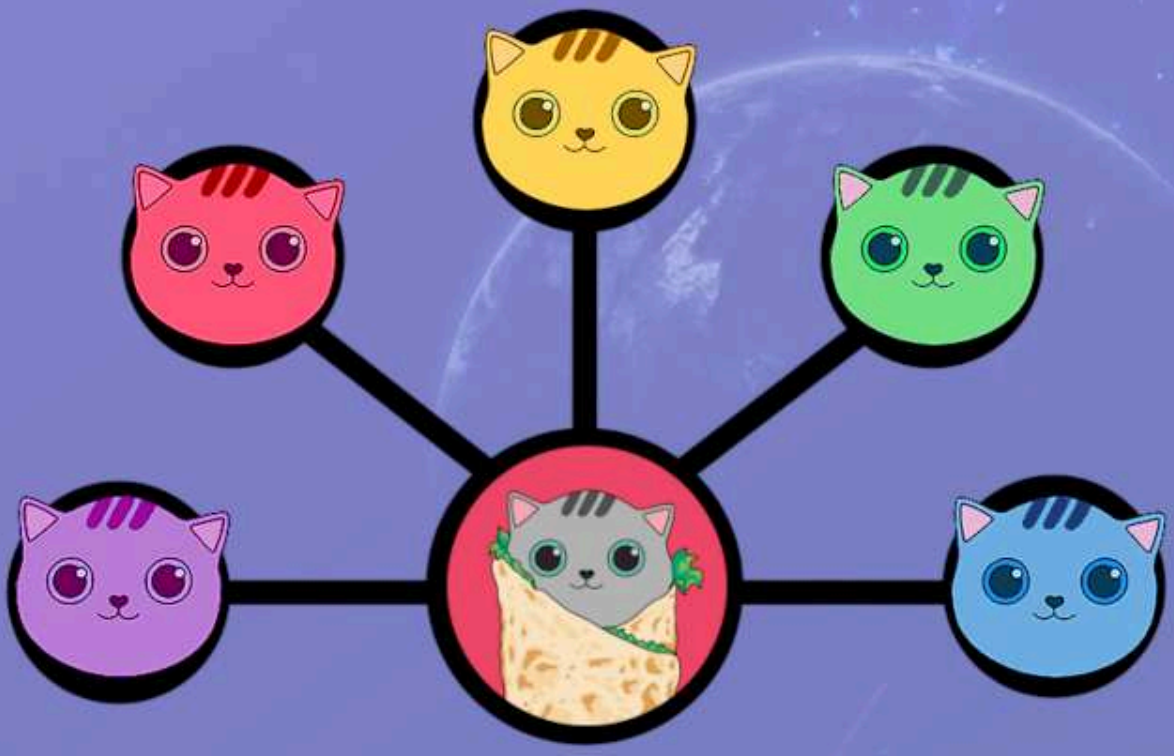
FUNCTIONAL EVENT-DRIVEN ARCHITECTURE

POWERED BY **SCALA 3**



GABRIEL VOLPE

Functional event-driven architecture

Powered by Scala 3

Gabriel Volpe

January 2, 2023

First Edition

Contents

Preface

With Scala 3[1] officially released in early 2021, and exponentially growing its ecosystem and production use in 2022, the future looks brighter than ever for this relatively new version of the language we will be exploring throughout the chapters.

To jazz things up even more, this book will delve into the event-driven architecture (EDA) in a purely functional way, mainly powered by Fs2[2] streams.

In the same spirit of Practical FP in Scala (PFPS)[3], we will develop a distributed system that meets the requirements of a modern software architecture capable of processing billions of events per day at scale using Apache Pulsar[4].

A lot of focus will be put on observability and monitoring as well, which is a necessity in distributed systems. Among other things, we will get into metrics exposed via Prometheus & Grafana and distributed open tracing.

We will also write a Web Sockets service powered by Http4s[5], accompanied by two identical Web applications written in Elm[6] and Scala.js[7], just because we can.

However, before we start designing the system, we will dive into some theory and concepts needed to understand why certain decisions are made. Though, always with the aim of keeping the theory at its minimum — and supplemented with code examples whenever appropriate — to make it a smoother reading experience.

It is worth mentioning that although the application picks on a particular design and implementation, most of the concepts should translate to other systems in the same space that can be built on top of Apache Kafka, RabbitMQ, or other message brokers.

[1] https://docs.scala-lang.org/scala3/getting-started.html
[2] https://fs2.io/
[3] https://leanpub.com/pfp-scala
[4] https://pulsar.apache.org/
[5] https://http4s.org/
[6] https://elm-lang.org/
[7] https://www.scala-js.org/

Acknowledgments

After writing two editions of PFPS, I thought my journey was over, yet I suddenly found myself with plenty of time to get this done. Many factors were in place for me to even consider starting such an ambitious project, but perhaps the most important was the available time on my hands after finishing a job contract.

Writing such a challenging book while having a full-time job would have taken years to complete. So more likely, this would not have happened. Thus, being jobless was somewhat the deciding factor, along with all of your support.

I am forever thankful to the Scala community for granting me the opportunity to make a meaningful contribution that will hopefully help others become more proficient in functional programming using Scala 3—an exquisite programming language.

People

I consider myself incredibly lucky to have had all these extraordinary human beings influencing the content of this book one way or another. This humble piece of work is dedicated:

- To my beloved wife Alicja for her endless support in life.
- To the talented @impurepics[8] for allowing me to use some of his excellent drawings in the book's cover.
- To Pavels Sisojevs[9] for those relentless reviews pointing out details that would escape the ordinary eye.
- To Dave Smith[10] (author of Tyrian) for carefully reviewing the Bonus Chapter.
- To Mrunal Badhe for the valuable early feedback.
- To everyone who reached out with encouraging words and constructive feedback.

Last but not least, I dedicate this book to all the people that make the Typelevel ecosystem as great as it is nowadays, especially to the maintainers and contributors of two of my favorite Scala libraries: Cats Effect and Fs2. This book wouldn't exist without all of your work! **#ScalaThankYou**

Although the book was thoroughly reviewed, I am the sole responsible for all of the opinionated sentences, and any remaining mistakes are only mine.

[8] https://twitter.com/impurepics
[9] https://scala.monster/
[10] https://twitter.com/davidjamessmith

Software

As a grateful open-source software contributor, this section is dedicated to all the free tools that have made this book possible.

- NeoVim[11]: my all-time favorite text editor, used to write this book and code the trading application and web clients.
- Pandoc[12]: a universal document converter written in Haskell, used to generate PDFs and ePub files.
- LaTeX[13]: a high-quality typesetting system to produce technical and scientific documentation, as well as books.
- DrawIO[14]: a configurable diagramming and white-boarding visualization application, used to create most of the diagrams in this book.

[11] https://neovim.io/
[12] https://pandoc.org/
[13] https://www.latex-project.org/
[14] https://github.com/jgraph/drawio

Fonts

This book's main font is Latin Modern Roman[15], distributed under The GUST Font License (GFL)[16]. Other fonts in use are listed below.

- JetBrainsMono[17] for code snippets, available under the SIL Open Font License 1.1[18]
- Linux Libertine[19] for some Unicode characters, licensed under the GNU General Public License version 2.0 (GPLv2)[20] and the SIL Open Font License[21].

[15] https://tug.org/FontCatalogue/latinmodernroman/
[16] https://www.ctan.org/license/gfl
[17] https://www.jetbrains.com/lp/mono/
[18] https://github.com/JetBrains/JetBrainsMono/blob/master/OFL.txt
[19] https://sourceforge.net/projects/linuxlibertine/
[20] https://opensource.org/licenses/gpl-2.0.php
[21] https://scripts.sil.org/cms/scripts/page.php?item_id=OFL

Prerequisites

As a consequence of wanting this book to be as practical as possible, we will only scratch the surface of subjects such as distributed systems, software architecture, streaming, event-driven architecture, clustering, observability, etc.

Besides the theoretical side of it, familiarity with the Scala language and its functional ecosystem is a requirement to understand this book.

Readers should be acquainted with effect monads (e.g. Cats Effect's `IO`) and concepts such as referential transparency, shared state, and tagless final. A minimal understanding of Fs2 streams is also desirable.

It is also worth mentioning that this is not a book about Kafka or Pulsar. It is more about helping users navigate the design space around message brokers by embracing the event-driven architecture and streaming systems in the realm of functional programming.

In the next section, you will find recommendations of reading material that I encourage you to check out regardless of your level.

Reading material

There is a vast availability[22] of learning material for those interested in expanding their knowledge in distributed systems, from which I would recommend the following two books:

- Designing Data-Intensive Applications[23] by Martin Kleppmann[24].
- Designing Event-Driven Systems[25] by Ben Stopford[26].

The former is a must-read for every Software Engineer. It covers the foundations of distributed systems and it delves into batch and stream processing at the end.

The latter is a short book (get it here[27] for free!) that covers design patterns around event-driven and streaming systems built on top of Apache Kafka.

Practical FP in Scala (PFPS)[28]—my previous book—covers all the design patterns and coding best practices you need to understand the code and techniques applied in the trading application, so I highly recommend reading it or ensuring a base knowledge of the topics detailed in its table of contents.

For any further discussions, please refer to the forum[29] (organized by chapter).

[22] https://www.goodreads.com/shelf/show/distributed-systems
[23] https://dataintensive.net/
[24] https://martin.kleppmann.com/
[25] https://www.confluent.io/designing-event-driven-systems/
[26] http://www.benstopford.com
[27] http://www.benstopford.com/2018/04/27/book-designing-event-driven-systems/
[28] https://leanpub.com/pfp-scala
[29] https://github.com/gvolpe/trading/discussions

How to read this book

This book is organized in three different parts:

- **Part I: Concepts** includes the first three chapters (1,2,3), briefly covering theoretical concepts such as event-driven architecture and distributed systems.
- **Part II: Coding** includes chapters 4 and 5, where we dive into Scala 3 and learn about best coding practices. We also learn about effectful streams, data pipelines, and interacting with message brokers from Scala code.
- **Part III: System** includes the last three chapters (6,7,8), where we design and develop a distributed trading system consisting of multiple services, placing each design under scrutiny.

When it comes to code snippets, you will find most of the imports and some datatype definitions are omitted for conciseness. Thus, readers are advised to read it by following along with the Scala project that supplements it.

- trading[30]: Trading application.

The trading project contains the source code of the full-fledged application we will develop throughout the chapters, including a test suite and deployment instructions.

Bear in mind that the presented trading application only acts as a guideline. To gain a better learning experience, readers are encouraged to write their application from scratch; **getting your hands dirty is the best way to learn**.

In case you skipped it, please check out the recommended reading material (see Reading material) for a deeper understanding of critical concepts.

[30]https://github.com/gvolpe/trading

Conventions used in this book

Colored boxes might indicate either notes, tips, or warnings.

Notes

`A note on what's being discussed`

Tips

`A tip about a particular topic`

Warning

`Claim or decision based on the author's opinion`

Part I: Concepts

Writing code is a craftsmanship that requires a solid knowledge of the underlying concepts that make up a system.

In this first part, we will review the definitions of event-driven architecture, microservices, and distributed systems, while analyzing the crucial aspects of each of them.

We will also learn about the key difference between stateless and stateful services, which is critical to choosing an application architecture.

1 Event-driven architecture

Before dialing into the functional programming world, let's set the foundations upon which we can build scalable and reliable distributed systems.

In this first chapter, we will explore software design and architecture. Specifically, we will analyze the pros and cons of event-driven architecture (EDA) and service-oriented architecture (SOA), among others.

Next time we have to design a system, we will be qualified to make an informed decision once we learn about EDA, CQRS, and event sourcing.

1.1 Introduction

Event-driven architecture[1] is a software architecture paradigm promoting the production, detection, consumption, and reaction to events.

An event can be defined as a significant change in the application's state, which is something that simply occurs. In practical terms, an event represents a past action, and other components can not change that fact but only react to it.

For example, an `UserEvent.SignedIn` indicates a user has signed in into the system, allowing consumers of such event to do something about it, such as increasing the current number of users online, and so on.

The responsible for producing such events are called producers (or publishers). The crucial distinction is that producers do not know about consumers and vice versa. Another essential component is the event channel (or message bus), where producers write messages to and consumers read from.

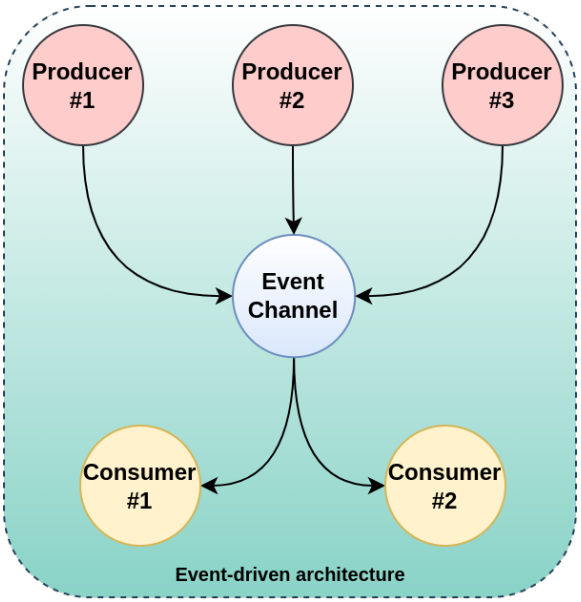


Figure 1.1.1: event-driven architecture

Figure 1.1.1 showcases a typical event-driven application in a simplistic fashion.

An event-driven architecture can be implemented on top of a publish-subscribe message paradigm[2] or an event streaming fashion, which can be a combination of simple event

[1] https://en.wikipedia.org/wiki/Event-driven_architecture
[2] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

processing, event stream processing (ESP)[3], complex event processing (CEP)[4], or online event processing (OLEP)[5]. This book will mainly focus on the former.

1.1.1 What problems does it solve?

To better understand whether event-driven architecture could be the solution to our problems or not, we are going to see how it compares against a traditional monolithic application. In this case, the problem to solve will be a user sign-in feature that roughly requires the following tasks:

1. Receive HTTP request with user credentials.
2. Read user from database to validate credentials.
3. Register user sign-in timestamp into a database.
4. Increase the current number of users online.
5. Notify registered devices about login activity.
6. Return HTTP response.

There are many points where things can go wrong in this minimal example. Suppose the most critical functionality is the ability to sign-in users. In that case, we could potentially agree that points 3, 4, and 5 are out of the equation, as the system will continue to process sign-in requests either way.

Another comparison we can draw (albeit more subtle) is event-driven microservices versus microservices communicating over HTTP. With the latter, what response code should we return to the user when the sign-in succeeds but the notification to registered devices fails? Do we return 200 (Ok) and schedule step 5 to retry later? Or do we return 500 (Internal server error) even though the sign-in action succeeded?

Event-driven microservices take on a different approach. Events are the source of truth, as opposed to synchronous HTTP communication.

We should strive to design services with clear responsibilities and always plan for **scalability** and **fault-tolerance**, which are two of the advantages of microservices.

In order to decouple the required functionality, the sign-in service could become a producer of a `UserEvent.SignedIn`, leading to the design showcased by fig. 1.1.2.

With this fan-out[6] design, other features can now be implemented by *reacting* to such event, which inherently enables parallel processing. Additionally, this allows us to treat different aspects of the system accordingly.

[3] https://en.wikipedia.org/wiki/Event_stream_processing
[4] https://en.wikipedia.org/wiki/Complex_event_processing
[5] https://queue.acm.org/detail.cfm?id=3321612
[6] https://en.wikipedia.org/wiki/Fan-out_(software)

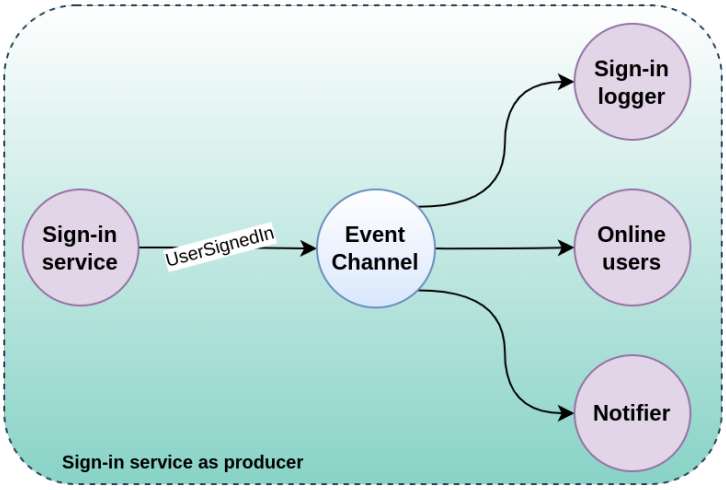


Figure 1.1.2: sign-in service

For instance, if sending notifications to user devices becomes an expensive operation, we could extract it out into an independent service while potentially increasing the available resources only for it.

Continuing with the decoupling example, the split components will be these three:

1. `Sign-in logger` receives `UserEvent.SignedIn` and registers new timestamp.
2. `Online users` receives the same event and increments the number of users online.
3. `Notifier` receives the same event and notifies the registered devices.

It all starts by decoupling producers from consumers, leading to enormous benefits, some of which we will discuss in the following sections. Another advantage of this approach is that all communication can occur asynchronously.

Next, we could try modeling the `UserEvent.SignedIn` in Scala 3 using enumerations.

```
enum UserEvent:
  def eventId: EventId
  def userId: UserId
  def createdAt: Timestamp

  case SignedIn(
      eventId: EventId,
      userId: UserId,
      device: UserDevice,
      createdAt: Timestamp
  )
```

In an event-driven application, it is essential to keep track of event IDs to uniquely identify them and event timestamps for observability purposes. In this case, we also make the `UserId` mandatory, as any `UserEvent` should have one. Furthermore, `UserEvent.SignedIn` contains the device information used to sign in.

We could potentially have other derived events from it with enriched information. Since the "notifier" service needs to read the registered user devices from the database to notify other devices about the login activity, it could also publish an `UserEvent.RegisterDevice` whenever it detects a new device, which might be picked up by a service dedicated to registering devices (write only).

Moreover, we could also add a `CorrelationId`, which will help us associate different system events involved in a particular transaction across services' boundaries.

1.1.2 When to use it? When not?

Event-driven architecture has been a proven design in the industry, as big companies like Microsoft[7] and AWS[8] currently embrace it.

Albeit having their encouragement, as Software Engineers, we must be aware of the pros and cons when designing a system and assess whether a determined architecture could help us deal with the business requirements.

So coming back to the monolithic example, we have started with a simple application that does the following:

1. Receive HTTP request with user credentials.
2. Read user from database to validate credentials.
3. Register user sign-in timestamp into a database.
4. Increase the current number of users online.
5. Notify registered devices about login activity.
6. Return HTTP response.

These appear to be just the functional requirements[9] of the application. However, when designing a system, we also need to consider the non-functional requirements (NFR)[10] to choose an appropriate architecture.

E.g. if we are designing a system for a client, we would probably get a service-level agreement (SLA)[11], which could specify the minimum required throughput (e.g. requests per second), the maximum downtime allowed (availability), etc.

[7] https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven
[8] https://aws.amazon.com/event-driven-architecture/
[9] https://en.wikipedia.org/wiki/Functional_requirement
[10] https://en.wikipedia.org/wiki/Non-functional_requirement
[11] https://en.wikipedia.org/wiki/Service-level_agreement

When we talk about availability, reliability, scalability, and so on, we are talking about a system's quality attributes[12].

Thus, if our monolithic application does the job and aligns with the SLA, introducing an event-driven architecture will more likely be over-killer. The time and resources needed to build such a system could be better assigned to delivering business features.

We may consider EDA when the SLA is demanding. For instance, achieving high availability—such as 99.9% uptime—may require our system to be scalable and tolerant to failures. Decoupling functionality increases our chances of being SLA-compliant.

Furthermore, auditability and observability could also be good reasons to adopt EDA. Events can be observed and audited in many different ways, e.g. by adopting the event sourcing pattern (see CQRS/ES).

[12] https://en.wikipedia.org/wiki/List_of_system_quality_attributes

1.2 Microservices architecture

A microservice (aka service) is a unit of functionality that can be deployed and scaled independently.

In microservices architecture—a variant of service-oriented architecture (SOA)[13]—services communicate via HTTP, Web Sockets, or a messaging protocol such as AMQP[14].

Coming back to our example, this does not necessarily mean that our `Sign-in logger` functionality needs to be implemented as a service. It could also be part of the sign-in service, except it should still listen to `UserEvent.SignedIn` events, effectively decoupling it from the critical path. This pattern is also known as "listen-to-yourself".

In practice, this means that a single service can run multiple producers and consumers, which is a subject we will discuss throughout the book.

Going over the steps again, the sign-in service will:

1. Receive HTTP request with user credentials.
2. Read user from database to validate credentials.
3. Emit `UserEvent.SignedIn` (or no event if credentials are invalid).
4. Return HTTP response.

While it will concurrently:

1. Read `UserEvent.SignedIn` events.
2. Persist user sign-in timestamp into a database.

The decoupling of functionality allows us to extract it into a new service if we find that necessary at some point. For example, we might need to scale the HTTP service that processes sign-in requests to keep up with demand, but no other functionality.

Notes

`Step 3 could be different in other cases. E.g. we could emit an UserEvent.SignInFailed including the failure details for analytics.`

Moreover, by publishing `UserEvent.SignedIn` events into a channel, we make it easy to plug-in new features based on such event—making the system *pluggable*.

The other two features—online users and notifications—can be similarly implemented.

Online users

1. Read `UserEvent.SignedIn` events.

[13] https://en.wikipedia.org/wiki/Service-oriented_architecture
[14] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

2. Increment the number of online users.

Notifications

1. Read `UserEvent.SignedIn` events.
2. Notify user's devices about new sign-in.

Once again, these features could be implemented in a single service or as separate services, depending on the circumstances.

So besides the decoupling of functionality, event-driven architecture and microservices architecture give us **scalability**, **fault-tolerance**, **observability**, and **versatility**.

When there is a bottleneck in the monolithic approach, such as slow database access or too many HTTP requests (or both), it becomes exceptionally challenging to keep up with demand, even when granting massive resources.

Conversely, when the functionality is separated into specific services, fixing this issue becomes as easy as scaling the right one, making it cost-efficient in the long run.

1.2.1 Scalability

Independent services can be scaled out (aka *horizontal scalability*) on demand.

A good example is the launch of a tickets platform for a trendy event such as the FIFA World Cup, where requests to the system reach a peak and can take the whole system down (fun fact: this usually happens with them, so we can guess they do not use EDA). There usually is a rush hour in specific systems where thousands of users (or even millions) access the system concurrently.

If we split responsibilities across unique small services, we can scale a specific functionality independently, which can also be cost-effective when running in the cloud.

We will discuss scalability in earnest in chapters 6 and 7, when we analyze every service.

1.2.2 Fault tolerance

Microservices also enable fault-tolerance, as our system might be capable of continuing to serve requests in the presence of failures.

Regarding our users' sign-in example, if the service responsible for increasing the number of users online has a temporary network issue, the system will still operate normally. The UI might show the wrong number of users for a moment, but it will eventually be right once the service is up again.

This property is known as *eventual consistency*, a topic we will be discussing in the next chapter (see Consistency: eventual vs strong).

1.2.3 Observability

An event-driven architecture can also be highly observable. If all the events in our system flow throughout a central event channel, we can inspect each of them.

This enables **auditability** of the system, which is usually a requirement in any banking or financial service.

We will come back to this topic in Chapter 8, which is dedicated to observability.

Sometimes we may need to recreate the application's state from events, also for this purpose. This is all possible in an EDA system without affecting other services.

1.2.4 Versatility

Having most functionality decoupled allows us to choose between the "listen-to-yourself" and the microservice patterns, making our system versatile.

Both patterns will be used in the system we will start developing in Chapter 6.

1.3 CQRS/ES

> Event sourcing is an append-only log of facts that happened in the past.
>
> *Greg Young*

CQRS stands for Command Query Responsibility Segregation, a specific implementation of the command query separation[15] principle, a term coined by Greg Young[16] circa 2006. It promotes the idea of separating an application between the "writing" and "reading" parts, leading to significant benefits.

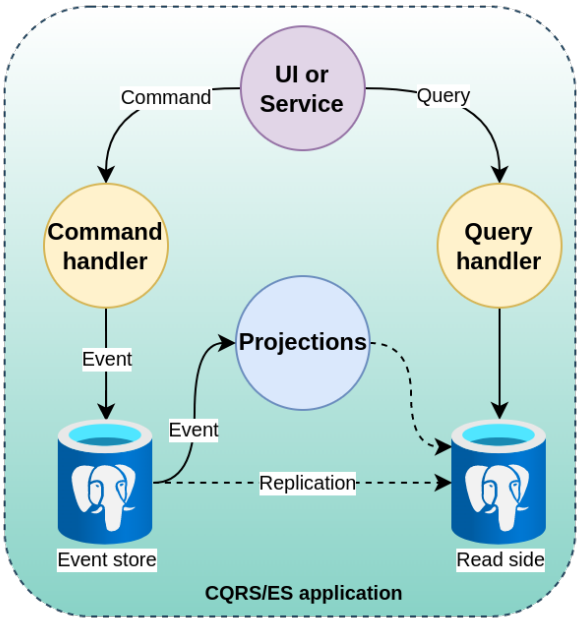


Figure 1.3.1: CQRS/ES application

We have previously learned that events reflect something that already happened in the system. Therefore, if we can persist every event in a store in the same order they occurred, we should be able to reconstruct the current state of the application by replaying the events. This is, in a nutshell, what event sourcing (ES)[17] is.

Figure 1.3.1 shows an example of a CQRS/ES application with queries and commands being produced either by the UI or some service.

[15] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation
[16] https://gregfyoung.wordpress.com/about/
[17] https://en.wikipedia.org/wiki/Domain-driven_design#Event_sourcing

The command handler takes care of the write side, whereas the query handler is responsible for the read side. We can also see how the former produces events, which are persisted in the event store, and also projections that are created from the current state of the application.

Even though the event sourcing term was introduced in the 2000s, it is not new. Accountants have been doing event sourcing the whole time! That's how they keep the books in order—by adding completed transactions to an append-only journal (if you're an accountant, please forgive the over-simplification).

Suppose we consider the typical application that needs to access the database to display a user profile and other user information. In that case, the number of reads vastly outnumbers the number of writes. Thus, separating an application between the reading and writing sides could be desirable.

1.3.1 Commands

Continuing with the user's sign-in feature, we can model the following commands.

```
enum UserCommand:
  def id: CommandId
  def cid: CorrelationId
  def createdAt: Timestamp

  case SignIn(
      id: CommandId,
      cid: CorrelationId,
      username: UserName,
      password: UserPassword,
      device: UserDevice,
      createdAt: Timestamp
  )

  case Register(
      id: CommandId,
      cid: CorrelationId,
      username: UserName,
      password: UserPassword,
      email: Email,
      device: UserDevice,
      createdAt: Timestamp
  )
```

The `UserCommand.SignIn` would eventually lead to the creation of `UserEvent.SignedIn`, if everything went right. Otherwise, other events such as `InvalidSignInAttempted` (e.g. when the wrong credentials are entered multiple times) might be emitted. Note that there could be zero, one, or more events emitted per command.

To expand on the example, we also have `UserCommand.Register`, representing a user sign-up command request. We can imagine events such as `UserEvent.Registered` or `UserEvent.NotRegistered` (e.g. `username` is taken) could be produced from such command.

Notice how commands have a unique ID and a timestamp. This gives us observability in the system, as we can analyze how messages flow in a determined time window.

We also have a `CorrelationId` portraying a unique transaction across service boundaries. For example, a `UserCommand.Register` followed by an `UserEvent.Registered` will share the same correlation ID. This way, we can associate multiple operations across services, especially useful when things go wrong in a distributed system.

It does not have to end here, though. We could have other services reacting to `UserEvent`s and producing other messages such as other events, notifications, or alerts, which inherently share the same `CorrelationId`.

Most CQRS applications have a dedicated *command handler,* which is responsible for consuming commands, processing them, and eventually producing events.

Services can be grouped based on the commands and events they handle. For instance, we could have a users service processing `UserCommand`s and producing `UserEvent`s.

1.3.2 Queries

User queries could come from a potential UI—where users can see their profile and other information—or from another service.

```
enum UserQuery:
  def id: QueryId
  def createdAt: Timestamp

  case GetProfile(
      id: QueryId,
      userId: UserId,
      userToken: UserToken,
      createdAt: Timestamp
  )
```

It will be processed by the query handler, which sits on the read side of the application. Since queries are read-only—i.e. they do not change the application's state—no events are produced.

As with commands, queries also have a unique ID and a timestamp. It might be practical to have a `CorrelationId` in certain cases, but since events are not produced from queries, it is more likely unnecessary.

The system we will develop will have the HTTP layer acting directly as the query handler instead of having messages of type "query" flowing throughout the system.

1.3.3 Reads & writes

The separation of concerns between reads and writes also leads to better database access control in an application. E.g. a service `A` could use a read-only component, and a service `B` could be the only one responsible for writing to the store.

I also endorse this separation of concerns at the components level, e.g. when creating an interface that gives us access to Redis, we can make both a reader and a writer.

```
trait UserCacheReader[F[_]]:
  def find(userId: UserId): F[Option[User]]

trait UserCacheWriter[F[_]]:
  def save(user: User): F[Unit]
```

This makes sense whenever separate services or applications use the reader and writer. If that's not the case, then it makes no difference.

1.3.4 When to use it? When not?

Although CQRS and event sourcing are frequently mentioned together, it does not mean we should pick both when designing an application. Depending on the problem we are solving, we could choose one or the other, or even build on a custom hybrid design that takes a bit of both.

Arguably the most substantial benefit of CQRS is the separation between reads and writes, which allows for optimizations on either side. On the other hand, ES gives us auditability and recoverability, as we can always recreate the current state of the application by replaying events from the event store.

So in a way, CQRS and ES complement each other, but they are two different things. The folks at Microsoft[18] write extensively about this pattern and how it integrates with their Azure platform.

[18] https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

CQRS/ES is an established pattern that solves a lot of problems. It generally works wonders in asynchronous applications where we don't need to wait for the response to a command that has been issued. However, a whole set of problems would be hard to solve, or at least inconvenient, using this pattern (e.g. any synchronous application, such as the classic HTTP request-response model).

Coming back to the user sign-in feature, we might conclude that it is not a suitable use case of CQRS/ES. As mentioned previously, we're better off processing sign-in requests directly in a users' service, where adding a command to process requests, and wait for a response (a natural synchronous model), could be over-killer.

However, we take a lot of value from publishing a `UserEvent.SignedIn`, making the ES part quite helpful in this regard.

To recap, here are the steps that may suit this problem better:

1. Receive HTTP request with user credentials.
2. Read user from database to validate credentials.
3. Emit `UserEvent.SignedIn`.
4. Return HTTP response.

Another example where CQRS/ES might not be a great fit is in dealing with ATM transactions. A request to withdraw money from an account is inherently synchronous, as we need to verify whether the customer has sufficient funds or not. However, emitting events of things that happened, such as `MoneyWithdrawn` once the transaction is over, could be useful downstream to process analytics, statistics, history, and so on.

When picking up this pattern for your application, there are different trade-offs, so it always boils down to the problem we are trying to solve. We will discuss this topic a bit more in the context of a specific system when we reach Chapter 6.

1.3.5 Frameworks

> All you need is a function, pattern matching, and a left fold.
>
> *Greg Young*

Greg's take on event-sourcing frameworks at his talk "A decade of DDD, CQRS, Event Sourcing"[19], meaning we don't really need one to build event-sourced applications.

Although I wholeheartedly agree with him on this, there are frameworks out there that provide much more than just event sourcing, which could be attractive to those looking for a batteries-included solution.

[19] https://youtu.be/LDW0QWie21s?t=1057

Arguably, the most mainstream framework in the Scala ecosystem is Akka Persistence[20], which builds around the Akka ecosystem of actors. It ships with fault-tolerance, distribution, clustering, and scalability out of the box, leading to indisputable popularity.

There is also Aecor[21], which provides a purely functional abstraction built on top of Akka for distribution and fault-tolerance.

In Chapter 3, we will learn more about stateful vs. stateless applications, clustering, and other solutions to event sourcing that don't involve frameworks and that are side-effects free and purely functional.

[20] https://doc.akka.io/docs/akka/current/typed/index-persistence.html
[21] https://github.com/notxcain/aecor

1.4 Summary

Microservices architecture facilitates maintainability in large, complex systems, as every service can have autonomous lifecycles and can be scaled on demand. Services are also fault-tolerant.

The event-driven architecture enables observability and separation of concerns, as most communication throughout the system is based on events.

CQRS introduces commands as instructions that may lead to events, whereas ES allows us to replay such events for auditability and recoverability.

These four critical concepts (Microservices, EDA, CQRS, and ES) make an excellent team for designing distributed systems at scale. However, there are also some downsides to choosing them, some of which have been mentioned in this chapter.

For instance, CQRS doesn't suit synchronous applications. Microservices allow to split business logic into independently deployable units, but it increases maintenance difficulty and roll-out coordination. Furthermore, observability (a topic we will discuss in earnest in Chapter 8) in an event-driven architecture is hard to get right.

Nevertheless, I believe there are more pros than cons to choosing this architecture. In later chapters, we will discuss these trade-offs from a particular perspective: designing a highly available distributed trading system.

2 Distributed systems

The following definition has been adapted from Wikipedia[1]:

> A distributed system is a system whose components are either located on different networked computers or are autonomous processes that run on the same physical computer. They communicate and coordinate their actions by passing messages to one another from any system.

In the previous chapter, we have discussed how microservices enable fault tolerance (see Fault tolerance), as a failure in one of the services does not affect the system as a whole. This is, in fact, one of the most important properties of a distributed system; there is no single point of failure (SPOF)[2].

Event-driven architectures rely on events for communication. Even though an EDA application may begin as a single service, once it scales to more services, it becomes a distributed system: a complex beast to tame.

Distributed systems have plenty of benefits but are also full of caveats. In this chapter, we will cover some of them and talk about other notable characteristics of distributed systems and system design patterns.

There are many written articles and books about this extensive topic (see Reading material). Thus, consider this chapter a merely introduction to some challenges we will face when designing and developing an event-driven system.

[1] https://en.wikipedia.org/wiki/Distributed_computing
[2] https://en.wikipedia.org/wiki/Single_point_of_failure

2.1 Overview

A distributed system comprising multiple services presents enormous benefits, such as scalability, high availability, and fault tolerance. However, it also introduces a fair amount of complexity.

For this reason, we need to carefully consider as many edge cases as possible during the design phase to avoid common pitfalls.

Scalability comes along easily when every service is stateless (more on this in Chapter 3) and takes on specific responsibilities, e.g. users sign-in.

Other properties, such as **fault tolerance** and **availability**, need a specific understanding of the system, as we will learn in the following two sections.

2.1.1 Identifying points of failure

To enable fault tolerance, we need to understand the importance of each service, as the entire system can only continue to operate when all the critical services are up.

So this is our first task when designing a distributed system: to identify the essential services that guarantee the correct functionality of the system. E.g.

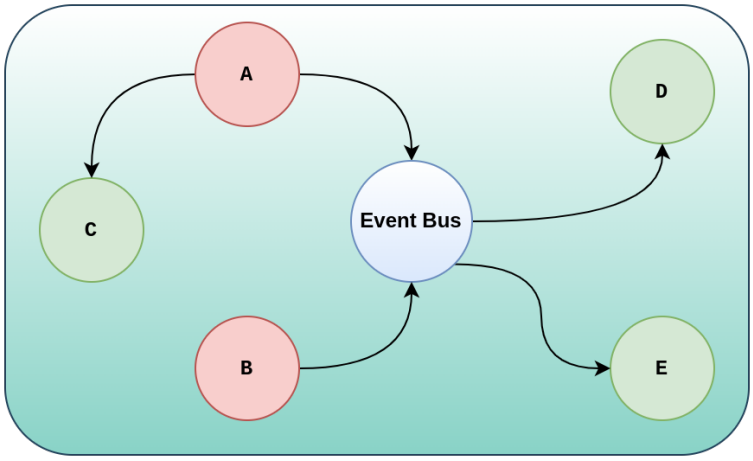


Figure 2.1.1: critical services (A & B)

Figure 2.1.1 shows a system consisting of five different services and a single event bus. By identifying the essential services, we can determine the critical points of failure in our system, as well as the less important ones.

In our example, both services A and B must always be up and running for the system to be in a functional state. We assume we can cope with the other services (colored in

green) being temporarily down (and perhaps the event bus too, though we should assess the situation in every scenario).

This means that services A and B need special treatment. If both services are stateless, we can easily make them fault-tolerant by spinning up at least one more instance of each. The number will always depend on the traffic load and the capabilities of each service.

Conversely, if the services are stateful (see Stateful services for a proper definition), we need to re-assess whether this needs to be the case.

To scale a service that writes to a database, we need to coordinate operations via transactions or any other mechanism that guarantees idempotency and consistency.

Scaling a service that needs to "remember" about previous states (e.g. in past sessions) is a bit more tricky. Ideally, we should avoid application state (e.g. move it over to a cache such as Redis when speed is crucial, or to a transactional database when consistency matters most). However, state must live somewhere, and getting rid of application state is not always possible. In such cases, we could try to reduce the responsibilities of the service and increase its runtime resources (aka *vertical scalability* or *scaling up*).

We will discuss stateless and stateful services at length in Chapter 3, so for now, all we need to know is that these two types of services need to be treated differently.

Returning to the user sign-in scenario given in Chapter 1 (see What problems does it solve?), we can also pinpoint the critical components, as fig. 2.1.2 shows.

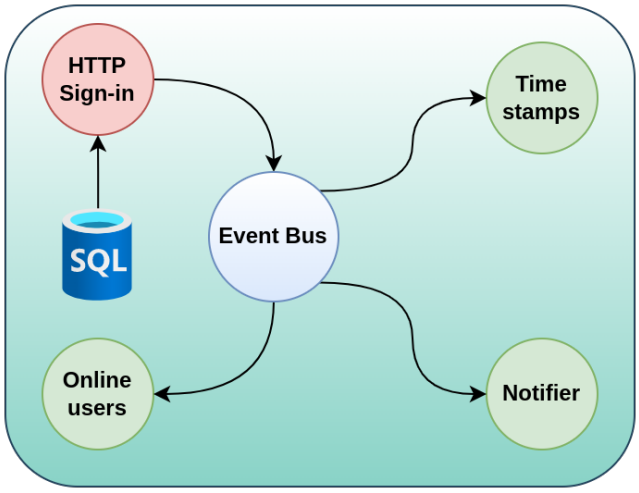


Figure 2.1.2: Critical services in users sign-in scenario

The HTTP users' sign-in service and the users' SQL database are critical points of failure, as our system would not be able to operate without these two components. The other services can always catch up with the influx of events.

