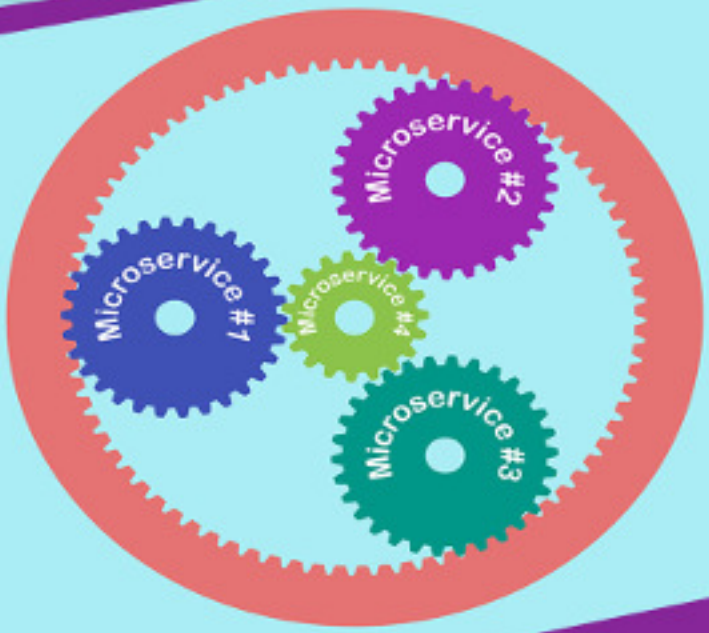
MICROSERVICES FOR JAVA DEVELOPERS

Hot Recipes for Microservices



ANDRIY REDKO

Java Code Geeks
Java 2 Java Developers Resource Center

Microservices for Java Developers

Contents

Preface

Microservices are a software development technique - a variant of the service-oriented architecture (SOA) structural style - that arranges an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. Computer microservices can be implemented in different programming languages and might use different infrastructures. Therefore the most important technology choices are the way microservices communicate with each other (synchronous, asynchronous, UI integration) and the protocols used for the communication (RESTful HTTP, messaging, ...). In a traditional system most technology choices like the programming language impact the whole systems. Therefore the approach for choosing technologies is quite different. (Source: https://en.wikipedia.org/wiki/Microservices)

In this book, we provide a comprehensive guide about Microservices for Java Developers. We cover a wide range of topics, from Microservices Communication and Implementing microservices to Managing Security, Testing, Monitoring and Alerting. With this guide you will be able to get your own projects up and running in minimum time. Enjoy!

About the Author

Andriy completed his Master Degree in Computer Science at Zhitomir Institute of Engineering and Technologies, Ukraine. For the last fifteen years he has been working as the Consultant/Software Developer/Senior Software Developer/Team Lead for a many successful projects including several huge software systems for customers from North America and Europe.

Through his career Andriy has gained a great experience in enterprise architecture, web development (ASP.NET, Java Server Faces, Play Framework), software development practices (test-driven development, continious integration) and software platforms (Sun JEE, Microsoft .NET), object-oriented analysis and design, development of the rich user interfaces (MFC, Swing, Windows Forms/WPF), relational database management systems (MySQL, SQL Server, PostgreSQL, Oracle), NoSQL solutions (MongoDB, Redis) and operating systems (Linux/Windows).

Andriy has a great experience in development of distributed (multi-tier) software systems, multi-threaded applications, desktop applications, service-oriented architecture and rich Internet applications. Since 2006 he is actively working primarily with JEE / JSE platforms.

As a professional he is always open to continuous learning and self-improvement to be more productive in the job he is really passionate about.

Chapter 1

Introduction

1.1 Introduction

Microservices, microservices, microservices ... One of the hottest topics in the industry nowadays and the new shiny thing everyone wants to be doing, often without really thinking about the deep and profound transformations this architectural style requires both from the people and organization perspectives.

In this chapter we are going to talk about practical microservice architecture, starting from the core principles and progressively moving towards making it production ready. There is tremendous amount of innovations happening in this space so please take everything we are going to discuss along the tutorial with some grain of salt: what is the accepted practice today may not hold its promise tomorrow. For better or worse, the industry is still building the expertise and gaining the experience around developing and operating microservices.

There are a lot of different opinions on doing the microservices the "right way" but the truth is, there is no really the magic recipe or advice which will get you there. It is a process of continuous learning and improvement while doing your best to keep the complexity under control. Please do not take everything discussed along this tutorial as granted, stay open-minded and do not be afraid to challenge things.

If you are looking to expand your bookshelf, there is not much literature available yet on the matter but Building Microservices: Designing Fine-Grained Systems by Sam Newman and Microservice Architecture: Aligning Principles, Practices, and Culture by Irakli Nadareishvili are certainly the books worth owning and reading.

1.2 Monoliths Around Us

For many years traditional single-tiered architecture or/and client/server architecture (practically, thin client talking to beefy server) were the dominant choices for building software applications and platforms. And fairly speaking, for the majority of the projects it worked (and still works) quite well but the appearance of microservice architecture suddenly put the scary monolith label on all of that (which many read as **legacy**).

This is a great example when the hype around the technology could shadow the common sense. There is nothing wrong with monoliths and there are numerous success stories to prove that. However, there are indeed the limits you can push them for. Let us briefly talk about that and outline a couple of key reasons to look towards adoption of microservice architecture.

Like it or not, in many organization monolith is the synonym to big ball of mud. The maintenance costs are skyrocketing very fast, the increasing amount of bugs and regressions drags quality bar down, business struggles with delivering new features since it takes too much time for developers to implement. This may look like a good opportunity to look back and analyze what went wrong and how it could be addressed. In many cases splitting the large codebase into a set of cohesive modules (or components) with well established APIs (without necessarily changing the packaging model per se) could be the simplest and cheapest solution possible.

But often you may hit the scalability issues, both scaling the software platform and scaling the engineering organization, which are difficult to solve while following the monolith architecture. The famous Conway's law summarized this pretty well.

"... that organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." - https://www.melconway.com/Home/Committees_Paper.html

Could the microservices be the light at the end of the tunnel? It is absolutely possible but please be ready to change dramatically the engineering organization and development practices you used to follow. It is going to be a bumpy ride for sure but it should be highlighted early on that along this tutorial we are not going to question the architecture choices (and push you towards microservices) but instead assume that you did you research and strongly believe that microservices is the answer you are looking for.

1.3 Saying "Yes!" to Microservices

So what the terms **"microservices"** and **"microservice achitecture"** actually mean? You may find that there are quite a few of slightly different definitions but arguably the most complete and understandable one is formulated by Martin Fowler in his essay on microservice architecture:

In short, the microservice architectural style ... is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. - https://www.martinfowler.com/articles/microservices.html

The prefix **"micro"** became the constant source of confusion as it is supposed to frame the size of the service somehow. Unsurprisingly, it turned out to be quite hard to justify exactly. The good rule of thumb is to split your system in such a way that every microservice has a single meaningful purpose to fulfill. Another somewhat controversial definition of the **"micro"** scale is that the microservice should be small enough to fit into the head of one developer (or more formally, maintainer).

There are basically two routes which may direct you towards embracing microservice architecture: starting the green-field application or evolving the architecture of the existing one (most likely, the monolith). In order to succeed while starting the journey by taking any of these routes, there are a number of the prerequisites and principles to be aware of.

First of all, microservices are all about domain modeling and domain design done right. There are two distinguishing books, The Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans and Implementing Domain-Driven Design by Vaughn Vernon, which are the credible sources on the subject and highly recommended reads. If you do not know your business domain well enough, the only advice would be to hold on with opening the microservices flood gate.

The importance of that is going to become clear in a few moments but it is particularly very difficult problem to tackle when you start the development of the application from scratch since there are too many unknowns in the equation.

1.3.1 Architecture(s) inside Architecture

Overall, microservice architecture mandates to structure your application as a set of modest services but does not dictate how the services (and communication between them) should be implemented. In some sense, it could be thought of some kind of supreme architecture.

As such the architectural choices applied to the individual microservices vary and, among many others, hexagonal architecture (also known as ports and adapters), clean architecture, onion architecture and the old buddy layered architecture could be often seen in the wild.

1.3.2 Bounded Context

The definition of the bounded context comes directly from the domain-driven design principles. When applied to the microservice architecture, it outlines the boundaries of the business subdomain each microservice is responsible for. The bounded context becomes the foundation of the microservice contract and essentially serves as the fence to the external world.

The cumulative business domain model of the whole application constitutes from the domain models of all its microservices, with bounded context in between to glue them together. Clearly, if the business domain is not well-defined and decomposed, so are the domain models, boundary contexts and microservices in front of them.

1.3.3 Ownership

Each microservice has to serve as the single source of truth of the domain it manages, including the data it owns. It is exceptionally important to cut off any temptations of data sharing (for example, consulting data stores directly), as such bypassing the contract microservice establishes.

In reality, the things will never stay completely isolated so there should be a solution. Indeed, since data sharing is not an option, you would immediately face the need to duplicate some pieces and, consequently, find a way to keep them in-sync.

Every implementation change which you are going to make should be deconstructed into pieces and land in the right microservice (or set of microservices), unambiguously.

From organizational perspective, the ownership translates into having a dedicated cross-functional team for each microservice (or perhaps, a few microservices). The cross-functional aspect is a significant step towards achieving the maturity and ultimate responsibility: **you build it, you ship it, you run it and you support it** (or to put it simply, **you build it, you own it**).

1.3.4 Independent Deployments

Each microservice should be independent from every other: not only during the development time but also at deployment time. Separate deployments, and most importantly, the ability to scale independently, are the strongest forces behind the microservice architecture. Think about microservices as the autonomous units, by and large agnostic to the platform and infrastructure, ready to be deployed anywhere, any time.

1.3.5 Versioning

Being independent also means that each microservice should have own lifecycle and release versioning. It is kind of flows out from the discussion around ownership however this time the emphasis is on collaboration. As in any loosely-coupled distributed system, it is very easy to break things. The changes have to be efficiently communicated between the owning teams so everyone is aware what is coming and could account for it.

Maintaining backward (and forward) compatibility becomes a must-have practice. This is another favor of responsibility: not only make sure the new version of your microservice is up and running smoothly, the existing consumers continue to function flawlessly.

1.3.6 Right Tool for the Job

The microservice architecture truly embraces **"pick the right tool for the job"** philosophy. The choices of the programming languages, frameworks, and libraries are not fixed anymore. Since different microservices are subjects to different requirements, it becomes much easier to mix and match various engineering decisions, as far as microservices could communicate with each other efficiently.

1.4 The Danger of the Distributed Monolith

Let us be fair, the microservice architecture has many things to offer but it also introduces a lot of complexity into the picture. Unsurprisingly, the choice of the programming languages and/or frameworks may amplify this complexity even more.

When decision is made to adopt microservices, many organizations and teams fall into the trap of applying the same development practices and processes that used to work in the past. This is probably the primary reason why the end result quite often becomes a distributed monolith, a nightmare for developers and horror for operations. Let us talk about that for a moment.

1.4.1 Every Function is (potentially) a Remote Call

Since each microservice lives in a separate process somewhere, every function invocation may potentially cause a storm of network calls to upstream microservices. The boundary here could be implicit and not easy to spot in the code, as such the proper instrumentation has to be present from day one. Network calls are expensive but, most importantly, everything could fail in spectacular ways.

1.4.2 Chattiness

Quite often one microservice needs to communicate with a few other upstream microservices. This is absolutely normal and expected course of action. However when the microservice in question needs to call dozens of other microservices (or issues a ton of calls to another microservice to accomplish its mission), it is huge red flag that the split was not done right. The high level of chattiness not only is subject to network latency and failures, it manifests the presence of the useless mediators or incapable proxies.

1.4.3 Dependency Cycles

The extreme version of chattiness is existence of the cycles between microservices, either direct or indirect. Cycles are often invisible but very dangerous beasts. Even if you find the magic sequence to deploy such interdependent microservices into production, sooner or later they are going to bring the application to its knees.

1.4.4 Sharing

Managing the common ground between microservices is particularly difficult problem to tackle. There are many best practices and patterns which we as the developers have learnt over the years. Among others the DRY and code reuse principles stand out.

Indeed, we know that the code duplication (also widely known as copy/paste programming) is bad and should be avoided at all costs. In context of microservice architecture though, sharing code between different microservices introduces the highest level of coupling possible.

It highly unrealistic (although worth trying) that you could completely get rid of shared libraries or alike, especially when your microservices are written using the same programming language or platform. The goal in this case would be to master the art of reducing the amount of shared pieces to absolutely necessary minimum, ideally to none. Otherwise, this kind of sharing will drag you down the monolith way, but this time the distributed one.

1.5 Conclusions

In this section we quite briefly talked about the microservice architecture, the benefits it delivers on the table, the complexity it introduces and the significant changes it brings to engineering organization. The opportunities this architectural style enables are tremendous but on the flip side of the coin, the price of the mistakes is equally very high.

1.6 What's next

In the next section of the tutorial we are going to talk about typical inter-service communication styles proven to fit well the microservice architecture.

Chapter 2

Microservices Communication

2.1 Introduction

Microservice architecture is essentially a journey into engineering of the distributed system. As more and more microservices are being developed and deployed, most likely than not they have to talk to each other somehow. And these means of the communication vary not only by transport and protocol, but also if they happen synchronously or asynchronously.

In this section of the tutorial we are going to talk about most widely used styles of communication applied to microservice architecture. As we are going to see, each one has own pros and cons, and the right choice heavily depends on the application architecture, requirements and business constraints. Most importantly, you are not obligated to pick just one and stick to it. Embodying different communication patterns between different groups of microservices, depending on their role, specification and destiny, is absolutely possible. It worth reminding one of the core principles of the microservices we have talked about in the opening part of the tutorial: pick the right tool for the job.

2.2 Using HTTP

In present-day world, HTTP is very likely the most widely used communication protocol out there. It is one of the foundational pieces of the World Wide Web and despite being unchanged for quite a long time, it went through the major revamp recently to address the challenges of modern web applications.

The semantic of the HTTP protocol is genuinely simple but at the same time flexible and powerful enough. There are several major interaction paradigms (or styles) built on top of HTTP protocol (more precisely, HTTP/1.1) which are clearly in dominant position with respect to the microservice architecture implementations.

2.2.1 SOAP

SOAP (or **Simple Object Access Protocol**) is one of the first specifications for exchanging structured information in the implementation of the web services. It was designed way back in 1998 and is centered on XML messages transferred primarily over HTTP protocol.

One particularly innovative idea which came out of the evolution of SOAP protocol is Web Services Description Language (or just WSDL): an XML-based interface definition language that was used for describing the functionality offered by SOAP web services. As we are going to see later on, the lessons learned from the WSDL taught us that, in some form or another, the notion of the explicit service contract (or schema, specification, description) is absolutely necessary to bridge providers and consumers together. SOAP is over 20 years old, why even bother mentioning it? Surprisingly, there are quite a lot of systems which interface using SOAP web services and are still very heavily utilized.

2.2.2 REST

For many, the appearance of the REST architectural style signified the end of the SOAP era (which turned out not to be true strictly speaking).

Representational State Transfer (**REST**) is an architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

By using a stateless protocol and standard operations, REST systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.

https://en.wikipedia.org/wiki/Representational_state_transfer

The roots of term representational state transfer go back to 2000 when Roy Fielding introduced and defined it in his famous doctoral dissertation "Architectural Styles and the Design of Network-based Software Architectures".

Interestingly, REST architectural style is basically agnostic to the protocol being used but gained tremendous popularity and adoption because of HTTP. This is not a coincidence, since the web applications and APIs represent a significant chunk of the applications these days.

There are six constraints which the system or application should meet in order to qualify as RESTful. All of them actually play very well by the rules of the microservice architecture.

- **Uniform Interface:** it does not matter who is the client, the requests look the same.
- **Separation of the client and the server** : servers and clients act independently (separation of concerns).
- **Statelessness** : no client-specific context is being stored on the server between requests and each request from any client contains all the information necessary to be serviced.
- **Cacheable** : clients and intermediaries can cache responses, whereas responses implicitly or explicitly define themselves as cacheable or not to prevent clients from getting stale data.
- **Layered system** : a client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way.
- **Code on demand (optional):** servers can temporarily extend or customize the functionality of a client by transferring executable code (usually some kind of scripts)

REST, when used in the context of HTTP protocol, relies on resources, uniform resource locators (URLs), standard HTTP methods, headers and status codes to design the interactions between servers and clients. The table below outlines the typical mapping of the HTTP protocol semantics to the imaginable library management web APIs designed after REST architectural style.

URL: https://api.library.com/books/						
GET	**PUT**	**PATCH**	**POST**	**DELETE**	**OPTIONS**	**HEAD**
Retrieve all resources in a collection.	**Replace** the entire collection with another collection.	Not generally used.	**Create** a new entry in the collection. The new entry's URI is usually returned by the operation.	**Delete** the entire collection.	**List** available HTTP methods (and may be other options).	**Retrieve** all resources in a collection (should return headers only).
URL: https://api.library.com/books/17						
GET	**PUT**	**PATCH**	**POST**	**DELETE**	**OPTIONS**	**HEAD**
Retrieve a representation of the single resource.	**Replace** the resource entirely (or **create** it if it does not exist yet).	**Update** the resource (usually, partially.	Not generally used.	**Delete** the resource.	**List** available HTTP methods (and may be other options).	**Retrieve** a single resource (should return headers only).

Idempotent: **yes**	Idempotent: **yes**	Idempotent: **no**	Idempotent: **no**	Idempotent: **yes**	Idempotent: **yes**	Idempotent: **yes**
Safe: **yes**	Safe: **no**	Safe: **no**	Safe: **no**	Safe: **no**	Safe: **yes**	Safe: **yes**

There are other subtle but exceptionally important expectations (or implicit premises if you will) associated with each HTTP method in context of REST (and microservices in particular): idempotency and safety. An operation is considered idempotent when even if the same input is sent to it multiple times, the effect will be the same (as sending this input only once). Consequently, the operation is safe if does not modify the resource (or resources). The assumptions regarding idempotency and safety are critical to handling failures and making the decisions about mitigating them.

To sum up, it is very easy to get started building RESTful web APIs since mostly every programming language has HTTP server and client baked into its base library. Consuming them is no brainer as well: either from command line (curl, httpie), using specialized desktop clients (Postman, Insomnia), or even from web browser (though not much you could do without installing the additional plugins).

This simplicity and flexibility of REST comes at a price: the lack of first-class support of discoverability and introspection. The agreement between server and client on the resources and the content of the input and output is out of band knowledge.

The API Stylebook with its Design Guidelines and Design Topics is a terrific resource to learn about the best practices and patterns for building magnificent RESTful web APIs. By the way, if you get an impression that REST architectural style restricts your APIs to follow the CRUD (Create/Read/Update/Delete) semantic, this is certainly a myth.

2.2.3 REST: Contracts on the Rescue

The lack of explicit, shareable, descriptive contract (besides the static documentation) for RESTful web APIs was always an area of active research and development in the community. Luckily, the efforts have been culminated recently into establishing the OpenAPI Initiative and releasing OpenAPI 3.0 specification (previously known as Swagger).

The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPI Specification removes guesswork in calling a service. - https://github.com/OAI/OpenAPI-Specification

OpenAPI is not the de-facto standard everyone is obligated to use but a well-thought, comprehensive mean to manage the contracts of your RESTful web APIs. Yet another benefit it comes with, as we are going to see later on in the tutorial, is that the tooling around OpenAPI is just amazing.

Among alternative options it is worth to mention API Blueprint, RAML, Apiary and Apigee. Honestly, it does not really matter what you are going to use, the shift towards contract-driven development and collaboration does.

2.2.4 GraphQL

Everything is moving forward and the dominant positions of REST were being shaken by the new kid on the block, namely GraphQL.

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools. - https://graphql.org/

GraphQL has an interesting story. It was originally created at Facebook in 2012 to address the challenges of handling their data models for client / server applications. The development of the GraphQL specification in the open started only in 2015 and since then this pretty much new technology is steadily gaining the popularity and widespread adoption.

GraphQL is not a programming language capable of arbitrary computation, but is instead a language used to query application servers that have capabilities defined in this specification. GraphQL does not mandate a particular programming language or storage system for application servers that implement it. Instead, application servers take their capabilities and map them to a

uniform language, type system, and philosophy that GraphQL encodes. This provides a unified interface friendly to product development and a powerful platform for tool-building. - https://facebook.github.io/graphql/June2018/

What makes GraphQL particularly appealing for microservices is a set of its core design principles:

- **It is hierarchical** : Most of the data these days is organized into hierarchical structures. To achieve congruence with such reality, a GraphQL query itself is structured hierarchically.
- **Strong -typing** : Every application declares own type system (also known as schema). Each GraphQL query is executed within the context of that type system whereas GraphQL server enforces the validity and correctness of such query before executing it.
- **Client -specified queries** : A GraphQL server publishes the capabilities that are available for its clients. It becomes the responsibility of the client to specifying exactly how it is going to consume those published capabilities so the given GraphQL query returns exactly what a client asks for.
- **Introspective** : The specific type system which is managed by a particular GraphQL server must be queryable by the GraphQL language itself.

GraphQL puts clients in control of what data they need. Although it has some drawbacks, the compelling benefits of strong typing and introspection often make GraphQL a favorable option.

Unsurprisingly, most of the GraphQL implementations are also HTTP-based and for good reasons: to serve as a foundation for building web APIs. In the nutshell, the GraphQL server should handle only HTTP `GET` and `POST` methods. Since the conceptual model in GraphQL is an entity graph, such entities are not identified by URLs. Instead, a GraphQL server operates on a single endpoint (usually `/graphql`) which handles all requests for a given service.

Surprisingly (or not?), many people treat GraphQL and REST as direct competitors: you have to pick one or another. But the truth is that both are excellent choices and can happily coexist to solve the business problems in a most efficient ways. This is what microservices are all about, right?

The implementations of GraphQL exist in many programming languages (for example, graphql-java for Java, Sangria for Scala, just to name a few) but the JavaScript one is outstanding and set the pace for entire ecosystem.

Let us take a look on a how the RESTful web APIs from the previous section could be described in the terms of GraphQL schema and types.

```
schema {
  query: Query
  mutation: Mutation
}

type Book {
  isbn: ID!
  title: String!
  year: Int
}

type Query {
  books: [Book]
  book(isbn: ID!): Book
}

# this schema allows the following mutation:
type Mutation {
  addBook(isbn: ID!, title: String!, year: Int): Book
  updateBook(isbn: ID!, title: String, year: Int): Book
  removeBook(isbn: ID!): Boolean
}
```

The separation between mutations and queries provides natural explicit guarantees about the safety of the particular operation.

It is fair to say that GraphQL slowly but steadily is changing the web APIs landscape as more and more companies are adapting it or have adapted already. You may not expect it but RESTful and GraphQL are often deployed side by side. One of the new

patterns emerged of such co-existence is backends for frontends (**BFF**) where the GraphQL web APIs are fronting the RESTful web services.

2.3 Not only HTTP

Although HTTP is the king, there are a couple of communication frameworks and libraries which go beyond that. Like, for example RPC-style conversations, the oldest form of inter-process communication.

Essentially, RPC is a request-response protocol where the client sends a request to a remote server to execute a specified procedure with supplied parameters. Unless the communication between client and server is asynchronous, the client usually blocks till the remote server sends a response back. Although quite efficient (most of the time the exchange format is a binary one), RPC used to have a huge issues with interoperability and portability across different languages and platforms. So why to rake over old ashes?

2.3.1 gRPC

The HTTP/2, a major revision of the HTTP protocol, unblocked the new ways to drive the communications on the web. gRPC, a popular, high performance, open-source universal RPC framework from Google, is the one who bridges the RPC semantics with HTTP/2 protocol.

To add a note here, although gRPC is more or less agnostic to the underlying transport, there is no other transport supported besides HTTP/2 (and there are no plans to change that in the immediate future). Under the hood, gRPC is built on top of another widely adopted and matured piece of the technology from Google, called protocol buffers.

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data - think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format. - https://developers.google.com/-protocol-buffers/docs/overview

By default, gRPC uses protocol buffers as both its Interface Definition Language (**IDL**) and as its underlying message interchange format. The IDL contains the definitions of all data structures and services and carry on the contract between gRPC server and its clients.

For example, here is very simplified attempt to redefine the web APIs from the previous sections using protocol buffers specification.

```
syntax = "proto3";

import "google/protobuf/empty.proto";

option java_multiple_files = true;
option java_package = "com.javacodegeeks.library";

package library;

service Library {
  rpc addBook(AddBookRequest) returns (Book);
  rpc getBooks(Filter) returns (BookList);
  rpc removeBook(RemoveBookRequest) returns (google.protobuf.Empty);
  rpc updateBook(UpdateBookRequest) returns (Book);
}

message Book {
  string title = 1;
  string isbn = 2;
  int32 year = 3;
}
```

```
message RemoveBookRequest {
  string isbn = 1;
}

message AddBookRequest {
  string title = 1;
  string isbn = 2;
  int32 year = 3;
}

message UpdateBookRequest {
  string isbn = 1;
  Book book = 2;
}

message Filter {
  int32 year = 1;
  string title = 2;
  string isbn = 3;
}

message BookList {
  repeated Book books = 1;
}
```

gRPC provides bindings for many mainstream programming languages and relies on the protocol buffers tools and plugins for code generation (but if you are programming in Go, you are in a luck since the Go language ecosystem is the state of the art there). gRPC is an excellent way to establish efficient channels for internal service-to-service or service-to-consumer communication.

A lot of exciting developments are happing around gRPC these days. The most promising one is gRPC for Web Clients (currently in beta) which is going to provide a JavaScript client library that lets browser clients to access gRPC servers directly.

2.3.2 Apache Thrift

To be fair, gRPC is not the only RPC-style framework available. The Apache Thrift is another one dedicated to scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly between many languages. Apache Thrift is specifically designed to support non-atomic version changes across client and server code. It is very similar to gRPC and protocol buffers and shares the same niche. While it is not as popular as gRPC, it supports bindings for 25 programming languages and relies on modular transport mechanism (HTTP included). Apache Thrift has own dialect of the Interface Definition Language which resembles protocol buffers quite a lot. To compare with, here is another version of our web APIs definition, rewritten using Apache Thrift.

```
namespace java com.javacodegeeks.library

service Library {
  void addBook(1: Book book),
  list getBooks(1: Filter filter),
  bool removeBook(1: string isbn),
  Book updateBook(1: string isbn, 2: Book book)
}

struct Book {
  1: string title,
  2: string isbn,
  3: optional i32 year
}

struct Filter {
  1: optional i32 year;
  2: optional string title;
```

```
  3: optional string isbn;
}
```

2.3.3 Apache Avro

Last but not least, Apache Avro, a data serialization system, is often used for RPC-style communication and message exchanges. What distinguishes Apache Avro from others is the fact that the schema is represented in JSON format, for example, here is our web APIs translated to Apache Avro.

```
{
  "namespace": "com.javacodegeeks.avro",
  "protocol": "Library",

  "types": [
    {
      "name": "Book",
      "type": "record",
      "fields": [
          {"name": "title", "type": "string"},
          {"name": "isbn", "type": "string"},
          {"name": "year", "type": "int"}
      ]
    }
  ],

  "messages": {
     "addBook": {
       "request": [{"name": "book", "type": "Book"}],
       "response": "null"
     },
     "removeBook": {
       "request": [{"name": "isbn", "type": "string"}],
       "response": "boolean"
     },
     "updateBook": {
       "request": [
           {"name": "isbn", "type": "string"},
           {"name": "book", "type": "Book"}
       ],
       "response": "Book"
     }
  }
}
```

Another unique feature of Apache Avro is to make out different kind of specifications, based on the file name extensions, for example:

- ***.avpr** : defines a Avro Protocol specification
- ***.avsc** : defines an Avro Schema specification
- ***.avdl:** defines an Avro IDL

Similarly to Apache Thrift, Apache Avro supports different transports (which additionally could be either stateless or stateful), including HTTP.

2.4 REST, GraphQL, gRPC, Thrift . . . how to choose?

To understand where each of these communication styles fit the best, the Understanding RPC, REST and GraphQL article is a great starting point.

2.5 Message passing

The request-response is not the only method to structure the communication in distributed systems and microservices in particular. Message passing is another communication style, asynchronous by nature, which revolves around exchanging messages between all the participants. Messaging is heart and soul of the even-driven applications and microservices. In practice, they are implemented primarily on the principles of Event Sourcing or Command Query Responsibility Segregation (**CQRS**) architectures however the definition of what it means to be event-driven goes broader than that.

To be fair, there is tremendous amount of different options to talk about and pick from. So to keep it sane, we are going to focus more on a core concept rather than concrete solutions.

2.5.1 WebSockets and Server-Sent Events

If your microservice architecture constitutes of RESTful web services, picking a native HTTP messaging solution is a logical way to go.

The WebSocket protocol enables bidirectional (full-duplex[full-duplex]) communication channels between a client and a server over a single connection. Interestingly, the WebSocket is an independent TCP-based protocol but at the same time "... it is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries ..." (https://tools.ietf.org/-html/rfc6455).

For non-bidirectional communication, server-sent events (or in short, **SSE**) is a great, simple way to enable servers to push the data to the clients over HTTP (or using dedicated server-push protocols).

With the raising popularity of HTTP/2, the role of WebSocket and server-sent events is slowly diminishing since most of their features are already backed into the protocol itself.

2.5.2 Message Queues and Brokers

Messaging is exceptionally interesting and crowded space in software development. Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), Simple (or Streaming) Text Orientated Messaging Protocol (STOMP), Apache Kafka, NATS, NSQ, ZeroMQ, not to mention Redis Pub/Sub, upcoming Redis Streams and tons of cloud solutions. What to say, even PostgreSQL includes one!

Depending on your application needs, it is very likely you could find more than one message broker to choose from. However, there is an interesting challenge which you may need to solve:

- efficiently publish the message schemas (to share what is packed into message)
- evolve the message schemas over time (ideally, without breaking things)

Surprisingly, our old friends protocol buffers, Apache Thrift and Apache Avro could be an excellent fit for these purposes. For example, Apache Kafka is often used with Schema Registry to store a versioned history of all message schemas. The registry is built on top of Apache Avro.

Other interesting libraries we have not talked about (since they are purely oriented on message formats, not services or protocols) are FlatBuffers, Cap'n Proto and MessagePack.

2.5.3 Actor Model

The actor model, originated in 1973, introduces the concept of actors as the universal primitives of concurrent computation which communicate with each other by sending messages asynchronously. Any actor, in the response to a message it receives, can do concurrently one of the following things:

- send a finite number of messages to other actors
- instantiate a finite number of new actors
- change the designated behavior to process the next message it receives

The consequences of using message passing are that actors do not share any state with each other. They may modify their own private state, but can only affect each other through messages.

You may have heard about Erlang, a programming language to build massively scalable soft real-time systems with requirements on high availability. It is one of the best examples of successful actor model implementation.

On JVM, the unquestionable leader is Akka: a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. It started as the actor model implementation but over the years has grown into full-fledged Swiss knife for distributed system developers.

Frankly speaking, the ideas and principles behind the actor model make it a serious candidate for implementing microservices.

2.5.4 Aeron

For a highly efficient and latency-critical communications the frameworks we have discussed so far may not be a best choice. You can certainly fallback to custom-made TCP/ UDP transport but there is a good set of options out there. Aeron is an efficient reliable UDP unicast, UDP multicast, and IPC message transport. It supports Java out of the box with performance being the key focus. Aeron is designed to be the highest throughput with the lowest and most predictable latency possible of any messaging system. Aeron integrates with Simple Binary Encoding (SBE) for the best possible performance in message encoding and decoding.

2.5.5 RSocket

RSocket is a binary protocol for use on byte stream transports such as TCP, WebSockets, and Aeron. It supports multiple symmetric interaction models via asynchronous message passing using just a single connection:

- request/response (stream of 1)
- request/stream (finite stream of many)
- fire-and-forget (no response)
- channel (bi-directional streams)

Among other things, it supports session resumption which allows to resume long-lived streams across different transport connections. This is particularly useful when network connections drop, switch, and reconnect frequently.

2.6 Cloud native

Cloud computing is certainly the place where the most of the applications are being deployed nowadays. The heated fights for the market share replenish the continuous streams of innovations. One of those is serverless computing where the cloud provider takes care of server management and capacity planning decisions dynamically. The presence of the term serverless is a bit confusing since the servers are still required, but the deployment and execution models change.

The exciting part is that serverless code can be used along with the application deployed as more traditional microservices. Even more, the whole application designed after microservice architecture could be built on top of purely serverless components, dramatically decreasing the operational burden.

In case the serverless computing sounds new to you, the good introduction to such architecture is given in this post on Martin Fowler's blog.

2.6.1 Function as a service

One of the best examples of serverless computing in action is function as a service (**FaaS**). As you may guess, the unit of deployment in such a model is a function (ideally, in any language, but Java, JavaScript and Go are most likely the ones you could realistically use right now). The functions are expected to start within a few milliseconds in order to handle the individual requests or to react on the incoming messages. When not used, the functions are not consuming any resources, incurring no charges at all.

Each cloud provider offers own flavor of function as a service platform but it is worth mentioning Apache OpenWhisk, OpenFaaS and riff projects, a couple of open-source well-established function as a service implementations.

2.6.2 Knative

This is literally a newborn member of the serverless movement, public announced by Google just a few weeks ago.

Knative components extends Kubernetes to provide a set of middleware components that are essential to build modern, source-centric, and container-based applications that can run anywhere: on premises, in the cloud, or even in a third-party data center. ... Knative components offer developers Kubernetes-native APIs for deploying serverless-style functions, applications, and containers to an auto-scaling runtime. - https://github.com/knative/docs

Knative is in very early stages of development but the potential impact of it on the serverless computing could be revolutionary.

2.7 Conclusions

Over the course of this section we have talked about many different styles to structure the communication between microservices (and their clients) in the applications which follow microservice architecture. We have understood the criticality and importance of the schema or/and contract as the essential mean of establishing healthy collaboration between service providers and consumers (think teams within organization). Last but not least, the combination of multiple communication styles is certainly possible and makes sense, however such decisions should be driven by real needs rather than hype (sadly, it happens too often in the industry).

2.8 What's next

In the next section of the tutorial we are going to evaluate the Java landscape and most widely used frameworks for building production-grade microservices on JVM.

The complete set of specification files is available for download.

Chapter 3

The Java / JVM Landscape

3.1 Introduction

In the previous part of the tutorial we have covered a broad range of communication styles widely used while building microservices. It is time to put this knowledge into practical perspective by talking about most popular and battle-tested Java libraries and frameworks which may serve as the foundation of your microservice architecture implementation.

Although there are quite a few old enough to remember the SOAP era, many of the frameworks we are going to discuss shortly are fairly young, and often quite opinionated. The choice of which one is right for you is probably the most important decision you are going to make early on. Beside the JAX-RS specification (and more generic Servlet specification), there are no industry-wide standards to guarantee the interoperability between different frameworks and libraries on JVM platform, so make the call wisely.

There will not be any comparison to promote one framework or library over another since each has own goals, philosophy, community, release cycles, roadmaps, integrations, scalability and performance characteristics. There are just too many factors to account for, taking into the context the application and organization specifics.

However, a couple of valuable resources could be of a great help. The Awesome Microservices repository is a terrific curated list of microservice architecture related principles and technologies. In the same vein, the TechEmpower's Web Framework Benchmarks provides a number of interesting and useful insights regarding the performance of several web application platforms and frameworks, and not only the JVMs ones.

3.2 Staying RESTy

The most crowded space is occupied by the frameworks and libraries which promote REST architectural style over HTTP protocol. Nearly all the nominees in this category are very matured and well-established brands, deployed in productions for years.

3.2.1 JAX-RS: RESTful Java in the Enterprise

The JAX-RS specification, also known as JSR-370 (and previously outlined in the JSR-339 and JSR-311), defines a set of Java APIs for the development of web services built according to the REST architectural style. It is fairly successful effort with many implementations available to select from and, arguably, the number one preference in the enterprise world.

The JAX-RS APIs are driven by Java annotations and generally could be ported from one framework to another quite smoothly. In addition, there is tight integration with other Java platform specifications, like Contexts and Dependency Injection for Java (JSR-365), Bean Validation (JSR-380), Java API for JSON Processing (JSR-374) to name a few.

Getting back to the imaginable library management web APIs we have talked about in the previous part of the tutorial, the typical (but very simplified) JAX-RS web service implementation may look like this:

```
@Path("/library")
public class LibraryRestService {
    @Inject private LibraryService libraryService;

    @GET
    @Path("/books")
    @Produces(MediaType.APPLICATION_JSON)
    public Collection<Book> getAll() {
        return libraryService.getBooks();
    }

    @POST
    @Path("/books")
    @Produces(MediaType.APPLICATION_JSON)
    public Response addBook(@Context UriInfo uriInfo, Book payload) {
        final Book book = libraryService.addBook(payload);

        return Response
            .created(
                uriInfo
                    .getRequestUriBuilder()
                    .path(book.getIsbn())
                    .build())
            .entity(book)
            .build();    }

    @GET
    @Path("/books/{isbn}")
    @Produces(MediaType.APPLICATION_JSON)
    public Book findBook(@PathParam("isbn") String isbn) {
        return libraryService
            .findBook(isbn)
            .orElseThrow(() -> new NotFoundException("No book found for ISBN: " + isbn));
    }
}
```

It should work on any framework which is fully compliant with the latest JAX-RS 2.1 specification (JSR-370) so let us take it from there.

3.2.2 Apache CXF

Apache CXF, an open source services framework, celebrates its 10th anniversary this year! Apache CXF helps to build and develop services using frontend programming APIs, like JAX-WS and JAX-RS, which can speak a variety of protocols such as SOAP, REST, SSE (even CORBA) and work over a variety of transports such as HTTP, JMS or JBI.

What makes Apache CXF a great fit for microservices is the fact that it has outstanding integrations with many other projects, notably: OpenAPI for contract-driven development, Brave / OpenTracing / Apache HTrace for distributed tracing, JOSE / OAuth2 / OpenID Connect for security.

3.2.3 Apache Meecrowave

Meecrowave is a very lightweight, easy to use microservices framework , built exclusively on top of other great Apache projects: Apache OpenWebBeans (CDI 2.0), Apache CXF (JAX-RS 2.1), and Apache Johnzon (JSON-P). It significantly reduces the development time since all the necessary pieces are already wired together.

3.2.4 RESTEasy

RESTEasy from RedHat / JBoss is another fully certified and portable implementation of the JAX-RS 2.1 specification. One of

the advantages of the RESTEasy framework is tighter integration with WildFly Application Server, in case it might be important in your context.

3.2.5 Jersey

Jersey is an open source, production quality framework for developing RESTful web services in Java. In fact, it serves as a JAX-RS (JSR-370, JSR-339 and JSR-311) reference implementation. Similarly to other frameworks, Jersey goes way beyond just being JAX-RS implementation and provides additional features, extensions and utilities to further simplify the development of RESTful web APIs and clients.

3.2.6 Dropwizard

Dropwizard is yet another great Java framework for developing RESTful web services and APIs with the emphasis on the operational friendliness and high performance. It is built on top of Jersey framework and combines together best of breed libraries from the entire Java ecosystem. In that sense, it is quite opinionated, but at the same time known to be simple, stable **,** mature and light-weight **.**

Dropwizard has out-of-the-box support for sophisticated configuration, application metrics, logging, operational tools, and much more, allowing you and your team to ship a production-quality web service in the shortest time possible. - https://www.dropwizard.io/-1.3.5/docs

It is worth noting that Dropwizard truly established the instrumentation and monitoring baseline for modern Java applications (you may have heard about Metrics library born from it) and is really good choice for building microservices.

3.2.7 Eclipse Microprofile: thinking in microservices from the get-go

While talking about microservices in Java universe it is impossible not to mention very recent initiative undertaken by Eclipse Foundation, known as MicroProfile.

The MicroProfile is a baseline platform definition that optimizes Enterprise Java for a microservices architecture and delivers application portability across multiple MicroProfile runtimes. The initially planned baseline is JAX-RS + CDI + JSON-P, with the intent of community having an active role in the MicroProfile definition and roadmap. - https://microprofile.io/faq

The primary goal of MicroProfile is to accelerate the pace of innovation in the space of the enterprise Java, which traditionally is suffering from very slow processes. It is certainly worth keeping an eye on.

3.2.8 Spring WebMvc / WebFlux

Long time ago Spring Framework, initially the implementation of the inversion of control (IoC) design principle, had shaken the world of the enterprise Java, filled with monstrous frameworks and specifications. It provided an easy and straightforward way to use dependency injection capabilities in Java applications.

The IoC and dependency injection are still at the core of Spring Framework but there are so many projects under its umbrella that nowadays it is more like a platform than anything else. As for this section, the one we are interested in is Spring Web MVC, the original web framework built on the Servlet API and widely used for traditional RESTful web services, like our library management APIs for example.

```
@RestController
@RequestMapping("/library")
public class LibraryController {
    @Autowired private LibraryService libraryService;

    @RequestMapping(path = "/books/{isbn}", method = GET, produces = APPLICATION_JSON_VALUE ↩
        )
    public ResponseEntity<Book> findBook(@PathVariable String isbn) {
        return libraryService
            .findBook(isbn)
```

```
            .map(ResponseEntity::ok)
            .orElseGet(ResponseEntity.notFound()::build);
    }

    @RequestMapping(path = "/books", method = GET, produces = APPLICATION_JSON_VALUE)
    public Collection<Book> getAll() {
        return libraryService.getBooks();
    }

    @RequestMapping(path = "/books", method = POST, consumes = APPLICATION_JSON_VALUE)
    public ResponseEntity<Book> addBook(@RequestBody Book payload) {
        final Book book = libraryService.addBook(payload);

        return ResponseEntity
            .created(linkTo(methodOn(LibraryController.class).findBook(book.getIsbn())). ↩
                toUri())
            .body(book);
    }
}
```

It would be unfair not to mention a relatively new Spring WebFlux project, the counterpart of the Spring Web MVC, built on top of the reactive stack and non-blocking I/O.

Innovative, productive and hugely successful, Spring Framework is the number one choice for Java developers these days, particularly with respect to microservice architecture.

3.2.9 Spark Java

Spark, also often referred as Spark-Java to eliminate the confusion with similarly named hyper-popular data processing framework, is a micro framework for creating web applications in Java with a minimal effort. It is heavily relying on its expressive and simple DSL, designed by leveraging the power of Java 8 lambda expressions. It indeed leads to quite compact and clean code. For example, here is a sneak peak on our library management APIs definition:

```
path("/library", () -> {
    get("/books",
        (req, res) -> libraryService.getBooks(),
        json()
    );

    get("/books/:isbn",
        (req, res) -> libraryService
            .findBook(req.params(":isbn"))
            .orElseGet(() -> {
                res.status(404);
                return null;
            }),
        json()
    );

    post("/books",
        (req, res) -> libraryService
            .addBook(JsonbBuilder.create().fromJson(req.body(), Book.class)),
        json()
    );
});
```

Beside just Java, Spark-Java has first class Kotlin support and although it is mainly used for creating REST APIs, it integrates with a multitude of template engines.

3.2.10 Restlet

Restlet framework aims to help Java developers build better web APIs that follow REST architectural style. It provides pretty powerful routing and filtering capabilities along with offering numerous extensions. It has quite unique way to structure and bundle things together.

```
public class BookResource extends ServerResource {
    @Inject private LibraryService libraryService;

    @Get
    public Book findBook() {
        final String isbn = (String) getRequest().getAttributes().get("isbn");
        return libraryService.findBook(isbn).orElse(null);
    }
}

public class BooksResource extends ServerResource  {
    @Inject private LibraryService libraryService;

    @Get
    public Collection<Book> getAll() {
        return libraryService.getBooks();
    }

    @Post("json")
    public Representation addBook(Book payload) throws IOException {
        return toRepresentation(libraryService.addBook(payload));
    }
}
```

And here are the bindings between the resources and their URI, basically the typical router.

```
final Router router = new Router(getContext());
router.attach("/library/books/{isbn}", BookResource.class);
router.attach("/library/books", BooksResource.class);
```

Interestingly, Restlet is one of the few open-source frameworks which was able to grew up from a simple library to a full-fledged RESTful APIs development platform.

