SAMPLE CHAPTER

AWS Lambda IN ACTION

Event-driven serverless applications

Danilo Poccia

FOREWORD BY James Governor

MANNING

AWS Lambda in Action

by Danilo Poccia

Chapter 1

brief contents

1 Running functions in the cloud

This chapter covers

- Understanding why functions can be the primitives of your application
- Getting an overview of AWS Lambda
- Using functions for the back end of your application
- Building event-driven applications with functions
- Calling functions from a client

In recent years, cloud computing has changed the way we think about and implement IT services, allowing companies of every size to build powerful and scalable applications that could disrupt the industries in which they operated. Think of how Dropbox changed the way we use digital storage and share files with each other, or how Spotify changed the way we buy and listen to music.

Those two companies started small, and needed the capacity to focus their time and resources on bringing their ideas to life quickly. In fact, one of the most important advantages of cloud computing is that it frees developers from spending their time on tasks that don't add real value to their work, such as managing and scaling

the infrastructure, patching the operating system (OS), or maintaining the software stack used to run their code. Cloud computing lets them concentrate on the unique and important features they want to build.

You can use cloud computing to provide the *infrastructure* for your application, in the form of virtual servers, storage, network, load balancers, and so on. The infrastructure can be scaled automatically using specific configurations. But with this approach you still need to prepare a whole environment to execute the code you write. You install and prepare an operating system or a virtual environment; you choose and configure a programming framework; and finally, when the overall stack is ready, you can plug in our code. Even if you use a container-based approach in building the environment, with tools such as Docker, you're still in change of managing versioning and updates of the containers you use.

Sometimes you need infrastructure-level access because you want to view or manage low-level resources. But you can also use cloud computing services that abstract from the underlying infrastructure implementation, acting like a *platform* on top of which you deploy your own customizations. For example, you can have services that provide you with a database, and you only need to plug in your data (together with a data model) without having to manage the installation and availability of the database itself. Another example is services where you provide the code of your application, and a standard infrastructure to support the execution of your application is automatically implemented.

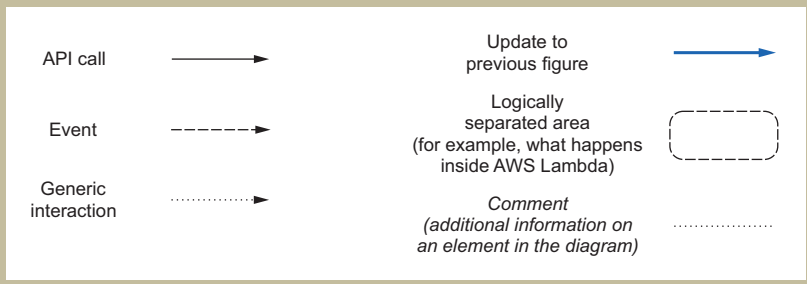
If that's true for a development environment, as soon as you get closer to production things become more complex and you may have to take care of the scalability and availability of the solution. And you must never forget to think about security—considering who can do what, and on which resources—during the course of the design and implementation of an application.

With the introduction of AWS Lambda, the abstraction layer is set higher, allowing developers to upload their code grouped in *functions*, and letting those functions be executed by the platform. In this way you don't have to manage the programming framework, the OS, or the availability and scalability. Each function has its own configuration that will help you use standard security features provided by Amazon Web Services (AWS) to define what a function can do and on which resources.

Those functions can be invoked directly or can *subscribe* to events generated by other *resources*. When you subscribe a function to a resource such as a file repository or a database, the function is automatically executed when something happens in that resource, depending on which kinds of events you've subscribed to. For example, when a file has been uploaded or a database item has been modified, an AWS Lambda function can react to those changes and do something with the new file or the updated data. If a picture has been uploaded, a function can create thumbnails to show the pictures on the screens with different resolutions. If a new record is written in an operational database, a function can keep the data warehouse in sync. In this way you can design applications that are driven by events.

Book graphical conventions

This book uses the following graphical conventions to help present information more clearly.



Using multiple functions together, some of them called directly from a user device, such as a smartphone, and other functions subscribed to multiple repositories, such as a file share and a database, you can build a complete event-driven application. You can see a sample flow of a media-sharing application built in this way in figure 1.1. Users use a mobile app to upload pictures and share them with their friends.

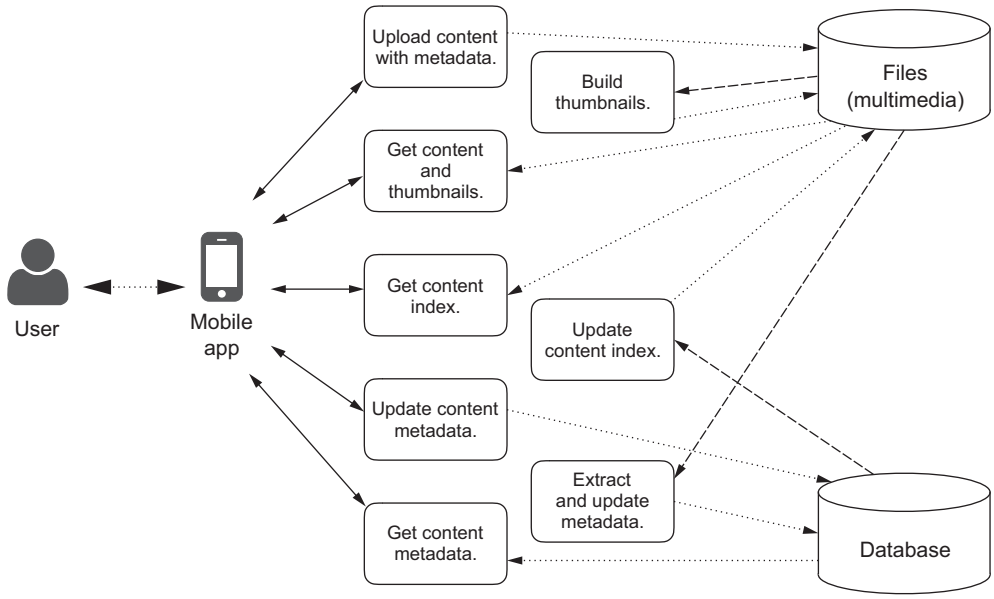


Figure 1.1 An event-driven, media-sharing application built using multiple AWS Lambda functions, some invoked directly by the mobile app. Other functions are subscribed to storage repositories such as a file share or a database.

