Book Collection

Learning Path
Building Microservices with JavaScript

Learn quick and practical methods for developing microservices

Diogo Resende and Paul Osman

Packt
www.packt.com

Building Microservices with JavaScript

Learn quick and practical methods for developing microservices

Diogo Resende
Paul Osman

Packt>

BIRMINGHAM - MUMBAI

Building Microservices with JavaScript

First published: May 2019

Production reference: 1170519

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83882-619-2

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Diogo Resende is a developer with more than 15 years of experience, working with Node.js almost from the beginning. His computer education and experience in many industries and telecommunication projects have given him a wider background knowledge of other architecture components that influence the overall performance.

Paul Osman has been building external and internal platforms for over 10 years. From public APIs targeted at third parties to internal platform teams, he has helped build distributed systems that power large-scale consumer applications. He has managed teams of engineers to rapidly deliver service-based software systems with confidence. Paul has published articles and given multiple conference talks on microservices and DevOps. He is a passionate advocate of open technology platforms and tools.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Microservices are a popular way to build distributed systems that power modern web and mobile apps. With the help of this Learning Path, you'll learn how to develop your applications as a suite of independently deployable and scalable services.

Using an example-driven approach, this Learning Path will uncover how you can dismantle your monolithic application and embrace microservice architecture, right from architecting your services and modeling them to integrating them into your application. You'll also explore ways to overcome challenges in testing and deploying these services by setting up deployment pipelines that break down the application development process into several stages. You'll study serverless architecture for microservices and understand its benefits. Furthermore, this Learning Path delves into the patterns used for organizing services, helping you optimize request handling and processing. You'll then move on to learn the fault-tolerance and reliability patterns that help you use microservices to isolate failures in your applications.

By the end of this Learning Path, you'll have the skills necessary to build enterprise-ready applications using microservices.

This Learning Path includes content from the following Packt products:

- Hands-On Microservices with Node.js by Diogo Resende
- Microservices Development Cookbook by Paul Osman

Who this book is for

If you're a JavaScript developer looking to put your skills to work by building microservices and moving away from the monolithic architecture, this book is for you. To understand the concepts explained in this Learning Path, you must have knowledge of Node.js and be familiar with the microservices architecture.

What this book covers

`Chapter 1`, The Age of Microservices, covers the evolution of computing and how development has changed and shifted from paradigm to paradigm depending on processing capacity and user demand, ultimately resulting in the age of microservices.

`Chapter 2`, Breaking the Monolith, shows how to make the transition from monolith to microservices, with the recipes focused on architectural design. You'll learn how to manage some of the initial challenges when you begin to develop features using this new architectural style.

`Chapter 3`, Edge Services, teaches you how to use open source software to expose your services to the public internet, control routing, extend your service's functionality, and handle a number of common challenges when deploying and scaling microservices.

`Chapter 4`, Modules and Toolkits, introduces you to some modules that help you create a microservice, detailing different approaches: from very raw and simple modules, such as Micro and Express, to full toolkits, such as Hydra and Seneca.

`Chapter 5`, Building a Microservice, covers the development of a simple microservice using the most common module, Express, with a very simple HTTP interface.

`Chapter 6`, State and Security, covers the development of our microservice: from using the server filesystem to moving to a more structured database service, such as MySQL.

`Chapter 7`, Testing, shows how to use Mocha and Chai to add test coverage to our previous microservice.

`Chapter 8`, Deploying Microservices, introduces you to Docker and helps you create a container image to use to run our microservice.

`Chapter 9`, Scaling, Sharding, and Replicating, covers the concept of replication when using Docker Swarm and Kubernetes locally to scale our microservice.

`Chapter 10`, Cloud-Native Microservices, shows how to migrate our microservice from the local Kubernetes to Google Cloud Platform, as an example of a fully cloud-native microservice.

`Chapter 11`, Design Patterns, enumerates some of the most common architectural design patterns and reviews the continuous integration and deployment loop.

`Chapter 12`, Inter-Service Communication, discusses recipes that will enable you to confidently handle the various kinds of interactions we're bound to require in a microservice architecture.

`Chapter 13`, Client Patterns, discusses techniques for modeling dependent service calls and aggregating responses from various services to create client-specific APIs. We'll also discuss managing different microservices environments and making RPC consistent with JSON and HTTP, as well as the gRPC and Thrift binary protocols.

`Chapter 14`, Reliability Patterns, discusses a number of useful reliability patterns that can be used when designing and building microservices to prepare for and reduce the impact of system failures, both expected and unexpected.

`Chapter 15`, Security, includes recipes that will help you learn a number of good practices to consider when building, deploying, and operating microservices.

`Chapter 16`, Monitoring and Observability, introduces several tenants of monitoring and observability. We'll demonstrate how to modify our services to emit structured logs. We'll also take a look at metrics, using a number of different systems for collecting, aggregating, and visualizing metrics.

`Chapter 17`, Scaling, discusses load testing using different tools. We will also set up autoscaling groups in AWS, making them scalable on demand. This will be followed by strategies for capacity planning.

`Chapter 18`, Deploying Microservices, discusses containers, orchestration, and scheduling, and various methods for safely shipping changes to users. The recipes in this chapter should serve as a good starting point, especially if you're accustomed to deploying monoliths on virtual machines or bare metal servers.

To get the most out of this book

You should have basic Node.js skills and be somewhat comfortable with the language. We will cover Docker and Kubernetes, and it can be helpful to know the concepts of containers—but it's not mandatory. You need to have Node.js (and npm) installed. We recommend using the current stable version, but you're free to use a previous version if it's an LTS one, with possible adaptions. If you want to deploy Kubernetes locally, you'll need to install it later on.

This books assumes basic knowledge of microservices architectures. Other instructions are mentioned in the respective recipes as needed.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/TrainingByPackt/BuildingMicroserviceswithJavaScript` .In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalogue of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/9781838826192_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `input()` method is used to get an input from the user."

A block of code is set as follows:

```
{
"username": "paulosman",
"followings": [
"johnsmith",
"janesmith",
"petersmith"
]
}
```

Any command-line input or output is written as follows:

```
docker build . -t gcr.io/imagini-205120/imagini
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "If you need something different, click on the **DOWNLOADS** link in the header for all possible downloads: "

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

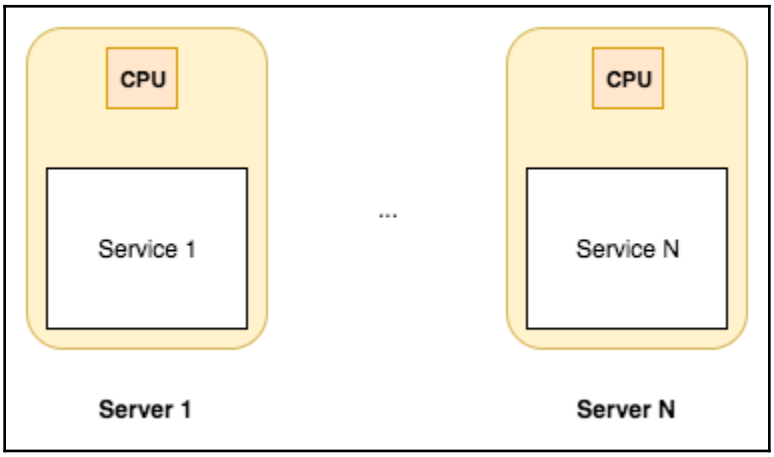
1
The Age of Microservices

Decades ago, more specifically in 1974, Intel introduced 8080 to the world, which is an 8-bit processor with a 2 MHz clock speed and 64 KB of memory. This processor was used in Altair and began the revolution in personal computers.

It was sold pre-assembled or as a kit for hobbyists. It was the first computer to have enough power to actually be used for calculations. Even though it had some poor design choices and needed an engineering major to be able to use and program it, it started the spread of personal computers to the general public.

The technology evolved rapidly and the processor industry followed Moore's law, almost doubling speed every two years. Processors were still single core, with a low-efficiency ratio (power consumption per clock cycle). Because of this, servers usually did one specific job, called a service, like serving HTTP pages or managing a **Lightweight Directory Access Protocol** (**LDAP**) directory. Services were the monolith, with very few components, and were compiled altogether to be able to take the most out of the hardware processor and memory.

In the 90s, the internet was still only available for the few. Hypertext, based on HTML and HTTP, was in its infancy. Documents were simple and browsers developed language and protocol as they pleased. Competition for market share was ferocious between Internet Explorer and Netscape. The latter introduced JavaScript, which Microsoft copied as JScript:



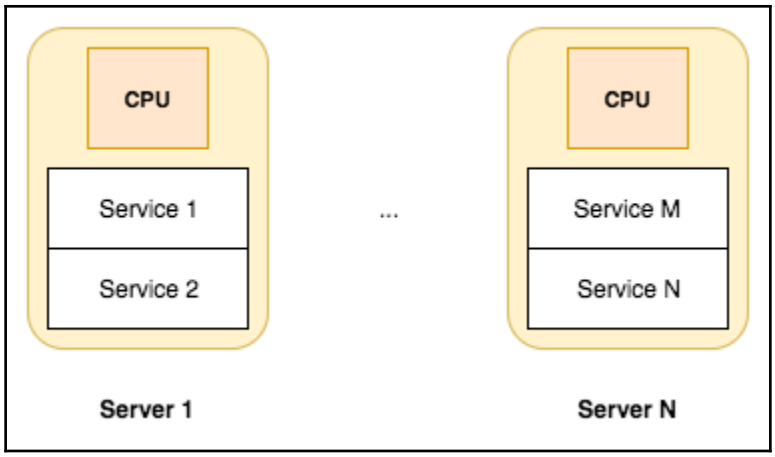
Simple single-core servers

After the turn of the century, processor speed continued to increase, memory grew to generous sizes, and 32-bit became insufficient for allocating memory addresses. The all-new 64-bit architecture appeared and personal computer processors hit the 100 W consumption mark. Servers gained muscle and were able to handle different services. Developers still avoided breaking the service into parts. Interprocess communication was considered slow and services were kept in threads, inside a single process.

The internet was starting to become largely available. Telcos started offering triple play, which included the internet bundled with television and phone services. Cellphones became part of the revolution and the age of the smartphone began.

JSON appeared as a subset of the JavaScript language, although it's considered a language-independent data format. Some web services began to support the format.

The following is an example of servers with a couple of services running, but still having only one processor.



Powerful but single-core servers

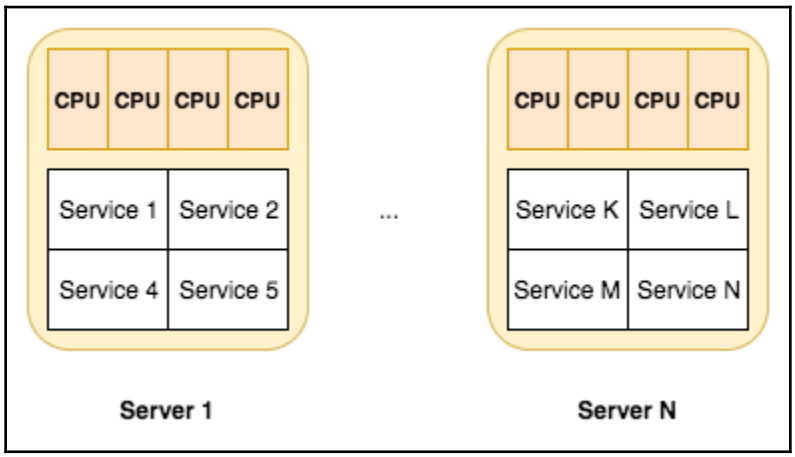
Processor evolution then shifted. Instead of the increased speed that we were used to, processors started to appear with two cores, and then four cores. Eight cores followed, and it seemed the evolution of the computer would follow this path for some time.

This also meant a shift in architecture in the development paradigms. Relying on the system to take advantage of all processors is unwise. Services started to take advantage of this new layout and now it's common to see services having at least one processor per core. Just look at any web server or proxy, such as Apache or Nginx.

The internet is now widely available. Mobile access to the internet and its information corresponds to more or less half of all internet access.

In 2012, the **Internet Engineering Task Force** (**IETF**) began its first drafts for the second version of HTTP or HTTP/2, and **World Wide Web Consortium** (**W3C**) did the same for HTML/HTML5, as both standards were old and needed a remake. Thankfully, browsers agreed on merging new features and specifications and developers no longer have the burden of developing and testing their ideas on the different browser edge cases.

The following is an example of servers with more services running as we reach a point where each server has more than one processor:



Powerful multi-core servers

Access to information in real time is a growing demand. The **Internet of Things** (**IoT**) multiplies the number of devices connected to the internet. People now have a couple of devices at home, and the number will just keep rising. Applications need to be able to handle this growth.

On the internet, HTTP is the standard protocol for communication. Routers usually do not block it, as it is considered a low traffic protocol (in contrast with video streams). This is actually not true nowadays, but it's now so widely used that changing this behavior would probably cause trouble.

Nowadays, it's actually so common to have the HTTP serving developer API working with JSON that most programming languages that release any version after 2015 probably support this data format natively.

As a consequence of processor evolution, and because of the data-demanding internet we now have, it's important to not only be able to scale a service or application to the several available cores, but also to scale outside a single hardware machine.

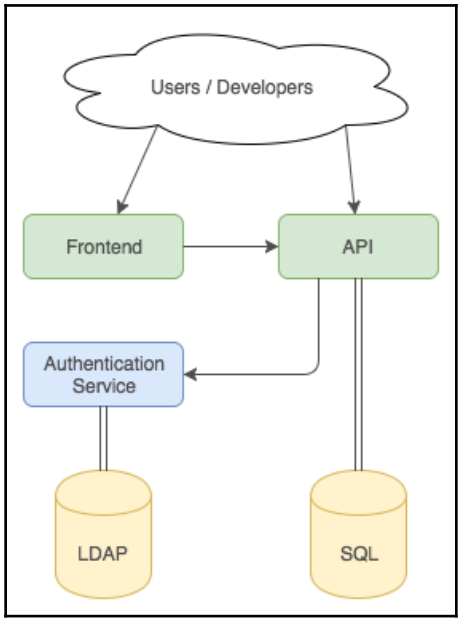
Many developers started using and following the **Service-Oriented Architecture** (**SOA**) principle. It's a principle where the architecture is focused on services, and each service presents itself to others as an application component and provides information to other application components, passing messages over some standard communication protocol.

Introducing microservices

Microservices, which are a variation of SOA, have become more and more appealing. Many projects have embraced this architecture, and it's not difficult to understand why. With the constant increase in demand for information, applications become more complex, especially with more information being transferred from new data sources to new data visualization devices.

New communication technologies have emerged, social communities spring up like mushrooms, and people expect an application to be able to merge into today's cyber lifestyle.

Microservices come to the rescue by defining a simple strategy: break every complex service into a small, simpler service that is aiming for common functionality. The idea is that services should be small and lightweight - so small that they can be easily maintained, developed, and tested, and so lightweight that they can be responsive and scale more easily:



Example of a simple microservices architecture

The preceding diagram is an example of an application that has been split into small microservices (marked as green and blue), with one for the frontend interface, another one for the API, and one just for authentication.

The idea is to decompose the business logic into small and reusable parts, easily understandable in separate chunks, enabling parallel development by different teams or groups. This way, people can develop parts without being worried about breaking an other's code. Each part should be considered a black box to other parts.

It is only important that communication is well-described. It's common for microservices to communicate over HTTP and use JSON as the data format. There are other formats available, such as XML, but they have fallen into desuetude. It's also common to use AMQP as an inter-service communication, but usually not as a public API service.

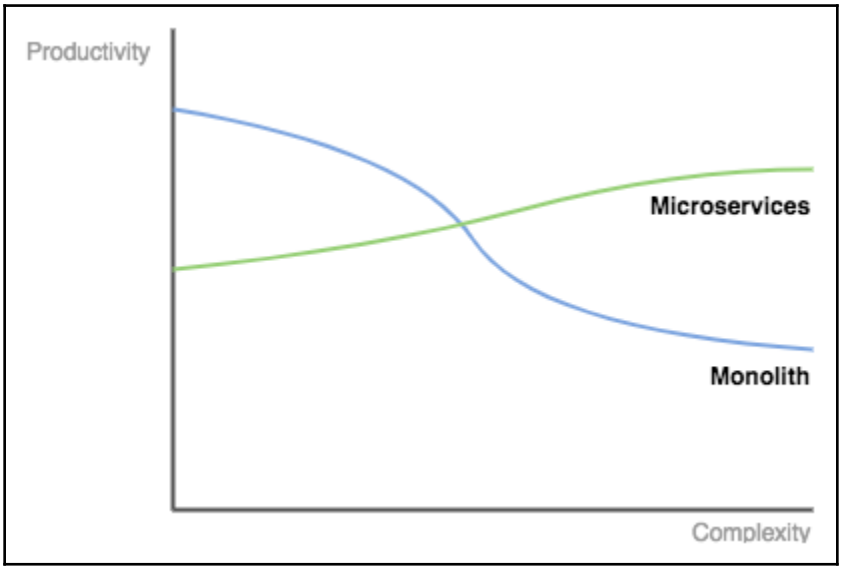
To summarize, there are several advantages of using this architecture:

- **Maintenance**: Services, when separate, become easier to develop, test, and deploy because they should be simpler and small
- **Design enforcement**: A proper and good design is enforced on the application being developed
- **Knowledge encapsulation**: Services will have specific objectives, such as delivering emails, which will lead to service re-usage and knowledge about specific tasks being grouped together in services
- **Replaceable**: Services become easier to swap because their functionality and communication is well-known
- **Technology agnostic**: Each service can be developed using the best tools and languages to build it correctly
- **Performant**: Services are small and lightweight, and, as mentioned previously, use the best tools available
- **Upgradable**: Services should be interchangeable and upgradable separately
- **Productivity**: When complexity starts to grow, productivity will be better than in a monolith application

There are also costs associated with this architecture, namely:

- **Dependencies**: Because of this architecture being technology agnostic, different dependencies for different services may arise

- **Complexity**: For small applications, the bootstrap complexity is bigger compared to the monolith
- **End-to-end testing**: It becomes more complex to test the application from end to end as the number of services to inter-connect is definitely bigger than in a monolith application



Variation in productivity as complexity grows for both monolith and microservices architecture

The graph is not to be taken very seriously; it's just an approximation of the difference between monolith and microservice architectures. In the beginning, when complexity was just beginning, productivity for microservices was poor as the architecture bootstrap demanded more work and thought.

As complexity started to increase, monolith applications became more difficult to manage and productivity began to decrease. On the other hand, as the microservices architecture started to separate services, productivity increased as the bootstrap already passed and each service was easier to manage.

Some may argue that microservices productivity will not grow as complexity will eventually also hit every service, but that's not true if a team follows the number one rule: if the complexity of a service is too much, split the service into smaller ones.

This architecture design brings long-term advantages if used correctly and across several applications. Services can be reused, which can potentially lead to more intensive usage, which will eventually lead to a more resilient and better-tested service.

Also, future applications can bootstrap faster if a development team has already bootstrapped one before. Previous services can also be integrated, which might lead to gaining an initial application testbed faster.

Using a microservices approach also helps to eliminate any long-term commitment to a technology stack. In the near future, when a team feels the need to change the stack, they can start new services using the new stack, and upgrade the old services one by one if they want to, without compromising the entire application.

Introducing Node.js

Node.js has become a very popular language, so to speak. It's not actually a language, it's a wraparound language, like JavaScript, or ECMAScript. JavaScript was developed for the browser and it is actually small by definition. Then, browsers created a layer of access to the page elements and events, called DOM. That's one of the reasons why people hate the language so much. Node.js takes only the base language and adds an API so that developers have access to I/O, namely, the filesystem and network.

Ryan Dahl started developing Node.js back in 2009. He felt the need for a performant and less blocking program than the ones that were available. Node.js used Google's V8 JavaScript engine from the beginning and was first introduced at the JSConf in Berlin in 2009.

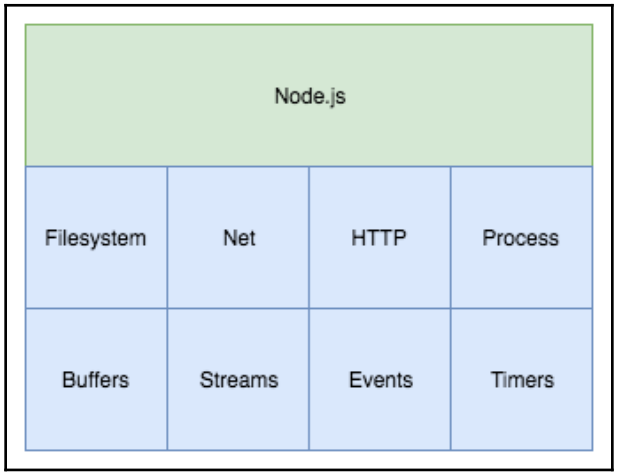
Looking just at the language, it's actually a sound and small, functional, object-oriented, prototype-based language. Everything is an object or inherits from it. Even numbers and functions inherit from an object. The good parts are as follows:

- Functions are first-class objects
- Functions and block-scoped variables
- Closures and anonymous functions
- Loose typing (can be seen as a bad aspect)

Node.js introduced JavaScript to a group of API modules that enable developers to access the filesystem, run and manage processes, and communicate over the network. Since it was first designed to replace a traditional web server, it also has HTTP and HTTPS modules to perform the roles of the client or server. Some other modules exist around these ones, which can be built as separate modules from the core (like DNS or URL), but live and are maintained inside the core.

At first, Node.js was very unstable, and not only the code's stability, but also the API's stability. Methods could change between versions dramatically. Modules got deprecated and replaced by others rapidly (search for the sys module for more information). Only brave developers would use it in production.

By the time it hit version 0.8, it became more reliable and the API had been stabilized. Large companies started supporting it and the community grew. Although there was a fork in 2014 because of internal conflicts, the community survived and the two code trees merged back in 2015:



Some of the most frequently used Node.js modules

Because of its use of Google's V8, and because it has a small and stable API, it's very fast and reliable. The reason for having a small API is that one of the guidelines of the community is to only have core functionality in the core API since everything else should go into a module. This has also become a major advantage of Node.js. It has a huge community with hundreds of thousands of modules available.

If you stop and think about it, this is the microservices approach – having separate modules to do one job and one job only, and do it well. You can easily find good, stable, and mature modules for specific needs, which are used by thousands of developers. These mature modules are all easy to deploy and have test suites to ensure they keep stable and functional.

But Node.js is not only about the modules. JavaScript has also evolved in the last couple of years and Google's V8 has always been an early adopter, so Node.js developers get access to the latest new features. Some of them have given developers new ways of simplifying code and removing some of the so-called **callback hell**.

Node.js's **Long Term Support** (**LTS**) version already has stable support for many new language features. Let's see some of them and how they're useful.

Modules

You develop Node.js code in separate files, called modules. There are three module types:

- Core modules, which you can load anywhere
- Dependency modules, which you can also load anywhere
- Local modules, which you need to load based on the relative path

Modules are loaded synchronously and cached. So, a repeated load will actually not be a load; instead, Node.js will pass you a reference to the already loaded module. This is true for all three types of modules:

```
# loading a JSON file with settings from same path as module
const settings = require("./settings");
```

Local modules are simple files where you need to know the relative or full path. You can also load a path and Node.js will look up the `index.js` file inside it. You can also load JSON files. You don't need to specify the file extension since Node.js will look for `.js` and `.json` files.

A module is nothing more than an object. The module developers decide what to expose and what not to. When you load a module, the code inside can have timers and I/O operations just like your own code. Even without you initializing it, it can run immediately after you load it.

There are hundreds of thousands of modules available. Take some time to search for something you need instead of writing one of your own. Some modules are structured so you can load parts of it (like async and lodash), thereby avoiding the memory footprint of loading everything when you just need a function or two.

Arrow functions

Arrow functions are a shorter expression function syntax without their own function scope, meaning `this` reference will point to the parent scope. This helps developers avoid saving a reference to the parent scope so they are able to reach it later on.

```
function start() {
  this.uptime = process.uptime();
  setTimeout(() => {
    console.log(this.uptime);
  }, 5000);
}
start();
```

It also helps to write less code for simple operation functions, for example, on array methods. Arrow functions are quite useful when it comes to manipulating arrays of information, whether for filtering, transforming, or reducing them to single values:

```
let double = function (value) {
    return value * 2;
};

[ 1, 2, 3 ].map(double);     // [ 2, 4, 6 ]
[ 1, 2, 3 ].map(v => v * 2); // [ 2, 4, 6 ]
```

Classes

JavaScript's classes are syntactic sugar over the inheritance model. They introduced a new way of defining object-oriented inheritance that existed in JavaScript but that new developers were not used to. They also introduced a simpler way of extending and defining an object prototype:

```
class Rectangle {
    constructor (w, h) {
        this.w = w;
        this.h = h;
    }
    get area () {
```

```
        return this.w * this.h;
    }
    static clone(r) {
        return new Rectangle(r.w, r.h);
    }
}
```

We just created a `Rectangle` class, with a constructor to specify dimensions and an area method, too. We also added a static method to clone a `Rectangle` instance.

Unlike the previous, and still possible, prototype definition, using this syntax will force a stricter development. More specifically:

- There's no hoisting, which means the class must be defined before usage
- There's no prototype redefinition

Promises and async/await

A `Promise` is an object that represents the completion or failure of an asynchronous operation. The `Promise` can be chained to perform serial operations, run in parallel until all operations execute, or even race operations and wait only for the first completion or failure:

```
Promise.race([
    new Promise((resolve, reject) => {
        // some possibly long operation
    }),
    new Promise((resolve, reject) => {
        setTimeout(reject, 5000);
    })
]).then(() => {
    console.log("success!");
}, () => {
    console.log("failed");
});
```

More recently, a syntactic sugar was created around `Promise` to mimic synchronous code syntax. Basically, you can indicate that a function is asynchronously using the `async` keyword. The function will then return a `Promise` when called. When the function returns a value, the `Promise` is resolved with that value. If the function throws an error, the `Promise` is rejected.

You can then use asynchronous functions inside other asynchronous functions, as follows:

```
function delay(timeout) {
  return new Promise((resolve) => {
    setTimeout(resolve, timeout);
  });
}
async function run() {
  await delay(1000);
  console.log("done");
}
run();
```

Spread and rest syntax

The spread syntax allows an iterable to be expanded in places where arguments (in functions) or elements (in arrays) are expected. It's very useful when, for example, a function accepts an initial set of arguments and then an unlimited one:

```
const concat = (separator, ...parts) => (parts.join(separator));

concat(", ", 1, 2, 3); // "1, 2, 3"
```

This example is especially important in arrow functions since you don't have access to the arguments object. It's also quite useful to merge arrays, as in the following example:

```
const a = [ 1, 2, 3 ];
const b = [ 4, 5, 6 ];

[ ...a, ...b ]; // [ 1, 2, 3, 4, 5, 6 ]
```

Default function parameters

There's no longer the need to use logical operators or check argument types to define default argument values. They can be defined directly in the prototype:

```
function pad(text, len, char = " ") {
    return text.substr(0, len) +
           (text.length < len ? char.repeat(text.length - len) : "");
}
pad("John", 10, "=");
```

If you've developed in languages like Python before, remember default arguments in JavaScript are evaluated at call time, not when defining the function. This means a new object is created every time:

```
function add(value, list = []) {
    list.push(value);
    return list;
}

add(1); // [ 1 ]
add(2); // [ 2 ] , not [ 1, 2 ]
```

Destructuring

Destructuring is the convenient method of constructing (target) or extracting (source) properties from an object. It gives the developer the ability to pick specific object properties from arguments or swap variable values, for example:

```
// head = 1, tail = [ 2, 3, 4]
let [ head, ...tail ] = [ 1, 2, 3, 4 ];

// list = [ "john", "jane" ]
let { users: list } = { users: [ "john", "jane" ] };
```

You can also have more complex destructuring in assignments and function arguments. You can also assign default values:

```
class Rectangle {
    constructor({ width = 100, height = 50 } = {}) {
        this.width = width;
        this.height = height;
    }
}
```

Template literals

Template literals are string literals that allow embedded expressions. Another advantage of these literals is that they can be written in multiple lines:

```
function hello(name) {
    console.log(`Hello ${name}`);
}
```

It's not a simple variable substitution, since it evaluates expressions of any kind:

```
function hello(name, age) {
    console.log(`hi ${name}, you were born in ${(new Date).getFullYear() -
age}`);
}
```

Advantages of using Node.js

Node.js has become a very strong competitor for building all kinds of applications. Large companies use it nowadays to deliver frontend interfaces and information, but also for specific backend services and developer API interfaces. Twitter, LinkedIn, and eBay are examples of these companies.

Let's go through all of the advantages of using Node.js one by one.

Node.js Package Manager

The huge number of modules available today makes it easy to start developing any kind of application or service, and npm ensures you're able to deploy your code in other servers easily. In fact, npm is also one of the advantages of Node.js and is also responsible for its spread.

```
# look how fast it checks dependencies (infinite levels), downloads them
# and installs them locally

$ npm i express
+ express@4.16.2
added 48 packages in 3.129s
```

There has always been some criticism about using Node.js, with people considering it slow and inefficient for some tasks. Some usually point to the fact that the code is single-threaded, but forget that the core API, which is your only way of communicating over the network or with the filesystem, uses a thread pool of workers to handle your actions:

```
const http = require("http");

// request is done in a separate thread, code execution continues
http
.request("http://www.google.com")
.once("response", (res) => {
    console.log(res.headers);
})
```

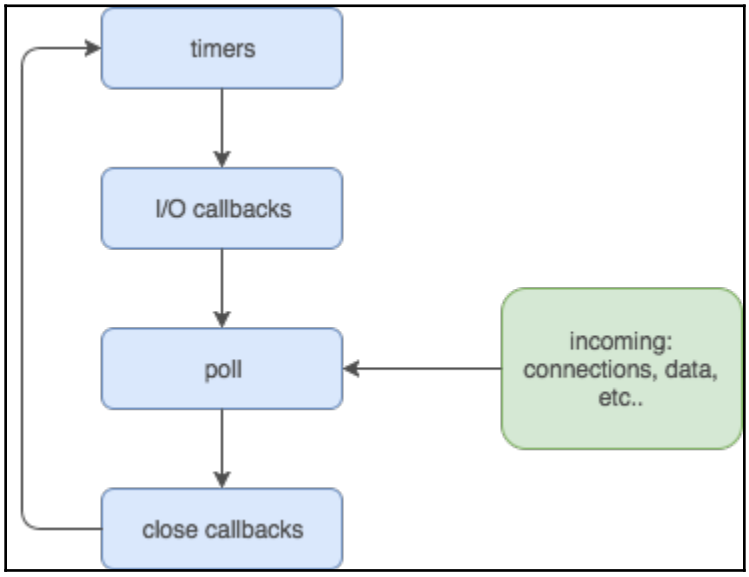
```
.end();

