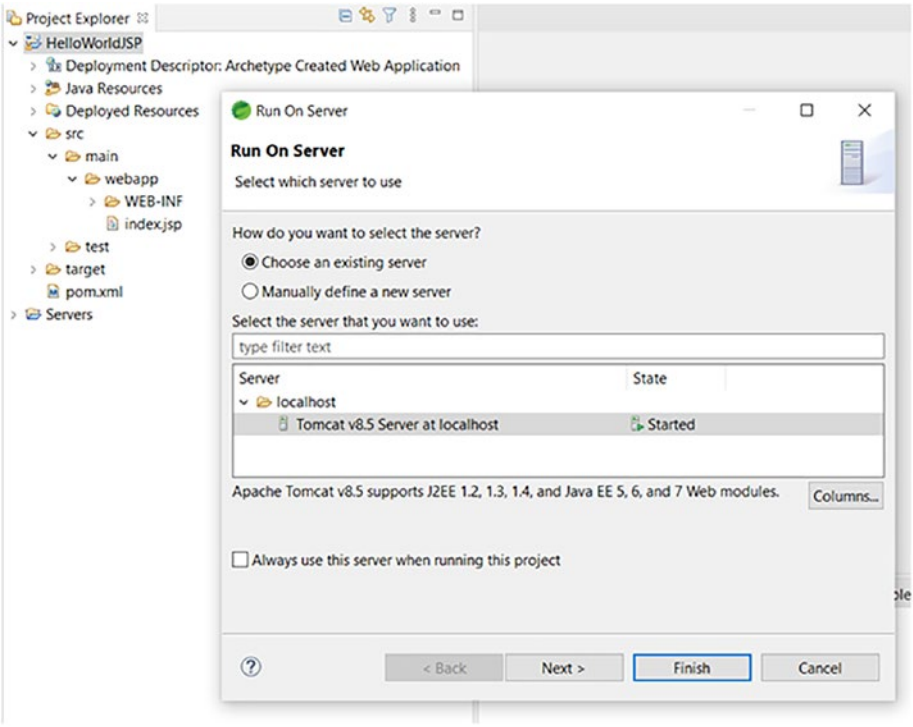


Figure 1-34. *Run On Server*

Type `http://localhost:8080/HelloWorldJSP/` in your favorite web browser to see the "Hello World!" message, as shown in Figure 1-35.

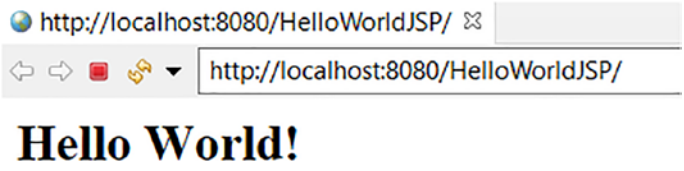


Figure 1-35. *Hello World! in browser*

Package a WAR File Using Maven

Now, let's package a WAR file using Maven in STS. Right-click the project and select **Run As ➤ Maven install**, as shown in Figure 1-36.

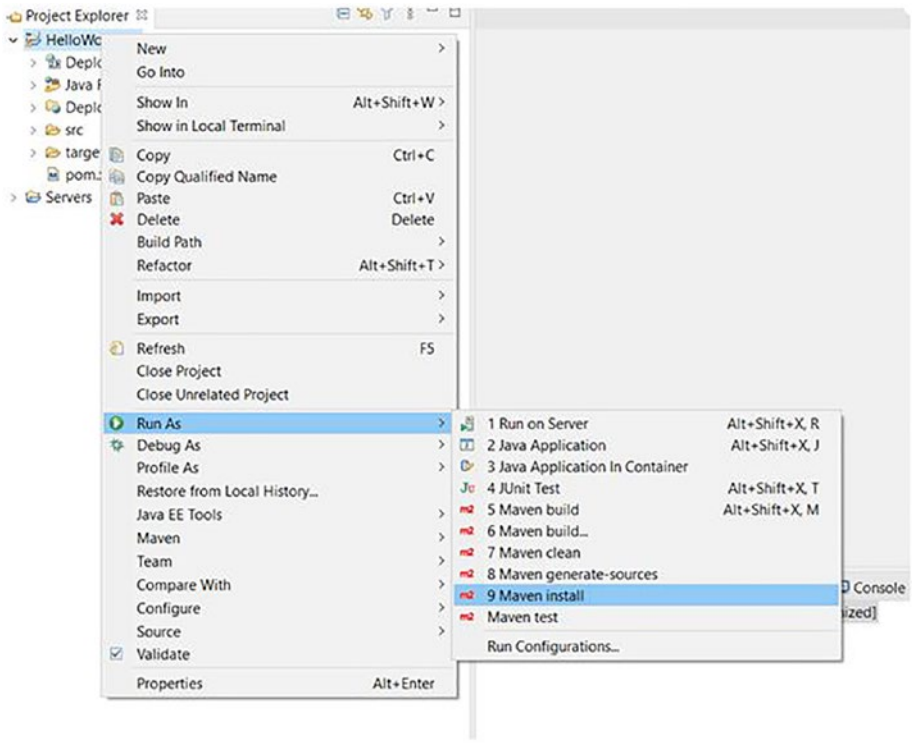


Figure 1-36. *Run As Maven install*

Once the build is successful, you can validate it with a success message in the console, as shown in Figure 1-37. This generates a WAR file.

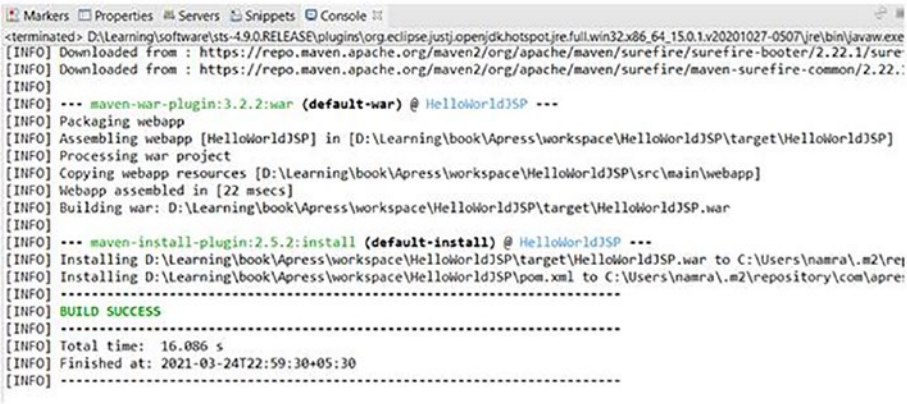


Figure 1-37. *Build success*

Refresh the project folder structure and expand the `target` folder, where you find a WAR file named `HelloWorldJSP.war`, as shown in Figure 1-38.

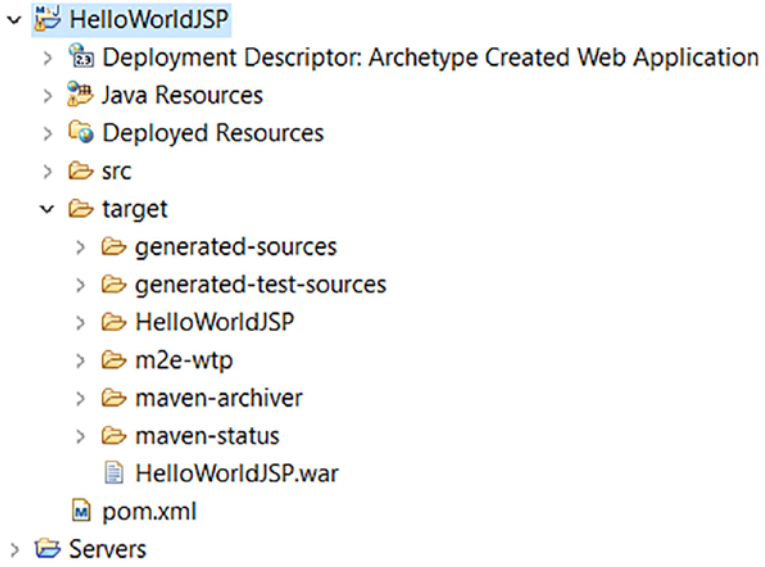


Figure 1-38. *Generated WAR file in the target folder*

Upload WAR to Elastic Beanstalk

Now, let's create a new application in Elastic Beanstalk and then upload the WAR file to it.

On the AWS console, go to the Elastic Beanstalk page. Figure 1-39 shows the application named My First Elastic Beanstalk Application.

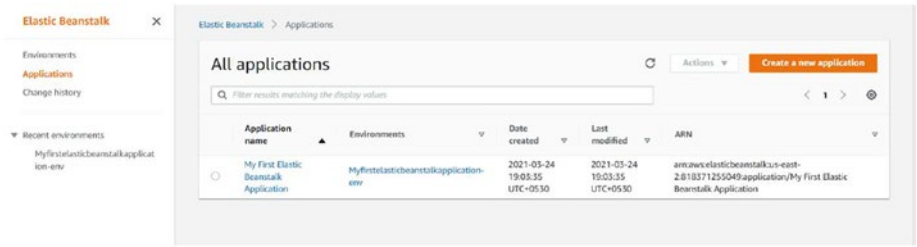


Figure 1-39. *Elastic Beanstalk application*

Now let's create a brand-new application by clicking the **Create a new application** button. Enter the application name as **helloworld**, as shown in Figure 1-40. Then, click the Create button.

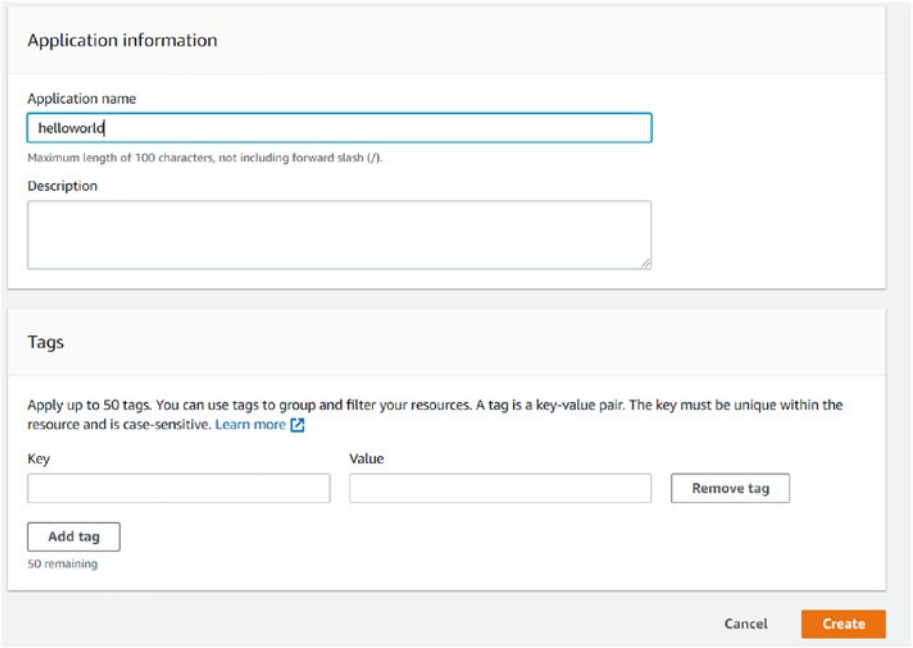


Figure 1-40. *Application information*

There is no environment that's already set up, as shown in Figure 1-41.

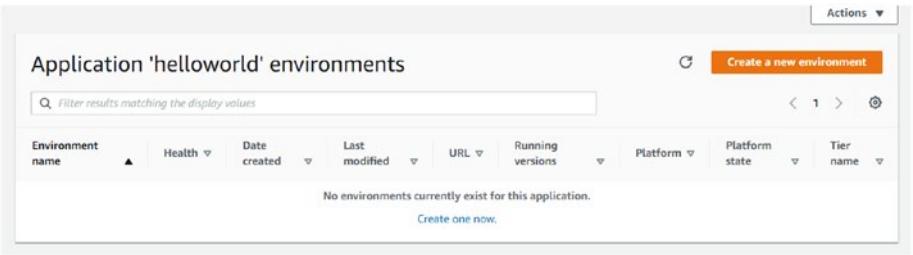


Figure 1-41. *Application environments*

Create an environment by clicking Create one now to select the environment tier, as shown in Figure 1-41.

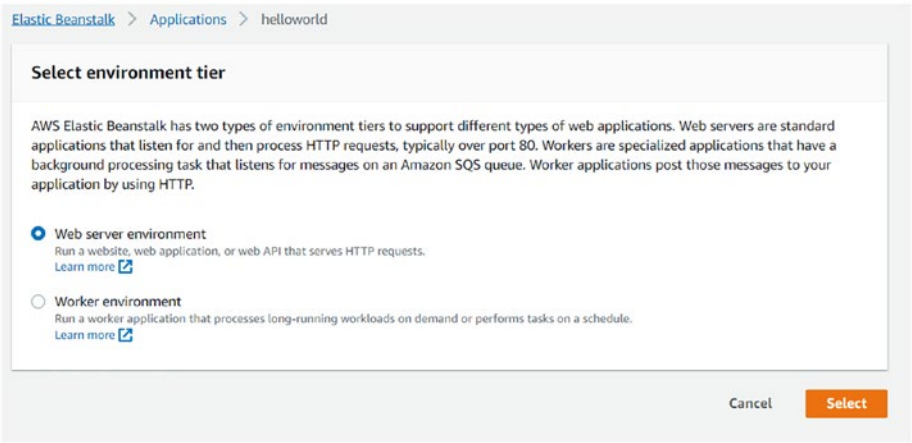


Figure 1-42. *AWS grouped by category*

For the environment tier, let's use a web server environment to run a web application. Elastic Beanstalk creates the server for us. Next, click the Select button. Now, you need to provide the environment information, as shown in Figure 1-43.

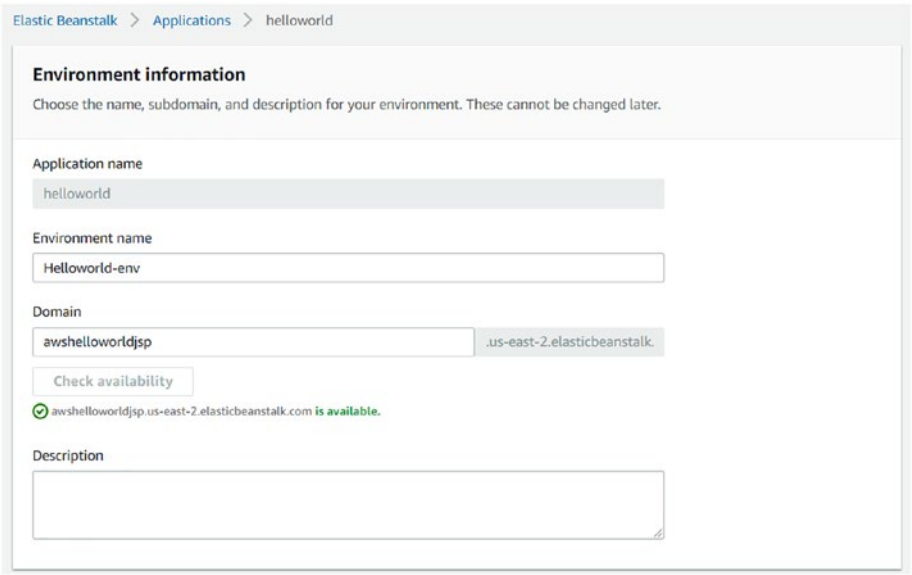


Figure 1-43. *Environment information*

Here, you need to provide details like a name for the environment and domain. Make sure that the environment URL is unique; here, name it **awshelloworldjsp**, which indicates that it is available for use. Then choose the platform details for the server, as shown in Figure 1-44.

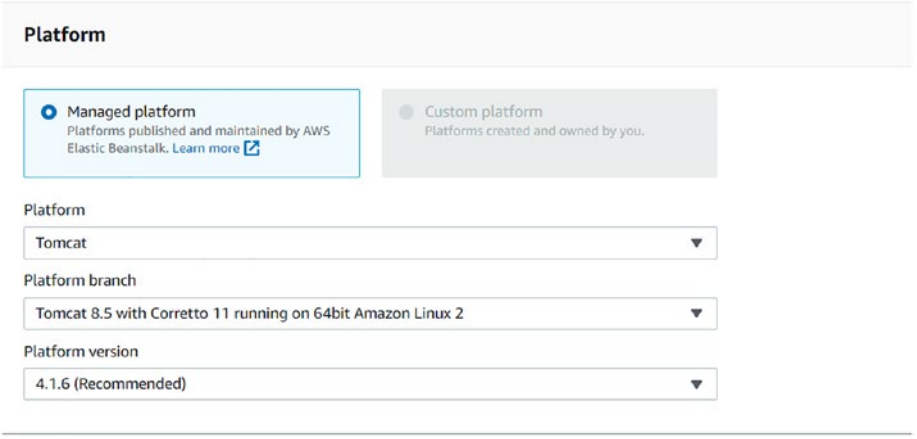


Figure 1-44. *Platform for server*

Here, select Managed platform, which is published and managed by AWS Elastic Beanstalk, and from the Platform drop-down list, choose Tomcat. So, Elastic Beanstalk creates a Tomcat server for you when it's spinning up the environment.

Now, you need to upload the WAR file to Elastic Beanstalk. Click Choose **file**, and select the HelloWorldJSP.war file from the local system, as shown in Figure 1-45.

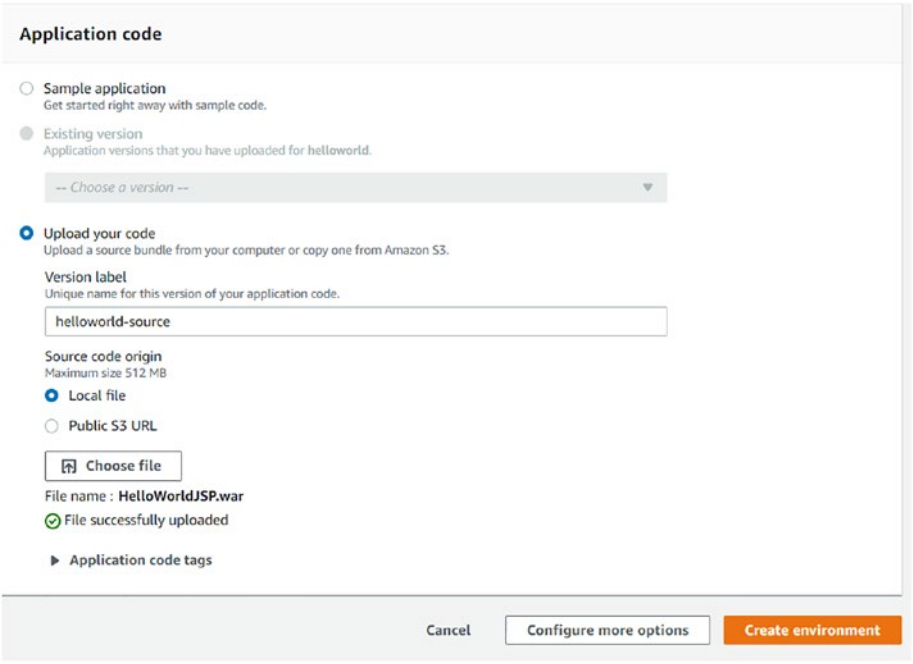


Figure 1-45. *Application code*

Once the file successfully uploads to Elastic Beanstalk, hit the Create environment button.

Behind the scenes, Amazon provisions a server to use with the operating system. They install Java and Tomcat and deploy your WAR file to the Tomcat environment. You get a green checkbox indicating success when everything is done, as shown in Figure 1-46. Here, the logs confirm that the environment successfully launched.

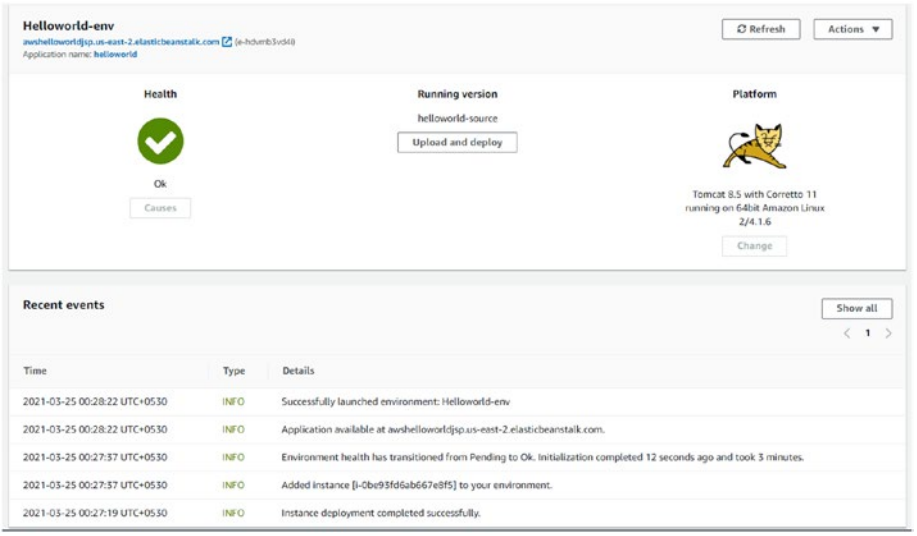


Figure 1-46. *Health OK and Recent events*

Click the application's link. The page opens in the browser displaying "Hello World!" (see Figure 1-47).

Figure 1-47. *Accessing application from browser by hitting URL*

This is your new application. The WAR file is deployed on the Amazon cloud in Elastic Beanstalk, and it's up and running.

Make sure to stop any unused AWS Elastic Beanstalk apps that you don't need. This helps prevent any overuse charges from Amazon.

Summary

This chapter overviewed Amazon Web Services (AWS). You learned about some AWS key services, such as EC2, Elastic Beanstalk, Amazon RDS, and Amazon Route 53. You created a free AWS account, a server, A HelloWorld JSP application, a WAR file with Maven, and uploaded the file to Elastic Beanstalk. Finally, you accessed your application in the browser to see the "Hello World!" message.

The next chapter deploys a Spring Boot application as a REST API in AWS.

CHAPTER 2

Deploy a Spring Boot Application as a REST API in AWS

The previous chapter provided an overview of Amazon Web Services (AWS), including services like Amazon Elastic Compute Cloud (Amazon EC2), AWS Elastic Beanstalk, Amazon Relational Database Service (Amazon RDS), and Amazon Route 53. First, you created a free AWS account for developers, explored Elastic Beanstalk, and created a server. Then, you created the HelloWorld JSP application. Finally, you built a WAR file with Maven and uploaded WAR to Elastic Beanstalk.

In this chapter, you create a Spring Boot application as a REST API in your local system. Then, you build the JAR using Maven for our Spring Boot application and deploy this JAR in Elastic Beanstalk so that anyone can access the REST API on the Internet. Finally, you explore logs from Elastic Beanstalk.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0_2

Build a Spring Boot Application as a REST API

Why use Spring Boot as a back-end framework? There are many frameworks available for developing web applications, and Spring Boot is just one among them. But, if you wish to build something fast, Spring Boot may be the primary choice as a web application development framework.

> *Working with Spring Boot is like pair programming with the Spring developers.*
>
> *—Josh Long @starbuxman*

Spring Boot provides production-ready applications and services that anyone can run with minimum fuss. Spring Boot is *opinionated*, which suggests ensuring decisions for developers and supporting ranges of nonfunctional features that are common in enterprise applications (embedded servers, security, health checks, metrics, and externalized configuration).

In this section, you develop your Spring Boot application, step by step. If you're already acquainted with this build process, you can skip to the end of this section to see how it all fits together. Spring offers different options for starting a brand-new project. For more information, refer to `https://spring.io/`.

Introduction to REST

Representational state transfer (REST) is an architectural style that describes how one system communicates or shares its state with another system. HTTP (Hypertext Transfer Protocol) may be a commonly used protocol to support a RESTful architecture. Standard HTTP methods like `POST`, `GET`, `PUT`, and `DELETE` access and manipulate RESTful web resources.

- The POST method performs a create operation by sending data to a server.
- The GET method retrieves data from a specified resource.
- The PUT method performs an update operation by sending data to a server.
- The DELETE method performs the delete operation.

A meaningful HTTP response status code always helps clients to utilize RESTful API. Table 2-1 describes several HTTP status codes that may be returned as the server response when calling a RESTful API.

Table 2-1. *HTTP Response Status Codes*

Code	Message	Description
200	OK	Successful response. The request has succeeded. (This is a standard HTTP response status code for a successful HTTP request.)
201	Created	Successful response. This is typically the HTTP response sent after POST or PUT requests are fulfilled, and a new resource has been created as a result.
204	Not Content	Successful response. This HTTP response code means that the request has been processed successfully but is not returning any content for this request.
400	Bad Request	Client error response. The request could not be fulfilled due to invalid syntax.
401	Unauthorized	Client error response. The request requires user authorization to get the requested response.

(*continued*)

Table 2-1. (*continued*)

Code	Message	Description
403	Forbidden	Client error response. The server refuses to fulfill the request because the client does not have access rights to the requested content.
404	Not Found	Client error response. The requested resource could not be found by the server.
409	Conflict	Client error response. The request cannot be completed because of a resource conflict with the current state of the server.

System Requirements

Spring Boot 2.0.3.RELEASE requires (at least) Java 8. So, the first thing that is required is the Java 8 SDK. If you have already set up the JDK in your system, you should check the current version of Java installed on your system before you begin.

```
$ java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

Spring offers the following three approaches to create a brand-new Spring Boot application.

- Use the Spring Boot **CLI** tool
- Use the Spring **STS** IDE
- Use Spring **Initializr** (http://start.spring.io/)

Create Spring Boot Application Using Spring Tool Suite

In this chapter, you build a RESTful application called `HelloSpringBoot` with REST endpoints, using STS IDE. The REST API layer is responsible for handling client requests and generating a response.

You create `HelloSpringBoot` by generating a Spring Boot application using Spring Tool Suite (STS). STS comes as a ready-to-use distribution of the latest Eclipse releases with pre-installed Spring IDE components.

Use a Spring Starter Project wizard to create a Spring Boot application, as shown in Figure 2-1. By default, the Spring Boot application runs on port 8080.

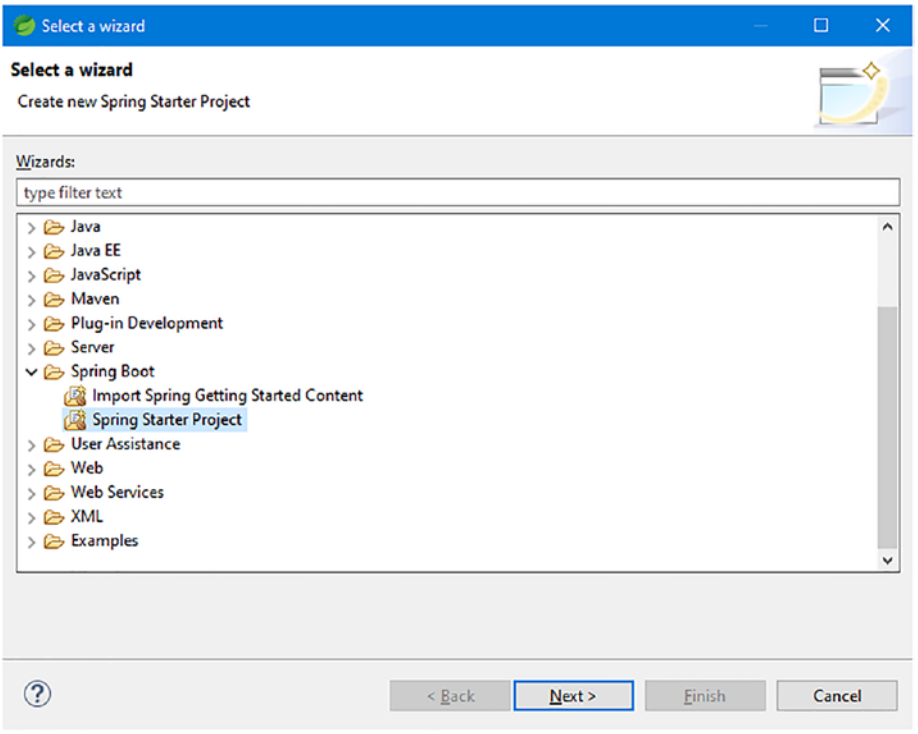


Figure 2-1. *The wizard to create a Spring Boot application*

Spring Boot provides *starters*. You need to provide project-related information, as shown in Figure 2-2.

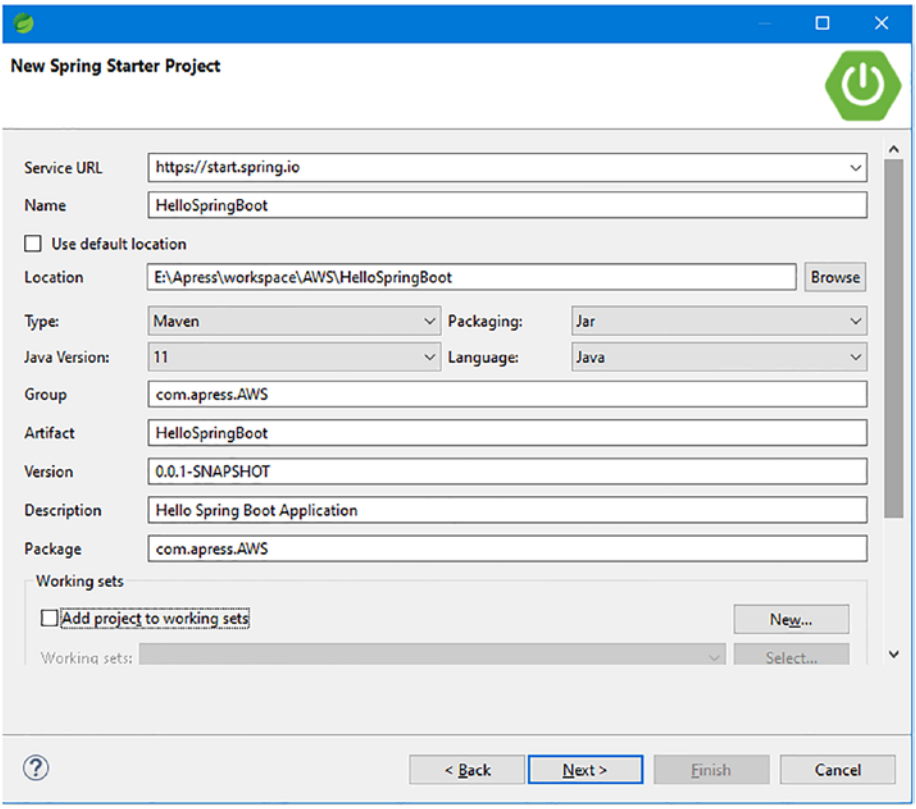


Figure 2-2. *Creating HelloSpringBoot using Spring Starter Project*

A starter in Spring Boot is a set of classpath dependencies that autoconfigure an application and enables a developer to build an application without any configuration.

In this chapter, you pick the web dependencies to build a simple `HelloSpringBoot` RESTful service, as shown in Figure 2-3.

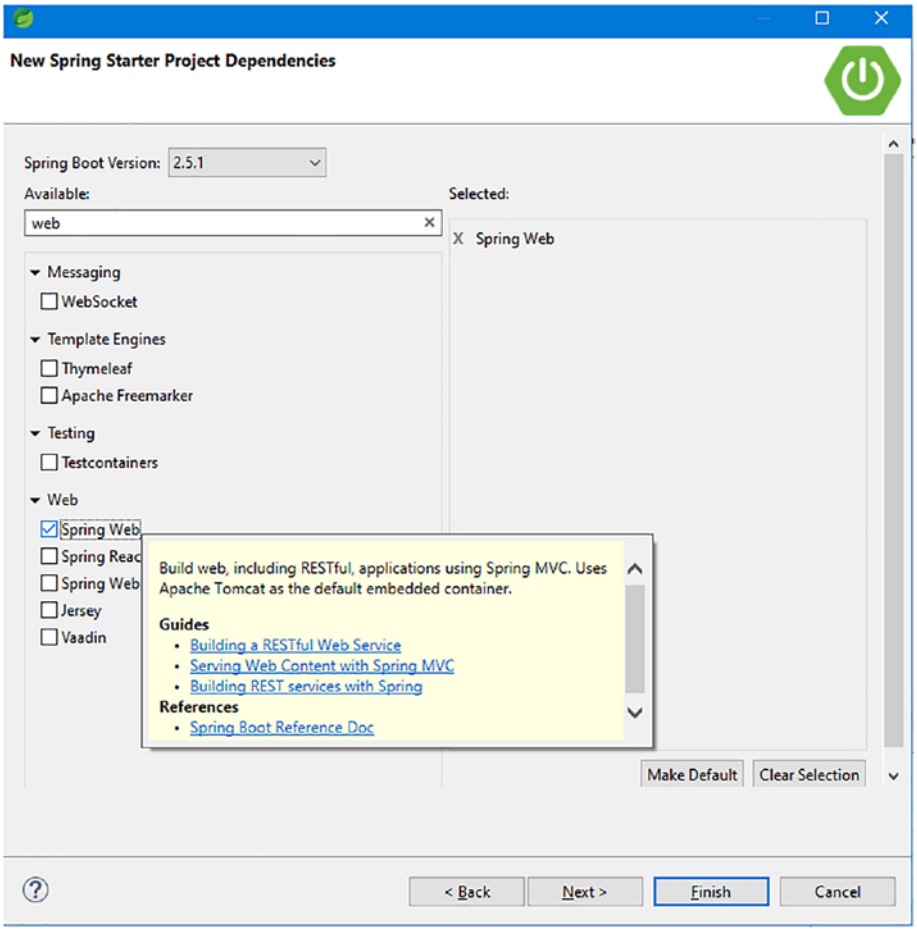


Figure 2-3. *Selecting a web dependency in the Spring starter*

Clicking the `Finish` button generates a workspace to create a new package, class, and static files in your resources. The final structure of the project looks like Figure 2-4.

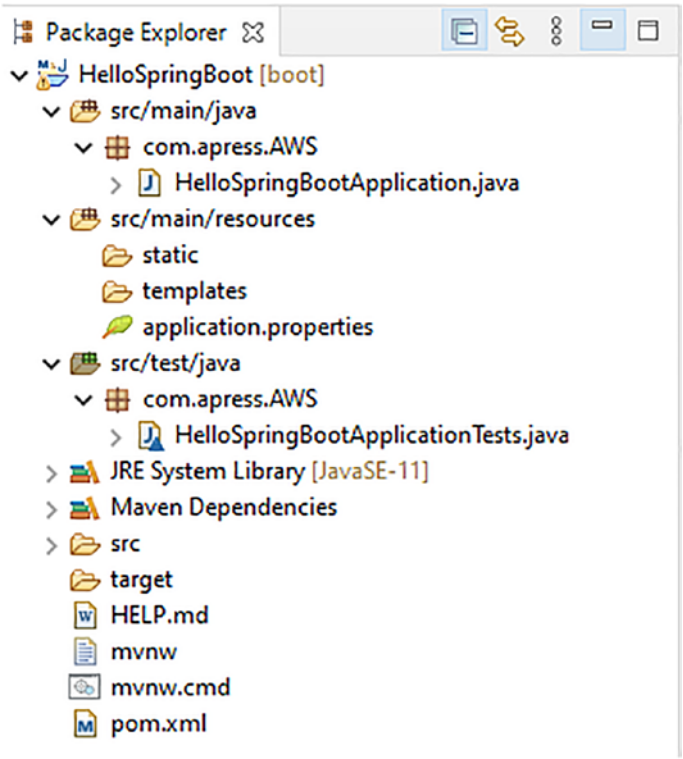


Figure 2-4. *Project structure*

Let's go through the code in the next section.

A Walk-Through

Let's walk through the code by going through the `pom.xml` file and the Java class files. Let's start with `pom.xml`.

pom.xml

When creating a Spring Boot application, all the dependencies that you select in the starter dialog are available in `pom.xml`, as shown in Listing 2-1. The `pom.xml` file is the recipe that builds your project.

Listing 2-1. pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.5.1</version>
            <relativePath/>
      </parent>
      <groupId>com.apress.AWS</groupId>
      <artifactId>HelloSpringBoot</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <name>HelloSpringBoot</name>
      <description>Hello Spring Boot Application</description>
      <properties>
            <java.version>11</java.version>
      </properties>
      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                 <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                  <scope>test</scope>
            </dependency>
```

```
        </dependencies>
        <build>
              <plugins>
                    <plugin>
                         <groupId>org.springframework.boot</groupId>
                          <artifactId>spring-boot-maven-plugin</
                          artifactId>
                    </plugin>
              </plugins>
        </build>
</project>
```

Note the following about Listing 2-1.

- The `<parent>` element specifies the parent POM of Spring Boot, which contains definitions for common components.
- The `<dependency>` element on spring-boot-starter-web tells Spring Boot that this is a web application and lets Spring Boot to form its opinions accordingly.

Before going further, let's look at Spring Boot's opinions and how it uses a starter like `spring-boot-starter-web to` form its configuration opinions.

The `HelloSpringBoot` application has used `spring-boot-starter-web` as Spring Boot's web application starter. And, based on this starter, Spring Boot has formed the following opinions.

- Spring web MVC for the REST framework
- Apache Jackson for the JSON binding
- Tomcat embedded web server container

After Spring Boot forms an opinion about the kind of application you plan to build, it delivers a collection of Maven dependencies supporting the POM contents and starter specified for the `HelloSpringBoot` application.

Write the Code

To bootstrap a Spring Boot application, you can start from a `main(...)` method. Most likely, you can delegate to the static `SpringApplication.run()` method, as shown in Listing 2-2.

Listing 2-2. \src\main\java\com\apress\AWS\HelloSpringBootApplication.java

```
package com.apress.AWS;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author RaviKantSoni
 *
 */
@SpringBootApplication
@RestController
public class HelloSpringBootApplication {

      public static void main(String[] args) {
            SpringApplication.run(HelloSpringBootApplication.
            class, args);
      }
```

```
        @RequestMapping("/greeting")
        public String greetingMessage() {
              return "Welcome to Hello Spring Boot Application!";
        }
}
```

Let's step through the important parts.

@SpringBootApplication Annotation

The first annotation in the `HelloSpringBootApplication` class is `@SpringBootApplication`, introduced in Spring Boot 1.2.0. It adds the following annotations.

- **`@Configuration`**: A class annotated with the `@Configuration` annotation can be used by the Spring Boot container as a source of Spring Bean definitions, which is not specific to Spring Boot. This class may contain one or more Spring Bean declarations by annotated methods with the `@Bean` annotation.
- **`@EnableAutoConfiguration`**: This annotation is part of the Spring Boot project that tells Spring Boot to start adding beans using classpath definitions. Autoconfiguration intelligently guesses and automatically creates and registers beans that you are likely to run with the application, thus simplifying the developer's work.
- **`@ComponentScan`**: This annotation tells Spring Boot to look for specific packages to scan for annotated components, configurations, and services.

@RestController and @RequestMapping Annotations

`@RestController` is another annotation in the `HelloSpringBootApplication` class. It is a stereotype annotation. The `@RequestMapping` annotation provides "routing" information and tells Spring Boot that any HTTP request with the path `/greeting` should be mapped to the `greetingMessage()` method.

The `@RestController` and `@RequestMapping` annotations come from Spring MVC (these annotations are not specific to Spring Boot).

The main Method

The most important part of the `HelloSpringBootApplication` class is the `main(...)` method. The application developed using Spring Boot contains the main method, which internally calls Spring Boot's `SpringApplication.run()` method to launch an application. The class that contains a `main` method is the main class and is annotated with the `@SpringBootApplication` annotation.

Run a Spring Boot Application in STS

Spring Boot application created using the Spring Starter Project wizard comes in two flavors: WAR and JAR. This wizard allows you to choose between WAR and JAR in its packaging option.

> *As* Josh Long *said in one of his talks in the Spring IO, "Make JAR, not WAR."*
>
> —https://twitter.com/springcentral/status/598910532008062976

Spring Boot favors JAR over WAR by allowing you to easily create stand-alone JAR packaged projects that add an embedded web server (Apache Tomcat is the default web server) inside the created artifact. It helps developers reduce the overhead of setting up local or remote Tomcat servers, WAR packaging, and deploying.

You don't need any special tooling from STS to run your Spring Boot application locally. You can run it by selecting **Run As ➤ Java Application**, either from the standard Eclipse Java debugging tools or STS. The benefits of using STS over other IDEs are that it provides a dedicated launcher, which does the same thing as other IDE does, but STS adds a few useful bells and whistles on the top of it. So, let's use STS to run the Spring Boot application, as shown in Figure 2-5. Simply right-click the `HelloSpringBoot` project, and then select **Run As ➤ Spring Boot App**.