> **NOTE** Don't worry if you don't completely understand the flow of the application in figure 1.1. Reading this book, you'll first learn the architectural principles used in the design of event-driven applications, and then you'll implement this media-sharing application using AWS Lambda together with an authentication service to recognize users.

When using third-party software or a service not natively integrated with AWS Lambda, it's still easy to use that component in an event-driven architecture, adding the capacity to generate those events by using one of the AWS software development kits (SDKs), which are available for multiple platforms.

The event-driven approach not only simplifies the development of production environments, but also makes it easier to design and scale the *logic* of the application. For example, let's take a function that's subscribed to the upload of a file in a repository. Every time this function is invoked, it extracts information from the content of the file and writes this in a database table. You can think of this function as a logical connection between the file repository and the database table: every time any component of the application—including the client—uploads a file, the subscribed events are triggered and, in this case, the database is updated.

As you add more features, the logic of any application becomes more and more complex to manage. But in this case you created a *relationship* between the file repository and the database, and this connection works independently from the process that uploads the file. You'll see more advantages of this approach in this book, along with more practical examples.

If you're building a new application for either a small startup or a large enterprise, the simplifications introduced by using functions as the building blocks of your application will allow you to be more efficient in where to spend your time and faster in introducing new features to your users.

1.1 Introducing AWS Lambda

AWS Lambda is different from a traditional approach based on physical or virtual servers. You only need to give your logic, grouped in functions, and the service itself takes care of executing the functions, if and when required, by managing the software stack used by the runtime you chose, the availability of the platform, and the scalability of the infrastructure to sustain the throughput of the invocations.

Functions are executed in *containers*. Containers are a server virtualization method where the kernel of the OS implements multiple isolated environments. With AWS Lambda, physical servers still execute the code, but because you don't need to spend time managing them, it's common to define this kind of approach as *serverless*.

> **TIP** For more details on the execution environment used by Lambda functions, please visit http://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html.

When you create a new function with AWS Lambda, you choose a *function name*, create your code, and specify the configuration of the execution environment that will be used to run the function, including the following:

- The maximum *memory size* that can be used by the function
- A *timeout* after which the function is terminated, even if it hasn't completed
- A *role* that describes what the function can do, and on which resources, using AWS Identity and Access Management (IAM)

TIP When you choose the amount of memory you want for your function, you're allocated proportional CPU power. For example, choosing 256 MB of memory allocates approximately twice as much CPU power to your Lambda function as requesting 128 MB of memory and half as much CPU power as choosing 512 MB of memory.

AWS Lambda implements the execution of those functions with an efficient use of the underlying compute resources that allows for an interesting and innovative cost model. With AWS Lambda you pay for

- The number of invocations
- The hundreds of milliseconds of execution time of all invocations, depending on the memory given to the functions

The execution time costs grow linearly with the memory: if you double the memory and keep the execution time the same, you double that part of the cost. To enable you to get hands-on experience, a free tier allows you to use AWS Lambda without any cost. Each month there's no charge for

- The first one million invocations
- The first 400,000 seconds of execution time with 1 GB of memory

If you use less memory, you have more compute time at no cost; for example, with 128 MB of memory (1 GB divided by 8) you can have up to 3.2 million seconds of execution time (400,000 seconds multiplied by 8) per month. To give you a scale of the monthly free tier, 400,000 seconds corresponds to slightly more than 111 hours or 4.6 days, whereas 3.2 million seconds comes close to 889 hours or 37 days.

TIP You'll need an AWS account to follow the examples in this book. If you create a new AWS account, all the examples that I provide fall in the *Free Tier* and you'll have no costs to sustain. Please look here for more information on the AWS Free Tier and how to create a new AWS account: http://aws.amazon.com/free/.

Throughout the book we'll use JavaScript (Node.js, actually) and Python in the examples, but other runtimes are available. For example, you can use Java and other

languages running on top of the Java Virtual Machine (JVM), such as Scala or Clojure. For object-oriented languages such as Java, the function you want to expose is a method of an object.

To use platforms that aren't supported by AWS Lambda, such as C or PHP, it's possible to use one of the supported runtimes as a *wrapper* and bring together with the function a static binary or anything that can be executed in the OS container used by the function. For example, a statically linked program written in C can be embedded in the archive used to upload a function.

When you call a function with AWS Lambda, you provide an event and a context in the input:

- The *event* is the way to send input parameters for your function and is expressed using JSON syntax.
- The *context* is used by the service to describe the execution environment and how the event is received and processed.

Functions can be called *synchronously* and return a *result* (figure 1.2). I use the term "synchronous" to indicate this kind of invocation in the book, but in other sources, such as the AWS Lambda API Reference documentation or the AWS command-line interface (CLI), this is described as the `RequestResponse` invocation type.

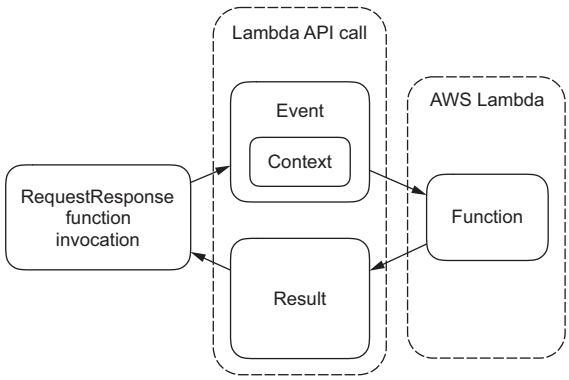


Figure 1.2 Calling an AWS Lambda function synchronously with the RequestResponse invocation type. Functions receive input as an event and a context and return a result.

