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 8

brief contents

8 Designing an authentication service

This chapter covers

- Designing a sample event-driven application
- Interacting with your users via JavaScript
- Sending emails from Lambda functions
- Storing data in Amazon DynamoDB
- Managing encrypted data

In the previous chapter you learned how to use standalone Lambda functions from different client applications:

- A web page, using JavaScript
- A native Mobile App, with the help of the AWS Mobile Hub to generate your starting code
- An Amazon API Gateway to generate server-side dynamic content for web browsers

Now it's time to build your first event-driven serverless application, using multiple functions together to achieve your purpose. Your goal is to implement a sample authentication service that can be used by itself or together with Amazon Cognito with developer-authenticated identities.

NOTE The authentication service you're going to build is an example of an event-driven serverless application and hasn't been validated by a security audit. If you need such a service, my advice is to use an already built and production-ready implementation, such as Amazon Cognito User Pools.

You'll define the architecture of your serverless back end built with AWS Lambda. In the chapter after this one, you'll implement all the required components. The first step is to define how your users interact with the application.

8.1 The interaction model

To make your application easy to use for a broad range of use cases, the main interface for your users is the web browser. Via a web browser, users can access static HTML pages that include JavaScript code, which can call one or more Lambda functions to execute code in the back end. At the end of the chapter, you'll see how it is easy to reuse the same flow and architecture with a mobile app.

Using the Amazon API Gateway

Another option, instead of calling Lambda functions directly from the client application, is to model a RESTful API with the Amazon API Gateway, using features similar to what you learned in chapter 3. The advantage of this approach is the decoupling of the client application from the actual back-end implementation:

- You call a Web API from the client application and not a Lambda function.
- You can easily change the back end implementation to (or from) AWS Lambda at any time, without affecting the development of the client application (for example, a web or mobile app).
- You can potentially open your back end to other services, publishing a public API that can further extend the reach of your application.

The Amazon API Gateway provides other interesting features, such as

- SDK generation
- Caching of function results
- Throttling to withstand traffic spikes

However, for the purpose of this book, I decided to use AWS Lambda directly in the authentication service. This makes the overall implementation simpler to build and more understandable for a first-time learner.

If you're building a new application, I advise you to evaluate the pros and cons of using the Amazon API Gateway as I did and make an informed decision.

The HTML pages, JavaScript code, and any other file required to render the page correctly on the web browser (such as CSS style sheets) can be stored on Amazon S3 as publicly readable objects. To store structured data, such as user profiles and passwords,

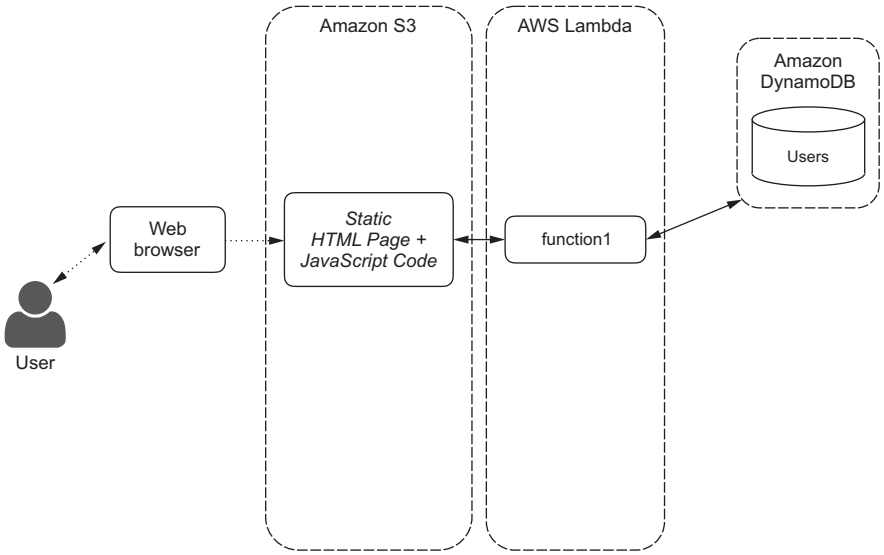

Figure 8.1 The first step in implementing the interaction model for your application: using a web browser to execute back end logic via Lambda functions that can store data on DynamoDB tables

Lambda functions can use DynamoDB tables. A summary of this interaction model is shown in figure 8.1.

