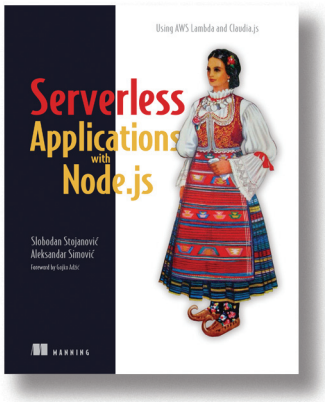
Author Picks

FREE

Exploring Serverless Applications with Node.js

Chapters selected by Slobodan Stojanović and Aleksandar Simović

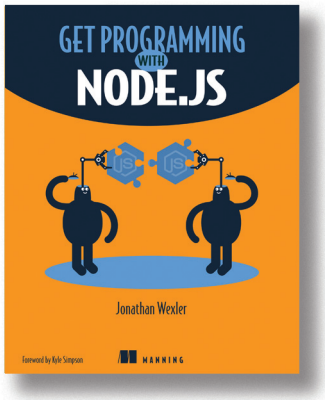
manning



Serverless Applications with Node.js

by Slobodan Stojanović and Aleksandar Simović

ISBN 9781617294723
352 pages
$35.99



Get Programming with Node.js

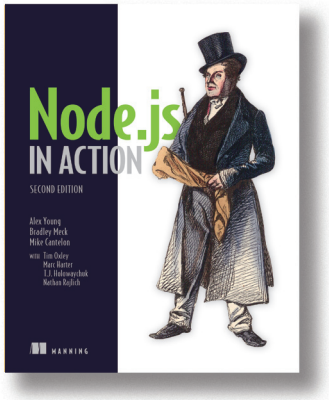
by Jonathan Wexler

ISBN 9781617294747
480 pages
$39.99

Node.js in Motion

by PJ Evans

Course duration: 6h 14m
$59.99



Node.js in Action, Second Edition

by Alex Young, Bradley Meck, and Mike Cantelon

ISBN 9781617292576
392 pages
$39.99

Exploring Serverless Applications with Node.js

Chapters selected by Slobodan Stojanović and Aleksandar Simović

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617297854

contents

introduction

Serverless computing has garnered a lot of interest in recent years, owing its popularity to a host of benefits including quicker packaging and deployment, easier scalability, less software complexity, and decreased operational costs when compared to traditional server-dependent architectures.

This mini ebook, featuring the first three chapters of *Serverless Applications with Node.js*, will get you on your way to learning to build serverless web apps. In the first chapter, you'll get an introduction to core concepts of serverless as well as when and why to use it, and you'll take a look at the features the Node.js library Claudia offers for developing serverless applications.

You'll go hands-on in the second chapter as you build a serverless pizzeria API with Claudia and deploy it to AWS Lambda and API Gateway. Building on the basic concepts you learned in the previous chapter, you'll look more closely at what Claudia is doing under the hood as you structure and deploy your API.

Once you've built your API, which you've programmed with features to create, update, and cancel pizza orders, you'll need a way to store them, and that's where chapter 3 comes in. Since Node.js executes asynchronously, you'll discover how serverless affects asynchronization and how it works with Claudia. You'll also see how easy it is to connect AWS Lambda to an external service. Then you'll use it to store your pizza orders with AWS DynamoDB.

This quick-start guide has plenty of examples, exercises, and tips to help lock in the concepts you'll learn along the way, making it a great place to start learning to build serverless web apps. If you want to delve further into serverless development, chapter 4 of *Serverless Applications with Node.js* continues on with topics including debugging, leveling up your API, building a serverless chatbot, and more. Congratulations on taking this step toward learning these valuable serverless development skills!

Chapter 1 from *Serverless Applications with Node.js*
by Slobodan Stojanović and Aleksandar Simović

This chapter explains what serverless is and why it matters to you, giving you a solid foundation of core serverless concepts.

1 Introduction to serverless with Claudia

This chapter covers

- What serverless is
- The core concepts of serverless
- The difference between serverless and hosted web applications
- How Claudia fits
- Why use serverless

Serverless is a method of deploying and running applications on cloud infrastructure, on a pay-per-use basis and without renting or buying servers. Instead of you, the serverless platform provider is responsible for capacity planning, scaling, balancing, and monitoring; the provider is also able to treat your applications as functions.

Wait, no servers? Seems like a new buzzword, a hipster cloud trend promising to revolutionize your life for the better.

This book explains what serverless is, what problems it solves, and where it does or doesn't fit into your application development process, without fanboyishly showing off and selling serverless like some trendy cloud cult that everyone needs to follow. It does so with a pragmatic approach, explaining the concepts by teaching you how to build reliable and scalable serverless applications with Node.js and Claudia.js, while saving time and resources.

This chapter focuses on the concepts of serverless: what it is, why it matters, and how it compares to server-hosted web application development. Your main goal for this chapter is to gain a good understanding of basic serverless concepts and build a good foundation.

1.1 Servers and washing machines

To understand serverless, consider for a moment washing machines. A clothes-cleaning appliance might sound like a crazy place to start, but owning a server nowadays is similar to owning a washing machine. Everybody needs clean clothes, and the most logical solution seems to be buying a washing machine. But most of the time the washing machine is plugged in, doing nothing. At best, it's used 5 to 15 hours per week. The same goes with servers. Most of the time, your average application server is just waiting to receive a request, doing nothing.

Interestingly, servers and washing machines have many common issues. They both have a maximum weight or volume they can process. Owning a small server is similar to owning a small washing machine; if you accumulate a big pile of laundry, the machine can't process all of it at once. You can buy a bigger one that can take up to 20 pounds of clothes, but then you'll find yourself in a situation where you want to clean just one shirt, and running a 20-pound machine for a single shirt seems wasteful. Also, setting up all your applications to run safely together on one server is tricky, and sometimes impossible. A correct setup for one app can completely mess up another one with a different setting. Similarly, with washing machines you have to separate clothes by color, and then choose the proper program, detergent, and softener combinations. If you don't handle setup properly, the machine can ruin your clothing.

These issues, along with the problem that not everyone is able to own a washing machine, led to the rise of laundromats or launderettes—coin laundry machines that you rent for the time needed to wash your clothes. For servers, the same need has led many companies to start providing server rental services, either locally or in the cloud. You can rent a server, and the server provider takes care of the storage, power, and basic setup. But both laundromats and rental servers are just partial solutions.

For rentals of washing machines and servers, you still need to know how to combine your clothes or applications and set up the machines, choosing appropriate detergents or environments. You also still have to balance the number of machines and their size limitations, planning how many you will need.

In the world of dirty laundry, in the second half of the twentieth century, a new trend of "fluff and fold" services started. You can bring these services a single piece or a bag of clothes, and they will clean, dry, and fold your laundry for you. Some even deliver to your address. They usually charge by the piece, so you don't need to wait to gather a specifically sized batch to clean, and you don't have to worry about washing machines, detergents, and cleaning programs at all.

Compared to the clothes cleaning industry, the software industry is still in the era of self-service laundromats, as many of us still rent servers or use Platform as a Service (PaaS) providers. We are still estimating the number of potential requests (quantity of

clothes) that we're going to handle and reserving enough servers to (we hope) deal with the load, often wasting our money on servers that either are not operating at full capacity or are overloaded and unable to process all our customer requests.

1.2 The core concepts

So how does serverless change that? The name, implying having no server at all, doesn't seem to be a logical solution. Go back to the definition:

> **What is serverless**
>
> *Serverless* is a method of deploying and running applications on cloud infrastructure, on a pay-per-use basis and without renting or buying servers.

Contrary to its name, serverless does not exclude the existence of servers; software requires hardware to run. Serverless just removes the need for companies, organizations, or developers to physically rent or buy a server.

You are probably wondering why serverless is so named. The answer lies in the serverless abstraction of the server concept. Rather than renting a server for your application, setting up the environment, and deploying it, you upload your application to your serverless provider, which takes care of assigning servers, storage, application handling, setup, and execution.

> **NOTE** Some of you may be wondering whether serverless removes a company's need for large DevOps teams. For most situations, the answer is yes.

More precisely, the provider stores your application inside a certain container. The container represents an isolated environment that contains everything your application needs to run. You can think of the container as being a pot for houseplants. The plant pot contains earth filled with all the minerals your plant needs to live.

Like the plant pot, the container allows the serverless provider to safely move and store your application, and to execute it and make copies of it depending on your needs. But the main benefit of serverless is that you don't do any server configuration, balancing, scaling—basically, any kind of server management. The serverless provider manages all of that for you while also guaranteeing that if a large number of calls to your application occur at the same time, it will clone enough containers to handle all the calls, and each clone will be an exact copy of the first. If necessary, the provider will create thousands of clones. The serverless provider decides to replicate a container only when the number of requests to your application becomes so big that the current container can't handle all incoming requests.

Unless there is a request (a call) to your application, not a single instance of your application is running, so it isn't wasting space, server time, or energy. The serverless provider is responsible for all the operational details, such as knowing where your application is stored, how and where to replicate it, when to load new containers, and when to reduce the number of replicated containers to unload the unused servers.

From the washing machine perspective, the process is like calling a fluff and fold cleaning service; the delivery guy appears at your door to pick up your dirty laundry, and the service cleans and then returns the laundry to you. No matter how much clothing you have and no matter what kinds (wool, cotton, leather, and so on), the cleaning company is responsible for all the processes of separation, detergent choice, and program selection.

Serverless and FaaS

Initially, the term *serverless* was interpreted differently from what it means now. In the early days of serverless, it was defined as a *Backend as a Service* (BaaS), because it represents applications that are partly or completely dependent on third-party services for server-based logic. Later, it was almost exclusively described as a *Function as a Service* (FaaS), because the serverless providers treat applications as functions, invoking them only when requested.

1.3 How does serverless work?

As previously described, serverless providers supply an isolated compute container for your application. The compute container is event-driven, so it's activated only when a certain event triggers.

Events are specific external actions that behave exactly like physical triggers. Take your home lights as an example: the events that turn them on can differ. A classic light switch is invoked by pressure; a motion sensor is tied to motion detection; a daylight sensor turns your lights on when the sun goes down. But containers are not limited to listening to the specified events and invoking your contained functions; they also provide a way for your functions to create events themselves, or, more precisely, to emit them. In a more technical manner, with serverless, your function containers are both *event listeners* and *event emitters*.

Finally, serverless providers offer various triggers that can run your functions. The list of triggers depends on the provider and implementation, but some of the most common triggers are HTTP requests, file uploads to file storage, database updates, and Internet of Things (IoT) events. There are many more.

NOTE A serverless function runs only when triggered, and you pay only for its execution time. After execution, the serverless provider shuts the function down, while keeping its trigger active.

1.4 Serverless in practice

The whole serverless landscape contains lots of moving parts, so we introduce it gently. We build an example application and bring in one piece at a time, so you can see how it fits. As you slowly pick up each new concept, you'll expand your example application.

This book takes a greenfield approach to its example application (it will be built from scratch), and it handles the problems of a small company—more precisely, a pizzeria. The pizzeria is managed by your fictional Aunt Maria. During the course of the book, Aunt Maria will face a lot of real-world problems, and your goal will be to help her while grasping serverless concepts along the way. Serverless, like every new technology, introduces a lot of new concepts that can be difficult to handle all at once.

NOTE For a brownfield situation (migrating your current application to serverless), feel free to jump to the last part of the book. If you're not familiar with serverless, you should go through at least the first few chapters before jumping to the last part of the book.

1.4.1 Aunt Maria's serverless pizzeria

Aunt Maria is a strong-willed person. For more than three decades, she has been managing her pizzeria, which was the place where many generations of people from the neighborhood spent time with their families, laughed together, and even went on romantic dates. But recently, her pizzeria has seen some rough times. She's told you that she's seeing fewer and fewer customers. Many of her customers now prefer ordering online via websites or their phones rather than visiting in person. Some new companies have started stealing her customers. The new Chess's pizzeria, for example, has a mobile app with pizza previews and online ordering, and also a chatbot for ordering via various messenger applications. Your aunt's customers like her pizzeria, but most want to order from their homes, so her three-decades-old business has started to wane. The pizzeria already has a website, but it needs a back-end application to process and store information on pizzas and orders.

1.4.2 A common approach

Given Aunt Maria's limited resources, the easiest solution is to build a small API with a popular Node.js framework, such as Express.js or Hapi, and set up a pizza database in the same instance (most likely MongoDB, MySQL, or PostgreSQL).

A typical API would have its code structured in a few layers resembling a three-tier architecture, meaning that the code is split into presentational, business, and data tiers or layers.

Three-tier architecture

Three-tier architecture is a client/server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), and computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

To learn more about three-tier architecture, visit https://en.wikipedia.org/wiki/Multitier_architecture#Three-tier_architecture.

The typical three-tier application design would be similar to figure 1.1, with separate routes for pizzas, orders, and users. It would also have routes for webhooks for both chatbots and a payment processor. All the routes would trigger some handler functions in the business layer, and the processed data would be sent to the data layer—database and file and image storage.

This approach fits perfectly for any given small application. It would work fine for your Pizza API, at least until online pizza orders grow to a certain level. Then you would need to scale your infrastructure.

But to be able to scale a monolithic application, it's necessary to detach the data layer (because you don't want to replicate your database, for the sake of data consistency). After that, your application would look like the one shown in figure 1.2. But you'd still have one conglomerate of an application with all its API routes and the business logic for everything. Your application could be replicated if you had too many users, but each instance would have all its services replicated as well, regardless of their usage.

Monolithic application

A *monolithic application* is a software application in which the user interface and data access code are combined into a single program on a single platform. A monolithic application is self-contained and independent from other computing applications.