For example, a simple synchronous function computing the sum of two numbers can be implemented in AWS Lambda using the JavaScript runtime as

```
exports.handler = (event, context, callback) => {
    var result = event.value1 + event.value2;
    callback(null, result);
};
```

The same can be done using the Python runtime:

```
def lambda_handler(event, context):
    result = event['value1'] + event['value2']
    return result
```

We'll dive deep into the syntax in the next chapter, but for now let's focus on what the functions are doing. Giving as input to those functions an event with the following JSON payload would give back a result of `30`:

```
{
  "value1": 10,
  "value2": 20
}
```

> **NOTE** The values in JSON are given as numbers, without quotation marks; otherwise the + used in both the Node.js and Python functions would change the meaning, becoming a concatenation of two strings.

Functions can also be called *asynchronously*. In this case the call returns immediately and no result is given back, while the function is continuing its work (figure 1.3). I use the term "asynchronous" to indicate this kind of invocation in the book, but in other sources, such as the AWS Lambda API Reference documentation and the AWS CLI, this is described as the `Event` invocation type.

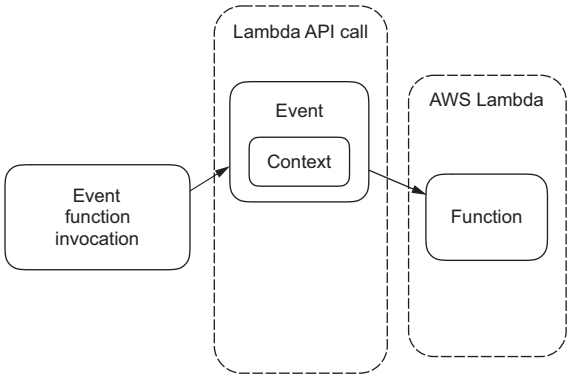


Figure 1.3 Calling an AWS Lambda function asynchronously with the Event invocation type. The invocation returns immediately while the function continues its work.

When a Lambda function terminates, no session information is retained by the AWS Lambda service. This kind of interaction with a server is usually defined as *stateless*. Considering this behavior, calling Lambda functions asynchronously (returning no value) is useful when they are accessing and modifying the status of other resources (such as files in a shared repository, records in a database, and so on) or calling other services (for example, to send an email or to send a push notification to a mobile device), as illustrated in figure 1.4.

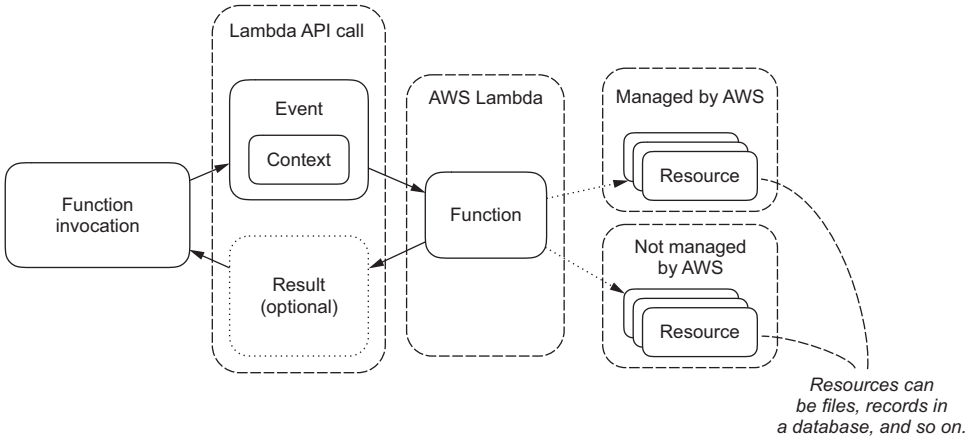


Figure 1.4 Functions can create, update, or delete other resources. Resources can also be other services that can do some actions, such as sending an email.

For example, it's possible to use the logging capabilities of AWS Lambda to implement a simple logging function (that you can call asynchronously) in Node.js:

```
exports.handler = function(event, context) {
    console.log(event.message);
    context.done();
};
```

In Python that's even easier because you can use a normal `print` to log the output:

```
def lambda_handler(event, context):
    print(event['message'])
    return
```

You can send input to the function as a JSON event to log a message:

```
{
  "message": "This message is being logged!"
}
```

In these two logging examples, we used the integration of AWS Lambda with Amazon CloudWatch Logs. Functions are executed without a default output device (in what is usually called a *headless environment*) and a default logging capability is given for each AWS Lambda runtime to ship the logs to CloudWatch. You can then use all the features provided by CloudWatch Logs, such as choosing the retention period or creating metrics from logged data. We'll give more examples and use cases regarding logging in part 4.

Asynchronous calls are useful when functions are *subscribed* to events generated by other resources, such as Amazon S3, an object store, or Amazon DynamoDB, a fully managed NoSQL database.

When you subscribe a function to events generated by other resources, the function is called asynchronously when the events you selected are generated, passing the events as input to the function (figure 1.5).

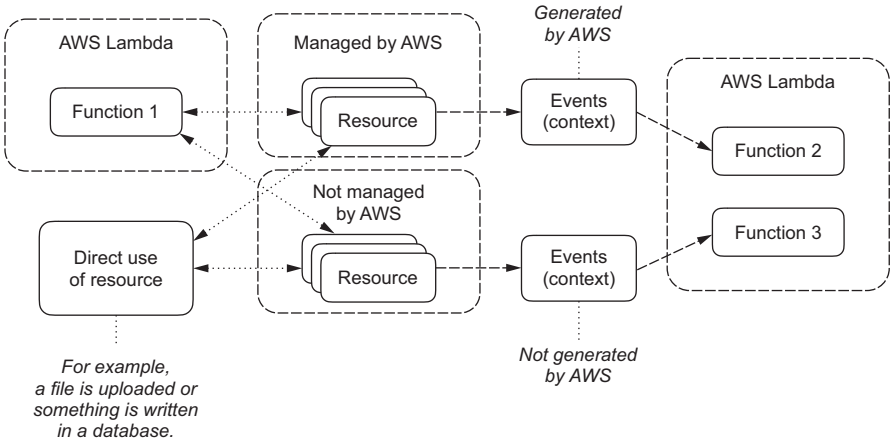


Figure 1.5 Functions can subscribe to events generated by direct use of resources, or by other functions interacting with resources. For resources not managed by AWS, you should find the best way to generate events to subscribe functions to those resources.