3.2.11 Vert.x

The Vert.x project from Eclipse Foundation is an open-source toolkit for building reactive applications on the JVM platform. It follows the reactive paradigm and is designed from the ground up to be event-driven and non-blocking.

It has tons of different components. One of them is Vert.x-Web, designated for writing sophisticated modern web applications and HTTP-based microservices. The snippet below showcases our library management web API implemented on top of Vert.x-Web.

```
final LibraryService libraryService = ...;

final Vertx vertx = Vertx.vertx();
final HttpServer server = vertx.createHttpServer();
final Router router = Router.router(vertx);

router
    .get("/library/books")
    .produces("application/json")
    .handler(context ->
        context
            .response()
            .putHeader("Content-Type", "application/json")
            .end(Json.encodePrettily(libraryService.getBooks())))
```

```
    );

router
    .get("/library/books/:isbn")
    .produces("application/json")
    .handler(context ->
        libraryService
            .findBook(context.request().getParam("isbn"))
            .ifPresentOrElse(
                book -> context
                        .response()
                        .putHeader("Content-Type", "application/json")
                        .end(Json.encodePrettily(book)),
                () -> context
                        .response()
                        .setStatusCode(204)
                        .putHeader("Content-Type", "application/json")
                        .end()
            )
    );

router
    .post("/library/books")
    .consumes("application/json")
    .produces("application/json")
    .handler(BodyHandler.create())
    .handler(context -> {
        final Book book = libraryService
            .addBook(context.getBodyAsJson().mapTo(Book.class));

        context
            .response()
            .putHeader("Content-Type", "application/json")
            .end(Json.encodePrettily(book));
    });

server.requestHandler(router::accept);
server.listen(4567);
```

Some of the notable characteristics of the Vert.x are high performance and modular design. More to that, it is pretty lightweight, scales really well and natively supports Java, JavaScript, Groovy, Ruby, Ceylon, Scala and Kotlin.

3.2.12 Play Framework

Play is high velocity, hyper-productive web framework for Java and Scala. It is based on a lightweight, stateless, web-friendly architecture and is built on top of Akka Toolkit. Although Play is full-fledged framework with exceptionally powerful templating engine, it is very well suited for RESTful web services development as well. Our library management web APIs could be easily designed in Play.

```
GET    /library/books           controllers.LibraryController.list
GET    /library/books/:isbn     controllers.LibraryController.show(isbn: String)
POST   /library/books           controllers.LibraryController.add
```

```
public class LibraryController extends Controller {
    private LibraryService libraryService;

    @Inject
    public LibraryController(LibraryService libraryService) {
        this.libraryService = libraryService;
    }
```

```
    public Result list() {
        return ok(Json.toJson(libraryService.getBooks()));
    }

    public Result show(String isbn) {
        return  libraryService
            .findBook(isbn)
            .map(resource -> ok(Json.toJson(resource)))
            .orElseGet(() -> notFound());
    }

    public Result add() {
        JsonNode json = request().body().asJson();
        final Book book = Json.fromJson(json, Book.class);
        return created(Json.toJson(libraryService.addBook(book)));
    }
}
```

The ability of Play to serve both backend (RESTful) and frontend (using for example Angular, React, Vue.js) endpoints, in other words going full-stack, might be an attractive offering with respect to implementing microservices using a single framework (although such decisions should be taken with a great care).

3.2.13 Akka HTTP

Akka HTTP, the part of amazing Akka Toolkit family, provides a full server-side and client-side HTTP stack implemented on top of actor model. It's not a full-fledged web framework (like Play, for example) but crafted specifically to manage HTTP-based services. Similarly to other frameworks, Akka HTTP has own DSL to elegantly define RESTful web API endpoints. The best way to give it a try is to create our library management API definitions.

```
public class LibraryRoutes extends AllDirectives {
    private final LibraryService libraryService;

    // ...

    public Route routes() {
        return route(
            pathPrefix("library", () ->
                pathPrefix("books", () ->
                    route(
                        pathEndOrSingleSlash(() ->
                            route(
                                get(() ->
                                    complete(StatusCodes.OK, libraryService.getBooks(),  ↩
                                        Jackson.marshaller())
                                ),
                                post(() ->
                                    entity(
                                        Jackson.unmarshaller(Book.class),
                                        payload -> complete(StatusCodes.OK, libraryService. ↩
                                            addBook(payload), Jackson.marshaller())
                                    )
                                )
                            )
                        ),
                        path(PathMatchers.segment(), isbn ->
                            route(
                                get(() ->
                                    libraryService
```

```
                                        .findBook(isbn)
                                        .map(book -> (Route)complete(StatusCodes.OK, book,  ↩
                                            Jackson.marshaller()))
                                        .orElseGet(() -> complete(StatusCodes.NOT_FOUND))
                                )
                            )
                        )
                    )
                )
            )
        );
    }
}
```

Akka HTTP has first-class support of Java and Scala and is an excellent choice for building RESTful web services and scalable microservice architecture in general.

3.2.14 Micronaut

Micronaut is a modern, JVM-based, full-stack framework for building modular, easily testable microservice applications. It is truly polyglot (in JVM sense) and offers support for Java, Groovy, and Kotlin out of the box. Micronaut's focus is a first-class support of reactive programming paradigm and compile-time dependency injection. Here is the skeleton of our library management web APIs declaration in Micronaut.

```
@Controller("/library")
public class LibraryController {
    @Inject private LibraryService libraryService;

    @Get("/books/{isbn}")
    @Produces(MediaType.APPLICATION_JSON)
    public Optional<Book> findBook(String isbn) {
        return libraryService.findBook(isbn);
    }

    @Get("/books")
    @Produces(MediaType.APPLICATION_JSON)
    public Observable<Book> getAll() {
        return Observable.fromIterable(libraryService.getBooks());
    }

    @Post("/books")
    @Consumes(MediaType.APPLICATION_JSON)
    public HttpResponse<Book> addBook(Book payload) {
        return HttpResponse.created(libraryService.addBook(payload));
    }
}
```

Comparing to others, Micronaut is very young but promising framework, focused on modern programming. It is a fresh start without the baggage accumulated over the years.

3.3 GraphQL, the New Force

The popularity of the GraphQL is slowly but steadily growing. It has proven to be an indispensable tool to address a wide range of the problems although there are not many choices available for Java developers. Along this section we are going to use the same GraphQL schema we have discussed in the previous part of the tutorial, repeated below just for convenience.

```
schema {
  query: Query
```

```
  mutation: Mutation
}

type Book {
  isbn: ID!
  title: String!
  year: Int
}

type Query {
  books: [Book]
  book(isbn: ID!): Book
}

type Mutation {
  addBook(isbn: ID!, title: String!, year: Int): Book
  updateBook(isbn: ID!, title: String, year: Int): Book
  removeBook(isbn: ID!): Boolean
}
```

3.3.1 Sangria

Sangria is the GraphQL implementation in Scala. This is a terrific framework with a vibrant community and seamless integration with Akka HTTP and/or Play Framework. Although it does not provide Java-friendly APIs at the moment, it is worth mentioning nonetheless. It takes a bit different approach by defining the schema along the resolvers in the code, for example.

```
object SchemaDefinition {
  val BookType = ObjectType(
    "Book", "A book.", fields[LibraryService, Book](
      Field("isbn", StringType, Some("The book's ISBN."), resolve = _.value.isbn),
      Field("title", StringType, Some("The book's title."), resolve = _.value.title),
      Field("year", IntType, Some("The book's year."), resolve = _.value.year)
    ))

  val ISBN = Argument("isbn", StringType, description = "ISBN")
  val Title = Argument("title", StringType, description = "Book's title")
  val Year = Argument("year", IntType, description = "Book's year")

  val Query = ObjectType(
    "Query", fields[LibraryService, Unit](
      Field("book", OptionType(BookType),
        arguments = ISBN :: Nil,
        resolve = ctx => ctx.ctx.findBook(ctx arg ISBN)),
      Field("books", ListType(BookType),
        resolve = ctx =>  ctx.ctx.getBooks())
    ))

  val Mutation = ObjectType(
    "Mutation", fields[LibraryService, Unit](
      Field("addBook", BookType,
        arguments = ISBN :: Title :: Year :: Nil,
        resolve = ctx => ctx.ctx.addBook(Book(ctx arg ISBN, ctx arg Title, ctx arg Year))),
      Field("updateBook", BookType,
        arguments = ISBN :: Title :: Year :: Nil,
        resolve = ctx => ctx.ctx.updateBook(ctx arg ISBN, Book(ctx arg ISBN, ctx arg Title, ↩
            ctx arg Year))),
      Field("removeBook", BooleanType,
        arguments = ISBN :: Nil,
        resolve = ctx => ctx.ctx.removeBook(ctx arg ISBN))
    ))
```

```
    val LibrarySchema = Schema(Query, Some(Mutation))
}
```

Although there are some pros and cons to the code-first schema development, it may be a good solution in certain microservice architecture implementations.

3.3.2 graphql-java

Easy to guess, graphql-java is the GraphQL implementation in Java. It has pretty good integration with Spring Framework as well as any other Servlet-compatible framework or container. In the case of such a simple GraphQL schema as ours, the implementation is just a matter of defining resolvers.

```
public class Query implements GraphQLQueryResolver {
    private final LibraryService libraryService;

    public Query(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    public Optional<Book> book(String isbn) {
        return libraryService.findBook(isbn);
    }

    public List<Book> books() {
        return new ArrayList<>(libraryService.getBooks());
    }
}
```

```
public class Mutation implements GraphQLMutationResolver {
    private final LibraryService libraryService;

    public Mutation(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    public Book addBook(String isbn, String title, int year) {
        return libraryService.addBook(new Book(isbn, title, year));
    }

    public Book updateBook(String isbn, String title, int year) {
        return libraryService.updateBook(isbn, new Book(isbn, title, year));
    }

    public boolean removeBook(String isbn) {
        return libraryService.removeBook(isbn);
    }
}
```

And this is literally it. If you are considering using GraphQL in some or across all services in your microservice architecture, the graphql-java could be a robust foundation to build upon.

3.4 The RPC Style

Comparing to RESTful or GraphQL, the efficiency of RPC conversations, specifically for service-to-service communication, is really hard to beat. The gRPC framework from Google is taking a lead here but this is not the only player.

As we are going to see, many RPC systems are built on the idea of defining a service contract: in Java it is typically an annotated interface definition. The server side implements this interface and exposes it over the wire, whereas on the client side this interface is used to get a proxy or a stub.

3.4.1 java-grpc

We have briefly glanced through gRPC generic concepts in the previous part of the tutorial, but in this section we are going to talk about its Java implementation - java-grpc. Since mostly everything is generated for you from the Protocol Buffers service definition, the only thing left to the developers is to provide the relevant service implementations. Here is the gRPC version of our library management service.

```
static class LibraryImpl extends LibraryGrpc.LibraryImplBase {
    private final LibraryService libraryService;

    public LibraryImpl(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    @Override
    public void addBook(AddBookRequest request, StreamObserver<Book> responseObserver) {
        final Book book = Book.newBuilder()
            .setIsbn(request.getIsbn())
            .setTitle(request.getTitle())
            .setYear(request.getYear())
            .build();

        responseObserver.onNext(libraryService.addBook(book));
        responseObserver.onCompleted();
    }

    @Override
    public void getBooks(Filter request, StreamObserver<BookList> responseObserver) {
        final BookList bookList = BookList
            .newBuilder()
            .addAllBooks(libraryService.getBooks())
            .build();
        responseObserver.onNext(bookList);
        responseObserver.onCompleted();
    }

    @Override
    public void updateBook(UpdateBookRequest request, StreamObserver<Book> responseObserver ↩
        ) {
        responseObserver.onNext(libraryService.updateBook(request.getIsbn(), request. ↩
            getBook()));
        responseObserver.onCompleted();
    }

    @Override
    public void removeBook(RemoveBookRequest request, StreamObserver<Empty>  ↩
        responseObserver) {
        libraryService.removeBook(request.getIsbn());
        responseObserver.onCompleted();
    }
}
```

It is worth mentioning that Protocol Buffers is the default but not the only serialization mechanism, gRPC could be used with JSON encoding as well. By and large, gRPC works amazingly well, and is certainly a safe bet with respect to implementing service-to-service communication in the microservice architecture. But more to come, stay tuned, grpc-web is around the corner.

3.4.2 Reactive gRPC

In the recent years reactive programming is steadily making its way to the mainstream. The Reactive gRPC is a suite of libraries to augment gRPC to work with Reactive Streams implementations. In the nutshell, it just generates alternative gRPC bindings, with respect to the library of your choice (RxJava 2 and Spring Reactor as of now), everything else stays pretty much unchanged. To prove it, let us take a look on the `LibraryImpl` implementation using Reactive Streams APIs.

```
static class LibraryImpl extends RxLibraryGrpc.LibraryImplBase {
    private final LibraryService libraryService;

    public LibraryImpl(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    @Override
    public Single<Book> addBook(Single<AddBookRequest> request) {
        return request
            .map(r ->
                Book
                    .newBuilder()
                    .setIsbn(r.getIsbn())
                    .setTitle(r.getTitle())
                    .setYear(r.getYear())
                    .build())
            .map(libraryService::addBook);
    }

    @Override
    public Single<BookList> getBooks(Single<Filter> request) {
        return request
            .map(r ->
                BookList
                    .newBuilder()
                    .addAllBooks(libraryService.getBooks())
                    .build());
    }

    @Override
    public Single<Book> updateBook(Single<UpdateBookRequest> request) {
        return request
            .map(r -> libraryService.updateBook(r.getIsbn(), r.getBook()));
    }

    @Override
    public Single<Empty> removeBook(Single<RemoveBookRequest> request) {
        return request
            .map(r -> {
                libraryService.removeBook(r.getIsbn());
                return Empty.newBuilder().build();
            });
    }
}
```

To be fair, Reactive gRPC is not the full-fledged gRPC implementation but rather an excellent addition to the java-grpc.

3.4.3 Akka gRPC

Yet another great tool from Akka Toolkit box, Akka gRPC provides support for building streaming gRPC servers and clients on top of Akka Streams (and Akka Toolkit in general). The Java-based implementation is relying on `CompletionStage` from the standard library and is quite straightforward to use.

```
public class LibraryImpl implements Library {
    private final LibraryService libraryService;

    public LibraryImpl(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    @Override
    public CompletionStage<Book> addBook(AddBookRequest in) {
        final Book book = Book
            .newBuilder()
            .setIsbn(in.getIsbn())
            .setTitle(in.getTitle())
            .setYear(in.getYear())
            .build();

        return CompletableFuture.completedFuture(libraryService.addBook(book));
    }

    @Override
    public CompletionStage<BookList> getBooks(Filter in) {
        return CompletableFuture.completedFuture(
            BookList
                .newBuilder()
                .addAllBooks(libraryService.getBooks())
                .build());
    }

    @Override
    public CompletionStage<Book> updateBook(UpdateBookRequest in) {
        return CompletableFuture.completedFuture(libraryService.updateBook(in.getIsbn(), in ↩
            .getBook()));
    }

    @Override
    public CompletionStage<Empty> removeBook(RemoveBookRequest in) {
        libraryService.removeBook(in.getIsbn());
        return CompletableFuture.completedFuture(Empty.newBuilder().build());
    }
}
```

Akka gRPC is quite a new member of the Akka Toolkit and is currently in the preview mode. It could be used already today but certainly expect some changes in the future.

3.4.4 Apache Dubbo

Apache Dubbo, currently under the incubation within Apache Software Foundation but initially developed at Alibaba, is a high-performance, Java-based, open source RPC framework. Our library service could be up and running just in a few lines of code.

```
final ServiceConfig serviceConfig = new ServiceConfig();
serviceConfig.setApplication(new ApplicationConfig("library-provider"));
serviceConfig.setRegistry(new RegistryConfig("multicast://224.5.6.7:1234"));
serviceConfig.setInterface(LibraryService.class);
serviceConfig.setRef(new LibraryServiceImpl());
serviceConfig.export();
```

It is purely Java-oriented so if you are aiming to build the polyglot microservice architecture, Apache Dubbo might not help you there natively but through additional integrations, like for example RPC over REST or RPC over HTTP.

3.4.5 Finatra and Finagle

Finagle, the RPC library, and Finatra, the services framework built on top of it, were born at Twitter and open-sourced shortly after.

Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers. Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency. - https://github.com/twitter/finagle

Finatra is a lightweight framework for building fast, testable, scala applications on top of TwitterServer and Finagle. - https://github.com/-twitter/finatra

They are both Scala-based and are actively used in production. Finagle was one of the first libraries to use Apache Thrift for service generation and binary serialization. Let us grab the library service IDL from the previous part of the tutorial and implement it as a Finatra service (as usual, most of the scaffolding code is generated on our behalf).

```
@Singleton
class LibraryController @Inject()(libraryService: LibraryService) extends Controller with  ↩
    Library.BaseServiceIface {
  override val addBook = handle(AddBook) { args: AddBook.Args =>
    Future.value(libraryService.addBook(args.book))
  }

  override val getBooks = handle(GetBooks) { args: GetBooks.Args =>
    Future.value(libraryService.getBooks())
  }

  override val removeBook = handle(RemoveBook) { args: RemoveBook.Args =>
    Future.value(libraryService.removeBook(args.isbn))
  }

  override val updateBook = handle(UpdateBook) { args: UpdateBook.Args =>
    Future.value(libraryService.updateBook(args.isbn, args.book))
  }
}
```

One of the distinguishing features of the Finagle is the out of the box support of distributed tracing and statistics for monitoring and diagnostics, invaluable insights for operating the microservices in production.

3.5 Messaging and Eventing

In microservice architecture, and generally in many loosely coupled distributed systems, some form of message and event passing is one of the basic building blocks. Along this section we are going to talk about few frameworks and outline a number of messaging solutions you may use independently of the framework of your choice.

3.5.1 Axon Framework

Axon is a lightweight, open-source Java framework to build scalable, extensible event-driven applications. It is one of the pioneers to employ the sound architectural principles of domain-driven design (DDD) and Command and Query Responsibility Segregation (CQRS) in practice.

3.5.2 Lagom

Lagom is an opinionated, open source framework for building reactive microservice systems in Java or Scala. Lagom stands on the shoulders of giants, Akka Toolkit and Play! Framework, two proven technologies that are battle-tested in production in many of the most demanding applications. Its designed after the principles of the domain-driven design (DDD), Event Sourcing and Command and Query Responsibility Segregation (CQRS) and strongly encourages usage of these patterns.

3.5.3 Akka

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. As we have seen, Akka serves as a foundation for a several other high-level frameworks (like for example Akka HTTP and Play Framework), however it is by itself is a great way to build microservices which could be decomposed into independent actors. The library management actor is an example of such a solution.

```
public class LibraryActor extends AbstractActor {
    private final LibraryService libraryService;

    public LibraryActor(final LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(GetBooks.class, e ->
                getSender().tell(libraryService.getBooks(), self()))
            .match(AddBook.class, e -> {
                final Book book = new Book(e.getIsbn(), e.getTitle());
                getSender().tell(libraryService.addBook(book), self());
            })
            .match(FindBook.class, e ->
                getSender().tell(libraryService.findBook(e.getIsbn()), self()))
            .matchAny(o -> log.info("received unknown message"))
            .build();
    }
}
```

Communication between Akka actors is very efficient and is not based on HTTP protocol (one of the preferred transports is Aeron, which we have briefly talked about in the previous part of the tutorial). The Akka Persistence module enables stateful actors to persist their internal state and recover it later. There are certain complexities you may run into while implementing microservice architecture using Akka but overall it is a solid and trusted choice.

3.5.4 ZeroMQ

ZeroMQ is not a typical messaging middleware. It is completely brokerless (originally the **zero** prefix in ZeroMQ was meant as "zero broker").

ZeroMQ (also known as OMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. - https://zguide.zeromq.org/page:all#ZeroMQ-in-a-Hundred-Words

It is widely used in the applications and services where achieving the low latency is a must. Those are the spots in microservice architecture where ZeroMQ might be of great help.

3.5.5 Apache Kafka

Apache Kafka, started from the idea of building the distributed log system, has expanded way beyond that into a distributed streaming platform. It is horizontally scalable, fault-tolerant, wicked fast, and is able to digest insanely massive volumes of messages (or events).

The hyper-popularity of the Apache Kafka (and for good reasons) had an effect that many other message brokers became forgotten and undeservedly are fading away. It is highly unlikely that Apache Kafka will not be able to keep up with the demand of your microservice architecture implementation, but more often than not a simpler alternative could be an answer.

3.5.6 RabbitMQ and Apache Qpid

RabbitMQ and Apache Qpid are classical examples of the message brokers which speak AMQP protocol. Not much to say beside that RabbitMQ is most known for the fact it is written in Erlang. Both are open source and good choices to serve as the messaging backbone between your microservices.

3.5.7 Apache ActiveMQ

Apache ActiveMQ is one of the oldest and most powerful open source messaging solutions out there. It supports a wide range of the protocols (including AMQP, STOMP, MQTT) while being fully compliant with Java Messaging Service (JMS 1.1) specification.

Interestingly, there are quite a few different message brokers hidden under Apache ActiveMQ umbrella. One of those is ActiveMQ Artemis with a goal to be a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.

Another one is ActiveMQ Apollo, a development effort to come up with a faster, more reliable and easier to maintain messaging broker. It was built from the foundations of the original Apache ActiveMQ with radically different threading and message dispatching architecture. Although very promising, it seems to be abandoned.

It is very likely that Apache ActiveMQ has every feature you need in order for your microservices to communicate the messages reliably. Also, the JMS support might be an important benefit in the enterprise world.

3.5.8 Apache RocketMQ

Apache RocketMQ is an open-source distributed messaging and streaming data platform (yet another contribution from Alibaba). It aims for extremely low latency, high availability and massive message capacity.

3.5.9 NATS

NATS is a simple, high performance open source messaging system for cloud-native applications, IoT messaging, and microservice architectures. It implements a highly scalable and elegant publish-subscribe message distribution model.

3.5.10 NSQ

NSQ is an open-source realtime distributed messaging platform, designed to operate at scale and handle billions of messages per day. It also follows a broker-less model and as such has no single point of failure, supports high-availability and horizontal scalability.

3.6 Get It All

There is a certain class of libraries which have a sole intention to seamlessly bridge many different communication channels together. Collectively, they are known as integration frameworks and are heavily inspired by terrific Enterprise Integration Patters book.

3.6.1 Apache Camel

Apache Camel is a powerful and mature open source integration framework. It is a surprisingly small library with a minimal set of dependencies and easily embeddable in any Java application. It abstracts away the kind of transports used behind a concise API layer which allows the interaction with more than 300 components provided out of the box.

3.6.2 Spring Integration

Spring Integration, another great member of the Spring projects portfolio, enables lightweight messaging and integration with external systems. It is primarily used within Spring-based applications and services, providing outstanding interoperability with all other Spring projects.

Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code. - https://spring.io/projects/spring-integration

If your microservices are built on top of Spring Framework, the Spring Integration is a logical choice to make (in case its need is justified and it fits into the overall architecture).

3.7 What about Cloud?

Along this part of the tutorial we have talked about open-source libraries and frameworks, which could be used either in on-premise deployments or in the cloud. They are generally agnostic to the vendor but many cloud providers have own biases towards one framework or another. More to that, besides offering own managed services, specifically in the messaging and data streaming space, the cloud vendors are heavily gravitating towards serverless computing, primarily in the shape of the function as a service.

3.8 But There Are a Lot More . . .

Fairly speaking, there are too many different libraries and frameworks to talk about. We have been discussing the most widely used ones but there are much, much more. Let us just mention some of them briefly. OSGi and its distributed counterpart, DOSGi, were known to be the only true way to build modular system and platforms in Java. Although it is not a simple one to deal with, it could be very well suited for implementing microservices. RxNetty is a pretty low-level reactive extension adaptor for Netty, quite helpful if you need an engine with very low overhead. Rest.li (from Linkedin) is a framework for building robust, scalable RESTful web services architectures using dynamic discovery and simple asynchronous APIs. Apache Pulsar is a multi-tenant, high performance, very low latency solution for server-to-server messaging which was originally developed by Yahoo.

3.9 Java / JVM Landscape - Conclusions

In this section of the tutorial we paged through tons of the different libraries and frameworks which are used today to build microservices in Java (and JVM in general). Every one of them has own niche but it does not make the decision process any easier. Importantly, the today's architecture may not reflect the tomorrow's reality: you have to pick the stack which could scale with your microservices for years to come.

3.10 What's next

In the next section of the tutorial we are going to talk about monoglot versus polyglot microservices and outline the reference application to serve as a playground for our future topics.

The complete set of sample projects is available for download.

Chapter 4

Monoglot or Polyglot?

4.1 Introduction

Along the previous parts of the tutorial we have talked quite a lot about the benefits of the microservice architecture. It is essentially a loosely coupled distributed system which provides a particularly important ability to pick the right tool for the job. It could mean not just a different framework, protocol or library but a completely different programming language.

In this part we are going to discuss the monoglot and polyglot microservices, the value each choice brings to the table and hopefully come up with the rational conclusions to help you make the decisions. Also, we will present the architecture of the reference application we are about to start developing. Its main purpose is to serve as the playground for numerous further topics we are going to look at.

4.2 There is Only One

Over the years many organizations have accumulated tremendous expertise around one particular programming language and its ecosystem, like for example Java, the subject of our tutorial. They have skilled developers, proven record of successful projects, in-depth knowledge of certain libraries and frameworks including the deep understanding of their quirks and peculiarities. Should all of that be thrown away in order to adopt the microservice architecture? Is this knowledge even relevant or useful?

Those are very hard questions to answer since many organizations get stuck with very old software stacks. Making such legacy systems fit the microservice architecture may sound quite impractical. However, if you have been lucky enough to bet on the frameworks and libraries we have discussed in the previous part of the tutorial, you are pretty much well positioned. You may certainly look around for better, modern options but starting with something you already know and familiar with is a safe bet. And frankly speaking, things do evolve over time, you may never feel the need to get off the Java train or your favorite set of frameworks and libraries.

There is nothing wrong with staying monoglot and building your microservices all the way on Java. But there is a trap where many adopters may fall into: very tight coupling between different services which eventually ends up with a birth of the distributed monolith. It stems from the decisions to take a shortcuts and share Java-specific artifacts (known as JARs) instead of relying on more generic, language-agnostic contracts and schemas.

Even if you prefer to stay monoglot, please think polyglot!

4.3 Polyglot on the JVM

Beside just Java, there are many other languages which natively run on JVM, like for example Scala, Kotlin, Clojure, Groovy, Ceylon to mention a few. Most of them have an excellent level of the interoperability with the code written in the plain old Java so it is really easy to take the polyglot microservices route staying entirely on JVM platform. Nonetheless, since everything is still packaged and distributed in JARs, the danger to build the distributed monolith remains very real.

While touching upon development of the polyglot applications on the JVM, it is unforgivable not to mention the cutting edge technology which came out of Oracle Labs and is bearing the name GraalVM.

GraalVM is a universal virtual machine for running applications written in JavaScript, Python 3, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++. GraalVM removes the isolation between programming languages and enables interoperability in a shared runtime. It can run either standalone or in the context of OpenJDK, Node.js, Oracle Database, or MySQL. - https://www.graalvm.org/

In the spirit of the true innovation, the GraalVM opens whole new horizons for the JVM platform. It is not ready for production use yet (still in the release candidate phase as of today) but it has all the potential to revolutionize the way we are building the applications on the JVM, especially the polyglot ones.

4.4 The Language Zoo

In the industry driven by hype and unrealistic promises, the new shiny things appear all the time and the developers are eager to use them right away in production. And indeed, the microservice architecture enables us to make such choices regarding the best language or/and framework to solve the business (or even technical) problems in a most efficient manner (but certainly does not mandate doing that).

In the same vein of promoting responsibility and ownership it looks logical to let the individual teams make the technological decisions. The truth is though in reality it is quite expensive to deal with the zoo of different languages and frameworks. That is why if you look around, you will see that most of the industry leaders bet on 2-3 primary programming languages, an important observation to keep in mind while evolving your microservices implementations.

4.5 Reference Application

To shift our discussions from the theory to practice, we are going to introduce the reference project we are about to start working on. Unsurprisingly, it is going to be built following the guiding principles of the microservice architecture. Since our tutorial is Java-oriented, most of our components will be written in this language but it is very important to see the big picture and realize that Java is not the only one. So let us roll up the sleeves and start building the polyglot microservices!

Our reference project is called **JCG Car Rentals** : a simplistic (but realistic!) application to provide car rental services to the various customers. There are several goals and objectives it pursues:

- Demonstrate the benefits of the microservice architecture
- Showcase how the various technology stacks (languages and frameworks) integrate with each other to compose a cohesive living platform
- Introduce the best practices, emerging techniques and tools for every aspect of the project lifecycle, ranging from development to operation in production
- And, hopefully, conclude that developing complex, heterogeneous distributed systems (which microservices are) is really difficult and challenging journey, full of tradeoffs

The **JCG Car Rentals** is quite far from the modern large-scale systems with hundreds of the microservices deployed in production. However even an application as simple as that posses many questions and problems. Let us take a look on the **JCG Car Rentals** architecture diagram below.

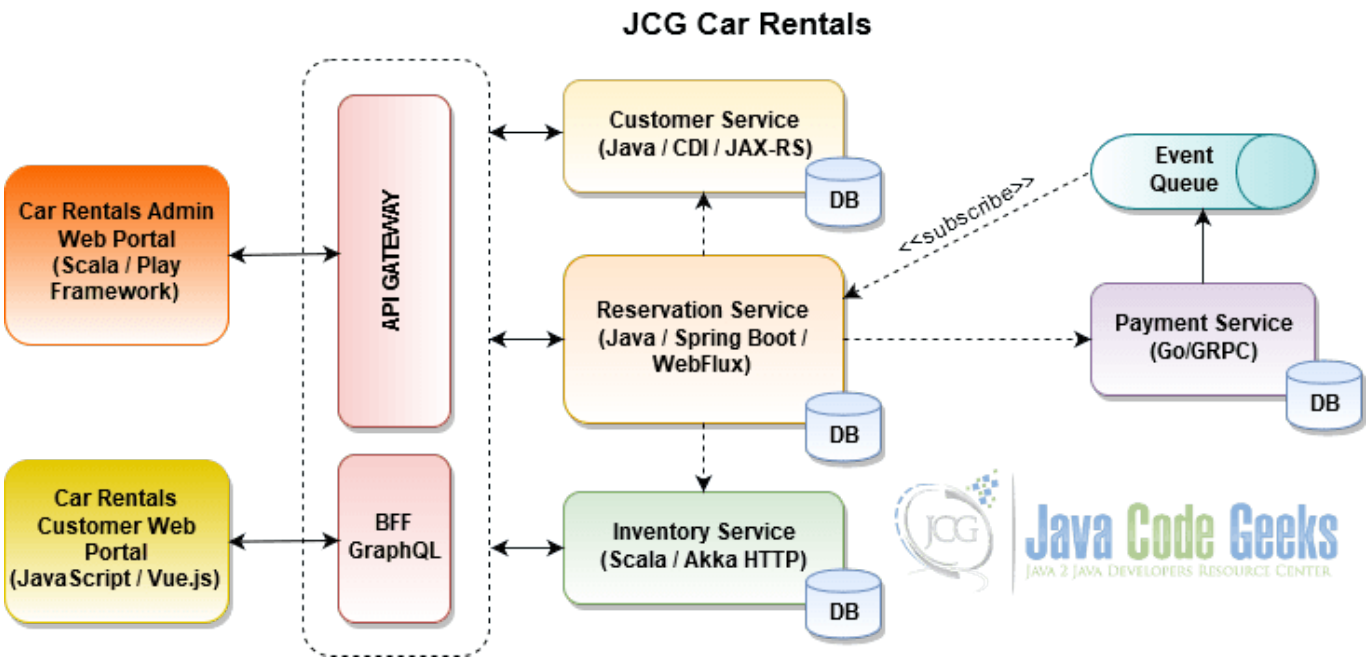


Figure 4.1: JCG Car Rentals Microservice Architecture

JCG Car Rentals Microservice Architecture

You may notice that there are quite a few contractions to some principles we have talked before. For example, each service uses own programming language or/and framework, the same happens to apply for frontends as well. Isn't it the language zoo we have been cautiously warned before? Yes, indeed, we have deliberately weakened some of the criteria in order to cover larger set of topics and interoperability scenarios. Please do not adopt this architecture as the blueprint for your applications but focus on the parts which work best for you.

With that, it would be useful to go over each box we have drawn and briefly explain the reasoning behind the choices we have made.

4.5.1 Customer Service

The responsibility of the customer service would be to manage the personal data of the **JCG Car Rentals** customers. This service has no upstream dependencies and we are going to implement it in Java as the RESTful web API, using one of the JAX-RS implementations (to be precise, we are going to rely on Apache CXF framework). **Why:** objectively, JAX-RS is a number one choice in the world of enterprise Java. The Apache CXF is not only JAX-RS compliant but provides a lot of additional must-have features for building successful microservice architectures.

4.5.2 Inventory Service

The inventory service is going to be an authoritative source of the cars and quantities available in the stock for rentals. It is also has no upstream dependencies and our implementation choice for it is going to be RESTful web service built on top of Scala and Akka HTTP. **Why:** Akka HTTP has proven to be an outstanding framework for implementing RESTful web services on JVM. Although it has Java DSL, the usage of Scala in the first place gives the most out of it.

4.5.3 Payment Service

The payment service would handle all the customer's charges for the services provided by **JCG Car Rentals** . We are picking a completely different technology stack here by building this service in Go and using gRPC protocol over HTTP/2 to communicate

with it. This is an example of the purely internal service which we may not want to expose externally. **Why:** gRPC has proven to be an efficient and an effective protocol for service to service communication. On the other side, Go is tremendously popular and has outstanding support of the gRPC.

4.5.4 Reservation Service

The reservation service is the core of the **JCG Car Rentals** microservice architecture. It is solely dedicated to manage the car reservations and depends on payment service, customer service and inventory service. Since it is most critical and important piece, we are going to implement it as RESTful web APIs using Spring Boot and Spring WebFlux. **Why:** Spring Boot, and more generally Spring Platform, absolutely dominates the Java ecosystem. Built on proven foundation of Spring Framework, it offers exceptional productivity, smart configuration capabilities and seamless integrations with most popular libraries and frameworks. The choice of Spring WebFlux (versus traditional Spring Web MVC) may not be so obvious at first but hopefully the next part of the tutorial is going to clarify that.

4.5.5 API Gateway

The API gateways secured their firm place in the microservice architecture since the early days. It turns out that exposing all (or most) of your services to be publicly accessible is in fact easy but not a very good idea. We are going to talk about API gateways in great details later on in the tutorial but just to highlight a few key issues they help to address:

- **discoverability** : the consumers do not need to know where the upstream dependencies live (for example hosts and ports)
- **partitioning** : the consumers do not need to know what are the exact services application constitutes of since the architecture may change over time
- **unified protocol** : the consumers do not need to worry about all kinds of different protocols each service speaks
- **client friendliness** : different clients may need to shape data differently

Our choices here are going to vary over time. We will start with Zuul, the Java-based gateway service from Netflix, and slowly take it from there. **Why:** Netflix run microservices at massive scale. Zuul (or more precisely Zuul 2) incorporates the invaluable experience of how the gateway service should operate.

4.5.6 BFF

The backend for frontend (or just BFF) came into the view not so long ago as an alternative to general-purpose API gateways. In short, each frontend is unique. At some point it became clear that the demands of the mobile frontends are quite different from let say a full-fledged desktop ones. To address this disparity, the BFF basic promise is to provide the outstanding support for one particular frontend, hence the name - backend for frontend.

There are quite a few frameworks to help us develop efficient BFFs but arguably the possibilities which are provided by GraphQL make the latter a particularly appealing foundation to build BFFs upon. And who can do the job better than Apollo platform. **Why:** Apollo platform is the bleeding edge of the GraphQL ecosystem. It is significantly more advanced and feature-rich than the JVM alternatives we have looked before.

4.5.7 Admin Web Portal

The admin web portal is the back-office for **JCG Car Rentals** platform. It will expose the ability to perform certain administrative functions, including the ones needed for customer support. Since this is an internal component, we are going to build it in Scala using Play Framework. **Why:** quite often the back-office applications fall in hands of backend developers. The Play Framework, backed by Scala, is truly a hyper productive choice for them.

4.5.8 Customer Web Portal

The customer web portal is the public entry point into **JCG Car Rentals** application. This is the place we expect people to search for deals and make the reservations, all that powered by handful of microservices. It is going to be developed using JavaScript and Vue.js. **Why:** this is a typical frontend stack for modern, single-page web applications (SPA). The preference to Vue.js (and not React or Angular for example) is given because of its simplicity and ease of use.

4.6 Conclusions

In this section we have talked about the opportunities the microservice architecture provides with respect to having a freedom to make technical choices. We have discussed the pros and cons of monoglot versus polyglot microservices and some pitfalls you should be aware of.

4.7 What's next

In the next section of the tutorial we are going to have a conversation about different programming paradigms which are often used to build microservices and modern distributed systems.

Chapter 5

Implementing microservices (synchronous, asynchronous, reactive, non-blocking)

5.1 Introduction

The previous parts of the tutorial were focused on more or less high-level topics regarding microservice architecture, like for example different frameworks, communication styles and interoperability in the polyglot world. Although it was quite useful, starting from this part we are slowly getting down to earth and direct our attention to the practical side of things, as developers say, closer to the code.

We are going to begin with a very important discussion regarding the variety of paradigms you may encounter while implementing the internals of your microservices. The deep understanding of the applicability, benefits and tradeoffs each one provides would help you out to make the right implementation choices in every specific context.

5.2 Synchronous

Synchronous programming is the most widely used paradigm these days because of its simplicity and ease to reason about. In the typical application it usually manifests as the sequence of function calls, where the each one is executed after another. To illustrate it in action, let us take a look on the implementation of the JAX-RS resource endpoint from the **Customer Service** .

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response register(@Context UriInfo uriInfo, @Valid CreateCustomer payload) {
    final CustomerInfo info = conversionService.convertTo(payload, CustomerInfo.class);
    final Customer customer = customerService.register(info);

    return Response
         .created(
             uriInfo
                 .getRequestUriBuilder()
                 .path(customer.getUuid())
                 .build())
         .build();
}
```

As you read this code, there are no surprises along the way (aside from possibility to get the exceptions). First, we convert customer information from the RESTful web API payload to the service object, after that we invoke the service to register the new customer and finally we return the response back to the caller. When the execution completes, the result of it is known and fully evaluated.

So what are the problems with such a paradigm? Surprisingly, it is the fact that the current invocation must wait for the previous one to finish. As an example, what if we have to send a confirmation email to the customer upon successful registration? Should we absolutely wait for the confirmation to be sent out or we could just return the response and make sure the confirmation is scheduled for delivery? Let us try to find the right answer in the next section.

5.3 Asynchronous

As we just discussed, the results of some operations may not necessarily be required in order for the execution flow to continue. Such operations could be executed asynchronously: concurrently, in parallel or even at some point in the future. The result of the operation may not be available immediately.

In order to understand how it works under the hood, we have to talk a little bit about concurrency and parallelism in Java (and on JVM in general), which is based on threads. Any execution in Java takes place in the context of the thread. As such, typical ways to achieve the execution of the particular operation asynchronously are to borrow the thread from the thread pool (or spawn a new thread manually) and perform the invocation in its context.

It looks pretty straightforward but it would be good to know when the asynchronous execution actually completes and, most importantly, what is the result of it. Since we are focusing primarily on Java, the key ingredient here is CompletableFuture which represents the result of an asynchronous computation. It has a number of the callback methods which allow the caller to be notified when the results are ready.

The veteran Java developers definitely remember the predecessor of the CompletableFuture, the Future interface. We are not going to talk about Future nor recommend using it since its capabilities are very limited.

Let us get back to sending a confirmation email upon successful customer registration. Since our **Customer Service** is using CDI 2.0, it would be natural to bind the notification to the customer registration event.

```
@Transactional
public Customer register(CustomerInfo info) {
    final CustomerEntity entity = conversionService.convertTo(info, CustomerEntity.class);
    repository.saveOrUpdate(entity);

    customerRegisteredEvent
        .fireAsync(new CustomerRegistered(entity.getUuid()))
        .whenComplete((r, ex) -> {
            if (ex != null) {
                LOG.error("Customer registration post-processing failed", ex);
            }
        });

    return conversionService.convertTo(entity, Customer.class);
}
```

The `CustomerRegistered` event is fired asynchronously and registration process continues the execution without awaiting for all observers to process it. The implementation is somewhat naive (since the transaction may fail or application may crash while processing the event) but it is good enough to illustrate the point: asynchronicity makes it harder to understand and reason about the execution flows. Not to mention the hidden costs of it: threads are precious and expensive resource.

The interesting property of the asynchronous invocation is the possibility to time it out (in case it is taking too long) or/and request the cancellation (in case the results are not needed anymore). However, as you may expect, not all operations could be interrupted, certain conditions apply.

5.4 Blocking

The synchronous programming paradigm in the context of executing I/O operations is often referred as blocking. Fairly speaking, synchronous and blocking are often used interchangeably but with respect to our discussion, only I/O operations are assumed to fall into this category.

Indeed, although from the execution flow perspective there is no much difference (each operation has to wait for previous one to complete), the mechanics of doing I/O is quite contrasting from, let say, pure computational work. What would be the typical examples of such blocking operation in mostly any Java application? Just think of relational databases and JDBC drivers.