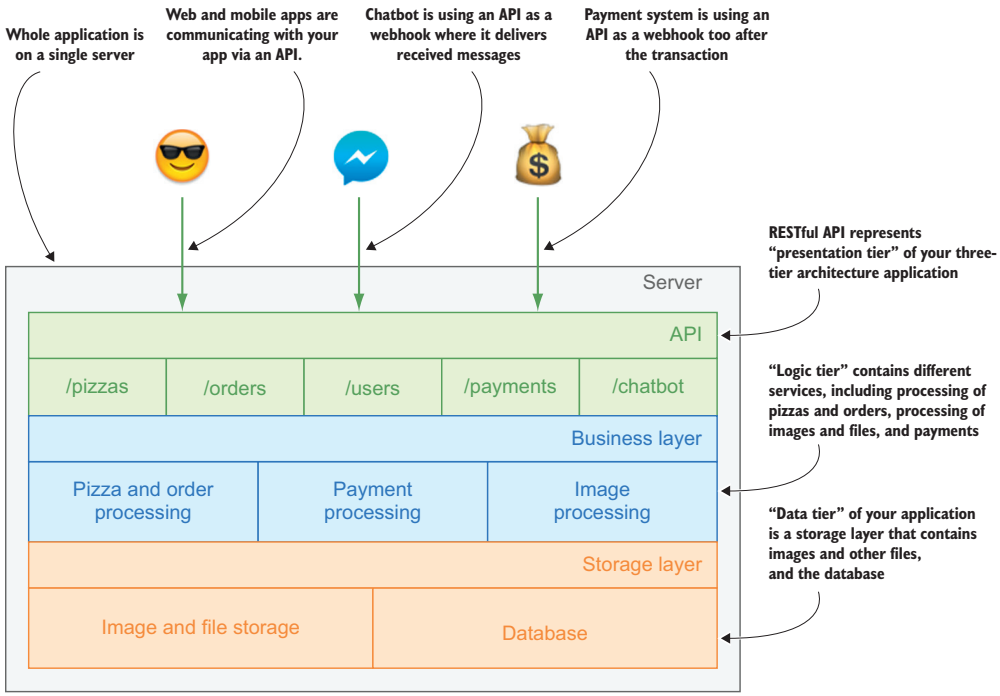


Figure 1.1 The typical three-tier design for the Pizza API

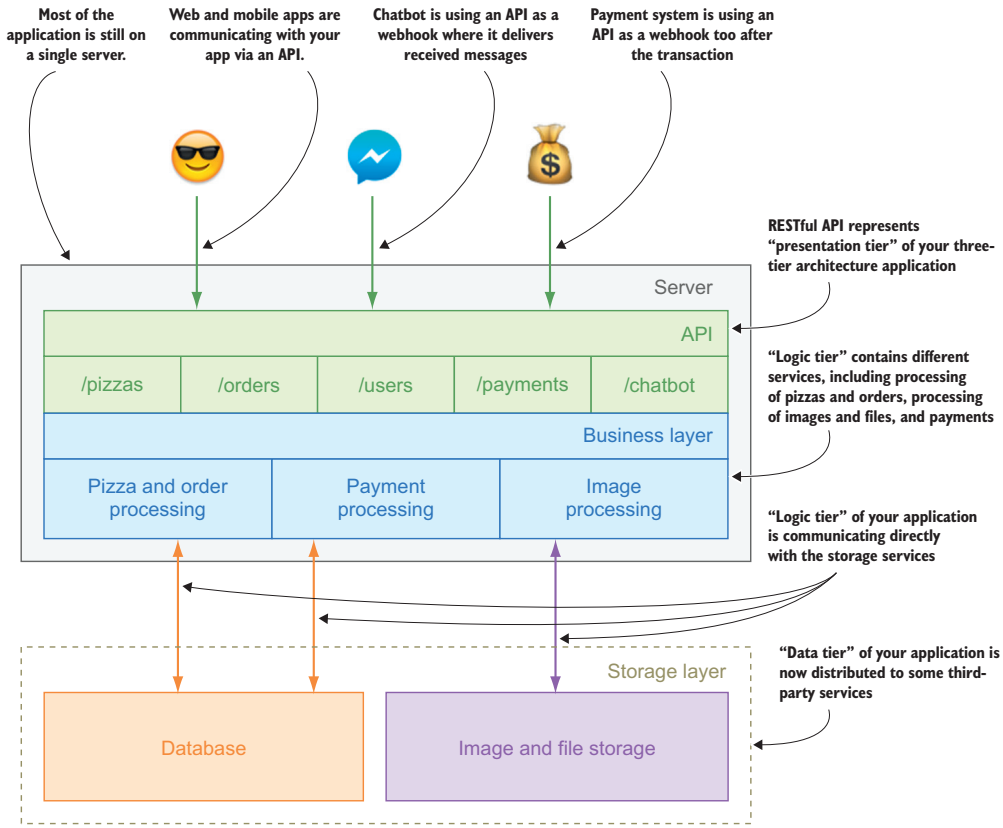


Figure 1.2 A common approach with an external database and file storage for the Pizza API

1.4.3 Serverless approach

Creating serverless applications requires a different approach, as these applications are event-driven and fully distributed.

Instead of having one server with the API endpoints and your business logic, each part of your application is isolated to independent and autoscalable containers.

In a serverless application, your requests are processed with an API router layer that has only one job: it accepts HTTP requests and routes them to the underlying business layer services. The API router in a serverless architecture is always independently managed. That means that application developers don't maintain the API router, and it's scaled automatically by the serverless provider to accept all the HTTP requests your API is receiving. Also, you pay only for the requests that are processed.

In the case of your Pizza API, the router will receive all the API requests from the mobile and web applications, and if necessary handle webhooks from chatbots and the payment processor.

After an API request is routed, it is passed to another container with the business layer service to be processed.

Instead of having one monolithic application, the business logic of a serverless application is often split into smaller units. The size of each unit depends on your preferences. A unit can be as small as a single function or as large as a monolithic application. Most of the time, its size does not directly affect the infrastructure cost, because you are paying for function execution. Units are also scaled automatically, and you won't pay for units that aren't processing anything, so owning one or a dozen of them costs the same.

However, for small applications and situations in which you don't have a lot of information, you can save money on hosting and maintenance by bundling functionalities related to one service into a single business unit. For your Pizza API, a sensible solution is to have one unit for processing pizzas and orders, one for handling payments, one for handling chatbot functionality, and one for processing images and files.

The last part of your serverless API is the data layer, which can be similar to the data layer in a scaled monolithic application, with a separately scaled database and file storage service. It would be best if the database and file storage were also independent and autoscalable.

Another benefit of a serverless application is that the data layer can trigger a serverless function out of the box. For example, when a pizza image is uploaded to the file storage, an image processing service can be triggered, and it can resize the photo and associate it with the specific pizza.

You can see the flow of the serverless Pizza API in figure 1.3.

1.5 Serverless infrastructure — AWS

Your serverless Pizza API needs infrastructure to run on. Serverless is very young and at the moment has several infrastructure choices. Most of these choices are owned by big vendors, because serverless requires a big infrastructure for scaling. The best-known and most advanced infrastructures are Amazon's AWS Lambda serverless compute container, Microsoft's Azure Functions, and Google's Cloud Functions.

This book focuses on AWS Lambda because AWS has the most mature serverless infrastructure available in the market, with a stable API and many successful stories behind it.

AWS Lambda is an event-driven serverless computing platform provided by Amazon as part of Amazon Web Services. It is a compute service that runs code in response to events and automatically manages the compute resources required by that code.

Google Cloud Functions and Microsoft Azure Functions

Google launched Google Cloud Functions, its answer to Amazon's AWS Lambda, in mid-2016. Google Cloud Functions are explained as lightweight event-based microservices that allow you to run JavaScript functions in a Node.js runtime. Your function can be triggered by an HTTP request, Google Cloud Storage, and other Google Cloud Pub/Sub services. At the time this book was written, Google Cloud Functions were still in alpha, so pricing was not known. You can learn more at the official website: https://cloud.google.com/functions/.

(continued)

Microsoft's implementation of serverless—Azure Functions—is part of its Azure cloud computing platform. Microsoft describes it as an event-based serverless compute experience that accelerates your development, scales based on demand, and charges you for only the resources you consume. Azure Functions allows you to develop functions in JavaScript, C#, F#, Python, and other scripting languages. Azure pricing is similar to that of AWS Lambda: You're charged 20 cents per 1 million executions and $0.000016 per GB of resource consumption per month, with a free tier for the first 1 million requests and 400,000 GB each month. For more information, visit the official website at https://azure.microsoft.com/en-us/services/functions/.

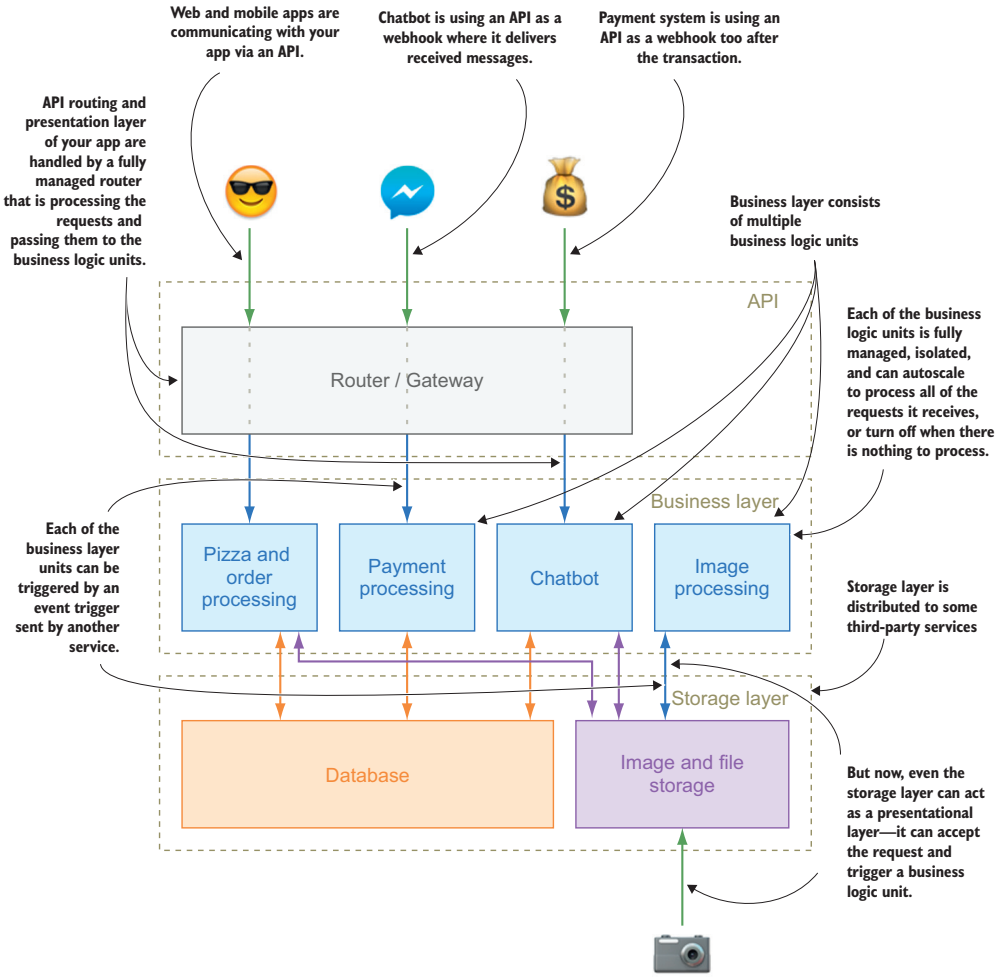


Figure 1.3 The serverless approach for the Pizza API

NOTE Most of the things you'll learn in this book are also feasible with other serverless providers, but some services might differ, so some of the solutions might need a slightly different approach.

In the Amazon platform, the word *serverless* is usually directly related to AWS Lambda. But when you are building a serverless application such as your Pizza API, AWS Lambda is just one of the building blocks. For a full application, you often need other services, such as storage, database, and routing services. In table 1.1 you can see that AWS has fully developed services for all of them:

- Lambda is used for computing.
- API Gateway is a router that accepts HTTP requests and invokes other services depending on the routes.
- DynamoDB is an autoscalable database.
- Simple Storage Service (S3) is a storage service that abstracts the standard hard drives and offers you unlimited storage.

Table 1.1 The building blocks of serverless applications in AWS

Functionality	AWS service	Short description
Computing	Lambda	Computing component, used for your business logic
Router	API Gateway	Routing component, used to route HTTP request data to your Lambda function
Database	DynamoDB	Autoscalable document database
Storage	S3	Autoscalable file storage service

Lambda is the most important serverless puzzle piece you need to understand, because it contains your business logic. Lambda is AWS's serverless computing container that runs your function when an event trigger occurs. It gets scaled automatically if many events trigger the function at the same time. To develop your Pizza API as a serverless application, you will need to use AWS Lambda as its serverless compute container.

When a certain event occurs, such as an HTTP request, a Lambda function is triggered, with the data from the event, context, and a way to reply to the event as its arguments. The Lambda function is a simple function handler written in one of the supported languages. At this time of this writing, AWS Lambda support the following languages:

- Node.js
- Python
- Java (Java 8 compatible) and other JVM languages
- C# (.NET Core)

In Node.js, event data, context, and a function callback are passed as JSON objects. The `context` object contains details about your Lambda function and its current

execution, such as execution time, what triggered the function, and other information. The third argument that your function receives is a callback function that allows you to reply with some payload that will be sent back to the trigger, or an error. The following listing shows a Node.js sample of a small AWS Lambda function that returns the text *Hello from AWS Lambda*.

Listing 1.1 An example of the smallest working Lambda function with Node.js

A function accepts an event, a context, and a callback function.

The callback function returns a success message.

```
function lambdaFunction(event, context, callback) {
  callback(null, 'Hello from AWS Lambda')
}

exports.handler = lambdaFunction
```

The function is exported as a handler.

NOTE As shown in listing 1.1, a function is exported with an `exports.handler` instead of the standard Node.js export, `module.exports`. This is because AWS Lambda requires the module export to be an object with a named `handler` method, rather than the function directly.

As mentioned before, the `event` in your Lambda function is the data passed by the service that triggered your Lambda function. In AWS, functions can be invoked by many things, from common events such as an HTTP request via API Gateway or file manipulation by S3, to more exotic ones such as code deployment, changes in the infrastructure, and even console commands using the AWS SDK.

Here's a list of the most important events and services that can trigger an AWS Lambda function and how they would translate to your Pizza API:

- *HTTP requests via API Gateway*—A website pizza request is sent.
- *Image uploading, deleting, and file manipulation via S3*—A new pizza image is uploaded.
- *Changes in the database via DynamoDB*—A new pizza order is received.
- *Various notifications via the Simple Notification Service (AWS SNS)*—A pizza is delivered.
- *Voice commands via Amazon Alexa*—A customer orders a pizza from home using voice commands.

For the full list of triggers, see http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html.

Lambda functions come with some limitations, such as limited execution time and memory. For example, by default, your Lambda function's execution time is up to three seconds, which means that it will time out if your code tries to process something longer. It also has 128 MB of RAM, which means that it is not suitable for complex computations.

NOTE Both of those limitations can be configured in the function settings. Timeout can be increased up to 15 minutes, and memory can be increased up to 3 GB. Increasing both of the limits can affect the cost per execution of your function.

Another important characteristic of Lambda functions is that they are stateless, and therefore state is lost between subsequent invocations.

Serverless pricing

One of the major selling points of serverless is the price. Amazon prices its standard virtual servers, Elastic Compute Cloud (Amazon EC2) servers, per hour. AWS Lambda is more expensive than EC2 in hourly cost, but by contrast, you don't pay for it unless your function is working. You pay 20 cents per million executions of your AWS Lambda function and $0.000016 per GB of resource consumption per month. Amazon also gives you a free tier with 1 million requests and 400,000 GB at no cost each month.

For your Pizza API, Aunt Maria won't have to pay anything until she reaches 1 million executions per month. If she reaches that number, you have succeeded in your goal of helping her.

For more information about pricing, visit the official website at https://aws.amazon.com/lambda/pricing/.

As you can see in figure 1.4, the flow of a Lambda function goes like this:

- A certain event happens, and the service that handles the event triggers the Lambda function.
- The function, such as the one shown in listing 1.1, starts its execution.
- The function execution ends, either with a success or error message, or by timing out.

Another important thing that can affect your serverless Pizza API is function latency. Because the Lambda function containers are managed by the provider, not the application operators, there's no way to know if a trigger will be served by an existing container or if the platform will instantiate a new one. If a container needs to be created and initialized before the function executes, it requires a bit more time and is called a *cold start*, as shown in figure 1.5. The time it takes to start a new container depends on the size of

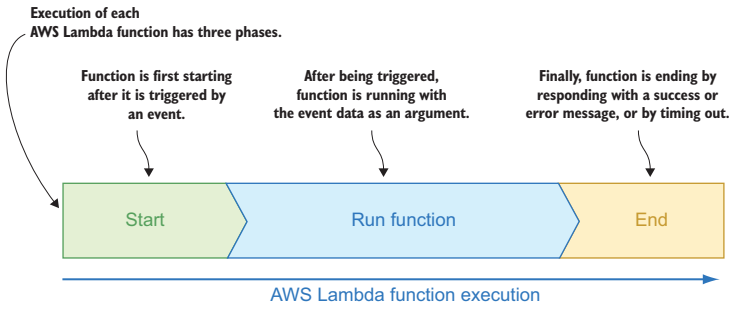


Figure 1.4 The flow of an AWS Lambda function

the application and the platform used to run it. Empirically and at the time of writing of this book, there are noticeably lower latencies with Node.js and Python than with Java.