> **TIP** Because the client side of the application is built using HTML pages and JavaScript code, it's relatively easy to repackage it as a hybrid mobile app, using frameworks such as Apache Cordova (formerly PhoneGap). Hybrid apps are popular because you can develop a mobile client once and use it in multiple environments, such as iOS, Android, and Windows Mobile. For more information on using Apache Cordova to implement mobile apps, please look at: https://cordova.apache.org.

It's important for an authentication service to verify contact data provided by users. A common use case is to verify that the email address given by a user is valid. To do that, the Lambda functions in the back end need to send emails to the users. To avoid the complexity of configuring and managing an email server, you can use Amazon Simple Email Services (SES) to send emails. This allows you to extend your interaction model adding this capability (figure 8.2).

> **NOTE** Amazon SES is a fully managed email service that you can use to send any volume of email, and receive emails that can be automatically stored on Amazon S3 or processed by AWS Lambda. When you receive an email with Amazon SES, you can also send a notification using Amazon Simple Notification Service (SNS). For more information on Amazon SES, see https://aws.amazon.com/ses/.

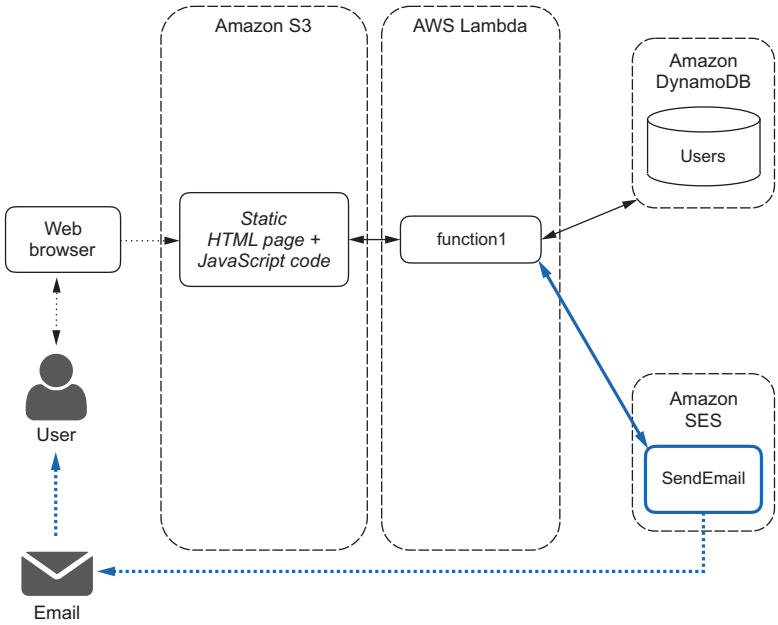

Figure 8.2 Adding the capability for Lambda functions to send emails to the users, via Amazon SES. In this way you can verify the validity of email addresses provided by users.

When a user receives an email sent by Amazon SES, you need a way of interacting with your back end to complete the verification process. To do that, you can include in the body of the email a link to the URL of another static HTML page on Amazon S3. When the user clicks the link, the web browser will open that page and execute the JavaScript code that's embedded in the page. The execution includes the invocation of another Lambda function that can interact with the data stored in Amazon DynamoDB (figure 8.3).

Now that you know how to interact with your users using a web browser or by sending emails, you can design the overall architecture of the authentication service.

8.2 The event-driven architecture

Every static HTML page you put on Amazon S3 can potentially be used as an *interactive step* to engage the user. If you compare this with a native mobile app, each of those pages can behave similarly to an *activity* in Android or a *scene* in iOS.

As the first step, you'll implement a menu of all possible actions users can perform (such as sign-up, login, or change password) and put that in an `index.html` page (figure 8.4). For now, this page doesn't require any client logic, so you have no JavaScript code to execute; it's a list of actions linking to other HTML pages.