2.1.1.1 Consensus protocols

It is worth mentioning that fault tolerance in stateful services can be solved via consensus protocols. Especially when working with state machines, these could be converted into a distributed implementation via consensus algorithms such as Paxos[3] and Raft[4], which are two of the most notorious.

A lot of distributed databases are built on top of these algorithms. For example, the TiKV[5] key-value database is powered by Raft.

Other popular applications such as Google Spanner[6] (distributed SQL database), Ceph[7] (distributed storage system), and Neo4j[8] (graph database), are built on top of Paxos.

Although this is low-level machinery that should not be a concern to application engineers relying on stateful message brokers such as Kafka and Pulsar, it is nevertheless necessary to build a solid understanding of the underlying functionality of our architecture.

[3] https://en.wikipedia.org/wiki/Paxos_(computer_science)
[4] https://raft.github.io/
[5] https://tikv.org/
[6] https://en.wikipedia.org/wiki/Spanner_(database)
[7] https://ceph.io/en/
[8] https://neo4j.com/

2.1.2 Consistency vs availability

These are two of the concepts that make the CAP theorem[9] (fig. 2.1.3 credits[10]).

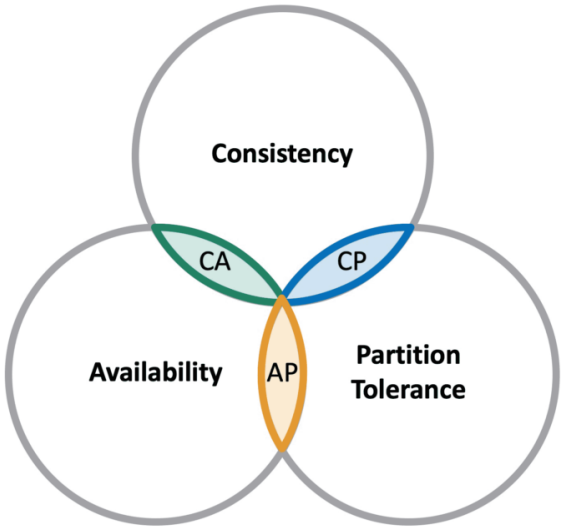


Figure 2.1.3: CAP theorem

A distributed system cannot be immune to network failures. Therefore, we must always choose between consistency (CP) and availability (AP).

Tips

`Understanding the relationship between the CAP theorem and NoSQL databases`[a] `can also be very interesting.`

[a] https://www.ibm.com/cloud/learn/cap-theorem#toc-cap-theore-ovhB7WL3

Even when choosing AP, we still need to select a consistency model[11], which dictates how data is viewed and updated across nodes.

Yet, not everything is about CAP. Generally, a system would be operating under normal conditions. In such cases, we still need to choose between latency and consistency, as the PACELC theorem[12] states—an extension of the CAP theorem.

[9] https://en.wikipedia.org/wiki/CAP_theorem
[10] https://hazelcast.com/glossary/cap-theorem/
[11] https://en.wikipedia.org/wiki/Consistency_model
[12] https://en.wikipedia.org/wiki/PACELC_theorem

2.1.2.1 Consistency: eventual vs strong

Eventual consistency[13]—also called optimistic replication—is a consistency model used in distributed computing to achieve **high availability** that informally guarantees that, if no new updates are made to a given data item, ultimately all accesses to that item will return the last updated value.

Eventually-consistent services are often classified as providing BASE semantics (basically-available, soft-state, eventual consistency), in contrast to traditional ACID[14] (atomicity, consistency, isolation, durability) semantics. However, we could argue such definitions are somewhat vague (see Martin Kleppmann's take on the topic in Designing Data-Intensive Applications).

A social media application with a "like" feature is a good use case for eventual consistency, favoring scalability and availability. A user may not see the number of likes immediately increase after another user gives it a like. Yet, the data should eventually become consistent (e.g. after a page refresh a few seconds later).

We have a few options when strong consistency[15] is more important than high availability. Nowadays, databases such as Google Spanner and Fauna[16] promise strong consistency and high availability over distributed systems, claiming network partitions are rare. Although this is true, whenever a network partition occurs, both implementations make some compromises over availability.

Another great model that can be directly implemented in the application layer is **strong eventual consistency (SEC)**, which can be achieved via conflict-free replicated data types (CRDT)[17], giving us the missing safety property[18] of eventual consistency.

Event-driven applications usually favor eventual consistency, for the most part. However, we could also opt-in for strong consistency in particular system areas. Thus, it is fair to say we can combine both consistency models depending on our use case.

Ultimately, something to remember is that eventual consistency is almost always a reading problem. Yet, it can still be a writing problem in multi-master clusters where multiple instances can be handling the writes, as is the case with geo-replicated databases.

[13] https://en.wikipedia.org/wiki/Eventual_consistency
[14] https://en.wikipedia.org/wiki/ACID
[15] https://en.wikipedia.org/wiki/Strong_consistency
[16] https://fauna.com/
[17] https://crdt.tech/
[18] https://en.wikipedia.org/wiki/Safety_property

2.2 Idempotence

In the shopping cart application[19] we developed as part of PFPS, we worked with an allegedly idempotent third-party payment client. Given the same payment request, it guarantees the credit card is charged at most once.

If the payment was already processed, it returns a different HTTP status code (409: conflict), including the payment ID, indicating there is no need to keep on retrying.

```
client.run(POST(payment, uri)).use { resp ⇒
  resp.status match {
    case Status.Ok | Status.Conflict ⇒
      resp.asJsonDecode[PaymentId]
    case st ⇒
      PaymentError(
        Option(st.reason).getOrElse("unknown")
      ).raiseError[F, PaymentId]
  }
}
```

This allows us to code without fear of creating duplicate payments, potentially charging the customer's credit card twice—with all that implicates.

In most cases, the remote HTTP service will be invoked only once per payment, but when something goes wrong (e.g. network failure or local service crashes), we can retry the request knowing the service is idempotent. Without this guarantee, it would be much harder to deal with failures.

Idempotence[20]—aka idempotency—is vital in eventually-consistent services. Considering that a process can be interrupted at any time and then it would start over upon a restart, we need a way to ensure that a repeated operation is only performed once.

It is one of the must-have properties when working with *at-least-once* delivery guarantees (see Delivery guarantees)—i.e. a message could be duplicated.

2.2.1 Deduplication

In our trading application, we will implement a few services where deduplication of messages must be guaranteed to enable idempotency upon restarts and rollouts.

Most modern message brokers, such as Kafka and Pulsar, can deduplicate messages before sending them to the consumers (see Apache Pulsar in Chapter 3), so we can (almost) forget about this and focus on solving business problems.

[19] https://github.com/gvolpe/pfps-shopping-cart
[20] https://en.wikipedia.org/wiki/Idempotence

Another approach to deduplication and atomicity that popular message brokers also support is distributed transactions, usually taking a toll on performance.

However, distributed transactions can only make atomic what happens between the application and the message broker—i.e. publishing and consuming messages. If other technologies are involved (e.g. writing to a database), we would need to handle it ourselves (e.g. using the Saga pattern[21]).

We can leverage two main strategies to deduplicate messages: deduplication at the producer and consumer sides, as we will learn in the next sections.

2.2.1.1 Producer-side deduplication

This is the most common strategy supported by message brokers, where a unique sequence identifier (aka sequence ID) is assigned to every message. If a message is retried for some reason, it can be deduplicated before being sent down to the consumers by keeping track of previously delivered messages' sequence identifiers.

This approach works well with redeliveries and retries (see Delivery guarantees), but it doesn't help us deduplicate a message "a" from another message "a" even though these are more likely a duplicate to the human eye. Each message will be assigned a unique sequence ID, making them distinct.

To solve this, most message brokers allow us to dictate how to assign sequence IDs. However, this means we need to do the broker's job and keep track of previous sequence IDs, making our application stateful. While not ideal in most cases, it may serve our purposes after considering the second strategy.

2.2.1.2 Consumer-side deduplication

With multiple producers publishing to the same topic, it can be intricate to deduplicate at the producer side, as that would require distributed coordination to issue unique sequence IDs—highly increasing complexity.

For these cases, it is easier to deduplicate at the consumer side. However, the only way to achieve this is by making our application stateful and keeping track of consumed messages that can be uniquely identified (e.g. by a message ID).

An immediate question comes to mind: **How long do we keep track of previous messages?** This applies to tracking messages on either side.

A common approach is only to keep track of the last messages processed in the last N minutes—e.g. all messages processed within a time window of 30 minutes or so.

[21] https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

Generally, this technique covers messages that are retried after a consumer's or a producer's crash. If we still have messages arriving after our flexible time window, we may have bigger problems than duplicate messages. Thus, this approach is well accepted in the software architecture space.

As a matter of fact, this is also the most common deduplication approach used by streaming platforms such as Kafka Streams[22].

2.2.1.3 Deduplication engine

Writing a deduplication engine that works both at producing and consuming times is a fun exercise to understand what other systems must do to guarantee no duplicates.

Under the `demo` module of the reference application, there is an implementation that follows the following approach: we keep track of the IDs processed so far over a configurable time window and compare this registry with the current command ID to decide whether it should be processed.

This can either be done in memory or be serialized to survive restarts. Whether to select one method or the other will always depend on several factors.

For instance, when we have a service running either in exclusive or fail-over mode (see Subscription types), we can get away with the in-memory deduplication, relying on the acknowledgement mechanism and downstream responsibilities. Most times, deduplication may not even be necessary.

Yet, deduplication is critical when running in shared or key-shared subscription mode, and serialization is necessary to guarantee the correct functionality of the system.

[22] https://docs.confluent.io/platform/current/streams/index.html

2.3 Atomicity

The A in ACID stands for atomicity[23]. In database systems, an atomic transaction is an indivisible and irreducible series of operations such that either all occur (COMMIT) or nothing occurs (ROLLBACK).

However, atomic transactions (or operations) also extend to concurrent programming, where the concept is also defined as linearizability[24]. A few popular models of linearizability are compare-and-swap (CAS) and lock[25], among others.

Some of the concurrent data structures from Cats Effect use these mechanisms. For example, `cats.effect.kernel.Ref` is implemented on top of Java's AtomicReference[26] and its `compareAndSet` method. Furthermore, `cats.effect.std.Semaphore` is an implementation of a lock (aka mutex).

Only with Cats Effect, we already have a fantastic toolbox to work with in-memory atomic transactions. However, when we need distributed atomic transactions, we must rely on other tools external to our code.

Most relational databases are ACID-compliant, meaning they support atomic transactions, making them an excellent fit for distributed systems requiring strong consistency. Yet, we sometimes need atomicity working with NoSQL databases that do not support transactions.

2.3.1 Distributed transactions

Many SQL databases support distributed transactions[27], which involve two or more different servers that need to coordinate writes for a transaction to succeed, generally implemented via a two-phase commit protocol (2PC)[28].

However, this is not a feature specific to databases. Message brokers such as Kafka and Pulsar also support distributed transactions (see Pulsar transactions).

Transactional support in message brokers allows us to consume, process, and produce messages in one atomic operation—possibly involving multiple topics. Furthermore, distributed transactions enable strong consistency semantics, at the expense of latency (they can be expensive at scale).

In our application, we will employ distributed transactions and learn how easy it is to make strong consistency guarantees, even though it comes at a cost.

[23] https://en.wikipedia.org/wiki/Atomicity_(database_systems)
[24] https://en.wikipedia.org/wiki/Linearizability
[25] https://en.wikipedia.org/wiki/Lock_(computer_science)
[26] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicReference.html
[27] https://en.wikipedia.org/wiki/Distributed_transaction
[28] https://en.wikipedia.org/wiki/Two-phase_commit_protocol

2.3.2 Change data capture

CDC stands for Change Data Capture[29], a topic discussed at length in the Designing Data-Intensive Applications book. Its popularity has dramatically increased in streaming systems over the last few years, for a good reason.

It elegantly solves the "atomically write to multiple stores" issue. To better understand what this means, let's examine the following example.

Suppose we have a service responsible for registering authors. It needs to write the author record into a PostgreSQL table, persist the author name in Redis, and publish an `AuthorRegistered` event to Pulsar once this is done.

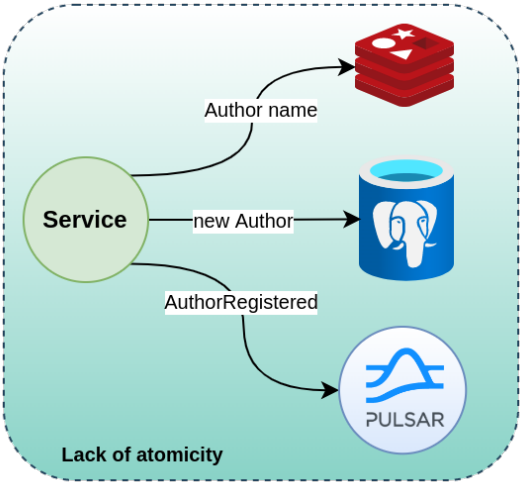


Figure 2.3.1: lack-of-atomicity

It is virtually impossible to guarantee atomicity for all these operations. Without such a guarantee, we can either make all operations idempotent or implement a roll-back mechanism for every procedure, e.g. using the saga pattern.

While these solutions work, it is not always possible to guarantee idempotency or provide a roll-back mechanism. Moreover, making multiple operations atomic becomes more challenging when complexity increases.

CDC enables linearizability by directly reading the transactional log of the database, as fig. 2.3.3 shows. In the case of PostgreSQL, it does so by leveraging its logical decoding feature[30].

[29] https://en.wikipedia.org/wiki/Change_data_capture
[30] https://www.postgresql.org/docs/current/logicaldecoding-explanation.html

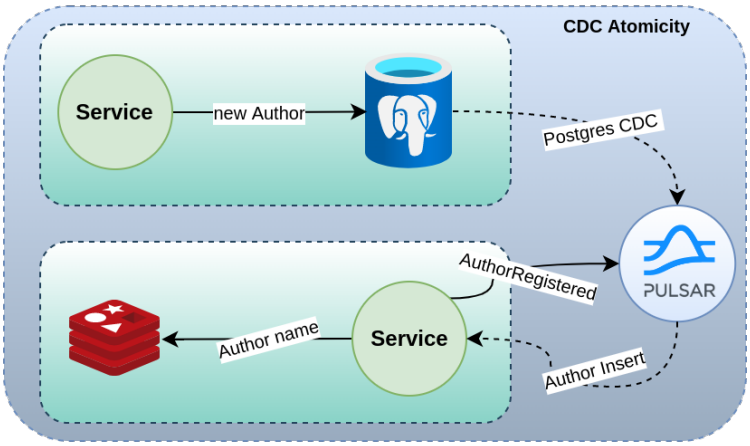


Figure 2.3.2: cdc-atomicity

Furthermore, we could move the write to Redis further by reacting to the `AuthorRegistered` event, making it a linear sequence of distributed operations, as fig. 2.3.3 shows.

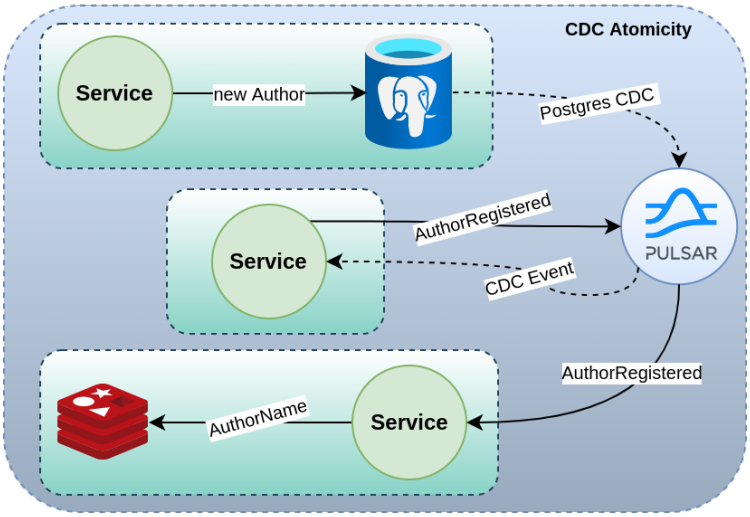


Figure 2.3.3: cdc-atomicity-ext

The service appearing multiple times in fig. 2.3.3 could be one or multiple services. When it is only one, reacting to the same events the service publishes is also known as the "listen to yourself" pattern.

Products such as the Debezium connector for PostgreSQL[31] let us plug it into multiple databases and message brokers, making legacy services' migrations a trivial task.

[31] https://debezium.io/documentation/reference/stable/connectors/postgresql.html

Notice that the Debezium connector requires an extra process running in our system, increasing operational complexity. However, it can be used in similar scenarios to avoid transactions once you have it in place.

2.3.2.1 Outbox pattern

Another popular pattern enabled by CDC is the outbox pattern—aka Transactional outbox[32]. Instead of reading incoming CDC events directly from the table we insert data into, it promotes the introduction of an *outbox* table where events that need to be published once the database transaction succeeds are stored.

Continuing with the author's registration example, fig. 2.3.4 showcases this pattern.

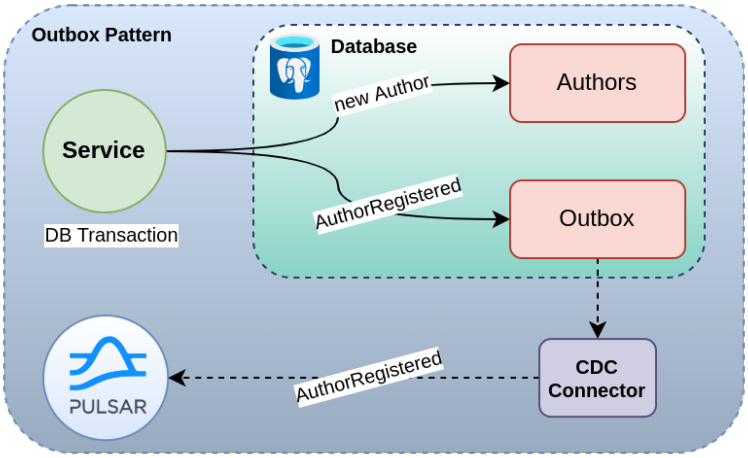


Figure 2.3.4: outbox-pattern

The author entity and the `AuthorRegistered` event are persisted in the same database transaction, so either both operations succeed, or every change is rolled-back. Once the transaction is committed, the CDC connector will capture the outbox insert operation and emit the event to the message broker.

The outbox pattern enables strong consistency by relying on a single database transaction. However, it makes the database a bottleneck, which is one thing we want to avoid in streaming and event-driven systems; a drawback we need to be aware of.

Patterns such as *listen to yourself* could be a better solution in some scenarios. We must evaluate our options when designing a system to avoid abusing the outbox pattern[33].

[32] https://microservices.io/patterns/data/transactional-outbox.html
[33] https://www.squer.at/en/blog/stop-overusing-the-outbox-pattern/

2.3.3 Distributed locks

A distributed lock manager (DLM)[34] is essential to synchronize access to shared resources in distributed systems. Many operating systems, cluster managers, and distributed databases use them.

An efficient and lightweight distributed lock can be implemented on top of Redis[35], which is by design safe (mutual exclusion), dead-lock free, and fault-tolerant.

The idea is simple enough whenever we have a single Redis instance: a client can acquire a lock by creating a key with an expiration time (aka time-to-live or TTL).

```
SET my_lock client_uuid NX PX 30000
```

Essentially, set the `my_lock` key with the `client_uuid` value if it does not exist (`NX`), with an expiration time of 30000 milliseconds (`PX`). If the key already exists, it means another client currently holds the lock and we need to retry (usually after a few milliseconds).

Using the `redis4cats` library, the same operation looks as follows.

```
redis.set("my_lock", "client_uuid", SetArgs(Nx, Px(30000.millis)))
```

The client that holds the lock can release it at any time by deleting the key (usually when it's done using the resource). If the client crashes and fails to do so, the lock will become available once again when the key expires (dead-lock free).

Challenge

Implement a distributed lock using a single Redis instance and at least three different clients trying to acquire it.

A more comprehensive example in Scala may look as follows.

```
type Lock = String

val lockName = "my_lock"
val clientId = "porcupine-ea4190d4-0807-4c90-aea9-41c19e249c84"
```

[34] https://en.wikipedia.org/wiki/Distributed_lock_manager
[35] https://redis.io/topics/distlock

```
val acquireLock: IO[Lock] =
  redis
    .set(lockName, clientId, SetArgs(Nx, Px(30000.millis)))
    .flatMap {
      case true  ⇒ IO.pure(clientId)
      case false ⇒ IO.sleep(50.millis) >> acquireLock
    }

val deleteLock: Lock ⇒ IO[Unit] =
  id ⇒
    redis.get(lockName).flatMap {
      _.traverse_ { v ⇒
        redis.del(lockName).whenA(v == id)
      }
    }

val lock: Resource[IO, Lock] =
  Resource.make(acquireLock)(deleteLock)
```

It is critical to verify that the lock's value corresponds to our application ID; otherwise, we may be deleting a lock acquired by another client.

With this lock modeled as a resource, we can perform any computation that may require access to a distributed shared resource with some guarantees.

```
val program: IO[Unit] =
  lock.surround {
    IO.println("some computation")
  }
```

If you are interested in learning about other popular implementations of distributed locks, look at Google's Chubby[36] and Apache ZooKeeper[37].

[36] https://research.google/pubs/pub27897/
[37] https://zookeeper.apache.org/doc/r3.1.2/recipes.html#sc_recipes_Locks

2.4 Summary

We get a lot of power and scalability by distributing computations across multiple machines, but it all comes at a cost: **distributed systems are hard**.

Still, hopefully you got enough understanding of the topics discussed so far to put to use during the development of the trading application. Regardless, we will revisit most of the concepts from a practical point of view when we reach Chapter 6 and discuss design decisions for every service.

Disclaimer: The idea of this short chapter is to highlight the concepts we will need the most in developing our event-driven application. As brief as it is, you can imagine we are only skimming through the surface of each topic.

To further enhance your understanding, I would encourage you to read specific books about distributed systems (see the recommended Reading material).

3 Stateless vs. Stateful

Let's begin with the definition of *state* by adapting a quote from Wikipedia[1]

> A system is described as stateful if it is designed to remember preceding events or user interactions; the retained information is called the state of the system.

A stateful service generally persists its state in an external database or cache to survive restarts and to synchronize writes with other instances, which highly increases its complexity.

Conversely, a stateless service can usually operate independently of other instances without the need for synchronizing writes and can be restarted without second thoughts.

Naturally, stateless services are the easiest to scale and maintain. However, every application needs state somewhere, whether in someone else's cloud or our system.

Therefore, the difference between stateless and stateful is key to scalability in microservices. This is a crucial knowledge to have in software architecture.

In the trading application, we will deal with both types of services.

[1] https://en.wikipedia.org/wiki/State_(computer_science)

3.1 Stateless services, stateful brokers

The premise behind stateless services in an event-driven architecture is that most of the state lives on a message broker. This way, services perform computations based on incoming messages and publish the resulting state back to the broker, similar to what the actor model[2] enables.

This does not mean we do not need a database or a cache here or there. Nonetheless, it allows us to easily separate the stateless services from the stateful ones.

For instance, two critical services might be performing computations based on messages they receive while publishing the result back to the message broker. Thus, enabling parallel processing and high throughput.

Suppose the final result (or part of it) needs to be persisted for analytics or auditability. A third service could exclusively consume such messages and write them down to a database in the required form.

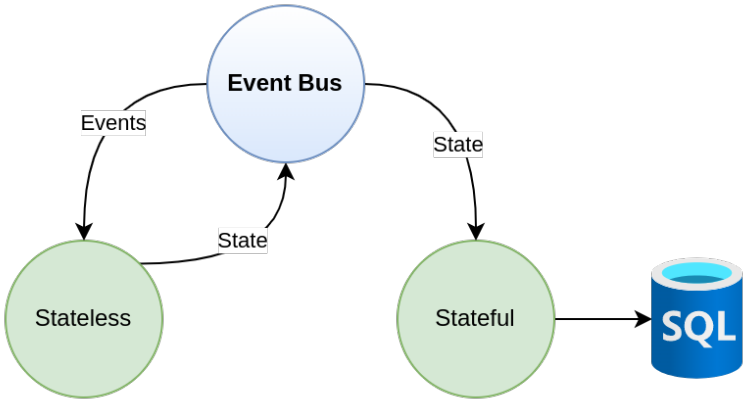


Figure 3.1.1: stateless-vs-stateful

It is a simple yet powerful design pattern that we will be using in the trading system.

3.1.1 Stateful services

Although fig. 3.1.1 labels one of the services as stateful, writing data to a database alone does not make a service stateful; let's clarify this.

As stated at the beginning of the chapter, a stateful service is that one that needs to remember what happened in previous sessions when starting up (e.g. on restart). Usually, this is represented as internal state and is persisted in a data storage for later retrieval.

[2] https://en.wikipedia.org/wiki/Actor_model

So such services depend on reading the previous state (if any) from an external storage to start operating. In most cases, they also require the ability to write their internal state back into storage to resume where they left off in case of restarts.

This type of approach will be discussed further down (see State snapshots).

Although the stateful service definition is clear enough, it is still typical to confuse them with stateless services that hold in-memory state or that interact with a database.

For this reason, the service labeled as stateful in fig. 3.1.1 might as well be labeled as stateless. Here is the key question that would make the difference:

- Does it need an initial state from an external storage to start operating?

If the answer is yes, then it is a stateful service. Otherwise, it is stateless.

To give one more example of a stateless service that holds state, we can use Kafka Streams once again. Such streaming services perform aggregations and are backed by `KTable`s, a specific data store that runs in the same node as the streaming service.

We consider such services to be stateless because Kafka Streams handles the state and replicates it to local `KTable`s as needed, liberating the service from all the hassle.

3.1.1.1 Users' sign-in state

Let's bring back the users' sign-in feature to analyze where the state may live.

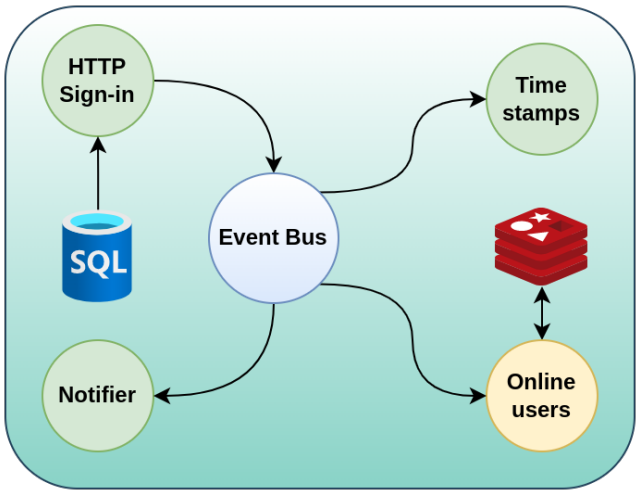


Figure 3.1.2: user-sign-in-state

From this diagram, we could deduce most services are stateless. Let's illustrate why.

- HTTP service only reads user credentials from the database.
- Notifier receives events and notifies other devices.
- Timestamps' service potentially writes to an append-only file.

The online users' service could be either (depending on its responsibilities). In this case, we imagine it is stateful due to requiring to read the current number of online users from a cache when starting up. As we will learn soon, it would only take a compacted topic (see Topic compaction) to remove this limitation and make it a stateless service.

3.1.2 Application clustering

The opposite example of stateless services and stateful brokers are the so-called clustered services. Such applications need to run in a network together with other nodes (or service instances), where the number of instances is known to all the nodes.

In such systems, it is also common to have leader election (master) and worker nodes (slave), which are generally elected via consensus algorithms (see Consensus protocols).

Akka Cluster[3] is unquestionably the most popular Scala framework that allows us to write this kind of applications. It leverages the actor model in multiple machines (making it distributed), which many believe is its killer feature.

Clustering introduces a lot of complexity, though this is inevitable. When we use stateless services, the clustering state lives in the message broker—a monster that also needs taming—but this is usually Dev Ops land.

When we go for things like Akka Cluster, application developers are also responsible for managing the cluster of nodes formed by the different instances, making the boundaries with system infrastructure and application code less clear.

Both approaches have pros and cons. Therefore, it always boils down to what trade-offs we are willing to accept.

[3] https://doc.akka.io/docs/akka/current/typed/cluster-concepts.html

3.2 Message-driven architecture

Both event-driven and actor-based architectures fall within the umbrella of message-driven architecture. It can even be a combination of both.

Yet, there is a crucial difference between messages and events: messages are addressed to a specific destination while events are not. Events occur, and others can observe and react to them, contrary to a message explicitly delivered to a recipient.

Nevertheless, both messages and events need to be delivered somehow, making no difference when it comes to delivery guarantees.

3.2.1 Delivery guarantees

There are three types of delivery guarantees: *at-most-once*, *at-least-once*, and *exactly-once*, each having different trade-offs.

- `at-most-once`: a message is delivered zero or one time; i.e. messages may be lost.
- `at-least-once`: a message is delivered one or more times; i.e. messages are never lost but could be duplicated.
- `exactly-once`: literally, a message is delivered just once; i.e. there can neither be lost nor duplicated messages.

The first method is the simplest of them all, as a message can be sent in a fire-and-forget way—without waiting for acknowledgement. Message brokers commonly use the second one, which requires retries and acknowledgements.

Exactly-once semantics are nearly impossible to achieve. A sender needs acknowledgement from a receiver. However, what happens if the message is successfully delivered and received, but the receiver fails to acknowledge the message? The sender has two options in this case: either re-deliver the message or assume it was delivered. Either way, the exactly-once semantics can not be guaranteed at this point.

Therefore, in most common scenarios, we will be using at-least-once semantics, and whenever we can afford to lose messages, at-most-once semantics.

The latter can be valuable in systems where the subsequent incoming message makes the previous message irrelevant. It is increasingly employed in the IoT[4] realm, where any acknowledgement mechanism becomes expensive.

[4] https://en.wikipedia.org/wiki/Internet_of_things

3.2.2 Apache Kafka

Apache Kafka[5] is arguably the most popular open-source message broker, developed initially at LinkedIn. Companies use it for data pipelines, analytics, and other mission-critical applications.

Implemented as a distributed commit log[6], Kafka aims to be the go-to platform for high throughput real-time data feeds.

It consists of producers and consumers, as well as topics partitioned across different brokers to achieve high throughput.

Before version 2.8, Kafka heavily depended on Apache ZooKeeper[7] as a metadata store for partitions and brokers, but it will be deprecated in Kafka 3.4 and entirely removed in Kafka 4 in favor KRaft[8]: the new consensus protocol for Kafka.

So most companies running Kafka in production are still on this previous setup, as the upgrade is not very trivial. The requirements for ZooKeeper made Kafka more complex than what it is, so this is excellent news for those looking to give Kafka 3 a try.

Kafka is not the focus of the book, so we will not be diving too much into it.

Those interested in learning more about system architectures built on Kafka should definitely read "Designing Event-Driven Systems" (see Reading material).

3.2.3 Apache Pulsar

Apache Pulsar[9] is the new kid on the block when it comes to message brokers, initially created at Yahoo!

It is also applicable to data pipelines, analytics, and low latency real-time data feeds.

Unlike Kafka, topics in Pulsar are not partitioned by default and are served by a single broker. However, this feature is optional, making it much easier to deal with topics.

We will be using Pulsar extensively in our trading application, so we are now going to go through a subset of its features and see how it differs from Kafka.

Ultimately, I will try to give a personal opinion on how the two compare.

[5] https://kafka.apache.org/
[6] https://en.wikipedia.org/wiki/Commit_(data_management)
[7] https://zookeeper.apache.org/
[8] https://developer.confluent.io/learn/kraft/
[9] https://pulsar.apache.org/

3.2.3.1 Subscriptions

A feature I value dearly is the ability to select a subscription type[10] when subscribing to a topic, which allows for different design patterns such as fan-out pub-sub messaging (exclusive mode) and message queueing (shared, fail-over, and key-shared mode).

The official documentation does a great job; thus, you can learn mostly everything right on their site. However, since subscription types are essential to understanding the system we will develop, let's look at the different types.

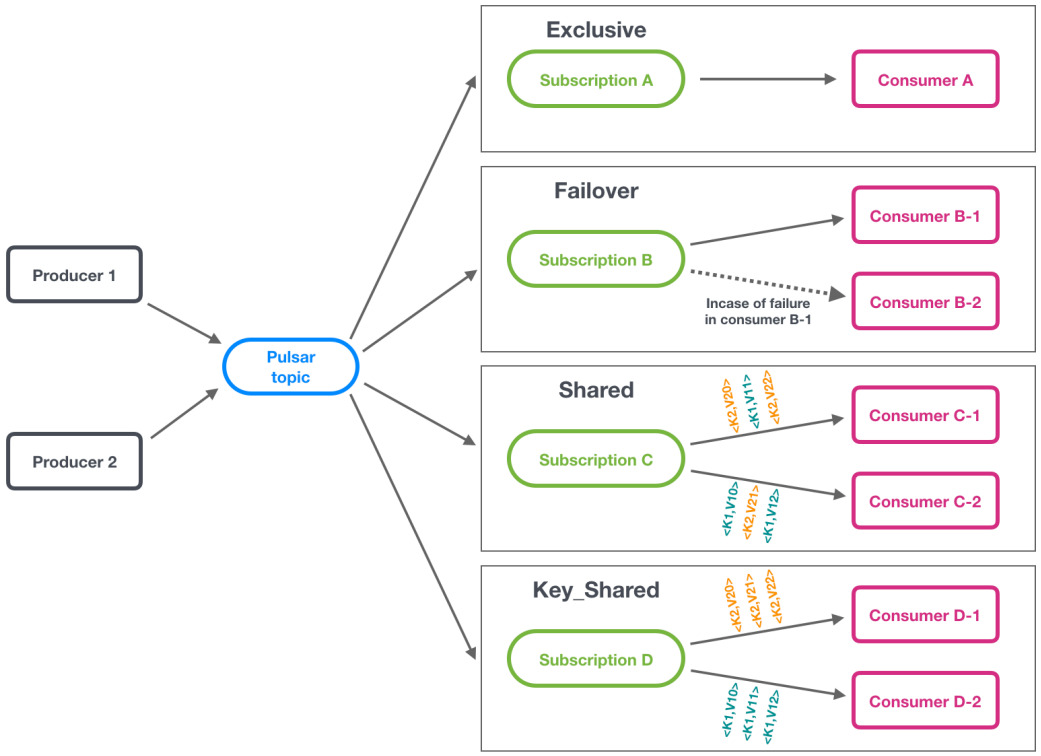


Figure 3.2.1: subscription types

[10] https://pulsar.apache.org/docs/en/concepts-messaging/#subscriptions

Figure 3.2.1 conveys the idea quite well, but let's add a simple definition of each type.

- **Exclusive**: only a single consumer is allowed to attach to a subscription.
- **Fail-over**: multiple consumers are allowed to attach to a subscription. A master consumer is picked and is the only one that gets the messages. The rest are there just in case something goes wrong with the former.
- **Shared**: multiple consumers can attach to a subscription, messages are delivered in a *round-robin* distribution across consumers, and any given message is delivered to only one consumer.
- **Key-Shared**: same as `Shared`, except messages are delivered according to a message or ordering key consumers can define instead of using round-robin.

It is important to grasp subscriptions well to better understand how these affect the decisions we make in our system.

3.2.3.2 Deduplication

Another essential feature is deduplication[11], allowing streaming applications to consume from topics without worrying about duplicates. This is a feature heavily used by Kafka Streams (see Streaming support further down).

In a few words, Pulsar deduplicates messages by assigning a unique sequence ID, which should be set on every message. We will learn how the Neutron library handles deduplication for us when we get to Chapter 5 (see Distributed via Apache Pulsar).

This could be enabled either at the system level or directly on a namespace or topic. These options are disabled by default both on the server and client sides, so make sure to consult the official documentation.

To expand on what we learned in Chapter 2 (see Deduplication), let's analyze some of the challenges that might come up in our system.

Suppose we are publishing messages—each with an ever-incremental sequence ID—and our service crashes at some point. Upon restart, we need to ensure the next message is assigned a greater sequence ID than the last known one.

This is easy when we have a single producer: all we need is to give it a unique name for Pulsar to be able to associate a producer to a last known sequence ID. In Scala terms, it would be something like `Map[ProducerName, LastSequenceId]`.

The real challenge appears when multiple instances of our service are involved, each with one producer publishing messages to the same topic. In such cases, we need to consider a few other variables.

Say we have the following case per instance:

[11] https://pulsar.apache.org/docs/en/concepts-messaging/#message-deduplication

1. Consume message A.
2. Process message A.
3. Produce message B.
4. Acknowledge message A.

If the service crashes before the acknowledgement is performed (4), another instance will re-process message A, resulting in the publishing of message B with a sequence ID corresponding to the current instance, which will more likely differ from the sequence ID that the former instance would assign.

Executing all these operations in a transaction could help as long as the external effects involve only Pulsar; if we have a database or HTTP call, these need to be idempotent for this to work correctly. That said, transactions are not cheap: they incur a performance penalty, which is why it needs to be explicitly enabled at the broker level.

Having multiple producers publishing messages to the same topic is fine, and Pulsar deduplication should work smoothly as long as we have fixed number of producers with unique names. In any other case, it is recommended to either use Pulsar transactions or, when not possible, to deduplicate on the consumer side.

In a nutshell, relying on Pulsar's deduplication mechanism makes it easier. For multiple instances, distributed transactions (see Transactions further down) are the only way to guarantee no duplicates on the producer side.

3.2.3.3 Topic compaction

By default, Pulsar stores all unacknowledged messages produced on a topic while preserving message ordering. Any consumer can subscribe to a topic and start consuming messages from the very beginning (useful for event sourcing), but it can significantly slow down the startup of applications.

In a lot of cases, applications do not need the entire history of events but a selection of messages that yield the same latest state.

For this kind of use cases, Pulsar offers topic compaction[12], which allow for faster rewinds.

Compaction is not new to message brokers, though; databases use this mechanism to make file segments (how some of them store data) smaller.

In a few words, it works by setting a partition key to every message.

Say we publish messages containing the latest price of a product, and our service requires the latest price to compute the total of a shopping cart. We can set a unique property as the partition key (e.g. the product ID) to enable topic compaction.

[12]https://pulsar.apache.org/docs/en/concepts-topic-compaction/

```
Item(id = 100, price = 456.12)
Item(id = 263, price = 799.99)
Item(id = 100, price = 510.95)
```

Next time compaction gets triggered (can also be done manually), the topic will retain only the latest value per partition key.

```
Item(id = 263, price = 799.99)
Item(id = 100, price = 510.95)
```

We will see this feature in action in our application when we get to Chapter 6.