> **COST OF THE PIZZA API** Developing the Pizza API as described in the following chapters of this book should cost less than a cup of coffee. AWS Lambda by itself will be free, but some of the services used for Pizza API development, such as DynamoDB and the Simple Storage Service, charge a small fee for the storage of your data. Both of those services and their pricing are described in later chapters. The final price of the application will depend on the amount of data and its usage, but if you are following the book's examples, it should be less than $1 per month.

Lambda functions are quite easy to understand and use. The most complex part is the deployment process.

There are a few ways to deploy your serverless application to AWS Lambda. You can deploy through the visual UI on the AWS Lambda console or the terminal via the AWS command-line interface using the AWS API, either directly or via the AWS SDK for one of the supported languages. Deploying a serverless application is simpler than deploying a traditional one, but it can be made even easier.

1.6 What is Claudia, and how does it fit?

Claudia is a Node.js library that eases the deployment of Node.js projects to AWS Lambda and API Gateway. It automates all the error-prone deployment and configuration tasks, and sets everything up the way JavaScript developers expect out of the box.

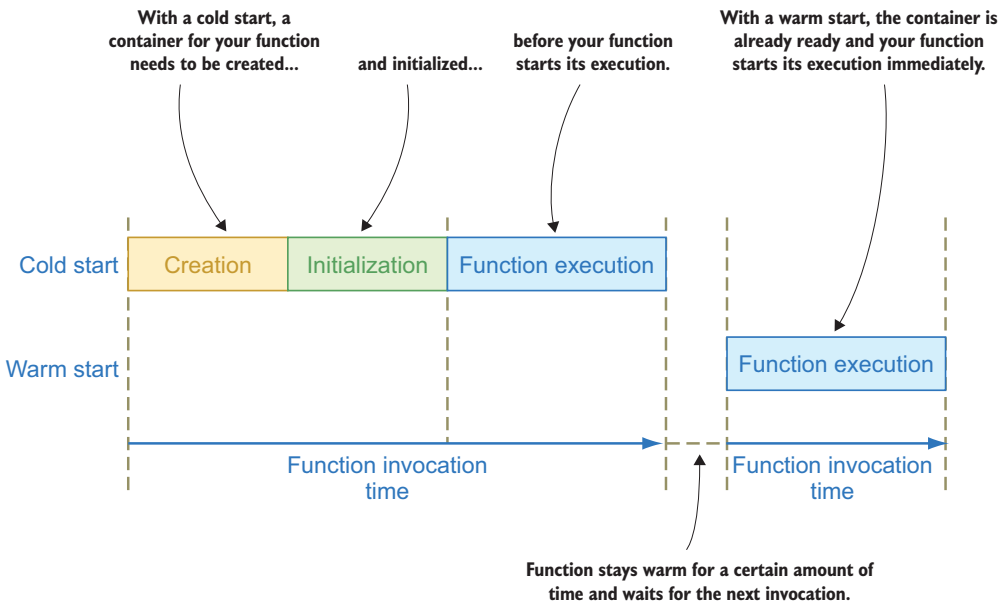


Figure 1.5 Cold start versus hot start for an AWS Lambda function

Claudia is built on top of the AWS SDK to make development easier. It is not a replacement for the AWS SDK or AWS CLI, but an extension that makes some common tasks, such as deployment and setting triggers, easy and fast.

Some of the core values of Claudia are

- Creating and updating the function with a single command (removing the need to manually zip your application and then upload the zip file via the AWS Dashboard UI)
- Not using boilerplate, which allows you to focus on your work and keep your preferred project setup
- Managing multiple versions easily
- Getting started in minutes with a very flat learning curve

Claudia acts as a command-line tool, and it allows you to create and update your functions from the terminal. But the Claudia ecosystem comes with two other useful Node.js libraries: Claudia API Builder allows you to create the API on API Gateway, and Claudia Bot Builder allows you to create chatbots for many messaging platforms.

As opposed to Claudia, which is a client-side tool never deployed to AWS, API Builder and Bot Builder are always deployed to AWS Lambda (see figure 1.6).

You can work with AWS Lambda and API Gateway without Claudia, either by using the AWS ecosystem directly or by using some of the alternatives.

The best-known alternatives are the following:

- Serverless Application Model (SAM), created by AWS, which allows you to create and deploy serverless applications via AWS CloudFormation. For more information, visit https://github.com/awslabs/serverless-application-model.
- Serverless Framework, which has a similar approach to SAM but also supports other platforms, such as Microsoft Azure. To learn more about it, visit https://serverless.com.

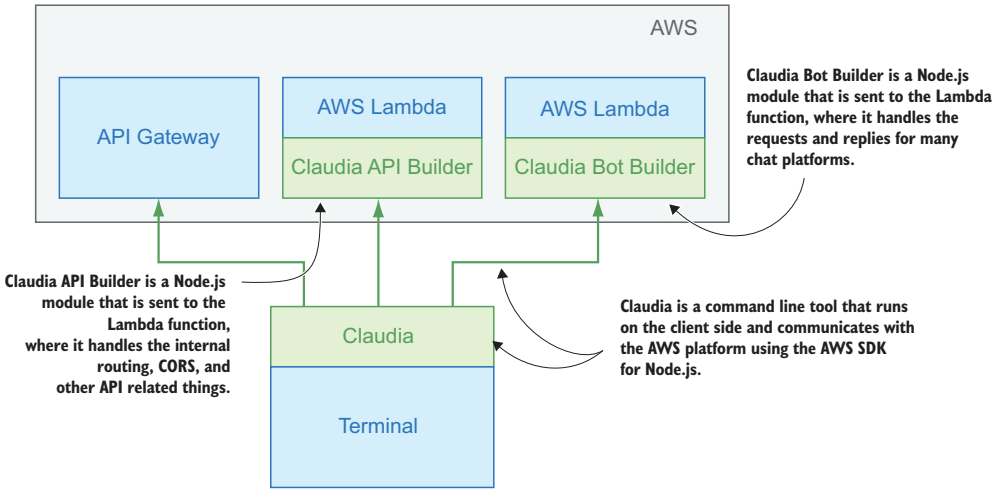


Figure 1.6 A visual representation of the relationships of Claudia, API Builder, and Bot Builder with the AWS platform

- Apex, another command-line tool that helps you deploy serverless applications but has support for more programming languages, such as Go. To learn more about it, visit http://apex.run.

NOTE Everything written in this book can most likely be done with one of these Claudia alternatives.

You are probably wondering why we chose to use Claudia. The Claudia FAQs provide the best explanation:

- *Claudia is a deployment utility, not a framework.* It does not abstract away AWS services, but instead makes them easier to get started with. As opposed to Serverless and Seneca, Claudia is not trying to change the way you structure or run projects. The optional API Builder, which simplifies web routing, is the only additional runtime dependency, and it's structured to be minimal and standalone. Microservice frameworks have many nice plugins and extensions that can help kick-start standard tasks, but Claudia intentionally focuses only on deployment. One of our key design goals is not to introduce too much magic, and let people structure the code the way they want to.
- *Claudia is focused on Node.js.* As opposed to Apex and similar deployers, Claudia has a much narrower scope. It works only for Node.js, but does so really well. Generic frameworks support more runtimes, but leave the developers to deal with language-specific issues. Because Claudia focuses on Node.js, it automatically installs templates to convert parameters and results into objects that JavaScript can consume easily, and makes things work the way JavaScript developers expect out of the box.

For more details, see https://github.com/claudiajs/claudia/blob/master/FAQ.md.

The idea of this book is to teach you how to think in a serverless way and how to easily develop and deploy quality serverless applications. Using the AWS ecosystem directly would involve a lot of distractions, such as learning how to interact with and configure different parts of the AWS platform. Rather than try to replace the AWS SDK, Claudia is built on top of it, and Claudia automates most common workflows with single commands.

Claudia favors code over configuration and as a result has almost no configuration at all. That makes it easier to learn, understand, and test. Writing a high-quality application requires proper testing; having lots of configuration doesn't mean you don't need to test it.

Claudia has a minimal set of commands that allow you to build serverless applications with a pleasant developer experience. Two of the main ideas behind Claudia are to minimize the magic and to be transparent in showing what happened when a command was invoked.

Despite its small API, Claudia enables you to develop many things: you can build serverless applications from scratch, migrate your current Express.js applications to serverless, and even build your own serverless chatbots and voice assistants.

1.7 When and where you should use it

Serverless architecture is not a silver bullet. It doesn't solve all problems, and it might not solve yours.

For example, if you are building an application that relies heavily on web sockets, serverless is not for you. AWS Lambda can work for up to 15 minutes, and it can't stay awake to listen for web socket messages after that.

If latency is critical for your application, even though waking containers is fast, there is always a price to pay for waking up them up. That price is a few dozen milliseconds, but for some applications, that can be too much.

The absence of configuration is one of the main selling points for serverless, but that advantage can be a huge setback for some application types. If you are building an application requiring a system-level configuration, you should consider the traditional approach instead. You can customize AWS Lambda to some extent; you can provide a static binary and use Node.js to invoke it, but that can be overkill in many cases.

Another important disadvantage is so-called vendor lock-in. Functions themselves are not a big problem because they are just standard Node.js functions, but if your full application is built as a serverless application, some services are not easy to migrate. However, this problem is a common one that is not related only to serverless, and it can be minimized with good application architecture.

That said, serverless has many more upsides than downsides, and the rest of this book shows you some of the good use cases.

Summary

- Serverless is abstracting servers away from software development.
- A serverless application differs from a traditional one in that serverless applications are event-driven, distributed, and autoscalable.
- There are a few choices for serverless infrastructure, and the most advanced one is Amazon's AWS Lambda.
- AWS Lambda is an event-driven, serverless computing platform that allows you to run functions written in Node.js, Python, C#, or Java and other JVM languages.
- AWS Lambda has certain limitations, such as execution time, which can be up to 15 minutes, and available memory, which can be up to 3 GB.
- The most complex parts of a serverless application in AWS are deployment and function configuration.
- Some tools and frameworks can help you deploy and configure your application more easily. The easiest one to use is Claudia, with its API Builder and Bot Builder.
- Claudia is a command-line tool that offers a minimal set of commands to allow you to build serverless applications with a pleasant developer experience.
- Serverless architecture is not a silver bullet, and there are some situations in which it isn't the best choice, such as for real-time applications with web sockets.

Chapter 2 from *Serverless Applications with Node.js*
by Slobodan Stojanović and Aleksandar Simović

Now that you've got a good grasp on the basic concepts of serverless, you'll take a closer look at Claudia in action as you use it to build and deploy an API, and you'll learn firsthand how API Gateway works along the way.

2 Building your first serverless API

This chapter covers

- Creating and deploying an API using Claudia
- How Claudia deploys an API to AWS
- How API Gateway works

The main goal of this chapter for you is to build your first serverless API with Claudia and deploy it to AWS Lambda and API Gateway. You'll also see the differences between a traditional and a serverless application structure and gain a better grasp of Claudia as you learn what Claudia is doing under the hood. To get the most from this chapter, you should understand the basic concepts of serverless described in chapter 1.

2.1 Assembling pizza ingredients: building the API

Your Aunt Maria is happy and grateful that you are going to help her get back on her feet. She even made you her famous pepperoni pizza! (Try not to be hungry at this moment!)

Aunt Maria already has a website, so you will build a back-end application—more precisely, an API—to enable her customers to preview and order pizzas. The API will be responsible for serving pizza and order information, as well as handling pizza orders. Later, Aunt Maria would also like to add a mobile application, which would consume your API services.

To start gently, the first API endpoints will handle some simple business logic and return static JSON objects. You can see the broad overview of your initial application structure in figure 2.1. The figure also shows the crude HTTP requests flow through your API.

Here is the list of features we cover for the initial API:

- Listing all pizzas
- Retrieving the pizza orders
- Creating a pizza order
- Updating a pizza order
- Canceling a pizza order

These features are all small and simple; therefore, you will implement them in a single Lambda function.

Even though you might feel that you should separate each feature into a separate function, for now it's simplest to put everything in the same Lambda, because the functions are tightly coupled. If you were to do inventory tracking as well, you would create that as a separate function from the start.

Each of the listed features will need to have a separate route to the corresponding handler within your function. You can implement the routing yourself, but Claudia has a tool to help you with that task: Claudia API Builder.

Claudia API Builder is an API tool that helps you handle all your incoming API Gateway requests and responses, as well as their configuration, context, and parameters, and enables you to have internal routing within your Lambda function. It has an Express-like endpoint syntax, so if you are familiar with Express, Claudia API Builder will be easy to use.

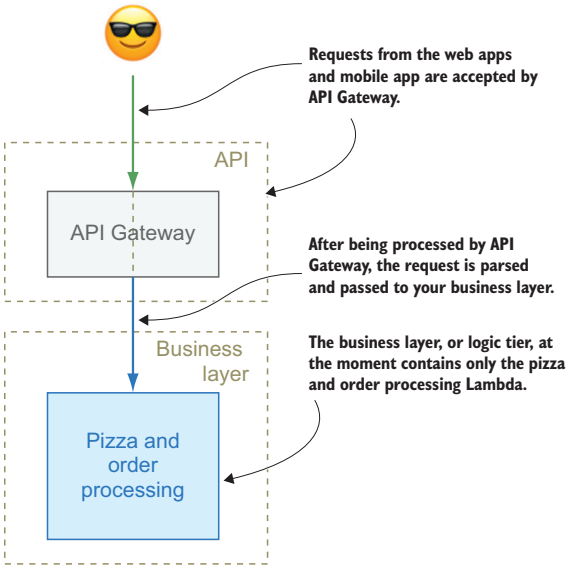


Figure 2.1 **A broad overview of the Pizza API you will build in this chapter**

Figure 2.2 shows a more detailed overview of how to route and handle the pizza and order features within your Lambda function by using Claudia API Builder. The figure shows that upon receiving requests from API Gateway, Claudia API Builder will redirect the requests to your defined routes and their corresponding handlers.

NOTE At the time of this writing, you can use AWS API Gateway in two modes:

- With models and mapped templates for requests and responses
- With proxy pass-through

Claudia API Builder uses proxy pass-through to capture all the HTTP request details and structure them in a JS developer-friendly way.

To learn more about proxy pass-through and models and mapped templates, you can read the official documentation at http://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-method-settings.html.