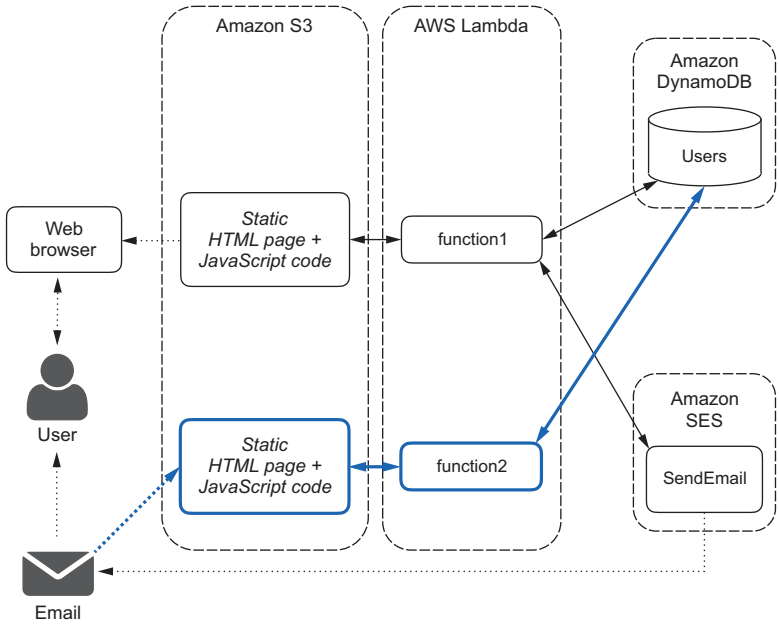

Figure 8.3 Emails received by users can include links to other HTML pages that can execute JavaScript code and invoke other Lambda function to interact with back-end data repositories such as DynamoDB tables.

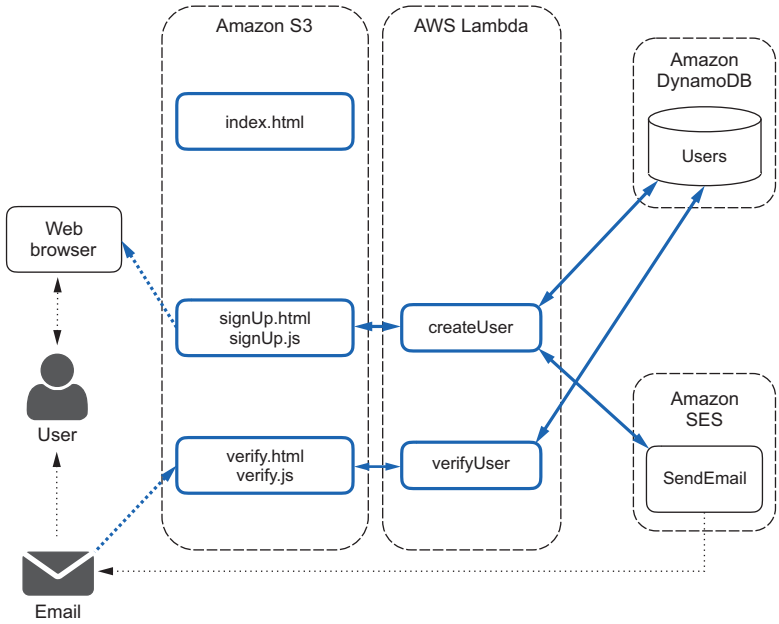

Figure 8.4 The first HTML pages, JavaScript files, and Lambda functions required to sign up new users and verify their email addresses

Next, you'll want users to sign up and create a new account using a `signUp.html` page. This page needs JavaScript code to invoke the `createUser` Lambda function (see figure 8.4).

> **TIP** To simplify separate management of the user interface (in the HTML page) and the client-side login (in the JavaScript code), put the JavaScript code in a separate file, with the same name as the HMTL page, but with the .js extension (for example, signUp.js in this case).

The `createUser` Lambda function takes as input all the information provided by a new user (such as the email and the password) and stores it in the `Users` DynamoDB table. A new user is flagged as unverified on the table because you don't know if the provided email address is correct. To verify that the email address given by the user is valid and that the user can receive emails at that address, the `createUser` function sends an email to the user (via Amazon SES).