```
@Inject @CustomersDb
private EntityManager em;

@Override
public Optional findById(String uuid) {
    final CriteriaBuilder cb = em.getCriteriaBuilder();

    final CriteriaQuery query = cb.createQuery(CustomerEntity.class);
    final Root root = query.from(CustomerEntity.class);
    query.where(cb.equal(root.get(CustomerEntity_.uuid), uuid));

    try {
        final CustomerEntity customer = em.createQuery(query).getSingleResult();
        return Optional.of(customer);
    } catch (final NoResultException ex) {
        return Optional.empty();
    }
}
```

Our **Customer Service** implementation does not use JDBC APIs directly, relying on high-level JPA specification (JSR-317, JSR-338) and its providers instead. Nonetheless, it easy to spot where the call to database is happening:

```
final CustomerEntity customer = em.createQuery(query).getSingleResult();
```

The execution flow is going to hit the wall here. Depending on the capabilities of the JDBC drivers, you may have some control over the transaction or query, like for example, cancelling it or setting the timeout. But by and large, it is a blocking call: the execution resumes only when the query is completed and results are fetched.

5.5 Non-Blocking

During the I/O cycles a lot of time is spent in waiting, typically for the disk operations or network transfers. Consequently, as we have seen in the previous section, the execution flow has to pay the price of that by being blocked from the further progress.

Since we already have gotten the brief introduction into the concept of the asynchronous programming, the obvious question would be why not to invoke such I/O operations asynchronously? All in all it makes perfect sense, however at least with respect to JVM (and Java) that would just offload the problem from one execution thread to another one (for instance, borrowed from the dedicated I/O pool). It is still looking quite inefficient from resource utilization perspective. Even more, the scalability is going to suffer as well since the application cannot just spawn or borrow new threads indefinitely.

Luckily, there are a number of techniques to attack this problem, collectively known as non-blocking I/O (or asynchronous I/O). One of the most widely used implementation of non-blocking, asynchronous I/O is based on Reactor pattern. The picture below depicts a simplified view of the it.

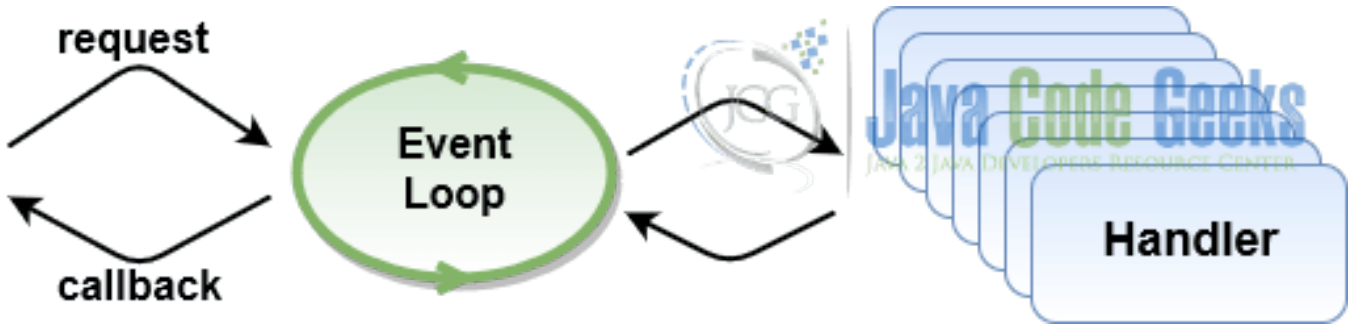


Figure 5.1: Reactor Pattern

Reactor Pattern

In the core of the Reactor pattern is a single-threaded event loop. Upon receiving the request for I/O operation, it is delegated to the pool of handlers (or to more efficient implementation specific to the operating system the application is running on). The results of I/O operation may be injected back (as the events) to the event loop and eventually, upon the completion, the outcome is dispatched to the application.

On JVM, the Netty framework is the de-facto choice for implementing an asynchronous, event-driven network servers and clients. Let us take a look on how the **Reservation Service** may call the **Customer Service** to lookup the customer by its identifier in a truly non-blocking fashion using AsyncHttpClient library, built on top of Netty (the error handling is omitted to keep the snippet short).

```
final AsyncHttpClient client = new DefaultAsyncHttpClient();

final CompletableFuture customer = client
    .prepareGet("https://localhost:8080/api/customers/" + uuid)
    .setRequestTimeout(500)
    .setReadTimeout(100)
    .execute()
    .toCompletableFuture()
    .thenApply(response -> fromJson(response.getResponseBodyAsStream()));

// ...

client.close();
```

Interestingly enough, for the caller the non-blocking invocation is no different from the asynchronous one, but the internals of how it is done matter a lot.

5.6 Reactive

The reactive programming lifts the asynchronous and non-blocking paradigms to a completely new level. There are quite a few definitions what the reactive programming really is, but the most compelling one is this.

Reactive programming is programming with asynchronous data streams. - https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

This rather short definition is worth a book. To keep the discussion reasonably short, we are going to focus on a practical side of things, the reactive streams.

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. - https://www.reactive-streams.org/

What is so special about it? The reactive streams unify the way we are dealing with data in our applications by emphasizing on a few key points:

- (mostly) everything is a stream
- the streams are asynchronous by nature
- the streams supports non-blocking back pressure to control the flow of data

The code is worth thousand words. Since Spring WebFlux comes with reactive, non-blocking HTTP client, let us take a look on how the **Reservation Service** may call the **Customer Service** to lookup the customer by its identifier in the reactive way (for simplicity, the error handling is omitted).

```
final HttpClient httpClient = HttpClient
    .create()
    .tcpConfiguration(client ->
        client
            .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 500)
            .doOnConnected(conn ->
```

```
                conn
                    .addHandlerLast(new ReadTimeoutHandler(100))
                    .addHandlerLast(new WriteTimeoutHandler(100))
    ));

final WebClient client = WebClient
    .builder()
    .clientConnector(new ReactorClientHttpConnector(httpClient))
    .baseUrl("https://localhost:8080/api/customers")
    .build();

final Mono customer = client
    .get()
    .uri("/{uuid}", uuid)
    .retrieve()
    .bodyToMono(Customer.class);
```

Conceptually, it looks pretty much like the AsyncHttpClient example, just a bit more ceremony. However, the usage of reactive types (like `Mono<Customer>`) unleashes the full power of reactive streams.

The discussions around reactive programming could not be complete without mentioning the The Reactive Manifesto and its tremendous impact on the design and architecture of the modern applications.

. . . We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable . This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback. - https://www.reactivemanifesto.org/

The foundational principles and promises of the reactive systems fit exceptionally well the microservice architecture, spawning a new class of microservices, the reactive microservices.

5.7 The Future Is Bright

The pace of innovation in Java has increased dramatically over the last couple of years. With the fresh release cadence, the new features become available to the Java developers every 6 months. However, there are many ongoing projects which may have a dramatic impact on the future of JVM and Java in particular.

One of those is Project Loom. The goal of this project is to explore the implementation of lightweight user-mode threads (fibers), delimited continuations (of some form), and related features. As of now, fibers are not supported by JVM natively, although there are some libraries like Quasar from Parallel Universe which are trying to fill this gap.

Also, the introduction of fibers, as the alternative to threads, would make it possible to have efficient support of the coroutines on JVM.

5.8 Implementing microservices - Conclusions

In this part of the tutorial we have talked about different paradigms you may consider while implementing your microservices. We went from the traditional way of structuring the execution flows as a sequence of consecutive blocking steps up to the reactive streams.

The asynchronous and reactive ways of thinking and writing the code may look difficult at first. Fear no more, but it does not mean all your microservices must be reactive. Every choice is a tradeoff and it is up to you to make the right ones in the context of your microservice architecture and organization.

5.9 What's next

In the next part of the tutorial we are going to discuss the fallacies of the distributed computing and how to mitigate their impact in the microservice architectures.

Chapter 6

Microservices and fallacies of the distributed computing

6.1 Introduction

The journey of implementing microservice architecture inherently implies building the complex distributed system. And fairly speaking, most of the real world software systems are far from being simple, but the distributed nature of the microservices amplifies the complexity a lot.

In this part of the tutorial we are going to talk about some of the common traps that many developers could fall into, known as fallacies of distributed computing. All these false assumptions should not mislead us and we are going to spend a fair amount of time talking about different patterns and techniques for building resilient microservices.

Any complex system can (and will) fail in surprising ways . . . - https://queue.acm.org/detail.cfm?id=2353017

6.2 Local != Distributed

How many times you have been caught by surprise discovering that the invocation of the seemingly innocent method or function causes a storm of remote calls? Indeed, these days most of the frameworks and libraries out there are hiding the genuinely important details behind multiple levels of convenient abstractions, trying to make us believe that there is no difference between local (in-process) and remote calls. But the truth is, network is not reliable and network latency is not equal to zero.

Although most of our topics are going to be centered on traditional request / response communication style, asynchronous message-driven microservices are not hassle-free either. You still have to reach the remote brokers and be ready to deal with idempotency and message de-duplication.

6.3 SLA

We are going to start off with service-level agreement, or simply SLA. It is very often overlooked subject but each service in your microservices ensemble should better have one defined. It is difficult and thoughtful process, which is unique to the nature of the service in question and should take into account a lot of different constraints.

Why it is so essential? First of all, it gives the development team a certain level of freedom in picking the right technology stack. And secondly, it hints the consumers of the service what to expect in terms of response times and availability (so the consumers could derive own SLAs).

In the next sections we are going to discuss a number of techniques the consumers (which are often the other services) may use to protect themselves from the instability or outages of the services they depend upon.

6.4 Health Checks

Is there a quick way to check if the service is up and running, even before embarking on a potentially complex business transaction? The health checks are the standard practice for the service to report its readiness to take the work.

All the services of the **JCG Car Rentals** platform expose the health check endpoints by default. Below, the **Customer Service** is picked to showcase the health check in action:

```
$ curl https://localhost:18800/health
{
  "checks": [
    {
      "data": {},
      "name": "db",
      "state": "UP"
    }
  ],
  "outcome": "UP"
}
```

As we are going to see later on, the health checks are actively used by infrastructure and orchestration layers to probe the service, alert or/and apply the compensating actions.

6.5 Timeouts

When one party calls another one over the wire, configuring the proper timeouts (connection, read, write, request, . . .) is probably the simplest but the most effective strategy to use. We have already seen it in the previous part of the tutorial, here is just a short remainder.

```
final CompletableFuture customer = client
    .prepareGet("https://localhost:8080/api/customers/" + uuid)
    .setRequestTimeout(500)
    .setReadTimeout(100)
    .execute()
    .toCompletableFuture();
```

When the other side is irresponsive, or communication channels are unreliable, waiting indefinitely in the hope that response may finally come in is not a best option. Now, the question is what the timeouts should be set to? There is no single magic number which works for everyone but the service SLAs we have discussed earlier are the key source of information to answer this question.

Great, so let us assume the right values are in place, but what should the consumer do if the call to the service times out? Unless the consumer does not care about the response anymore, the typical strategy in this case is to retry the call. Let us talk about that for a moment.

6.6 Retries

From the consumer perspective, retrying the request to the service in case of intermittent failures is the easiest thing to do. For these purposes, the libraries like Spring Retry, failsafe or resilience4j are of great help, offering a range of retry and back-off policies. For example, the snippet below demonstrates the Spring Retry approach.

```
final SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy(5);
final ExponentialBackOffPolicy backOffPolicy = new ExponentialBackOffPolicy();
backOffPolicy.setInitialInterval(1000);
backOffPolicy.setMaxInterval(20000);

final RetryTemplate template = new RetryTemplate();
```

```
template.setRetryPolicy(retryPolicy);
template.setBackOffPolicy(backOffPolicy);

final Result result = template.execute(new RetryCallback<Result, IOException>() {
    public Result doWithRetry(RetryContext context) throws IOException {
        // Any logic which needs retry here
        return ...;
    }
});
```

Besides these general-purpose one, most of the libraries and frameworks have own built-in idiomatic mechanism to perform retries. The example below comes from the Spring Reactive WebClient we have touched upon in the previous part of the tutorial.

```
final WebClient client = WebClient
    .builder()
    .clientConnector(new ReactorClientHttpConnector(httpClient))
    .baseUrl("https://localhost:8080/api/customers")
    .build();

final Mono customer = client
    .get()
    .uri("/{uuid}", uuid)
    .retrieve()
    .bodyToMono(Customer.class)
    .retryBackoff(5, Duration.ofSeconds(1));
```

The importance of the back-off policy rather than fixed delays should not be neglected. The retry storms, better known as thundering herd problem, are often causing the outages since all the consumers may decide to retry the request at the same time.

And last but not least, one serious consideration when using any retry strategy is idempotency: the preventing measures should be taken both from consumer side and service side to make sure there are no unexpected side-effects.

6.7 Bulk-Heading

The concept of bulkhead is borrowed from the ship building industry and found its direct analogy in software development practices.

Bulkheads are used in ships to create separate watertight compartments which serve to limit the effect of a failure - ideally preventing the ship from sinking. - https://skife.org/architecture/fault-tolerance/2009/12/31/bulkheads.html

Although we are not building ships but software, the main idea stays the same: minimize the impact of the failures in the applications, ideally preventing them from crashes or becoming irresponsive. Let us discuss a few scenarios where bulkheading manifests itself, especially in microservices.

The **Reservation Service**, part of the **JCG Car Rentals** platform, might be asked to retrieve all reservations for a particular customer. To do that, it first consults the **Customer Service** to make sure the customer exists, and in case of successful response, fetches the available reservations from the underlying data store, limiting the results to first 20 records.

```
@Autowired private WebClient customers;
@Autowired private ReservationsByCustomersRepository repository;
@Autowired private ConversionService conversion;

@GetMapping("/customers/{uuid}/reservations")
public Flux findByCustomerId(@PathVariable UUID uuid) {
    return customers
        .get()
        .uri("/{uuid}", uuid)
        .retrieve()
        .bodyToMono(Customer.class)
        .flatMapMany(c -> repository
```

```
            .findByCustomerId(uuid)
            .take(20)
            .map(entity -> conversion.convert(entity, Reservation.class)));
}
```

The conciseness of the Spring Reactive stack is amazing, isn't it? So what could be the problem with this code snippet? It all depends on `repository`, really. If the call is blocking, the catastrophe is about to happen since the even loop is going to be blocked as well (remember, the Reactor pattern). Instead, the blocking call should be isolated and offloaded to a dedicated pool (using `subscribeOn`).

```
    return customers
        .get()
        .uri("/{uuid}", uuid)
        .retrieve()
        .bodyToMono(Customer.class)
        .flatMapMany(c -> repository
            .findByCustomerId(uuid)
            .take(20)
            .map(entity -> conversion.convert(entity, Reservation.class))
            .subscribeOn(Schedulers.elastic()));
```

Arguably, this is one example of the bulkheading, to use dedicated thread pools, queues, or processes to minimize the impact on the critical parts of application. Deploying and balancing over multiple instances of the service, isolating the tenants in the multitenant applications, prioritizing the request processing, harmonizing the resource utilization between background and foreground workers, this is just a short list of interesting challenges you may run into.

6.8 Circuit Breakers

Awesome, so we have learned about the retry strategies and bulkheading, we know how to apply these principles to isolate the failures and progressively get the job done. However, our goal is not really that, we have to stay responsive and fulfill the SLA promises. And even if you do not have ones, responding within reasonable time frame is a must. The circuit breaker pattern, popularized by Michael Nygard in the terrific and highly recommended for reading Release It! book, is what we would really need.

The circuit breaker implementation could get quite sophisticated but we are going to focus on its two core features: ability to keep track of the status of the remote invocation and use the fallback in case of failures or timeouts. There are quite a few excellent libraries which provide the circuit breaker implementations. Beside failsafe and resilience4j we have mentioned before, there are also Hystrix, Apache Polygene and Akka. The Hystrix is probably the best known and battle-tested circuit breaker implementation as of today and is the one we are going to use as well.

Getting back to our **Reservation Service**, let us take a look on how Hystrix could be integrated into the reactive flow.

```
public Flux findByCustomerId(@PathVariable UUID uuid) {
    final Publisher customer = customers
        .get()
        .uri("/{uuid}", uuid)
        .retrieve()
        .bodyToMono(Customer.class);

    final Publisher fallback = HystrixCommands
        .from(customer)
        .eager()
        .commandName("get-customer")
        .fallback(Mono.empty())
        .build();

    return Mono
        .from(fallback)
        .flatMapMany(c -> repository
```

```
            .findByCustomerId(uuid)
            .take(20)
            .map(entity -> conversion.convert(entity, Reservation.class)));
}
```

We have not tuned any Hystrix configuration in this example but if you are curious to learn more about the internals, please check out this article.

The usage of the circuit breakers not only helps the consumer to make the smart decisions based on the operational statistics, it also potentially may help the service provider to recover from the intermittent load conditions faster.

6.9 Budgets

The circuit breakers along with sensitive timeouts and retry strategies are helping your service to deal with failures but they also eat your service SLA budget. It is absolutely possible that when the service has finally gotten all the data it needs to assemble the final response, the other side is not interested anymore and has dropped the connection long ago.

This is difficult problem to solve although there is one quite straightforward technique to apply: consider calculating the approximate time budget the service has while progressively fulfilling the request. Going over the budget should be rather the exception than the rule, but when it happens, you are well prepared by cutting off the throwaway work.

6.10 Persistent Queues

This is somewhat obvious but if your microservice architecture is built using asynchronous message passing, the queues which store the messages or events must be persistent (and very desirably, replicated). Most of the message brokers we have discussed previously support durable persistent storage out of the box but there are special cases when you may be trapped.

Let us get back to the example of sending the confirmation email upon successful customer registration, which we implemented using asynchronous CDI 2.0 events.

```
customerRegisteredEvent
    .fireAsync(new CustomerRegistered(entity.getUuid()))
    .whenComplete((r, ex) -> {
        if (ex != null) {
            LOG.error("Customer registration post-processing failed", ex);
        }
    });
```

The problem with this approach is that the event queuing is happening all in memory. If the process crashes before the event gets delivered to the listeners, it is going to be lost forever. Perhaps in case of confirmation email it is not a big deal, but issue is still there.

For the cases when the lost of such events or messages is not desired, one of the options is to use persistent in-process queue, like for example Chronicle Queue. But in the long run using the dedicated message broker or data store might be a better choice overall.

6.11 Rate Limiters

One of the unpleasant but unfortunately very realistic situations you should prepare your services for is to deal with abusive clients. We would exclude the purposely malicious and DDoS attacks, since those require the sophisticated mitigation solutions. But bugs do happen and even internal consumers may go wild and try to put your service on its knees. Rate limiting is an efficient technique to control the rate of requests from the particular source and shed the load in case when the limits are violated.

Although it is possible to bake the rate limiting into each service (using, for example, Redis to coordinate all service instances), it makes more sense to offload such responsibility to API gateways and orchestration layers. We are going to get back to this topic in more details later in the tutorial.

6.12 Sagas

Let us forget about individual microservices for a moment and look at the big picture. The typical business flow is usually a multistep process and relies on several microservices to successfully do their part. The reservation flow which the **JCG Car Rentals** implements is a good example of that. There are at least three services involved in it:

- the **Inventory Service** has to confirm the vehicle availability
- the **Reservation Service** has to check that vehicle is not already booked and make the reservation
- the **Payment Service** has to process the charges (or refunds)

The flow is a bit simplified but the point is, every step may fail for variety of reasons. The traditional approach the monoliths take is to wrap everything in the huge all-or-nothing database transaction but it is not going to work here. So what are the options?

One of them is to use distributed transaction and two-phase commit protocol, with all the complexity and scalability issues it brings on the table. Another approach, more aligned with the microservice architecture, is to use sagas.

A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions. - https://microservices.io/patterns/data/saga.html

It is very likely that you may need to rely on sagas while implementing business flows spanning multiple microservices. The Axon and Eventuate Tram Saga are two examples of the frameworks which support sagas but the chances to end up in DIY situation are high.

6.13 Chaos

At this point it may look like building microservices is the fight against chaos: anything anywhere could break and you have to deal with that somehow. In some sense it is true and this is probably why the discipline of chaos engineering was born.

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production. - https://principlesofchaos.org/

The goal of chaos engineering is not to crash the system but make sure that mitigation strategies work and reveal the problems if any. In the part of the tutorial dedicated to testing we are going to spend some time discussing the faults injection but if you are curious to learn more right away, please check out this great introductory article.

6.14 Conclusions

In this part of the tutorial we have talked about the importance of thinking about and mitigating failures while implementing microservice architecture. Network is unreliable and staying resilient and responsive should be among the core guiding principles to follow by each microservice in your fleet.

We have covered the set of generally applicable techniques and practices but this is just a tip of the iceberg. The advanced approaches like, for example, Java GC pause detection or load balancing, left out of scope in our discussion however the upcoming parts of the tutorial will dive into some of those.

To run a bit ahead, it is worth to mention that a lot of concerns which used to be the responsibility of the applications are moving up to the infrastructure or orchestration layers. Still, it is valuable to known such problems exist and how to deal with them.

6.15 What's next

In the next part of the tutorial we are going to talk about security and secret management, exceptionally important topics in the age when everything is deployed into the public cloud.

Chapter 7

Managing Security and Secrets

7.1 Introduction

Security is an exceptionally important element of the modern software systems. It is a huge topic by itself which includes a lot of different aspects and should never come as an afterthought. It is hard to get everything right, particularly in the context of the distributed microservice architecture, nonetheless along this part of the tutorial we are going to discuss the most critical areas and suggest on how you may approach them.

If you have the security expert in your team or organization, this is a great start of the journey. If not, you should better hire one, since the developer's expertise may vary greatly here. No matter what please restrain from rolling out your own security schemes.

And at last, before we get started, please make the Secure Coding Guidelines for Java SE a mandatory reading for any Java developer in your team. Additionally, Java SE platform official documentation includes a good summary of all the specifications, guides and APIs related to Java security.

7.2 Down to the Wire

In any distributed system a lot of data travels between different components. Projecting that to the microservice architecture, each service either directly communicates with other services or passes the messages or/and events around.

Using the secure transport is probably the most fundamental way to protect data in the transit from being intercepted or tampered. For web-based communication, it typically means the usage of HTTPS (or better to say, HTTP over SSL / TLS) to shelter privacy and preserve data integrity. Interestingly, although for HTTP/2 the presence of the secure transport is still optional, it is mostly exclusively used with SSL / TLS.

Beside just HTTPS, there are many other protocols which rely on TLS for securing the communication, like for example DTLS, SFTP[SFTP], WSS and SMTPS, to name a few. It is also worth mentioning Message Security Layer, an extensible and flexible secure messaging framework which was open-sourced by Netflix.

7.3 Security in Browser

On the web browser side, a lot of the efforts are invested in making the web sites more secure by supporting mechanisms like HTTP Strict Transport Security (HSTS), HTTP Public Key Pinning (HPKP), Content Security Policy (CSP), secure cookies and same-site cookies (the **JCG Car Rentals** customer and administration web portals would certainly rely on some of those).

On the scripting side we have Web Cryptography API specification which describes a JavaScript API for performing basic cryptographic operations in the web applications (hashing, signature generation and verification, and encryption and decryption).

7.4 Authentication and Authorization

Identifying all kinds of possible participants (users, services, partners, and external systems) and what they are allowed to do in the system is yet another aspect of securing your microservices. It is closely related to two distinct processes, authentication and authorization. Authentication is the process to ensure that the entity is who or what it claims to be. Whereas the authorization is the process of specifying and imposing the access rights, permissions and privileges this particular entity has.

For the majority of the applications out there, the single-factor authentication (typically, based on providing the password) is still the de-facto choice, despite its numerous weaknesses. On the bright side, the different methods of multi-factor authentication slowly but surely are getting more widespread adoption.

With respect to the authorization, there are basically two prevailing models: role-based access control (also called RBAC) and access control lists (ACLs). As we are going to see later on, most of the security frameworks support both these models so it is a matter of making the deliberate decision which one fits the best to the context of your microservice architecture.

If we shift the authentication and authorization towards the web applications and services, like with our **JCG Car Rentals** platform, we are most likely to end up with two industry standards, OAuth 2.0 and OpenID Connect 1.0.

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. - https://tools.ietf.org/html/rfc6749

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner - https://openid.net/connect/

Those two standards are closely related to the JSON Web Token (JWT) specification, which is often used to serve as OAuth 2.0 bearer token.

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. - https://tools.ietf.org/html/rfc7519

Amid numerous data breaches and leaks of the personal information (hello, Mariott), security becomes as important as never before. It is absolutely essential to familiarize, follow and stay up to date with the best security practices and recommendations. The two excellent guides on the subject, OAuth 2.0 Security Best Current Practices and JSON Web Token Best Current Practices, certainly fall into must-read category.

7.5 Identity Providers

Once the authentication and authorization decisions are finalized, the next obvious question is should you implement everything yourself or may be look around for existing solutions? To admit the truth, are your requirements so unique that you have to waste engineering time and build your own implementations? Is it in the core of your business? It is surprising how many organizations fall into DIY mode and reinvent the wheel over and over again.

For **JCG Car Rentals** platform we are going to use Keycloak, the established open-source identity and access management solution which fully supports OpenID Connect.

Keycloak is an open source Identity and Access Management solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code. - https://www.keycloak.org/about.html

The Keycloak comes with quite comprehensive configuration and installation guides but it is worth to mention that we are going to use it to manage the identity of the **JCG Car Rentals** customers and support staff.

Beside the Keycloak, another notable open-source alternative to consider is WSO2 Identity Server, which could have worked for **JCG Car Rentals** as well.

WSO2 Identity Server is an extensible, open source IAM solution to federate and manage identities across both enterprise and cloud environments including APIs, mobile, and Internet of Things devices, regardless of the standards on which they are based. - https://wso2.com/identity-and-access-management/features/

In you are looking to completely outsource the identity management of your microservices, there is a large number of certified OpenID providers and commercial offerings to choose from.

7.6 Securing Applications

The security on the applications and services side is probably where the most efforts will be focused on. In the Java ecosystem there are basically two foundational frameworks for managing authentication and authorization mechanisms, Spring Security and Apache Shiro.

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. - https://spring.io/projects/spring-security

Indeed, since our **Reservation Service** is built on top of the Spring Boot and Spring WebFlux, the choice in favor of Spring Security is obvious. It takes literally a few lines of configuration to have full-fledged OpenID Connect integrated into the service. The code snippet below illustrates just one of the possible ways to do that.

```
@EnableReactiveMethodSecurity
@EnableWebFluxSecurity
@Configuration
public class WebSecurityConfiguration {
    @Value("${spring.security.oauth2.resourceserver.jwt.issuer-uri}")
    private String issuerUri;

    @Bean
    SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http){
        http
            .cors()
            .configurationSource(corsConfigurationSource())
            .and()
            .authorizeExchange()
            .pathMatchers(HttpMethod.OPTIONS).permitAll()
            .anyExchange()
            .authenticated()
            .and()
            .oauth2ResourceServer()
            .jwt();
        return http.build();
    }

    @Bean
    ReactiveJwtDecoder jwtDecoder() {
        return ReactiveJwtDecoders.fromOidcIssuerLocation(issuerUri);
    }
}
```

The `spring.security.oauth2.resourceserver.jwt.issuer-uri` points out to the **JCG Car Rentals** instance of the Keycloak realm. In case when Spring Security is out of scope, Apache Shiro is certainly worth considering.

Apache Shiro is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management... - https://shiro.apache.org/

A little bit less known is the pac4j security engine also focused on protecting the web applications and web services. The **Admin Web Portal** of the **JCG Car Rentals** platform relies on pac4j to integrate with Keycloak using OpenID Connect.

Since OpenID Connect uses JWT (and related specifications) you may need to onboard one of the libraries which implements the specs in question. The most widely used ones include Nimbus JOSE+JWT, jose4j, Java JWT and Apache CXF.

If we expand our coverage beyond just Java to a broader JVM landscape, there are a few other libraries you may run into. One of them is Silhouette, used primarily by Play Framework web applications.

7.7 Keeping Secrets Safe

Most (if not all) of the services in typical microservice architecture would depend on some sort of configuration in order to function as intended. This configuration is usually specific to an environment the service is deployed into and, if you follow the 12 Factor App methodology, you already know that such configuration has to be externalized and separated from the code.

Still, many organizations store the configuration close by to the service, in the configuration files or even hardcoded in the code. What makes the matter worse is that often such configuration includes sensitive information, like for example credentials to access data stores, service accounts or encryption keys. Such pieces of data are classified as secrets and should never leak in plain. The projects like git-secrets would help you to prevent committing secrets and credentials into source control repositories.

Luckily, there are several options to look at. The simplest one is to use the encryption and store only the encrypted values. For Spring Boot applications, you may use Spring Boot CLI along with Spring Cloud CLI to encrypt and decrypt property values.

```
$ ./bin/spring encrypt --key <secret> <value>
d66bcc67c220c64b0b35559df9881a6dad8643ccdec9010806991d4250ecde60
```

Such encrypted values should be prefixed in the configuration with the special `{cipher}` prefix, like in this YAML fragment:

```
spring:
  data:
    cassandra:
      password:"{cipher}d66bcc67c220c64b0b35559df9881a6dad8643ccdec9010806991d4250ecde60"
```

To configure the symmetric key we just need to set `encrypt.key` property or better, use `ENCRYPT_KEY` environment variable. The Jasypt's Spring Boot integration works in similar fashion by providing the encryption support for property sources in Spring Boot applications.

Using encrypted properties works but is quite naive, the arguably better approach would be to utilize a dedicated secret management infrastructure, like for example Vault from HashiCorp.

Vault secures, stores, and tightly controls access to tokens, passwords, certificates, API keys, and other secrets in modern computing. - https://learn.hashicorp.com/vault/#getting-started

Vault makes secrets management secure and really easy. The services built on top of Spring Boot, like **Reservation Service** from **JCG Car Rentals** platform, may benefit from first-class Spring Cloud Vault integration.

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-vault-config</artifactId>
</dependency>
```

One of the very power features which Spring Cloud Vault provides is the ability to plug Vault key/value store as the application property source. The **Reservation Service** leverages that using `bootstrap.yml` configuration file.

```
spring:
  application:
    name: reservation-service
  cloud:
    vault:
      host: localhost
      port: 8200
      scheme: https
      authentication: TOKEN
      token: <token>
      kv:
        enabled: true
```

Although Vault is probably the most known one, there are a number of decent alternatives which fit nicely into microservice architecture. One of the pioneers is Keywhiz, a system for managing and distributing secrets, which was developed and open-sourced by Square. Another one is Knox, the service for storing and rotating secrets, keys, and passwords used by other services, came out of Pinterest.

7.8 Taking Care of Your Data

Data is probably the most important asset you may ever have and as such, it should be managed with a great care. Some pieces of data like credit card numbers, social security numbers, bank accounts or/and personally identifiable information (PII) are very

sensitive and should be operated securely. These days that typically assumes it has to be encrypted (which prevents data visibility in the case of unauthorized access or theft).

In the Keeping Secrets Safe section we have discussed the ways to manage encryption keys however you still have to decide if the data should be encrypted at application level or at storage level. Although both approaches have own pros and cons, not all data stores support encryption at rest, so you may not have a choice here. In this case, you may find invaluable Cryptographic Storage and Password Storage cheat sheets which OWASP Foundation publishes and keeps up to date with respect to the latest security practices.

7.9 Scan Your Dependencies

It is very likely that each microservice in your microservices ensemble depends on multiple frameworks or libraries, which in turn have own set of dependencies. Keeping your dependencies up to date is yet another aspect of security measures since the vulnerabilities may be discovered in any of those.

The OWASP dependency-check is an open source solution which can be used to scan the Java applications in order to identify the use of known vulnerable components. It has dedicated plugins for Apache Maven, Gradle, SBT and integrated into build definitions of each **JCG Car Rentals** service.

The OWASP dependency-check is an open source solution which can be used to scan the Java applications in order to identify the use of known vulnerable components. It has dedicated plugins for Apache Maven, Gradle, SBT and is integrated into build definitions of each **JCG Car Rentals** service. The following fragment in the **Reservation Service** 's `pom.xml` illustrates the usage scenario.

```
<plugin>
        <groupId>org.owasp</groupId>
        <artifactId>dependency-check-maven</artifactId>
        <version>4.0.0</version>
        <executions>
                <execution>
                        <goals>
                                <goal>check</goal>
                        </goals>
                </execution>
        </executions>
</plugin>
```

To give an idea what the OWASP dependency-check report prints out, let us take a look at some of the dependencies identified with known vulnerabilities in the **Reservation Service** .

Dependency	CPE	Coordinates	Highest Severity	CVE Count	CPE Confidence	Evidence Count
jul-to-slf4j-1.7.25.jar	cpe:/a:slf4j:slf4j:1.7.25	org.slf4j:jul-to-slf4j:1.7.25	High	1	Highest	28
jcl-over-slf4j-1.7.25.jar	cpe:/a:slf4j:slf4j:1.7.25	org.slf4j:jcl-over-slf4j:1.7.25	High	1	Highest	29
slf4j-api-1.7.25.jar	cpe:/a:slf4j:slf4j:1.7.25	org.slf4j:slf4j-api:1.7.25	High	1	Highest	29

Figure 7.1: Table Reservation Service

Since **JCG Car Rentals** has a couple of components which use Node.js, it is important to audit their package dependencies for security vulnerabilities as well. The recently introduced `npm audit` command scans each project for vulnerabilities and automatically installs any compatible updates to vulnerable dependencies. Below is an example of the audit command execution.

```
$ npm audit
```

```
                        !!! npm audit security report !!!
found 0 vulnerabilities
 in 20104 scanned packages
```

7.10 Packaging

Ever since Docker brought the containers to the masses, they become the de-facto packaging and distribution model for all kind of applications, including the Java ones. But with the great power comes great responsibility: the vulnerabilities inside the container could make your applications severely exposed. Fortunately, we have Clair, vulnerability static analysis for containers.

Clair is an open source project for the static analysis of vulnerabilities in application containers (currently including appc and Docker). - https://github.com/coreos/clair

Please do not ignore the threats which may come from the containers and make the vulnerabilities scan a mandatory step before publishing any images.

Going further, let us talk about gVisor, the container runtime sandbox, which takes another perspective on security by fencing the containers at runtime.

gVisor is a user-space kernel, written in Go, that implements a substantial portion of the Linux system surface. It includes an Open Container Initiative (OCI) runtime called `runsc` that provides an isolation boundary between the application and the host kernel. - https://github.com/google/gvisor

This technology is quite new and certain limitations still exists but it opens a whole new horizon for running the containers securely.

7.11 Watch Your Logs

It is astonishing how often the application or service logs become the security hole by leaking the sensitive or personally identifiable information (PII). The common approaches to address such issues are to use masking, filtering, sanitization and data anonymization.

This OWASP Logging Cheat Sheet is a focused document to provide the authoritative guidance on building application logging mechanisms, especially related to security logging.

7.12 Orchestration

Up to now we have been mostly concentrated on how to make the security measures an integral part of the applications and services, using dedicated libraries and frameworks. It is all good but over time you may see the same patterns reappearing over and over again. Won't it be wonderful if we could offload such repeating cross-cutting concerns somewhere else? It makes a perfect sense and at some extent, it is already happening ...

In case you orchestrate your microservices deployments using Apache Mesos or Kubernetes, there are quite a lot of the security-related features which you get for free. However, the most interesting developments are happening in the newborn infrastructure layer, called service meshes.

The most advanced, production-ready service meshes as of now include Istio, Linkerd and Consul Service Mesh. Although we are going to talk more about these things later in the tutorial, it is worth to mention that they take upon themselves a large set of concerns by following best security practices and conventions.

7.13 Sealed Cloud

So far we have talked about open-sourced solutions which are by and large agnostic to the hosting environment but if you are looking towards deploying your microservices in the cloud, it would make more sense to learn the managed options your cloud provider offers.

Let us take a quick look at what the leaders in the space have for you. Microsoft Azure includes Key Vault to encrypt keys and small secrets (like passwords) but the complete list of security-related services is quite comprehensive. It also has a security center, one-stop service for unified security management and advanced threat protection.

The list of security-related products which are offered by Google Cloud is impressive. Among many, there is also a dedicated service to manage encryption keys, Key Management Service (or shortly KMS) which, surprisingly, does not directly store secrets (it could only encrypt secrets that you should store elsewhere). The security portal is a great resource to learn your options. AWS , the long-time leader in the space, has many security products to choose from. It even offers two distinct services to manage your encryption keys and secrets, Key Management Service (KMS) and Secrets Manager. Besides the managed offering,it is worth to mention the open-sourced Confidant from Lyft which stores secrets in DynamoDB using encryption at rest. This reference to the cloud security web page will help you to get started.

7.14 Conclusions

For most of us security is a difficult subject. And applying proper security boundaries and measures while implementing microservice architecture is even more difficult but absolutely necessary. In this part of the tutorial we have highlighted a set of key topics you may very likely run into but it is far from being exhaustive.

7.15 What's next

In the next section of the tutorial we are going to talk about various testing practices and techniques in the context of microservice architecture.

Chapter 8

Testing

8.1 Introduction

Since Kent Beck coined the idea of test-driven development (TDD) more than a decade ago, testing became an absolutely essential part of every software project which aims for success. Years passed, the complexity of the software systems has grown enormously so did the testing techniques but the same foundational principles are still there and apply.

Efficient and effective testing is a very large subject, full of opinions and surrounded by never ending debates of **Dos** and **Don'ts** . Many think about testing as an art, and for good reasons. In this part of the tutorial we are not going to join any camps but instead focus on testing the applications which are implemented after the principles of the microservice architecture. Even in such narrowed subject, there are just too many topics to talk about so the upcoming parts of the tutorial will be dedicated to performance and security testing respectively.

But before we start off, please take some time to go over Marin Fowler's Testing Strategies in a Microservice Architecture, the brilliant, detailed and well-illustrated summary of the approaches to manage the testing complexity in the world of microservices.

8.2 Unit Testing

Unit testing is the probably the simplest, yet very powerful, form of testing which is not really specific to the microservices but any class of applications or services.

A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected. - https://martinfowler.com/articles/microservice-testing/#testing-unit-introduction

Unit tests usually should constitute the largest part of the application test suite (as per practical test pyramid) since they supposed to be very easy to write and fast to execute. In Java, JUnit framework (JUnit 4 and JUnit 5) is the de-facto pick these days (although other frameworks like TestNG or Spock are also widely used).

What could be a good example of unit test? Surprisingly, it is very difficult question to answer, but there are a few rules to follow: it should test one specific component ("unit") in isolation, it should test one thing at a time and it should be fast.

There are many unit tests which come as part of service test suites of the **JCG Car Rentals** platform. Let us pick the **Customer Service** and take a look on the fragment of the test suite for `AddressToAddressEntityConverter` class, which converts the `Address` data transfer object to corresponding JPA persistent entity.

```
public class AddressToAddressEntityConverterTest {
    private AddressToAddressEntityConverter converter;

    @Before
    public void setUp() {
        converter = new AddressToAddressEntityConverter();
    }
```

```
    @Test
    public void testConvertingNullValueShouldReturnNull() {
        assertThat(converter.convert(null)).isNull();
    }

    @Test
    public void testConvertingAddressShouldSetAllNonNullFields() {
        final UUID uuid = UUID.randomUUID();

        final Address address = new Address(uuid)
            .withStreetLine1("7393 Plymouth Lane")
            .withPostalCode("19064")
            .withCity("Springfield")
            .withStateOrProvince("PA")
            .withCountry("United States of America");

        assertThat(converter.convert(address))
            .isNotNull()
            .hasFieldOrPropertyWithValue("uuid", uuid)
            .hasFieldOrPropertyWithValue("streetLine1", "7393 Plymouth Lane")
            .hasFieldOrPropertyWithValue("streetLine2", null)
            .hasFieldOrPropertyWithValue("postalCode", "19064")
            .hasFieldOrPropertyWithValue("city", "Springfield")
            .hasFieldOrPropertyWithValue("stateOrProvince", "PA")
            .hasFieldOrPropertyWithValue("country", "United States of America");
    }
}
```

The test is quite straightforward, it is easy to read, understand and troubleshoot any failures which may occur in the future. In real projects, the unit tests may get out of control very fast, become bloated and difficult to maintain. There is no universal treatment for such disease, but the general advice is to look at the test cases as the mainstream code.

8.3 Integration Testing

In reality, the components (or "units") in our applications often have dependencies on other components, data storages, external services, caches, message brokers, … Since unit tests are focusing on isolation, we need to go up one level and switch over to integration testing.

An integration test verifies the communication paths and interactions between components to detect interface defects. - https://martinfowler.com/articles/microservice-testing/#testing-integration-introduction

Probably the best example to demonstrate the power of the integration testing is to come up with the suite to test the persistence layer. This is the area where frameworks like Arquillian, Mockito, DBUnit, Wiremock, Testcontainers, REST Assured (and many others) take the lead.

Let us get back to the **Customer Service** and think about how to ensure that the customer data is indeed persistent in the database. We have a dedicated **RegistrationService** to manage the registration process, so what we need is to provide the database instance, wire all the dependencies and initiate the registration process.

```
@RunWith(Arquillian.class)
public class TransactionalRegistrationServiceIT {
    @Inject private RegistrationService service;

    @Deployment
    public static JavaArchive createArchive() {
        return ShrinkWrap
            .create(JavaArchive.class)
            .addClasses(CustomerJpaRepository.class, PersistenceConfig.class)
            .addClasses(ConversionService.class, TransactionalRegistrationService.class)
            .addPackages(true, "org.apache.deltaspike")
```

```
            .addPackages(true, "com.javacodegeeks.rentals.customer.conversion")
            .addPackages(true, "com.javacodegeeks.rentals.customer.registration.conversion" ↩
                );
    }

    @Test
    public void testRegisterNewCustomer() {
        final RegisterAddress homeAddress = new RegisterAddress()
            .withStreetLine1("7393 Plymouth Lane")
            .withPostalCode("19064")
            .withCity("Springfield")
            .withCountry("United States of America")
            .withStateOrProvince("PA");

        final RegisterCustomer registerCustomer = new RegisterCustomer()
            .withFirstName("John")
            .withLastName("Smith")
            .withEmail("john@smith.com")
            .withHomeAddress(homeAddress);

        final UUID uuid = UUID.randomUUID();
        final Customer customer = service.register(uuid, registerCustomer);

        assertThat(customer).isNotNull()
            .satisfies(c -> {
                assertThat(c.getUuid()).isEqualTo(uuid);
                assertThat(c.getFirstName()).isEqualTo("John");
                assertThat(c.getLastName()).isEqualTo("Smith");
                assertThat(c.getEmail()).isEqualTo("john@smith.com");
                assertThat(c.getBillingAddress()).isNull();
                assertThat(customer.getHomeAddress()).isNotNull()
                    .satisfies(a -> {
                        assertThat(a.getUuid()).isNotNull();
                        assertThat(a.getStreetLine1()).isEqualTo("7393 Plymouth Lane");
                        assertThat(a.getStreetLine2()).isNull();
                        assertThat(a.getCity()).isEqualTo("Springfield");
                        assertThat(a.getPostalCode()).isEqualTo("19064");
                        assertThat(a.getStateOrProvince()).isEqualTo("PA");
                        assertThat(a.getCountry()).isEqualTo("United States of America");
                    });
            });
    }
}
```

This is an Arquillian-based test suite where we have configured in-memory H2 database engine in PostgreSQL compatibility mode (through the properties file). Even in this configuration it may take up to 15-25 seconds to run, still much faster than spinning the dedicated instance of PostgreSQL database.

Trading integration tests execution time by substituting the integration components is one of the viable techniques to obtain feedback faster. It certainly may not work for everyone and everything so we will get back to this subject later on in this part of the tutorial.

If your microservices are built on top of Spring Framework and Spring Boot, like for example our **Reservation Service**, you would definitely benefit from auto-configured test slices and beans mocking. The snippet below, part of the `ReservationController` test suite, illustrate the usage of the `@WebFluxTest` test slice in action.

```
@WebFluxTest(ReservationController.class)
class ReservationControllerTest {
    private final String username = "b36dbc74-1498-49bd-adec-0b53c2b268f8";

    private final UUID customerId = UUID.fromString(username);
    private final UUID vehicleId = UUID.fromString("397a3c5c-5c7b-4652-a11a-f30e8a522bf6");
```

```
    private final UUID reservationId = UUID.fromString("3f8bc729-253d-4d8f-bff2- ↩
        bc07e1a93af6");

    @Autowired
    private WebTestClient webClient;
    @MockBean
    private ReservationService service;
    @MockBean
    private InventoryServiceClient inventoryServiceClient;

    @Test
    @DisplayName("Should create Customer reservation")
    @WithMockUser(roles = "CUSTOMER", username = username)
    public void shouldCreateCustomerReservation() {
        final OffsetDateTime reserveFrom = OffsetDateTime.now().plusDays(1);
        final OffsetDateTime reserveTo = reserveFrom.plusDays(2);

        when(inventoryServiceClient.availability(eq(vehicleId)))
            .thenReturn(Mono.just(new Availability(vehicleId, true)));

        when(service.reserve(eq(customerId), any()))
            .thenReturn(Mono.just(new Reservation(reservationId)));

        webClient
            .mutateWith(csrf())
            .post()
            .uri("/api/reservations")
            .accept(MediaType.APPLICATION_JSON_UTF8)
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            .body(BodyInserters
                .fromObject(new CreateReservation()
                    .withVehicleId(vehicleId)
                    .withFrom(reserveFrom)
                    .withTo(reserveTo)))
            .exchange()
            .expectStatus().isCreated()
            .expectBody(Reservation.class)
            .value(r -> {
                assertThat(r)
                    .extracting(Reservation::getId)
                    .isEqualTo(reservationId);
            });
    }
}
```

To be fair, it is amazing to see how much efforts and thoughts the Spring team invests into the testing support. Not only we are able to cover the most of the request and response processing without spinning the server instance, the test execution time is blazingly fast.