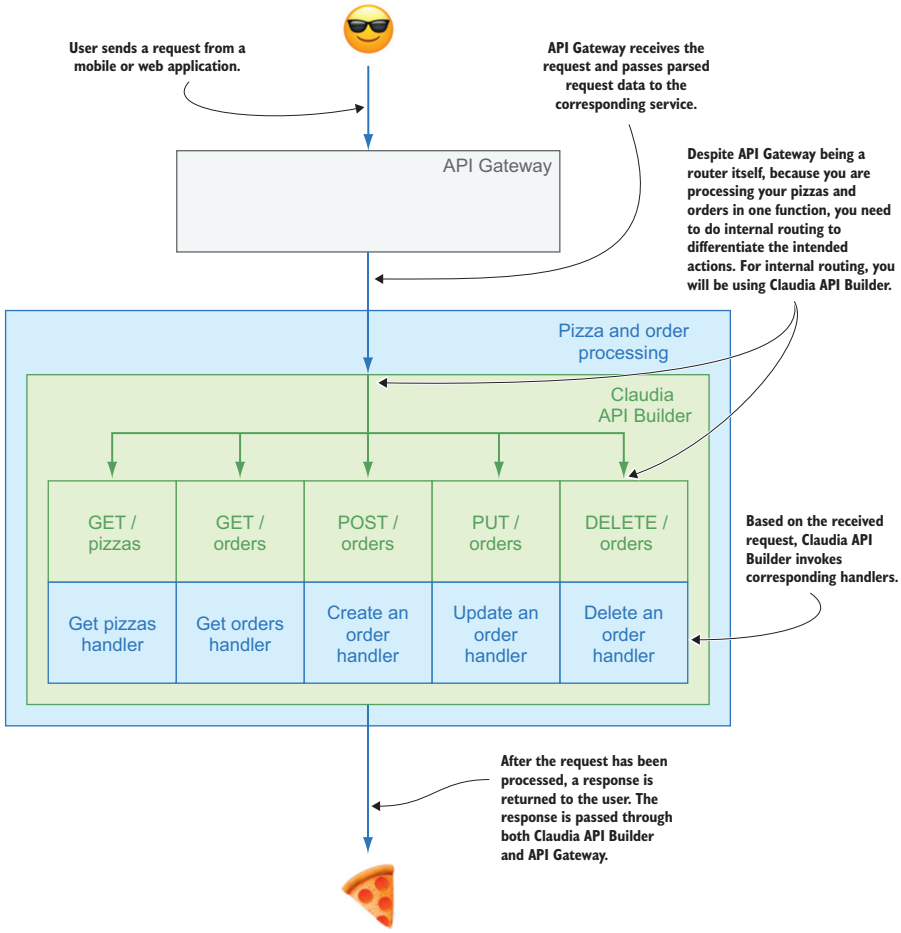


Figure 2.2 A visual representation of the AWS Lambda function that handles pizza and order processing

2.1.1 Which pizza can I GET?

As the first method of your Pizza API you will create a `GET` pizza service that lists all available pizzas. To do so, you will need to fulfill these prerequisites:

- Own an AWS account and properly set up the AWS credentials file
- Install Node.js and its package manager, NPM
- Install Claudia from NPM as a global dependency

If you're not familiar with these steps or are not sure whether you have completed them, jump to appendix A, which guides you through each setup process.

> **CODE EXAMPLES** From this point onward, you'll see a lot of code examples. We highly recommend that you try them all, even if they feel familiar. You can use your favorite code editor unless stated otherwise.

Now that you're fully set up, you can start by creating an empty folder for your first serverless application. You can name your project folder as you like, but in this book the application folder's name is pizza-api. After you've created it, open your terminal, navigate to your new folder, and initialize the Node.js application. After your app is initialized, install the `claudia-api-builder` module from NPM as a package dependency, as explained in appendix A.

The next step is to create your application's entry point. Create a file named api.js inside your pizza-api folder, and open it with your favorite code editor.

> **ES6 SYNTAX FOR THE CODE EXAMPLES** All the code examples in the book use the ES6/ES2015 syntax. If you are not familiar with ES6 features, such as arrow functions and/or template strings, see Manning's *ES6 in Motion*, by Wes Higbee, or the second edition of *Secrets of the JavaScript Ninja*, by John Resig.

To create an API route, you need an instance of Claudia API Builder, as it is a class and not a utility function. At the beginning of your api.js file, require and instantiate `claudia-api-builder`.

Now you're able to use Claudia API Builder's built-in router. To implement the `GET /pizzas` route, you need to use the `get` method of your Claudia API Builder instance. The `get` method receives two arguments: a route and a handler function. As the route parameter, pass the string `/pizzas`, and as the handler, pass an anonymous function.

The Claudia API Builder anonymous handler function has one major difference compared with Express.js. In Express.js, you have both the response and the request as callback function arguments, but Claudia API Builder's callback function has only the request. To send back the response, you just return the result.

Your `GET /pizzas` route should show a list of pizzas, so for now, you will return a static array of pizzas from Aunt Maria's pizzeria: Capricciosa, Quattro Formaggi, Napoletana, and Margherita.

Finally, you need to export your API instance, which Claudia API Builder is fitting into your Lambda function as middleware.

At this point, your code should look like the following listing.

Listing 2.1 The `GET /pizzas` handler of your Pizza API

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

api.get('/pizzas', () => {
  return [
    'Capricciosa',
    'Quattro Formaggi',
    'Napoletana',
    'Margherita'
  ]
})

module.exports = api
```

Require the Claudia API Builder module.

Create an instance of Claudia API Builder.

Define a route and a handler.

Return a simple list of all pizzas.

Export your Claudia API Builder instance.

That's all it takes to make a simple serverless function. Before popping a champagne bottle in celebration, however, you should deploy your code to your Lambda function. To do so, jump back to your terminal and unleash the power of Claudia.

Because one of Claudia's main goals is single-command deployment, deploying your API takes just a simple `claudia create` command. This command requires only two options: the AWS region where you want your API to be deployed, and your application's entry point. The options are passed as flags, so to deploy your API, just execute the `claudia create` command with `--region` and `--api-module` flags, as shown in listing 2.2. The intricacies of the `claudia create` command are explained in more detail in section 2.2.

SHELL COMMANDS FOR WINDOWS USERS Some of the commands in the book are split into multiple lines for readability and annotation purposes. If you are a Windows user, you might need to join those commands into a single line and remove backslashes (\).

Listing 2.2 Deploying an API to AWS Lambda and API Gateway using Claudia

```
claudia create \
  --region eu-central-1 \
  --api-module api
```

Create and deploy a new Lambda function.

Select the region where you want your function to be deployed.

Tell Claudia that you are building an API and that your API's entry point is api.js.

For your region, choose the closest one to your users to minimize latency. The closest region to Aunt Maria's pizzeria is in Frankfurt, Germany, and it's called `eu-central-1`. You can see all the available regions in the official AWS documentation: http://docs.aws.amazon.com/general/latest/gr/rande.html#lambda_region.

Your api.js file is your API's entry point. Claudia automatically appends the .js extension, so just type `api` as your application's entry point.

NOTE The name and location of your entry point are up to you; you just need to provide a correct path to the entry point in the `claudia create` command. For example, if you name it index.js and put it in the src folder, the flag in the Claudia command should be `--api-module src/index`.

After a minute or so, Claudia will successfully deploy your API. You'll see a response similar to listing 2.3. The command response has useful information about your Lambda function and your API, such as the base URL of your API, the Lambda function's name, and the region.

DEPLOYMENT ISSUES If you encounter deployment issues, such as a credentials error, make sure you've properly set up everything as described in appendix A.

Listing 2.3 The `claudia create` command response

```
{
  "lambda": {
    "role": "pizza-api-executor",
    "name": "pizza-api",
    "region": "eu-central-1"
  },
  "api": {
    "id": "g8fhlgccof",
    "module": "api",
    "url": "https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/latest"
  }
}
```

Lambda function information

API information

Your API's base URL

During the deployment, Claudia created a claudia.json file in the root of your project along with some similar information, but without your base API URL. This file is for Claudia to relate your code to a certain Lambda function and API Gateway instance. The file is intended for Claudia only; don't change it by hand.

Now it's time to "taste" your API. You can try it directly from your favorite browser. Just visit the base URL from your `claudia create` response, remembering to append your route to the base URL. It should look similar to https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/latest/pizzas. When you open your modified base URL link in your browser, you should see the following:

```
["Capricciosa","Quattro Formaggi","Napoletana","Margherita"]
```

URLS FOR THE EXAMPLES FROM THE BOOK Instead of `latest`, each example from the book will contain different versions in the following format: `chapterX_Y`, where `X` is the number of the chapter and `Y` is the number of the example in that chapter. We did this so you can run the examples simply by copying the URL from the book. When you run the code by yourself, the output URL will contain `latest` as a version, instead of `chapterX_Y` that you'll see in the book.

For example, the first example can be accessed at the following URL: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_1/pizzas.

Congratulations—you just built a serverless API with Claudia! If this was your first time, you should be proud of yourself, and this is a good time to pause.

2.1.2 Structuring your API

Before rushing to add more features, you should always try to spend a few minutes rethinking your API structure and organization. Adding all the route processors directly into the main file makes it difficult to understand and maintain, so you should ideally split handlers from routing/wiring. Smaller code files are easier to understand and work with than with one monster file.

Considering application organization, at the time of this writing there aren't any specific best practices. Also, Claudia gives you complete freedom on that topic. For your Pizza API, because the part for handling pizzas and orders isn't going to be huge, you can move all route handlers to a separate folder and keep only the routes within your

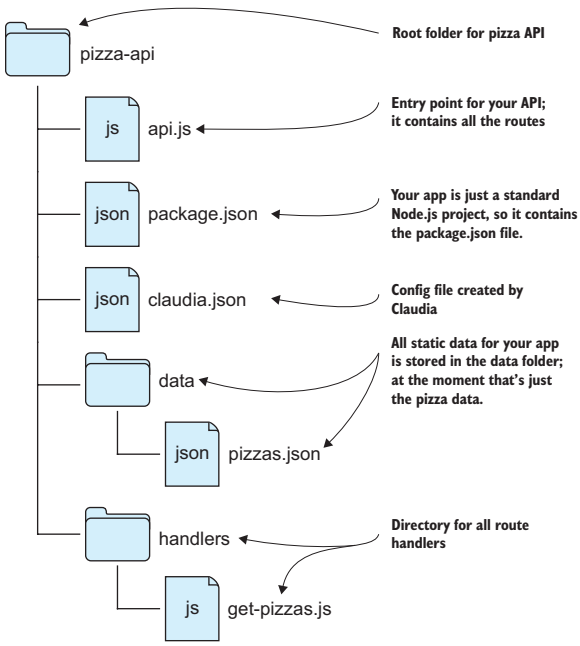


Figure 2.3 The file structure of the Pizza API project

api.js file. After that, because the pizza list should have more pizza attributes than just pizza names, you should move it to a separate file. You can even go a step further and create a folder for the data, as you did for the pizza list we mentioned earlier. After you apply these recommendations, your code structure should look similar to figure 2.3.

The first modification is moving the list of pizzas to a separate file and extending the list with additional information, such as pizza IDs and ingredients. To do so, create a folder in the root of your Pizza API project, and name it data. Then create a file in your new folder, and name it pizzas.json. Add the content from the following listing to the new file.

Listing 2.4 JSON containing the pizza info

Each pizza object has a pizza ID, name, and ingredients.

This JSON file is an array of pizza objects.

```
[
  {
    "id": 1,
    "name": "Capricciosa",
    "ingredients": [
      "tomato sauce", "mozzarella", "mushrooms", "ham", "olives"
    ]
  },
  {
    "id": 2,
    "name": "Quattro Formaggi",
    "ingredients": [
      "tomato sauce", "mozzarella", "parmesan cheese", "blue cheese", "goat
     cheese"
    ]
  },
  {
    "id": 3,
    "name": "Napoletana",
    "ingredients": [
      "tomato sauce", "anchovies", "olives", "capers"
    ]
  },
  {
    "id": 4,
    "name": "Margherita",
    "ingredients": [
      "tomato sauce", "mozzarella"
    ]
  }
]
```

Your next step is to move the `getPizzas` handler to a separate file. Create a folder called handlers in your project root, and create a get-pizzas.js file inside it.

In your new get-pizzas.js file will be the `getPizzas` handler, which returns the list of pizzas from listing 2.4. First, you need to import the pizza list from the JSON file you created. Second, you need to create a `getPizzas` handler function and export it so

that you can require it from your entry file. Then, instead of just returning the pizza list, go a step further and return just one pizza if a pizza ID was passed as a parameter to your `getPizzas` handler. To return just one pizza, you can use the `Array.find` method, which searches for a pizza by the pizza ID from your pizza list. If it finds a pizza, return it as a handler result. If there aren't any pizzas with that ID, have your application throw an error.

The updated code of your new pizza handler should look similar to the next listing.

Listing 2.5 Your `getPizzas` handler with a pizza ID filter in a separate file

```
const pizzas = require('../data/pizzas.json')

function getPizzas(pizzaId) {
  if (!pizzaId)
    return pizzas

  const pizza = pizzas.find((pizza) => {
    return pizza.id == pizzaId
  })

  if (pizza)
    return pizza

  throw new Error('The pizza you requested was not found')
}

module.exports = getPizzas
```

Create the getPizzas handler function.

Import the list of pizzas from the data directory.

If a pizza ID is not passed, return the full pizza list.

Otherwise, search the list by the passed pizza ID.

Note == instead of ===. That's because pizzaId is passed as a string, and you don't want it to be a strict match, as in the database it may be an integer.

Throw an error if the application doesn't find the selected pizza.

Export the getPizzas handler.

You should also remove the previous `getPizzas` handler code from your API entry point file, api.js. Delete everything between importing Claudia API Builder and the end, where you're exporting your Claudia API Builder instance.

After the line where you're importing Claudia API Builder, import the new `get-pizzas` handler from your handlers folder:

```
const getPizzas = require('./handlers/get-pizzas')
```

> **NOTE** You should also create a handler for the `GET` route for the root path, `/`, which should return a static message to the user. Though this is optional, we highly recommend it. Your API is more user-friendly when it returns some friendly message instead of an error when someone is querying just your API's base URL.

Next you should add the route for getting the pizza list, but this time, you'll use the `get-pizzas` handler you created for the route handling. You should import the file at the beginning of your api.js entry file. If you remember, your `get-pizzas` handler

can also filter pizzas by ID, so you should add another route that returns a single pizza. Write that route so that it accepts a `GET` request for the `/pizzas/{id}` url. The `/{id}` part is the dynamic route parameter that tells your handler which pizza ID the user requested. Like Express.js, Claudia API Builder supports dynamic route parameters, but it uses a different syntax, which is why it has `/{id}` instead of `/:id`. The dynamic path parameters are available in the `request.pathParams` object. Finally, if your handler hasn't found the pizza you wanted, return a 404 error:

```
api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})
```

By default, API Gateway returns HTTP status 200 for all requests. Claudia API Builder helps you by setting some sane defaults, such as status 500 for errors, so your client application can handle request errors in promise catch blocks.

To customize the error status, you can pass a third parameter to the `api.get` function. For example, in your get `/pizza/{id}` function handler, besides the path and your handler function, you can pass an object with custom headers and statuses. To set the status error to 404, pass an object with the `error: 404` value in it.

You can see how your fully updated api.js file should look in the following listing.

Listing 2.6 The updated api.js

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')

api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})

api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

module.exports = api
```

Import the get-pizzas handler from your handlers directory.

Add a simple root route that returns static text to make your API user-friendly.

Replace the inline handler function with the new one you imported.

Add the route for finding one pizza by its ID.

Customize success and error status codes.

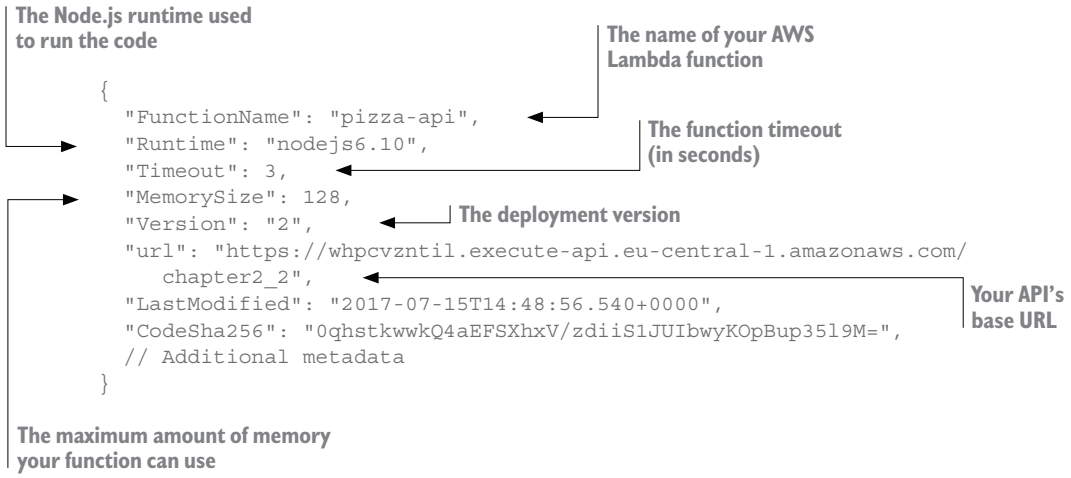
Now deploy your API again. To update your existing Lambda function along with its API Gateway routes, run the Claudia `update` command from your terminal:

```
claudia update
```

NOTE Because of the claudia.json file, the `claudia update` command knows exactly which Lambda function the files are deployed to. The command can be customized with a `--config` flag. For more information, see the official documentation at https://github.com/claudiajs/claudia/blob/master/docs/update.md.

