Amazon Web Services IN ACTION

Andreas Wittig
Michael Wittig

FOREWORD BY Ben Whaley

SAMPLE CHAPTER

MANNING

Amazon Web Services in Action

by Michael Wittig
and Andreas Wittig

Chapter 13

brief contents

13 Designing for fault-tolerance

This chapter covers

- What fault-tolerance is and why you need it
- Using redundancy to remove single point of failures
- Retrying on failure
- Using idempotent operations to achieve retry on failure
- AWS service guarantees

Failure is inevitable for hard disks, networks, power, and so on. Fault-tolerance deals with that problem. A fault-tolerant system is built for failure. If a failure occurs, the system isn't interrupted, and it continues to handle requests. If your system has a single point of failure, it's not fault-tolerant. You can achieve fault-tolerance by introducing redundancy into your system and by decoupling the parts of your system in such a way that one side doesn't rely on the uptime of the other.

The most convenient way to make your system fault-tolerant is to compose the system of fault-tolerant blocks. If all blocks are fault-tolerant, the system is fault-tolerant

as well. Many AWS services are fault-tolerant by default. If possible, use them. Otherwise you'll need to deal with the consequences.

Unfortunately, one important service isn't fault-tolerant by default: EC2 instances. A virtual server isn't fault-tolerant. This means a system that uses EC2 isn't fault-tolerant by default. But AWS provides the building blocks to deal with that issue. The solution consists of auto-scaling groups, Elastic Load Balancing (ELB), and SQS.

It's important to differentiate among services that guarantee the following:

- *Nothing (single point of failure)*—No requests are served in case of failure.
- *High availability*—In case of failure, it takes some time until requests are served as before.
- *Fault-tolerance*—In case of failure, requests are served as before without any availability issues.

Following are the guarantees of the AWS services covered in this book in detail. *Single point of failure* (SPOF) means this service will fail if, for example, a hardware failure occurs:

- *Amazon Elastic Compute Cloud (EC2) instance*—A single EC2 instance can fail for many reasons: hardware failure, network problems, availability-zone problems, and so on. Use auto-scaling groups to have a fleet of EC2 instances serve requests in a redundant way to achieve high availability or fault-tolerance.
- *Amazon Relational Database Service (RDS) single instance*—A single RDS instance can fail for many reasons: hardware failure, network problems, availability zone problems, and so on. Use Multi-AZ mode to achieve high availability.

Highly available (HA) means that when a failure occurs the service won't be available for a short time but will come back automatically:

- *Elastic Network Interface (ENI)*—A network interface is bound to an AZ (availability zone), so if this AZ goes down, your network interface is down.
- *Amazon Virtual Private Cloud (VPC) subnet*—A VPC subnet is bound to an AZ, so if this AZ goes down, your subnet is down. Use multiple subnets in different AZs to remove the dependency on a single AZ.
- *Amazon Elastic Block Store (EBS) volume*—An EBS volume is bound to an AZ, so if this AZ goes down, your volume is unavailable (your data won't be lost). You can create EBS snapshots from time to time so you can recreate an EBS volume in another AZ.
- *Amazon Relational Database Service (RDS) Multi-AZ instance*—When running in Multi-AZ mode, a short downtime (one minute) is expected if an issue occurs with the master instance while changing DNS records to switch to the standby instance.

Fault-tolerant means that if a failure occurs, you won't notice it:

- Elastic Load Balancing (ELB), deployed to at least two AZs
- Amazon EC2 Security Group
- Amazon Virtual Private Cloud (VPC) with an ACL and a route table
- Elastic IP Address (EIP)
- Amazon Simple Storage Service (S3)

- Amazon Elastic Block Store (EBS) snapshot
- Amazon DynamoDB
- Amazon CloudWatch
- Auto-scaling group
- Amazon Simple Queue Service (SQS)
- AWS Elastic Beanstalk
- AWS OpsWorks
- AWS CloudFormation
- AWS Identity and Access Management (IAM, not bound to a single region; if you create an IAM user, that user is available in all regions)

Why should you care about fault-tolerance? Because in the end, a fault-tolerant system provides the highest quality to your end users. No matter what happens in your system, the user is never affected and can continue to consume content, buy stuff, or have conversations with friends. A few years ago it was expensive to achieve fault-tolerance, but in AWS, providing fault-tolerant systems is an affordable standard.

Chapter requirements

To fully understand this chapter, you need to have read and understood the following concepts:

- EC2 (chapter 3)
- Auto-scaling (chapter 11)
- Elastic Load Balancing (chapter 12)
- SQS (chapter 12)

The example makes intensive use of the following:

- Elastic Beanstalk (chapter 5)
- DynamoDB (chapter 10)
- Express, a Node.js web application framework

In this chapter, you'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault-tolerant by default).

13.1 Using redundant EC2 instances to increase availability

Unfortunately, EC2 instances aren't fault-tolerant. Under your virtual server is a host system. These are a few reasons your virtual server might suffer from a crash caused by the host system:

- If the host hardware fails, it can no longer host the virtual server on top of it.
- If the network connection to/from the host is interrupted, the virtual server loses the ability to communicate via network as well.
- If the host system is disconnected from a power supply, the virtual server also goes down.

But the software running on top of the virtual server may also cause a crash:

- If your software has a memory leak, you'll run out of memory. It may take a day, a month, a year, or more, but eventually it will happen.
- If your software writes to disk and never deletes its data, you'll run out of disk space sooner or later.
- Your application may not handle edge cases properly and instead just crashes.

Regardless of whether the host system or your software is the cause of a crash, a single EC2 instance is a single point of failure. If you rely on a single EC2 instance, your system will blow up—the only question is when.

13.1.1 Redundancy can remove a single point of failure

Imagine a production line that makes fluffy cloud pies. Producing a fluffy cloud pie requires several production steps (simplified!):

1. Produce a pie crust.
2. Cool the pie crust.
3. Put the fluffy cloud mass on top of the pie crust.
4. Cool the fluffy cloud pie.
5. Package the fluffy cloud pie.

The current setup is a single production line. The big problem with this setup is that whenever one of the steps crashes, the entire production line must be stopped. Figure 13.1 illustrates the problem when the second step (cooling the pie crust) crashes. The following steps no longer work, either, because they don't receive cool pie crusts.

Why not have multiple production lines? Instead of one line, suppose we have three. If one of the lines fails, the other two can still produce fluffy cloud pies for all the hungry customers in the world. Figure 13.2 shows the improvements; the only downside is that we need three times as many machines.

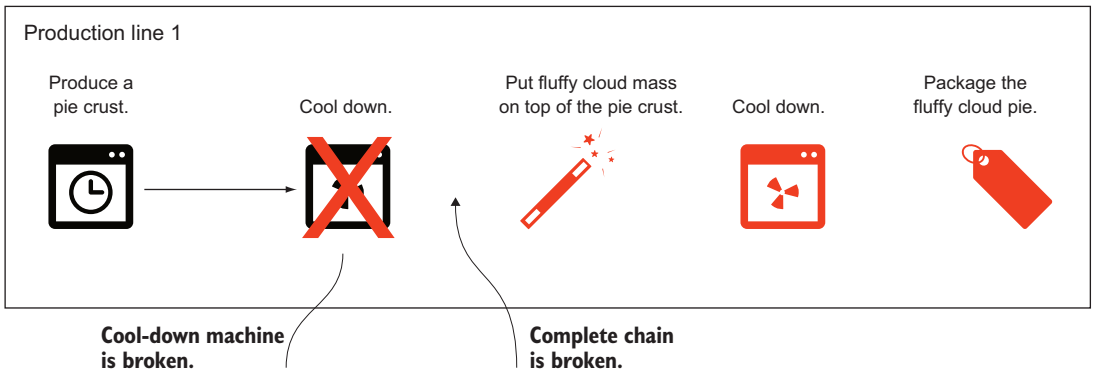

Figure 13.1 A single point of failure affects not only itself, but the entire system.