console.log("getting google.com headers..");
```

Asynchronous I/O

The main purpose and the initial idea behind Node.js was to be able to handle asynchronous I/O effectively. To achieve that goal, Node.js has a very good toolkit. It's built around libuv, which empowers the JavaScript language to do asynchronous I/O.

This means that it's kind of a silver bullet in this field. As long as your application is slim, not very CPU intensive, and is able to handle I/O efficiently, Node.js is the right tool for you.

Although it's true that your code runs in a single thread, as soon as your code needs to open a file or make an HTTP request, it uses other threads to do so. So, to really take advantage of the Node.js architecture, you should use it for writing code that actually needs the core API.



The simplified version of the Node.js Event Loop

The Event Loop is a loop mechanism that is responsible for handling asynchronous I/O code. The code you write synchronously runs immediately. The rest, like connecting to a third-party API, a database, or opening a file, will be queued in the poll. After that, if any timeout occurs (by a `setTimeout` or `setInterval`), they run. After that, run the I/O callbacks, if any. This is data from a file or a socket, for example. Finally, the close callbacks are executed. This is an over-simplified idea of the loop. There are several other tasks in-between (such as `setImmediate` before the close callbacks).

If you're writing code to perform some processor-intensive calculations, with big number manipulation or with fraction precision, Node.js will perform poorly. You may need to find modules to help you address these disadvantages by moving its weak points out of the JavaScript context.

If you still want to use Node.js for performance tasks, try creating a C++ module and use that instead. You can then pass the calculations to this module and still be able to use Node.js for other tasks that you would normally need more code for in C++.

Community

People around the language can also influence it, and in this case positively. There were some strange moments in the past, more specifically, sometime before the `io.js` fork. But the community didn't give up and the commitment to the language has brought it to what it is now.

It's a strong community around a structured language that has gained traction like no other. Developers know what to expect and there's an effort to keep it stable and secure, which, for me, is one of the fundamental principles for a developer to even consider using a language.

Summing up, the advantages of Node.js are as follows:

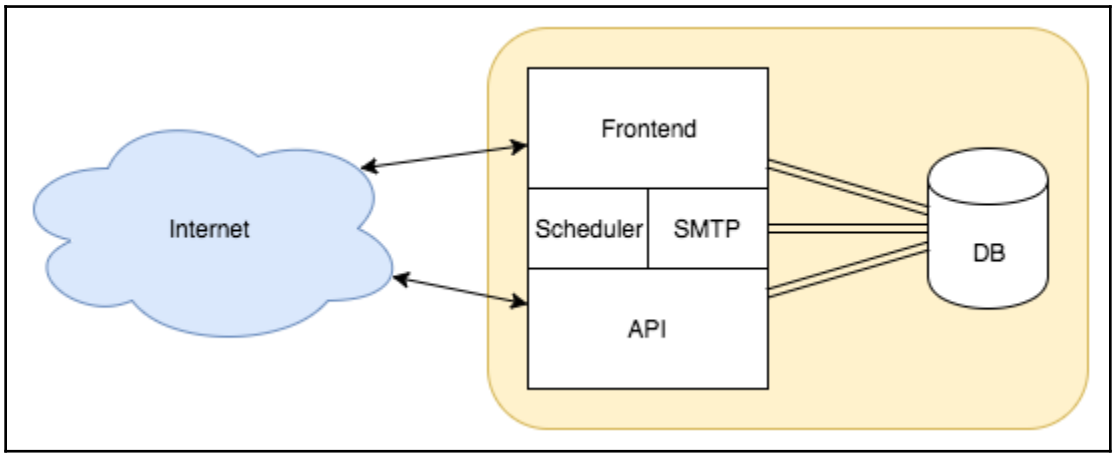
- A stable and mature language, used and known by web developers for years
- A performant core, based on Google's V8 engine
- A huge community and large module base, with dozens of modules for each task that you can choose from
- An active and maintained core, with a focus on stability and security

From monolith to microservices

As we described previously, a microservices architecture is based on a loosely coupled set of services that work together to achieve a specific target application. At the end of the spectrum, there are monolith applications.

A monolith application is composed of a set of components that are tightly coupled. These components are usually developed using the same language and the application runs as a whole. The first noticeable difference is probably the slow start. Deploying might also be slow since you might need a couple of dependencies before having anything up and running.

Let's imagine an event application, a simple one, an application that lets users define events and be notified when those events are about to start.



A monolith event application

Let's describe what the event application does:

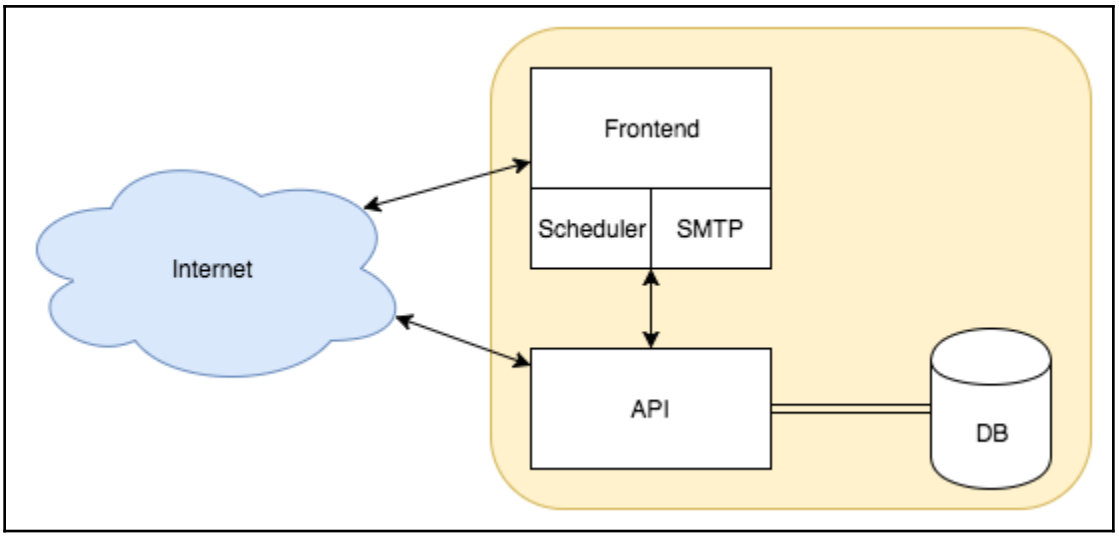
- It allows users to register themselves and add events to a calendar
- A few minutes before the event starts (that's what the **Scheduler** component is for), the users receive an email with the event information (that's the **SMTP** component)
- Users can use the frontend interface or the **API** interface

Imagine the preceding application as being a monolith (the greyed out area on the right). Imagine that all four parts are part of the same process, even though they could be in separate threads. Imagine that the database is accessed directly across the application. Sound good?

Well, it sounds terrible, perhaps not for a small application, but for a medium one, this would be a representation of chaos. Having a group of developers making new features or improvements would be a nightmare, and for new developers entering the group, it would take some time before having the base knowledge to make some changes.

The first principle that you should follow is the **Don't Repeat Yourself** (**DRY**) principle. Avoiding multiple components from accessing a data source helps developers in the future. Later on, if there's a need to change the data source or part of its structure, it will be easier if only one component manipulates it. This is not always possible, but if it is possible, you should keep the data source access to a minimum.

In our example, the API should probably have access, and all others should use the API.



A single service accessing a data source

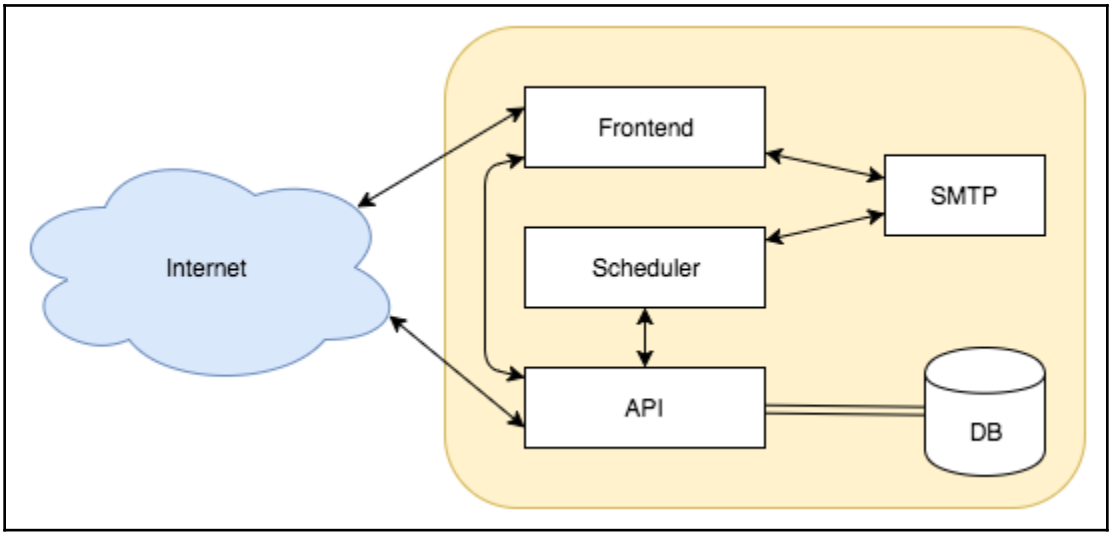
We now have two services:

- The API, which is the only service accessing the data source
- The frontend, which is the user's interface to change the data source

Although the frontend is used to manage events, it uses the API service to manipulate data sources. Besides having only one service managing data sources, it forces you to think of the API for external developers. It's a win-win.

There is still room for improvement. The **Frontend** could be in a separate service, allowing you to scale the interface according to user traffic, and have the other parts on separate services. **Scheduler** and **SMTP** are both candidates for separate services. **SMTP** should be thought of as a reusable service for other applications you might develop later on.

Let's take a look at how we could build the same application using the microservices approach:



A microservices event application

It looks more complex. Well, the architecture is more complex. The difference is that now, we have loosely coupled components, and each one of them is easily understandable and maintainable. Summing up the changes and advantages:

- The **API** is the only one accessing the database so it can change from SQLite, MongoDB, MySQL, or anything else, and no other component is affected
- **SMTP** can be used from the **Frontend** and **Scheduler**, and if you decide to change it from using a local service to using a third-party email sending **API**, you can make the change easily

- **SMTP** is a candidate for being a reusable service in other applications, meaning you can use it in other applications or event share the same service between multiple applications

You can think of these components as capabilities of your application. They can be swapped, upgraded, maintained, and scaled, all without affecting other components or your application.

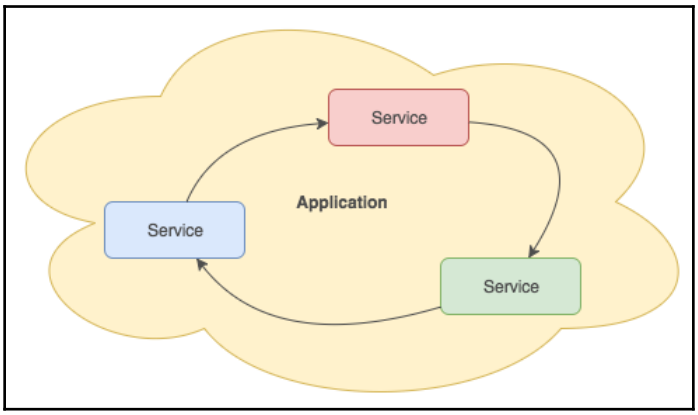
A commonly underestimated advantage of using this approach is that your application is much more resilient to failures. In a monolith application, any part can bring your application offline. In this microservices approach, this application might not send emails but can still be running and accessible. Add caching into the mixture and the API can restart in moments.

Patterns of microservices

Microservices architecture, like other archicectures, has a set of patterns that are easily identifiable and form the basis for this application development approach.

Some of these patterns can make the initial bootstrap a burden and can eventually be postponed. Others are essential from the beginning or you will have difficulty, later on, in migrating to a full microservice approach.

The following patterns are not an extensive list but they represent a solid foundation:



An example of services working together to form an application

Decomposable

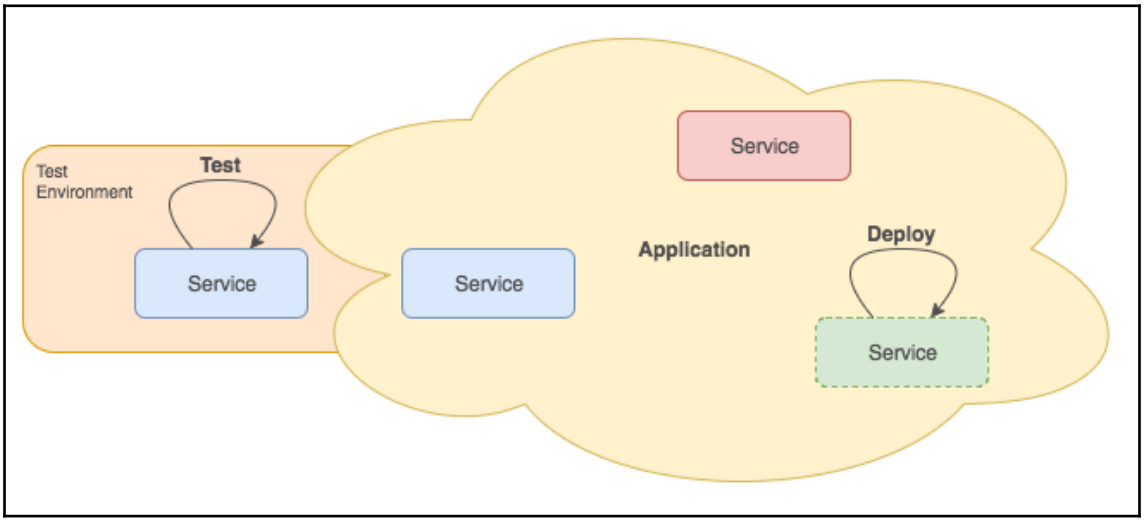
The main pattern behind a microservice architecture is the ability to have loosely coupled services. These services are decomposed, separated into smaller parts. This decomposition should create a set of services that implement a set of strongly related functions.

Each service should be small but complete, meaning it should run a set of functions in a given context. Those functions should represent all the functions you need or need to support for that context. What this means is that if you have a service that handles meeting events, all meeting event functions should be done using that service, whether it's creating an event, changing, removing, or getting information about a specific event. This ensures that an implementation change to events will affect that service only.

Decomposing an application can take one of two main approaches:

- By capability, when a service has a specific power or set of powers, such as sending emails, regardless of its content
- By subdomain, when a service has the complete knowledge of a subdomain or module of your application domain

In our previous event application, a service that was decomposed by capability is, for example, the SMTP service. A service decomposed by the domain could be the API service, assuming the application only manages events:



An example of services being tested and deployed autonomously, instead of the whole application

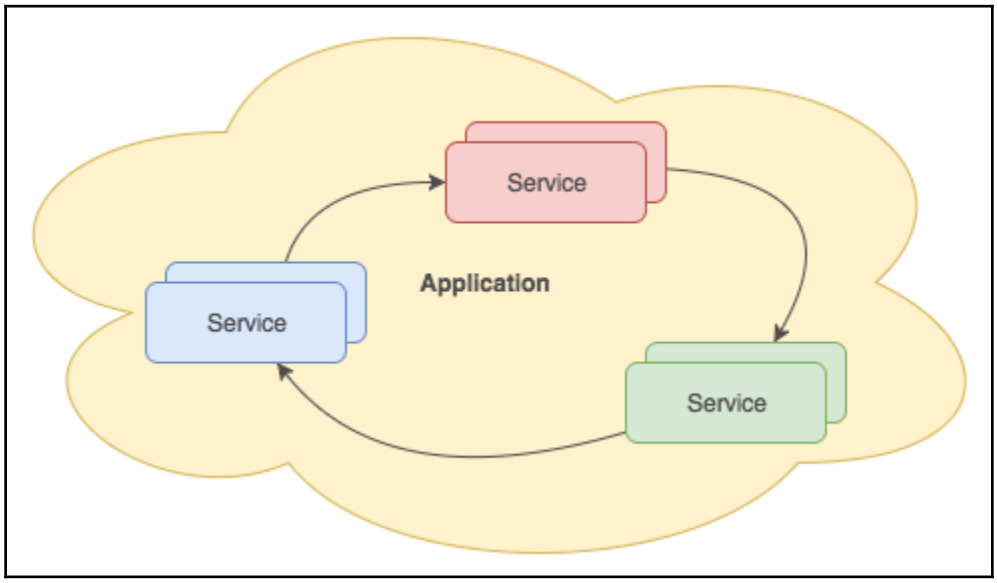
Autonomous

In a microservice architecture, each service should be autonomous. A small team should be able to run it without the other services that make your application. That team should also be able to develop autonomously and make changes to implementation without affecting the application.

The development team should be able to:

- Test, creating business logic and unit tests to ensure the service functions work as expected
- Deploy, upgrading functionality, without restarting other services in the process

Services should be able to evolve regardless of others, keeping backward compatibility, adding new functions, and scaling to several locations, with minimal changes to the architecture:



An example of an application with two instances per service, making it fault-tolerant

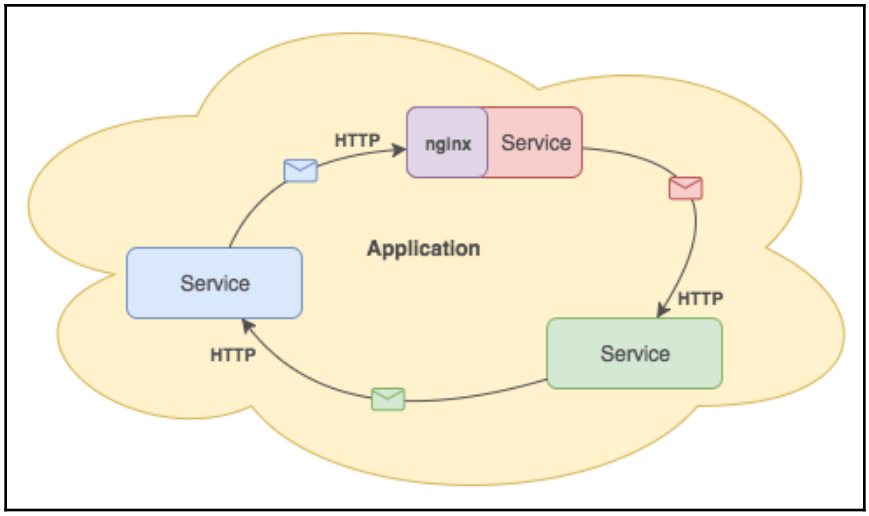
Scalable

A service should be scalable. At least two instances should be able to run in parallel, enabling failure tolerance and maintenance downtime. A service can also, later on, scale geographically, be near your customers, and improve apparent performance and application response.

For this scaling to be effective, the application platform will need service discovery and routing, a service that could be used by other services to register themselves and expose their capabilities. Other services could, later on, inquire this service directory and know how to reach these capabilities.

To reduce complexity for other services, a service router can redirect requests to service instances. For example, to send emails, you could have three instances and one central router that would redirect requests in a round-robin manner. If any of those instances go offline, the router will stop redirecting to it and the rest of the application doesn't need to care about it.

Another approach could be to use a DNS approach. The name service is capable of handling registrations to a subdomain, and then, when another service makes simple requests, it will receive one or all of the addresses and connect it as if there was only one service operating:



An example of communication between services of an application

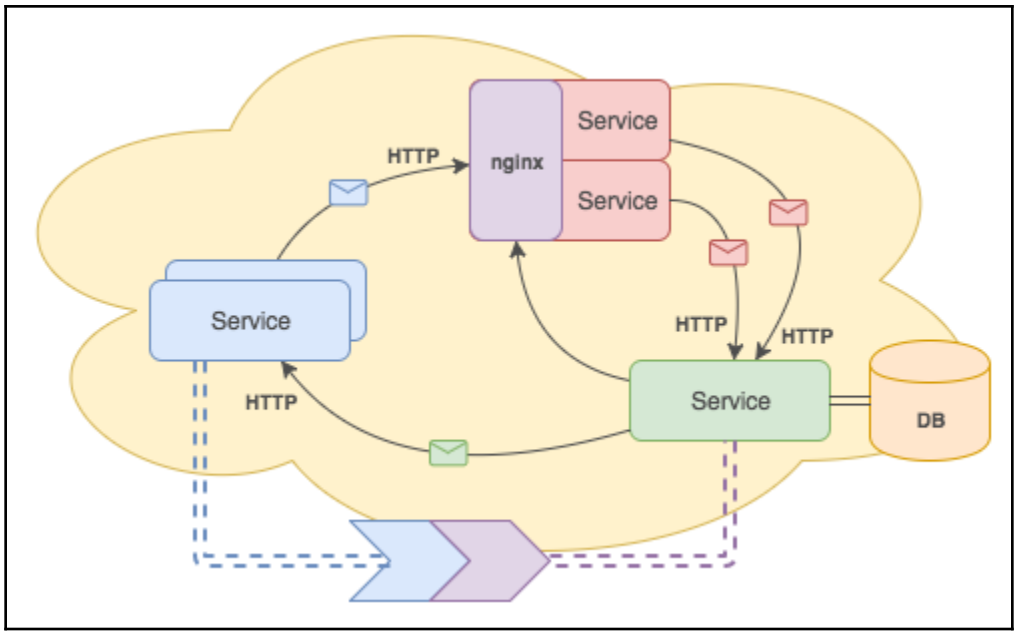
Communicable

Usually, services communicate over HTTP using a REST-compliant API. This is not a pattern that you must follow, but it's something that comes naturally based on how common HTTP is nowadays, making it an obvious choice.

There are plenty of HTTP servers out there, making it easy to expose a non-HTTP service with minimum effort.

HTTP is also a mature communication transport layer. It's a stateless protocol, giving developers and operations many features, such as:

- Caching commonly used and often updated resources
- Proxying and routing requests
- Securing communication over TLS



A complex application with several services and streams of communication

Disadvantages of microservices

Microservices have a lot of advantages, and they become more evident in the long term when the application becomes more complex. The microservice patterns primarily introduce complexity and enforce you to be very strict around development in order to avoid losing track of the entire application.

To start with, developers add to the complexity of the distributed system. This distributed system enforces a network communication -- not only do the services need to communicate with each other; they also need to find one another using some kind of service discovery technique.

This adds the need for a stable network to exchange requests. It adds layers of complexity just to exchange a simple request. It will also add an extra layer of security, both for development that needs to support it and for the operations that need to address it. As has been said previously, using bootstrap is harder than using a monolith application.

The operations team can also see some complexity in deployment. It's better to look at several services as separate components for deployment. This is actually as they should look for this architecture. They can, in the long term, even use some of the deployed architecture for new applications. Still, complexity for operations deployment and monitoring is even bigger.

Although not a direct disadvantage, because the architecture is loosely coupled and each service can be built using the best tools for the job, this can increase the heterogeneous environment, which will require operations to have knowledge of a broader technology scope. This can potentially lead to mistakes. Try to look for the second-best tool for the job if that tool is already used in operations.

The operations team will also have to possibly manage more third-party services, such as message queues, as they're commonly used to enable service communication. Other added services that might be added for scalability are service discovery. In a fully scalable application, operations just deploy new services and they register themselves and get used by all of your application's ecosystem.

As you can see, although development might get easier and you'll be able to parallel the application development across teams, there's also an added complexity if you intend to fully scale your application.

As a recommendation, do not plan your application to scale to millions of requests unless you really know it will handle that amount in the near future. Plan your application to have small and lightweight services to start with. It will be easier to do than upgrade the services to scale.

As a possible macro strategy, split your application using these three guidelines:

- Split services by capabilities
- Try to keep subdomains on a single service
- Prepare for scale, but don't scale while there's no need to

Summary

In conclusion, microservices architecture is a good, clear pattern that helps tackle more complex projects. In the long term, it reduces the complexity associated with new projects by appealing to service reuse. It helps to structure an application into loosely coupled services that can be independently developed and tested by small, different teams. It comes at the cost of initial proper planning and a more complex deployment.

We shall now take a look at the recipes mentioned in the next chapter that would explain in much more detail on how to make the transition from monolith to microservices

2
Breaking the Monolith

In this chapter, we will cover the following recipes:

- Organizing your team to embrace microservices
- Decomposing by business capability
- Identifying bounded contexts
- Migrating data in production
- Refactoring your monolith
- Evolving your monolith into services
- Evolving your test suite
- Using Docker for local development
- Routing requests to services

Introduction

One of the hardest things about microservices is getting started. Many teams have found themselves building features into an ever-growing, hard-to-manage monolithic code base and don't know how to start breaking it apart into more manageable, separately deployable services. The recipes in this chapter will explain how to make the transition from monolith to microservices. Many of the recipes will involve no code whatsoever; instead, they will be focused on architectural design and how best to structure teams to work on microservices.

You'll learn how to begin moving from a single monolithic code base to suites of microservices. You'll also learn how to manage some of the initial challenges when you begin to develop features using this new architectural style.

Organizing your team

Conway's law tells us that organizations will produce designs whose structure is a copy of their communication structure. This often means that the organizational chart of an engineering team will have a profound impact on the structure of the designs of the software it produces. When a new startup begins building software, the team is small—sometimes it is comprised of just one or two engineers. In this setup, engineers work on everything, including frontend and backend systems, as well as operations. Monoliths suit this organizational structure very well, allowing engineers to work on any part of the system at any given time without moving between code bases.

As a team grows, and you start to consider the benefits of microservices, you can consider employing a technique commonly referred to as an the **Inverse Conway Maneuver**. This technique recommends evolving your team and organizational structure to encourage the kind of architectural style you want to see emerge. With regard to microservices, this will usually involve organizing engineers into small teams that you will eventually want to be responsible for a handful of related services. Setting your team up for this structure ahead of time can motivate engineers to build services by limiting communication and decision-making overhead within the team. Simply put, monoliths continue to exist when the cost of adding features as services is greater than the cost of adding a feature to the monolith. Organizing your teams in this way reduces the cost of developing services.

This recipe is aimed at managers and other leaders in companies who have the influence to implement changes to the structure of the organization.

How to do it…

Re-organizing a team is never a simple task, and there are many non-obvious factors to consider. Factors such as personality, individual strengths and weaknesses, and past histories are outside the scope of this recipe, but they should be considered carefully when making any changes. The steps in this recipe provide one possible way to move a team from being organized around a monolithic code base to being optimized for microservices, but there is no one-size-fits-all recipe for every organization.

Use the following steps as a guide if you think they apply, but otherwise use them for inspiration and to encourage thought and discussion:

1. Working with other stakeholders in your organization, build out a product roadmap. You may have limited information about the challenges your organization will face in the short term, but do the best you can. It's perfectly natural to be very detailed for short-term items on a roadmap and very general for the longer term.
2. Using the product roadmap, try to identify technical capabilities that will be required to help you deliver value to your users. For example, you may be planning to work on a feature that relies heavily on search. You may also have a number of features that rely on content uploading and management. This means that search and uploading are two technical capabilities you know you will need to invest in.
3. As you see patterns emerge, try to identify the main functional areas of your application, paying attention to how much work you anticipate will go into each area. Assign higher priorities to the functional areas you anticipate will need a lot of investment in the short to medium term.
4. Create new teams, ideally consisting of four to six engineers, who are responsible for one of the functional areas within your application. Start with the functional areas that you anticipate will require the most work over the next quarter or so. These teams can be focused on the backend services or they can be cross-functional teams that include the mobile and web engineers. The benefit of having cross-functional teams is that the team can then deliver the entire vertical component of the application autonomously. The combination of service engineers with engineers consuming their services will also enable more information sharing, and hopefully, empathy.

Discussion

Using this approach, you should end up with small, cohesive, and focused teams responsible for core areas of your application. The nature of teams is that individuals within the team should start to see the benefit of creating separately managed and deployed code bases that they can work in autonomously without the costly overhead of coordinating changes and deployments with other teams.

To help illustrate these steps, imagine your organization builds an image-messaging application. The application allows users to take a photo with their smart phone and send it, along with a message, to a friend in their contacts list. Their friends can also send them photos with messages. A fictional roadmap for this fictional product could involve the need to add support for short videos, photo filters, and support for emojis. You now know that the ability to record, upload, and play videos, the ability to apply photo filters, and the ability to send rich text will be important to your organization. Additionally, you know from experience that users need to register, log in, and maintain a friends list.

Using the preceding example, you may decide to organize engineers into a media team, responsible for uploading, processing and playing, filters, and storage and delivery, a messaging team, responsible for the sending of photo or video messages with associated text, and a users team, responsible for providing reliable authentication, registration, on-boarding, and social features.

Decomposing by business capability

In the early stages of product development, monoliths are the best suited to delivering features to users as quickly and simply as possible. This is appropriate, as at this point in a products development you do not have luxury problems of having to scale your teams, code bases or ability to serve customer traffic. Following good design practices, you separate your applications concerns into easy-to-read, modular code patterns. Doing so allows engineers to work on different sections of the code autonomously and limits the possibility of having to untangle complicated merge conflicts when it comes time to merge your branch into the master and deploy your code.

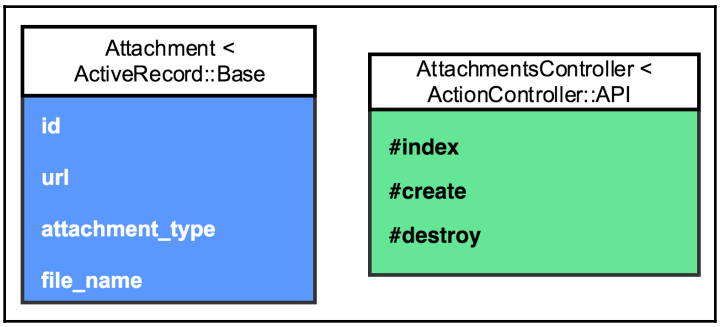
Microservices require you to go a step further than the good design practices you've already been following in your monolith. To organize your small, autonomous teams around microservices, you should consider first identifying the core business capabilities that your application provides. Business capability is a business school term that describes the various ways your organization produces value. For example, your internal order management is responsible for processing customer orders. If you have a social application that allows users to submit user-generated content such as photos, your photo upload system provides a business capability.

When thinking about system design, business capabilities are closely related to the **Single Responsibility Principle** (**SRP**) from **object-oriented design** (**OOD**). Microservices are essentially SRP extended to code bases. Thinking about this will help you design appropriately sized microservices. Services should have one primary job and they should do it well. This could be storing and serving images, delivering messages, or creating and authenticating user accounts.

How to do it...

Decomposing your monolith by business capability is a process. These steps can be carried out in parallel for each new service you identify a need for, but you may want to start with one service and apply the lessons you learn to subsequent efforts:

1. Identify a business capability that is currently provided by your monolith. This will be the target for our first service. Ideally this business capability is something that has some focus on the roadmap you worked on in the previous recipe and ownership can be given to one of your newly created teams. Let's use our fictional photo messaging service as an example and assume we'll start with the ability to upload and display media as our first identified business capability. This functionality is currently implemented as a single model and controller in your **Ruby on Rails** monolith:



2. In the preceding screenshot, **AttachmentsController** has four methods (called **actions** in Ruby on Rails lingo), which roughly correspond to the **create, retrieve, update, delete** (**CRUD**) operations you want to perform on an **Attachment** resource. We don't strictly need it, and so will omit the update action. This maps very nicely to a RESTful service, so you can design, implement, and deploy a microservice with the following API:

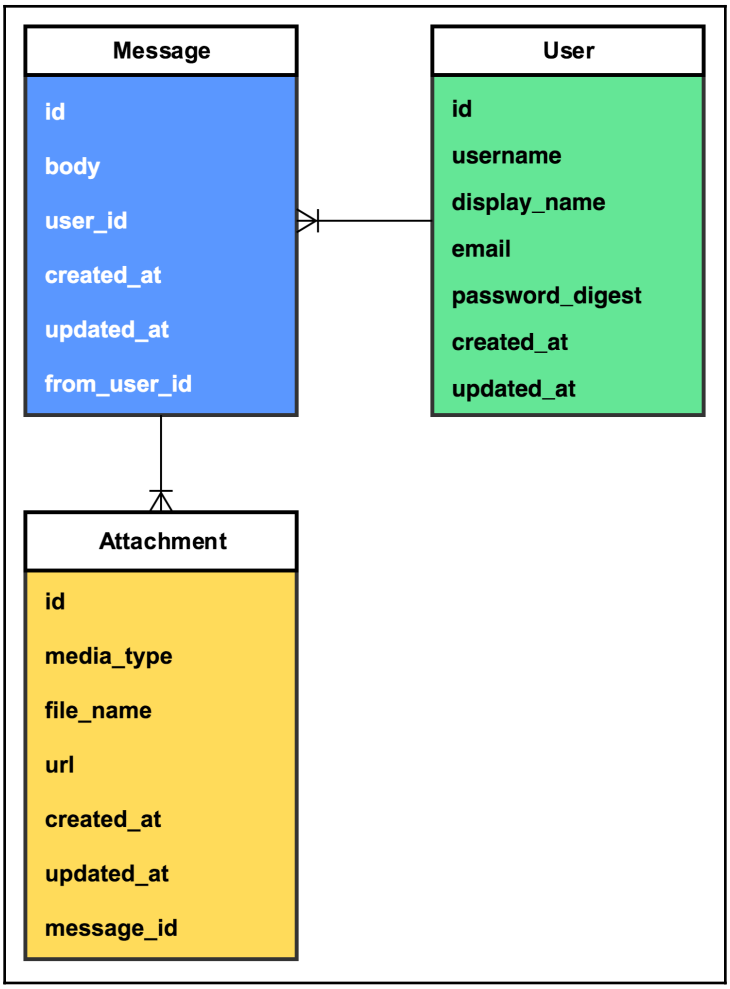
```
POST /attachments
GET /attachments/:id
DELETE /attachments/:id
```

3. With the new microservice deployed (migrating data is discussed in a later recipe), you can now begin modifying client code paths to use the new service. You can begin by replacing the code in the **AttachmentsController** action's methods to make an HTTP request to our new microservice. Techniques for doing this are covered in the *Evolving your monolith into services* recipe later in this chapter.

Identifying bounded contexts

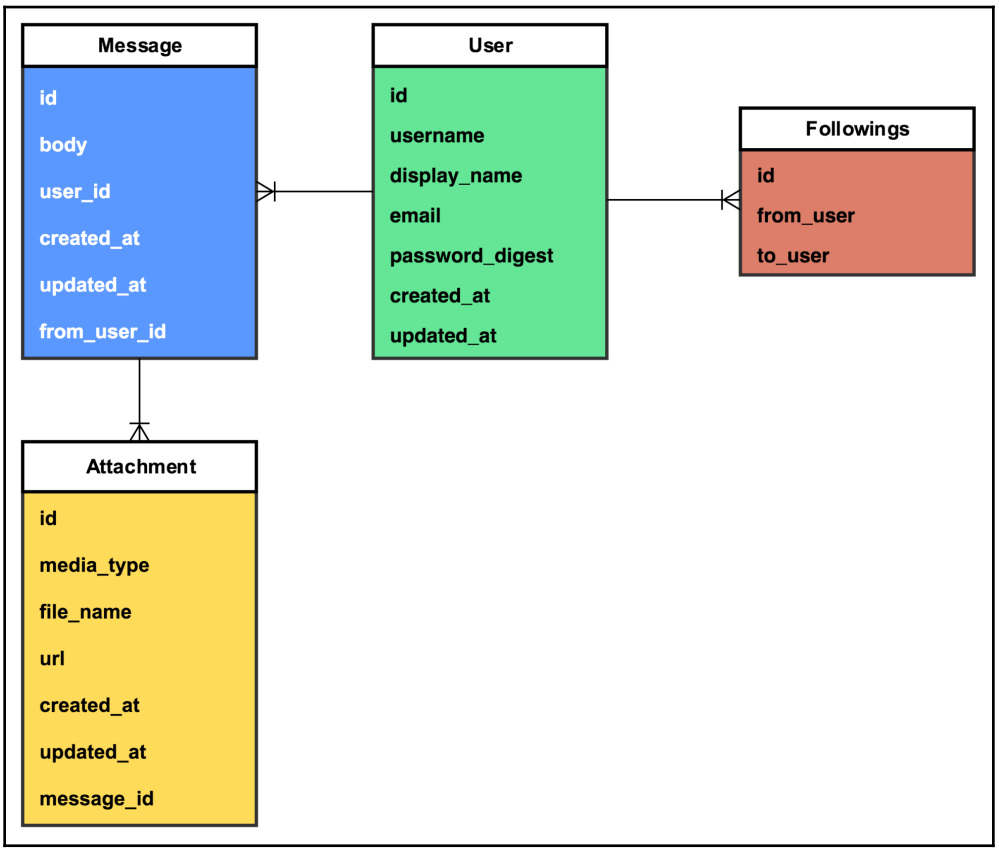
When designing microservices, a common point of confusion is how big or small a service should be. This confusion can lead engineers to focus on things such as the number of lines of code in a particular service. Lines of code are an awful metric for measuring software; it's much more useful to focus on the role that a service plays, both in terms of the business capability it provides and the domain objects it helps manage. We want to design services that have low coupling with other services, because this limits what we have to change when introducing a new feature in our product or making changes to an existing one. We also want to give services a single responsibility.

When decomposing a monolith, it's often useful to look at the data model when deciding what services to extract. In our fictional image-messaging application, we can imagine the following data model:



We have a table for messages, a table for users, and a table for attachments. The **Message** entity has a one-to-many relationship with the **User** entity; every user can have many messages that originate from or are targeted at them, and every message can have multiple attachments. What happens as the application evolves and we add more features? The preceding data model does not include anything about social graphs. Let's imagine that we want a user to be able to follow other users. We'll define the following as a asymmetric relationship, just because user 1 follows user 2, that does not mean that user 2 follows user 1.

There are a number of ways to model this kind of relationship; we'll focus on one of the simplest, which is an adjacency list. Take a look at the following diagram:



We now have an entity, **Followings**, to represent a follow relationship between two users. This works perfectly in our monolith, but introduces a challenge with microservices. If we were to build two new services, one to handle attachments, and another to handle the social graph (two distinct responsibilities), we now have two definitions of the user. This duplication of models is often necessary. The alternative is to have multiple services access and make updates to the same model, which is extremely brittle and can quickly lead to unreliable code.

This is where bounded contexts can help. A bounded context is a term from **Domain-Driven Design** (**DDD**) and it defines the area of a system within which a particular model makes sense. In the preceding example, the social-graph service would have a **User** model whose bounded context would be the users social graph (easy enough). The media service would have a **User** model whose bounded context would be photos and videos. Identifying these bounded contexts is important, especially when deconstructing a monolith—you'll often find that as a monolithic code base grows, the previously discussed business capabilities (uploading and viewing photos and videos, and user relationships) would probably end up sharing the same, bloated **User** model, which will then have to be untangled. This can be a tricky but enlightening and important process.

How to do it...

Deciding on how to define bounded contexts within a system can be a rewarding endeavor. The process itself encourages teams to have many interesting discussions about the models in a system and the various interactions that must happen between various systems:

1. Before a team can start to define the bounded contexts it works with, it should first start listing the models that are owned by the parts of the system it works on. For example, the media team will obviously own the **Attachment** model, but it will also need to have information about users, and messages. The **Attachment** model may be entirely maintained within the context of the media teams services, but the others will have to have a well-defined bounded context that can be communicated to other teams if necessary.
2. Once a team has identified potentially shared models, it's a good idea to have a discussion with other teams that use similar models or the same model.
3. In those discussions, hammer out the boundaries of the model and decide whether it makes sense to share a model implementation (which in a microservice world would necessitate a service-to-service call) or go their separate ways and develop and maintain separate model implementations. If the choice is made to develop separate model implementations, it'll become important to clearly define the bounded context within which the model applies.
4. The team should document clear boundaries in terms of teams, specific parts of the application, or specific code bases that should make use of the model.

Migrating data in production

Monolith code bases usually use a primary relational database for persistence. Modern web frameworks are often packaged with **object-relational mapping** (**ORM**), which allows you to define your domain objects using classes that correspond to tables in the database. Instances of these model classes correspond to rows in the table. As monolith code bases grow, it's not uncommon to see additional data stores, such as document or key value stores, be added.

Microservices should not share access with the same database your monolith connects to. Doing so will inevitably cause problems when trying to coordinate data migrations, such as schema changes. Even schema-less stores will cause problems when you change the way data is written in one code base but not how data is read in another code base. For this and other reasons, it's best to have microservices fully manage the data stores they use for persistence.

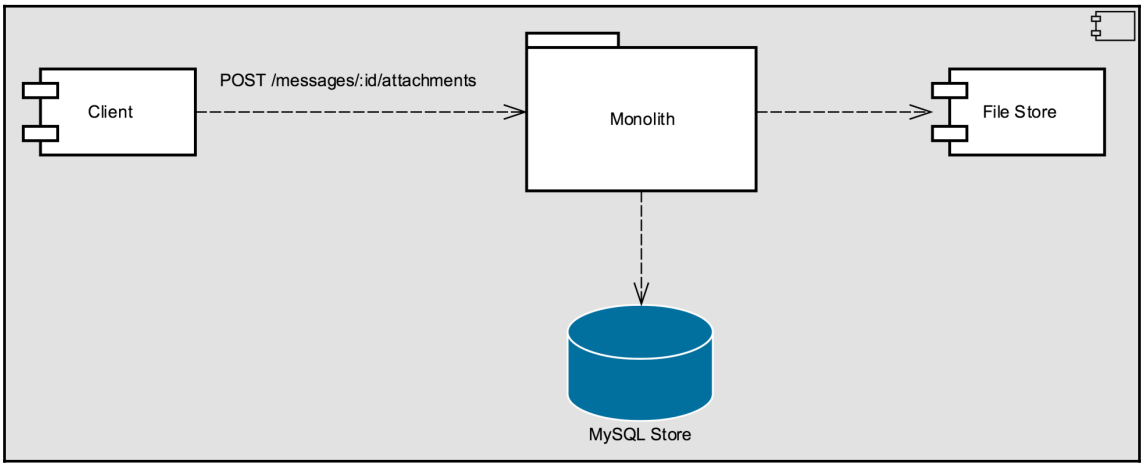
When transitioning from a monolith to microservices, it's important to have a strategy for how to migrate data. All too often, a team will extract the code for a microservice and leave the data, setting themselves up for future pain. In addition to difficulty managing migrations, a failure in the monolith relational database will now have cascading impacts on services, leading to difficult-to-debug production incidents.

One popular technique for managing large-scale data migrations is to set up dual writing. When your new service is deployed, you'll have two write paths–one from the original monolith code base to its database and one from your new service to its own data store. Make sure that writes go to both of these code paths. You'll now be replicating data from the moment your new service goes into production, allowing you to backfill older data using a script or a similar offline task. Once data is being written to both data stores, you can now modify all of your various read paths. Wherever the code is used to query the monolith database directly, replace the query with a call to your new service. Once all read paths have been modified, remove any write paths that are still writing to the old location. Now you can delete the old data (you have backups, right?).

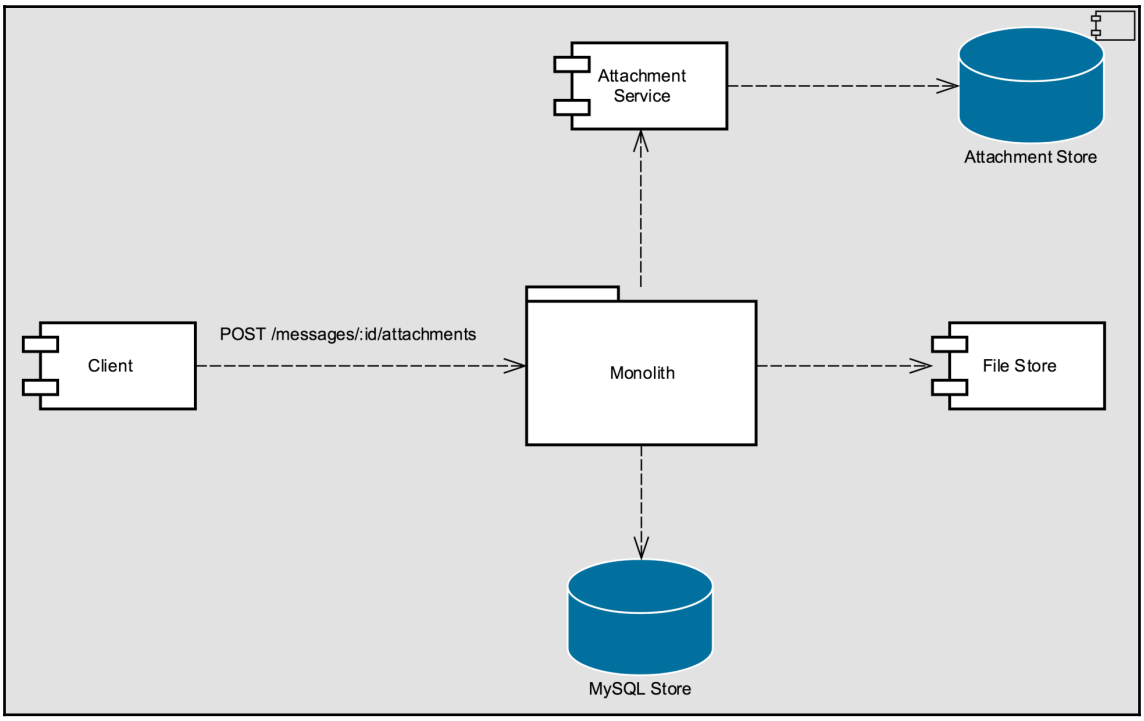
How to do it...

Migrating data from a monolith database to a new store fronted by a new service, without any impact on availability or consistency, is a difficult but common task when making the transition to microservices. Using our fictional photo-messaging application, we can imagine a scenario where we want to create a new microservice responsible for handling media uploads. In this scenario, we'd follow a common dual-writing pattern:

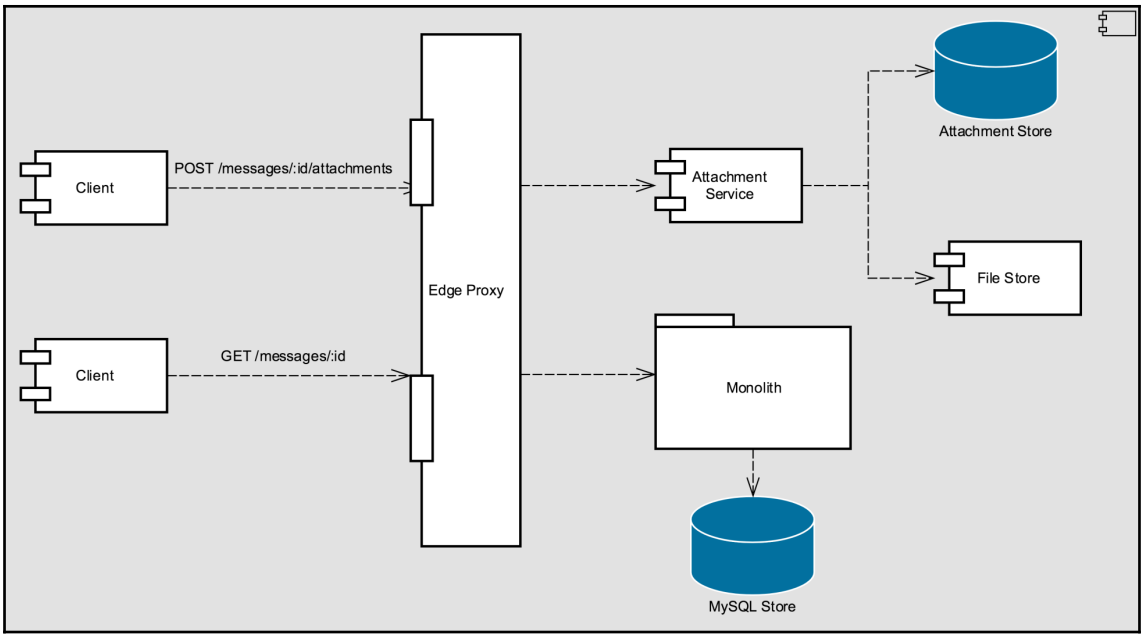
1. Before writing a new service to handle media uploads, we'll assume that the monolith architecture looks something like the following diagram, where HTTP requests are being handled by the monolith, which presumably reads the multipart/form-encoded content body as a binary object and stores the file in a distributed file store (Amazon's S3 service, for example). Metadata about the file is then written to a database table, called **attachments**, as shown in the following diagram:



2. After writing a new service, you now have two write paths. In the write path in the monolith, make a call to your service so that you're replicating the data in the monolith database as well as the database fronted by your new service. You're now duplicating new data and can write a script to backfill older data. Your architecture now looks something like this:



3. Find all read paths in your **Client** and **Monolith** code, and update them to use your new service. All reads will now be going to your service, which will be able to give consistent results.
4. Find all write paths in your **Client** and **Monolith** code, and update them to use your new service. All reads and writes are now going to your service, and you can safely delete old data and code paths. Your final architecture should look something like the following (we'll discuss edge proxies in later chapters):



Using this approach, you'll be able to safely migrate data from a monolith database to a new store created for a new microservice without the need for downtime. It's important not to skip this step; otherwise, you won't truly realize the benefits of microservice architectures (although, arguably, you'll experience all the downsides!).

Refactoring your monolith

A common mistake when making the transition to microservices is to ignore the monolith and just build new features as services. This usually happens when a team feels that the monolith has gotten so out of control, and the code so unwieldy, that it would be better to declare bankruptcy and leave it to rot. This can be especially tempting because the idea of building green field code with no legacy baggage sounds a lot nicer than refactoring brittle, legacy code.

Resist the temptation to abandon your monolith. To successfully decompose your monolith by business capability and start evolving it into a set of nicely factored, single-responsibility microservices, you'll need to make sure that your monolith code base is in good shape and is well factored, and well tested. Otherwise, you'll end up with a proliferation of new services that don't model your domain cleanly (because they overlap with functionality in the monolith), and you'll continue to have trouble working with any code that exists in your monolith. Your users won't be happy and your teams' energy will most likely start to decline as the weight of technical debt becomes unbearable.

Instead, take constant, proactive steps to refactor your monolith using good, solid design principles. Excellent books have been written on the subject of refactoring (I recommend *Refactoring* by Martin Fowler and *Working Effectively with Legacy Code* by Michael Feathers), but the most important thing to know is that refactoring is never an all-or-nothing effort. Few product teams or companies will have the patience or luxury to wait while an engineering team stops the world and spends time making their code easier to change, and an engineering team that tries this will rarely be successful. Refactoring has to be a constant, steady process.

However your team schedules its work, make sure you're reserving an appropriate time for refactoring. A guiding principle is, whenever you go to make a change, first make the change easy to make, then make the change. Your goal is to make your monolith code easier to work with, easier to understand, and less brittle. You should also be able to develop a robust test suite that will come in handy.

Once your monolith is in better shape, you can start to continuously shrink the monolith as you factor out services. Another aspect of most monolith code bases is serving dynamically generated views and static assets served through browsers. If your monolith is responsible for this, consider moving your web application component into a separately served JavaScript application. This will allow you to shrink your monolith from multiple directions.

How to do it...

Refactoring any code base is a process. For monoliths, there are a few techniques that can work quite well. In this example, we'll document the steps that can be taken to make refactoring a Ruby on Rails code base easy:

1. Using the techniques described in previous recipes, identify business capabilities and bounded contexts within your application. Let's focus on the ability to upload pictures and videos.

2. Create a directory called `app/services` alongside `controllers`, `models`, and `views`. This directory will hold all of your service objects. Service objects are a pattern used in many Rails applications to factor out a conceptual service into a ruby object that does not inherit any Ruby on Rails functionality. This will make it easier to move the functionality encapsulated within a service object into a separate microservice. There is no one way to structure your service objects. I prefer to have each object represent a service, and move operations I want that service to be responsible for to that service object as methods.
3. Create a new file called `attachments_service.rb` under `app/services` and give it the following definition:

```
class AttachmentsService

  def upload
    # ...
  end

  def delete!
    # ...
  end

end
```

4. Looking at the source code for the `AttachmentsController#create` method in the `app/controllers/attachments_controller.rb` file, it currently handles the responsibility for creating the `Attachment` instance and uploading the file data to the attachment store, which in this case is an Amazon S3 bucket. This is the functionality that we need to move to the newly created service object:

```
# POST /messages/:message_id/attachments
def create
  message = Message.find_by!(params[:message_id], user_id:
  current_user.id)
  file = StorageBucket.files.create(
    key:  params[:file][:name],
    body: StringIO.new(Base64.decode64(params[:file][:data]),
    'rb'),
    public: true
  )
  attachment = Attachment.new(attachment_params.merge!(message:
  message))
  attachment.url = file.public_url
  attachment.file_name = params[:file][:name]
```

```
    attachment.save
    json_response({ url: attachment.url }, :created)
  end
```

5. Open the newly created service object in the `app/services/attachments_service.rb` file and move the responsibility for uploading the file to the `AttachmentsService#upload` method:

```
class AttachmentsService

  def upload(message_id, user_id, file_name, data, media_type)
    message = Message.find_by!(message_id, user_id: user_id)
    file = StorageBucket.files.create(
      key:  file_name,
      body: StringIO.new(Base64.decode64(data), 'rb'),
      public: true
    )
    Attachment.create(
      media_type: media_type,
      file_name:  file_name,
      url:        file.public_url,
      message:    message
    )
  end

  def delete!
  end
end
```

6. Now upload the `AttachmentsController#create` method in `app/controllers/attachments_controller.rb` to use the newly created `AttachmentsService#upload` method:

```
# POST /messages/:message_id/attachments
def create
  service = AttachmentService.new
  attachment = service.upload(params[:message_id], current_user.id,
   params[:file][:name], params[:file][:data],
   params[:media_type])
  json_response({ url: attachment.url }, :created)
end
```

7. Repeat this process for code in the `AttachmentsController#destroy` method, moving the responsibility to the new service object. When you're finished, no code in `AttachmentsController` should be interacting with the `Attachments` model directly; instead, it should be going through the `AttachmentsService` service object.

You've now isolated responsibility for the management of attachments to a single service class. This class should encapsulate all of the business logic that will eventually be moved to a new attachment service.

Evolving your monolith into services

One of the most complicated aspects of transitioning from a monolith to services can be request routing. In later recipes and chapters, we'll explore the topic of exposing your services to the internet so that the mobile and web client applications can communicate directly with them. For now, however, having your monolith act as a router can serve as a useful intermediary step.

As you break your monolith into small, maintainable microservices, you can replace code paths in your monolith with calls to your services. Depending on the programming language or framework you used to build your monolith, these sections of code can be called controller actions, views, or something else. We'll continue to assume that your monolith was built in the popular Ruby on Rails framework; in which case, we'll be looking at controller actions. We'll also assume that you've begun refactoring your monolith and have created one or more service objects as described in the previous recipe.

It's important when doing this to follow best practices. In later chapters, we'll introduce concepts, such as circuit breakers, that become important when doing service-to-service communication. For now, be mindful that HTTP calls from your monolith to a service could fail, and you should consider how best to handle that kind of situation.

How to do it...

1. Open the service object we created in the previous recipe. We'll modify the service object to be able to call an external microservice responsible for managing attachments. For the sake of simplicity, we'll use an HTTP client that is provided in the Ruby standard library. The service object should be in the `app/services/attachments_service.rb` file:

```
class AttachmentsService

  BASE_URI = "http://attachment-service.yourorg.example.com/"

  def upload(message_id, user_id, file_name, data, media_type)
    body = {
      user_id: user_id,
      file_name: file_name,
      data: StringIO.new(Base64.decode64(params[:file]
      [:data]), 'rb'),
      message: message_id,
      media_type: media_type
    }.to_json
    uri = URI("#{BASE_URI}attachment")
    headers = { "Content-Type" => "application/json" }
    Net::HTTP.post(uri, body, headers)
  end

end
```

2. Open the `attachments_controller.rb` file, located in `pichat/app/controllers/`, and look at the following create action. Because of the refactoring work done, we require only a small change to make the controller work with our new service object:

```
class AttachmentsController < ApplicationController
  # POST /messages/:message_id/attachments
  def create
    service = AttachmentService.new
    response = service.upload(params[:message_id], current_user.id,
     params[:file][:name], params[:file][:data],
     params[:media_type])
    json_response(response.body, response.code)
  end
  # ...
end
```

Evolving your test suite

Having a good test suite in the first place will help tremendously as you move from a monolith to microservices. Each time you remove functionality from your monolith code base, your tests will need to be updated. It's tempting to replace unit and functional tests in your Rails app with tests that make external network calls to your services, but this approach has a number of downsides. Tests that make external calls will be prone to failures caused by intermittent network connectivity issues and will take an enormous amount of time to run after a while.

Instead of making external network calls, you should modify your monolith tests to stub microservices. Tests that use stubs to represent calls to microservices will be less brittle and will run faster. As long as your microservices satisfy the API contracts you develop, the tests will be reliable indicators of your monolith code base's health. Making backwards-incompatible changes to your microservices is another topic that will be covered in a later recipe.

Getting ready

We'll use the `webmock` gem for stubbing out external HTTP requests in our tests, so update your monolith gemfile to include the `webmock` gem in the test group:

```
group :test do
  # ...
  gem 'webmock'
end
```

You should also update `spec/spec_helper.rb` to disable external network requests. That will keep you honest when writing the rest of your test code:

```
require 'webmock/rspec'
WebMock.disable_net_connect!(allow_localhost: false)
```

How to do it...

Now that you have `webmock` included in your project, you can start stubbing HTTP requests in your specs. Once again, open `specs/spec_helper.rb` and add the following content:

```
stub_request(:post, "attachment-service.yourorg.example.com").
  with(body: {media_type: 1}, headers: {"Content-Type" => /image\/.+/}).
  to_return(body: { foo: bar })
```

Using Docker for local development

As we've discussed, microservices solve a particular set of problems but introduce some new challenges of their own. One challenge that engineers on your team will probably run into is doing local development. With a monolith, there are fewer moving parts that have to be managed—usually, you can get away with just running a database and an application server on your workstation to get work done. As you start to create new microservices, however, the situation gets more complicated.

Containers are a great way to manage this complexity. Docker is a popular, open source software containerization platform. Docker allows you to specify how to run your application as a container—a lightweight standardized unit for deployment. There are plenty of books and online documentation about Docker, so we won't go into too much detail here, just know that a container encapsulates all of the information needed to run your application. As mentioned, a monolith application will often require an application server and a database server at a minimum—these will each run in their own container.

Docker Compose is a tool for running multicontainer applications. Compose allows you to define your applications containers in a YAML configuration file. Using the information in this file, you can then build and run your application. Compose will manage all of the various services defined in the configuration file in separate containers, allowing you to run a complex system on your workstation for local development.

Getting ready

Before you can follow the steps in this recipe, you'll need to install the required software:

1. Install Docker. Download the installation package from the Docker website (`https://www.docker.com/docker-mac`) and follow the instructions.
2. Install `docker-compose` by executing the following command line on macOS X:

```
brew install docker-compose
```

On Ubuntu Linux, you can execute the following command line:

```
apt-get install docker-compose
```

With those two packages installed, you'll be ready to follow the steps in this recipe.

How to do it...

1. In the root directory of your Rails application, create a single file called `Dockerfile` with the following contents:

```
FROM ruby:2.3.3
RUN apt-get update -qq && apt-get install -y build-essential
libpq-dev nodejs
RUN mkdir /pichat
WORKDIR /pichat
ADD Gemfile /pichat/Gemfile
ADD Gemfile.lock /pichat/Gemfile.lock
RUN bundle install
ADD . /pichat
```

2. Create a file called `docker-compose.yml` with the following contents:

```
version: '3'
services:
  db:
    image: mysql:5.6.34
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: root

  app:
    build: .
    environment:
```

```
  RAILS_ENV: development
command: bundle exec rails s -p 3000 -b '0.0.0.0'
volumes:
  - .:/pichat
ports:
  - "3000:3000"
depends_on:
  - db
```

3. Start your application by running the `docker-compose up app` command. You should be able to access your monolith by entering `http://localhost:3000/` in your browser. You can use this approach for new services that you write.

Routing requests to services

In previous recipes, we focused on having your monolith route requests to services. This technique is a good start since it requires no client changes to work. Your clients still make requests to your monolith and your monolith marshals the request to your microservices through its controller actions. At some point, however, to truly benefit from a microservices architecture, you'll want to remove the monolith from the critical path and allow your clients to make requests to your microservices. It's not uncommon for an engineer to expose their organization's first microservice to the internet directly, usually using a different hostname. However, this starts to become unmanageable as you develop more services and need a certain amount of consistency when it comes to monitoring, security, and reliability concerns.

Internet-facing systems face a number of challenges. They need to be able to handle a number of security concerns, rate limiting, periodic spikes in traffic, and so on. Doing this for each service you expose to the public internet will become very expensive, very quickly. Instead, you should consider having a single edge service that supports routing requests from the public internet to internal services. A good edge service should support common features, such as dynamic path rewriting, load shedding, and authentication. Luckily, there are a number of good open source edge service solutions. In this recipe, we'll use a Netflix project called **Zuul**.

How to do it...

1. Create a new Spring Boot service called `Edge Proxy` with a main class called `EdgeProxyApplication`.
2. Spring Cloud includes an embedded Zuul proxy. Enable it by adding the `@EnableZuulProxy` annotation to your `EdgeProxyApplication` class:

```
package com.packtpub.microservices;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy
@SpringBootApplication
public class EdgeProxyApplication {

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

}
```

3. Create a file called `application.properties` under `src/main/resources/` with the following contents:

```
zuul.routes.media.url=http://localhost:8090
ribbon.eureka.enabled=false
server.port=8080
```

   In the preceding code, it tells `zuul` to route requests to `/media` to a service running on port `8090`. We'll touch on that `eureka` option in later chapters when we discuss service discovery, for now just make sure it's set to `false`.

At this point, your service should be able to proxy requests to the appropriate service. You've just taken one of the biggest steps toward building a microservices architecture. Congratulations!

3
Edge Services

In this chapter, we will cover the following recipes:

- Controlling access to your service with an edge proxy server
- Extending your services with sidecars
- Using API Gateway to route requests to services
- Rate limiting with an edge proxy server
- Stopping cascading failure with Hystrix
- Using a service mesh to factor out shared concerns

Introduction

Now that you've had some experience breaking a monolith into microservices, you've seen that many of the challenges exist outside the monolith or service code bases themselves. Exposing your service to the internet, controlling routing, and building in resiliency are all concerns that can be addressed by what are commonly called **edge services**. These are services that exist at the edge of our architecture, generally handling requests from the public internet. Luckily, because many of these challenges are so common, open source projects exist to handle most of them for us. We'll use a lot of great open source software in this chapter.

With the recipes in this chapter, you'll learn how to use open source software to expose your services to the public internet, control routing, extend your service's functionality, and handle a number of common challenges when deploying and scaling microservices. You'll also learn about techniques for making client development against services easier and how to standardize the monitoring and observability of your microservice architecture.

Controlling access to your service with an edge proxy server

In `Chapter 2`, *Breaking the Monolith*, we modified a monolith code base to provide easy routing to our microservices. This approach works and requires little effort, making it an ideal intermediary step. Eventually, your monolith will become a bottleneck in the development and resiliency of your architecture. As you try to scale your service and build more microservices, your monolith will need to be updated and deployed every time you make an API change to your service. Additionally, your monolith will have to handle connections to your services and is probably not well-configured to handle edge concerns such as load shedding or circuit breaking. In the *Routing requests to services* recipe of `Chapter 2`, *Breaking the Monolith*, we introduced the concept of edge proxies. Using an edge proxy server to expose your service to the public internet allows you to factor out most of the shared concerns a publicly exposed service must address. Requirements such as request routing, load shedding, back pressure, and authentication can all be handled in a single edge proxy layer instead of being duplicated by every service you need to have exposed to the internet.

An edge proxy is a proxy server that sits on the edge of your infrastructure, providing access to internal services. You can think of an edge proxy as the "front door" to your internal service architecture—it allows clients on the internet to make requests to internal services you deploy. There are multiple open source edge proxies that have a robust feature set and community, so we don't have to write and maintain our own edge proxy server. One of the most popular open source edge proxy servers is called **Zuul** and is built by Netflix. Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security, and more. Zuul is packaged as a Java library. Services written in the Java framework Spring Boot can use an embedded Zuul service to provide edge-proxy functionality. In this recipe, we'll walk through building a small Zuul edge proxy and configuring it to route requests to our services.

Operational notes

Continuing with our example application from the previous chapter, imagine that our photo-messaging application (we'll call it `pichat` from now on) was originally implemented as a Ruby on Rails monolithic code base. When the product first launched, we deployed the application to Amazon Web Services behind a single **Elastic Load Balancer** (**ELB**). We created a single **Auto Scale Group** (**ASG**) for the monolith, called `pichat-asg`.

Each EC2 instance in our ASG is running NGINX, which handles requests for static files (images, JavaScript, CSS) and proxies requests to unicorns running on the same host that is serving our Rails application. SSL is terminated at the ELB, and HTTP requests are forwarded to NGINX. The ELB is accessed through the DNS `monolith.pichat-int.me` name from within the **Virtual Private Cloud** (**VPC**).

We've now created a single `attachment-service`, which handles videos and images attached to messages being sent through the platform. The `attachment-service` is written in Java, using the Spring Boot platform and is deployed in its own ASG, called `attachment-service-asg`, that has its own ELB. We've created a private DNS record, called `attachment-service.pichat-int.me`, that points to this ELB.

With this architecture and topology in mind, we now want to route requests from the public internet to our Rails application or our newly created attachment service, depending on the path.

How to do it...

1. To demonstrate using Zuul to route requests to services, we'll first create a basic Java application that will serve as our edge proxy service. The Java project Spring Cloud provides an embedded Zuul service, making it pretty simple to create a service that uses the `zuul` library. We'll start by creating a basic Java application. Create the `build.gradle` file with the following content:

```
group 'com.packtpub.microservices'
version '1.0-SNAPSHOT'

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:1.4.4.RELEASE"
        classpath "io.spring.gradle:dependency-management-
plugin:0.5.6.RELEASE"
    }
}

apply plugin: 'java'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'
```

```
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencyManagement {
    imports {
        mavenBom 'org.springframework.cloud:spring-cloud-
netflix:1.4.4.RELEASE'
    }
}

dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version: '1.4.4.RELEASE'
    compile group: 'org.springframework.cloud', name: 'spring-
cloud-starter-zuul'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

2. Create a single class called `EdgeProxyApplication`. This will serve as the entry point to our application:

```
package com.packtpub.microservices.ch02.edgeproxy;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy
@SpringBootApplication
public class EdgeProxyApplication {
    public static void main(String[] args) {
        SpringApplication.run(EdgeProxyApplication.class, args);
    }
}
```

3. Create a file called `application.yml` in the `src/main/resources` directory of your application. This file will specify your route configurations. In this example, we'll imagine that our monolith application can be accessed on the `monolith.pichat-int.me` internal host and we want to expose the `/signup` and `/auth/login` paths to the public internet:

```
zuul:
 routes:
  signup:
   path: /signup
   url: http://monolith.pichat-int.me
  auth:
   path: /auth/login
   url: http://monolith.pichat-int.me
```

4. Start the project with `./gradlew bootRun` and you should be able to access the `/signup` and `/auth/login` URLs, which will be proxied to our monolith application.
5. We want to expose the `attachment-service` URLs to the internet. The attachment service exposes the following endpoints:

```
POST / # Creates an attachment
GET / # Fetch attachments, can filter by message_id
DELETE /:attachment_id # Deletes the specified attachment
GET /:id # Get the specific attachment
```

6. We'll need to decide which paths we want to use in our public API. Modify `application.properties` to add the following entries:

```
zuul:
 routes:
  signup:
   path: /signup
   url: http://monolith.pichat-int.me
  auth:
   path: /auth/login
   url: http://monolith.pichat-int.me
  attachments:
   path: /attachments/**
   url: http://attachment-service.pichat-int.me
```

7. Now all requests to `/attachments/*` will be forwarded to the attachment service and signup, and `auth/login` will continue to be served by our monolith application.

8. We can test this by running our service locally and sending requests to `localhost:8080/signup`, `localhost:8080/auth/login`, and `localhost:8080/attachments/foo`. You should be able to see that requests are routed to the respected services. Of course, the service will respond with an error because `attachment-service.pichat-int.me` cannot be resolved, but this shows that the routing is working as expected:

```
$ curl -D - http://localhost:8080/attachments/foo
HTTP/1.1 500
X-Application-Context: application
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 27 Mar 2018 12:52:21 GMT
Connection: close

{"timestamp":1522155141889,"status":500,"error":"Internal Server
Error","exception":"com.netflix.zuul.exception.ZuulException","mess
age":"attachment-service.pichat-int.me"}%
```

Extending your services with sidecars

When you start developing microservices, it's common to embed a certain amount of boilerplate into each service. Logging, metrics, and configuration are all functionalities that are commonly copied from service to service, resulting in a large amount of boilerplate and copied and pasted code. As your architecture grows and you develop more services, this kind of setup becomes harder and harder to maintain. The usual result is that you end up with a bunch of different ways of doing logging, metrics, service discovery, and so on, which results in systems that are hard to debug and maintain. Changing something as simple as a metrics namespace or adding a feature to your service discovery clients can require the coordination of multiple teams and code bases. More realistically, your microservices architecture will continue to grow with inconsistent logging, metrics, and service discovery conventions, making it harder for developers to operate, contributing to overall operational pain.

The sidecar pattern describes a pattern whereby you extend the functionality of a service with a separate process or container running on the same machine. Common functionalities, such as metrics, logging, service discovery, configuration, or even network RPC, can be factored out of your application and handled by a sidecar service running alongside it. This pattern makes it easy to standardize shared concerns within your architecture by implementing them in a separate process that can be used by all of your services.

A common method for implementing a sidecar is to build a small, separate process that exposes some functionality over a commonly used protocol, such as HTTP. Imagine, for instance, that you want all of your services to use a centralized service-discovery service instead of relying on DNS hosts and ports to be set in each application's configuration. To accomplish this, you'd need to have up-to-date client libraries for your service-discovery service available in all of the languages that your services and monolith are written in. A better way would be to run a sidecar parallel to each service that runs a service-discovery client. Your services could then proxy requests to the sidecar and have it determine where to send them. As an added benefit, you could configure the sidecar to emit consistent metrics around network RPC requests made between services.

This is such a common pattern that there are multiple open source solutions available for it. In this recipe, we'll use `spring-cloud-netflix-sidecar`, a project that includes a simple HTTP API that allows non-JVM applications to use JVM client libraries. The Netflix sidecar assumes you are using Eureka, a service registry designed to support the service-discovery needs of clients. We'll discuss service discovery in more detail in later chapters. The sidecar also assumes your non-JVM application is serving a health-check endpoint and will use this to advertise its health to Eureka. Our Rails application exposes such an endpoint, /health, which, when running normally, will return a small JSON payload with a key status and the `UP` value.

How to do it...

1. Start by creating a basic Spring Boot service. Include the Spring Boot Gradle plugin and add dependencies for Spring Boot and the Spring Cloud Netflix sidecar project:

```
group 'com.packtpub.microservices'
version '1.0-SNAPSHOT'

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.springframework.boot:spring-boot-gradle-
plugin:1.4.4.RELEASE"
        classpath "io.spring.gradle:dependency-management-
plugin:0.5.6.RELEASE"
    }
}

apply plugin: 'java'
```

```
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencyManagement {
    imports {
        mavenBom 'org.springframework.cloud:spring-cloud-
netflix:1.4.4.RELEASE'
    }
}

dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version: '1.4.4.RELEASE'
    compile group: 'org.springframework.cloud', name: 'spring-
cloud-netflix-sidecar', version: '1.4.4.RELEASE'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

2. We're ready to create a simple Spring Boot application. We'll use the `@EnableSidecar` annotation, which also includes the `@EnableZuulProxy`, `@EnableCircuitBreaker`, and `@EnableDiscoveryClient` annotations:

```
package com.packtpub.microservices;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.cloud.netflix.sidecar.EnableSidecar;
import org.springframework.stereotype.Controller;

@EnableSidecar
@Controller
@EnableAutoConfiguration
public class SidecarController {
    public static void main(String[] args) {
        SpringApplication.run(SidecarController.class, args);
    }
}
```

3. The Netflix sidecar application requires a few configuration settings to be present. Create a new file called `application.yml` with the following content:

```
server:
 port: 5678

sidecar:
 port: 3000
 health-uri: http://localhost:3000/health
```

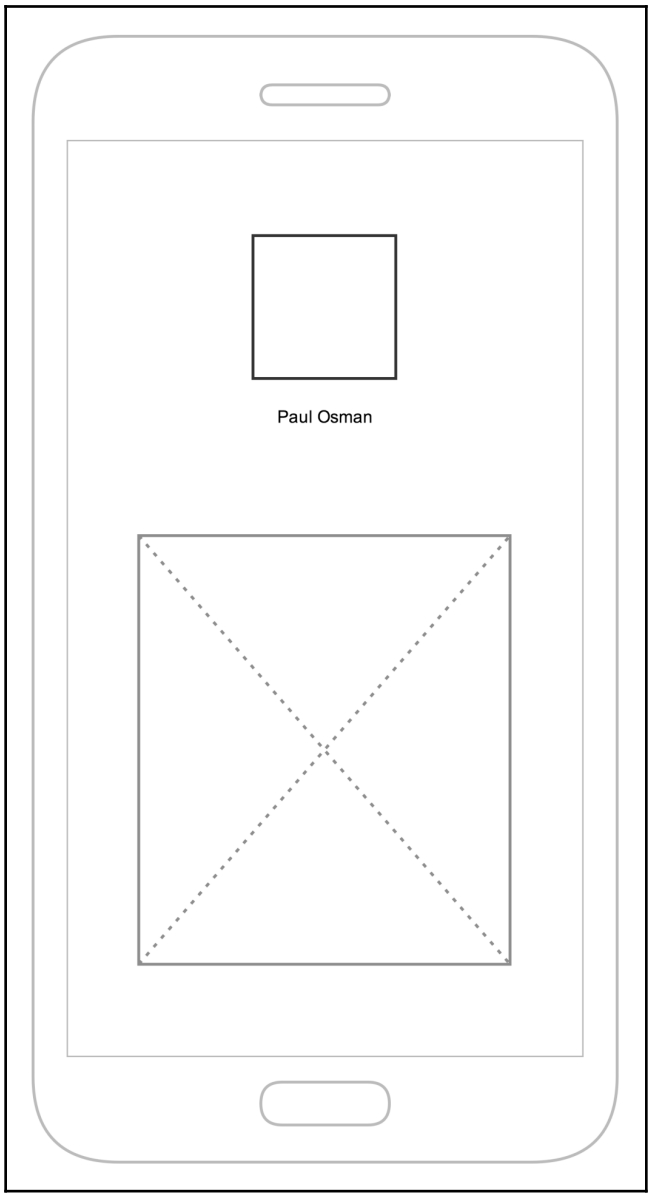
4. The sidecar will now expose an API that allows non-JVM applications to locate services registered with Eureka. If our `attachment-service` is registered with Eureka, the sidecar will proxy requests to `http://localhost:5678/attachment/1234` to `http://attachment-service.pichat-int.me/1234`.

Using API Gateways for routing requests to services

As we've seen in other recipes, microservices should provide a specific business capability and should be designed around one or more domain concepts, surrounded by a bounded context. This approach to designing service boundaries works well to guide you toward simple, independently-scalable services that can be managed and deployed by a single team dedicated to a certain area of your application or business.

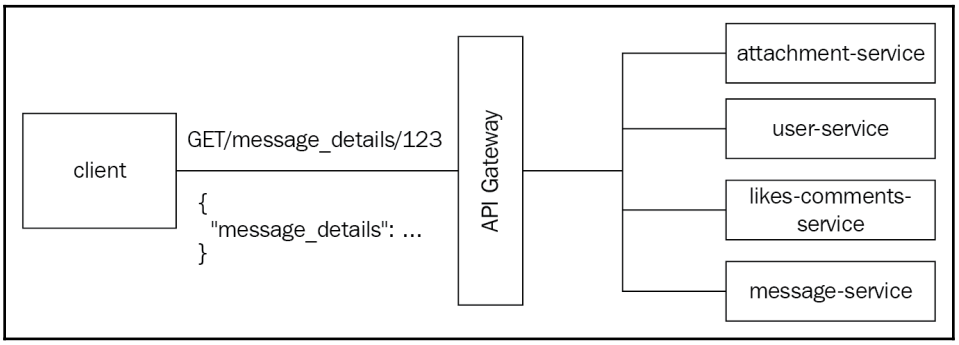
When designing user interfaces, clients often aggregate related but distinct entities from various backend microservices. In our fictional messaging application, for instance, the screen that shows an actual message might have information from a message service, a media service, a likes service, a comments service, and so on. All of this information can be cumbersome to collect and can result in a large number of round-trip requests to the backend.

Porting a web application from a monolith with server-side-rendered HTML to a single-page JavaScript application, for example, can easily result in hundreds of `XMLHttpRequests` for a single page load:



To reduce the amount of round-trip requests to the backend services, consider creating one or more API Gateways that provide an API that is catered to the client's needs. API Gateways can be used to present a single view of backend entities in a way that makes it easier for clients who use the API. In the preceding example, a request to a single message endpoint could return information about the message itself, media included in the message, likes and comments, and other information.

These entities can be concurrently collected from various backend services using a fan-out request pattern:



Design considerations

One of the benefits of using an API Gateway to provide access to microservices is that you can create a single, cohesive API for a specific client. In most cases, you'll want to create a specific API for mobile clients, perhaps even one API for iOS and one for Android. This implementation of API Gateways is commonly referred to as the **Backend for Frontend** (**BFF**) because it provides a single logical backend for each frontend application. A web application has very different needs than a mobile device.

In our situation, we'll focus on creating one endpoint that provides all the data needed by the message-view screen. This includes the message itself as well as the attachment(s), the user details of the sender, and any additional recipients of the message. If the message is public, it can also have likes and comments, which we'll imagine are served by a separate service. Our endpoint could look something like this:

```
GET /message_details/:message_id
```

The endpoint will return a response similar to the following:

```
{
  "message_details": {
    "message": {
      "id": 1234,
      "body": "Hi There!",
      "from_user_id": "user:4321"
    },
    "attachments": [{
      "id": 4543,
      "media_type": 1,
      "url": "http://..."
    }],
    "from_user": {
      "username": "paulosman",
      "profile_pic": "http://...",
      "display_name": "Paul Osman"
    },
    "recipients": [
      ...
    ],
    "likes": 200,
    "comments": [{
      "id": 943,
      "body": "cool pic",
      "user": {
        "username": "somebody",
        "profile_pic": "http://..."
      }
    }]
  }
}
```

This response should have everything a client needs to show our message-view screen. The data itself comes from a variety of services, but, as we'll see, our API Gateway does the hard work of making those requests and aggregating the responses.

How to do it...

An API Gateway is responsible for exposing an API, making multiple service calls, aggregating the results, and returning them to the client. The **Finagle Scala** framework makes this natural by representing service calls as futures, which can be composed to represent dependencies. To stay consistent with other examples in this book, we'll build a small example gateway service in Java using the Spring Boot framework:

1. Create the project skeleton. Create a new Java project and add the following dependencies and plugins to the Gradle build file. We'll be using Spring Boot and Hystrix in this recipe:

```
plugins {
    id 'org.springframework.boot' version '1.5.9.RELEASE'
}

group 'com.packtpub.microservices'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version: '1.5.9.RELEASE'
    compile group: 'com.netflix.hystrix', name: 'hystrix-core',
version: '1.0.2'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

   Looking at the JSON example in the previous section, it's clear that we are collecting and aggregating some distinct domain concepts. For the purposes of this example, we'll imagine that we have a message service that retrieves information about messages, including likes, comments, and attachments, and a user service. Our gateway service will be making a call to the message service to retrieve the message itself, then calls to the other services to get the associated data, which we'll stitch together in a single response. For the purposes of this recipe, imagine the message service is running on port `4567` and the user service on port `4568`. We'll create some stub services to mock out the data for these hypothetical microservices.

2. Create a model to represent our `Message` data:

```
package com.packtpub.microservices.gateway.models;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = false)
```

```
public class Message {

    private String id;
    private String body;

    @JsonProperty("from_user_id")
    private String fromUserId;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public String getFromUserId() {
        return fromUserId;
    }

    public void setFromUserId(String fromUserId) {
        this.fromUserId = fromUserId;
    }
}
```

It's important that non-dependent service calls be done in a non-blocking, asynchronous manner. Luckily, Hystrix has an option to execute commands asynchronously, returning `Future<T>`.

3. Create a new package, say, `com.packtpub.microservices.gateway.commands` with the following classes:

    - Create the `AttachmentCommand` class with the following content:

```
package com.packtpub.microservices.gateway.commands;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class AttachmentCommand extends HystrixCommand<String> {
    private String messageId;

    public AttachmentCommand(String messageId) {
super(HystrixCommandGroupKey.Factory.asKey("AttachmentCommand"));
        this.messageId = messageId;
    }

    @Override
    public String run() {
        RestTemplate template = new RestTemplate();
        String attachmentsUrl = "http://localhost:4567/message/" +
messageId + "/attachments";
        ResponseEntity<String> response =
template.getForEntity(attachmentsUrl, String.class);
        return response.getBody();
    }
}
```

- Create the `CommentCommand` class with the following content:

```
package com.packtpub.microservices.commands;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class CommentCommand extends HystrixCommand<String> {

    private String messageId;

    public CommentCommand(String messageId) {
super(HystrixCommandGroupKey.Factory.asKey("CommentGroup"));
        this.messageId = messageId;
    }

    @Override
    public String run() {
        RestTemplate template = new RestTemplate();
        String commentsUrl = "http://localhost:4567/message/" +
messageId + "/comments";
        ResponseEntity<String> response =
template.getForEntity(commentsUrl, String.class);
        return response.getBody();
```

```
    }
}
```

- Create the `LikeCommand` class with the following content:

```
package com.packtpub.microservices.commands;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class LikeCommand extends HystrixCommand<String> {

    private String messageId;

    public LikeCommand(String messageId) {
        super(HystrixCommandGroupKey.Factory.asKey("Likegroup"));
        this.messageId = messageId;
    }

    @Override
    public String run() {
        RestTemplate template = new RestTemplate();
        String likesUrl = "http://localhost:4567/message/" +
messageId + "/likes";
        ResponseEntity<String> response =
template.getForEntity(likesUrl, String.class);
        return response.getBody();
    }
}
```

- Our `MessageClient` class is a bit different than the previous examples—instead of returning the JSON string from the service response, it'll return an object representation, in this case, an instance of our `Message` class:

```
package com.packtpub.microservices.commands;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import com.packtpub.microservices.models.Message;
import org.springframework.web.client.RestTemplate;

public class MessageClient extends HystrixCommand<Message> {

    private final String id;
```

```
    public MessageClient(String id) {
super(HystrixCommandGroupKey.Factory.asKey("MessageGroup"));
        this.id = id;
    }

    @Override
    public Message run() {
        RestTemplate template = new RestTemplate();
        String messageServiceUrl = "http://localhost:4567/message/"
+ id;
        Message message = template.getForObject(messageServiceUrl,
Message.class);
        return message;
    }
}
```

- Create the `UserCommand` class with the following content:

```
package com.packtpub.microservices.commands;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class UserCommand extends HystrixCommand<String> {

    private String id;

    public UserCommand(String id) {
        super(HystrixCommandGroupKey.Factory.asKey("UserGroup"));
        this.id = id;
    }

    @Override
    public String run() {
        RestTemplate template = new RestTemplate();
        String userServiceUrl = "http://localhost:4568/user/" + id;
        ResponseEntity<String> response =
template.getForEntity(userServiceUrl, String.class);
        return response.getBody();
    }
}
```

4. Stitch together the execution of these Hystrix commands in a single controller that exposes our API as the `/message_details/:message_id` endpoint:

```
package com.packtpub.microservices;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.packtpub.microservices.commands.*;
import com.packtpub.microservices.models.Message;
import org.springframework.boot.SpringApplication;
import org.springframework.http.MediaType;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.io.IOException;
import java.io.StringWriter;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

@SpringBootApplication
@RestController
public class MainController {

    @RequestMapping(value = "/message_details/{id}", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
    public Map<String, HashMap<String, String>>
messageDetails(@PathVariable String id)
            throws ExecutionException, InterruptedException,
IOException {

        Map<String, HashMap<String, String>> result = new
HashMap<>();
        HashMap<String, String> innerResult = new HashMap<>();

        Message message = new MessageClient(id).run();
        String messageId = message.getId();

        Future<String> user = new
UserClient(message.getFromUserId()).queue();
        Future<String> attachments = new
AttachmentClient(messageId).queue();
        Future<String> likes = new LikeClient(messageId).queue();
        Future<String> comments = new
CommentClient(messageId).queue();
```

```
            ObjectMapper mapper = new ObjectMapper();
            StringWriter writer = new StringWriter();
            mapper.writeValue(writer, message);

            innerResult.put("message", writer.toString());
            innerResult.put("from_user", user.get());
            innerResult.put("attachments", attachments.get());
            innerResult.put("comments", comments.get());
            innerResult.put("likes", likes.get());

            result.put("message_details", innerResult);

            return result;
        }

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

5. There you have it. Run the service with `./gradlew bootRun` and test it by making a request to:

```
$ curl -H "Content-Type: application/json"
http://localhost:8080/message_details/1234
```

Stopping cascading failures with Hystrix

Failures in a complex system can be hard to diagnose. Often, the symptom can appear far away from the cause. Users might start experiencing higher-than-normal error rates during login because of some downstream service that manages profile pictures or something else tangentially related to user profiles. An error in one service can often propagate needlessly to a user request and adversely impact user experience and therefore trust in your application. Additionally, a failing service can have cascading effects, turning a small system outage into a high-severity, customer-impacting incident. It's important when designing microservices to consider failure isolation and decide how you want to handle different failure scenarios.

A number of patterns can be used to improve the resiliency of distributed systems. Circuit breakers are a common pattern used to back off from making requests to a temporarily overwhelmed service. Circuit breakers were first described in Michael Nygard's book *Release It!*. A calling service defaults to a closed state, meaning requests are sent to the downstream service.

If the calling service receives too many failures too quickly, it can change the state of its circuit breaker to open, and start failing fast. Instead of waiting for the downstream service to fail again and adding to the load of the failing service, it just sends an error to upstream services, giving the overwhelmed service time to recover. After a certain amount of time has passed, the circuit is closed again and requests start flowing to the downstream service.

There are many available frameworks and libraries that implement circuit breakers. Some frameworks, such as Twitter's Finagle, automatically wrap every RPC call in a circuit breaker. In our example, we'll use the popular Netflix library, `hystrix`. Hystrix is a general-purpose, fault-tolerance library that structures isolated code as commands. When a command is executed, it checks the state of a circuit breaker to decide whether to issue or short circuit the request.

How to do it...

Hystrix is made available as a Java library, so we'll demonstrate its use by building a small Java Spring Boot application:

1. Create a new Java application and add the dependencies to your `build.gradle` file:

```
plugins {
    id 'org.springframework.boot' version '1.5.9.RELEASE'
}

group 'com.packetpub.microservices'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version: '1.5.9.RELEASE'
    compile group: 'com.netflix.hystrix', name: 'hystrix-core',
version: '1.0.2'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

2. We'll create a simple `MainController` that returns a simple message. This is a contrived example, but it demonstrates an upstream service making downstream calls. At first, our application will just return a hardcoded `Hello, World!` message. Next, we'll move the string out to a Hystrix command. Finally, we'll move the message to a service call wrapped in a Hystrix command:

```
package com.packtpub.microservices;

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

@SpringBootApplication
@EnableAutoConfiguration
@RestController
public class MainController {
    @RequestMapping("/message")
    public String message() {
        return "Hello, World!";
    }

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

3. Move the message out to `HystrixCommand`:

```
package com.packtpub.microservices;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

public class CommandHelloWorld extends HystrixCommand<String> {

    private String name;

    CommandHelloWorld(String name) {
super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
```

```
    public String run() {
        return "Hello, " + name + "!";
    }
}
```

4. Replace the method in `MainController` to use `HystrixCommand`:

```
@RequestMapping("/message")
public String message() {
    return new CommandHelloWorld("Paul").execute();
}
```

5. Move the message generation to another service. We're hardcoding the hypothetical message service URL here, which is not a good practice but will do for demonstration purposes:

```
package com.packtpub.microservices;

import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class CommandHelloWorld extends HystrixCommand<String> {

    CommandHelloWorld() {
super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
    }

    @Override
    public String run() {
        RestTemplate restTemplate = new RestTemplate();
        String messageResourceUrl = "http://localhost:4567/";
        ResponseEntity<String> response =
restTemplate.getForEntity(messageResourceUrl, String.class);
        return response.getBody();
    }

    @Override
    public String getFallback() {
        return "Hello, Fallback Message";
    }
}
```

6. Update the `MainController` class to contain the following:

```
package com.packetpub.microservices;

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

@SpringBootApplication
@EnableAutoConfiguration
@RestController
public class MainController {

    @RequestMapping("/message")
    public String message() {
        return new CommandHelloWorld().execute();
    }

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

7. Our `MainController` class now makes a service call, wrapped in a Hystrix command, to generate a message to send back to the client. You can test this by creating a very simple service that generates a message string. `sinatra` is a simple-to-use Ruby library ideal for creating test services. Create a new file called `message-service.rb`:

```
require 'sinatra'

get '/' do
  "Hello from Sinatra"
end
```

8. Run the service by running `ruby message-service.rb` and then make a few sample requests to your Hystrix-enabled service. You can simulate a failure by modifying the service to return a `503`, indicating that it is temporarily overwhelmed:

```
require 'sinatra'

get '/' do
  halt 503, 'Busy'
end
```

   Your Spring service should now attempt to reach the service but use the value in the fallback when it encounters a `503`. Furthermore, after a number of attempts, the command's circuit breaker will be tripped and the service will start defaulting to the fallback for a period of time.

Rate limiting

In addition to techniques such as circuit breaking, rate limiting can be an effective way to prevent cascading failures in a distributed system. Rate limiting can be effective at preventing spam, protecting against **Denial of Service** (**DoS**) attacks, and protecting parts of a system from becoming overloaded by too many simultaneous requests. Typically implemented as either a global or per-client limit, rate limiting is usually part of a proxy or load balancer. In this recipe, we'll use NGINX, a popular open source load balancer, web server, and reverse proxy.