After a minute or so, you should see a response similar to the one in listing 2.7. After processing the command and redeploying your application, Claudia will print out some useful information about your Lambda function and your API in the terminal. That information includes the function name, Node.js runtime, timeout, function memory size, and base URL of your API.

Listing 2.7 The printed information after running the `claudia update` command




```
{
  "FunctionName": "pizza-api",
  "Runtime": "nodejs6.10",
  "Timeout": 3,
  "MemorySize": 128,
  "Version": "2",
  "url": "https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/
     chapter2_2",
  "LastModified": "2017-07-15T14:48:56.540+0000",
  "CodeSha256": "0qhstkwwkQ4aEFSXhxV/zdiiS1JUIbwyKOpBup35l9M=",
  // Additional metadata
}
```

If you open this route link again from your browser (which should look similar to https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas), you see the array of all pizza objects from your data/pizza.js file.

When you open the other route link (something similar to https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas/1), you see only the first pizza. This response should look something like this:

```
{"id":1,"name":"Capricciosa","ingredients":["tomato
     sauce","mozzarella","mushrooms","ham","olives"]}
```

To test whether your API is working as expected, you should also try to get a pizza that doesn't exist. Visit your API URL with a nonexistent pizza ID, such as this one: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas/42. In this case, the response should look similar to this:

```
{"errorMessage" : "The pizza you requested wasn't found"}
```

Congratulations—your Pizza API is now capable of showing a list of pizzas to Aunt Maria's customers! This will make your Aunt Maria happy, but your API is not done yet. You need to implement the core feature of the API: creating a pizza order.

2.1.3 POSTing an order

Being able to create a pizza order via your API is important to Aunt Maria. Even though she is not as technically proficient as you are, she's aware that it will speed up pizza ordering and help her to quickly serve all the customers from her whole neighborhood, or even the whole town.

> **NOTE** In this example, you will learn about basic application structure, so to simplify things you will not store the orders anywhere. You will work with persistent storage in chapter 3.

To implement pizza order creation, you need to have a "create pizza order" route and a "create an order" handler, which means that you will need to create a new file in the handlers folder in your Pizza API project. As always, try to create simple and readable filenames. In this case, a good name for your handler file would be create-order.js.

First, create the new handler file, and open it in your favorite code editor. Next, create the `createOrder` function, and export it at the end of the file. Your handler function needs to accept some order data or an `order` object. At this moment, this `order` object should have only two attributes: the ID of the pizza a customer ordered and the customer address where the pizza should be delivered.

As a first step, check whether those two values have been passed within the `order` object. If not, throw an error.

The following part should implement storing the order to the database, but at the moment, you will just return an empty object if the `order` object is valid. You could store the object in a file, but a Lambda function can be deployed on multiple containers, and you have no control over that, so it's important not to rely on the local state. In the next chapter, you will learn how to connect your serverless function to a database and actually save an order.

Your create-order.js file should look like the one in the next listing.

Listing 2.8 Creating a pizza order handler

If the order object doesn't contain a pizzaId or a customer `address`, throw an error.

The createOrder handler function accepts the order object.

```
function createOrder(order) {
  if (!order || !order.pizzaId || !order.address)
    throw new Error('To order pizza please provide pizza type and address
     where pizza should be delivered')

  return {}
}

module.exports = createOrder
```

Otherwise, return an empty object.

Export the handler function.

Now that you have the handler for creating an order, it's time to create a route—but this one should accept `POST` requests. To do that, you'll need to go back to your api.js file. Like `api.get`, Claudia API Builder has an `api.post` method that receives three parameters: path, handler function, and options.

NOTE Besides `GET`, Claudia API Builder supports `POST`, `PUT`, and `DELETE` as HTTP verbs.

For the route path, you should write `/orders`, as your app is creating a new order. As the route handler function, import the create-order.js file you just made in your handlers folder. Finally, for the options parameter, pass customized statuses for both success and error: 201 and 400, respectively. Use the `success` attribute to add a custom status for success.

The `POST` request body is automatically parsed for you and available in the `request.body` attribute, which means that you don't need to use any additional middleware to parse the received data, such as the Express.js `body_parser`.

Parsing POST request body

The body of the `POST` request is automatically parsed by API Gateway. Claudia checks the body and normalizes it. For example, if the content type of the request is `application/json`, Claudia converts the empty body to an empty JSON object.

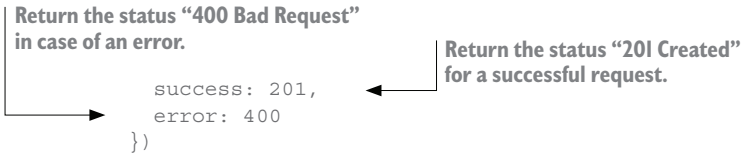
After you add the new route, your api.js file should look like the following listing.

Listing 2.9 Main API file updated with the new routes

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')
const createOrder = require('./handlers/create-order')

api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})

api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

api.post('/orders', (request) => {
  return createOrder(request.body)
}, {
```

Import the create-order handler from the handlers directory.

Add the POST /orders route to create an order and pass the request.body to the handler.




```
    success: 201,
    error: 400
})

module.exports = api
```

Again, deploy the API by running the `claudia update` command.

Trying out a `POST` request can be a bit trickier than testing a `GET`. You can't test it by opening the route URL in the browser. Hence, for the `POST` routes, you should use one of the free HTTP testing tools, such as `curl` or Postman.

NOTE From now on, you will see `curl` commands for all examples where you should try out your API endpoints. They aren't obligatory; you are free to use any tool you prefer.

curl and Postman

`curl` is a tool used in command lines or scripts to transfer data. It is also used in cars, television sets, routers, printers, audio equipment, mobile phones, tablets, set-top boxes, and media players, and is the internet transfer backbone for thousands of software applications affecting billions of humans daily. `curl` is designed to work without user interaction.

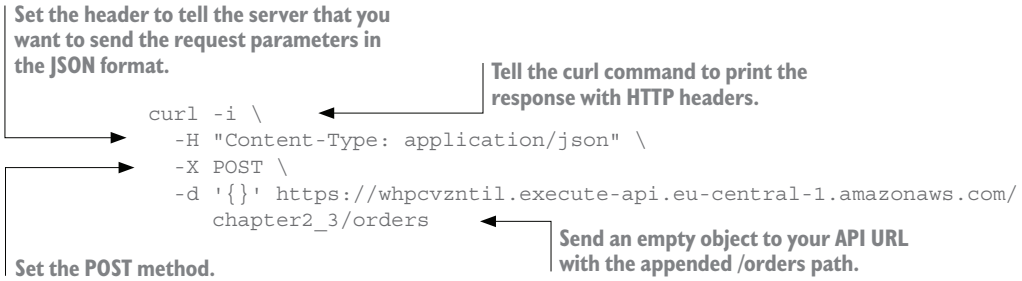
Postman is an application with a graphical user interface (GUI) that can also help you test your APIs. It can also speed up development, as you can build API requests and documentation through testing. It is available as an application for Mac, Windows, and Linux and as a Chrome plug-in.

You are going to test your `POST /orders` endpoint by using a `curl` command. In this command, you'll send an empty request body so you can check the validation error. Besides the `POST` body, you need to specify the method, provide a header to tell your API you are sending a JSON request, and specify the full URL you want to send the request to.

NOTE By default, `curl` doesn't print out the response HTTP status code. To check if your API is returning the correct status, use the `-w` flag and append the HTTP status after the API response.

You can see the command format in the following listing. This command has an empty body so you can test the error response.

Listing 2.10 `curl` command for testing `POST /orders` route (error response)



After you run the `curl` command from listing 2.10 in your terminal, the response should look like this, with a few additional headers:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 104
Date: Mon, 25 Sep 2017 06:53:36 GMT

{"errorMessage":"To order pizza please provide pizza type and address where
     pizza should be delivered"}
```

Now that you've verified the returned error when no order data is passed, you should also test a successful response. To do so, run a similar `curl` command from your terminal; change only the request body, as now it needs to contain a pizza ID and an address. The following listing shows the updated `curl` command. This command has a valid body so you can test the successful response.

Listing 2.11 `curl` command for testing the `POST /orders` route (successful response)

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizzaId":1,"address":"221B Baker Street"}' \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_3/orders
```

Send the pizza ID and address as a POST body.

This command returns the following:

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 2
Date: Mon, 25 Sep 2017 06:53:36 GMT

{}
```

This confirms that your API works correctly.

Now that you've learned the serverless API basics, it's time to take a look at what Claudia did when you ran the `claudia create` command.

2.2 How Claudia deploys your API

The previous examples demonstrated one of the main ideas of Claudia: single-command application deployment. There is no magic behind the tool, so every command can be explained easily.

Figure 2.4 represents the flow of events that happened when you ran the `claudia create` command. This simplified diagram is focused on the most important parts of the process for easier understanding. Also, some of the events described in this flow can be skipped or modified if you provide some flags with the `create` command. For example, Claudia can skip the first step and copy your code with all the local dependencies if you provide the flag `--use-local-dependencies`. For the full list of options, see https://github.com/claudiajs/claudia/blob/master/docs/create.md.

When you run the `claudia create` command, the first thing that Claudia does is zip your code without the dependencies and hidden files, using the `npm pack` command. Then it creates a copy of your project in a temporary folder in your system. This action ensures a clean and reproducible release, always starting from a well-known point and preventing problems caused by potential local dependencies. During this step, Claudia ignores your node_modules folder and all files ignored by Git or NPM. It also installs your production and optional dependencies using the `npm install --production` command.

Because the Lambda function requires the code with all its dependencies to be uploaded as a zip file, Claudia installs all production and optional NPM dependencies before compressing your project into a zip file.

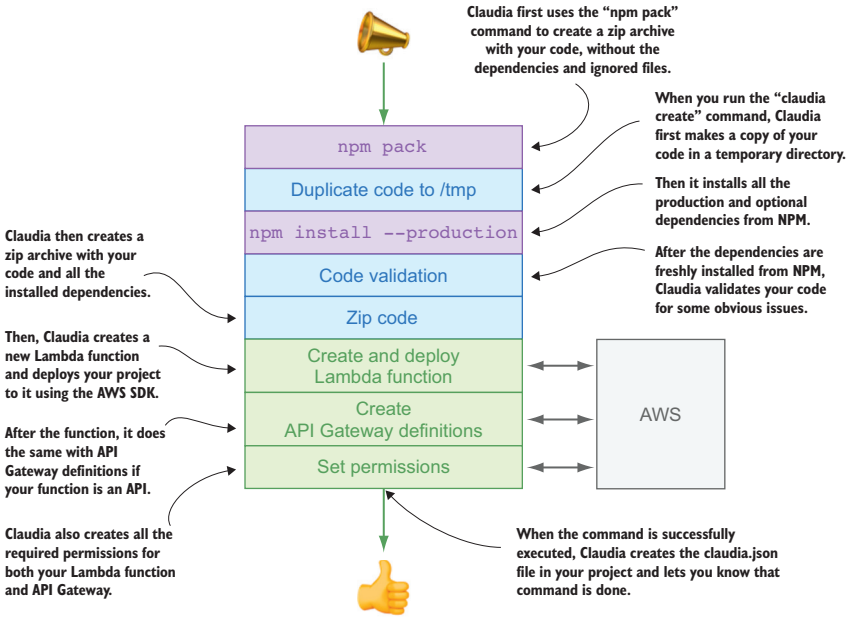


Figure 2.4 The `claudia create` process

Also, because debugging Lambda functions isn't straightforward, as you will see in chapter 5, Claudia also verifies that your project doesn't have any obvious issues, such as typos or your application invoking an undefined module. Take this step with a grain of salt, because it will do just a shallow validation. If you have a typo or an undefined function or module invocation inside the function or handler body, this step won't catch it.

As the next step, Claudia creates a zip file with your code with all the dependencies installed in the first step.

The last three steps in figure 2.4 aren't executed sequentially, but in parallel.

When the zip file is created, Claudia invokes the AWS API to create your Lambda function and uploads the archive. The interaction with the AWS platform is done through the AWS SDK module for Node.js. Before the code is uploaded, Claudia creates a new IAM user and assigns to the IAM user certain permissions to allow it to interact with AWS Lambda and API Gateway.

AWS IAM users, roles, and permissions

AWS Identity and Access Management (IAM) enables you to securely control access to AWS services and resources for your users. Using IAM, you can create and manage AWS users and groups, as well as use permissions to allow or deny any user or group access to your AWS resources.

A deeper explanation of IAM is beyond the scope of this book, but we highly recommend that you read more about it before progressing to the next few chapters. You can start with the official documentation: https://aws.amazon.com/iam/.

After your Lambda function is fully set up, Claudia sets up an API Gateway instance to it, defines all the routes, and sets their required permissions.

The `claudia update` command flow is almost identical to that of the `claudia create` command, but without some steps that have already been completed, such as role creation and permissions setup.

If you want to dive even deeper into Claudia and its commands, you can see its source code here: https://github.com/claudiajs/claudia.

Now that you know how Claudia works under the hood, the last piece of the API puzzle is understanding how API Gateway does the routing for your Pizza API.

2.3 Traffic control: How API Gateway works

In chapter 1 you learned that users can't interact with AWS Lambda outside of the AWS platform unless a trigger wakes up the function. One of Lambda's most important triggers is API Gateway.

As you can see in figure 2.5, API Gateway acts like a router or a traffic controller. It accepts HTTP requests (such as Pizza API requests from your web or mobile application), parses them to a common format, and routes them to one of your connected AWS services.

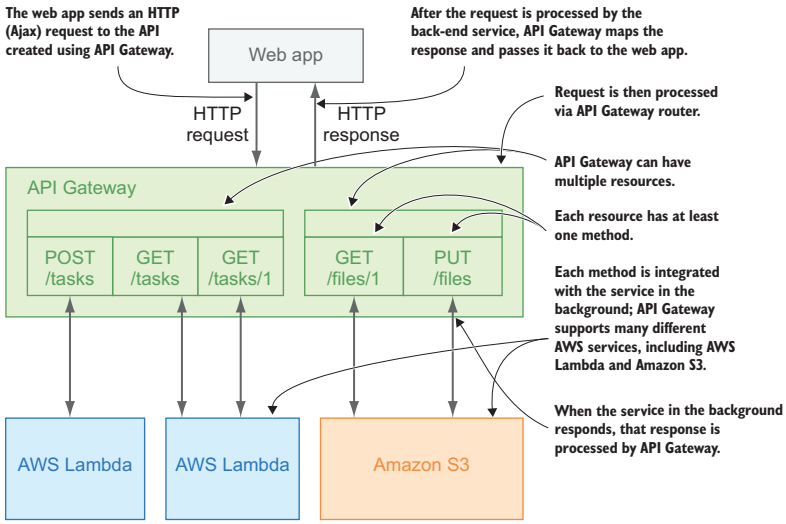


Figure 2.5 API Gateway routes requests to your AWS services.