The email sent to the user has a link to the `verify.html` page that includes a query parameter with a unique identifier (for example, a `token`) that's randomly generated for that specific user and stored in the `Users` DynamoDB table. For example, the link in the HTML page would be similar to the following:

```
http://some.domain/verify.html?token=<some unique identifier>
```

The JavaScript code in the `verify.html` page can read the unique identifier (`token`) from the URL and send it as input (as part of the event) to the `verifyUser` Lambda function. The function can check the validity of the `token` and change the status of the user to "verified" on the DynamoDB table.

A verified user can log in using the provided credentials (email, password). You can use a `login.html` page and a `login` Lambda function to check in the User table that the user is verified and the credentials are correct (figure 8.5). At first, this function can return the login status as a Boolean value (`true` or `false`). You'll learn later in this chapter how to federate the authentication service you're building with Amazon Cognito as a developer-authenticated identity.

Another important capability is for your users to change their passwords. Changing passwords periodically (for example, every few months) is a good practice to reduce the risk associated with compromised credentials.

You can add a `changePassword.html` page that can use a `changePassword` Lambda function to update credentials in the `Users` DynamoDB table (figure 8.6). But this page is different from others: only an authenticated user can change their own password.

There are two possible implementations that you can use for secure access to the `changePassword` function:

1. Add the current password to the input event of the function, to check the authentication of the user before changing the password.
2. Use Amazon Cognito, via the `login` function, to provide an authenticated status to the user.

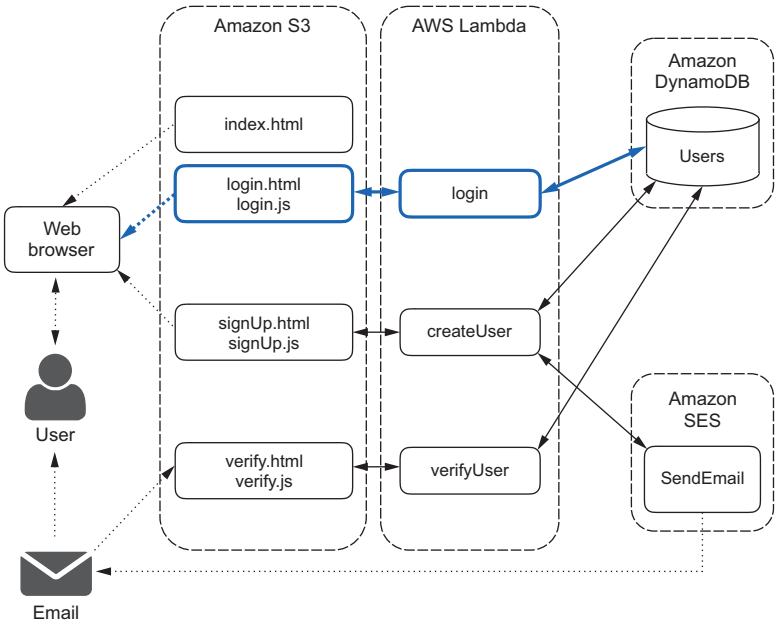

Figure 8.5 Adding a login page to test the provided credentials and the validity of the user in the Users repository