Another interesting concept you will often encounter, specifically in integration testing, is using fakes, stubs, test doubles and/or mocks.

8.4 Testing Asynchronous Flows

It is very likely that sooner or later you may face the need to test some kind of functionality which relies on asynchronous processing. To be honest, without using dedicated dispatchers or executors, it is really difficult due to the non-deterministic nature of the execution flow.

If we rewind a bit to the moment when we discussed microservices implementation, we would run into the flow in the **Customer Service** which relies on asynchronous event propagation provided by CDI 2.0. How would we test that? Let us find out one of

the possible ways to approach this problem by dissecting the snippet below.

```
@RunWith(Arquillian.class)
public class NotificationServiceTest {
    @Inject private RegistrationService registrationService;
    @Inject private TestNotificationService notificationService;

    @Deployment
    public static JavaArchive createArchive() {
        return ShrinkWrap
            .create(JavaArchive.class)
            .addClasses(TestNotificationService.class, StubCustomerRepository.class)
            .addClasses(ConversionService.class, TransactionalRegistrationService.class,  ↩
                RegistrationEventObserver.class)
            .addPackages(true, "org.apache.deltaspike.core")
            .addPackages(true, "com.javacodegeeks.rentals.customer.conversion")
            .addPackages(true, "com.javacodegeeks.rentals.customer.registration.conversion" ↩
                );
    }

    @Test
    public void testCustomerRegistrationEventIsFired() {
        final UUID uuid = UUID.randomUUID();
        final Customer customer = registrationService.register(uuid, new RegisterCustomer() ↩
            );

        await()
            .atMost(1, TimeUnit.SECONDS)
            .until(() -> !notificationService.getTemplates().isEmpty());

        assertThat(notificationService.getTemplates())
            .hasSize(1)
            .hasOnlyElementsOfType(RegistrationTemplate.class)
            .extracting("customerId")
            .containsOnly(customer.getUuid());
    }
}
```

Since the event is fired and consumed asynchronously, we cannot make the assertions predictably but take into account the timing aspect by using Awaitility library. Also, we do not really need to involve the persistence layer in this test suite so we provide our own (quite dumb to be fair) `StubCustomerRepository` implementation to speed up test execution.

```
@Singleton
public static class StubCustomerRepository implements CustomerRepository {
    @Override
    public Optional<CustomerEntity> findById(UUID uuid) {
        return Optional.empty();
    }

    @Override
    public CustomerEntity saveOrUpdate(CustomerEntity entity) {
        return entity;
    }

    @Override
    public boolean deleteById(UUID uuid) {
        return false;
    }
}
```

Still, even with this approach there are opportunities for instability. The dedicated test dispatchers and executors may yield better results but not every framework provides them or supports easy means to plug them in.

8.5 Testing Scheduled Tasks

The work, which is supposed to be done (or scheduled) at specific time, poses an interesting challenge from the testing perspective. How would we make sure the schedule meets the expectations? It would be impractical (but believe it or not, realistic) to have test suites running for hours or days waiting for task to be triggered. Luckily, there are few options to consider. For application and services which use Spring Framework the simplest but quite reliable route is to use `CronTrigger` and mock (or stub) `TriggerContext`, for example.

```
class SchedulingTest {
    private final class TestTriggerContext implements TriggerContext {
        private final Date lastExecutionTime;

        private TestTriggerContext(LocalDateTime lastExecutionTime) {
            this.lastExecutionTime =  Date.from(lastExecutionTime.atZone(ZoneId. ↩
                systemDefault()).toInstant());
        }

        @Override
        public Date lastScheduledExecutionTime() {
            return lastExecutionTime;
        }

        @Override
        public Date lastActualExecutionTime() {
            return lastExecutionTime;
        }

        @Override
        public Date lastCompletionTime() {
            return lastExecutionTime;
        }
    }

    @Test
    public void testScheduling(){
        final CronTrigger trigger = new CronTrigger("0 */30 * * * *");

        final LocalDateTime lastExecutionTime = LocalDateTime.of(2019, 01, 01, 10, 00, 00);
        final Date nextExecutionTime = trigger.nextExecutionTime(new TestTriggerContext( ↩
            lastExecutionTime));

        assertThat(nextExecutionTime)
            .hasYear(2019)
            .hasMonth(01)
            .hasDayOfMonth(01)
            .hasHourOfDay(10)
            .hasMinute(30)
            .hasSecond(0);
    }
}
```

The test case above uses fixed `CronTrigger` expression and verifies the next execution time but it could be also populated from the properties or even class method annotations.

Alternatively to verifying the schedule itself, you may find it very useful to rely on virtual clock and literally "travel in time". For example, you could pass around the instance of the `Clock` abstract class (the part of Java Standard Library) and substitute it with stub or mock in the tests.

8.6 Testing Reactive Flows

The popularity of the reactive programming paradigm has had a deep impact on the testing approaches we used to employ. In fact, testing support is the first class citizen in any reactive framework: RxJava, Project Reactor or Akka Streams, you name it.

Our **Reservation Service** is built using Spring Reactive stack all the way and is great candidate to illustrate the usage of dedicated scaffolding to test the reactive APIs.

```
@Testcontainers
@SpringBootTest(
    classes = ReservationRepositoryIT.Config.class,
    webEnvironment = WebEnvironment.NONE
)
public class ReservationRepositoryIT {
    @Container
    private static final GenericContainer<?> container = new GenericContainer<>("cassandra ↩
        :3.11.3")
        .withTmpFs(Collections.singletonMap("/var/lib/cassandra", "rw"))
        .withExposedPorts(9042)
        .withStartupTimeout(Duration.ofMinutes(2));

    @Configuration
    @EnableReactiveCassandraRepositories
    @ImportAutoConfiguration(CassandraMigrationAutoConfiguration.class)
    @Import(CassandraConfiguration.class)
    static class Config {
    }

    @Autowired
    private ReservationRepository repository;

    @Test
    @DisplayName("Should insert Customer reservations")
    public void shouldInsertCustomerReservations() {
        final UUID customerId = randomUUID();

        final Flux<ReservationEntity> reservations =
            repository
                .deleteAll()
                .thenMany(
                    repository.saveAll(
                        Flux.just(
                            new ReservationEntity(randomUUID(), randomUUID())
                                .withCustomerId(customerId),
                            new ReservationEntity(randomUUID(), randomUUID())
                                .withCustomerId(customerId))));

        StepVerifier
            .create(reservations)
            .expectNextCount(2)
            .verifyComplete();
    }
}
```

Besides utilizing Spring Boot testing support, this test suite relies on outstanding Spring Reactor test capabilities in the form of `StepVerifier` where the expectations are defined in terms of events to expect on each step. The functionality which `StepVerifier` and family provide is quite sufficient to cover arbitrary complex scenarios.

One more thing to mention here is the usage of Testcontainers framework and bootstrapping the dedicated data storage instance (in this case, Apache Cassandra) for persistence. With that, not only the reactive flows are tested, the integration test is using the real components, sticking as close as possible to real production conditions. The price for that is higher resource demands and significantly increased time of the test suites execution.

8.7 Contract Testing

In a loosely coupled microservice architecture, the contracts are the only things which each service publishes and consumes. The contract could be expressed in IDLs like Protocol Buffers or Apache Thrift which makes it comparatively easy to communicate, evolve and consume. But for HTTP-based RESTful web APIs it would be more likely some form of blueprint or specification. In this case, the question becomes: how the consumer could assert the expectations against such contracts? And more importantly, how the provider could evolve the contract without breaking existing consumers?

Those are the hard problems where consumer-driven contract testing could be very helpful. The idea is pretty simple. The provider publishes the contract. The consumer creates the tests to make sure it has the right interpretation of the contract. Interestingly, the consumer may not need to use all APIs but just the subset it really needs to have the job done. And lastly, consumer communicates these tests back to provider. This last step is quite important as it helps the provider to evolve the APIs without disrupting the consumers.

In the JVM ecosystem, Pact JVM and Spring Cloud Contract are two the most popular libraries for consumer-driven contract testing. Let us take a look on how the **JCG Car Rentals Customer Admin Portal** may use Pact JVM to add the consumer-driven contract test for one of the **Customer Service** APIs using the OpenAPI specification it publishes.

```
public class RegistrationApiContractTest {
    private static final String PROVIDER_ID = "Customer Service";
    private static final String CONSUMER_ID = "JCG Car Rentals Admin";

    @Rule
    public ValidatedPactProviderRule provider = new ValidatedPactProviderRule(getContract() ↩
        , null, PROVIDER_ID,
        "localhost", randomPort(), this);

    private String getContract() {
        return getClass().getResource("/contract/openapi.json").toExternalForm();
    }

    @Pact(provider = PROVIDER_ID, consumer = CONSUMER_ID)
    public RequestResponsePact registerCustomer(PactDslWithProvider builder) {
        return builder
            .uponReceiving("registration request")
            .method("POST")
            .path("/customers")
            .body(
                new PactDslJsonBody()
                    .stringType("email")
                    .stringType("firstName")
                    .stringType("lastName")
                    .object("homeAddress")
                        .stringType("streetLine1")
                        .stringType("city")
                        .stringType("postalCode")
                        .stringType("stateOrProvince")
                        .stringType("country")
                        .closeObject()
            )
            .willRespondWith()
            .status(201)
            .matchHeader(HttpHeaders.CONTENT_TYPE, "application/json")
            .body(
                new PactDslJsonBody()
                    .uuid("id")
                    .stringType("email")
                    .stringType("firstName")
                    .stringType("lastName")
                    .object("homeAddress")
                        .stringType("streetLine1")
```

```
                        .stringType("city")
                        .stringType("postalCode")
                        .stringType("stateOrProvince")
                        .stringType("country")
                        .closeObject())
            .toPact();
    }

    @Test
    @PactVerification(value = PROVIDER_ID, fragment = "registerCustomer")
    public void testRegisterCustomer() {
        given()
            .contentType(ContentType.JSON)
            .body(Json
                .createObjectBuilder()
                .add("email", "john@smith.com")
                .add("firstName", "John")
                .add("lastName", "Smith")
                .add("homeAddress", Json
                    .createObjectBuilder()
                    .add("streetLine1", "7393 Plymouth Lane")
                    .add("city", "Springfield")
                    .add("postalCode", "19064")
                    .add("stateOrProvince", "PA")
                    .add("country", "United States of America"))
                .build())
            .post(provider.getConfig().url() + "/customers");
    }
}
```

There are many ways to write the consumer-driven contract tests, above is just one favor of it. It does not matter much what approach you are going to follow, the quality of your microservice architecture is going to improve.

Pushing it further, the tools like swagger-diff, Swagger Brake and assertj-swagger are very helpful in validating the changes in the contracts (since it is a living thing in most cases) and making sure the service is properly implementing the contract it claims to.

If this is not enough, one of the invaluable tools out there is Diffy from Twitter which helps to find potential bugs in the services using running instances of new version and old version side by side. It behaves more like a proxy which routes whatever requests it receives to each of the running instances and then compares the responses.

8.8 Component Testing

On the top of the testing pyramid of a single microservice sit the component tests. Essentially, they exercise the real, ideally production-like, deployment with only external services stubbed (or mocked).

Let us get back to **Reservation Service** and walk through the component test we may come up with. Since it relies on the **Inventory Service**, we need to mock this external dependency. To do that, we could benefit from Spring Cloud Contract WireMock extension which, as the name implies, is based on WireMock. Besides **Inventory Service** we also mock the security provider by using `@MockBean` annotation.

```
@AutoConfigureWireMock(port = 0)
@Testcontainers
@SpringBootTest(
    webEnvironment = WebEnvironment.RANDOM_PORT,
    properties = {
        "services.inventory.url=https://localhost:${wiremock.server.port}"
    }
)
class ReservationServiceIT {
```

```
    private final String username = "ac2a4b5d-a35f-408e-a652-47aa8bf66bc5";

    private final UUID vehicleId = UUID.fromString("4091ffa2-02fa-4f09-8107-47d0187f9e33");
    private final UUID customerId = UUID.fromString(username);

    @Autowired private ObjectMapper objectMapper;
    @Autowired private ApplicationContext context;
    @MockBean private ReactiveJwtDecoder reactiveJwtDecoder;
    private WebTestClient webClient;

    @Container
    private static final GenericContainer<?> container = new GenericContainer<>("cassandra ↩
        :3.11.3")
        .withTmpFs(Collections.singletonMap("/var/lib/cassandra", "rw"))
        .withExposedPorts(9042)
        .withStartupTimeout(Duration.ofMinutes(2));

    @BeforeEach
    public void setup() {
        webClient = WebTestClient
            .bindToApplicationContext(context)
            .apply(springSecurity())
            .configureClient()
            .build();
    }

    @Test
    @DisplayName("Should create Customer reservations")
    public void shouldCreateCustomerReservation() throws JsonProcessingException {
        final OffsetDateTime reserveFrom = OffsetDateTime.now().plusDays(1);
        final OffsetDateTime reserveTo = reserveFrom.plusDays(2);

        stubFor(get(urlEqualTo("/" + vehicleId + "/availability"))
            .willReturn(aResponse()
                .withHeader("Content-Type", "application/json")
                .withBody(objectMapper.writeValueAsString(new Availability(vehicleId, true) ↩
                    ))));

        webClient
            .mutateWith(mockUser(username).roles("CUSTOMER"))
            .mutateWith(csrf())
            .post()
            .uri("/api/reservations")
            .accept(MediaType.APPLICATION_JSON_UTF8)
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            .body(BodyInserters
                .fromObject(new CreateReservation()
                    .withVehicleId(vehicleId)
                    .withFrom(reserveFrom)
                    .withTo(reserveTo)))
            .exchange()
            .expectStatus().isCreated()
            .expectBody(Reservation.class)
            .value(r -> {
                assertThat(r)
                    .extracting(Reservation::getCustomerId, Reservation::getVehicleId)
                    .containsOnly(vehicleId, customerId);
            });
    }
}
```

Despite the fact that a lot of things are happening under the hood, the test case is still looking quite manageable but the time it

needs to run is close to 50 seconds now.

While designing the component tests, please keep in mind that there should be no shortcuts taken (like for example mutating the data in the database directly). If you need some prerequisites or the way to assert over internal service state, consider introducing the supporting APIs which are available at test time only (enabled, for example, using profiles or configuration properties).

8.9 End-To-End Testing

The purpose of end-to-end tests is to verify that the whole system works as expected and as such, the assumption is to have a full-fledge deployment of all the components. Though being very important, the end-to-end tests are the most complex, expensive, slow and, as practice shows, most brittle ones.

Typically, the end-to-end tests are designed after the workflows performed by the users, from the beginning to the end. Because of that, often the entry point into the system is some kind of mobile or web frontend so the testing frameworks like Geb, Selenium and Robot Framework are quite popular choices here.

8.10 Fault Injection and Chaos Engineering

It would be fair to say most of tests are biased towards a "happy path" and do not explore the faulty scenarios, unless trivial ones, like for example the record is not present in the data store or input is not valid. How often have you seen test suites which deliberately introduce database connectivity issues?

As we have stated in the previous part of the tutorial, the bad things will happen and it is better to be prepared. The discipline of chaos engineering gave the birth to many different libraries, frameworks and toolkits to perform fault injection and simulation.

To fabricate different kind of network issues, you may start with Blockade, Saboteur or Comcast, all of those are focused on network faults and partitions injection and aim to simplify resilience and stability testing.

The Chaos Toolkit is a more advanced and disciplined approach for conducting the chaos experiments. It also integrates quite well with most popular orchestration engines and cloud providers. In the same vein, the SimianArmy from Netflix is one of the earliest (if not the first) cloud-oriented tools for generating various kinds of failures and detecting abnormal conditions. For services built on top of Spring Boot stack, there is a dedicated project called Chaos Monkey for Spring Boot which you may have heard of already. It is pretty young but evolves fast and is very promising.

This kind of testing is quite new for the most organizations out there, but in the context of the microservice architecture, it is absolutely worth considering and investing. These tests give you a confidence that the system is able to survive outages by degrading the functionality gradually instead of catching the fire and burning in flames. Many organizations (like Netflix for example) do conduct chaos experiments in production regularly, proactively detecting the issues and fixing them.

8.11 Conclusions

In this part of the tutorial we were focused on testing. Our coverage it is far from being exhausting and complete, since there are some many different kind of tests. In many regards, testing individual microservices is not much different, the same good practices apply. But the distributed nature of such architecture brings a lot of unique challenges, which Contract Testing along with Fault Injection and Chaos Engineering are trying to address.

To finish up, the series of articles Testing Microservices, the sane way and Testing in Production, the safe way are the terrific sources of great insights and advices on what works and how to avoid the common pitfalls while testing the microservices.

8.12 What's next

In the next section of the tutorial we are going to continue the subject of testing and talk about performance (load and stress) testing.

Chapter 9

Performance and Load Testing

9.1 Introduction

These days numerous frameworks and libraries make it is pretty easy to get from literally nothing to a full-fledged running application or service in a matter of hours. It is really amazing and you may totally get away with that but more often than not the decisions which frameworks make on your behalf (often called "sensitive defaults") are far from being optimal (or even sufficient) in the context of the specific application or service (and truth to be said, it is hardly possible to come up with one-size-fits-all solution).

In this section of the tutorial we are going to talk about performance and load testing, focusing on the tools to help you with achieving your goals and also highlight the typical areas of the application to tune. It is worth noting that some techniques may apply to one individual microservice but in most cases the emphasis should gravitate towards the entire microservice architecture ensemble.

Often enough the terms performance and load testing are used interchangeably, however this is a misconception. It is true that these testing techniques often come together but each of them sets different goals. The performance testing helps you to assess how fast the system under test is whereas the load testing helps you to understand the limits of the system under the test. These answers are very sensitive to the context the system is running in, so it is always recommended to design the simulations as close to production conditions as possible.

The methodology of the performance and load testing is left out of this part of the tutorial since there are just too many things to cover there. I would highly recommend the book Systems Performance: Enterprise and the Cloud by Brendan Gregg to deeply understand the performance and scalability aspects throughout the complete software stack.

9.2 Make Friends with JVM and GC

The success of the Java is largely indebted to its runtime (JVM) and automatic memory management (GC). Over the years JVM has turned into a very sophisticated piece of technology with a lot of things being built on top of it. This is why it is often referred as "JVM platform".

There are two main open sourced, production-ready JVM implementations out there: HotSpot and Eclipse OpenJ9. Fairly speaking, HotSpot is in dominant position but Eclipse OpenJ9 is looking quite promising for certain kind of applications. The picture would be incomplete without mentioning the GraalVM, a high-performance polyglot VM, based on SubstrateVM. Picking the right JVM could be an easy win from the start.

With respect to memory management and garbage collection (GC), the things are much more complicated. Depending on the version of the JDK (8 or 11) and the vendor, we are talking about Serial GC, Parallel GC, CMS, G1, ZGC and Shenandoah. The JDK 11 release introduced an experimental Epsilon GC, which is effectively a no-op GC.

Tuning GC is an art and requires deep understanding on how JVM works. The JVM Anatomy Park is one of the best and up-to-date sources of the invaluable insights on JVM and GC internals. But how would you diagnose the problems in your applications and actually figure out what to tune?

To our luck, this is now possible with the help of two great tools, Java Mission Control and Java Flight Recorder, which have been open sourced as of the JDK 11 release. These tools are available for HotSpot VM only and are exceptionally easy to use, even in production.

Last but not least, let us talk for a moment about how the containerization (or better to say, Docker'ization) impacts the JVM behavior. Since the JDK 10 and JDK 8 Update 191 the JVM has been modified to be fully aware that it is running in a Docker container and is able to properly extract the allocated number of CPUs and total memory.

9.3 Microbenchmarks

Adjusting GC and JVM settings to the needs of your applications and services is difficult but rewarding exercise. However, it very likely will not help when JVM stumbles upon the inefficient code. More often than not the implementation has to be rewritten from the scratch or refactored, but how to make sure that it outperforms the old one? The microbenchmarking techniques backed by JHM tool are here to help.

JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targetting the JVM. - https://openjdk.java.net/projects/code-tools/jmh/

You may be wondering why use the dedicated tool for that? In the nutshell, the benchmarking looks easy, just run the code in question in a loop and measure the time, right? In fact, writing the benchmarks which properly measure the performance of the reasonably small parts of the application is very difficult, specifically when JVM is involved. There are many optimizations which JVM could apply taking into the account the much smaller scope of the isolated code fragments being benchmarked. This is the primary reason you need the tools like JHM which is aware of the JVM behavior and guides you towards implementing and running the benchmark correctly, so you would end up with the measurements you could trust.

The JHM repository has a large number of samples to look at and derive your own, but if you want to learn more on the subject, Optimizing Java: Practical Techniques for Improving JVM Application Performance by Benjamin J Evans, James Gough and Chris Newland is a terrific book to look into.

Once you master the JHM and start to use it day by day, the comparison of the microbenchmarks may become a tedious process. The JMH Compare GUI is a small GUI tool which could help you to compare these results visually.

9.4 Apache JMeter

Let us switch gears from micro- to macrobenchmarking and talk about measuring the performance of the applications and services deployed somewhere. The first tool we are going to look at is Apache JMeter, probably one of the oldest tools in this category.

The Apache JMeter application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions. - https://jmeter.apache.org/

Apache JMeter advocates the UI-based approach to create and manage quite sophisticated test plans. The UI itself is pretty intuitive and it won't take long to have your first scenario out. One of the strongest sides of the Apache JMeter is high level of extensibility and scripting support.

The **Reservation Service** is a core of the **JCG Car Rentals** platform, so the screenshot below gives a sneak peak on the simple test plan against reservation RESTful API.

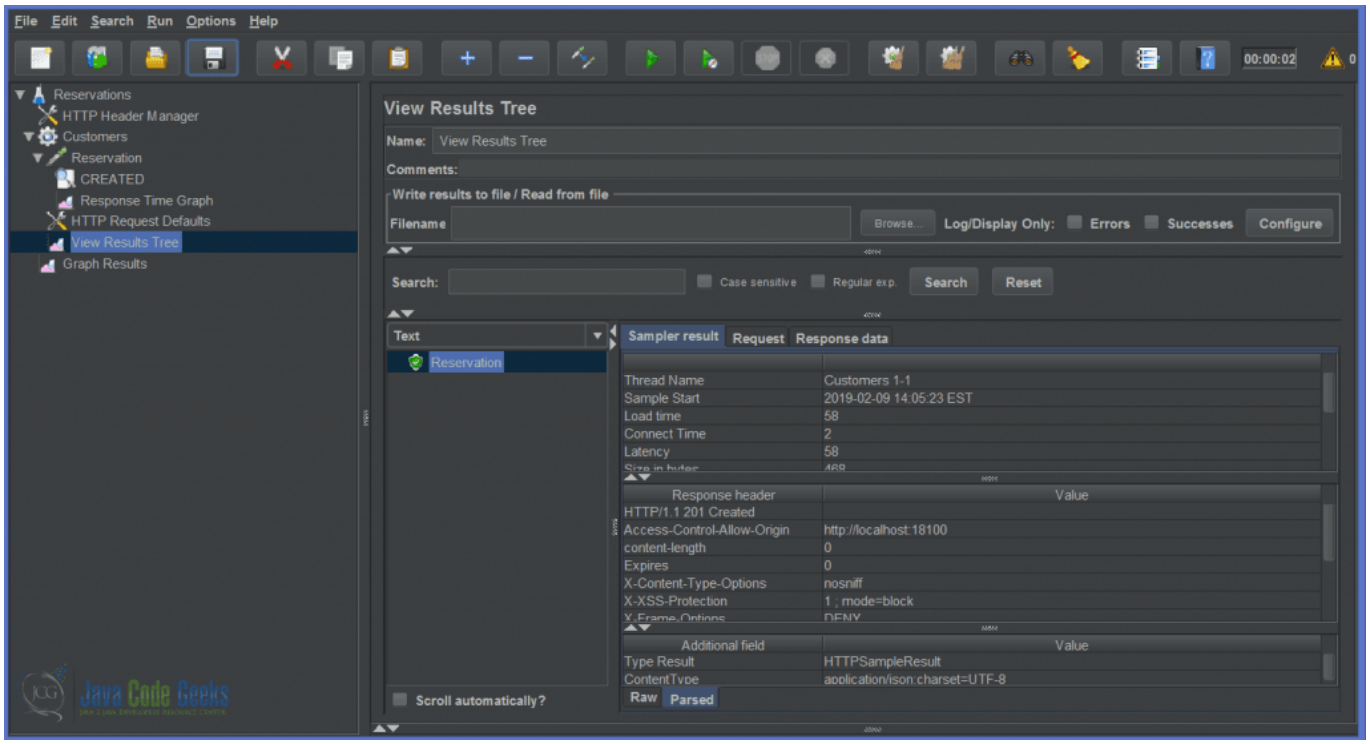


Figure 9.1: Image

The presence of the user-friendly interface is great for humans but not for automated tooling. Luckily, the Apache JMeter test plans could be run from command line, using Apache Maven plugin, Gradle plugin or even embedded into the application test harness.

The ability to be easily injected into continuous integration pipelines makes Apache JMeter a great fit for developing automated load and performance test scenarios.

9.5 Gatling

There are quite a few load testing frameworks which promote the code-first approach to test scenarios, with Gatling being one of the best examples.

Gatling is a highly capable load testing tool. It is designed for ease of use, maintainability and high performance. - https://gatling.io/docs/current/

The Gatling scenarios are written in Scala but this aspect is abstracted away behind the concise DSL, so the knowledge of Scala is desired although not required. Let us re-implement the Apache JMeter test scenario for **Reservation Service** using Gatling code-first approach.

```
class ReservationSimulation extends Simulation {
  val tokens: Map[String, String] = TrieMap[String, String]()
  val customers = csv("customers.csv").circular()

  val protocol = http
    .baseUrl("https://localhost:17000")
    .contentTypeHeader("application/json")

  val reservation = scenario("Simulate Reservation")
    .feed(customers)
    .doIfOrElse(session => tokens.get(session("username").as[String]) == None) {
      KeycloakToken
```

```
        .token
        .exec(session => {
          tokens.replace(session("username").as[String], session("token").as[String])
          session
        })
    } {
      exec(session => {
        tokens.get(session("username").as[String]).fold(session)(session.set("token", _))
      })
    }
    .exec(
      http("Reservation Request")
        .post("/reservations")
        .header("Authorization", "Bearer ${token}")
        .body(ElFileBody("reservation-payload.json")).asJson
        .check(status.is(201)))

  setUp(
    reservation.inject(rampUsers(10) during (20 seconds))
  ).protocols(protocol)
}
```

The test scenario, or in terms of Gatling, simulation, is pretty easy to follow. A minor complication arises from the need to obtain the access token using the Keycloak APIs but there are several ways to resolve it. In the simulation above we made it a part of the reservation flow backed by in-memory token cache. As you could see, complex, multi-step simulations are looking easy in Gatling.

The reporting side of Gatling is really amazing. Out of the box you get the simulation results in a beautiful HTML markup, the picture below is just a small fragment of it. You could also extract this data from the simulation log file and interpret it in the way you need.

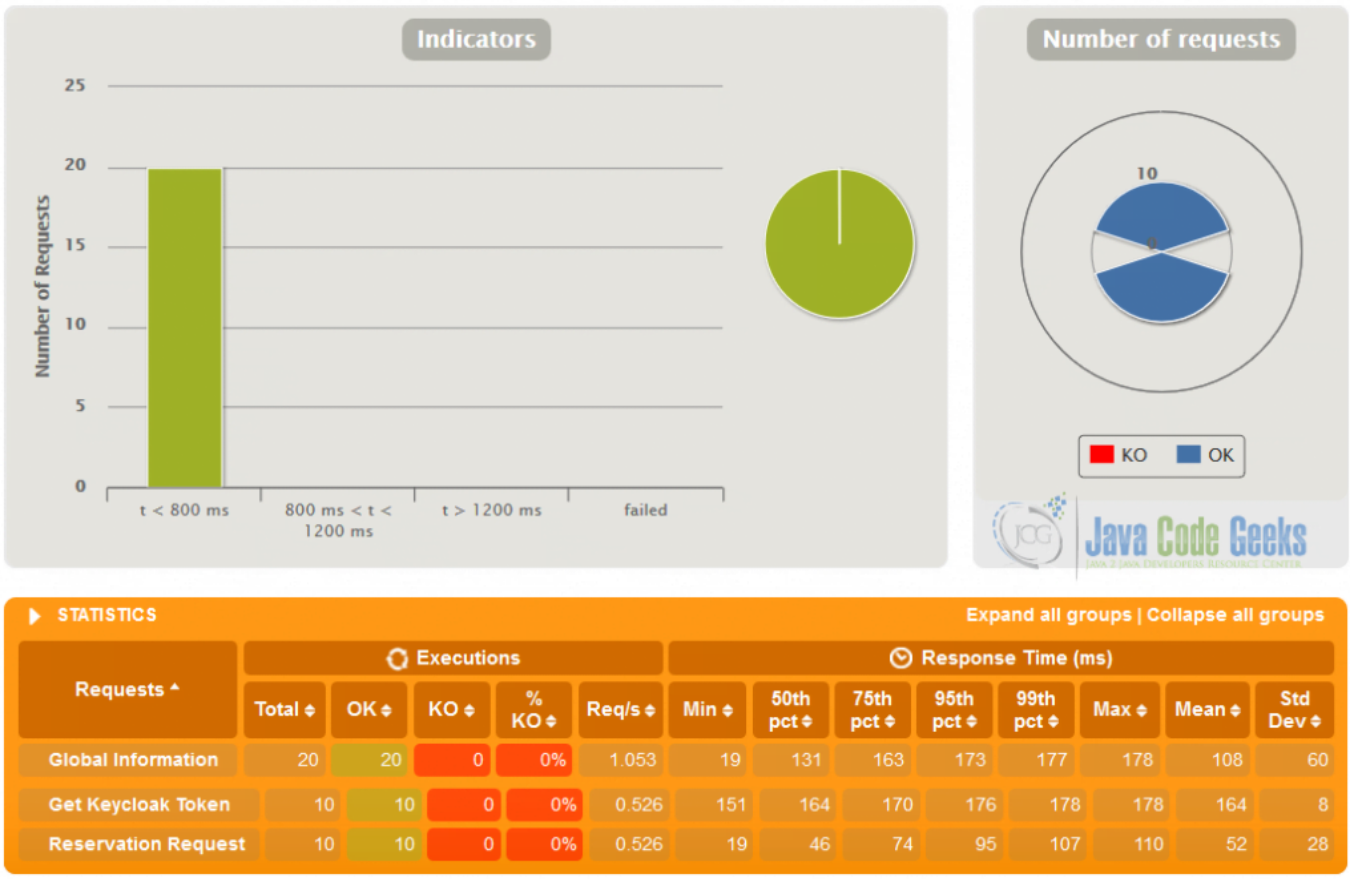


Figure 9.2: Image

From the early days Gatling was designed for continuous load testing and integrates very well with Apache Maven, SBT, Gradle, and continuous integration pipelines. There are a number of extensions available to support wide variety of the protocols (and you are certainly welcome to contribute there).

9.6 Command-Line Tooling

The command line tools are probably the fastest and most straightforward way to put some load on your services and get this so needed feedback quickly. We are going to start with Apache Bench (better known as ab), a tool for benchmarking HTTP-based services and applications.

For example, the same scenario for the **Reservation Service** we have seen in the previous sections could be load tested using ab, assuming the security token has been obtained before.

```
$ ab -c 5 -n 1000 -H "Authorization: Bearer $TOKEN" -T "application/json" -p reservation- ↩
    payload.json  https://localhost:17000/reservations

This is ApacheBench, Version 2.3
Copyright 1996 Adam Twiss, Zeus Technology Ltd, https://www.zeustech.net/
Licensed to The Apache Software Foundation, https://www.apache.org/

Benchmarking localhost (be patient)
...
Completed 1000 requests
Finished 1000 requests
```

```
Server Software:
Server Hostname:        localhost
Server Port:            17000

Document Path:          /reservations
Document Length:        0 bytes

Concurrency Level:      5
Time taken for tests:   22.785 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      487000 bytes
Total body sent:        1836000
HTML transferred:       0 bytes
Requests per second:    43.89 [#/sec] (mean)
Time per request:       113.925 [ms] (mean)
Time per request:       22.785 [ms] (mean, across all concurrent requests)
Transfer rate:          20.87 [Kbytes/sec] received
                        78.69 kb/s sent
                        99.56 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0       1
Processing:     6  113 449.4     27    4647
Waiting:        5  107 447.7     19    4645
Total:          6  114 449.4     28    4648

Percentage of the requests served within a certain time (ms)
  50%     28
  66%     52
  75%     57
  80%     62
  90%     83
  95%    326
  98%   1024
  99%   2885
 100%   4648 (longest request)
```

When the simplicity of ab becomes a show stopper, you may look at wrk, a modern HTTP benchmarking tool. It has power scripting support, baked by Lua, and is capable of simulating the complex load scenarios.

```
$ wrk -s reservation.lua -d60s -c50 -t5 --latency -H "Authorization: Bearer $TOKEN" https ↩
    ://localhost:17000/reservations

Running 1m test @ https://localhost:17000/reservations
  5 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   651.87ms   93.89ms   1.73s    85.20%
    Req/Sec    16.94     10.00    60.00     71.02%
  Latency Distribution
     50%  627.14ms
     75%  696.23ms
     90%  740.52ms
     99%    1.02s
  4579 requests in 1.00m, 2.04MB read
Requests/sec:     76.21
Transfer/sec:     34.83KB
```

If scripting is not something you are willing to use, there is another great option certainly worth mentioning, vegeta, a HTTP load testing tool (and library). It has enormous amount of features and even includes out of the box plotting.

```
$ echo "POST https://localhost:17000/reservations" | vegeta attack -duration=60s -rate=20 - ↩
    header="Authorization: Bearer $TOKEN" -header="Content-Type: application/json" -body  ↩
    reservation-payload.json > results.bin
```

Once the corresponding load test results are stored (in our case, in the file called **results.bin**), they could be easily converted into textual report:

```
$ cat results.bin | vegeta report

Requests      [total, rate]            1200, 20.01
Duration      [total, attack, wait]    59.9714976s, 59.9617223s, 9.7753ms
Latencies     [mean, 50, 95, 99, max]  26.286524ms, 9.424435ms, 104.754362ms, 416.680833ms, ↩
     846.8242ms
Bytes In      [total, mean]            0, 0.00
Bytes Out     [total, mean]            174000, 145.00
Success       [ratio]                  100.00%
Status Codes  [code:count]             201:1200
Error Set:
```

Or just converted into the graphical chart representation:

```
$ cat results.bin | vegeta plot
```

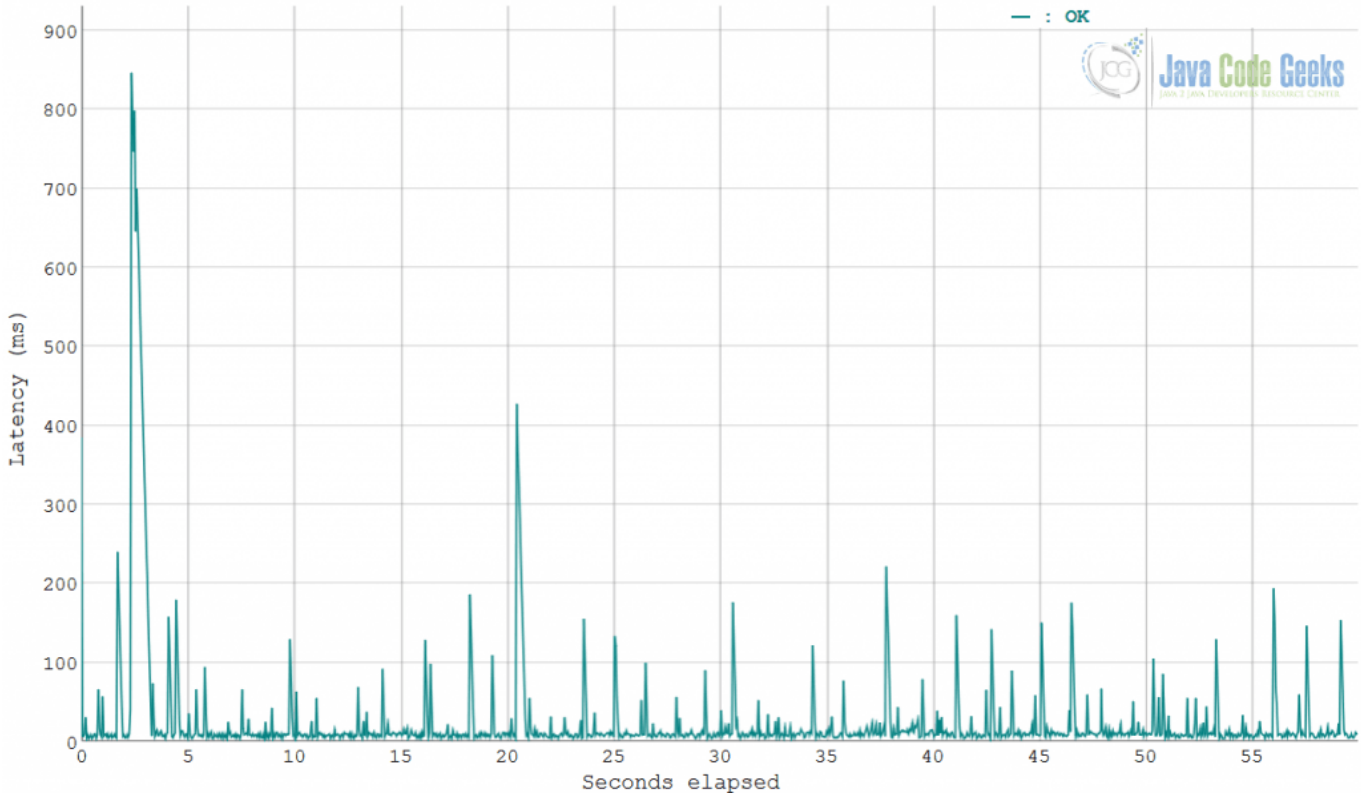


Figure 9.3: Image

As we have seen, each of these command line tools fits to the different needs you may have in mind for a particular load or performance scenario. Although there are many others out there, those three are pretty safe choices to pick from.

9.7 What about gRPC? HTTP/2? TCP?

All of the tools we have talked about so far support performance testing of the HTTP-based web services and APIs from the get-go. But what about stressing the services which rely on gRPC, HTTP/2 or even plain old UDP protocols?

Although there is no magic Swiss Army knife kind of tool yet, there are certainly some options. For example, gatling.io[Gatling] has HTTP/2 support built-in since the `3.0.0` release, whereas gRPC and UDP are supported by community extensions. From the other side, vegeta has HTTP/2 support whereas Apache JMeter has SMTP, FTP and TCP support.

Digging into specifics, there is official gRPC benchmarking guide which summarizes the performance benchmarking tools, the scenarios considered by the tests, and the testing infrastructure for gRPC-based services.

9.8 More Tools Around Us

Beside the tools and frameworks we have discussed up to now, it worth mentioning a few other choices which are great but might not be native for the Java developers. The first one is Locust, an easy-to-use, distributed, scalable load testing framework written in Python. The second one is Tsung, an open-source multi-protocol distributed load testing tool written in Erlang.

One of the promising projects to watch for is Test Armada, a fleet of tools empowering developers to implement quality automation at scale, which is also on track to introduce the support of the performance testing (based off Apache JMeter).

And it will be unfair to finish up without talking about Grinder, one of the earliest Java load testing framework that makes it easy to run a distributed test using many load injector machines. Unfortunately, the project seems to be dead, without any signs of the development for the last few years.

9.9 Performance and Load Testing - Conclusions

In this part of the tutorial we have talked about the tools, techniques and frameworks for performance and load testing, specifically in the context of the JVM platform. It is very important to take enough time, set the goals upfront and design the realistic performance and load test scenarios. This is a discipline by itself, but most importantly, the outcomes of these simulations could guide the service owners to properly shape out many SLA aspects we discussed before.

9.10 What's next

In the next section if the tutorial we are going to wrap up the discussion related to testing the microservices by focusing on the tooling around the security testing.

The samples and sources for this section are available for download here.

Chapter 10

Security Testing and Scanning

10.1 Introduction

This part of the tutorial, which is dedicated to the security testing, is going to wrap up the discussions around testing strategies proven to be invaluable in the world of software development (microservices included). Although the security aspects in the software projects become more and more important every single day, it is astonishing to consider how many companies neglect security practices altogether. At least once a month you hear about a new major vulnerability or data breach disclosure. Most of them could be prevented way before reaching the production!

The inspiration for this part of the tutorial mostly comes out of the Open Web Application Security Project (shortly, OWASP), a worldwide not-for-profit charitable organization focused on improving the security of software. It is one of the best and up-to-date resources on software security, available free of charge. You may recall that some of the OWASP tooling we have seen already along the tutorial.

10.2 Security Risks

Security is a very, very broad topic. So what kind of security risks attribute to the microservice architecture? One of the OWASP initiatives is to maintain the Top 10 Application Security Risks, a list of the most widely discovered and exploited security flaws in the applications, primarily web ones. Although the last version is dated **2017**, most risks (if not all of them) are still relevant even these days.

For an average developer, it is very difficult to be aware of all possible security flaws the applications may exhibit. Even more difficult is to uncover and mitigate these flaws without expertise, dedicated tooling or/and automation. Having the security experts on the team is probably the best investment but it is surprisingly difficult to find good ones. With that, the tooling aspect is exactly what we are going to be focusing on, narrowing the discussion only to the open-sourced solutions.

10.3 From the Bottom

Security should be a comprehensive measure, not an afterthought. It is equally as important to follow the secure coding practices as to secure the infrastructure. Like in the construction industry, it is absolutely necessary to start from a solid foundation.

There are a couple of tools which perform the security audit of the Java code bases. The most widely known one is the Find Security Bugs, the SpotBugs plugin for security audits of Java web applications which relies on static code analysis. Besides the IDE integrations, there are dedicated plugins for Apache Maven and Gradle so the analysis could be baked right into the build process and automated.

Let us take a look on Find Security Bugs usage. Since most of the **JCG Car Rentals** microservices are built using Apache Maven, the SpotBugs and Find Security Bugs are among the mandatory set of plugins.

```
<plugin>
    <groupId>com.github.spotbugs</groupId>
    <artifactId>spotbugs-maven-plugin</artifactId>
    <version>3.1.11</version>
    <configuration>
        <effort>Max</effort>
        <threshold>Low</threshold>
        <failOnError>true</failOnError>
        <plugins>
            <plugin>
                <groupId>com.h3xstream.findsecbugs</groupId>
                <artifactId>findsecbugs-plugin</artifactId>
                <version>LATEST</version>
            </plugin>
        </plugins>
    </configuration>
    s<executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
            <phase>verify</phase>
        </execution>
    </executions>
</plugin>
```

By default, the build is going to fail in case any issues have been discovered during the analysis (but the configuration is really flexible in this regard). Specifically for Find Security Bugs there is also an SBT integration (for Scala-based projects) although it looks abandoned.

To run a bit ahead, if you are employing a continuous code quality solution, like for example SonarQube (which we are going to talk about later in the tutorial), you will benefit from the code security audits as part of the quality checks pipeline.

10.4 Zed Attack Proxy

Leaving the static code analysis behind, the next tool we are going to look at is Zed Attack Proxy, widely known simply as ZAP.