API Gateway can be integrated with many AWS services, including AWS Lambda and Amazon S3. Each API on API Gateway can be connected to multiple services. For example, certain routes can invoke Lambda functions, whereas others can interact with some other service.

API Gateway offers another approach to HTTP request routing, called a *proxy router*. Instead of creating each route, a proxy router sends all requests to a single AWS Lambda function. This approach can be useful when you are creating a small API or when you want to speed up your deployment, because creating and updating multiple routes on API Gateway can take a few minutes, depending on your internet connection speed and the number of routes.

2.4 When a serverless API is not the solution

Even though we've just scratched the surface, you can already see how easy it is to build serverless APIs with Claudia.js and Claudia API Builder. Serverless APIs can be powerful and incredibly scalable, but in certain situations traditional APIs are a much better solution, such as the following:

- When request time and latency are critical. You can't guarantee minimal latency with serverless applications.
- When you need to guarantee a certain level of availability. In most cases, AWS will provide a pretty good level of availability, but sometimes that's not enough.
- When your application requires intensive and complex computing.
- When your API requires compliance with a specific standard. AWS Lambda and API Gateway might not be flexible enough.

2.5 Taste it!

After going through each chapter, have a "do it yourself" session. Most of the chapters give you a certain task, and you should try to implement it yourself. We provide a few useful hints, and the solutions are in the next section.

2.5.1 Exercise

In this chapter, you implemented the `GET /pizzas` and `POST /orders` API routes. To make your API more useful, there are two routes left: `PUT /orders` and `DELETE /orders`.

For the first exercise, do the following:

1. Create a handler for updating a pizza order, and add an API route for it.
2. Create a handler for deleting a pizza order, and add an API route for it.

In case you need some hints, here are a few:

- To add a `PUT` route, use the `api.put` method provided by Claudia API Builder.
- To add a `DELETE` route, use the `api.delete` method provided by Claudia API Builder.
- Both methods accept three arguments: a route path, a handler function, and an options object.
- Both paths require a dynamic parameter: an order ID.
- The `updateOrder` handler also requires a body with the new order details.
- Because you don't have the database yet, just return an empty object or a simple text message as a response.

When you finish the exercise, the file structure of your Pizza API should look like the one in figure 2.6.

If this exercise is too easy for you, and you want an additional challenge, try to add an API route for listing the pizza orders. There's no solution for this challenge in the next section, but that handler exists in the source code of Pizza API included with this book, so feel free to check the source and compare the solutions.

2.5.2 Solution

We hope you managed to finish the exercise on your own. Here are our solutions, so you can compare.

The first part of the exercise was to create a handler to update an order. To begin, you needed to create a file in your handlers folder, and name it update-order.js. In the file, you needed to create and export an `updateOrder` function that accepts an ID and the updated order details. The function should throw an error if the order ID or the updated order details object is not provided, or return a success message if successful. The code should look like the following listing.

Listing 2.12 Updating an order handler

If the ID or updates object is not passed, throw an error.

A handler function accepts the order ID and the order updates.

```
function updateOrder(id, updates) {
  if (!id || !updates)
    throw new Error('Order ID and updates object are required for updating
     the order')

  return {
    message: `Order ${id} was successfully updated`
  }
}

module.exports = updateOrder
```

Otherwise, return a success message.

Export the handler function.

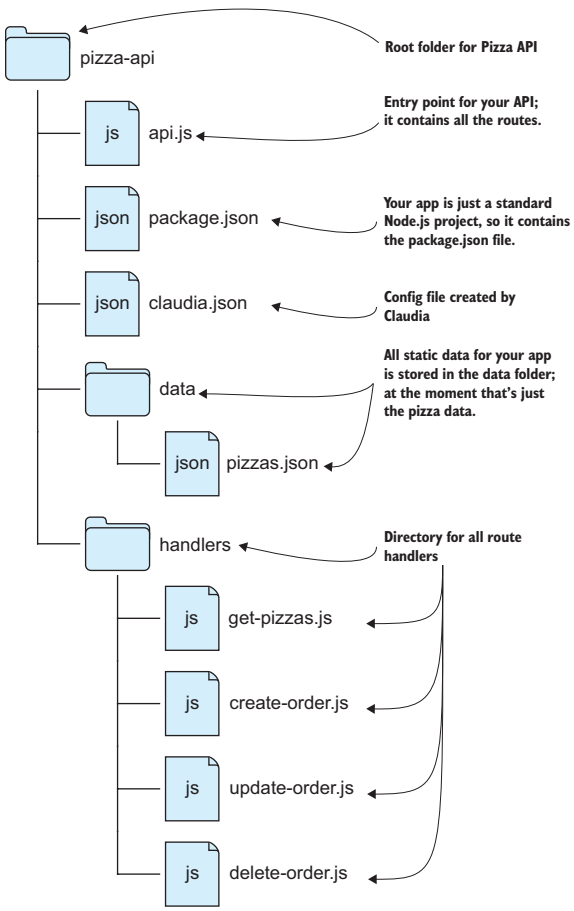


Figure 2.6 The updated file and folder structure of the Pizza API project

After you created the `updateOrder` function, you should have done the same for the handler to delete an order. First, you needed to create the delete-order.js file in your handlers folder. Then you should have created an exported `deleteOrder` function in the file. That function should accept an order ID. If the order ID isn't passed, the handler should throw an error; otherwise, it should return an empty object. The code should look like the following listing.

Listing 2.13 Deleting an order handler

```
function deleteOrder(id) {
  if (!id)
    throw new Error('Order ID is required for deleting the order')

  return {}
}

module.exports = deleteOrder
```

If an ID is not passed, throw an error.

A handler function accepts the order ID.

Otherwise, return an empty object.

Export the handler function.

Now, with the handlers implemented, your next step is to import them to api.js and create routes for updating and deleting the orders.

To update an order, use the `api.put` method, and use the `/orders/{id}` URL as the path; then set the handler function and the options with 400 as the status code for errors. You can't just pass the handler function you created in the previous step because it doesn't accept the full request object; instead, pass an anonymous function that invokes the `updateOrder` handler with an order ID from the received request body. The `DELETE /orders` route is the same except for two differences: it uses the `api.delete` method, and it doesn't pass the request body to the `deleteOrder` handler function.

After this step, your api.js file should look like the following listing.

Listing 2.14 The Pizza API with `PUT /orders` and `DELETE /orders` routes

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')
const createOrder = require('./handlers/create-order')
const updateOrder = require('./handlers/update-order')
const deleteOrder = require('./handlers/delete-order')

// Define routes
api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})
api.get('/pizzas/{id}', (request) => {
```

Import the update-order handler from the handlers directory.

Import the delete-order handler from the handlers directory.

```
  return getPizzas(request.pathParams.id)
}, {
  error: 400
})

api.post('/orders', (request) => {
  return createOrder(request.body)
}, {
  success: 201,
  error: 400
})
api.put('/orders/{id}', (request) => {
  return updateOrder(request.pathParams.id, request.body)
}, {
  error: 400
})
api.delete('/orders/{id}', (request) => {
  return deleteOrder(request.pathParams.id)
}, {
  error: 400
})

module.exports = api
```

Add a route for PUT /orders and connect a handler.

Add a route for DELETE /orders and connect a handler.

Both routes return status 400 in case of an error.

As always, open your terminal, navigate to your pizza-api folder, and run the `claudia update` command from it to update your Lambda function and API Gateway definition.

When Claudia updates your Pizza API, you can use the `curl` commands from listings 2.15 and 2.16 to test your new API endpoints. These commands are almost the same as the command you used for the `POST` request, with the following differences:

- The HTTP method is different: You use `PUT` for updating and `DELETE` for deleting the order.
- Updating the order needs to pass the body with the updates.
- Deleting the order doesn't require the request body.

These commands each have a valid body and should return a successful response.

Listing 2.15 `curl` command for testing `PUT /orders/{id}` route

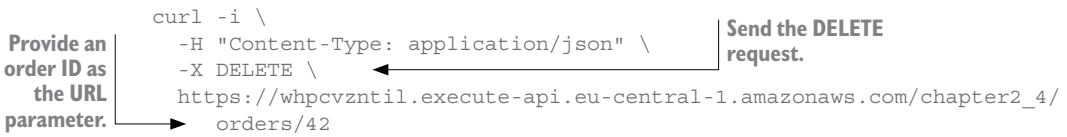
Add extra pepperoni to your order.

```
curl -i \
  -H "Content-Type: application/json" \
  -X PUT \
  -d '{"pizzaId":2}' \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_4/
     orders/42
```

Send the PUT request.

Add an order ID as a path parameter.

Listing 2.16 `curl` command for testing `DELETE /orders/{id}` route




```
curl -i \
  -H "Content-Type: application/json" \
  -X DELETE \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_4/
    orders/42
```

When you execute the commands in your terminal, they return the responses `{"message":"Order 42 was successfully updated"}` and `{}`, respectively, both with status 200.

Summary

- Claudia enables you to deploy your API to API Gateway and AWS Lambda in a single command.
- Updating your API takes a single Claudia command, too.
- A serverless API on AWS Lambda doesn't require any specific folder structure or organization.
- API Gateway acts as a router and can invoke various services.
- If you want to bundle more routes into a single AWS Lambda function, you need internal routing.
- Claudia API Builder has a router identical to the routers in other popular Node.js web API libraries.
- Serverless APIs are powerful, but they are not a silver bullet, so depending on your case, a traditional API might work better.

Chapter 3 from *Serverless Applications with Node.js*
by Slobodan Stojanović and Aleksandar Simović

In this chapter, you'll discover how serverless affects asynchronous communication and how it works with Claudia. You'll also learn the basics of JavaScript promise, and continuing on with your API, you'll add the ability to store orders by connecting to DynamoDB from Claudia and AWS Lambda.

3 Asynchronous work is easy, we Promise()

This chapter covers

- Handling asynchronous operations with Claudia
- The basics of JavaScript promises
- Connecting to DynamoDB from Claudia and AWS Lambda

In the previous chapter, you created a simple API for handling pizza information and orders. You also learned that unlike with a traditional Node.js server, AWS Lambda state is lost between subsequent invocations. Therefore, a database or an external service is required to store Aunt Maria's pizza orders or any other data you want to keep.

As Node.js executes asynchronously, you will first learn how serverless affects asynchronous communication: how it works with Claudia, and, more importantly, the recommended way of developing your serverless applications. As you grasp these concepts, you will see how easy it is to connect AWS Lambda to an external service, and you will learn how to use it to store your pizza orders by using AWS DynamoDB.

Because our brains aren't good at asynchronous reading, and books are written in a synchronous manner, let's go step by step.

3.1 Storing the orders

Ring, ring! You just had a short phone call with Aunt Maria. She is impressed by your speed, though she still can't use your application, as you aren't storing any of her pizza orders. She still needs to use the old pen-and-paper method. To complete the basic version of your Pizza API, you need to store your orders somewhere.

Before starting development, you should always have an idea of which details you want to store. In your case, the most elementary pizza order is defined by the selected pizza, the delivery address, and the order status. For clarity, this kind of information is usually drawn as a diagram. So as a small exercise, take a minute to try to draw it yourself.

Your diagram should be similar to figure 3.1.

Now that you have an idea of what to store, let's see how you should structure it for the database. As you previously learned, you can't rely on AWS Lambda to store state, which means that storing order information in your Lambda filesystem is off the table.

In a traditional Node.js application, you would use some popular database, such as MongoDB, MySQL, or PostgreSQL. In the serverless world, each of the serverless providers has a different combination of data storage systems. AWS doesn't have an out-of-the-box solution for any of those databases.

As the easiest alternative, you can use Amazon DynamoDB, a popular NoSQL database that can be connected to AWS Lambda easily.

NOTE AWS Lambda is not limited to DynamoDB, and you can use it with other databases, but that's beyond the scope of this book.

So what is DynamoDB?

DynamoDB is a fully managed, proprietary NoSQL database service offered by Amazon as part of its AWS portfolio. DynamoDB exposes a similar data model to and derives its name from Dynamo, a highly available key-value structured storage system with a different underlying implementation.

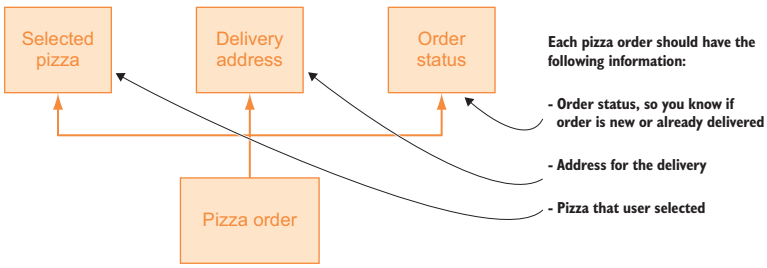


Figure 3.1 The most basic pizza order

To put it simply, DynamoDB is just a database building block for serverless applications. DynamoDB is to NoSQL databases what AWS Lambda is to computing functions: a fully managed, autoscaled, and relatively cheap cloud database solution.

DynamoDB stores the data in its data tables. A data table represents a collection of data. Each table contains multiple items. An item represents a single concept described by a group of attributes. You can think of an item as a JSON object, because it has the following similar characteristics:

- Its keys are unique.
- It doesn't limit how many attributes you can have.
- Values can be different types of data, including numbers, strings, and objects.

The table is just the storage representation of the model you previously defined, as shown in figure 3.1.

Now you need to transform your previously defined model to the structure your database understands: a database table. While you are doing that, keep in mind that DynamoDB is almost schemaless, which means that you need to define only your primary key and can add everything else later. As a first step, you'll design a minimum viable table for your orders.

Ready?

As in any other database, you want to store each order as one item in the database table. For your pizza order storage, you'll use a single DynamoDB table, which will be a collection of your orders. You want to receive your orders via an API and store them to the DynamoDB table. Each order can be described by a set of its characteristics:

- Unique order ID
- Pizza selection
- Delivery address
- Order status

You can use those characteristics as keys in your table. Your orders table should look like table 3.1.

Table 3.1 The structure of an orders table in DynamoDB

Order ID	Order status	Pizza	Address
1	pending	Capricciosa	221B Baker Street
2	pending	Napoletana	29 Acacia Road

The next step is to create your table—let's name it `pizza-orders`. As with most things in AWS, you can do this several ways; our preferred method is to use the AWS CLI. To create a table for the orders, you can use the `aws dynamodb create-table` command, as shown in listing 3.1.

You need to supply a few required parameters when creating the table. First, you need to define your table name; in your case, it will be `pizza-orders`. Then you need to define your attributes. As we mentioned, DynamoDB requires only primary key definition, so you can define only the `orderId` attribute and tell DynamoDB that it will be of type string. You also need to tell DynamoDB that `orderId` will be your primary key (or, in DynamoDB's world, *hash key*).

After that, you need to define the provisioned throughput, which tells DynamoDB what read and write capacity it should reserve for your application. Because this is a development version of your application, setting both read and write capacity to 1 will work perfectly fine, and you can change that later through the AWS CLI. DynamoDB supports autoscaling, but it requires the definition of the minimum and maximum capacity. At this point, you won't need to use autoscaling, but if you want to learn more about it, visit http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html.

Finally, you need to select the region where you want to create your table. Pick the same region as you did with your Lambda function to decrease latency in database communication. The following listing shows the complete command.

Listing 3.1 Create a DynamoDB table using the AWS CLI

Create a pizza-orders table using the AWS CLI.

Provide an attribute definition and tell DynamoDB that your primary key is of type string (S).