For example, if a user of a mobile application uploads a new high-resolution picture to a file store, a function can be triggered with the location of the new file in its input as part of the event. The function could then read the picture, build a small thumbnail to use in an index page, and write that back to the file store.

Now you know how AWS Lambda works at a high level, and that you can expose your code as functions and directly call those functions or subscribe them to events generated by other resources.

In the next section, you'll see how to use those functions in your applications.

1.2 Functions as your back end

Imagine you're a mobile developer and you're working on a new application. You can implement features in the mobile app running on the client device of the end user, but you'd probably keep part of the logic and status outside of the mobile app. For example:

- A mobile banking app wouldn't allow an end user to add money to their bank account without a good reason; only logic executed outside of the mobile device, involving the business systems of the bank, can decide if a transfer of money can be done or not.
- An online multiplayer game wouldn't allow a player to go to the next level without validating that the player has completed the current level.

This is a common pattern when developing client/server applications and the same applies to web applications. You need to keep part of the logic outside of the client (be it a web browser or a mobile device) for a few reasons:

- *Sharing*, because the information must be used (directly or indirectly) by multiple users of the application
- *Security*, because the data can be accessed or changed only if specific requirements are satisfied and the client cannot be trusted to check those requirements by itself
- *Access* to computing resources or storage capacity not available on a client device

We refer to this external logic required by a front end application as the *back end* of the application.

To implement this external logic, the normal approach is either to build a web application that can be called by the mobile app or to integrate it into an already existing web application rendering the content for a web browser. But instead of building and deploying a whole back end web application or extending the functionalities of your current back end, you can have your web page or your mobile application call one or more AWS Lambda functions that implement the logic you need. Those functions become your *serverless back end*.

Security is one of the reasons why you implement back end logic for an application, and you must always check the authentication and authorization of end users accessing your back end. AWS Lambda uses the standard security framework provided by AWS to control what a function can do, and on which resources. For example, a function can read from only a specific path of a file share, and write in a certain database table. This framework is based on AWS Identity and Access Management policies and roles. In this way, taking care of the security required to execute the code is simpler and becomes part of the development process itself. You can tailor security permissions specifically for each function, making it much easier to implement a least-privilege approach for each module (function, in this case) of your application.

DEFINITION By *least privilege*, I mean a security practice in which you always use the least privilege you need to perform an action in your application. For example, if you have a part of your application that's reading the user profiles from a central repository to publish them on a web page, you don't need to have write access to the repository in that specific module; you only need to read the subset of information you need to publish. Every other permission on top of that is in excess of what's required and can amplify the effects of a possible attack—for example, allowing malicious users that discover a security breach in your application to do more harm.

1.3 A single back end for everything

We can use AWS Lambda functions to expose the back end logic of our applications. But is that enough, or do we need something different to cover all the possible use cases for a back end application? Do we still need to develop traditional web applications, beyond the functions provided by AWS?

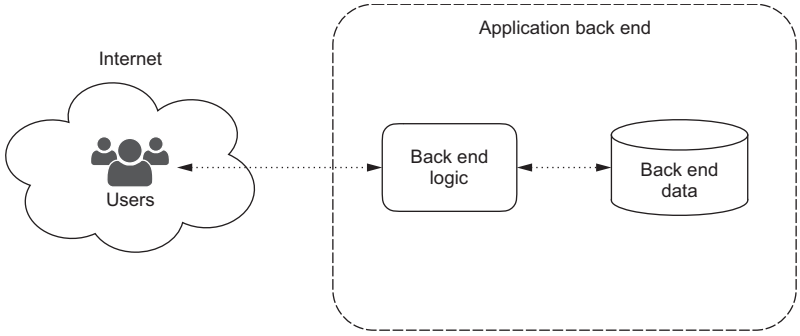


Figure 1.6 How users interact via the internet with the back end of an application. Note that the back end has some logic and some data.

Let's look at the overall flow and interactions of an application that can be used via a web browser or a mobile app (figure 1.6). Users interact with the back end via the internet. The back end has some logic and some data to manage.

The users of your application can use different devices, depending on what you decide to support. Supporting multiple ways to interact with your application, such as a web interface, a mobile app, and public application programming interfaces (APIs) that more advanced users can use to integrate third-party products with your application, is critical to success and is a common practice for new applications.

But if we look at the interfaces used by those different devices to communicate with the back end, we discover that they aren't always the same: a web browser expects more than the others, because both the content required by the user interface (dynamically generated HTML, CSS, JavaScript, multimedia files) and the application back end logic (exposed via APIs) are required (figure 1.7).

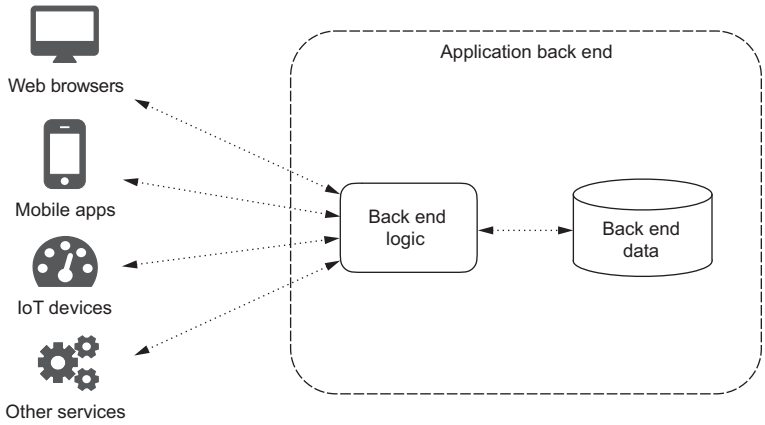


Figure 1.7 Different ways in which users can interact with the back end of an application. Users using a web browser receive different data than other front end clients.