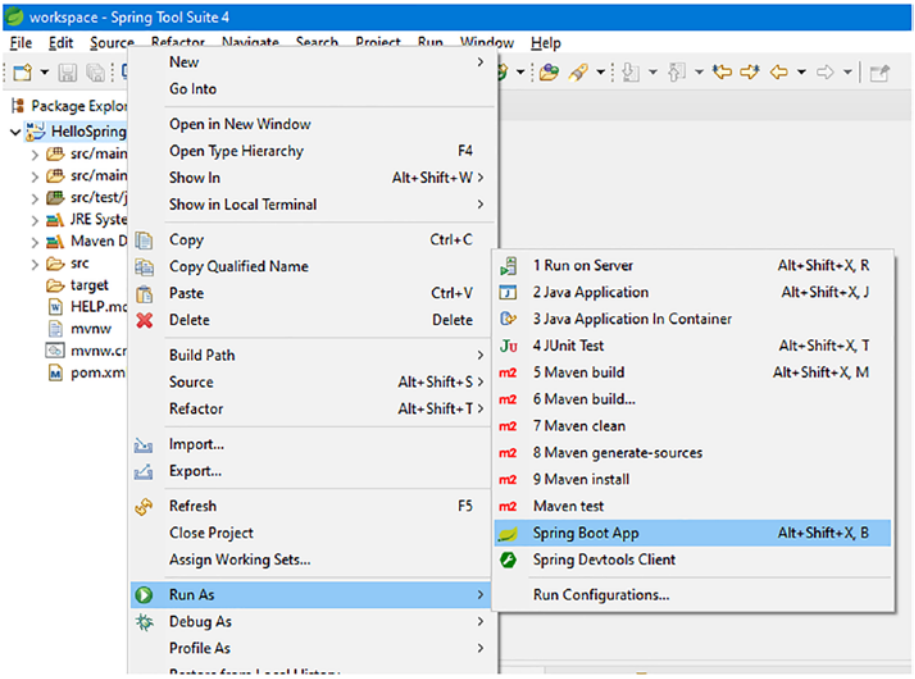


Figure 2-5. *Wizard in STS to run the application*

The Spring Boot application starts with output in the console, as shown in Figure 2-6.

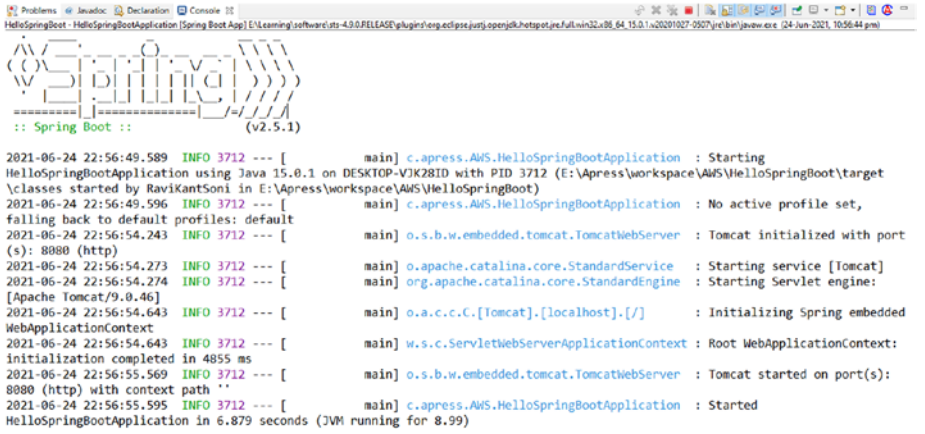


Figure 2-6. *Output on the STS console*

If the Spring Boot application runs successfully, the last line on the console states, `Started HelloSpringBootApplication`.

Congratulations! You have successfully set up and run the application using Spring Boot. Now it's time to visit `http://localhost:8080/greeting` in the browser to see the web page, as shown in Figure 2-7.

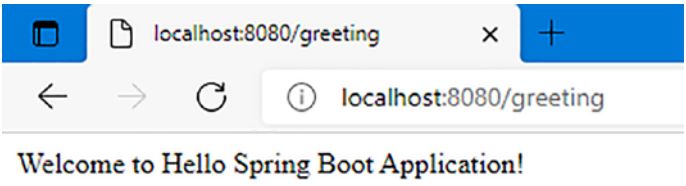


Figure 2-7. *Accessing the REST endpoint from the browser*

Add Swagger UI to a Spring Boot Application

Nowadays, front-end components and back-end components usually isolate a web application. Building a back-end API layer introduces new challenges that have gone beyond implementing endpoints. Usually, you expose REST APIs as a back-end component for the front-end component or any third-party app integrations.

Thus, your REST API documentation becomes more fragile. REST API documentation should be well structured so that it's informative, concise, and easy to read. In such a scenario, it is essential to have a proper specification for the back-end REST API. Moreover, reference API documentation should simultaneously describe each change in the API. Fulfilling this manually is a time-consuming and tedious exercise, so automation of this process was inevitable.

Swagger supports generating the API documentation automatically, and it also ensures that any changes made to the API are available to the customer immediately. In this section, you learn how to use Swagger 2 in a Spring Boot application to generate REST API documentation.

Introduction to Swagger 2

Swagger 2 is an open source project that documents RESTful APIs. It is language-agnostic and is extensible to new technologies and protocols beyond HTTP protocol.

This Swagger 2 version defines a set of HTML (HyperText Markup Language), JavaScript, and CSS assets to dynamically generate documentation from a Swagger-compliant API. The Swagger UI project bundled these sets of files to display the API on the browser, and it returns response data in the JSON format. Besides rendering documentation, Swagger UI also allows other API developers or API consumers to interact

with the API's resources without having any of the API implementation logic in place.

The Swagger 2 specification, which is understood as the OpenAPI specification, has several implementations. Springfox has recently replaced Swagger-SpringMVC (Swagger 1.2 and older) and is popular for Spring Boot applications. Springfox supports both Swagger 1.2 and 2.0.

Let's use Swagger 2 for our Spring Boot REST web service, using the Springfox implementation of the Swagger 2 specification.

Add Dependency in a Maven POM

Let's use the Springfox implementation of the Swagger specification. Its latest version can be found on Maven Central. To add it to our Spring Boot–based projects, you need to add a single `springfox-boot-starter` dependency, as shown in Listing 2-3.

Listing 2-3. Add Springfox Dependency in pom.xml

```
<dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-boot-starter</artifactId>
      <version>3.0.0</version>
</dependency>
```

Configure Swagger 2 into a Project

The configuration of Swagger 2 mainly focused on the Docket bean. For our Spring Boot application, let's create a Docket bean in a Spring Boot configuration class file to configure Swagger 2 for our application. A Springfox Docket instance provides the primary API configuration with default methods for configuration. Listing 2-4 shows our Spring Boot `SwaggerConfig` configuration class.

Listing 2-4. \src\main\java\com\apress\AWS\config\SwaggerConfig.java

```
package com.apress.AWS.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.
RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.
EnableSwagger2;

/**
 * @author RaviKantSoni
 *
 */
@Configuration
@EnableSwagger2
public class SwaggerConfig {
      @Bean
      public Docket productApi() {
            return new Docket(DocumentationType.SWAGGER_2)
                        .select()
                        .apis(RequestHandlerSelectors.any())
                        .paths(PathSelectors.any())
                        .build();
      }

}
```

In this `SwaggerConfig` configuration class, the `@EnableSwagger2` annotation enables Swagger support in the class. The `select()` method called on the Docket bean instance returns an `ApiSelectorBuilder`, which provides a way to control the endpoints exposed by Swagger.

In the code, the `RequestHandlerSelectors` and `PathSelectors` use `any()` to make documentation for our entire API available through Swagger.

Configuration Verification

At this point, you should be able to test the Swagger configuration by restarting the application and go to `http://localhost:8080/v2/api-docs`.

As shown in Figure 2-8, the result is a JSON response with a large number of key/value pairs, which is not very human-readable.

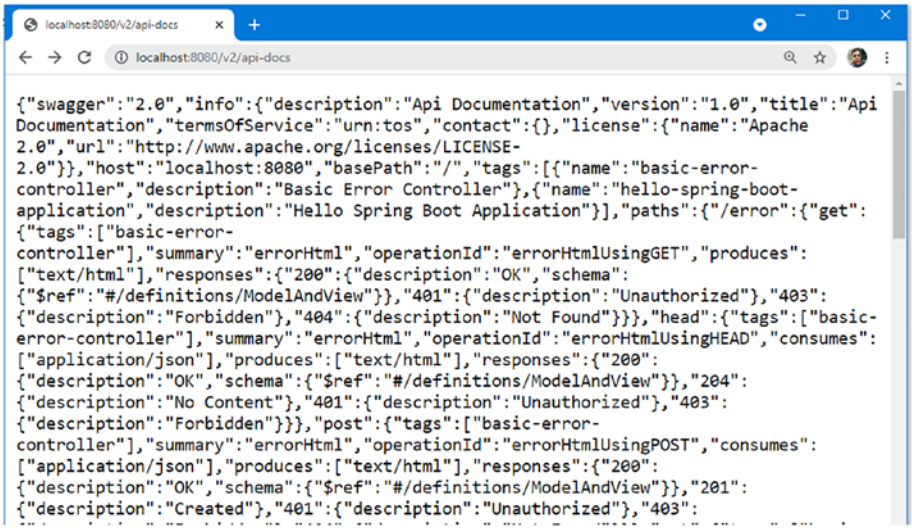


Figure 2-8. *Swagger JSON output*

Swagger UI

You want human-readable structured documentation. Swagger UI is a built-in solution that makes user interaction with the Swagger-generated API documentation much easier. In your browser, go to `http://localhost:8080/swagger-ui/`.

You see the generated documentation rendered by Swagger UI, as shown in Figure 2-9.

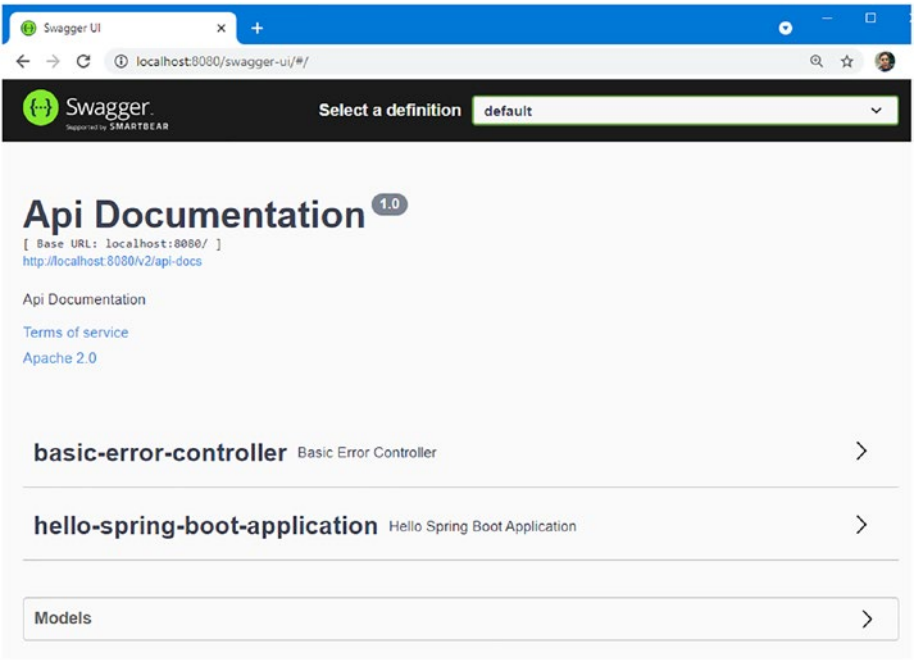


Figure 2-9. *The Swagger API documentation page*

The Basic Error Controller is the API that comes with Spring MVC. Models show all the `Model` objects.

Within Swagger's response is a list of all controllers defined in our application. Clicking any of them lists the operation endpoints with valid HTTP methods (`DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT`), as shown in Figure 2-10.

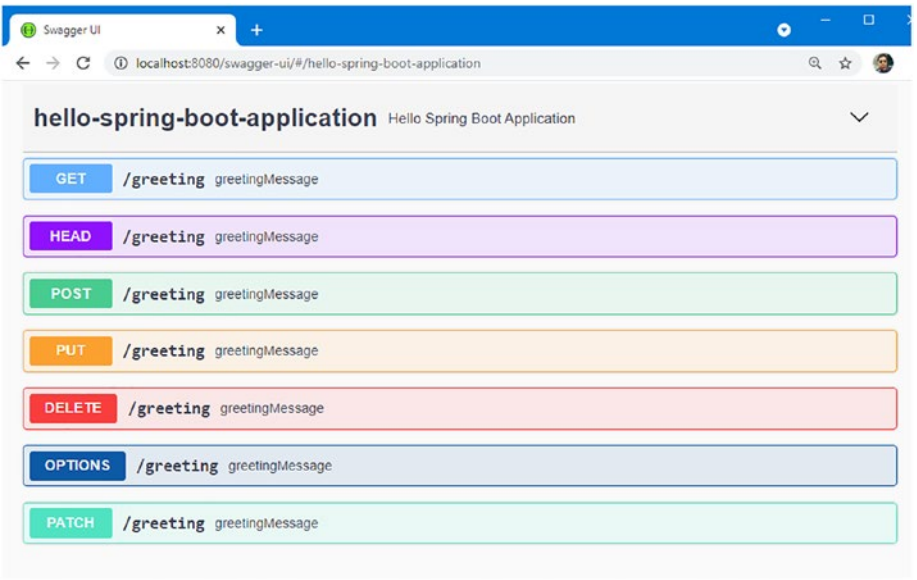


Figure 2-10. *Swagger UI lists REST endpoints*

For more information on Swagger, refer to the official documentation page at https://swagger.io/docs/specification/2-0/basic-structure/.

Configure the Server Port for a Spring Boot Project

The default Port with which the Spring Boot application has been configured is 8080, which means a Spring Boot application starts with an embedded Tomcat server at a default port 8080. You can change this default embedded server port to any other port.

AWS Elastic Beanstalk assumes that the Spring Boot application listens on port 5000. You can change the default port by simply making an entry in the `application.properties` file in your Spring Boot application, as shown in Listing 2-5.

Listing 2-5. \src\main\resources\application.properties

```
server.port=5000
```

Let's build and run our Spring Boot application in another port and then open the browser to access our application. This time, you are not using default port 8080 in the browser; rather, port 5000. In your browser, go to `http://localhost:5000/swagger-ui/`.

Figure 2-11 shows the generated documentation rendered by Swagger UI.

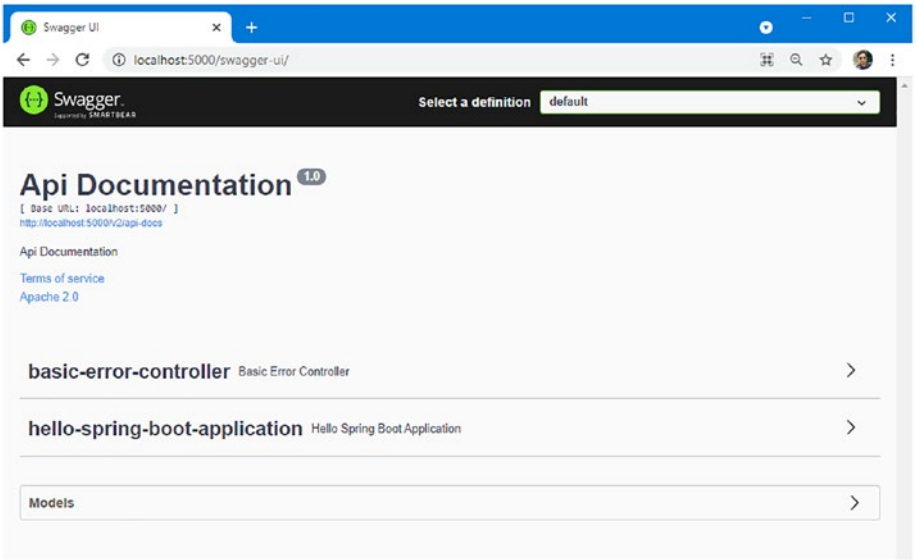


Figure 2-11. *URI with port 5000*

Build a JAR for a Spring Boot Application

Since you have successfully created a Spring Boot application as a REST API, let's deploy it to AWS Elastic Beanstalk. To achieve this goal, you need a deployable unit of our project.

Before starting the actual process, make sure that you have Apache Maven (a command-line tool for building and managing any Java-based project) installed in your local system. If you do not already have Maven installed, you can follow the instructions at `maven.apache.org`.

A Spring Boot application's default mode packages executable JARs (also known as *fat JARs*). So, a JAR is used as a deployable unit for this project. To build a JAR, you can either use STS or the command prompt.

From STS, right-click the `HelloSpringBoot` project, and then select **Run As ➤ Maven build**, as shown in Figure 2-12.

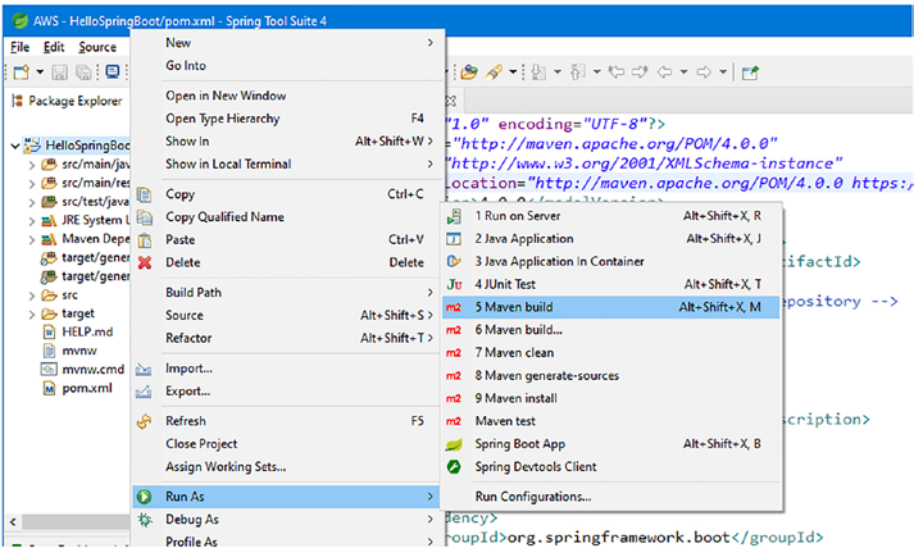


Figure 2-12. *Maven build using STS*

This opens the Edit Configuration window. Enter **package** in the Goals text box. Click Apply, and then click Run, as shown in Figure 2-13.

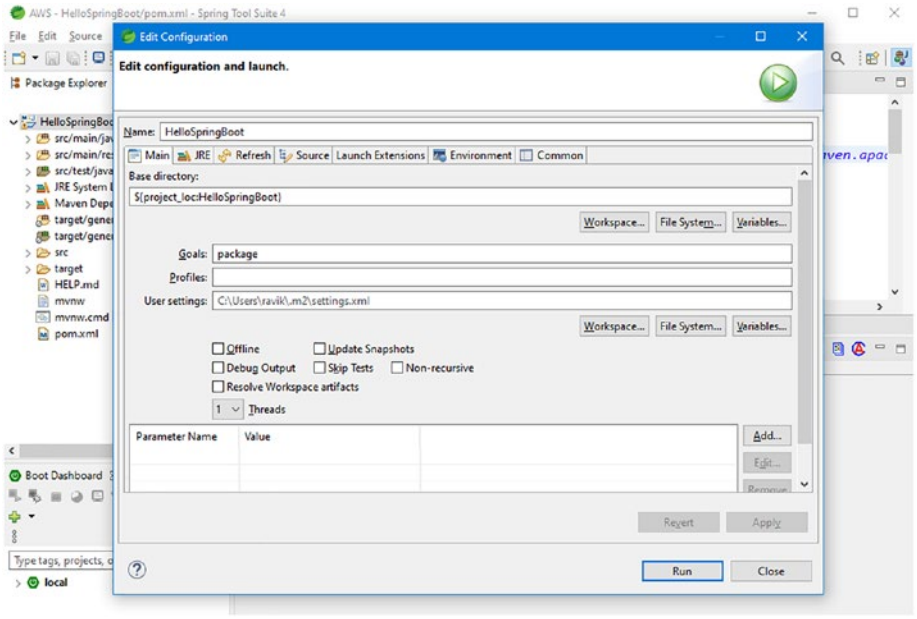


Figure 2-13. *Edit configuration window*

The HelloSpringBoot application starts building. You see that the entire Maven build runs, as shown in Figure 2-14.

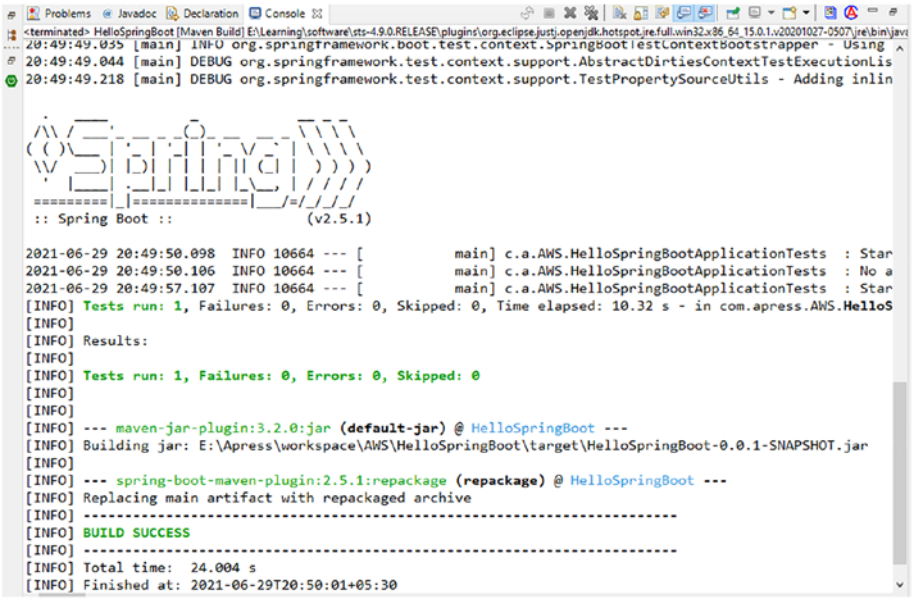


Figure 2-14. *Build output on console in STS*

A JAR named `HelloSpringBoot-0.0.1-SNAPSHOT.jar` has been created in the project's `target` folder.

```
Building jar: E:\Apress\workspace\AWS\HelloSpringBoot\target\
HelloSpringBoot-0.0.1-SNAPSHOT.jar
```

To build a JAR using the command prompt, go to your project directory (where you have created the Spring Boot project) and copy the path. Change the working directory to the project path on the command prompt, as shown in Figure 2-15.

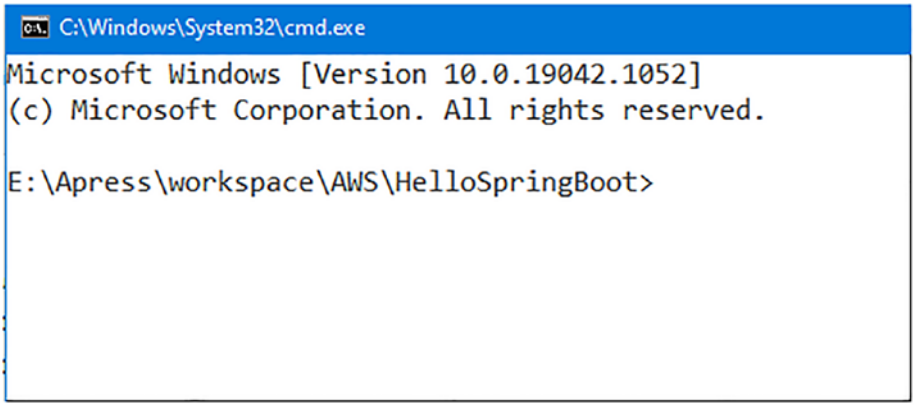


Figure 2-15. *Directory to the project path on the command prompt*

Build the project using the following command in the command prompt.

```
E:\Apress\workspace\AWS\HelloSpringBoot>mvn clean install
```

This starts building the application. The JAR file named `HelloSpringBoot-0.0.1-SNAPSHOT.jar` has been created, as shown in Figure 2-16.

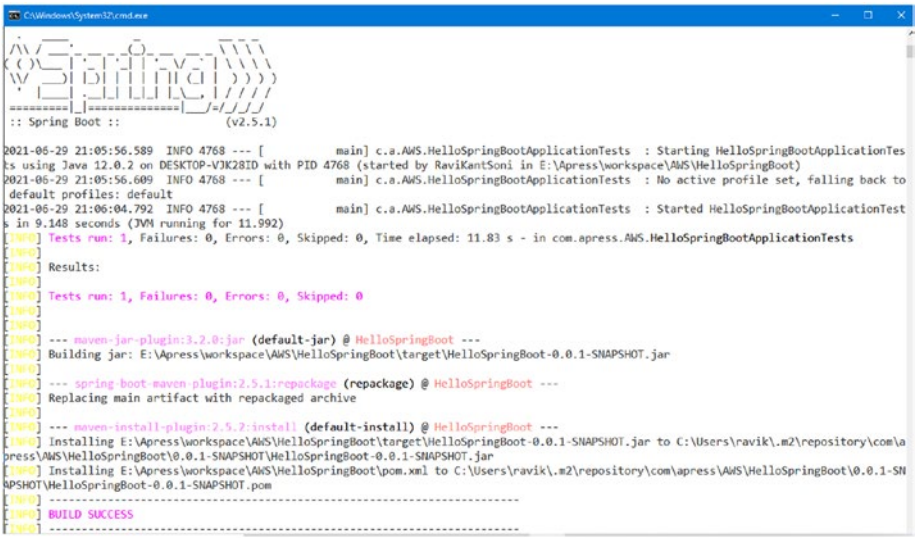


Figure 2-16. *Build success in command prompt*

You need to pick up and deploy the generated JAR file to AWS Elastic Beanstalk.

Deploy a Spring Boot Application in AWS Elastic Beanstalk

You have locally created and run the HelloSpringBoot REST API and created a JAR file in the `target` folder. Now, let's deploy to AWS Elastic Beanstalk.

Sign in to the AWS Management Console using AWS credentials, and select Elastic Beanstalk as the service. There are already two applications, named `My First Elastic Beanstalk Application` and `helloworld`, created in Chapter 1 (see Figure 2-17).

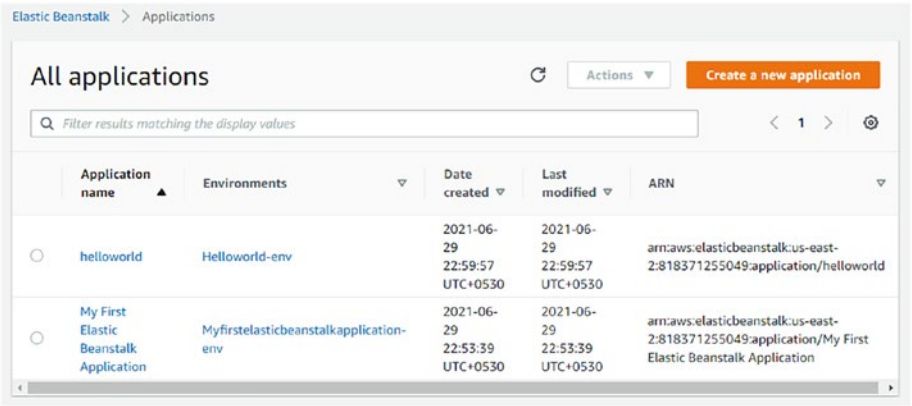


Figure 2-17. *Elastic Beanstalk application*

Now let's create a brand-new application for our Spring Boot REST API. First, click the **Create a new application** button, and enter **HelloSpringBoot** the application name. Next, click the Create button to create a new environment for the application. Then, click the **Create one now** link. Select **Web server environment** as the environment tier, and then click the Select button.

On the environment information page, name the domain `HelloSpringBoot`, and check for availability. Then, select Java as the managed platform, as shown in Figure 2-18.

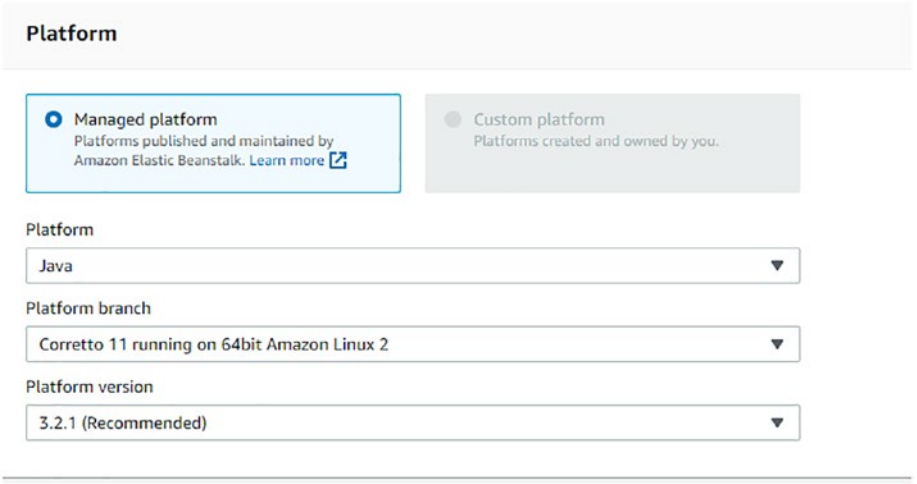


Figure 2-18. *Platform as Java*

Finally, upload your code by selecting the JAR file from the `target` folder (e.g., in the authors' local it is `E:\Apress\workspace\AWS\HelloSpringBoot\target\HelloSpringBoot-0.0.1-SNAPSHOT.jar`) of the project, and then click the **Create environment** button.

Elastic Beanstalk coordinates the creation and deployment of all AWS resources required to support the environment during the launch process. This includes, but is not limited to, launching two EC2 instances, creating a load balancer, and creating a security group, as shown in Figure 2-19.

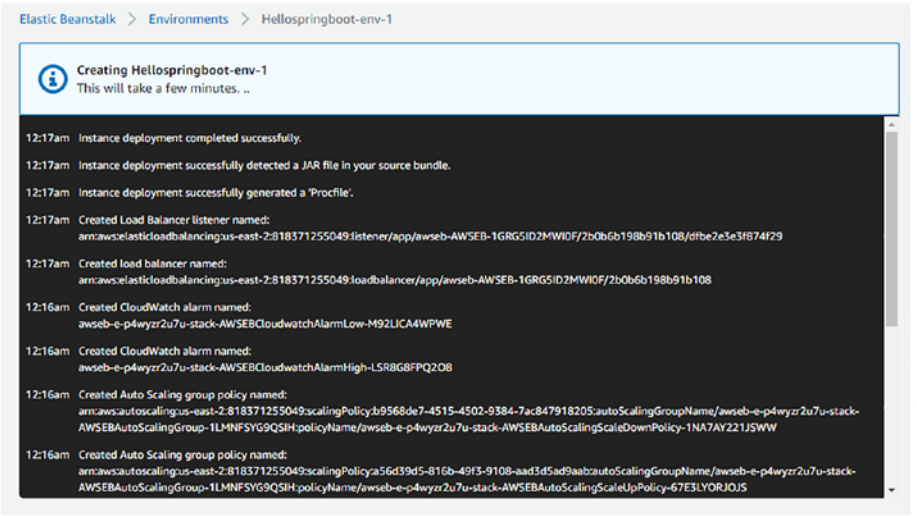


Figure 2-19. *Creating Hellospringboot-env in Elastic Beanstalk*

Once the environment has been created and the resources have been deployed, notice that Health is reported as severe (see Figure 2-20). This is because the Spring Boot application still needs some configuration.

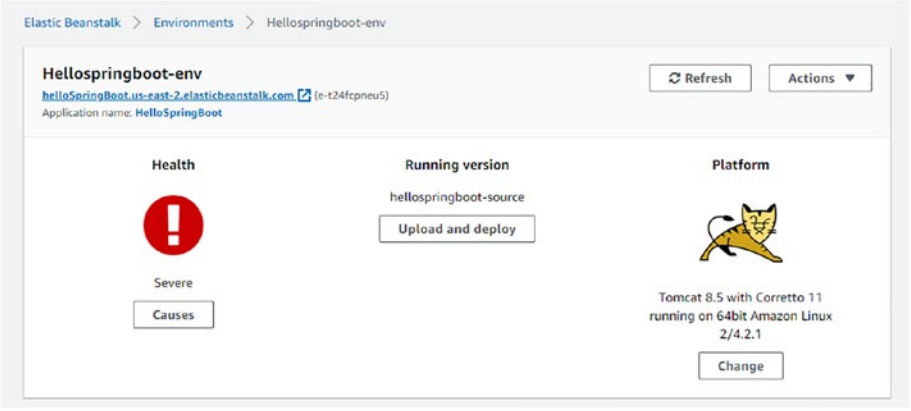


Figure 2-20. *Severe health of Spring Boot application*

AWS Elastic Beanstalk assumes that the application listens on port 5000. To fix the discrepancy, change the port the Spring Boot application listens on. So, you need to specify the SERVER_PORT environment variable in the Elastic Beanstalk environment and set the value to 5000.

Go to the Configuration page in your environment. Under Configuration, click the Edit icon, as shown in Figure 2-21.

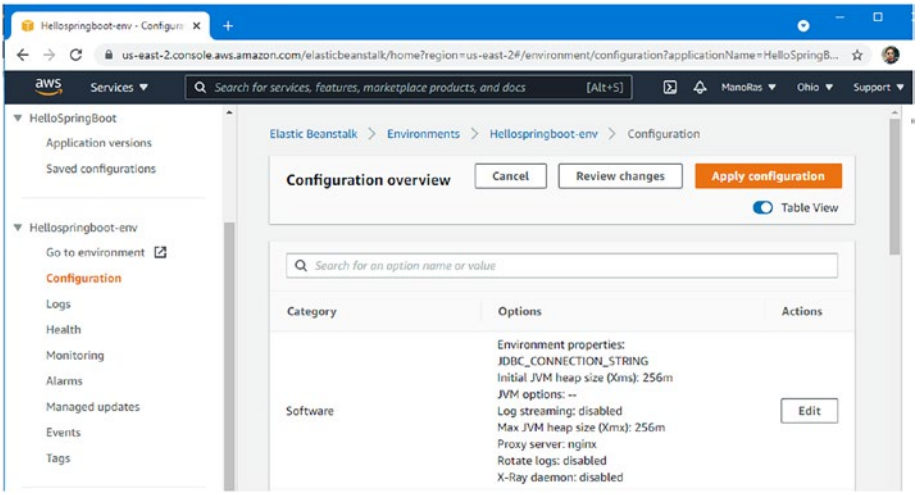


Figure 2-21. *Spring Boot application severe health*

In the Environment properties, you see that there are already some environment variables set by Elastic Beanstalk when it is configured to use the Java platform.

To change the port that the Spring Boot application listens on, add a new environment variable, SERVER_PORT, with a value of 5000, as shown in Figure 2-22.

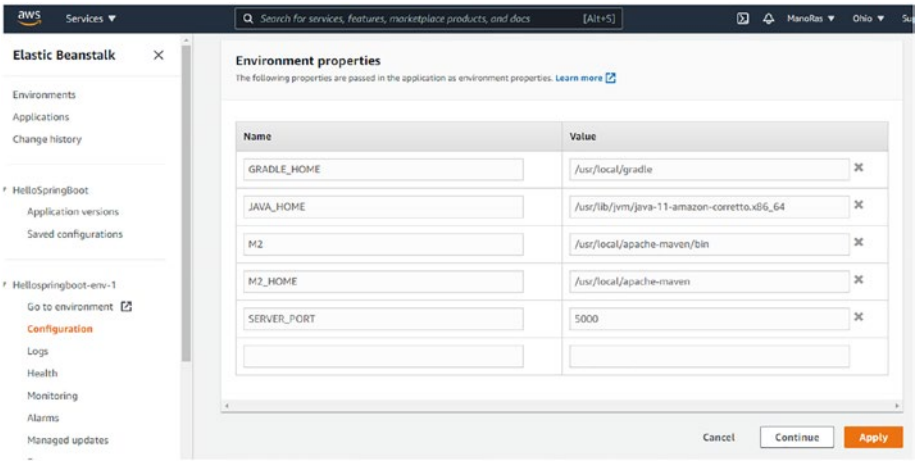


Figure 2-22. *Environment properties on software configuration*

As soon as you click Apply, the configuration change propagates to the application servers. The application restarts. When it restarts, it picks up the new configuration through the environment variables. After a minute, you see a healthy application on the dashboard, as shown in Figure 2-23.

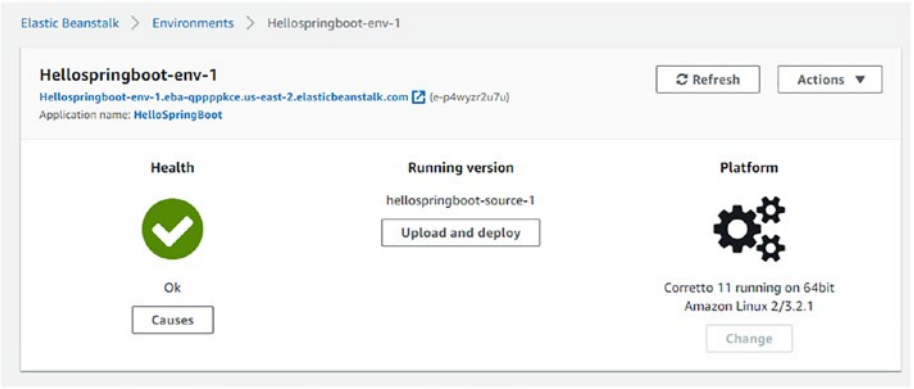


Figure 2-23. *Health OK*

Test a Spring Boot Application as a REST API in the Cloud

Now, let's test the deployed REST API endpoint in AWS. Use the URL you configured in the environment to access the service. For this example, the specified URL is `http://hellospringboot-env-1.eba-qppppkce.us-east-2.elasticbeanstalk.com`.

For the first test, from the browser, use an HTTP GET on the greeting URI at `http://hellospringboot-env-1.eba-qppppkce.us-east-2.elasticbeanstalk.com/greeting`. The service responds with a welcome greeting message, as shown in Figure 2-24.

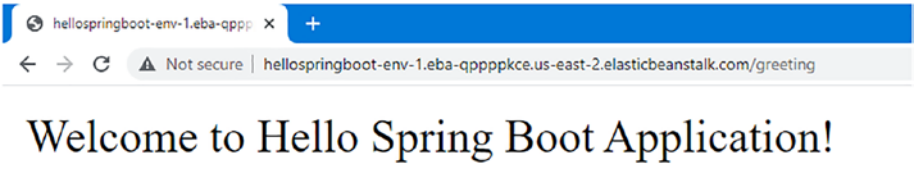


Figure 2-24. *Accessing REST API deployed on the cloud from browser*

Next, access the Swagger UI dashboard at `http://hellospringboot-env-1.eba-qppppkce.us-east-2.elasticbeanstalk.com/swagger-ui/` from your browser, as shown in Figure 2-25.

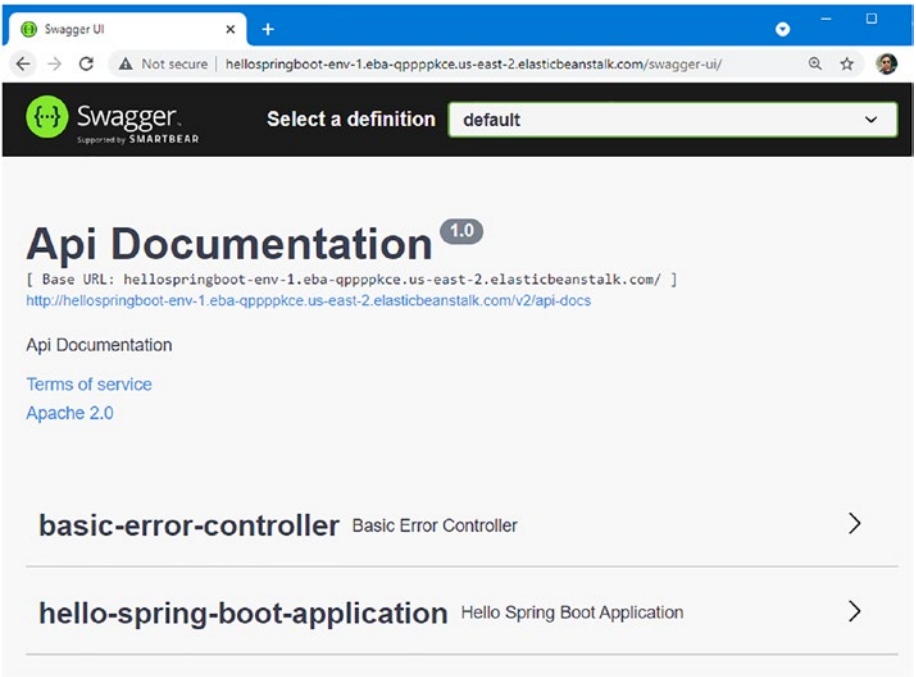


Figure 2-25. *Accessing Swagger-UI dashboard from browser*

Explore Logs from Elastic Beanstalk

You can explore the Spring Boot logs from Elastic Beanstalk. Select **Logs ➤ Request** Logs to retrieve the last 100 lines of a log or the entire set of logs from each EC2 instance, as shown in Figure 2-26.