3.2.3.4 Transactions

Another Pulsar feature worth mentioning is transactions[13], which allows us to consume, process, and produce messages in one atomic operation.

It allows us to coordinate the consumption and production of multiple messages—possibly involving multiple topics—as one single operation.

This is not an exclusive Pulsar feature by any means; Apache Kafka has offered transactional support[14] for a long time.

We will learn more about how Neutron handles transactions from the Scala perspective when we get to Chapter 5 (see Distributed via Apache Pulsar), and how transactions are used in some critical pieces of our trading application.

[13] https://pulsar.apache.org/docs/en/transactions/
[14] https://www.confluent.io/blog/transactions-apache-kafka/

3.2.3.5 Pulsar IO

Pulsar IO connectors[15] offer a native way to interact with external systems like Apache Cassandra, Aerospike, and many others. It is the Pulsar answer to Kafka Connect[16].

There are *sources* and *sinks*. Sources feed data from external systems into Pulsar; sinks feed data from Pulsar into external systems.

Arguably, the most intriguing connectors are the CDC[17] ones (see Change Data Capture in Chapter 2).

Pulsar CDC connectors react to events coming in directly from the many supported datastores such as PostgreSQL and publish an event into a Pulsar topic, which preserves the order of the database operations.

The trading repository comes with a demonstration of this feature using PostgreSQL. Follow the README file instructions to run the `PulsarCDC` program found under the `demo` module.

In Chapter 7, we will discuss how we could take advantage of this feature to solve a specific application issue (see Forecasts' command handler).

3.2.4 What should I use?

There is no easy answer to this simple question. Both technologies deliver similar features, so we could implement the same system using either of them.

Depending on the requirements, we could also consider other technologies such as RabbitMQ or Kinesis (a good choice if we run our system on AWS, though vendor-locked).

However, having run both Kafka and Pulsar in production in past years, allow me to share with you my personal insights.

3.2.4.1 Setup and maintenance difficulty

Both message brokers require a hands-on system administrator who understands the broker's configuration pretty well—or a developer that takes on that role.

Kafka is probably the one that requires more tuning and maintenance—at least with the version requiring ZooKeeper—but I can't speak of newer versions using KRaft as I have no experience with them.

Pulsar is easier to get started with, and the configuration comes with sane defaults but still requires JVM tuning and other configurations we need to be aware of.

[15] https://pulsar.apache.org/docs/io-overview
[16] https://docs.confluent.io/platform/current/connect/index.html
[17] https://pulsar.apache.org/docs/io-cdc

3.2.4.2 Level of maturity

Arguably, Kafka with ZooKeeper is the most battle-tested solution. It has been used (and still is) by many Fortune 100 companies for years, so you won't go wrong with it.

On the other hand, if you do not mind trying out new technology that is simpler to use, delivers similar features, and is backed by a bunch of brilliant people, then Apache Pulsar could be a great option.

Kafka with KRaft is another great choice. However, both solutions are not as widely used as the former, so we should expect more bugs.

Although all software has bugs, it is a good measure for you to go through their repositories and evaluate the ratio between the number of issues and pull requests.

3.2.4.3 Usability

I may be a little biased, as I am the creator and maintainer of a Scala client, but I believe Pulsar is easier to use. In contrast, Kafka has a lot of complicated concepts that usually translate to complex clients (e.g. the many single-threaded operations).

Dealing with partitioned topics in Kafka is perhaps one of the pain points that make it more difficult than Pulsar, where these are completely optional.

However, partitioned topics usually deliver higher throughput, as multiple brokers can serve them, which is something to consider.

3.2.4.4 Clients availability

Suppose you work with multiple teams that use different programming languages but share the same infrastructure with the message broker. In that case, it is always good to check whether there are clients available for those languages or not.

Kafka, being the most mature, ships with support for multiple languages[18]. Pulsar is still playing catch-up[19], so this is something else to have in mind.

[18] https://cwiki.apache.org/confluence/display/KAFKA/Clients
[19] https://pulsar.apache.org/docs/en/client-libraries/

3.2.4.5 Streaming support

Kafka Streams allows us to group and aggregate data from different topics, which can then be pushed down to another topic for further processing. The concept is not new, but it has established itself as one of the most common streaming platforms with a rich community and ecosystem.

Pulsar Functions[20] is the lightweight equivalent on the Pulsar side, which supports more or less the same features, albeit not having a rich DSL.

We will come back to this topic when we get to Chapter 8 (see Centralized tracing).

Moreover, Pulsar ships with a few features that Kafka does not have, such as Geo-Replication[21].

3.2.4.6 Final thoughts

Caveats apply to all new software, and Pulsar and Kafka 3 (based on KRaft) are not the exception. Although both have a vibrant community and ecosystem behind them (even commercial support!), bugs now and then are to be expected. After all, it is free open source software we are leveraging.

Both Kafka and Pulsar would make for a solid choice, but ultimately, the decision is up to you. I think Pulsar shines in most areas, but it is still trying hard to earn its place in the main league.

You will need to analyze and assess which one fits your use case better.

[20] https://pulsar.apache.org/docs/en/functions-overview/
[21] https://pulsar.apache.org/docs/en/administration-geo/

3.3 State snapshots

Event sourcing allows us to retrieve the current application state by replaying all the events that occurred from the beginning. These are called *projections*.

As we process events over time, the number tends to become too big to handle, so performing a replay on every restart or deployment becomes a slow task.

A solution to this problem is to create snapshots of the current state every now and then, so when the application starts up, it only needs to fetch the latest snapshot and replay a small number of events instead of starting over from the beginning.

We saw how topic compaction enables faster rewinds (see Topic compaction), but it is not always possible to compact a topic, especially in event-sourced topics that need the entire history for auditability. In such cases, snapshots are the alternative.

We will have a lightweight implementation of snapshots written by a dedicated service and read by two other services in the trading application.

Lastly, it is worth mentioning this is one of the features provided by Akka Persistence[22], which is the go-to solution for Akka Cluster (see Application clustering).

3.3.1 Retention policy

Event sourcing implies persisting events in a durable topic, so services can have the ability to replay them from any point in time.

However, persisting events forever would take a considerable amount of disk space if we process millions of events per day. This is where retention policies come into play.

Since it is usually a good practice to have snapshots in event-sourcing applications, we do not need to persist all events unless required for auditability purposes.

Even then, there exist better solutions for auditability. E.g. we can move older events to another storage such as a data warehouse or keep relevant data stored in a proper database instead of relying on trillion events.

With state snapshots, we can configure a topic's retention policy with a short amount of time. Most brokers offer a sensitive default we can always tweak.

Pulsar supports persistence using an internal tool named Apache BookKeeper[23], a distributed write-ahead log (WAL). By default, only unacknowledged messages are persisted, but this is something that can be tuned via retention policies[24].

[22] https://doc.akka.io/docs/akka/current/typed/persistence-snapshot.html
[23] https://bookkeeper.apache.org/
[24] https://pulsar.apache.org/docs/en/cookbooks-retention-expiry/#retention-policies

Summing up, a system administrator needs to know these settings, working together with application developers to understand what configurations are the most suitable for the system at hand.

However, it does not end here. Monitoring and alerts are also a must in such systems; a topic we will discuss in earnest in Chapter 8 (see Monitoring).

3.4 Schema evolution

Schema evolution is all about compatibility between changes across time. Every time we change the shape of a datatype, we need to ensure other systems are aware and prepared to handle it. Otherwise, this can lead to incompatibility capable of breaking the entire system if we are not careful enough.

Schema changes need to be thought through thoroughly during the design phase. Any time later is already too late.

There are two ways to deal with schema evolution: schema compatibility and versioning. We can also combine both, though one should suffice.

3.4.1 Schema compatibility

Schema compatibility means that when we deploy a new version of a service with breaking schema changes, more recent instances are still capable of processing older messages and vice-versa.

There are a few different types of compatibility strategies:

- **backward**: newer instances can read data produced by older instances.
- **forward**: older instances can read data produced by newer instances.
- **full**: both backward and forward compatibility.

In most messaging systems, it is enough to have backward compatibility, even though forward compatibility is a nice feature to have (albeit not always possible).

Let's see an example using JSON to make things more evident. Say we start with this.

```
{
  "uuid": "171c546a-734c-479e-927e-33ddea086e50",
  "value": "foo"
}
```

What happens if we now add an extra field?

```
{
  "uuid": "171c546a-734c-479e-927e-33ddea086e50",
  "value": "foo",
  "code": 403
}
```

The answer is: it depends. If the JSON value needs to be produced by a component that does not know about anything about the `code` field, then we can only hard-code that value to be able to produce it.

On the other hand, if we only care about a consumer reading the JSON value, then the old decoder should still work, but it won't be aware of `code`.

If we had a way to know whether `code` was optional or not, we could learn more about how compatible a change is. This is precisely what schemas do for us. They provide a contract every party needs to agree upon for the entire system to be functional.

Unfortunately, JSON alone does not have the capability of modeling schemas. However, if the change is between Scala services that share the domain model, we can easily model the optionality of the newly added field.

```
case class Event(uuid: UUID, value: String, code: Option[Int])
```

Any JSON decoder we get from this case class will know how to deal with the presence and absence of such field in the raw JSON value.

When we need cross-boundary compatibility, schemas are the answer. Protocols such as Apache Avro[25], Protocol Buffers[26], and Thrift[27] are some popular choices.

For example, the last change can be represented as follows using Avro.

```
{
  "type" : "record",
  "name" : "Event",
  "namespace" : "org.arxiv..domain",
  "fields" : [ {
    "name" : "uuid",
    "type" : {
      "type" : "string",
      "logicalType" : "uuid"
    }
  }, {
    "name" : "value",
    "type" : "string"
  }, {
    "name" : "code",
    "type" : [ "null", "int" ],
    "default" : null
  } ]
}
```

[25] https://avro.apache.org/
[26] https://developers.google.com/protocol-buffers/
[27] https://thrift.apache.org/docs/concepts.html

As previously mentioned, it integrates perfectly with Kafka, while Pulsar ships with native schema support using the Avro format.

3.4.2 Versioning strategies

The second way of dealing with schema evolution is by deploying breaking changes under new endpoints. If we have a web service, then versioning our REST API is crucial.

I always recommend doing this, even if you do not deem it necessary during the design phase. E.g.

```
GET /api/v1/users/96bf41a4
```

As opposed to.

```
GET /api/users/96bf41a4
```

By design, we make the version we are dealing with crystal clear. Another way is to send the version in the HTTP headers, but I believe having it visible in the URL makes it more transparent.

Deploying a breaking change is now a matter of releasing a new HTTP endpoint.

```
GET /api/v2/users/96bf41a4
```

This has pros and cons. On the one hand, we can deploy breaking changes without coordinating with other teams. On the other hand, our service still needs to maintain code capable of processing both versions, which can quickly become a nightmare unless we plan accordingly.

This not only applies to HTTP endpoints. We can also deploy breaking changes of internal messages to versioned topics. E.g. `v1/users-topic` vs. `v2/users-topic`.

Let's say we have two user services producing and consuming messages—namely Royal and Thunder for internal reference—as fig. 3.4.1 shows.

If we deploy a new version of the Royal service, which includes breaking changes in the event schema, we would be breaking the Thunder service, now incapable of decoding the latest version of the events.

We can always deploy the new version to publish to a new versioned topic. However, it would create a large message backlog for our message broker without a consumer.

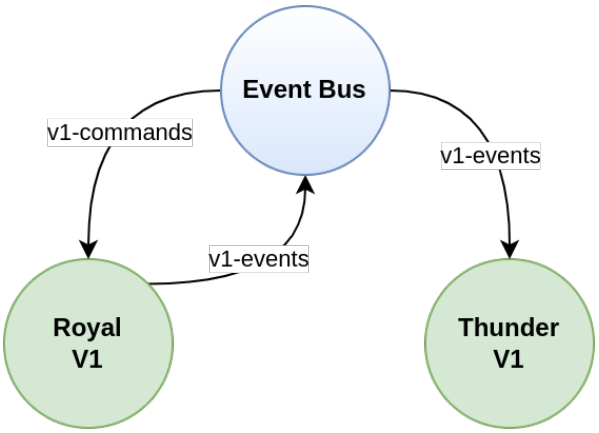


Figure 3.4.1: versioning strategy

So we should always start by deploying changes on the consumer side in such cases.

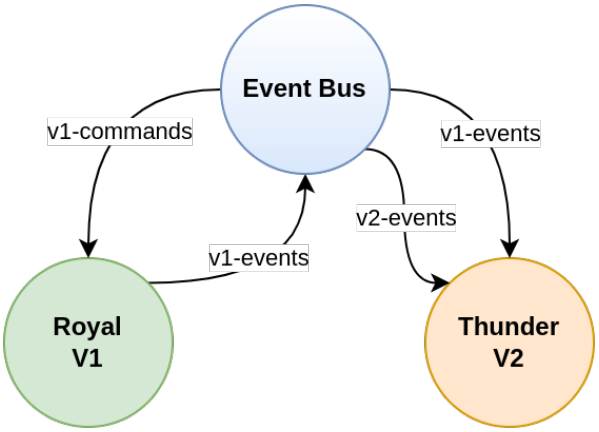


Figure 3.4.2: versioning strategy (step 1)

The strategy is to deploy a new version of the Thunder service that consumes from both versioned topics—and understands both schemas—even if there are no incoming messages yet from the `v2-events` topic.

Once this is deployed, we are free to deploy the publishing side, as fig. 3.4.3 shows.

Like the HTTP service, the consuming service also needs to maintain code capable of understanding both versions, so this requires careful coordination to allow the removal of the old code when no longer needed.

This also includes the removal of the old topics such as `v1-events`, checking all relevant messages have been successfully processed.

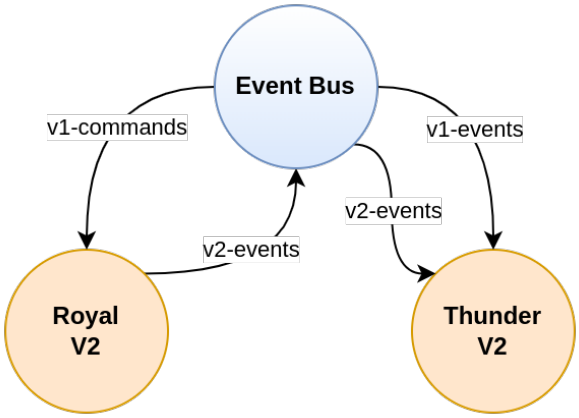


Figure 3.4.3: versioning strategy (step 2)

Overall, it is a decent and straightforward approach that works well in most scenarios. Still, schema compatibility checks provide stronger guarantees you may wish to have.

3.4.3 Schema registry

Apache Kafka can operate together with a schema registry running on a machine external to the brokers. Its responsibility is to maintain a record of all the data schemas written into the different topics, usually read at start-up.

As mentioned previously, Avro is an excellent choice, and there exist multiple solutions out there, both self-managed and as a cloud service.

On the other hand, Apache Pulsar has a built-in schema registry that enables clients to upload data schemas per topic. These schemas dictate which data types are recognized as valid for that topic.

In practice, consumers and producers check for schema compatibility strategy on start-up and fail to start (runtime error) if any incompatibility is detected.

It is highly recommended to keep track of your schemas in a version control system that any team member can easily access in both cases.

3.5 Summary

We have gone through many fundamental topics in very few pages—as we did with the two previous chapters—aiming to keep theory at its minimum and avoid reading what would make quite the boring book.

Nonetheless, the amount of given information should be sufficient to point you in the right direction when in need of more.

With that in mind, let me congratulate you for making it until the end of the theoretical section! In the following chapters, we will get hands-on on a lot of Scala 3 code!

Part II: Coding

Understanding the foundations of event-driven architecture and distributed systems is a wonderful skill to acquire, but where is the fun if we don't get to apply it to our code?

In this second part, we shift our focus to Scala 3 code while learning a few tricks and design patterns along the way.

We will also dive into effectful streams and data pipelines and learn how to deal with producer-consumer applications powered by Apache Kafka and Apache Pulsar.

4 Functional programming in Scala 3

Time to dive straight into the code, and what better way to do so than with Scala 3! This new major version of the language comes with exciting new features, some of which we will be exploring in this chapter, together with other functional libraries.

We will start with domain modeling and everything that applies to it: typeclass derivation, newtypes, refinement types, and orphan instances.

Next, we will briefly see typeclasses, capability traits, and HTTP routes. At last, we will explore a design pattern that leverages one of the brand-new features of the language.

4.1 Domain modeling

This topic has been extensively covered in PFPS, where we worked with newtypes and refinement types as the grand duo. This section will cover what this means in Scala 3, where some of the previously endorsed libraries are no longer supported.

4.1.1 Typeclass derivation

I recommend Derevo[1] for typeclass derivation in Scala 2, but it unfortunately relies on experimental macros—a feature dropped[2] in Scala 3—putting it off the table.

However, a great effort has been ongoing in Shapeless 3[3], the grounds upon which Kittens[4] builds on.

The following example showcases derivation for a few common typeclasses.

```
import cats.*
import cats.derived.*

case class Person(
    name: String,
    age: Int
) derives Eq, Order, Show
```

4.1.1.1 JSON codecs

The Circe[5] library—in its latest versions—ships with experimental support for typeclass derivation for Scala 3. E.g.

```
import io.circe.Codec

case class Address(
    streetName: String,
    streetNumber: Int,
    flat: Option[String]
) derives Codec.AsObject
```

It generates both `Decoder[Address]` and `Encoder[Address]` instances, as you can see in the following examples.

[1] https://github.com/tofu-tf/derevo
[2] https://docs.scala-lang.org/scala3/reference/dropped-features/macros.html
[3] https://github.com/typelevel/shapeless-3
[4] https://github.com/typelevel/kittens
[5] https://github.com/circe/circe

```
import io.circe.parser.decode
import io.circe.syntax.*

object Demo:
  @main def run =
    val address = Address("Baker", 221, Some("B"))
    val json    = address.asJson.spaces2
    assert(decode[Address](json) == Right(address))
```

If we print out that JSON, we will get the following value.

```
{
  "streetName" : "Baker",
  "streetNumber" : 221,
  "flat" : "B"
}
```

Another great JSON library is jsoniter-scala[6]. It requires us to first indicate how codecs are generated, but we can then leverage Scala 3's typeclass derivation mechanism.

```
import com.github.plokhotnyuk.jsoniter_scala.core.*
import com.github.plokhotnyuk.jsoniter_scala.macros.*

sealed trait JsonCodec[A] extends JsonValueCodec[A]

object JsonCodec:
  inline def derived[A]: JsonCodec[A] = new:
    private val impl = JsonCodecMaker.make[A](
      CodecMakerConfig.withDiscriminatorFieldName(Some("field"))
    )
    export impl._
```

For instance, the following datatypes can easily derive JSON codecs.

```
case class Person(
    age: Int,
    name: String
) derives JsonCodec

enum Digits derives JsonCodec:
  case One
  case Two(name: String)
```

Which can be verified with the following examples.

[6] https://github.com/plokhotnyuk/jsoniter-scala

```scala
val person = Person(40, "Joe")

val one = Digits.One
val two = Digits.Two("dos")

def show[A: JsonCodec](a: A): String =
  new String(writeToArray(a))

show(person) // {"age":40,"name":"Joe"}
show(one) // {"field":"One"}
show(two) // {"field":"Two","name":"dos"}
```

Although we will be using Circe in the trading application, the choice is always yours.

4.1.2 Newtypes

Furthermore, I endorse the Newtype[7] library in Scala 2, which gives us zero-costs wrappers. In other words, it allows us to define newtypes like the one shown below.

```scala
object domain {
  @newtype case class Age(value: Int)
}
```

Unfortunately, like Derevo, it also heavily depends on experimental macros. Nevertheless, it is not all bad news. Scala 3 ships with opaque types[8], which gives us the basic blocks upon which we can build newtypes.

The following newtypes encoding—which draws some inspiration from Monix Newtypes[9]—will be used in the trading application we are going to start developing in Chapter 6.

[7] https://github.com/estatico/scala-newtype
[8] https://docs.scala-lang.org/scala3/reference/other-new-features/opaques.html
[9] https://newtypes.monix.io/

```
abstract class Newtype[A](using
    eqv: Eq[A],
    ord: Order[A],
    shw: Show[A],
    enc: Encoder[A],
    dec: Decoder[A]
):
  opaque type Type = A

  inline def apply(a: A): Type = a

  protected inline final def derive[F[_]](using ev: F[A]): F[Type] = ev

  extension (t: Type) inline def value: A = t

  given Wrapper[A, Type] with
    def iso: Iso[A, Type] =
      Iso[A, Type](apply(_))(_.value)

  given Eq[Type]       = eqv
  given Order[Type]    = ord
  given Show[Type]     = shw
  given Encoder[Type]  = enc
  given Decoder[Type]  = dec
  given Ordering[Type] = ord.toOrdering
```

It is highly opinionated, thus it ought to be used with caution. It automatically derives common typeclass instances for our newtypes. Here's how we can use it.

```
type Name = Name.Type
object Name extends Newtype[String]

type Age = Age.Type
object Age extends Newtype[Int]
```

Not as clean as using macro-annotations, but it is the best we have for now.

Notes

The `Wrapper` typeclass defines an isomorphism that allows us to convert back and forth from the underlying wrapped type.

```
trait Wrapper[A, B]:
  def iso: Iso[A, B]
```

Moreover, we can define a few other abstract newtypes ubiquitous in any application. The first one is a newtype over `UUID`, which can automatically derive `IsUUID` for us.

```
abstract class IdNewtype extends Newtype[UUID]:
  given IsUUID[Type] = derive[IsUUID]
```

The `IsUUID` typeclass is defined as follows (also seen in PFPS).

```
trait IsUUID[A]:
  def iso: Iso[UUID, A]

object IsUUID:
  def apply[A: IsUUID]: IsUUID[A] = summon

  given IsUUID[UUID] with
    def iso: Iso[UUID, UUID] =
      Iso[UUID, UUID](identity)(identity)
```

The second is a numeric newtype, where we can add some extension methods.

```
abstract class NumNewtype[A](using
    eqv: Eq[A],
    ord: Order[A],
    shw: Show[A],
    enc: Encoder[A],
    dec: Decoder[A],
    num: Numeric[A]
) extends Newtype[A]:

  extension (x: Type)
    inline def -[T](using inv: T =:= Type)(y: T): Type =
      apply(num.minus(x.value, inv.apply(y).value))
    inline def +[T](using inv: T =:= Type)(y: T): Type =
      apply(num.plus(x.value, inv.apply(y).value))
```

4.1.2.1 Is it worth the trouble?

Let's face it: the user experience with this encoding could be better. For example, using Monix Newtypes—which has a very similar encoding—we define a newtype as follows.

```
type Name = Name.Type
object Name extends NewtypeWrapped[String]
```

Compare that to the following encoding.

```
case class Name(value: String)
```

It is only one more line of code to make it a zero-cost wrapper. However, if we consider typeclass derivation, the difference becomes more significant.

```
type Surname = Surname.Type
object Surname extends NewtypeWrapped[String]:
  given Eq[Surname] = derive
  given Order[Surname] = derive
  given Show[Surname] = derive
```

With a `case class`, it remains a one-liner.

```
case class Surname(value: String) derives Eq, Order, Show
```

Much nicer for the eyes! Of course, it's not a zero-cost wrapper. Still, the additional boxing of your newtypes will not be the bottleneck of your application unless you are doing some crazy memory-performant stuff.

The most important aspect of newtyping is to **avoid using primitive types** more than the boxing costs. Therefore, going for the encoding that gives the best developer experience is a sensible choice.

We get around the boilerplate by auto-deriving such typeclass instances directly in our custom `Newtype` encoding, but there are always exceptions for which we will need something different.

Another point worth discussing is the infamous `copy` method on case classes, as we will see in the next section.

So, is it worth going through all this trouble to have zero-cost wrappers? This is a good question you will need to answer with your team. After reading the next section, you will be equipped with the necessary information to make a reasonable decision.

4.1.3 Refinement types

Another library I usually endorse is Refined[10], which provides refinement types for Scala. However, it only supports runtime validation at the moment of writing, but compile-time validation should arrive at some point.

Given the current limitations, we will use Iron[11] instead, as it exclusively targets Scala 3. Here's an example taken directly from its documentation.

```
def log(x: Double :| Greater[0.0]): Double =
  Math.log(x) // used like a normal `Double`

log(-1d) // compile-time error
```

The `Symbol` datatype we will see in Chapter 6 leverages refinement types.

```
type SymbolR = DescribedAs[
  Match["^[a-zA-Z0-9]{6}$"],
  "A Symbol should be an alphanumeric of 6 digits"
]
```

The R in `SymbolR` stands for Refinement. We use them together with newtypes.

```
type Symbol = Symbol.Type
object Symbol extends Newtype[String :| SymbolR]
```

As shown by the `Symbol` encoding, we can define custom refinement types. The `SymbolR` constraint builds upon the existing `Match` datatype, but we can define our own by defining a `Constraint` instance.

```
final class Positive

given Constraint[Int, Positive] with
  override inline def test(value: Int): Boolean = value > 0
  override inline def message: String = "Should be strictly positive"

// compile-time error: Should be strictly positive
val x: Int :| Positive = 0
```

However, custom constraints on `String` can not always be inlined, unless we use macros. This means we would only get runtime support for such cases. The official documentation is a good entry point to learn more about it.

The following example showcases Iron's support for Cats and Circe in combination with Kittens (omitting imports for conciseness).

[10] https://github.com/fthomas/refined
[11] https://github.com/Iltotore/iron

```
type AgeR = DescribedAs[
  Greater[0] & Less[151],
  "Valid alien's age between 1 and 150"
]

type NameR = DescribedAs[
  Alphanumeric & MinLength[1] & MaxLength[50],
  "Valid alien's name, alphanumeric max 50 letters"
]

case class Alien(
    name: String :| NameR,
    age: Int :| AgeR
) derives Codec.AsObject, Eq, Show
```

Moreover, notice how we can combine constraints via intersection types (`&`).

Like Refined in Scala 2, we get both compile-time and runtime validation for free.

```
val alien1 = Alien("Bob", 120)
val alien2 = Alien("Bob", 500) // compile-time error: invalid age
val alien3 = Alien("", 50)     // compile-time error: invalid name
```

For the latter, we can leverage the `refine*` methods from the Cats module.

```
object Alien:
  def make(
      name: String,
      age: Int
  ): EitherNel[String, Alien] =
    (
      name.refineNel[NameR],
      age.refineNel[AgeR]
    ).parMapN(Alien.apply)
```

Using `EitherNel` together with `parMapN` gives us a great user experience.

4.1.3.1 The copy method

The `copy` method on case classes is always something to be aware of when not using refinement types, as it can bypass any previous validation.

Say we have the following case class with a smart constructor validating its inputs.

```
case class Pet(name: String)

object Pet:
  def make(name: String): Either[String, Pet] =
    if (name =!= "") then Pet(name).asRight
    else "Pet name must be non-blank!".asLeft
```

We could bypass validation by accessing the `copy` method in the following way.

```
Pet.make("Czela").map(_.copy(name = ""))
```

This wouldn't be a problem if we used refinement types, so this is one option.

```
case class Pet(name: String :| Not[Blank])
```

However, if we insist on using primitive types, we have two other options.

```
case class Pet1 private (name: String)

object Pet1:
  def make(name: String): Either[String, Pet1] =
    Pet1(name).asRight

sealed abstract case class Pet2(name: String)

object Pet2:
  def make(name: String): Either[String, Pet2] =
    Right(new Pet2(name) {})
```

Both `Pet1` and `Pet2` no longer synthesize a `copy` method for their instances.

```
// compile error: value copy is not a member of Pet2
def tryMe(pet: Pet2): Pet2 =
  pet.copy(name = "By-pass?")
```

Still, these approaches contribute to boilerplate that could go away with the usage of refinement types, making another compelling case for them.

4.1.4 Orphan instances

Orphan instances are those typeclass instances for types we don't control, e.g. from a third-party library. In Scala 2, I recommend creating a `trait` with the instances, which is then mixed in with the `domain` package object or wherever you like to keep your datatypes. E.g.

```
package object domain extends OrphanInstances

trait OrphanInstances {
  implicit val instantEq: Eq[Instant] =
    Eq.by(_.getEpochSecond)

  implicit val instantOrder: Order[Instant] =
    Order.by(_.getEpochSecond)

  implicit val instantShow: Show[Instant] =
    Show.show[Instant](_.toString)
}
```

However, Scala 3 ships with export clauses[12], a feature we can leverage to manage orphan instances. Instead of a `trait`, we can define all the instances within an `object`. Following the previous example, this is what we will end up with.

```
object OrphanInstances:
  given Eq[Instant]    = Eq.by(_.getEpochSecond)
  given Order[Instant] = Order.by(_.getEpochSecond)
  given Show[Instant]  = Show.show[Instant](_.toString)
```

Wait! If the instances are defined within an object, don't we have to import them every time we need them in scope? Well, yes. But we are not done just yet.

Scala 3 has dropped support for package objects[13], as we can now have top-level definitions of case classes, type aliases, and so on. So instead, we can have a `domain.scala` file that defines a bunch of newtypes in the `domain` namespace. Here is where we can export all the orphan instances—which will incidentally import them into scope—so they become available wherever we have `import domain.given`.

```
package domain

export OrphanInstances.given
```

[12] https://docs.scala-lang.org/scala3/reference/other-new-features/export.html
[13] https://docs.scala-lang.org/scala3/reference/dropped-features/package-objects.html

```
type Timestamp = Timestamp.Type
object Timestamp extends Newtype[Instant]
```

Neat! Isn't it? It feels great to take advantage of new features.

Furthermore, you will find that all the configuration instances for Ciris are defined in its own file. The sole reason for doing this is to be able to use the entire domain in Scala.js, where Ciris is not yet supported. In the trading repository, you will find more details.

4.2 Typeclasses

When it comes to typeclasses, there is not much of a difference between the different Scala versions. In Scala 3, however, the syntax is more concise.

Here we have a `Time` effect—also seen in PFPS—which we will be using once again in the trading application.

```
trait Time[F[_]]:
  def timestamp: F[Timestamp]

object Time:
  def apply[F[_]](using ev: Time[F]): Time[F] = ev

  given forSync[F[_]](using F: Sync[F]): Time[F] with
    def timestamp: F[Timestamp] =
      F.delay(Instant.now()).map(t ⇒ Timestamp(t))
```

Besides the more concise syntax and the update in the keywords `using` and `given` instead of `implicit`, the definition remains the same.

Scala 3 also supports context bound notation[14], so we could rewrite it as follows.

```
object Time:
  def apply[F[_]: Time]: Time[F] = summon

  given forSync[F[_]: Sync]: Time[F] with
    def timestamp: F[Timestamp] =
      Sync[F].delay(Instant.now()).map(t ⇒ Timestamp(t))
```

The `apply` method—aka the summoner—can now be implemented via `summon`, effectively replacing the old `implicitly` method.

Moreover, we do not need to name the given instance; it can be anonymous[15].

```
given [F[_]: Sync]: Time[F] with
  def timestamp: F[Timestamp] =
    Sync[F].delay(Instant.now()).map(t ⇒ Timestamp(t))
```

This is my preference; thus, we will stick with it in this book.

[14] https://docs.scala-lang.org/scala3/reference/contextual/context-bounds.html
[15] https://docs.scala-lang.org/scala3/reference/contextual/givens.html

4.3 HTTP routes

In Scala 2, my preference is to define HTTP routes as **`final case class`**es, mainly because we don't need to use the **`new MyRoutes`** syntax to instantiate them. E.g.

```
final case class HealthRoutes[F[_]: Monad]() extends Http4sDsl[F] {
  val routes: HttpRoutes[F] = HttpRoutes.of {
    case GET → Root / "health" ⇒ Ok()
  }
}

val rt: HttpRoutes[F] = HealthRoutes[F]().routes
```

The good news is this is no longer needed, as Scala 3 gives us universal apply methods[16], so we can define our routes as follows.

```
final class HealthRoutes[F[_]: Monad] extends Http4sDsl[F]:
  val routes: HttpRoutes[F] = HttpRoutes.of {
    case GET → Root / "health" ⇒ Ok()
  }
```

At call site, it can be instantiated in the following way.

```
val rt: HttpRoutes[F] = HealthRoutes[F].routes
```

Yay! Scala 3 makes everything more readable and concise. Notice how the empty parentheses are neither needed in the **`class`** definition nor when we instantiate it.

[16] https://docs.scala-lang.org/scala3/reference/other-new-features/creator-applications.html

4.4 Effectful context

In medium-to-big size applications, carrying a handful of dependencies across different components is ubiquitous, making wiring them not very simple.

The approach I have been recommending—and the one I use—is to group dependencies into different modules[17], which can either be traits or classes.

There are also those implicit dependencies that are not typeclasses but that we treat as capability traits because creating an instance is usually an effectful operation.

A good example is `Supervisor`, which can be passed around as an implicit dependency, so we don't need to manually thread it across the entire application when only a few components need it.

```
Supervisor[IO].use { implicit sp ⇒
  restOfTheProgram
}
```

If `Supervisor` is the only effectful dependency we have to share in our application, it should be enough to thread it across components as if it were a typeclass. E.g.

```
def foo[F[_]: Supervisor]: F[Unit] = ???

def bar[F[_]](using sp: Supervisor[F]): F[Unit] = ???
```

However, as the application grows, you might find yourself having two or more effectful dependencies and other information that needs to be available to most components. This can also be solved with modules, as previously noted.

Be that as it may, I would like to discuss a new Scala 3 feature that we can leverage in similar scenarios. For instance, let's say we want our application to be context-aware. We can first model our context datatype.

```
final class Log(ref: Ref[IO, List[String]]):
  def add(str: ⇒ String): IO[Unit] = ref.update(_ :+ str)
  def get: IO[List[String]]        = ref.get

final case class Ctx(
    id: UUID,
    sp: Supervisor[IO],
    log: Log
)
```

[17] https://github.com/gvolpe/pfps-shopping-cart/tree/second-edition/modules/core/src/main/scala/shop/modules

We have a unique application ID, a `Supervisor`, and a `Log`. It's all fixed to `IO` to keep things simple, but we could easily abstract over it if that's what we desire.

A typical way to deal with this would be to create the context and share it with the rest of the application as an implicit dependency. E.g.

```
def p1(using ctx: Ctx): IO[Unit] =
  IO.println("Running program 1") *> p2

def p2(using ctx: Ctx): IO[Unit] =
  IO.println("Running program #2") *>
    ctx.sp
      .supervise {
        ctx.log.add(s"Start: ${ctx.id}") >>
          IO.sleep(1.second) >>
          ctx.log.add(s"Done: ${ctx.id}")
      }
      .flatMap { fb ⇒
        ctx.log.add(s"Waiting: ${ctx.id}") >>
          fb.join.void
      }

def p3(using ctx: Ctx): IO[Unit] =
  IO.sleep(100.millis) *> IO.println("Running program 3") *> p4

def p4(using ctx: Ctx): IO[Unit] =
  IO.println(s"Running program 4: ${ctx.id.show}")

val mkCtx = for
  id <- Resource.eval(IO(UUID.randomUUID()))
  sp <- Supervisor[IO]
  lg <- Resource.eval(Ref.of[IO, List[String]](List.empty))
yield Ctx(id, sp, Log(lg))

val run: IO[Unit] =
  mkCtx.use { implicit ctx ⇒
    (p1 &> p3) *>
      ctx.log.get.flatMap(_.traverse_(IO.println))
  }
```

Or we could also replace `implicit` with a pattern-bound given instance.

```
mkCtx.use { case given Ctx ⇒ ??? }
```

Running this program should produce a similar output to the one shown below.

```
[info] running demo.EffectfulContext
Running program #1
Running program #2
Running program #3
Running program #4: 447c6c84-6d7b-4acd-8574-95145878c820
Start: 93dd53d1-ed2f-4d54-96b9-f446a0e503ff
Waiting: 93dd53d1-ed2f-4d54-96b9-f446a0e503ff
Done: 93dd53d1-ed2f-4d54-96b9-f446a0e503ff
```

This approach is very much acceptable, and it also works in Scala 2. However, notice how `p1` and `p3` do not use the context directly, but still need to declare it via `using ctx: Ctx` everywhere. If threading the implicit context was not a requirement, these methods could be simple values defined as `val` instead of `def`.

Scala 3 introduces context functions[18], an alternative solution we can explore. We could declare our program as a context function via the arrow notation whenever the context is not required. E.g.

```
val p1: Ctx ?=> IO[Unit] =
  IO.println("Running program 1") *> p2
```

Furthermore, `p1` does not need to be a method (i.e. defined via `def`) and can be made a value. Following this design, we can rewrite the previous example accordingly, starting with creating a helper method to initialize the context.

```
def withCtx(f: Ctx ?=> IO[Unit]): IO[Unit] =
  mkCtx.use { ctx =>
    f(using ctx) *> ctx.log.get.flatMap(_.traverse_(IO.println))
  }
```

Having it separated allows us to run other actions before and after the main program without mixing concerns. In this case, we print out the log contents to the standard output afterward.

Next, we can proceed with declaring the rest of the program.

```
val p1: Ctx ?=> IO[Unit] =
  IO.println("Running program #1") *> p2

def p2(using ctx: Ctx): IO[Unit] =
  IO.println("Running program #2") *> ???

val p3: Ctx ?=> IO[Unit] =
  IO.sleep(100.millis) *> IO.println("Running program #3") *> p4
```

[18] https://docs.scala-lang.org/scala3/reference/contextual/context-functions.html

```
def p4(using ctx: Ctx): IO[Unit] =
  IO.println(s"Running program 4: ${ctx.id.show}")

val run: IO[Unit] =
  withCtx {
    p1 &> p3
  }
```

Notice that we could also write both `p2` and `p4` as values, but we would need to summon the context manually. E.g.

```
val p2: Ctx ?=> IO[Unit] =
  val ctx = summon[Ctx]
  restOfTheProgram
```

Or we could also define a summoner method in the `Ctx` companion object.

This feature is mainly recommended to write pretty DSLs, and that's more or less what we have done with the `withCtx` method. The only difference with the Scala 2 approach, is that we do not need to declare that the instantiated context should be `implicit` (or `given`). Instead, context functions make sure it is synthesized as such.

This is quite useful when we have multiple levels of nesting, as in the official example.

```
table {
  row {
    cell("top left")
    cell("top right")
  }
  row {
    cell("bottom left")
    cell("bottom right")
  }
}
```

Though, it seems to be an over-killer for wiring dependencies. Nonetheless, it is a feature worth exploring!