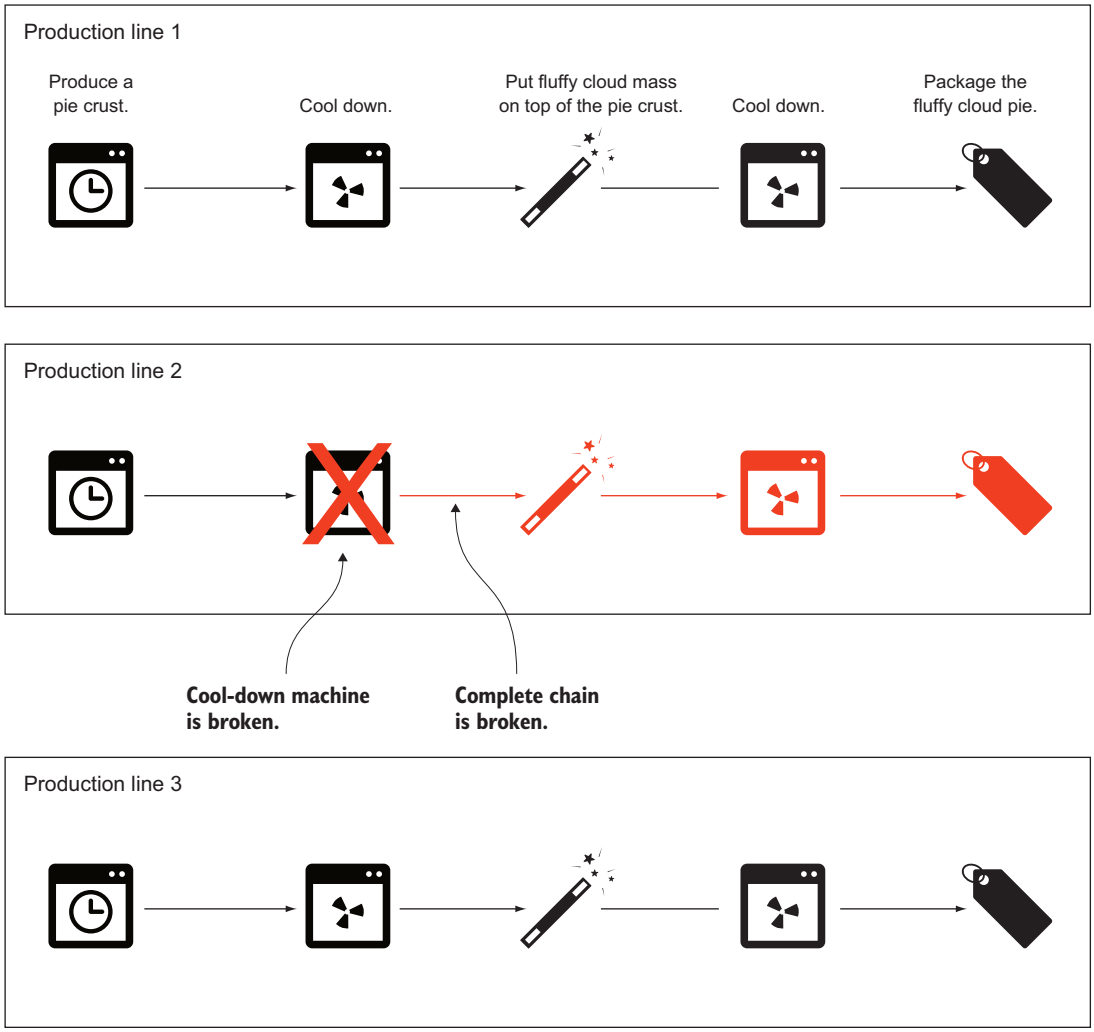

Figure 13.2 **Redundancy eliminates single points of failure and makes the system more stable.**

The example can be transferred to EC2 instances as well. Instead of having only one EC2 instance, you can have three of them running your software. If one of those instances crashes, the other two are still able to serve incoming requests. You can also minimize the cost impact of one versus three instances: instead of one large EC2 instance, you can choose three small ones. The problem that arises with a dynamic server pool is, how can you communicate with the instances? The answer is *decoupling*: put a load balancer between your EC2 instances and the requestor or a message queue. Read on to learn how this works.

13.1.2 Redundancy requires decoupling

Figure 13.3 shows how EC2 instances can be made fault-tolerant by using redundancy and synchronous decoupling. If one of the EC2 instances crashes, ELB stops to route requests to the crashed instances. The auto-scaling group replaces the crashed EC2 instance within minutes, and ELB begins to route requests to the new instance.

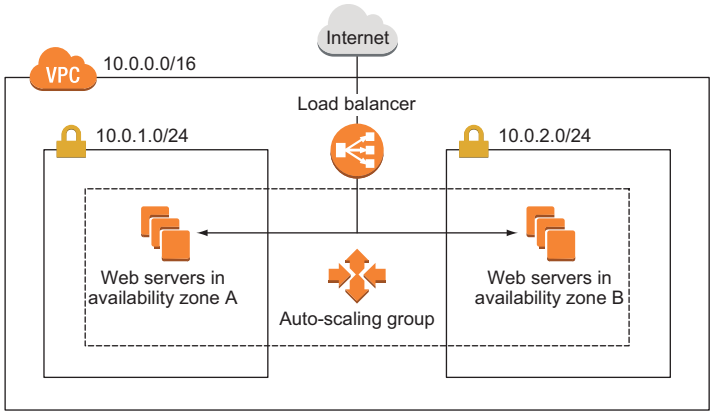

Figure 13.3 Fault-tolerant EC2 servers with an auto-scaling group and ELB

Take a second look at figure 13.3 and see what parts are redundant:

- *Availability zones*—Two are used. If one AZ goes down, we still have EC2 instances running in the other AZ.
- *Subnets*—A subnet is tightly coupled to an AZ. Therefore we need one subnet in each AZ, and subnets are also redundant.
- *EC2 instances*—We have multi-redundancy for EC2 instances. We have multiple instances in a single subnet (AZ), and we have instances in two subnets (AZs).

Figure 13.4 shows a fault-tolerant system built with EC2 that uses the power of redundancy and asynchronous decoupling to process messages from an SQS queue.

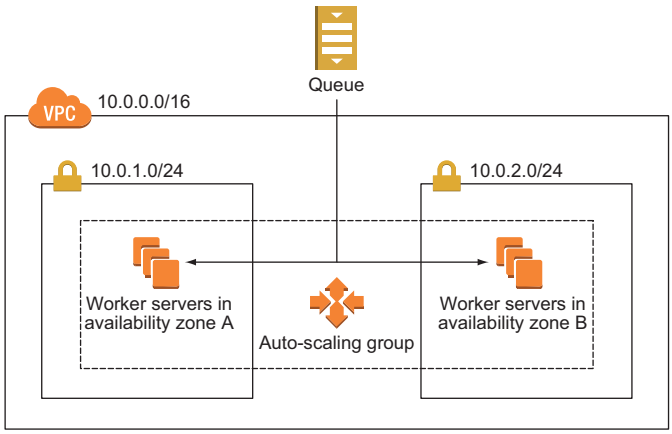

Figure 13.4 Fault-tolerant EC2 servers with an auto-scaling group and SQS

In both figures, the load balancer/SQS queue appears only once. This doesn't mean ELB or SQS is a single point of failure; on the contrary, ELB and SQS are fault-tolerant by default.

13.2 Considerations for making your code fault-tolerant

If you want fault-tolerance, you must achieve it within your code. You can design fault-tolerance into your code by following two suggestions presented in this section.

13.2.1 Let it crash, but also retry

The Erlang programming language is famous for the concept of "let it crash." That simply means whenever the program doesn't know what to do, it crashes, and someone needs to deal with the crash. Most often people overlook the fact that Erlang is also famous for retrying. Letting it crash without retrying isn't useful—if you can't recover from a crashed situation, your system will be down, which is the opposite of what you want.

You can apply the "let it crash" concept (some people call it "fail-fast") to synchronous and asynchronous decoupled scenarios. In a synchronous decoupled scenario, the sender of a request must implement the retry logic. If no response is returned within a certain amount of time, or an error is returned, the sender retries by sending the same request again. In an asynchronous decoupled scenario, things are easier. If a message is consumed but not acknowledged within a certain amount of time, it goes back to the queue. The next consumer then grabs the message and processes it again. Retrying is built into asynchronous systems by default.

"Let it crash" isn't useful in all situations. If the program wants to respond to tell the sender that the request contained invalid content, this isn't a reason for letting the server crash: the result will stay the same no matter how often you retry. But if the server can't reach the database, it makes a lot of sense to retry. Within a few seconds the database may be available again and able to successfully process the retried request.

Retrying isn't that easy. Imagine that you want to retry the creation of a blog post. With every retry, a new entry in the database is created, containing the same data as before. You end up with many duplicates in the database. Preventing this involves a powerful concept that's introduced next: idempotent retry.

13.2.2 Idempotent retry makes fault-tolerance possible

How can you prevent a blog post from being added to the database multiple times because of a retry? A naïve approach would be to use the title as primary key. If the primary key is already used, you can assume that the post is already in the database and skip the step of inserting it into the database. Now the insertion of blog posts is *idempotent*, which means no matter how often a certain action is applied, the outcome must be the same. In the current example, the outcome is a database entry.

Let's try it with a more complicated example. Inserting a blog post is more complicated in reality, and the process looks something like this:

1. Create a blog post entry in the database.
2. Invalidate the cache because data has changed.
3. Post the link to the blog's Twitter feed.

Let's take a close look at each step.

1. CREATING A BLOG POST ENTRY IN THE DATABASE

We covered this step earlier by using the title as a primary key. But this time, let's use a universally unique identifier (UUID) instead of the title as the primary key. A UUID like `550e8400-e29b-11d4-a716-446655440000` is a random ID that's generated by the client. Because of the nature of a UUID, it's unlikely that two equal UUIDs will be generated. If the client wants to create a blog post, it must send a request to the ELB containing the UUID, title, and text. The ELB routes the request to one of the back-end servers. The back-end server checks whether the primary key already exists. If not, a new record is added to the database. If it exists, the insertion continues. Figure 13.5 shows the flow.