If the mobile app of a specific service is developed after the web browser interface is already implemented, the back end application should be refactored to split API functionalities from web rendering—but that's usually not an easy task, depending on how the original application was developed. This sometimes causes developers to support two different back end platforms: one for web browsers serving web content and one for mobile apps, new devices (for example, wearable, home automation, and Internet of Things devices), and external services consuming their APIs. Even if the two back end platforms are well designed and share most of the functionalities (and hence the code), this wastes the developer's resources, because for each new feature they have to understand the impact on both platforms and run more tests to be sure those features are correctly implemented, while not adding value for their end users.

If we split the back end data between structured content that can go in one or more databases and unstructured content, such as files, we can simplify the overall architecture in a couple of steps:

1. Adding a (secure) web interface to the file repository so that it becomes a standalone resource that clients can directly access
2. Moving part of the logic into the web browser using a JavaScript client application and bringing it on par with the logic of the mobile app

Such a JavaScript client application, from an architectural point of view, behaves in the same way as a mobile app, in terms of functionality implemented, security, and (most importantly for our use case) the interactions with the back end (figure 1.8).

Looking at the back end logic, we now have a *single architecture for all clients* and the same interactions and data flows for all the consuming applications. We can abstract our back end from the actual implementation of the client and design it to serve a

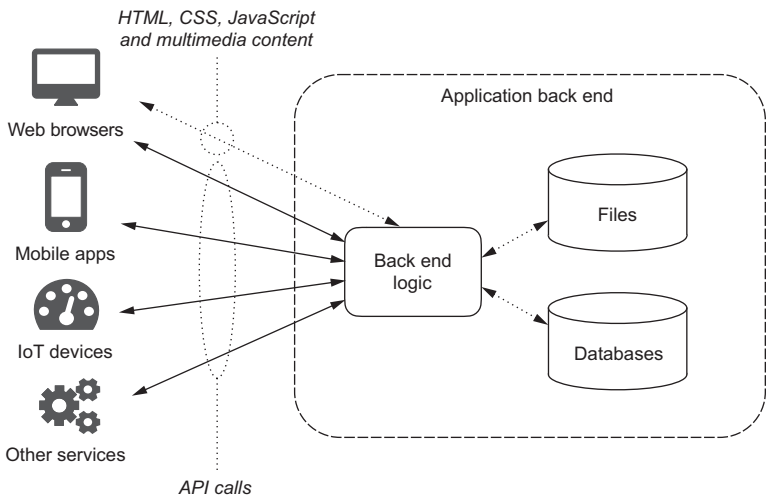


Figure 1.8 Using a JavaScript application running in the browser, back end architecture is simplified by serving only APIs to all clients.

generic *client application* using standard API calls that we define once and for all possible end users.

This is an important step because we've now *decoupled* the front end implementations, which could be different depending on the supported client devices, from the back end architecture (figure 1.9). Also, later you can add a new kind of client application (for example, an application running on wearable devices) without affecting the back end.

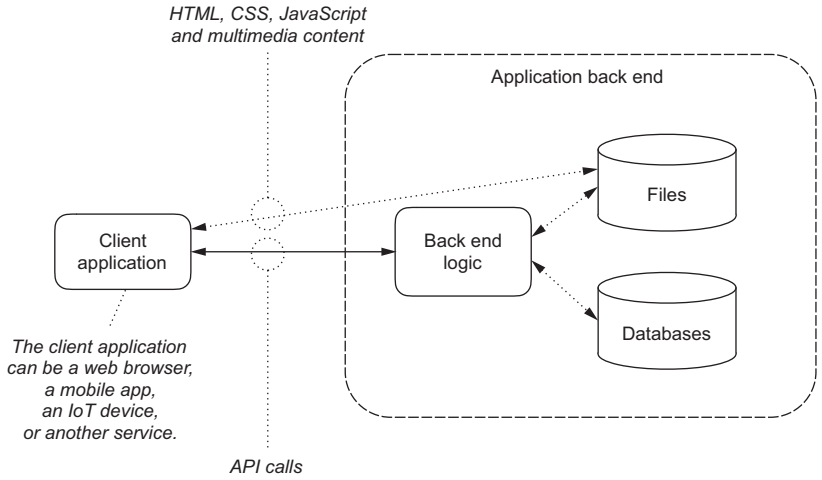


Figure 1.9 Think of your clients as a single client application consuming your APIs, which is possible when you decouple the implementation of the back end from the different user devices that interact with your application.

Looking again at the decoupled architecture, you can see that each of those API calls takes input parameters, does something in the back end, and returns a result. Does that remind you of something? Each API call is a *function exposed by the back end* that you can implement using AWS Lambda. Applying the same approach, all back end APIs can be implemented as functions managed by AWS Lambda.

In this way you have a *single serverless back end*, powered by AWS Lambda, that serves the same APIs to all clients of your application.

1.4 Event-driven applications

Up to now, we've used the functions provided by AWS Lambda directly, calling them as back end APIs from the client application. This is what's usually referred to as a *custom event* approach. But you could subscribe a function to receive events from another resource, for example if a file is uploaded to a repository or if a record in a database is updated.

Using subscriptions, you can change the internal behavior of the back end so that it can react not only to direct requests from client applications, but also to changes in the

resources that are used by the application. Instead of implementing a centralized workflow to support all the interactions among the resources, each interaction is described by the *relationship between the resources* involved. For example, if a file is added in a repository, a database table is updated with new information extracted from the file.

> **NOTE** This approach simplifies the design and the future evolution of the application, because we're inherently capitalizing on one of the advantages that microservices architectures bring: bottom-up *choreography* among software modules is much easier to manage than top-down *orchestration*.

With this approach, our back end becomes a distributed application, because it's not centrally managed and executed anymore, and we should apply best practices from distributed systems. For example, it's better to avoid synchronous transactions across multiple resources, which are difficult and slow to manage, and design each function to work independently (thanks to event subscriptions) with eventual consistency of data.

> **DEFINITION** By *eventual consistency*, I mean that we shouldn't expect the state of data to always be in sync across all resources used by the back end, but that the data will eventually converge over time to the last updated state.

Applications designed to react to internal and external events without a centralized workflow to coordinate processing on the resources are *event-driven applications*. Let's introduce this concept with a practical example.

Imagine you want to implement a media-sharing application, where the users can upload pictures from their client, a web browser or a mobile app, and share those pictures publicly with everyone or only with their friends.