The OWASP Zed Attack Proxy (ZAP) is one of the world's most popular free security tools and is actively maintained by hundreds of international volunteers. It can help you automatically find security vulnerabilities in your web applications while you are developing and testing your applications. It's also a great tool for experienced pentesters to use for manual security testing. - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

There are several modes which ZAP could be exploited. The simplest one is just to run the active scan against the URL where the web frontend is hosted. But to get most out of ZAP, it is recommended to configure it as a man-in-the-middle proxy.

Besides that, what is interesting about ZAP is the fact it could be used to find the vulnerabilities by scanning web services and APIs, using their OpenAPI or SOAP contracts. Unfortunately, ZAP does not support OpenAPI v3.x yet but the issue is opened and hopefully is going to be fixed at some point.

Out of all **JCG Car Rentals** microservices only **Reservation Service** uses the older OpenAPI specification which ZAP understands and is able to perform the scan against. Assuming the valid access token is obtained from the Keycloak, let us run our first ZAP API scan.

```
$ docker run -t owasp/zap2docker-weekly zap-api-scan.py
    -z "-config replacer.full_list(0).description=keycloak
        -config replacer.full_list(0).enabled=true
        -config replacer.full_list(0).matchtype=REQ_HEADER
        -config replacer.full_list(0).matchstr=Authorization
        -config replacer.full_list(0).regex=false
        -config replacer.full_list(0).replacement=Bearer\\ $TOKEN"
```

```
    -t https://host.docker.internal:18900/v2/api-docs
    -f openapi -a

...

Total of 15 URLs
PASS: Directory Browsing [0]
PASS: In Page Banner Information Leak [10009]
PASS: Cookie No HttpOnly Flag [10010]
PASS: Cookie Without Secure Flag [10011]
PASS: Incomplete or No Cache-control and Pragma HTTP Header Set [10015]
PASS: Web Browser XSS Protection Not Enabled [10016]
PASS: Cross-Domain JavaScript Source File Inclusion [10017]
PASS: Content-Type Header Missing [10019]
PASS: X-Frame-Options Header Scanner [10020]
PASS: X-Content-Type-Options Header Missing [10021]
PASS: Information Disclosure - Debug Error Messages [10023]
PASS: Information Disclosure - Sensitive Information in URL [10024]
PASS: Information Disclosure - Sensitive Information in HTTP Referrer Header [10025]

...

PASS: Cross Site Scripting (Persistent) [40014]
PASS: Cross Site Scripting (Persistent) - Prime [40016]
PASS: Cross Site Scripting (Persistent) - Spider [40017]
PASS: SQL Injection [40018]
PASS: SQL Injection - MySQL [40019]
PASS: SQL Injection - Hypersonic SQL [40020]
PASS: SQL Injection - Oracle [40021]
PASS: SQL Injection - PostgreSQL [40022]
PASS: Possible Username Enumeration [40023]
PASS: Source Code Disclosure - SVN [42]
PASS: Script Active Scan Rules [50000]
PASS: Script Passive Scan Rules [50001]
PASS: Path Traversal [6]
PASS: Remote File Inclusion [7]

...

FAIL-NEW: 0   FAIL-INPROG: 0  WARN-NEW: 1   WARN-INPROG: 0  INFO: 0 IGNORE: 0   PASS: 97
```

As the report says, no major issues have been discovered. It is worth to note that ZAP project is very automation-friendly and provides a convenient set of the scripts and Docker images along with dedicated Jenkins plugin.

10.5 Archery

Moving forward, let us spend some time and look at Archery, basically a suite of the different tools (including Zed Attack Proxy by the way) to perform the comprehensive security analysis.

Archery is an opensource vulnerability assessment and management tool which helps developers and pentesters to perform scans and manage vulnerabilities. Archery uses popular opensource tools to perform comprehensive scanning for web application and network. - https://github.com/archerysec/archerysec

The simplest way to get started with Archery is to use prebuilt Docker container image (but in this case the integrations with other tools would need to be done manually):

```
$ docker run -it -p 8000:8000 archerysec/archerysec:latest
```

Arguably the better way to have Archery up and running in Docker is to use the Docker Compose with the deployment blueprint provided. It bundles all the tooling and wires it with Archery.

Although the typical way to interface with Archery is through its web UI, it also has a RESTful web APIs for automation purposes and could be integrated into CI/ CD pipelines. The management part of the Archery feature set includes integration with JIRA for ticket management.

Please notice nonetheless the project is still in development phase, it has been showing quite promising adoption, certainly worth keeping an eye on.

10.6 XSStrike

Cross-Site Scripting (XSS) is steadily one of the most exploited vulnerabilities in the modern web applications (and is the second most prevalent issue in the OWASP Top 10, found in around two thirds of the applications). Since the **JCG Car Rentals** platform has a public web frontend, the XSS is the real issue to take care of and the tools like XSStrike are enormously helpful in detecting it.

XSStrike is a Cross Site Scripting detection suite equipped with four hand written parsers, an intelligent payload generator, a powerful fuzzing engine and an incredibly fast crawler. - https://github.com/s0md3v/XSStrike

The XSStrike is written in Python so you would need the **3.7.x** release to be installed in advance. Sadly, the XSStrike does not play well with the single-page web applications (like **JCG Web Portal** for example, which is based on Vue.js). But still, we could benefit from running it against **JCG Admin Web Portal** instead.

```
$ python3 xsstrike.py -u https://localhost:19900/portal?search=bmw
       XSStrike v3.1.2

[~] Checking for DOM vulnerabilities
[+] WAF Status: Offline
[!] Testing parameter: search
[!] Reflections found: 1
[~] Analysing reflections
[~] Generating payloads
[-] No vectors were crafted.
```

It turned out to be not very helpful for **JCG Car Rentals** web frontends but let this fact not discourage you from giving XSStrike a try.

10.7 Vulas

Just a few weeks ago SAP had open-sourced the Vulnerability Assessment Tool (Vulas), composed from several independent microservices, that it has been used to perform 20K+ scans of more than 600+ Java development projects.

The open-source vulnerability assessment tool supports software development organizations in regards to the secure use of open-source components during application development. The tool analyzes Java and Python applications ... - https://github.com/-SAP/vulnerability-assessment-tool

The Vulas tool is targeting one of the OWASP Top 10 security threats, more specifically using components with known vulnerabilities. It is powered by vulnerability assessment knowledge base, also open-sourced by SAP, which basically aggregates public information about the security vulnerabilities in open source projects.

Once Vulas is deployed (using Docker is probably the easiest way to get up to speed) and vulnerabilities database is filled in, you may use Apache Maven plugin, Gradle plugin or just plain command line tooling to integrate the scanning into Java-based applications.

To illustrate how useful Vulas could be, let us take a look on the sample vulnerabilities discovered during the audit of the **Customer Service** microservice, one of key components of the **JCG Car Rentals** platform.

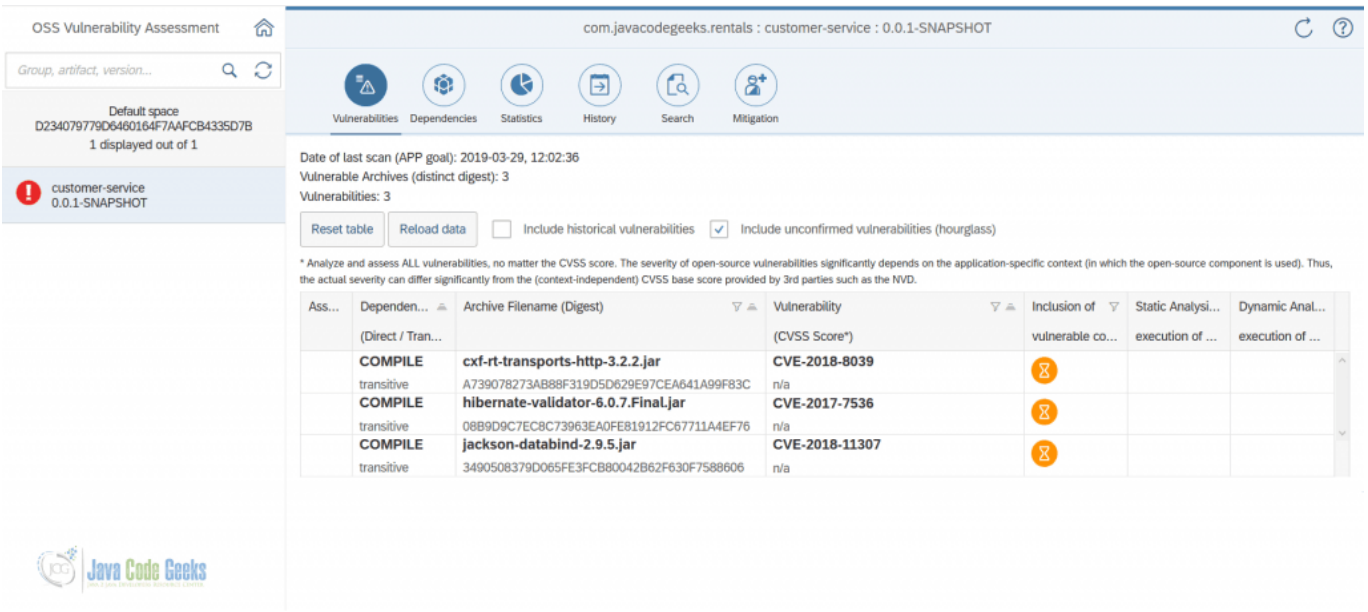


Figure 10.1: Image

Although the Vulas web UI is quite basic, the amount of the details presented along with each uncovered vulnerability is just amazing. Functionally, it is somewhat similar to the OWASP dependency-check we have talked about in the previous part of the tutorial.

10.8 Another Vulnerability Auditor

AVA , or Another Vulnerability Auditor in full, is a pretty recent open-source contribution from the Indeed[Indeed] security team.

AVA is a web scanner designed for use within automated systems. It accepts endpoints via HAR-formatted files and scans each request with a set of checks and auditors. The checks determine the vulnerabilities to check, such as Cross-Site Scripting or Open Redirect. The auditors determine the HTTP elements to audit, such as parameters or cookies. - https://github.com/-indeedsecurity/ava

Similarly to the XSStrike, it is also Python-based and is quite easy to install. Let us use AVA to perform the XSS audit for **JCG Admin Web Portal** .

```
$ ava -a parameter -e xss vectors.har

2019-03-27 01:56:38Z : INFO : Loading vectors.
2019-03-27 01:56:38Z : INFO : Loading scanner.
2019-03-27 01:56:41Z : INFO : Found 0 issues in 0:00:02.
```

The results are promising, no issues have been discovered.

10.9 Orchestration

The tremendous popularity of the orchestration solutions and service meshes could give a false impression that you would get the secure infrastructure with zero efforts. In reality, there are a lot of things to take care of and the tools like kubeaudit from Shopify may be of great help here.

10.10 Cloud

Secure applications deployed into poorly secured environments may not get you too far. The things go even wilder by including the cloud computing into equation. How would you ensure that your configuration is hardened properly? How to catch the potential security flaws? And how to scale that across multiple cloud providers, when each one has own vision on cloud security? Netflix has faced these challenges early on and made the contribution to the community by open-sourcing the Security Monkey project.

Security Monkey monitors your AWS and GCP accounts for policy changes and alerts on insecure configurations. Support is available for OpenStack public and private clouds. Security Monkey can also watch and monitor your GitHub organizations, teams, and repositories. - https://github.com/Netflix/security_monkey

There are also many other open-source projects for continuous auditing the cloud deployments, tailored for a specific cloud provider. Please make sure you are covered there.

10.11 Conclusions

In this section of the tutorial we have talked about security testing. The discussion revolved around three main subjects: static code analysis, auditing vulnerable components and scanning the instances of the web applications and APIs. This is great start but certainly not enough.

Complex distributed systems, like microservices, have a very wide surface area to attack. Hiring security experts and making them the part of your team could greatly reduce the risks of being hacked or unintentionally leak sensitive data.

One of the interesting initiatives with respect to Java ecosystem is the establishment of the Central Security Project to serve as one-stop place for the security community to report security issues found in open source Apache Maven components.

10.12 What's next

This part wraps up the testing subject. In the next part of the tutorial we are going to switch over to continuous delivery and continuous integration.

Chapter 11

Continuous Integration and Continuous Delivery

11.1 Introduction

If we look back at the number of challenges associated with the microservice architecture, ensuring that every single microservice is able to speak the right language with each of its peer is probably one of the most difficult ones. We have talked a lot about testing lately but there is always opportunity for bugs to sneak in. Maybe it is last minute changes in the contracts? Or maybe it is that security requirements have been hardened? And what about unintentionally pushing the improper configuration?

In order to address these concerns, along with many others, we are going to have a conversation about the practices of continuous integration and continuous delivery. So what are these practices, how they are helpful and what are the differences between them?

The continuous integration paradigm advocates for pushing your changes to the mainstream source repository as often as possible paired with running the complete build and executing the suite of the automated tests and checks. The goal here is to keep the builds rolling and tests passing all the time, avoiding the scenario when everyone tries to merge the changes at the last moment, dragging the project into the integration hell.

The continuous delivery practice lifts the continuous integration to the next level by bringing in the release process automation and ensuring that the projects are ready to be released at any time. Surprisingly, not many organizations understand the importance of continuous delivery, but this practice is an absolutely necessary prerequisite in order to follow the principles of the microservice architecture.

Fairly speaking, continuous delivery is not the end of it. The continuous deployment process closes the loop by introducing the support of the automated release deployments, right into the live system. The presence of the continuous deployment is an indicator of mature development organization.

11.2 Jenkins

For many, the term continuous integration immediately rings Jenkins to mind. Indeed, it is probably one of the most widely deployed continuous integration (and continuous delivery) platforms, particularly in the JVM ecosystem.

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. - https://jenkins.io/doc/

Jenkins has an interesting story which essentially spawns two radically different camps: the ones who hate it and the ones who love it. Luckily, the release of Jenkins version 2.0 a few years ago was a true game changer which sprawled out the tsunami of innovations.

To illustrate the power of Jenkins, let us take a look at how **JCG Car Rentals** platform is using pipelines to continuously build and test its microservice projects.

Figure 11.1: Image

The true gems of Jenkins are its extensibility and enormous amount of the community plugins available. As you may remember, all **JCG Car Rentals** projects integrate with SpotBugs for static code analysis and OWASP dependency-check for catching the vulnerable dependencies. Unsurprisingly, Jenkins has plugins for both, which are easily injectable into the build pipeline, like the one **Customer Service** has.

```
pipeline {
   agent any

   options {
       disableConcurrentBuilds()
       buildDiscarder(logRotator(numToKeepStr:'5'))
   }

   triggers {
       pollSCM('H/15 * * * *')
   }

   tools {
       jdk "jdk-8u202"
   }

   stages {
       stage('Cleanup before build') {
           steps {
                cleanWs()
           }
       }

       stage('Checkout from SCM') {
            steps {
                checkout scm
            }
       }

       stage('Build') {
           steps {
               withMaven(maven: 'mvn-3.6.0') {
                   sh "mvn clean package"
               }
           }
        }

        stage('Spotbugs Check') {
           steps {
               withMaven(maven: 'mvn-3.6.0') {
                    sh "mvn spotbugs:spotbugs"
               }

               script {
```

```
                    def spotbugs = scanForIssues tool: [$class: 'SpotBugs'], pattern: '**/ ↩
                        target/spotbugsXml.xml'
                    publishIssues issues:[spotbugs]
                }
            }
        }

        stage('OWASP Dependency Check') {
            steps {
                dependencyCheckAnalyzer datadir: '', hintsFile: '', includeCsvReports: false ↩
                    , includeHtmlReports: true, includeJsonReports: false, includeVulnReports ↩
                    : false, isAutoupdateDisabled: false, outdir: '', scanpath: '',  ↩
                    skipOnScmChange: false, skipOnUpstreamChange: false, suppressionFile: '', ↩
                     zipExtensions: ''
                dependencyCheckPublisher canComputeNew: false, defaultEncoding: '', healthy: ↩
                     '', pattern: '', unHealthy: ''
            }
        }
    }

    post {
         always {
             archiveArtifacts artifacts: 'target/*.jar', fingerprint: true
             archiveArtifacts artifacts: '**/dependency-check-report.xml', onlyIfSuccessful ↩
                 : true
             archiveArtifacts artifacts: '**/spotbugsXml.xml', onlyIfSuccessful: true
         }
    }
}
```

Once the pipeline job is triggered on Jenkins, the SpotBugs and OWASP dependency-check reports are published as part of the build results.

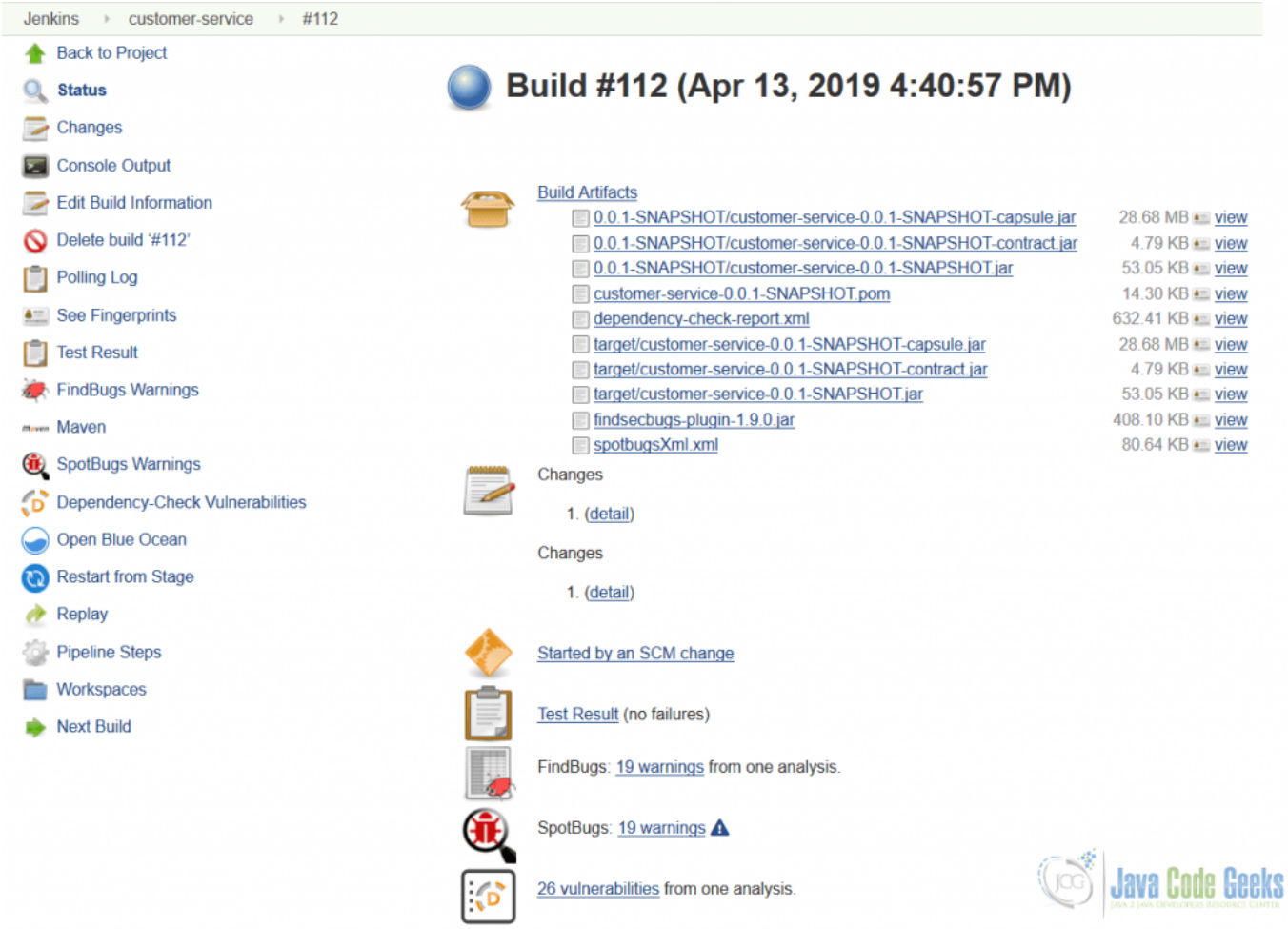


Figure 11.2: Image

It is critically important to stay disciplined and to follow the principles of the continuous integration. The builds should be kept healthy and passing all the time.

There are a lot of things to say about Jenkins, particularly with respect to its integration with Docker but let us better glance over other options.

11.3 SonarQube

We have talked about SonarQube along previous parts of the tutorial. To be fair, it does not fit into continuous integration or continuous delivery bucket but rather forms a complementary one, a continuous code quality inspection.

SonarQube is an open source platform to perform automatic reviews with static analysis of code to detect bugs, code smells and security vulnerabilities on 25+ programming languages including Java, C#, JavaScript, TypeScript, C/C++, COBOL and more. ... - https://www.sonarqube.org/about/

The results of the SonarQube code qualify inspections are of immense value. To get a glimpse of them, let us take a look at the **Customer Service** code quality dashboard.

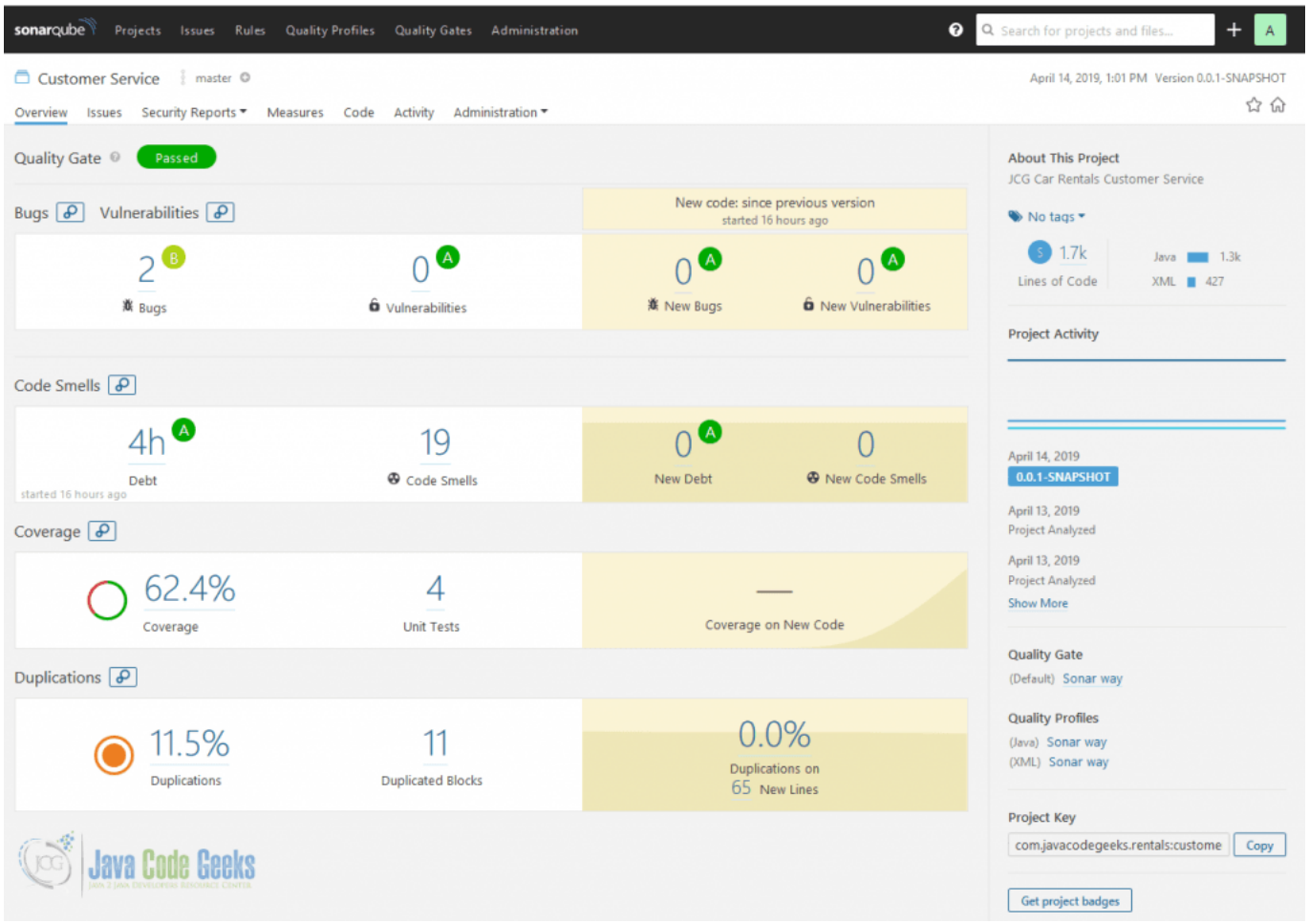


Figure 11.3: Image

As you may see, there is some intersection with the reports generated by SpotBugs and OWASP dependency-check, however SonarQube checks are much broader in scope. But how hard it is to make SonarQube a part of your continuous integration pipelines? As easy as it could possibly get since SonarQube has outstanding integrations with Apache Maven, Gradle and even Jenkins.

With over 25 programming languages supported, SonarQube would certainly help you to raise the bar of code quality and maintainability across a whole microservices fleet (even if there may be no out of the box integration with the continuous integration platform of your choice).

11.4 Bazel

Bazel came out of Google as a flavor of the tool used to build company's server software internally. It is not designated to serve as continuous integration backbone but the build tool behind it.

Bazel is an open-source build and test tool similar to Make, Maven, and Gradle. It uses a human-readable, high-level build language. Bazel supports projects in multiple languages and builds outputs for multiple platforms. Bazel supports large codebases across multiple repositories, and large numbers of users. - https://docs.bazel.build/versions/master/bazel-overview.html

What is interesting about Bazel is its focus on faster builds (advanced local and distributed caching, optimized dependency analysis and parallel execution), scalability (handles codebases of any size, across many repositories or a huge monorepo) and support of the multiple languages (Java included). In the world of polyglot microservices, having the same build tooling experience may be quite advantageous.

11.5 Buildbot

In essence, Buildbot is a job scheduling system which supports distributed, parallel execution of jobs across multiple platforms, flexible integration with different source control systems and extensive job status reporting.

Buildbot is an open-source framework for automating software build, test, and release processes. - https://buildbot.net/

Buildbot fits well to serve the needs of the mixed language applications (like polyglot microservices). It is written in Python and extensively relies on Python scripts for configuration tasks.

11.6 Concourse CI

Concourse takes a generalized approach to the automation which makes it a good fit for backing continuous integration and continuous delivery, in particular.

Concourse is an open-source continuous thing-doer. - https://concourse-ci.org/

Everything in Concourse runs in a container, it is very easy to get started with and its core design principles are encouraging to use the declarative pipelines (which, frankly speaking, are quite different from Jenkins or GoCD ones). For example, the basic pipeline for the **Customer Service** may look like that:

```
resources:
  - name: customer-service
    type: git
    source:
      uri:
      branch: master
jobs:
  - name: build
    plan:
    - get: customer-service
      trigger: true
    - task: compile
      config:
        platform: linux
        image_resource:
          type: docker-image
          source:
            repository: maven
        inputs:
          - name: customer-service
        outputs:
          - name: build
        caches:
          - path: customer-service/.m2
        run:
          path: sh
          args:
          - -c
          - mvn -f customer-service/pom.xml package -Dmaven.repo.local=customer-service/.m2
```

Also, Concourse comes with command line tooling and pretty basic web UI which is nonetheless helpful in visualizing your pipelines, like at the picture below.

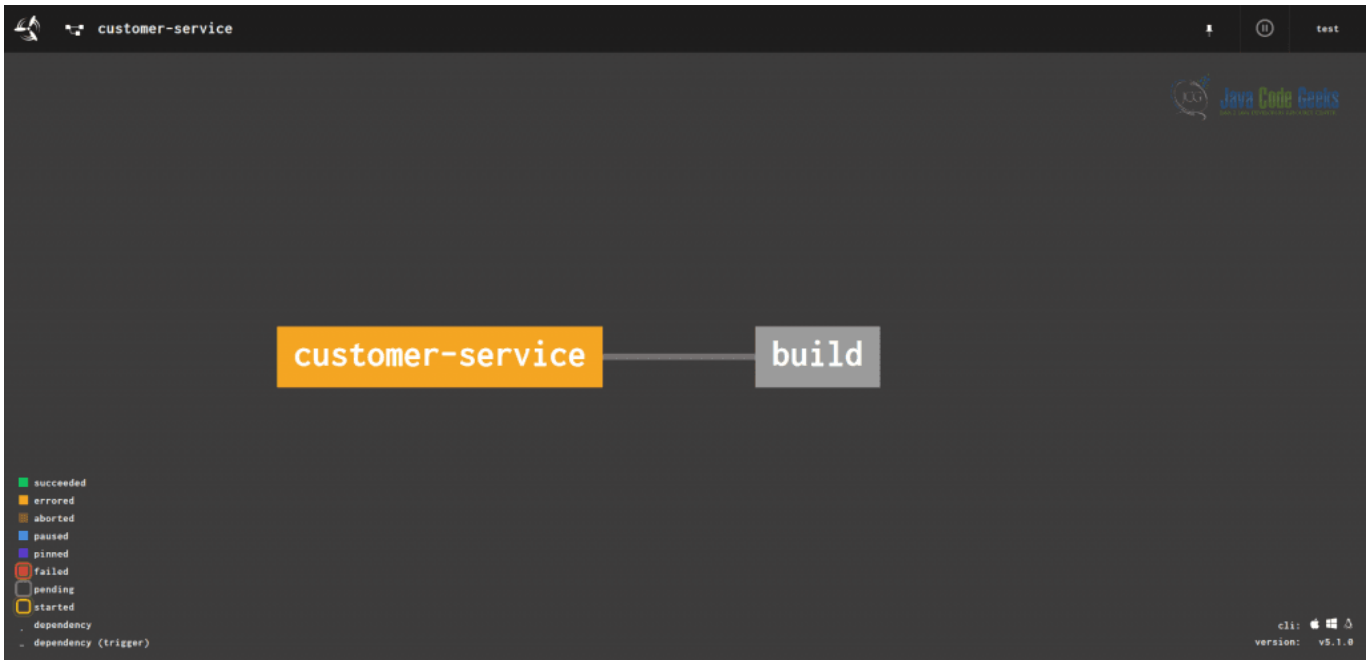


Figure 11.4: Image

11.7 Gitlab

Gitlab is a full-fledged open source end-to-end software development platform with built-in version control, issue tracking, code review, continuous integration and continuous delivery.

GitLab is a single application for the entire software development lifecycle. From project planning and source code management to CI/CD, monitoring, and security. - https://about.gitlab.com/

If you are looking for all-in-one solution, which could be either self-hosted or managed in the cloud, Gitlab is certainly an option to consider. Let us take a look at the **Customer Service** project development in case of Gitlab being chosen.

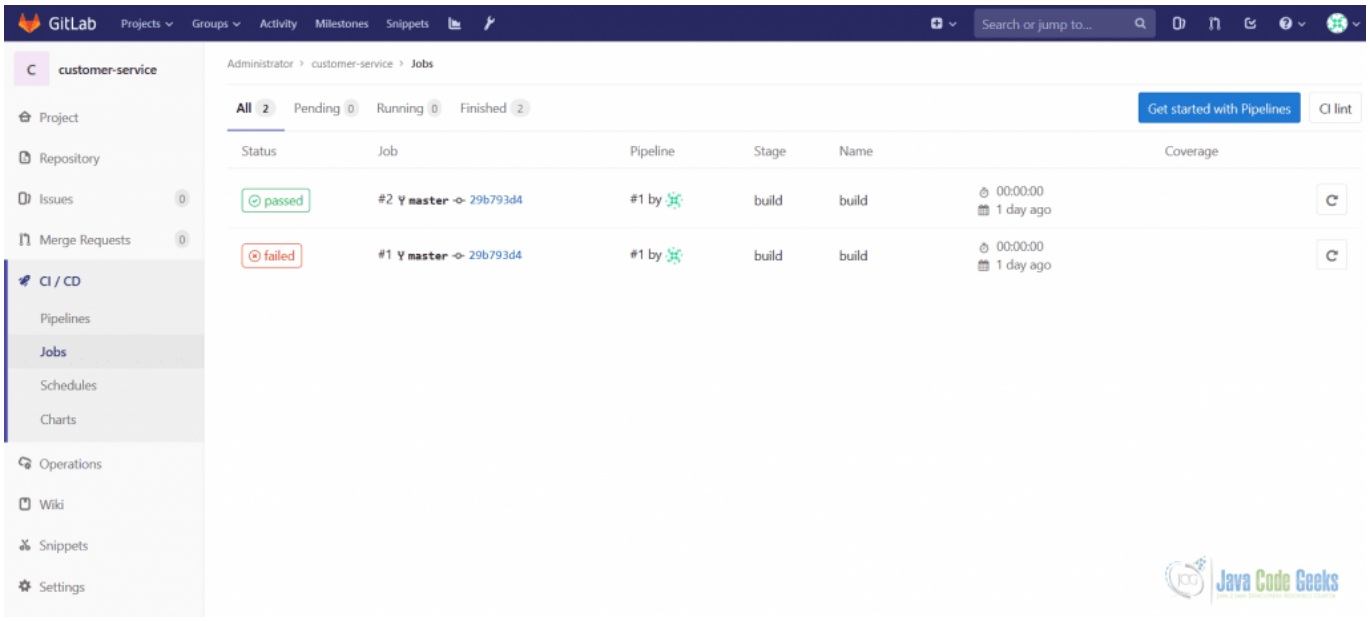


Figure 11.5: Image

The continuous integration and continuous delivery pipelines are pretty extensible and by default include at least 3 stages: build, test and code quality.

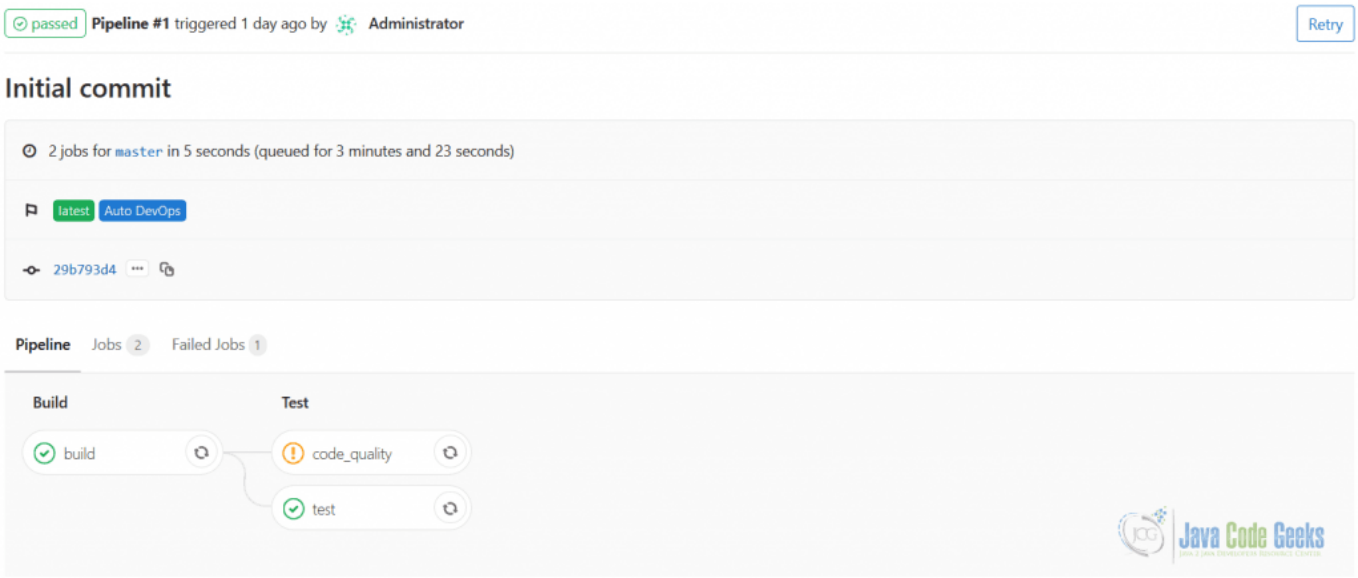


Figure 11.6: Image

It is worth noting that Gitlab is being used by quite a large number of companies and its popularity and adoption are steadily growing.

11.8 GoCD

The next subject we are going to talk about, GoCD, came out of ThoughtWorks, the organization widely known for employing the world-class experts in mostly every area of software development.

GoCD is an open source build and release tool from ThoughtWorks. GoCD supports modern infrastructure and helps enterprise businesses get software delivered faster, safer, and more reliably. - https://www.gocd.org/

Unsurprisingly, pipelines are central piece in GoCD as well. They serve as the representation of a workflow or a part of a workflow. The web UI GoCD comes with is quite intuitive, simple and easy to use. The **Customer Service** pipeline in the image below is a good demonstration of that.

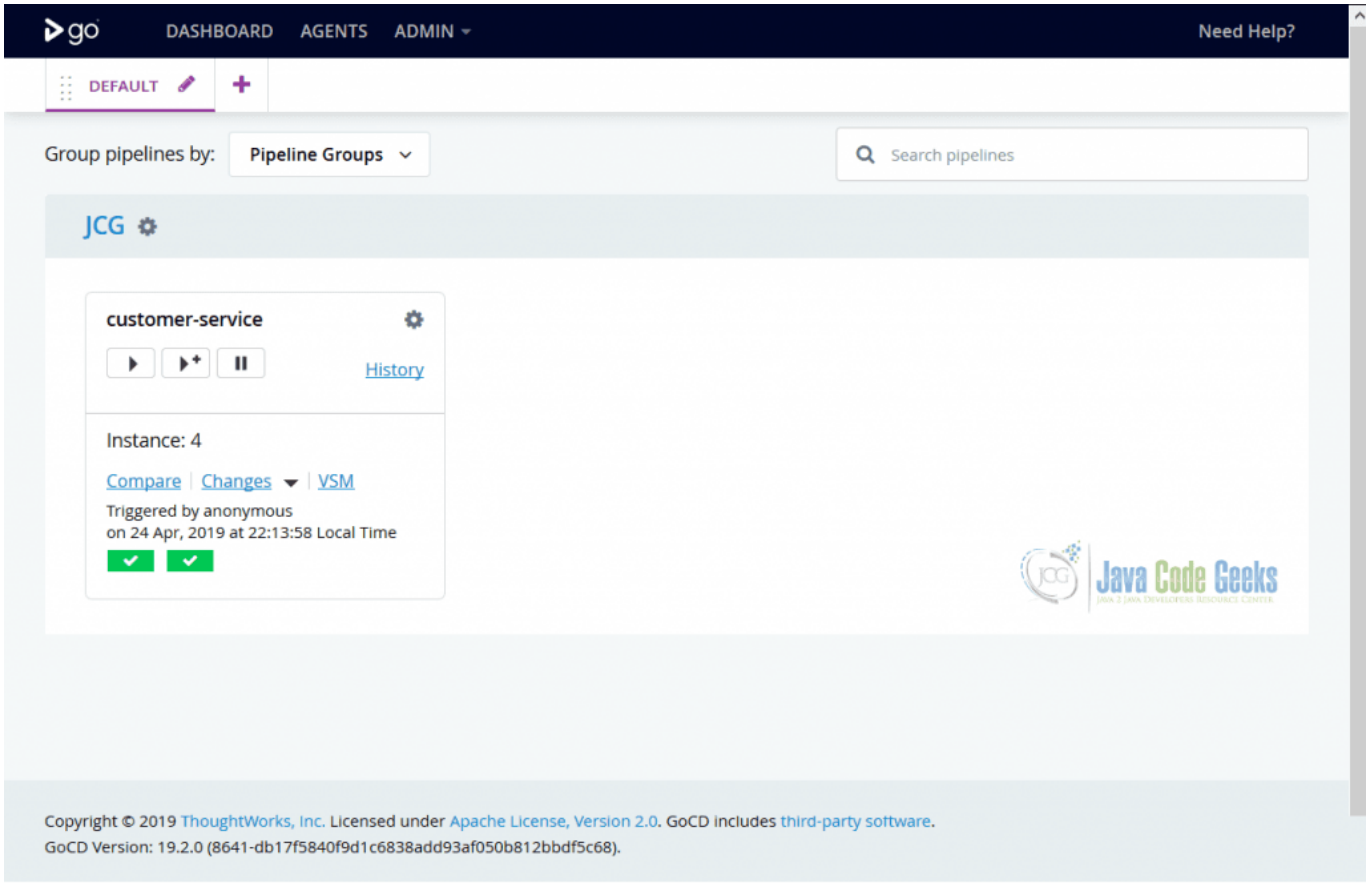


Figure 11.7: Image

The pipeline itself may include an arbitrary amount of stages, for example the **Customer Service** 's one has two stages configured, **Build** and **Test** . The dedicated view shows off the execution of the each stage in great details.

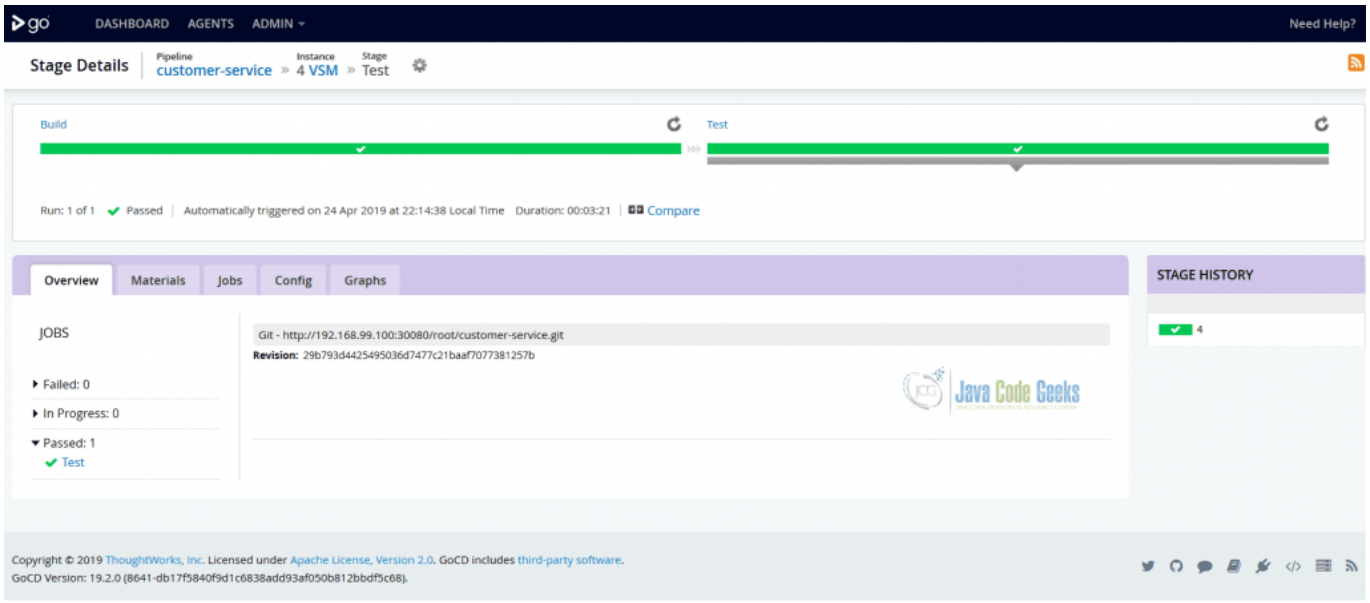


Figure 11.8: Image

The GoCD pipelines are very generic and not biased towards any programming language or development platform, as such addressing the needs of the polyglot microservice projects.

11.9 CircleCI

If the self-hosted (or to say it differently, on-premise) solutions are not aligned with your plans, there are quite a few SaaS offerings around. The CircleCI is one of the popular choices.