Most rate-limiting implementations use the *leaky-bucket algorithm*—an algorithm that originated in computer network switches and telecommunications networks. As the name suggests, the leaky-bucket algorithm is based on the metaphor of a bucket with a small leak in it that controls a constant rate. Water is poured into the bucket in bursts, but the leak guarantees that water exists in the bucket at a steady, fixed rate. If the water is poured in faster than the water exits the bucket, eventually the bucket will overflow. In this case, the overflow represents requests that are dropped.

It's certainly possible to implement your own rate-limiting solution; there are even implementations of the algorithms out there that are open source and available to use. It's a lot easier, however, to use a product such as NGINX to do rate limiting for you. In this recipe, we'll configure NGINX to proxy requests to our microservice.

How to do it...

1. Install NGINX by running the following command:

```
apt-get install nginx
```

2. `nginx` has a `config` file, `nginx.conf`. On an Ubuntu-based Linux system, this will probably be in the `/etc/nginx/nginx.conf` directory. Open the file and look for the `http` block and add the following content:

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;
server {
    location /auth/signin {
        limit_req zone=mylimit;
        proxy_pass http://my_upstream;
    }
}
```

As you can see from the preceding code, rate limiting is implemented with two configuration directives. The `limit_req_zone` directive defines the parameters for rate limiting. In this example, we're implementing a rate limit, based on the client's IP address, of 10 requests per second. The `limit_req` directive applies our rate limiting to a specific path or location. In this case, we're applying it to all requests to `/auth/signin`, presumably because we don't want bots scripting the creation of accounts!

Using service mesh for shared concerns

As web services' frameworks and standards evolve, the amount of boilerplate or shared application concerns is reduced. This is because, collectively, we figure out what parts of our applications are universal and therefore shouldn't need to be re-implemented by every programmer or team. When people first started networking computers, programmers writing network-aware applications had to worry about a lot of low-level details that are now abstracted out by the operating system's networking stack. Similarly, there are certain universal concerns that all microservices share. Frameworks such as Twitter's Finagle wrap all network calls in a circuit breaker, increasing fault tolerance and isolating failures in systems. Finagle and Spring Boot, the Java framework we've been using for most of these recipes, both support exposing a standard metrics endpoint that standardizes basic network, JVM, and application metrics collected for microservices.

Every microservice should consider a number of shared application concerns. From an observability perspective, services should strive to emit consistent metrics and structured logs. To improve the reliability of our systems, services should wrap network calls in circuit breakers and implement consistent retry and back-off logic. To support changes in network and service topology, services should consider implementing client-side load balancing and use centralized service discovery.

Instead of implementing all of these features in each of our services, it would be ideal to abstract them out to something outside our application code that could be maintained and operated separately. Like the features of our operating systems network stack, if each of these features is implemented by something our application could rely on being present, we would not have to worry about them being available. This is the idea behind a service mesh.

Running a service mesh configuration involves running each microservice in your system behind a network proxy. Instead of services speaking directly to one another, they communicate via their respective proxies, which are installed as sidecars. Practically speaking, your service would communicate with its own proxy running on localhost. As network requests are sent through a services proxy, the proxy can control what metrics are emitted and what log messages are output. The proxy can also integrate directly with your service registry and distribute requests evenly among active nodes, keeping track of failures and opting to fail fast when a certain threshold has been reached. Running your system in this kind of configuration can ease the operational complexity of your system while improving the reliability and observability of your architecture.

Like most of the recipes discussed in this chapter, there are numerous open source solutions for running a service mesh. We'll focus on **Linkerd**, an open source proxy server built and maintained by buoyant. The original authors of Linkerd worked at Twitter before forming buoyant and as such, Linkerd incorporates many of the lessons learned by teams at Twitter. It shares many features with the Finagle Scala framework, but can be used with services written in any language. In this recipe, we'll walk through installing and configuring Linkerd and discuss how we can use it to control communication between our Ruby on Rails monolith API and our newly developed media service.

How to do it...

To demonstrate running a service behind a proxy, we'll install and run an instance of Linkerd and configure it to handle requests to and from your service. There are instructions on the Linkerd website for running it in Docker, Kubernetes, and other options. To keep things simple, we'll focus on running Linkerd and our service locally:

1. Download the latest Linkerd release at `https://github.com/linkerd/linkerd/releases`.
2. Extract the tarball by executing the following command:

```
$ tar xvfz linkerd-1.3.4.tgz
$ cd linkerd-1.3.4
```

3. By default, `linkerd` ships with a configuration that uses file-based service discovery. We'll discuss alternatives to this approach next, but, for now, create a new file called `disco/media-service` with the following contents:

```
localhost 8080
```

4. This maps the hostname and port to a service called `media-service`. Linkerd uses this file to look up services by name and determines the hostname and port mappings.
5. Run Linkerd as follows:

```
$ ./linkerd-1.3.4-exec config/linkerd.yaml
```

6. Start the service on port `8080`. Change into the `media-service` directory and run the service:

```
$ ./gradlew bootRun
```

7. Linkerd is running on port `4140`. Test that proxying is working with the following request:

```
$ curl -H "Host: attachment-service" http://localhost:4140/
```

4
Modules and Toolkits

Now we'll review a couple of options and build a more or less simple microservice so that we can point out advantages and disadvantages for each approach.

We'll look at four different modules:

- **Express**: One of the most commonly used modules in the Node.js ecosystem
- **Micro**: A very minimalistic microservices approach
- **Seneca**: A microservice toolkit based on property matching
- **Hydra**: A package that bundles a couple of modules to help you resolve many microservices concerns, such as distribution and monitoring

Express

If you're familiar with Node.js, you probably know of one module that is the core for most Node.js platforms and applications that are around. You have probably heard of Express, and it's not by accident. It's eight years old and it's been there since the beginning of Node.js.

Express is a rock-solid layer that helps you create applications faster. As it states on its website, it's a *fast, unopinionated,* and *minimalist web framework*.

Node.js has an HTTP module that you can use to create a simple HTTP server. The problem is that it's too raw and you'll have to do a ton of work so that you can make it usable and modular. Instead of you creating that layer, you can use Express.

To install Express, create a folder and run the following command:

```
npm init -y
npm install express --save
```

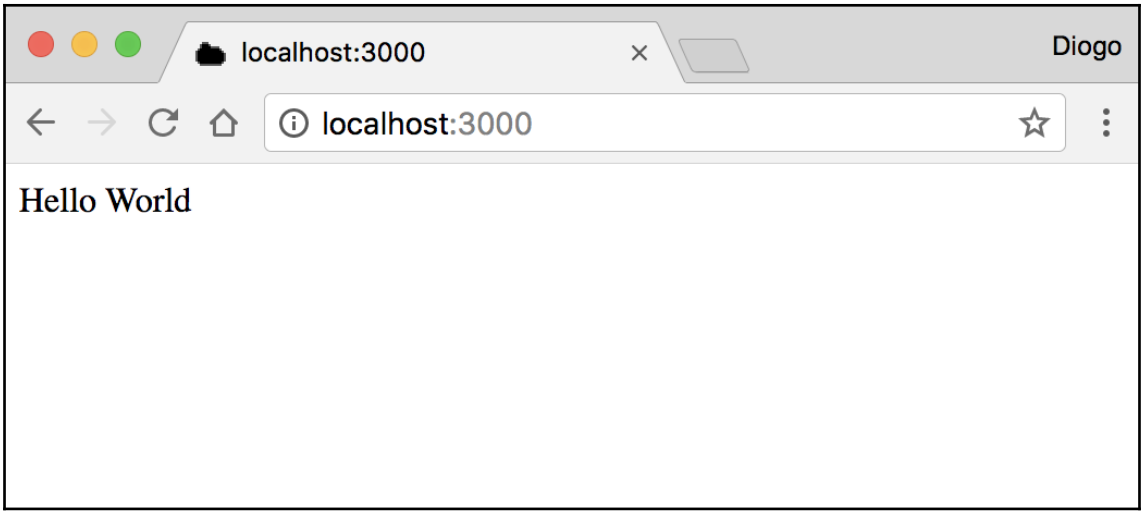
Then, create a file called `app.js` and place the following code inside it:

```
let express = require("express");
let app     = express();

app.get("/", (req, res) => {
    res.send("Hello World");
});

app.listen(80);
```

Then, run your code using the following command:

```
node app
```

Go to your browser and type the address `http://localhost:3000/`. You should see something like the following:

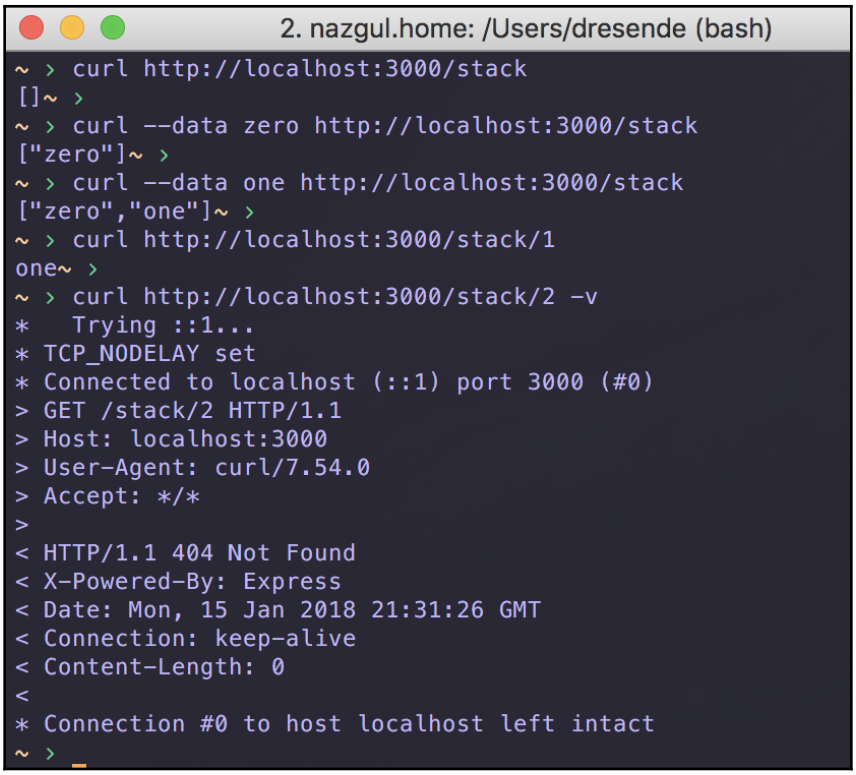


With Express, you can associate functions to routes. A route is a URL with a specific HTTP verb, but it doesn't necessarily have to. You can have automatic parameters parsing inside routes. You can even associate more functions to the same route and have them be called sequentially. You can also stop the sequence at any time. It's up to you; remember, Express is not opinionated.

Change your `app.js` to the following code and run it again:

```
let express = require("express");
let app     = express();
let stack   = [];

app.post("/stack", (req, res, next) => {
    let buffer = "";

    req.on("data", (data) => {
        buffer += data;
    });
    req.on("end", () => {
        stack.push(buffer);
        return next();
    });
});

app.delete("/stack", (req, res, next) => {
    stack.pop();
    return next();
});

app.get("/stack/:index", (req, res) => {
    if (req.params.index >= 0 && req.params.index < stack.length) {
        return res.end("" + stack[req.params.index]);
    }
    res.status(404).end();
});

app.use("/stack", (req, res) => {
    res.send(stack);
});

app.listen(3000);
```

You'll have trouble testing it in the browser because we're using `POST` and `DELETE` verbs. Instead, let's use `curl` and make requests in the command line. Follow this sequence and see if you understood the preceding code:



We first make a request to `/stack` just to see if it will return an empty array. Then, we use `--data` to pass `zero` as the HTTP request body and `curl` assumes it's a `POST`. We do it again, this time, with body `one`. For both requests, we get the final stack result. We then request stack index 1 and it returns `one`. We then request index 2 and, as you can see by the `-v` parameter, our code returns a proper `404 Not Found` error.

You might have wondered how the stack is returned on both `GET` and `POST` requests and that's the reason for calling `next()`, which tells Express to pass to the next route (if there are any) available. In our case, it's the route that's been defined that's using the use method. This is a catch all for all `HTTP` verbs. Remember: route definition order is important and that's why the catch all is done at the end.

As you can see, we can scale pretty fast to complex logic. It's important to note that although Express really helps you create an `HTTP` service easier, it's still just an improved layer over the `http` and `https` modules that are available.

To help you even more, it has an integrated modular system, called middleware, and hundreds of compatible published modules that help you create services even faster, such as session handling, templating, caching, and security modules.

Let's make our example a little bit more complex so that you can see what I mean. First, install some middleware:

```
npm i body-parser --save
```

Now, change your code to this:

```
let express = require("express");
let body    = require("body-parser");
let route   = express.Router();
let app     = express();
let stack   = [];

app.use(body.text({ type: "*/*" }));

route.post("/", (req, res, next) => {
    stack.push(req.body);

    return next();
});

route.delete("/", (req, res, next) => {
    stack.pop();

    return next();
});

route.get("/:index", (req, res) => {
    if (req.params.index >= 0 && req.params.index < stack.length) {
        return res.end("" + stack[req.params.index]);
    }
    res.status(404).end();
```

```
});

route.use((req, res) => {
    res.send(stack);
});

app.use("/stack", route);
app.listen(3000);
```

It has the same number of lines, but we improved our code by making two important changes:

- Instead of reading the request body, we now use a middleware, `body-parser`, which handles it for us and supports several body types and compression.
- We created an `express.Route` and attached it to our service at the last but one line. We can use this to move our route definition to a separate file and make it URL agnostic (see how we only mention `/stack` once, when attaching).

Micro

Let's look at another interesting tool. This module is called micro and was created by ZEIT, a team of some of the most influential Node.js developers. It is meant to be a minimalistic framework to create slim and fast microservices.

Let's repeat our first example. Create a folder and run the following command:

```
npm init -y
npm install --save micro
```

Now, create a file called `app.js` and write the following code inside it:

```
module.exports = (req, res) => {
    res.end("Hello World");
};
```

Change your `scripts` property in `package.json` to this:

```
"scripts": {
    "start": "micro"
},
```

Now, just run this:

```
npm start
```

You'll see something like this, which indicates that a micro is running on its default port, `3000`:

```
micro: Accepting connections on port 3000
```

You can now refresh your browser and you will get the same page you saw earlier. But now, you have it with only three lines of code (and some configuration).

That's it. Micro is very minimalistic. You'll have to declare and install all your dependencies as it won't do anything other than enable you to write very slim microservices. This is handy if you don't want to carry the 2.3 MB of dependencies from our first Express example.

Seneca

Now it's time for a completely different approach. Let's look at another framework called Seneca. This framework was designed to help you develop message-based microservices. It has two distinct characteristics:

- **Transport agnostic**: Communication and message transport is separated from your service logic and it's easy to swap transports
- **Pattern matching**: Messages are JSON objects and each function exposes what sort of messages they can handle based on object properties

Being able to change transports is not a big deal; many tools allow you to do so. What is really interesting about this framework is its ability to expose functions based on object patterns. Let's start by installing Seneca:

```
npm install seneca
```

For now, let's forget the transport and create a producer and consumer in the same file. Let's look at an example:

```
const seneca  = require("seneca");
const service = seneca();

service.add({ math: "sum" }, (msg, next) => {
    next(null, {
        sum : msg.values.reduce((total, value) => (total + value), 0)
    });
});

service.act({ math: "sum", values: [ 1, 2, 3 ] }, (err, msg) => {
    if (err) return console.error(err);
```

```
    console.log("sum = %s", msg.sum);
});
```

There's a lot to absorb. The easy part comes first as we include the `seneca` module and create a new service.

We then expose a producer function that matches an object that has `math` equal to `sum`. This means that any request object to the service that has the property `math` and that is equal to `sum` will be passed to this function. This function accepts two arguments. The first, which we called `msg`, is the request object (the one with the `math` property and anything else the object might have). The second argument, `next`, is the callback that the function should invoke when finished or in case of an error. In this particular case, we're expecting an object that also has a `values` list and we're returning the sum of all values by using the `reduce` method that's available in arrays.

Finally, we invoke `act`, expecting it to consume our producer. We pass an object with the `math` equal to `sum` and a list of `values`. Our producer should be invoked and should return the sum.

Assuming you have this code in `app.js`, if you run this in the command line, you should see something like this:

```
$ node app

sum = 6
```

Let's try and replicate our previous stack example. This time, instead of having the consumer and producer in the code, we'll use `curl` as the consumer, just like we did previously.

First, we need to create our `service`. We do that, as we've seen before, by loading Seneca and creating an instance:

```
const seneca  = require("seneca");
const service = seneca({ log: "silent" });
```

We explicitly tell it that we don't care about logging for now. Now, let's create a variable to hold our stack:

```
const stack = [];
```

We then create our producers. We'll create three of them: one for adding an element to the stack, called push; one to remove the last element from the stack, called pop; and one to see the stack, called get. Both push and pop will return the final stack result. The third producer is just a helper function so that we can see the stack without performing any operations.

To add elements to the stack, we define:

```
service.add("stack:push,value:*", (msg, next) => {
    stack.push(msg.value);

    next(null, stack);
});
```

There are a few new things to see here:

- We defined our pattern as a string instead of an object. This action string is a shortcut to the extended object definition.
- We explicitly indicate that we need a value.
- We also indicate that we don't care what the value is (remember, this is pattern matching).

We now define a simpler function to remove the last element of the `stack`:

```
service.add("stack:pop", (msg, next) => {
    stack.pop();

    next(null, stack);
});
```

This one is simpler as we don't need a value, we're just removing the last one. We're not addressing the case where the stack is empty already. An empty array won't throw an exception, but perhaps, in a real scenario, you would want another response.

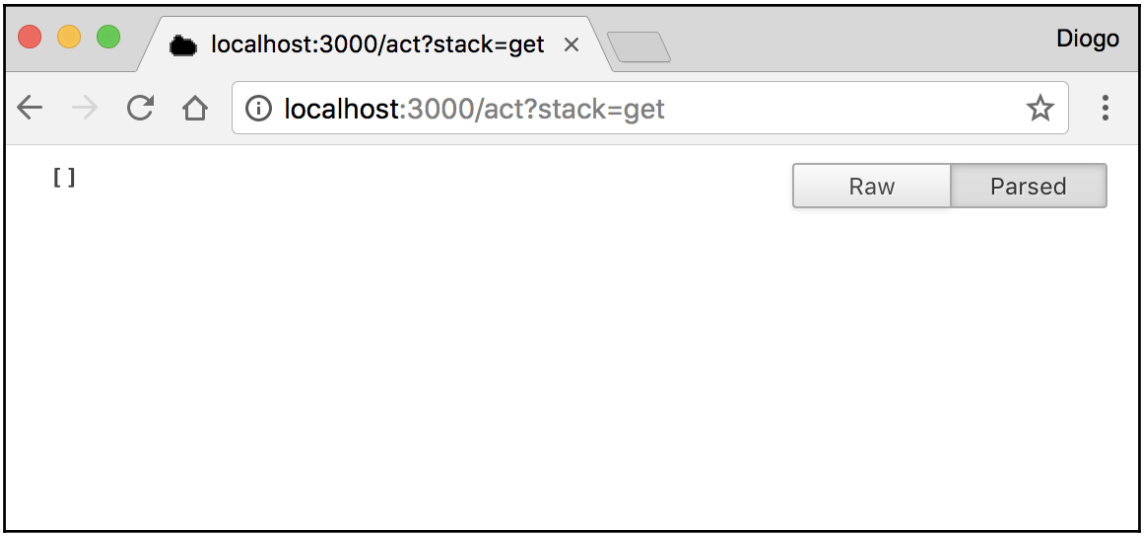
Our third function is even simpler as we just return the `stack`:

```
service.add("stack:get", (msg, next) => {
    next(null, stack);
});
```

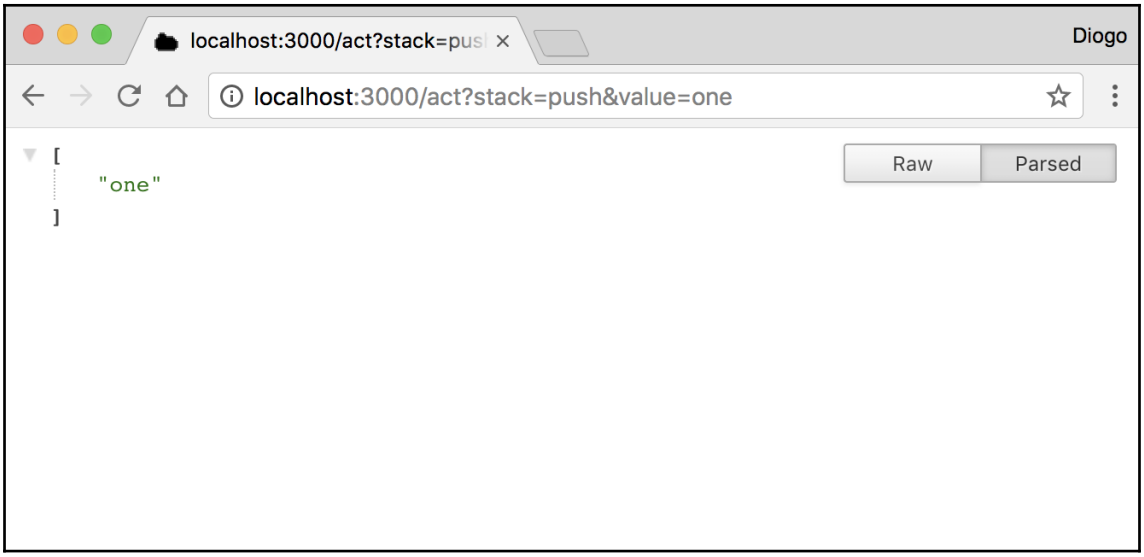
Finally, we need to tell our `service` to listen for messages. The default transport is HTTP and we just indicate port `3000` as we did in our previous examples:

```
service.listen(3000);
```

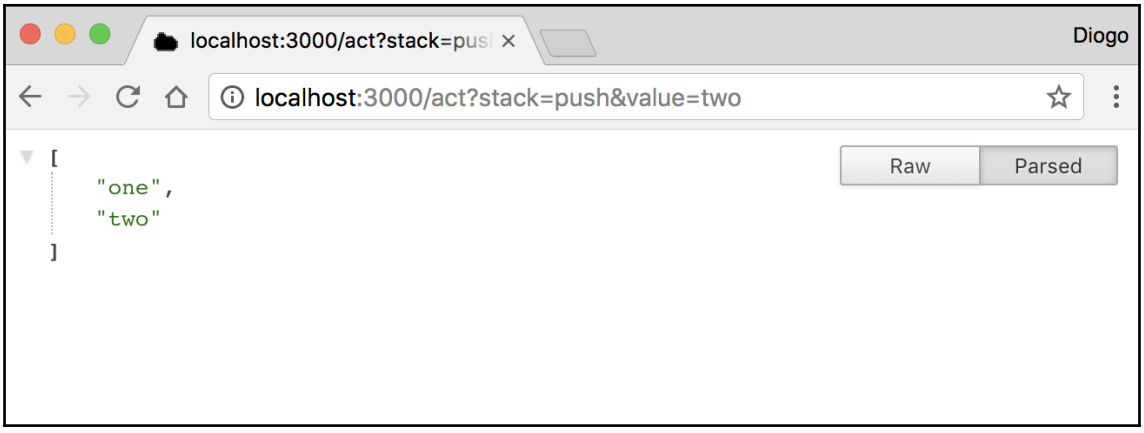
Wrap all this code in a file and try it out. You can use curl or just try it in your browser. Seneca won't differentiate between HTTP verbs in this case. Let's begin by checking our `stack`. The URL describes an action (`/act`) we want to perform and the query parameter gets converted to our pattern:



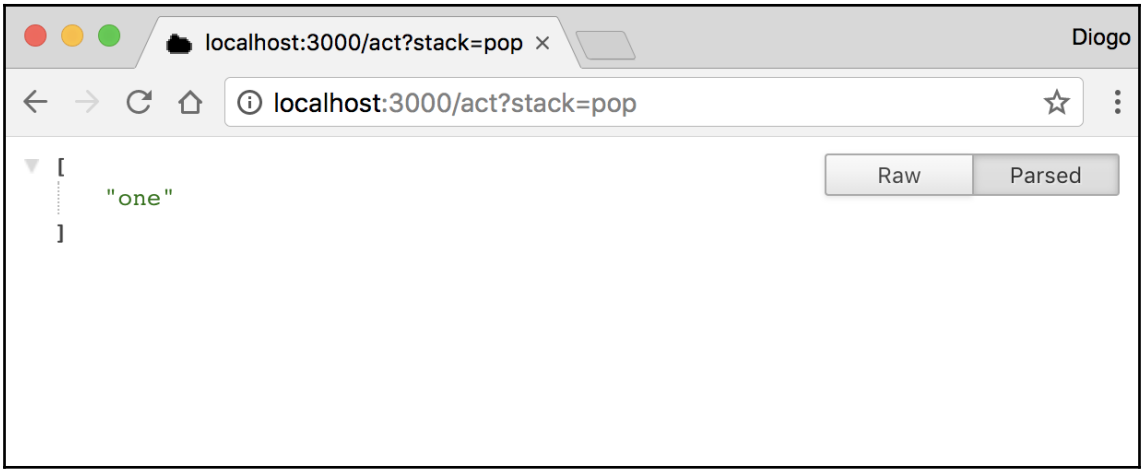
We can then try adding the value `one` to our `stack` and see the final `stack`:



We can continue and add the value `two` and see how the `stack` grows:



If we then try to remove the last element, we'll see the `stack` shrinking:



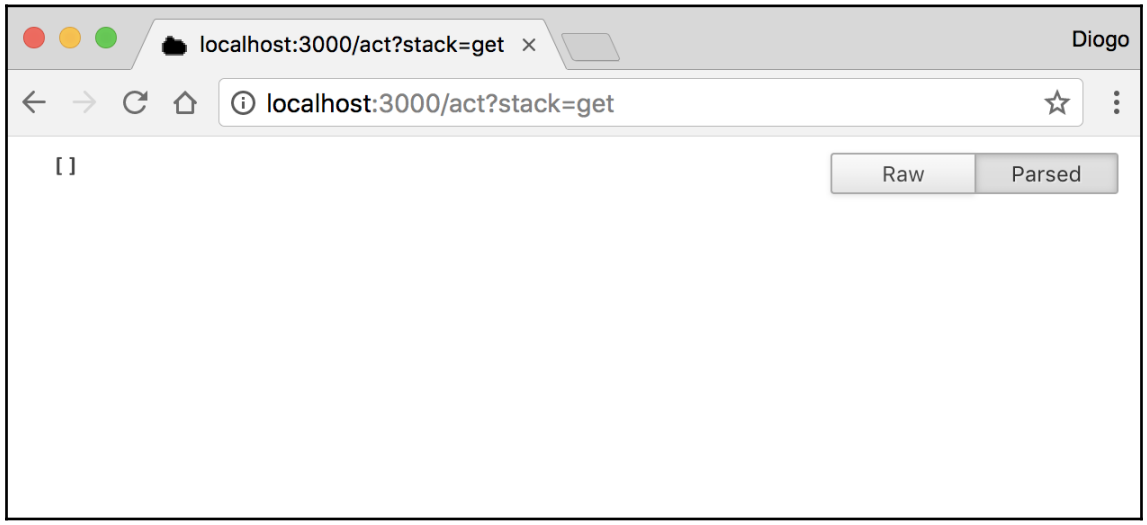
As in Express, Seneca also has middleware that you can install and use. In this case, the middleware is called plugins. By default, Seneca includes a number of core plugins for transport, and both HTTP and TCP transports are supported. There are more transports available, such as **Advanced Message Queuing Protocol** (**AMQP**) and Redis.

There are also storage plugins for persistent data and there's support for several database servers, both relational and non-relational. Seneca exposes an **object-relational mapping** (**ORM**)-like interface to manage data entities. You can manipulate entities and use a simple storage in development and then move to production storage later on. Let's see a more complex example of this:

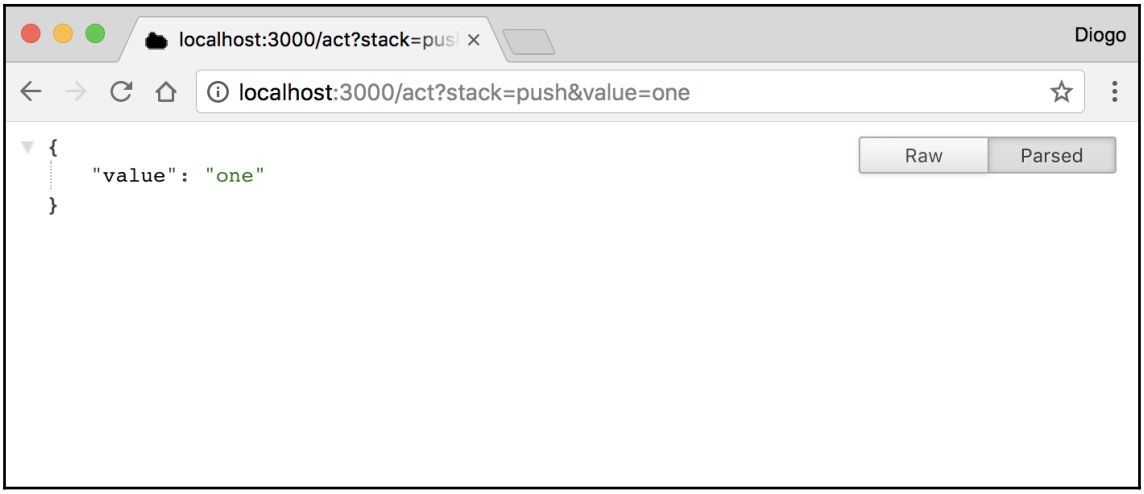
```
const async   = require("async");
const seneca  = require("seneca");
const service = seneca();

service.use("basic");
service.use("entity");
service.use("jsonfile-store", { folder : "data" });

const stack = service.make$("stack");

stack.load$((err) => {
    if (err) throw err;

    service.add("stack:push,value:*", (msg, next) => {
        stack.make$().save$({ value: msg.value }, (err) => {
            return next(err, { value: msg.value });
```

```
        });
    });

    service.add("stack:pop,value:*", (msg, next) => {
        stack.list$({ value: msg.value }, (err, items) => {
            async.each(items, (item, next) => {
                item.remove$(next);
            }, (err) => {
                if (err) return next(err);

                return next(err, { remove: items.length });
            });
        });
    });

    service.add("stack:get", (msg, next) => {
        stack.list$((err, items) => {
            if (err) return next(err);

            return next(null, items.map((item) => (item.value)));
        });
    });

    service.listen(3000);
});
```

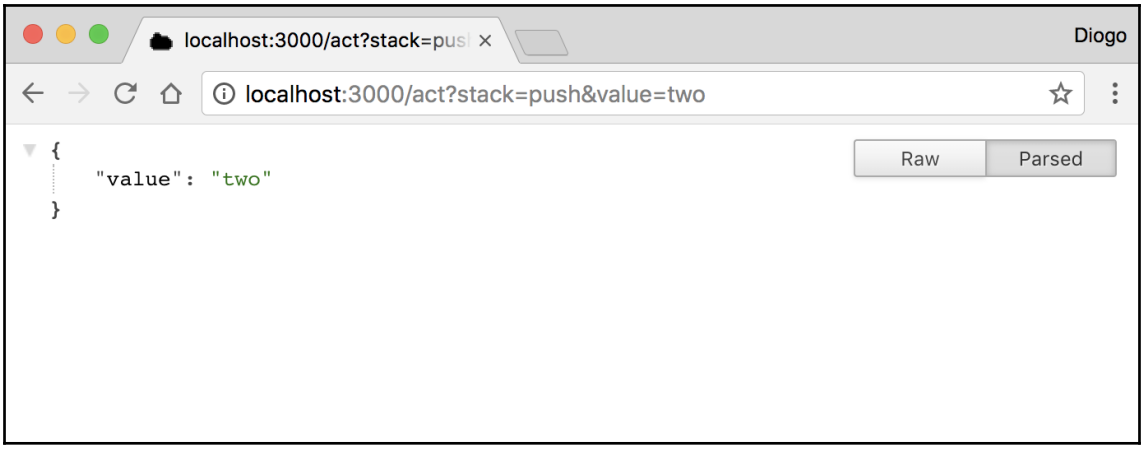
Just run this new code and we'll see how this code behaves by making some requests to test it. First, let's see how our `stack` is by requesting it:



Nothing different. Now, let's add the value `one` to the `stack`:



Well, we haven't received the final `stack`. We could, but instead we changed the service to return the exact item that was added. It's actually a good way to confirm what we just did. Let's add another one:



Again, it returns the value we just added. Now, let's see how our stack is:

Our `stack` now has our two values. Now comes one big difference compared with the previous code. We're using *entities*, an API exposed by Seneca, which helps you store and manipulate data objects using a simple abstraction layer similar to an ORM, or to people who are familiar with Ruby, an `ActiveRecord`.

Our new code, instead of just popping out the last value, removes a value we indicate. So, let's remove the value `one` instead of `two`:

Success! We removed exactly one item. Our code will remove all items from the `stack` that match the value (it has no duplication check so you can have repeated items). Let's try to remove the same item again:

No more items match `one`, so it didn't remove anything. We can now check our `stack` and confirm that we still have the value `two`:

Correct! And, as a bonus, you can stop and restart the code and your stack will still have the value `two`. That's because we're using the JSON file store plugin.

When testing using Chrome or any other browser, be aware that sometimes, browsers make requests in advance while you're typing. Because we already tested our first code, which had the same URL addresses, the browser might duplicate requests and you might get a stack with duplicated values without knowing why. This is why.

Hydra

Let's get back to Express. As you've seen before, it's a rock-solid layer on top of the `http` module. Although it adds an important base layer in the somewhat raw module, it still lacks many features you need to make a good microservice.

As there are lots of plugins out there to extend Express, it can be hard to pick a useful list for us to use.

After picking the right list, you'll still need to make other decisions:

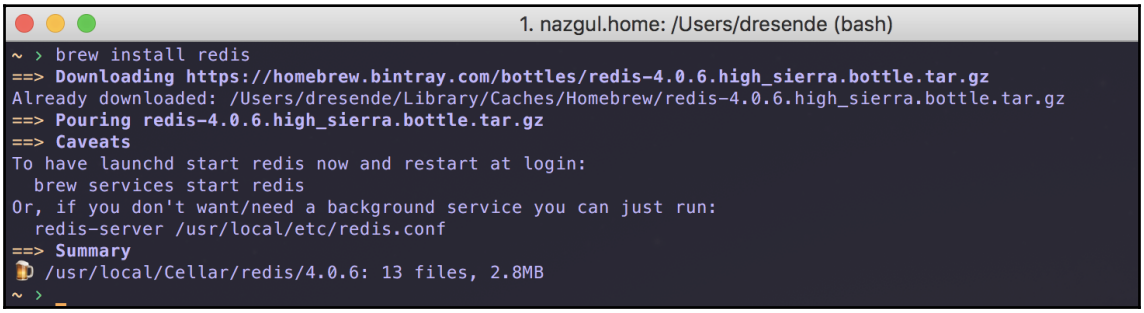
- How can I distribute my service with multiple instances?
- How can the service be discovered?
- How can I monitor whether my service is running properly?

Enter Hydra, a framework that facilitates building distributed microservices. Hydra leverages the power of Express and helps you create microservices or communicate with microservices.

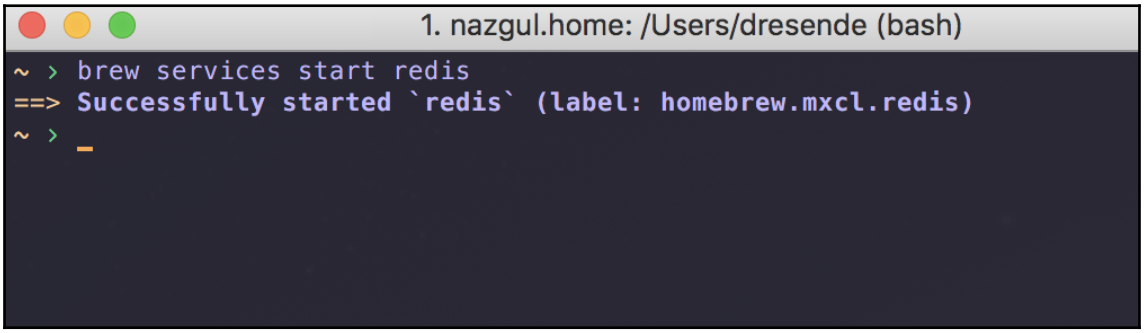
It will, out of the box, enable you to:

- Do service registration and service discovery, allowing your microservices to discover and be discoverable
- Communicate with microservices and load balance communication between multiple instances, taking care of failed instances and automatically rerouting requests to other running instances
- Monitor instances, checking whether the microservice is available and operating normally

Unlike the other modules we've reviewed so far, Hydra has a dependency that is not installable directly using the NPM. Hydra uses Redis to accomplish its goal. Look for information on the Redis website at `https://redis.io/` to install it on your operating system before continuing. If you have macOS and use Homebrew, type the following to install `redis`:



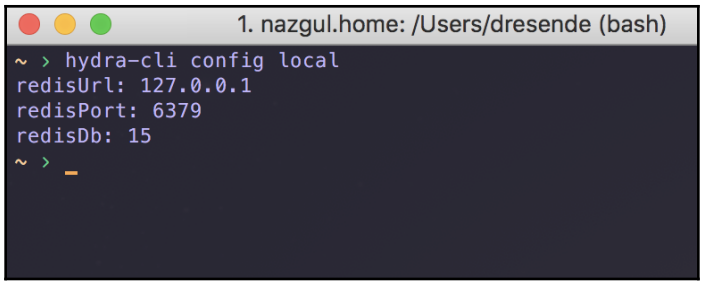
Now, let's make sure `redis` has successfully started:



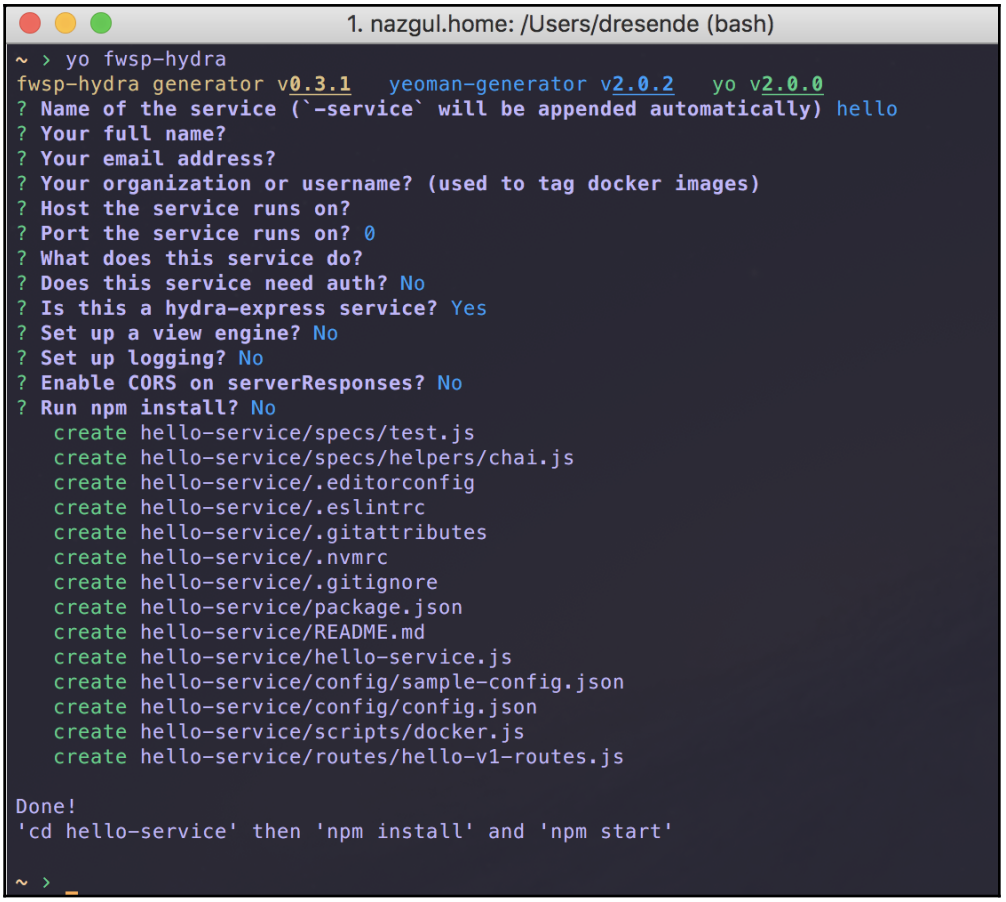
After that, we need to install Hydra command-line tools:

```
sudo npm install -g yo generator-fwsp-hydra hydra-cli
```

We now need to configure the connection to Redis. We do this by creating a configuration. Type in the command and follow the instructions. If you installed it locally (or used the preceding instructions), you should answer something similar to the following screenshot:

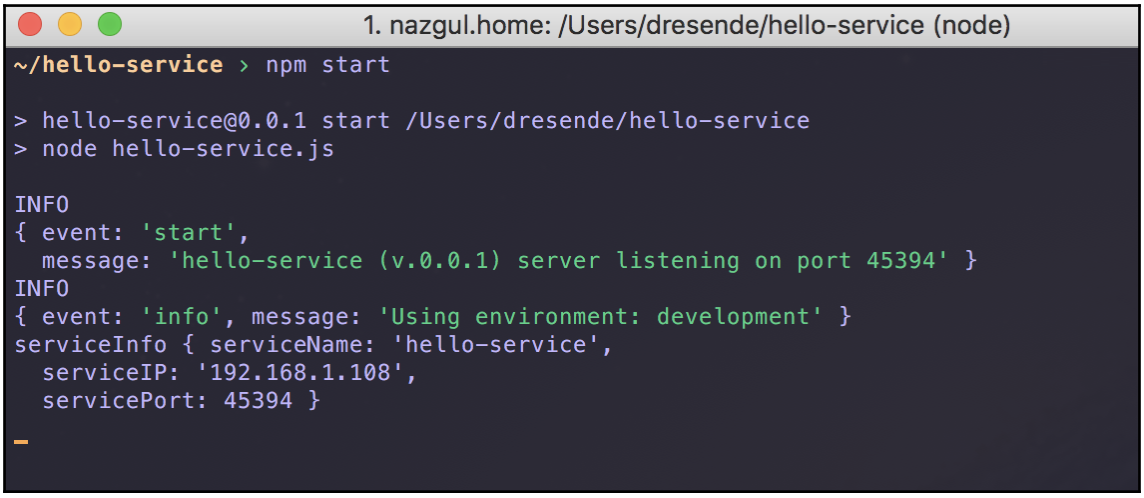


Now, let's create a very simple microservice, just to see what the workflow is like. Hydra has a scaffolding tool using `yeoman`. To create a service, type the following command and follow the instructions:

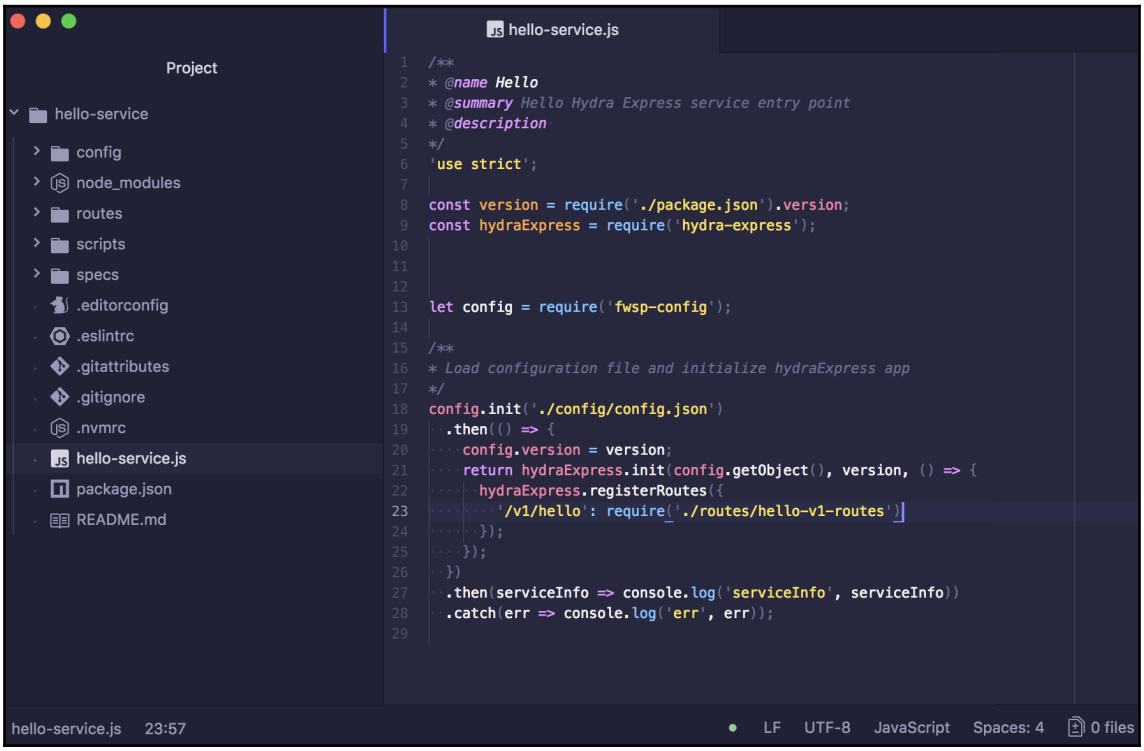


On the name of the service, just type **hello**. Just hit *Enter* to the rest of the questions to use the defaults. In the end, enter the folder that was created and install the dependencies:

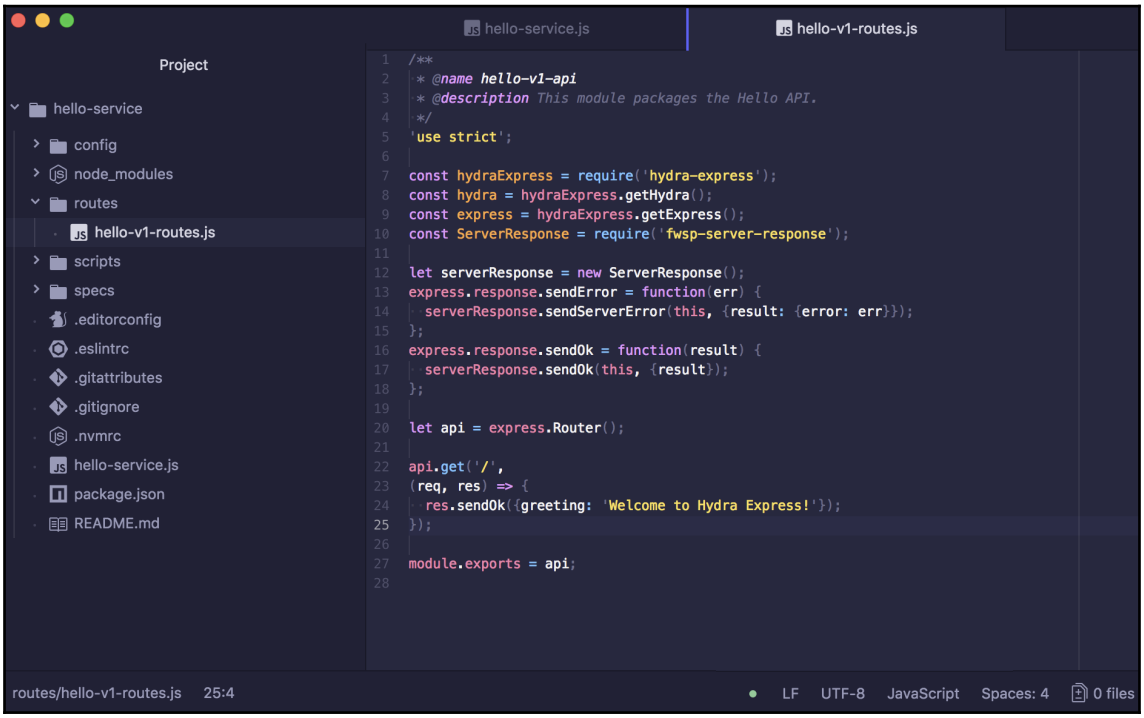
The service is now ready to start. You might have already seen the instructions when scaffolding the service. Let's start the service:



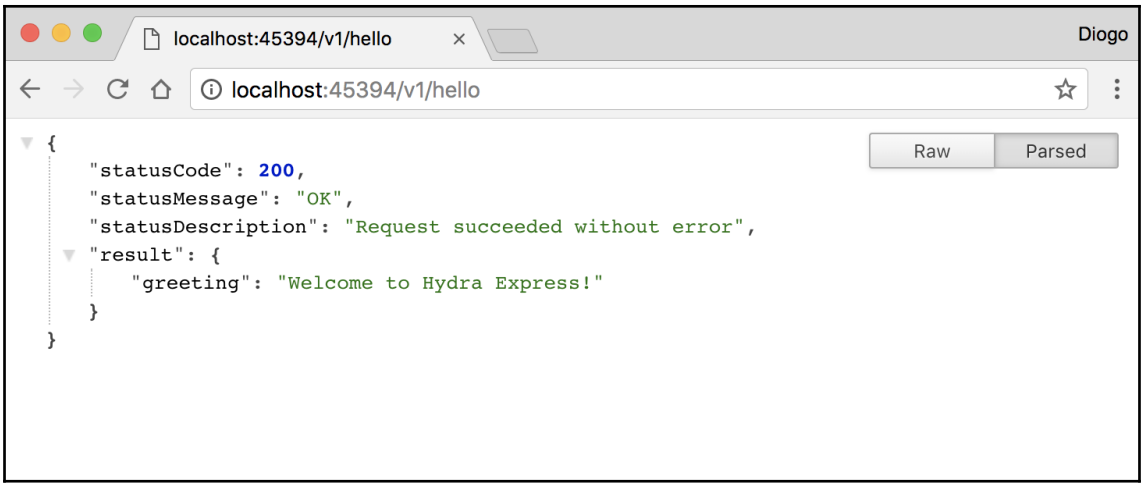
As we can see from the preceding screenshot, the service has started and has been attached to a local IP (**192.168.1.108**) and port (**45394**). Open up the folder in your code editor:



You'll see a file in the base folder called `hello-service.js`, which has the service routes inside it. You'll find the `/v1/hello` route, which points to another file in `routes/hello-v1-routes.js`:



Inside that file, you'll see the response to that route. Now, let's jump to a web browser and see if it's up and running:



What we saw in the file is inside the result property of the JSON response. We just deployed our first Hydra microservice without writing a single line of code!

Summary

We've just covered a range of different modules and toolkits to help develop microservices. From the tiny Micro, through patterns in Seneca, to the Hydra bundle, many approaches are available.

They all target different audiences and fill different needs. I would advise you to experiment with some of them to help you make a better choice instead of just picking one.

Let's dig into some of these tools and start creating a more complete microservice. In the next chapter, we'll be making a useful microservice, covering different use cases, while we develop a fully functional and distributed microservice.

5
Building a Microservice

Now that we've seen some examples of building microservices using some tools, let's dig deeper and create a microservice from scratch using these tools. To accomplish our goal, we'll first use Express, then see how we could refactor it using Hydra, and finally, we'll create our microservice using the Seneca approach.

There are many microservices we could create, but some are more interesting than others. More specifically, a microservice that you can use in several applications is obviously more useful.

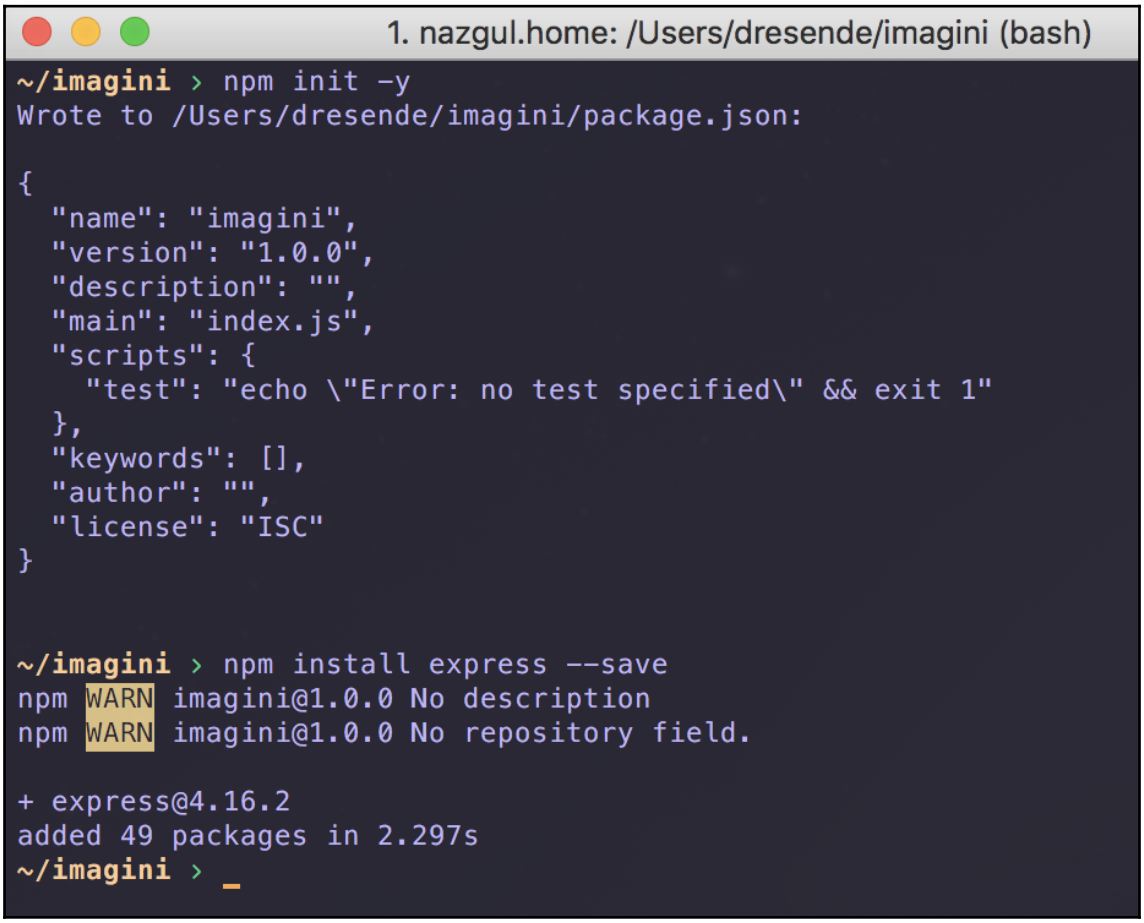
Let's create an image processing microservice. We'll start with a simple thumbnail service, and then we'll evolve to make some simple image transformations. We'll be covering how to:

- Build a microservice using Express
- Use external modules to manipulate images
- Build our previous microservice in Hydra and Seneca

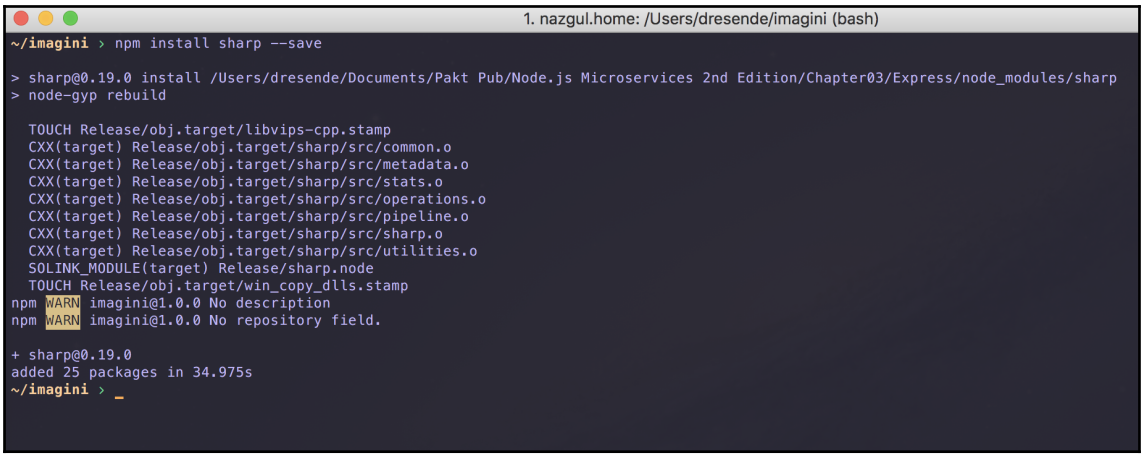
The microservice name is very important, as it gives identity. Let's name it *imagini*, the Latin name for image.

Using Express

As we've seen before, Express gives us a very simple but very useful layer on top of the default Node.js HTTP module. Let's create an empty folder, initialize our `package.json` file, and install Express:



To help us get around images, we'll use the `sharp` module, a very fast image manipulation tool for the modern web. It will allow us to easily transform and resize images in a simple serial interface. Let's just install it now:



Let's start by exposing an address to return thumbnails. We'll need to first define our default parameters. Later on, we'll allow the user to change the default values. Our service will run on port `3000` and will accept thumbnail requests on both PNG and JPEG formats. Here's our not-so-simple service:

```
const express = require("express");
const sharp   = require("sharp");
const app     = express();

app.get(/\/thumbnail\.(jpg|png)/, (req, res, next) => {
    let format    = (req.params[0] == "png" ? "png" : "jpeg");
    let width     = 300;
    let height    = 200;
    let border    = 5;
    let bgcolor   = "#fcfcfc";
    let fgcolor   = "#ddd";
    let textcolor = "#aaa";
    let textsize  = 24;
    let image     = sharp({
        create : {
            width      : width,
            height     : height,
            channels   : 4,
            background : { r: 0, g: 0, b: 0 },
        }
    });
```

```
    const thumbnail = new Buffer(
`<svg width="${width}" height="${height}">
    <rect
        x="0" y="0"
        width="${width}" height="${height}"
        fill="${fgcolor}" />
    <rect
        x="${border}" y="${border}"
        width="${width - border * 2}" height="${height - border * 2}"
        fill="${bgcolor}" />
    <line
        x1="${border * 2}" y1="${border * 2}"
        x2="${width - border * 2}" y2="${height - border * 2}"
        stroke-width="${border}" stroke="${fgcolor}" />
    <line
        x1="${width - border * 2}" y1="${border * 2}"
        x2="${border * 2}" y2="${height - border * 2}"
        stroke-width="${border}" stroke="${fgcolor}" />
    <rect
        x="${border}" y="${(height - textsize) / 2}"
        width="${width - border * 2}" height="${textsize}"
        fill="${bgcolor}" />
    <text
        x="${width / 2}" y="${height / 2}" dy="8"
        font-family="Helvetica" font-size="${textsize}"
        fill="${textcolor}" text-anchor="middle">${width} x
${height}</text>
</svg>`
    );

    image.overlayWith(thumbnail)[format]().pipe(res);
});

app.listen(3000, () => {
    console.log("ready");
});
```

We start by getting access to both `express` and `sharp` modules. We then initialize our Express application:

```
const express = require("express");
const sharp   = require("sharp");
const app     = express();
```

We then create a route to get an image, with a regular expression that will catch the address `/thumbnail.png` and `/thumbnail.jpg`. We'll use the extension to ascertain what image type the user wants. We define some default parameters, such as sizes and colors:

```
let format    = (req.params[0] == "png" ? "png" : "jpeg");
let width     = 300;
let height    = 200;
let border    = 5;
let bgcolor   = "#fcfcfc";
let fgcolor   = "#ddd";
let textcolor = "#aaa";
let textsize  = 24;
```

We then create an empty image using `sharp`:

```
let image = sharp({
    create : {
        width      : width,
        height     : height,
        channels   : 4,
        background : { r: 0, g: 0, b: 0 },
    }
});
```

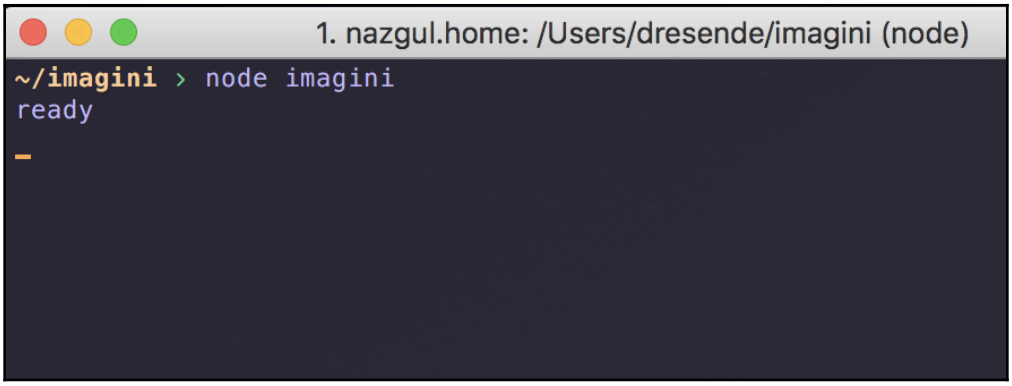
We then create an SVG file with an outer border, two crossing lines, and a text in the middle with the size of the image. Next, we overlay the SVG on our empty image, and output the result to the user:

```
image.overlayWith(thumbnail)[format]().pipe(res);
```

Finally, we initialize our service on port `3000`:

```
app.listen(3000, () => {
    console.log("ready");
});
```

Save the full code shown previously as `imagini.js`, and run it on the console:



You can now head to your web browser and just type the address of our service, and you'll see something like the following image:

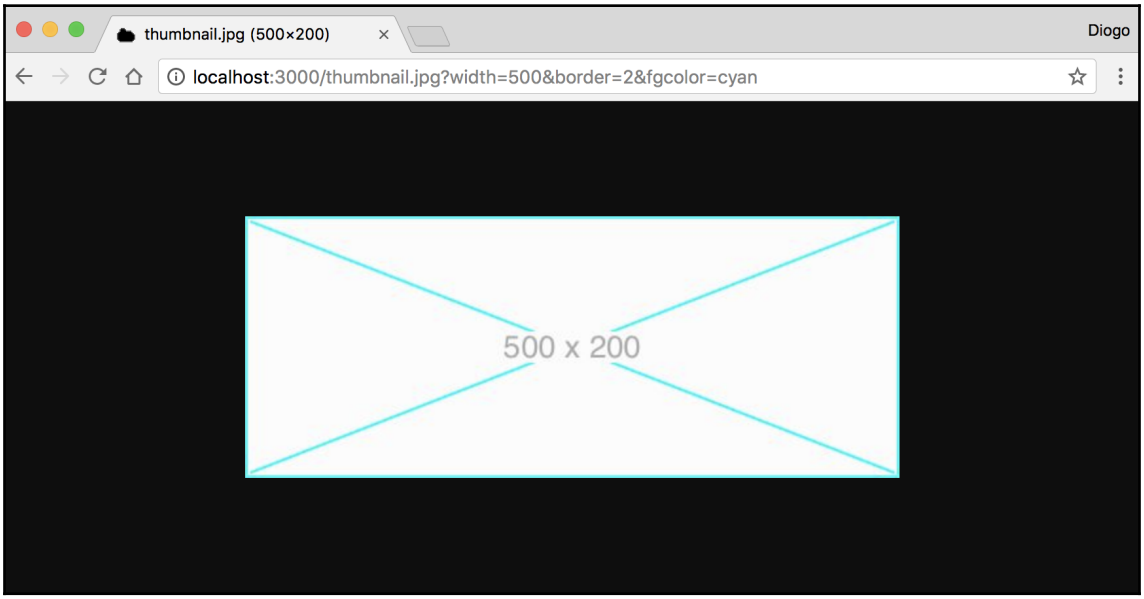
We can now start accepting some changes. Let's change our default variables to something like this:

```
let width     = +req.query.width || 300;
let height    = +req.query.height || 200;
let border    = +req.query.border || 5;
```

```
let bgcolor   = req.query.bgcolor || "#fcfcfc";
let fgcolor   = req.query.fgcolor || "#ddd";
let textcolor = req.query.textcolor || "#aaa";
let textsize  = +req.query.textsize || 24;
```

The preceding code is not safe for production. Use it only for demonstration purposes. We'll take a look at security in the next chapter.

Restart our service and play with query parameters to see the result. Here's an example of changing the width to 500 px, the border to 2 px, and the foreground color to cyan:



Well, it looks awesome, but it's not very useful as it is, as it just shows the same empty image. Though perhaps more applicable for prototyping, in the real world, it would be more useful if it could create thumbnails from images we upload. Let's do that.

To do so, we need to be able to:

- Upload images, which should be stored somewhere
- Check whether an image exists
- Download an image thumbnail

To simplify, we'll create dynamic routes in a specific path, and we'll use HTTP verbs (`GET`, `POST`, `DELETE`, and so on) to distinguish the actions. When uploading images, we'll use `POST` and the body should have the image data. To check if an image exists, we'll use `HEAD`. To download an image thumbnail, we'll use `GET`.

Uploading images

Let's first add our route to handle uploading images. We'll use a body parser module that will handle compressed requests automatically for us.

```
npm install body-parser --save
```

We'll include it along with other core modules that we'll need. Add these lines on the top of our `service` file:

```
const bodyparser = require("body-parser");
const path       = require("path");
const fs         = require("fs");
```

We can now create the function that will handle our upload:

```
app.post("/uploads/:image", bodyparser.raw({
    limit : "10mb",
    type  : "image/*"
}), (req, res) => {
    let image = req.params.image.toLowerCase();

    if (!image.match(/\.(png|jpg)$/)) {
        return res.status(403).end();
    }

    let len = req.body.length;
    let fd  = fs.createWriteStream(path.join(__dirname, "uploads",
    image), {
        flags    : "w+",
        encoding : "binary"
    });

    fd.write(req.body);
    fd.end();

    fd.on("close", () => {
        res.send({ status : "ok", size: len });
    });
});
```

We're expecting an HTTP `POST` on the `/uploads` path. It must be an image with a maximum size of 10 MB. Let's analyze the code in detail.

```
let image = req.params.image.toLowerCase();

if (!image.match(/\.(png|jpg)$/)) {
    return res.status(403).end();
}
```

We start by checking whether the image name passed ends with `.png` or `.jpg`. If not, we reply with an HTTP 403 response code, which means **forbidden**, as we're just accepting those types of images:

```
let len = req.body.length;
let fd  = fs.createWriteStream(path.join(__dirname, "uploads", image), {
    flags    : "w+",
    encoding : "binary"
});
```

We then create a stream to the local file where we'll save our image. The name of the file will be the name of the image. We also store the image size so we can return that to the user when we finish saving. This enables the microservice user to check if we received all the data.

```
fd.write(req.body);
fd.end();

fd.on("close", () => {
    res.send({ status : "ok", size: len });
});
```

Lastly, we write the image to file and, after properly closing the stream, we reply to the user with a JSON response with a status and a size property.

Don't forget to create the `uploads` folder inside the `microservice` folder. Then, restart the `service`. We'll continue to use `curl` to test our services. We'll need an image to try it out. I searched for a Google image and saved it locally. I then uploaded it to our service using this command:

```
curl -X POST -H 'Content-Type: image/png' \
    --data-binary @example.png \
    http://localhost:3000/uploads/example.png
```

We're telling `curl` that we want to:

- Send a `POST` request
- Define the `Content-Type` header saying it's a PNG image
- Add the content of the `example.png` file (I downloaded) inside the request body
- Send the request to the `/upload/example.png` path of our microservice

After executing the command, I received a JSON response. The size matches my file:

```
{ "status" : "ok", "size" : 55543 }
```

We can't see the image using our microservice just yet, but we can check it locally. You should have a copy of the file in the `uploads` folder. For reference, here's the image I'm using:

Checking an image exists in the folder

We can now create a route to check whether an image exists in our `uploads` folder. We'll use the HEAD verb. If you're not familiar with it, it's just like a `GET` request, but without a body (no content). It's used to request only information (headers) from a path.

```
app.head("/uploads/:image", (req, res) => {
    fs.access(
        path.join(__dirname, "uploads", req.params.image),
        fs.constants.R_OK ,
        (err) => {
            res.status(err ? 404 : 200);
            res.end();
        }
    );
});
```

We'll look for a similar route, but this time we're only handling HEAD requests. This is a simple check. We just question if the current process has read access to the local file.

```
fs.access(path, mode, callback);
```

If so, we'll reply with HTTP response code `200`, which means *Found*. If not, we'll reply with HTTP response code `404`, which means *Not Found*:

```
res.status(err ? 404 : 200);
```

Add the preceding code and restart the service.

If you use `curl` to check for our previously uploaded file, you'll receive something like this:

```
curl --head 'http://localhost:3000/uploads/example.png'

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Length: 0
Connection: keep-alive
```

If you change the path to something else, you should receive something like this:

```
curl --head 'http://localhost:3000/uploads/other.png'

HTTP/1.1 404 Not Found
X-Powered-By: Express
Connection: keep-alive
```

Downloading images

Now that we have uploaded our file and checked whether it's stored on the server, we want to be able to download it anytime. Let's create a route to download it. It will be just like the HEAD route we created, but now using the `GET` verb. Here's an example of how we can implement the route:

```
app.get("/uploads/:image", (req, res) => {
    let ext = path.extname(req.params.image);

    if (!ext.match(/^\.(png|jpg)$/)) {
        return res.status(404).end()
    }

    let fd = fs.createReadStream(path.join(__dirname, "uploads",
    req.params.image));
```

```
        fd.on("error", (e) => {
            if (e.code == "ENOENT") {
                return res.status(404).end()
            }

            res.status(500).end();
        });

        res.setHeader("Content-Type", "image/" + ext.substr(1));

        fd.pipe(res);
    });
```

We first start by checking the image extension:

```
let ext = path.extname(req.params.image);
```

If the extension is not `.png` or `.jpg`, we immediately return an HTTP response code `404`, which means *Not Found*:

```
if (!ext.match(/^\.(png|jpg)$/)) {
    return res.status(404).end();
}
```

If the extension is acceptable, we create a readable stream to the image file path:

```
let fd = fs.createReadStream(path.join(__dirname, "uploads",
req.params.image));
```

We then attach an error handler that will catch errors when reading the local file. If the error code is `ENOENT`, it means the file does not exist, so we again return an HTTP response code `404`. For any other error codes, we return an HTTP response code `500`, which means *Internal Server Error*:

```
fd.on("error", (e) => {
    if (e.code == "ENOENT") {
        return res.status(404).end()
    }

    res.status(500).end();
});
```

Returning code `500` is up to you. Perhaps you prefer to always return a `404` and hide the error differences from the user. That's a design decision; it depends on how you want your users to see your microservice and what target users will use it.

Then, we use the previously stored extension to set the content type returned to the user, in order to identify the image type we're sending. The content type should be in the format image/extension (without a dot):

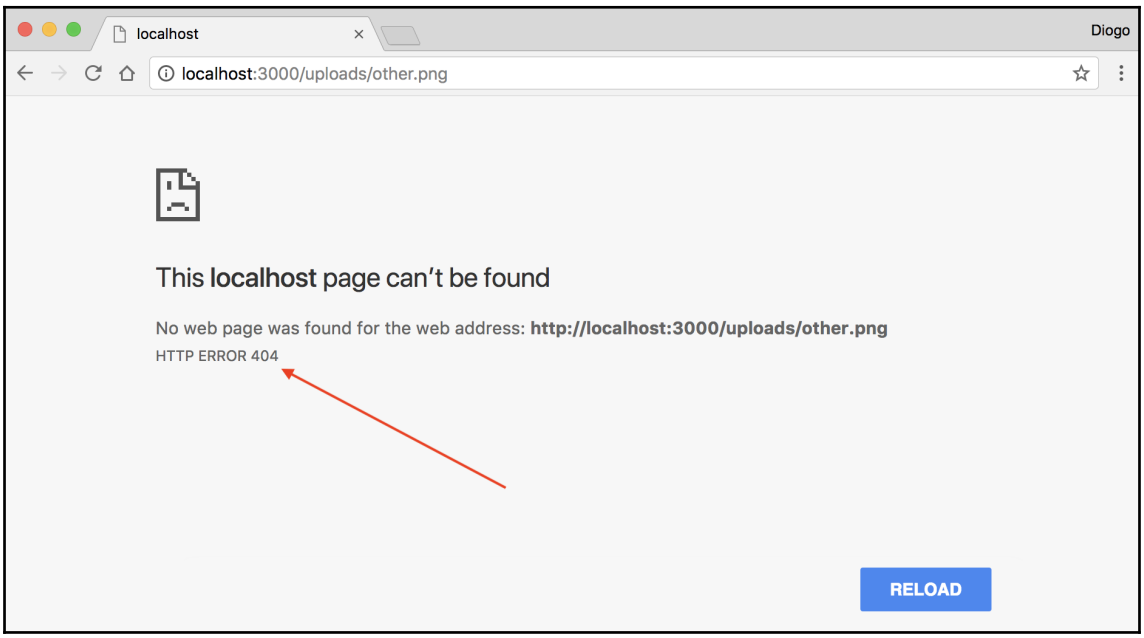
```
res.setHeader("Content-Type", "image/" + ext.substr(1));
```

Finally, we pipe the file content to the response. This is a helper method that will trigger reads from the file stream and write them to the response. When the whole file has been read, the response is ended automatically:

```
fd.pipe(res);
```

Let's try it out. Restart our service and open your web browser. If you point to the previously uploaded image, you should get a copy of it:

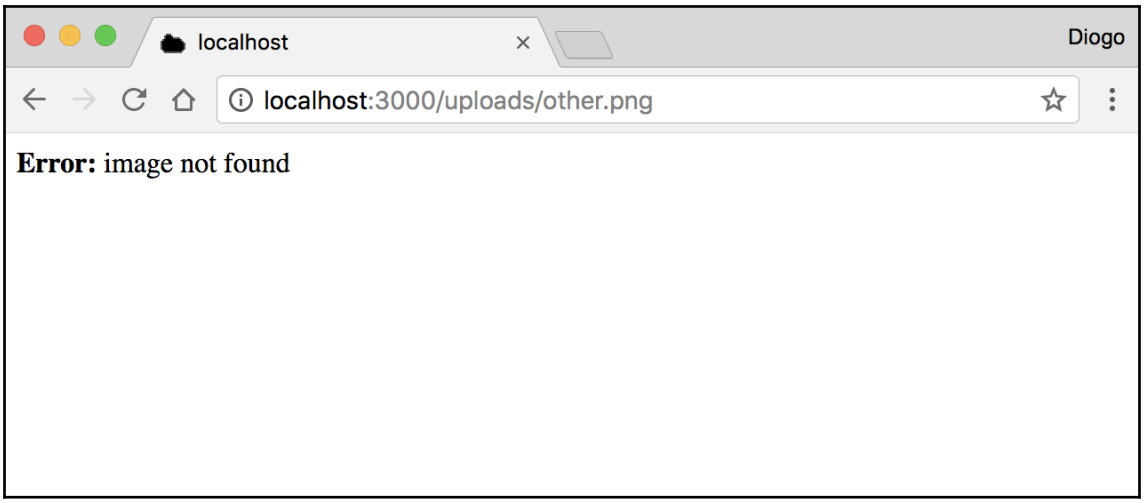
If you change the path to something that you haven't uploaded, you should get an **HTTP ERROR 404**:



This is not very appealing. We can change our error handler to check whether the request might accept HTML content and return a custom message:

```
fd.on("error", (e) => {
    if (e.code == "ENOENT") {
        res.status(404);

        if (req.accepts('html')) {
            res.setHeader("Content-Type", "text/html");

            res.write("<strong>Error:</strong> image not found");
        }

        return res.end();
    }

    res.status(500).end();
});
```

You should now receive a nicer response:



If you're familiar with Express, you could instead use a render engine and define a custom error template, or look for error handler modules that generate beautiful error pages.

Using route parameters

Now that we've written more than 100 lines of code, let's take a step back and look at it. There's certainly space for optimizations. Let's use another awesome feature of Express – route parameters. They allow you to preprocess any route that uses a parameter and check whether it's valid, and do all kinds of stuff such as fetching additional information from a database or another server.

We'll use it to validate our image name for now:

```
app.param("image", (req, res, next, image) => {
    if (!image.match(/\.(png|jpg)$/i)) {
        return res.status(req.method == "POST" ? 403 : 404).end();
    }

    req.image     = image;
    req.localpath = path.join(__dirname, "uploads", req.image);

    return next();
});
```

We start by checking whether the name matches a PNG or JPG image and, if not, we immediately reply with a `403` (for `POST` requests) or `404` error code (for anything else). If the name is acceptable, we store the image and the expected local path in the request object, for consulting it later.

We can now rewrite our routes, so let's start with our upload route:

```
app.post("/uploads/:image", bodyparser.raw({
    limit : "10mb",
    type  : "image/*"
}), (req, res) => {
    let fd  = fs.createWriteStream(req.localpath, {
        flags    : "w+",
        encoding : "binary"
    });

    fd.end(req.body);

    fd.on("close", () => {
        res.send({ status : "ok", size: req.body.length });
    });
});
```

Our initial check was completely removed since the parameter was pre-validated. Also, we can now use `req.localpath`. I also optimized code by removing an unnecessary variable allocation (`len`) and just using `.end()` by avoiding `.write()`.