To do that, you need two repositories:

- A file repository for the multimedia content (pictures)
- A database to handle user profiles (user table), friendships among the users (friendship table), and content metadata (content table).

You need to implement the following basic functionalities:

- Allow users to upload new multimedia content (pictures) with its own metadata. (By metadata, I mean: Is this content public or shared only among friends? Who uploaded the file? Where was the picture taken? At what time? Is there a caption?)
- Allow users to get specific content (pictures) shared by other users, but only if they have permission.
- Get an index of the content a specific user can see (all public content plus what has been shared with that user by their friends).
- Update content metadata. For example, a user can upload pictures only for their friends, and then change their mind and make a picture public for everyone to see.
- Get content metadata to be shown on the client together with the picture thumbnails; for example, adding the owner of the content, a date, a location, and a caption.

Of course, a real application needs more features (and more functions), but for the sake of simplicity we'll consider only the features listed here for now. You'll build a more complex (but still relatively simple) media-sharing application in chapter 8.

Because the content won't change too quickly, it's also effective to compute in advance (precompute) what each user can see in terms of content: end users will probably look at recent content often, and when they do, they want to see the result quickly. Using a precomputed *index* for the most recent content makes the rendering fast for users and makes the application use fewer computing resources in the back end. If users go back to older content outside the scope of the precomputed index, you can still compute that dynamically, but it happens less often and is easier to manage. The precomputed indexes must be updated each time the content (files or metadata) is updated and when the friendships between users change (because picture visibility is based on friendship).

You can see those features, and how they access repositories, implemented using one AWS Lambda function for each feature in figure 1.10.

In this way all interactions from the client application are covered, but you still miss basic back end functionalities here:

- What happens if a user uploads a new piece of content?
- What happens to the index if the user changes the metadata?
- You need to build thumbnails for the pictures to show them as a preview to end users.

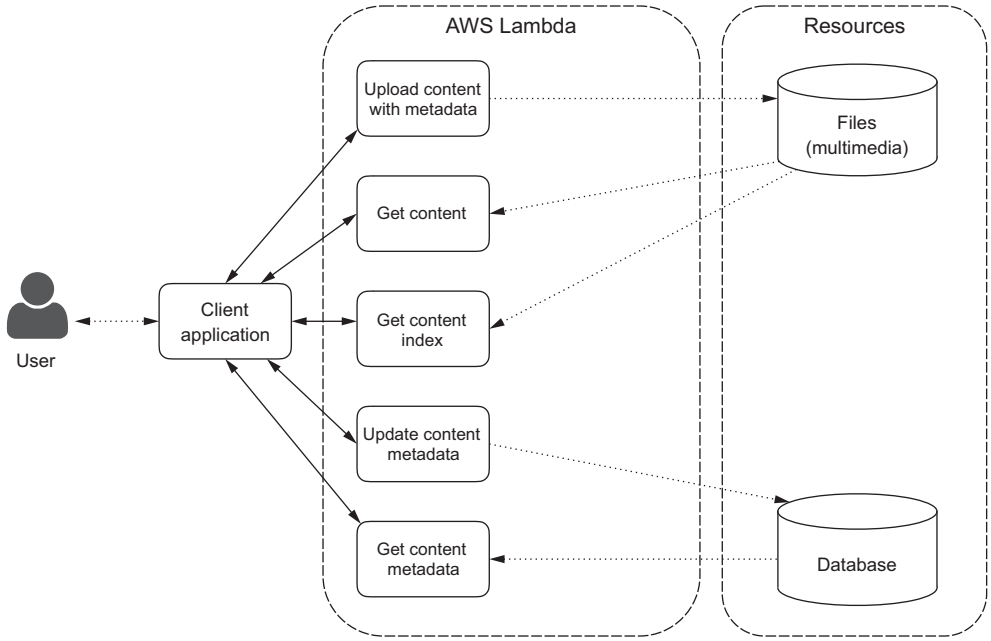


Figure 1.10 Features of a sample media-sharing application implemented as AWS Lambda functions, still missing basic back end functionalities

Those new back end features that you want to introduce are different from the previous ones, because they depend on what's happening in the back end repositories (files and database tables, in this case). You can implement those new features as additional functions that are subscribed to events coming from the repositories. For example:

- If a file (picture) is added or updated, you build the new thumbnail and add it back to the file repository.
- If a file (picture) is added or updated, you extract the new metadata and update the database (in the content table).
- Whenever the database is updated (user, friendship, or content table), you rebuild the dependent precomputed indexes, changing what a user can see.

Implementing those functionalities as AWS Lambda functions and subscribing those functions to the relevant events allows you to have an efficient architecture that drives updates when something relevant happens in the repositories, without enforcing a centralized workflow of activities that are required when data is changed by the end users. You can see a sample architecture implementing those new features as functions subscribed to events in figure 1.11.

Consider in our example the function subscribed to database events: that function is activated when the database is changed directly by end users (explicitly changing something in the metadata) or when an update is made by another function (because a new picture has been uploaded, bringing new metadata with it).