```
aws dynamodb create-table --table-name pizza-orders \
  --attribute-definitions AttributeName=orderId,AttributeType=S \
  --key-schema AttributeName=orderId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
  --region eu-central-1 \
  --query TableDescription.TableArn --output text
```

Provide a key schema.

Select the region for your DynamoDB table.

Set the throughput (read and write capacity) for the DynamoDB table.

Print back the table's Amazon Resource Name (ARN) to confirm that everything is set up correctly.

TIP Adding the `--query` attribute in AWS CLI commands will filter the output and return only the values you need. For example, `--query TableDescription.TableArn` returns only the table's ARN.

You can also define the type of your output by using the `--output` attribute along with the value. For example, `--output text` returns the result as plain text.

When you run the command in listing 3.1, it prints the ARN of your DynamoDB table and looks similar to this:

```
arn:aws:dynamodb:eu-central-1:123456789101:table/pizza-orders
```

That's it! Now you have the `pizza-orders` DynamoDB table. Let's see how you can connect it to your API's route handlers.

To be able to connect to your DynamoDB table from Node.js, you need to install the AWS SDK for Node.js. You can get the `aws-sdk` from NPM, as you would any other module. In case you are unfamiliar with that process, see appendix A.

You now have all the ingredients, and it's time for the most important step: combine all the pieces, just as you would prepare a pizza. (Fortunately for you, we have a pizza recipe in the last appendix.)

The easiest way to communicate with DynamoDB from your Node.js application is through the `DocumentClient` class, which requires asynchronous communication. `DocumentClient`, like any part of the AWS SDK, works perfectly with Claudia, and you will use it in the API route handlers you made in chapter 2.

DynamoDB DocumentClient

`DocumentClient` is a class of the DynamoDB subset of the AWS SDK. Its goal is to simplify working with table items by abstracting the operations. It exposes a simple API, and we'll cover only the pieces you need later in this chapter. In case you want to see the API documentation, it's available here: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html.

Connecting your pizza order API to the newly created database is easy. Storing an order to your DynamoDB table takes just two steps:

1. Import the AWS SDK, and initialize the DynamoDB `DocumentClient`.
2. Update your `POST` method to save an order.

Because you split your code into separate files in chapter 2, let's start with the create-order.js file in the handlers folder. The following listing shows how to update create-order.js to save a new order to the `pizza-orders` DynamoDB table.

Listing 3.2 Saving an order to the DynamoDB table

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function createOrder(request) {
  if (!request || !request.pizza || !request.address)
    throw new Error('To order pizza please provide pizza type and address
     where pizza should be delivered')

  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: 'some-id',
      pizza: request.pizza,
      address: request.address,
      orderStatus: 'pending'
    }
  }).promise()
```

Import the AWS SDK and initialize the DynamoDB DocumentClient.

Put the new pizza order in the DynamoDB table.

The ID can be any string—hardcode it for now.

The DocumentClient instance has a .promise method that returns a promise.

```
    .then((res) => {
      console.log('Order is saved!', res)
      return res
    })
    .catch((saveError) => {
      console.log(`Oops, order is not saved :(`, saveError)
      throw saveError
    })
}

module.exports = createOrder
```

Log the response and return the data if the promise is fulfilled.

If the promise is rejected, log an error and throw it again so you can use the error in the api.js file.

Export the handler function.

When you finish this step, the `POST /orders` method of your Pizza API should look and work the way it is presented in figure 3.2.

Let's explain what happens here. After importing the AWS SDK, you need to initialize the DynamoDB `DocumentClient`. Then you can replace the empty object you are returning on line 7 of your create-order.js handler with the code that saves an order to your table, using the `DocumentClient` you imported previously.

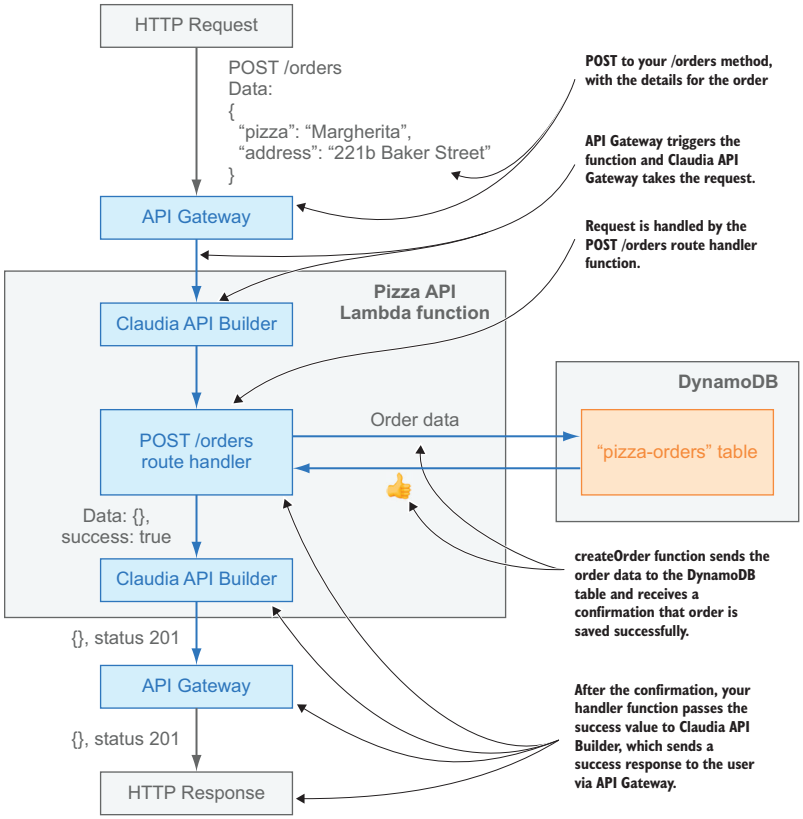


Figure 3.2 The flow of the `POST /orders` method of your Pizza API with DynamoDB integration

To save an order to DynamoDB, you use the `DocumentClient.put` method that puts a new item in the database, either by creating a new one or replacing an existing item with the same ID. The `put` method expects an object that describes your table by providing the `TableName` attribute and the item by providing the following `Item` attribute as an object. In your database table plan, you decided that your item should have four attributes—ID, pizza, address, and order status—and that's exactly what you want to add to the `Item` object you are passing to the `DocumentClient.put` method.

As Claudia API Builder expects a promise for async operations, you should use the `.promise` method of `DocumentClient.put`. The `.promise` method converts a reply to a JavaScript promise. Some of you are probably wondering if there are any differences in how promises work in serverless applications and how Claudia handles asynchronous communication. The following section gives a short explanation of promises and how they work with Claudia and Claudia API Builder. If you are already familiar with these concepts, jump to section 3.3.

3.2 Promising to deliver in less than 30 minutes!

The pizzeria processes include dough rising, baking, pizza ordering, and so on. These are asynchronous operations. If they were synchronous, Aunt Maria's pizzeria would be blocked and stopped from working on anything else until the operation in progress finished. For example, you would wait until the dough had risen, and then do something else. And for such time-wasting, Aunt Maria would fire anyone, even you! Because most of the JavaScript runtimes are single-threaded, many longer operations, such as network requests, are executed asynchronously. Asynchronous code execution is handled by two known concepts: callbacks and promises. At the time of this writing, promises are the recommended way to go in all Node.js applications. We do not explain callbacks, as you are most likely already familiar with them.

Asynchronous promise

A *promise* represents an eventual result of an asynchronous operation.

A promise is like a real-world promise made to partners, friends, parents, and kids:

- "Honey, will you please take out the garbage?"
- "Yes, dear, I promise!"

And a couple of hours later, guess who took out the garbage?

Promises are just pretty wrappers around callbacks. In real-world situations, you wrap a promise around a certain action or operation. A promise can have two possible outcomes: it can be *resolved* (fulfilled) or *rejected* (unfulfilled).

Promises can have conditions related to them, and this is where their asynchronous power comes into play:

- "Johnny, when you finish your homework, you will be able to go out and play!"

This example displays how certain actions can occur only after fulfilling a certain asynchronous operation. In the same way, the execution of certain code blocks waits for the completion of a defined promise.

The following listing is a JavaScript promise representation of the example sentence.

Listing 3.3 Johnny's play—the promise way

```
function tellJohhny(homework) {
  return finish(homework)
    .then(finishedHomework => {
      return getOut(finishedHomework);
    })
    .then(result => {
      return play();
    })
    .catch(error => {
      console.log(error);
    });
}
```

finish, getOut, and play are async functions, but they all return promises, which can be chained.

getOut is invoked only after Johnny finishes his homework.

Catch the errors from any of the functions in the promise chain.

Promises have several features:

- *Promise chaining*—As in listing 3.3, you can easily *chain* one promise to another, passing the results from one code block to the next without any hassle.
- *Parallel execution*—You can execute two functions at the same time and get the results of both just once.
- *Proper asynchronous operation rejection*—If a function gives an error or doesn't give a good result, you can reject it and stop its execution at any time. By contrast, with callbacks, rejecting the promise stops the full chain of promises.
- *Error recovery*—The promise `catch` block allows you to easily and properly manage errors and propagate them to the responsible error handler.

Some customers order multiple pizzas in one order, but those pizzas are not delivered one by one. If they were, customers would be furious with such an inefficient process. Instead, the pizza chef usually bakes them all at the same time; then the delivery person waits until *all* of them are finished before delivery.

The following listing is a code representation of this process.

Listing 3.4 Pizza parallel baking

```
function preparePizza(pizzaName) {
  return new Promise((resolve, reject) => {
    // prepare pizza
    resolve(bakedPizza);
  });
}

function processOrder(pizzas) {
  return Promise.all([
```

```
    preparePizza('extra-cheese'),
    preparePizza('anchovies')
  ]);
}

return processOrder(pizzas)
  .then((readyPizzas) => {
    console.log(readyPizzas[0]); // prints out the result from the extra-
     cheese pizza
    console.log(readyPizzas[1]); // prints out the result from the anchovies
     pizza
    return readyPizzas;
  })
```

As you can see in listings 3.3 and 3.4, promises help a lot. They allow you to handle any situation Aunt Maria's pizzeria could have and also help you properly describe all the processes. Claudia fully supports all promise features, so you can easily use them.

In the next listing, you can see a simple Claudia example of a handler replying after one second. Because `setTimeout` is not returning a promise, you need to wrap it by using a `new Promise` statement.

Listing 3.5 Wrapping an async operation that doesn't support promises with a promise

```
const Api = require('claudia-api-builder')
const api = new Api()

api.get('/', request => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello after 1 second')
    }, 1000)
  })
})

module.exports = api
```

Wrap the async operation with a JavaScript promise.

Use the resolve method to send a response back to Claudia API Builder.

Execute setTimeout with a one-second delay.

As you see in listing 3.5, as opposed to some popular Node.js frameworks, Claudia API Builder only exposes the request in the route handler. In chapter 2, to reply to it you would return a value, but in the case of an asynchronous operation, you should return a JavaScript promise. Claudia API Builder receives it, waits for it to be resolved, and uses the value returned as a reply.

> **NOTE** The AWS SDK has out-of-the-box support for JavaScript promises. All the SDK classes have a `promise` method that can, instead of default callback behavior, return a promise.

3.3 Trying out your API

After the small detour into the world of promises, run `claudia update` again from your pizza-api folder and deploy the code. In less than a minute, you'll be able to test your API and see if it works.

To test your API, reuse the `curl` command from chapter 2:

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizza":4,"address":"221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_1/orders
```

> **NOTE** Don't forget to replace the URL in your `curl` command with the URL you got from the `claudia update` command.

Oh! The `curl` command returns this:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 219
Date: Mon, 25 Sep 2017 06:53:36 GMT

{"errorMessage":"User: arn:aws:sts::012345678910:assumed-role/pizza-api
     -executor/book-pizza-api
is not authorized to perform: dynamodb:PutItem on resource:
arn:aws:dynamodb:eu-central-1:012345678910:table/pizza-orders"}
```

What's wrong?

This error is telling you that the role your Lambda function is using (`arn:aws:sts::012345678910:assumed-role/pizza-api-executor/book-pizza-api`) is not allowed to perform a `dynamodb:PutItem` command on your DynamoDB database (`arn:aws:dynamodb:eu-central-1:012345678910:table/pizza-orders`).

To fix the issue, you need to add an IAM policy that allows your Lambda function to communicate with your database. You can do that with `claudia create` by providing a `--policies` flag. Be careful, though; that flag doesn't work with the `claudia update` command, as Claudia never duplicates things that you can do with a single AWS CLI command.

> **NOTE** In AWS, everything is enclosed in IAM policies, which are something like authorization policies. An IAM policy is similar to a passport visa. To enter a certain country, you need to have a valid visa.

First, define a role in a JSON file. Create a new folder in your project root, and call it roles. Then create a role file for DynamoDB. Call it dynamodb.json, and use the content from the following listing. You want to allow your Lambda function to get, delete, and put items in the table. Because you might have more tables in the future, apply this rule to all tables, not just the one you have right now.

Listing 3.6 JSON file that represents DynamoDB role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:Scan",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Define a version.

Define a statement for this role.

Define the specific actions this role allows or denies.

Enable (allow) the actions you defined.

This rule applies the role to all DynamoDB tables, not a specific one.

TIP You probably want to have more precise roles in a production app, and you definitely don't want your Lambda function to be able to access all DynamoDB tables. To read more about roles and policies, visit http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

Now you can use the AWS CLI `put-role-policy` command to add a policy to your role, as shown in the next listing. To do so, you'll need to provide the role that your Lambda function is using, the name of your policy, and the absolute path to your dynamodb.json file. Where can you find the role? Remember the claudia.json file that Claudia created in the root folder of your project? Open that file, and you'll see the `role` attribute in the `lambda` section.

Listing 3.7 Add a policy to the Lambda role to allow it to communicate with DynamoDB tables.

Use the put-role-policy command from the iam section of the AWS CLI to add the policy.

Attach the policy to the Lambda role you got from the claudia.json file.

```
aws iam put-role-policy \
  --role-name pizza-api-executor \
  --policy-name PizzaApiDynamoDB \
  --policy-document file://./roles/dynamodb.json
```

Name your policy.

Use the dynamodb.json file as a source for creating the policy.

NOTE You need to provide a path to dynamodb.json with the `file://` prefix. If you are providing an absolute path, keep in mind that you will have three slashes after `file:`. The first two are for `file://`, and the third one is from the absolute path, because it starts with a slash.

When you run the command from listing 3.7, you won't get any response. That's OK, because an empty response means that everything went well.

Now, rerun the same `curl` command and try to add an order:

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizza":4,"address":"221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_1/orders
```

NOTE You don't need to redeploy your code, because you didn't change it. The only thing you updated was the role for your Lambda function.

The `curl` command should return {} with status 201. If that's the case, congratulations! Your database connection is working! But how do you see whether the order was really saved to the table?

The AWS CLI has an answer to that question, too. To list all the items in your table, you can use the `scan` command in the `dynamodb` section of the AWS CLI. The `scan` command returns all the items in the table unless you provide a filter. To list all the items in the table, run the command in the following listing from your terminal.

Listing 3.8 An AWS CLI command that lists all the items from the `pizza-orders` table

The scan command lists all the items from the table.

The command requires a table name as a parameter.