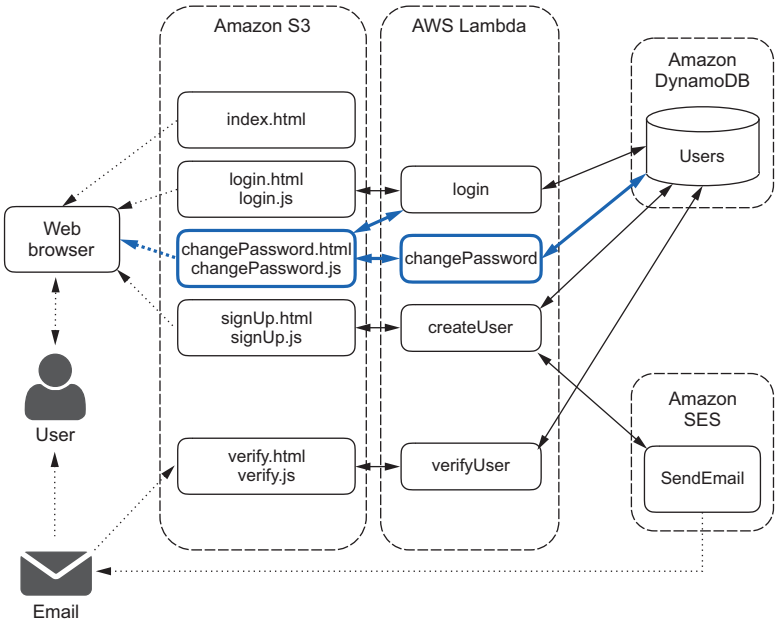

Figure 8.6 The page to allow users to change their passwords is calling a function that must be protected so that only authenticated users can use it.

The first solution is easy to implement (for example, reusing code from the `login` function), but because we're going to federate this authentication service with Amazon Cognito, let's make this example more interesting and go for the second option.

As you may recall, HTML pages need to get AWS credentials from Amazon Cognito to invoke Lambda functions. In all examples so far, we used only unauthenticated users; to allow those users to invoke a Lambda function, we added those functions to the unauthenticated IAM role associated with the Cognito identity pool.

To protect access to the `changePassword` function, you'll add this function to the authenticated IAM role (and not to the unauthenticated role). The same approach will work for any function that needs to be executed by only authenticated users.

Sometimes users need to change passwords because they forgot their current one. In those cases, you can use their email address to validate their request in a way similar to what you did for the initial sign-up: send an email with an embedded link and a unique identifier.

The `lostPassword.html` page is calling a `lostPassword` Lambda function to generate a unique identifier (`resetToken`) that's stored in the `Users` DynamoDB table. The `resetToken` is then sent to the user as a query parameter in a link embedded in a verification email (figure 8.7).

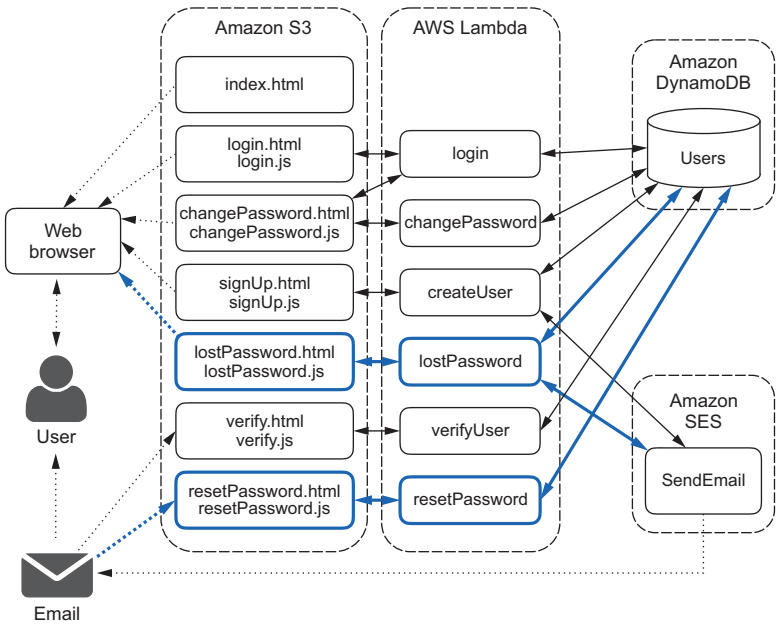

Figure 8.7 In case of a lost password, a lost password page is used to send an email with an embedded link to reset the password. A unique identifier, stored in the DynamoDB table and part of the reset password link, is used to verify that the user making the request to reset the password is the same user who receives the email.

For example, the link can be something similar to the following:

```
http://some.domain/resetPassword?resetToken=<some unique identifier>
```

The user can then open the email and click the link to the `resetPassword.html` page, which will ask for a new password and then call a `resetPassword` Lambda function to check the unique identifier (`resetToken`) in the `Users` DynamoDB table. If the identifier is correct, the function will change the password to the new value.