4.5 Dependent types

The last new feature we will discuss is Match Types[19].

```
type Elem[X] = X match
  case String      ⇒ Char
  case Array[t]    ⇒ t
  case Iterable[t] ⇒ t
```

Match types enable dependent typing, and we will leverage them when declaring dependencies in our finite state machines (see Chapter 5).

Let's start with an example. Say we have the following graph datatype defining a state `St`, a dependency `Dep`, an input `In`, and an output `Out`.

```
abstract class Graph[F[_], St, Dep, In, Out]:
  val dep: Dep
```

As well as the following concrete state datatypes.

```
type CatState = Map[String, String]
type DogState = Set[Int]
type FoxState = Array[Long]
```

We can then create a graph of each animal, for instance.

```
val cat = new Graph[IO, CatState, List[String], String, Unit]:
  val dep = List.empty

val dog = new Graph[IO, DogState, Vector[Int], Int, Unit]:
  val dep = Vector.empty

val fox = new Graph[IO, FoxState, Set[Long], Long, Unit]:
  val dep = Set.empty
```

In these examples, we can observe an implicit relationship between the different types. We can see that when the input type is `String`, the state type is `CatsState`, and the dependency type `List[String]`. However, nothing stops us from creating another cat instance with different state and dependency types.

```
val wrongCat = new Graph[IO, FoxState, Boolean, String, Unit]:
  val dep = List.empty
```

To better convey our intentions, we can enforce this relationship by leveraging match types—assuming there is a unique state and dependency type for a given input.

[19] https://docs.scala-lang.org/scala3/reference/new-types/match-types.html

```
type GraphSt[In] = In match
  case String ⇒ CatState
  case Int    ⇒ DogState
  case Long   ⇒ FoxState

type GraphDep[In] = In match
  case String ⇒ List[String]
  case Int    ⇒ Vector[Int]
  case Long   ⇒ Set[Long]
```

We can then declare a type that represents this relationship, and create a smart constructor within the companion object of `Graph`.

```
type Graf[In] = Graph[IO, GraphSt[In], GraphDep[In], In, Unit]

object Graph:
  def make[In](_dep: GraphDep[In]): Graf[In] = new:
    val dep = _dep
```

We should also make its abstract class constructor private. Now creating correct instances should be straightforward, with clear explicit rules at the type level.

```
val _cat = Graph.make[String](List.empty)
val _dog = Graph.make[Int](Vector.empty)
val _fox = Graph.make[Long](Set.empty)
```

Attempting to create wrong instances will now result in a compilation error.

```
// TypeError: Match type reduction failed...
val no1 = Graph.make[Long](List.empty)
val no2 = Graph.make[Int]("wrong")
```

Another benefit we get is type inference, which works flawlessly in all these examples.

Match types help us to enforce the popular rule in the FP world: **Make illegal state unrepresentable.** In the final chapter, we will unveil its usage in the tracing finite state machines (see FSM Dependent Types).

4.6 Summary

We have explored the world of domain modeling, from typeclass derivation via the Kittens library to newtypes with a custom encoding. Furthermore, we also looked into refinement types and orphan instances.

Moreover, we have learned about the subtle changes in typeclass encoding and HTTP routes. Finally, we discovered new language features by learning about effectful context and dependent types.

As exciting as this new major version of the language is, remember that it is still evolving, which means many of the features discussed in this chapter might change in newer releases.

5 Effectful streams

The Fs2[1] library—aka Functional streams for Scala—is undoubtedly my old-time favorite. It came to life originally as scalaz-stream[2], as part of the famous red book[3] (now on its second edition[4]), to evolve for many years to the current state of the art.

In this chapter, we will review practical examples and software design concepts that will be key when we get to work on our streams-powered distributed system.

[1] https://fs2.io/
[2] https://github.com/scalaz/scalaz-stream/tree/master
[3] https://www.manning.com/books/functional-programming-in-scala
[4] https://www.manning.com/books/functional-programming-in-scala-second-edition

5.1 Finite state machines

We have briefly seen this topic in PFPS, but let's review it once again, as we will use it a lot in the trading application. Quoting Wikipedia[5]:

> A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

The following representation written in Scala 3 is the one we will use.

```
import cats.syntax.all.*
import cats.{ Functor, Id }

case class FSM[F[_], S, I, O](run: (S, I) ⇒ F[(S, O)]):
  def runS(using F: Functor[F]): (S, I) ⇒ F[S] =
    (s, i) ⇒ run(s, i).map(_._1)

object FSM:
  def id[S, I, O](run: (S, I) ⇒ Id[(S, O)]) = FSM(run)
```

The `run` function takes a state `S` and an input `I` and produces a new state `S` and an output `O` within a context `F`. When we don't need any context, we can use the identity FSM, represented via `cats.Id`.

The `runS` method is an extension that runs the state machine and discards its output, only producing a new state. We could also write a similar method that only returns the output and discards the new state if we need it at some point.

5.1.0.1 Trading FSM

Here's a sneak peek of the code we will be writing and using in the next chapter, which models the trading engine as a finite state machine.

[5] https://en.wikipedia.org/wiki/Finite-state_machine

```
object TradeEngine:
  val fsm =
    FSM.id[
      TradeState,
      TradeCommand | SwitchCommand,
      (EventId, Timestamp) ⇒ TradeEvent | SwitchEvent
    ] {
      // Trading status: On
      case (st @ TradeState(On, _), cmd @ Create(_, cid, sl, ac, p, q, _, _)) ⇒
        val nst = st.modify(sl)(ac, p, q)
        nst → ((id, ts) ⇒ CommandExecuted(id, cid, cmd, ts))
      case (st @ TradeState(On, _), cmd @ Update(_, cid, sl, ac, p, q, _, _)) ⇒
        val nst = st.modify(sl)(ac, p, q)
        nst → ((id, ts) ⇒ CommandExecuted(id, cid, cmd, ts))
      case (st @ TradeState(On, _), cmd @ Delete(_, cid, sl, ac, p, _, _)) ⇒
        val nst = st.remove(sl)(ac, p)
        nst → ((id, ts) ⇒ CommandExecuted(id, cid, cmd, ts))
      // Trading status: Off
      case (st @ TradeState(Off, _), cmd: TradeCommand) ⇒
        val rs = Reason("Trading is off")
        st → ((id, ts) ⇒ CommandRejected(id, cmd.cid, cmd, rs, ts))
      // Trading switch: On / Off
      case (st @ TradeState(Off, _), Start(_, cid, _)) ⇒
        val nst = TradeState._Status.replace(On)(st)
        nst → ((id, ts) ⇒ Started(id, cid, ts))
      case (st @ TradeState(On, _), Stop(_, cid, _)) ⇒
        val nst = TradeState._Status.replace(Off)(st)
        nst → ((id, ts) ⇒ Stopped(id, cid, ts))
      case (st @ TradeState(On, _), Start(_, cid, _)) ⇒
        st → ((id, ts) ⇒ Ignored(id, cid, ts))
      case (st @ TradeState(Off, _), Stop(_, cid, _)) ⇒
        st → ((id, ts) ⇒ Ignored(id, cid, ts))
    }
```

Without getting into details about the datatypes on display (we will learn more about them in Chapter 6), we can observe how we are able to model the state transitions via `TradeState` by processing incoming commands and producing events.

We can get away by using the identity FSM because we are "delaying" the creation of `EventId` and `Timestamp` to create each event. If we wanted `F[TradeEvent | SwitchEvent]` as an output, then our effect type `F` would need different capabilities such as `Applicative`, `GenUUID`, and `Time`, which `Id` could not fulfill.

Because state machines are plain functions, these are extremely easy to test by feeding them an initial state and input (command) and writing assertions on the expected output

(state and event). E.g.

```
test("Trade engine fsm") {
  val st1 = fsm.runS(TradeState.empty, createCmd)
  val ex1 = TradeState(On, pricesMap)
  expect.same(st1, ex1)
}
```

We don't get into details here, but it illustrates what such a test may look like.

5.1.0.2 Streams integration

Fs2 provides two methods that fit the shape of the finite state machine's `run` function: `mapAccumulate` and `evalMapAccumulate`.

Below is the simplified type signature of both (omitting variance).

```
def mapAccumulate[S, O](s: S)(f: (S, I) ⇒ (S, O)): Stream[F, (S, O)]
def evalMapAccumulate[S, O](s: S)(f: (S, I) ⇒ F[(S, O)]): Stream[F, (S, O)]
```

Continuing with the trading FSM, we could use it as follows.

```
val commands: Stream[IO, TradeCommand | SwitchCommand] = ???
commands.evalMapAccumulate(TradeState.empty)(fsm.run)
```

For example, the commands can result from consuming messages from a Pulsar topic.

Most of the code we will write in the following chapter will roughly follow this shape whenever we need to process state transitions.

To see more FSM examples, check out this blog post[6] I wrote a while ago.

[6] https://gvolpe.com/blog/fsm-fs2-a-match-made-in-heaven/

5.2 Resources and lifecycle

All libraries that integrate with Cats Effect can leverage the Resource[7] datatype (and its instances), which models the necessity of performing a clean-up action in case of completion or failure.

A resource is usually expensive to acquire in the case of an HTTP server or a connection to a database. Though, it could only be that we must ensure a final computation is executed before the resource is shutdown.

A clear example is the default Http4s server: *Ember*.

```
val mkServer: Resource[IO, Server] =
  EmberServerBuilder
    .default[IO]
    .withHost(host"0.0.0.0")
    .withPort(port"8080")
    .build
```

Another example is creating a Redis connection.

```
Redis[IO].utf8("redis://localhost").use { redis ⇒
  for
    _ <- redis.set("foo", "123")
    x <- redis.get("foo")
    _ <- redis.setNx("foo", "should not happen")
    y <- redis.get("foo")
    _ <- IO(assert(x == y))
  yield ()
}
```

However, there is a difference between these two types of resources. The former is generally run as a long-living computation, ignoring the produced `Server` instance.

```
// same as `.use(_ ⇒ IO.never)`
mkServer.useForever
```

Conversely, a Redis connection is commonly shared with other components. E.g.

```
Redis[IO].utf8("redis://localhost").use { redis ⇒
  serviceOne(redis) *> serviceTwo(redis)
}
```

[7] https://typelevel.org/cats-effect/docs/std/resource

Unquestionably, there is one thing these two have in common: they typically appear at the top layer of the application, where we initialize a sequence of resources so we can build our application and run it either in `IO` or `Stream`.

Fs2's `Stream` ships with a `resource` method to integrate with Cats Effect's `Resource`, making it a perfect fit for our Redis connection. It also fits our HTTP server, but not without its caveats.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (serverRes, redisRes) ⇒
      Stream.eval(serverRes.useForever).concurrently {
        Stream.resource(redisRes).evalMap { redis ⇒
          serviceOne(redis) *> serviceTwo(redis)
        }
      }
    }
    .compile
    .drain
```

In this example, we still prefer the `useForever` method lifted into a stream via `Stream.eval`, as that's the shortest version. Although we could still use `Stream.resource`, it needs to be followed by `Stream.never` to achieve the same semantics.

```
(Stream.resource(serverRes) >> Stream.never[IO])
  .concurrently {
    Stream.resource(redisRes).evalMap { redis ⇒
      serviceOne(redis) *> serviceTwo(redis)
    }
  }
```

In our application, we will stick with the former `useForever`, but you are free to choose.

5.3 Data pipelines

Fs2 is an incredible tool for building data pipelines[8]. It allows us to hook up different data sources such as files, databases, or even incoming messages from a message broker.

Every case is different, however, generally demanding distinctive treatment. For instance, a pipeline built for real-time data processing should not be treated as an analytics pipeline or a batch-processing pipeline.

The following diagram showcases a hybrid data pipeline.

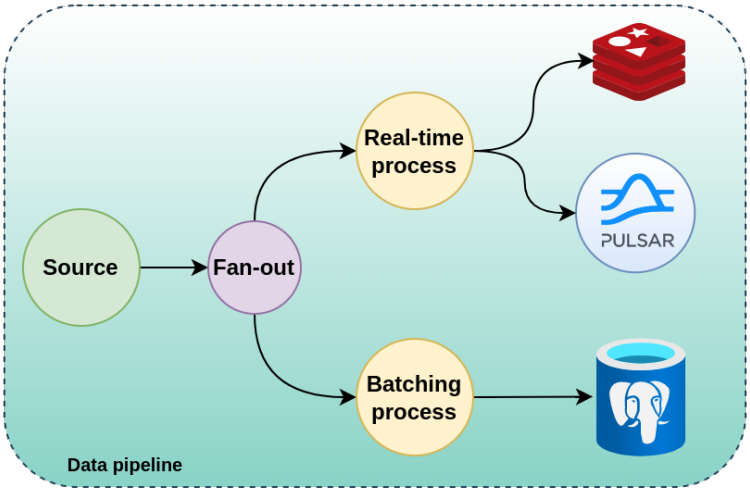


Figure 5.3.1: data pipeline

In this section, we will study a few examples and ideas that will not only be applicable in the system we will develop together in the next chapter, but also in your career as a Software Engineer, especially when it comes to the field of system design.

5.3.1 Real-time

Data pipelines designed for real-time processing require high throughput. E.g. receiving events related to a football match—such as `GoalScored` or `CornerKickAwarded`—followed by reflecting the changes in our system, which may involve some computations, updating a cache, publishing other messages, and so on.

On the other hand, if every event needs to be stored in a database, we will be sacrificing throughput. Therefore, it is essential to separate real-time processing from this case, as we will see in further examples.

[8] https://en.wikipedia.org/wiki/Pipeline_(computing)

These pipelines can be split between those that need the order to be preserved and those where retaining order is not required. The former can usually be achieved either via `evalMap` or `parEvalMap`.

```
val events: Stream[IO, FootballEvent] = ???

events.parEvalMap(maxConcurrent = 10) {
  process(_).flatMap { e ⇒
    sendMessage(e) &> updateCache(e)
  }
}
```

In general, `parEvalMap` performs better when the source produces a large amount of data. If this is not the case, the differences with `evalMap` may not be noticeable.

Conversely, when retaining the order of events is unnecessary, we can leverage `parEvalMapUnordered` instead (aka `mapAsyncUnordered`).

5.3.2 Batching

Another mainstream case for data pipelines relates to processing expensive computations for every message received, such as writing to a SQL database. This is known as the *fast producer, slow consumer* scenario, which can drastically affect performance when overlooked. To deal with it, we can process the data in batches.

If we only try using `evalMap` or `parEvalMap` as in the previous example, our application will more likely struggle to keep up with the incoming data, leading to potential inconsistencies and even runtime crashes.

For such cases, Fs2 lets us operate in *chunks* via the `fs2.Chunk` datatype. It represents a strict, finite sequence of values that allows index-based random access of elements.

```
scala> Stream(1,2,3).repeat.chunkN(2).take(5).toList
res0: List[Chunk[Int]] = List(
  Chunk(1, 2), Chunk(3, 1), Chunk(2, 3), Chunk(1, 2), Chunk(3, 1)
)
```

You can learn about its intricacies in this talk[9] given by Michael Pilquist.

In our example, we can convert our stream into a stream of chunks before persisting the events, as shown in the code snippet below.

[9] https://mpilquist.github.io/fs2-chunk-talk

```
events
  .chunkN(1000)
  .zip(Stream.iterate(0)(_ + 1))
  .parEvalMap { (c, n) ⇒
    IO.println(s"Chunk #$n") *> persist(c)
  }

def persist(chunks: Chunk[FootballEvent]): IO[Unit] = ???
```

We `zip` it with an index to log how many chunks we are processing, but that's optional.

5.3.3 Analytics

Depending on the use case, an analytics data pipeline could either be a real-time pipeline, a batching pipeline, or a combination of both.

Every scenario is different and might require special considerations. Nonetheless, we can follow some guidelines. For example, we can perform real-time processing and only after that run the batching operations in chunks whenever the latter depends on the results of the real-time computations.

```
events
 .parEvalMap(10) {
   process(_).flatMap { e ⇒
     sendMessage(e) &> updateCache(e).as(e)
   }
 }
 .chunkN(1000)
 .parEvalMap(10)(persist)
```

However, we should be wary of doing this, as the batch-processing will still affect the real-time processing. A better way to make these processes more independent is to introduce an internal topic to achieve a fan-out topology. E.g.

```
Stream.eval(Topic[IO, FootballEvent]).flatMap { topic ⇒
  val realTime = consumer.receive.parEvalMap(10) {
    process(_).flatMap { e ⇒
      (
        sendMessage(e), updateCache(e), topic.publish1(e)
      ).parSequence_
    }
  }

  val batching =
    topic
```

```
      .subscribe(1000)
      .chunkN(1000)
      .parEvalMap(10)(persist)

  Stream(realTime, batching).parJoinUnbounded
}
```

Even better, if the batching process does not depend on real-time results, we can perform both operations by subscribing to our local topic.

```
Stream.eval(Topic[IO, FootballEvent]).flatMap { topic ⇒
  val realTime =
    topic.subscribe(1000).parEvalMap(50) {
      process(_).flatMap { e ⇒
        sendMessage(e) &> updateCache(e)
      }
    }

  val batching =
    topic
     .subscribe(1000)
     .chunkN(1000)
     .parEvalMap(10)(persist)

  Stream(
    realTime,
    batching,
    consumer.receive.through(topic.publish)
  ).parJoinUnbounded
}
```

The previous paragraph says "more independent" because these computations would not be 100% independent of each other. They will run as a single application, potentially sharing the same thread pools and some other resources.

5.3.4 Data source

Every data pipeline begins with a data source; it can originate from a file, a database, a cache, a network connection, a message broker, etc.

5.3.4.1 Files

The fs2.io.file[10] API might be sufficient for processing files, but if you need to process known file formats such as CSV, XML, or JSON, I highly recommend checking out the fs2-data[11] library.

Here's an example that reads and parses a CSV file into a `Movie` datatype.

```
import fs2.data.csv.*
import fs2.data.csv.generic.semiauto.*
import fs2.io.file.{ Files, Path }

// Stream[IO, Movie]
Files[IO]
  .readAll(Path("dataset/movies.csv"))
  .through(fs2.text.utf8.decode)
  .through(decodeUsingHeaders[Movie]())
```

Or how about a good old XML file?

```
import fs2.data.xml.*

// Stream[IO, XmlEvent]
Files[IO]
  .readAll(Path("dataset/demo.xml"))
  .through(fs2.text.utf8.decode)
  .through(events[IO, String])
```

Last but not least, it is worth mentioning that other libraries support these and other file formats that are also compatible with Fs2 streams. Readers are always encouraged to research the area and pick one; `fs2-data` is only my preferred library.

This is another great advantage of using Fs2: its incredible ecosystem[12]!

5.3.4.2 Databases

When it comes to databases, it depends on whether the client you use supports Fs2 streams. If it does not, you may be able to add it yourself, as Fs2 gives you the tools. However, it might need to be done directly in the client library.

[10] https://fs2.io/#/io?id=files
[11] https://github.com/satabin/fs2-data
[12] https://fs2.io/#/ecosystem

Speaking of PostgreSQL, both Doobie[13] and Skunk[14] support streaming queries. Here's a quick example using the former.

```
sql"SELECT name FROM country"
  .query[String]
  .stream // Stream[ConnectionIO, String]
```

If Skunk is your jam, the following snippet will suit you better.

```
val e: Query[String, String] =
  sql"SELECT name FROM country".query(varchar)

Stream
  .resource(session.prepare(e))
  .flatMap(_.stream("U%", 64))
```

Both libraries have excellent native integration with Fs2.

5.3.4.3 Networking

For networking data sources such as raw TCP and UDP connections, the fs2.io.net[15] API is a good starting point.

With it, you can write a minimalistic echo TCP server in a few lines of code.

```
Network[IO].server(port = Some(port"5555")).map { client ⇒
  client.reads
    .through(fs2.text.utf8.decode)
    .through(fs2.text.lines)
    .interleave(Stream.constant("\n"))
    .through(fs2.text.utf8.encode)
    .through(client.writes)
    .handleErrorWith(_ ⇒ Stream.empty)
}.parJoin(100)
```

On the other hand, for something more high level, you may find existing third-party libraries such as fs2-grpc[16], among others.

[13] https://tpolecat.github.io/doobie/docs/04-Selecting.html#internal-streaming
[14] https://tpolecat.github.io/skunk/tutorial/Query.html#parameterized-query
[15] https://fs2.io/#/io?id=networking
[16] https://github.com/typelevel/fs2-grpc

5.3.4.4 Message brokers

In the next section, we will learn more about communicating with Apache Pulsar and Apache Kafka from purely functional Scala code. Notwithstanding, you should know there are many other message brokers and protocols such as RabbitMQ, ZeroMQ, MQTT (Mosquito), AWS Kinesis, AWS SQS, Google Cloud Pub/Sub, etc.

For these too, you may find a suitable library. E.g. here's a code snippet using the fmq[17] library (ZeroMQ client).

```
import io.fmq.*
import io.fmq.socket.pubsub.Subscriber
import io.fmq.syntax.literals.*

val topic = Subscriber.Topic.utf8String("demo"))

Stream.resource {
  Context
    .create[IO](1)
    .evalMap(_.createSubscriber(topic))
    .flatMap(_.connect(tcp"://localhost:31234"))
}.flatMap { socket ⇒
  Stream.repeatEval(socket.receiveFrame[String])
}
```

Followed up by examples of AWS Kinesis and AWS SQS using the fs2-aws[18] library.

```
import fs2.aws.*

val kinesis: Stream[IO, CommittableRecord] =
  readFromKinesisStream[IO]("appName", "streamName")

val sqs: Stream[IO, String] =
  sqsStream[IO, String](
    sqsConfig,
    (cfg, cb) ⇒ SQSConsumerBuilder(cfg, cb)
  ).map(_.body())
```

Regardless of your data source, all the data pipelines previously outlined can be considered for your application, depending on the use case.

[17] https://github.com/iRevive/fmq
[18] https://github.com/laserdisc-io/fs2-aws

5.4 Producer-consumer

A producer and consumer can represent a specific message broker such as Kafka or Pulsar. From Scala code, we can interact with them via predefined interfaces from the client library in use.

However, nothing stops us from writing our own abstractions. Let's start with the interface of the `Producer`, which is the simplest one in our case.

```
trait Producer[F[_], A]:
  def send(a: A): F[Unit]
  def send(a: A, properties: Map[String, String]): F[Unit]
```

It is parameterized with `F[_]` (the effect type) and `A` (the message type). So the action of producing a message is represented as a function `A ⇒ F[Unit]` if the `send` method were to be converted. A second variant allows us to send properties (aka *metadata*) in addition to the message payload.

Likewise, the `Consumer` is parameterized with `F[_]` and `A`, albeit defining a few different methods.

```
trait Acker[F[_], A]:
  def ack(id: Consumer.MsgId): F[Unit]
  def ack(ids: Set[Consumer.MsgId]): F[Unit]
  def nack(id: Consumer.MsgId): F[Unit]

trait Consumer[F[_], A] extends Acker[F, A]:
  def receiveM: Stream[F, Consumer.Msg[A]]
  def receiveM(id: Consumer.MsgId): Stream[F, Consumer.Msg[A]]
  def receive: Stream[F, A]
  def lastMsgId: F[Option[Consumer.MsgId]]

object Consumer:
  type MsgId = String
  type Properties = Map[String, String]

  final case class Msg[A](id: MsgId, props: Properties, payload: A)
```

We have all the acknowledgement-related functions grouped in a separate interface named `Acker`, which we will see in action in Chapter 7.

Having these interfaces (or tagless algebras), we can already model some logic. For example, a producer-consumer program can be expressed via the `concurrently` method, which has the following simplified type signature.

```
def concurrently[O](that: Stream[F, O]): Stream[F, O]
```

The stream will be interrupted whenever the left-hand side stream finishes. So generally, we may want to gracefully shut down the program only when the consumer terminates.

```
val c1 =
  consumer.receive
    .evalMap(n ⇒ IO.println(s"Consumed: $n"))

val p2 =
  Stream.range(0, 100)
    .evalMap(producer.send)

c1.concurrently(p2)
```

The producer program `p2` will terminate once the consumer program `c1` finishes. If, on the other hand, we may prefer to run them as independent processes, we should use `parJoin` or `parJoinUnbounded` instead.

```
Stream(c1, p2).parJoin(2)
```

The latter is also useful when we want to run multiple streams, which may include an HTTP server. The approach is so common that most of our services will look similar.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (consumer, topic, server) ⇒
      val http =
        Stream.eval(server.useForever)

      val subs =
        topic.subscribers.evalMap { n ⇒
          Logger[IO].info(s"WS connections: $n")
        }

      val alerts =
        consumer.receive.through(topic.publish)

      Stream(http, subs, alerts).parJoin(3)
    }
    .compile
    .drain
```

This is actually a preview of the initial design of the service that handles Web Sockets and alerts (see Web Sockets in Chapter 6). It runs three tiny programs:

1. HTTP server (represented as a `Resource`, so we call `useForever` within `Stream.eval`).

2. Logger of the current number of WS connections (subscribers).
3. Consumer of alerts that publishes them to an internal topic.

All of these programs run independently of each other. Only in case of failure the whole stream would fail, so we need to handle this one way or another or simply let it crash.

5.4.1 In-memory via Queue

We can now write some interpreters for our `Consumer` and `Producer`. Let's start with the latter by defining the following constructor its companion object.

```
import cats.effect.std.Queue

def local[F[_]: Applicative, A](
    queue: Queue[F, Option[A]]
): Resource[F, Producer[F, A]] =
  Resource.make[F, Producer[F, A]](
    Applicative[F].pure(
      new:
        def send(a: A): F[Unit] = queue.offer(Some(a))
        def send(a: A, properties: Map[String, String]): F[Unit] = send(a)
    )
  )(_ ⇒ queue.offer(None))
```

We use an internal `Queue[F, Option[A]]` so our program gracefully terminates whenever a `None` value is published.

> **Notes**
>
> We could also use an `fs2.concurrent.Topic` instead of a `Queue`, depending on the semantics we are looking for.

Next comes the `Consumer`, also based on the same `Queue`, as these should have a way of communication.

```
def local[F[_]: Applicative, A](
    queue: Queue[F, Option[A]]
): Consumer[F, A] = new:
  def receiveM: Stream[F, Msg[A]]       = receive.map(Msg("N/A", _))
  def receive: Stream[F, A]             = Stream.fromQueueNoneTerminated(queue)
  def ack(id: Consumer.MsgId): F[Unit]  = Applicative[F].unit
  def nack(id: Consumer.MsgId): F[Unit] = Applicative[F].unit
  ...
```

Both `ack` and `nack` are a no-op in this implementation, as there is no such concept with an in-memory queue. The `receiveM` does not make much sense either, but it still can be implemented.

Lastly, the `receive` method uses the `fromQueueNoneTerminated` streaming method, which terminates whenever a `None` value is published to the internal queue.

We are now ready to write a demo program using what we have so far.

```
def run: IO[Unit] =
  Queue.bounded[IO, Option[String]](500).flatMap { q ⇒
    val consumer = Consumer.local(q)
    val producer = Producer.local(q)

    val p1 =
      consumer.receive
        .evalMap(s ⇒ IO.println(s">>> GOT: $s"))

    val p2 =
      Stream
        .resource(producer)
        .flatMap { p ⇒
          Stream
            .sleep[IO](100.millis)
            .as("test")
            .repeatN(3)
            .evalMap(p.send)
        }

    IO.println(">>> Initializing in-memory demo <<<") *>
      p1.concurrently(p2).compile.drain
  }
```

It will produce the following output when executed.

```
>>> Initializing in-memory demo <<<
>>> GOT: test
>>> GOT: test
>>> GOT: test
```

You can find all the standalone examples under the demo module[19].

This is fun, isn't it? Now let's take it to the next level.

[19] https://github.com/gvolpe/trading/tree/main/modules/x-demo/src/main/scala/demo

5.4.2 Distributed via Apache Pulsar

The real fun begins with distributed consumers and producers, powered by Apache Pulsar. We will be using the Neutron[20] library, so it is recommended to check out its documentation for a complete insight.

Let's now write interpreters for our `Consumer` and `Producer`, starting with the latter.

5.4.2.1 Producer

One of the supported Pulsar subscription modes is Key-Shared[21], as briefly mentioned in Chapter 3 (see Subscriptions).

Producers can set an *ordering key* to every message to dictate how these are delivered to consumers attached to a subscription in such mode. This key can be seen as another metadata field, for example:

```
val m1 = Message("key-1", "a1")
val m2 = Message("key-2", "b1")
val m3 = Message("key-3", "a2")
```

If we have two consumers subscribed in key-shared mode, namely `c1` and `c2`, both `m1` and `m3` will be sent to `c1`, whereas `c2` will only receive `m2`.

Setting an ordering key (or *message key*) to control how messages are delivered is also known as *sharding*, which is the terminology we will be using from now on.

So before we start writing the `Producer` interpreter, let's introduce a `Shard` typeclass that defines how a type can produce a `ShardKey`.

```
import dev.profunktor.pulsar.ShardKey

trait Shard[A]:
  def key: A ⇒ ShardKey
```

We can create a default instance—meaning no sharding—or an instance for our type.

```
object Shard:
  def apply[A: Shard]: Shard[A] = summon

  def default[A]: Shard[A] = new:
    val key: A ⇒ ShardKey = _ ⇒ ShardKey.Default
```

[20] https://neutron.profunktor.dev/
[21] https://pulsar.apache.org/docs/en/concepts-messaging/#key_shared

```
given Shard[String] with
  val key: String ⇒ ShardKey =
    str ⇒ ShardKey.Of(str.getBytes(UTF_8))
```

The default `Shard[String]` instance produces a sharding key by serializing the given string into UTF-8-encoded bytes.

Opinionated information

Neutron does not provide such a typeclass because it is not a generic abstraction. Though, it makes sense in our application.

Furthermore, published messages need a partition key for compacted topics (see Topic compaction in Chapter 3). For this case, we can introduce a similar typeclass named `Compaction`.

```
import dev.profunktor.pulsar.MessageKey

trait Compaction[A]:
  def key: A ⇒ MessageKey
```

Like `Shard`, we also have a default instance, but no instance for `String`.

```
object Compaction:
  def apply[A: Compaction]: Compaction[A] = summon

  def default[A]: Compaction[A] = new:
    val key: A ⇒ MessageKey = _ ⇒ MessageKey.Empty

  def by(s: String): MessageKey = MessageKey.Of(s)
```

Pulsar terminology makes this a bit confusing. On the one hand, the message `orderingKey` is used for routing in `KeyShared` subscriptions. On the other hand, the message `key` is mainly used for topic compaction—even though it is also used for routing when the `orderingKey` is absent.

Notes

A `ShardKey` manages the `orderingKey` (used for sharding), whereas a `MessageKey` corresponds to the message `key` mainly used for topic compaction (but also for sharding when the former is absent).

We can now proceed to write a constructor that instantiates a Pulsar producer.

```
import dev.profunktor.pulsar.{ Producer as PulsarProducer, * }
```

```
def pulsar[F[_]: Async: Logger: Parallel, A: Encoder](
    client: Pulsar.T,
    topic: Topic.Single,
    settings: Option[PulsarProducer.Settings[F, A]] = None
): Resource[F, Producer[F, A]] =
  val _settings =
    settings
      .getOrElse(PulsarProducer.Settings[F, A]())
      .withLogger(Logger.pulsar[F, A]("out"))

  val encoder: A ⇒ Array[Byte] =
    _.asJson.noSpaces.getBytes(UTF_8)

  PulsarProducer
    .make[F, A](client, topic, encoder, _settings)
    .map { p ⇒
      new:
        def send(a: A): F[Unit] = p.send_(a)
        def send(
          a: A,
          properties: Map[String, String]
        ): F[Unit] = p.send_(a, properties)
        def send(a: A, tx: Txn): F[Unit] = p.send_(a, tx.get)
    }
```

Notice that the `Logger` constraint is a custom effect that you can find in the trading project, but we will skip its implementation for brevity.

We also have an `Encoder` constraint on `A`, so we can encode our messages as JSON, as the `encoder` function demonstrates. Furthermore, we have added one extra `send` method that takes a `Txn` argument to support transactions, as we will learn shortly.

The constructor takes a Pulsar client, a topic, and an optional producer settings as arguments. By default, we add a JSON-structured logger, which leverages literal types and union types.

```
object Logger:
  def pulsar[F[_]: Logger, A: Encoder](
      flow: "in" | "out"
  ): A ⇒ Topic.URL ⇒ F[Unit] =
    p ⇒ t ⇒ Logger[F].info {
      Json.obj(
        "flow"    → flow.asJson,
        "payload" → p.asJson,
        "topic"   → t.value.asJson
```

```
    )
    .noSpaces
}
```

We learned that Apache Pulsar bases its deduplication mechanism on Sequence ID (see Deduplication), set per message. By default, Pulsar will set it to N+1, where N is the last known sequence ID. Neutron allows us to customize how these are set via `SeqIdMaker`, which can be set via the producer's settings.

As we will see in Chapter 6, we will rely on Pulsar's default implementation, as this mechanism of deduplication is only useful for message redeliveries (retries).

5.4.2.2 Consumer

Pulsar consumers are inherently more complex than producers. Creating a consumer requires a Pulsar client, a topic, a subscription, a message decoder, a decoding error handler, and the consumer settings.

In the following implementation, we add a default dead-letter topic policy and the same JSON-structured logger we used for the producer, except `flow="in"` this time.

```
import dev.profunktor.pulsar.{ Consumer as PulsarConsumer, * }

def pulsar[F[_]: Async: Logger, A: Decoder: Encoder](
    client: Pulsar.T,
    topic: Topic,
    sub: Subscription,
    settings: Option[PulsarConsumer.Settings[F, A]] = None
): Resource[F, Consumer[F, A]] =
  val deadLetterPolicy =
    DeadLetterPolicy.builder
      .deadLetterTopic("dead-letter")
      .maxRedeliverCount(2)
      .build

  val _settings =
    settings
      .getOrElse(PulsarConsumer.Settings[F, A]())
      .withLogger(Logger.pulsar[F, A]("in"))
      .withDeadLetterPolicy(deadLetterPolicy)

  val decoder: Array[Byte] ⇒ F[A] =
    bs ⇒ Async[F].fromEither(jsonDecode[A](new String(bs, UTF_8)))

  val handler: Throwable ⇒ F[PulsarConsumer.OnFailure] =
    e ⇒ Logger[F].error(e.getMessage).as(PulsarConsumer.OnFailure.Nack)
```

The `handler` dictates what should be done when a message fails to be decoded (can happen when we don't handle schema evolution properly). Whenever we can afford losing messages (e.g. old messages are no longer relevant in the presence of new ones), we can use the `OnFailure.Ack` strategy.

For most other cases, it is best to use `OnFailure.Nack` or `OnFailure.Raise`. However, nacking on failure means the same message will be scheduled for redeliver multiple times and our service will still be unable to decode it.

To mitigate the issue, we set a `DeadLetterPolicy` so that these problematic messages get published to a dead-letter topic after a certain number of attempted redeliveries to allow the continued operation of our service.

Next, we proceed with the creation of the `Consumer` based on `PulsarConsumer`.

```
PulsarConsumer.make[F, A](
  client, topic, sub, decoder, handler, _settings
).map { c ⇒
  new:
    def receiveM: Stream[F, Msg[A]] = c.subscribe.map { m ⇒
      Msg(MsgId.from(m.id), m.properties, m.payload)
    }
    def receiveM(id: MsgId): Stream[F, Consumer.Msg[A]] =
      c.subscribe(MsgId.to(id)).map { m ⇒
        Msg(MsgId.from(m.id), m.properties, m.payload)
      }

    def receive: Stream[F, A] = c.autoSubscribe
    def lastMsgId: F[Option[MsgId]] = c.lastMessageId.map(_.map(MsgId.from))
    def ack(id: MsgId): F[Unit] = c.ack(MsgId.to(id))
    def ack(ids: Set[MsgId]): F[Unit] = c.ack(ids.map(MsgId.to))
    def ack(id: MsgId, tx: Txn): F[Unit] = c.ack(MsgId.to(id), tx.get)
    def nack(id: MsgId): F[Unit] = c.nack(MsgId.to(id))
}
```

We will learn more about the configuration of consumers and producers when we get to use them in our trading application in the next chapter.

5.4.2.3 Example

We can now proceed with creating producers and consumers using our abstraction. Here's a simplified sneak peek of one of the services we will be writing in the next chapter (see Processor).

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url, Pulsar.Settings().withTransactions)
    cmdTopic = AppTopic.TradingCommands.make(config.pulsar)
    evtTopic = AppTopic.TradingEvents.make(config.pulsar)
    producer <- Producer.pulsar[IO, TradeEvent](pulsar, evtTopic, evtSettings)
    consumer <- Consumer.pulsar[IO, TradeCommand](pulsar, cmdTopic, sub)
  yield (server, consumer, Engine.fsm(producer, Txn.make(pulsar), consumer))
```

Until we dive into details in Chapter 6, let's only analyze what we understand so far.

We first load a configuration containing the connection details, then acquire a Pulsar connection with transactional support, followed by creating the topics.

Lastly, we create a producer and a consumer, ending with the instantiation of the FSM, which also requires a `Resource[F, Txn]` for transactions.

`Txn` is a custom interface we use to abstract over Neutron's implementation.

```
import dev.profunktor.pulsar.transactions.{ PulsarTx, Tx }

trait Txn:
  def get: Tx
```

We can create a default one by passing a Pulsar client as argument.

```
object Txn:
  def make(pulsar: Pulsar.T): Resource[IO, Txn] =
    PulsarTx.make[IO](
      pulsar, 30.seconds, Logger[IO].debug
    ).map { tx ⇒
      new:
        def get: Tx = tx
    }
```

Most of our services will be shaped as such, one way or another.

5.4.3 Distributed via Apache Kafka

How about creating interpreters for Apache Kafka as well? Although we will not use it in our application, it will demonstrate that this is possible, with some caveats that will be disclosed in the final summary.

We will base our implementation on top of the excellent fs2-kafka[22] library.

5.4.3.1 Producer

Starting out with the `Producer` implementation.

[22] https://github.com/fd4s/fs2-kafka

```
import fs2.kafka.{ KafkaProducer, ProducerSettings }

def kafka[F[_]: Async, A](
    settings: ProducerSettings[F, String, A],
    topic: String
): Resource[F, Producer[F, A]] =
  KafkaProducer.resource(settings).map { p ⇒
    new:
      def send(a: A): F[Unit] =
        p.produceOne_(topic, "key", a).flatten.void
      def send(a: A, properties: Map[String, String]): F[Unit] = send(a)
  }
```

Notice how we use the underlying `produceOne_` method to implement our `send`. Understandably, this is where things start getting funky, as Kafka and Pulsar are two different beasts that need taming.

Still, this is perfectly fine with `Producer`, but contrasting is the story with `Consumer`.

5.4.3.2 Consumer

Next up is our `Consumer` implementation, where things get tricky. First off, we start by creating a `Ref` to keep track of the Kafka committable offsets that can be later committed via our `ack` method.

Then we create the underlying Kafka consumer, ensuring auto-commit is disabled. Once done, we subscribe to a single topic, followed by building our `Consumer` instance.