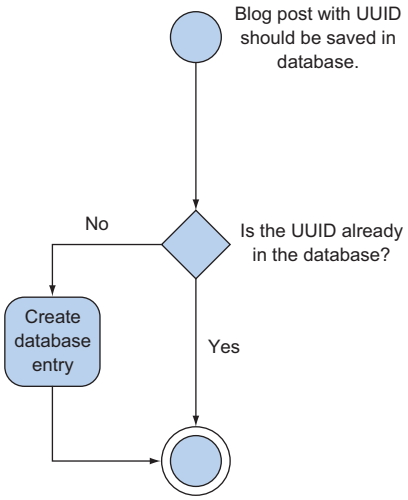

Figure 13.5 Idempotent database insert: creating a blog post entry in the database only if it doesn't already exist

Creating a blog post is a good example of an idempotent operation that's guaranteed by code. You can also use your database to handle this problem. Just send an insert to your database. Three things can happen:

- Your database inserts the data. The step is successfully completed.
- Your database responds with an error that the primary key is already in use. The step is successfully completed.
- Your database responds with a different error. The step crashes.

Think twice about the best way of implementing idempotence!

2. INVALIDATING THE CACHE

This step sends an invalidation message to a caching layer. You don't need to worry about idempotency too much here: it doesn't hurt if the cache is invalidated more often than needed. If the cache is invalidated, then the next time a request hits the cache, the cache won't contain data, and the original source (in this case, the database) will be queried for the result. The result is then put in the cache for subsequent requests. If you invalidate the cache multiple times because of a retry, the worst thing that can happen is that you may need to make a few more calls to your database. That's easy.

3. POSTING TO THE BLOG'S TWITTER FEED

To make this step idempotent, you need to use some tricks because you interact with a third party that doesn't support idempotent operations. Unfortunately, no solution will guarantee that you post exactly one status update to Twitter. You can guarantee the creation is at least one (one or more than one) status update, or at most one (one or none) status update. An easy approach could be to ask the Twitter API for the latest status updates; if one of them matches the status update that you want to post, you skip the step because it's already done.

But Twitter is an eventually consistent system: there's no guarantee that you'll see a status update immediately after you post it. You can end up having your status update posted multiple times. Another approach would be to save in a database whether you already posted the blog post status update. But imagine saving to the database that you posted to Twitter and then making the request to the Twitter API—but at that moment, the system crashes. Your database will say that the Twitter status update was posted, but in reality it wasn't. You need to make a choice: tolerate a missing status update, or tolerate multiple status updates. Hint: it's a business decision. Figure 13.6 shows the flow of both solutions.

Now it's time for a practical example! You'll design, implement, and deploy a distributed, fault-tolerant web application on AWS. This example will demonstrate how distributed systems work and will combine most of the knowledge in this book.

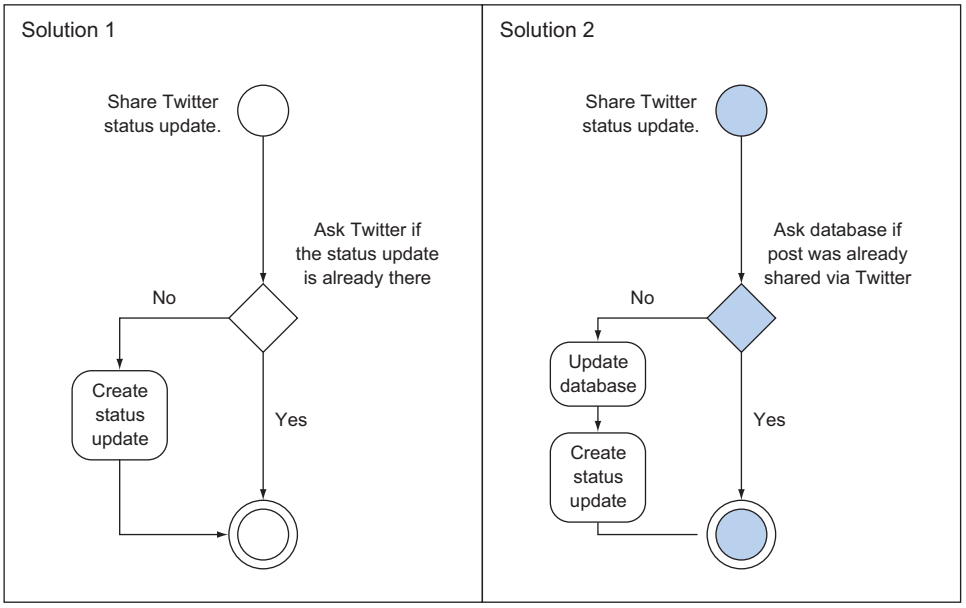

Figure 13.6 Idempotent Twitter status update: only share a status update if it hasn't already been done.

13.3 Architecting a fault-tolerant web application: Imagery

Before you begin the architecture and design of the fault-tolerant Imagery application, we'll talk briefly about what the application should do in the end. A user should be able to upload an image. This image is then transformed with a sepia filter so that it looks old. The user can then view the sepia image. Figure 13.7 shows the process.

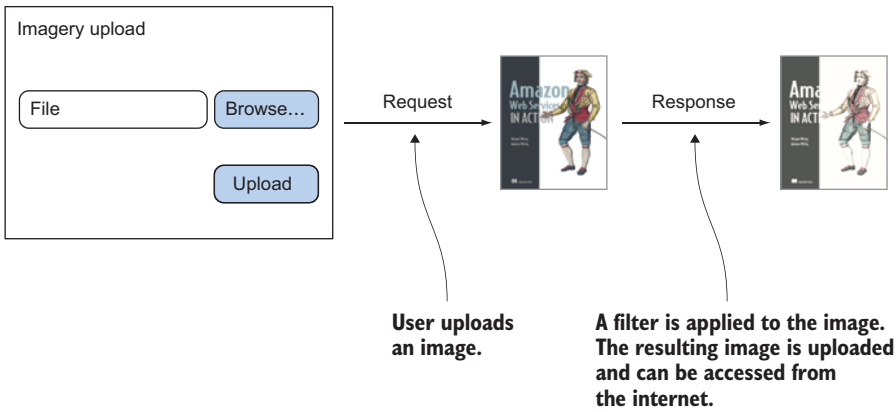

Figure 13.7 The user uploads an image to Imagery, where a filter is applied.

The problem with the process shown in figure 13.7 is that it's synchronous. If the server dies during request and response, the user's image won't be processed. Another problem arises when many users want to use the Imagery app: the system becomes busy and may slow down or stop working. Therefore the process should be turned into an asynchronous one. Chapter 12 introduced the idea of asynchronous decoupling by using a SQS message queue, as shown in figure 13.8.

When designing an asynchronous process, it's important to keep track of the process. You need some kind of identifier for it. When a user wants to upload an image, the user creates a process first. This process creation returns a unique ID. With that ID, the user is able to upload an image. If the image upload is finished, the server begins to process the image in the background. The user can look up the process at any time

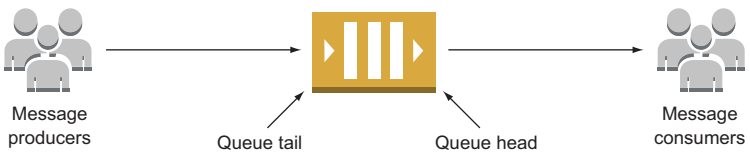

Figure 13.8 Producers send messages to a message queue, and consumers read messages.

with the process ID. While the image is being processed, the user can't see the sepia image. But as soon as the image is processed, the lookup process returns the sepia image. Figure 13.9 shows the asynchronous process.

Now that you have an asynchronous process, it's time to map that process to AWS services. Keep in mind that most services on AWS are fault-tolerant by default, so it makes sense to pick them whenever possible. Figure 13.10 shows one way of doing it.

To make things as easy as possible, all the actions will be accessible via a REST API, which will be provided by EC2 instances. In the end, EC2 instances will provide the process and make calls to all the AWS services shown in figure 13.10.

You'll use many AWS services to implement the Imagery application. Most of them are fault-tolerant by default, but EC2 isn't. You'll deal with that problem using an idempotent image-state machine, as introduced in the next section.