You've now designed the overall flow and the necessary components to cover the basic functionalities for implementing the authentication service. But before you move into the implementation phase in the next chapter, you'll learn how to *federate* the authentication with Amazon Cognito, and define how to implement other details. By *identity federation* I mean having an authorization service (Amazon Cognito in this case) trusting the authentication of an external service (the sample authentication service you are building).

> **NOTE** Instead of creating multiple Lambda functions, one for each HTML page, you could create a single Lambda function and pass the kind of action (for example `signUp` or `resetPassword`) as part of the input event. You'd have fewer functions to manage (potentially, only one) but the codebase of that function would be larger and more difficult to evolve and extend with further functionalities. Following a microservices approach, my advice is to have multiple smaller functions, each one with a well-defined input/output interface that you can update and deploy separately. However, the right balance between function size and the number of functions to implement depends on your actual use case and programming style. If you need to aggregate multiple functions into a single service call, the Amazon API Gateway is the place to do that instead of the functions themselves.

8.3 Working with Amazon Cognito

To use the authentication service with Amazon Cognito, you need to add to the `login` Lambda function a call to Amazon Cognito to get a token for a developer identity. The `login` function can then return the authentication token for a correct authentication.

The JavaScript code in the page can use that token to authenticate with Amazon Cognito and get AWS temporary credentials for the authenticated role (figure 8.8).

> **WARNING** The AWS credentials returned by Amazon Cognito are temporary and expire after a period of time. You need to manage credential rotation—for example, using the JavaScript `setInterval()` method to periodically call Amazon Cognito to refresh the credentials.

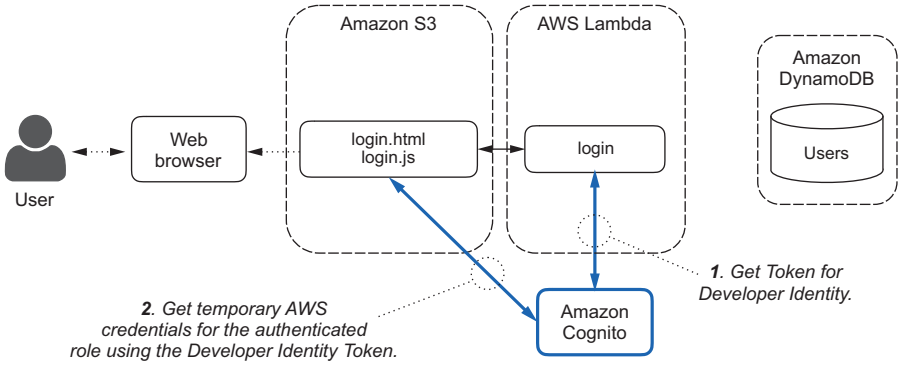

Figure 8.8 Integrating the login function with Cognito Developer Authenticated Identities. The login function gets the token from Amazon Cognito, and then the JavaScript code running in the browser can get AWS Credentials for the authenticated role by passing the token as a login.

8.4 Storing user profiles

To store user profiles, you're using the `Users` DynamoDB table in this sample application. Generally speaking, in a Lambda function you can use any repository reachable via the internet, or that's deployed on AWS in an Amazon Virtual Private Cloud (VPC), or deployed on-premises and connected to an Amazon VPC with a VPN connection. I'm using Amazon DynamoDB because it's a fully-managed NoSQL database service that embraces the serverless approach of this book.

In Amazon DynamoDB, when you create a new table, only the primary key must be declared and must be used in all items in the table. The rest of the table schema is flexible and other attributes can be used (or not) to add more information to any item.

> **NOTE** A DynamoDB item is a collection of attributes, and each attribute has a name and a value. For more details on how to work with items, see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html.

The primary key must be unique for an item and can be composed of a single *hash key* (for example, a user ID), or of a hash key together with a *range key* (such as a user ID and a validity date).

For this authentication service, the email of the user is a unique identifier that you can use as hash key, without a range key. If you want to have multiple items for the same users—for example, to keep track of changes and updates in the user profile—you could use a composed primary key with the email as hash key and a validity date in the sort key.