```
import fs2.kafka.{ ConsumerSettings, KafkaConsumer }
import org.apache.kafka.clients.consumer.OffsetAndMetadata
import org.apache.kafka.common.TopicPartition

def kafka[F[_]: Async, A](
    settings: ConsumerSettings[F, String, A],
    topic: String
): Resource[F, Consumer[F, A]] =
  Resource.eval(
    Ref.of[F, List[CommittableOffset[F]]](List.empty)
  ).flatMap { ref ⇒
    KafkaConsumer
      .resource[F, String, A](settings.withEnableAutoCommit(false))
      .evalTap(_.subscribeTo(topic))
      .map { c ⇒
        new:
```

```
        def receiveM: Stream[F, Msg[A]] =
          c.stream.evalMap { c ⇒
            ref
             .update(_ :+ c.offset)
             .as(Msg(MsgId.Dummy, Map.empty, c.record.value))
          }

        def receiveM(id: MsgId): Stream[F, Consumer.Msg[A]] = receiveM

        def receive: Stream[F, A] = c.stream.flatMap { s ⇒
          Stream
            .emit(s.offset)
            .through(commitBatchWithin[F](500, 10.seconds))
            .as(s.record.value)
        }

        def ack(ids: Set[MsgId]): F[Unit] =
          ref.get.flatMap(CommittableOffsetBatch.fromFoldable(_).commit)

        def lastMsgId: F[Option[MsgId]]      = none.pure[F]
        def ack(id: MsgId): F[Unit]          = ack(Set(id))
        def ack(id: MsgId, tx: Txn): F[Unit] = ack(Set(id))
        def nack(id: MsgId): F[Unit]         = Applicative[F].unit
    }
}
```

The underlying `c.stream` method returns a `Stream[F, CommitableConsumerRecord[F, String, A]]`, so here things are entirely different compared to Pulsar, as we can manipulate *records* and *offsets* directly.

Notice we don't implement `nack` either, as it is not relevant to Kafka, but we do implement `ack` via `CommittableOffsetBatch` with the offsets we store in memory.

We can get away with this simple interpreter for a demo, though it is not recommended for production use. Mainly because it is easy to misuse the `ack` and `nack` methods without knowing their implementation.

5.4.3.3 Example

Since we got this far, let's see how we can put all the pieces together with an example, starting by creating a topic and both the consumer and producer settings.

```
val topic = "trading-kafka"
```

```
val consumerSettings =
  ConsumerSettings[IO, String, TradeEvent]
    .withAutoOffsetReset(AutoOffsetReset.Earliest)
    .withBootstrapServers("localhost:9092")
    .withGroupId("group")

val producerSettings =
  ProducerSettings[IO, String, TradeEvent]
    .withBootstrapServers("localhost:9092")
```

We use the `TradeEvent` datatype, which will appear in our trading application. Because it is a custom datatype, we need to provide instances for `Serializer` and `Deserializer`.

```
import fs2.kafka.*
import io.circe.parser.decode as jsonDecode

given Deserializer[IO, TradeEvent] =
  Deserializer.lift[IO, TradeEvent] { bs ⇒
    IO.fromEither(jsonDecode[TradeEvent](new String(bs, UTF_8)))
  }

given Serializer[IO, TradeEvent] =
  Serializer.lift[IO, TradeEvent] { e ⇒
    IO.pure(e.asJson.noSpaces.getBytes(UTF_8))
  }
```

Now we are ready to create the consumer and producer as resources.

```
def resources =
  for
    c <- Consumer.kafka[IO, TradeEvent](consumerSettings, topic)
    p <- Producer.kafka[IO, TradeEvent](producerSettings, topic)
  yield c → p
```

At last, we can run our program with some random events.

```
val event: Option[TradeEvent] = ??? // random data

def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (consumer, producer) ⇒
      val p1 =
        consumer.receive
          .evalMap(e ⇒ IO.println(s">>> KAFKA: $e"))
```

```
    val p2 =
      Stream
        .awakeEvery[IO](1.second)
        .as(event)
        .evalMap(_.traverse_(producer.send))

    p1.concurrently(p2)
  }
  .interruptAfter(5.seconds)
  .compile
  .drain
```

Notes

Random events' implementation is skipped for brevity.

It should produce a similar output to the one shown below when executed.

```
>>> Initializing kafka demo <<<
>>> KAFKA: CommandExecuted(...)
>>> KAFKA: CommandExecuted(...)
>>> KAFKA: CommandExecuted(...)
>>> KAFKA: CommandExecuted(...)
[success] Total time: 8 s, completed Nov 18, 2021, 1:33:28 PM
```

5.5 Summary

We have learned that Kafka and Pulsar's semantics are absolutely different. Although creating a `Consumer` and `Producer` abstraction is a fun exercise—and can even pass the test for proof of concepts—you need to understand the pros and cons of the message broker of choice, pick your tool and use the library directly (e.g. either `fs2-kafka` or `neutron`).

In the application we will write in the next chapter, we will still keep the interfaces that abstract over the creation of consumers and producers. Nonetheless, you will see a lot of Pulsar-specific code—such as subscriptions and consumer-producer settings—gets mixed-in in our services. The same would happen if we choose Kafka instead.

That said, creating a common constructor for all your services with default settings (e.g. a JSON-structured logger) and semantics is always good to have.

So besides learning about the clients for these two popular message brokers, we have seen finite-state machines, data pipelines, and other streaming goodies that will be immensely useful when we get hands down on the application code.

Part III: System

After learning about event-driven architecture and distributed systems—and having had fun with Scala 3 code and effectful streams talking to message brokers—we should now feel ready to take on more significant challenges.

In this last extensive part, we will focus on designing and developing a distributed trading system to make all the pieces come together in complete harmony.

Ready? Fasten your seatbelts.

6 Trading system (core services)

It is now the time to put all that theory into practice by designing and developing a distributed system, analyzing every piece of it from various angles.

We will explore the world of trading. More specifically, the stock exchange market[1]—including foreign exchange[2]—and the different aspects of such a system.

[1] https://en.wikipedia.org/wiki/Stock_exchange
[2] https://en.wikipedia.org/wiki/Foreign_exchange_market

6.1 Business requirements

A fictitious SLA indicates the distributed system we need to develop must be highly available, can afford eventual consistency, and should cope with at least a million users a day. There are two sides to it: *trading* and *forecasting*.

Trading

The trading feature should allow users to buy and sell foreign currency via *bids and asks* (aka *bids and offers*). Trading can also be turned on and off, like a light switch.

A trade or transaction occurs when a buyer in the market is willing to pay the best offer available or to sell at the highest bid. The bid price represents the maximum price that a buyer is willing to pay for a currency. The ask price represents the minimum price that a seller is ready to take for that same currency.

Users should also be able to subscribe to alerts from currency symbols such as EURPLN or GBPCHF via Web Sockets. Anytime a bid or ask is placed, the system should assess whether a new alert should be emitted or not.

There already exists a web application with the following user interface.

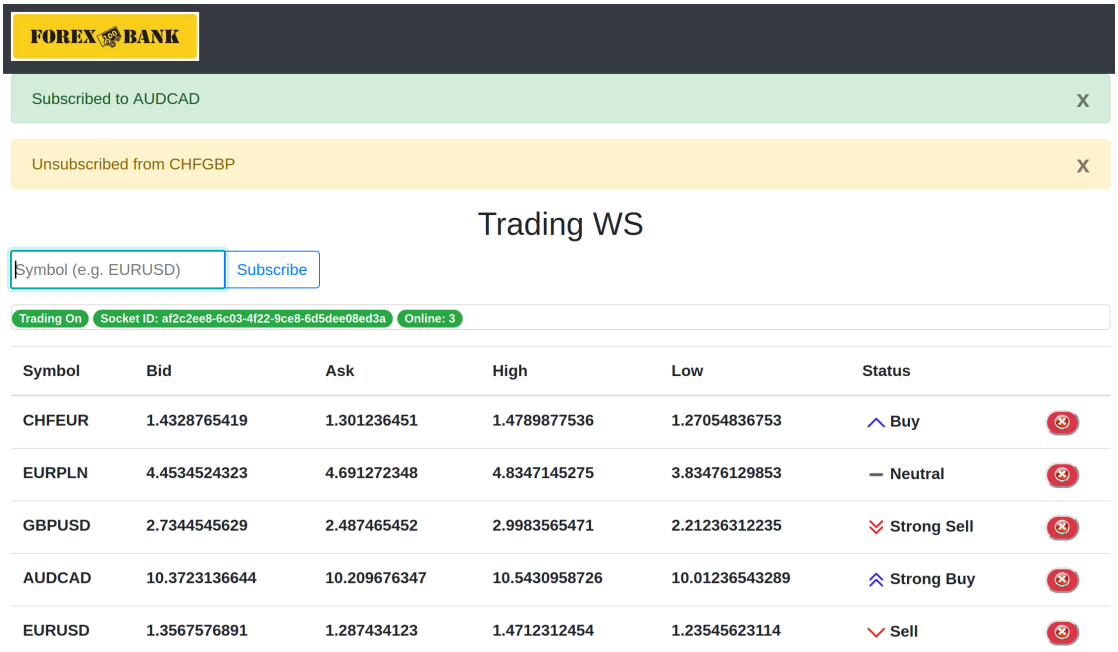


Figure 6.1.1: web app

The alert assessment part has not been specified yet, so for now, we can place any dummy logic until further notice.

Forecasting

The forecasting functionality allows authors to self-register to later publish market forecasting articles such as "GBPUSD short trading opportunity" or so. Every forecast can then be rated up or down—as in like or dislike—anonymously.

Unlike trading, there is neither a web application nor further requirements for this feature, but that's something that may come later on.

Therefore, we will only have a single `forecasts` service instance for now, as we will learn in Chapter 7 (see Forecasts).

6.1.1 Overview

We will have the following services representing the core trading functionality.

- `processor`: reads trading commands (e.g. placing bids and asks) and switch commands (trading switch, on/off), updates the trading status when appropriate, and emits trading and switch events.
- `alerts`: reads trading and switch events and emits alerts according to the configured settings (dummy logic for now).
- `ws`: reads alerts and broadcasts them via Web Sockets to all users subscribed to the symbol associated with the alert.

Commands are being published to a Pulsar topic by an external development team, so our focus will be to integrate our system with it (pluggable system).

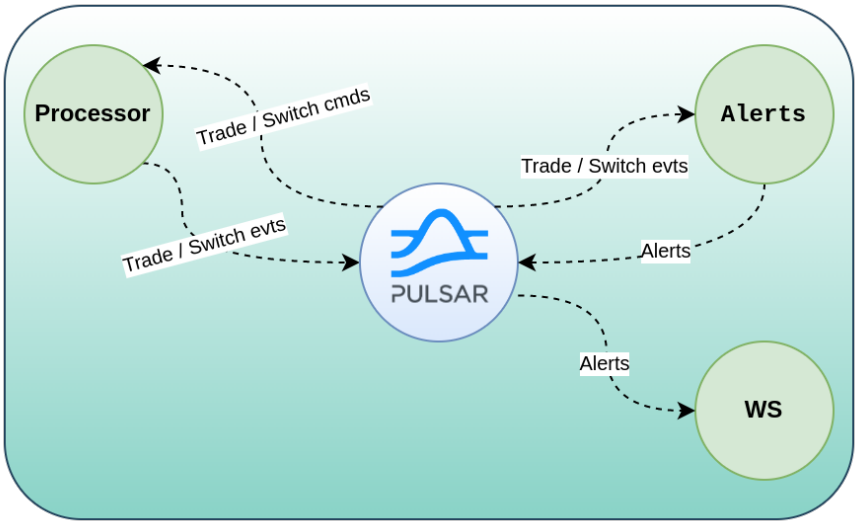


Figure 6.1.2: core services

In this chapter, we will dive into the specifics of each service, and we will learn more about the overall system design decisions.

Let's start by analyzing the domain and purpose of each core service.

6.1.2 Domain modeling

Our application's domain model lives under the **domain** module shared across all services. Among other datatypes, we also define most newtypes using the encoding we learned in Chapter 4 (see Newtypes). E.g.

```
package trading.domain

type Quantity = Quantity.Type
object Quantity extends NumNewtype[Int]

type Timestamp = Timestamp.Type
object Timestamp extends Newtype[Instant]
```

All these are defined as top-level datatypes within the **trading.domain** package. We also have a few enumerations, for instance.

```
enum TradeAction derives Codec.AsObject, Eq, Show:
  case Ask, Bid
```

Typeclass derivation works with a few nuances at the moment of writing, but most of the issues should be resolved by the time this book is completed.

Other datatypes such as commands, events, and state will be disclosed as we *traverse* (functional pun intended) this chapter.

6.1.3 Shared modules

All services depend on the **core** and **lib** modules, in addition to the **domain** module.

Under the former, we define business logic and interpreters that need to be accessed by more than one service. E.g. the trading FSM.

```
object TradeEngine:
  val fsm: FSM[
    Id,
    TradeState,
    TradeCommand | SwitchCommand,
    (EventId, Timestamp) ⇒ SwitchEvent | TradeEvent
  ] = FSM.id { ... }
```

As well as a static definition of Pulsar topics.

```
sealed abstract class AppTopic:
  def name: String
  def make(cfg: Config): Topic.Single

object AppTopic:
  case object Alerts extends AppTopic:
    val name: String = "trading-alerts"
    def make(cfg: Config): Topic.Single = mkPersistent(cfg, name)
```

On the other hand, under `lib`, we find capability traits, among other things.

```
trait GenUUID[F[_]]:
  def make[A: IsUUID]: F[A]

trait Time[F[_]]:
  def timestamp: F[Timestamp]
```

You can get a proper view of all the shared components by looking directly into the trading repository[3] and following along with this book.

[3] https://github.com/gvolpe/trading

6.2 Processor

Now is time to get down to business and start exploring the responsibilities of the `processor` service. As we can observe in the application diagram, it consumes `TradeCommand`s and `SwitchCommand`s, and it produces `TradeEvent`s and `SwitchEvent`s, respectively.

This follows the CQRS design pattern (see CQRS/ES), as far as the commands go. We will not have queries, though, as these are not needed here.

Firstly, let's study the definition of these datatypes.

6.2.1 Commands

A `TradeCommand` represents either an instruction to create, update, or delete a bid or ask. In Scala 3, we encode most commands and events as `enum`s (but they could also be `sealed trait`s) with a few abstract methods.

```
enum TradeCommand derives Codec.AsObject, Eq, Show:
  def id: CommandId
  def cid: CorrelationId
  def symbol: Symbol
  def createdAt: Timestamp
```

These four fields represent essential information in our system.

- `id`: a unique command identifier.
- `cid`: a correlation identifier used to trace a system transaction.
- `symbol`: a unique symbol (e.g. EURUSD) for the ask or bid.
- `createdAt`: a timestamp indicating when the command was created at.

Next, we have multiple datatypes that are part of this enumeration.

```
case Create(
    id: CommandId,
    cid: CorrelationId,
    symbol: Symbol,
    tradeAction: TradeAction,
    price: Price,
    quantity: Quantity,
    source: Source,
    createdAt: Timestamp
)
```

```
case Update(
    id: CommandId,
    ...
)

case Delete(
    id: CommandId,
    ...
)
```

Most fields have been omitted for brevity, but you get the idea.

A `SwitchCommand` conveys instructions to turn trading on or off. Its encoding is very similar to the previous one.

```
enum SwitchCommand derives Codec.AsObject, Eq, Show:
  def id: CommandId
  def cid: CorrelationId
  def createdAt: Timestamp

  case Start(
      id: CommandId,
      cid: CorrelationId,
      createdAt: Timestamp
  )

  case Stop(
      id: CommandId,
      cid: CorrelationId,
      createdAt: Timestamp
  )
```

We only have three essential fields in this case, as a switch applies to trading in general, not just to a specific symbol.

6.2.2 Events

A `TradeEvent` represents something that has occurred as a consequence of reacting to a command.

```
enum TradeEvent derives Codec.AsObject:
  def id: EventId
  def cid: CorrelationId
  def command: TradeCommand
  def createdAt: Timestamp
```

The `CorrelationId` always corresponds to that of the command that triggered the event, allowing us to track such system transactions. We will learn more about the importance of this simple field when we dive into observability (see Chapter 8).

Here are the two events defined as part of the enumeration.

```
case CommandExecuted(
    id: EventId,
    cid: CorrelationId,
    command: TradeCommand,
    createdAt: Timestamp
)

case CommandRejected(
    id: EventId,
    ...
)
```

Once again, skipping some implementation details for briefness.

A `SwitchEvent` follows through with three different outcomes.

```
enum SwitchEvent derives Codec.AsObject, Show:
  def id: EventId
  def cid: CorrelationId
  def createdAt: Timestamp

  case Started(...)
  case Stopped(...)
  case Ignored(...)
```

We will learn how commands relate to the different events in the following section.

6.2.3 Command-event relationship

Understanding the relationship between commands and events is key to getting a complete picture of the different state transitions—defined in the main `TradeEngine` finite-state machine.

STATUS	COMMAND	EVENT
On	Create	CommandExecuted
On	Update	CommandExecuted
On	Delete	CommandExecuted
Off	C-U-D	CommandRejected("trading off")
Off	Start	Started
On	Stop	Stopped
On	Start	Ignored
Off	Stop	Ignored

The first two columns represent the inputs of the state machine, whereas the last column represents the outputs.

When trading is on, and we receive any command such as `Create`, `Update`, or `Delete`, a `CommandExecuted` event is produced, as shown in the first three rows.

Conversely, a `CommandRejected` event is produced either when any of these three commands (C-U-D) are received and trading is off.

The trading status can be switched on and off via the `Start` and `Stop` commands, respectively, as we can observe in the transitions displayed on the relationships table.

Also, notice how these commands can sometimes be ignored when the expected status is already in place.

We had a peek at the trading FSM in Chapter 5, which has the following signature.

```
val fsm = FSM.id[
  TradeState,
  TradeCommand | SwitchCommand,
  (EventId, Timestamp) ⇒ TradeEvent | SwitchEvent
]
```

Outputs are represented as a function `(EventId, Timestamp) ⇒ TradeEvent | SwitchEvent` to make it pure. Yes, we could make it `TradeEvent | SwitchEvent` instead, but that would require some effect capabilities to create those two necessary inputs.

This only showcases a different design; both approaches are valid. We will learn more about the state machines in use by the processor service shortly.

We also have a second state machine that operates on events instead of commands.

```
val eventsFsm = FSM.id[TradeState, TradeEvent | SwitchEvent, Unit]
```

However, nothing is produced (`Unit`) from this state machine; we are only interested in state transitions.

STATUS	EVENT	NEW STATE
On	CommandExecuted	Yes
Off	CommandExecuted	No
Off	Started	Yes
On	Stopped	Yes
_	CommandRejected	No
_	SwitchEvent	No

Now that we have a basic understanding of how events relate to commands and the internal state, we can look into the core implementation of the `processor` service.

6.2.4 Entry point

All services have a similar entry point—an object named `Main`—extending `IOApp.Simple` and defining a sequence of resources like consumers, producers, etc.

Let's analyze `processor`'s `Main` in detail to get acquainted with the design, as many moving pieces are common with other services.

First, we have a Pulsar subscription for `TradeCommand` using the `KeyShared` type (see Subscriptions), which allows us to have many instances of `processor` running concurrently (more on this in the Scalability section further down).

```
def cmdSub(id: AppId) =
  Subscription.Builder
    .withName(id.name)
    .withType(Subscription.Type.KeyShared)
    .build
```

The `AppId` is a simple datatype that uniquely identifies every service instance.

```
case class AppId(name: String, id: UUID)
```

Next, we have a Pulsar subscription for `SwitchCommand`, which is `Exclusive` to every instance (hence, the unique identifier included in `id.show`).

```
def swtSub(id: AppId) =
  Subscription.Builder
    .withName(id.show)
    .withType(Subscription.Type.Exclusive)
    .build
```

Another essential property is that the corresponding `SwitchCommand` topic is set up for compaction, so the consumer should be configured accordingly to read from it.

```
val compact =
  PulsarConsumer.Settings[IO, SwitchCommand]().withReadCompacted.some
```

Next, we have the producer settings, starting with `TradeEvent`. It includes deduplication support, and is also sharded by symbol.

```
val evtSettings =
  PulsarProducer
    .Settings[IO, TradeEvent]()
    .withDeduplication
    .withShardKey(Shard[TradeEvent].key)
    .some
```

Followed by the `SwitchEvent` producer settings.

```
val swtSettings =
  PulsarProducer
    .Settings[IO, TradeEvent.Switch]()
    .withDeduplication
    .withMessageKey(Compaction[TradeEvent.Switch].key)
    .some
```

Also with deduplication support, but partitioned instead of sharded (see Scalability for details). This is mainly used for topic compaction, as seen in Chapter 3.

At last, we have the initialization of all the resources that have a lifecycle.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url, Pulsar.Settings().withTransactions)
    _      <- Resource.eval(Logger[IO].info(s"Initializing: ${config.appId.show}"))
    server   = Ember.default[IO](config.httpPort)
    cmdTopic = AppTopic.TradingCommands.make(config.pulsar)
    evtTopic = AppTopic.TradingEvents.make(config.pulsar)
    swcTopic = AppTopic.SwitchCommands.make(config.pulsar)
    sweTopic = AppTopic.SwitchEvents.make(config.pulsar)
```

```
    trProducer <- Producer.pulsar[IO, TradeEvent](pulsar, evtTopic, evtSettings)
    swProducer <- Producer.pulsar[IO, SwitchEvent](pulsar, sweTopic, swtSettings)
    sub1 = cmdSub(config.appId)
    sub2 = swtSub(config.appId)
    trConsumer <- Consumer.pulsar[IO, TradeCommand](pulsar, cmdTopic, sub1)
    swConsumer <- Consumer.pulsar[IO, SwitchCommand](pulsar, swcTopic, sub2, compact)
    engine = Engine.fsm(trProducer, swProducer, Txn.make(pulsar), trConsumer, swConsumer)
  yield (server, trConsumer, swConsumer, engine)
```

The first line loads the configuration, which uses the Ciris[4] library, followed by creating a connection to Apache Pulsar with transactional support.

The `Ember.default` method lives under the `core` module, which creates an HTTP server with support for a simple health-check endpoint and Prometheus metrics.

Next, producers for both `TradingEvent` and `SwitchEvent` are created. This is followed by the creation of the corresponding command's consumers.

6.2.5 FSM

Finally, we create the FSM that dictates the main logic of this service. Most of our business logic will be written as finite-state machines, so this is a pattern you will see repeatedly in other services.

The funny thing about this state machine is that it invokes the trading FSM within its logic, which is also used in other services, as we will learn soon.

Let's recap on the shape of FSM.

```
case class FSM[F[_], S, I, O](
  run: (S, I) ⇒ F[(S, O)]
)
```

It has an internal state `S`, and it processes state transitions by receiving inputs `I` and producing outputs `O`.

In our case, `S` is `TradeState`, and `I` is either a message of type `TradeCommand` or `SwitchCommand`. We do not produce anything as output (`O` = `Unit`), which usually indicates an effect is performed within `F`.

The `TradeState` datatype is defined as follows.

```
final case class TradeState(
    status: TradingStatus,
    prices: Map[Symbol, Prices]
) derives Eq, Show
```

[4] https://cir.is

Where `TradingStatus` is a simple enumeration, isomorphic to a boolean.

```
enum TradingStatus derives Eq, Show:
  case On, Off
```

And `Prices` is another case class with four properties.

```
final case class Prices(
    ask: Prices.Ask,
    bid: Prices.Bid,
    high: Price,
    low: Price
) derives Eq, Show

object Prices:
  type Ask = Map[AskPrice, Quantity]
  type Bid = Map[BidPrice, Quantity]
```

As shown below, we need to write tedious code to modify only the `ask` price.

```
val st: TradeState = ???

val pold = st.prices.get(Symbol.EURUSD).getOrElse(Prices.empty)
val pnew = pold.copy(ask = pold.ask.updated(Price(3.41), Quantity(2)))
val nst  = st.copy(prices = st.prices.updated(Symbol.EURUSD, pnew))
```

For this reason, we have a few Monocle[5] optics to help us operate on the deeply nested structures of the state—all law-checked under `TradeStateSuite`.

```
import monocle.{ Focus, Optional }
import monocle.function.{ At, Index }

object TradeState:
  def empty: TradeState = TradeState(TradingStatus.On, Map.empty)

  val _Status = Focus[TradeState](_.status)
  val _Prices = Focus[TradeState](_.prices)

  object __Prices:
    def at(s: Symbol): Optional[TradeState, Option[Prices]] =
      _Prices.andThen(At.atMap[Symbol, Prices].at(s))

    def index(s: Symbol): Optional[TradeState, Prices] =
      _Prices.andThen(Index.mapIndex[Symbol, Prices].index(s))
```

[5] https://github.com/optics-dev/Monocle

For those unfamiliar with optics, these are well-defined composable functional abstractions for data manipulation that obey a set of algebraic laws.

For instance, lenses operate on product types (e.g. case classes), prisms on sum types (e.g. ADTs), optionals inside product types with elements that may not exist, etc.

Monocle abstracts away most of this complexity with its Focus[6] datatype, providing a great starting point to using optics without even knowing what a lens is.

The `At` and `Index` optics appearing in the code above let us operate on *indexable data structures* such as `List` and `Map`. These are typeclasses roughly defined as follows.

```
class Index[S, -I, A]:
  def index(i: I): Optional[S, A]

class At[S, -I, A]:
  def at(i: I): Lens[S, A]
```

They are very similar, except one returns an `Optional` and the other a `Lens`. The PFPS book covers some optics in detail and recommends further material for those wanting to dig deeper into the fantastic world of optics.

There are more optics used in the different state machines. Additionally, we have the following helper methods defined directly on `TradeState` to reduce the boilerplate.

```
def modify(symbol: Symbol)(
  action: TradeAction, price: Price, quantity: Quantity
): TradeState = ???

def remove(symbol: Symbol)(
  action: TradeAction, price: Price
): TradeState = ???
```

Apologies for the optics deviation. Let's have a look at the FSM constructor.

```
object Engine:
  type In = Either[Consumer.Msg[TradeCommand], Consumer.Msg[SwitchCommand]]

  def fsm[F[_]: GenUUID: Logger: MonadCancelThrow: Time](
      producer: Producer[F, TradeEvent],
      switcher: Producer[F, SwitchEvent],
      pulsarTx: Resource[F, Txn],
      tradeAcker: Acker[F, TradeCommand],
      switchAcker: Acker[F, SwitchCommand]
  ): FSM[F, TradeState, In, Unit] =
    FSM { ??? }
```

[6] https://www.optics.dev/Monocle/docs/focus

It takes two producers (one for each event type), a Pulsar transactional resource, and two command ackers.

Next, we have the processing of both commands running within a Pulsar transaction.

```
FSM {
  case (st, Right(Consumer.Msg(msgId, _, cmd))) ⇒
    sendEvent(switchAcker.ack(msgId, _), st, cmd)
  case (st, Left(Consumer.Msg(msgId, _, cmd))) ⇒
    sendEvent(tradeAcker.ack(msgId, _), st, cmd)
}
```

The `sendEvent` helper method is defined as follows.

```
def sendEvent(
    ack: Txn ⇒ F[Unit],
    st: TradeState,
    cmd: TradeCommand | SwitchCommand
): F[(TradeState, Unit)] =
  pulsarTx.use { tx ⇒
    val (nst, evt) = TradeEngine.fsm.run(st, cmd)
    (GenUUID[F].make[EventId], Time[F].timestamp).mapN(evt).flatMap {
      case e: TradeEvent  ⇒ producer.send(e, tx)
      case e: SwitchEvent ⇒ switcher.send(e, tx)
    } *> ack(tx).tupleLeft(nst)
  }.handleErrorWith { e ⇒
    Logger[F].warn(s"Transaction failed: ${e.getMessage}").tupleLeft(st)
  }
```

In this case, a transaction guarantees that the acknowledgement of the consumption of the command and the publishing of the corresponding event are part of a single atomic operation.

We will analyze this solution as well as the alternatives shortly (see Deep analysis).

6.2.5.1 Testing

One of the great benefits of finite-state machines is that they are easy to test in isolation, as they take inputs and produce outputs, like a plain old function.

In the following trading FSM test, we can observe the relationship between commands and events and the internal state transitions (see Command-event relationship).

```
test("Trade engine commands fsm") {
  val cmd1 = TradeCommand.Create(id, cid, s, TradeAction.Ask, p1, q1, "test", ts)
  val (st1, ev1) = fsm.run(TradeState.empty, cmd1)

  val xst1 = TradeState(
               status = On,
               prices = Map(s → Prices(ask = Map(p1 → q1), bid = Map.empty, p1, p1))
             )
  val xev1 = TradeEvent.CommandExecuted(eid, cid, cmd1, ts)

  val cmd5 = SwitchCommand.Stop(id, cid, ts)
  val (st5, ev5) = fsm.run(st4, cmd5)
  val xst5 = TradeState(Off, xst4.prices)
  val xev5 = SwitchEvent.Stopped(eid, cid, ts)

  val cmd6 = TradeCommand.Create(id, cid, s, TradeAction.Bid, p1, q1, "test", ts)
  val (st6, ev6) = fsm.run(st5, cmd6)
  val xst6 = xst5
  val xev6 = TradeEvent.CommandRejected(eid, cid, cmd6, Reason("Trading is Off"), ts)

  // expectations 2 to 5 skipped for brevity
  NonEmptyList
    .of(
      expect.same(st1, xst1),
      expect.same(st6, xst6),
      expect.same(ev1(eid, ts), xev1),
      expect.same(ev6(eid, ts), xev6)
    )
    .reduce
}
```

We combine all Weaver[7]'s expectations (aka assertions) in a single `NonEmptyList` and then call `reduce` on it, as they do form a `Monoid`.

All the hidden values are constants declared at the top level of the test object. Please refer to the source code for the complete version.

[7] https://github.com/disneystreaming/weaver-test

6.2.6 Deep analysis

Let's now analyze the decisions made in the main FSM and the potential issues we may encounter.

When we get either a `TradeCommand` or a `SwitchCommand`, we run the trading FSM and get back a new `TradeState` and a function `(EventId, Timestamp) ⇒ TradeEvent | SwitchEvent`.

Next, we perform a series of Pulsar operations within a transactional block:

1. Publish either a trading or switch event.
2. Acknowledge the command has been successfully received.

```
pulsarTx.use { tx ⇒
  val (nst, evt) = TradeEngine.fsm.run(st, cmd)
  (GenUUID[F].make[EventId], Time[F].timestamp).mapN(evt).flatMap {
    case e: TradeEvent  ⇒ producer.send(e, tx)
    case e: SwitchEvent ⇒ switcher.send(e, tx)
  } *> ack(tx).tupleLeft(nst)
}.handleErrorWith { e ⇒
  Logger[F].warn(s"Transaction failed: ${e.getMessage}").tupleLeft(st)
}
```

Either all these operations are performed together as one atomic action, or the transaction is aborted, in which case the internal state is not updated.

Without a transaction, the command would be re-processed, and the same event would be produced whenever the acknowledgement fails. To successfully deduplicate this event, we would need to keep track of previously processed commands to assign the same Sequence ID or deduplicate at the consumer side (see Deduplication), increasing complexity. Therefore, we choose transactions to make our lives easier, even though we may take a performance hit.

It is not magic, though. Atomicity and idempotency are guaranteed only for Pulsar operations. If we perform other effects mid-transaction, that's on us.

As all the external effects in our case represent Pulsar interactions, we have idempotency guaranteed. However, what would happen if we had a database operation in between?

```
for
  _ <- producer.send(evt, tx)
  _ <- hypotheticalDB.save(nst)
  _ <- ack(msgId, tx)
yield ()
```

Let's inspect what would happen in case of failure in every line. What happens if the publishing of the event fails?

```
producer.send(evt, tx)
```

Nothing. The service will crash (if the error is not handled), and the command will be picked up on restart (or by other instances).

What happens if the persistence of the current `TradeState` fails in our hypothetical DB?

```
hypotheticalDB.save(nst)
```

The Pulsar transaction will fail, as the acknowledgement wouldn't be reached, but we need to understand what kind of failure we are against. Can we safely retry the operation? What if the state was persisted, but the connection dropped before we got a response? It would not be safe to retry if the operation is not idempotent.

In such cases, our only option would be to guarantee the database write is idempotent, which could be achieved by assigning the data a primary key. An alternative would be to use CRDTs, briefly mentioned in Chapter 2 (see Consistency).

It may tempt to combine database transactions with Pulsar transactions—any error raised within the transactional block would trigger the abortion of both.

```
(pulsarTx, hypotheticalDB.tx).tupled.use {
  case (tx, _) ⇒
    for
      _ <- producer.send(evt, tx)
      _ <- hypotheticalDB.save(nst)
      _ <- ack(msgId, tx)
    yield ()
}.handleErrorWith { e ⇒
  Logger[F].error(e) *> nack(msgId)
}
```

However, we cannot guarantee that the "release" of both resources would succeed—i.e. committing the database and Pulsar transactions could fail with a network issue.

Instead, patterns like the transactional outbox (see Outbox pattern in Chapter 2) solve this issue elegantly.

This kind of analysis is crucial to the understanding of the system. *What if...?* questions can help us achieve that goal, putting our design under scrutiny.

6.2.7 Scalability

The `KeyShared` subscription type allows us to have multiple instances of `processor` running concurrently. So, how does this look in practice? How do we handle state spread across the many instances?

Figure 6.2.1 shows three different instances of `processor` running in this mode.

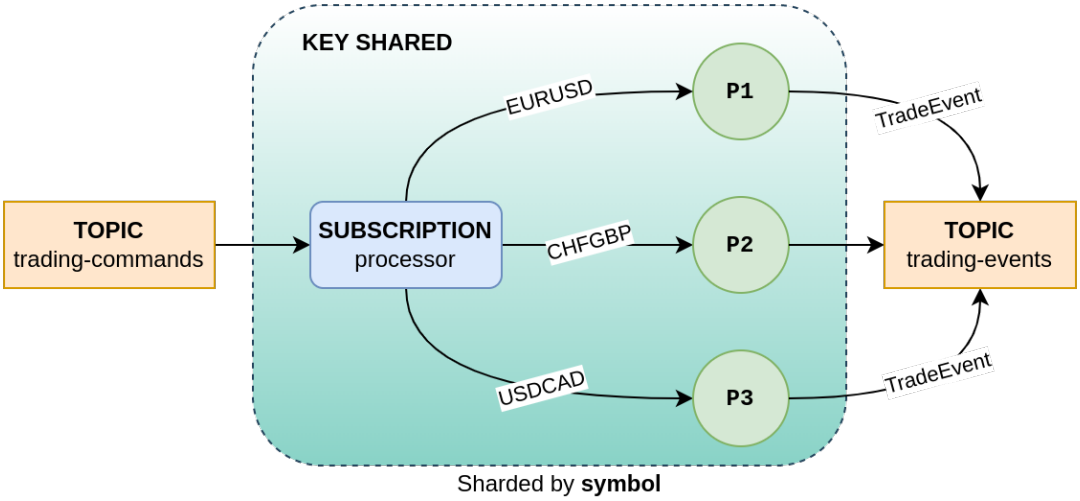


Figure 6.2.1: processor scalability (trading)

The sharding logic is configured via the `Shard` instance (see Pulsar Producer).

```
given Shard[TradeCommand] with
  val key: TradeCommand ⇒ ShardKey =
    cmd ⇒ Shard.by(cmd.symbol.show)
```

This guarantees that only one instance at a time would be processing the same symbol, so we can safely process aggregations independently.

Still, the `processor` service does not care about fluctuation of prices, only about the generation of events, which are dependent on the current trading status.

For this reason, the service can always start from an empty `TradeState`. Every instance has an exclusive subscription to the `switch-commands` compacted topic, which guarantees a fast startup as it only needs to read the latest value (i.e. on/off).

Figure 6.2.2 shows the handling of switch commands.

Reading from compacted topics requires either an `Exclusive` or a `Failover` subscription. We make this possible by assigning a unique identifier to each instance (`AppId`).

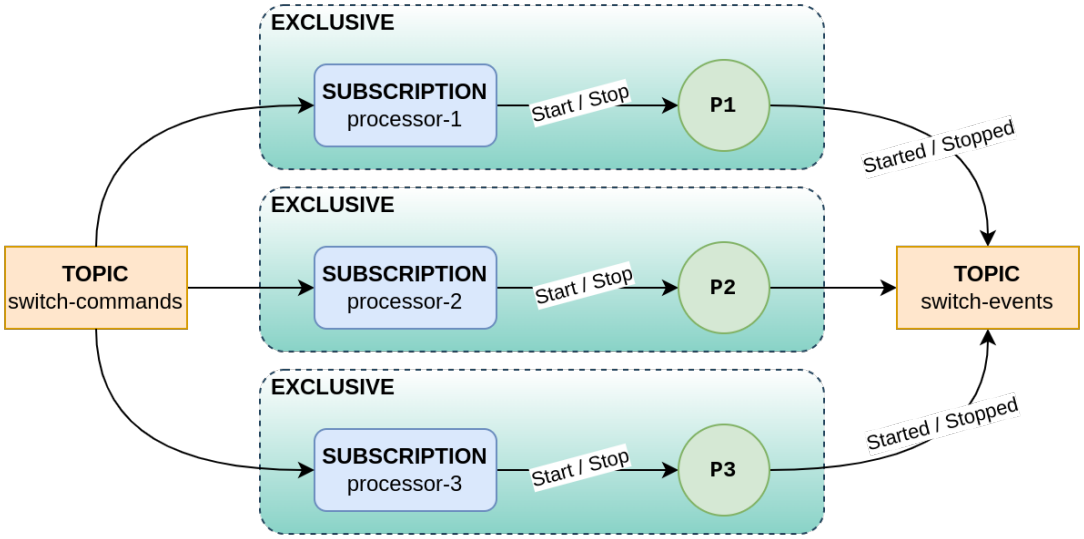


Figure 6.2.2: processor scalability (switch)

Furthermore, both `SwitchCommand`s and `SwitchEvent`s need to be partitioned by key, dictated by the `Compaction` instances.

```
given Compaction[SwitchCommand] with
  val key: SwitchCommand ⇒ MessageKey = {
    case _: SwitchCommand.Start ⇒ by("start")
    case _: SwitchCommand.Stop  ⇒ by("stop")
  }

given Compaction[SwitchEvent] with
  val key: SwitchEvent ⇒ MessageKey = {
    case _: SwitchEvent.Started ⇒ by("started")
    case _: SwitchEvent.Stopped ⇒ by("stopped")
    case _: SwitchEvent.Ignored ⇒ by("ignored")
  }
```

When a topic gets compacted, we would have maximum two commands and three events, respectively.