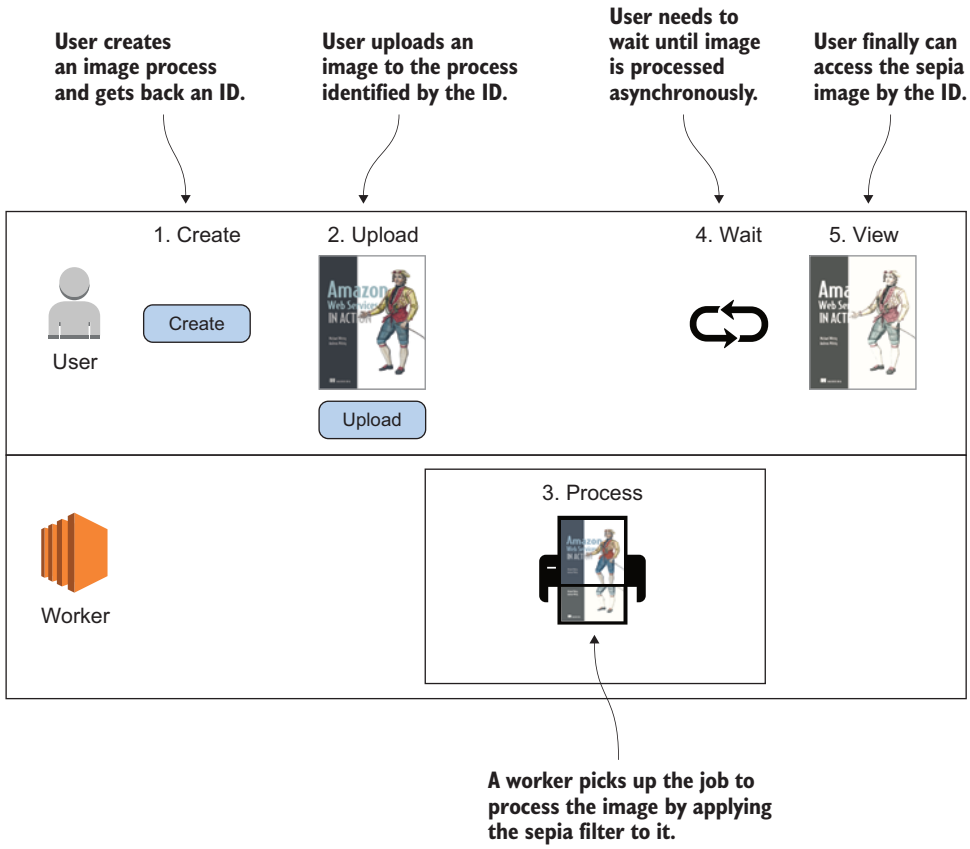

Figure 13.9 **The user asynchronously uploads an image to Imagery, where a filter is applied.**

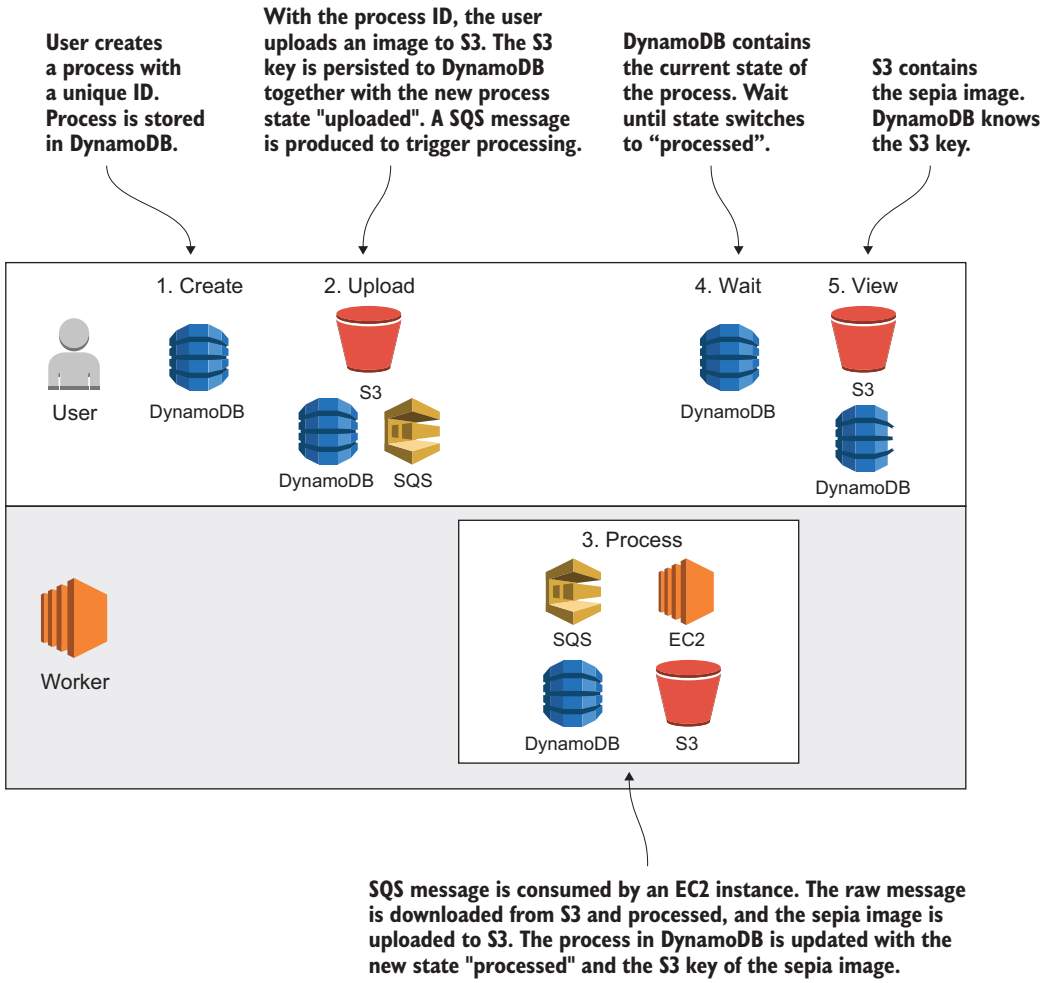

Figure 13.10 **Combining AWS services to implement the asynchronous Imagery process**

Example is 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there are no other things going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

AWS Lambda and Amazon API Gateway are coming

AWS is working on a service called Lambda. With Lambda, you can upload a code function to AWS and then execute that function on AWS. You no longer need to provide your own EC2 instances; you only have to worry about the code. AWS Lambda is made for short-running processes (up to 60 seconds), so you can't create a web server with Lambda. But AWS will offer many integration hooks: for example, each time an object is added to S3, AWS can trigger a Lambda function; or a Lambda function is triggered when a new message arrives on SQS. Unfortunately, AWS Lambda isn't available in all regions at the time of writing, so we decided not to include this service.

Amazon API Gateway gives you the ability to run a REST API without having to run any EC2 instances. You can specify that whenever a `GET /some/resource` request is received, it will trigger a Lambda function. The combination of Lambda and Amazon API Gateway lets you build powerful services without a single EC2 instance that you must maintain. Unfortunately, Amazon API Gateway isn't available in all regions at the time of writing.

13.3.1 The idempotent image-state machine

An idempotent image-state machine sounds complicated. We'll take some time to explain it because it's the heart of the Imagery application. Let's look at what a *state machine* is and what *idempotent* means in this context.

THE FINITE STATE MACHINE

A state machine has at least one start state and one end state (we're talking about finite state machines). Between the start and the end state, the state machine can have many other states. The machine also defines transitions between states. For example, a state machine with three states could look like this:

```
(A) -> (B) -> (C).
```

This means

- State A is the start state.
- There is a transition possible from state A to B.
- There is a transition possible from state B to C.
- State C is the end state.

But there's no transition possible between (A) -> (C) or (B) -> (A). The Imagery state machine could look like this:

```
(Created) -> (Uploaded) -> (Processed)
```

Once a new process (state machine) is created, the only transition possible is to `Uploaded`. To make this transition happen, you need the S3 key of the uploaded raw image. The transition between `Created` -> `Uploaded` can be defined by the function `uploaded(s3Key)`. Basically, the same is true for the transition `Uploaded` -> `Processed`. This transition can be done with the S3 key of the sepia image: `processed(s3Key)`.

Don't be confused because the upload and the image filter processing don't appear in the state machine. These are the basic actions that happen, but we're only interested in the results; we don't track the progress of the actions. The process isn't aware that 10% of the data has been uploaded or that 30% of the image processing is done. It only cares whether the actions are 100% done. You can probably imagine a bunch of other states that could be implemented but that we're skipping for the purpose of simplicity in this example; `Resized` and `Shared` are just two examples.

IDEMPOTENT STATE TRANSITIONS

An idempotent state transition must have the same result no matter how often the transition takes place. If you know that your state transitions are idempotent, you can do a simple trick: in case of a failure during transitioning, you retry the entire state transition.

Let's look at the two state transitions you need to implement. The first transition `Created` -> `Uploaded` can be implemented like this (pseudo code):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== "Created") {
    throw new Error("transition not allowed")
  }
  DynamoDB.updateItem(processId, {"state": "Uploaded", "rawS3Key": s3Key})
  SQS.sendMessage({"processId": processId, "action": "process"});
}
```

The problem with this implementation is that it's not idempotent. Imagine that `SQS.sendMessage` fails. The state transition will fail, so you retry. But the second call to `uploaded(s3Key)` will throw a "transition not allowed" error because `DynamoDB.updateItem` was successful during the first call.

To fix that, you need to change the `if` statement to make the function idempotent:

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== "Created" && process.state !== "Uploaded") {
    throw new Error("transition not allowed")
  }
  DynamoDB.updateItem(processId, {"state": "Uploaded", "rawS3Key": s3Key})
  SQS.sendMessage({"processId": processId, "action": "process"});
}
```

If you retry now, you'll make multiple updates to DynamoDB, which doesn't hurt. And you may send multiple SQS messages, which also doesn't hurt, because the SQS message consumer must be idempotent as well. The same applies to the transition `Uploaded` -> `Processed`.

Next, you'll begin to implement the Imagery server.

13.3.2 Implementing a fault-tolerant web service

We'll split the Imagery application into two parts: a server and a worker. The server is responsible for providing the REST API to the user, and the worker handles consuming SQS messages and processing images.

> **Where is the code located?**
> As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code. Imagery is located in /chapter13/.