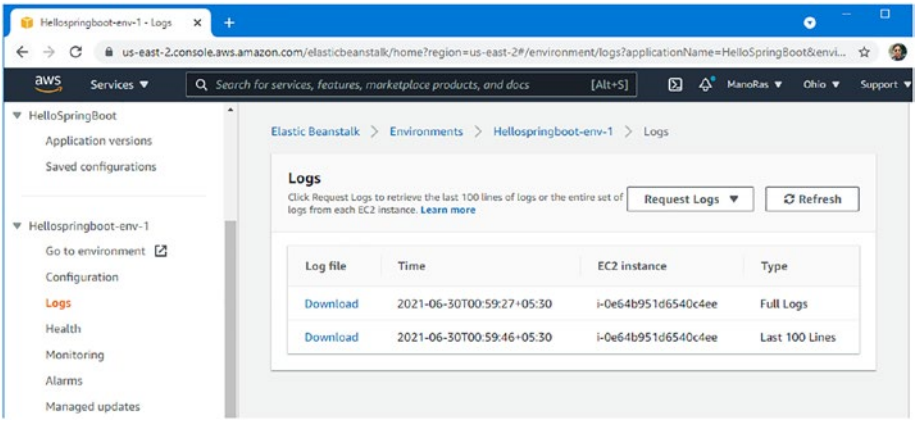


Figure 2-26. *AWS Elastic Beanstalk logs*

Once you click Download, you see that the entire Spring Boot log is visible.

Summary

In this chapter, you deployed a REST API to Elastic Beanstalk. You created a Spring Boot project application as a REST API and then generated a JAR file for the project. You deployed this JAR to Elastic Beanstalk, resolved server issues in AWS. And finally, you accessed the deployed application in the cloud.

In the next chapter, you deploy a MySQL database in AWS with RDS.

CHAPTER 3

Deploy MySQL as a Database in AWS with RDS

In Chapter 2, you deployed the REST API to AWS Elastic Beanstalk. You created a Spring Boot application as REST API, and then you generated a JAR file of our project. You were able to deploy the JAR file to Elastic Beanstalk and resolved the server issue on AWS to make the application. And finally, you were able to access the application deployed on the AWS cloud.

Amazon RDS makes it easy to set up and operate a MySQL database and easy to scale MySQL deployment in the Amazon cloud. Self-managing a database offers a lot of challenges and takes upkeep. This chapter introduces Amazon Relational Database Service (RDS), and you learn how to deploy it.

If you look at the application architecture from Chapter 2, Elastic Beanstalk is where our Java-based Spring Boot application was deployed. Now let's use the Amazon RDS, which is a database in the cloud. MySQL runs on AWS. An instance of a MySQL database is created and configured in AWS. Tables are also created in the MySQL database.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0_3

Introduction to Amazon RDS (Amazon Relational Database Service)

Data can be understood as a collection of the distinct unit of information that can be translated into a required form for efficient movement and processing. A database can be defined as an organized collection of structured data so that it can be easily accessed, managed, and updated. In simple words, a database is where the data is stored.

Amazon RDS is a web service that allows you to quickly deploy and scale a relational database on the Amazon cloud. Once you have deployed your database, you can manage it using a normal admin tool like MySQL Workbench, Oracle SQL Developer, or another admin tool. AWS also supports NoSQL databases like MongoDB.

For more information on Amazon RDS, refer to `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`.

Create an Instance of the RDS Database in AWS

Let's begin configuring the RDS MySQL environment by signing up on AWS Management Console. Select RDS under the Database section in All Services, as shown in Figure 3-1.

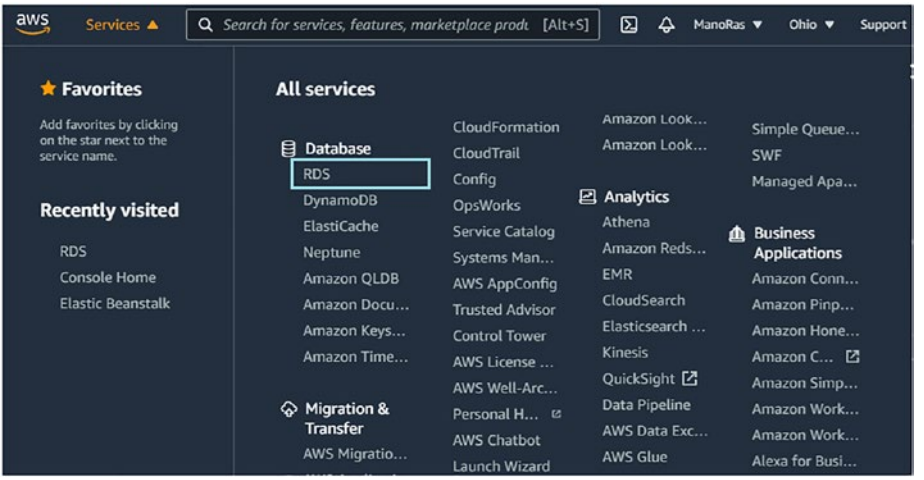


Figure 3-1. *RDS under Database section in All Services*

You are redirected to the Amazon RDS dashboard page, as shown in Figure 3-2. This page gives information about the resources you are using. Let's create an instance of Amazon Relational Database by clicking the **Create database** button.

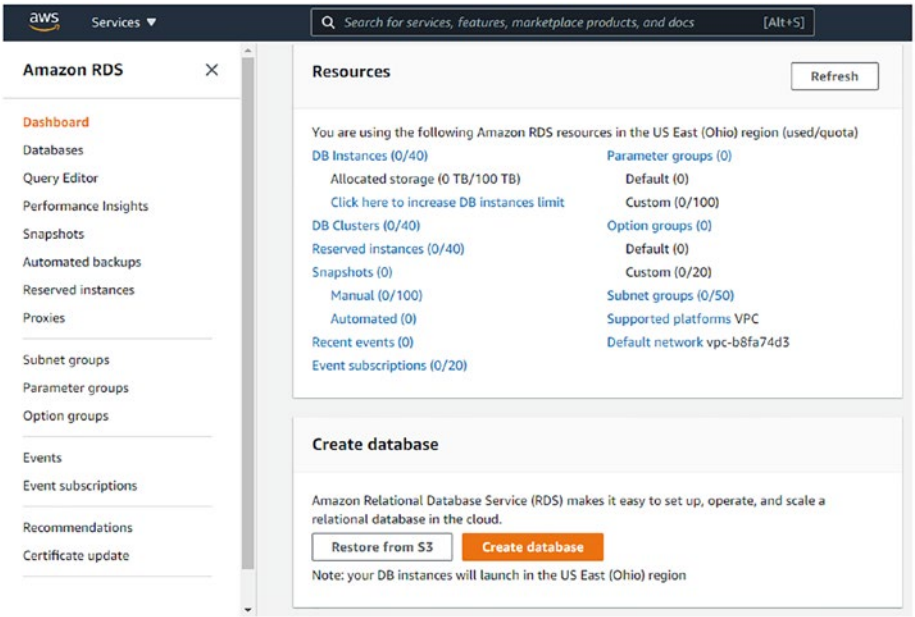


Figure 3-2. *Amazon RDS dashboard*

A new page opens, where you can define the database creation method and other options. Let's start creating a database.

First, select the database engine from the **Engine options**, as shown in Figure 3-3. There are a lot of options available, but let's use the MySQL database engine. MySQL is a widely-used open source relational database management system. MySQL is mostly used for web databases.

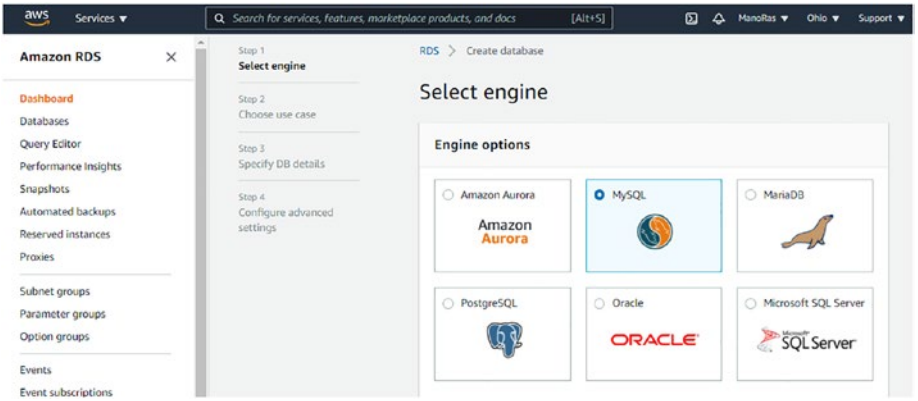


Figure 3-3. *Engine options to select*

As shown in Figure 3-4, click the check box to only enable options for the RDS Free Usage Tier, which allows you to use a database for free in the AWS cloud. And then click the Next button.

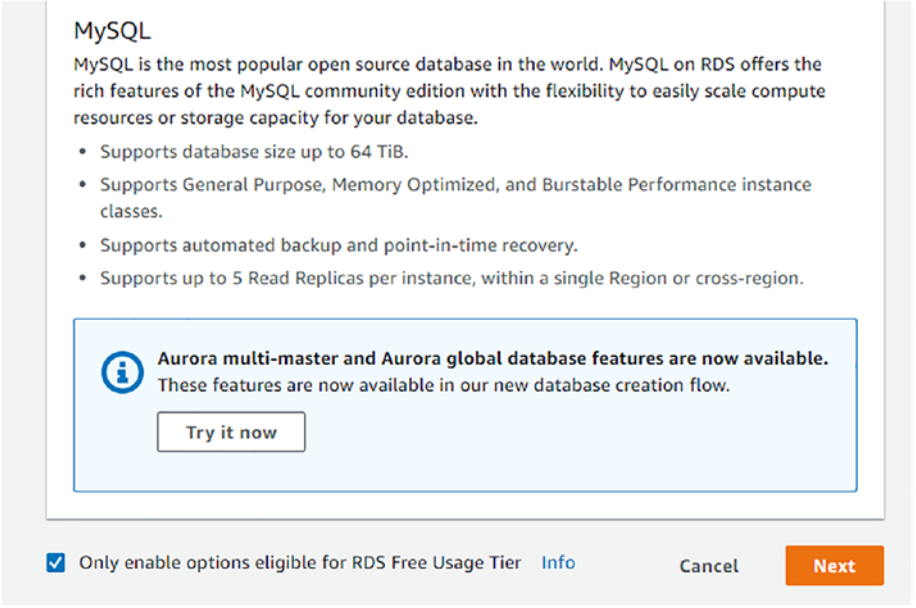


Figure 3-4. *RDS Free Usage Tier*

Next, specify the database details, as shown in following Figure 3-5.

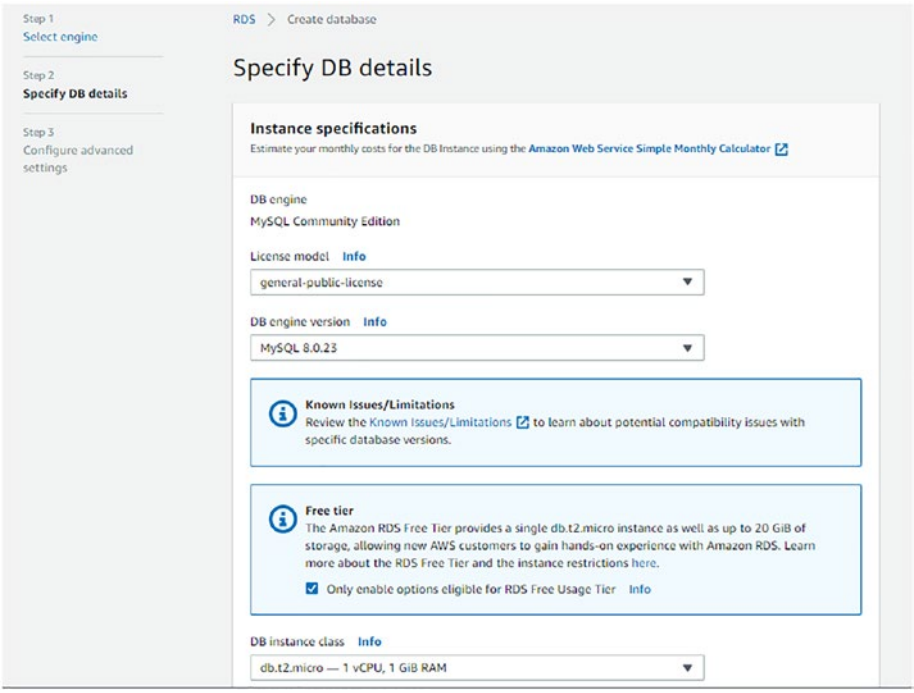


Figure 3-5. *Specify DB details*

Keep the defaults for the license model and DB engine version. Check the box to only enable the option for the RDS free tier. In the DB instance class, keep the default selected value, `db.t2.micro`, for the free tier.

The database instance identifier is a unique name that you create to find or reference a database instance. Next, provide a suitable name for the database; let's use `spring-aws-db`, as shown in Figure 3-6.

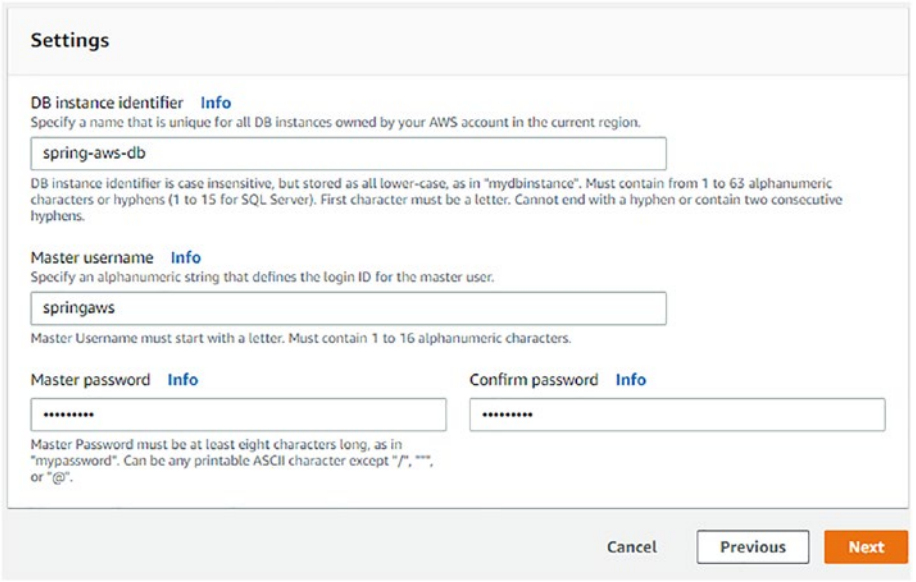


Figure 3-6. *Setting database details*

Similarly, provide the master username and password. We used `springaws` for both to keep things simple, but you can use any value you want. You can connect to the MySQL instance using this username and password later, so keep these credentials safe. And then click the Next button.

Finally, you need to configure some advanced settings that are essential to setting up an RDS MySQL environment, as shown in Figure 3-7.

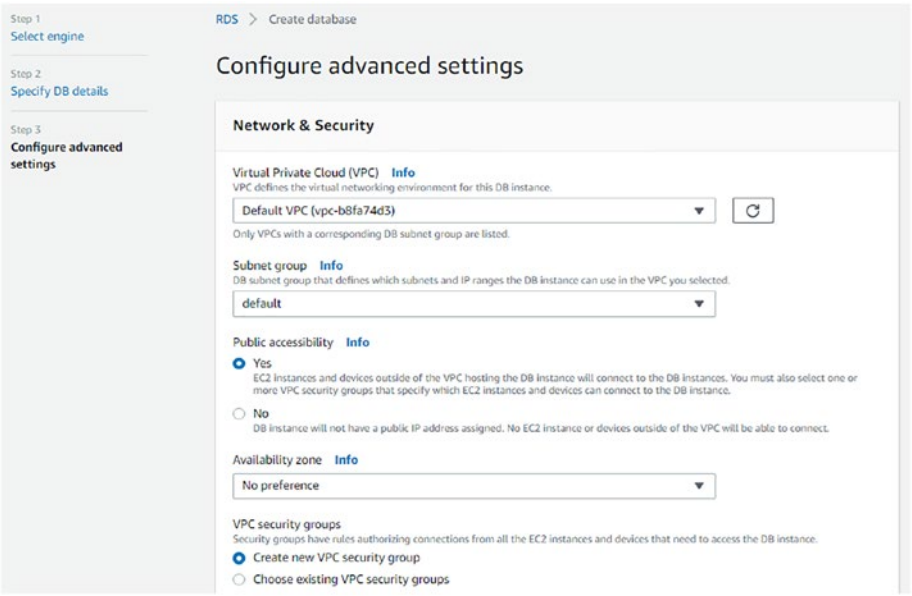


Figure 3-7. *Configure advanced settings*

Keep all the defaults in the Network & Security section. Make sure the public accessibility of the DB instance is Yes. This allows the database instance to be available on the Internet and connect with other hosts.

Next, the database options include the name, port, and so on, as shown in Figure 3-8. Keep all the defaults as they are. The port number is 3306, which is the default port. Other options are also available.

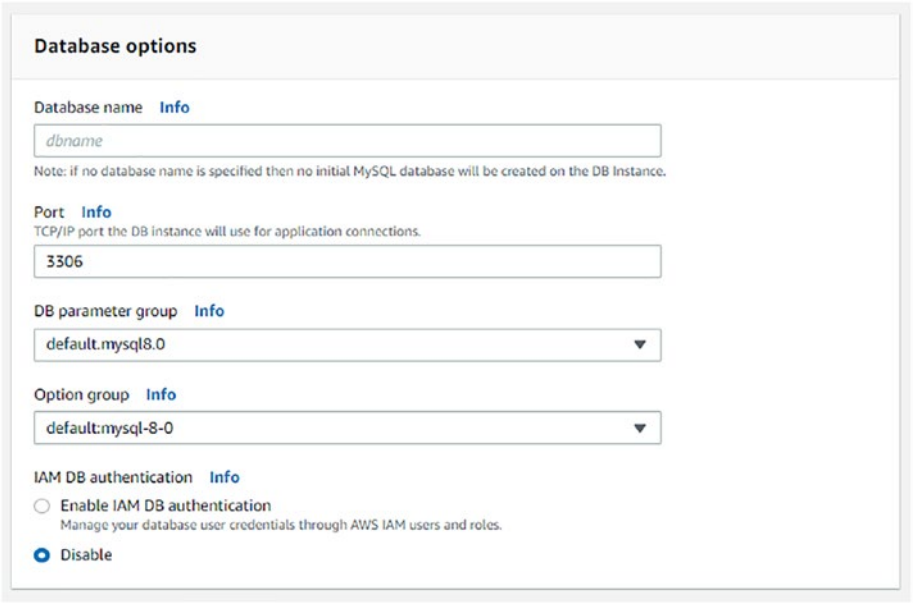


Figure 3-8. *Database options*

Click the **Create database** button to launch the Amazon RDS database instance, as shown in Figure 3-9.

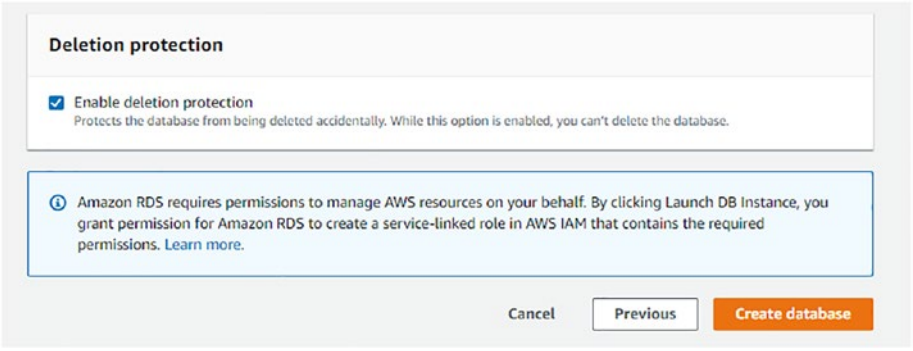


Figure 3-9. *Launch the Amazon RDS database instance*

You see that your database instance is being created, as shown in Figure 3-10.

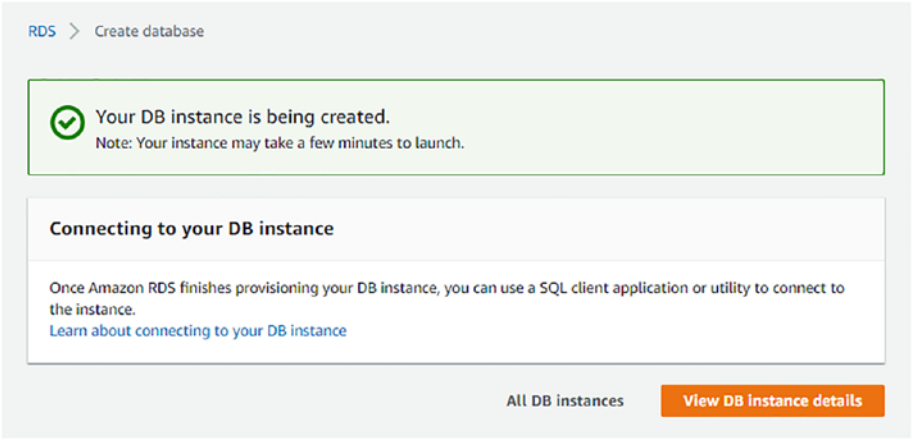


Figure 3-10. *Amazon RDS DB instance creation status*

Your DB instance normally takes a few minutes to launch.

Configure Amazon RDS

The current status shows that you have a database instance available in the AWS cloud, which you created as an instance of the RDS database server. Unfortunately, this database instance is empty because there's no database schema, tables, or data available in the RDS database instance.

You need to do some configuration work for the relational database service, connect it to MySQL Workbench, and access it. As a development process, the first thing is to configure security for inbound connection rules. And, then you need to test the database connectivity with MySQL Workbench.

Before going ahead, let's check the Amazon RDS database instance status. Click **Databases** under Amazon RDS, as shown in Figure 3-11.

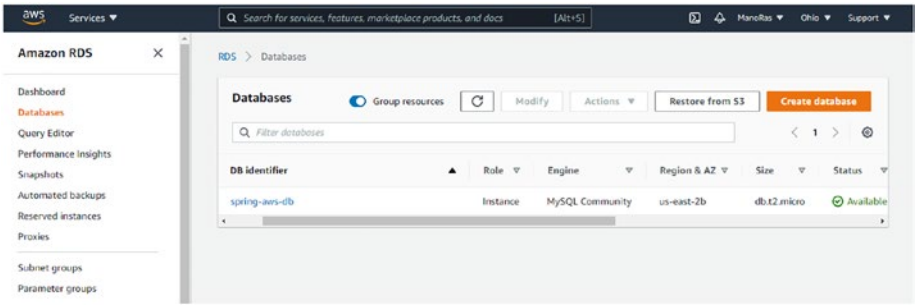


Figure 3-11. *Database instance status*

Here, you can see that the database instance `spring-aws-db` is added to the list. Its status is available, which indicates that the database has been created and is available for use. Click the `spring-aws-db` link in the Databases table. Figure 3-12 shows the summary.

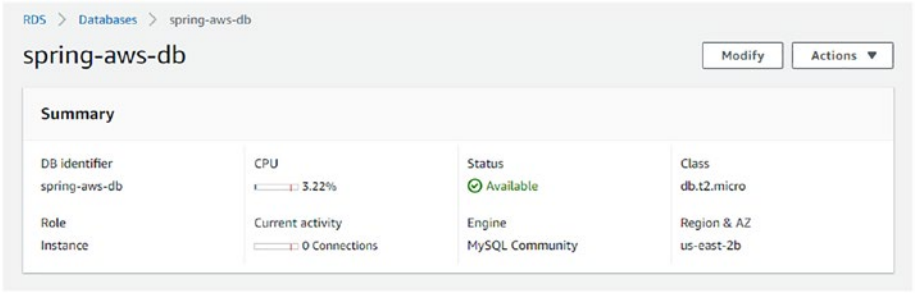


Figure 3-12. *Amazon RDS database instance summary*

Here, you can see the information on the `spring-aws-db` database instance. The class is `db.t2.micro`, the engine is MySQL Community, and the status is available.

Step 1. Configure Security for Inbound Connection Rules

First, you need to configure the security group rules for the inbound connection rules. Scroll down to the **Security group rules** section, as shown in Figure 3-13.

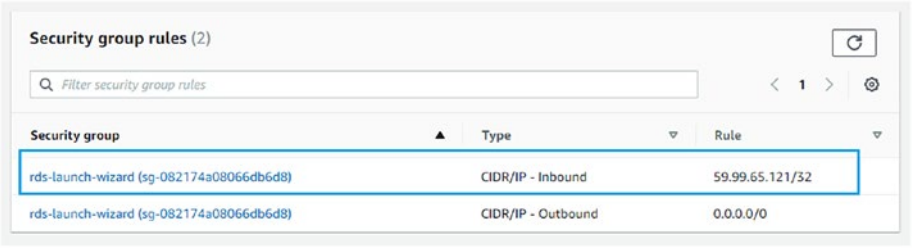


Figure 3-13. *Security group rules*

The inbound rule defines the traffic allowed on the server and who can connect to the database instance. Click `rds-launch-wizard` for CIDR/IP - Inbound, which redirects you to the Security Groups Info page, as shown in Figure 3-14.

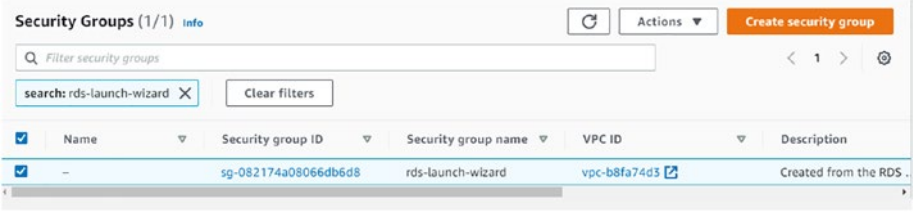


Figure 3-14. *Security groups*

At the bottom of this page, you see tabs named Details, Inbound rules, Outbound rules, and Tags, as shown in Figure 3-15.

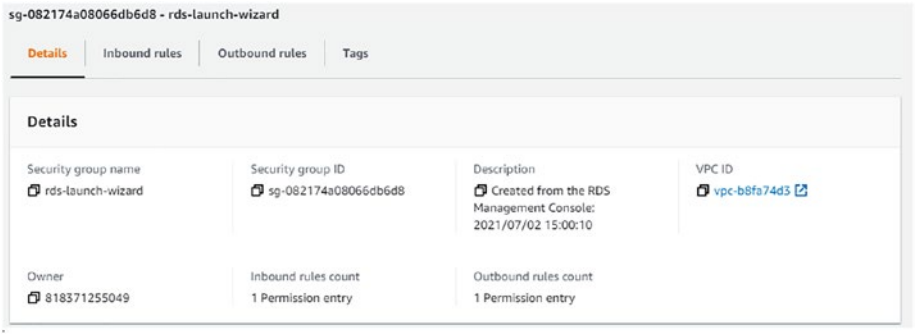


Figure 3-15. *rds-launch-wizard*

Click the **Inbound rules** tab, as shown in Figure 3-16.

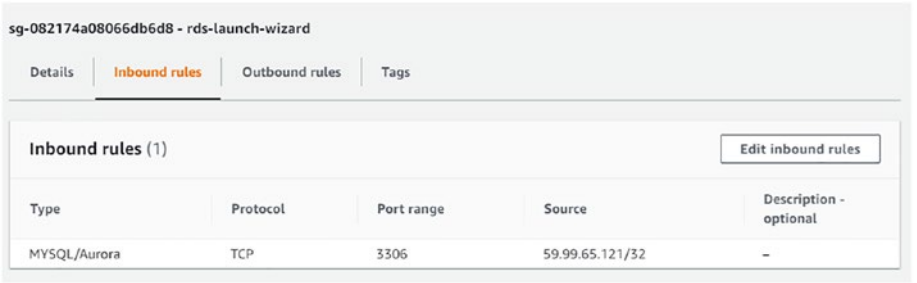


Figure 3-16. *Inbound rules*

You see that the database is only accessible from the IP address 59.99.65.121/32. You need to make some modifications here. Click the **Edit inbound rules** button, which redirects to the **Edit inbound rules** page. Here you can edit the IP address that has access to the Amazon RDS MySQL database instance, as shown in Figure 3-17.

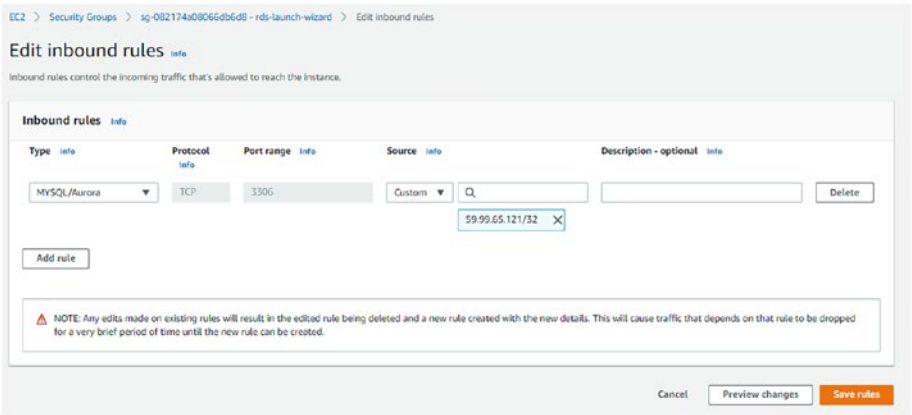


Figure 3-17. *Edit inbound rules*

You can determine the traffic that can reach the database instance. From the Source drop-down list, select the Anywhere option, as shown in Figure 3-18.

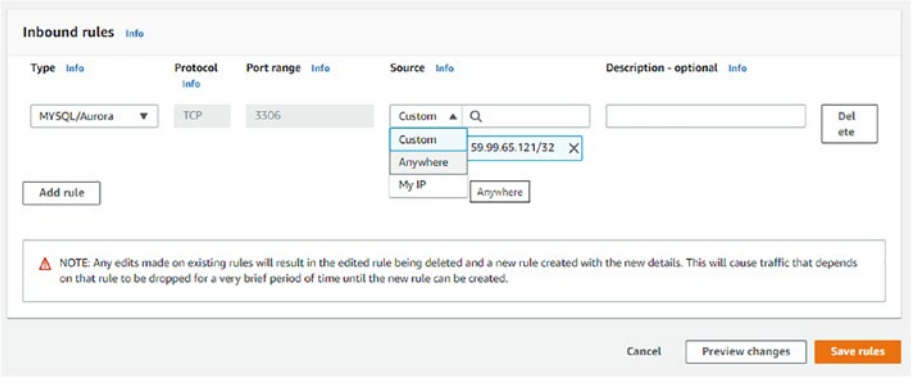


Figure 3-18. *Select Anywhere from Source drop-down list*

Now, anyone can find the database instance or connect to it, but they still have to provide a correct user ID and password. The Anywhere source option is good for dev and testing, but it is recommended to only allow access from the Elastic Beanstalk app IP address for production.

Click the **Save rules** button. Now you can see that the inbound rule has been set up, as shown in Figure 3-19.

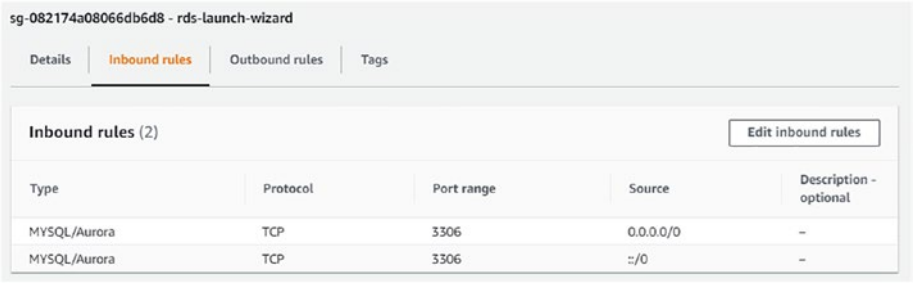


Figure 3-19. *Updated source in Inbound rules*

These updated Inbound rules allow connection from anywhere.

Step 2. Test an Amazon RDS Database Instance Connection with MySQL Workbench

Once you have successfully created the Amazon RDS MySQL database instance and all the necessary configurations are done, the second step is to test the RDS database instance connection with MySQL Workbench.

Return to the previous page in the browser. In the Databases section, click the **Connectivity & security** tab, as shown in Figure 3-20.

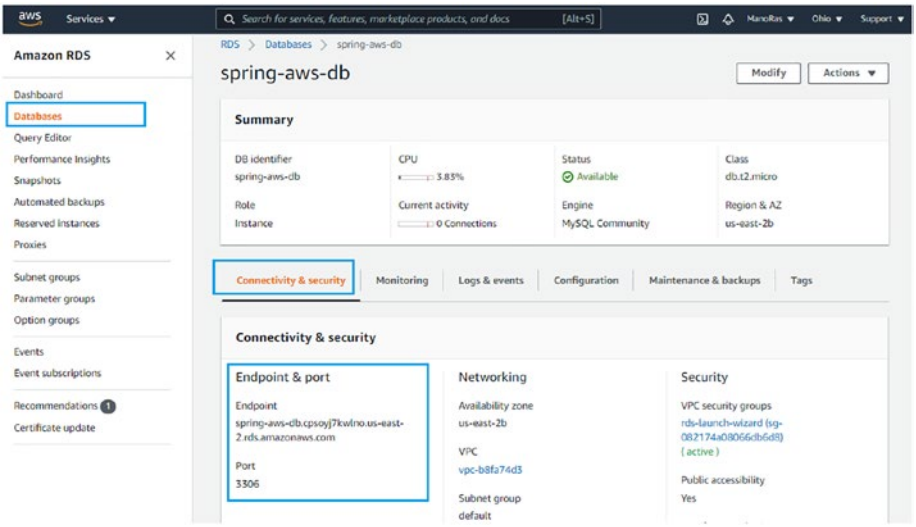


Figure 3-20. *Updated source in Inbound rules*

In the Connectivity & Security tab, there is a section called **Endpoint & port**. The endpoint indicates the hostname of the database instance, which you can use in MySQL Workbench to connect to the RDS database instance. In this case, it is

```
spring-aws-db.cpsoyj7kwlno.us-east-2.rds.amazonaws.com
```

Note Refer to Appendix A for the MySQL Workbench installation guide.

Connect MySQL Workbench to an Amazon RDS MySQL Database Instance

Open MySQL Workbench in your local system. Then, click the + icon to create a MySQL connection, as shown in Figure 3-21.

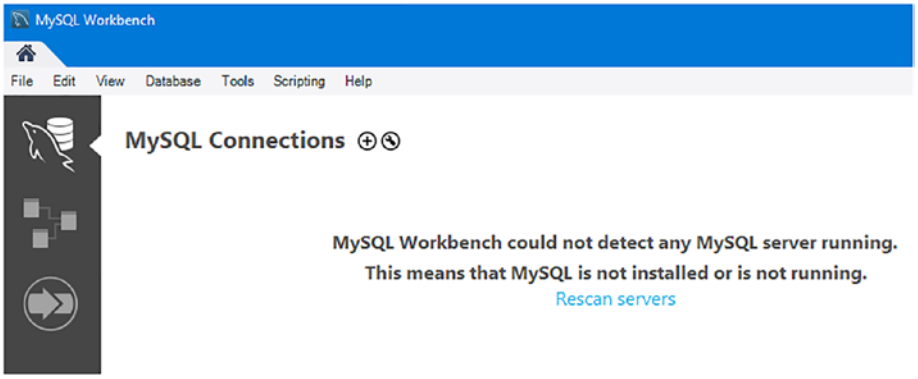


Figure 3-21. *MySQL Workbench*

This opens the Setup New Connection wizard, as shown in Figure 3-22.

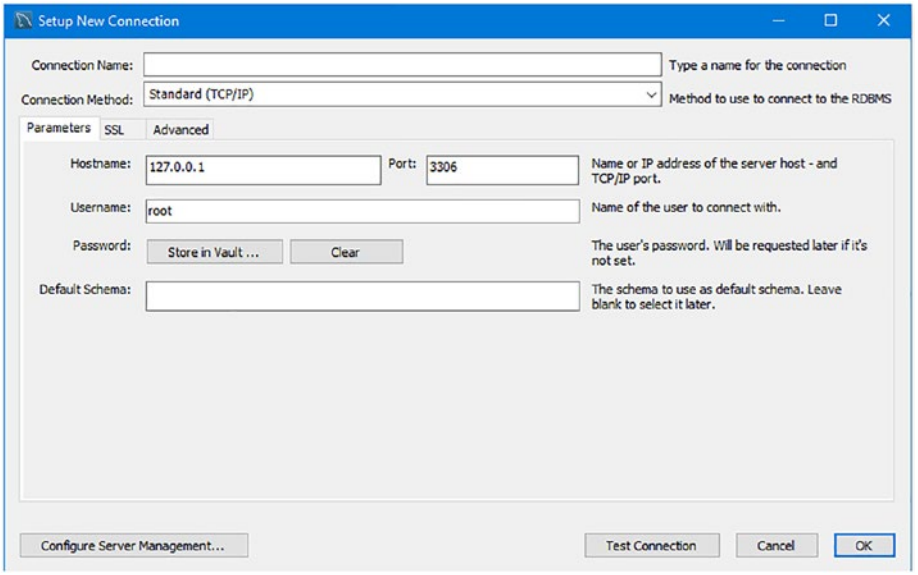


Figure 3-22. *Setup New Connection wizard*

For the connection name, enter the value as **spring-aws-db**. In the hostname field, the default value is 127.0.0.1, which is known as the localhost. Replace the default IP address with the following RDS database instance hostname from AWS Management Console.

```
spring-aws-db.cpsoyj7kwlno.us-east-2.rds.amazonaws.com
```

Leave the port number as it is because 3306 is the port for the database instance from the AWS console.

Use the same username and password that you created for the RDS database instance. So, enter **springaws** as the username, as shown in Figure 3-23. Click the **Store in Vault** button for password.

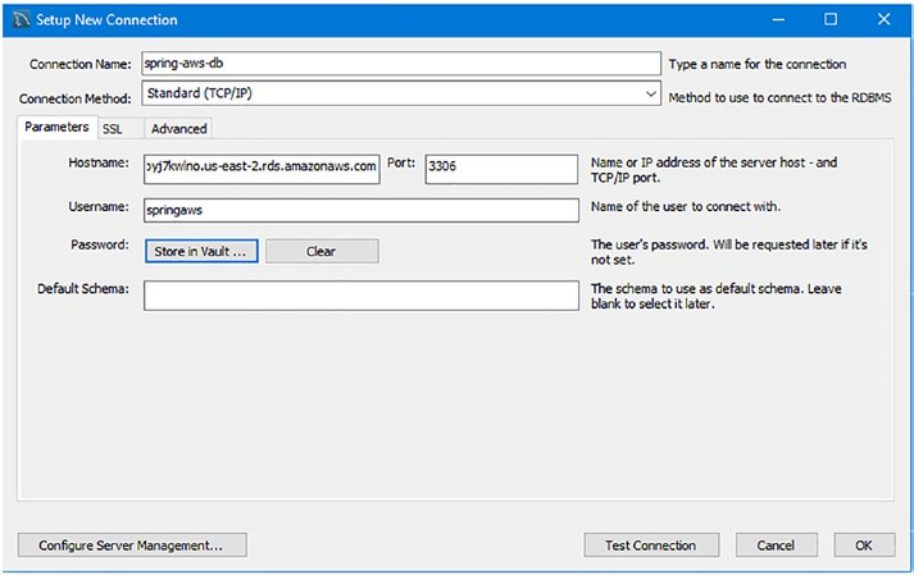


Figure 3-23. *Updated value in Setup New Connection wizard*

Enter **springaws**, and then click OK, as shown in Figure 3-24.