8.5 Adding more data to user profiles

Because Amazon DynamoDB doesn't enforce a schema outside of the primary key, you can freely add more attributes to any item in a table. Different items can have

different attributes. For example, to flag newly created users as unverified, you can add an `unverified` attribute equal to `true`.

When a user email is verified, instead of keeping the `unverified` attribute with a `false` value, you can remove it from the item using the assumption that if the `unverified` attribute isn't present, the user is verified. This approach (that can be easily used with Boolean values) provides a compact and efficient usage of the database storage, especially if you create an index on the `unverified` attribute, because only items with that attribute are part of the index.

Amazon DynamoDB also supports a JSON Document Model, so that the value of an attribute can be a JSON document. In this way, you can further extend the possibility of storing data in a hierarchical and structured way. For example, in the AWS JavaScript SDK, you can use the document client to have native JavaScript data types mapped to and from DynamoDB attributes.

For more information on the document client in the AWS JavaScript SDK, see http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html.

8.6 Encrypting passwords

When managing passwords, certain interactions are critical and must be secured. For example, the following are not secure:

- Storing passwords in plain text in a database table, because any user who has read access to the database table can intercept user credentials
- Sending passwords on an insecure channel, where malicious eavesdropping users can intercept user credentials

For this authentication service, you'll store the password as encrypted using a *salt*. In cryptography, a salt is random data that's generated for each password and used as an additional input to a one-way function that computes a hash of the password that's stored in the user profile, together with the salt:

```
hashingFunction(password, salt) = hash
```

To test the password in a login, the salt is read from the user profile and the same hashing function is used to compare the result with the stored hash. For example,

```
if hashingFunction(inputPassword, salt) == hash then // Logged in...
```

If user profiles are compromised and a malicious user has access to the database content, the use of a salt can protect against dictionary attacks, which use a list of common passwords versus a list of password hashes.

> **TIP** Common hashing functions, used in the past for salting passwords, were MD5 and SHA1, but they've been demonstrated to not be robust enough to protect against specific attacks. You have to check the robustness of a hashing function at the time you use it.

In the login phase, you send the password over a secure channel, because the AWS API, used by the `login.html` page to invoke the `login` Lambda function, is using HTTPS as transport.

> **TIP** This approach is secure enough for a sample implementation, but for a more robust solution you should never send the password as plain text. Use a challenge-response authentication, such as that implemented by the Secure Remote Password (SRP) protocol, used by Amazon Cognito User Pools. For more information on the SRP protocol, see http://srp.stanford.edu.

For a more in-depth analysis of password security in case of remote access, I suggest you to have a look at "Password Security: A Case History" by Robert Morris and Ken Thompson (1978), https://www.bell-labs.com/usr/dmr/www/passwd.ps.

Summary

In this chapter you designed the overall architecture of your first event-driven application, a sample authentication service using AWS lambda to implement the back-end logic.

In particular, you learned how to do the following:

- Interact with a client application via a static HTML page using JavaScript
- Differentiate between authenticated and unauthenticated access
- Send emails and interact using custom links in the email body
- Map application functionality to different components in the architecture
- Federate the custom authentication service with Amazon Cognito
- Use Amazon DynamoDB to store user profiles
- Use encryption to protect passwords from being intercepted and compromised

In the next chapter, you'll implement this sample authentication service.

EXERCISE

1. To send an email from a web page, you can
   a. Use JavaScript in the browser to use SMTP
   b. Use JavaScript in the browser to use IMAP
   c. Use a Lambda function to call Amazon SES
   d. Use a Lambda function to call Amazon SQS
2. To give access to a Lambda function only to authenticated users coming from a web or mobile app, you can
   a. Use AWS IAM users and groups to give access to the function to authenticated users only
   b. Use Amazon Cognito and give access to the function to the authenticated role only

c Use AWS IAM users and groups to give access to the function to unauthenticated users only

d Use Amazon Cognito and give access to the function to the unauthenticated role only

3 The most secure way to validate a user password with a login service is to

a Use a challenge-response interface such as CAPTCHA

b Send the password over HTTP

c Use a challenge-response protocol such as SRP

d Send the password via email

Solution

1 c

2 b

3 c

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]