The server will support the following routes:

- `POST /image`—A new image process is created when executing this route.
- `GET /image/:id`—This route returns the state of the process specified with the path parameter `:id`.
- `POST /image/:id/upload`—This route offers a file upload for the process specified with the `path` parameter `:id`.

To implement the server, you'll again use Node.js and the Express web application framework. You'll only use Express framework a little, so you won't be bothered by it.

SETTING UP THE SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and things like that, as shown in the next listing.

Listing 13.1 Initializing the Imagery server (server/server.js)

```
var express = require('express');
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var uuid = require('node-uuid');
var multiparty = require('multiparty');

var db = new AWS.DynamoDB({
  "region": "us-east-1"
});
var sqs = new AWS.SQS({
  "region": "us-east-1"
});
var s3 = new AWS.S3({
  "region": "us-east-1"
});

var app = express();
app.use(bodyParser.json());

[...]
```

Loads Node.js modules (dependencies)

Creates a DynamoDB endpoint

Creates an SQS endpoint

Creates an S3 endpoint

Creates an Express application

Tells Express to parse the request bodies

```
app.listen(process.env.PORT || 8080, function() {
  console.log("Server started. Open http://localhost:"
  + (process.env.PORT || 8080) + " with browser.");
});
```

Starts Express on the port defined by the environment variable PORT, or defaults to 8080

Don't worry too much about the boilerplate code; the interesting parts will follow.

CREATING A NEW IMAGERY PROCESS

To provide a REST API to create image processes, a fleet of EC2 instances will run Node.js code behind a load balancer. The image processes will be stored in DynamoDB. Figure 13.11 shows the flow of a request to create a new image process.

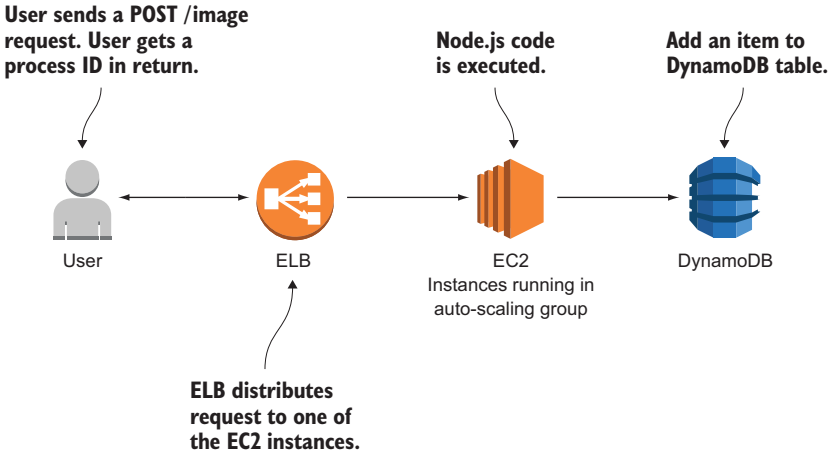

Figure 13.11 **Creating a new image process in Imagery**

You'll now add a route to the Express application to handle `POST /image` requests, as shown in the next listing.

Listing 13.2 **Imagery server: `POST /image` creates an image process**

```
app.post('/image', function(request, response) {
  var id = uuid.v4();
  db.putItem({
    "Item": {
      "id": {
        "S": id
      },
      "version": {
        "N": "0"
      },
```

Registers the route with Express

Creates a unique ID for the process

Invokes the putItem operation on DynamoDB

The id attribute will be the primary key in DynamoDB.

Use the version for optimistic locking (explained in the following sidebar).

A new process can now be created.

Optimistic locking

To prevent multiple updates to a DynamoDB item, you can use a trick called *optimistic locking*. When you want to update an item, you must tell which version you want to update. If that version doesn't match the current version of the item in the database, your update will be rejected.

Imagine the following scenario. An item is created in version 0. Process A looks up that item (version 0). Process B also looks up that item (version 0). Now process A wants to make a change by invoking the `updateItem` operation on DynamoDB. Therefore process A specifies that the expected version is 0. DynamoDB will allow that modification because the version matches; but DynamoDB will also change the item's version to 1 because an update was performed. Now process B wants to make a modification and sends a request to DynamoDB with the expected item version 0. DynamoDB will reject that modification because the expected version doesn't match the version DynamoDB knows of, which is 1.

To solve the problem for process B, you can use the same trick introduced earlier: retry. Process B will again look up the item, now in version 1, and can (you hope) make the change.

There's one problem with optimistic locking: if many modifications happen in parallel, a lot of overhead is created because of many retries. But this is only a problem if you expect a lot of concurrent writes to a single item, which can be solved by changing the data model. That's not the case in the Imagery application. Only a few writes are expected to happen for a single item: optimistic locking is a perfect fit to make sure you don't have two writes where one overrides changes made by another.

The opposite of *optimistic locking* is *pessimistic locking*. A pessimistic lock strategy can be implemented by using a semaphore. Before you change data, you need to lock the semaphore. If the semaphore is already locked, you wait until the semaphore becomes free again.

The next route you need to implement is to look up the current state of a process.

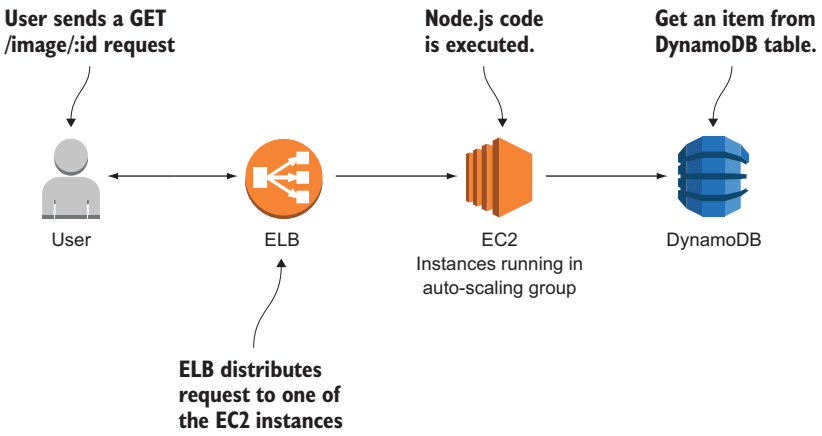

Figure 13.12 **Looking up an image process in Imagery to return its state**

LOOKING UP AN IMAGERY PROCESS

You'll now add a route to the Express application to handle `GET /image/:id` requests. Figure 13.12 shows the request flow.

Express will take care of the path parameter `:id` by providing it within `request.params.id`. The implementation needs to get an item from DynamoDB based on the path parameter ID.

Listing 13.3 **Imagery server: `GET /image/:id` looks up an image process**

```
function mapImage(item) {
  return {
    "id": item.id.S,
    "version": parseInt(item.version.N, 10),
    "state": item.state.S,
    "rawS3Key": [...]
    "processedS3Key": [...]
    "processedImage": [...]
  };
};

function getImage(id, cb) {
  db.getItem({
    "Key": {
      "id": {
        "S": id
      }
    },
    "TableName": "imagery-image"
  }, function(err, data) {
    if (err) {
```

Helper function to map a DynamoDB result to a JavaScript object

Invokes the getItem operation on DynamoDB

id is the primary hash key.

```
      cb(err);
    } else {
      if (data.Item) {
        cb(null, mapImage(data.Item));
      } else {
        cb(new Error("image not found"));
      }
    }
  });
}
app.get('/image/:id', function(request, response) {
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);
    }
  });
});
```

Registers the route with Express

Responds with the image process

The only thing missing is the upload part, which comes next.

UPLOADING AN IMAGE

Uploading an image via `POST` request requires several steps:

1. Upload the raw image to S3.
2. Modify the item in DynamoDB.
3. Send an SQS message to trigger processing.

Figure 13.13 shows this flow.

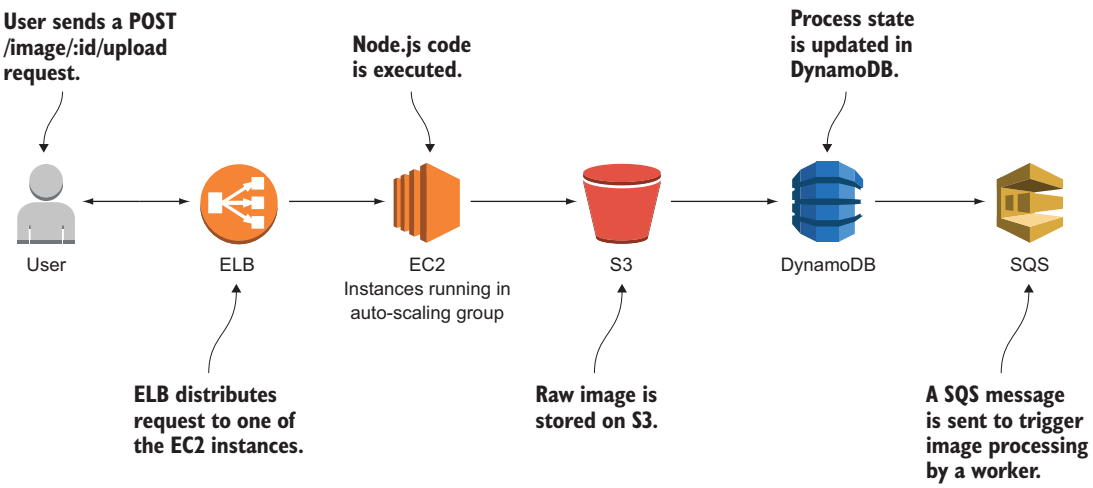

Figure 13.13 Uploading a raw image to Imagery and triggering image processing

The following listing shows the implementation of these steps.