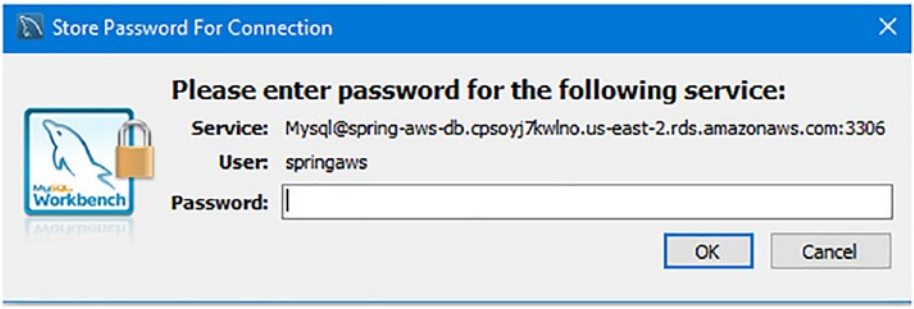


Figure 3-24. *Store password for connection*

Click the Test Connection button. You should receive a notification saying you have successfully made the MySQL connection, as shown in Figure 3-25.

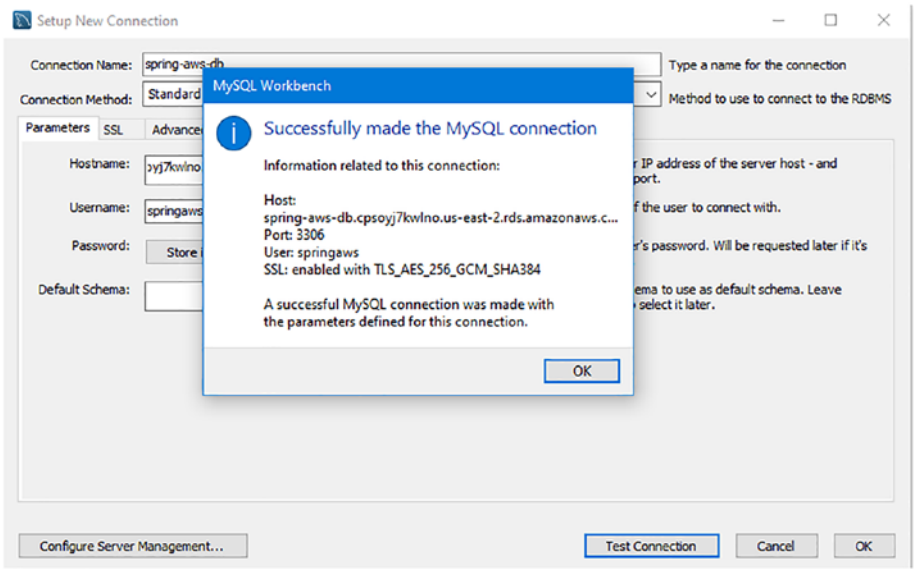


Figure 3-25. *Successfully made the MySQL connection*

That's how you know that the database instance is available and running in the AWS cloud. You can use your local MySQL Workbench to connect to it. Click the OK button in the Connection wizard, which lets Workbench list the database connection details, as shown in Figure 3-26.

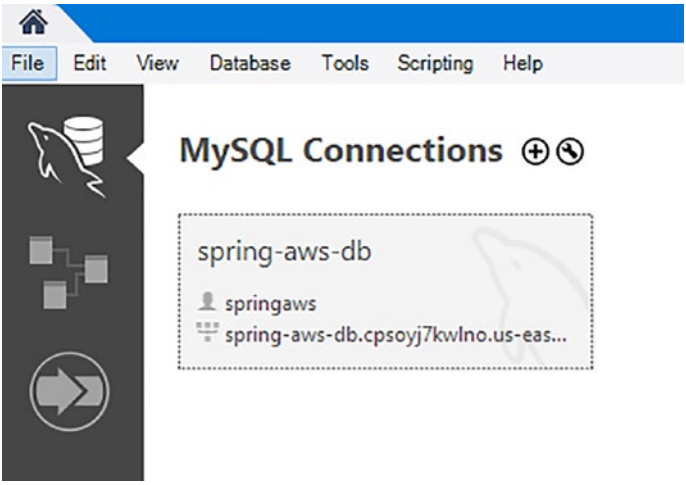


Figure 3-26. *MySQL Workbench with Amazon RDS db connection details*

Create a Table Inside an RDS Database Instance

MySQL is set up correctly. You can access the remote RDS database instance by clicking `spring-aws-db`, which opens in the SQL editor, as shown in Figure 3-27.

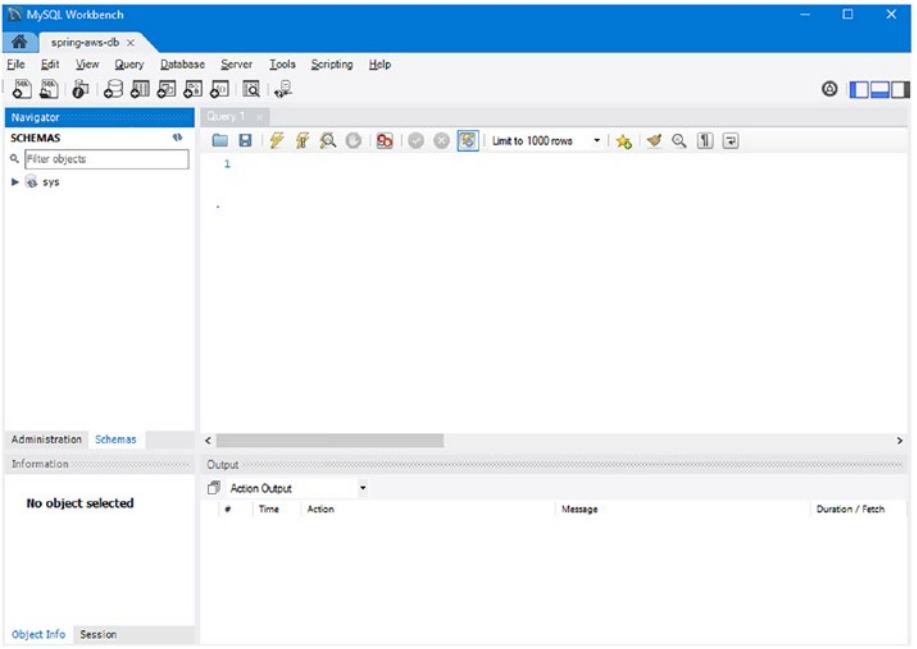


Figure 3-27. *SQL editor instance for* `spring-aws-db`

Currently, there is no database, table, or data available for our RDS database instance. You need to run some scripts to provide anything that you can query.

First, let's create a database using the `CREATE DATABASE` command. The syntax to create a new database is `CREATE DATABASE DB_NAME`, where `DB_NAME` is the database name that you want to create. For example, to create a database named `UserRegistration`, type the following query into the Query tab and run it.

```
CREATE DATABASE UserRegistration;
```

Once the query is executed successfully, the Schema tab should display the `UserRegistration` database, as shown in Figure 3-28.

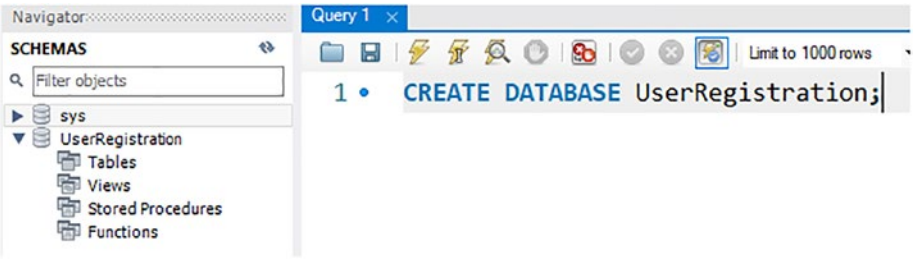


Figure 3-28. *Database created*

Now, let's create a user table in the `UserRegistration` database. A table displays and stores the records in a structured format. The `CREATE TABLE` command creates a new table into the existing database. The syntax to create a MySQL table is shown in Listing 3-1.

Listing 3-1. Syntax to Create MySQL Table

```
CREATE TABLE [IF NOT EXISTS] table_name(
    column_1 datatype(size) [NULL | NOT NULL],
    column_2 datatype(size) [NULL | NOT NULL],
    column_3 datatype(size) [NULL | NOT NULL],
    ........,
    column_N datatype(size) [NULL | NOT NULL],
    table_constraints
);
```

`table_name` is the name of the table, which should always be unique in a MySQL database. The `IF NOT EXISTS` clause helps prevent errors when the same table name already exists in the database.

`column_` specifies the column name. `datatype` specifies the type of data for that column, and columns are separated using a comma operator.

`table_constraints` specifies the table's constraints, such as primary key, foreign key, and unique key. For example, to create a table called `users`, enter the query shown in Listing 3-2 in the Query tab and run it.

Listing 3-2. Create Users Table in UserRegistration Database

```
use UserRegistration;

CREATE TABLE [IF NOT EXISTS]  users(
    id int NOT NULL AUTO_INCREMENT,
    first_name varchar(45) NOT NULL,
    last_name varchar(45) NOT NULL,
    address varchar(35) NOT NULL,
    age int NOT NULL,
    created_date DATE,
    PRIMARY KEY (id)
);
```

Here, the `use UserRegistration` command selects the database under which the table is created. Once the query is executed successfully, the `UserRegistration` database should display the `users` table, as shown in Figure 3-29.

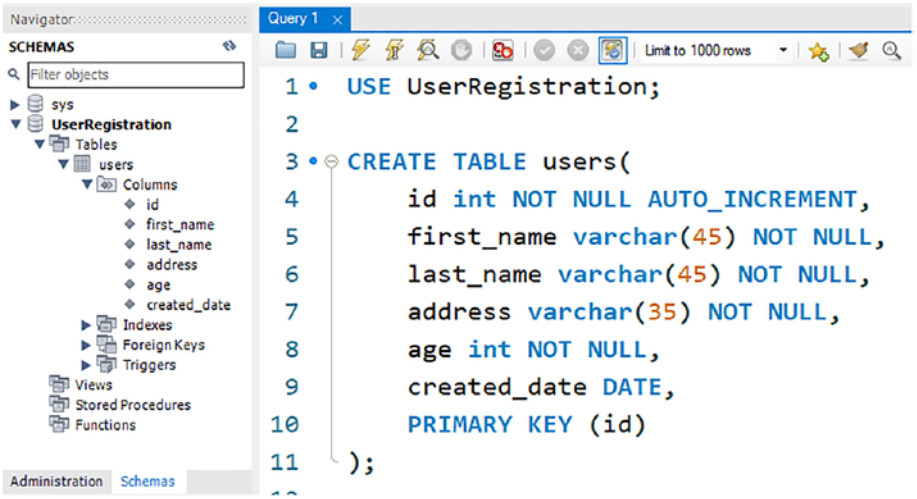


Figure 3-29. *Table created*

Now, let's insert some data into the `users` table. The `INSERT INTO` command adds or stores data in a table. The syntax to insert data into a table is shown in Listing 3-3.

Listing 3-3. Syntax to Insert Data into the Table

```
INSERT INTO DATABASE.table_name (column_1, column_2,... column_N)
VALUES
( value_1, value_2,...value_N );
```

First, specify the database name followed by a dot (.), followed by the table name, and then a list of comma-separated columns. Next, provide the list of values corresponding to the column's name after the `VALUES` clause. For example, to insert data in the `users` table, type the query shown in Listing 3-4 in the Query tab, and then run it.

Listing 3-4. Insert Data in users Table in UserRegistration Database

```
INSERT INTO UserRegistration.users (first_name, last_name,
address, age, created_date)
VALUES
('Ravi', 'Soni', 'Sasaram-Bihar-India', 34, '2021-07-04');
```

The default date format in MySQL is YYYY-MM-DD, where YYYY represents the year in four digits, MM represents the month in two digits, and DD represents the day in two digits.

Once the insert query is executed successfully, you can use the `SELECT` command to fetch data from the MySQL database. The data returned from the database is stored in a result table, called `result-set`. The `SELECT` command syntax to fetch data from a MySQL table is shown in Listing 3-5.

Listing 3-5. Syntax of SELECT Command to Fetch Data from Database

```
SELECT column_1, column_2, ...
FROM
DATABASE.table_name;
```

For example, to fetch data from the `users` table, type the query shown in Listing 3-6 into the Query tab, and then run it.

Listing 3-6. Fetch Data from UserRegistration Database

```
SELECT first_name, last_name, address, age, created_date
FROM
UserRegistration.users;
```

Once the `SELECT` query is executed successfully, the result appears as shown in Figure 3-30.

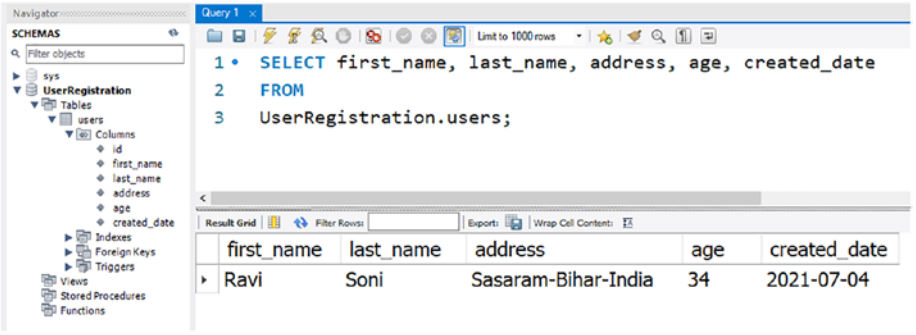


Figure 3-30. *Table created*

Summary

This chapter introduced Amazon RDS. First, you created a MySQL database instance in AWS and configured the database. Then, you created a table in the database and inserted data into it using MySQL Workbench.

The next chapter overviews CRUD operations in a Spring Boot application, and you deploy Spring Boot application that talks to MySQL in AWS.

CHAPTER 4

Deploy a Spring Boot Application Talking to MySQL in AWS

Chapter 3 introduced Amazon RDS, and you learned how to deploy it on the Amazon cloud. You created an instance of an Amazon RDS MySQL database in AWS and configured the database. You also created tables in this database and inserted data into it using MySQL Workbench.

In Chapter 2, you created and deployed a Spring Boot REST API containing some endpoints to AWS Elastic Beanstalk. However, that's not how real applications run. The real application uses a real-time database to perform CRUD operations.

This chapter creates a Spring Boot application as a REST API talking to an Amazon RDS MySQL database from your local system.

Create Spring Boot UserRegistrationApp Talking to MySQL Database

In this section, you create the `UserRegistrationApp` Spring Boot application using Spring Initializr (`http://start.spring.io/`). Here, you select Web, JPA, MySQL, and Lombok as dependencies, as shown in Figure 4-1.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0_4

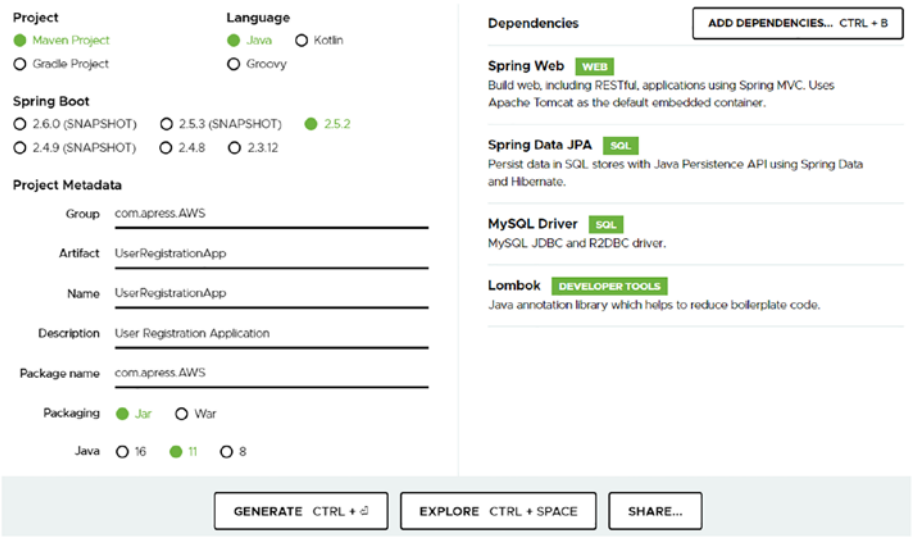


Figure 4-1. *Creating UserRegistrationApp using Spring Initializr*

Table 4-1 lists all the `UserRegistrationApp` settings.

Table 4-1. *Project-Related Details*

Field	Value
Group	com.apress.AWS
Artifact	UserRegistrationApp
Name	UserRegistrationApp
Description	User registration application
Package Name	com.apress.AWS
Packaging	JAR
Java Version	11
Language	Java
Project	Maven

After entering the project metadata, click the Generate button to download the `UserRegistrationApp.zip` file. Unzip it, and import it as a Maven project into the Spring Source Tool (STS) IDE. The initial project structure looks like what's shown in Figure 4-2.

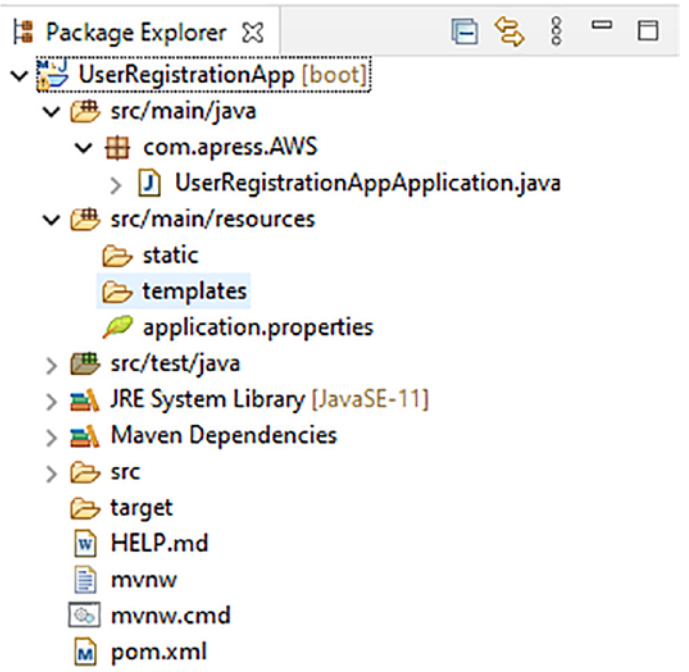


Figure 4-2. *Project structure*

Let's walk through the code for more information and explore Maven dependencies defined in `pom.xml`.

Maven Dependency in pom.xml

All the required dependencies you selected in Spring Initializr when creating the Spring Boot application are available in `pom.xml`, as shown in Listing 4-1. The `pom.xml` file is the recipe that builds the Spring Boot application.

Listing 4-1. pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      https://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.5.2</version>
            <relativePath/>
      </parent>
      <groupId>com.apress.AWS</groupId>
      <artifactId>UserRegistrationApp</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <name>UserRegistrationApp</name>
      <description>User Registration Application</description>
      <properties>
            <java.version>11</java.version>
      </properties>
      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-web
                  </artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-data-jpa
                  </artifactId>
            </dependency>
```

```
        <dependency>
              <groupId>mysql</groupId>
              <artifactId>mysql-connector-java</artifactId>
              <scope>runtime</scope>
        </dependency>
        <dependency>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
              <optional>true</optional>
        </dependency>
        <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-test
              </artifactId>
              <scope>test</scope>
        </dependency>
</dependencies>
<build>
        <plugins>
              <plugin>
                    <groupId>org.springframework.boot
                    </groupId>
                    <artifactId>spring-boot-maven-plugin
                    </artifactId>
                    <configuration>
                          <excludes>
                                <exclude>
                                      <groupId>org.project
                                      lombok</groupId>
                                      <artifactId>lombok
                                      </artifactId>
                                </exclude>
```

```
                        </excludes>
                    </configuration>
                </plugin>
            </plugins>
        </build>
</project>
```

Also, update `pom.xml` with a Springfox dependency for the Swagger UI, as shown in Listing 4-2.

Listing 4-2. Add Springfox Dependency in pom.xml

```
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-boot-starter</artifactId>
        <version>3.0.0</version>
</dependency>
```

Project Lombok

You selected Lombok dependency while creating the project. Let's look at the main objective of Project Lombok. "Project Lombok is a small Java library that plugs into your IDE like Eclipse, IntelliJ, STS, etc. Also, it can plug into build tools like Maven, Ant, etc. [The] Lombok library reduces the amount of boilerplate Java code by [preventing you from writing] another getter, setter, toString, or equals method again. And this implementation is automatically done during compile time." (https://projectlombok.org)

Project Lombok automatically generates the `getter, setter, toString`, and `equals` method for the object by using the `@Data` Lombok. The following are the steps to plug in the Lombok Java library to the STS IDE.

1. For the STS IDE, get the Lombok executable JAR file.

2. Do Maven build in the Spring Boot project. Figure 4-3 shows the Lombok JAR is at `\.m2\repository\org\projectlombok\lombok\1.18.20\`.

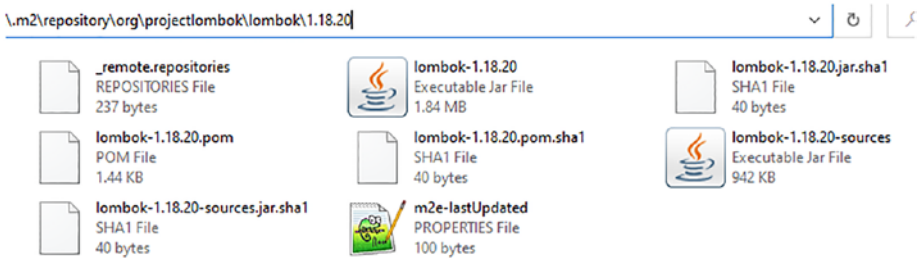


Figure 4-3. *Lombok JAR file under .m2 directory*

3. Double-click `Lombok.jar` to open the installer UI. Specify the location of the STS.exe path, and then click the Install/Update button, as shown in Figure 4-4.

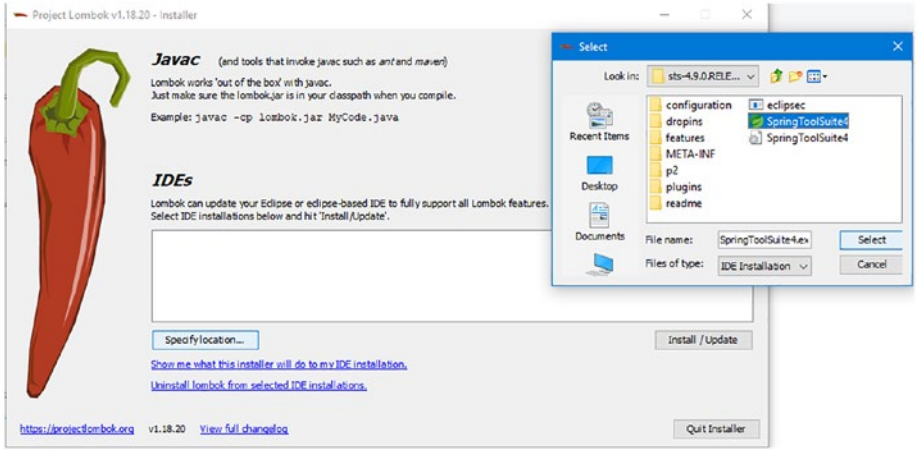


Figure 4-4. *Lombok Installer UI*

4. You should see a "Install successful" message, as shown in Figure 4-5. Click the Quit Installer `button` to exit the installer.

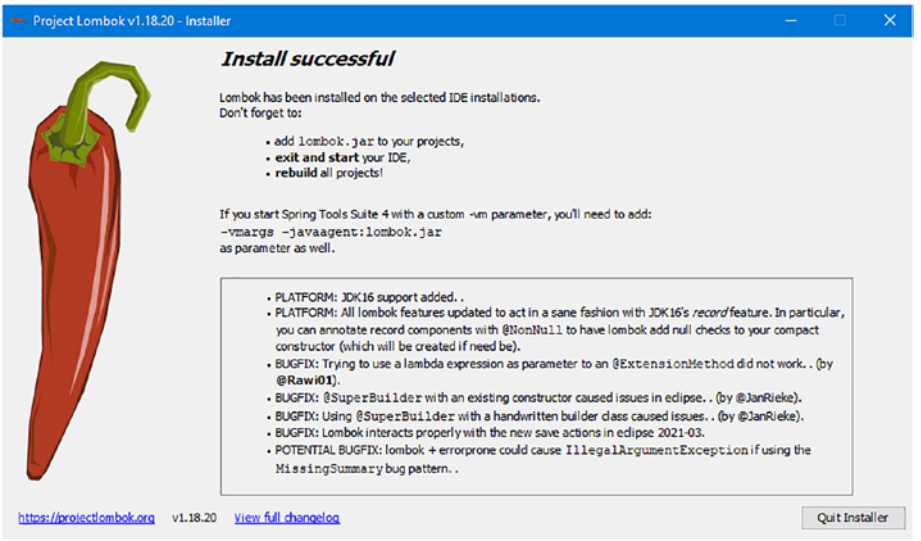


Figure 4-5. *Lombok installation success*

5. Restart the STS IDE to ensure that Lombok is correctly configured. Verify this in STS by going to the Help option and clicking the About option, as shown in Figure 4-6.

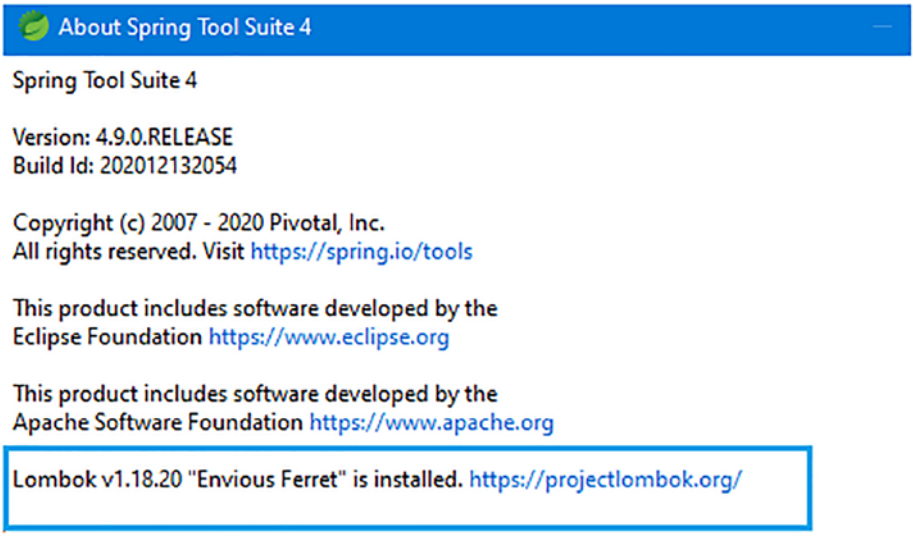


Figure 4-6. *Spring Tool Suite with Lombok details*

Application Properties

You need to configure how you can connect to the Amazon RDS MySQL database. In Chapter 3, you captured the MySQL database information, such as URL, username, and password, which you used in the MySQL Workbench connection with the Amazon RDS MySQL database instance.

Let's add code to the `/src/main/resources/application.properties` file, as shown in Listing 4-3.

Listing 4-3. /src/main/resources/application.properties

```
server.port=5000
# MySQL database settings
spring.datasource.url=jdbc:mysql://spring-aws-db.cpsoyj7kwlno.
us-east-2.rds.amazonaws.com:3306/UserRegistration
spring.datasource.username=springaws
spring.datasource.password=springaws
# db-creation settings
```

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Here, you configure the datasource URL, username, and the corresponding password that you want to connect to the MySQL database. `spring.jpa.hibernate.ddl-auto` can be none, update, create, or create-drop.

- `none` is the default for MySQL. It indicates that there are no changes made to the database structure.
- `update` instructs Hibernate to change the database according to the given entity structures.
- `create` instructs Hibernate to create the database every time the application restarts but does not drop it when `SessionFactory` closes.
- `create-drop` instructs Hibernate to create the database every time the application restarts and drops it when `SessionFactory` closes.

In the `application.properties` file, configure `ddl-auto = update` to make sure that whenever the application is restarted, Hibernate compares the tables in the database with the entities declared in the class. If there are any changes in the entity structure, those changes are updated in the database.

Domain Implementation: UserDTO Entity Class

In the `UserRegistrationApp` project, you create a DTO (data transfer object) class named `UserDTO` corresponding to the user domain's object inside a `com.apress.AWS.dto` subpackage. The `UserDTO` class contains only data. It transfers data between different layers of the application when there is a separation of concerns.

You can annotate the UserDTO class with JPA (Java Persistence API) annotations, which allow it to be easily persisted and retrieved using the JPA technology. A formal overview of JPA is beyond the scope of this book.

Let's implement the UserDTO entity class, as shown in Listing 4-4.

Listing 4-4. \src\main\java\com\apress\AWS\dto\UserDTO.java

```
package com.apress.AWS.dto;

import java.time.LocalDateTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Entity
@Table(name = "users")
@Data
public class UserDTO {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      @Column(name = "id", nullable = false)
      private Long id;
      @Column(name = "first_name")
      private String firstName;
      @Column(name = "last_Name")
      private String lastName;
      @Column(name = "address")
      private String address;
```

```
    @Column(name = "age")
    private Integer age;
    @Column(name = "created_date")
    private LocalDateTime createdDate;

}
```

The `UserDTO` class has six attributes: `id`, `firstName`, `lastName`, `address`, `age`, and `createdDate`. The `UserDTO` class is annotated with the `@Entity` annotation to make it a JPA entity. This entity class is also annotated with the `@Table` annotation to define the table name as `Users`. The `id` property in `UserDTO` is annotated with the `@Id` annotation to make it the primary key. The `id` attribute has been annotated with the `@GeneratedValue` annotation to indicate that the ID value should be generated automatically. The `id` attribute is annotated with the `@Column` annotation to specify the details of the column to which a field or property is mapped. The other five properties are annotated with the `@Column` annotation.

The `@Data` Lombok annotation is used, so you don't have to create a getter and setter for attributes, and at the compile, it is automatically generated. The next step is to provide the repository implementation.

Repository Implementation: UserJpaRepository

The Data Access Object (DAO) design pattern supports separation of concern by providing separation between business layer (services) and data access operation, as shown in Figure 4-7.

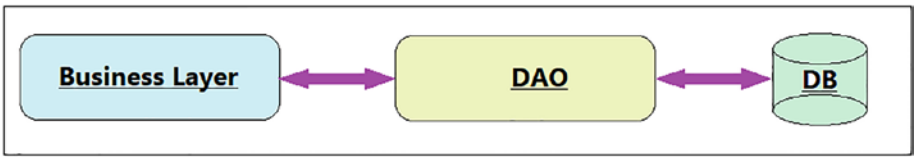


Figure 4-7. *Separation of concern*

The DAO layer sits between the business layer and the database and performs CRUD (create, retrieve, update, delete) operations in the database. To support `JpaRepository`, you need to add the Spring Data JPA dependency shown in Listing 4-5 to the `pom.xml` file.

Listing 4-5. Spring Data JPA Dependency

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Listing 4-6 creates a repository interface named `UserJpaRepository` by extending the `org.springframework.data.jpa.repository.JpaRepository` interface that helps in persisting the `UserDTO` domain object into a relational database.

Listing 4-6. \src\main\java\com\apress\AWS\repository\UserJpaRepository.java

```
package com.apress.AWS.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.apress.AWS.dto.UserDTO;

@Repository
public interface UserJpaRepository extends
JpaRepository<UserDTO, Long> {

}
```

In Listing 4-6, the JpaRepository interface takes a domain object. The domain object's identifier field is UserDTO and Long. Its generic parameters are T and ID. The UserJpaRepository interface inherits all the CRUD methods provided by JpaRepository.

Next, let's create a Service class that autowires UserJpaRepository.

Service Implementation: UserService

Let's begin the service implementation by creating a Service class named UserService, as shown in Listing 4-7, where you call the CRUD methods of the UserJpaRepository interface to handle SQL operations.

Listing 4-7. \src\main\java\com\apress\AWS\service\UserService.java

```
package com.apress.AWS.service;

import java.util.List;

import javax.transaction.Transactional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.apress.AWS.dto.UserDTO;
import com.apress.AWS.repository.UserJpaRepository;

@Service
@Transactional
public class UserService {

      @Autowired
      private UserJpaRepository useRepository;
```

```
        public List<UserDTO> listAll() {
              return useRepository.findAll();
        }

        public void save(UserDTO user) {
              useRepository.save(user);
        }

        public UserDTO get(Long id) {
              return useRepository.findById(id).get();
        }

        public void delete(Long id) {
              useRepository.deleteById(id);
        }
}
```

This `UserService` class uses the `@Autowired` annotation that autowires `UserJpaRepository`.

Next, let's create a REST controller class to define different REST endpoints to retrieve and manipulate the `UserDTO` domain object.

REST Controller Implementation: UserRegistrationController

Let's create a Spring REST controller named `UserRegistrationController` and implement different REST API endpoints to perform CRUD operations. Listing 4-8 is the code implementation for the `UserRegistrationController` class.

Listing 4-8. \src\main\java\com\apress\AWS\controller\UserRegistrationController.java

```
package com.apress.AWS.controller;

import java.util.List;
import java.util.NoSuchElementException;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.apress.AWS.dto.UserDTO;
import com.apress.AWS.service.UserService;

import lombok.extern.slf4j.Slf4j;

@Slf4j
@RestController
@RequestMapping("/api/")
public class UserRegistrationController {

      @Autowired
      private UserService userService;

      // URI - /api/users
      @GetMapping(value = "users")
      public ResponseEntity<List<UserDTO>> istAllUsers() {
```

```
        List<UserDTO> users = this.userService.listAll();
        return new ResponseEntity<List<UserDTO>>(users,
        HttpStatus.OK);
}

// URI - /api/user/id/1
@GetMapping("user/id/{id}")
public ResponseEntity<UserDTO> getUserById(
@PathVariable(name = "id") final Long userId) {
        try {
                final UserDTO user = this.userService.
                get(userId);
                return new ResponseEntity<UserDTO>
                (user, HttpStatus.OK);
        } catch (NoSuchElementException e) {
                return new ResponseEntity<UserDTO>
                (HttpStatus.NOT_FOUND);
        }
}

// URI - /api/user/save
@PostMapping(value = "user/save")
public ResponseEntity<UserDTO> save(@RequestBody UserDTO
user) {
        this.userService.save(user);
        return new ResponseEntity<UserDTO>(user,
        HttpStatus.CREATED);
}

// URI - /api/user/delete/id/1
@DeleteMapping("user/delete/id/{id}")
public ResponseEntity<UserDTO> delete(@PathVariable
(name = "id") final Long userId) {
```

```
            this.userService.delete(userId);
            return new ResponseEntity<UserDTO>(HttpStatus.
            NO_CONTENT);
        }
}
```

Here, the `UserRegistrationController` class was annotated with `@RestController` annotation. `@RequestMapping("/api")` was defined, which indicates that all REST API endpoint URLs start with `/api`, and it maps incoming HTTP requests to handler methods!

The `@Autowired` annotation autowires `UserService` to the RESTful controller. Table 4-2 explores the different REST endpoints defined in the `UserRegistrationController` class to retrieve and manipulate `UserDTO`.

Table 4-2. *REST Endpoints Defined in the UserRegistrationController Class*

Annotation	URI	Description
@GetMapping	/api/users	Retrieve all users available in database
@PostMapping	/api/user/save	Create a new user in database
@GetMapping (“/ {id}”)	/api/user/id/{id}	Retrieve an individual user based on ID
@ DeleteMapping	/api/user/delete/id/{id}	Delete an individual user based on ID

Now, build the `UserRegistrationApp` using a Maven build and run it locally to test defined REST endpoints.

Run and Test UserRegistrationApp Locally

To run the `UserRegistrationApp` using the STS IDE in a local system, right-click the `UserRegistrationAppApplication.java` class under the `com.apress.AWS` package, and then click **Run As ➤ Spring Boot App**, as shown in Figure 4-8.

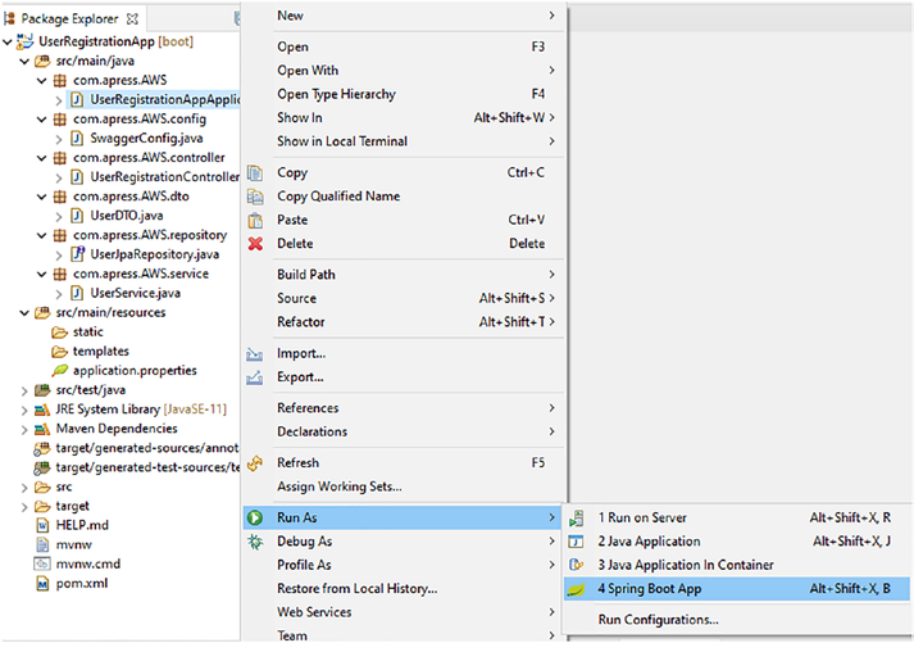


Figure 4-8. *Run the UserRegistrationApp using STS IDE*

Once `UserRegistrationApp` started successfully, the last line in the STS console should state, `Started UserRegistrationAppApplication`, as shown in Figure 4-9.

Figure 4-9. *Output on the STS console*

Now, it's time to test the REST API using Postman (`www.postman.com`). You added data to the database using MySQL Workbench in Chapter 3. You should get that data during the REST API call.

Retrieve All Users: /api/users

Let's test the first REST endpoint to retrieve all users. Launch the Postman tool in your local system, select GET as the request type, and enter **`http://localhost:5000/api/users`** to retrieve and display all user data. You should see a 200 OK HTTP status, as shown in Figure 4-10.

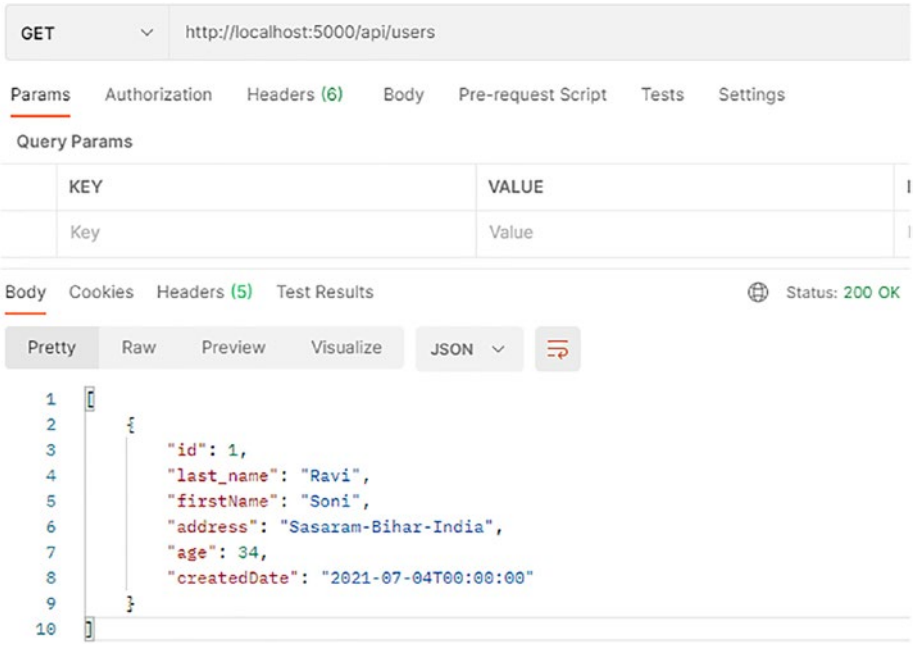


Figure 4-10. *Retrieve all users*

Retrieve an Individual User: /api/user/id/{id}

Now, let's test another REST endpoint to retrieve an individual user based on id. To test this REST endpoint, launch Postman, select GET as the request type, and enter the URL (`http://localhost:5000/api/user/id/1`) to retrieve and display individual user data. You should see a 200 OK HTTP status, as shown in Figure 4-11.

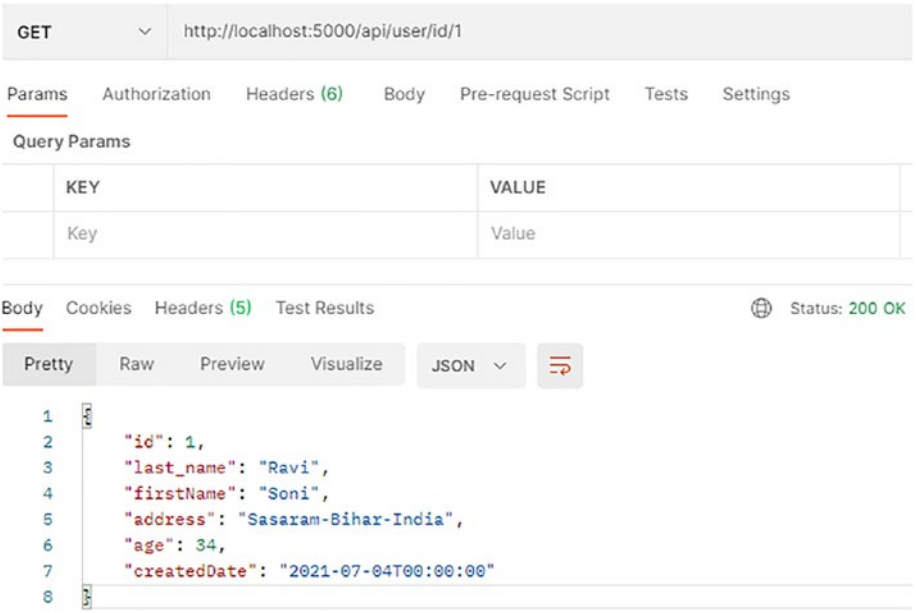


Figure 4-11. *Retrieving an individual user*

Create a New User: /api/user/save

Next, let's test the REST endpoint to create a new user in the database. Launch Postman, select POST as the request type, and enter **`http://localhost:5000/api/user/save`**. Click the Body radio button, and then select **raw**. From the drop-down list, select JSON (application/json) as the content-type header. Use the JSON data in the request body as shown in Listing 4-9, and hit Send.

Listing 4-9. JSON Data in the Body to Create a New User