CircleCI's continuous integration and delivery platform makes it easy for teams of all sizes to rapidly build and release quality software at scale. Build for Linux, macOS, and Android, in the cloud or behind your firewall. - https://circleci.com/

Besides being a great product, one of the reasons the CircleCI is included in our list is the presence of the free tier to let you get started quickly.

11.10 TravisCI

TravisCI falls into the same bucket of the SaaS offerings as CircleCI but with the one important difference - it is always free for open source projects.

Travis CI is a hosted continuous integration and deployment system. - https://github.com/travis-ci/travis-ci

TravisCI is probably the most popular continuous integration service used to build and test software projects hosted on GitHub. On the not so bright side, the future of TravisCI is unclear since it was acquired in January 2019 by private equity firm and reportedly the original development team was let go.

11.11 CodeShip

CodeShip is yet another SaaS for doing continuous integration and continuous delivery, acquired by CloudBees recently, which also has a free plan available.

Codeship is a fast and secure hosted Continuous Integration service that scales with your needs. It supports GitHub, Bitbucket, and Gitlab projects. - https://cms.codeship.com/

One of the distinguishing advantages of the CodeShip is that it takes literally no (or little) time to set it up and get going.

11.12 Spinnaker

Most of the options we discussed so far are trying to cover the continuous integration and continuous delivery under the same umbrella. On the other hand, the Spinnaker, originally created at Netflix, is focusing purely on continuous delivery side of things.

Spinnaker is an open source, multi-cloud continuous delivery platform for releasing software changes with high velocity and confidence. - https://www.spinnaker.io/

It is truly unique solution which combines a flexible continuous delivery pipeline management with integrations to the leading cloud providers.

11.13 Cloud

Hosting your own continuous integration and continuous delivery infrastructure might be far beyond one's purse. The SaaS offerings we have talked about could significantly speed up the on-boarding and lower the upfront costs, at least when you just starting. However, if you are in the cloud (which is more than likely these days), it makes a lot of sense to benefit from the offerings the cloud provider has for you. Let us take a look at what leaders in the cloud computing came up with.

The first one in our list is for sure AWS, which has two offerings related to continuous integration and continuous delivery, AWS CodeBuild and AWS CodePipeline respectively.

AWS CodeBuild is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don't need to provision, manage, and scale your own build servers. ... - https://aws.amazon.com/codebuild/

AWS CodePipeline is a fully managed continuous delivery service that helps you automate your release pipelines for fast and reliable application and infrastructure updates. CodePipeline automates the build, test, and deploy phases of your release process every time there is a code change, based on the release model you define. ... - https://aws.amazon.com/codepipeline/

Moving on to the Google Cloud, we are going to stumble upon Cloud Build offering, the backbone of the Google Cloud continuous integration efforts.

Cloud Build lets you build software quickly across all languages. Get complete control over defining custom workflows for building, testing, and deploying across multiple environments ... - https://cloud.google.com/cloud-build/

The Microsoft Azure took another route and started with the managed Jenkins offering. But shortly after GitHub acquisition, the Azure DevOps has emerged, the completely new offering which spawns across continuous integration, continuous delivery and continuous deployment.

Azure DevOps Services provides development collaboration tools including high-performance pipelines, free private Git repositories, configurable Kanban boards, and extensive automated and continuous testing capabilities. .... - https://docs.microsoft.com/-en-ca/azure/devops/index?view=azure-devops

11.14 Cloud Native

Before wrapping up, it would be great to look on how existing continuous integration and continuous delivery solutions adapt to the constantly changing infrastructural and operational landscape. There is a lot of innovation happening in this area, let us just look through a few interesting developments.

To begin with, Jenkins has recently announced the new subproject, Jenkins X, to specifically target the Kubernetes-based deployments. I think this trend is going to continue and other players are going to catch up since the popularity of Kubernetes is skyrocketing.

The spread of the serverless execution model put the continuous integration and continuous delivery into the new perspective. In this regards, it worth to note LambCI, a continuous integration system built on AWS Lambda. It is very likely that we are going to see more options emerging on this front.

11.15 Conclusions

The importance of the continuous integration poses no questions these days. From the other side, the continuous delivery and continuous deployment are falling behind but they are undoubtedly the integral part of the microservice principles.

Along this section of the tutorial, we have glanced over quite a number of options but obviously there are many others in the wild. The emphasis is not on the particular solution though but the practices themselves. Since you embarked yourself on the microservices journey, it is also your responsibility to make it a smooth one.

11.16 What's next

In the next section of the tutorial we are going to dig more into operational concerns associated with the microservice architecture and talk about configuration management, service discovery and load balancing.

Chapter 12

Configuration, Service Discovery and Load Balancing

12.1 Configuration, Service Discovery and Load Balancing - Introduction

Slowly but steadily we are moving towards getting our microservices ready to be deployed in production. In this section of the tutorial we are going to talk about three main subjects: configuration, service discovery and load balancing.

Our goal is to understand the essential basic concepts rather to cover every option available. As we will see later in the tutorial, the configuration management, service discovery and load balancing are going to pop up over and over again, although in the different contexts and shapes.

12.2 Configuration

It is very likely that the configuration of each of your microservices is going to vary from environment to environment. It is perfectly fine but raises the question: how to tell the microservice in question what configuration to use?

Many frameworks offer different mechanisms to configuration management (like profiles, configuration files, command line options, ...) but the approach we are going to advocate here is to follow The Twelve-Factor App methodology (which we have touched upon already).

The twelve-factor app stores config in environment variables (often shortened to env vars or env). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard. - https://12factor.net/config

The environment variables exhibit only one major limitation: they are static in nature. Any change in their values may require the full microservice restart. It may not be an issue for many but it is often desirable to have some kind of flexibility to modify the service configuration at runtime.

12.2.1 Dynamic Configuration

Ability to update the configuration without restarting the service is a very appealing feature to have. But the price to pay is also high since it requires a descent amount of instrumentation and not too many frameworks or libraries offer such transparent support.

For example, let us think about changing the database JDBC URL connection string on the fly. Not only the underlying data-sources have to be transparently recreated, also the JDBC connection pools have to be drained and reinitialized as well.

The mechanics behind the dynamic configuration really depends on what kind of the configuration management approach you are using (Consul, Zookeeper, Spring Cloud Config, ...), however some frameworks, like Spring Cloud for example, take a lot of this burden away from the developers.

12.2.2 Feature Flags

The feature flags (or feature toggles) do not fall precisely into the configuration bucket but it is a very powerful technique to dynamically change the service or application characteristics. They are tremendously useful and widely adopted for A/B testing, new features roll out, introducing experimental functionality, just to name a few areas.

In the Java ecosystem, FF4J is probably the most popular implementation of the feature flags pattern. Another library is togglz[Togglz] however it is not actively maintained these days. If we go beyond just Java, it is worth looking at Unleash, an enterprise-ready feature toggles service. It has impressive list of SDKs available for many programming languages, including Java.

12.2.3 Spring Cloud Config

If your microservices are built on top of the Spring Platform, then Spring Cloud Config is the one of the most accessible configuration management options to start with. It provides both server-side and client-side support (the communication is based on HTTP protocol), is exceptionally easy to integrate with and even to embed into existing services.

Since the **JCG Car Rentals** platform needs the configuration management service, let us see how simple it is to configure one using Spring Cloud Config backed by Git.

```
server:
  port: 20100

spring:
  cloud:
    config:
      server:
        git:
          uri: file://${rentals.home}/rentals-config
  application:
    name: config-server
```

To run the embedded configuration server instance, the idiomatic Spring Boot annotation-driven approach is the way to go.

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

Although Spring Cloud Config has out of the box encryption and decryption support, you may also use it in conjunction with HashiCorp Vault to manage sensitive configuration properties and secrets.

12.2.4 Archaius

In the space of the general-purpose library for configuration management, probably Archaius from Netflix would be the best known one (in case of JVM platform). It does support dynamic configuration properties, complex composite configuration hierarchies, has native Scala support and could be used along with Zookeeper backend.

12.3 Service Discovery

One of the most challenging problems, incidental to microservices which use at least some form of direct communication, is how the peers discover each other. For example, in the context of the **JCG Car Rentals** platform the **Reservation Service** should know where the **Inventory Service** is. How to solve this particular challenge?

One may argue that it is feasible to pass the list of host/port pairs of all running **Inventory Service** instances to **Reservation Service** using the environment variables but this is not really sustainable. What if the **Inventory Service** was scaled up or down? Or what if some **Inventory Service** instances become inaccessible due to network hiccups or just crashed? This is truly a dynamic, operational data and should be treated as such.

To be fair, service discovery often goes side by side with cluster management and coordination. It is a very interesting but broad subject so our focus is going to gravitate towards service discovery side. There are many open source options available, ranging from quite low-level to full-fledged distributed coordinators, and we are going to glance over the most widely used ones.

12.3.1 JGroups

JGroups, one of the oldest of its kind, is a toolkit for reliable messaging which, among its many other features, serves as backbone for cluster management and membership detection. It is not the dedicated service discovery solution per se but could be used at the lowest level to implement one.

12.3.2 Atomix

In the same vein, Atomix framework provides capabilities for cluster management, communicating across nodes, asynchronous messaging, group membership, leader election, distributed concurrency control, partitioning, replication and state changes coordination in distributed systems. Fairly speaking, it is also not a direct service discovery solution, rather an enabler to have your own given that the framework has all the necessary pieces in place.

12.3.3 Eureka

Eureka, developed at Netflix, is a REST-based service that is dedicated to be primarily used for service discovery purposes (with an emphasis on AWS support). It is written purely in Java and includes server and client components.

It is really independent of any kind of framework. However, Spring Cloud Netflix provides outstanding integration of the Spring Boot applications and services with a number of Netflix components, including the abstractions over Eureka servers and clients. Let us take a look on how **JCG Car Rentals** platform may benefit from Eureka.

```
server:
  port: 20200

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    healthcheck:
      enabled: true
    serviceUrl:
      defaultZone: https://localhost:20201/eureka/
  server:
    enable-self-preservation: false
    wait-time-in-ms-when-sync-empty: 0
  instance:
    appname: eureka-server
    preferIpAddress: true
```

We could also profit from the seamless integration with Spring Cloud Config instead of hard-coding the configuration properties.

```
spring:
  application:
    name: eureka-server
  cloud:
    config:
      uri:
      - https://localhost:20100
```

Similarly to Spring Cloud Config example, running embedded Eureka server instance requires just a single annotated class.

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

On the services side, the Eureka client should be plugged in and configured to communicate with the Eureka server we just implemented. Since the **Reservation Service** is built on top of Spring Boot, the integration is really simple and concise, thanks to Spring Cloud Netflix.

```
eureka:
  instance:
    appname: reservation-service
    preferIpAddress: true
  client:
    register-with-eureka: true
    fetch-registry: true
    healthcheck:
      enabled: true
    service-url:
      defaultZone: https://localhost:20200/eureka/
```

For sure, picking these properties from Spring Cloud Config or similar configuration management solution would be preferable. When we run multiple instances of the **Reservation Service**, each will register itself with the Eureka service discovery, for example:

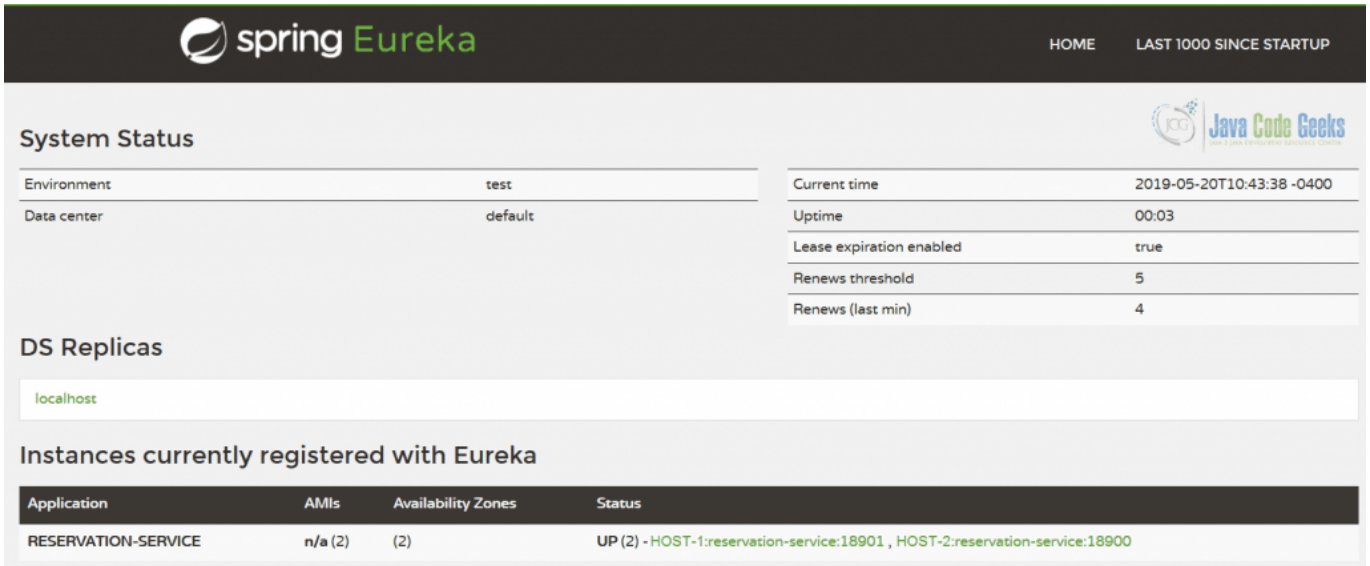


Figure 12.1: Image

In case you are looking for self-hosted service discovery, Eureka could be a very good option. We have not done anything sophisticated yet but Eureka has a lot of features and configuration parameters to tune.

12.3.4 Zookeeper

Apache ZooKeeper is a centralized, highly available service for managing configuration and distributed coordination. It is one

of the pioneers of the open source distributed coordinators, is battle-tested for years and serves as the reliable backbone for many other projects.

For JVM-based applications, the ecosystem of the client libraries to work with Apache ZooKeeper is pretty rich. Apache Curator, originally started at Netflix, provides high-level abstractions to make using Apache ZooKeeper much easier and more reliable. Importantly, Apache Curator also includes a set of recipes for common use cases and extensions such as service discovery.

Even more good news for Spring-based applications since the Spring Cloud Zookeeper is solely dedicated to provide Apache Zookeeper integrations for Spring Boot applications and services. Similarly to Apache Curator, it comes with the common patterns baked in, including service discovery and configuration.

12.3.5 Etcd

In the essence, etcd is a distributed, consistent and highly-available key value store. But don't let this simple definition to mislead you since etcd is often used as the backend for service discovery and configuration management.

12.3.6 Consul

Consul by HashiCorp has started as a distributed, highly available, and data center aware solution for service discovery and configuration. It is one of the first products which augmented service discovery to become a first-class citizen, not a recipe or a pattern. Consul's API is purely HTTP-based so there is no special client needed to start working with it. Nonetheless, there are a couple of dedicated JVM libraries which make integration with Consul even easier, including the Spring Cloud Consul project for Spring Boot applications and services.

Unsurprisingly, Consul is evolving very fast and has become much more than just service discovery and key/value store. In the upcoming parts of the tutorial we are going to meet it again, in the different incarnations though.

12.4 Load Balancing

The service discovery is foundational for building scalable and resilient microservice architecture. The knowledge about how many service instances are available and their locations is essential for the consumer to have the job done but also unexpectedly reveals another issue: among many, what service instance should be picked up for a next request? This is known as load balancing and aims to optimize the resource efficiency, overall system reliability and availability.

By and large, there are two types of load balancing, client-side and server-side (including the DNS ones). In case of the client-side load balancing, each client fetches the list of available peers through service discovery and makes its own decision which one to call. In contrast, in case of server-side load balancing though, there is single entry point (load balancer) for clients to communicate with, which forwards the requests to the respective service instances.

As you may expect, load balancers may sit on different OSI levels and are expected to support different communication protocols (like TCP, UDP, HTTP, HTTP/2, gRPC, ...). From the microservices implementation perspective, the presence of the health checks is a necessary operational requirement in order for the load balancer to maintain the up-to-date set of live instances. It is really exciting to see the revived efforts regarding health checks formalization, namely Health Check Response Format for HTTP APIs and GRPC Health Checking Protocol.

In the rest of this section we are going to discuss typical open source solutions which you may find useful for load balancing (and reverse proxying).

12.4.1 nginx

nginx is an open source software for web serving, reverse proxying, caching, load balancing of TCP/ HTTP/ UDP traffic (including HTTP/2 and gRPC as well), media streaming, and much more. What makes nginx extremely popular choice is the fact that its capabilities go way beyond just load balancing and it works really, really well.

12.4.2 HAProxy

HAProxy is free, very fast, reliable, high performance TCP/ HTTP (including HTTP/2 and gRPC) load balancer. Along with nginx, it has become the de-facto standard open source load balancer, suitable for the most kinds of deployment environments and workloads.

12.4.3 Synapse

Synapse is a system for service discovery, developed and open sourced by Airbnb. It is part of the SmartStack framework and is built on top of battle-tested Zookeeper, HAProxy (or nginx).

12.4.4 Traefik

Traefik is a very popular open source reverse proxy and load balancer. It integrates exceptionally well with existing infrastructure components and supports HTTP, Websocket and HTTP/2 and gRPC. One of the strongest sides of the Traefik is its operational friendliness, since it exposes metrics, access logs, bundles web UI and REST(ful) web APIs (beside clustering, retries and circuit breaking).

12.4.5 Envoy

Envoy is a representative of the new generation of edge and service proxies. It supports advanced load balancing features (including retries, circuit breaking, rate limiting, request shadowing, zone local load balancing, etc) and has first class support for HTTP/2 and gRPC. Also, one of the unique features provided by Envoy is a transparent HTTP/1.1 to HTTP/2 proxying. Envoy assumes a side-car deployment model, which basically means to have it running alongside with applications or services. It has become a key piece of the modern infrastructure and we are going to meet Envoy shortly in the next parts of the tutorial.

12.4.6 Ribbon

So far we have seen only server-side load balancers or side-cars but it is time to introduce Ribbon, open sourced by Netflix, a client-side IPC library with built-in software load balancing. It supports TCP, UDP and HTTP protocols and integrates very well with Eureka. Ribbon is also supported by Spring Cloud Netflix so its integration into Spring Boot applications and services is really no-brainer.

12.5 Cloud

Every single cloud provider offers its own services with respect to configuration management, service discovery and load balancing. Some of them are built-in, others might be available in certain contexts, but by and large, leveraging such offerings could save you a lot of time and money.

The move to serverless execution model lays down different requirements to discovery and scaling in response to changing demands. It really becomes the part of the platform and the solutions we have talked about so far my not shine there.

12.6 Conclusions

In this part of the tutorials we have discussed such important subjects as configuration management, service discovery and load balancing. It was not an exhaustive overview but focused on the understanding of the foundational concepts. As we are going to see later on, the new generation of infrastructure tooling goes even further by taking care of many concerns posed by the microservices architecture.

12.7 What's next

In the next part of the tutorial we are going to talk about API gateways and aggregators, yet other critical pieces of the microservices deployment.

Chapter 13

API Gateways and Aggregators

13.1 Introduction

In the last part of the tutorial we were talking about the different means of how services in the microservices architecture discover each other. Hopefully it was a helpful discussion, but we left completely untouched the topic of how other consumers, like desktop, web frontends or mobile clients, are dealing with this kind of challenge.

The typical frontend or mobile application may need to communicate with dozens of microservices, which in case of REST(ful) service backends for example, requires the knowledge of how to locate each endpoint in question. The usage of service discovery or service registry is not practical in such circumstances since these infrastructure components should not be publicly accessible. This does not leave many options besides pre-populating the service connection details in some sort of configuration settings (usually configuration files) that the client is able to consume.

By and large, this approach works but raises another problem: the number of round-trips between clients and services is skyrocketing. It is particularly painful for the mobile clients which often communicate over quite slow and unreliable network channels. Not only that, it could be quite expensive in case of cloud-based deployments where many cloud providers charge you per number of requests (or invocations). The problem is real and needs to be addressed, but how? This is the moment where API gateway pattern appears on the stage.

There are many formal and informal definitions of what API gateway actually is, the one below is trying to encompass all aspects of it in a few sentences.

API gateway is server that acts as an API front-end, receives API requests, enforces throttling and security policies, passes requests to the back-end service and then passes the response back to the requester. A gateway often includes a transformation engine to orchestrate and modify the requests and responses on the fly. A gateway can also provide functionality such as collecting analytics data and providing caching. The gateway can provide functionality to support authentication, authorization, security, audit and regulatory compliance. - https://en.wikipedia.org/wiki/API_management

A more abstract and shorter description of the API gateway definition comes from excellent blog post An API Gateway is not the new Unicorn, a highly recommended reading.

The API gateway is a way to solve the problem of how clients consume their use-cases in a microservice-based ecosystem within the microservice pattern - https://www.krakend.io/blog/what-is-an-api-gateway/

Along the rest of the tutorial we are going to talk about different kind of API gateways available in the wild and in which circumstances they can be useful.

13.2 Zuul 2

Zuul was born at Netflix and serves as the front door for all requests coming to their streaming backends. It is a gateway service (also sometimes called edge service) that provides dynamic routing, monitoring, resiliency, security, and a lot more. It went through a major revamp recently and has been rebranded as Zuul 2, the next generation.

Essentially, Zuul provides basic building blocks but everything else, like for example routing rules, is subject of customization through filters and endpoints abstractions. Such extensions should be implemented in Groovy, the scripting language of choice. For example, the **JCG Car Rentals** platform heavily uses Zuul to front the requests to all its services by providing its own inbound filter implementation.

```
class Routes extends HttpInboundSyncFilter {
    @Override
    int filterOrder() {
        return 0
    }

    @Override
    boolean shouldFilter(HttpRequestMessage httpRequestMessage) {
        return true
    }

    @Override
    HttpRequestMessage apply(HttpRequestMessage request) {
        SessionContext context = request.getContext()

        if (request.getPath().equals("/inventory") || request.getPath().startsWith("/ ↩
            inventory/")) {
            request.setPath("/api" + request.getPath())
            context.setEndpoint(ZuulEndPointRunner.PROXY_ENDPOINT_FILTER_NAME)
            context.setRouteVIP("inventory")
        } else if (request.getPath().equals("/customers") || request.getPath().startsWith(" ↩
            /customers/")) {
            request.setPath("/api" + request.getPath())
            context.setEndpoint(ZuulEndPointRunner.PROXY_ENDPOINT_FILTER_NAME)
            context.setRouteVIP("customers")
        } else if (request.getPath().equals("/reservations") || request.getPath(). ↩
            startsWith("/reservations/")) {
            request.setPath("/api" + request.getPath())
            context.setEndpoint(ZuulEndPointRunner.PROXY_ENDPOINT_FILTER_NAME)
            context.setRouteVIP("reservations")
        } else if (request.getPath().equals("/payments") || request.getPath().startsWith("/ ↩
            payments/")) {
            request.setPath("/api" + request.getPath())
            context.setEndpoint(ZuulEndPointRunner.PROXY_ENDPOINT_FILTER_NAME)
            context.setRouteVIP("payments")
        } else {
            context.setEndpoint(NotFoundEndpoint.class.getCanonicalName())
        }

        return request
    }
}
```

Zuul is very flexible and gives you a full control over the APIs management strategies. Beside many other features, it integrates very well with Eureka for service discovery and Ribbon for load balancing. The server initialization is pretty straightforward.

```
public class Bootstrap {
    public static void main(String[] args) {
        Server server = null;

        try {
            ConfigurationManager.loadCascadedPropertiesFromResources("application");
            final Injector injector = InjectorBuilder.fromModule(new RentalsModule()). ↩
                createInjector();
            final BaseServerStartup serverStartup = injector.getInstance(BaseServerStartup. ↩
                class);
            server = serverStartup.server();
```

```
            server.start(true);
        } catch (final IOException ex) {
            throw new UncheckedIOException(ex);
        } finally {
            // server shutdown
            if (server != null) {
                server.stop();
            }
        }
    }
}
```

It is battle-tested in production for years and its effectiveness as API gateway and/or edge service is proven at Netflix's scale.

13.3 Spring Cloud Gateway

Spring Cloud Gateway, a member of the Spring platform, is a library to facilitate building your own API gateways leveraging Spring MVC and Spring WebFlux. The first generation of Spring Cloud Gateway was built on top of Zuul but it is not the case anymore. The new generation has changed the power train to Spring's own Project Reactor and its ecosystem.

Let us take a look on how **JCG Car Rentals** platform could leverage Spring Cloud Gateway to have an edge entry point for its APIs.

```
server:
  port: 17001

spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
      routes:
      - id: inventory
        uri: lb://inventory-service
        predicates:
        - Path=/inventory/**
        filters:
        - RewritePath=/(?.*), /api/$\\{path}
      - id: customers
        uri: lb://customer-service
        predicates:
        - Path=/customers/**
        filters:
        - RewritePath=/(?.*), /api/$\\{path}
      - id: reservations
        uri: lb://reservation-service
        predicates:
        - Path=/reservations/**
        filters:
        - RewritePath=/(?.*), /api/$\\{path}
      - id: payments
        uri: lb://payment-service
        predicates:
        - Path=/payments/**
        filters:
        - RewritePath=/(?.*), /api/$\\{path}

eureka:
  instance:
```

```
    appname: api-gateway
    preferIpAddress: true
  client:
    register-with-eureka: false
    fetch-registry: true
    healthcheck:
      enabled: true
    service-url:
      defaultZone: https://localhost:20200/eureka/
```

As you can spot right away, we have used a purely configuration-driven approach along with Eureka integration for service discovery. Running the server using Spring Boot requires just a few lines of code.

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

Similarly to Zuul 2, Spring Cloud Gateway allows you to slice and dice whatever features your microservices architecture demands from the API gateway. However, it also becomes your responsibility to maintain and learn how to operate it.

One of the benefits to building your own API gateway is the freedom to perform aggregations and fan-outs over multiple services. This way the number of round-trips which clients have to perform otherwise could be reduced significantly since API gateway would be responsible for stitching the multiple responses together. There is the dark side of this path though, please stay tuned.

13.4 HAProxy

In the previous part of the tutorial we talked about HAProxy primarily as a load balancer, however its capabilities allow it to serve as API gateway as well. If HAProxy already made its way into your microservice architecture, trying it in a role of API gateway is worth considering.

13.5 Microgateway

Microgateway by StrongLoop is a great illustration of the innovations happening in the world of JavaScript and particularly Node.js ecosystem.

The Microgateway is a developer-focused, extensible gateway framework written in Node.js for enforcing access to Microservices and APIs. - https://strongloop.com/projects/

13.6 Kong

Kong is among the first API gateways which emerged at Mashape to address the challenges of their microservice deployments.

Kong is a scalable, open source API Layer (also known as an API Gateway, or API Middleware). Kong runs in front of any RESTful API and is extended through Plugins, which provide extra functionality and services beyond the core platform. - https://konghq.com/about-kong/

Written in Lua, Kong is built on solid foundation of nginx (which we have talked about in the previous part of the tutorial) and is distributed along with OpenResty, a full-fledged nginx-powered web platform.

13.7 Gravitee.io

From the focused API gateway solutions we are gradually moving towards more beefy options, starting from Gravitee.io, an open-source API platform.

Gravitee.io is a flexible, lightweight and blazing-fast open source API Platform that helps your organization control finely who, when and how users access your APIs. - https://gravitee.io/

The Gravitee.io platform consists of three core components: in the center is API Gateway, surrounded by Management API and Management Web Portal.

13.8 Tyk

Tyk is yet another example of lightweight and comprehensive API platform, with the API gateway in the heart of it.

Tyk is an open source API Gateway that is fast, scalable and modern. Out of the box, Tyk offers an API Management Platform with an API Gateway, API Analytics, Developer Portal and API Management Dashboard. - https://tyk.io/

Tyk is written in Go and is easy to distribute and deploy. It has quite large list of the key features, with the emphasis on API analytics and access management.

13.9 Ambassador

The hyper-popularity of Kubernetes led to the rise of API gateways which could natively run on it. One of the pioneers in this category is Ambassador by Datawire.

Ambassador is an open source Kubernetes-native API Gateway built on Envoy, designed for microservices. Ambassador essentially serves as an Envoy ingress controller, but with many more features. - https://github.com/datawire/ambassador

Since most of the organizations are leveraging Kubernetes to deploy their microservices fleet, Ambassador has occupied the leading positions there.

13.10 Gloo

Yet another notable representative of the Kubernetes-native API gateways is Gloo, open-sourced and maintained by solo.io.

Gloo is a feature-rich, Kubernetes-native ingress controller, and next-generation API gateway. Gloo is exceptional in its function-level routing; its support for legacy apps, microservices and serverless; its discovery capabilities; its numerous features; and its tight integration with leading open-source projects. - https://gloo.solo.io/

Gloo is built on top of the Envoy. The seamless integration with a number of serverless offerings makes Gloo a truly unique solution.

13.11 Backends for Frontends (BFF)

One of the challenges that many microservice-based platforms face these days is dealing with the variety of different types of consumers (mobile devices, desktop applications, web frontends, ...). Since every single consumer has own unique needs and requirements, it clashes with the reality to run against one-size-fit-all backend services.

The API gateway could potentially help, often trading the convenience for the explosion of the APIs, tailored for each consumer. To address these shortcomings, the Backends For Frontends (or BFF) pattern has been emerged and gained some traction. In particular, backed by GraphQL, it becomes a very efficient solution to the problem.

Let us quickly look though the **JCG Car Rentals** platform which includes the BFF component, based on GraphQL and Apollo GraphQL stack. The implementation itself uses REST Data Source to delegate the work to **Reservation Service**, **Customer Service** or/and **Inventory Service**, transparently to the consumer which just asks for what it needs using GraphQL queries.

```
query {
  reservations(customerId: $customerId) {
    from
    to
  }
  profile(id: $customerId) {
    firstName
    lastName
  }
}
```

BFF, especially GraphQL ones, may not be classified as traditional API gateway however it is certainly a very useful pattern to consider when dealing with the multitude of different clients. The major benefit BFFs bring on the table is the ability to optimize for specific client or platform but they may also sidetrack to the danger zone easily.

13.12 Build Your Own

In case none of the existing approaches look appealing to your microservice architecture needs, there is always the possibility to build your own. Leveraging the full-fledged integration frameworks like Apache Camel or Spring Integration could be the best and shortest path to get you there. Moreover, if you are already betting on these frameworks, using the familiar paradigm is much more efficient than learning yet another technology (spoiler alert, do not let hype to mislead you).

13.13 Cloud

Every major cloud provider has at least one kind of API gateway offering. It may not be the best in class but hopefully is good enough. On the bright side, it has seamless integration with numerous other offerings, specifically related to security and access control.

One interesting subject to touch upon is to understand what is the role of API gateways in the serverless computing? It will not be an overstatement to say that the API gateway is a very core component in such architecture since it provides one of the entry points into the serverless execution model. If a trivial microservices deployment may get along without an API gateway, the serverless may not get far without it.

13.14 On the Dark Side

The fight for leadership in the niche of the API gateways forces the vendors to pour more and more features into their products, which essentially reshapes the definition of what an API gateway is and leads straight to the identity crisis. The problem is exceptionally well summarized by ThoughtWorks in their terrific technology radar.

We remain concerned about business logic and process orchestration implemented in middleware, especially where it requires expert skills and tooling while creating single points of scaling and control. Vendors in the highly competitive API gateway market are continuing this trend by adding features through which they attempt to differentiate their products. This results in overambitious API gateway products whose functionality - on top of what is essentially a reverse proxy - encourages designs that continue to be difficult to test and deploy. API gateways do provide utility in dealing with some specific concerns - such as authentication and rate limiting - but any domain smarts should live in applications or services. - https://www.thoughtworks.com/-radar/platforms/overambitious-api-gateways

The issues are not imaginable and this is indeed happening in many organizations. Please take it seriously and avoid this trap along your journey.

13.15 Microservices API Gateways and Aggregators - Conclusions

In this part of the tutorial we have identified the role of the API gateway, yet another key piece in the modern microservice architecture. Along with BFF, it takes care of many cross-cutting concerns and removes unnecessary burden from the consumers, but at the cost of increasing complexity. We have also discussed the common pitfalls the organizations fall into while introducing API gateways and BFFs, the mistakes other did and you should learn from and avoid.

13.16 What's next

In the next section of the tutorial we are going to talk about deployment and orchestration, specifically suited for microservices.

Chapter 14

Deployment and Orchestration

14.1 Introduction

These days more and more organizations are relying on cloud computing and managed service offerings to host their services. This strategy has a lot of benefits but you still have to choose the best deployment game plan for your microservices fleet.

Using some sort of PaaS is probably the easiest option but for many it is not sustainable in the long run due to the inherited constraints and limitations such model has. From the other side, using IaaS does relief the costs of infrastructure management and maintenance, but still requires a significant amount of work with respect to deployment of the applications and services and keeping them afloat. Last but not least, a lot of the organizations still prefer to manage their software stacks internally, only offloading the virtual (or bare-metal) machines management to cloud providers.

The challenge to decide which model is right for the majority of the organizations stayed unsolved (at large) for quite a long time, waiting for some kind of breakthrough to happen. And luckily, the "big bang" came in due time.

14.2 Containers

Although the seeds have been planted long before, the revolution has been initiated by Docker and has drastically changed the way we used to approach the distribution, deployment and development of the applications and services. The game changer popped up in a form of operating system level virtualization and containers. It is an exceptionally lightweight (comparing to traditional virtual machines) architecture, imposes little to no overhead, share the same operating system kernel and do not require special hardware support to perform efficiently.

Nowadays the container images became the de-facto packaging and distribution blueprint whereas the containers serve as the mainstream execution and isolation model. There are a lot to say about Docker and container-based virtualization, specifically with respect to the applications and services on the JVM platform, but along this part of the tutorial we are going to focus on the deployment and operational aspects.

The tooling around containers is available for mostly any programming language or/and platform and in most cases it could be easily integrated into the build and deployment pipelines. With respect to the **JCG Car Rentals** platform, for example, the **Customer Service** builds a container image using jib (more precisely jib-maven-plugin) which assembles and publishes the image without requiring Docker daemon.

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>1.3.0</version>
    <configuration>
        <to>
            <image>jcg-car-rentals/customer-service</image>
            <tags>
                <tag>${project.version}</tag>
```

```
            </tags>
        </to>
        <from>
            <image>openjdk:8</image>
        </from>
        <container>
            <user>1000</user>
            <mainClass>ws.ament.hammock.Bootstrap</mainClass>
        </container>
    </configuration>
</plugin>
```

So what we could do with the container now? The move towards container-based runtimes spawned a new category of the infrastructure components, the container orchestration and management.

14.3 Apache Mesos

We are going to start with Apache Mesos, one of the oldest and well-established open-source platforms for fine-grained resource sharing.

Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. - https://mesos.apache.org/

Strictly speaking, Apache Mesos is not a container orchestrator, more like a cluster-management platform, but it also gained a native support for launching containers not so long ago. There are certain overlaps with traditional cluster management frameworks (like for example Apache Helix), so Apache Mesos is often called the operating system for the datacenters, to emphasize its larger footprint and scope.

14.4 Titus

Titus, yet another open-source project from Netflix, is an example of the dedicated container management solution.

Titus is a container management platform that provides scalable and reliable container execution and cloud-native integration with Amazon AWS. Titus was built internally at Netflix and is used in production to power Netflix streaming, recommendation, and content systems. - https://netflix.github.io/titus/

In the essence, Titus is a framework on top of Apache Mesos. The seamless integration with AWS as well as Spinnaker, Eureka and Archaius make it quite a good fit after all.

14.5 Nomad

Nomad, one more open-sourced gem from the HashiCorp, is a workload orchestrator which is suitable for deploying a mix of microservices, batch jobs, containerized and non-containerized applications.

Nomad is a highly available, distributed, data-center aware cluster and application scheduler designed to support the modern datacenter with support for long-running services, batch jobs, and much more. - https://www.nomadproject.io/

Besides being really very easy to use, it has outstanding native integration with Consul and Vault to complement the service discovery and secret management (which we have introduced in the previous parts of the tutorial).

14.6 Docker Swarm

If you are an experienced Docker user, you may know about the swarm, a special Docker operating mode for natively managing a cluster of Docker Engines. It is probably the easiest way to orchestrate the containerized deployments but at the same time not widely adopted.

14.7 Kubernetes

The true gem we left to the very end. Kubernetes, built upon 15 years of experience of running production workloads at Google, is an open-source, hyper-popular and production-grade container orchestrator.

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. - https://kubernetes.io/

Undoubtedly, Kubernetes is a dominant container management platform these days. It could be run literally on any infrastructure, and as we are going to see shortly, is offered by all major cloud providers.

The **JCG Car Rentals** platform is going to leverage Kubernetes capabilities to run all its microservices and supporting components (like API gateways and BFFs). Although we could start crafting the YAML manifests right away, there is one more thing to talk about.

Kubernetes is a platform for building platforms. It's a better place to start; not the endgame.- https://twitter.com/-kelseyhightower/status/935252923721793536?lang=en

It is quite interesting statement which is already being put into life these days by the platforms like OpenShift and a number of commercial offerings.

14.8 Service Meshes

The container orchestration platforms significantly simplified and improved the deployment and operational practices, specifically related to microservices. However there were a number of crosscutting concerns which services and applications developers have to take care of, like secure communication, resiliency, authentication and authorization, tracing and monitoring, to name a few. The API gateways took some of that burden off but the service-to-service communication still struggled. The search for the viable solution led to rise of the service meshes.

A service mesh is an infrastructure layer that handles service-to-service communication, freeing applications from being aware of the complex communication network. The mesh provides advanced capabilities, including encryption, authentication and authorization, routing, monitoring and tracing. - https://medium.com/solo-io/https-medium-com-solo-io-supergloo-ff2aae1fb96f

To be fair, service meshes came in like a life savers. Deployed along the container orchestrator of choice, they allowed the organizations to keep the focus on implementing the business concerns and features, whereas the mesh was taking care of the rest.

Service mesh is the future of cloud-native apps - https://medium.com/solo-io/https-medium-com-solo-io-supergloo-ff2aae1fb96f

There are a number of service meshes in the wild, quite mature and battle-tested in production. Let us briefly go over some of them.

14.8.1 Linkerd

We are going to start with Linkerd, one of the pioneers of the open source service meshes. It has taken off as the dedicated layer for managing, controlling, and monitoring service-to-service communication. Since then, it has been rewritten and refocused on Kubernetes integration.

Linkerd is an ultralight service mesh for Kubernetes . It gives you observability, reliability, and security without requiring any code changes. - https://linkerd.io/

The meaning of "ultralight" may not sound significant but it actually is. You might be surprised by how much cluster resources a service mesh may consume and, depending on your deployment model, may incur substantial additional costs.

14.8.2 Istio

If there is a service mesh everyone have heard of, it is very likely Istio.

It is a completely open source service mesh that layers transparently onto existing distributed applications. It is also a platform, including APIs that let it integrate into any logging platform, or telemetry or policy system. Istio 's diverse feature set lets you successfully, and efficiently, run a distributed microservice architecture, and provides a uniform way to secure, connect, and monitor microservice - https://istio.io/docs/concepts/what-is-istio/

Although Istio is used mostly with Kubernetes, it is in fact platform independent. For example, as of now it could be run along with Consul-based deployments (with or without Nomad).

The ecosystem around Istio is really flourishing. One notable community contribution is Kiali, which visualizes the service mesh topology and provides visibility into features like request routing, circuit breakers, request rates, latency and more.

The need of the service mesh for **JCG Car Rentals** platform is obvious and we are going to deploy Istio to fulfill this gap. Here is the simplistic example of the Kubernetes deployment manifest for **Customer Service** using Ist io and previously built container image.

```
apiVersion: v1
kind: Service
metadata:
  name: customer-service
  labels:
    app: customer-service
spec:
  ports:
  - port: 18800
    name: http
  selector:
    app: customer-service

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: customer-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: customer-service
  template:
    metadata:
      labels:
        app: customer-service
    spec:
      containers:
      - name: customer-service
        image: jcg-car-rentals/customer-service:0.0.1-SNAPSHOT
        resources:
          requests:
            cpu: "200m"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 18800
        volumeMounts:
        - name: config-volume
          mountPath: /app/resources/META-INF/microprofile-config.properties
          subPath: microprofile-config.properties
      volumes:
        - name: config-volume
          configMap:
            name: customer-service-config

---
```

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: customer-service-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"

---

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customer-service
spec:
  hosts:
  - "*"
  gateways:
  - customer-service-gateway
  http:
  - match:
    - uri:
        prefix: /api/customers
    route:
    - destination:
        host: customer-service
        port:
          number: 18800
```

14.8.3 Consul Connect

As we know from the previous part of the tutorial, Consul started off as service discovery and configuration storage. One of the recent additions to the Consul is Connect feature which allowed it to enter into the space of the service meshes.

Consul Connect provides service-to-service connection authorization and encryption using mutual Transport Layer Security (TLS). Applications can use sidecar proxies in a service mesh configuration to automatically establish TLS connections for inbound and outbound connections without being aware of Connect at all. - https://www.consul.io/docs/connect/-index.html

Consul already had the perfect foundation each service mesh needed, adding the missing features was a logical step towards adapting to this fast changing landscape.

14.8.4 SuperGloo

With quite a few service meshes available, it becomes really unclear which one is the best choice for your microservices, and how to deploy and operate one? If that is the problem you are facing right now, you may take a look at SuperGloo, the service mesh orchestration platform.

SuperGloo , an open-source project to manage and orchestrate service meshes at scale. SuperGloo is an opinionated abstraction layer that will simplify the installation, management, and operation of your service mesh, whether you use

(or plan to use) a single mesh or multiple mesh technologies, on-site, in the cloud, or on any topology that best fits you. - https://supergloo.solo.io/

From the service meshes perspective, SuperGloo currently supports (to some extent) Istio, Consul Connect, Linkerd and AWS App Mesh.

On the same subject, the wider Service Mesh Interface (SMI) specification was announced recently, an undergoing initiative to align different service mesh implementations so they could be used interchangeably. .

14.9 Cloud

The industry-wide shift towards container-based deployments has forced the cloud providers to come up with the relevant offerings. As of today, every major player in the cloud business has managed Kubernetes offering along with the service mesh.

14.9.1 Google Kubernetes Engine (GKE)

Since Kubernetes emerged from Google and from its experience managing world's largest computing clusters, it is only natural that Google Cloud has an outstanding support for it. And that is really the case, Google's Kubernetes Engine (GKE) is a fully managed Kubernetes platform hosted in the Google Cloud.

Kubernetes Engine is a managed, production-ready environment for deploying containerized applications. It brings our latest innovations in developer productivity, resource efficiency, automated operations, and open source flexibility to accelerate your time to market. - https://cloud.google.com/kubernetes-engine/

As for the service mesh, Google Cloud provides Istio support through Istio on GKE add-on for Kubernetes Engine (currently in beta).

14.9.2 Amazon Elastic Kubernetes Service (EKS)

For quite a while AWS offers the support for running containerized applications in the form of Amazon Elastic Container Service (ECS). But since the last year AWS announced the general availability of the Amazon Elastic Kubernetes Service (EKS).

Amazon EKS runs the Kubernetes management infrastructure for you across multiple AWS availability zones to eliminate a single point of failure. - https://aws.amazon.com/eks/

From the service mesh side, you are covered by AWS App Mesh which could be used with Amazon Elastic Kubernetes Service. Under the hood it is powered by Envoy service proxy.

14.9.3 Azure Container Service (AKS)

The Microsoft Azure Cloud followed a similar to AWS approach by offering Azure Container Service first (which by the way could have been deployed with Kubernetes or Docker Swarm) and then deprecating it in favor of the Azure Kubernetes Service (AKS).