Listing 13.4 Imagery server: `POST /image/:id/upload` uploads an image

```
function uploadImage(image, part, response) {
  var rawS3Key = 'upload/' + image.id + '-' + Date.now();
  s3.putObject({
    "Bucket": process.env.ImageBucket,
    "Key": rawS3Key,
    "Body": part,
    "ContentLength": part.byteCount
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
        "Key": {
          "id": {
            "S": image.id
          }
        },
        "UpdateExpression": "SET #s=:newState,
        ➥ version=:newVersion, rawS3Key=:rawS3Key",
        "ConditionExpression": "attribute_exists(id)
        ➥ AND version=:oldVersion
        ➥ AND #s IN (:stateCreated, :stateUploaded)",
        "ExpressionAttributeNames": {
          "#s": "state"
        },
        "ExpressionAttributeValues": {
          ":newState": {
            "S": "uploaded"
          },
          ":oldVersion": {
            "N": image.version.toString()
          },
          ":newVersion": {
            "N": (image.version + 1).toString()
          },
          ":rawS3Key": {
            "S": rawS3Key
          },
          ":stateCreated": {
            "S": "created"
          },
          ":stateUploaded": {
            "S": "uploaded"
          }
        },
        "ReturnValues": "ALL_NEW",
        "TableName": "imagery-image"
      }, function(err, data) {
        if (err) {
```

Creates a key for the S3 object

Invokes putObject on S3

The S3 bucket name is passed in as an environment variable (the bucket will be created later in the chapter).

body is the uploaded stream of data.

Invokes updateItem on DynamoDB

Updates the state, version, and raw S3 key

Updates only when item exists. Version equals the expected version, and state is one of those allowed.

```
        throw err;
      } else {
        sqs.sendMessage({
          "MessageBody": JSON.stringify({
                           "imageId": image.id,
                           "desiredState": "processed"
                         }),
          "QueueUrl": process.env.ImageQueue,
        }, function(err) {
          if (err) {
            throw err;
          } else {
            response.json(lib.mapImage(data.Attributes));
          }
        });
      }
    });
  }
});

app.post('/image/:id/upload', function(request, response) {
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      var form = new multiparty.Form();
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
```

Invokes sendMessage on SQS

Message contains the process ID

The queue URL is passed in as an environment variable.

Registers the route with Express

Magic lines to handle uploads

The server side is finished. Next you'll continue to implement the processing part in the Imagery worker. After that, you can deploy the application.

13.3.3 Implementing a fault-tolerant worker to consume SQS messages

The Imagery worker does the asynchronous stuff in the background: processing images into sepia images while applying a filter. The worker handles consuming SQS messages and processing images. Fortunately, consuming SQS messages is a common task that's solved by Elastic Beanstalk, which you'll use later to deploy the application. Elastic Beanstalk can be configured to listen to SQS messages and execute an HTTP `POST` request for every message. In the end, the worker implements a REST API that's invoked by Elastic Beanstalk. To implement the worker, you'll again use Node.js and the Express framework.

SETTING UP THE SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and so on, as shown in the following listing.

Listing 13.5 Initializing the Imagery worker (worker/worker.js)

```
var express = require('express');
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var assert = require('assert-plus');
var Caman = require('caman').Caman;
var fs = require('fs');

var db = new AWS.DynamoDB({
  "region": "us-east-1"
});
var s3 = new AWS.S3({
  "region": "us-east-1"
});

var app = express();
app.use(bodyParser.json());

app.get('/', function(request, response) {
  response.json({});
});

[...]

app.listen(process.env.PORT || 8080, function() {
  console.log("Worker started on port " + (process.env.PORT || 8080));
});
```

Loads Node.js modules (dependencies)

Creates a DynamoDB endpoint

Creates an S3 endpoint

Creates an Express application

Registers a route for health checks that returns an empty object

Starts Express on a port defined by the environment variable PORT, or defaults to 8080

The Node.js module caman is used to create sepia images. You'll wire that up next.

HANDLING SQS MESSAGES AND PROCESSING THE IMAGE

The SQS message to trigger the raw image processing is handled in the worker. Once a message is received, the worker starts to download the raw image from S3, applies the sepia filter, and uploads the processed image back to S3. After that, the process state in DynamoDB is modified. Figure 13.14 shows the steps.

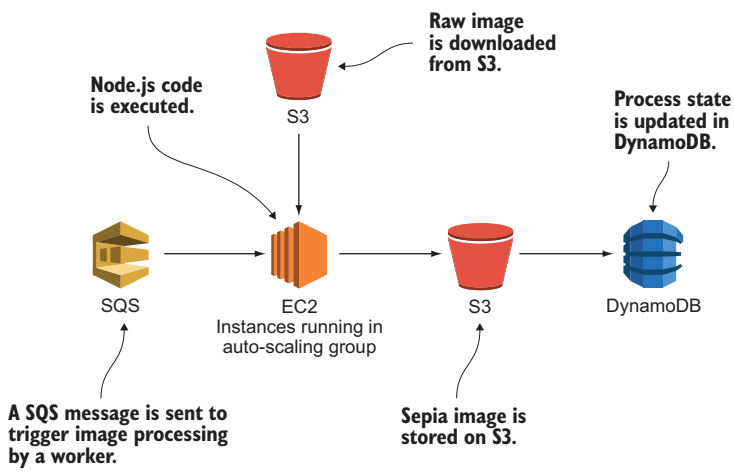

Figure 13.14 Processing a raw image to upload a sepia image to S3

Instead of receiving messages directly from SQS, you'll take a shortcut. Elastic Beanstalk, the deployment tool you'll use, provides a feature that consumes messages from a queue and invokes a HTTP `POST` request for every message. You configure the `POST` request to be made to the resource `/sqs`. The following listing shows the implementation.

Listing 13.6 Imagery worker: `POST /sqs` handles SQS messages

```
function processImage(image, cb) {
  var processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png';
  // download raw image from S3
  // process image
  // upload sepia image to S3
  cb(null, processedS3Key);
}

function processed(image, request, response) {
  processImage(image, function(err, processedS3Key) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
        "Key": {
          "id": {
            "S": image.id
          }
        },
        "UpdateExpression": "SET #s=:newState,
        ➥ version=:newVersion, processedS3Key=:processedS3Key",
        "ConditionExpression": "attribute_exists(id)
        ➥ AND version=:oldVersion
        ➥ AND #s IN (:stateUploaded, :stateProcessed)",
        "ExpressionAttributeNames": {
          "#s": "state"
        },
        "ExpressionAttributeValues": {
          ":newState": {
            "S": "processed"
          },
          ":oldVersion": {
            "N": image.version.toString()
          },
          ":newVersion": {
            "N": (image.version + 1).toString()
          },
          ":processedS3Key": {
            "S": processedS3Key
          },
          ":stateUploaded": {
            "S": "uploaded"
          },
```

The implementation of processImage isn't shown here; you can find it in the book's source folder.

Invokes the updateItem operation on DynamoDB

Updates the state, version, and processed S3 key

Updates only when an item exists, version equals the expected version, and state is one of those allowed

```
            ":stateProcessed": {
              "S": "processed"
            }
          },
          "ReturnValues": "ALL_NEW",
          "TableName": "imagery-image"
        }, function(err, data) {
          if (err) {
            throw err;
          } else {
            response.json(lib.mapImage(data.Attributes));
          }
        });
      }
    });
}

app.post('/sqs', function(request, response) {
  assert.string(request.body.imageId, "imageId");
  assert.string(request.body.desiredState, "desiredState");
  getImage(request.body.imageId, function(err, image) {
    if (err) {
      throw err;
    } else {
      if (request.body.desiredState === 'processed') {
        processed(image, request, response);
      } else {
        throw new Error("unsupported desiredState");
      }
    }
  });
});
```

Responds with the process's new state

Registers the route with Express

The implementation of getImage is the same as on the server.

Invokes the processed function if the SQS message's desiredState equals "processed".

If the `POST /sqs` route responds with a 2XX HTTP status code, Elastic Beanstalk considers the message delivery successful and deletes the message from the queue. Otherwise the message is redelivered.

Now you can process the SQS message to process the raw image and upload a sepia image to S3. The next step is to deploy all that code to AWS in a fault-tolerant way.

13.3.4 Deploying the application

As mentioned previously, you'll use Elastic Beanstalk to deploy the server and the worker. You'll use CloudFormation to do so. This may sounds strange because you use an automation tool to use another automation tool. But CloudFormation does a bit more than deploy two Elastic Beanstalk applications. It defines the following:

- S3 bucket for raw and processed images
- DynamoDB table `imagery-image`
- SQS queue and dead-letter queue
- IAM roles for the server and worker EC2 instances
- Elastic Beanstalk application for the server and worker

It takes quite a while to create that CloudFormation stack; that's why you should do so now. After you've created the stack, we'll look at the template. After that, the stack should be ready to use.