```
aws dynamodb scan \
  --table-name pizza-orders \
  --region eu-central-1 \
  --output json
```

You can specify the format of the response.

This command "scans" your `pizza-orders` table and returns the result as a JSON object. You can change the output value to `text`, and you'll get the result in text format. A few more formats are available, including XML.

The command should return something like the value in the following listing: a JSON response with the count and an array of all your table items.

Listing 3.9 Response from the `scan` command for your `pizza-orders` table

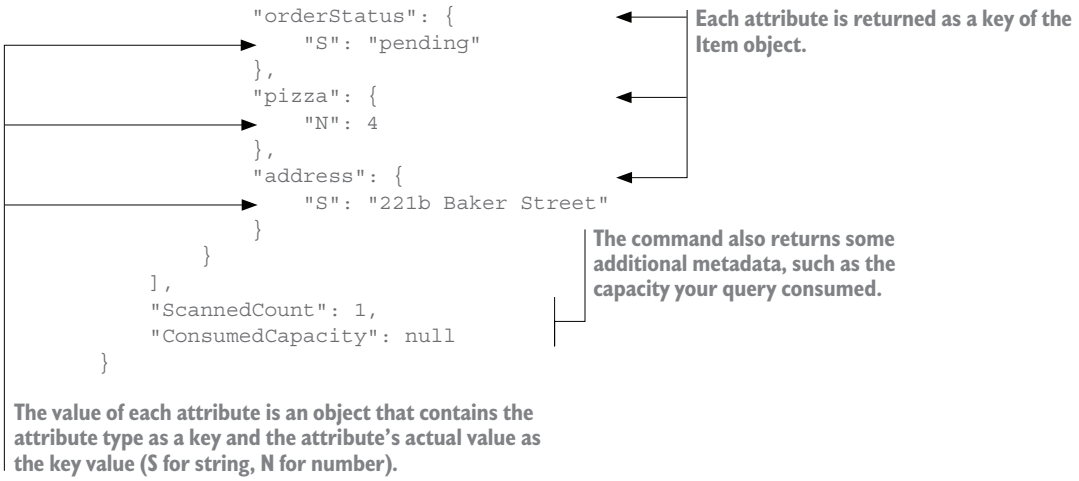
This returns a count of all table items.

Items are returned as objects in the Items array.

Each attribute is returned as a key of the Item object.

The value of each attribute is an object that contains the attribute type as a key and the attribute's actual value as the key value (S for string, N for number).

```
{
    "Count": 1,
    "Items": [
        {
            "orderId": {
                "S": "some-id"
            },
```



Awesome—it seems that your API is working as expected!

Try to add another pizza order now with the same `curl` command—for example, a Napoletana for 29 Acacia Road. If you then run the AWS CLI command from listing 3.8 again to scan the database, you'll see only one item in your table; the previous one doesn't exist anymore.

Why did that happen?

Remember that you hardcoded an `orderId` in your create-order.js handler, as shown in listing 3.2?

Each of the orders should have a unique primary key, and you used the same one, so your new entry replaced the previous one.

You can fix that by installing the `uuid` module from NPM and saving it as a dependency. `uuid` is a simple module that generates universally unique identifiers.

Universally unique identifiers

A *universally unique identifier* is a 128-bit value used to identify information in computer systems. It's better known by the abbreviation UUID. Sometimes it's called a globally unique identifier (GUID).

UUIDs are standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). To learn more about the UUID standard, see RFC 4122 (the specification that describes it), available here: http://www.ietf.org/rfc/rfc4122.txt.

After you download the module, update your create-order.js handler as shown in the next listing. You can simply import and invoke the `uuid` function to get a unique ID for the order. Keep in mind that this listing shows only the part of the create-order.js file affected by this change; the rest of the file is the same as the one in listing 3.2.

Listing 3.10 Adding UUIDs for the orders while creating them

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const uuid = require('uuid')

function createOrder(request) {
  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: uuid(),
      pizza: request.pizza,
      address: request.address,
      status: 'pending'
    }
  }).promise()
// Rest of the file stays the same
```

Import the uuid module that you've installed from NPM.

Invoke the uuid function to get a unique ID for the order.

The rest of the file stays as it is in listing 3.2.

After you redeploy the code by invoking the `claudia update` function, use the same `curl` command to test your API again and then scan the database with the AWS CLI command from listing 3.8. As you can see, the new `orderId` for your new order is some unique string like this one: `8c499027-a2d7-4ad9-8360-a49355021adc`. If you add more orders, you'll see that all of them are now saved in the database, as expected.

3.4 Getting orders from the database

After storing an order in the database, retrieving one should be fairly easy. The `DocumentClient` class has a `scan` method, which you can use to retrieve the orders.

The `scan` method works the same way as in the AWS CLI, with a small difference: You need to pass an object to it as a parameter, along with some options. In the options, the only required attribute is the name of your table.

Besides scanning the database, your get-orders.js handler can get a single item by an ID. You can do that with a scan by filtering the results, but that's inefficient. A more efficient way is to use the `get` method, which works almost the same way but requires a key for your item, too.

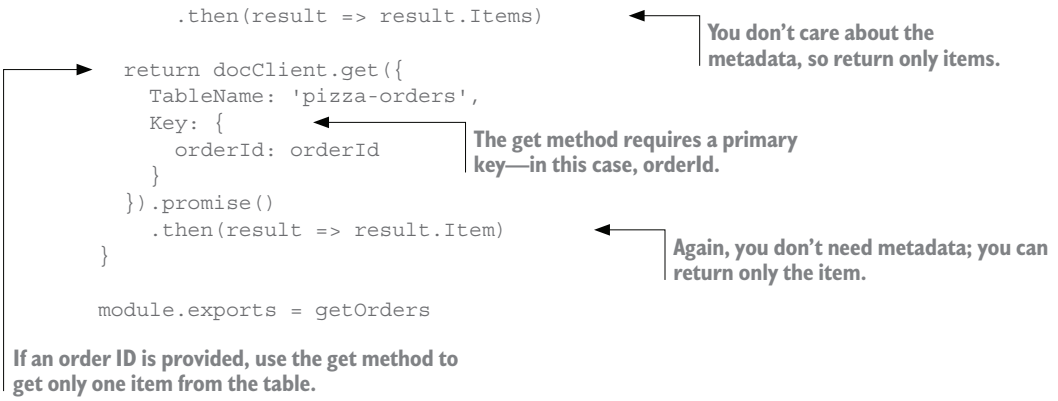
Let's update your get-orders.js file in the handlers folder to scan the orders from your table, or to get a single item if an order ID is provided. When you update your code, it should look like the code in the following listing. Once you've made these changes, deploy the code using the `claudia update` command.

Listing 3.11 get-orders.js handler reads the data from the `pizza-orders` table

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function getOrders(orderId) {
  if (typeof orderId === 'undefined')
    return docClient.scan({
      TableName: 'pizza-orders'
    }).promise()
```

Import and initialize DocumentClient.

Scan the pizza-orders table.



Let's test it! First, scan all the orders with the following `curl` command:

```
curl -i \
  -H "Content-Type: application/json" \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_2/orders
```

When you run it, it should display something like this:

```
HTTP/1.1 200 OK

[{
  "address": "29 Acacia Road",
  "orderId": "629d4ab3-f25e-4110-8b76-aa6d458b1fce",
  "pizza": 4,
  "orderStatus":"pending"
}, {
  "address": "29 Acacia Road",
  "orderId": "some-id",
  "pizza": 4,
  "status": "pending"
}]
```

Don't worry if the order ID is different from yours; it should be unique.

Now try using an ID from one of the returned orders to get a single order. You can do that by running the following `curl` command from your terminal:

```
curl -i \
  -H "Content-Type: application/json" \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_2/
     orders/629d4ab3-f25e-4110-8b76-aa6d458b1fce
```

The result should look something like this:

```
HTTP/1.1 200 OK

{
  "address": "29 Acacia Road",
  "orderId": "629d4ab3-f25e-4110-8b76-aa6d458b1fce",
  "pizza": 4,
  "status": "pending"
}
```

It works! Awesome and easy, right?

3.5 Taste it!

As you've seen, saving orders to the database and retrieving them is easy. But Aunt Maria has told you that sometimes customers make mistakes and order the wrong pizza, so she wants the capability to change or cancel a pizza order.

3.5.1 Exercise

To fulfill Aunt Maria's request, you need to connect two more API endpoints to the database:

1. Update the update-order.js handler to update an existing order in the `pizza-orders` DynamoDB table.
2. Update the delete-order.js handler to delete an order from the `pizza-orders` DynamoDB table.

When you finish both endpoints, your API should have the same structure as the one in figure 3.3.

The solution's code is in the next section. Before looking at it, try to complete the exercise yourself, but if you're struggling, peek a little.

A few hints:

- You should use DynamoDB's `DocumentClient` for both updates and deletions.
- To update an existing order, use the `DocumentClient.update` method. Besides `TableName`, this method requires a few more items in the object you are providing, including `Key`, `UpdateExpression`, and others. See the official documentation

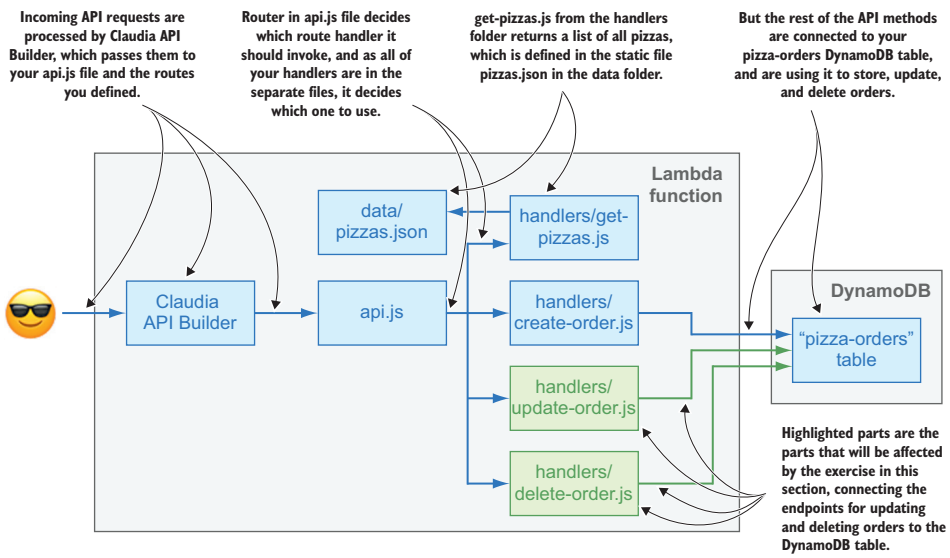


Figure 3.3 The Pizza API after connecting all order endpoints to the DynamoDB table, with the parts of the app that need to be addressed by this exercise highlighted

for the full list: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html#update-property.

- If the `update` method seems too complex for you, remember that `DocumentClient.put` will replace an existing order with a new one, so you can try using that one.
- To delete an existing order, use the `DocumentClient.delete` method. To delete an item, you need to provide an object that contains the `TableName` and the `Key` for that item. For more information, see the official documentation: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html#delete-property.
- Don't forget to return a promise and to pass the value.

In case this is too easy, here are a few additional things you can do:

- Update update-order.js and delete-order.js to affect pending orders only, because you don't want customers to be able to change an order if the pizza is ready and being delivered.
- Update get-orders.js to be able to filter by order status, and by default return only pending orders.

The solutions to these additional tasks are available in the final application source code, along with code annotations.

3.5.2 Solution

Finished already or peeking a little? If you're finished, that's great, but even if you weren't able to complete the exercise without any help, don't worry. DynamoDB is a bit different from the other popular noSQL databases, and you may need more time and practice to understand it.

Let's take a look at the solution. The following listing shows the updates for the update-order.js file in the handlers folder of your project.

Listing 3.12 Updating an order in the pizza-orders DynamoDB table

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function updateOrder(orderId, options) {
  if (!options || !options.pizza || !options.address)
    throw new Error('Both pizza and address are required to update an order')

  return docClient.update({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    },
    UpdateExpression: 'set pizza = :p, address=:a',
```

Import and initialize the DynamoDB DocumentClient.

Pass an ID and an object with the attributes you want to update.

Define the key for your order.

Describe how the update will modify attributes of an order.

```
    ExpressionAttributeValues: {
      ':p': options.pizza,
      ':a': options.address
    },
    ReturnValues: 'ALL_NEW'
  }).promise()
    .then((result) => {
      console.log('Order is updated!', result)
      return result.Attributes
    })
    .catch((updateError) => {
      console.log(`Oops, order is not updated :(`, updateError)
      throw updateError
    })
}

module.exports = updateOrder
```

Provide the values to the UpdateExpression expression.

Tell DynamoDB that you want a whole new item to be returned.

Just log the response or error and pass the value—you'll use this in chapter 5 for debugging purposes.

Export the handler.

It's not that different from create-order.js. The two major differences are

- Using the `DocumentClient.update` method with a `Key`, which is `orderId` in your case
- Passing more values to the function because you need an `orderId` and new values to update (`pizza` and `address`)

TIP Update syntax can be a bit confusing because of its `UpdateExpression`, `ExpressionAttributeValues`, and `ReturnValues` attributes. But the attributes are quite simple. The annotations of listing 3.12 provide a basic explanation. For more details, check the official documentation at http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.UpdateExpressions.html.

The following listing shows the updates for the delete-order.js file in your handlers folder. The required updates are similar to those in both the create-order.js and update-order.js files; the only difference is that you're using the `DocumentClient.delete` method here.

Listing 3.13 Deleting an order from the pizza-orders DynamoDB table

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function deleteOrder(orderId) {
  return docClient.delete({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    }
  }).promise()
    .then((result) => {
```

Use the DocumentClient.delete method to delete an order.

Import and initialize the DynamoDB DocumentClient.

Pass an order ID.

Provide an orderId, the primary key for your table.

Don't forget to use the .promise method to return a promise.

```
      console.log('Order is deleted!', result)
      return result
    })
    .catch((deleteError) => {
      console.log(`Oops, order is not deleted :(`, deleteError)
      throw deleteError
    })
}

module.exports = deleteOrder
```

Export the handler.

Log the response or the error, and pass the value.

Seems easy, right?

Now you need to run the `claudia update` command from your pizza-api folder one more time to deploy your code. To test whether everything works, you can use the same `curl` commands you were using in chapter 2. Copy them from listings 3.14 and 3.15, and paste them in your terminal. Don't forget to update your `orderId` value. Using the one provided in those listings won't work because it's just a placeholder.

Listing 3.14 `curl` command for testing `PUT /orders/{orderId}` route

```
curl -i \
  -H "Content-Type: application/json" \
  -X PUT \
  -d '{"pizza": 3, "address": "221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_3/
     orders/some-id
```

Remember to replace some-id with the real ID of your order.

This command should return the following:

```
HTTP/1.1 200 OK

{
  "address": "221b Baker Street",
  "orderId": "some-id",
  "pizza": 3
  "status": "pending"
}
```

Listing 3.15 `curl` command for testing `DELETE /orders/{orderId}` route

```
curl -i \
  -H "Content-Type: application/json" \
  -X DELETE \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_3/
     orders/some-id
```

Remember to replace some-id with the real ID of your order.

This command should return

```
HTTP/1.1 200 OK

{}
```

Summary

- To build a useful serverless application, you'll often need to use external services—either for saving and retrieving data in a database, or to get needed information from another API.
- Communication to an external service is asynchronous.
- Claudia allows you to handle asynchronous functions by using JavaScript promises.
- JavaScript promises simplify the way you handle async operations. They also fix the problem often known as "callback hell" by allowing you to chain async operations, pass the values, and bubble the errors up.
- The simplest way to store data with AWS Lambda is to use DynamoDB, a NoSQL database offered as part of the AWS ecosystem.
- You can use DynamoDB in Node.js by installing the `aws-sdk` Node module. Among other things, the AWS SDK also exposes the DynamoDB `DocumentClient` class, which allows you to save, query, edit, and delete items in DynamoDB tables.
- DynamoDB tables are similar to collections in traditional NoSQL databases. Unfortunately, they only allow queries by primary key, which can be a combination of hash and range keys.

index