6.2.7.1 Rebalance

What happens when a new instance is added? Or when an instance crashes? In both cases, Pulsar will automatically rebalance the active consumers.

Figure 6.2.3 shows the `P2` instance becoming unavailable, triggering a rebalance.

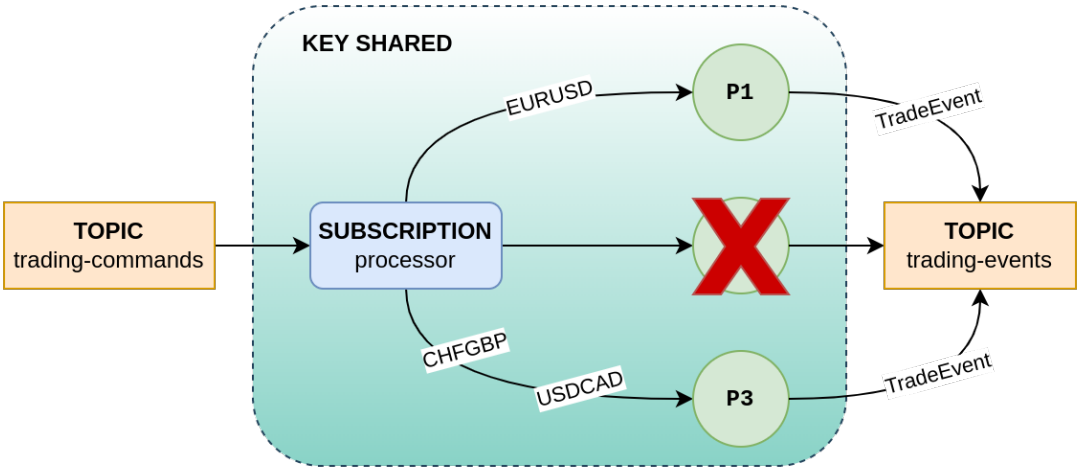


Figure 6.2.3: processor scalability (rebalance)

This could be quite challenging in stateful services (see Alerts Scalability), but it makes our lives easier when it is not an issue.

6.2.8 Run

At last, this is how we run the `processor` application.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (server, trConsumer, swConsumer, fsm) ⇒
      Stream.eval(server.useForever).concurrently {
        trConsumer.receiveM
          .either(swConsumer.receiveM)
          .evalMapAccumulate(TradeState.empty)(fsm.run)
      }
    }
    .compile
    .drain
```

All services have an HTTP endpoint for health and metrics, so we run it via `useForever` while `concurrently`, we consume commands and run them through the FSM.

By default, `receiveM` starts consuming from the last unacknowledged message.

6.3 Alerts

The next essential service is `alerts`. It consumes both `TradeEvent`s and `SwitchEvent`s produced by `processor`, and it emits `Alerts`, as we can observe in fig. 6.1.2.

Moreover, it consumes `PriceUpdate`s, which we will see in the next section (see Datatypes).

Unlike the previous service, this one is a kind of *listen-and-react* type, so it does not follow the CQRS design pattern—i.e. it reacts to events by producing alerts.

However, we could also see the `processor` service in the same way as it reacts to commands by producing events. The only meaningful distinction is that commands convey instructions, whereas events only inform of something that has occurred.

So these are essentially different types of services, albeit sharing a lot in common.

We are already acquainted with `TradeEvent`, so let's now learn about `Alert` and `PriceUpdate`.

6.3.1 Datatypes

Alerts can be seen as other kinds of events. In fact, it could be renamed to something like `AlertEvent`, `NotificationEvent`, or so, without changing its meaning.

As we did with other crucial datatypes, we have some mandatory fields.

```
enum Alert derives Codec.AsObject, Show:
  def id: AlertId
  def cid: CorrelationId
  def createdAt: Timestamp
```

Next, we have two different cases.

```
case TradeAlert(
    alertType: AlertType,
    symbol: Symbol,
    askPrice: AskPrice,
    bidPrice: BidPrice,
    high: HighPrice,
    low: LowPrice
)

case TradeUpdate(
    status: TradingStatus,
)
```

A `TradeAlert` represents a change in a specific `Symbol`: a newtype over a string of length equal to six like `USDEUR` and `GBPCHF` defined via refinement types.

```
type SymbolR = DescribedAs[
  Match["^[a-zA-Z0-9]{6}$"],
  "A Symbol should be an alphanumeric of 6 digits"
]

type Symbol = Symbol.Type
object Symbol extends Newtype[String :| SymbolR]
```

Next, `TradeUpdate` represents a change in the trading status: whether it is `On` or `Off`.

Among other datatypes we have not seen before, we have `AlertType`.

```
enum AlertType derives Show:
  case StrongBuy, StrongSell, Neutral, Buy, Sell
```

All the price-related datatypes are newtypes over `BigDecimal`, and `TradingStatus` is just `On` and `Off`, as we have previously learned.

At last, we have `PriceUpdate` representing a price change regarding a symbol.

```
final case class PriceUpdate(
    symbol: Symbol,
    prices: Prices
) derives Codec.AsObject, Eq, Show
```

We will soon learn about the need for this datatype.

6.3.2 Event-alert relationship

As previously stated, the relationship between data is essential to our understanding of the system, so let's see how alerts are being produced in the service's FSM.

STATUS	EVENT	ALERT
Off	Started	TradeUpdate(On)
On	Stopped	TradeUpdate(Off)
On	Started	N/A
Off	Stopped	N/A
*	Ignored	N/A
*	PriceUpdate	N/A
*	CommandRejected	N/A
*	CommandExecuted	TradeAlert(*) / PriceUpdate

The first two rows represent a change in the trading status. If we receive a `Started` event while the status is `On`, it will not have any effect, as shown in the third row; only the message is acknowledged. Same with `Stopped`.

Nothing is produced when receiving a `PriceUpdate`, only internal state changes, as we will learn in the next section (see FSM).

Next, a `SwitchEvent.Ignored` and a `TradeEvent.CommandRejected` are simply acknowledged without producing any alerts.

At last, we produce a `TradeAlert` when receiving a `CommandExecuted` event, and a `PriceUpdate` under certain conditions (details to follow up).

6.3.3 FSM

The `alerts`' finite-state machine also operates on `TradeState` as the state, but on a mix of messages `In` as inputs and no outputs (`Unit`).

```
type In = Msg[TradeEvent | SwitchEvent | PriceUpdate]
```

The FSM constructor looks as follows.

```
def fsm[F[_]: GenUUID: Logger: MonadCancelThrow: Time](
    appId: AppId,
    alertProducer: Producer[F, Alert],
    pricesProducer: Producer[F, PriceUpdate],
    pulsarTx: Resource[F, Txn],
    tradeAcker: Acker[F, TradeEvent],
    switchAcker: Acker[F, SwitchEvent],
    pricesAcker: Acker[F, PriceUpdate]
): FSM[F, TradeState, In, Unit] =
    def mkIdTs: F[(AlertId, Timestamp)] =
      (GenUUID[F].make[AlertId], Time[F].timestamp).tupled

    FSM { ... }
```

Figure 6.3.1 shows the `alerts` service producing and consuming `PriceUpdate`s, which is enough to understand why the FSM has `PriceUpdate` as one of the input types and its constructor takes a `Producer[F, PriceUpdate]` as argument. We will explain the reason for doing so shortly (see Scalability).

Within the FSM brackets block shown above, we have all the transitions described in the relationship table. Let's first look at the simple ones that only acknowledge the message.

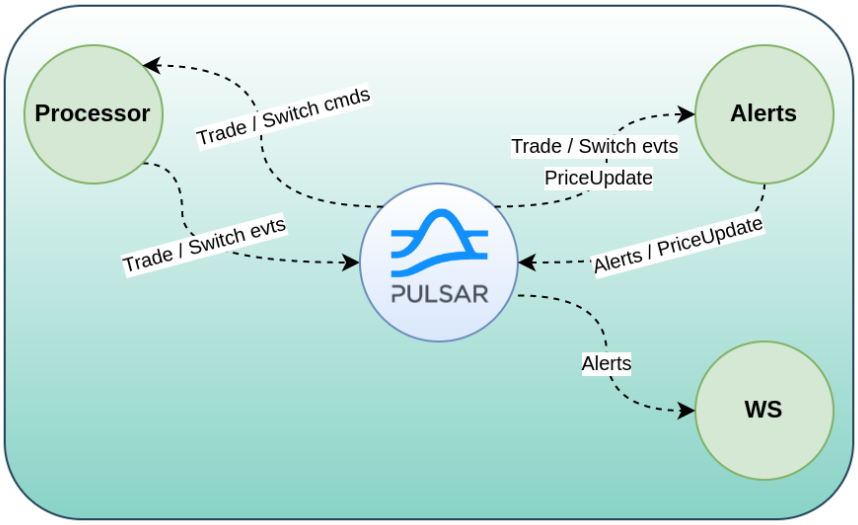


Figure 6.3.1: core services v2

```
case (st, Msg(msgId, _, SwitchEvent.Ignored(_, _, _))) ⇒
  switchAcker.ack(msgId).tupleLeft(st)
case (st, Msg(msgId, _, TradeEvent.CommandRejected(_, _, _, _, _))) ⇒
  tradeAcker.ack(msgId).tupleLeft(st)
```

Next, we have `PriceUpdate`, which only triggers internal state changes.

```
case (st, Msg(msgId, props, PriceUpdate(symbol, prices))) ⇒
  val nst = props.get("app-id") match
    case Some(id) if id =!= appId.id.show ⇒
      TradeState.__Prices.at(symbol).replace(Some(prices))(st)
    case _ ⇒ st
  pricesAcker.ack(msgId).tupleLeft(nst)
```

We first look into the message properties: if it contains an `app-id` different from the current one, we update the internal prices for the given symbol. Otherwise, we return the previous state. Details will be unveiled in the scalability section.

Next, let's analyze how the main `CommandExecuted` event is handled.

```
case (st, Msg(msgId, _, evt: TradeEvent.CommandExecuted)) ⇒
  val nst = TradeEngine.eventsFsm.runS(st, evt)
  val cmd = evt.command
  val p   = st.prices.get(cmd.symbol)
  val c   = nst.prices.get(cmd.symbol)

  val previousAskMax: AskPrice =
```

```
    p.flatMap(_.ask.keySet.maxOption).getOrElse(Price(0.0))
  val currentAskMax: AskPrice =
    c.flatMap(_.ask.keySet.maxOption).getOrElse(Price(0.0))

  // dummy logic to simulate the trading market
  def alert(id: AlertId, ts: Timestamp): Alert =
    if previousAskMax - currentAskMax > Price(0.3) then
      TradeAlert(id, cid, StrongBuy, symbol, ..., ts)
    else
      TradeAlert(id, cid, Neutral, symbol, ..., ts)

  val priceUpdate = c.flatMap { prices ⇒
    (p =!= c).guard[Option].as(PriceUpdate(cmd.symbol, prices))
  }

  mkIdTs.map(mkAlert).flatMap { alert ⇒
    sendAck(alert, priceUpdate, tradeAcker.ack(msgId, _))
      .tupleLeft(nst)
      .handleErrorWith { e ⇒
        Logger[F].warn(s"Transaction failed: ${e.getMessage}").tupleLeft(st)
      }
  }
```

We first acquire a new `TradeState` by running the event through the trading FSM.

The `alert` method simulates the emission of alerts (omitting some details for brevity), whereas the `priceUpdate` produces a `Some` whenever the previous and current prices for the symbol are different, signalling a price change.

Ultimately, we invoke the `sendAck` helper method, which does a few things.

```
def sendAck(
    alert: Alert,
    priceUpdate: Option[PriceUpdate],
    ack: Txn ⇒ F[Unit]
): F[Unit] =
  pulsarTx.use { tx ⇒
    for
      _ <- alertProducer.send(alert, tx)
      _ <- priceUpdate.traverse_ {
             pricesProducer.send(_, Map("app-id" → appId.id.show), tx)
           }
      _ <- ack(tx)
    yield ()
  }
```

Within a Pulsar transaction (see Transactions), it:

1. produces an `Alert`.
2. when non-empty, produces a `PriceUpdate` including the current `AppId`.
3. acknowledges the input message.

If something goes wrong and the transaction is aborted, the internal state is left untouched.

Finally, we have both `Started` and `Stopped` events.

```
case (st, Msg(msgId, _, evt: SwitchEvent)) ⇒
  switch(evt.cid, msgId, st, TradeEngine.eventsFsm.runS(st, evt))
```

This appears after the handling of `SwitchEvent.Ignored` (for an exhausted pattern matching), invoking the `switch` helper method.

```
def switch(
    cid: CorrelationId,
    msgId: MsgId,
    st: TradeState,
    nst: TradeState
): F[(TradeState, Unit)] =
  mkIdTs.map(TradeUpdate(_, cid, nst.status, _)).flatMap { alert ⇒
    sendAck(alert, None, switchAcker.ack(msgId, _)).tupleLeft(nst)
      .handleErrorWith { e ⇒
        Logger[F].warn(s"Transaction failed: ${e.getMessage}").tupleLeft(st)
      }
  }
```

It sends a `TradeUpdate` alert with the new `TradingStatus`, while handling a potential transaction failure.

6.3.4 Entry point

The service's entry point is also a `Main` object, with two different subscriptions.

```
def trSub(appId: AppId) =
  Subscription.Builder
    .withName(appId.name)
    .withType(Subscription.Type.KeyShared)
    .build

def swSub(appId: AppId) =
  Subscription.Builder
```

```
    .withName(appId.show)
    .withType(Subscription.Type.Exclusive)
    .build
```

The former being a `KeyShared` subscription for `TradeEvent`s, and the latter an `Exclusive` subscription with a unique identifier for `SwitchEvent`s and `PriceUpdate`s.

Next, we have the producer settings for `Alert` and `PriceUpdate`, respectively.

```
val altSettings =
  PulsarProducer
    .Settings[IO, Alert]()
    .withDeduplication
    .withMessageKey(Compaction[Alert].key)
    .some

val priSettings =
  PulsarProducer
    .Settings[IO, PriceUpdate]()
    .withDeduplication
    .withMessageKey(Compaction[PriceUpdate].key)
    .some
```

Both featuring deduplication and a message key, mainly used for topic compaction.

Last but not least, we have the consumer settings for `SwitchEvent`s.

```
val compact =
  PulsarConsumer.Settings[IO, SwitchEvent]().withReadCompacted.some
```

A sequence of resource acquisitions follows up.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url, Pulsar.Settings().withTransactions)
    _      <- Resource.eval(Logger[IO].info(s"Initializing: ${config.appId.show}"))
    alertsTopic  = AppTopic.Alerts.make(config.pulsar)
    pricesTopic  = AppTopic.PriceUpdates.make(config.pulsar)
    switchTopic  = AppTopic.SwitchEvents.make(config.pulsar)
    tradingTopic = AppTopic.TradingEvents.make(config.pulsar)
    snapshots      <- SnapshotReader.make[IO](config.redisUri)
    alertProducer  <- Producer.pulsar[IO, Alert](pulsar, alertsTopic, altSettings)
    pricesProducer <- Producer.pulsar[IO, PriceUpdate](pulsar, pricesTopic, priSettings)
    xSub = swSub(config.appId)
    tSub = trSub(config.appId)
```

```
    trConsumer <- Consumer.pulsar[IO, TradeEvent](pulsar, tradingTopic, tSub)
    swConsumer <- Consumer.pulsar[IO, SwitchEvent](pulsar, switchTopic, xSub, compact)
    puConsumer <- Consumer.pulsar[IO, PriceUpdate](pulsar, pricesTopic, xSub)
    server = Ember.default[IO](config.httpPort)
    txn = Txn.make(pulsar)
    engine = Engine.fsm(
      config.appId, alertProducer, pricesProducer, txn, trConsumer, swConsumer, puConsumer
    )
  yield (server, trConsumer, swConsumer, puConsumer, snapshots, engine)
```

We start by loading the application configuration followed by acquiring a connection to Pulsar with transactional support and creating a few topics.

`SnapshotReader` is used to read trade state snapshots when the service starts up, making it a stateful service.

```
trait SnapshotReader[F[_]]:
  def latest: F[Option[(TradeState, Consumer.MsgId)]]
```

This service only reads snapshots. The writing side lives on a different service that will be explored in Chapter 7.

Next, we have both producers for `Alert`s and `PriceUpdate`s, respectively. We also create the three required consumers to feed the state machine.

At last, we create the HTTP server and the FSM.

6.3.5 Scalability

At first sight, the architecture of this service resembles that of `processor`.

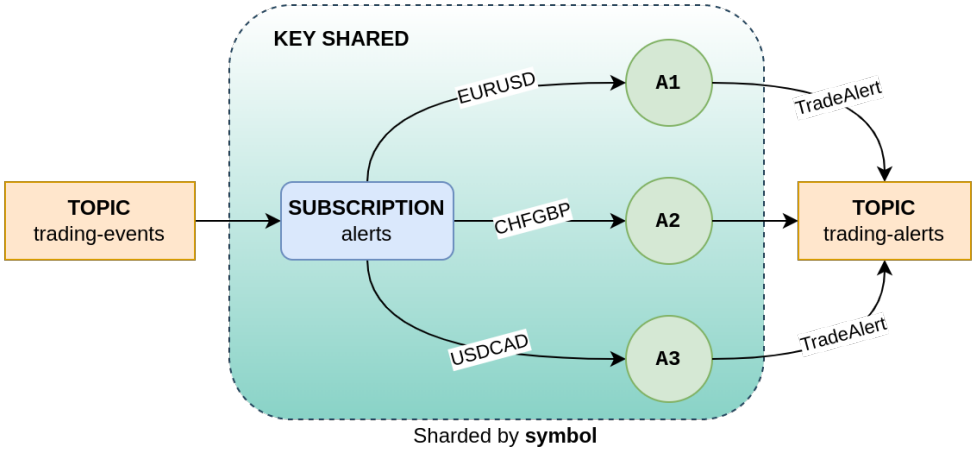


Figure 6.3.2: alerts scalability (trading)

The consumption of `TradeEvent`s is sharded by `Symbol` (`KeyShared` subscription), meaning different instances of the service will process unique symbols, as fig. 6.3.2 shows.

Besides publishing `TradeEvent`s, we can observe from the previous code that the `alerts` service also publishes `PriceUpdate`s to a different compacted topic.

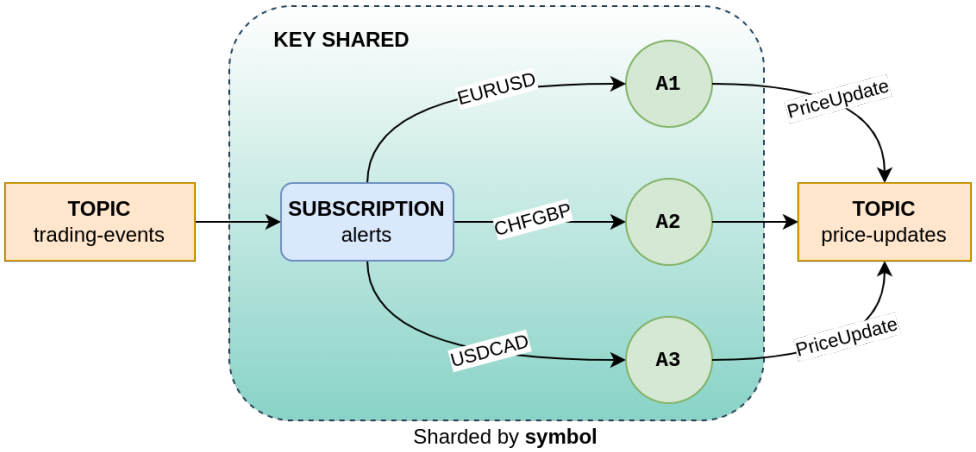


Figure 6.3.3: alerts scalability (prices)

`PriceUpdate`s are consumed by every active consumer, as we will learn shortly.

Similarly to how `SwitchCommand`s are processed by the `processor` service, here we process `SwitchEvent`s via an `Exclusive` subscription, as fig. 6.3.4 shows.

Figure 6.3.4: alerts scalability (switch)

Finally, fig. 6.3.5 shows the `PriceUpdate` exclusive subscription.

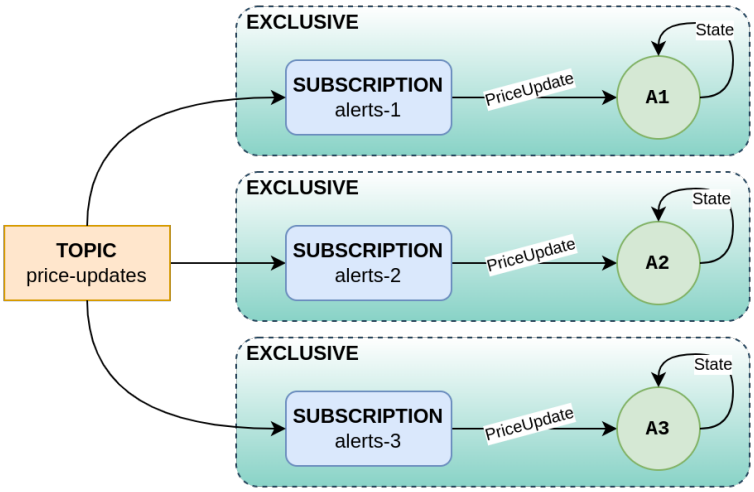


Figure 6.3.5: alerts scalability (price updates)

Why do we need to produce and consume `PriceUpdate`s?

In a few words, because of the *rebalancing of consumers*; the inherent complexity of the `KeyShared` subscription type in stateful services.

6.3.5.1 Rebalance

When an instance crashes or a new one is added, Pulsar will automatically rebalance the active consumers, as we have learned with the previous service (see Processor rebalance).

Figure 6.3.6 shows the `A2` instance becoming unavailable, triggering a rebalance.

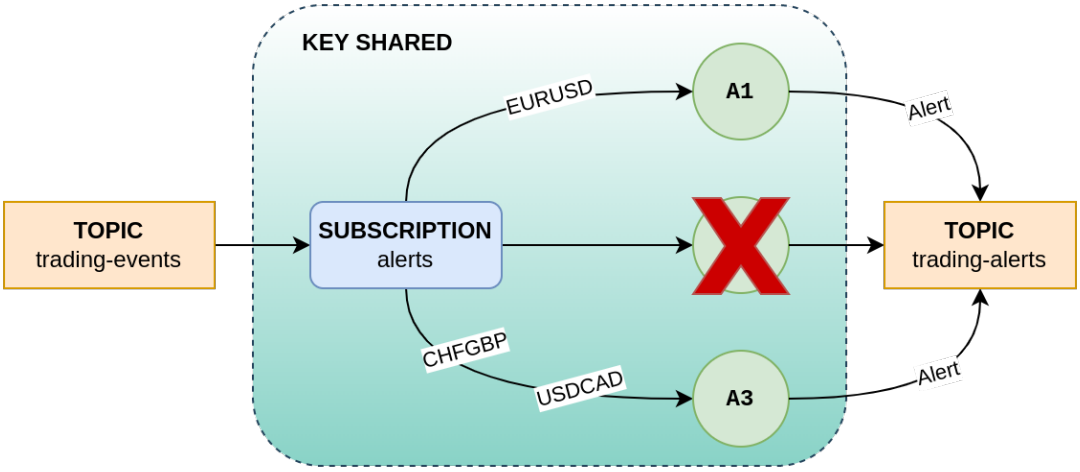


Figure 6.3.6: alerts scalability (rebalance)

On startup, every instance picks up the latest snapshot of `TradeState` and it continues with its own aggregation via the FSM. Let's recap on the state's shape.

```
final case class TradeState(
    status: TradingStatus,
    prices: Map[Symbol, Prices]
) derives Eq, Show
```

The initial snapshot contains the aggregation of the symbols processed by all instances. We will get the complete picture once we unveil the writing side (see Snapshots).

So, when a service starts processing incoming events, Pulsar will be delivering a specific set of symbols *only* to this particular instance, ensuring no other instance processes the price fluctuations of the same symbol.

This means that, after a consumers' rebalance, a running instance will still have the initial (now stale) state for the symbols that were being processed by the instance that went down.

For this reason, each instance shares the `PriceUpdate`s of the symbols they process, so that other instances are kept updated and ready for when a rebalance occurs.

By including the application ID in the message metadata, we ensure we only process incoming `PriceUpdate`s from other instances to avoid duplicate work.

In the example showcased by fig. 6.3.6, the `A3` instance picks up the processing of alerts for `CHFGBP` after `A2` goes down, able to continue processing alerts thanks to having an up-to-date state for such symbol.

Whenever `A2` comes back online, another rebalance of consumers will occur. In this case, `A2` will start over from a `TradeState` snapshot while also having `PriceUpdate`s and `SwitchEvent`s at hand.

This solution stems from the accidental complexity of the `KeyShared` subscription in a stateful service like this one. However, we have two other potential solutions:

1. Use a `Failover` subscription instead of `KeyShared`.
2. Use Pulsar's sticky consistent hashing for `KeyShared` subscriptions.

`1` is the easiest (in fact, that's how this service started), as we don't have to deal with the rebalancing problem, but **it doesn't scale** using single topics: we can only have one instance running at a time. It could work using partitioned topics[8], but not without introducing another set of challenges, some of which are shared with the second solution (i.e. fixed number of consumers, hard to update).

`2` is a solid choice, even though undocumented at the moment of writing. Instead of having dynamic routing of messages based on a sharding key consumers, it forces us to declare how many consumers we have from the start, and what hashes (computed from the sharding key) every consumer gets.

This guarantees no rebalancing of consumers, which removes the need for `PriceUpdate` sharing among instances. However, it is quite limiting to start with a fixed number of instances. A dynamic number of instances allows us to increase them during peak times while reducing them in quiet times, becoming more cost-effective.

The other big limitation of this solution is **downtime**. If an instance becomes unavailable, all consumers would stop processing messages until it comes back up. This is due to implementation details of this technique in Pulsar (supported since v2.7.0), but it may change in future versions.

Being aware of the different solutions to solve a problem puts us ahead of the game, so performing this kind of analysis should be part of any system design.

[8] https://pulsar.apache.org/docs/en/concepts-messaging/#partitioned-topics

6.3.6 Run

The way in which events start being fed to the FSM presents some challenges as well. Let's start by looking at the `run` method of the `alerts` service.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (server, trConsumer, swConsumer, puConsumer, snapshots, fsm) ⇒
      Stream.eval(server.useForever).concurrently {
        Stream.eval(snapshots.latest).flatMap {
          case Some(st, id) ⇒
            Stream.eval(IO.deferred[Unit]).flatMap { gate ⇒
              trConsumer
                .rewind(id, gate)
                .either(Stream.exec(gate.get) ++ swConsumer.receiveM)
                .either(Stream.exec(gate.get) ++ puConsumer.receiveM)
                .union2
                .evalMapAccumulate(st)(fsm.run)
            }
          case None ⇒
            trConsumer.receiveM
              .either(swConsumer.receiveM)
              .either(puConsumer.receiveM)
              .union2
              .evalMapAccumulate(TradeState.empty)(fsm.run)
        }
      }
    }
    .compile
    .drain
```

As usual, we start by running the HTTP server in the background while concurrently fetching the latest `TradeState` snapshot and consuming messages that are run through the FSM.

However, we have logic we are seeing for the first time. When we get a snapshot, we rewind the `TradeEvent`'s consumer to the message that corresponds with the `TradeState` we got. This is where it gets tricky.

If we rewind that consumer but continue to process both `SwitchEvent`s and `PriceUpdate`s, we will end up messing up the internal state, as the last two come from compacted topics (always yielding their latest unique values).

For this reason, we synchronize the rewinding of the consumer via `Deferred[Unit]`.

```
extension [F[_]: Monad, A](c: Consumer[F, A])
  def rewind(
      id: MsgId,
      gate: Deferred[F, Unit]
  ): Stream[F, Msg[A]] =
    Stream.eval(c.lastMsgId).flatMap { lastId ⇒
      c.receiveM(id).evalTap { msg ⇒
        gate.complete(()).whenA(lastId == Some(msg.id))
      }
    }
```

When we have processed all the `TradeEvent`s up to the last message, the `Deferred` is completed, allowing both the `SwitchEvent` and `PriceUpdate` consumers to resume.

```
.either(Stream.exec(gate.get) ++ swConsumer.receiveM)
.either(Stream.exec(gate.get) ++ puConsumer.receiveM)
```

Last but not least, we have a `union2` extension method defined as follows.

```
extension [F[_], A, B, C](
    src: Stream[F, Either[Either[Msg[A], Msg[B]], Msg[C]]]
)
  def union2: Stream[F, Msg[A | B | C]] =
    src.map {
      case Left(Left(ma))  ⇒ ma.asInstanceOf[Msg[A | B | C]]
      case Left(Right(mb)) ⇒ mb.asInstanceOf[Msg[A | B | C]]
      case Right(mc)       ⇒ mc.asInstanceOf[Msg[A | B | C]]
    }
```

Support for operations on top of union types is still playing catch-up in many libraries, so we have to add this ourselves. In some cases, `merge` works with union types when we explicitly indicate the expected type, but this is not one of those.

Summing up, we can come up with a solution such as this one only after thoroughly thinking about scalability and edge cases. Otherwise, we would not even find out we have a problem.

6.4 Web Sockets

The final core service is the one responsible for handling alerts and subscriptions via Web Sockets (WS).

Once connected, users can subscribe and unsubscribe to alerts from determined symbols such as `EURUSD` or `GBPCHF`—all anonymously, as there are no requirements for authentication.

As fig. 6.4.1 shows, this service consumes `Alert`s and handles WS messages. We will learn about these in the section below.

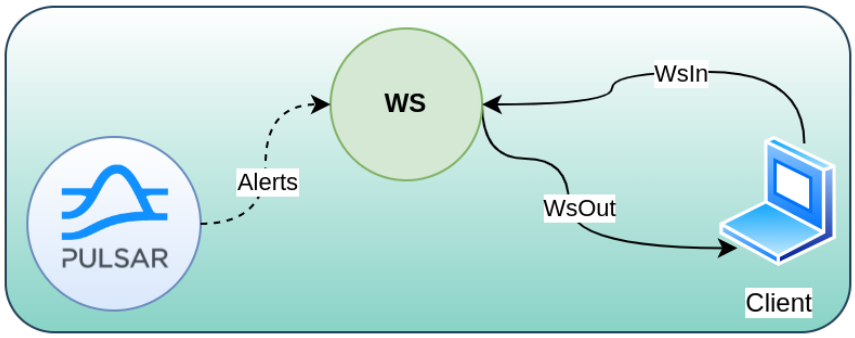


Figure 6.4.1: WS service

6.4.1 Datatypes

We have two types of messages to communicate via WS: the outgoing messages.

```
enum WsOut derives Codec.AsObject, Show:
  case Attached(sid: SocketId)
  case Notification(alert: Alert)
  case OnlineUsers(n: Int)
```

And the incoming messages.

```
enum WsIn derives Codec.AsObject, Show:
  case Close
  case Heartbeat
  case Subscribe(symbol: Symbol)
  case Unsubscribe(symbol: Symbol)
```

`Attached` is a message sent to the client when a connection is acquired, including a unique socket identifier. `Notification` is any alert we receive from the `alerts` service. Ultimately, `OnlineUsers` indicates the number of active users connected to the server.

Additionally, we have an extension method for converting any `Alert` to `WsOut`.

```
extension (alert: Alert)
  def wsOut: WsOut = WsOut.Notification(alert)
```

As far as the incoming messages go, users can only `Subscribe` and `Unsubscribe` to symbols, so this is all interaction this service will have with them.

The `Close` and `Heartbeat` messages are meant for internal use, helping us keep a healthy Web Sockets connection while gracefully handling shutdowns, as we will learn in a few sections (see Events handler).

6.4.2 HTTP routes

The HTTP interface for Web Sockets is defined as follows.

```
final class WsRoutes[F[_]: GenUUID: Logger: Monad](
    ws: WebSocketBuilder[F],
    mkHandler: SocketId ⇒ F[Handler[F]]
) extends Http4sDsl[F]:

  val routes: HttpRoutes[F] = HttpRoutes.of {
    case GET → Root / "v1" / "ws" ⇒
      GenUUID[F].make[SocketId]
        .flatMap(mkHandler)
        .flatMap(h ⇒ ws.build(h.send, h.receive))
  }
```

The `mkHandler` function takes a unique socket identifier and returns a function that constructs a `Handler[F]`, which we will see shortly.

The `WebSocketBuilder` provided by Http4s defines a `build` method with the following type signature.

```
def build(
    send: Stream[F, WebSocketFrame],
    receive: Pipe[F, WebSocketFrame, Unit],
): F[Response[F]]
```

Therefore, when a new connection is initiated, a new `Handler` with a unique `SocketId` is created, followed by returning a Web Socket response.

6.4.3 Events handler

The WS events handler is one of the most complex pieces of the system, albeit having a simple interface matching the arguments of `WebSocketBuilder#build`.

```
trait Handler[F[_]]:
  def send: Stream[F, WebSocketFrame]
  def receive: Pipe[F, WebSocketFrame, Unit]
```

Its main constructor takes three different arguments.

```
def make[F[_]: Concurrent: Logger](
    sid: SocketId,
    conns: WsConnections[F],
    alerts: Stream[F, Alert]
): F[Handler[F]] = ???
```

- `sid`: unique socket identifier.
- `conns`: interface keeping track of total WS connections.
- `alerts`: stream of `Alert`s to be processed.

Next, we have the definition of the `WsConnections` interface.

```
trait WsConnections[F[_]]:
  def get: F[Int]
  def subscriptions: Stream[F, Int]
  def subscribe(sid: SocketId): F[Unit]
  def unsubscribe(sid: SocketId): F[Unit]
```

Its main implementation is backed by Fs2's `SignallingRef`.

```
object WsConnections:
  def make[F[_]: Concurrent]: F[WsConnections[F]] =
    SignallingRef.of[F, Set[SocketId]](Set.empty).map { ref ⇒
      new:
        def get: F[Int] = ref.get.map(_.size)
        def subscriptions: Stream[F, Int] = ref.discrete.map(_.size)
        def subscribe(sid: SocketId): F[Unit] = ref.update(_ + sid)
        def unsubscribe(sid: SocketId): F[Unit] = ref.update(_ - sid)
    }
```

It gives us access to a discrete stream (`subscriptions`) of the current number of connections—i.e. it emits a value (the set's size) for every change.

Continuing with the implementation of `Handler`, we start by creating the following internal primitives.

```
(
  Deferred[F, Either[Throwable, Unit]],
  Deferred[F, Unit],
  Ref.of[F, Set[Symbol]](Set.empty)
).mapN { case (switch, fuze, subs) ⇒
    ???
}
```

- `switch`: synchronizes the termination of the handler.
- `fuze`: synchronizes the first received message to avoid losing subscriptions.
- `subs`: keeps track of symbol subscriptions.

We also have a few helper functions within the blocks—the first one to encode outgoing messages.

```
val toWsFrame: WsOut ⇒ WebSocketFrame =
  out ⇒ Text(out.asJson.noSpaces)

val encode: WsOut ⇒ F[Option[WebSocketFrame]] =
  case out @ WsOut.Notification(t: TradeAlert) ⇒
    subs.get.map(_.find(_ === t.symbol).as(toWsFrame(out)))
  case out ⇒
    toWsFrame(out).some.pure[F].widen
```

When we get a `Notification` with a trade alert, it is sent out to the client *if and only if* there is a subscription for the alert's symbol. In any other case, the message is sent out without further checks.

The following helper function decodes WS messages into incoming messages.

```
val decode: WebSocketFrame ⇒ Either[String, WsIn] =
  case Close(_)     ⇒ WsIn.Close.asRight
  case Text(msg, _) ⇒ jsonDecode[WsIn](msg).leftMap(_.getMessage)
  case e            ⇒ s">>> [$sid] - Unexpected WS message: $e".asLeft
```

We encode messages as JSON, so the format needs to be agreed upon with the possible clients of the API. To ensure we do not break compatibility, it is advisable to have golden tests or round-trip conversion tests with fixed inputs.

The `WsCodecSuite` in the trading project showcases the latter, which adds some coverage for potential breakage.

Next, we have the implementation of `send`.

```
val attached =
  Stream.eval {
    conns.subscribe(sid) *> encode(WsOut.Attached(sid))
  }

val onlineUsers =
  conns.subscriptions.evalMap { n ⇒
    encode(WsOut.OnlineUsers(n))
  }

val send: Stream[F, WebSocketFrame] =
  onlineUsers
    .mergeHaltR {
      attached ++ alerts.evalMap { x ⇒
        fuze.get *> encode(x.wsOut)
      }
    }
    .interruptWhen(switch)
    .unNone
```

The `onlineUsers` stream sends out a message every time there is a change in the number of subscriptions, and it runs concurrently with the `attached` and `alerts` streams.

The `attached` stream first registers the connection (`subscribe`), followed by sending the initial `Attached` message. The `alerts` stream encodes every input as a `WebSocketFrame` in the appropriate format.

That `fuze.get` is only necessary on the first run, when there could be a race condition between subscribing and emitting alerts, which is completed by `receive`, as we will analyze next.

Recalling that `Pipe[F, I, O]` is merely a type alias for `Stream[F, I] ⇒ Stream[F, O]`, let's look at this implementation.

```
val close = conns.unsubscribe(sid)

val receive: Pipe[F, WebSocketFrame, Unit] =
  _.evalMap {
    decode(_) match
      case Left(e) ⇒
        Logger[F].error(e)
      case Right(WsIn.Heartbeat) ⇒
        ().pure[F]
      case Right(WsIn.Close) ⇒
```

```
        close *> switch.complete(().asRight).void
      case Right(WsIn.Subscribe(symbol)) ⇒
        subs.update(_ + symbol)
      case Right(WsIn.Unsubscribe(symbol)) ⇒
        subs.update(_ - symbol)
  }.onFinalize {
      Logger[F].info(s"[$sid] - WS connection terminated") *> close
  }.onFirstMessage(fuze.complete(()).void)
```

It receives `WebSocketFrame` messages and tries to decode them into `WsIn` messages. The first case `Left(e)` is a decoding error, where we simply log the error and continue processing messages.

We do this instead of sending a negative acknowledgement (in fact, we use auto-ack) because alerts fit the use case where a new message make the previous one obsolete. Reprocessing an old message afterwards would give us incorrect results.

The `Heartbeat` only matters to the underlying connection, so we ignore it.

The `Close` message signals that the connection has been gracefully shutdown, in which case we `unsubscribe` the connection and complete the `switch` that will trigger the interruption of the `send` stream.

We update the internal subscriptions either on `Subscribe` or `Unsubscribe`.

Finally, when the first message is received, we complete the `fuze`, ensuring both `send` and `receive` are in sync.

This is done via a custom extension method defined as follows.