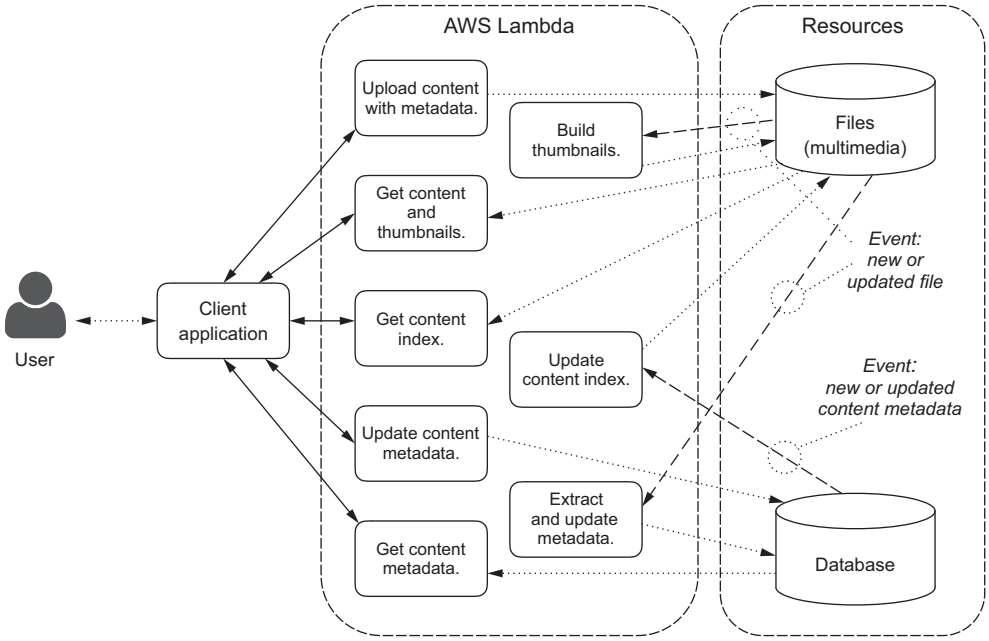


Figure 1.11 Sample media-sharing application with event-driven functions in the back end, subscribed to events from back end resources, such as file shares or databases

You don't need to manage the two use cases separately; they're both managed by the same subscription, a subscription that describes the relationship among the resources and the action you need to do when something changes.

You'll see when implementing this media-sharing application that some of the Lambda functions can be replaced by direct interactions to back end resources. For example, you can upload new or updated content (together with its own metadata) directly in a file share. Or update content metadata by directly writing to a database. The Lambda functions subscribed to those resources will implement the required back end logic.

This is a simplified but working example of a media-sharing application with an event-driven back end. Functions are automatically chained one after the other by the relationships we created by subscribing them to events. For example, if a picture is updated with new metadata (say, a new caption), a first function is invoked by the event generated in the file repository, updating the metadata in the database content table. This triggers a new event that invokes a second function to update the content index for all users who can see that content.

> **NOTE** In a way, the behavior I described is similar to a spreadsheet, where you update one cell and all the dependent cells (sums, average, more complex functions) are recomputed automatically. A spreadsheet is a good example of an event-driven application. This is a first step toward reactive programming, as you'll see later in the book.

Try to think of more features for our sample media-sharing application, such as creating, updating, and deleting a user; changing friendships (adding or removing a friend) and adding the required functions to the previous diagram to cover those aspects; subscribing (when it makes sense) the new functions to back end resources to have the flow of the application driven by events and avoid putting all the workflow logic in the functions themselves.

For example, suppose you have access to a mobile push notification service such as the Amazon Simple Notification Service (SNS). Think about the best way to use that in the back end to notify end users if new or updated content is available for them. What would you need to add, in terms of resources, events, and functions, to figure 1.11?

1.5 Calling functions from a client

In the previous discussion we didn't consider how, technically, the client application interacts with the AWS Lambda functions, assuming that a sort of direct invocation is possible.

As mentioned previously, each function can be invoked synchronously or asynchronously, and a specific AWS Lambda API exists to do that: the Invoke API (figure 1.12).

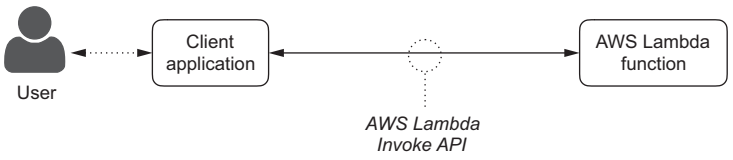


Figure 1.12 Calling AWS Lambda functions from a client application using the Invoke API

To call the Invoke API, AWS applies the standard security checks and requires that the client application has the right permissions to invoke the function. As per all other AWS APIs, you need AWS credentials to authenticate, and based on that authentication, AWS verifies whether those credentials have the right authorization to execute that API call (Invoke) on that specific resource (the function).

TIP We'll discuss the security model used by AWS Lambda in more detail in chapter 4. The most important thing to remember now is to *never put security credentials in a client application*, be that a mobile app or a JavaScript web application. If you put security credentials in something you deliver to end users, such as a mobile app or HTML or JavaScript code, an advanced user can find the credentials and compromise your application. In those cases, you need to use a different approach to authenticate a client application with the back end.

In the case of AWS Lambda, and all other AWS APIs, it's possible to use a specific service to manage authentication and authorization in an easy way: Amazon Cognito.

With Amazon Cognito, the client can authenticate using an external social or custom authentication (such as Facebook or Amazon) and get temporary AWS credentials to invoke the AWS Lambda functions the client is authorized to use (figure 1.13).

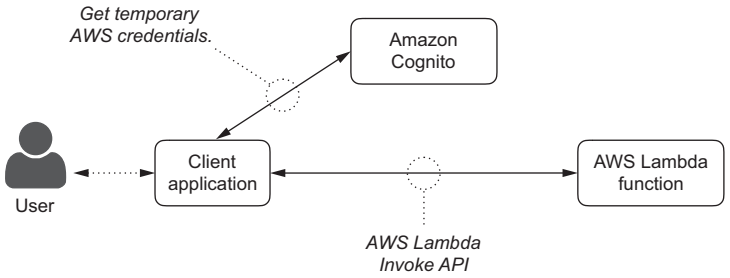


Figure 1.13 Using Amazon Cognito to authenticate and authorize invocation for AWS Lambda functions

NOTE Amazon Cognito provides a simplified interface to other AWS services, such as AWS Identity and Access Management (IAM) and AWS Security Token Service (STS). Figure 1.12 makes the flow easier to visualize, not including all details for the sake of simplicity.

Moving a step forward, it's possible to replace the direct use of the AWS Lambda Invoke API by clients with your own web APIs that you can build by mapping the access to AWS Lambda functions to more generic HTTP URLs and verbs.

For example, let's implement the web API for a bookstore. Users may need to list books, get more information for a specific book, and add, update, or delete a book. Using the Amazon API Gateway, you can map the access to a specific resource (the URL of the bookstore or a specific book) with an HTTP verb (`GET`, `POST`, `PUT`, `DELETE`, and so on) to the invocation of an AWS Lambda function. See table 1.1 for a sample configuration.