The fully managed Azure Kubernetes Service (AKS) makes deploying and managing containerized applications easy. It offers serverless Kubernetes, an integrated continuous integration and continuous delivery (CI/CD) experience, and enterprise-grade security and governance. - https://azure.microsoft.com/en-us/services/kubernetes-service/

Interestingly, as of moment of this writing Microsoft Azure Cloud does not bundle the support of any service mesh with its Azure Kubernetes Service offering but it is possible to install Istio components on AKS following the manual procedure.

14.9.4 Rancher

It is very unlikely that your microservices fleet will be deployed in one single Kubernetes cluster. At least, you may have production and staging ones and these should be better kept separated. If you care about your customers, you would probably think hard about the high-availability and disaster recovery, which essentially means multi-region or multi-cloud deployments.

Managing many Kubernetes clusters across the wide range of the environments could be cumbersome and difficult, unless you know about Rancher.

Rancher is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters across any infrastructure, while providing DevOps teams with integrated tools for running containerized workloads. - https://rancher.com/what-is-rancher/overview/

By and large, Rancher becomes a single platform to operate your Kubernetes clusters, including managed cloud offerings or even bare-metal servers.

14.10 Deployment and Orchestration - Conclusions

In this section of the tutorial we have talked about the container-based deployments and orchestration. Nonetheless there are a few options on the table it is fair to say that Kubernetes is the de-facto choice nowadays and for good reasons. Although not strictly required, the presence of the service mesh is going to greatly relief certain pains of the microservices operational concerns and let you focus on what is important for business instead.

14.11 What's next

In the next section of the tutorial we are going to talk about log management, consolidation and aggregation.

Chapter 15

Log Management

15.1 Introduction

With this part of the tutorial we are entering the land of the observability. Sounds like another fancy buzzword, so what is that exactly?

In a distributed system, which is inherently implied by microservice architecture, there are too many moving pieces that interact and could fail in unpredictable ways.

Observability is the activities that involve measuring, collecting, and analyzing various diagnostics signals from a system. These signals may include metrics, traces, logs, events, profiles and more. - https://medium.com/observability/-microservices-observability-26a8b7056bb4

As quickly as possible spot the problems, pin-point the exact place (or places) in the system where they emerged, and figure out the precise cause, these are the ultimate goals of the observability in the context relevant to the microservices. It is indeed a very difficult target to achieve and requires a compound approach.

The first pillar of the observability we are going to talk about is logging. When logs are done well, they can contain valuable (and often, invaluable) details about the state your applications or/and services are in. Logs are the primary source to tap you directly into application or/and service errors stream. Beyond that, on the infrastructure level, logs are exceptionally helpful in identifying security issues and incidents.

Unsurprisingly, we are going to focus on application and service logs. The art of logging is probably the skill we, developers, are perfecting throughout the lifetime. We know that the logs should be useful, easy to understand (more often than not it will be us or our teammates running over them) and contain enough meaningful data to reconstruct the flow and troubleshoot the issue. Logs bloat or logs shortage, both lead to waste of precious time or/and resources, finding the right balance is difficult. Moreover, the incidents related to leaking the personal data through careless logging practices are not that rare but the consequences of that are far-reaching.

The distributed nature of the microservices assumes the presence of many services, managed by different teams, very likely implemented using different frameworks, and running on different runtimes and platforms. It leads to proliferation of log formats and practices but despite that, you have to be able to consolidate all logs in a central searchable place and be able to correlate the events and flows across the microservice and infrastructure boundaries. It sounds like impossible task, isn't it? Although it is certainly impossible to cover every single logging framework or library out there, there is a core set of principles to start with.

15.2 Structured or Unstructured?

It is unrealistic to come up and enforce the universally applicable format for logs since every single application or service is just doing different things. The general debate however unfolds around structured versus unstructured logging.

To understand what the debate is about, let us take a look at how the typical Spring Boot application does logging, using **Reservation Service**, part of the **JCG Car Rentals** platform, as an example.

```
...
2019-07-27 14:13:34.080  INFO 15052 --- [           main] o.c.cassandra.migration. ↩
    MigrationTask    : Keyspace rentals is already up to date at version 1
2019-07-27 14:13:34.927  INFO 15052 --- [           main] d.s.w.p. ↩
    DocumentationPluginsBootstrapper : Documentation plugins bootstrapped
2019-07-27 14:13:34.932  INFO 15052 --- [           main] d.s.w.p. ↩
    DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2019-07-27 14:13:34.971  INFO 15052 --- [           main] s.d.s.w.s. ↩
    ApiListingReferenceScanner     : Scanning for api listing references
2019-07-27 14:13:35.184  INFO 15052 --- [           main] o.s.b.web.embedded.netty. ↩
    NettyWebServer  : Netty started on port(s): 18900
...
```

As you may notice, the logging output follows some pattern, but in general, it is just a just a freestyle text which becomes much more interesting when exceptions come to the picture.

```
2019-07-27 14:30:08.809  WARN 12824 --- [nfoReplicator-0] com.netflix.discovery. ↩
    DiscoveryClient    : DiscoveryClient_RESERVATION-SERVICE/********:reservation-service ↩
    :18900 - registration failed Cannot execute request on any known server

com.netflix.discovery.shared.transport.TransportException: Cannot execute request on any  ↩
    known server
        at com.netflix.discovery.shared.transport.decorator.RetryableEurekaHttpClient. ↩
            execute(RetryableEurekaHttpClient.java:112) ~[eureka-client-1.9.12.jar:1.9.12]
        at com.netflix.discovery.shared.transport.decorator.EurekaHttpClientDecorator. ↩
            register(EurekaHttpClientDecorator.java:56) ~[eureka-client-1.9.12.jar:1.9.12]
        at com.netflix.discovery.shared.transport.decorator.EurekaHttpClientDecorator$1. ↩
            execute(EurekaHttpClientDecorator.java:59) ~[eureka-client-1.9.12.jar:1.9.12]
...
```

Extracting meaningful data out of such logs is not fun. Essentially, you have to parse and pattern-match every single log statement, determine if it is single or multiline, extract timestamps, log levels, thread names, key/value pairs, as so on. It is feasible in general but also time-consuming, computationally heavy, fragile and difficult to maintain. Let us compare that with the structured logging where the format is more or less standard (let say, JSON) but the set of fields may (and in reality will) differ.

```
{"@timestamp":"2019-07-27T22:12:19.762-04:00","@version":"1","message":"Keyspace rentals is ↩
     already up to date at version 1","logger_name":"org.cognitor.cassandra.migration. ↩
    MigrationTask","thread_name":"main","level":"INFO","level_value":20000}
{"@timestamp":"2019-07-27T22:12:20.545-04:00","@version":"1","message":"Documentation  ↩
    plugins bootstrapped","logger_name":"springfox.documentation.spring.web.plugins. ↩
    DocumentationPluginsBootstrapper","thread_name":"main","level":"INFO","level_value" ↩
    :20000}
{"@timestamp":"2019-07-27T22:12:20.550-04:00","@version":"1","message":"Found 1 custom  ↩
    documentation plugin(s)","logger_name":"springfox.documentation.spring.web.plugins. ↩
    DocumentationPluginsBootstrapper","thread_name":"main","level":"INFO","level_value" ↩
    :20000}
{"@timestamp":"2019-07-27T22:12:20.588-04:00","@version":"1","message":"Scanning for api  ↩
    listing references","logger_name":"springfox.documentation.spring.web.scanners. ↩
    ApiListingReferenceScanner","thread_name":"main","level":"INFO","level_value":20000}
{"@timestamp":"2019-07-27T22:12:20.800-04:00","@version":"1","message":"Netty started on  ↩
    port(s): 18900","logger_name":"org.springframework.boot.web.embedded.netty. ↩
    NettyWebServer","thread_name":"main","level":"INFO","level_value":20000}
```

Those are the same logs represented in a structural way. From the indexing and analysis perspective, dealing with such structured data is significantly easier and more convenient. Please consider to favor the structuring logging by your microservices fleet, it will certainly pay off.

15.3 Logging in Containers

The next question after settling on logs format is where these logs should be written to. To find the right answer, we may turn back to The Twelve-Factor App principles.

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to `stdout` . During local development, the developer will view this stream in the foreground of their terminal to observe the app's behavior. - https://12factor.net/logs

Since all of **JCG Car Rentals** microservices are running within the containers, they should not be concerned with how to write or store the logs but rather stream them to `stdout/stderr` . The execution/runtime environment is to make a call on how to capture and route the logs. Needless to say that such model is well supported by all container orchestrators (f.e. docker logs, kubectl logs, . . .). On the side note, dealing with multiline log statements is going to be a challenge.

It worth to mention that in certain cases you may encounter the application or service which writes its logs to a log file rather than `stdout/stderr` . Please keep in mind that since the container filesystem is ephemeral, you will have to either configure a persistent volume or forward logs to a remote endpoint using data shippers, to prevent the logs being lost forever.

15.4 Centralized Log Management

So far we have talked about the easy parts. The next one, probably the most important out of all, is logs management and consolidation.

15.4.1 Elastic Stack (formerly ELK)

The first option we are going to talk about is what is used to be known as ELK. It is an acronym which stands for three open source projects: Elasticsearch, Logstash, and Kibana.

Elasticsearch is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases. As the heart of the Elastic Stack, it centrally stores your data so you can discover the expected and uncover the unexpected. - https://www.elastic.co/products/elasticsearch

Logstash is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to your favorite "stash." - https://www.elastic.co/products/logstash

- Kibana lets you visualize your Elasticsearch data and navigate the Elastic Stack so you can do anything from tracking query load to understanding the way requests flow through your apps. -* **https://www.elastic.co/products/kibana**

ELK has gained immense popularity in the community since it provided a complete end-to-end pipeline for logs management and aggregation. The Elastic Stack is the next evolution of the ELK which also includes another open source project, Beats.

Beats is the platform for single-purpose data shippers. They send data from hundreds or thousands of machines and systems to Logstash or Elasticsearch . - https://www.elastic.co/products/beats

The Elastic Stack (or its predecessor ELK) is the number one choice if you are considering to own your logs management infrastructure. But be aware that from the operational perspective, keeping your Elasticsearch clusters up and running might be challenging.

The **JCG Car Rentals** platform uses Elastic Stack to consolidate logs across all services. Luckily, it is very easy to ship structured logs to Logstash using, for example, Logback and Logstash Logback Encoder. The `logback.xml` configuration snippet is shown below.

```
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>

    <appender name="logstash" class="net.logstash.logback.appender. ↩
        LogstashTcpSocketAppender">
        <destination>logstash:4560</destination>
```

```
        <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="logstash" />
    </root>
</configuration>
```

The logs become immediately available for searching and Kibana is a literally one-stop shop to do quite complex analysis or querying over them.

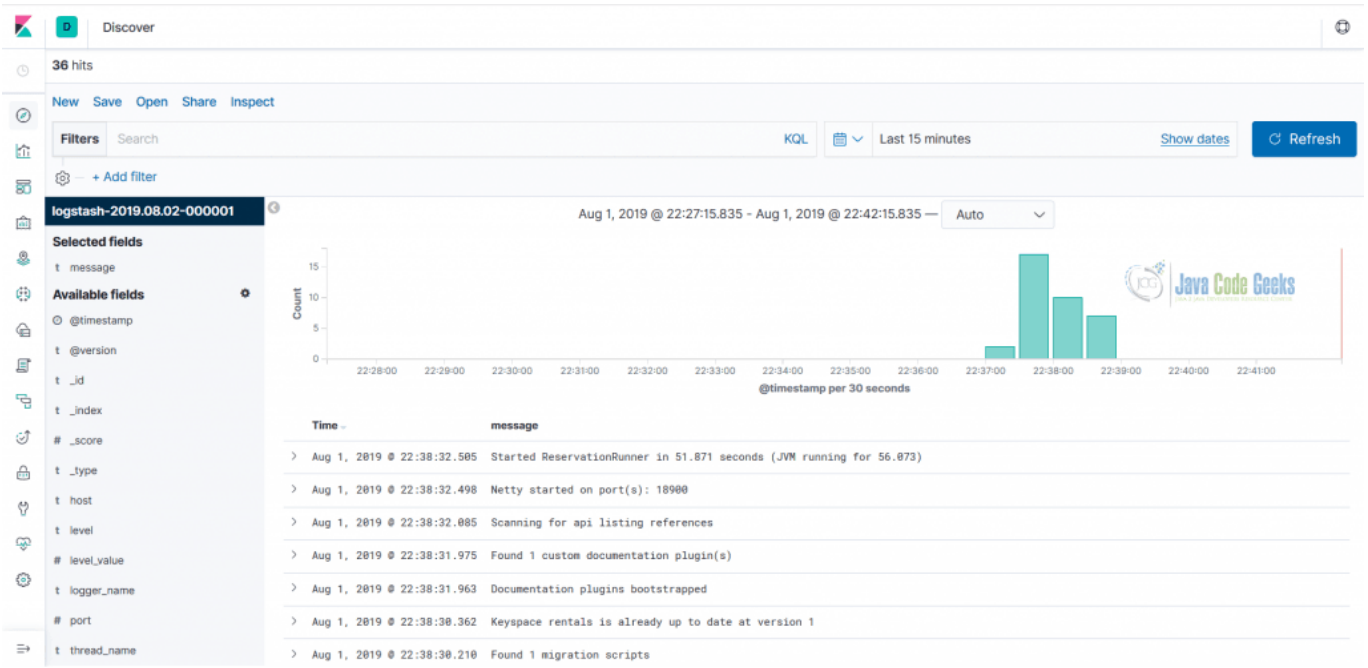


Figure 15.1: Image

Alternatively, you may just write logs to `stdout/stderr` using Logstash Logback Encoder and just tail the output to the Logstash.

15.4.2 Graylog

Graylog is yet another centralized open source log management solution which is built on top of the Elasticsearch and MongoDB.

Graylog is a leading centralized log management solution built to open standards for capturing, storing, and enabling real-time analysis of terabytes of machine data. We deliver a better user experience by making analysis ridiculously fast, efficient, cost-effective, and flexible. - https://www.graylog.org/

One of the key differences compared to Elastic Stack is that Graylog can receive structured logs (in GELF format) directly from an application or service over the network (mostly every logging framework or library is supported).

15.4.3 GoAccess

GoAccess is an open source solution which is tailored for analyzing the logs from the web servers in the real-time.

GoAccess is an open source real-time web log analyzer and interactive viewer that runs in a terminal in *nix systems or through your browser. - https://goaccess.io/

It is not a full-fledged log management offering but it has really unique set of capabilities which might be well aligned with your operational needs.

15.4.4 Grafana Loki

Loki by Grafana Labs is certainly a newcomer to the space of open source log management, with announcement being made at the end of 2018, less than a year ago.

Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus . It is designed to be very cost effective and easy to operate. It does not index the contents of the logs, but rather a set of labels for each log stream. - https://github.com/grafana/loki

Loki has a goal to stay as lightweight as possible, thus the indexing and crunching of logs is deliberately left out of scope. It comes with the first class Kubernetes support but please make a note that Loki is currently in alpha stage and is not recommended to be used in production just yet.

15.5 Log Shipping

Let us switch gears a bit from the complete off the shelf log management solutions to log data collectors and shippers. Their role is to detach the source log streams from the underlying backend systems by sitting in between. Logstash and Beats, part of the Elastic Stack, are great example of those.

15.5.1 Fluentd

Fluentd is widely used open source data collector which is now a member of the Cloud Native Computing Foundation (CNCF).

Fluentd is an open source data collector, which lets you unify the data collection and consumption for a better use and understanding of data. - https://www.fluentd.org/

One of the benefits of being CNCF member is the opportunity to closely integrate with Kubernetes and Fluentd undoubtedly shines there. It is often used as the log shipper in Kubernetes deployments.

15.5.2 Apache Flume

Apache Flume is probably one of oldest open source log data collectors and aggregators.

Flume is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store. - https://flume.apache.org/index.html

15.5.3 rsyslog

rsyslog is a powerful, modular, secure and high-performance log processing system. It accepts data from variety of sources (system or application), optionally transforms it and outputs to diverse destinations. The great thing about rsyslog is that it comes preinstalled on most Linux distributions so basically you get it for free in mostly any container.

15.6 Cloud

The leading cloud providers have quite different approaches with respect to centralized logging. As we are going to see, some do have dedicated offerings whereas others include log management as part of the larger ones.

15.6.1 Google Cloud

Google Cloud has probably one of the best real-time log management and analysis tooling out there, called Stackdriver Logging, part of the Stackdriver offering.

Stackdriver Logging allows you to store, search, analyze, monitor, and alert on log data and events from Google Cloud Platform and Amazon Web Services (AWS). Our API also allows ingestion of any custom log data from any source. Stackdriver Logging is a fully managed service that performs at scale and can ingest application and system log data from thousands of VMs. Even better, you can analyze all that log data in real time. - https://cloud.google.com/logging/

The AWS integration comes as a pleasant surprise but it is actually powered by the customized distribution of the Fluentd.

15.6.2 AWS

In the center of the AWS logs management offering is CloudWatch Logs.

CloudWatch Logs enables you to centralize the logs from all of your systems, applications, and AWS services that you use, in a single, highly scalable service. You can then easily view them, search them for specific error codes or patterns, filter them based on specific fields, or archive them securely for future analysis. - https://docs.aws.amazon.com/-AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html

Besides CloudWatch Logs, AWS also brings a use case of the centralized logging solution implementation, backed by Amazon Elasticsearch Service **.**

AWS offers a centralized logging solution for collecting, analyzing, and displaying logs on AWS across multiple accounts and AWS Regions. The solution uses Amazon Elasticsearch Service (Amazon ES), a managed service that simplifies the deployment, operation, and scaling of Elasticsearch clusters in the AWS Cloud, as well as Kibana, an analytics and visualization platform that is integrated with Amazon ES. In combination with other AWS managed services, this solution offers customers a customizable, multi-account environment to begin logging and analyzing their AWS environment and applications. - https://aws.amazon.com/solutions/centralized-logging/

15.6.3 Microsoft Azure

The Microsoft Azure's dedicated offering for managing logs went through a couple of incarnations and as of today is a part of Azure Monitor.

Azure Monitor logs is the central analytics platform for monitoring, management, security, application, and all other log types in Azure. - https://azure.microsoft.com/en-ca/blog/azure-monitor-is-providing-a-unified-logs-experience/

15.7 Serverless

It is interesting to think about subtleties of logging in the context of the serverless. At first, it is not much different, right? The evil is in details: careless logging instrumentation may considerably impact the execution time, as such directly influencing the cost. Please keep it in mind.

15.8 Microservices: Log Management - Conclusions

In this section of the tutorial we have started to talk about observability pillars, taking off from logs. The times when tailing a single log file was enough are long gone. Instead, the microservice architecture brings the challenge of logs centralization and consolidation from many different origins. Arguably, logs are still the primary source of the information to troubleshoot problems and issues in the software systems, but there are other powerful means to complement them.

The friendly reminder that along the whole tutorial we are focusing on free and open-source offerings but the market of the commercial log management solutions is just huge. Many organizations prefer to offload log management to SaaS vendors and just pay for it.

15.9 What's next

In the next section of the tutorial we are going to continue our discussion about observability, this time focusing on metrics.

Chapter 16

Metrics

16.1 Introduction

In this part of the tutorial we are going to continue our journey into observability land and tackle its next foundational pillar, metrics. While logs are descriptive, metrics take the inspiration from measurements.

If you can't measure it, you can't improve it. - Peter Drucker

Metrics are serving multi-fold purposes. First of all, they give you quick insights into the current state of your service or application. Secondly, metrics could help to correlate the behavior of different applications, services and/or infrastructure components under heavy load or outages. As a consequence, they could lead to faster problems identification and bottlenecks detection. And last but not least, metrics could help to proactively and efficiently mitigate the potential issues, minimizing the risk of them to grow into serious problems or widespread outages.

Yet there are more. One of the most valuable properties of the metrics is the ability to capture the overall system performance characteristics, as such establishing the baseline to compare and to trend over. Backed by continuous integration and delivery practices, they assist in detection of any undesirable regressions early enough, before ones sneak into production.

It sounds really useful, but what kind of metrics our systems need? How could we instrument our applications and services? And what exactly should we measure? Those are the hard questions we will be trying to attack in this part of the tutorial.

16.2 Instrument, Collect, Visualize (and Alert)

Metrics do not show up from nowhere, the applications and/or services should be instrumented in order to expose the relevant insights. Luckily, JVM ecosystem is flourishing here, there are a few excellent instrumentation libraries (notably Micrometer and Dropwizard Metrics) and most of the widely used frameworks have out-of-the box integrations with at least one of them.

Once exposed, metrics need to be collected (pushed or scraped) and persisted somewhere in order to provide the historical trends over time and aggregations. Typically, this is fulfilled by using one of the time series databases.

A time series database is built specifically for handling metrics and events or measurements that are time-stamped. A TSDB is optimized for measuring change over time. Properties that make time series data very different than other data workloads are data lifecycle management, summarization, and large range scans of many records. - https://www.influxdata.com/time-series-database/

The final phase of the metrics lifecycle is visualization, usually through pre-built dashboards, using charts / graphs, tables, heatmaps, etc. From the operational perspective, this is certainly useful but the true value of metrics is to serve as the foundation for real-time alerts: the ability to oversee the trends and proactively notify about anomalies or emerging issues. It is so critical and important for real-world production systems that we are going to devote a whole part of the tutorial to talk about alerting.

16.3 Operational vs Application vs Business

There is enormous amount of metrics which could be collected and acted upon. Roughly, they could be split into three classes: operational metrics, application metrics and business metrics.

To put things into perspective, let us focus on **JCG Car Rentals** platform which constitutes of multiple HTTP-based microservices, data storages and message brokers. These components are probably running on some kind of virtual or physical host, very likely inside the container. At minimum, at every layer we would be interested to collect metrics for CPU, memory, disk I/O and network utilization.

In the case of HTTP-based microservices, what we want to be aware of, at minimum, are the following things:

- **Requests Per Second (RPS)** . This is a core metric which indicates how many requests are travelling through the application or service.
- **Response time** . Yet another core metric which shows off how much time it takes to the application or service to respond to the requests.
- **Errors** . This metric indicates the rate of erroneous application or service responses. In case of HTTP protocol, we are mostly interested in `5xx` errors (the server-side ones), however practically `4xx ` errors should not be neglected either.

Those are typical examples of the operational metrics, and to be fair, there are hundreds and hundreds of them. Some are straightforward, others are not. For example, what could be a good, indicative metrics for the message brokers taking into the account the differences in their architectures? Luckily, in majority of cases the vendors and maintainers already took care of exposing and documenting the relevant metrics, and even further, publishing the dashboards and templates to ease up the operations.

So what about the application metrics? As you may guess, those are really dependent on the implementation context and vary. For example, the applications built on top of actor model should expose a number of metrics related to actor system and actors. In the same vein, the Tomcat-based applications may need to expose the metrics related to server thread pools and queues.

The business metrics are essentially intrinsic to each system's domain and vary significantly. For example, for **JCG Car Rentals** platform the important business metric may include the number of reservations over time interval.

16.4 JVM Peculiarities

In the Java world, there is one thing in between operating system and the application: the JVM. It is terrific but equally complex piece of technology which has to be watched out: CPU, heap consumption, garbage collection, metaspace, classloading, off-heap buffers, ... and so on. To our luck, JVM exposes tons of metrics out of the box so it becomes the matter of using them properly.

To generalize this point, always learn the runtime your applications and services are running under and make sure that you have the right metrics in place to understand what is going on.

16.5 Pull or Push?

Depending on the monitoring backend you are using, there two basic strategies how metrics are being gathered from the applications or services: either they are periodically pushed or pulled (scraped). Each of these strategies has own pros and cons (for example, well-known weakness of pull-based strategy is ephemeral and batch jobs which may not exist long enough to be scraped) so please spend some time to understand which one fits the best to your context.

16.6 Storage

As we have touched upon before, storing and querying metrics efficiently requires the use of the dedicated time series database. There are quite a few good choices out there we are going to talk about.

16.6.1 RRDTool

If you are looking for something really basic, the RRDtool (or the longer version, Round Robin Database tool) is probably the one you need.

RRDtool is the OpenSource industry standard, high performance data logging and graphing system for time series data. RRDtool can be easily integrated in shell scripts, perl, python, ruby, lua or tcl applications. - https://oss.oetiker.ch/-rrdtool/

The idea behind round robin databases is quite simple and exploits the circular buffers, thus keeping the system storage footprint constant over time.

16.6.2 Ganglia

Once quite popular, Ganglia is probably the oldest open source monitoring systems out there. Although you may find mentions about Ganglia in the wild, unfortunately it is not actively developed anymore.

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. - https://ganglia.info/

16.6.3 Graphite

Graphite is one of the first open source projects emerged as the full-fledged monitoring tool. It was created back in **2006** but is still being actively maintained.

Graphite is an enterprise-ready monitoring tool that runs equally well on cheap hardware or Cloud infrastructure. Teams use Graphite to track the performance of their websites, applications, business services, and networked servers. It marked the start of a new generation of monitoring tools, making it easier than ever to store, retrieve, share, and visualize time-series data. - https://graphiteapp.org/#overview

Interestingly, the Graphite's storage engine is very similar in design and purpose to round robin databases, such as RRDTool.

16.6.4 OpenTSDB

Some of the time series databases are built on top of more traditional (relation or non-relational) data storage, like for example OpenTSDB, which relies on Apache HBase.

OpenTSDB is a distributed, scalable Time Series Database (TSDB) written on top of HBase . OpenTSDB was written to address a common need: store, index and serve metrics collected from computer systems (network gear, operating systems, applications) at a large scale, and make this data easily accessible and graphable. - https://github.com/OpenTSDB/-opentsdb

16.6.5 TimescaleDB

TimescaleDB is yet another example of the open-source time series database built on top of the proven data store, in this case PostgreSQL.

TimescaleDB is an open-source time-series database optimized for fast ingest and complex queries. It speaks "full SQL" and is correspondingly easy to use like a traditional relational database, yet scales in ways previously reserved for NoSQL databases. - https://docs.timescale.com/latest/introduction

From the development perspective, TimescaleDB is implemented as an extension on PostgreSQL so it basically means running inside the PostgreSQL instance.

16.6.6 KairosDB

KairosDB was originally forked from OpenTSDB but with the time it evolved into independent, promising open-source time series database.

- KairosDB is a fast distributed scalable time series database written on top of Cassandra. - https://github.com/kairosdb/-kairosdb*

16.6.7 InfluxDB (and TICK Stack)

InfluxDB is an open source time series database which is developed and maintained by InfluxData.

InfluxDB is a time series database designed to handle high write and query load - https://www.influxdata.com/products/-influxdb-overview/

InfluxDB is rarely used alone but as the part of more comprehensive platform called the TICK stack, which includes Telegraf, Chronograf and Kapacitor. The next generation of the InfluxDB, currently in alpha, intends to unify this time series platform in a single redistributable binary.

16.6.8 Prometheus

These days Prometheus is the number one choice as the metrics, monitoring and alerting platform. Besides its simplicity and ease of deployment, it natively integrates with container orchestrators like Kubernetes for example.

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud . - https://prometheus.io/-docs/introduction/overview/

In 2016, Prometheus joined the Cloud Native Computing Foundation (CNCF). For the **JCG Car Rentals** platform, Prometheus is going be an obvious pick to collect, store and query metrics. In case of simple static Prometheus configuration (with static IP addresses), here is how the subset of the **JCG Car Rentals** platform services is shown up on the **Targets** web page.

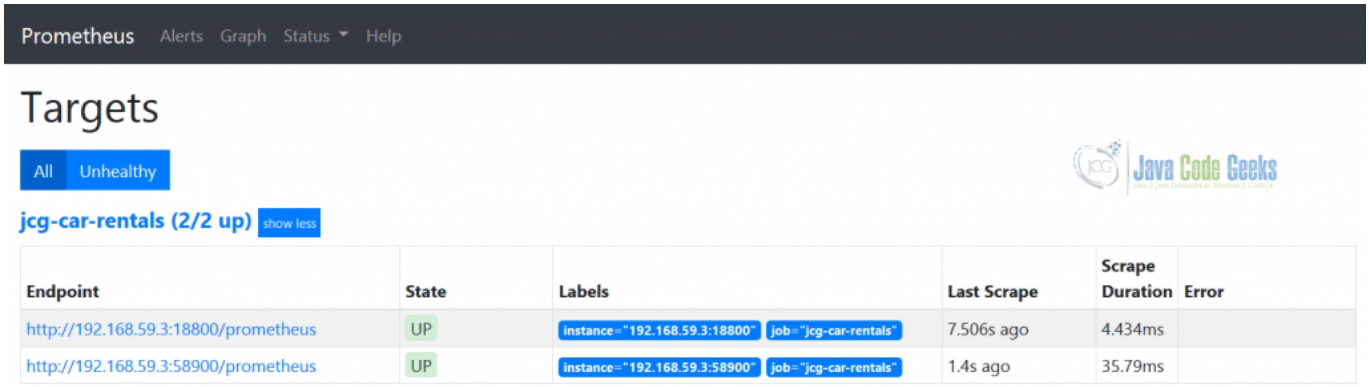


Figure 16.1: Image

16.6.9 Netflix Atlas

Atlas was born (and open-sourced later) at Netflix, driven by the need to cope with increased number of metrics which has to be collected by its streaming platform.

Atlas was developed by Netflix to manage dimensional time series data for near real-time operational insight. Atlas features in-memory data storage, allowing it to gather and report very large numbers of metrics, very quickly. - https://github.com/Netflix/atlas/wiki

It is a great system but please keep in mind that the choice to use in-memory data storage is one of Atlas's sore points and may incur additional costs.

16.7 Instrumentation

The choice of framework plays an important role in order to facilitate the instrumentation of the applications and services. For example, since **Reservation Service** is using Spring Boot, there is out of the box support for standard set of metrics for web servers and web clients, baked by Micrometer.

```
management:
  endpoint:
    prometheus:
      enabled: true
    metrics:
      enabled: true
  metrics:
    distribution:
      percentiles-histogram:
        http.server.requests: true
    export:
      prometheus:
        enabled: true
    web:
      client:
        max-uri-tags: 150
        requests-metric-name: http.client.requests
      server:
        auto-time-requests: true
        requests-metric-name: http.server.requests
```

And even more, Spring Boot comes with the handy customizers to enrich the metrics with additional configuration and metadata (labels or/and tags).

```
@Configuration
public class MetricsConfiguration {
    @Bean
    MeterRegistryCustomizer<MeterRegistry> metricsCommonTags(@Value("${spring.application. ↩
        name}") String application) {
        return registry -> registry.config().commonTags("application", application);
    }
}
```

The integration with Prometheus, the monitoring choice of the **JCG Car Rentals** platform, is also seamless and is bundled with Micrometer.

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

From the other side, the **Customer Service** uses Dropwizard Metrics and needs a bit of customization to collect and expose the desired metrics in accordance to Prometheus protocol.

```
@ApplicationScoped
public class PrometheusServletProvider implements ServletContextAttributeProvider{
    @Inject private MetricsConfig metricsConfig;

    @PostConstruct
    public void init() {
        CollectorRegistry.defaultRegistry.register(new DropwizardExports(metricsConfig. ↩
            getMetricRegistry()));
        DefaultExports.initialize();
    }
```

```
    @Produces
    public ServletDescriptor prometheusServlet() {
        String[] uris = new String[]{"/prometheus"};
        WebInitParam[] params = null;
        return new ServletDescriptor("Prometheus", uris, uris, 1, params, false,  ↩
            MetricsServlet.class);
    }

    @Override
    public Map<String, Object> getAttributes() {
        return Collections.emptyMap();
    }
}
```

16.7.1 Statsd

Outside of pure JVM-specific options, statsd would be the one worth mentioning. Essentially, it is a front-end proxy for different metric backends.

A network daemon that runs on the Node.js platform and listens for statistics, like counters and timers, sent over UDP or TCP and sends aggregates to one or more pluggable backend services (e.g., Graphite). - https://github.com/statsd/statsd

There is a large number of client implementations available, thereafter positioning statsd as a very appealing choice for polyglot microservice architectures.

16.7.2 OpenTelemetry

As we have seen so far, there are quite a lot of opinions on how the metrics instrumentation and collection should be done. Recently, the new industry-wide initiative has been announced under OpenTelemetry umbrella.

OpenTelemetry is made up of an integrated set of APIs and libraries as well as a collection mechanism via an agent and collector. These components are used to generate, collect, and describe telemetry about distributed systems. This data includes basic context propagation, distributed traces, metrics, and other signals in the future. OpenTelemetry is designed to make it easy to get critical telemetry data out of your services and into your backend(s) of choice. For each supported language it offers a single set of APIs, libraries, and data specifications, and developers can take advantage of whichever components they see fit. - https://opentelemetry.io/

The goals of OpenTelemetry go way beyond metrics, and we are going to talk about some of them more in the upcoming parts of the tutorial. As of now, the OpenTelemetry is available as specification draft only. But if you would like to give it a try, it is based on well-known OpenSensus project, which also includes metrics instrumentation.

16.7.3 JMX

For JVM applications, there is yet another way to expose real-time metrics, using Java Management Extensions (JMX). To be fair, JMX is quite old technology and you may find it awkward to use, however it is probably the simplest and fastest way to get insights about your JVM-based applications and services.

The standard way to connect to the JVM applications over JMX is to use JConsole, JVisualVM or the newest way, using JDK Mission Control (JMC). For example, the screenshot below illustrates JVisualVM in action, which visualizes the Apache Cassandra's **requests** metric exposed by **Reservation Service** over JMX.

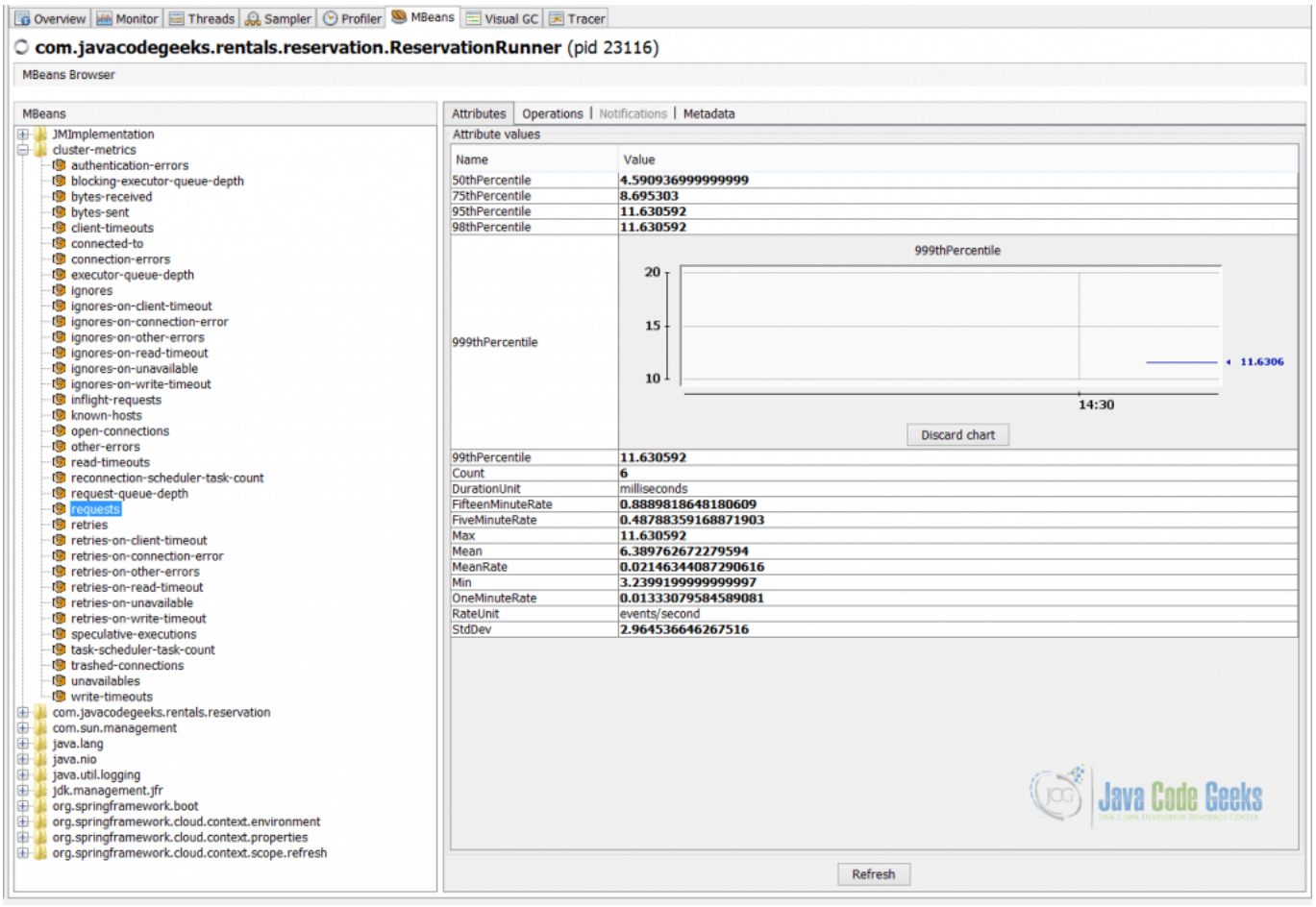


Figure 16.2: Image

The metrics exposed through JMX are ephemeral and available only during the time when applications and services are up and running (to be precise, persistence is optional, non portable and is rarely used). Also, please keep in mind that the scope of JMX is not limited to metrics but management in general.

16.8 Visualization

As we already understood, the typical JVM application or service exposes a lot of metrics. Some of them are rarely useful whereas others are critical indicators of the application or service health. What means do we have to make this distinction obvious and, more importantly, meaningful and useful? One of the answers is visualization and construction of the real-time operational or/and business dashboards.

The monitoring and metrics management platforms like Graphite, Prometheus and InfluxDB do support quite sophisticated query languages and graphs so you may not even dig further. But in case you are looking for building state of the art dashboards or consolidating over multiple metric sources, you would need to search around.

16.8.1 Grafana

Undoubtedly, as of today Grafana is a one stop shop for metrics visualization and creating truly beautiful dashboards (with a large number of pre-built ones already available).

Grafana is the leading open source project for visualizing metrics. Supporting rich integration for every popular database like Graphite, Prometheus and InfluxDB. - https://grafana.com/ For **JCG**

For **JCG Car Rentals** platform, Grafana fits exceptionally well since it has outstanding integration with Prometheus. In case of **Reservation Service**, which is using Micrometer library, there are a few community built dashboards to get you started quickly, one to them is shown below.

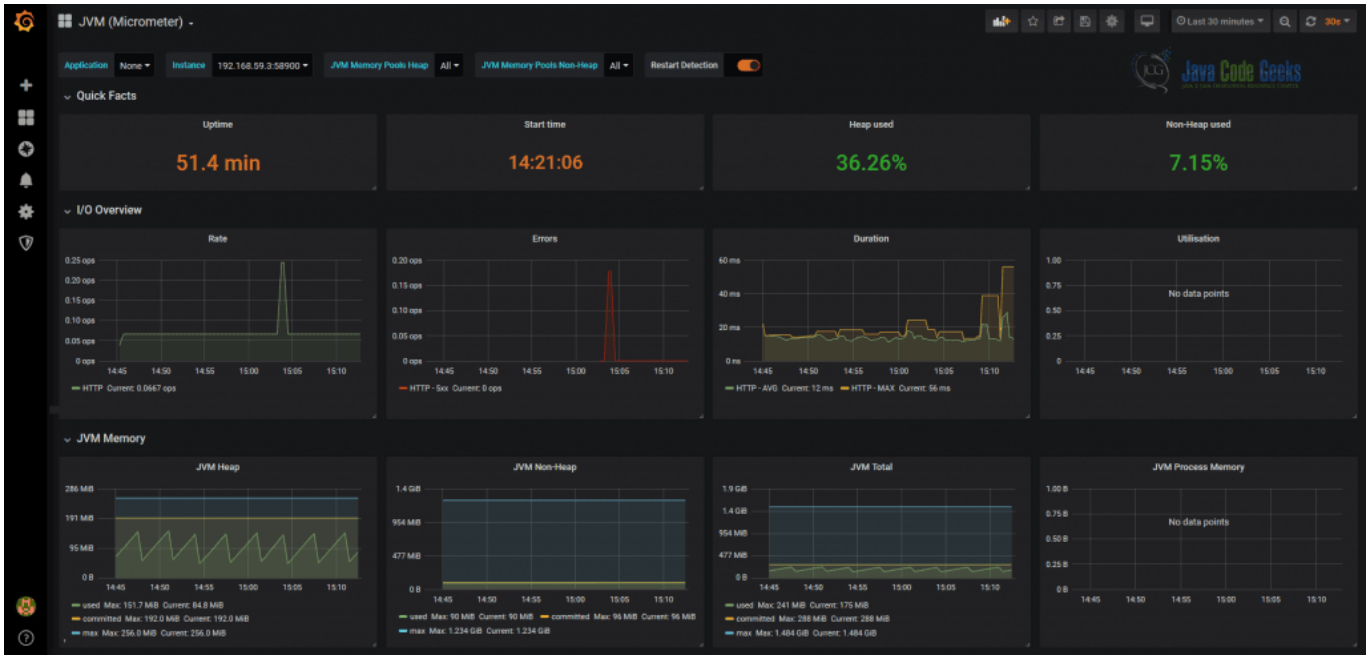


Figure 16.3: Image

It is worth to emphasize that Grafana is highly customizable and extensible, so if you make a choice to use it as your metrics visualization platform, it is unlikely this decision is going to be regretted in the future.

16.9 Cloud

For the applications and services deployed in the cloud, the importance of the metrics (and alerting, more on that in the upcoming part of the tutorial) is paramount. The pattern you will discover quickly is that the metrics management comes along with the same offerings we have talked about in the previous part of the tutorial, so let us quickly glance over them.

If you are running applications, services, API gateways or functions on AWS, the Amazon CloudWatch automatically collects and tracks a large amount of metrics (as well as other operational data) on your behalf without any additional configuration (including the infrastructure). In addition, if you are looking just for a storage part, it is certainly worth exploring Amazon Timestream, a fast, scalable, fully managed time series database offering.

The Microsoft Azure's offering for metrics collection and monitoring is a part of the Azure Monitor data platform.

Similarly to others, Google Cloud does not have standalone offering just for metrics management but bundles it along with Stackdriver Monitoring, part of a Stackdriver offering.

16.10 Serverless

The most significant mindset shift for serverless workloads is that the metrics related to the host systems are not your concern anymore. From the other side, you need to understand what kinds of metrics are relevant in serverless world and collect those. So what are they?

- **Invocation Duration** . The distribution of the function execution times (since this is what you primarily pay for).

- **Invocations Count** . How many times the function was invoked.
- **Erroneous Invocations Count** . How many times the function did not complete successfully.

Those are a good starting point however the most important metrics will be the business or application ones, intrinsic to what each function should be doing.

Most of the cloud providers collect and visualize metrics for their serverless offerings and the good news are that the popular open source serverless platforms like jazz, Apache OpenWhisk, OpenFaas, Serverless Framework come with at least basic instrumentation and expose a number of metrics out of the box as well.

16.11 What is the Cost?

Up to now, we have been focused on the importance of the metrics to gather the insights, oversee trends and patterns. However, we have not talked about the cost of doing that, both from storage and computational perspectives.

It is difficult to come up with the universal cost model, but there are a number of factors and trade-offs to consider. The most important ones are:

- The total number of metrics.
- The number of distinct time series which exists per particular metric.
- The backend storage (for example, keeping all data in memory is expensive, disk is much cheaper option).
- Collecting raw metrics versus pre-aggregated ones.

Another risk you might face is related to running queries and aggregations over large amount of time series. In most cases, this is very expensive operation, and it is better to plan the capacity ahead of time if you really need to support that.

As you may guess, when left adrift, things may get quite expensive.