```
extension [F[_], A](src: Stream[F, A])
  /* Perform an action when we get the first message without consuming it twice */
  def onFirstMessage(action: F[Unit]): Stream[F, A] =
    src.pull.uncons.flatMap {
      case Some((chunk, tl)) ⇒
        Pull.eval(action) >> Pull.output(chunk) >> tl.pull.echo
      case None ⇒ Pull.done
    }.stream
```

The complexity of this implementation lives in all the inevitable race conditions exposed by the underlying WS connection implementation. Although it strives for simplicity, it does not allow for much customization when it comes to handling stream interruptions.

6.4.4 Unit tests

Inherently, testing this complex implementation requires a little bit of thinking. We have a dummy implementation of the `WsConnections` named `conns`, and the following primitives.

```
test("WS message handler") {
  (
    GenUUID[IO].make[SocketId],
    IO.ref(List.empty[WebSocketFrame]),
    IO.deferred[Unit],
    IO.deferred[Either[Throwable, Unit]]
  ).tupled.flatMap { (sid, out, switch, connected) ⇒ ??? }
}
```

- `sid`: unique socket identifier.
- `out`: all `WsOut` messages to be sent.
- `switch`: to synchronize the sending of the `WsIn.Close` message.
- `connected`: to know when there is an active subscription.

Next, we have the body of the test.

```
Handler.make(sid, conns, Stream.emits(alerts)).flatMap { h ⇒
  val recv =
    Stream.emits(input).append {
      Stream.eval(switch.get.as(Text(WsIn.Close.asJson.noSpaces)))
    }.through(h.receive)
     .void

  val send =
    h.send
      .evalMap(wsf ⇒ out.update(_ :+ wsf))
      .onFinalize(switch.complete(()).void)

  Stream(recv, send).parJoin(2).compile.drain
} >> out.get.flatMap {
  case Text(x, _) :: Text(y, _) :: xs ⇒
    NonEmptyList.of(
      expect((x + y).contains("Attached")),
      expect((x + y).contains("OnlineUsers")),
      expect.same(xs.size, alerts.size - 1)
    )
    .reduce
    .pure[IO]
  case _ ⇒
```

```
    out.get
     .flatMap(_.traverse_(IO.println))
     .as(failure("expected non-empty list"))
}
```

The `recv` stream emits a finite number of `WsIn` messages to simulate interaction with a client and sends them through the handler's `receive` function.

```
val input = List(
  WsIn.Subscribe(sl1),
  WsIn.Heartbeat,
  WsIn.Unsubscribe(sl2),
  WsIn.Subscribe(sl1),
  WsIn.Heartbeat
).map(in ⇒ Text(in.asJson.noSpaces))
```

As soon as these are sent, it waits for the `switch` to be completed so it can finally send out the `WsIn.Close` message.

On the other hand, the `send` stream consumes messages from the handler's `send` function, updating the list of `WsOut` messages received.

When the stream finalizes, the `switch` is completed, which will trigger the graceful termination of the entire stream.

Ultimately, we retrieve the content of `out` and run some expectations on it, based on the `Attached` and the `OnlineUsers` messages, and the number of alerts produced by our `subscribe` test implementation.

```
val alerts = List(
  Alert.TradeAlert(id, cid, AlertType.Buy, sl1, p1, p1, p1, p1, ts),
  Alert.TradeUpdate(id, cid, TradingStatus.Off, ts),
  Alert.TradeAlert(id, cid, AlertType.Sell, sl1, p1, p1, p1, p1, ts),
  Alert.TradeUpdate(id, cid, TradingStatus.On, ts),
  Alert.TradeAlert(id, cid, AlertType.StrongSell, sl1, p1, p1, p1, p1, ts),
  Alert.TradeAlert(id, cid, AlertType.Neutral, sl2, p1, p1, p1, p1, ts)
)
```

Notice that the first message could either be `Attached` or `OnlineUsers`, as these two streams run concurrently. Hence the assertion `(x + y).contains("Attached")`.

6.4.5 Entry point

This service does a few things differently, so we will perform a deep analysis.

To begin with, we have a function that takes a `SocketId` and it builds a non-durable and exclusive Pulsar subscription. The non-durable mode represents a lightweight subscription that doesn't have a durable cursor associated to it.

```
val mkSub = (sid: SocketId) ⇒
  Subscription.Builder
    .withName(s"ws-server-${sid.show}")
    .withMode(Subscription.Mode.NonDurable)
    .withType(Subscription.Type.Exclusive)
    .build
```

This means we will create a consumer per open socket (aka *ephemeral topic*), as we will discuss shortly.

Next, we have the settings enabling the consumer to read from a compacted topic.

```
val compact =
  PulsarConsumer
    .Settings[IO, Alert]()
    .withInitialPosition(SubscriptionInitialPosition.Earliest)
    .withReadCompacted
    .some
```

We also set the initial position to `Earliest`, so that we get all the alerts from the beginning of time (as we do in event sourcing). We can do this because the topic is compacted, so we will always get the latest alerts of each symbol without affecting the startup time.

Following up, we have a sequence of resources.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url)
    _      <- Resource.eval(Logger[IO].info("Initializing ws-server service"))
    ptopic = AppTopic.Alerts.make(config.pulsar)
    conns <- Resource.eval(WsConnections.make[IO])
    mkConsumer = (sid: SocketId) ⇒ Consumer.pulsar[IO, Alert](
                   pulsar, ptopic, mkSub(sid), compact
                 )
    mkAlerts   = (sid: SocketId) ⇒ Stream.resource(mkConsumer(sid)).flatMap(_.receive)
    mkHandler  = (sid: SocketId) ⇒ Handler.make[IO](sid, conns, mkAlerts(sid))
    server     = Ember.websocket[IO](config.httpPort, WsRoutes[IO](_, mkHandler).routes)
  yield conns → server
```

As previously mentioned, we use **receive** on the consumer, which performs auto-ack (see Events handler).

Unlike previous services, we use **Ember.websocket**, a custom constructor defined under the **core** module.

```
def websocket[F[_]: Async: Console](
    port: Port,
    f: WebSocketBuilder[F] ⇒ HttpRoutes[F]
): Resource[F, Server] =
  metrics[F].flatMap { mid ⇒
    make[F](port)
      .withHttpWebSocketApp { ws ⇒
        mid(f(ws) <+> HealthRoutes[F].routes).orNotFound
      }
      .build
      .evalTap(showBanner[F])
  }
```

This is the implementation, without getting too much into details. The key difference is that we use **withHttpWebSocketApp** instead of the classic **withHttpApp**.

6.4.6 Run

At last, the execution of the `ws` application.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (conns, server) ⇒
      Stream.eval(server.useForever).concurrently {
        conns.subscriptions.evalMap { n ⇒
          Logger[IO].info(s"WS connections: $n")
        }
      }
    }
    .compile
    .drain
```

We run the HTTP server while concurrently logging the number of WS connections—triggered every time a client registers or unregisters.

Subscriptions

New Subscription

Subscription Name	Type
ws-server-3958b342-dd92-487d-9794-a1ade9be8710-subscription	Exclusive
ws-server-eec3a57c-6b0e-4ee2-bfa3-ca9c4d9b1f2b-subscription	Exclusive
ws-server-b16d3f33-9487-4d47-902e-754db283841a-subscription	Exclusive
tracing-subscription	Exclusive

Figure 6.4.2: Alerts' consumers

Figure 6.4.2 shows three different consumers originated from this service (ignore the other one for now). Every consumer runs in exclusive mode with a unique subscription name corresponding to the `SocketId`.

6.4.7 Scalability

Scalability is another important aspect of this service. In the following example showcased by fig. 6.4.3, we can observe two `ws` instances running concurrently, each of them handling exclusive subscriptions to the `trading-alerts` topic.

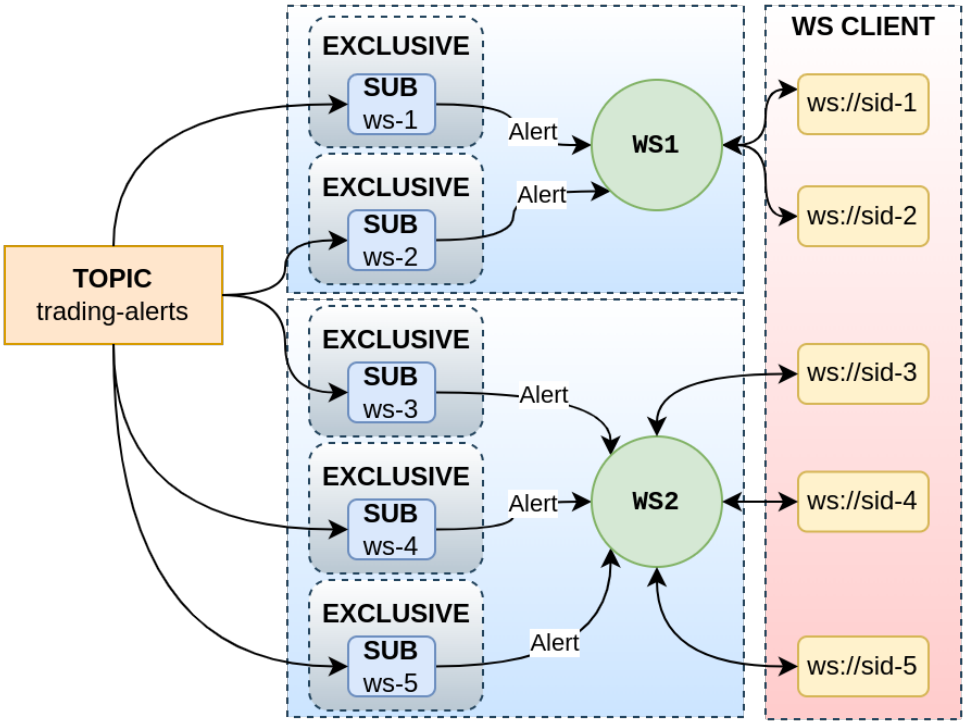


Figure 6.4.3: WS scalability

This particular design enables horizontal scalability, allowing us to run as many instances as needed to handle traffic during peak time while reducing costs over quiet times.

At a massive scale, a potential bottle-neck could be the single `trading-alerts` topic. To increase the throughput, the next move would be to use a partitioned topic so that it can be served by multiple brokers.

A valid option is to start with a partitioned topic of a single partition straight away to make the change much easier in the future.

Each consumer serves a WS client uniquely identified by `SocketId`, as fig. 6.4.3 shows.

We make the subscription non-durable because there are no requirements (yet) to allow a client to reconnect and continue where it left off. This means that when the client disconnects, the subscription gets scheduled for deletion, allowing us to fearlessly create ephemeral topics as we go.

6.4.8 Addendum

To show that not everything is perfect from the start—and that code evolves—here's a fun design-change anecdote that I decided to keep thinking is a valuable lesson.

Initially, this service had a single alerts consumer broadcasting them into an internal `Topic` (from `fs2.concurrent`). That was sufficient for a while, and perhaps simpler.

The pertinent code looked as follows (where `topic` has the type `Topic[F, Alert]`).

```
consumer.receiveM.evalMap { case Consumer.Msg(id, _, alert) ⇒
  topic.publish1(alert) *> consumer.ack(id)
}
```

The `Handler` instances would then subscribe to this internal topic instead of consuming alerts directly from Pulsar.

```
topic.subscribe(100) // Stream[F, Alert]
```

The issue showed up when I started thinking about duplicate alerts. On the producer side, alerts are being deduplicated, so that is not a problem.

The problem was not relying on Pulsar (i.e. using an internal topic), which means we lose all guarantees of deduplication made by the broker.

It all boiled down to the two following operations.

```
topic.publish1(alert) *> consumer.ack(id)
```

It could happen that we broadcast an alert to all the internal topic's subscribers, and then the acknowledgement would fail. In such cases, Pulsar would schedule the same message for redelivery.

That means we could have duplicate alerts we would need to take care of. Inherently, this complicates the design and maintenance of the service.

So getting rid of the internal topic and instead shifting the duplicate responsibility on Pulsar greatly simplifies the implementation.

That is how the "one consumer per socket" solution came into place. You can find the corresponding initial changes in this pull request[9].

[9] https://github.com/gvolpe/trading/pull/78

6.5 Summary

So far, we have learned about the three core services capable of delivering the first milestone. Yet, a lot of work remains ahead of us.

We have discussed critical design decisions and learned about the capabilities of every service, including how each of them handles scalability challenges.

In the following two chapters, we will learn about the remaining services and some important features of distributed systems, such as monitoring and observability.

7 Trading system (alt services)

Now that we know about the core services, let's look at the few other ones that make the entire system, as shown in fig. 7.0.1.

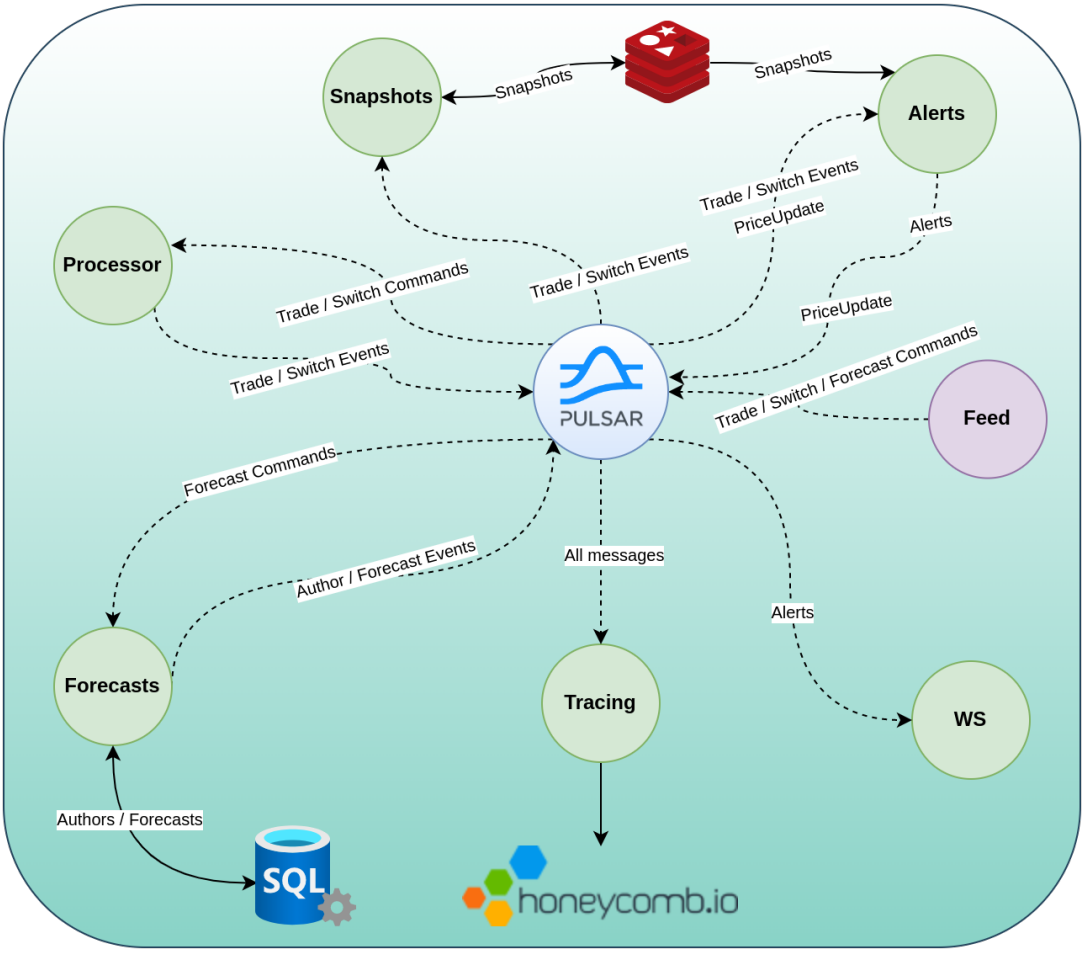


Figure 7.0.1: overview

Apache Pulsar is the core of our distributed system. A few services also require access to databases, caches, or external services, as we will learn in the two remaining chapters.

Additionally, we have a `feed` service generating random trading and forecasting commands, so that we can see the whole system in action (used for manual testing).

This chapter will focus on the `snapshots`, `forecasts`, and `feed` services. At last, we will also work on the definition of some integration tests.

We will explore the remaining `tracing` service in Chapter 8.

7.1 Snapshots

As we can observe in the system diagram, this service consumes both `TradeEvent`s and `SwitchEvent`s, and persists snapshots to Redis. So what does this really mean?

We briefly touched on this topic in Chapter 3 (see State snapshots). State snapshots represent the internal state at a given point in time. In our case, modeled as `TradeState`.

These are helpful to enable faster restarts, so the service does not need to replay the events from the beginning of time to recreate the current state.

What this service does, in essence, is to compute snapshots from `CommandExecuted` events in the form of `TradeState` every some configurable time and persist it together with the corresponding message ID, so that other services can read the latest version on startup.

Furthermore, every `SwitchEvent` results in an immediate write to Redis whenever the current status is different, as these events are somewhat rare.

We are already familiar with both datatypes, so let's get straight down to business.

7.1.1 Scalability

Before we dive into the subscription details and resources used in this service, let's discuss the design decisions that went into it.

The responsibility of this service is to write `TradeState` snapshots associated with a `MessageId` corresponding to the last `TradeEvent` processed, which needs to be coordinated when we have more than one running instance.

The simplest way to handle this is to ensure only one instance performs the writes at a given time. This is known as the *single writer principle*.

When talking about data replication (keeping a copy of the same data in multiple machines), the same principle is known as *leader-based* replication (also known as *active/passive* or *master-slave* replication).

For this type of applications, the `Failover` subscriptions seems ideal, allowing us to have a single active consumer with at least one fail-over consumer ready to make the switch when things go wrong.

This scenario is showcased in Figure 7.1.1, showing an instance named `S2` elected as the leader, while both `S1` and `S3` are ready to take over in case of failure.

However, further analysis of the design exposes a few flaws with this architecture.

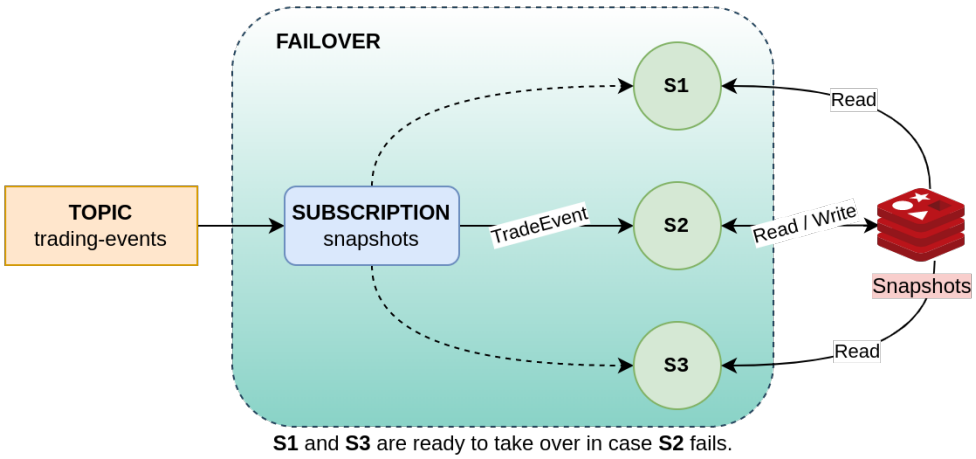


Figure 7.1.1: Snapshots in fail-over mode

7.1.1.1 Why not fail-over mode?

For a faster startup, we initially read the latest snapshot and rewind the subscription to the specific message ID.

However, the first time we run the application there will not be a snapshot. Our only option is to rewind the subscription to the beginning of the topic, slowing down its startup. So ideally, we would always want to start from the latest snapshot when possible.

Suppose all the instances shown in fig. 7.1.1 have started up at the same time and `S2` has been processing events for 30 hours before crashing. In that case, the internal state of both `S1` and `S3` will be outdated by the time they take over. To amend this, they need to re-fetch the latest snapshot and rewind the subscription accordingly.

This might sound easy, but it is not. First of all, because Pulsar does not have a mechanism to allow other instances to know when they become active consumers. If that was the case, we could be notified and prepare the state of our application before we start processing events.

Instead, Pulsar starts sending messages as soon as it assigns a new active consumer during a fail-over switch. It does not give us the chance to fetch the snapshot and rewind the subscription before starting.

This limitation can probably be overcome at the application level. When we detect a first message, we can discard it, fetch the snapshots and rewind the subscription (e.g. via the `onFirstMessage` extension method we saw in Chapter 6). Though, this does not feel natural, and requires a few code shenanigans to synchronize instances.

Mainly for this reason, we will implement this service using an exclusive subscription that is synchronized using a distributed lock powered by Redis (see Distributed locks in Chapter 2).

In the end, it will look as though we were using a `Failover` subscription with the switching mechanism being in our control via a distributed lock instead of being handled by Pulsar.

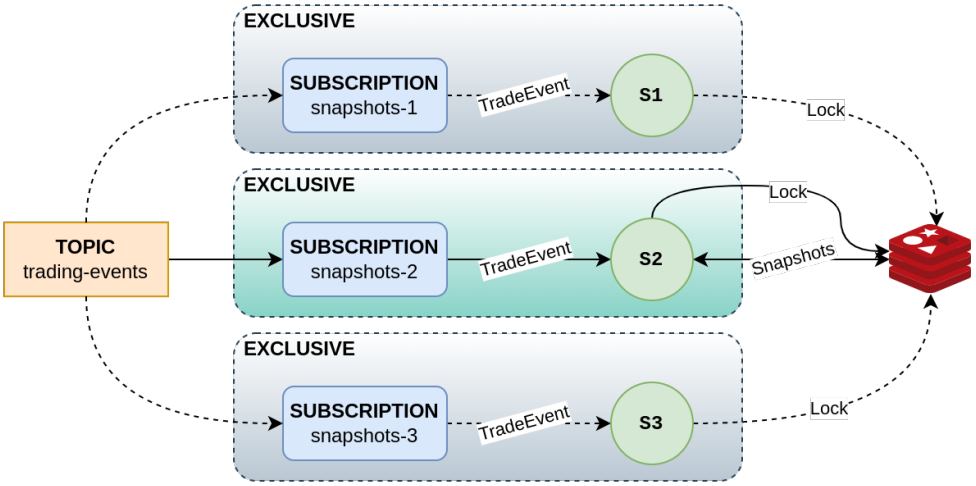


Figure 7.1.2: Snapshots scalability

Figure 7.1.2 showcases our chosen application architecture.

Another potential solution could be to publish snapshot updates to a compacted Pulsar topic instead of writing them to Redis, ensuring the message key is always the same. When compaction is triggered, only the latest snapshot will remain.

This would require services reading snapshots to synchronize the consumption of additional messages like the `alerts` service does with `PriceUpdate`s (see Alerts Rebalance).

Nevertheless, this service will follow the distributed lock design we discussed earlier to showcase how we can solve this problem differently.

7.1.2 Entry point

Our entry point starts with an `Exclusive` subscription per instance (via `AppId`).

```
def mkSub(appId: AppId) =
  Subscription.Builder
    .withName(appId.show)
    .withType(Subscription.Type.Exclusive)
```

```
    .withMode(Subscription.Mode.NonDurable)
    .build
```

We also make it non-durable, as there is no need to persist the cursor of the subscription. If the instance goes down, the next time will reconnect with a different application ID.

The same subscription is valid both for `TradeEvent`s and `SwitchEvent`s. However, the latter comes from a compacted topic, for which we need extra configuration settings.

```
val compact =
  PulsarConsumer.Settings[IO, SwitchEvent]().withReadCompacted.some
```

Next, we have a sequence of initialization of resources.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url)
    _      <- Resource.eval(Logger[IO].info("Initializing: ${config.appId.show}"))
    teTopic = AppTopic.TradingEvents.make(config.pulsar)
    swTopic = AppTopic.SwitchEvents.make(config.pulsar)
    redis <- RedisClient[IO].from(config.redisUri.value)
    distLock = DistLock.make[IO]("snap-lock", config.appId, redis)
    reader <- SnapshotReader.fromClient[IO](redis)
    writer <- SnapshotWriter.fromClient[IO](redis, config.keyExpiration)
    sub = mkSub(config.appId)
    trConsumer <- Consumer.pulsar[IO, TradeEvent](pulsar, teTopic, sub)
    swConsumer <- Consumer.pulsar[IO, SwitchEvent](pulsar, swTopic, sub, compact)
    fsm    = Engine.fsm(trConsumer, swConsumer, writer)
    server = Ember.default[IO](config.httpPort)
  yield (server, distLock, trConsumer, swConsumer, reader, fsm)
```

Besides resources we saw before, we have a `DistLock` interface with a `refresh` method—in addition to the implementation we saw in Chapter 2 (see Distributed locks).

```
trait DistLock[F[_]]:
  def refresh: F[Unit]
```

Its main constructor takes three arguments.

```
object DistLock:
  def make[F[_]: Logger: MkRedis: Temporal](
      lockName: String,
      appId: AppId,
      client: RedisClient
  ): Resource[F, DistLock[F]] = ???
```

Every instance will be competing to acquire a lock, uniquely identified by `AppId`. The one that gets it will become the elected leader responsible for processing writes.

Moreover, we have the pertinent `SnapshotWriter`.

```
trait SnapshotWriter[F[_]]:
  def save(state: TradeState, id: Consumer.MsgId): F[Unit]
  def saveStatus(st: TradingStatus): F[Unit]
```

The `MsgId` correlates to the captured `TradeState`, allowing us to later rewind a topic subscription to that specific point, to then continue aggregating the state.

Snapshots are persisted to Redis, for which the keys should always have an expiration time, as it is considered bad practice to store cache data indefinitely.

```
object SnapshotWriter:
  def from[F[_]: MonadThrow](
      redis: RedisCommands[F, String, String],
      exp: KeyExpiration
  ): SnapshotWriter[F] = ???
```

However, its implementation is not relevant here, so we will skip it. Please refer to the accompanying source code for details.

7.1.3 FSM

The corresponding state machine handles a few interesting edge cases, defining the following constructor.

```
type Tick = Unit

object Engine:
  type In = Either[Msg[TradeEvent], Msg[SwitchEvent]] | Tick

  def fsm[F[_]: MonadThrow: Logger](
      tradeAcker: Acker[F, TradeEvent],
      switchAcker: Acker[F, SwitchEvent],
      writer: SnapshotWriter[F]
  ): FSM[F, (TradeState, List[MsgId]), In, Unit] = FSM { ... }
```

Notice that we do not require de-duplication in this service because it is guaranteed only one instance at a time will be consuming such events.

First of all, we have `(TradeState, List[MsgId])` as the internal state and a combination of messages as inputs. We already know about `TradeState` so let's focus on that `List[MsgId]` and the state transitions.

Here's what happens every time it processes a `CommandExecuted` event.

```
case ((st, ids), Left(Msg(msgId, _, evt @ CommandExecuted(_, _, _, _)))) ⇒
  ().pure[F].tupleLeft(TradeEngine.eventsFsm.runS(st, evt) → (ids :+ msgId))
```

We get a new `TradeState` by running the event through the trading FSM and add the current `MsgId` to the internal state. We do not acknowledge the message yet.

Conversely, when we get a `CommandRejected` event, we simply acknowledge the message.

```
case (st, Left(Msg(msgId, _, CommandRejected(evId, _, _, _, _)))) ⇒
  Logger[F].debug(s"Event ID: $evId, no persistence") *>
    tradeAcker.ack(msgId).tupleLeft(st)
```

Next, we process a `SwitchEvent` by running it through the FSM to acquire a new state, followed by persisting the trading status (only if it has changed) and acknowledging the message.

```
case ((st, ids), Right(Msg(msgId, _, evt))) ⇒
  val nst = TradeEngine.eventsFsm.runS(st, evt)
  writer.saveStatus(nst.status).whenA(nst.status =!= st.status) *>
    switchAcker.ack(msgId).tupleLeft(nst, ids)
```

Ultimately, we handle the `Tick` message, which is arguably the most interesting one.

```
case ((st, ids), (_: Tick)) if ids.nonEmpty ⇒
  val lastId = ids.last
  writer.save(st, lastId).attempt.flatMap {
    case Left(e) ⇒
      Logger[F].warn(
        s"Failed to persist state: $lastId"
      ).tupleLeft(st → ids)
    case Right(_) ⇒
      Logger[F].debug(
        s"State persisted: $lastId. Acking ${ids.size} messages."
      ) *>
        tradeAcker.ack(ids.toSet).attempt.map {
          case Left(_)  ⇒ (st → ids)        → ()
          case Right(_) ⇒ (st → List.empty) → ()
        }
  }
case (st, (_: Tick)) ⇒
  ().pure[F].tupleLeft(st)
```

On every `Tick`, we attempt to persist the current state in Redis. If it fails, we log the error and continue processing messages without clearing the state.

When it succeeds, we perform a batch acknowledgement of the multiple `MsgId`s we have accumulated so far. If it all goes well, we clear the `List[MsgId]` and continue processing messages.

`Tick` messages are produced by ourselves in `Main`, as we will learn shortly (see Run).

7.1.3.1 Unit tests

All these cases are unit-tested to ensure a solid implementation. There are three major cases defined in terms of a `baseTest` method with the following type signature.

```
def baseTest(
    gen: Gen[Either[TradeEvent, SwitchEvent]],
    mkWriter: Ref[IO, Option[TradeState]] ⇒ SnapshotWriter[IO],
    expWrites: TradeState ⇒ Option[TradeState],
    expAcks: List[MsgId]
): IO[Expectations] = ???
```

The leading case is tested by generating `TradeEvent.CommandExecuted` values and ensuring the writing of the snapshots always succeeds (including a `Tick`).

The last three values prefixed with `exp` are our expectations. The first one indicates the expected `TradeState` is the same one we get from the trading FSM. The other one tracks acknowledgements.

```
test("fsm w/ command executed events should ack AND write new state") {
  baseTest(
    gen = genCommandExecEvt.map(_.asLeft),
    mkWriter = succesfulWriter,
    expWrites = _.some,
    expAcks = List(msgId)
  )
}
```

In the same way, we can test how our state machine handles other events different from `CommandExecuted` (i.e. `CommandRejected` and `SwitchEvent`s).

```
test("fsm w/ other events should ack without writing new state") {
  baseTest(
    gen = genTradeEventNoCmdExec,
    mkWriter = succesfulWriter,
    expWrites = _ ⇒ None,
    expAcks = List(msgId)
```

```
  )
}
```

Last but not least, how does our FSM deal with a writer that fails to persist snapshots?

```
test("snapshot fsm w/ failing snapshot writer should NOT ack") {
  baseTest(
    gen = genCommandExecEvt.map(_.asLeft),
    mkWriter = _ ⇒ failingWriter,
    expWrites = _ ⇒ None,
    expAcks = List.empty
  )
}
```

These few tests give us enormous guarantees, so we can fearlessly continue to build robust software on top of it.

7.1.4 Run

Ultimately, here is how we run the `snapshots` application.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (server, distLockRes, trConsumer, swConsumer, reader, fsm) ⇒
      Stream.eval(server.useForever).concurrently {
        Stream.resource(distLockRes).flatMap { distLock ⇒
          val ticks: Stream[IO, Engine.In] =
            Stream.fixedDelay[IO](2.seconds).evalMap(_ ⇒ distLock.refresh)

          Stream.eval(IO.deferred[Unit]).flatMap { gate ⇒
            def process(st: TradeState, trading: Stream[IO, Msg[TradeEvent]]) =
              trading
                .either(Stream.exec(gate.get) ++ swConsumer.receiveM)
                .merge(ticks)
                .evalMapAccumulate(st → List.empty)(fsm.run)

            Stream.eval(reader.latest).flatMap {
              case Some(st, id) ⇒
                process(st, trConsumer.rewind(id, gate))
              case None ⇒
                process(TradeState.empty, trConsumer.rewind(MsgId.earliest, gate))
            }
          }
        }
      }
    }
    .compile
    .drain
```

It runs the HTTP server while concurrently trying to acquire the distributed lock.

On success, it retrieves the latest snapshot and performs a synchronized rewind (similarly to the `alerts` service). Afterwards, it continues consuming both `TradeEvent`s and `SwitchEvent`s while running them through the FSM.

Additionally, we send `Tick`s every two seconds to trigger the persistence of snapshots, and use this very same tick to refresh the TTL (time-to-live) of the lock.

The idea of persisting snapshots every few seconds while performing batch acknowledgements enables a much higher throughput, though the two seconds value can always be adjusted after further monitoring and analysis.

Summarizing, we rewind the subscription to the given offset when we get a snapshot. Otherwise, we start from the very beginning and rebuild the state from scratch, as one would do in event-sourced applications.

7.2 Forecasts

At the beginning of Chapter 6, we briefly described what forecasting means.

It allows authors to self-register, so they can publish market forecasts' articles, which can then be anonymously rated up or down.

These actions can be modeled as three different messages of type `ForecastCommand`, as shown in the following section.

7.2.1 Commands

The `ForecastCommand` defines a few essential fields in the same way `TradeCommand` does.

```
enum ForecastCommand derives Codec.AsObject, Eq, Show:
  def id: CommandId
  def cid: CorrelationId
  def createdAt: Timestamp
```

Next, we have the three commands mentioned above as part of this enumeration.

```
case Register(
    authorName: AuthorName,
    authorWebsite: Option[Website]
)

case Publish(
    authorId: AuthorId,
    symbol: Symbol,
    description: ForecastDescription,
    tag: ForecastTag
)

case Vote(
    forecastId: ForecastId,
    result: VoteResult
)
```

For brevity, essential fields are omitted in this section.

Most new datatypes we see here are newtypes over `String` or `UUID`, except for `ForecastTag`, defined as another enumeration.

```
enum ForecastTag derives Codec.AsObject, Eq, Show:
  case Long, Short, Unknown
```

Same goes for `VoteResult`.

```
enum VoteResult derives Eq, Show:
  case Up, Down
```

These are probably self-explanatory, but here is a short description of each.

- `Register`: register a new author with a unique name and optional website.
- `Publish`: publish a new article for a specific symbol (only for registered authors).
- `Vote`: cast a vote to a specific article identified by forecast ID.

7.2.2 Events

Unlike `TradeCommand`s, where the only possible events that can be emitted from their processing are `TradeEvent`s, two different types of events can be produced from the processing of `ForecastCommand`s.

The first one is `ForecastEvent`, which makes the `forecastId` field mandatory.

```
enum ForecastEvent derives Codec.AsObject:
  def id: EventId
  def cid: CorrelationId
  def forecastId: ForecastId
  def createdAt: Timestamp
```

Next, we have two possible outcomes (omitting essential fields).

```
case Published(
    authorId: AuthorId,
    symbol: Symbol
)

case Voted(
    result: VoteResult
)
```

The second type is `AuthorEvent`, defined as follows.

```
enum AuthorEvent derives Codec.AsObject:
  def id: EventId
  def cid: CorrelationId
  def createdAt: Timestamp
```

Which can only be the following concrete event for now.

```
case Registered(
    authorId: AuthorId,
    authorName: AuthorName,
    authorWebsite: Option[Website]
)
```

Once again, skipping essential fields for conciseness.

7.2.3 Command-event relationship

Let's examine the relationship between commands and events to gain a better perspective of how the different outcomes relate to the inputs.

COMMAND	AUTHOR EVENT	FORECAST EVENT
Register	Registered	N/A
Publish	N/A	Published
Vote	N/A	Voted

The idea of splitting these two types of events stems from the potential necessity of allowing their processing to happen in different services.

In reality, however, there is no consumer of such events due to a lack of requirements for this functionality in this particular moment.

As we can observe, there is no state in this relationship table, as the forecasts' engine does not require a state machine. We will learn more about it next.

7.2.4 Engine

The main business logic of this service is not defined as a state machine.

```
trait Engine[F[_]]:
  def run: Msg[ForecastCommand] ⇒ F[Unit]
```

Instead, we directly process messages of type `ForecastCommand`s.

```
object Engine:
  def make[F[_]: GenUUID: Logger: MonadCancelThrow: Time](
      producer: Producer[F, ForecastEvent],
      atStore: AuthorStore[F],
      fcStore: ForecastStore[F],
      acker: Acker[F, ForecastCommand]
  ): Engine[F] = new:
    def run: Msg[ForecastCommand] ⇒ F[Unit] =
      case Msg(msgId, _, cmd) ⇒ ???
```

Its primary constructor requires a `ForecastEvent` producer, an `Acker` for `ForecastCommand`, and stores for `Author`s and `Forecast`s.

We are already acquainted with `Producer` and `Acker`, so let's dive into the remaining types.

7.2.4.1 Store

Let's start with `AuthorStore`, as there are a few datatypes we have not seen before.

```
trait AuthorStore[F[_]]:
  def fetch(id: AuthorId): F[Option[Author]]
  def tx: Resource[F, TxAuthorStore[F]]

trait TxAuthorStore[F[_]]:
  def save(author: Author): F[Unit]
  def outbox(evt: AuthorEvent): F[Unit]
```

The `Author` datatype may contain a set of `ForecastId`, among other information.

```
final case class Author(
    id: AuthorId,
    name: AuthorName,
    website: Option[Website],
    forecasts: Set[ForecastId]
) derives Codec.AsObject, Show
```

As we will learn shortly, the `save` and `outbox` operations need to run within a transaction that needs to be composed externally, so we define it under a different interface.

At the `Engine` level, we only deal with the store via its interface, so at this point, it is not very relevant to know whether the interpreter writes to a SQL or NoSQL database. Nevertheless, this is something we will explore in the next section.

Next comes `ForecastsStore`, where writing operations are also transactional.

```
trait ForecastStore[F[_]]:
  def fetch(fid: ForecastId): F[Option[Forecast]]
  def tx: Resource[F, TxForecastStore[F]]

trait TxForecastStore[F[_]]:
  def save(aid: AuthorId, fc: Forecast): F[Unit]
  def castVote(fid: ForecastId, res: VoteResult): F[Unit]
  def registerVote(evt: ForecastEvent.Voted): F[Unit]
  def outbox(evt: ForecastEvent): F[Unit]
```

A `Forecast` is always linked to a specific `Symbol`, which also contains a score calculated from the total number of votes.