```
{
      "last_name": "Soni",
      "firstName": "Namrata",
      "address": "Bangalore-India",
```

```
    "age": 25,
    "createdDate": "2021-07-04T00:00:00"
}
```

On successful completion of the POST request, a new user is created in the database, and the response HTTP status is 201 Created, as shown in Figure 4-12.

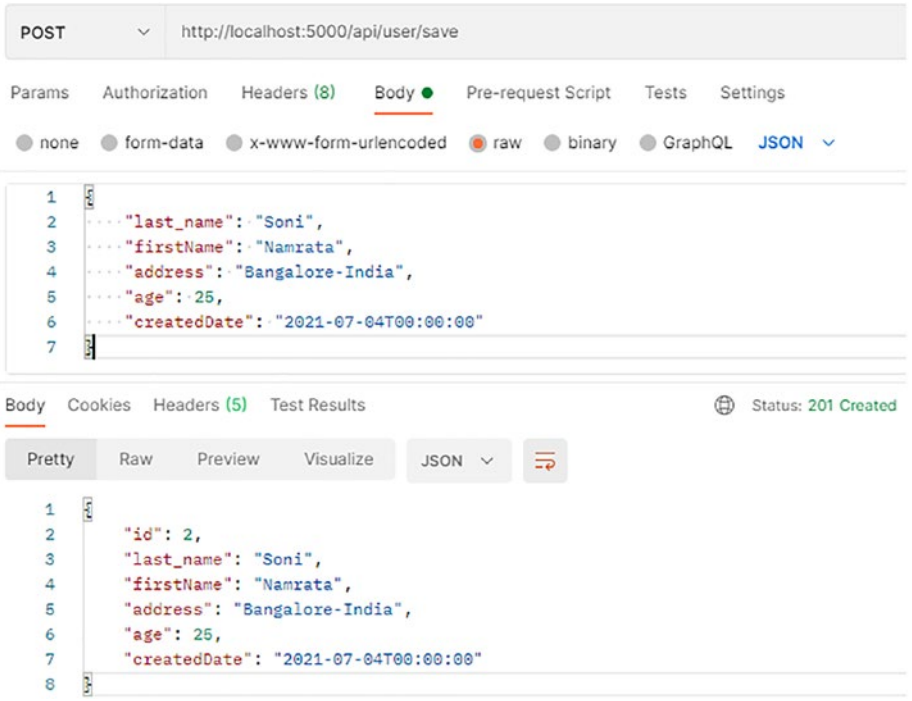


Figure 4-12. *Creating a new user*

Delete an Existing User: /api/user/delete/id/{id}

The last endpoint to test deletes an existing user from the database based on ID. To test this REST endpoint, launch Postman, select DELETE as the request type, and enter the URL (`http://localhost:5000/api/user/id/2`) to delete the existing user with `id=1`. On successful completion of the DELETE request, this user is deleted from the database. The response HTTP status after deleting the user is 204 No Content, as shown in Figure 4-13.

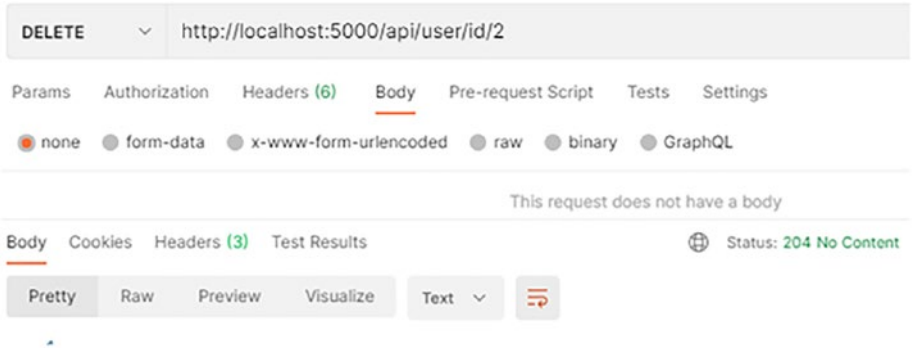


Figure 4-13. *Delete an existing user*

Swagger UI: API Documentation

In a browser, open the Swagger UI page at `http://localhost:8080/swagger-ui/`. You see the generated API documentation, as shown in Figure 4-14.

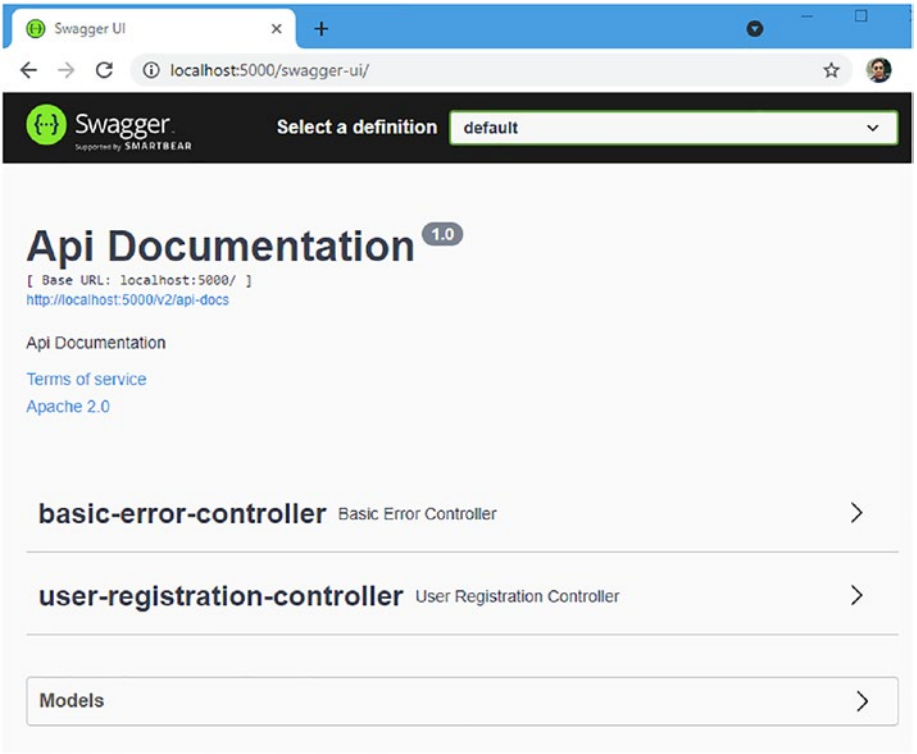


Figure 4-14. *Swagger API documentation page*

`user-registration-controller` is defined in the application. Clicking it lists the REST endpoints and their valid HTTP methods. Clicking Models displays the model structure. Figure 4-15 shows the defined REST endpoints and the UserDTO model structure.

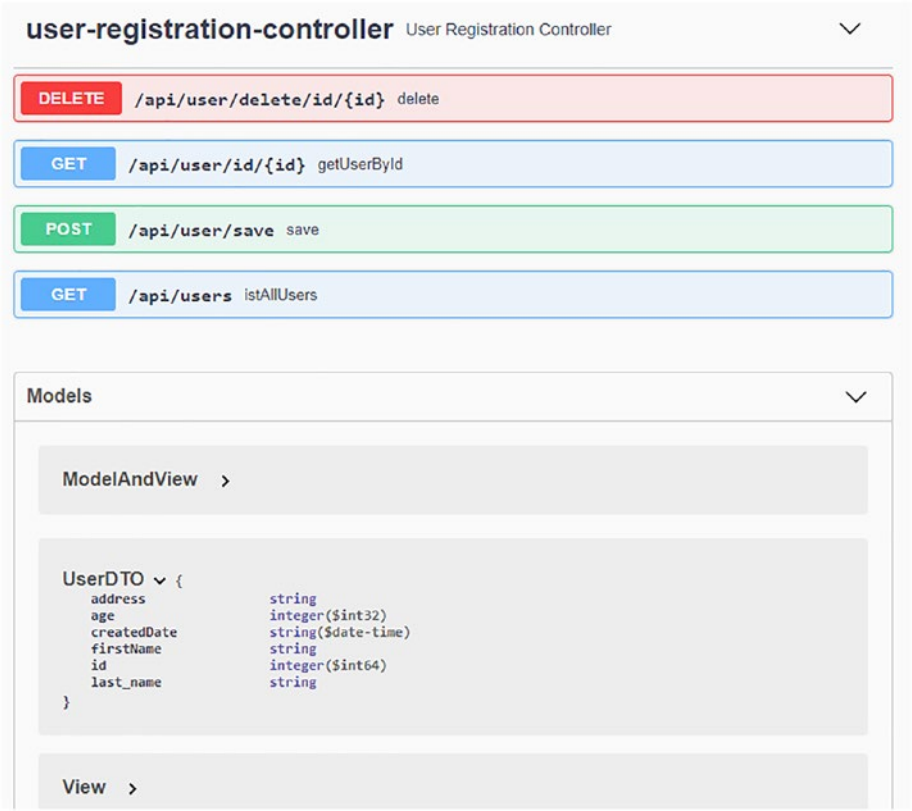


Figure 4-15. *Swagger UI lists REST endpoints*

Build a JAR for a Spring Boot Application

To build JAR for the Spring Boot application from a command prompt, go to the project directory where you created the Spring Boot project and copy the project path. Now, change the working directory to the project path on the command prompt. Build the project using the following command executed in the command prompt, as shown in Figure 4-16.

```
E:\Apress\workspace\AWS\UserRegistrationApp>mvn clean install
```

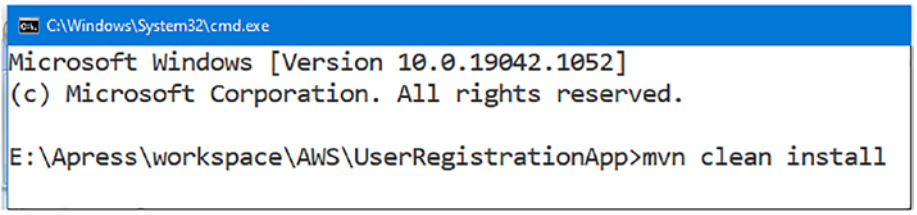


Figure 4-16. *Build JAR from the command prompt*

This starts building the `UserRegistrationApp` project. Once the build is successful, you are informed that the JAR file named `UserRegistrationApp-0.0.1-SNAPSHOT.jar` has been created, as shown in Figure 4-17.

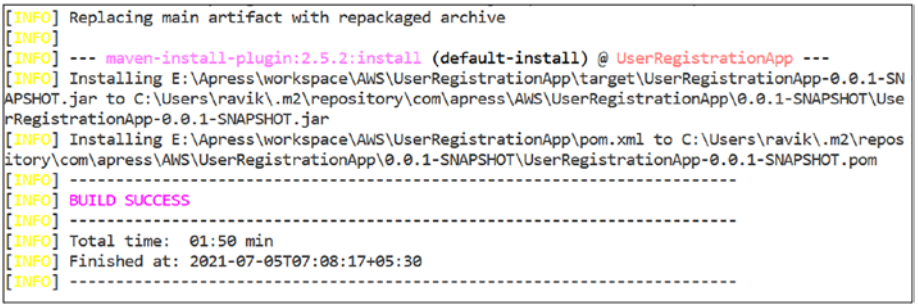


Figure 4-17. *Build success*

You need to deploy the generated JAR file into Elastic Beanstalk.

Deploy the UserRegistrationApp Spring Boot Application in AWS Elastic Beanstalk

Since you have successfully created a JAR file for the `UserRegistrationApp` application in your local system, now, you must deploy this JAR file to Elastic Beanstalk.

Let's sign in to the AWS Management Console using your AWS credentials and select service as Elastic Beanstalk. Figure 4-18 shows that three applications are already available: `My First Elastic Beanstalk Application`, `helloworld`, and `HelloSpringBoot`. You created them in previous chapters.

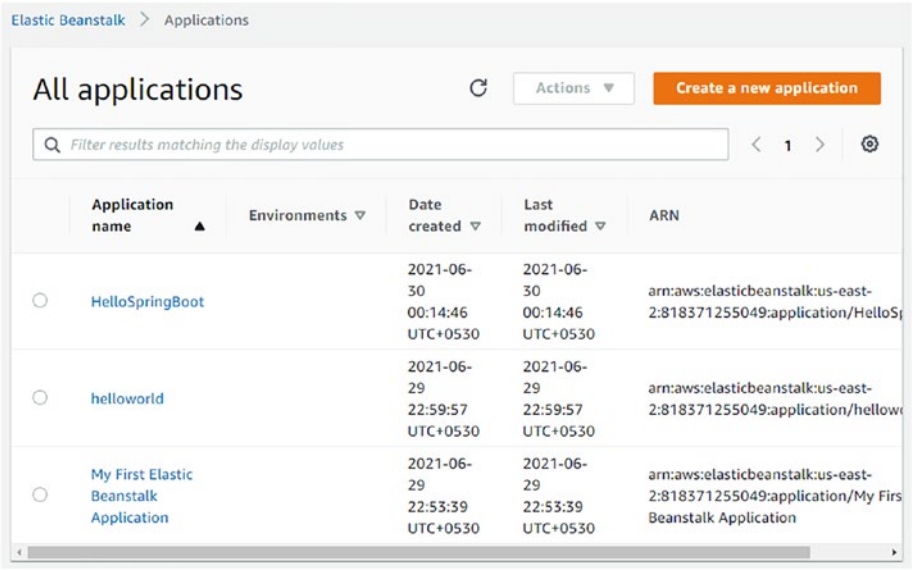


Figure 4-18. *List of all applications available in Elastic Beanstalk*

Next, let's create a new application for `UserRegistrationApp` Spring Boot application talking to the MySQL database. Click the **Create a new application** button, enter the application name as **`UserRegistrationApp`**, and click the Create button.

Next, create a new environment for this application by clicking the **Create one now** link. Select **Web server environment** as the environment tier, and then click the Select button.

On the **Environment information** page, enter **`userregistration`** as the domain name, and check for domain availability (see Figure 4-19).

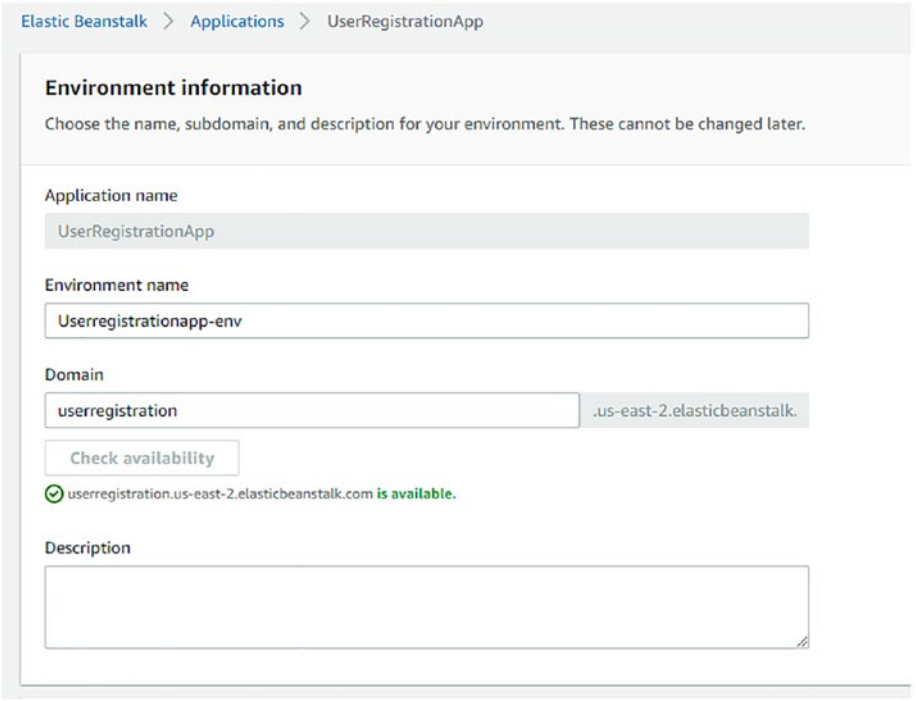


Figure 4-19. *Environment information*

Next, select Java as the managed platform, as shown in Figure 4-20.

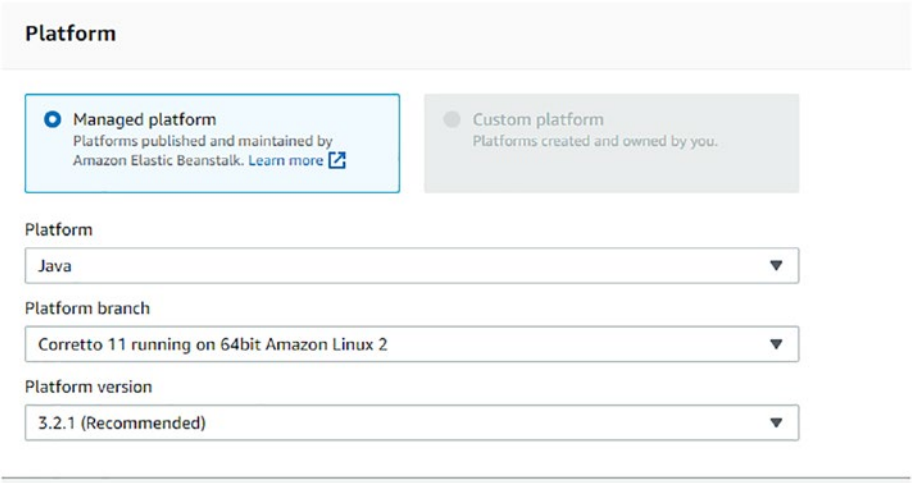


Figure 4-20. *Java is the managed platform*

Finally, upload the code by selecting the JAR file from the project's `target` folder (e.g., in the authors' local system, it is `E:\Apress\workspace\AWS\UserRegistrationApp\target\UserRegistrationApp-0.0.1-SNAPSHOT.jar`), and then click the **Create environment** button, as shown in Figure 4-21.

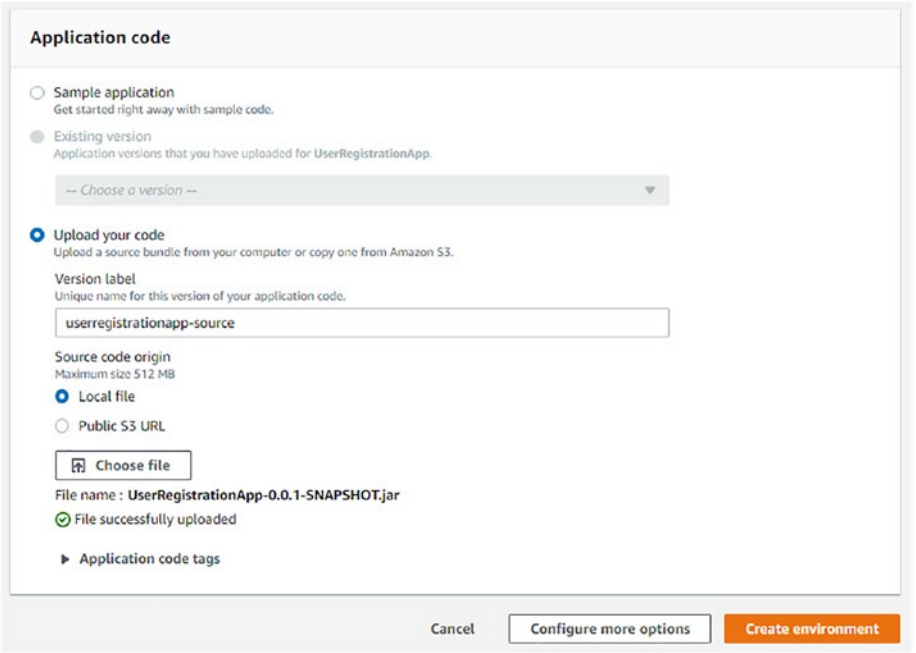


Figure 4-21. *Upload application code*

Once the environment has been created, and the resources have been deployed, change the server port the Spring Boot application listens on. So, you need to specify the `SERVER_PORT` environment variable in the Elastic Beanstalk environment and set the value to 5000.

On the Configuration page in your environment, under Software, click the Edit icon, as shown in Figure 4-22.

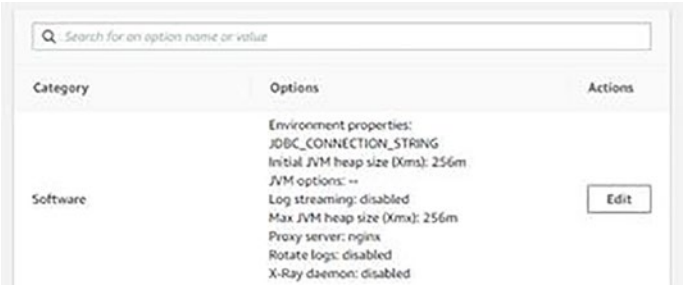


Figure 4-22. *Edit software configuration*

And then add a new environment variable `SERVER_PORT`, with a value 5000 to change the port that the Spring Boot application listens on, as shown in Figure 4-23.

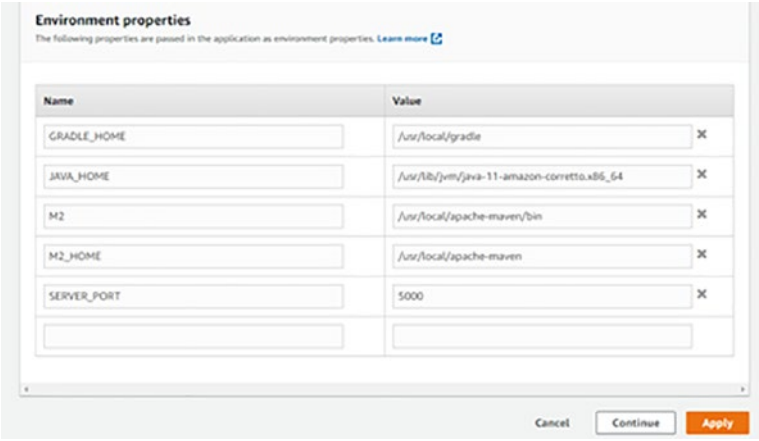


Figure 4-23. *Environment properties in software configuration*

As soon as you click the Apply button, the configuration changes are propagated to the application servers, and the application is restarted.

When it restarts the application, it picks up the new configuration through the environment variables. And, in about a minute, you see a healthy application on the dashboard, as shown in Figure 4-24.

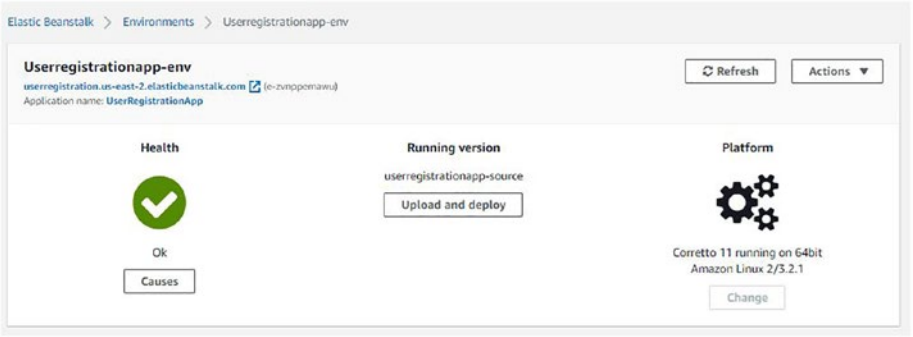


Figure 4-24. *Health OK*

You are now ready to test the `UserRegistrationApp` application deployed in the Amazon cloud.

Test Deployed REST API in AWS Using Swagger UI

Now, it's time to test the deployed REST API endpoints in AWS. Use the URL that you configured on the AWS environment to access the service. For this example, the specified URL is `http://userregistration.us-east-2.elasticbeanstalk.com`.

Let's open the Swagger UI page in the browser at `http://userregistration.us-east-2.elasticbeanstalk.com/swagger-ui/`. You see the generated API documentation, as shown in Figure 4-25.

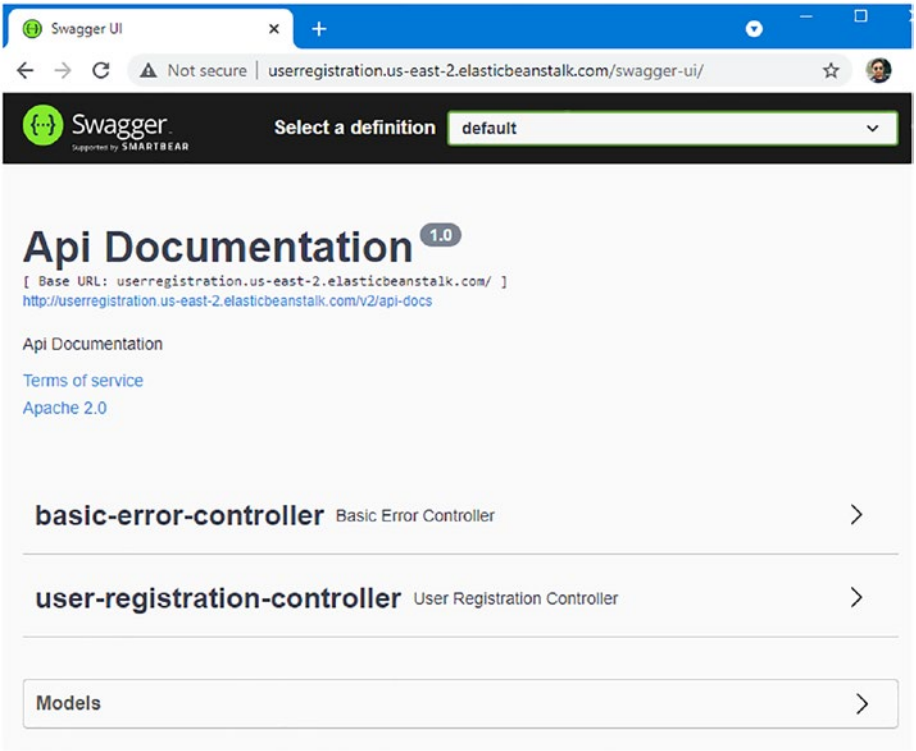


Figure 4-25. *Swagger API documentation page*

Here, clicking `user-registration-controller` shows the list of defined REST endpoints, and by clicking the Models display domain model structure, as shown in Figure 4-26.

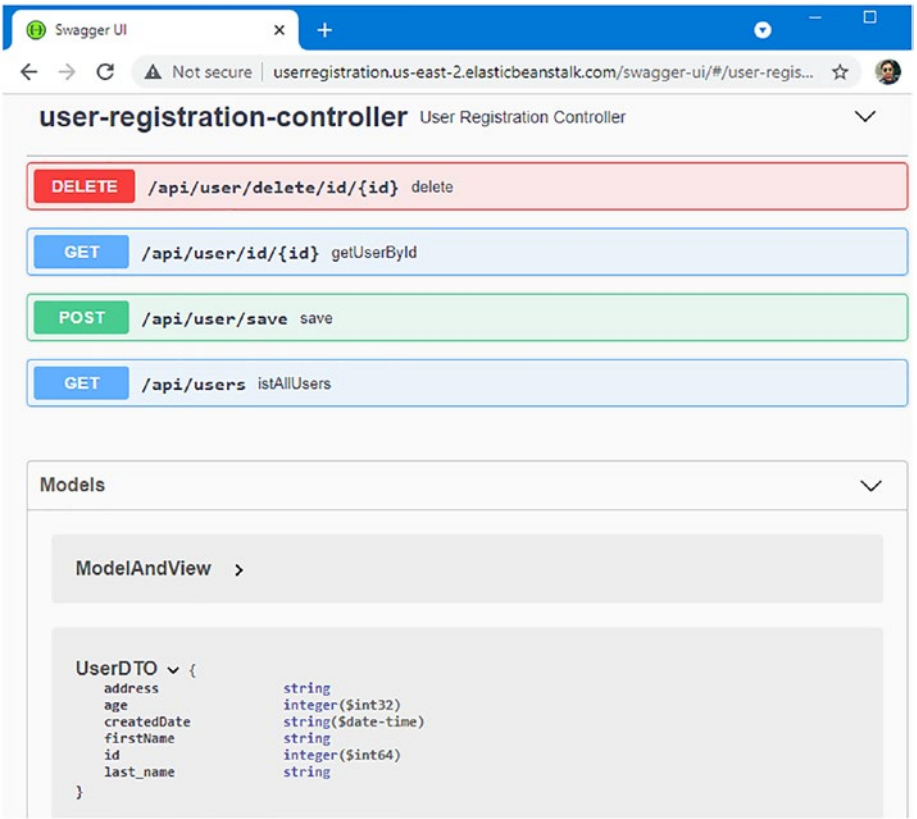


Figure 4-26. *Swagger UI lists REST endpoints and model structure*

Using Swagger, let's test the REST Endpoints deployed on AWS.

List All Users: /api/users

On the Swagger UI page, expand `GET /api/users`, and click the Try It Out button. And then, click the Execute button to call this REST endpoint. Figure 4-27 shows that the HTTP status response code should be 200 OK, and the response body should contain the list of users.

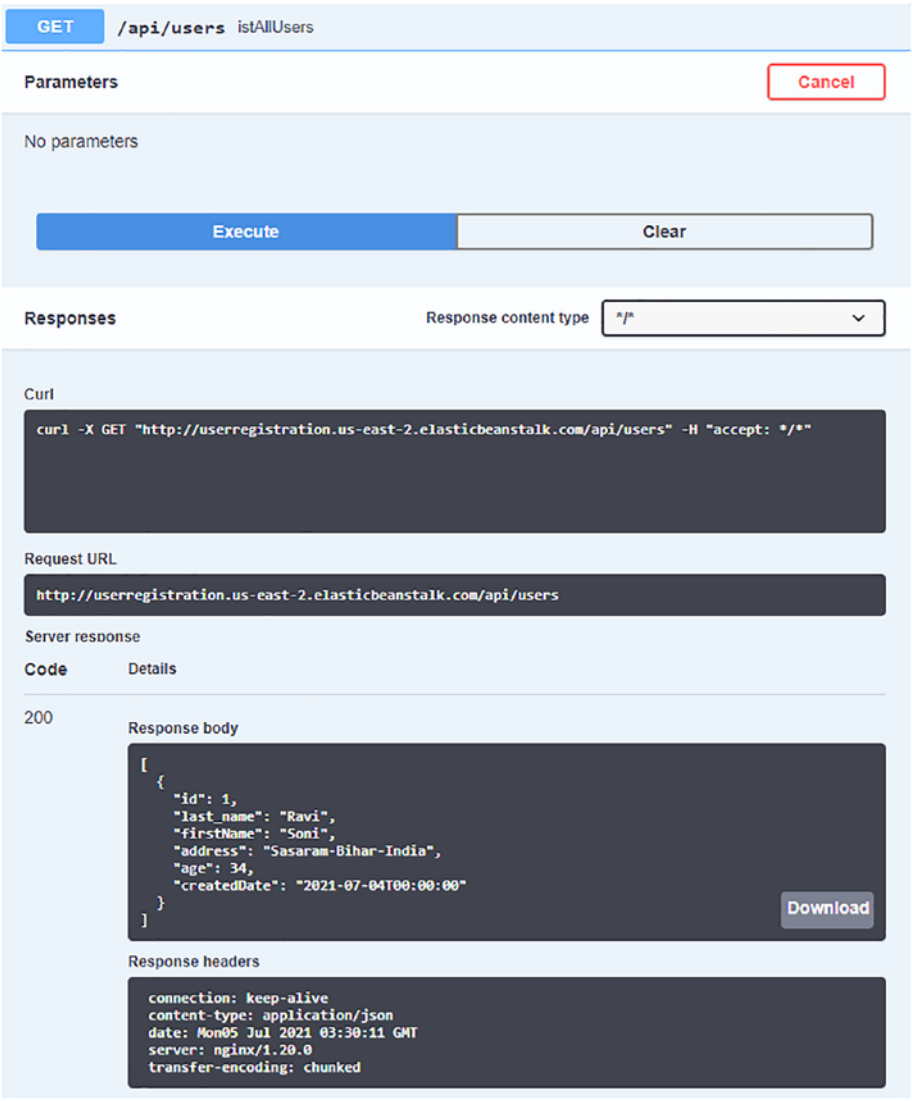


Figure 4-27. *List all users using Swagger UI*

Create New Users: /api/users

On the Swagger UI page, expand `POST /api/user/save`, and click the Try It Out button. Next, enter the user JSON data shown in Listing 4-10 in the request body input box, and select **application/json** as the content-type parameter.

Listing 4-10. User JSON Data

```
{
      "last_name": "Soni",
      "firstName": "Namrata",
      "address": "Bangalore-India",
      "age": 25,
      "createdDate": "2021-07-04T00:00:00"
}
```

Next, click the Execute button to call this REST endpoint. As shown in Figure 4-28, the response HTTP status code should be 201 Created.

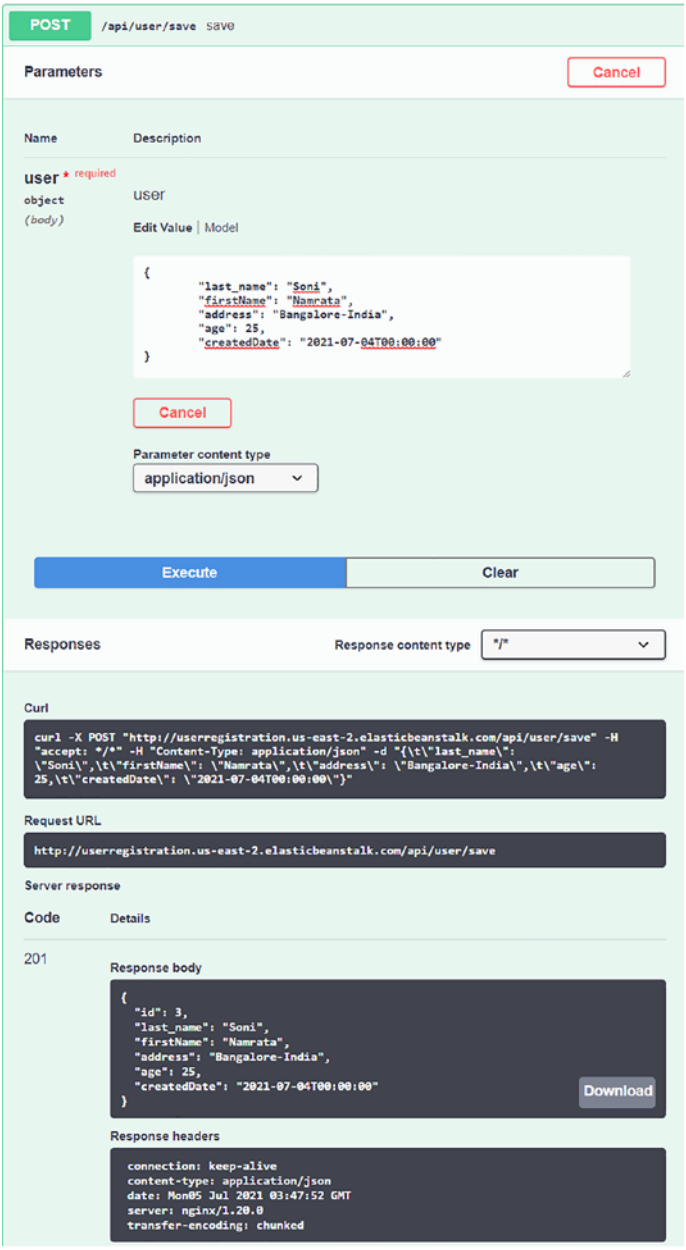


Figure 4-28. *Create a new user using Swagger UI*

Summary

In this chapter, you created `UserRegistrationApp` Spring Boot REST API talking to an Amazon RDS MySQL database. You explored different Maven dependencies that have been used in the `pom.xml` file, such as Lombok, JPA, and so on. You learned how to configure Project Lombok to STS IDE. You updated the `application.properties` file with database details such as URL, username, and password, and many more. And then, you created an Entity class using JPA annotation, a repository interface that extends the JpaRepository interface, a service class for CRUD methods, and a REST controller to define different REST endpoints.

First, you tested the `UserRegistrationApp` application locally using Postman. Then you built a JAR that you deployed in Elastic Beanstalk. Finally, you tested the deployed REST endpoints to the AWS cloud using the Swagger UI.

The next chapter explores how to deploy a full stack Spring Boot React application in AWS and S3.

CHAPTER 5

Deploy a Full Stack Spring Boot React Application in AWS and S3

In Chapter 4, you created the `UserRegistrationApp` Spring Boot RESTful web service that talks to the Amazon RDS MySQL database to perform CRUD operations. You learned how to configure Project Lombok to STS IDE. You created an `Entity` class using JPA annotation, a repository interface that extends the JpaRepository interface, a `Service` class for CRUD methods, and a REST controller to define different REST endpoints. Afterward, you tested the `UserRegistrationApp` application locally using Postman. Then you built an executable JAR that was deployed in AWS Elastic Beanstalk. Finally, you tested the deployed REST endpoints using Swagger UI.

The world sees the front end, including the design using some languages such as HTML and CSS. The main aim of the front end is to present data in a well-defined style and allows interaction with the client to perform CRUD operations. There are so many amazing JavaScript libraries available that can develop front-end applications.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0_5

React is an open source, front-end JavaScript library for building single-page applications. React is a perfect solution for a client-side library for a clean and structured approach.

This chapter introduces React as a front-end framework and its major components. You can develop a single-page application using React as the front end to consume APIs exposed by the `UserRegistrationApp` back-end application developed using Spring Boot, as shown in Figure 5-1.

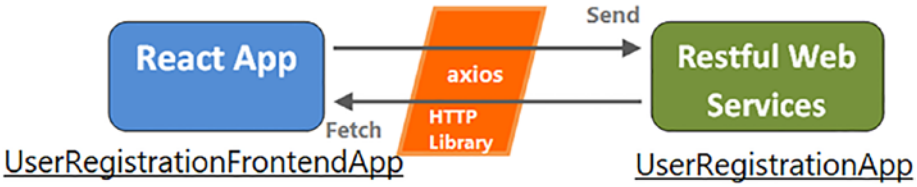


Figure 5-1. *Full stack application overview*

You set up a development environment to develop your React front-end application. In this chapter, you learn the following.

- How to develop and run React as a local front-end application
- How to deploy the React front end to AWS S3

This front-end application has a home page, an Add New User page, and a List All Users page with a Delete option. You make an API call to AWS, where you have already deployed the back-end RESTful services named `UserRegistrationApp`. You are introduced to AWS S3 (Simple Storage Service), where you deploy the React front-end application. And, finally, you verify successful deployment of the React front-end application.

Develop and Run React as a Front-End Application

Let's start developing and running the interactive front-end application with React in the local system. We assume that you have good knowledge of JavaScript, HTML5, CSS, and React. If you want an in-depth understanding of React, refer to https://reactjs.org.

Introducing React as a Front-end Framework

React is an open source, component-based JavaScript library for building fast and interactive UI (user interface) components. It was created in 2011 by a Facebook software engineer named Jordan Walke. Initially, it was developed and maintained by Facebook. React application is made up of independent, isolated, and reusable components, which are the heart of React application, and each component is responsible for building complex and reusable user interfaces. Every React application has at least one component known as the *root component*. This root component represents the internal application and contains other child components.

You build a user registration front-end app using React with CRUD features. This React application has different components, as shown in Figure 5-2.

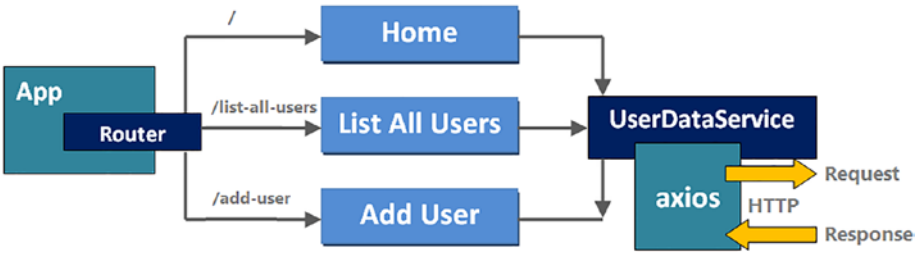


Figure 5-2. *React components with Router and Axios*

- The `App` component is a root component that contains `react-router`. This also contains a navbar that links to the route's paths.
- The `Home` component displays a welcome message.
- The `ListAllUsers` component displays a list of all users with a Delete option.
- The `AddUser` component has a form for new user submission.

All these components call required methods in `UserDataService`, which internally uses the Axios HTTP library to make HTTP requests and receive responses.

React Components

In React, a component is considered as the core part of the user interface. Each component has its own structure and is independent of other components, and when all the components merge in a parent component results in the final UI of the application. A component is typically implemented as a JavaScript class with some state and a render method, as shown in Listing 5-1.

Listing 5-1. Structure of Component with State and Render Method