16.12 Conclusions

In this part of the tutorial we have talked about metrics, another pillar of the observability. Metrics and logs constitute the absolutely required foundation for every distributed system built after microservice architecture. We have learned how applications and services are instrumented, how metrics are collected and stored, and last but not least, how they could be represented in a human-friendly way using dashboards (the alerting piece will come after).

To finish up, it would be fair to say that our focus was primarily pointed towards metrics management platforms and not analytics ones, like Apache Druid or ClickHouse, or monitoring ones, like Nagios or Hawkular (although there are some intersections here). Nonetheless please stay tuned, we are going to get back to broader monitoring and alerting subject in the last part of the tutorial.

16.13 What's next

In the next part of the tutorial we are going to talk about distributed tracing.

Chapter 17

Distributed Tracing

17.1 Introduction

This part of the tutorial is going to conclude the observability discussions by dissecting its last pillar, distributed tracing.

Distributed tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance. - https://opentracing.io/docs/overview/what-is-tracing/

In distributed systems, like a typical microservice architecture, the request could travel through dozens or even hundreds of services before the response is assembled and sent back. But how are you supposed to know that? At some extent, logs are able to provide these insights, but they are inherently flat: it becomes difficult to understand the causality between calls or events, extract latencies, and reconstruct the complete path the request has taken through the system. This is exactly the case where distributed tracing comes to the rescue.

The story of the distributed tracing (as we know it these days) started in **2010**, when Google published the famous paper Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Although Dapper was never open-sourced, the paper has served as an inspirational blueprint for a number of the open source and commercial projects designed after it. So let us take a closer look at distributed tracing.

17.2 Instrumentation + Infrastructure = Visualization

Before we dig into the details, it is important to understand that even though distributed tracing is terrific technology, it is not magical. Essentially, it consists of three key ingredients:

- **Instrumentation** : language-specific libraries which help to enrich the applications and services with tracing capabilities.
- **Infrastructure** : a tracing middleware (collectors, servers, ...) along with the store engine(s) where traces are being sent, collected, persisted and become available for querying later on.
- **Visualization** : the frontends for exploring, visualizing and analyzing collected traces.

What it practically means is that rolling out distributed tracing support across a microservices fleet requires not only development work but also introduces operational overhead. Essentially, it becomes yet another piece of infrastructure to manage and monitor. The good news is, it is out of the critical path, most of the instrumentation libraries are designed to be resilient against tracing middleware outages. At the end, the production flows should not be impacted anyhow, although some traces might be lost.

For many real-world applications, recording and persisting the traces for every single request could be prohibitively expensive. For example, it may introduce non-negligible overhead in the systems highly optimized for performance, or put a lot of pressure on the storage in the case of systems with very high volume of requests. To mitigate the impact and still get useful insights, different sampling techniques are widely used.

17.3 TCP, HTTP, gRPC, Messaging, . . .

One of the challenges which modern distributed tracing implementations face is the wide range of communication means employed by microservice architectures (and distributed systems in general). The context propagation strategies are quite different not only because of protocols, but communication styles as well. For example, adding tracing instrumentation for the services which use request / response communication over HTTP protocol is much more straightforward than instrumenting Apache Kafka producers and consumers or gRPC services.

The distributed tracing as a platform works across programming languages and runtime boundaries. The only language-specific pieces are the instrumentation libraries which bridge applications, services and distributed tracing platforms together. Most luckily, as it stands today, the tracing instrumentation you are looking for is already available, either from community, vendors or maintainers. However, in rare circumstances, especially when using the cutting edge technologies, you may need to roll your own.

17.4 OpenZipkin

Zipkin is one of the first open source projects implemented after Dapper paper by Twitter engineers. It quickly got a lot of traction and soon after changed home to OpenZipkin.

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data. - https://github.com/openzipkin/zipkin

By all means, Zipkin is the leading distributed tracing platform these days, with a large number of integrations available for many different languages. The **JCG Car Rentals** platform is going to use Zipkin to collect and query the traces across all its microservices.

Let us have a sneak-peak on typical integration flow. For example, in case of the **Payment Service**, which we have decided to implement in Go, we could use zipkin-go instrumentation.

```
reporter := httpreporter.NewReporter("https://localhost:9411/api/v2/spans"))
defer reporter.Close()

// create our local service endpoint
endpoint, err := zipkin.NewEndpoint("payment-service", "localhost:29080")
if err != nil {
    log.Fatalf("unable to create local endpoint: %+v\\n", err)
}

tracer, err := zipkin.NewTracer(reporter, zipkin.WithLocalEndpoint(localEndpoint))
if err != nil {
    log.Fatalf("unable to create tracer: %+v\\n", err)
}
```

Not only zipkin-go provides the necessary primitives, it also has outstanding instrumentation capabilities for gRPC-based services, as **Payment Service** is.

```
func Run(ctx context.Context, tracer *zipkin.Tracer) *grpc.Server {
    s := grpc.NewServer(grpc.StatsHandler(zipkingrpc.NewServerHandler(tracer)))
    payment.RegisterPaymentServiceServer(s, newPaymentServer())
    reflection.Register(s)
    return s
}
```

17.5 OpenTracing

Zipkin was among the first but the number of different distributed tracing platform inspired by its success started to grow quickly, with each one promoting own APIs. OpenTracing initiative has emerged early on as an attempt to establish the common ground among all these implementations.

OpenTracing is comprised of an API specification, frameworks and libraries that have implemented the specification, and documentation for the project. OpenTracing allows developers to add instrumentation to their application code using APIs that do not lock them into any one particular product or vendor. - https://opentracing.io/docs/overview/what-is-tracing/

Luckily, the benefits of such the effort were generally understood and as of today the list of distributed tracers which support OpenTracing includes mostly every major player.

17.6 Brave

Brave is one of the most widely employed tracing instrumentation library for JVM-based applications which is typically used along with OpenZipkin tracing platform.

Brave is a distributed tracing instrumentation library. Brave typically intercepts production requests to gather timing data, correlate and propagate trace contexts. - https://github.com/openzipkin/brave

The amount of instrumentations provided by Brave out of the box is very impressive. Although it could be integrated directly, many libraries and frameworks introduce the convenient abstractions on top of Brave to simplify the idiomatic instrumentation. Let us take a look what that means for different **JCG Car Rentals** services.

Since **Reservation Service** is built on top of Spring Boot, it could benefit from outstanding integration with Brave provided by Spring Cloud Sleuth.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

Most of the integration settings could be tuned through configuration properties.

```
spring:
  sleuth:
    enabled: true
    sampler:
      probability: 1.0
  zipkin:
    sender:
      type: WEB
    baseUrl: https://localhost:9411
    enabled: true
```

From the other side, the **Customer Service** uses native Brave instrumentation, following Project Hammock conventions. Below is a code snippet to illustrate how it could be configured.

```
@ApplicationScoped
public class TracingConfig {
    @Inject
    @ConfigProperty(name = "zipkin.uri", defaultValue = "https://localhost:9411/api/v1/ ↩
        spans")
    private String uri;

    @Produces
    public Brave brave() {
        return new Brave.Builder("customer-service")
            .reporter(AsyncReporter.create(OkHttpSender.create(uri)))
            .traceSampler(Sampler.ALWAYS_SAMPLE)
```

```
                .build();
    }

    @Produces
    public SpanNameProvider spanNameProvider() {
        return new DefaultSpanNameProvider();
    }
}
```

The web frontends which come as part of Zipkin server distribution allow to visualize individual traces across all participating microservices.

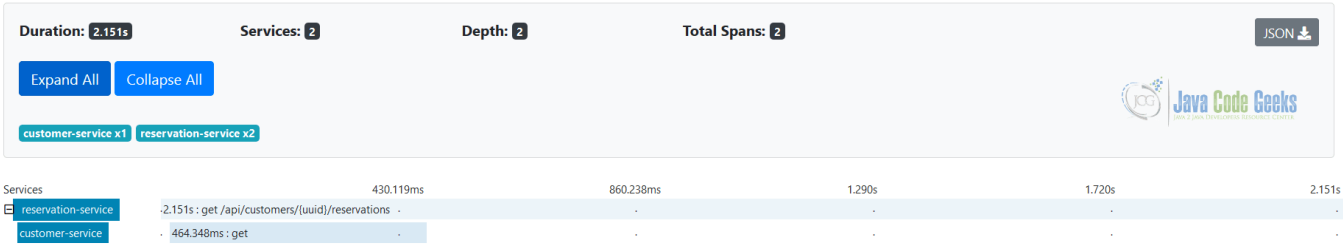


Figure 17.1: Traces Reservation and Customer services

Obviously, the most useful application of distributed tracing platforms is to speed up troubleshooting and problems detection. The right visualization plays a very important role here.

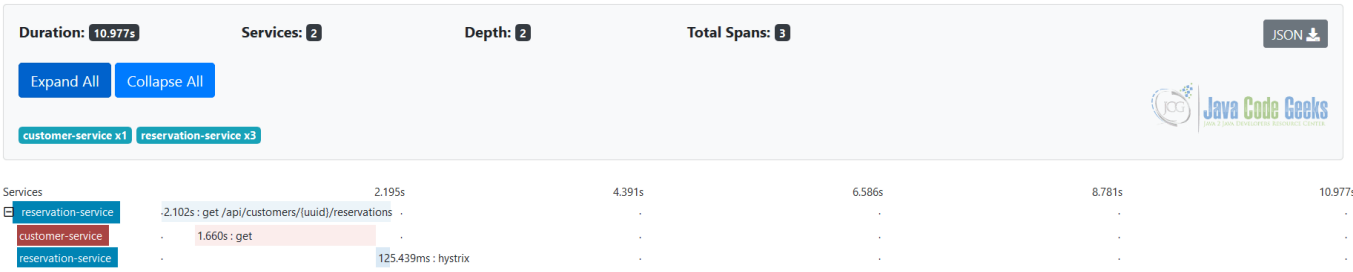


Figure 17.2: An issue between Reservation and Customer services is shown in the trace

Last but not least, like many other distributed tracing platforms Zipkin continues to evolve and innovate. One of the recent additions to its tooling is a new alternative web frontend called Zipkin Lens, shown on the picture below.

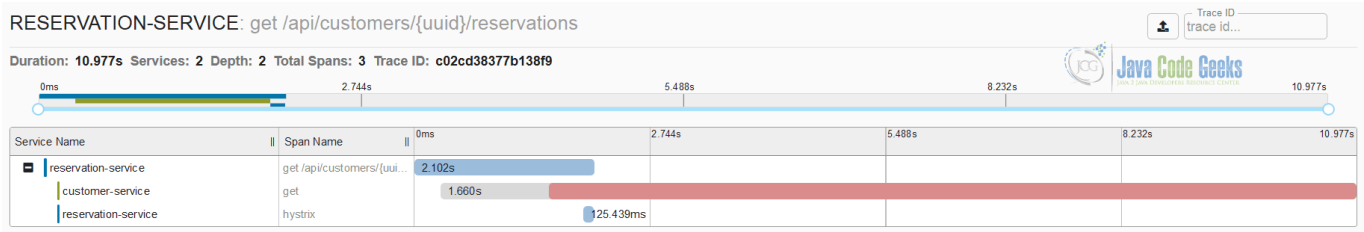


Figure 17.3: Reservation and Customer services through Zipkin Lens

17.7 Jaeger

Jaeger is yet another popular distributed tracing platform which was developed at Uber and open sourced later on.

Jaeger , inspired by Dapper and OpenZipkin , is a distributed tracing system released as open source by Uber Technologies . It can be used for monitoring microservices-based distributed systems - https://github.com/jaegertracing/jaeger

Besides being hosted under the Cloud Native Computing Foundation (CNCF) umbrella, Jaeger natively supports OpenTracing specification and also provides backwards compatibility with Zipkin. What it practically means is that the instrumentation we have done for **JCG Car Rentals** services would seamlessly work with Jaeger tracing platform.

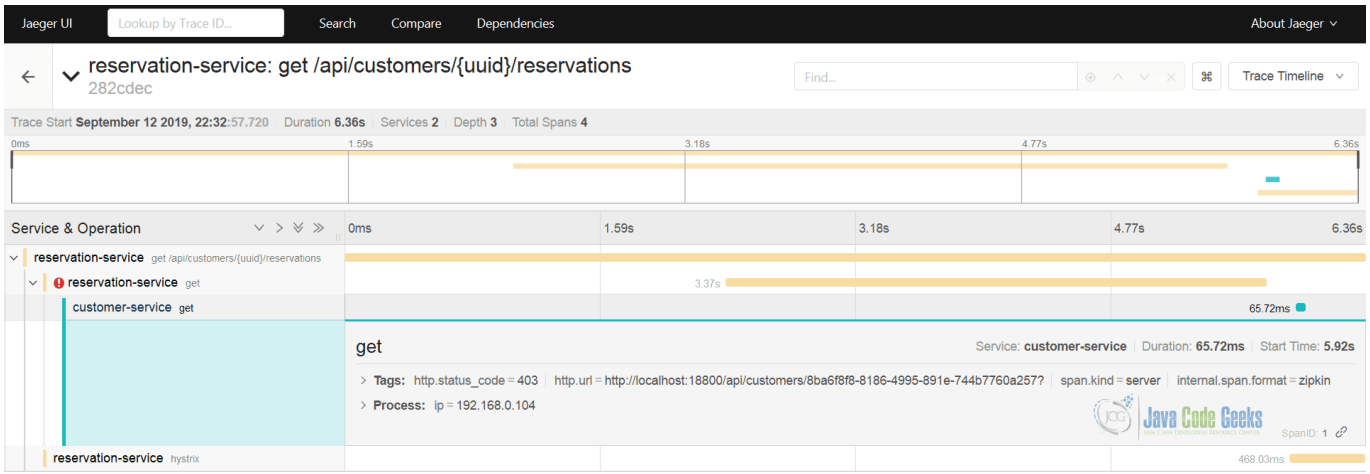


Figure 17.4: Reservation and Customer services in Jaeger

17.8 OpenSensus

OpenCensus originates from Google where it was used to automatically capture traces and metrics from the massive amount of services.

OpenCensus is a set of libraries for various languages that allow you to collect application **metrics** and **distributed traces**, then transfer the data to a backend of your choice in real time. - **https://opencensus.io/**

By and large, OpenCensus is an instrumentation layer only which is compatible (among many others) with Jaeger and Zipkin tracing backends.

17.9 OpenTelemetry

We have talked about the OpenTelemetry initiative in the previous part of the tutorial, just touching the metrics subject only. But truth to be told, OpenTelemetry is result of combining the efforts of two well-established projects, Jaeger and OpenSensus, under one umbrella, delivering a full-fledged, robust, and portable telemetry platform.

The leadership of OpenTracing and OpenCensus have come together to create OpenTelemetry, and it will supersede both projects. - https://opentelemetry.io/

As of the moment of this writing, the work around first official release of OpenTelemetry is still in progress but the early bits are on the plan to be available very soon.

17.10 Haystack

Haystack, born at Expedia, is an example of the distributed tracing platform which goes beyond just collecting and visualizing traces. It focuses on the analysis of the operation trends, service graphs and anomaly detection.

Haystack is an Expedia -backed open source distributed tracing project to facilitate detection and remediation of problems in microservices and websites. It combines an OpenTracing -compliant trace engine with a componentized backend architecture designed for high resiliency and high scalability. Haystack also includes analysis tools for visualizing trace data, tracking trends in trace data, and setting alarms when trace data trends exceed limits you set. - https://expediadotcom.github.io/haystack/docs/about/introduction.html

Haystack is a modular platform, which could be used in parts or as a whole. One of the exceptionally useful and powerful components of it is Haystack UI. Even if you don't use Haystack yet, you could use Haystack UI along with Zipkin as a drop-in replacement of its own frontend.

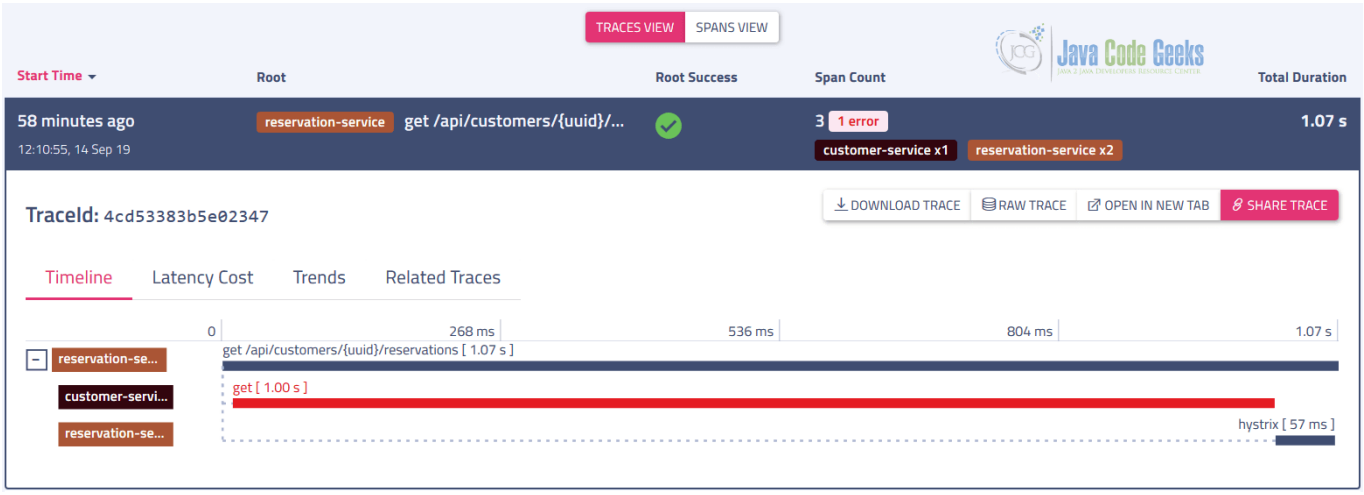


Figure 17.5: Reservation and Customer services trace in Haystack UI

When used with Zipkin only, not all components are accessible but even in that case a lot of analytics is made available out of the box.

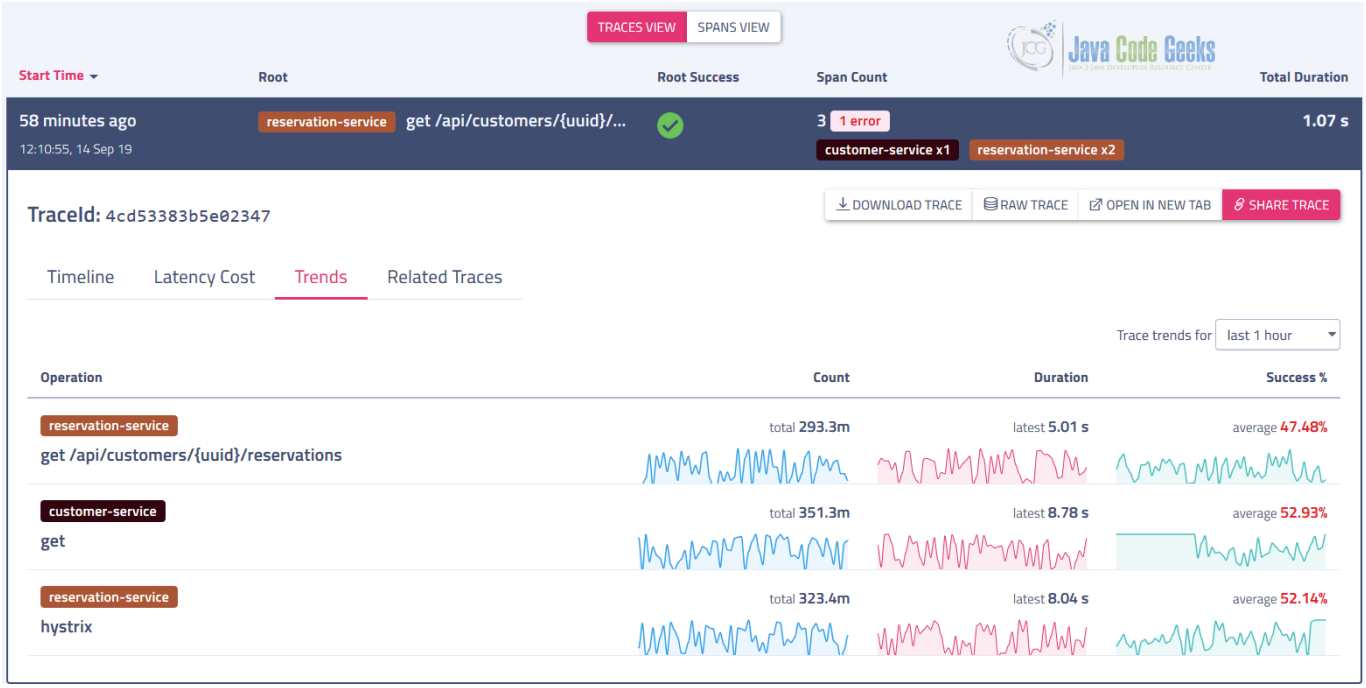


Figure 17.6: Trends in Haystack UI

Haystack is probably the most advanced open-source distributed tracing platforms at the moment. We have seen just a small subset of what is possible yet another its feature, adaptive alerting, is going to come back in the next part of the tutorial.

17.11 Apache SkyWalking

Apache SkyWalking is yet another great example of the mature open-source observability platform, where distributed tracing plays a key role.

SkyWalking : an open source observability platform to collect, analyze, aggregate and visualize data from services and cloud native infrastructures. - https://github.com/apache/skywalking/blob/master/docs/en/concepts-and-designs/overview.md

It is worth noting that Apache SkyWalking instrumentation APIs are fully compliant with the OpenTracing specification. On backend level, Apache SkyWalking also supports integration with Zipkin and Jaeger, although some limitations apply.

In the case of **JCG Car Rentals** platform, replacing Zipkin with Apache SkyWalking is seamless and all existing instrumentations continue to functional as expected.

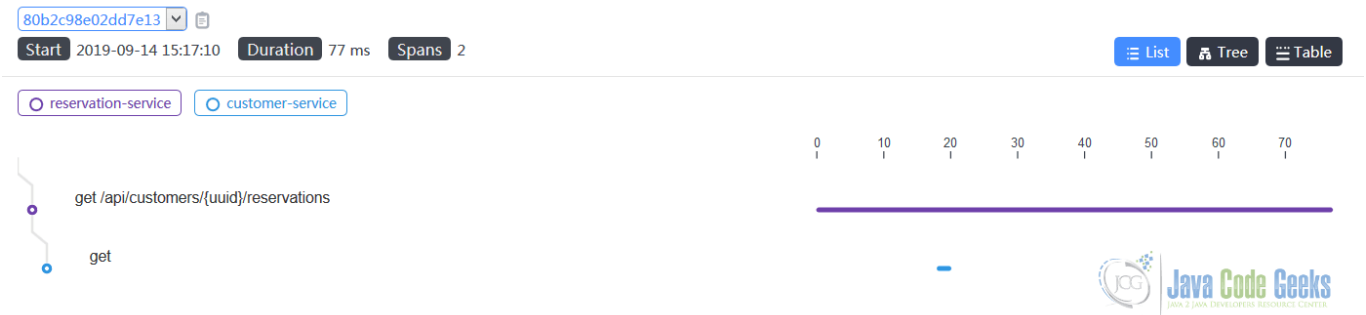


Figure 17.7: Reservation and Customer services trace in Apache SkyWalking

17.12 Orchestration

As we have discussed awhile back, the orchestrators and service meshes are deeply penetrating into the deployment of modern microservice architectures. Being invisible and just do the job is the mojo behind service meshes. But when things go wrong, it is critical to know if the service mesh or the orchestrator is the culprit.

Luckily, every major service mesh is built and designed with observability in mind, incorporating all three pillars: logs, metrics and distributed tracing. Istio is a true leader here and comes with Jaeger or/and Zipkin support, whereas Linkerd provides only some of the features that are often associated with distributed tracing. From the other side, Consul with Connect purely relies on Envoy's distributed tracing capabilities and does not go beyond that.

The context propagation from the service mesh up to the individual services enables to see the complete picture of how the request travels through the system, from the moment one has entered it to the moment last byte of the response has been sent.

17.13 The First Mile

As you might have noticed, the distributed tracing is often narrowed to the context of backend services or server-side applications; frontends are almost entirely ignored. Such negligence certainly removes some important pieces from the puzzle since in most cases the frontends are exactly the place where most server-side interactions are being initiated. There is even a W3C specification draft called Trace Context to address this gap so why is that?

The instrumentation of the JavaScript application is provided by many distributed tracing platform, for example, OpenSensus has one, so does OpenZipkin and OpenTracing. But any of those require some pieces of distributed tracing infrastructure to be publicly available to actually collect the traces sent from the browser. Although such practices are widely accepted for analytics data for example, it still poses security and privacy concerns since quite often traces indeed may contain sensitive information.

17.14 Cloud

The integration of the distributed tracing in cloud-based deployments used to be a challenge but these days most of the cloud providers have dedicated offerings.

We are going to start with AWS X-Ray which provides an end-to-end view of requests as they travel through the system and shows a map of the underlying components. In general, the applications and services should use X-Ray SDKs for distributed tracing instrumentation but some platforms, like OpenZipkin or OpenSensus to name a few, have the extensions to integrate with AWS X-Ray.

In the Google Cloud, distributed tracing is fulfilled by Stackdriver Trace, the member of the Stackdriver observability suite. The language-specific SDKs provide low-level interfaces for interacting directly with the Stackdriver Trace but you have the option to make use of OpenSensus or Zipkin instead.

The distributed tracing in Microsoft Azure is backed by Application Insights, part of a larger Azure Monitor offering. It also provides the dedicated Application Insights SDKs which applications and services should integrate with to unleash distributed tracing capabilities. What comes as a surprise is that Application Insights also supports distributed tracing through OpenSensus.

As you may see, every cloud provider have an own opinion regarding distributed tracing and in most cases you have no choice as to use their SDKs. Thankfully, the leading open source distributed tracing platforms take this burden off from you by maintaining such integrations.

17.15 Serverless

More and more organizations are looking towards serverless computing, either to cut the costs or to accelerate the rollout of the new features or offerings. The truth is that serverless systems scream for observability, otherwise troubleshooting the issues become more like searching for a needle in the haystack. It can be quite difficult to figure out where, in a highly distributed serverless system, things went wrong, particularly in the case of cascading failures.

This is the niche where distributed tracing truly shines and is tremendously helpful. The cloud-based serverless offerings are backed by provider-specific distributed tracing instrumentations, however the open source serverless platforms are trying to catch up here. Notably, Apache OpenWhisk comes with OpenTracing integration whereas Knative is using Zipkin. For others, like OpenFaas or Serverless, you may need to instrument your functions manually at the moment.

17.16 Conclusions

In this part of the tutorial we have talked about the third observability pillar, distributed tracing. We have covered only the necessary minimum however there are tons of materials to read and to watch on the subject, if you are interested to learn it further.

These days, there are a lot of innovations happening in the space of the distributed tracing. The new interesting tools and integrations are in work (traces comparison, latency analysis, JFR tracing, ...) and hopefully we are going to be able to use them in production very soon.

17.17 What's next

In the next, the final part of the tutorial, we are going to talk about monitoring and alerting.

Chapter 18

Monitoring and Alerting

18.1 Introduction

In this last part of the tutorial we are going to talk about the topic where all the observability pillars come together: monitoring and alerting. For many, this subject belongs strictly to operations and the only way you know it is somehow working is when you are on-call and get pulled in.

The goal of our discussion is to demystify at least some aspects of the monitoring, learn about alerts, and understand how metrics, distributed traces and sometimes even logs are being used to continuously observe the state of the system and notify about upcoming issues, anomalies, potential outages or misbehavior.

18.2 Monitoring and Alerting Philosophy

There are tons of different metrics which could (and should) be collected while operating a more or less realistic software system, particularity designed after microservice architecture principles. In this context, the process for collecting and storing such state data is usually referred as monitoring.

So what exactly should you monitor? To be fair, it is not easy to come up upfront with all the possible aspects of the system which have to be monitored and, as such, to decide which metrics (and other signals) you need to collect and which ones you do not, but the golden rule "more data is better than no data" certainly applies here. The ultimate goal is when things go wrong the monitoring subsystem should let you know right away. This is what alerting is all about.

Alert messaging (or alert notification) is machine-to-person communication that is important or time sensitive.

https://en.wikipedia.org/wiki/Alert_messaging

Obviously, you could alert on anything but there are certain rules you are advised to follow while defining your own alerts. The best summary regarding the alerting philosophy is laid out in these excellent articles, Alerting Philosophy by Netflix and My Philosophy on Alerting by Rob Ewaschuk. Please try to find the time to go over these resources, the insights presented in there are priceless.

To summarize some best practices, when an alert triggers, it should be easy to understand why, so keeping the alerts rules as simple as possible is a good idea. Once the alert sets off someone should be notified and look into it. As such, the alerts should indicate the real cause, be actionable and meaningful, the noisy ones should be avoided at all cost (and they will be ignored anyway).

Last but not least, no matter how many metrics you collect, how many dashboards and alerts you have had configured, there would be always something missed. Please consider this process to be a continuous improvement, reevaluate periodically your monitoring, logging, distributed tracing, metrics collection and alerting decisions.

18.3 Infrastructure Monitoring

The monitoring of the infrastructure components and layers is somewhat a solved problem. From the open-source perspective the well-established names like Nagios, Zabbix, Riemann, OpenNMS and Icinga are ruling there and it is very likely that your operations team is already betting on one of those.

18.4 Application Monitoring

The infrastructure certainly falls into the "must be monitored" category but the application side of monitoring is arguably much more interesting and closer to the subject. So let us direct the conversation towards that.

18.4.1 Prometheus and Alertmanager

We have talked about Prometheus already, primarily as a metrics storage, but the fact is that it also includes the alerting component called AlertManager makes it come back.

The AlertManager handles alerts sent by client applications such as the Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integrations such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts. - https://prometheus.io/docs/alerting/alertmanager/

Actually, AlertManager is a standalone binary process which handles alerts sent by Prometheus server instance. Since the **JCG Car Rentals** platform has chosen Prometheus as the metrics and monitoring platform, it becomes a logical choice to manage the alerts as well.

Basically, there are a few steps to follow. The procedure consists of configuring and running the instance of AlertManager, configuring Prometheus to talk to this AlertManager instance and finally defining the alert rules in the Prometheus. Taking one step at a time, let us start off with AlertManager configuration first.

```
global:
  resolve_timeout: 5m
  smtp_smarthost: 'localhost:25'
  smtp_from: 'alertmanager@jcg.org'
route:
  receiver: 'jcg-ops'
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 1h
  group_by: [cluster, alertname]
  routes:
  - receiver: 'jcg-db-ops'
    group_wait: 10s
    match_re:
      service: postgresql|cassandra|mongodb
receivers:
- name: 'jcg-ops'
  email_configs:
  - to: 'ops-alerts@jcg.org'
- name: 'jcg-db-ops'
  email_configs:
  - to: 'db-alerts@jcg.org'
```

If we supply this configuration snippet to the AlertManager process (usually by storing it in the `alertmanager.yml`), it should start successfully, exposing its web frontend at port `9093` .

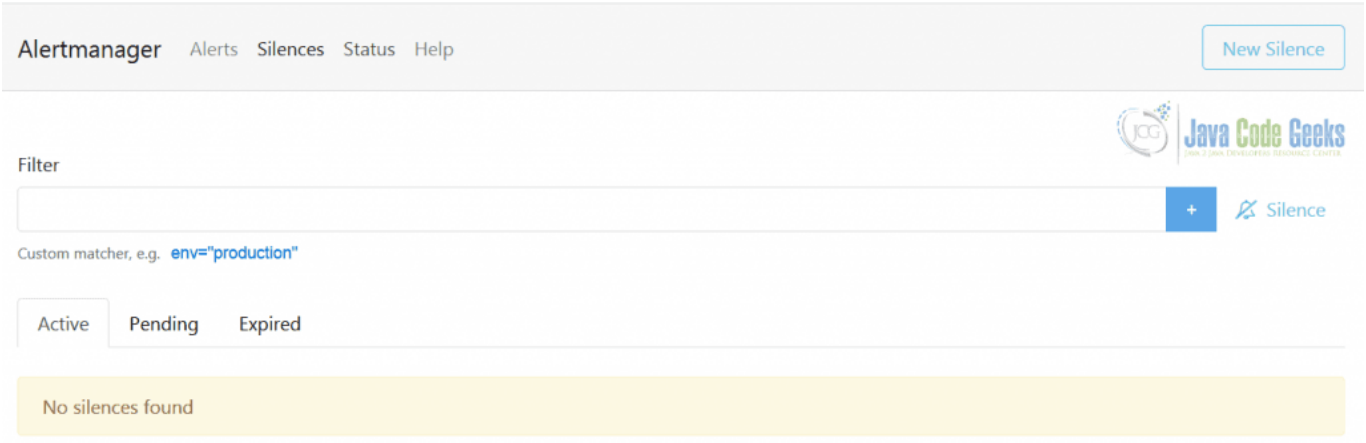


Figure 18.1: Image

Excellent, now we have to tell Prometheus where to look for AlertManager instance. As usual, it is done through configuration file.

```
rule_files:
  - alert.rules.yml

alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - 'alertmanager:9093'
```

The snippet above also includes the most interesting part, the alert rules, and this is what we are going to loot at next. So what would be a good, simple and useful example of meaningful alert in the context of **JCG Car Rentals** platform? Since most of the **JCG Car Rentals** services are run on JVM, the one which comes to mind first is heap usage: getting too close to the limit is a good indication of a trouble and possible memory leak.

```
groups:
- name: jvm
  rules:
  - alert: JvmHeapIsFillingUp
    expr: jvm_memory_used_bytes{area="heap"} / jvm_memory_max_bytes{area="heap"} > 0.8
    for: 5m
    labels:
      severity: warning
    annotations:
      description: 'JVM heap usage for {{ $labels.instance }} of job {{ $labels.job }} is ↩
          close to 80% for last 5 minutes.'
      summary: 'JVM heap for {{ $labels.instance }} is filling up'
```

The same alert rules could be seen in Prometheus using the **Alerts** view, confirming that the configuration has been picked up properly.

Figure 18.2: Image

Once the alert triggers, is it going to appear in the AlertManager immediately, at the same time notifying all affected recipients (the receivers). On the picture below you could see the example of the triggered `JvmHeapIsFillingUp` alert.

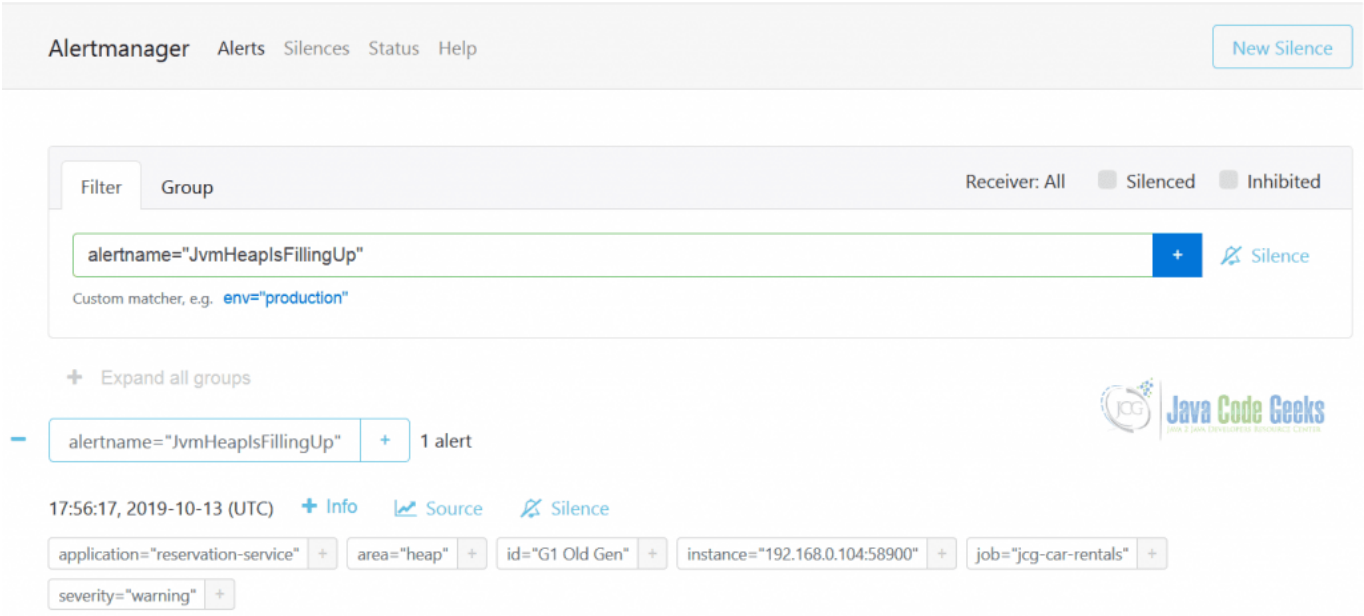


Figure 18.3: Image

As you may agree at this point, Prometheus is indeed a full-fledged monitoring platform, covering you not only from the metrics collection perspective, but the alerting as well.

18.4.2 TICK Stack: Chronograf

If the TICK stack sounds familiar to you that is because it popped up on our radar in the previous part of the tutorial. One of the components of the TICK stack (which corresponds to letter **C** in the abbreviation) is Chronograf.

Chronograf provides a user interface for Kapacitor - a native data processing engine that can process both stream and batch data from InfluxDB. You can create alerts with a simple step-by-step UI and see your alert history in Chronograf.

https://www.influxdata.com/time-series-platform/chronograf/

The InfluxDB 2.0 (still in alpha), the future of the InfluxDB and TICK stack in general, will incorporate Chronograf into its time series platform.

18.4.3 Netfix Atlas

Netflix Atlas, the last one from the old comers we have talked about before, also has support for alerting built-in into the platform.

18.4.4 Hawkular

Starting from the Hawkular, one of the Red Hat community projects, we are switching off the gears to the dedicated all-in-one open-source monitoring solutions.

Hawkular is a set of Open Source (Apache License v2) projects designed to be a generic solution for common monitoring problems. The Hawkular projects provide REST services that can be used for all kinds of monitoring needs.

https://www.hawkular.org/overview/

The list of the Hawkular components includes support for alerting, metrics collection and distributed tracing (based on Jaeger).

18.4.5 Stagemonitor

Stagemonitor is an example of the monitoring solution dedicated specifically to Java-based server applications.

Stagemonitor is a Java monitoring agent that tightly integrates with time series databases like Elasticsearch, Graphite and InfluxDB to analyze graphed metrics and Kibana to analyze requests and call stacks. It includes preconfigured Grafana and Kibana dashboards that can be customized.

https://github.com/stagemonitor/stagemonitor

Similarly to Hawkular, it comes with distributed tracing, metrics and alerting support out of the box. Plus, since it targets only Java applications, a lot of the Java-specific insights are being backed into the platform as well.

18.4.6 Grafana

It may sound least expected but Grafana is not only an awesome visualization tool but starting from version `4.0` it comes with own alert engine and alert rules. Alerting in Grafana is available on per-dashboard panel level (only graphs at this moment) and upon save, alerting rules are going to be extracted into separate storage and be scheduled for evaluation. To be honest, there are certain restrictions which make Grafana's alerting of limited use.

18.4.7 Adaptive Alerting

So far we have talked about more or less traditional approaches to alerting, based on metrics, rules, criteria or/and expressions. However, more advanced techniques like anomaly detection are slowly making its way into monitoring systems. One of the pioneers in this space is Adaptive Alerting by Expedia.

The main goal for Adaptive Alerting is to help drive down the Mean Time To Detect (MTTD). It does this by listening to streaming metric data, identifying candidate anomalies, validating them to avoid false positives and finally passing them along to downstream enrichment and response systems. - https://github.com/ExpediaDotCom/adaptive-alerting/wiki/Architectural-Overview

The Adaptive Alerting is behind the anomaly detection subsystem in the Haystack, a resilient, scalable tracing and analysis system we have talked about in the previous part of the tutorial.

18.5 Orchestration

The container orchestrators ruled by the service meshes is probably the most widespread microservices deployment model nowadays. In fact, the service mesh plays the role of the "shadow cardinal" who is in charge and knows everything. By pulling all this knowledge from the service mesh, the complete picture of your microservice architecture is going to emerge. One of the first projects that decided to pursue this simple but powerful idea was Kiali.

Kiali is an observability console for Istio with service mesh configuration capabilities. It helps you to understand the structure of your service mesh by inferring the topology, and also provides the health of your mesh. Kiali provides detailed metrics, and a basic Grafana integration is available for advanced queries. Distributed tracing is provided by integrating Jaeger .

Kiali consolidates most of the observability pillars in one place, combining it with the real-time topology view of your microservices fleet. If you are not using Istio, than Kiali may not help you much, but other service meshes are catching up, for example Linkerd comes with telemetry and monitoring features as well.

So what about alerting? It seems like the alerting capabilities are left out at the moment, and you may need to hook into Prometheus or / and Grafana yourself in order to configure the alert rules.

18.6 Cloud

The cloud story for alerting is a logical continuation of the discussion we have started while talking about metrics. The same offerings which take care of the collecting the operational data are the ones to manage alerts.

In case of AWS, the Amazon CloudWatch enables setting the alarms (the AWS notion of alerts) and automated actions based on either predefined thresholds or on machine learning algorithms (like anomaly detection for example).

The Azure Monitor, which backs metrics and logs collection in Microsoft Azure, allows to configure different kind of alerts based on logs, metrics or activities.

In the same vein, Google Cloud bundles alerting into Stackdriver Monitoring, which provides the way to define the alerting policy: the circumstances to be alerted on and how to be notified.

18.7 Serverless

The alerts are as equally important in the world of serverless as everywhere else. But as we already realized, the alerts related to hosts for example are certainly not on your horizon. So what is happening in the serverless universe with regards to alerting?

It is actually not an easy question to answer. Obviously, if you are using the serverless offering from the cloud providers, you should be pretty much covered (or limited?) by their tooling. On the other end of the spectrum we have standalone frameworks making own choices.

For example, OpenFaas uses Prometheus and AlertManager so you are pretty much free to define whatever alerts you may need. Similarly, Apache OpenWhisk exposes a number of metrics which could be published to Prometheus and further decorated by alert rules. The Serverless Framework comes with a set of preconfigured alerts but there are restrictions associated with their free tier.

18.8 Alerts Are Not Only About Metrics

In most cases, metrics are the only input fed into alert rules. By and large, it makes sense, but there are other signals you may want to exploit. Let us consider logs for example. What if you want to get an alert if some specific kind of exception appears in the logs?

Unfortunately, nor Prometheus nor Grafana, Netfix Atlas, Chronograf or Stagemonitor would help you here. On a positive note, we have Hawkular which is able to examine logs stored in Elasticsearch and trigger the alerts using pattern matching. Also, Grafana Loki is making a good progress towards supporting alerts based of logs. As the last resort, you may need to roll your own solution.

18.9 Microservices: Monitoring and Alerting - Conclusions

In this last part of the tutorial we have talked about alerting, the culmination of the observability discussions. As we have seen, it is very easy to create alerts, but it is very difficult to come up with the good and actionable ones. If you get paged at night, there should be a real reason for that. You should not spend hours trying to understand what this alert means, why it was triggered and what to do about it.

18.10 At the End

Admittedly, it was a long journey! Along the way we went over so many different topics that you may feel scared of microservice architecture. Fear no more, there are tremendous benefits it brings on the table however it also requires you to think about the systems you are building in a different way. Hopefully the end of this tutorial is just a beginning of your journey into the exciting world of the microservice architecture.

But keep your ears open. Yes, microservice architecture is not a silver bullet. Please do not buy it as a sales pitch or fall into the hype trap. It solves the real problems but you should actually run into them before choosing microservices as the solution. Please do not invert this simple formula.

Best of luck and with that, happy microservicing!