```
app.head("/uploads/:image", (req, res) => {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        res.status(err ? 404 : 200).end();
    });
});
```

Our image check route looks almost exactly the same. This route had no security, so at least we introduced the pre-validation automatically. It's still far from acceptable for production, but it's a good start:

```
app.get("/uploads/:image", (req, res) => {
    let fd = fs.createReadStream(req.localpath);

    fd.on("error", (e) => {
        res.status(e.code == "ENOENT" ? 404 : 500).end();
    });

    res.setHeader("Content-Type", "image/" +
path.extname(req.image).substr(1));
```

```
    fd.pipe(res);
});
```

Our image download route got a bit smaller. I removed the fancy error to keep the service clean as it will most probably be used by other services and programs, not users directly.

The change we introduced centralizes the image parameter check. We can go a little further and see if the name has special characters that would allow a malicious user to fetch a file from a location other than our local uploads folder. Let's save that for the next chapter.

Generating thumbnails

We created upload and download routes, but they do nothing to the image other than storing it locally. Picking on our first thumbnail route, let's change our download route to be able to tell us the size we want for the image:

```
app.param("width", (req, res, next, width) => {
    req.width = +width;

    return next();
});

app.param("height", (req, res, next, height) => {
    req.height = +height;

    return next();
});
```

We start by defining two parameters, width, and height. These are numbers, so we'll assume they can be casted to a number type and stored in the request object.

Since we want to be able to specify sizes, let's create different routes using the route parameters we just introduced. To keep our code DRY, let's create a function to download the image and enable us to avoid repeating a very similar code. Our function will be able to handle optional resize:

```
function download_image(req, res) {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) return res.status(404).end();

        let image = sharp(req.localpath);

        if (req.width && req.height) {
            image.ignoreAspectRatio();
        }
```

```
            if (req.width || req.height) {
                image.resize(req.width, req.height);
            }

            res.setHeader("Content-Type", "image/" +
            path.extname(req.image).substr(1));

            image.pipe(res);
        });
    }
```

Our function first starts by checking whether the image exists. This will avoid the `sharp` module from throwing for non-existent images. We then initialize the image processing by passing its local path. We then do something interesting:

```
if (req.width && req.height) {
    image.ignoreAspectRatio();
}
```

First, if we receive both width and height, we tell `sharp` to ignore aspect ratio and resize as we tell it. Otherwise, it would maintain aspect ratio and center the image in the final size. You can check the difference by commenting out those first three lines:

```
if (req.width || req.height) {
    image.resize(req.width, req.height);
}
```

Second, if we receive one of the `width` or `height` parameters, we resize the image. We can resize with only one of the parameters. The `sharp` module is also able to resize using only one of the parameters if the other is undefined, so the line just works and will maintain aspect ratio.

We then just do what we did before: set the content type response header and send the image to the user. Now, we can create multiple routes for this function.

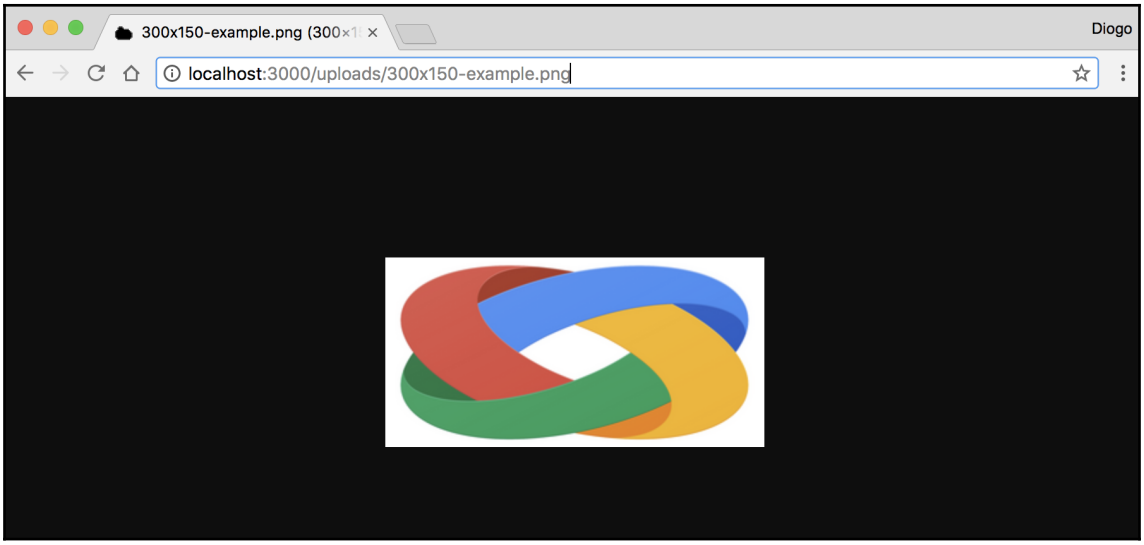
We'll allow three different download scenarios:

- A particular fixed-size image
- An aspect ratio resize by passing only width or height
- A full-size image

For the first scenario, we'll use our previous `width` and `height` parameters:

```
app.get("/uploads/:width(\\d+)x:height(\\d+)-:image", download_image);
```

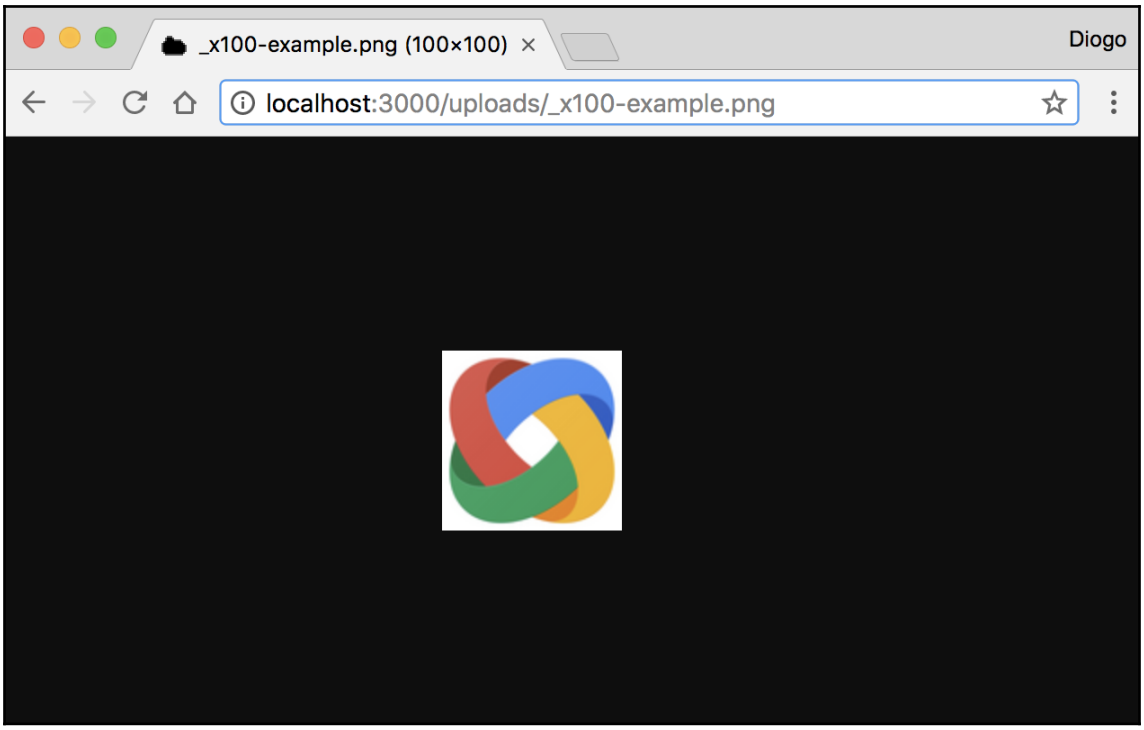
This will allow a route address similar to the following:



For the second scenario, we'll use similar routes, where the missing parameter is replaced by an underscore:

```
app.get("/uploads/_x:height(\\d+)-:image", download_image);
app.get("/uploads/:width(\\d+)x_-:image", download_image);
```

These will allow routes similar to the one as follows:



For the third and final scenario, we'll keep the same route:

```
app.get("/uploads/:image", download_image);
```

These four routes must be defined in this order. This is important because route parameters are based on regular expressions and they're greedy, so we need to define our more complicated routes first, and then the simpler ones.

Playing around with colors

Let's have some more fun and add a simple effect: `greyscale`. Let's add a parameter that will indicate to our service that the user wants the image in shades of grey:

```
app.param("greyscale", (req, res, next, greyscale) => {
    if (greyscale != "bw") return next("route");

    req.greyscale = true;
```

```
    return next();
});
```

Our parameter will only match the `bw` string. If it matches, it will mark a flag in the request object.

We can now change our image download function to handle this parameter, if defined:

```
function download_image(req, res) {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) return res.status(404).end();

        let image = sharp(req.localpath);

        if (req.width && req.height) {
            image.ignoreAspectRatio();
        }

        if (req.width || req.height) {
            image.resize(req.width, req.height);
        }

        if (req.greyscale) {
            image.greyscale();
        }

        res.setHeader("Content-Type", "image/" +
        path.extname(req.image).substr(1));

        image.pipe(res);
    });
}
```

Finally, for every one of our four download routes, we need to add another one to enable grey scaling. This means our user can take advantage of our resizing options and can `greyscale` the image if he prefers:

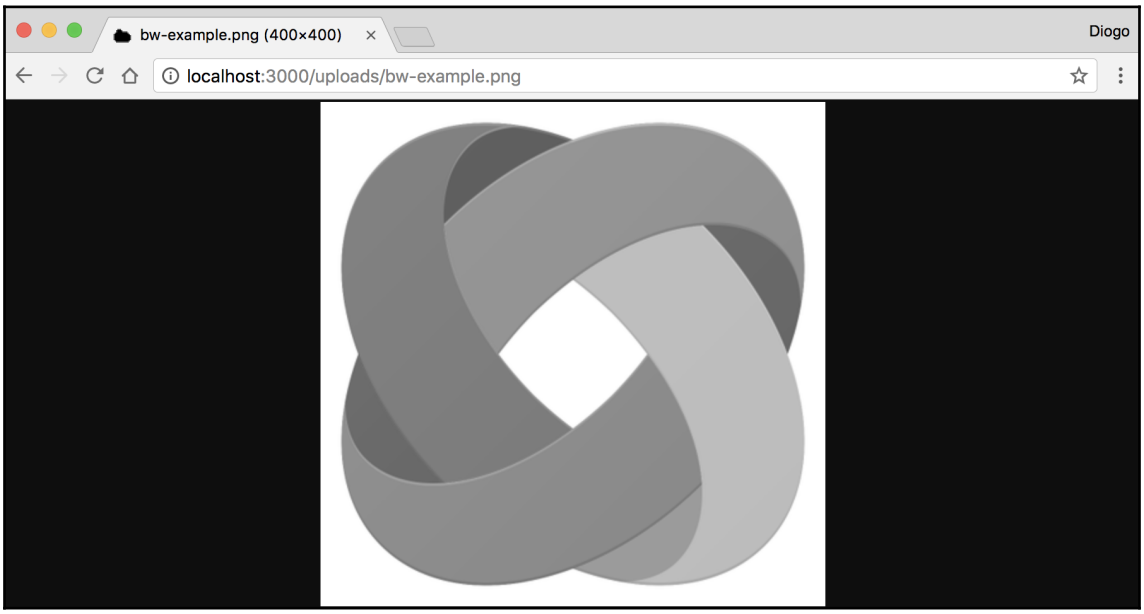
```
app.get("/uploads/:width(\\d+)x:height(\\d+)-:greyscale-:image",
download_image);
app.get("/uploads/:width(\\d+)x:height(\\d+)-:image", download_image);
app.get("/uploads/_x:height(\\d+)-:greyscale-:image", download_image);
app.get("/uploads/_x:height(\\d+)-:image", download_image);
app.get("/uploads/:width(\\d+)x_-:greyscale-:image", download_image);
app.get("/uploads/:width(\\d+)x_-:image", download_image);
app.get("/uploads/:greyscale-:image", download_image);
app.get("/uploads/:image", download_image);
```

For every one of our initial four routes, we added a similar route before that, with the greyscale parameter. The user just has to prefix the image name with `bw-` to automatically enable `greyscale`.

In the end, excluding our first generated thumbnail, you should have something similar to the following code. I removed the code blocks inside the routes and our download function, so you're able to see the base structure:

```
const bodyparser = require("body-parser");
const path       = require("path");
const fs         = require("fs");
const express    = require("express");
const sharp      = require("sharp");
const app        = express();

app.param("image", (req, res, next, image) => { ... });
app.param("width", (req, res, next, width) => { ... });
app.param("height", (req, res, next, height) => { ... });
app.param("greyscale", (req, res, next, greyscale) => { ... });

app.post("/uploads/:image", bodyparser.raw({ limit: "10mb", type: "image/*"
}), (req, res) => { ... });

app.head("/uploads/:image", (req, res) => { ... });

app.get("/uploads/:width(\\d+)x:height(\\d+)-:greyscale-:image",
download_image);
app.get("/uploads/:width(\\d+)x:height(\\d+)-:image", download_image);
app.get("/uploads/_x:height(\\d+)-:greyscale-:image", download_image);
app.get("/uploads/_x:height(\\d+)-:image", download_image);
app.get("/uploads/:width(\\d+)x_-:greyscale-:image", download_image);
app.get("/uploads/:width(\\d+)x_-:image", download_image);
app.get("/uploads/:greyscale-:image", download_image);
app.get("/uploads/:image", download_image);

app.listen(3000, () => {
    console.log("ready");
});

function download_image(req, res) {
    ...
}
```

You can now restart our service and make some tests. Taking our first image as an example, we can request a `greyscale` version:



You can still resize the image as we did before, with the optional `greyscale` filter:

We can take advantage of the `sharp` module and add some more interesting image manipulations for our service. This way, we can use it in more projects as it evolves to a full-featured image manipulation service.

We can start by adding a blurring option, something that will make the image seem unfocused. We can also do the opposite, and make the image sharper. We can also flip the image horizontally and vertically.

Refactor routes

These options look neat but, following the route we're doing, it will probably be a mess to create all the path possibilities. We're also narrowing on the image names as we're creating virtual prefixes for these image manipulations options.

Let's make a change while we're at the beginning. Instead of using route parameters, let's use query parameters. They can be added in the order we want, and don't affect the path. We could also use request headers, but these would be harder to test both on the browser and using fetching modules such as request.

So, before we add our neat image manipulations, let's refactor our code to use query parameters. Express handles and decodes query parameters automatically, so we just need to change our download `image` function:

```
function download_image(req, res) {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) return res.status(404).end();

        let image     = sharp(req.localpath);
        let width     = +req.query.width;
        let height    = +req.query.height;
        let greyscale = (req.query.greyscale == "y");

        if (width > 0 && height > 0) {
            image.ignoreAspectRatio();
        }

        if (width > 0 || height > 0) {
            image.resize(width || null, height || null);
        }

        if (greyscale) {
            image.greyscale();
        }

        res.setHeader("Content-Type", "image/" +
        path.extname(req.image).substr(1));

        image.pipe(res);
    });
}
```

We're not validating as we probably should, but we're ensuring the width and height are positive numbers. For greyscaling, we're checking whether the query parameter equals `y`. If you prefer to support more options on greyscaling such as `yes` or `true`, change the line to something like the following:

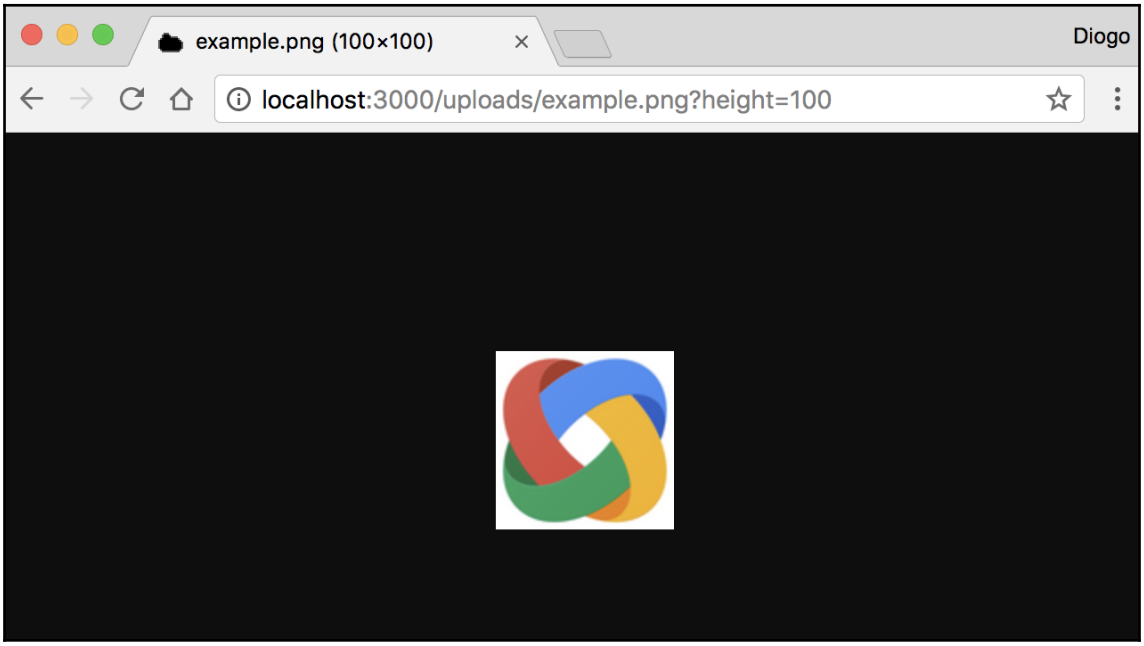
```
let greyscale = [ "y", "yes", "1", "on"].includes(req.query.greyscale);
```

Now, we can drop the `width`, `height`, and `greyscale` route parameters we added before, and just keep the `image` parameter. Also, all our download routes can be removed, and we just keep the simpler one. In the end, we'll have something like this:

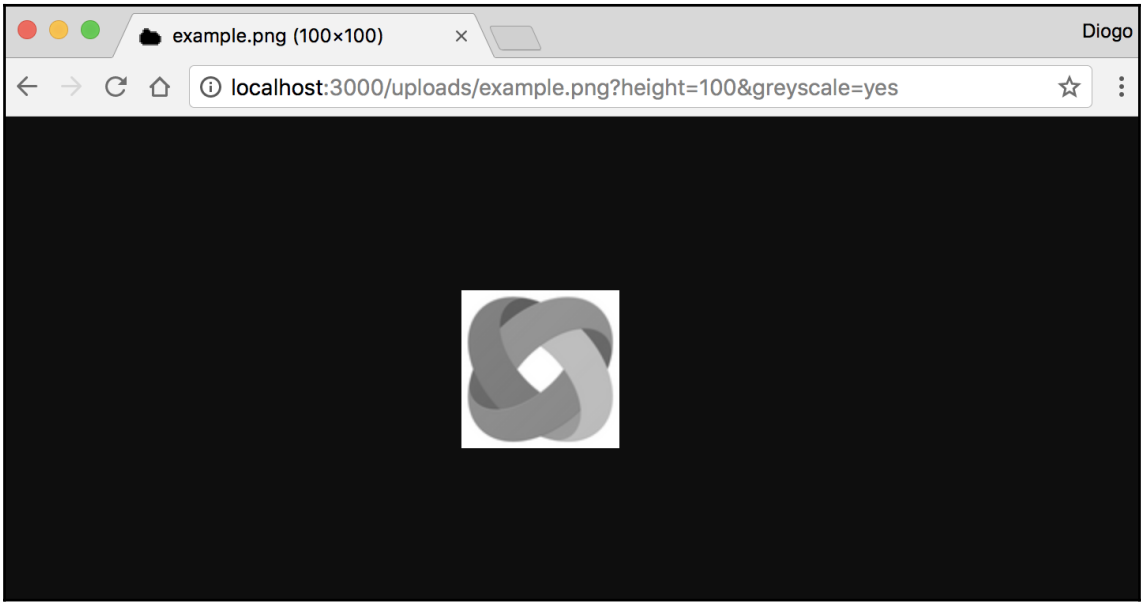
```
const bodyparser = require("body-parser");
const path       = require("path");
const fs         = require("fs");
const express    = require("express");
const sharp      = require("sharp");
const app        = express();

app.param("image", (req, res, next, image) => {
    if (!image.match(/\.(png|jpg)$/i)) {
        return res.status(req.method == "POST" ? 403 : 404).end();
    }

    req.image     = image;
    req.localpath = path.join(__dirname, "uploads", req.image);

    return next();
});

app.post("/uploads/:image", bodyparser.raw({
    limit : "10mb",
    type  : "image/*"
}), (req, res) => {
    let fd  = fs.createWriteStream(req.localpath, {
        flags    : "w+",
        encoding : "binary"
    });

    fd.end(req.body);

    fd.on("close", () => {
        res.send({ status : "ok", size: req.body.length });
    });
});

app.head("/uploads/:image", (req, res) => {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
```

```
            res.status(err ? 404 : 200).end();
        });
    });

    app.get("/uploads/:image", download_image);

    app.listen(3000, () => {
        console.log("ready");
    });
```

Let's try it out now. You should be able to do a resize like we did before, but with a different address:



We can also add the `greyscale` function to it:



Manipulating images

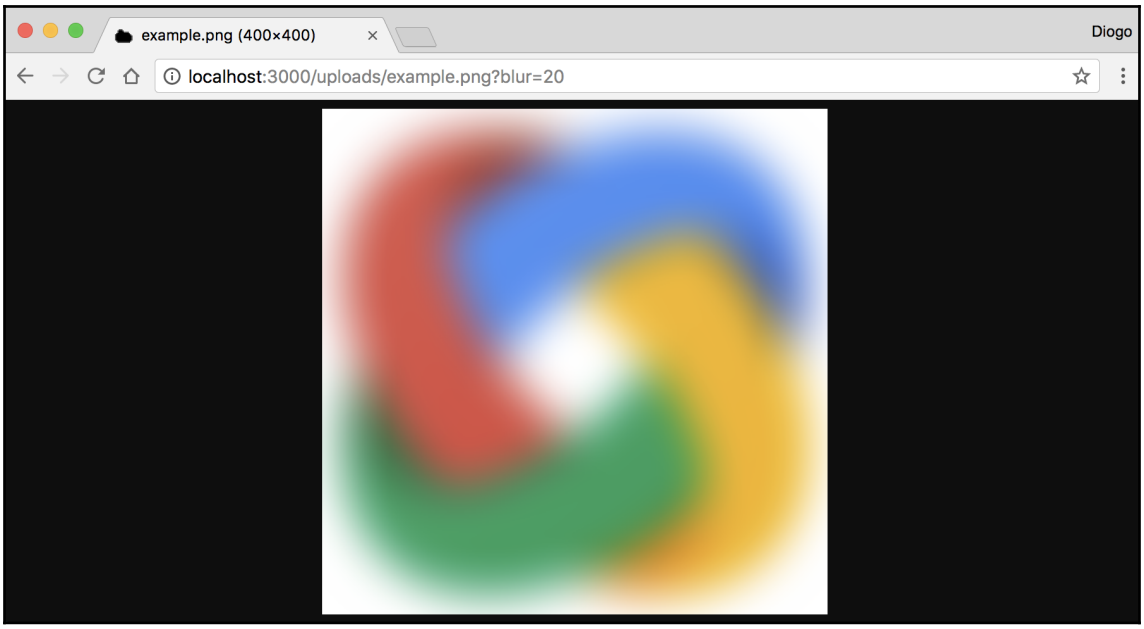
Let's add the other image manipulation options we mentioned before. Let's use the Sharp naming for flipping. We'll use the query parameter `flip` to flip the image vertically, and the query parameter `flop` to flip it horizontally. We'll also add the optional query parameters `blur` and `sharpen`, which should be positive numbers.

Our image download function can be reintegrated directly in our download route, since we only have one route now. In the end, we would end up with something like this:

```
app.get("/uploads/:image", (req, res) => {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) return res.status(404).end();

        let image     = sharp(req.localpath);
        let width     = +req.query.width;
        let height    = +req.query.height;
        let blur      = +req.query.blur;
        let sharpen   = +req.query.sharpen;
        let greyscale = [ "y", "yes", "1",

       "on"].includes(req.query.greyscale);
        let flip      = [ "y", "yes", "1",
        "on"].includes(req.query.flip);
        let flop      = [ "y", "yes", "1",
        "on"].includes(req.query.flop);

        if (width > 0 && height > 0) {
            image.ignoreAspectRatio();
        }

        if (width > 0 || height > 0) {
            image.resize(width || null, height || null);
        }

        if (flip)         image.flip();
        if (flop)         image.flop();
        if (blur > 0)     image.blur(blur);
        if (sharpen > 0) image.sharpen(sharpen);
        if (greyscale)    image.greyscale();

        res.setHeader("Content-Type", "image/" +
        path.extname(req.image).substr(1));

        image.pipe(res);
    });
});
```

We should be able to play with it now; flipping and sharpening the image leads to something like this:

Blurring the image leads to something similar to this:



Well, that looks awesome for a first iteration. Let's see what it would look like if we use Hydra instead.

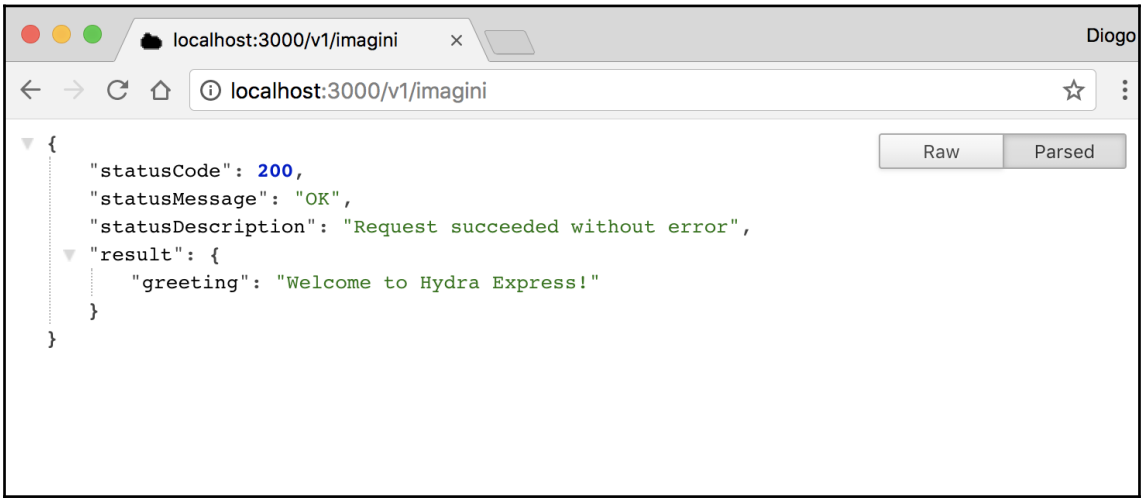
Using Hydra

As you may remember, Hydra has a scaffolding command that helps to bootstrap our service quickly. Let's use it, and prepare our base layout. Run `yo fwsp-hydra` and answer the questions. You can leave most of them as default. Depending on the versions you'll use, you should get something similar to the lines shown here:

```
fwsp-hydra generator v0.3.1   yeoman-generator v2.0.2   yo v2.0.1
? Name of the service (`-service` will be appended automatically) imagini
? Your full name? Diogo Resende
? Your email address? dresende@thinkdigital.pt
? Your organization or username? (used to tag docker images) dresende
? Host the service runs on?
? Port the service runs on? 3000
? What does this service do? Image thumbnail and manipulation
? Does this service need auth? No
? Is this a hydra-express service? Yes
? Set up a view engine? No
? Set up logging? No
? Enable CORS on serverResponses? No
? Run npm install? No
   create imagini-service/specs/test.js
   create imagini-service/specs/helpers/chai.js
   create imagini-service/.editorconfig
   create imagini-service/.eslintrc
   create imagini-service/.gitattributes
   create imagini-service/.nvmrc
   create imagini-service/.gitignore
   create imagini-service/package.json
   create imagini-service/README.md
   create imagini-service/imagini-service.js
   create imagini-service/config/sample-config.json
   create imagini-service/config/config.json
   create imagini-service/scripts/docker.js
   create imagini-service/routes/imagini-v1-routes.js

Done!

'cd imagini-service' then 'npm install' and 'npm start'
```

Well, let's do just that. Let's enter our service folder and install dependencies. If you then start it using `npm start`, and open the browser and point it to our service, you should get something like this:

Not surprising, because Hydra created a different base route. To enable versioning and having different services running on the same HTTP backend, Hydra scaffolding created a route under the `/v1/imagini` prefix. Remember, we scaffolded Hydra with Express integration, so many of the terms we discussed earlier will be the same here:



Before we pick our previous code and integrate into Hydra, we need to add our Sharp dependency to `package.json`. Look up the `dependencies` property and add `sharp`. You should end up with something along these lines:

```
(...)
"dependencies": {
    "sharp"                : "^0.19.0",
    "body-parser"          : "^1.18.2",
    "fwsp-config"          : "1.1.5",
    "hydra-express"        : "1.5.5",
    "fwsp-server-response" : "2.2.6"
},
(...)
```

Now, run `npm install` to install Sharp. Then, open the `imagini-v1-routes.js` file, which is under the `routes` folder. Basically, what it does is get a handler for Hydra and Express, prepare a generic JSON server response (that's what the `fwsp-server-response` module is), create an Express router, attach the `/` route, and then export it.

We'll keep this structure for now. I refactored the file as I'm a bit picky about indentation and quotes. I added our image route parameter and added the image upload route. I changed our previous route code to drop the `/uploads` route prefix, and use the new `sendOk` and `sendError` functions you see in the preceding code:

```
/**
 * @name          imagini-v1-api
 * @description   This module packages the Imagini API.
 */
"use strict";

const fs             = require("fs");
const path           = require("path");
const sharp          = require("sharp");
const bodyparser     = require("body-parser");
const hydraExpress   = require("hydra-express");
const ServerResponse = require("fwsp-server-response");
const hydra          = hydraExpress.getHydra();
const express        = hydraExpress.getExpress();

let serverResponse = new ServerResponse();

express.response.sendError = function (err) {
    serverResponse.sendServerError(this, { result : { error : err }});
};

express.response.sendOk = function (result) {
```

```
    serverResponse.sendOk(this, { result });
};

let api = express.Router();

api.param("image", (req, res, next, image) => {
    if (!image.match(/\.(png|jpg)$/i)) {
        return res.sendError("invalid image type/extension");
    }

    req.image     = image;
    req.localpath = path.join(__dirname, "../uploads", req.image);

    return next();
});

api.post("/:image", bodyparser.raw({
    limit : "10mb",
    type  : "image/*"
}), (req, res) => {
    let fd  = fs.createWriteStream(req.localpath, {
        flags    : "w+",
        encoding : "binary"
    });

    fd.end(req.body);

    fd.on("close", () => {
        res.sendOk({ size: req.body.length });
    });
});

module.exports = api;
```

Then, we restart our microservice, create the `uploads` folder under the `imagini-service` folder, and try to upload an image. Like before, I used `curl` to test it:

```
curl -X POST -H 'Content-Type: image/png' \
    --data-binary @example.png \
    http://localhost:3000/v1/imagini/example.png
```

As expected, I received a JSON response with our `size` property:

```
{
    "statusCode"        : 200,
    "statusMessage"     : "OK",
    "statusDescription" : "Request succeeded without error",
    "result" : {
```

```
            "size" : 55543
        }
    }
```

We can have our uploaded file in our `uploads` folder. We're getting there; just two more routes:

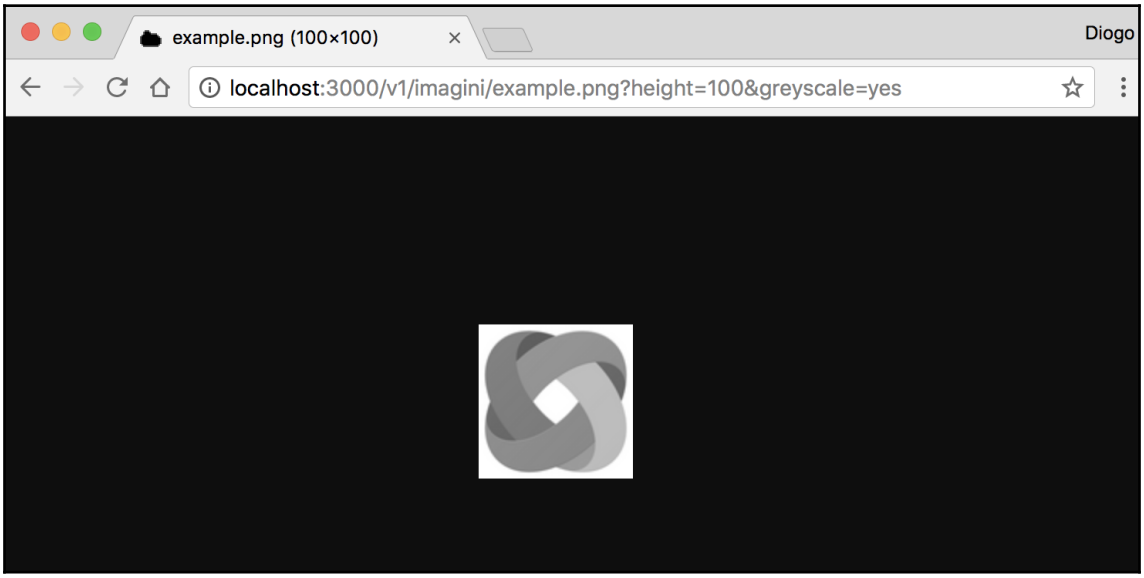
```
api.head("/:image", (req, res) => {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) {
            return res.sendError("image not found");
        }

        return res.sendOk();
    });
});
```

Our check route is very similar. We just changed the return methods to use the methods defined previously:

```
api.get("/:image", (req, res) => {
    fs.access(req.localpath, fs.constants.R_OK , (err) => {
        if (err) {
            return res.sendError("image not found");
        }

        let image     = sharp(req.localpath);
        let width     = +req.query.width;
        let height    = +req.query.height;
        let blur      = +req.query.blur;
        let sharpen   = +req.query.sharpen;
        let greyscale = [ "y", "yes", "true", "1",
        "on"].includes(req.query.greyscale);
        let flip      = [ "y", "yes", "true", "1",
        "on"].includes(req.query.flip);
        let flop      = [ "y", "yes", "true", "1",
        "on"].includes(req.query.flop);

        if (width > 0 && height > 0) {
            image.ignoreAspectRatio();
        }

        if (width > 0 || height > 0) {
            image.resize(width || null, height || null);
        }

        if (flip)        image.flip();
        if (flop)        image.flop();
```

```
            if (blur > 0)     image.blur(blur);
            if (sharpen > 0)  image.sharpen(sharpen);
            if (greyscale)    image.greyscale();

            res.setHeader("Content-Type", "image/" +
            path.extname(req.image).substr(1));

            image.pipe(res);
        });
    });
```

Our image download method is equally similar. For this route, we're not using the JSON responses, and instead just return our image directly. This allows us to try it out on our browser:



We just migrated our service from Express to Hydra. Not much of a difference, but Hydra gives you a more robust layout, about which we'll find out more later on. Let's take a look at our third framework: Seneca.

Using Seneca

Remember that routing on this framework is all about patterns. Let's keep it simple for now and use a role property to indicate what we want to do (upload, check, or download).

By default, every message should be JSON-encoded, so we'll encode the image in `base64` to pass it as a string inside the JSON messages to upload and download.

Create a folder for our Seneca service, and then create a child folder called `uploads`. Then, install `seneca` and `sharp` on that folder by running the following command:

```
npm install seneca sharp --save
```

Then, create a file called `imagini.js` with the following content:

```
const seneca  = require("seneca");
const sharp   = require("sharp");
const path    = require("path");
const fs      = require("fs");
const service = seneca();

service.add("role:upload,image:*,data:*", function (msg, next) {
    let filename = path.join(__dirname, "uploads", msg.image);
    let data = Buffer.from(msg.data, "base64");

    fs.writeFile(filename, data, (err) => {
        if (err) return next(err);

        return next(null, { size : data.length });
    });
});

service.listen(3000);
```

What this does is initiate a simple service with a route for uploading. Since we're receiving all the image content directly on an object property, I used `fs.writeFile`. It's a simpler method and gives me an error whether something incorrect happens, which we can pass on to the route response.

I also used `Buffer.from` to convert our image data, which we'll be uploading in `base64`.

So, let's just start it as we did with the others. I included the same `example.png` image and used `curl` to test this out.

```
curl -H "Content-Type: application/json" \
--data '{"role":"upload","image":"example.png","data":"'"$( base64
```

```
example.png)"'"}' \
http://localhost:3000/act
```

Seneca promptly replied with the following:

```
{"size":55543}
```

This is the image size. Notice I'm taking advantage of bash interpolation (variable substitution) to directly convert the image file to `base64` and pass it to `curl`, which then sends that JSON piece of data to our service:

```
service.add("role:check,image:*", function (msg, next) {
    let filename = path.join(__dirname, "uploads", msg.image);

    fs.access(filename, fs.constants.R_OK , (err) => {
        return next(null, { exists : !err });
    });
});
```

Our check route is very similar. Instead of just replying with an HTTP 404 response code, we reply with a stringified JSON object with a boolean property exists, which will indicate if the image was found.

Here, we are checking for our image using `curl`:

```
curl -H "Content-Type: application/json" \
    --data '{"role":"check","image":"example.png"}' \
    http://localhost:3000/act
```

We will respond with the following:

```
{"exists":true}
```

If you change the image name, it will respond with `false`:

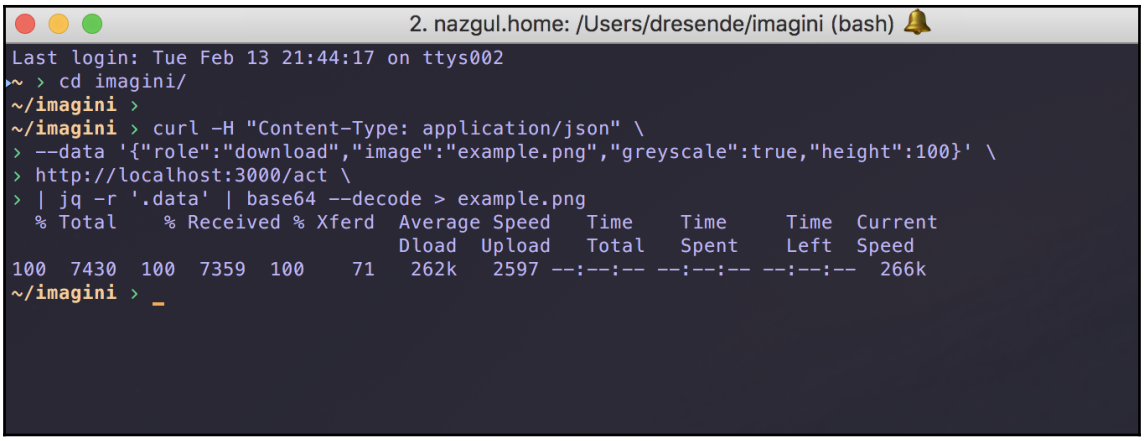
```
service.add("role:download,image:*", function (msg, next) {
    let filename = path.join(__dirname, "uploads", msg.image);

    fs.access(filename, fs.constants.R_OK , (err) => {
        if (err) return next(err);

        let image     = sharp(filename);
        let width     = +msg.width;
        let height    = +msg.height;
        let blur      = +msg.blur;
        let sharpen   = +msg.sharpen;
        let greyscale = !!msg.greyscale;
        let flip      = !!msg.flip;
```

```
            let flop      = !!msg.flop;

            if (width > 0 && height > 0) {
                image.ignoreAspectRatio();
            }

            if (width > 0 || height > 0) {
                image.resize(width || null, height || null);
            }

            if (flip)         image.flip();
            if (flop)         image.flop();
            if (blur > 0)     image.blur(blur);
            if (sharpen > 0)  image.sharpen(sharpen);
            if (greyscale)    image.greyscale();

            image.toBuffer().then((data) => {
                return next(null, { data: data.toString("base64") });
            });
        });
    });
```

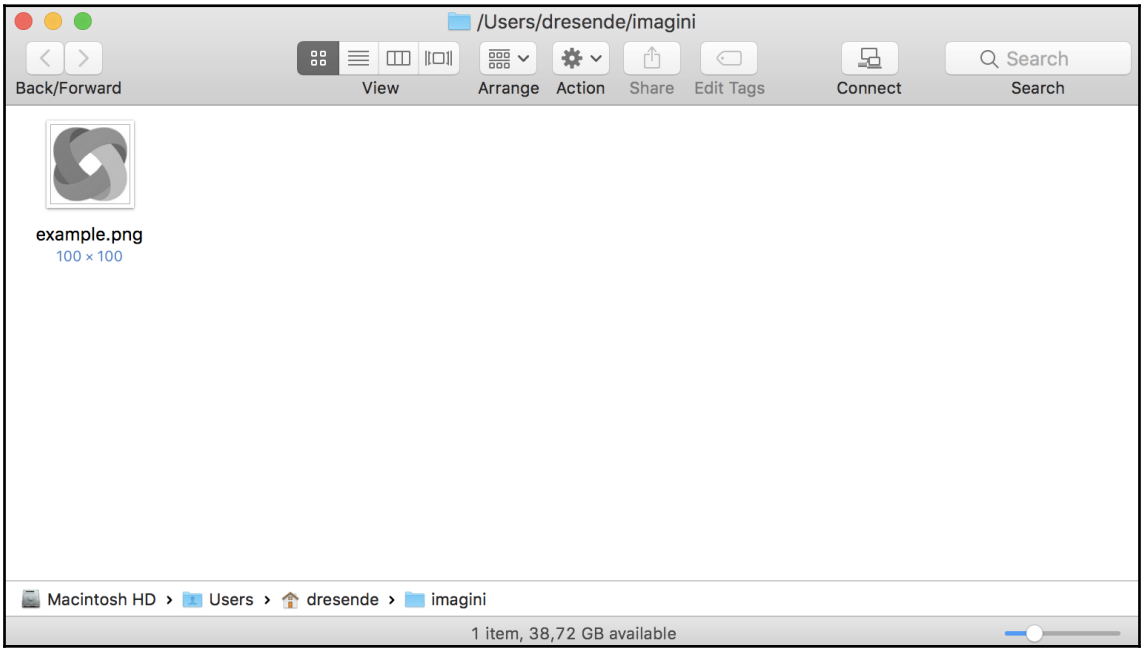
Our downloaded route has some changes:

- Instead of query parameters, we check directly on `msg`. One noticeable advantage is that we have types and not just strings, so we can use Boolean and numbers directly.
- Instead of returning the image in binary, so that we could just open in the browser, we convert it to `base64` and pass it on the JSON response.

We need some tools to test this on the command line. Since I use JSON a lot, I have `jq` installed. I strongly recommend you install it too and take a look at the tutorials. It will make your life easier. Using the `base64` command we previously used to encode, we can decode the content and pipe the data to a local file:



We can then open the folder and see the image is there. Notice I added `greyscale` and resized the image just by passing two more JSON parameters:



Plugins

In the spirit of Seneca, we should make a plugin for our `imagini` service. Let's split our code into two parts:

- The `imagini` plugin, a service that manipulates images
- A Seneca microservice, which exposes the `imagini` plugin, and possibly others later on

There's lot of room for improvement on our code, starting with code we repeated constantly. It's important to detect repetitions while our service is still very small.

The most repeated part is the local filename. This is actually something you might want to configure when starting the service, so let's change that to a function. Start by changing our `imagini.js` file to be a plugin. Clear all content and write this code:

```
const sharp   = require("sharp");
const path    = require("path");
const fs      = require("fs");

module.exports = function (settings = { path: "uploads" }) {
    // plugin code goes here
};
```

This is the basis of our plugin. We're loading the modules we need, but not Seneca, because our plugin will have access to the service directly. Seneca itself will load the plugin by calling our exported function. Following the idea of being able to configure the local image folder, we define an optional `settings` parameter, which will default to an object with the property `path` equal to `uploads`, which is the folder we've been using so far.

Now, let's add the content of our plugin, inside the preceding function:

```
const localpath = (image) => {
    return path.join(settings.path, image);
}
```

We start by defining a function that will convert our image parameter to the local path. We can actually reduce the function to be written in one single line:

```
const localpath = (image) => (path.join(settings.path, image));
```

Then, let's create another function that will check whether we have access to a local file, and return a Boolean (if it exists or not) and the filename we provided:

```
const access = (filename, next) => {
    fs.access(filename, fs.constants.R_OK , (err) => {
        return next(!err, filename);
    });
};
```

We can use this for our image check, and for our image download. This way, we can improve or even cache the results for greater performance, avoiding excessive filesystem hits. Our image check route can now be written in a very concise way:

```
this.add("role:check,image:*", (msg, next) => {
    access(localpath(msg.image), (exists) => {
        return next(null, { exists : exists });
    });
});
```

Notice that we're referring to the `this` object. Our Seneca service will call our plugin function and reference itself to `this`. Again, we can write it in a more concise way:

```
this.add("role:check,image:*", (msg, next) => {
    access(localpath(msg.image), (exists) => (next(null, { exists })));
});
```

Our upload route is fairly simple and has no changes:

```
this.add("role:upload,image:*,data:*", (msg, next) => {
    let data = Buffer.from(msg.data, "base64");

    fs.writeFile(localpath(msg.image), data, (err) => {
        return next(err, { size : data.length });
    });
});
```

The download route uses our previously created helper functions to avoid storing our local filename. We also made some tweaks to how `width` and `height` were treated:

```
this.add("role:download,image:*", (msg, next) => {
    access(localpath(msg.image), (exists, filename) => {
        if (!exists) return next(new Error("image not found"));

        let image     = sharp(filename);
        let width     = +msg.width || null;
        let height    = +msg.height || null;
        let blur      = +msg.blur;
        let sharpen   = +msg.sharpen;
```

```
        let greyscale = !!msg.greyscale;
        let flip      = !!msg.flip;
        let flop      = !!msg.flop;

        if (width && height) image.ignoreAspectRatio();
        if (width || height) image.resize(width, height);
        if (flip)            image.flip();
        if (flop)            image.flop();
        if (blur > 0)        image.blur(blur);
        if (sharpen > 0)     image.sharpen(sharpen);
        if (greyscale)       image.greyscale();

        image.toBuffer().then((data) => {
            return next(null, { data: data.toString("base64") });
        });
    });
});
```

There are actually a lot of variables we're using where we could just check the message parameter instead. We can rewrite our download function and get one-third reduction:

```
this.add("role:download,image:*", (msg, next) => {
    access(localpath(msg.image), (exists, filename) => {
        if (!exists) return next(new Error("image not found"));

        let image     = sharp(filename);
        let width     = +msg.width || null;
        let height    = +msg.height || null;

        if (width && height) image.ignoreAspectRatio();
        if (width || height) image.resize(width, height);
        if (msg.flip)        image.flip();
        if (msg.flop)        image.flop();
        if (msg.blur > 0)    image.blur(blur);
        if (msg.sharpen > 0) image.sharpen(sharpen);
        if (msg.greyscale)   image.greyscale();

        image.toBuffer().then((data) => {
            return next(null, { data: data.toString("base64") });
        });
    });
});
```

In the end, you should have an `imagini.js` file with the following content:

```
const sharp   = require("sharp");
const path    = require("path");
const fs      = require("fs");
```

```
module.exports = function (settings = { path: "uploads" }) {
    const localpath = (image) => (path.join(settings.path, image));
    const access    = (filename, next) => {
        fs.access(filename, fs.constants.R_OK , (err) => {
            return next(!err, filename);
        });
    };

    this.add("role:check,image:*", (msg, next) => {
        access(localpath(msg.image), (exists) => (next(null, { exists })));
    });

    this.add("role:upload,image:*,data:*", (msg, next) => {
        let data = Buffer.from(msg.data, "base64");

        fs.writeFile(localpath(msg.image), data, (err) => {
            return next(err, { size : data.length });
        });
    });

    this.add("role:download,image:*", (msg, next) => {
        access(localpath(msg.image), (exists, filename) => {
            if (!exists) return next(new Error("image not found"));

            let image     = sharp(filename);
            let width     = +msg.width || null;
            let height    = +msg.height || null;

            if (width && height) image.ignoreAspectRatio();
            if (width || height) image.resize(width, height);
            if (msg.flip)        image.flip();
            if (msg.flop)        image.flop();
            if (msg.blur > 0)    image.blur(blur);
            if (msg.sharpen > 0) image.sharpen(sharpen);
            if (msg.greyscale)   image.greyscale();

            image.toBuffer().then((data) => {
                return next(null, { data: data.toString("base64") });
            });
        });
    });
};
```

We just need to create our Seneca service and use our plugin. This is actually very straightforward. Create a file called `seneca.js`, and add the following:

```
const seneca  = require("seneca");
const service = seneca();

service.use("./imagini.js", { path: __dirname + "/uploads" });

service.listen(3000);
```

What the code does, line by line, is as follows:

1. Loads the `seneca` module
2. Creates a Seneca `service`
3. Loads the `imagini.js` plugin and passes our desired path
4. Starts `service` on port `3000`

That's it, our service is now a plugin and could be used by any Seneca service! You should now start our `service` by running the new file and not `imagini.js` directly:

```
node seneca
```

Summary

As you can see, writing a service does not change much between frameworks. Our code is very similar, with minor changes. Seneca is stricter about message format and content, so we used `base64` to encode our image inside the JSON message. Other than that, everything is the same.

For tools such as Express, which is a broader tool in terms of doing everything you might need, and not just microservices, you'll need to find the right plugins, and perhaps write several more plugins, in order to help you reach a production-ready microservice. You get the advantage of being able to choose everything about the service, but you need to write a lot of code. To facilitate the task, Hydra might be a good start for an initial set of plugins.

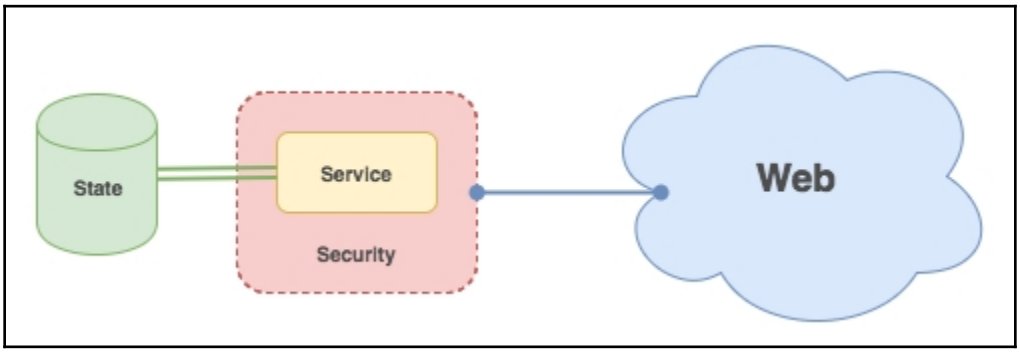
For other tools such as Seneca, some aspects of the microservice (for instance, the communication using JSON messages and service composition) are already packed. This comes at the price of a stricter service definition.

In the next chapter, we will discuss the security of our three service versions, and how we can store state.

6
State and Security

Now that we have created our microservice base layout on different frameworks, it's time to read our code more carefully and see if everything looks good. It's easy to just keep writing code and stop thinking about what we're doing, but later on, when we do stop for a while, we'll be wasting time deleting repeated code and reorganizing our service.

It's always better to think before we code. This is something you'll learn with time, to value the time you dedicate to plan your service or think about a new feature. It's never a good idea to just start coding. In theory, your **Service** should be inside a **Security** layer, with a good and stable connection to **State**:



State

Think of state as a person's memory. Usually, a service has state, which means it has memory of actions and information it's serving. The idea is that our service will run indefinitely, but sometimes we're forced to restart it or even stop it for some time because of maintenance or an upgrade.

Ideally, a service should resume without losing state, giving its users the perception that it never stopped. This is achieved by doing one of two things:

- Having state stored in a persistent storage
- Saving state in a persistent storage before stopping and loading that state after restarting

The first option will make your service a bit slower (nothing is faster than state in system memory) but should give you a more consistent state across restarts.

The second option is trickier, because sometimes our service might stop abruptly and be unable to save that state, but for those use cases, you may not care about the state. It's up to you.

There are a lot of options to store state; it will depend on what you want to store. For a microservice, you should avoid the filesystem so as to make your service more compatible with multiple operating systems.

Storing state

Depending on your service, you can store state using:

- A **relational database management system** (**RDBMS**), such as MySQL or PostgreSQL
- A **non-relational database management system**, or NoSQL, such as MongoDB or RethinkDB
- An **in-memory database** (**IMDB**), such as Redis or Memcached

The first option is still the most commonly used one. You'll rely on stable and more than proven database systems that run in multiple systems and that you can find on any cloud service where you might want to deploy your microservice. Apart from the maturity of most solutions, a relational database, if properly set up, should give you consistency.

The second option is more recent compared to the first one. Usually, there are no fixed tables as in RDBMS, and you normally work with collections of documents that are just common JSON structures. It's more agile as there are usually no restrictions, and each document might have a different structure. The more agile, the less consistent.

All three options, depending on the specific system you choose, support replication, which should enable fault tolerance and improve speed in geographically spaced instances.

Let's try each of the three options using one of the suggested systems. Let's begin with the relational database, and use MySQL.

MySQL

Installing MySQL is very simple. Just head to the official website and follow the instructions. You'll usually be asked for a password for the root user, which you can use later on to manage the server settings and user accounts.

There are some options to connect to a MySQL server using Node.js, but the best tools are the `mysql` and `mysql2` modules. They both serve the required purpose, and neither is the next version of the other, they're just a bit different in design and supported features.

First, let's add the dependency to our service. On the terminal, go to our service folder and type:

```
npm install mysql --save
```

We can now include our dependency and configure a connection to the database. To avoid having the credentials in our code, we can create a separate file and put settings there that we may change in the future, and that shouldn't belong in the code. We can take advantage of Node.js being able to include JSON files, and just write our settings in JSON.

Create a file called `settings.json`, and add the following content:

```
{
    "db": "mysql://root:test@localhost/imagini"
}
```

We defined a setting called `db` that has a database URI, which is a handy way of defining our database access and credentials using an address similar to any website address. Our database uses `mysql`; it's at `localhost` (using the default port), which can be accessed using the username `root` and the password `test`, and our database name is called `imagini`. We can now include the module and settings, and create the connection:

```
const settings = require("./settings");
const mysql    = require("mysql");
const db       = mysql.createConnection(settings.db);
```

This module only connects to the database when you make a query. This means the service would start and you wouldn't know whether your connection settings are correct until you make the first query. We don't want to figure out we can't connect to the database only when the service is used later on, so let's force a connection and check if the server is running and accepts our connection:

```
db.connect((err) => {
    if (err) throw err;

    console.log("db: ready");

    // ...
    // the rest of our service code
    // ...

    app.listen(3000, () => {
        console.log("app: ready");
    });
});
```

This way, if anything is wrong with the database, the service won't start and will throw an exception, which will notify you to check what's wrong. Here's an example of a possible error:

```
Error: ER_ACCESS_DENIED_ERROR: Access denied for user 'root'@'localhost'
(using password: YES)
```

This indicates you probably typed the password incorrectly, or the user doesn't match, or even the hostname or database may be wrong. Ensuring you connect to the database before setting up the service means your service won't be exposed to the public without a proper state.

Our microservice has a very simple state, so to speak. Our state is the images previously uploaded. Instead of using the filesystem, we can now use the database and create a table to store them all:

```
db.query(
    `CREATE TABLE IF NOT EXISTS images
    (
        id INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
        date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
        date_used TIMESTAMP NULL DEFAULT NULL,
        name VARCHAR(300) NOT NULL,
        size INT(11) UNSIGNED NOT NULL,
        data LONGBLOB NOT NULL,

        PRIMARY KEY (id),
```

```
        UNIQUE KEY name (name)
    )
    ENGINE=InnoDB DEFAULT CHARSET=utf8`
);
```

We can issue this query every time the service starts because it will create the images table only if it doesn't exist already. If we don't change its structure, it's fine to always do this.

You can see we're creating a table with a unique identification number (`id`), a creation date (`date_created`), a date to know when our image has been used(`date_used`), the `name` of the image, the `size` of it in bytes, and the image `data`. The size is a little redundant here as we could just check the data length, but bear with me, this is just an example.

We also defined our name as a unique key, meaning it has an index for quickly finding images by name, and also ensures our name does not repeat and that no one can overwrite an image (without removing it first).

Having the images stored this way on a database table gives you several advantages, such as regarding:

- How many images you have
- The size of every image and the total size
- When the images were created and last used

It also enables you to improve your service; for example, you can delete images that are not used for longer than a specific time period. You can also make this dependent on the image sizes. Later, you can add authentication (mandatory or not) and have user-specific rules.

It's also easy to back up and replicate the state to another site. There are plenty of tools for backing up databases, and you can have another MySQL server acting as a slave to this one and have your images replicated in real time to another geographical location.

Let's change our service from the previous chapter to use our table instead of the previously used folder on our filesystem. We can remove our `fs` module dependency (don't remove the path dependency for now):

```
app.param("image", (req, res, next, image) => {
    if (!image.match(/\.(png|jpg)$/i)) {
        return res.status(403).end();
    }

    db.query("SELECT * FROM images WHERE name = ?", [ image ], (err,
    images) => {
        if (err || !images.length) {
            return res.status(404).end();
```

```
        }

        req.image = images[0];

        return next();
    });
});
```

Our `app.param` is completely different. We now validate the `image` against our `image` table. If it doesn't find it, it returns code `404`. If it does find it, it stores the `image` information in `req.image`. We can now change our `image` upload to store the `image` on our table:

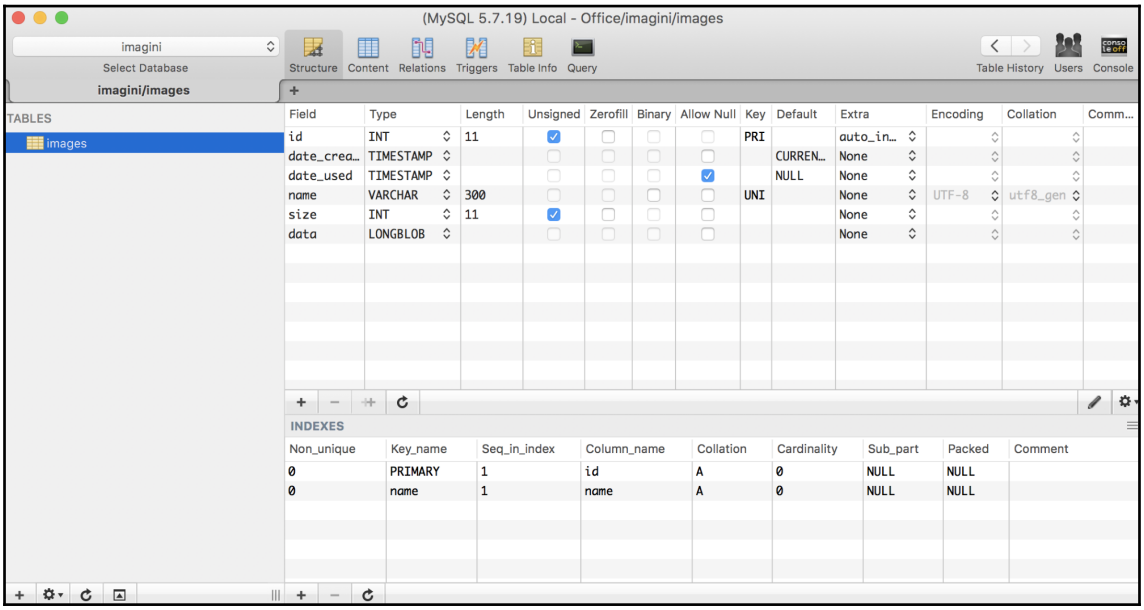
```
app.post("/uploads/:name", bodyparser.raw({
    limit : "10mb",
    type : "image/*"
}), (req, res) => {
    db.query("INSERT INTO images SET ?", {
        name : req.params.name,
        size : req.body.length,
        data : req.body,
    }, (err) => {
        if (err) {
            return res.send({ status : "error", code: err.code });
        }

        res.send({ status : "ok", size: req.body.length });
    });
});
```

Uploading images no longer use the filesystem and instead creates a new row on our table. We don't need to specify the `id` as it's automatic. Our creation date is also automatic as it defaults to the current timestamp. Our use date defaults to `NULL`, which means we haven't used the `image` yet:

```
app.head("/uploads/:image", (req, res) => {
    return res.status(200).end();
});
```

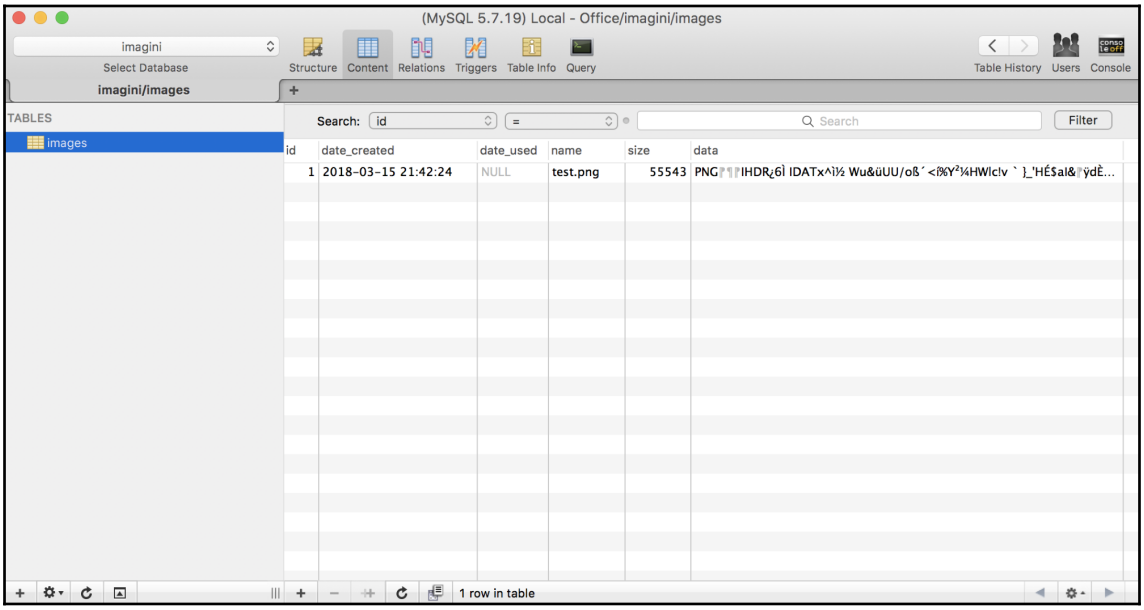
Our `image` check method now gets extremely simple as it relies on the previous `app.param` to check whether the `image` exists, so, if we get to this point, we already know the image exists (it's on `req.image`), so we just need to return the code `200`.

Before updating our image `fetch` method, let's try our service. If you start it on the console, you can immediately open any MySQL administration tool and check our database. I'm using Sequel Pro for macOS. Although there's a Pro in the name, it's free software and it's damn good:



Our table was created, and you can check it has all the properties and indexes we defined. Let's now upload an `image` once again:

As before, it returns a JSON response with a success status and the size of the `image`. If you look at Sequel again, on the content separator, you'll see our **images** data:

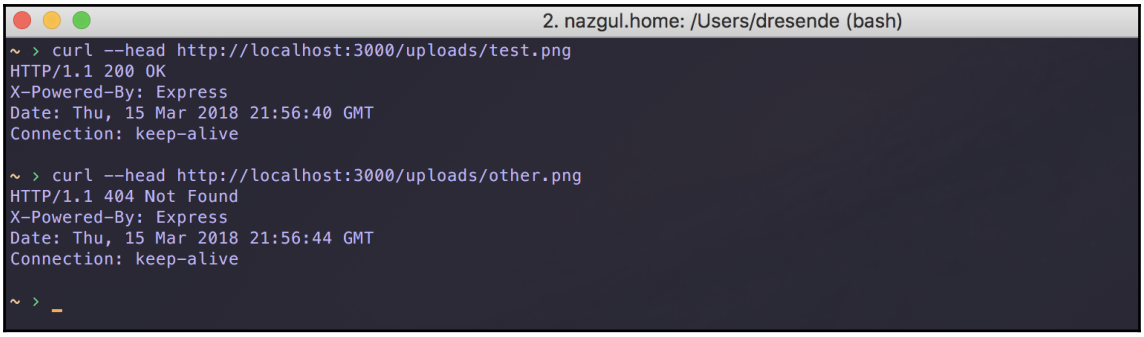


Let's try and upload the `image` again. Previously, our service would just overwrite it. Now, because of our unique index, it should deny an `INSERT` with the same name:

Great! The `ER_DUP_ENTRY` is the MySQL code for duplicate insertion. We can rely on that and deny overwriting images.

We can also check whether our `image` exists using the `check` method:

If we use another name, we'll get a code `404`:



It looks like everything is working great. Let's now change our final method, the `image` manipulation one. This method is almost the same; we just don't have to read the `image` file, as it's already available:

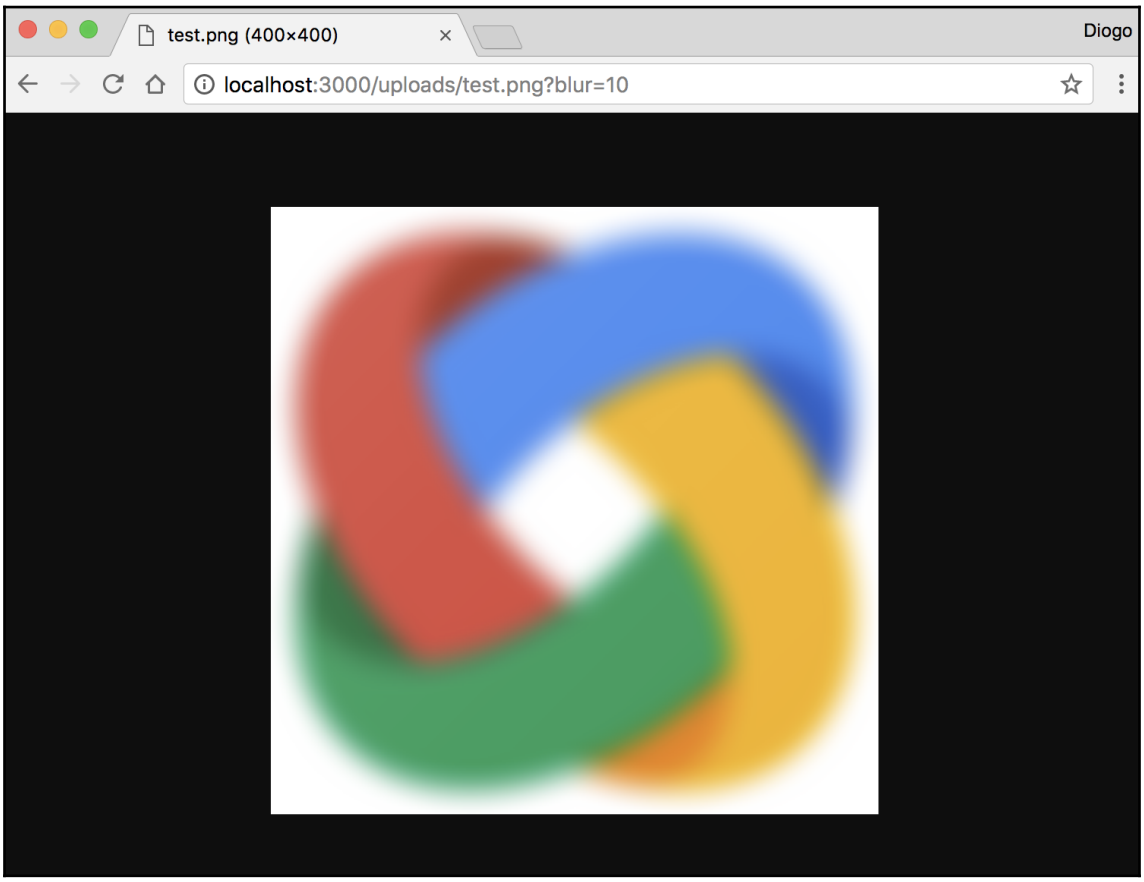
```
app.get("/uploads/:image", (req, res) => {
    let image     = sharp(req.image.data);
    let width     = +req.query.width;
    let height    = +req.query.height;
    let blur      = +req.query.blur;
    let sharpen   = +req.query.sharpen;
    let greyscale = [ "y", "yes", "true", "1",
    "on"].includes(req.query.greyscale);
    let flip      = [ "y", "yes", "true", "1",
    "on"].includes(req.query.flip);
    let flop      = [ "y", "yes", "true", "1",
    "on"].includes(req.query.flop);

    if (width > 0 && height > 0) {
        image.ignoreAspectRatio();
    }
```

```
    if (width > 0 || height > 0) {
        image.resize(width || null, height || null);
    }

    if (flip)         image.flip();
    if (flop)         image.flop();
    if (blur > 0)     image.blur(blur);
    if (sharpen > 0) image.sharpen(sharpen);
    if (greyscale)    image.greyscale();

    db.query("UPDATE images " +
             "SET date_used = UTC_TIMESTAMP " +
             "WHERE id = ?", [ req.image.id ]);

    res.setHeader("Content-Type", "image/" +
path.extname(req.image.name).substr(1));

    image.pipe(res);
});
```

You can see how we used the path dependency to get the extension of the `image` name. The rest is the same. We just add an update to our image every time we request this method.

We can use a web browser to test our method and see our previously uploaded image:

Everything should just work as before because we haven't changed our image manipulation dependency, so blurring and the other actions should work as expected:

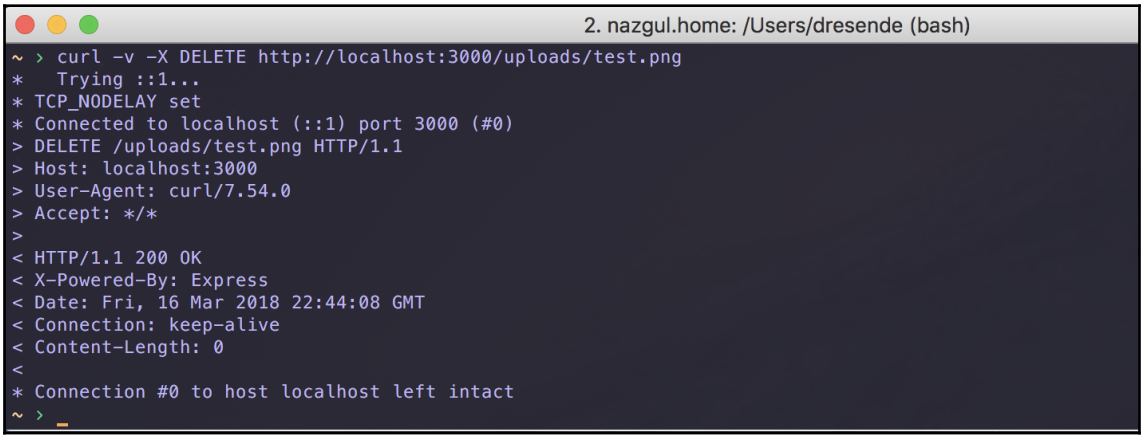


We can now improve our service and add a method we didn't expose before: deleting an `image`. To do that, we can use the `DELETE` verb from HTTP and just remove the `image` from our table:

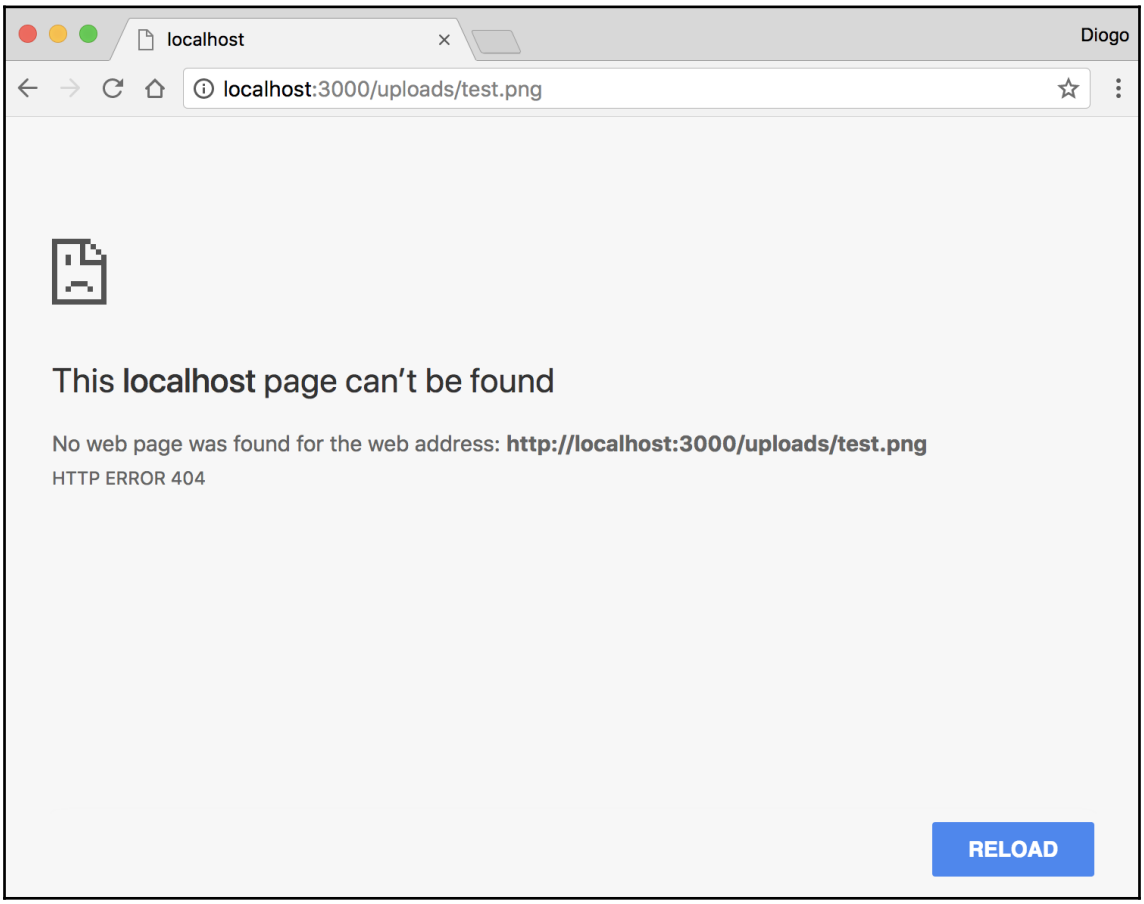
```
app.delete("/uploads/:image", (req, res) => {
    db.query("DELETE FROM images WHERE id = ?", [ req.image.id ], (err)
    => {
        return res.status(err ? 500 : 200).end();
    });
});
```

We just have to check whether the query resulted in an error. If so, we respond with a code `500` (internal server error). If not, we respond with the usual code `200`.

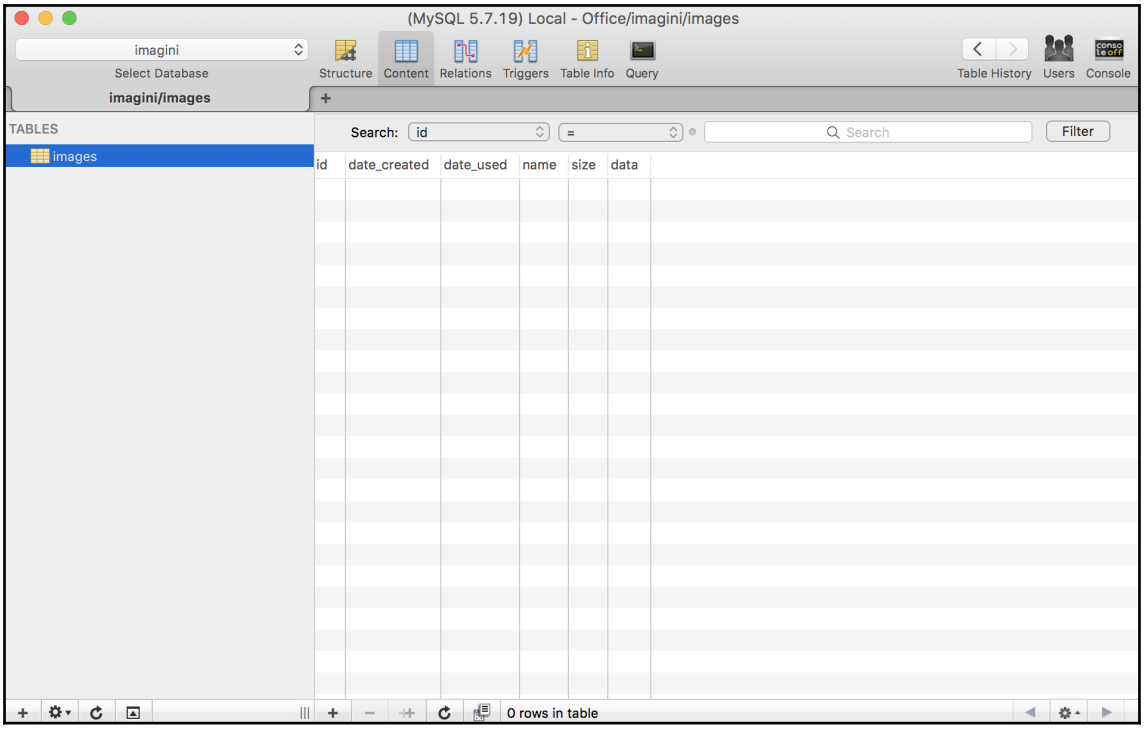
Let's restart our microservice and try to delete our `image`:



It looks like it worked; it responded with a code `200`. If we try to open our image in the web browser, we should see something like this:



On Sequel, the table should now be empty too:



We now have a functional microservice with a state that persists across restarts, as we intended. You can now deploy to any cloud service with no dependency on the filesystem, just a database.

You could easily change MySQL to another database or use an **object relational mapping** (**ORM**) module to enable you to change database server without changing your code. An ORM is a library that allows you to use a common interface to access different types of databases. Usually, this kind of abstraction involves not using SQL at all and reducing your interaction with the databases to simpler queries (to allow for interoperability between database servers).

Let's take this opportunity to go a little further and add a few methods that got simplified by this migration to the database. Let's create a method that exposes statistics about our database, and let's remove old images.

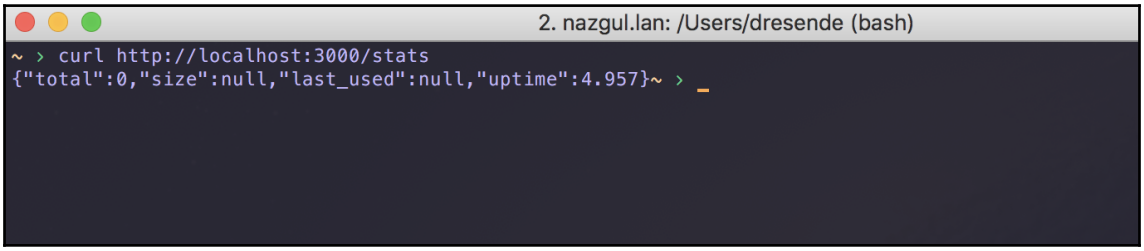
Our first statistics method should just return a JSON structure with some useful information. Let's expose the following:

- The total number of images
- The total size of the images
- How long our service is running
- When the last time was that we uploaded an image

Here's an example of how our statistics method could look:

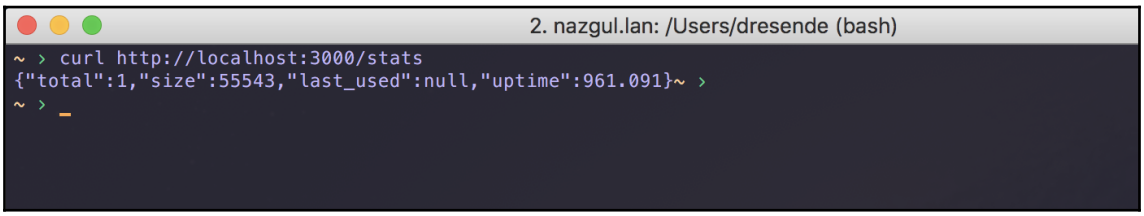
```
app.get("/stats", (req, res) => {
    db.query("SELECT COUNT(*) total" +
             ", SUM(size) size " +
             ", MAX(date_created) last_created " +
             "FROM images",
    (err, rows) => {
        if (err) {
            return res.status(500).end();
        }

        rows[0].uptime = process.uptime();

        return res.send(rows[0]);
    });
});
```

Restart the service, and let's try it:



As we can see, we have no images as we just removed our image previously. There's no size because we have no images. There's also no used date, and the service uptime is 5 seconds.

If we upload our previous image, we will get different results, something like the following screenshot:



Now, for our second task, deleting old images, we need to check our database periodically. We'll use an interval timer and just run a `DELETE` query. The intervals mentioned in the following query are just an example; you can write the conditions you want.