```
class UserClass {
   state = {};
   render() {

   }
}
```

There are mainly two types of components in React.

- Stateless functional components
  - These are JavaScript functions that don't have their own state and return HTML to describe UI.
- Stateful class components
  - These are regular ES6 classes that extend the `Component` class from the React library. They must contain a render method, which in turn returns React elements or HTML. They manage the local state.

React State

The state is an updatable structure that is managed within the component. A `Stateful` component has a state responsible for making the user interface dynamic and interactive. You need to declare some default set of values to define the initial state of components. A state can be set or changed using a `setState` method.

Constructor

In React, the constructor initializes an object's state of a class. This constructor is called automatically during the object creation of the class. It is called before the component is mounted. You need to call the `super(props)` method before any other statement in a constructor. Also, in React, the constructor binds the event handler method.

A React Component's Life Cycle

Let's explore the React component's life cycle. It primarily consists of four phases, as shown in Figure 5-3.

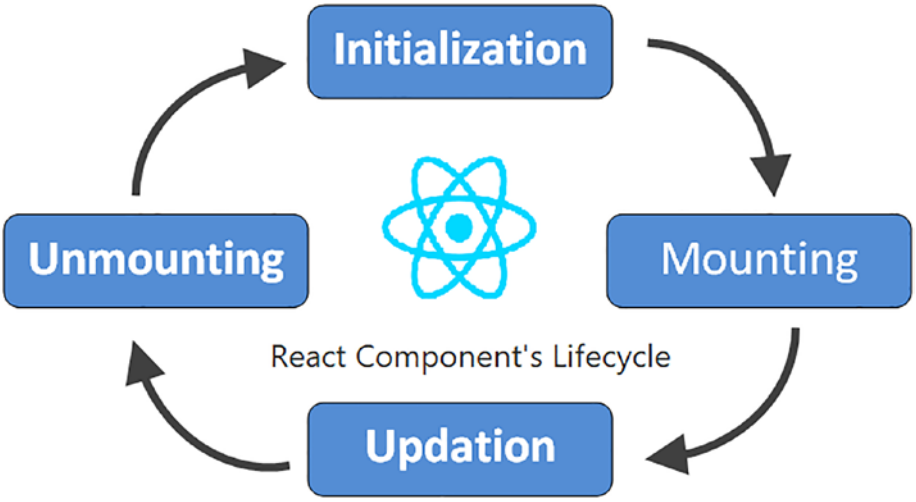


Figure 5-3. *React component's life cycle*

The different phases of the React component's life cycle provide different methods. React calls the life cycle method according to the component phase.

- Initialization is the birth phase of React components, where they start their journey by setting up the initial state and default props. This is done in the component's constructor.
- Mounting is the phase where the React component mounts (created and inserted) on the Document Object Model (DOM). After completing the initialization phase, the React component renders for the first time in this mounting phase.
- Updation is the third phase of a React component's life cycle. It is the state of the created component change. The React component data (e.g., props and state) is updated in response to user events like typing, clicking, and so on.

- Unmounting is the last phase in this life cycle. The React component instance is destroyed and unmounted from the DOM.

Set up a Development Environment

The following tools are needed to run any React application.

- A **code editor**, such as Visual Studio, to work with the project files. You can download it from `https://code.visualstudio.com`.
- Go to `https://nodejs.org` to download and install the latest version of **Node.js**, which is a JavaScript runtime environment.
- A package manager called **npm**, which downloads and runs JavaScript packages built on Node.js. It's automatically included in your installation of Node.js.

To check the Node.js and npm versions, run the `node -v` and `npm -v` commands in your terminal, as shown in Figure 5-4.

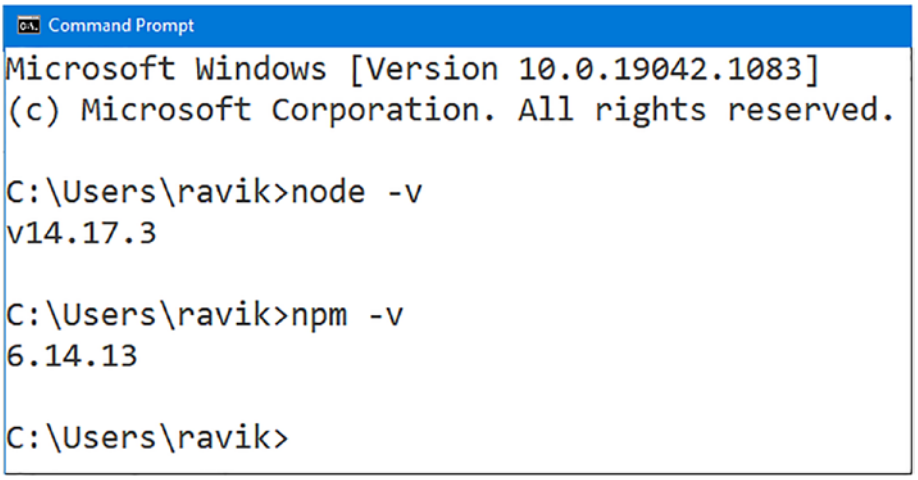


Figure 5-4. *Node.js and npm version in PC*

Cross-Origin Resource Sharing (CORS) Error

When you work on a front-end application in React that connects to a RESTful web service written in Spring Boot, you may get a CORS error whenever you make the request in your browser. Basically, this error means that the user agent (`http://localhost:3000`) doesn't have sufficient required permissions to access Spring Boot resources (`http://localhost:5000`).

The solution to this error required an update in the Spring Boot application to enable cross-origin requests for a RESTful web service. You must annotate the `Controller` class with `@CrossOrigin` annotation to support global CORS configuration, as shown in Listing 5-2. And, by default, all origins and the GET, HEAD, and POST HTTP methods are allowed.

Listing 5-2. \src\main\java\com\apress\AWS\controller\UserRegistrationController.java

```
@CrossOrigin
@RestController
@RequestMapping("/api/")
public class UserRegistrationController {
```

After updating the `Controller` class, Maven builds and runs the `UserRegistrationApp` Spring Boot application. And, also make sure that `UserRegistrationApp` should always be running when developing the front-end application using React.

Developing React Front-End Application with create-react-app

The `create-react-app` package makes developing React front-end applications a breeze. To create a React app using `create-react-app`, open a command prompt in the folder where you want to save the project folder and run the following `npx` command (see Figure 5-5).

```
npx create-react-app user-registartion-frontend-app
```

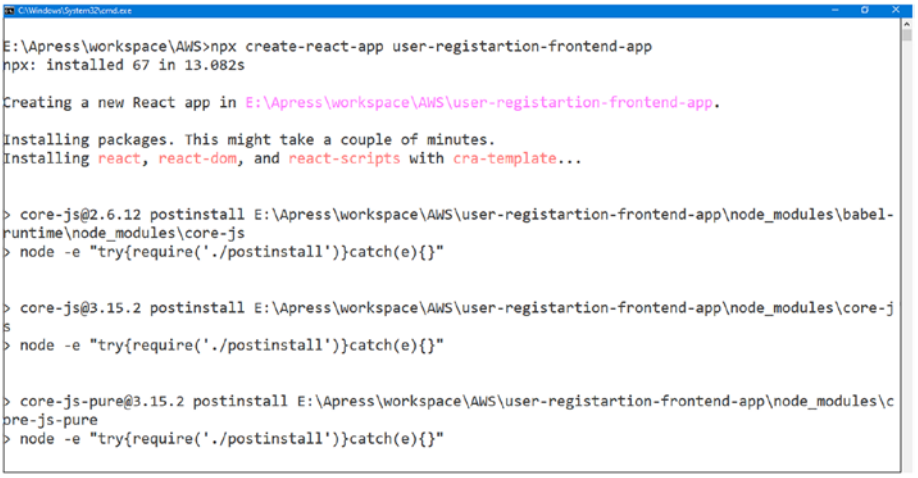


Figure 5-5. *npx command to create a React app using create-react-app*

Once the `npx` command has run successfully, a folder named `user-registration-frontend-app` is created, as shown in Figure 5-6; all the required packages are automatically installed.

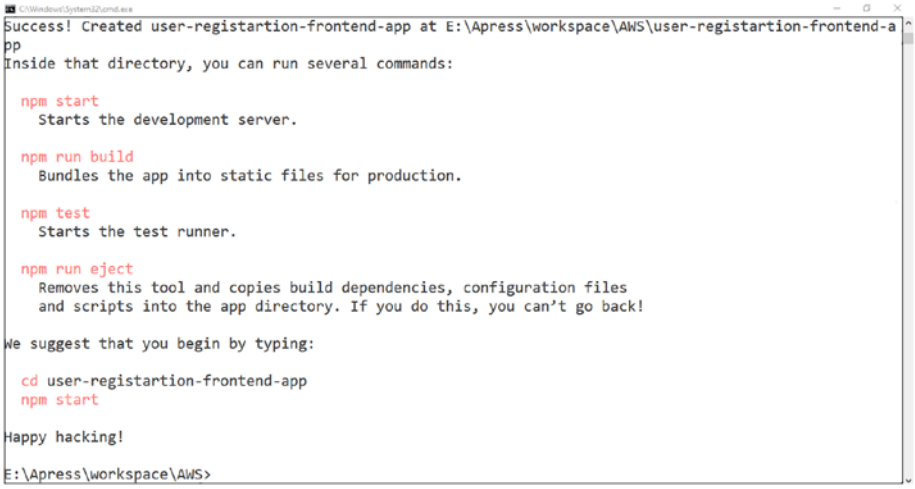


Figure 5-6. *Successfully created user-registration-frontend-app*

Review the Project Structure

Once the React project has been created and all the required dependencies have been installed, open the project in Visual Studio. The project structure should look like as shown in Figure 5-7.

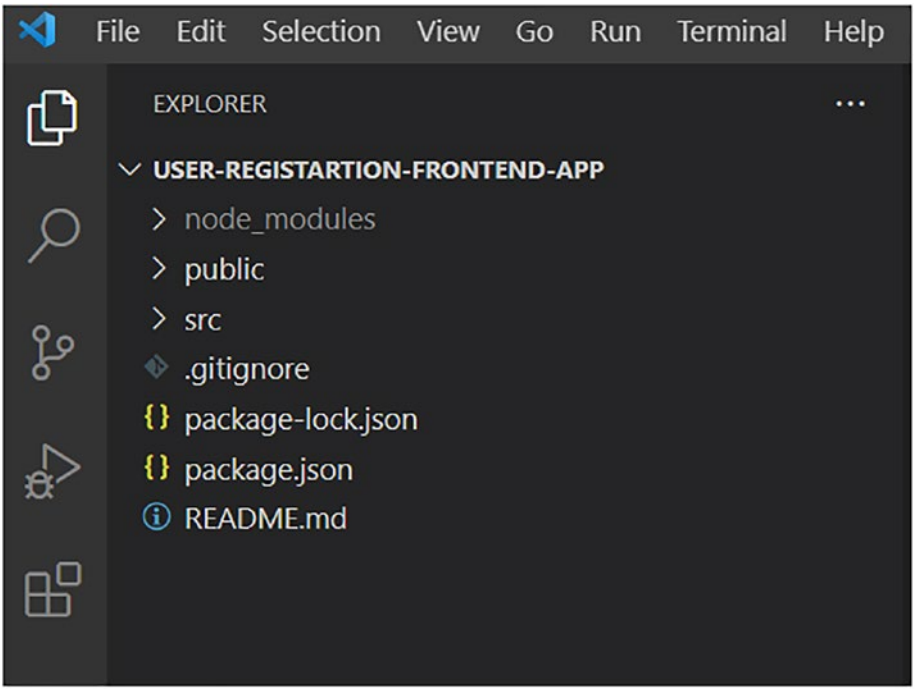


Figure 5-7. *Project structure in Visual Studio*

The project structure contains the following files and folders.

- The `README.md` file is a markdown file that includes a lot of helpful information.
- The `package.json` file manages the app's required dependencies and the scripts needed to run it.
- The `.gitignore` file excludes desired files and folders from being tracked by Git. Generally, you exclude large folders like the `node_modules` folder.
- The `src` folder contains React-related source code and all the components that you develop.

    - The `App.js` file in the `src` folder is a root component of the React application.
    - The `index.js` file is the top render file of the React application. You import `App` components using the `ReactDOM.render()` method in the `index.js` file.
- The `public` folder stores static assets, such as fonts and images, for the React app.
    - The `index.html` file is in the public folder. The React application uses this single file to render all the components. This supports the principle of a single-page application.
- The `node_modules` folder contains all the packages installed with Node.js and npm.

Run a React App

To build the React app, the following files must exist with the exact filenames.

- `public/index.html` is the only HTML file in the entire project. This HTML file is a template, and it is loaded first when the application starts.
    - Only those files which are there in the `public` folder can be used from `public/index.html`.
    - This file contains a line of code `<div id="root"></div>`, which signifies that all the React app components are loaded into this div.
- `src/index.js` is the JavaScript entry point.
- The `src/App.js` is the `App` component, which is the main component in React; it acts as a container for all the other components.

To start the React app, open the command prompt at `user-registration-frontend-app`, which is a newly created folder, and run the `npm start` command, as shown in Figure 5-8.

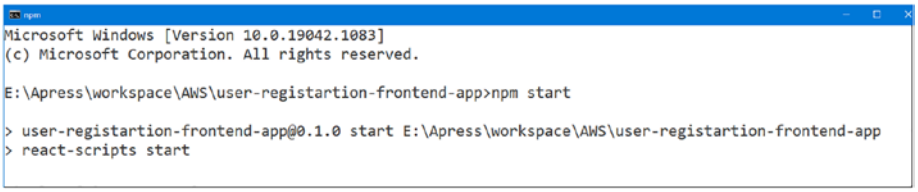


Figure 5-8. *npm start command to start React app*

A success message should appear in the command prompt, as shown in Figure 5-9.

Figure 5-9. *Compiled success message on command prompt*

This started the development server on localhost:3000. The great thing about this development server is that the server automatically refreshes to reflect the changes, and there is no need to refresh the browser manually.

You can view the application in the browser by hitting the URL (`http://localhost:3000`), as shown in Figure 5-10.

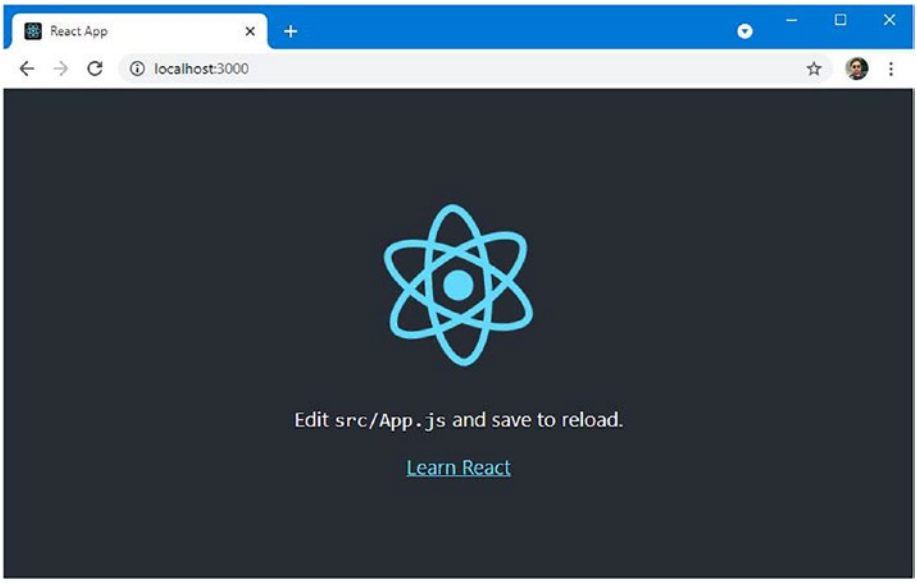


Figure 5-10. *Home page for React app*

Congratulations! You have successfully created a base source code for the React application to add more components as needed. This app content comes from the `src/App.js` file, which contains the code shown in Listing 5-3.

Listing 5-3. src/App.js

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
```

```
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

To support CRUD operation, let's create the following additional files in the React application.

- `src/services/user-registration.service.js`
- `src/components/add-user.component.js`
- `src/components/home.component.js`
- `src/components/list-users.component.js`

Add Twitter Bootstrap to Style the React App with CSS

By default, `create-react-app` comes with CSS support by providing an `App.css` file in the `src` folder, where you can add some style to improve appearance. Twitter Bootstrap is a front-end CSS framework that can style a website's contents.

Open the command prompt, and run the `npm install bootstrap` command, which installs Bootstrap in the `node_modules` folder, as shown in Figure 5-11.

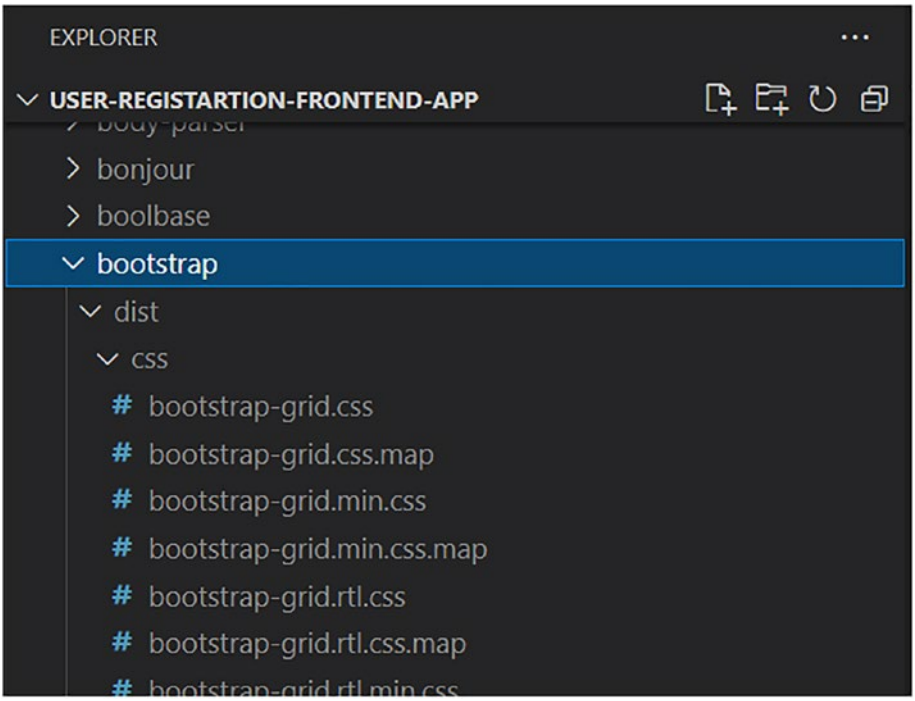


Figure 5-11. *Bootstrap installed in node_modules folder*

To import Twitter Bootstrap into the React app, open the `src/App.js` file and modify the code, as shown in Listing 5-4.

Listing 5-4. src/App.js

```
import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css'

function App() {
      // ...
}
```

Add a Navbar

Let's add a navbar to the App component, which is the root container for the React application. Update the `src/App.js` file with the code shown in Listing 5-5.

Listing 5-5. src/App.js

```
import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css'

function App() {
  return (
    <div className="App">
      <header className="App-header1">
        <div class="page-header text-center">
          <h2>User Registration App</h2>
        </div>
      </header>
      <div class="container-fluid">
        <nav class="navbar  bg-primary  justify-content-
        center">
            <div class="col-sm"></div>
            <a href="/"
              class="col-sm btn btn-outline-light"
```

```
              role="button">
              Home
            </a>
            <div class="col-sm"></div>
            <a href="/list-all-users"
              class="col-sm btn btn-outline-light"
              role="button">
              List All Users
            </a>
            <div class="col-sm"></div>
            <a href="/add-user"
              class="col-sm btn btn-outline-light"
              role="button">
              Add User
            </a>
            <div class="col-sm"></div>
        </nav>
      </div>
    </div>
  );
}

export default App;
```

Add react-router

Routing is a process that redirects users to different pages based on their request or action. The `react-router` package is a standard library system built on top of React and defines multiple routes using `react-router` in single-page web applications. When a user enters a specific URL in a browser, and the URL path matches a defined route, the user is routed to it.

By default, React doesn't come with routing. And, you need to add a react-router library in the project to enable routing. Open the command prompt and run the following command to install `react-router`.

```
npm install react-router-dom
```

Since you have successfully installed `react-router`, let's use it in the application.

BrowserRouter Object to Enable Routing

`BrowserRouter` uses the HTML5 history API to keep your user interface in sync with the URL. It is used in client-side routing with URL segments.

First, you need to import `BrowserRouter` from `react-router-dom` to enable routing in the project. Open and update `src/index.js` to wrap app components with the `BrowserRouter` object, as shown in Listing 5-6.

Listing 5-6. src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { BrowserRouter } from "react-router-dom";

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

reportWebVitals();
```

Switch and Route to Render Routes

`Switch` renders a route exclusively and helps with switching between pages without reloading it. Every route that matches the component and path renders inclusively.

The `path` property defines the path of the route; for example, `/` defines the path of the home page. `Route` loads the defined component; for example, it loads the home component. Update the `src/App.js` file with the source code shown in Listing 5-7.

Listing 5-7. Update src/App.js with react-router

```
import './App.css';
import React, {components} from 'react';
import { Switch, Route } from 'react-router-dom';
import 'bootstrap/dist/css/bootstrap.min.css'

import ListUsers from './components/list-users.component';
import Home from './components/home.component';
import AddUser from './components/add-user.component';

function App() {
  return (
    <div className="App">
      <header className="App-header1">
        <div class="page-header text-center">
          <h2>User Registration App</h2>
        </div>
      </header>
      <br/>
      <div class="container-fluid">
        <nav class="navbar  bg-primary  justify-content-center">
            <div class="col-sm"></div>
            <a href="/"
```

```
            class="col-sm btn btn-outline-light"
            role="button">
            Home
          </a>
          <div class="col-sm"></div>
          <a href="/list-all-users"
            class="col-sm btn btn-outline-light"
            role="button">
            List All Users
          </a>
          <div class="col-sm"></div>
          <a href="/add-user"
            class="col-sm btn btn-outline-light"
            role="button">
            Add User
          </a>
          <div class="col-sm"></div>
        </nav>
        <br/>
        <div className="container mt-3">
          <Switch>
            <Route exact path={["/"]} component={Home} />
            <Route exact path={["/list-all-users"]}
            component={ListUsers} />
            <Route exact path={["/add-user"]}
            component={AddUser} />
          </Switch>
        </div>
      </div>
    </div>
  );
}

export default App;
```

Three routes are defined in the React application.

- `/` for the home page
- `/list-all-users` for the List All Users page
- `/add-user` for the Add User page

Initialize Axios for a REST API Call

React is a JavaScript library that builds user interfaces. It is not concerned with HTTP. To make HTTP or REST API calls, you need to use a third-party HTTP library. Here, you use the Axios HTTP library.

Axios is a promise-based HTTP client that allows you to make an HTTP request to a given endpoint and has good defaults to work with JSON. To set up Axios with React, you need to install Axios with npm. Open the command prompt and run the `npm install axios` command. Let's create an `http-common.js` file in the `src` folder, as shown in Listing 5-8.

Listing 5-8. src/http-common.js

```
import axios from "axios";

export default axios.create({
  baseURL: "http://localhost:5000/api/",
  headers: {
    "Content-type": "application/json"
  },
});
```

Depending on the URL of REST API, you can update `baseURL` in the file.

Data Service to Send an HTTP Request

Next, create a data service that uses Axios to send HTTP requests to the REST API. Let's create a `service` folder in the `src` folder and a `user-registration.service.js` file in that folder, as shown in Listing 5-9.

Listing 5-9. src/user-registration.service.js

```
import http from '../http-common';

class UserDataService {

    getAllUsers() {
      return http.get("/users");
    }

    createUser(user) {
      return http.post("/user/save", user);
    }

    deleteUser(id) {
      return http.delete(`/user/delete/id/${id}`);
    }
}

export default new UserDataService();
```

`UserDataService` defines three methods: `getAllUsers, createUser`, and `deleteUser`. The Axios `get`, `post`, and `delete` methods are called corresponding to the HTTP GET, POST, and DELETE methods to make a CRUD operation.

Create React Components Corresponding to Routes

Create three components corresponding in the `src/components/` subfolder to the three routes defined before.

Home Component

Let's create the Home component, which displays welcome messages along with a navigation bar. Listing 5-10 shows the code for the home component.

Listing 5-10. src/components/home.component.js

```
import React, { Component } from "react";

export default class Home extends Component {
    render() {
        return (
            <div class="container">
                <div class="panel panel-default">
                    <div class="alert alert-success">
                        <span class="lead">
                            Welcome to User Registration App
                        </span>
                    </div>
                    <div class="panel-body ">
                        <div class="alert alert-info">
                            <ul>
                                <li>
                                    Please click on
                                    <strong> List All Users
                                    </strong>
                                    to get all users.
                                </li>
                                <li>
                                    Please click on
                                    <strong> Add User </strong>
                                    to add a new user.
```

```
                                    </li>
                                </ul>
                            </div>
                        </div>
                    </div>
                </div>
            );
        }
}
```

In this component, you create a `Home` class that extends the `Component` class, which contains a `render()` method that returns HTML code containing a welcome message.

When you save this home component file, the content on the browser is automatically refreshed. The result in the browser is shown in Figure 5-12.

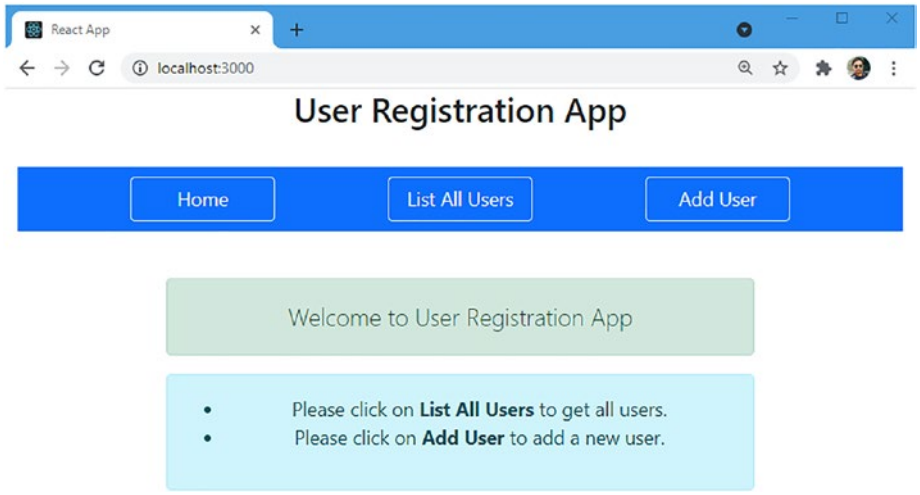


Figure 5-12. *User registration app home page*

Add Users Component

Let's create another component to add a new user in the components. This component has a form to submit a new user with four fields: First Name, Last Name, Age, and Address. Listing 5-11, 5-12, 5-13 and 5-14 shows the pieces of code for the `add-user` component.

Listing 5-11. Imports in src/components/add-user.component.js

```
import React, { Component } from "react";
import userRegistrationService from "../services/user-
registration.service";
```

Here, we have imported React and Component from `"react"` and `user-registration-service`.

Listing 5-12. Constructor and State in AddUser Class in src/components/add-user.component.js

```
export default class AddUser extends Component {

    constructor(props) {
      super(props);

      this.onChangeFirstName = this.onChangeFirstName.
      bind(this);
      this.onChangeLastName = this.onChangeLastName.bind(this);
      this.onChangeAge = this.onChangeAge.bind(this);
      this.onChangeAddress = this.onChangeAddress.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
      this.newUser = this.newUser.bind(this);

      this.state = {
        id: null,
        firstName: "",
```

```
        lastName: "",
        age: "",
        address: "",
        createdDate: ""
      };

    }
```

In the preceding code, the `AddUser` class extends components. The constructor of this class sets the initial state for `id, firstName, lastName, age, address,` and `createdDate` with a default value. Also, we bound it to different events, such as `onChangeFirstName`, `handleSubmit`, and so on.

Listing 5-13. Functions in AddUser Class in src/components/add-user.component.js

```
    onChangeFirstName(event) {
        this.setState({
            firstName: event.target.value
        });
    }

    onChangeLastName(event) {
        this.setState({
            lastName: event.target.value
        });
    }

    onChangeAge(event) {
        this.setState({
            age: event.target.value
        });
    }
```

```
    onChangeAddress(event) {
        this.setState({
            address: event.target.value
        });
    }

    handleSubmit(event) {
        console.log(this.state)

        var data = {
            firstName: this.state.firstName,
            lastName: this.state.lastName,
            age: this.state.age,
            address: this.state.address
        };

        event.preventDefault();

        userRegistrationService.createUser(data)
        .then(response => {
            alert('You submitted successfully! ' + data.
            firstName + ' User is created');
            this.setState({
                id: response.data.id,
                firstName: response.data.firstName,
                lastName: response.data.lastName,
                age: response.data.age,
                address: response.data.address
            });
            this.props.history.push("/list-all-users");
        })
```

```
        .catch(e => {
            console.log(e);
        });
    }

    newUser() {
        this.setState({
            id: null,
            firstName: "",
            lastName: "",
            age: "",
            address: "",
            createdDate: ""
        });
    }
```

Four functions (`onChangeFirstName`, `onChangeLastName`, `onChangeAge`, `onChangeAddress`) are created to track the input value and set the state for changes. A function named `handleSubmit` is defined to get the value of the form (state) and call the `createUser()` method of `userRegistrationService`, which internally sends HTTP POST requests to the REST API.

Listing 5-14. Render Method to Return HTML Code

```
    render() {
        return (
            <div className="submit-form">
                <div className="form-group">
                    <label htmlFor="firstName">First Name
                    </label>
                    <input
                        type="text"
```

```
                    className="form-control"
                    id="firstName"
                    required
                    value={this.state.firstName}
                    onChange={e => this.
                    onChangeFirstName(e)}
                    name="firstName"
                />
            </div>

            <div className="form-group">
                <label htmlFor="lastName">Last Name</label>
                <input
                    type="text"
                    className="form-control"
                    id="lastName"
                    required
                    value={this.state.lastName}
                    onChange={e => this.
                    onChangeLastName(e)}
                    name="lastName"
                />
            </div>

            <div className="form-group">
                <label htmlFor="age">Age</label>
                <input
                    type="text"
                    className="form-control"
                    id="age"
                    required
                    value={this.state.age}
```

```
                        onChange={e => this.onChangeAge(e)}
                        name="age"
                    />
                </div>

                <div className="form-group">
                    <label htmlFor="address">Address</label>
                    <input
                        type="text"
                        className="form-control"
                        id="address"
                        required
                        value={this.state.address}
                        onChange={e => this.onChangeAddress(e)}
                        name="address"
                    />
                </div>

                <button onClick={this.handleSubmit}
                className="btn btn-success">
                Submit
                </button>
            </div>
        )
    }
}
```

Here, the render method results in UI. `AddUser` contains input boxes for the first name, last name, age, and address, and it contains the Submit button for creating a new user, as shown in Figure 5-13.

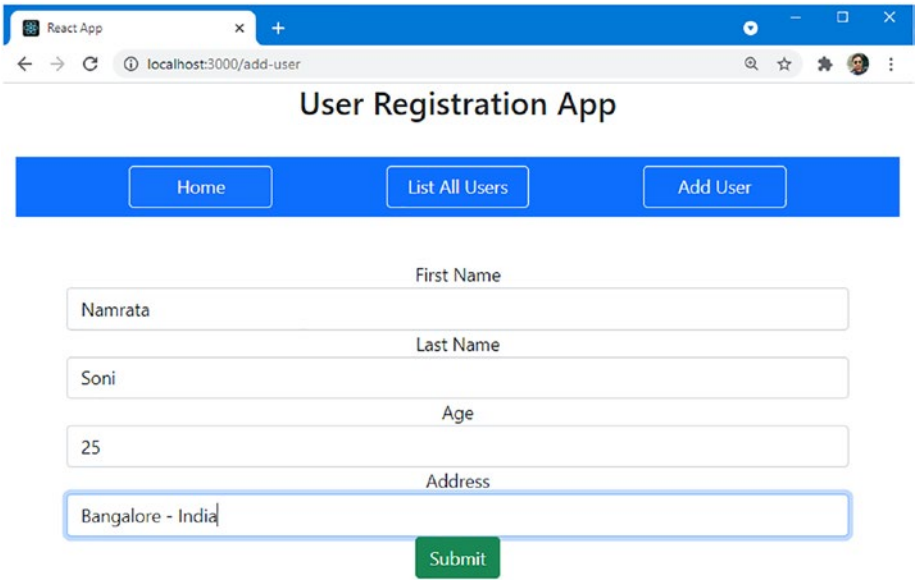


Figure 5-13. *Page to add new user*

List All Users Component

Let's create another component to list all the users in the `components` subfolder. This component has a *user array* to display a list of users in the table, and each row has a Delete button to delete specific users from the list. Listing 5-15 and 5-16 shows the pieces of code for the `list-user` component.

Listing 5-15. Imports, Constructor, State, and Functions in UsersList Class in src/components/list-users.component.js

```
import React, { Component } from "react";
import UserDataService from '../services/user-registration.
service';

export default class UsersList extends Component {
    constructor(props) {
      super(props);
```

```
  this.retrieveUsers = this.retrieveUsers.bind(this);
  this.deleteUser = this.deleteUser.bind(this);
  this.state = {
    users: []
  };
}

componentDidMount() {
    this.retrieveUsers();
}

retrieveUsers() {
    UserDataService.getAllUsers()
        .then(response => {
            this.setState({
            users: response.data
            });
            console.log(response.data);
        })
        .catch(e => {
            console.log(e.target);
        });
}

deleteUser(user, index) {
    UserDataService.deleteUser(user.id)
        .then(response => {
            alert('Deleted successfully! ' + user.
            firstName);
            this.retrieveUsers();
        })
```

```
            .catch(e => {
                console.log(e.target);
            });
    }
```

The `UsersList` class extends the `Components` class. `React, Component,` and `user-registration-service` import as `UserDataService`. We defined the constructor of this class that sets the initial state for the users array. Also, we bound this to the different events such as `retrieveUsers` and `deleteUser`.

The `retrieveUsers` function is defined to get the list of users by calling the `getAllUsers()` method of `UserDataService`, which internally sends HTTP GET requests to the REST API. A function named `deleteUser` is defined to delete users by calling the `deleteUser()` method of `UserDataService`, which internally sends HTTP DELETE requests to the REST API. The `componentDidMount()` method immediately executes the React code after a component is mounted (placed in the DOM).

Listing 5-16. Render Method to Return HTML Code

```
    render() {
        const { users } = this.state;

        return (
                <table class="table table-hover">
                    <caption>List of users</caption>
                    <thead class="thead-dark">
                    <tr>
                        <th scope="col">#</th>
                        <th scope="col">First Name</th>
                        <th scope="col">Last Name</th>
                        <th scope="col">Age</th>
                        <th scope="col">Address</th>
                        <th scope="col">Delete</th>
```

```
                    </tr>
                    </thead>
                    <tbody>
                        {users && users.map((user, index) => (
                            <tr>
                                <th scope="row">{index+1}</th>
                                <td>{user.firstName}</td>
                                <td>{user.lastName}</td>
                                <td>{user.age}</td>
                                <td>{user.address}</td>
                                <td>
                                    <button type="button"
                                        onClick={() => this.
                                        deleteUser(user,
                                        index)}
                                        class="btn btn-danger
                                        custom-width"
                                        key={index}
                                    >
                                        <span class="glyphicon
                                        glyphicon-remove">
                                            Delete
                                        </span>
                                    </button>
                                </td>
                            </tr>
                        ))}
                    </tbody>
                </table>
        );
    }
}
```

The `render` method results in a UI. The List of Users page displays a user list in a table. It also contains a Delete button for each user's row in a table, as shown in Figure 5-14.

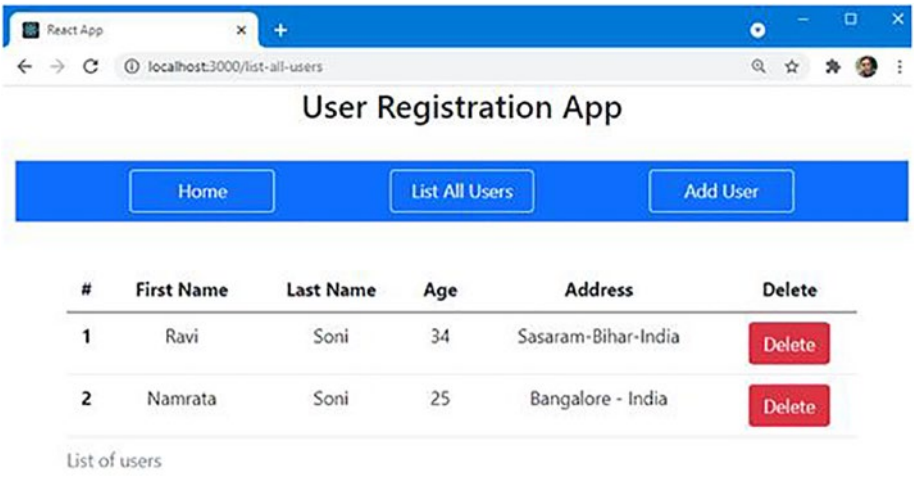


Figure 5-14. *List all users along with a delete user option*

Even though you added only one user in the previous section, the list shows two users. It's because the database already contains one user that was added in Chapter 4.

Here, clicking the Delete button deletes a specific user, as shown in Figure 5-15.