Table 1.1 A sample web API for a bookstore

Resource	+	HTTP verb	→	Method (function)
/books	+	GET	→	GetAllBooksByRange
/books	+	POST	→	CreateNewBook
/books/{id}	+	GET	→	GetBookById
/books/{id}	+	PUT	→	CreateOrUpdateBookById
/books/{id}	+	DELETE	→	DeleteBookById

Let's look at the example in table 1.1 in more detail:

- If you do an HTTP `GET` on the `/books` resource, you execute a Lambda function (`GetAllBooksByRange`) that will return a list of books, depending on a range you can optionally specify.
- If you do an HTTP `POST` on the same URL, you create a new book (using the `CreateNewBook` function) and get the `ID` of the book as the result.
- With an HTTP `GET` on `/books/ID`, you execute a function (`GetBookById`) that will give you a description (a representation, according to the REST architecture style) of the book with that specific `ID`.
- And so on for the other examples in the table.

NOTE You don't need to have a different Lambda function for every resource and HTTP verb (method) combination. You can send the resource and the method as part of the input parameters of a single function that can then process it to understand if it has been triggered by a `GET` or a `POST`. The choice between having more and smaller functions, or fewer and bigger ones, depends on your programming habits.

But the Amazon API Gateway adds more value than that, such as caching results to reduce load on the back end, throttling to avoid overloading the back end in peak moments, managing developer keys, generating the SDKs for the web API you design for multiple platforms, and other features that we'll start to see in chapter 2.

What's important is that by using the Amazon API Gateway we're *decoupling* the client from directly using AWS Lambda, exposing a clean web API that can be consumed by external services that should have no knowledge of AWS. However, even with the web API exposed by the Amazon API Gateway, we can optionally use AWS credentials (and hence Amazon Cognito) to manage authentication and authorization for the clients (figure 1.14).

With the Amazon API Gateway, we can also give public access to some of our web APIs. By *public access* I mean that no credentials are required to access those web APIs. Because one of the possible HTTP verbs that we can use in configuring an API is `GET`,

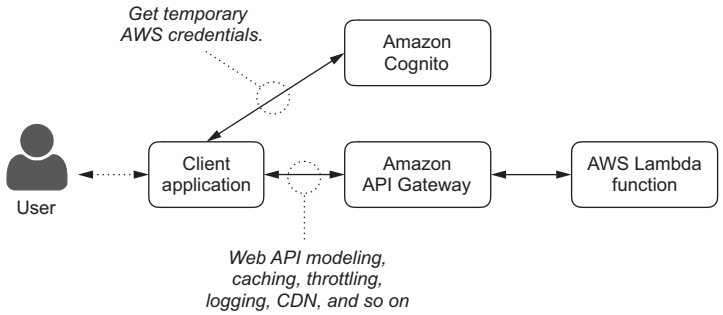


Figure 1.14 Using the Amazon API Gateway to access functions via web APIs

and `GET` is the default that is used when you type a URL in a web browser, we can use this configuration to create public websites whose URLs are dynamically served by AWS Lambda functions (figure 1.15).

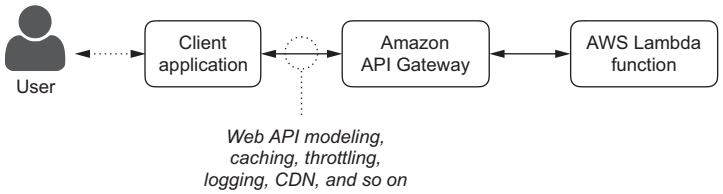


Figure 1.15 Using the Amazon API Gateway to give public access to an API and create public websites backed by AWS Lambda

In fact, the web API exposed publicly via the HTTP `GET` method can return any content type, including HTML content, such as a web page that can be seen in a browser.

> **TIP** For an example of a joint use of AWS Lambda and the Amazon API Gateway to build dynamic websites, see the Serverless framework at http://www.serverless.com/.

Summary

In this first chapter, I introduced the core topics that will be seen in depth in the rest of the book:

- An overview of AWS Lambda functions.
- Using functions to implement the back end of an application.
- Having a single back end for different clients, such as web browsers and mobile apps.
- An overview of how event-driven applications work.
- Managing authentication and authorization from a client.
- Using Lambda functions from a client, directly or via the Amazon API Gateway.

Now let's put all this theory into practice and build our first functions.

SOFTWARE DEVELOPMENT/CLOUD

AWS Lambda IN ACTION

Danilo Poccia

With AWS Lambda, you write your code and upload it to the AWS cloud. AWS Lambda responds to the events triggered by your application or your users, and automatically manages the underlying computer resources for you. Back-end tasks like analyzing a new document or processing requests from a mobile app are easy to implement. Your application is divided into small functions, leading naturally to a reactive architecture and the adoption of microservices.

AWS Lambda in Action is an example-driven tutorial that teaches you how to build applications that use an event-driven approach on the back-end. Starting with an overview of AWS Lambda, the book moves on to show you common examples and patterns that you can use to call Lambda functions from a web page or a mobile app. The second part of the book puts these smaller examples together to build larger applications. By the end, you'll be ready to create applications that take advantage of the high availability, security, performance, and scalability of AWS.

What's Inside

- Create a simple API
- Create an event-driven media-sharing application
- Secure access to your application in the cloud
- Use functions from different clients like web pages or mobile apps
- Connect your application with external services

Requires basic knowledge of JavaScript. Some examples are also provided in Python. No AWS experience is assumed.

Danilo Poccia is a technical evangelist at Amazon Web Services and a frequent speaker at public events and workshops.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/aws-lambda-in-action

"Clear and concise … the code samples are as well structured as the writing."
—From the Foreword by James Governor, RedMonk

"A superb guide to an important aspect of AWS."
—Ben Leibert, VillageReach

"Step-by-step examples and clear prose make this the go-to book for AWS Lambda!"
—Dan Kacenjar, Wolters Kluwer

"Like Lambda itself, this book is easy to follow, concise, and very functional."
—Christopher Haupt, New Relic

ISBN-13: 978-1-61729-371-9
ISBN-10: 1-61729-371-7
54999
9 781617 293719

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]