To help you deploy Imagery, we created a CloudFormation template located at https://s3.amazonaws.com/awsinaction/chapter13/template.json. Create a stack based on that template. The stack output `EndpointURL` returns the URL that can be accessed from your browser to use Imagery. Here's how to create the stack from the terminal:

```
$ aws cloudformation create-stack --stack-name imagery \
--template-url https://s3.amazonaws.com/\
awsinaction/chapter13/template.json \
--capabilities CAPABILITY_IAM
```

Now let's look at the CloudFormation template.

DEPLOYING S3, DYNAMODB, AND SQS

The following CloudFormation snippet describes the S3 bucket, DynamoDB table, and SQS queue.

Listing 13.7 Imagery CloudFormation template: S3, DynamoDB, and SQS

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "AWS in Action: chapter 13",
  "Parameters": {
    "KeyName": {
      "Description": "Key Pair name",
      "Type": "AWS::EC2::KeyPair::KeyName",
      "Default": "mykey"
    }
  },
  "Resources": {
    "Bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": {"Fn::Join": ["-",
        ["imagery", {"Ref": "AWS::AccountId"}]]},
        "WebsiteConfiguration": {
          "ErrorDocument": "error.html",
          "IndexDocument": "index.html"
        }
      }
    },
    "Table": {
      "Type": "AWS::DynamoDB::Table",
      "Properties": {
        "AttributeDefinitions": [{
          "AttributeName": "id",
          "AttributeType": "S"
        }],
        "KeySchema": [{
          "AttributeName": "id",
          "KeyType": "HASH"
```

S3 bucket for uploaded and processed images, with web hosting enabled

The bucket name contains the account ID to make the name unique.

DynamoDB table containing the image processes

The id attribute is used as the primary hash key.

```
        }],
        "ProvisionedThroughput": {
          "ReadCapacityUnits": 1,
          "WriteCapacityUnits": 1
        },
        "TableName": "imagery-image"
      }
    },
    "SQSDLQueue": {
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": "message-dlq"
      }
    },
    "SQSQueue": {
      "Type": "AWS::SQS::Queue",
      "Properties": {
        "QueueName": "message",
        "RedrivePolicy": {
          "deadLetterTargetArn": {"Fn::GetAtt":
          ["SQSDLQueue", "Arn"]},
          "maxReceiveCount": 10
        }
      }
    },
    [...]
  },
  "Outputs": {
    "EndpointURL": {
      "Value": {"Fn::GetAtt": ["EBServerEnvironment", "EndpointURL"]},
      "Description": "Load Balancer URL"
    }
  }
}
```

SQS queue that receives messages that can't be processed

SQS queue to trigger image processing

If a message is received more than 10 times, it's moved to the dead-letter queue.

Visit the output with your browser to use Imagery.

The concept of a dead-letter queue needs a short introduction here as well. If a single SQS message can't be processed, the message becomes visible again on the queue for other workers. This is called a *retry*. But if for some reason every retry fails (maybe you have a bug in your code), the message will reside in the queue forever and may waste a lot of resources because of many retries. To avoid this, you can configure a *dead-letter queue (DLQ)*. If a message is retried more than a specific number of times, it's removed from the original queue and forwarded to the DLQ. The difference is that no worker listens for messages on the DLQ. But you should create a CloudWatch alarm that triggers if the DLQ contains more than zero messages because you need to investigate this problem manually by looking at the message in the DLQ.

Now that the basic resources have been designed, let's move on to the more specific resources.

IAM ROLES FOR SERVER AND WORKER EC2 INSTANCES

Remember that it's important to only grant the privileges that are needed. All server instances must be able to do the following:

- `sqs:SendMessage` to the SQS queue created in the template to trigger image processing
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the upload/ key prefix)
- `dynamodb:GetItem`, `dynamodb:PutItem`, and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

All worker instances must be able to do the following:

- `sqs:ChangeMessageVisibility`, `sqs:DeleteMessage`, and `sqs:ReceiveMessage` to the SQS queue created in the template
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the processed/ key prefix)
- `dynamodb:GetItem` and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

If you don't feel comfortable with IAM roles, take a look at the book's code repository on GitHub at https://github.com/AWSinAction/code. The template with IAM roles can be found in /chapter13/template.json.

Now it's time to design the Elastic Beanstalk applications.

ELASTIC BEANSTALK FOR THE SERVER

Let's have a short refresher on Elastic Beanstalk, which we touched on in section 5.3. An Elastic Beanstalk consists of these elements:

- An *application* is a logical container. It contains versions, environments, and configurations. To use AWS Elastic Beanstalk in a region, you have to create an application first.
- A *version* contains a specific version of your application. To create a new version, you have to upload your executables (packed into an archive) to S3. A version is basically a pointer to this archive of executables.
- A *configuration template* contains your default configuration. You can manage the configuration of your application (such as the port your application listens on) as well as the configuration of the environment (such as the size of the virtual server) with your custom configuration template.
- An *environment* is the place where AWS Elastic Beanstalk executes your application. It consists of a version and the configuration. You can run multiple environments for one application by using the versions and configurations multiple times.

Figure 13.15 shows the parts of an Elastic Beanstalk application.

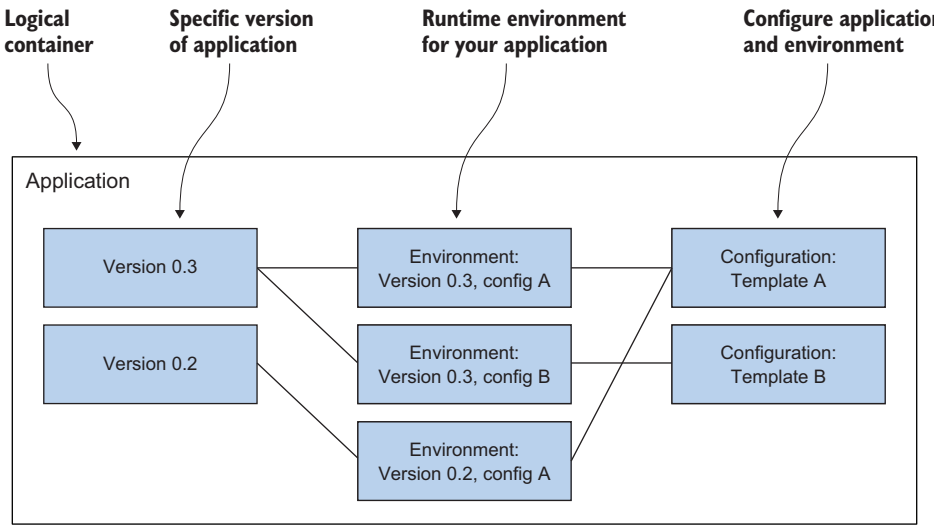

Figure 13.15 An AWS Elastic Beanstalk application consists of versions, configurations, and environments.

Now that you've refreshed your memory, let's look at the Elastic Beanstalk application that deploys the Imagery server.

Listing 13.8 Imagery CloudFormation template: Elastic Beanstalk for the server

Describes the server application container

```
"EBServerApplication": {
  "Type": "AWS::ElasticBeanstalk::Application",
  "Properties": {
    "ApplicationName": "imagery-server",
    "Description": "Imagery server: AWS in Action: chapter 13"
  }
},
"EBServerConfigurationTemplate": {
  "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",
  "Properties": {
    "ApplicationName": {"Ref": "EBServerApplication"},
    "Description": "Imagery server: AWS in Action: chapter 13",
    "SolutionStackName":
    "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",
    "OptionSettings": [{
      "Namespace": "aws:autoscaling:asg",
      "OptionName": "MinSize",
      "Value": "2"
    }, {
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "EC2KeyName",
      "Value": {"Ref": "KeyName"}
    }, {
      "Namespace": "aws:autoscaling:launchconfiguration",
      "OptionName": "IamInstanceProfile",
```

Uses Amazon Linux 2015.03 running Node.js 0.12.6

Minimum of two EC2 instances for fault-tolerance

Passes a value from the KeyName parameter