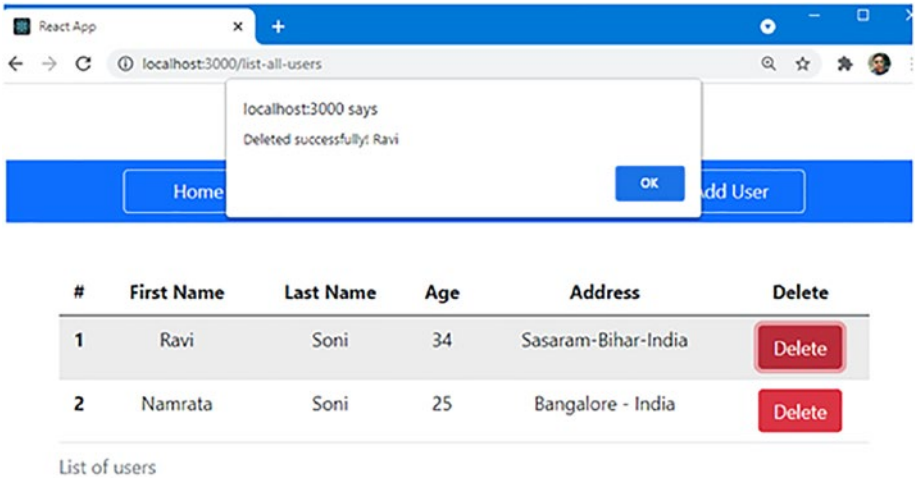


Figure 5-15. *Delete an existing user*

After successfully deleting a specific user, the table displays an updated user list, as shown in Figure 5-16.

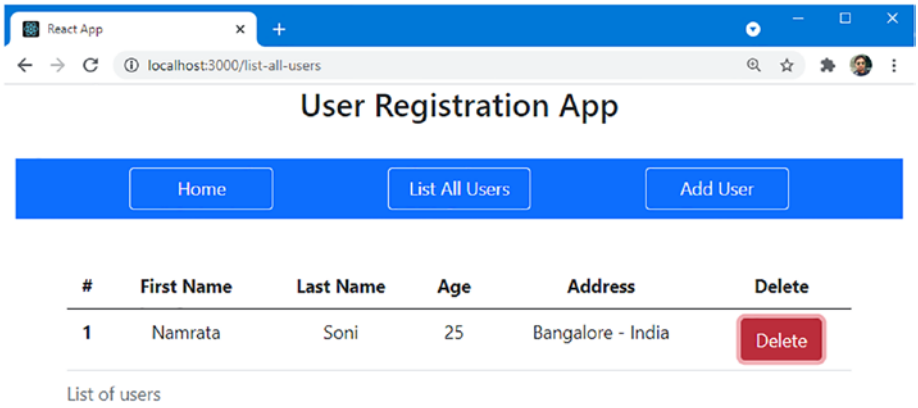


Figure 5-16. *Updated user list after the delete operation*

Build React Code as a Front-end Application for AWS

You have successfully developed and run a user registration front-end app using React with CRUD features in your local system that consumes data from `UserRegistrationApp` RESTful web services that also run in the local system. To deploy the React app to AWS, you need to build React code.

Verify the AWS Elastic Beanstalk Environment Is Up

You have updated the Spring Boot application, which should be deployed to Elastic Beanstalk. You already learned about the deployment process of the back-end application, so you need to follow the same here to complete the deployment of the `UserRegistrationApp` Spring Boot application. Once you have successfully deployed the updated code, you need to verify that the Elastic Beanstalk environment is up, as shown in Figure 5-17.

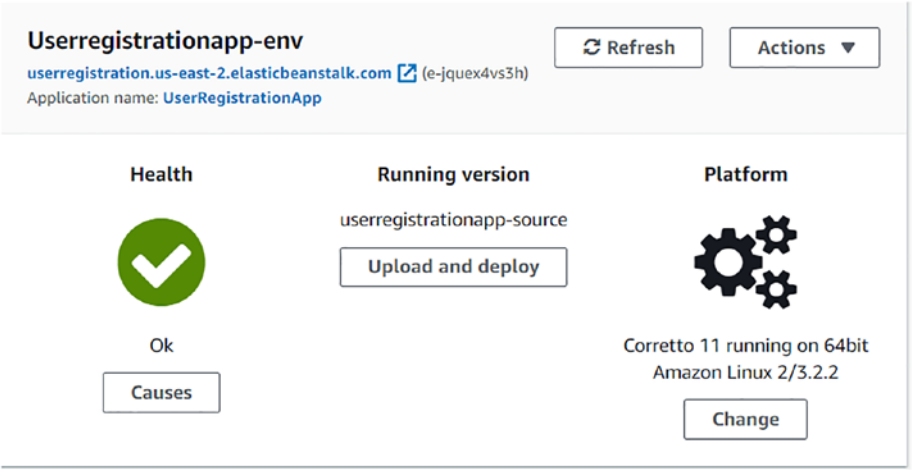


Figure 5-17. *Verify that the Elastic Beanstalk environment is up*

Update BaseURL in a React App with an AWS Elastic Beanstalk Environment URL

We provided the localhost URL of the RESTful app in the React front-end app in the `src/http-common.js` file so that Axios can make a REST API call from the front end to the back end.

Now, the React front-end app should interact with the RESTful web services deployed in Elastic Beanstalk. To achieve this, open the `src/http-common.js` file and update the base URL with the Elastic Beanstalk environment URL, as shown in Listing 5-17.

Listing 5-17. src/http-common.js

```
import axios from "axios";

export default axios.create({
  //baseURL: "http://localhost:5000/api/",
  baseURL: "http://userregistration.us-east-2.elasticbeanstalk.
  com/api/",
  headers: {
    "Content-type": "application/json"
  },

});
```

Before building, let's verify the changes locally. Once you access the List All Users page in the browser, you can see the result from AWS, as shown in Figure 5-18.

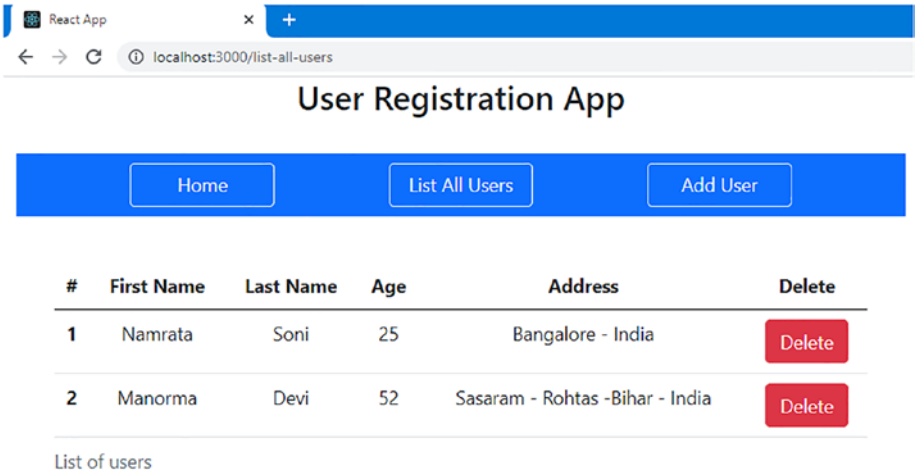


Figure 5-18. *React app interact with RESTful web services deployed in Elastic Beanstalk*

To cross verify the changes, open Developers Tools in a browser and validate the request URL, as shown in Figure 5-19, for the POST method to create a new user.

Figure 5-19. *Validate the Request URL in browser Developer Tools*

Build React Code for AWS Deployment

You have made the required changes in the React app and verified those changes to confirm that the React app interacts with RESTful web services deployed in AWS. Now, you would like to deploy this React front-end app to the AWS server. You need to create a build for the React app.

To create a build, you need to stop the React app and execute the following `npm` command in the command prompt.

```
E:\Apress\workspace\AWS\user-registartion-frontend-app>npm run
build
```

Once you run the build command, a folder named `build` is created in the React app, and it is populated with an optimized production build, as shown in Figure 5-20.

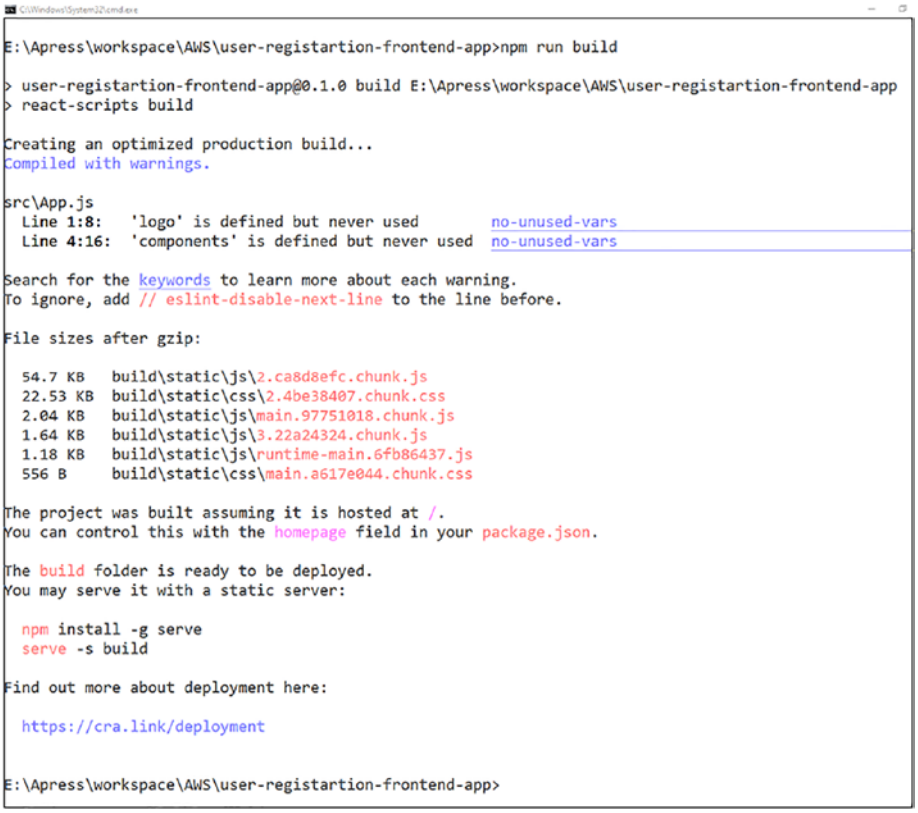


Figure 5-20. *Build React app using npm command*

So, now the `build` folder is ready. It contains a `static` folder and the `asset-manifest.json`, `fevicon.ico`, `index.html`, `manifest.json`, `logo.png`, and `robots.txt` files, as shown in Figure 5-21.

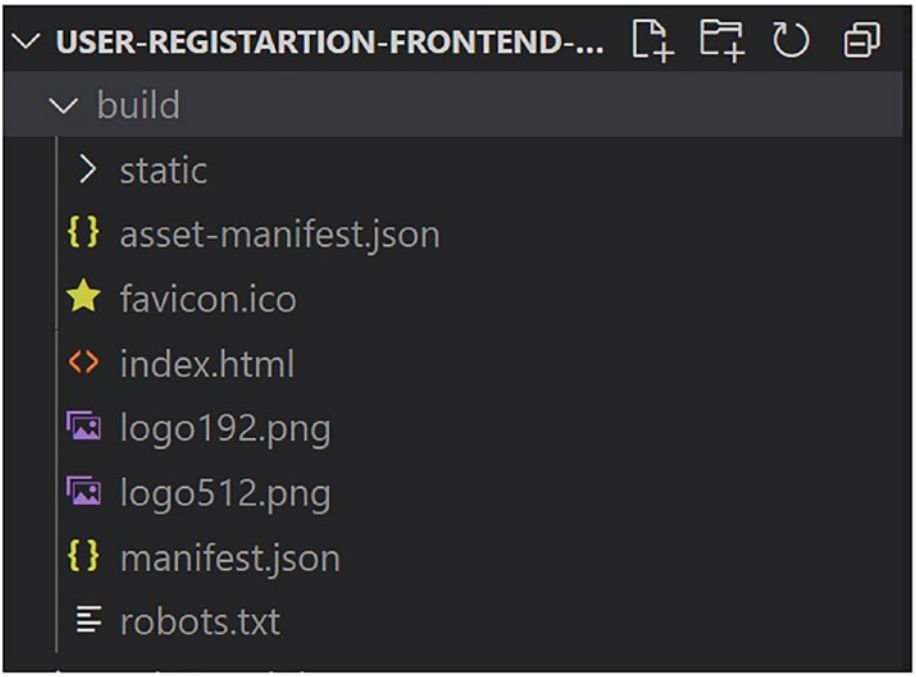


Figure 5-21. *The build folder in React app*

Deploy a React Front-End to AWS S3: Hosting a Static Website

In the previous section, you built a React front-end app that you want to deploy in AWS S3.

Introduction to S3: Simple Storage Service in AWS

S3 stands for Simple Storage Service, which is scalable storage in the cloud. S3 is basically an object-store.

Log in to AWS Console Management, and click the **All service** hyperlink at the top, and you find S3 under the Storage category, as shown in Figure 5-22.

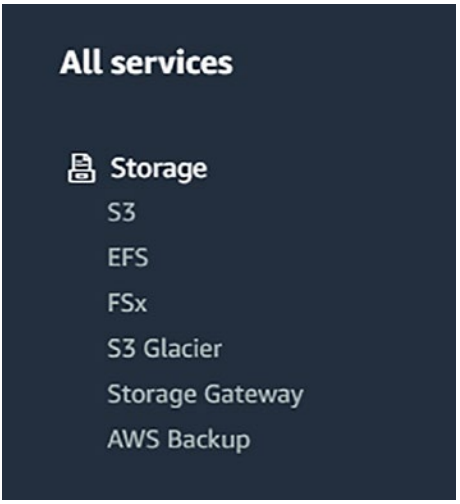


Figure 5-22. *S3 service under Storage category on AWS*

Clicking S3 brings you to the page containing the bucket's details, as shown in Figure 5-23.

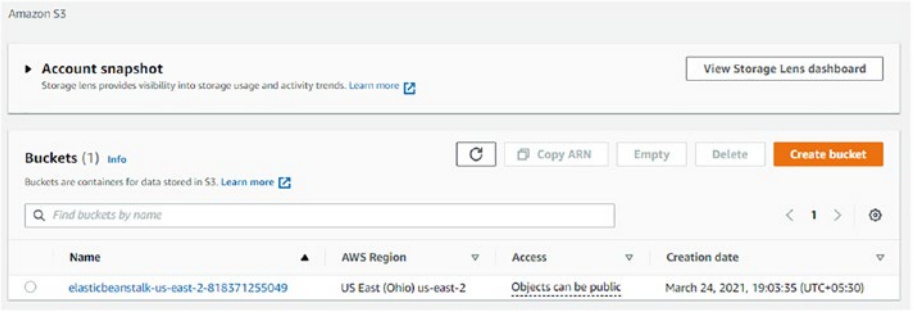


Figure 5-23. *Buckets details on Amazon S3*

A *bucket* is a collection of objects that are files belonging to that container. Figure 5-23 shows that a bucket is available in Amazon S3.

Figure 5-24 shows that this bucket contains all the JARS that you deployed in previous chapters, as shown in.

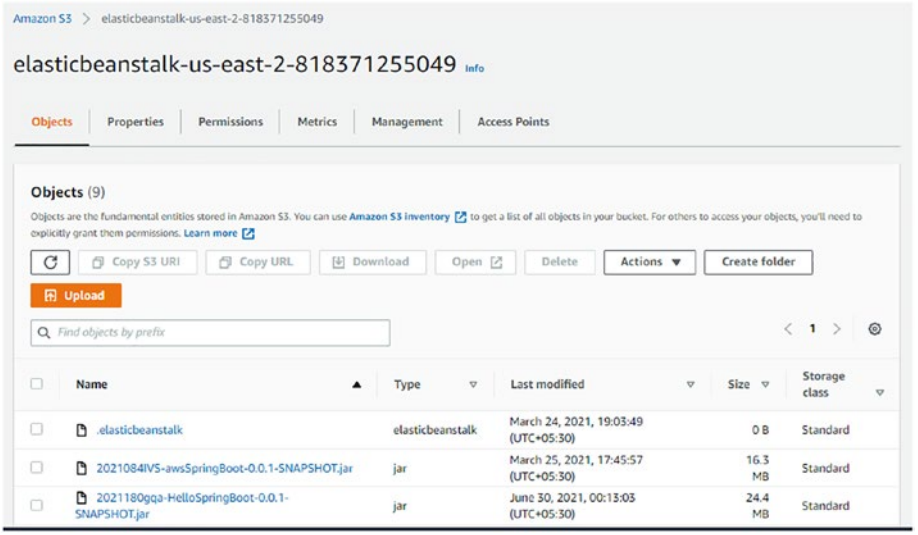


Figure 5-24. *Bucket contains JARs*

AWS fetches all the required JARs from S3, which you can think of as primarily a storage service in AWS. If you want to store something like a backup file, archival file, data staging, or logs file, you use S3 in AWS.

S3 can also serve static websites, and that is the feature which you deploy React applications. S3 provides high durability and high availability.

While buckets are associated with regions, when you use S3, you are in a global space that means a global service, and you are not really selecting a region, as shown in Figure 5-25.

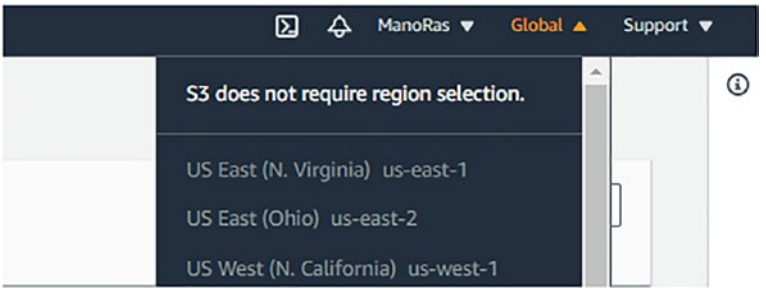


Figure 5-25. *Selecting S3 means global service*

Next, you deploy the React app in AWS S3.

Create a Bucket

Open the **Create bucket** page, as shown in Figure 5-26.

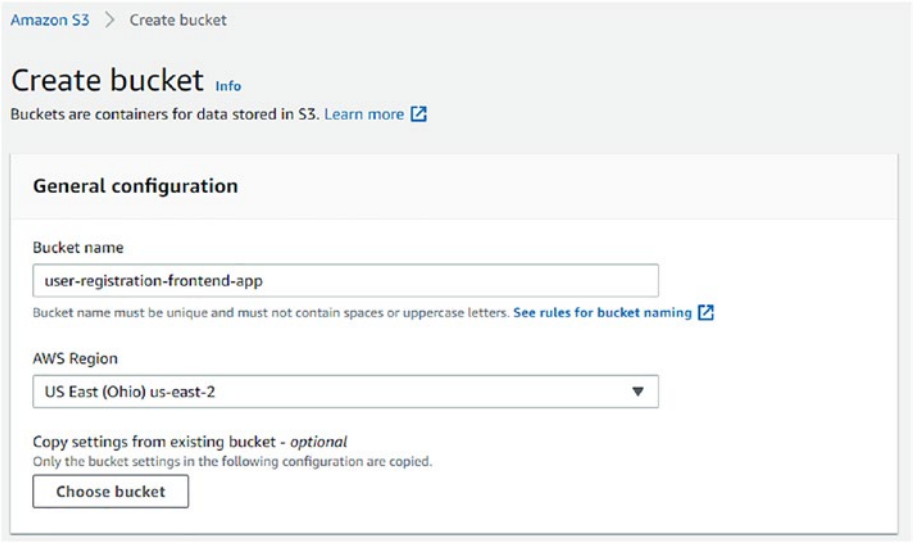


Figure 5-26. *Creating UserRegistrationApp using Spring Initializr*

Here, you need to provide general configuration information. While entering the bucket name, across AWS, the Bucket name should be globally unique. Enter **`user-registration-frontend-app`** in Bucket name, leave the other options on the page as is, and then click the **Create bucket** button. You should get a success message, as shown in Figure 5-27.

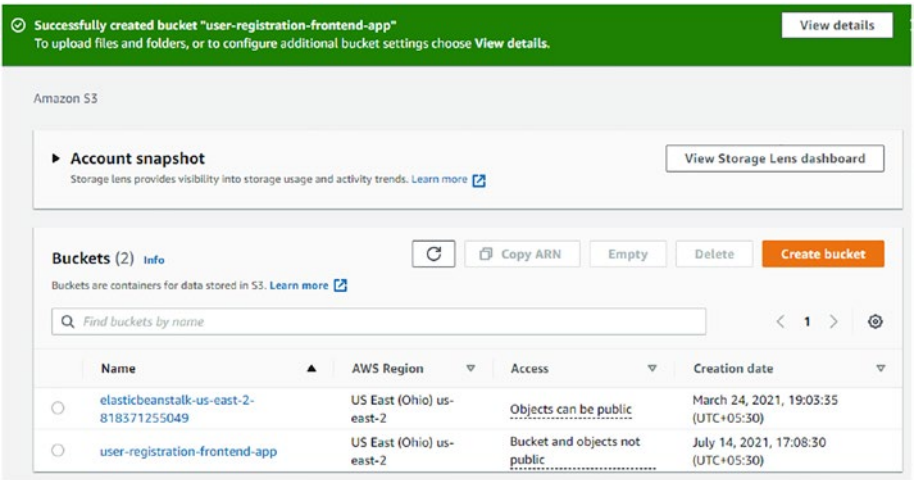


Figure 5-27. *Creating UserRegistrationApp using Spring Initializr*

Here, you can see that two buckets were created in AWS S3. Click the newly created bucket named `user-registration-frontend-app`, which takes you to `user-registration-frontend-app`, as shown in Figure 5-28.

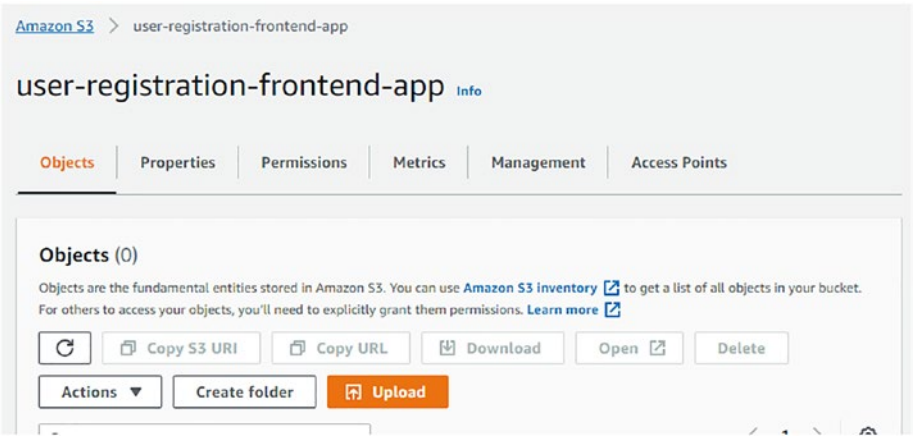


Figure 5-28. *user-registration-frontend-app with object details*

Here, the objects are empty because it is a newly created bucket. Click the Upload button to upload all the content from the local system in the `build` folder, as shown in Figure 5-29.

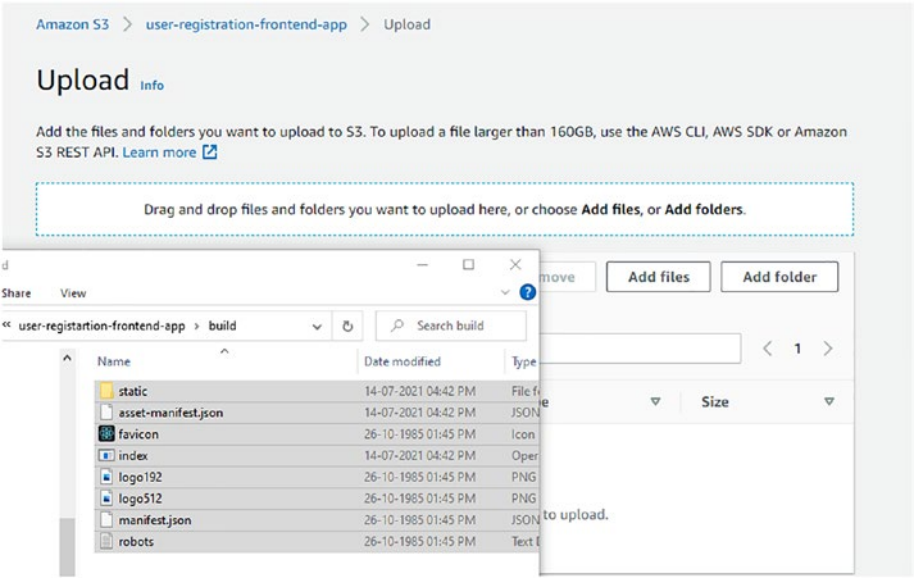


Figure 5-29. *Upload files in build folder to S3 bucket*

Next, click the Upload button at the bottom of the page. Once the files are uploaded successfully, you get a success message, as shown in Figure 5-30.

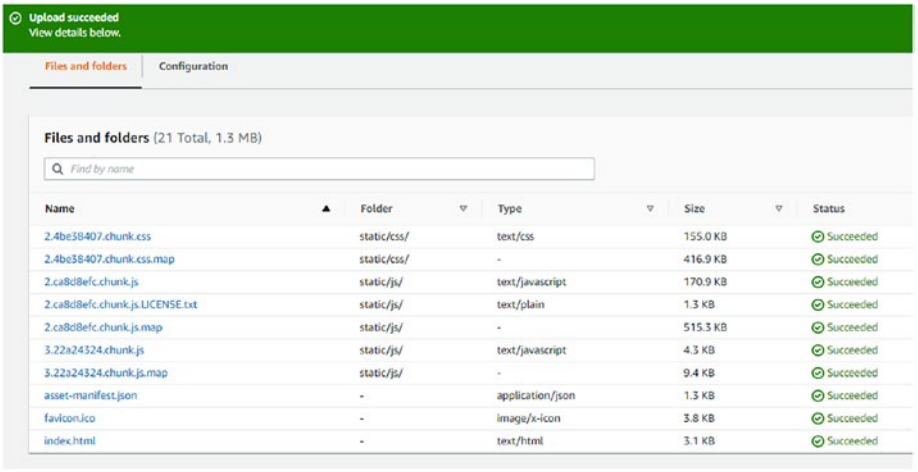


Figure 5-30. *Uploaded files and folder to AWS S3*

Now, under the Objects tab, you see all the objects present in the `user-registration-frontend-app` bucket. Figure 5-31 shows the `static` folder and all the files you have uploaded to the bucket.

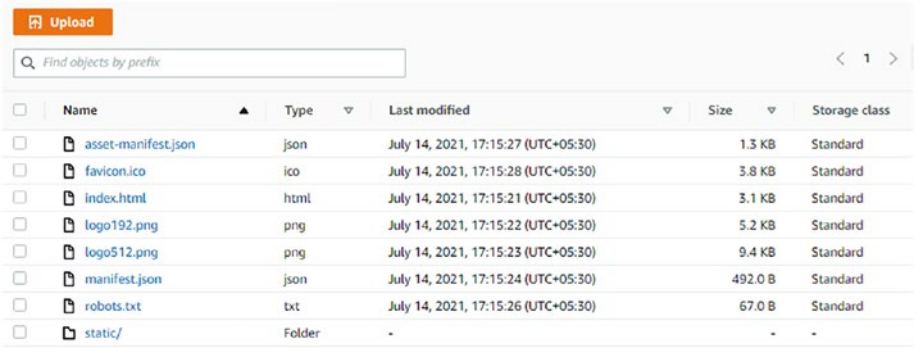


Figure 5-31. *Creating UserRegistrationApp using Spring Initializr*

To host a website, go to the Properties tab, scroll down to **Static website hosting**, and then click Edit, as shown in Figure 5-32.

Figure 5-32. *Static website hosting*

Next, select **Enable** for static website hosting, select **Host a static website** as the hosting type, and enter index.html as the index document, as shown in Figure 5-33.

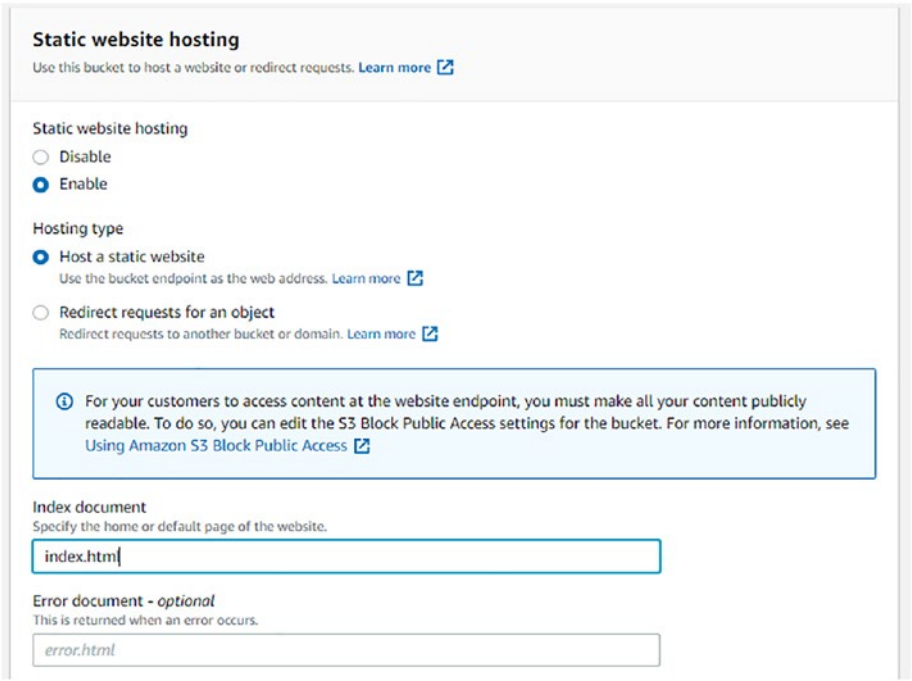


Figure 5-33. *Update static website hosting details*

The index.html file was uploaded to the S3 bucket. Save the changes. Now you can find the bucket website endpoint URL in the Properties tab, as shown in Figure 5-34.

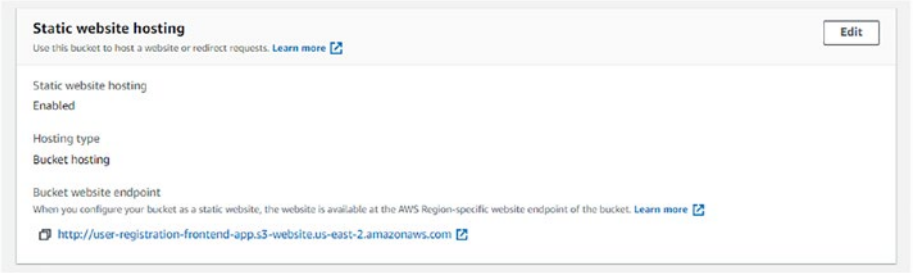


Figure 5-34. *Bucket website endpoint URL*

Clicking the bucket website endpoint URL gives a 403 Forbidden error, as shown in Figure 5-35.

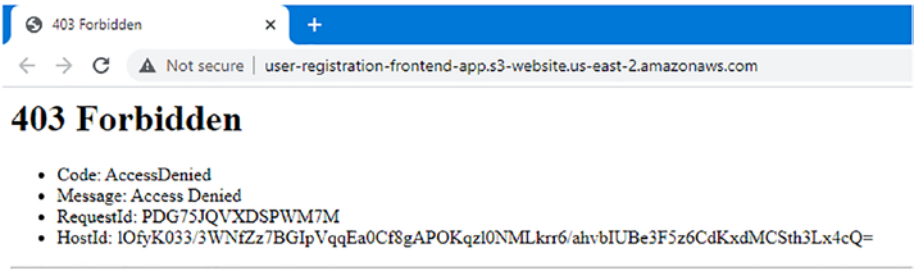


Figure 5-35. *Creating UserRegistrationApp using Spring Initializr*

The Access Denied error is due to S3 security issues. By default, all the objects you have uploaded have **Block public access** in the Permissions tab, as shown in Figure 5-36.

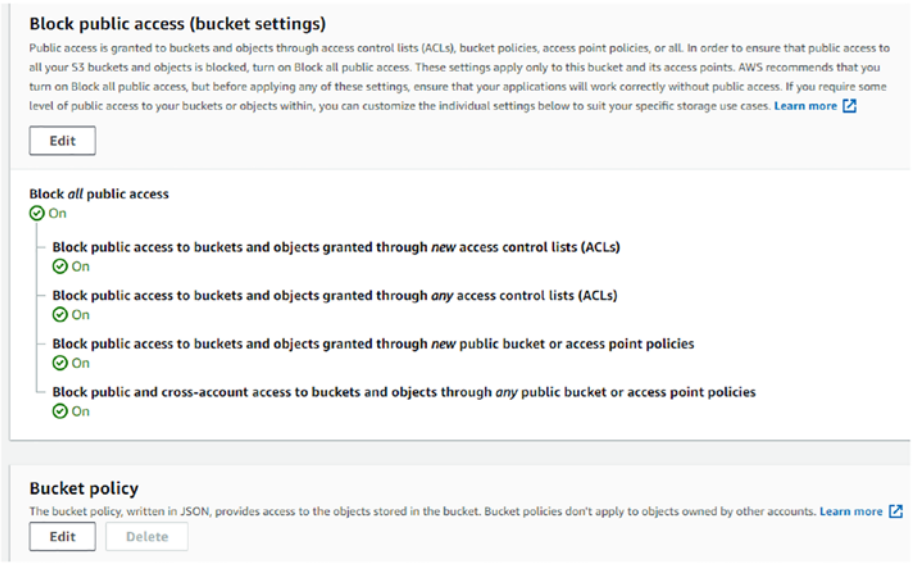


Figure 5-36. *By default, all objects block public access*

To make all the bucket's content public so that it is accessible on the Internet, click **Block public access**, uncheck **Block all public access**, and click **Save changes**, as shown in Figure 5-37.

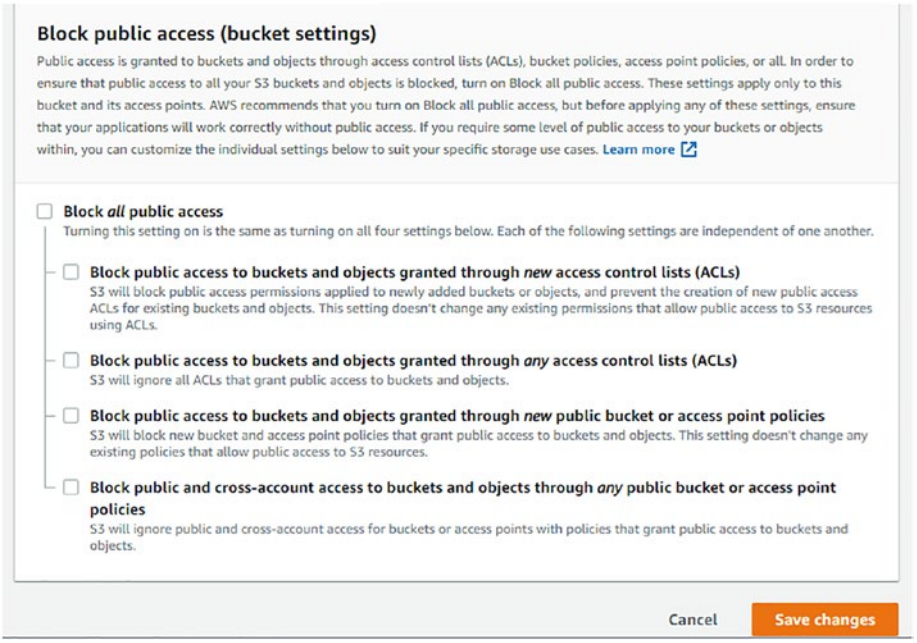


Figure 5-37. *Creating UserRegistrationApp using Spring Initializr*

A confirmation screen pops up to confirm the settings. You need to enter **confirm** in the input box and click the Confirm button, as shown in Figure 5-38.

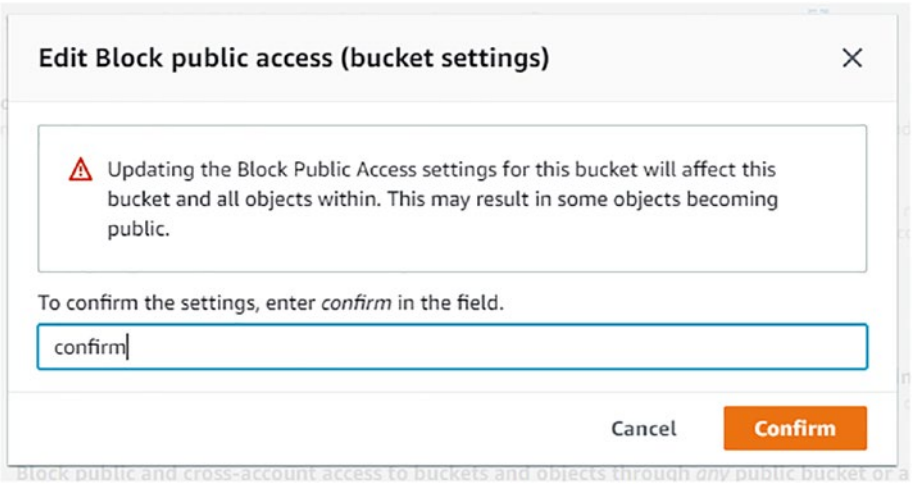


Figure 5-38. *To confirm the settings, enter confirm in the field*

A success message should appear, as shown in Figure 5-39.

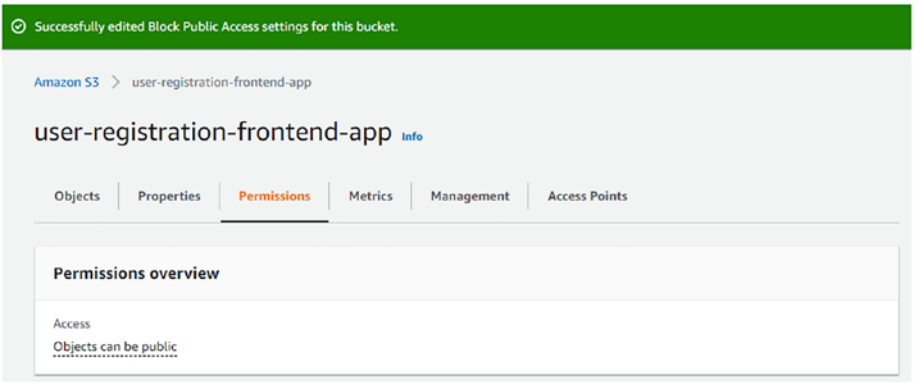


Figure 5-39. *Successfully edited Block Public Access settings for bucket*

Now, you need to edit the bucket policy, which is written in JSON. It provides access to the objects stored in the bucket. To edit bucket policy, in the Permissions tab, scroll down to the **Bucket policy** section, and click the Edit button, and enter the JSON under Policy, as shown in Figure 5-40.

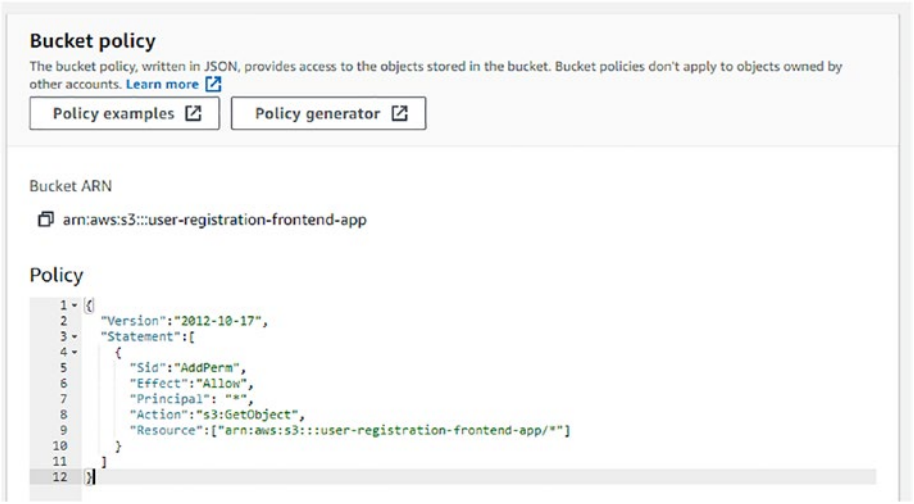


Figure 5-40. *Update bucket policy*

Listing 5-18 shows the JSON for a bucket policy.