```
setInterval(() => {
    db.query("DELETE FROM images " +
             "WHERE (date_created < UTC_TIMETSTAMP - INTERVAL 1 WEEK
             AND date_used IS NULL) " +
             " OR (date_used < UTC_TIMETSTAMP - INTERVAL 1 MONTH)");
}, 3600 * 1000);
```

This query deletes `images` that were not used in the past month (but were used before) or images that were not used in the past week (and never used before). This means that images uploaded need to be used at least once or they will get removed quickly.

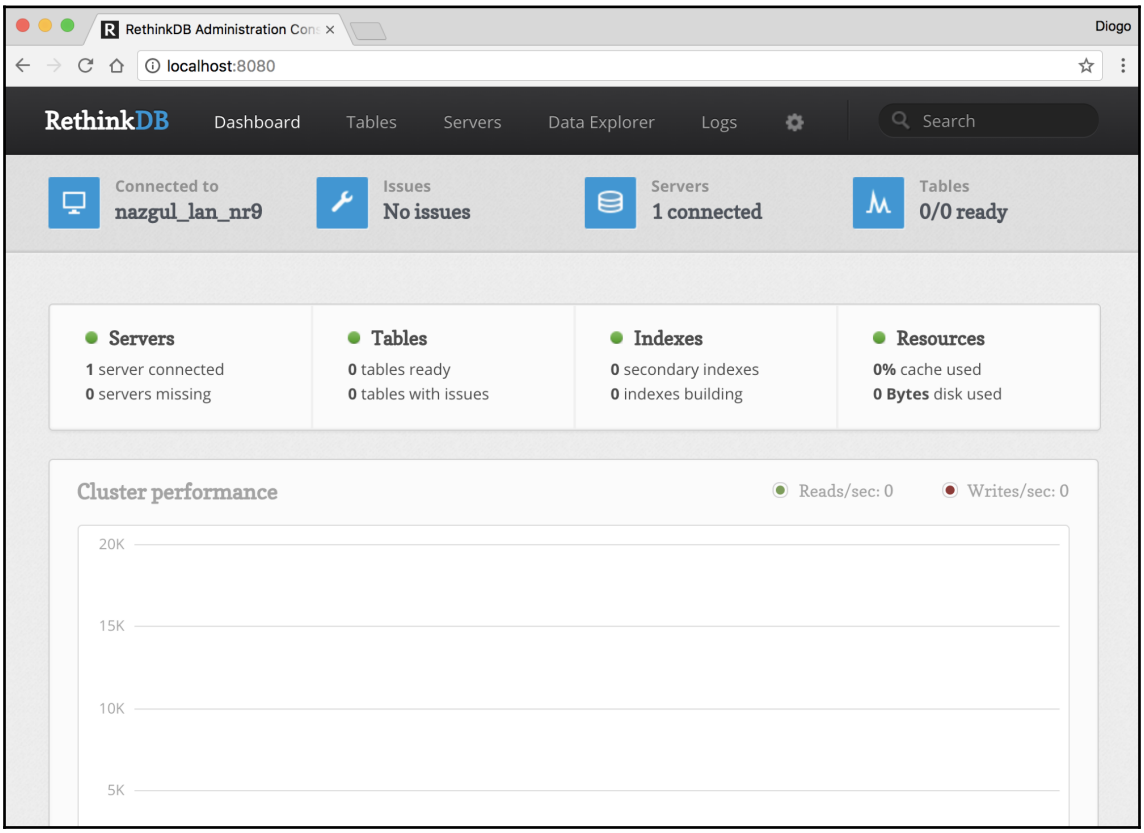
You can think of a different strategy, or use no strategy and delete manually if you want. Now that we've seen MySQL, let's move on and look at another kind of database server.

RethinkDB

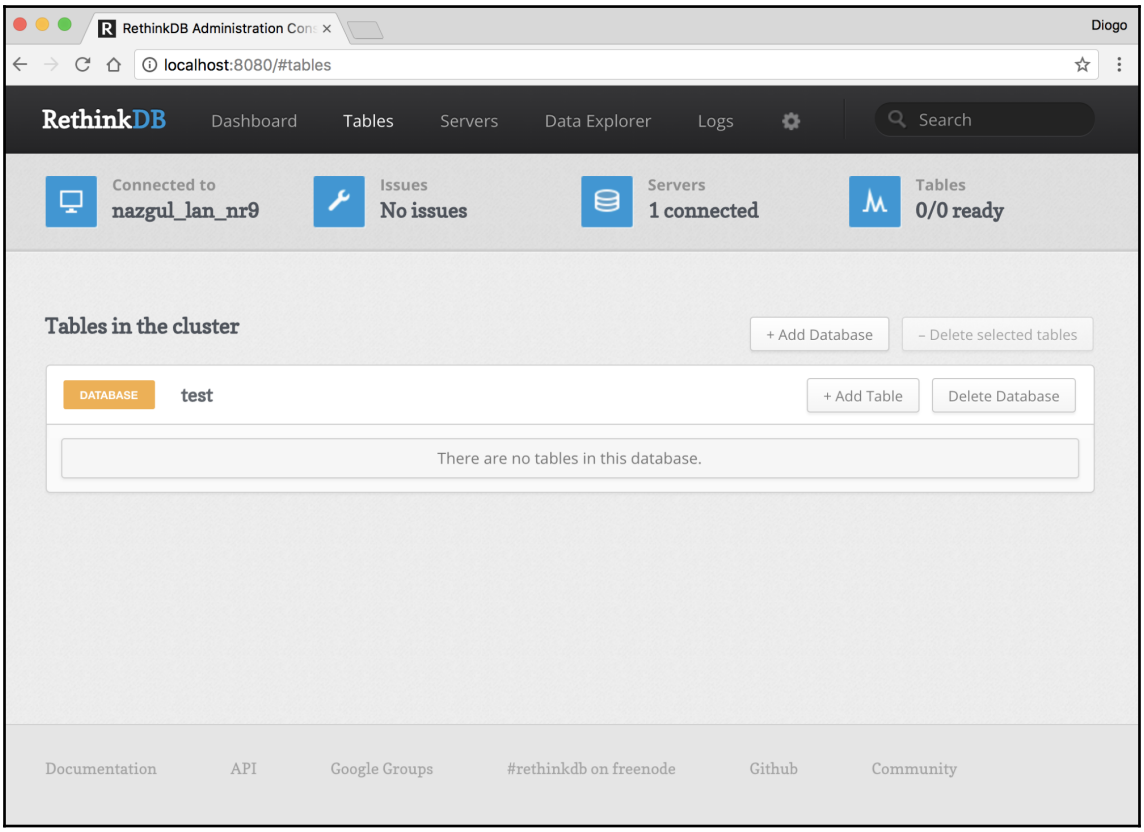
Let's see the differences for a non-relational database using RethinkDB. If you don't have it, just install it by following the official documentation (`https://www.rethinkdb.com/docs/`). Let's just start the server:

```
rethinkdb
```

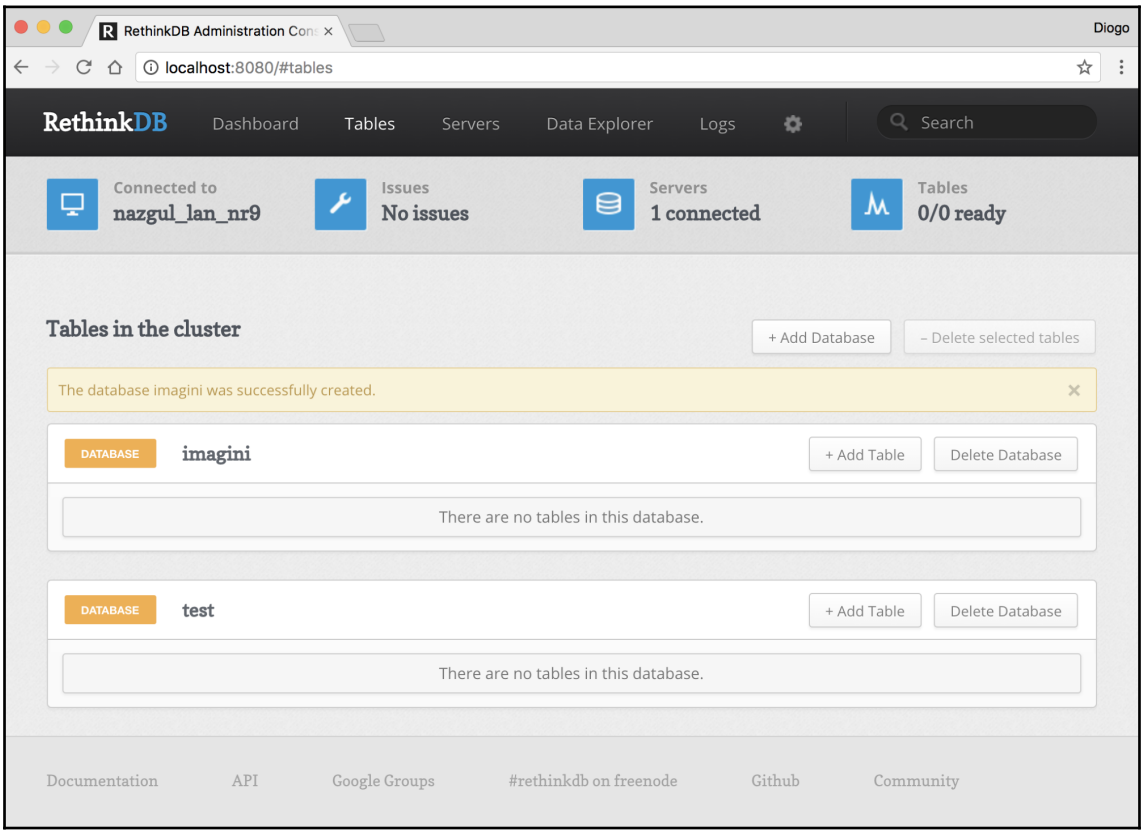
This will start the server, which comes with a very nice administration console on port `8080`. You can open it in the web browser:



Go to the **Tables** section on top to see the databases:



Create a database called `imagini` using the **Add Database** button. You should now have our database ready. You need nothing else here:



To use our new database, we need to install the `rethinkdb` dependency. You can remove the MySQL dependency:

```
npm uninstall mysql --save
npm install rethinkdb --save
```

Now, let's change our `settings` file. This module doesn't accept a connection string, so we'll use a JSON structure:

```
{
    "db": {
        "host" : "localhost",
        "db" : "imagini"
    }
}
```

To include our dependency, we just need to include the module:

```
const rethinkdb = require("rethinkdb");
```

Then, use this to connect to our server:

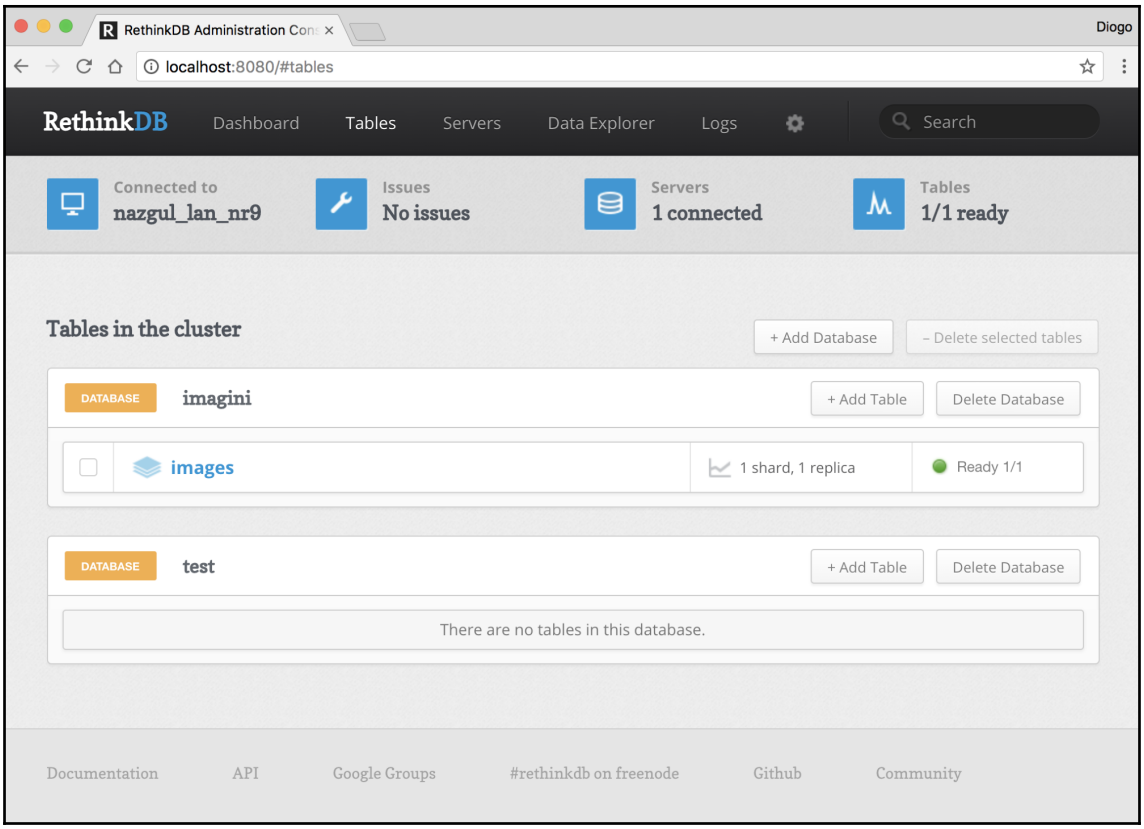
```
rethinkdb.connect(settings.db, (err, db) => {
    if (err) throw err;

    console.log("db: ready");

    // ...
    // the rest of our service code
    // ...

    app.listen(3000, () => {
        console.log("app: ready");
    });
});
```

After connecting, we can create our table as we did before. This time, we don't need to specify any structure:

```
rethinkdb.tableCreate("images").run(db);
```

The `rethinkdb` object is the one we'll use to manipulate our table, and the `db` object is a connection object used to reference the connection and to indicate where to run our manipulations.

If you restart our service just like this, you'll see a new table on our previously created database:



If you restart our service again, you'll get an error trying to create the table that already exists. We need to check whether it already exists, and only issue the command if not:

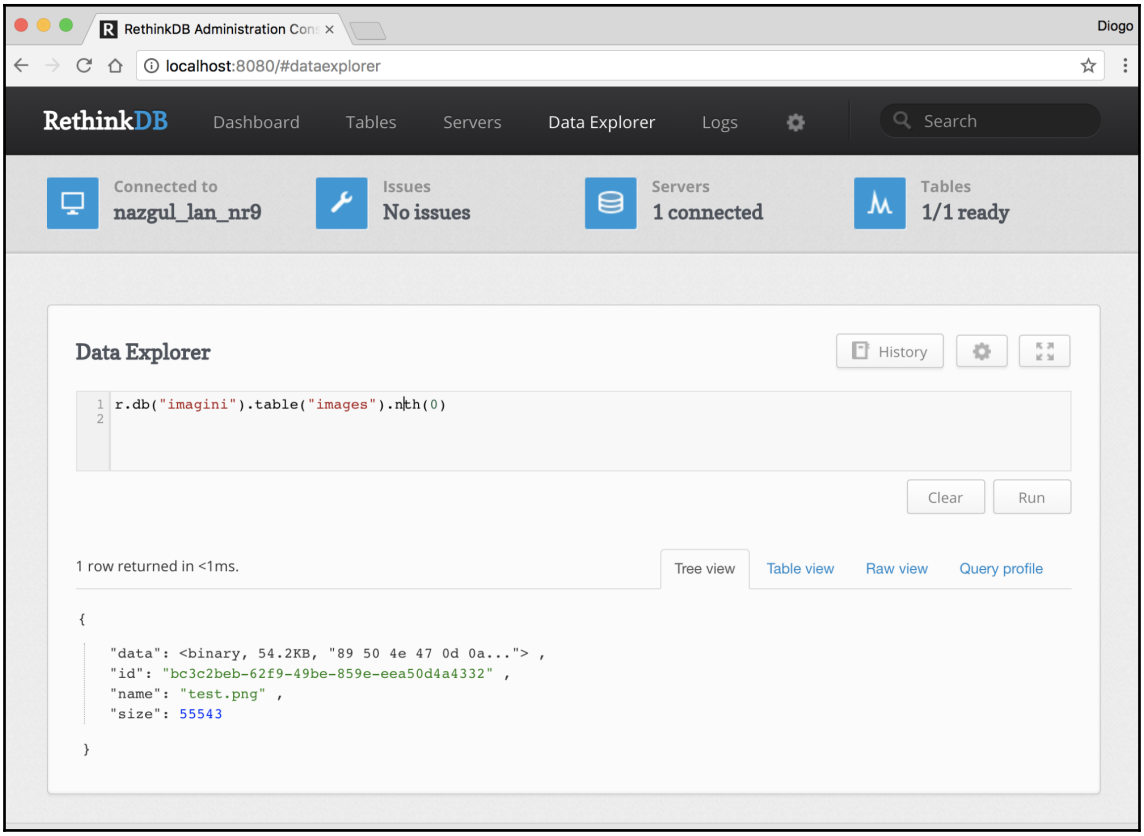
```
rethinkdb.tableList().run(db, (err, tables) => {
    if (err) throw err;

    if (!tables.includes("images")) {
        rethinkdb.tableCreate("images").run(db);
    }
});
```

Moving on, our upload method should be changed slightly to something like the following:

```
app.post("/uploads/:name", bodyparser.raw({
    limit : "10mb",
    type : "image/*"
}), (req, res) => {
    rethinkdb.table("images").insert({
        name : req.params.name,
        size : req.body.length,
        data : req.body,
    }).run(db, (err) => {
        if (err) {
            return res.send({ status : "error", code: err.code });
        }

        res.send({ status : "ok", size: req.body.length });
    });
});
```

If you restart the server just like this, you should be able to upload an `image`:

We receive the same response, just like with MySQL. We can go to the **Data Explorer** section in the administration console and get our record to see whether it's there:



Looks good. Notice our record ID is not a number, it's a **Universally Unique Identifier** (**UUID**). This is because RethinkDB has support for sharding (our table is sharded by default if there was more than one server) and it's easier to shard unique identifiers than an incremental number.

Moving on to our Express parameter:

```
app.param("image", (req, res, next, image) => {
    if (!image.match(/\.(png|jpg)$/i)) {
        return res.status(403).end();
    }

    rethinkdb.table("images").filter({
        name : image
    }).limit(1).run(db, (err, images) => {
        if (err) return res.status(404).end();

        images.toArray((err, images) => {
```

```
                if (err) return res.status(500).end();
                if (!images.length) return res.status(404).end();

                req.image = images[0];

                return next();
            });
        });
    });
```

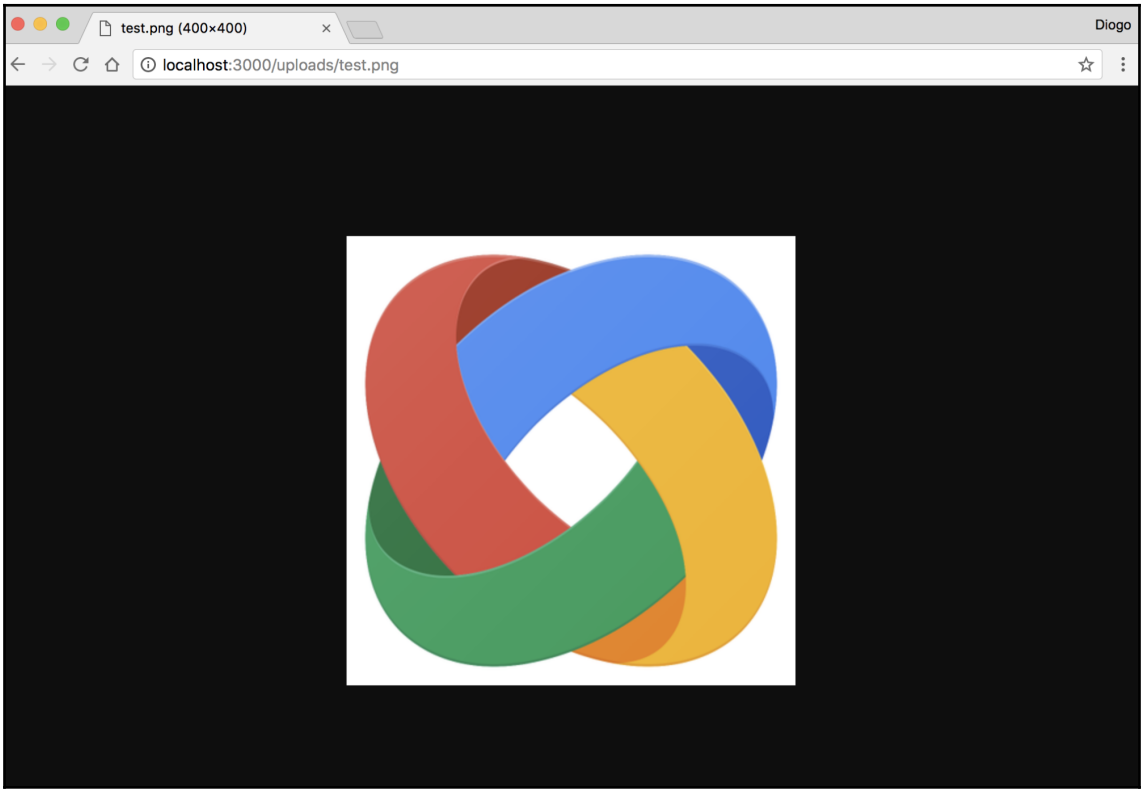
With this change, we can now restart our service and see whether our `image` exists:

We need to change the download just a little bit. We need to remove the previous query to update our usage date and replace it with a new one:

```
app.get("/uploads/:image", (req, res) => {
    let image     = sharp(req.image.data);
    let width     = +req.query.width;
    let height    = +req.query.height;
    let blur      = +req.query.blur;
    let sharpen   = +req.query.sharpen;
    let greyscale = [ "y", "yes", "true", "1",
    "on"].includes(req.query.greyscale);
    let flip      = [ "y", "yes", "true", "1",
    "on"].includes(req.query.flip);
    let flop      = [ "y", "yes", "true", "1",
    "on"].includes(req.query.flop);

    if (width > 0 && height > 0) {
        image.ignoreAspectRatio();
    }

    if (width > 0 || height > 0) {
        image.resize(width || null, height || null);
```

```
        }

        if (flip)         image.flip();
        if (flop)         image.flop();
        if (blur > 0)     image.blur(blur);
        if (sharpen > 0)  image.sharpen(sharpen);
        if (greyscale)    image.greyscale();

        rethinkdb.table("images").get(req.image.id).update({ date_used :
        Date.now() }).run(db);

        res.setHeader("Content-Type", "image/" +
        path.extname(req.image.name).substr(1));

        image.pipe(res);
    });
```

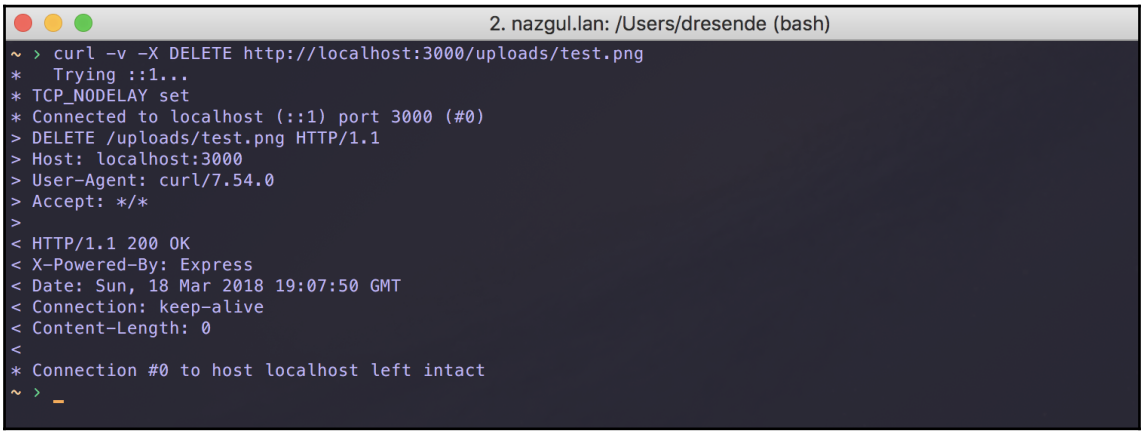
We can now download our image using the web browser:



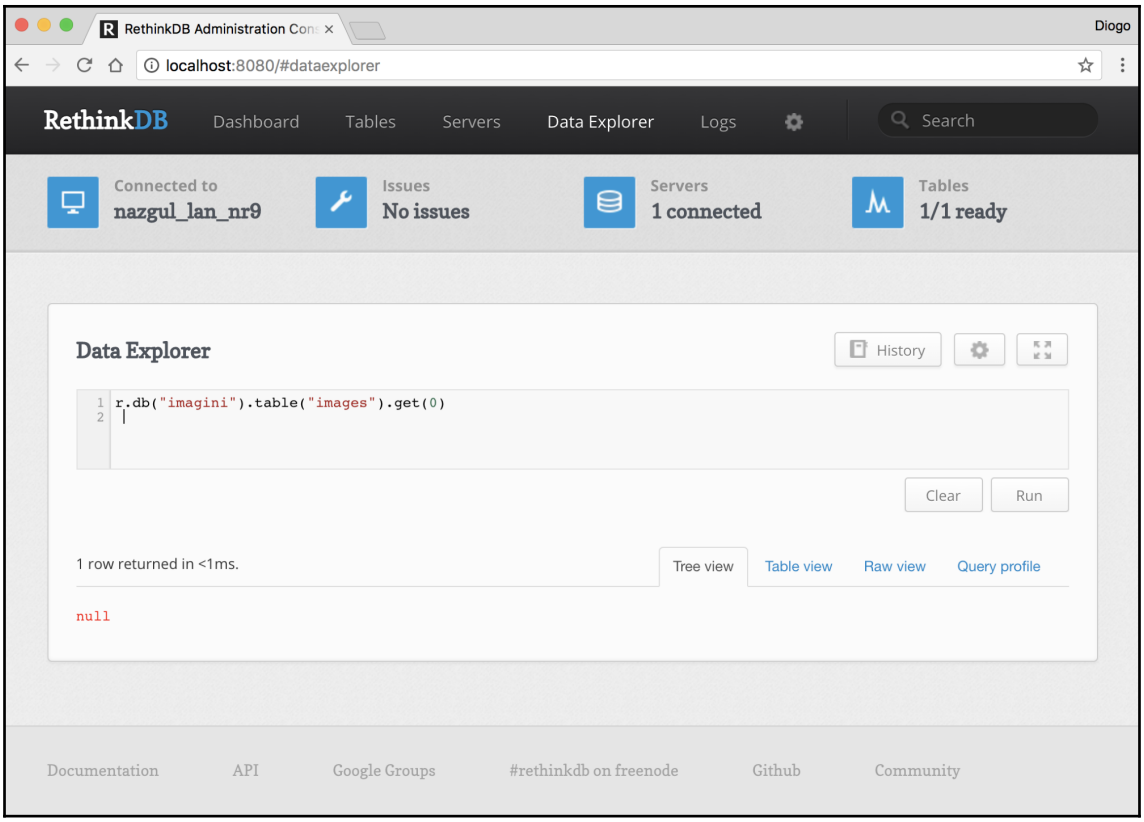
Next, we need to update our image removal method. It's as easy as our upload:

```
app.delete("/uploads/:image", (req, res) => {
    rethinkdb.table("images").get(req.image.id).delete().run(db, (err)
    => {
        return res.status(err ? 500 : 200).end();
    });
});
```

This time, we used the image unique ID to remove it. If we try again using the `curl` command, we'll receive a code `200`:



If we try to get the first record of our table, we'll see there's nothing there:



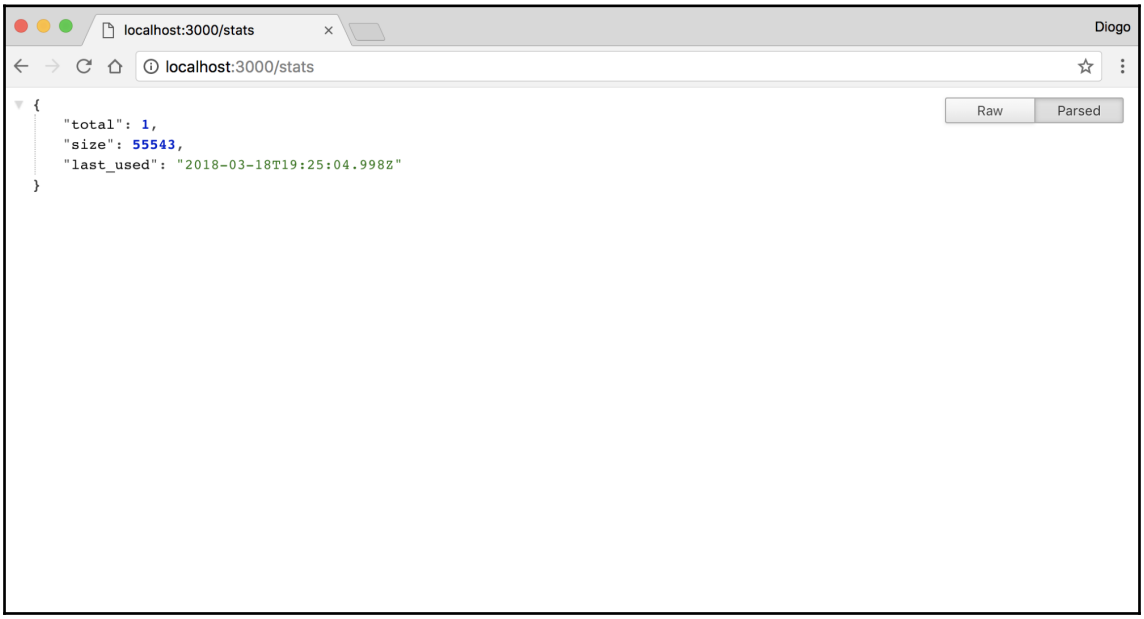
Finally, there are our two extra features that we added after introducing MySQL: the statistics and removing old unused images.

Our statistics method is not so simple as running an SQL query with aggregations. We must calculate each of our statistics:

```
app.get("/stats", (req, res) => {
    let uptime = process.uptime();

    rethinkdb.table("images").count().run(db, (err, total) => {
        if (err) return res.status(500).end();

        rethinkdb.table("images").sum("size").run(db, (err, size) => {
            if (err) return res.status(500).end();

            rethinkdb.table("images").max("date_created").run(db, (err,
            last_created) => {
                if (err) return res.status(500).end();
```

```
                    last_created = (last_created ? new
                    Date(last_created.date_created) : null);

                    return res.send({ total, size, last_created, uptime });
                });
            });
        });
    });
```

We should have a similar result to before:



Removing old images is more or less easy; we just need to filter the images we want to remove, and then remove them:

```
setInterval(() => {
    let expiration = Date.now() - (30 * 86400 * 1000);

    rethinkdb.table("images").filter((image) => {
        return image("date_used").lt(expiration);
    }).delete().run(db);
}, 3600 * 1000);
```

I simplified the previous strategy and am just removing `images` older than 1 month (30 days, times 86,400 seconds a day, times 1,000 milliseconds).

Redis

In-memory databases are different from the previous two types, as they're usually not structured, which means you have no tables. What you have is normally lists of some kind that you can look up and manipulate, or simple hash tables.

Taking advantage of the Redis instance we installed previously for Hydra, let's see another drawback, or actually feature, of this kind of database. Let's connect to our Redis instance and make the following sequence of instructions:

What we did here was to:

1. Connect to the Redis service using `redis-cli`.
2. Get the content of the counter, which is nil (nothing), because we haven't defined it yet.
3. Increment the counter, which is now automatically defined and set to `1`.
4. Increment the counter again, which is now `2`.
5. Get the content of the counter, which is of course `2`.
6. Shut down the Redis service.
7. Start the Redis service.
8. Connect to the Redis service again.
9. Get the content of the counter, which is nil (nothing).

Where's our counter? Well, this is an in-memory database, so everything is gone when we shut down the Redis service. This is the design of almost all kinds of in-memory databases.

They're designed to be fast and in-memory. Their purpose is normally to cache data that is expensive to get, such as some complex calculations, or extensive to download, and we want that to be available faster (in-memory).

I wasn't completely fair with Redis as it actually allows your data to be saved between service restarts. So, let's see how far we can go in using it to save our microservice state.

As before, let's uninstall `rethinkdb` and install the `redis` module:

```
npm uninstall rethinkdb --save
npm install redis --save
```

Let's ignore our `settings.json` file (you can remove it if you prefer) and assume Redis will be on our local machine.

First, we need to include the `redis` module and create a `Client` instance:

```
const redis = require("redis");
const db    = redis.createClient();
```

We then need to wait until it connects:

```
db.on("connect", () => {
    console.log("db: ready");

    // ...
    // the rest of our service code
    // ...

    app.listen(3000, () => {
        console.log("app: ready");
    });
});
```

There are a couple of ways we can use Redis to store our data. To make it simple, as we don't have tables, let's use hashes to store our images. Each image will have a different hash, and the name of the hash will be the name of the image.

As there are no tables in this kind of database, our initialization code can just be removed.

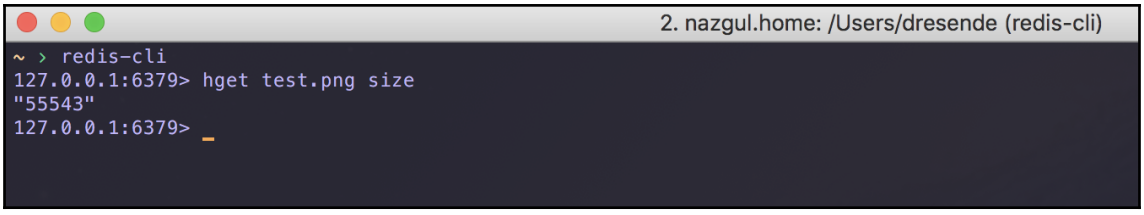
Next, let's change our upload method to store data on Redis. As I mentioned, let's store it in a hash with the name of the `image`:

```
app.post("/uploads/:name", bodyparser.raw({
```

```
        limit : "10mb",
        type : "image/*"
    }), (req, res) => {
        db.hmset(req.params.name, {
            size : req.body.length,
            data : req.body.toString("base64"),
        }, (err) => {
            if (err) {
                return res.send({ status : "error", code: err.code });
            }

            res.send({ status : "ok", size: req.body.length });
        });
    });
```

The `hmset` command lets us set multiple fields of a hash, in our case, `size` and `data`. Notice we're storing our image content in `base64` encoding, otherwise we'll lose data. If we restart our service and try to upload our test `image`, it should work fine:

We can then use `redis-cli` and see whether our image is there. Well, we're checking to see whether our hash has the field size and matches our image size:



Great! We can now change our Express parameter to look for the `image` hash:

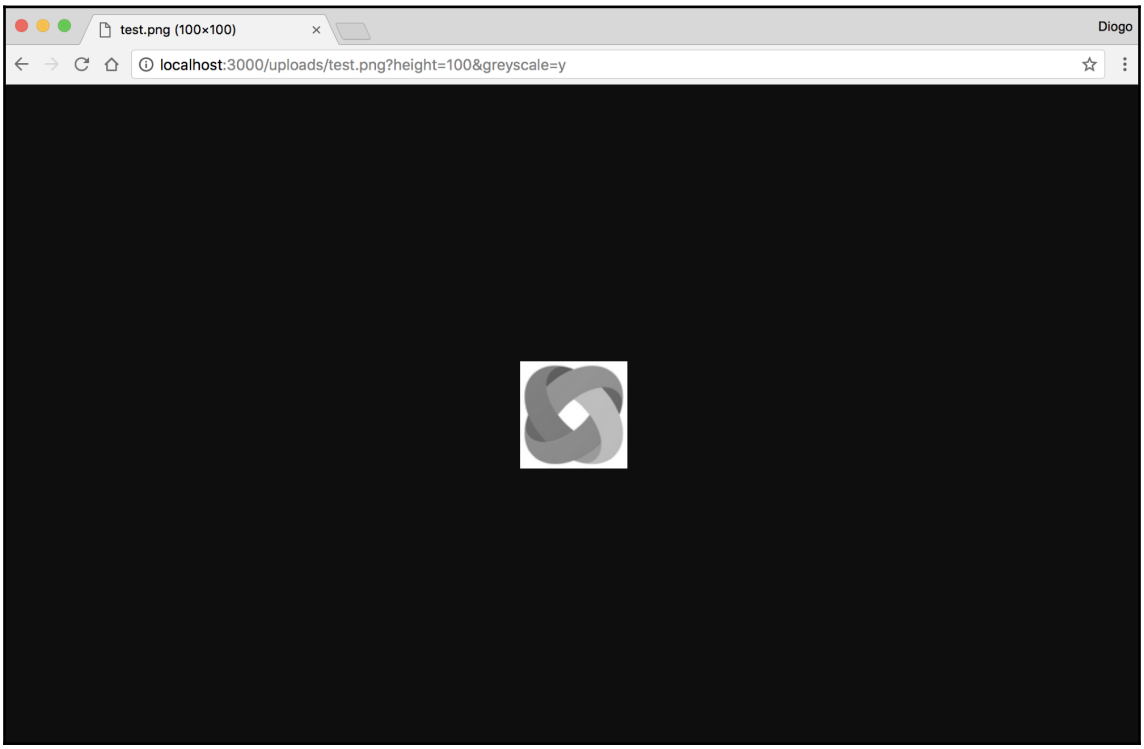
```
app.param("image", (req, res, next, name) => {
    if (!name.match(/\.(png|jpg)$/i)) {
        return res.status(403).end();
    }
```

```
    db.hgetall(name, (err, image) => {
        if (err || !image) return res.status(404).end();

        req.image      = image;
        req.image.name = name;

        return next();
    });
});
```

Our `image` check method should work now. And, for our download method to work, we just need to change the image loading to decode our previous `base64` encoding:

```
app.get("/uploads/:image", (req, res) => {
    let image     = sharp(Buffer.from(req.image.data, "base64"));
    let width     = +req.query.width;
    let height    = +req.query.height;
    let blur      = +req.query.blur;
    let sharpen   = +req.query.sharpen;
    let greyscale = [ "y", "yes", "true", "1",
"on"].includes(req.query.greyscale);
    let flip      = [ "y", "yes", "true", "1",
"on"].includes(req.query.flip);
    let flop      = [ "y", "yes", "true", "1",
"on"].includes(req.query.flop);

    if (width > 0 && height > 0) {
        image.ignoreAspectRatio();
    }

    if (width > 0 || height > 0) {
        image.resize(width || null, height || null);
    }

    if (flip)         image.flip();
    if (flop)         image.flop();
    if (blur > 0)     image.blur(blur);
    if (sharpen > 0) image.sharpen(sharpen);
    if (greyscale)    image.greyscale();

    db.hset(req.image.name, "date_used", Date.now());

    res.setHeader("Content-Type", "image/" +
path.extname(req.image.name).substr(1));

    image.pipe(res);
});
```

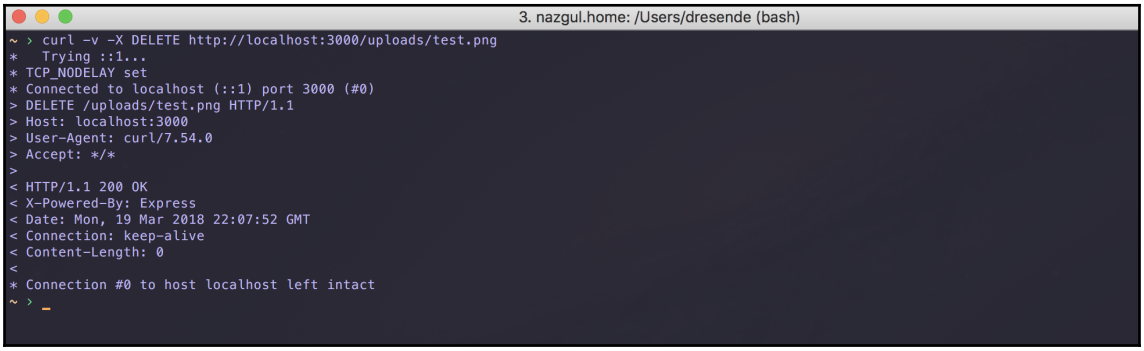
Our images are now being served from Redis. As a bonus, we're adding/updating a `date_used` field in our `image` hash to indicate when it was last used:



Removing our `image` is as simple as removing our hash:

```
app.delete("/uploads/:image", (req, res) => {
    db.del(req.image.name, (err) => {
        return res.status(err ? 500 : 200).end();
    });
});
```

We can then try to remove our `test` image:



Using `redis-cli` to check whether the hash exists, we see that it's gone:

The only two features missing are the statistics and removing old images.

For the statistics, that could be hard as we're using generic hash tables and we can't be sure how many hash tables are defined, and if all or any have image data. We would have to scan all hash tables, which is complex for large sets.

To remove old images, the problem is the same as there's no way of looking for hash tables with a specific condition, such as a field value.

There are still other paths available to tackle this problem. For example, we could have another hash table with just our image names and use dates. But, the complexity would increase, and the integrity could be at risk as we're splitting information through different hash tables with no certainty of making **Atomicity**, **Consistency**, **Isolation**, and **Durability** (**ACID**) operations.

Conclusion

As we've seen, there are plenty of options to store our microservice state. Depending on the type of information we're manipulating, there are databases better-prepared to handle our data.

It all depends on a few different questions we should ask ourselves:

- Is our data integrity important?
- Is our data structure complex?
- How and what type of information do we need to acquire?

If our data integrity is important or the data structure is complex, do not use in-memory databases. Depending on the complexity, see if you need a non-relational database, or whether you can go with a relational database that can handle more complex manipulation and data aggregation operations, which will help you to achieve the last point.

Security

One good practice is to write code iteratively, testing every time we make a new small feature or improvement, and always write code thinking of all the features we envision for our service.

Thinking about the service roadmap allows you to prepare the service for future improvements, reducing the amount of code wasted or replaced later on.

For instance, in terms of security:

- Is our service secure? Is it prepared for some types of malicious attacks?
- Is our service private? Should it have some kind of authentication or authorization mechanism?

Luckily, our frameworks allow our code to be composed and allow us to add layers of security later. For example, using Express or Hydra, we can add a precedent routing function that will run before any of our service methods, allowing us to enforce, for example, authentication.

Looking at our service, since it exposes its methods using HTTP, there are a couple of improvements we can add to it, for example:

- **Authentication**: Forcing anyone that uses it to identify themselves. Or, just the upload and removal methods. It's up to you. There could also be user accounts, and each user would see their respective list of images.
- **Authorization**: Restricting, for example, what networks could access the service, independently of having a valid authentication or not.
- **Confidentiality**: Giving your users protection against prying eyes over the network traffic.
- **Availability**: Restricting the maximum usage frequency of the service, per client, to ensure a single client cannot block your entire service.

To introduce these improvements, you may add an authentication module such as the Passport module, and use a certificate to give your users a more secure HTTPS experience.

Other types of insecurity come directly from your code and don't improve by adding a certificate or forcing authentication. I'm referring to:

- Bugs, programming logic flaws, and use cases not properly tested, which can lead to minor or serious problems
- Dependency bugs, which you might not be aware of but can still ruin your service and may force you to look for alternative dependencies, which is never a pleasant task

To minimize these events, you should always keep evolving your test suite, adding use cases as they show up, ensuring a new bug that is solved does not reappear later. Regarding dependency bugs, you can subscribe to the Node Security Project and even integrate it with your code to always know when one of your dependencies is a risk.

If there were source code commandments, the next four would surely be on the list:

- Keep the code simple. If the code is getting complex, stop, look back, and split the code into simpler parts.
- Validate external input, whether it's the user or another service. Never trust data from the outside.
- Deny by default and not the opposite, checking whether someone has access to a resource and denying anyone that is not.
- Add test cases from the beginning of the project.