```
        "Value": {"Ref": "ServerInstanceProfile"}
      }, {
        "Namespace": "aws:elasticbeanstalk:container:nodejs",
        "OptionName": "NodeCommand",
        "Value": "node server.js"
      }, {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "ImageQueue",
        "Value": {"Ref": "SQSQueue"}
      }, {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "ImageBucket",
        "Value": {"Ref": "Bucket"}
      }, {
        "Namespace": "aws:elasticbeanstalk:container:nodejs:staticfiles",
        "OptionName": "/public",
        "Value": "/public"
      }]
    }
  },
  "EBServerApplicationVersion": {
    "Type": "AWS::ElasticBeanstalk::ApplicationVersion",
    "Properties": {
      "ApplicationName": {"Ref": "EBServerApplication"},
      "Description": "Imagery server: AWS in Action: chapter 13",
      "SourceBundle": {
        "S3Bucket": "awsinaction",
        "S3Key": "chapter13/build/server.zip"
      }
    }
  },
  "EBServerEnvironment": {
    "Type": "AWS::ElasticBeanstalk::Environment",
    "Properties": {
      "ApplicationName": {"Ref": "EBServerApplication"},
      "Description": "Imagery server: AWS in Action: chapter 13",
      "TemplateName": {"Ref": "EBServerConfigurationTemplate"},
      "VersionLabel": {"Ref": "EBServerApplicationVersion"}
    }
  }
```

Links to the IAM instance profile created in the previous section

Start command

Passes the SQS queue into an environment variable

Passes the S3 bucket into an environment variable

Serves all files from /public as static files

Loads code from the book's S3 bucket

Under the hood, Elastic Beanstalk uses an ELB to distribute the traffic to the EC2 instances that are also managed by Elastic Beanstalk. You only need to worry about the configuration of Elastic Beanstalk and the code.

ELASTIC BEANSTALK FOR THE WORKER

The worker Elastic Beanstalk application is similar to the server. The differences are highlighted in the following listing.

Listing 13.9 Imagery CloudFormation template: Elastic Beanstalk for the worker

```
"EBWorkerApplication": {
  "Type": "AWS::ElasticBeanstalk::Application",
  "Properties": {
```

Describes the worker application container

```
      "ApplicationName": "imagery-worker",
      "Description": "Imagery worker: AWS in Action: chapter 13"
    }
  },
  "EBWorkerConfigurationTemplate": {
    "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",
    "Properties": {
      "ApplicationName": {"Ref": "EBWorkerApplication"},
      "Description": "Imagery worker: AWS in Action: chapter 13",
      "SolutionStackName":
      "64bit Amazon Linux 2015.03 v1.4.6 running Node.js",
      "OptionSettings": [{
        "Namespace": "aws:autoscaling:launchconfiguration",
        "OptionName": "EC2KeyName",
        "Value": {"Ref": "KeyName"}
      }, {
        "Namespace": "aws:autoscaling:launchconfiguration",
        "OptionName": "IamInstanceProfile",
        "Value": {"Ref": "WorkerInstanceProfile"}
      }, {
        "Namespace": "aws:elasticbeanstalk:sqsd",
        "OptionName": "WorkerQueueURL",
        "Value": {"Ref": "SQSQueue"}
      }, {
        "Namespace": "aws:elasticbeanstalk:sqsd",
        "OptionName": "HttpPath",
        "Value": "/sqs"
      }, {
        "Namespace": "aws:elasticbeanstalk:container:nodejs",
        "OptionName": "NodeCommand",
        "Value": "node worker.js"
      }, {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "ImageQueue",
        "Value": {"Ref": "SQSQueue"}
      }, {
        "Namespace": "aws:elasticbeanstalk:application:environment",
        "OptionName": "ImageBucket",
        "Value": {"Ref": "Bucket"}
      }]
    }
  },
  "EBWorkerApplicationVersion": {
    "Type": "AWS::ElasticBeanstalk::ApplicationVersion",
    "Properties": {
      "ApplicationName": {"Ref": "EBWorkerApplication"},
      "Description": "Imagery worker: AWS in Action: chapter 13",
      "SourceBundle": {
        "S3Bucket": "awsinaction",
        "S3Key": "chapter13/build/worker.zip"
      }
    }
  },
  "EBWorkerEnvironment": {
    "Type": "AWS::ElasticBeanstalk::Environment",
```

Configures the HTTP resource that's invoked when an SQS message is received

```
  "Properties": {
    "ApplicationName": {"Ref": "EBWorkerApplication"},
    "Description": "Imagery worker: AWS in Action: chapter 13",
    "TemplateName": {"Ref": "EBWorkerConfigurationTemplate"},
    "VersionLabel": {"Ref": "EBWorkerApplicationVersion"},
    "Tier": {
      "Type": "SQS/HTTP",
      "Name": "Worker",
      "Version": "1.0"
    }
  }
}
```

Switches to the worker environment tier (pushes SQS messages to your app)

After all that JSON reading, the CloudFormation stack should be created. Verify the status of your stack:

```
$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [{
    [...]
    "Description": "AWS in Action: chapter 13",
    "Outputs": [{
      "Description": "Load Balancer URL",
      "OutputKey": "EndpointURL",
      "OutputValue": "awseb-...582.us-east-1.elb.amazonaws.com"
    }],
    "StackName": "imagery",
    "StackStatus": "CREATE_COMPLETE"
  }]
}
```

Copy this output into your web browser.

Wait until CREATE_COMPLETE is reached.

The `EndpointURL` output of the stack is the URL to access the Imagery application. When you open Imagery in your web browser, you can upload an image as shown in figure 13.16.

Go ahead and upload some images. You've created a fault-tolerant application!

Cleaning up

To find out your 12-digit account ID (878533158213), you can use the CLI:

```
$ aws iam get-user --query "User.Arn" --output text
arn:aws:iam::878533158213:user/mycli
```

Delete all the files in the S3 bucket `s3://imagery-$AccountId` (replace `$AccountId` with your account ID) by executing

```
$ aws s3 rm s3://imagery-$AccountId --recursive
```

Execute the following command to delete the CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name imagery
```

Stack deletion will take some time.

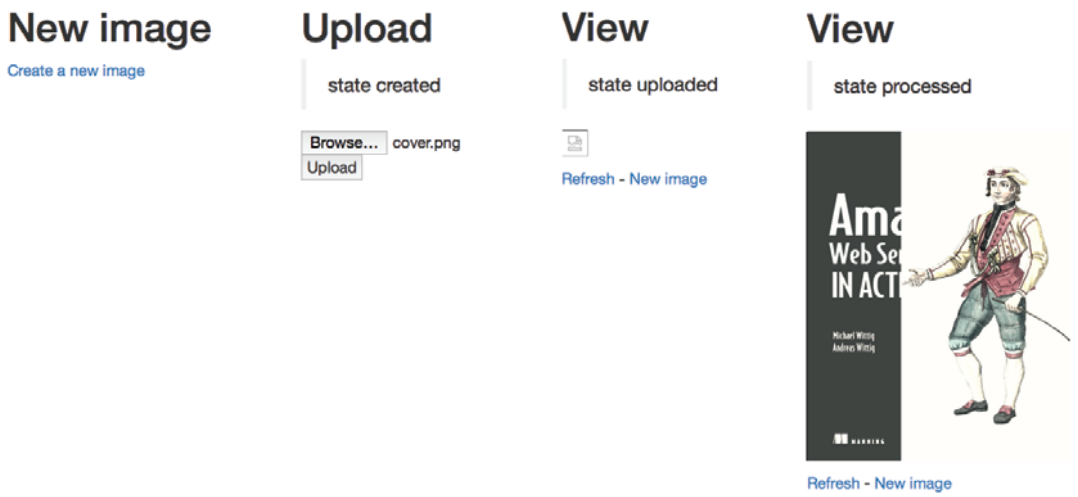

Figure 13.16 **The Imagery application in action**

13.4 Summary

- Fault-tolerance means to expect that failures happen. Design your systems in such a way that they can deal with failure.
- To create a fault-tolerant application, you can use idempotent actions to transfer from one state to the next.
- State shouldn't reside on the server (a stateless server) as a prerequisite for fault-tolerance.
- AWS offers fault-tolerant services and gives you all the tools you need to create fault-tolerant systems. EC2 is one of the few services that isn't fault-tolerant out of the box.
- You can use multiple EC2 instances to eliminate the single point of failure. Redundant EC2 instances in different availability zones, started with an auto-scaling group, are the way to make EC2 fault-tolerant.

SOFTWARE ENGINEERING

Amazon Web Services IN ACTION

Andreas and Michael Wittig

Physical data centers require lots of equipment and take time and resources to manage. If you need a data center, but don't want to build your own, Amazon Web Services may be your solution. Whether you're analyzing real-time data, building software as a service, or running an e-commerce site, AWS offers you a reliable cloud-based platform with services that scale.

Amazon Web Services in Action introduces you to computing, storing, and networking in the AWS cloud. You'll start with an overview of cloud computing and then begin setting up your account. You'll learn how to automate your infrastructure by programmatically calling the AWS API to control every part of AWS. Next, you'll learn options and techniques for storing your data. You'll also learn how to isolate your systems using private networks to increase security. Finally, this book teaches you how to design for high availability and fault tolerance.

What's Inside

- Overview of cloud concepts and patterns
- Deploy applications on AWS
- Integrate Amazon's pre-built services
- Manage servers on EC2 for cost-effectiveness

Written for developers and DevOps engineers moving distributed applications to the AWS platform.

Andreas Wittig and **Michael Wittig** are software engineers and consultants focused on AWS and web development.

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

"A confident, practical guide through the maze of the industry's leading cloud platform."
—From the Foreword by Ben Whaley

"Fantastic introduction to cloud basics with excellent real-world examples."
—Rambabu Posa, GL Assessment

"A very thorough and practical guide to everything AWS … highly recommended."
—Scott M. King, Amazon

"Cuts through the vast expanse of official documentation and gives you what you need to make AWS work now!"
—Carm Vecchio, Computer Science Corporation (CSC)

ISBN 13: 978-1-61729-288-0
ISBN 10: 1-61729-288-5
54999
9 781617 292880

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]