Listing 5-18. JSON for Bucket Policy

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"AddPerm",
      "Effect":"Allow",
      "Principal": "*",
      "Action":"s3:GetObject",
      "Resource":["arn:aws:s3:::user-registartion-frontend-
      app/*"]
    }
  ]
}
```

`Resource` contains the bucket name, which is `user-registration-frontend-app`, to identify the resource for the bucket policy. This JSON specifies a specific version. `GetObject` in `Action` allows access to all principals. All users can execute `GetObject` on `user-registration-frontend-app`.

Next, save the changes, which prompts a message stating, "This bucket has public access." Refresh the browser with the bucket website endpoint URL. You can now access your home page, as shown in Figure 5-41.

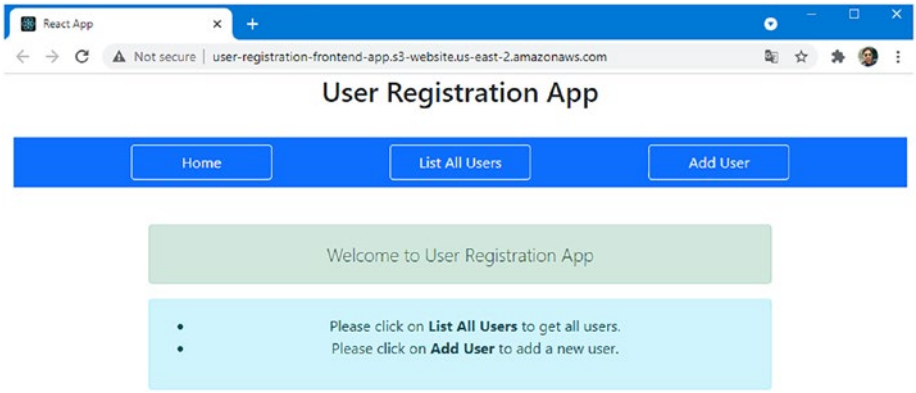


Figure 5-41. *Bucket website endpoint URL in home page*

Congratulations! You have successfully hosted your static React app in AWS S3 and can access the home page.

Verify the Successful Deployment of a React Front-end Application: Resolve a 404 Error

Click the List All Users button in the navigation bar on the home page. You get 404 Not Found errors, as shown in Figure 5-42.

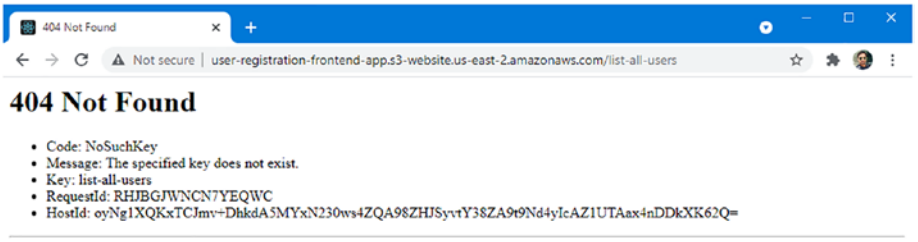


Figure 5-42. *List All Users page throws 404 error*

To resolve this issue, you need to update the **Error document** box to `index.html`. To make these changes, you need to go to the Properties tab under the bucket. Scroll down to **Static website hosting**, click Edit, and update the error document, as shown in Figure 5-43.

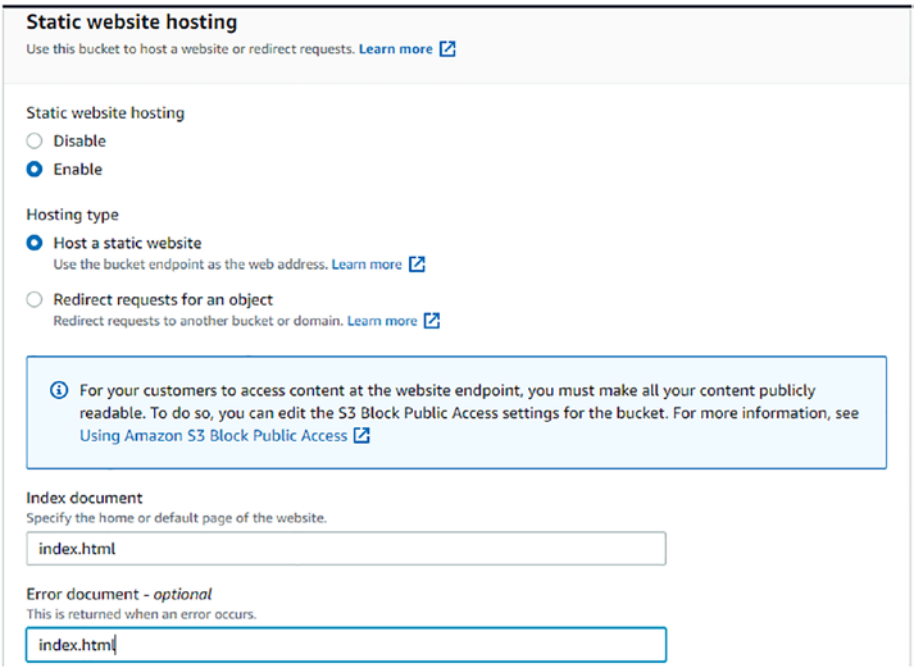


Figure 5-43. *Update Error document in Static Website Hosting*

This is the way `react-router` works. It handles the requests from the front-end and routes users to other routes. Save the changes and refresh the browser to view the List All Users page, as shown in Figure 5-44.

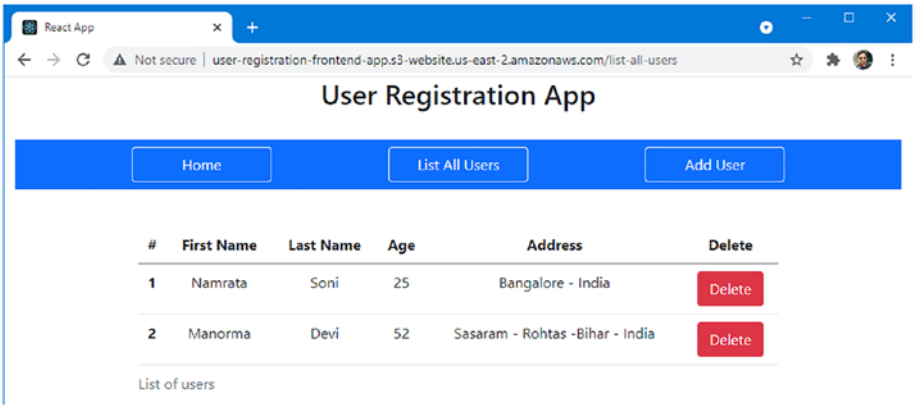


Figure 5-44. *Access list-all-users page hosted on AWS S3*

Summary

This chapter introduced React as a front-end framework and its major components to develop a single-page application using React as the front end to consume the API exposed by the back-end application. You set up a development environment to develop a React front-end application and were introduced to S3 in AWS, where you deployed a React front-end application.

APPENDIX A

Install MySQL Workbench on Windows 10

MySQL Workbench is a visual database designing and modeling access tool used to add functionality and ease to SQL development work. MySQL Workbench facilitates creating new physical data models or modifying existing MySQL databases and provides data modeling, SQL development, and various administration tools for configuration. It also offers a graphical interface to work with MySQL databases in a structured way.

Step 1. Download Workbench

Go to the official MySQL Workbench download site (`https://dev.mysql.com/downloads/workbench/`). You see the options to download

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0

Workbench, as shown in Figure A-1. The MySQL Workbench version that was available when writing this tutorial was 8.0.25.

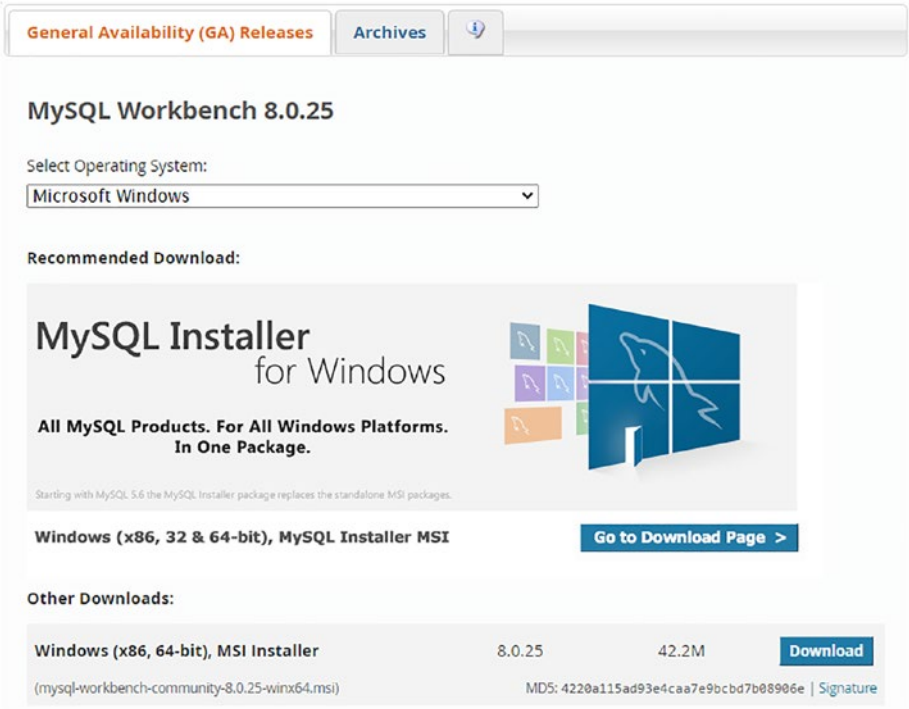


Figure A-1. *MySQL Workbench*

Clicking the Download button takes you to the next page, which asks you to either log in to download or download directly, as shown in Figure A-2.

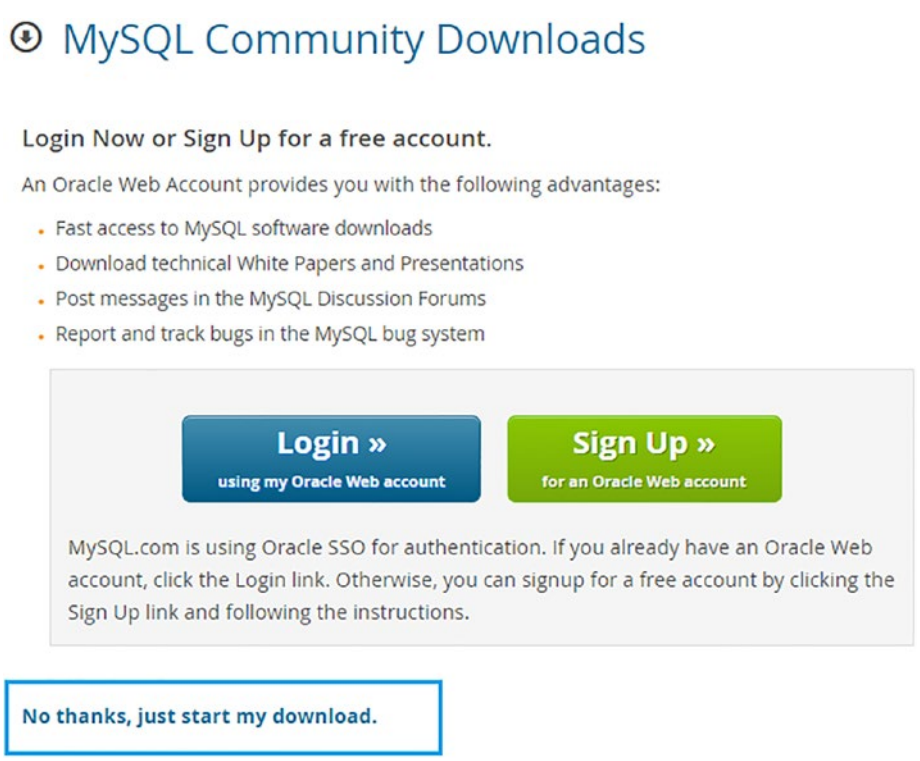


Figure A-2. *MySQL community download*

Complete the MySQL installer download by following either of the approaches.

Step 2. Install Workbench

Double-click the downloaded MySQL Workbench installer to execute it. It shows a "Welcome to the Setup Wizard" screen, as shown in Figure A-3.

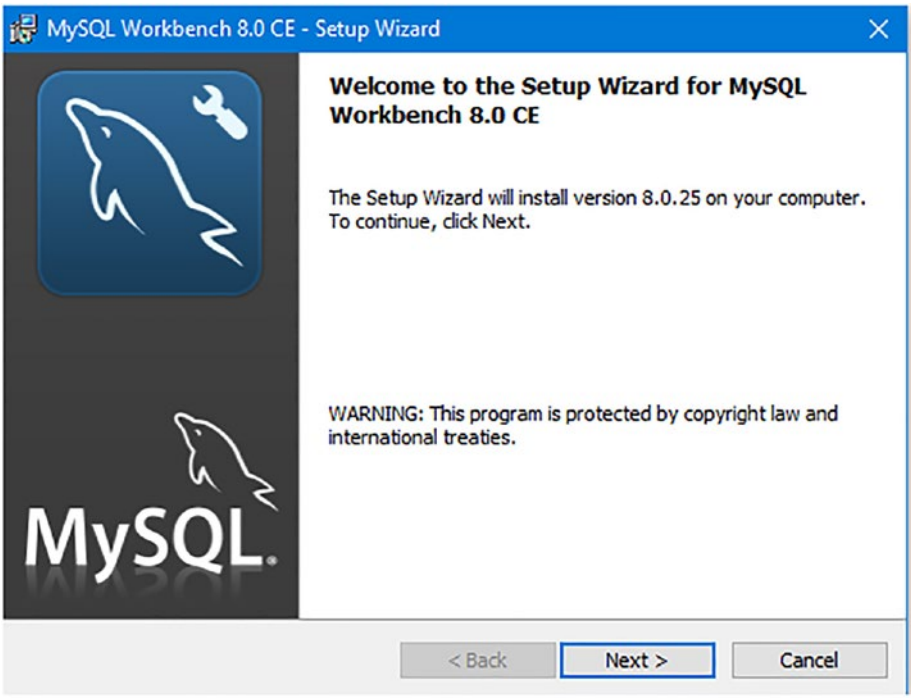


Figure A-3. *Welcome screen*

Click the Next button to continue the MySQL Workbench installation. The following screen asks you for the destination folder, as shown in Figure A-4.

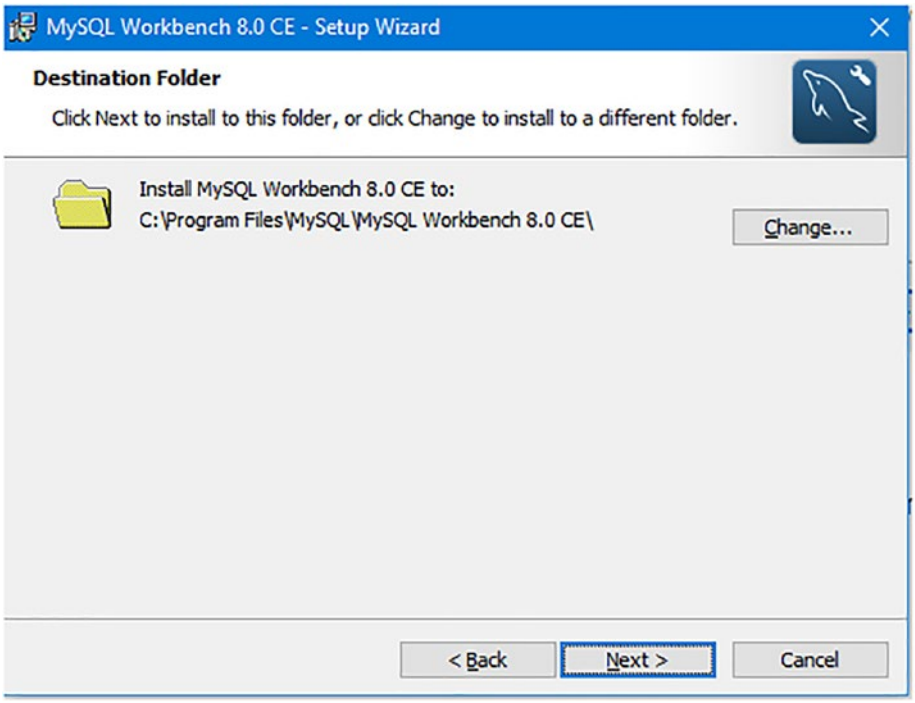


Figure A-4. *Destination folder*

Change the path if required and then click the Next button. The next screen offers the setup type options, as shown in Figure A-5.

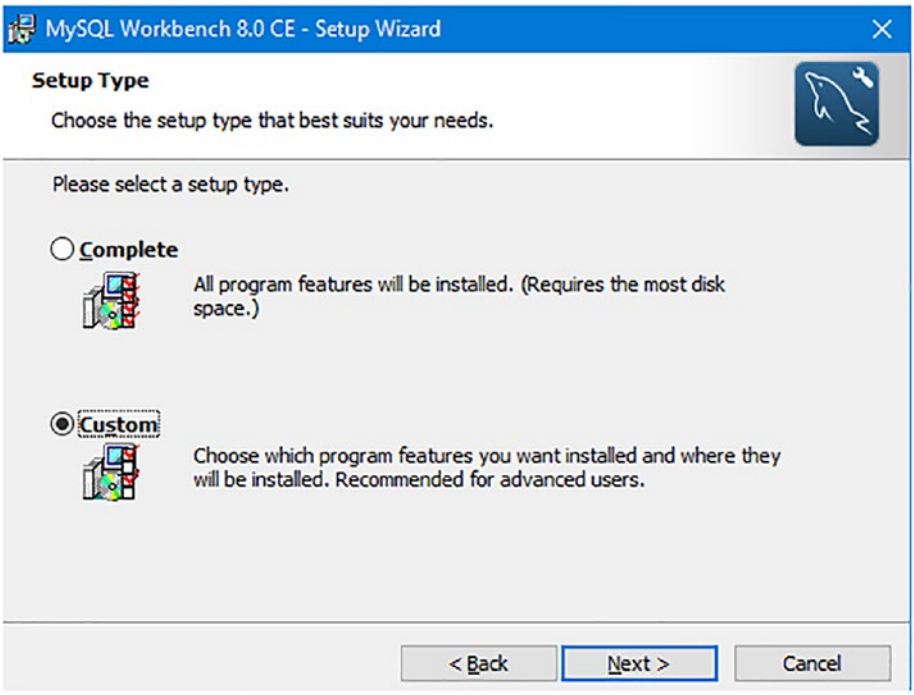


Figure A-5. *Destination folder*

Select the Custom setup type to make changes. Then, click the Next button to view the custom options, as shown in Figure A-6.

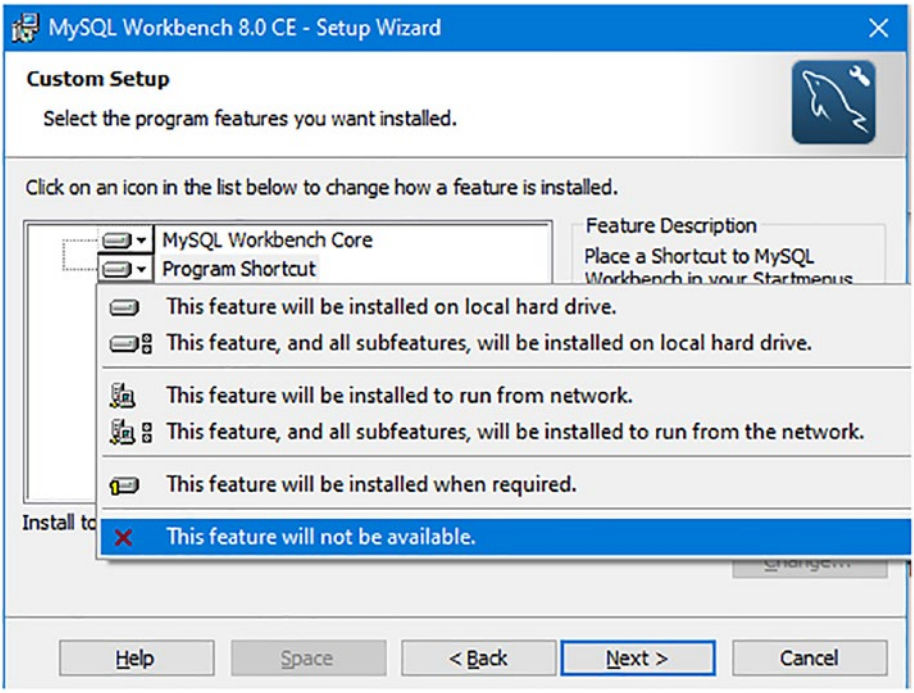


Figure A-6. *Custom setup*

You can omit Program Shortcut by clicking it and selecting **This feature will not be available** (if required). Then, click the Next button to confirm MySQL Workbench installation, as shown in Figure A-7.

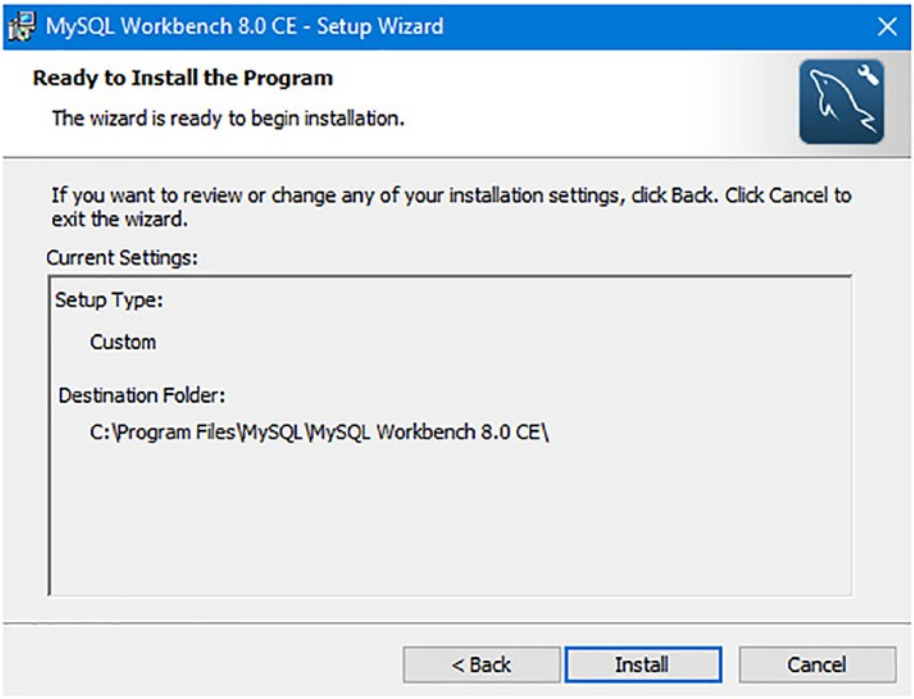


Figure A-7. *Ready to install*

Click the Install button to start the installation. The installer asks for your system's permission. Grant the permissions to allow the installation process. It displays the progress, as shown in Figure A-8.

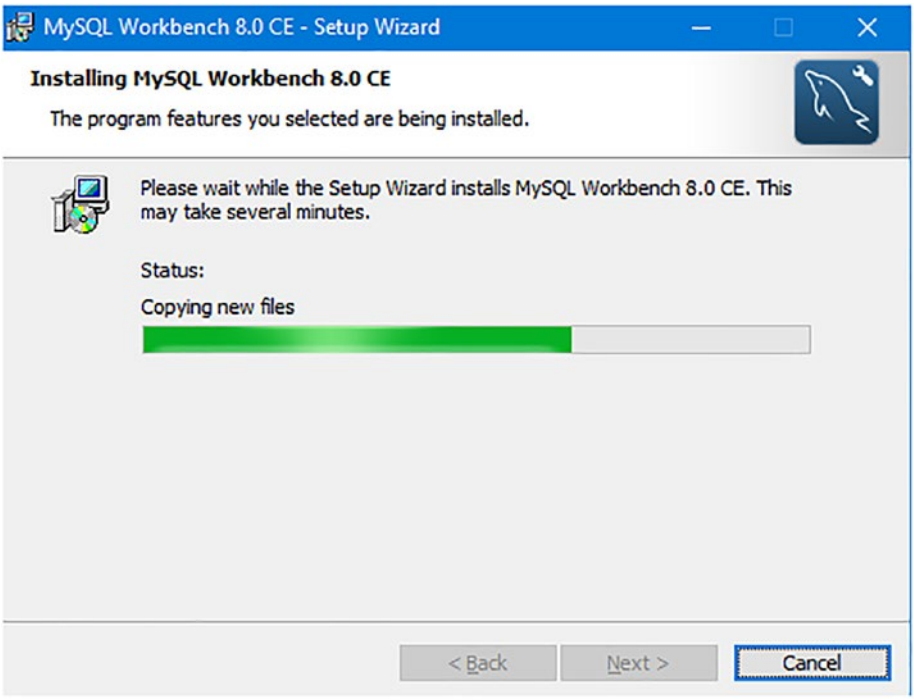


Figure A-8. *Copying new files*

After completing the installation, a Wizard Completed success screen is displayed, as shown in Figure A-9.

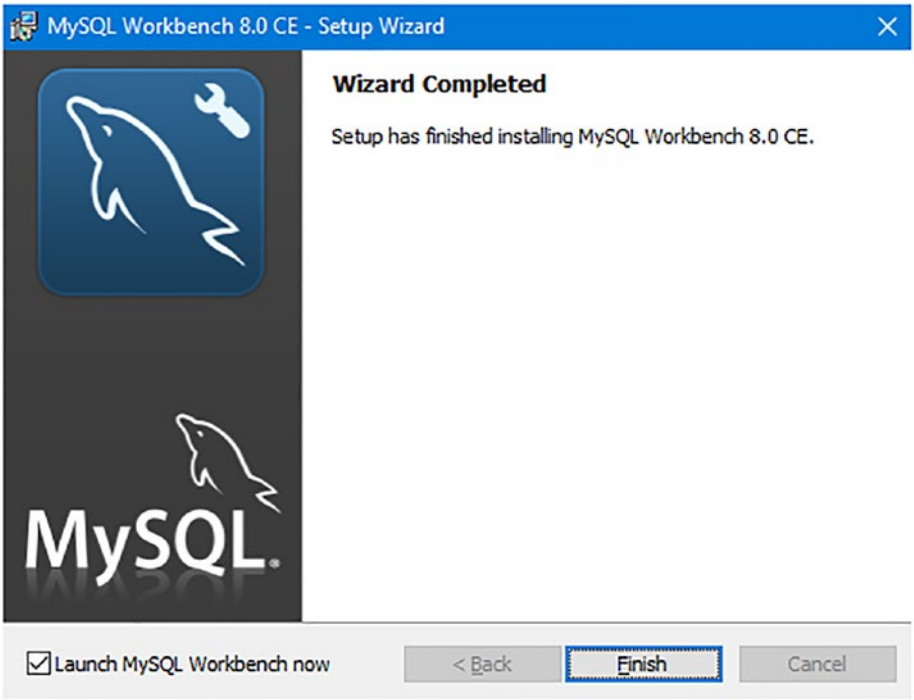


Figure A-9. *Wizard completed*

Once you click the Finish button, the installer starts MySQL Workbench. The default window looks like the one shown in Figure A-10.

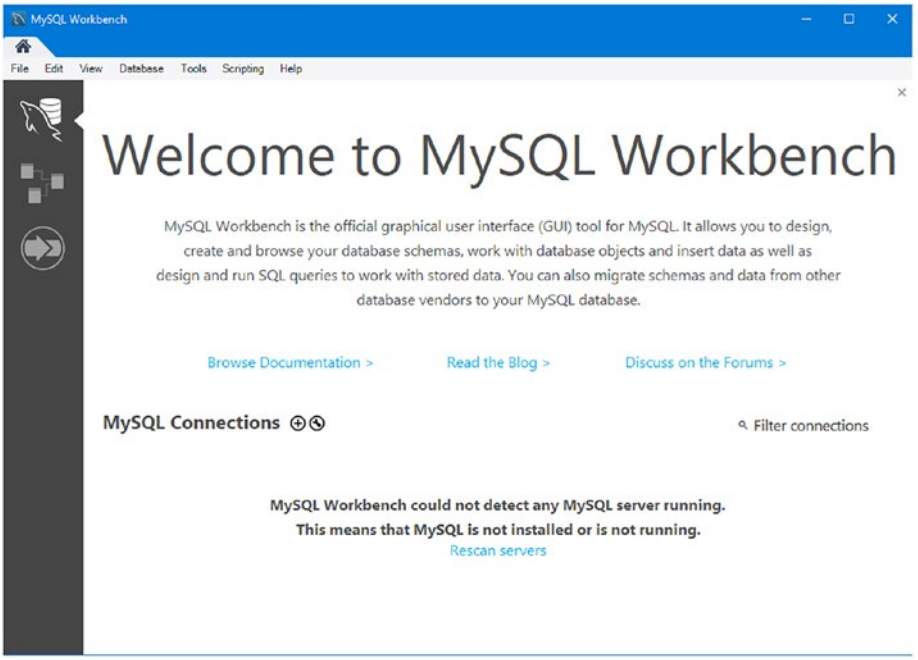


Figure A-10. *Welcome to MySQL Workbench*

Your MySQL server connection contains information about the target database server, including how to connect to it. Click the **+** icon on the MySQL Workbench home window to open the **Setup New Connection** wizard, as shown in Figure A-11.

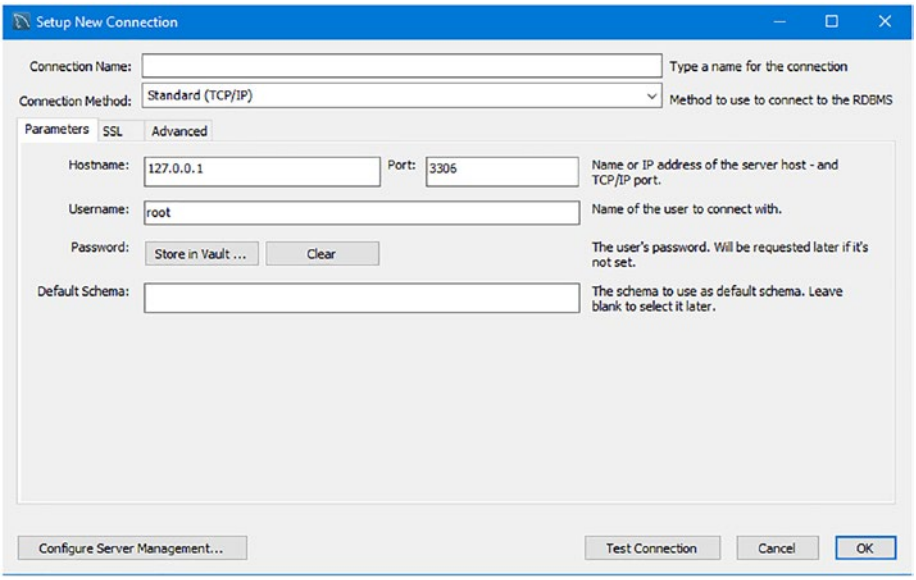


Figure A-11. *Setup New Connection wizard*

APPENDIX B

AWS Command-Line Interface (CLI)

The AWS Command Line Interface (CLI) manages AWS services from a terminal session that allows you to configure and control multiple AWS services by implementing a level of automation without logging in to the AWS Management Console.

Many popular tools, like Terraform, Jenkins, and Python scripts, support CLI access to create infrastructure as code (IAC), which creates the entire infrastructure. For example, if you want to create an S3 bucket in AWS, you don't have to log in to the AWS Management Console and visit different-different pages on AWS to enter lots of details for this bucket creation. Instead, create some code with the required information, like the bucket name and so on, and run that code, which creates the S3 bucket automatically.

Let's explore how to install AWS CLI in Windows and how to use the AWS CLI.

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0

Step 1. Download and Install the AWS CLI on a Windows Operating System

First, you need to download the AWS CLI (`https://aws.amazon.com/cli/`), which asks you to save the MSI standalone package in your local system. Once downloaded, run it, and follow the steps by clicking the Next buttons and the Finish button.

Once installation is completed, the program files are stored at `C:\Program Files\Amazon\AWSCLIV2`.

Step 2. Create an Access Key

When you create an AWS account using AWS Management Console, AWS creates a root user who has administrative rights to perform many talks in AWS. You need to create an IAM user in your AWS account to provide the necessary rights.

Log in to AWS Management Console, and in All Services, you can find IAM under the Security, Identity, & Compliance category, as shown in Figure B-1.

Figure B-1. *IAM under Security, Identity, & Compliance*

Clicking IAM takes you to the IAM page, where you find the **My access key** link, as shown in Figure B-2.

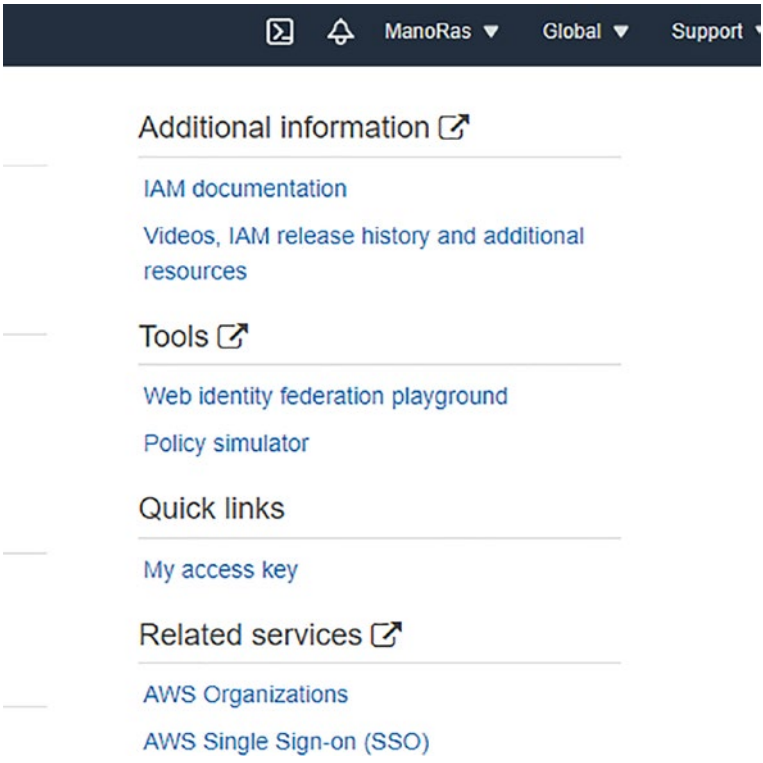


Figure B-2. *My access key*

Clicking **My access key** gives you the Create New Access Key option, as shown in Figure B-3.

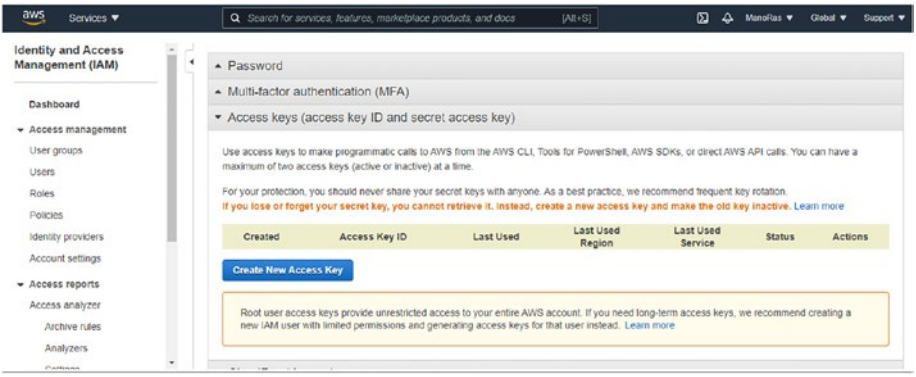


Figure B-3. *Create a new access key*

Clicking the Create New Access Key button opens a Create Access Key popup, with a Download Key File option and a Show Access Key option, as shown in Figure B-4.

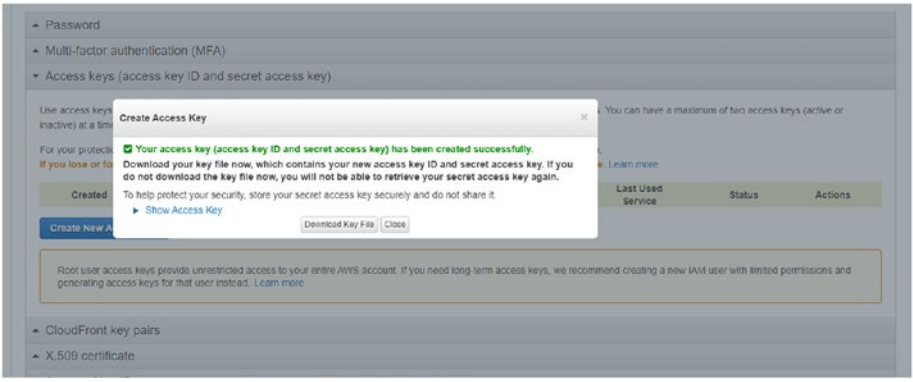


Figure B-4. *Create access key*

Download the file for future reference.

Configure AWS CLI

Once you have successfully installed the AWS CLI, you need to configure the application to connect to your AWS account. To achieve this, open the command prompt, and enter the `aws configure` command, which prompts you for four pieces of information, as shown in Figure B-5.

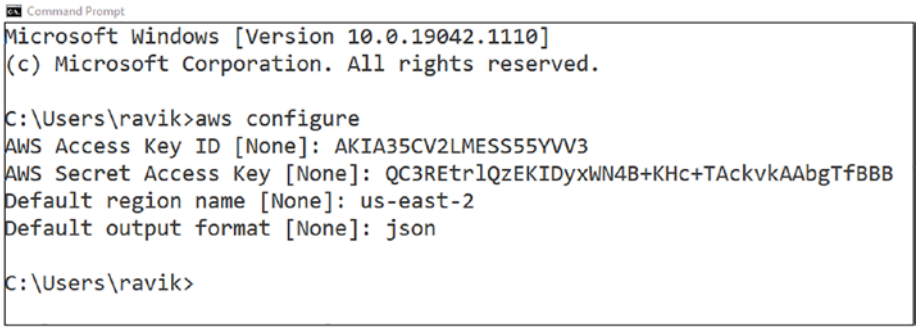


Figure B-5. *AWS configure*

Copy the access key ID and the secret access key from the downloaded key file, which authenticates your AWS account. The region name defines the region where the request from CLI is sent to. The output format specifies the result format: JSON, YAML, text, or table.

Example Commands That Work with S3

1. List all the S3 buckets in your AWS account.
   `aws s3 ls`

```
C:\Users\ravik>aws s3 ls
2021-03-24 19:03:57 elasticbeanstalk-us-east-2-818371255049
2021-07-14 17:57:15 user-registration-frontend-app

C:\Users\ravik>
```

2. Create a bucket.
   `aws s3 mb s3://user-registration-backup`

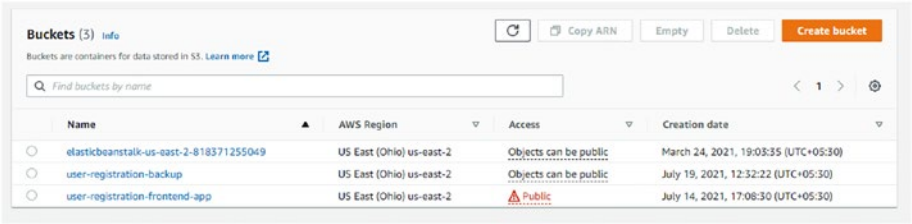
```
C:\Users\ravik>aws s3 ls
2021-03-24 19:03:57 elasticbeanstalk-us-east-2-818371255049
2021-07-14 17:57:15 user-registration-frontend-app

C:\Users\ravik>aws s3 mb s3://user-registration-backup
make_bucket: user-registration-backup

C:\Users\ravik>aws s3 ls
2021-03-24 19:03:57 elasticbeanstalk-us-east-2-818371255049
2021-07-19 12:32:22 user-registration-backup
2021-07-14 17:57:15 user-registration-frontend-app

C:\Users\ravik>
```

3. Verify in AWS Management Console.



4. Refer to the folder at `C:\Program Files\Amazon\AWSCLIV2\awscli\examples` for an example with a command that you can use based on your requirements.

Index

R. K. Soni and N. Soni, *Spring Boot with React and AWS*,
https://doi.org/10.1007/978-1-4842-7392-0

D

E

F, G

H

I

J, K

L

M, N, O, P, Q

R

S, T

U, V

W, X, Y, Z