```
final case class Forecast(
    id: ForecastId,
    symbol: Symbol,
    tag: ForecastTag,
    description: ForecastDescription,
    score: ForecastScore
) derives Codec.AsObject, Show
```

Moreover, we define a few custom error types so we can adapt the store interpreter's errors. To avoid stack traces, these are defined by extending `NoStackTrace`. E.g.

```
case object DuplicateAuthorError extends NoStackTrace
type DuplicateAuthorError = DuplicateAuthorError.type
```

These are all the new datatypes—time to move onto the command's processing.

7.2.4.2 Command handler

As the entry point of the engine, we have the following code.

```
def run: Msg[ForecastCommand] ⇒ F[Unit] =
  case Msg(msgId, _, cmd) ⇒
    val eid = EventId(cmd.id.value)
    Time[F].timestamp.flatMap { ts ⇒
```

```
      cmd match ???
    }
```

We pattern-match on the message to extract its ID and payload, then generate a `Timestamp` for the emission of events. Notice how we also reuse the `CommandId` to create an `EventId` to avoid duplicate events from being processed. This can only be done when there is a one-to-one relationship between commands and events.

Next, we pattern-match on the command, starting with `Register`.

```
case ForecastCommand.Register(_, cid, name, site, _) ⇒
  GenUUID[F].make[AuthorId].flatMap { aid ⇒
    atStore.tx
      .use { db ⇒
        db.save(Author(aid, name, site, Set.empty)) *>
          db.outbox(AuthorEvent.Registered(eid, cid, aid, name, site, ts))
      }
      .productR(acker.ack(msgId))
      .recoverWith { case DuplicateAuthorError ⇒
        Logger[F].error(s"Author name $name already registered!") *> acker.ack(msgId)
      }
      .handleNack
  }
```

We start by creating a unique `AuthorId`. Then, we try to persist the `Author` and the `AuthorEvent` within the same transaction. This architecture follows the outbox pattern we learned about in Chapter 2 (see Outbox pattern).

Persisting the author could raise three different errors (see interpreter further down), but we are only interested in handling the `DuplicateAuthorError`—meaning the username is already taken—as we always invoke it with an empty set of forecasts. In such cases, we handle it via `recoverWith` by logging the error message and acknowledging the message.

At last, we have the `handleNack` extension method.

```
extension (fa: F[Unit])
  def handleNack: F[Unit] =
    fa.handleErrorWith {
      case e: DuplicateEventId ⇒
        Logger[F].error(s"Ignoring $e") *> acker.ack(msgId)
      case e ⇒
        Logger[F].error(s"Unexpected: ${e.getMessage}") *> acker.nack(msgId)
    }
```

On `DuplicateEventId` (occurring when the event has already been processed in the outbox table), we log the error message and acknowledge the message.

On any other failure, we log the error and send a negative acknowledgement. We have this logic on an extension method because it is shared with the `Publish` command.

```
case ForecastCommand.Publish(_, cid, aid, symbol, desc, tag, _) ⇒
  GenUUID[F].make[ForecastId].flatMap { fid ⇒
    fcStore.tx
      .use { db ⇒
        db.save(aid, Forecast(fid, symbol, tag, desc, ForecastScore(0))) *>
          db.outbox(ForecastEvent.Published(eid, cid, aid, fid, symbol, ts))
      }
      .productR(acker.ack(msgId))
      .recoverWith { case AuthorNotFound ⇒
        Logger[F].error(s"Author not found: $aid") *> acker.ack(msgId)
      }
      .handleNack
  }
```

The procedure is very similar to the processing of the `Register` command, except we persist a `ForecastEvent` in the outbox table. The `save` operation could raise an `AuthorNotFound` error, which means the `AuthorId` we received in the command has not been yet registered.

Errors are also handled in a similar fashion via `recoverWith` and `handleNack`.

Ultimately, we reuse the `CommandId` as the `EventId` to handle the redelivery of commands in case the acknowledgement fails after the database transaction succeeds. Still, it would only present a problem to the `Publish` command, not to the `Register` command, where the `DuplicateAuthorError` will be triggered first.

7.2.4.3 Outbox handler

Both the `Register` and `Publish` commands persist an event into the outbox table. Upon consumption, these are then processed by the following handler.

```
trait OutboxHandler[F[_]]:
  def run: Msg[OutboxEvent] ⇒ F[Unit]
```

An `OutboxEvent` can either contain an `AuthorEvent` or a `ForecastEvent` in our case.

```
final case class OutboxEvent(
    event_id: EventId,
    correlation_id: CorrelationId,
    event: Either[AuthorEvent, ForecastEvent],
    created_at: Timestamp
) derives Codec.AsObject
```

Its main interpreter takes two producers (one for each event), and an `Acker` for the `OutboxEvent`.

```
object OutboxHandler:
  def make[F[_]: Applicative: Logger](
      authors: Producer[F, AuthorEvent],
      forecasts: Producer[F, ForecastEvent],
      acker: Acker[F, OutboxEvent]
  ): OutboxHandler[F] = new:
    def run: Msg[OutboxEvent] ⇒ F[Unit] =
      case Msg(msgId, _, OutboxEvent(_, _, ev, _)) ⇒
        ev.bitraverse(authors.send, forecasts.send) *> acker.ack(msgId)
```

On each incoming event, it extracts the inner event and publishes it, followed by acknowledging the message. This completes the cycle of the outbox pattern.

This is pretty much how it would look like using Debezium and its PostgreSQL-Pulsar connector. However, since we use the H2 database, such connector doesn't exist.

For this reason, in the reference application, you will find more code to make up for it. We will not get into details as it is not very relevant, but know it is using a database trigger[1] to react to changes and push events into an in-memory queue. Another process then repeatedly reads the queue and publishes `OutboxEvent`s to a Pulsar topic.

A full-fledged example using PostgreSQL and Pulsar can be found in the `PulsarCDC` demo application. Follow the README file instructions in the repository to learn more.

7.2.4.4 Votes handler

The last command is `Vote`. Its processing is fairly simple as far as the `Engine` goes.

```
case ForecastCommand.Vote(_, cid, fid, res, _) ⇒
  producer.send(ForecastEvent.Voted(eid, cid, fid, res, ts)) *>
    acker.ack(msgId)
```

Wait! What on Earth is happening here?

We use the "listen-to-yourself" pattern we learned about in Chapter 2 (see Change Data Capture). This means we have a consumer of `ForecastEvent`s running concurrently with the rest of the application, which are processed by the following handler.

```
trait VotesHandler[F[_]]:
  def run: Msg[ForecastEvent] ⇒ F[Unit]
```

Its main interpreter requires a `ForecastStore` and an `Acker` for `ForecastEvent`.

[1] http://h2database.com/html/features.html#triggers

```
object VotesHandler:
  def make[F[_]: Logger: MonadCancelThrow](
      store: ForecastStore[F],
      acker: Acker[F, ForecastEvent]
  ): VotesHandler[F] = new:
    def run: Msg[ForecastEvent] ⇒ F[Unit] = ???
```

The processing of the `Published` event is simple, as we are not interested in it (this is already handled using the outbox pattern directly in `Engine`).

```
case Msg(msgId, _, ForecastEvent.Published(_, _, _, _, _, _)) ⇒
  acker.ack(msgId)
```

Ultimately, we process the main `Voted` event as follows.

```
case Msg(msgId, _, evt @ ForecastEvent.Voted(_, _, fid, res, _)) ⇒
  store.tx
    .use { db ⇒
      db.registerVote(evt) *> db.castVote(fid, res)
    }
    .productR(acker.ack(msgId))
    .handleErrorWith {
      case DuplicateEventId(eid) ⇒
        Logger[F].error(s"Duplicate event ID: $eid") *> acker.ack(msgId)
      case e ⇒
        Logger[F].error(s"Vote registration error: $fid - ${e.getMessage}") *>
          acker.nack(msgId)
    }
```

We start a database transaction in which we cast the vote (updates the score in the forecasts table) and register the event (updates the votes table). The latter help us avoid duplicated votes, as the operation will fail if the `EventId` already exists.

If everything goes well or we get a `DuplicateEventId` error, we acknowledge the message. Otherwise, we log the error and send a negative acknowledgement.

7.2.4.5 Unit tests

All this logic is unit-tested under `EngineSuite`. Here is a sneak peek of the tests.

```
test("Successful author registration") {
  val out = AuthorEvent.Registered(
    eventId, cid, authorId, authorName, None, ts
  )
  baseTest(
```

```
    in = ForecastCommand.Register(id, cid, authorName, None, ts),
    ex1 = expect.same(_, Some(out)),
    ex2 = expect.same(_, None)
  )
}

test("Fail to register author (duplicate username)") {
  baseTest(
    mkAuthorStore = _ ⇒ failAuthorStore,
    in = ForecastCommand.Register(id, cid, authorName, None, ts),
    ex1 = expect.same(_, None),
    ex2 = expect.same(_, None)
  )
}
```

The `baseTest` method is defined as follows (omitting some details for brevity).

```
private def baseTest(
    mkAuthorStore: Ref[IO, Option[AuthorEvent]] ⇒ AuthorStore[IO],
    mkForecastStore: Ref[IO, Option[ForecastEvent]] ⇒ ForecastStore[IO],
    in: ForecastCommand,
    ex1: Option[AuthorEvent] ⇒ Expectations,
    ex2: Option[ForecastEvent] ⇒ Expectations
): IO[Expectations] = ???
```

Given a store and an input command, the provided assertions are run against the potential output event (there could be none). Since the events are not published directly (we use the outbox pattern), we use those `Ref`s to make up for it.

7.2.5 SQL store

As shown in the system diagram, both authors and forecasts are persisted in a relational SQL database.

For simplicity, we will be using the in-memory H2 database[2] to avoid yet another heavy service in our local stack. It suffices to say that you should use something like PostgreSQL in production.

As a client for the interpreters, we use the battle-tested Doobie[3] library, which supports several JDBC (Java Database Connectivity) engines.

[2] https://h2database.com/html/main.html
[3] https://github.com/tpolecat/doobie

If you go straight for PostgreSQL, another valid option is Skunk[4], which avoids the blocking nature of JDBC.

Without further ado, let's look at the implementation details from bottom to top.

7.2.5.1 Database connection

First of all, we create a connection and run a schema migration via Flyway[5].

```
object DB:
  private val uri = "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"

  def init[F[_]: Async]: Resource[F, H2Transactor[F]] =
    ExecutionContexts
      .fixedThreadPool[F](32)
      .flatMap { ce ⇒
        H2Transactor.newH2Transactor[F](uri, "sa", "", ce)
      }
      .evalTap {
        _.configure { ds ⇒
          Async[F].delay(
            Flyway.configure().dataSource(ds).load().migrate()
          )
        }
      }
```

Flyway looks for any versioned SQL files under `resource/db/migration`. We have the initial one named `V1__baseline.sql`, which defines the essential tables.

```
CREATE TABLE authors (
  id UUID PRIMARY KEY,
  name VARCHAR UNIQUE NOT NULL,
  website TEXT NULL
);

CREATE TABLE forecasts (
  id UUID PRIMARY KEY,
  symbol TEXT,
  tag TEXT,
  description TEXT,
  score INT DEFAULT 0
);
```

[4] https://github.com/tpolecat/skunk
[5] https://flywaydb.org/

```
CREATE TABLE author_forecasts (
  id UUID PRIMARY KEY,
  author_id UUID NOT NULL,
  CONSTRAINT author_key FOREIGN KEY (author_id) REFERENCES authors(id),
  CONSTRAINT forecast_key FOREIGN KEY (id) REFERENCES forecasts(id)
);
```

Two main tables and one for the *one-to-many* author-to-forecasts relationship.

Next, we have a file defining the `outbox` table used for author and forecast events.

```
CREATE TABLE outbox (
  event_id UUID PRIMARY KEY,
  correlation_id UUID NOT NULL,
  event TEXT NOT NULL,
  created_at DATETIME
);

CREATE TRIGGER h2_cdc
AFTER INSERT
ON outbox
FOR EACH ROW
CALL "trading.forecasts.cdc.H2OutboxTrigger";
```

Additionally, we define a trigger that invokes the `H2OutboxTrigger` class, filling the void left by the lack of a native H2-Pulsar-CDC connector.

Finally, the last migration file defines the `votes` table.

```
CREATE TABLE votes (
  event_id UUID PRIMARY KEY,
  fid UUID NOT NULL,
  result INT,
  created_at DATETIME,
  CONSTRAINT votes_forecast_key FOREIGN KEY (fid) REFERENCES forecasts(id)
);
```

It ensures no duplicate events are processed more than once while providing a way to audit the casting of votes.

7.2.5.2 Extension methods

Under the `trading.forecasts.store` package, we also have a few useful extensions that map specific database errors to business errors via `adaptError`.

```
extension [F[_]: MonadThrow, A](fa: F[A])
  /* duplicate key violates unique constraint */
  def onDuplicate(err: Throwable): F[A] =
    fa.adaptError {
      case e: SQLException if e.getSQLState == "23505" ⇒ err
    }

  /* referential integrity constraint violation */
  def onConstraintViolation(err: Throwable): F[A] =
    fa.adaptError {
      case e: SQLException if e.getSQLState == "23506" ⇒ err
    }

extension [F[_]: MonadThrow](fa: F[Int])
  /* for update-set statements */
  def onUpdateFailure(err: Throwable): F[Unit] =
    fa.flatMap {
      case 1 ⇒ ().pure[F]
      case _ ⇒ err.raiseError[F, Unit]
    }
```

We extensively use these methods in the main interpreters, as we will see soon.

7.2.5.3 Composable transactions

Thus far, we have learned that database writes are handled atomically via transactions, as the outbox pattern heavily relies on this feature.

Using transactions is never an easy decision to make, as not every system can afford an overall decrease in performance[6]. However, it guarantees correctness and fits this service given the current business requirements (or lack thereof).

Unlike Skunk's transactions[7], Doobie's transactions are not modeled as a `Resource`, making it a bit awkward to combine multiple operations externally from an interface. For this is the reason, we have a little helper behind the following abstraction.

```
trait DoobieTx[F[_]]:
  def transaction(xa: Transactor[F]): Resource[F, ConnectionIO ⇝ F]

extension [F[_]: DoobieTx](xa: Transactor[F])
  def transaction: Resource[F, ConnectionIO ⇝ F] =
    DoobieTx[F].transaction(xa)
```

[6] https://www.dbta.com/Columns/DBA-Corner/The-5-Key-Factors-for-Database-Performance-134830.aspx

[7] https://tpolecat.github.io/skunk/tutorial/Transactions.html

Implementation details are not that relevant here (you can refer to the accompanying source code), but notice the return type of the `transaction` method.

We get a resource of a natural transformation `ConnectionIO ~> F` that can be used in the interpreters to convert a SQL statement in `ConnectionIO` to our application effect `F`.

Running other I/O operations within a Doobie or Skunk transaction is not recommended, as we become responsible for any unrelated operations going wrong.

7.2.5.4 SQL queries and statements

In the `SQL` object, we have two things. Firstly, the required Doobie typeclass instances.

```
object SQL:
  given Meta[UUID] = Meta[String].imap[UUID](UUID.fromString)(_.toString)

  given Read[Author] = Read[(UUID, String, Option[String], Option[UUID])].map {
    (id, name, website, fid) ⇒
      Author(
        AuthorId(id),
        AuthorName(name),
        website.map(Website(_)),
        fid.toSet.map(ForecastId(_))
      )
  }

  given Read[Forecast] = Read[(UUID, String, String, String, Int)].map {
    (id, sl, tag, desc, sc) ⇒
      Forecast(
        ForecastId(id),
        Symbol.unsafeFrom(sl),
        ForecastTag.from(tag),
        ForecastDescription(desc),
        ForecastScore(sc)
      )
  }
```

We only define `Read` instances, and instead of defining `Write` instances, we determine how to write directly in the SQL statements.

As an example, here are the SQL queries and statements for `Author`.

```
/* ---------------------- authors table ----------------------- */
val selectAuthor: AuthorId ⇒ Query0[Author] = id ⇒ sql"""
    SELECT a.id, a.name, a.website, f.id FROM authors AS a
    LEFT JOIN author_forecasts AS f ON a.id=f.author_id
    WHERE a.id=${id.show}
  """.query[Author]

val insertAuthor: Author ⇒ Update0 = a ⇒ sql"""
    INSERT INTO authors (id, name, website)
    VALUES (${a.id.value}, ${a.name.value}, ${a.website.map(_.value)})
  """.update
```

In the `insertAuthor` statement, we manually indicate how the `Author` entity should be persisted, but it could also be done via the `Writer` typeclass.

7.2.5.5 Author store interpreter

Let's now see how the `AuthorStore` is implemented, starting with its main constructor.

```
object AuthorStore:
  def from[F[_]: DoobieTx: MonadCancelThrow](
      xa: Transactor[F]
  ): AuthorStore[F] = new: ???
```

It requires a single parameter of type `Transactor[F]`, which will be `H2Transactor` in our case, but it could be any of the supported database engines. We also leverage the `DoobieTx` capability trait for custom transactional support.

Next, we have the implementation of `fetch`.

```
def fetch(id: AuthorId): F[Option[Author]] =
  SQL.selectAuthor(id).accumulate[List].transact(xa).map {
    case Nil       ⇒ None
    case (x :: xs) ⇒ x.copy(
      forecasts = x.forecasts.union(xs.toSet.flatMap(_.forecasts))
    ).some
  }
```

It accumulates all the potential results in a list followed by analyzing the transaction result. If the list is non-empty, we take the head and proceed with merging all the forecasts from the tail in a single set via `union`, as the query is defined via `LEFT JOIN`.

The `save` method could be implemented as follows if we ignore the forecasts set (`Set[ForecastId]`) and the transactional extensibility for a moment.

```
def save(author: Author): F[Unit] =
  val saveAuthor =
    SQL
      .insertAuthor(author)
      .run
      .onDuplicate(DuplicateAuthorError)
      .transact(xa)
```

Leveraging the `onDuplicate` extension method, we raise a `DuplicateAuthorError` when the username is already taken, followed by executing the statement via `transact(xa)`.

Conversely, if we consider the set of forecasts, we need an extra statement that runs as part of the same transaction. In this case, we can perform a batch insert as well.

```
def save(author: Author): F[Unit] =
  val saveAuthor =
    SQL
      .insertAuthor(author)
      .run
      .onDuplicate(DuplicateAuthorError)

  val saveForecasts =
    SQL
      .insertAuthorForecasts(author)
      .whenA(author.forecasts.nonEmpty)
      .onDuplicate(DuplicateForecastError)
      .onConstraintViolation(ForecastNotFound)

  (saveAuthor *> saveForecasts).transact(xa)
```

The `insertAuthorForecasts` is powered by the underlying `ConnectionIO` and the `updateMany` function, as shown in the following code snippet.

```
def insertAuthorForecasts(a: Author): ConnectionIO[Int] =
  val sql = "INSERT INTO author_forecasts (id, author_id) VALUES (?, ?)"
  val ids = a.forecasts.toList.map(_.value → a.id.value)
  Update[(UUID, UUID)](sql).updateMany(ids)
```

We try to persist the set of forecast IDs only when the set is non-empty. Because the given forecast could either be assigned to another author or not exist, we map both cases to specific business errors.

Both the persistence of the author and the forecasts are performed in a single transaction. Therefore, everything can be rolled back if something goes wrong.

This is how most interpreters are generally implemented. However, it is not possible to combine Doobie's transactions from interface calls without the `DoobieTx` extension. For this reason, we introduced an explicit transactional interface: `TxAuthorStore`.

```
def tx: Resource[F, TxAuthorStore[F]] =
  xa.transaction.map(transactional)
```

We have a private constructor to build the transactional store instance.

```
private def transactional[F[_]: MonadCancelThrow](
    fk: ConnectionIO ~> F
): TxAuthorStore[F] = new: ???
```

The first implemented method is `outbox`.

```
def outbox(evt: AuthorEvent): F[Unit] = fk {
  SQL
    .insertOutbox(evt.asLeft)
    .run.void
    .onConstraintViolation(DuplicateEventId(evt.id))
}
```

We raise a `DuplicateEventId` error via the `onConstraintViolation` extension method whenever the event has already been processed.

Next comes the final implementation of the `save` method.

```
def save(author: Author): F[Unit] =
  val saveAuthor =
    SQL
      .insertAuthor(author)
      .run
      .onDuplicate(DuplicateAuthorError)

  val saveForecasts =
    SQL
      .insertAuthorForecasts(author)
      .whenA(author.forecasts.nonEmpty)
      .onDuplicate(DuplicateForecastError)
      .onConstraintViolation(ForecastNotFound)

  fk(saveAuthor *> saveForecasts)
```

Instead of running both SQL operations via `transact(xa)`, we do it via a natural transformation so that the calls to `outbox` and `save` can be externally composed while remaining transactional.

7.2.5.6 Forecasts store interpreter

The `ForecastsStore` has an identical constructor to the `AuthorStore`, so we skip it and get straight into the `fetch` method.

```
def fetch(fid: ForecastId): F[Option[Forecast]] =
  SQL.selectForecast(fid).option.transact(xa)
```

There are no joins in this query, making it very straightforward. Next, we have the `tx` method, built exactly as the one from `AuthorStore`.

```
def tx: Resource[F, TxForecastStore[F]] =
  xa.transaction.map(transactional)
```

Within the `TxForecastStore` implementation, we have a few methods.

```
def outbox(evt: ForecastEvent): F[Unit] = fk {
  SQL
    .insertOutbox(evt.asRight)
    .run.void
    .onConstraintViolation(DuplicateEventId(evt.id))
}
```

The `outbox` method inserts the outbox event containing the `ForecastEvent`, which may result in a `DuplicateEventId` error.

Next, we have the `castVote` method.

```
def castVote(fid: ForecastId, res: VoteResult): F[Unit] = fk {
  SQL
    .updateVote(fid, res)
    .run
    .onUpdateFailure(ForecastNotFound)
}
```

It involves a single update statement defined as follows.

```
def updateVote(id: ForecastId, res: VoteResult): Update0 =
  sql"""
    UPDATE forecasts
    SET score=COALESCE(score, 0)+${res.asInt}
    WHERE id=${id.show}
  """.update
```

The `onUpdateFailure` extension method operates on the `Int` value returned by the execution of such statement, which we adapt to a `ForecastNotFound` error.

That `res.asInt` extension method used in the update statement is defined as follows.

```
extension (res: VoteResult)
  def asInt: Int = res match
    case VoteResult.Up   ⇒ 1
    case VoteResult.Down ⇒ -1
```

As we have seen with the votes handler, the `castVote` operation runs in the same transaction as `registerVote`, defined as shown below.

```
def registerVote(evt: ForecastEvent.Voted): F[Unit] = fk {
  SQL
   .insertVote(evt)
   .run.void
   .onConstraintViolation(DuplicateEventId(evt.id))
}
```

Lastly, we have the `save` method, consisting of two different statements.

```
def save(aid: AuthorId, fc: Forecast): F[Unit] =
  val saveForecast =
    SQL.insertForecast(fc).run

  val saveRelationship =
    SQL
      .updateAuthorForecast(aid, fc.id)
      .run.void
      .onConstraintViolation(AuthorNotFound)

  fk(saveForecast *> saveRelationship)
```

We begin by persisting the forecast, and if everything goes well, we proceed with the persistence of the author-forecast relationship, all in a single transaction.

Note that the latter can fail if the given `AuthorId` does not exist, so we adapt the error.

The `EngineSuite` covers all the cases where either one event or another should be produced, depending on whether the set of operations was successful. Though, this only tests with an in-memory representations of both stores.

It is always recommendable to test the SQL queries and statements and see whether there are any database-specific issues.

We will learn how to achieve exactly this in the last section of this chapter (see Integration tests further down).

7.2.6 Scalability

Similarly to the `snapshots` service, we also require a single writer instance (even though we could rely on database transactions to coordinate writes, but this could make them more expensive due to potential conflicts with other writers).

For this purpose, we use a fail-over subscription: a perfect match for a single-writer stateless application.

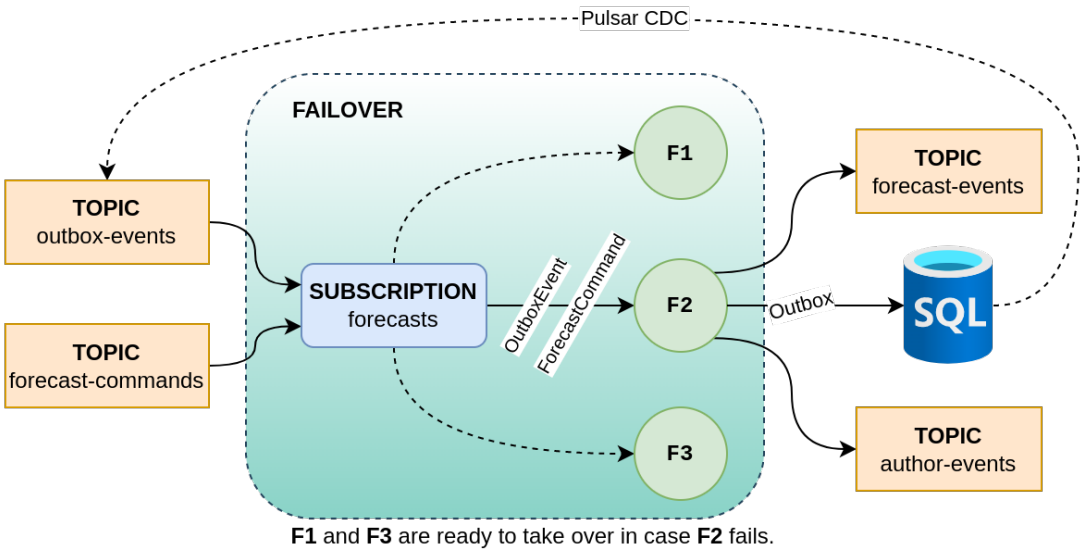


Figure 7.2.1: Forecasts scalability

Figure 7.2.1 shows the elected leader `F2` processing `ForecastCommand`s while writing data to a SQL database following the outbox pattern. It also processes incoming `OutboxEvent`s and publishes events to the corresponding Pulsar topics. The other two instances, `F1` and `F3`, are ready to take over in case something goes wrong with `F2`.

This service is not expected to receive big traffic, so we write to the database on every command we receive. In any other case, we could consider any of the data approaches discussed in Chapter 5 (see Data pipelines).

7.2.7 Entry point

Like many other services, we start off with a subscription type.

```
val sub =
  Subscription.Builder
    .withName("forecasts")
    .withType(Subscription.Type.Failover)
    .build
```

Followed by the producer settings, which needs a unique producer name to correctly perform deduplication after a restart (see Deduplication).

```
def settings[A](name: String) =
  PulsarProducer
    .Settings[IO, A]()
    .withDeduplication
    .withName(s"fc-$name-event")
    .some
```

Since we use the H2 database and there is no Pulsar CDC connector for it, we need to manually handle that the publishing of `OutboxEvent`s.

```
def cdcResources(
    pulsar: Pulsar.T,
    topic: Topic.Single
): Resource[IO, Stream[IO, Unit]] =
  Producer.pulsar[IO, OutboxEvent](
    pulsar, topic, settings("outbox")
  ).map(p ⇒ OutboxProducer.make(p).stream)
```

The `OutboxProducer` reads from the internal queue where the H2 database trigger publishes events to and publishes to the `outbox-events` topic. This is the kind of functionality we would get out of the box with any Pulsar CDC connector.

Next, we have the usual sequence of resources.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url, Pulsar.Settings().withTransactions)
    _      <- Resource.eval(Logger[IO].info("Initializing forecasts service"))
    cmdTopic = AppTopic.ForecastCommands.make(config.pulsar)
    atTopic  = AppTopic.AuthorEvents.make(config.pulsar)
    fcTopic  = AppTopic.ForecastEvents.make(config.pulsar)
    outTopic = AppTopic.OutboxEvents.make(config.pulsar)
```

```
    authors     <- Producer.pulsar[IO, AuthorEvent](pulsar, atTopic, settings("at"))
    forecasts   <- Producer.pulsar[IO, ForecastEvent](pulsar, fcTopic, settings("fc"))
    cmdConsumer <- Consumer.pulsar[IO, ForecastCommand](pulsar, cmdTopic, sub)
    evtConsumer <- Consumer.pulsar[IO, ForecastEvent](pulsar, fcTopic, sub)
    outConsumer <- Consumer.pulsar[IO, OutboxEvent](pulsar, outTopic, sub)
    cdcEvents   <- cdcResources(pulsar, outTopic)
    xa          <- DB.init[IO]
    atStore = AuthorStore.from(xa)
    fcStore = ForecastStore.from(xa)
    server  = Ember.default[IO](config.httpPort)
    engine  = Engine.make(forecasts, atStore, fcStore, cmdConsumer)
    handler = VotesHandler.make(fcStore, evtConsumer)
    outbox  = OutboxHandler.make(authors, forecasts, outConsumer)
  yield (
    server, cmdConsumer, evtConsumer, outConsumer, handler, outbox, cdcEvents, engine
  )
```

As previously discussed, we have a single consumer of commands and two producers for the different types of events—additionally, a database connection for the stores.

Furthermore, we have a consumer of `OutboxEvent`s (for the Outbox handler) and one consumer of `ForecastEvent`s (for the Votes handler).

7.2.8 Run

There is no need for initializing any state in this service, so we end up with a simplified implementation of the `run` method.

```
def run: IO[Unit] =
  Stream
    .resource(resources)
    .flatMap { (
       server, cmdConsumer, evtConsumer, outConsumer, handler, outbox, cdc, engine
    ) ⇒
      Stream.eval(server.useForever).concurrently {
        Stream(
          cmdConsumer.receiveM.evalMap(engine.run),  // ForecastCommand
          outConsumer.receiveM.evalMap(outbox.run),  // OutboxEvent
          evtConsumer.receiveM.evalMap(handler.run), // Voted events (no cdc)
          cdc                                        // CDC connector
        ).parJoin(5)
      }
    }
    .compile
    .drain
```

It runs the HTTP server in the background while consuming forecasting commands that are run through the engine. Running concurrently, we have the consumer of events that are run through the corresponding handlers (see Command handler and Votes handler), and a custom CDC producer for the H2 database.

7.3 Feed

The `feed` service generates random trading and forecasting commands, and publishes them to Pulsar to trigger the response of the entire system.

Its single purpose is intended for manual testing, mainly due to our system's lack of actual input commands until it is integrated with an external system.

Implementation details are not that relevant, so we will only look at the engine's main constructor, defined as follows.

```
object Feed:
  def random[F[_]: GenUUID: Logger: Parallel: Temporal: Time](
      trProducer: Producer[F, TradeCommand],
      switcher: Producer[F, SwitchCommand],
      fcProducer: Producer[F, ForecastCommand]
  ): F[Unit] = ???
```

It takes a producer for each command type, simulating the emission of random data via Scalacheck generators.

However, random forecast events don't represent a realistic flow of commands and events, so we have a `ForecastFeed` for this purpose. It guarantees that `Publish` commands are only sent with valid `AuthorId`s by listening to `Registered` events. Same goes for the `Vote` command, ensuring an existing `ForecastId` by consuming `Published` events.

```
object ForecastFeed:
  def stream(
      fp: Producer[IO, ForecastCommand],
      fc: Consumer[IO, ForecastEvent],
      ac: Consumer[IO, AuthorEvent]
  ): Stream[IO, Unit] = ???
```

By default, this is used in the application instead of the fully random one.

7.3.1 Generators

These are the main generators for `TradeCommand` and `SwitchCommand`.

```
val tradeCommandGen: Gen[TradeCommand] =
  Gen.oneOf(createCommandGen, updateCommandGen, deleteCommandGen)

val switchCommandGen: Gen[SwitchCommand] =
  Gen.oneOf(startCommandGen, stopCommandGen)
```

```
def tradeCommandListGen: List[TradeCommand | SwitchCommand] =
  val gen = Gen.frequency[TradeCommand | SwitchCommand](
    2 → switchCommandGen,
    9 → tradeCommandGen
  )
  Gen.listOfN(10, gen).sample.toList.flatten
```

Whereas these are the ones for `ForecastCommand`s.

```
val forecastCommandGen: Gen[ForecastCommand] =
  Gen.oneOf(publishCommandGen, registerCommandGen, voteCommandGen)

val forecastCommandListGen: List[ForecastCommand] =
  Gen.listOfN(10, forecastCommandGen).sample.toList.flatten
```

We prioritize the CRUD commands over the start-stop commands on the former for fair randomization.

Since this service requires access to the generators defined under the `domain`'s test module, the module declaration requires the following settings.

```
lazy val feed = (project in file("modules/feed"))
  .settings(commonSettings: _*)
  .settings(
    libraryDependencies += Libraries.scalacheck
  )
  .dependsOn(core)
  .dependsOn(domain.jvm % "compile→compile;compile→test")
```

That last `compile→test` configuration does the trick. Unfortunately, this means the `feed` service cannot be deployed as a Docker container without resorting to hacks. Still, this should not be an issue, given the nature of this service.

We will learn more about containers and deployment in the last chapter (see Build & run), but have in mind this service is intended to run via `sbt run` directly.

7.3.2 Run

Initially, we would only need producers. However, the `ForecastFeed` needs to listen to `AuthorEvent`s and `ForecastEvent`s in order to provide the aforementioned guarantees.

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url)
    _      <- Resource.eval(Logger[IO].info("Initializing feed service"))
    trTopic   = AppTopic.TradingCommands.make(config.pulsar)
    swTopic   = AppTopic.SwitchCommands.make(config.pulsar)
    fcTopic   = AppTopic.ForecastCommands.make(config.pulsar)
    atEvTopic = AppTopic.AuthorEvents.make(config.pulsar)
    fcEvTopic = AppTopic.ForecastEvents.make(config.pulsar)
    trading  <- Producer.pulsar[IO, TradeCommand](pulsar, trTopic, settings("trade"))
    switcher <- Producer.pulsar[IO, SwitchCommand](pulsar, swTopic, swSettings)
    forecast <- Producer.pulsar[IO, ForecastCommand](pulsar, fcTopic, settings("fc"))
    fcConsumer <- Consumer.pulsar[IO, ForecastEvent](pulsar, fcEvTopic, sub)
    atConsumer <- Consumer.pulsar[IO, AuthorEvent](pulsar, atEvTopic, sub)
    fcFeed = ForecastFeed.stream(forecast, fcConsumer, atConsumer)
    server = Ember.default[IO](config.httpPort)
  yield (server, Feed.random(trading, switcher, forecast), fcFeed)
```

Furthermore, every producer requires different settings.

For `TradeCommand` and `ForecastCommand`, we need to enable deduplication and set the `orderingKey` of Pulsar messages, which we achieve via the `Shard` typeclass we learned about in Chapter 5 (see Neutron Producer). Moreover, we need to set a unique producer name, so it can correctly perform deduplication after a restart.

```
def settings[A: Shard](name: String) =
  PulsarProducer
    .Settings[IO, A]()
    .withDeduplication
    .withName(s"feed-$name-command")
    .withShardKey(Shard[A].key)
    .some
```

We also enable deduplication for `SwitchCommand`. However, we set a partition key via the `Compaction` typeclass (used for compacted topics) instead.

```
val swSettings =
  PulsarProducer
    .Settings[IO, SwitchCommand]()
```

```
    .withDeduplication
    .withMessageKey(Compaction[SwitchCommand].key)
    .withName("feed-switch-command")
    .some
```

The consumer subscriptions are quite straightforward.

```
val sub =
  Subscription.Builder
    .withName("forecasts-gen")
    .withType(Subscription.Type.Exclusive)
    .build
```

Ultimately, The `run` method is defined as follows.

```
def run: IO[Unit] =
  resources.use { (server, feed, fcFeed) ⇒
    Stream
      .eval(server.useForever)
      .concurrently {
        Stream(
          Stream.eval(feed),
          fcFeed
        ).parJoin(2)
      }
      .compile
      .drain
  }
```

It runs the HTTP server while concurrently running the trading and forecasting feeds.

7.4 Integration tests

As integration tests, we will consider all the interpreters that access a data store. In our case, Redis and a SQL database.

We will not be testing the interaction with Apache Pulsar, as there is not much value in doing that, unless we run an automated simulation of the entire system.

The latter could be a good idea. However, such simulations are generally tricky to get right and become even more challenging and expensive to maintain over time.

Nonetheless, we will soon learn about a technique that verifies communication integrity across multiple services right before deployment (See Smoke tests in Chapter 8).

7.4.1 Redis suite

The `RedisSuite` starts with defining a Redis connection.

```
object RedisSuite extends ResourceSuite:
  type Res = RedisCommands[IO, String, String]

  override def sharedResource: Resource[IO, Res] =
    Redis[IO].utf8("redis://localhost").beforeAll(_.flushAll)
```

Next we have the snapshots interpreters to test.

```
test("snapshots reader and writer") { redis ⇒
  val reader = SnapshotReader.from[IO](redis)
  val writer = SnapshotWriter.from[IO](redis, KeyExpiration(30.seconds))
  val msgId  = MsgId.earliest
  NonEmptyList
    .of(tradeStateGen.sample.replicateA(3).toList.flatten.last)
    .traverse { ts ⇒
      for
        x <- reader.latest
        _ <- writer.save(ts, msgId)
        y <- reader.latest
      yield NonEmptyList.of(
        expect.same(None, x),
        expect.same(Some(ts.status), y.map(_._1.status)),
        expect.same(Some(ts.prices.keySet), y.map(_._1.prices.keySet)),
        expect.same(Some(msgId.serialize), y.map(_._2.serialize))
      )
    }.map(_.flatten.reduce)
}
```

A simple implementation that tests the reading and writing parts.

Notice that we do not use property-based tests here because of the stateful nature of writing to Redis. If we want to write deterministic assertions, it suffices to do it only once, so we instead take a sample of the `TradeState` generator and select the last entry.

7.4.2 SQL suite

We also have a `SQLSuite` that checks all the SQL queries and statements in a single test that could as well be considered a unit test, as we use the H2 in-memory database.

However, we could update the JDBC connection to something like PostgreSQL, making this the right place for it.

Here's a sneak peek of the test suite you can find in the `trading` repository, starting with the definition of the connection and some initial values.

```
object SQLSuite extends IOSuite:
  type Res = H2Transactor[IO]

  override def sharedResource: Resource[IO, Res] =
    DB.init[IO]

  val aid  = AuthorId(UUID.randomUUID())
  val fid  = ForecastId(UUID.randomUUID())
```

Followed by a test that triggers all the SQL queries and statements invoked in both stores (not all of them shown here).

```
test("authors and forecasts") { xa ⇒
  val at = AuthorStore.from(xa)
  val fc = ForecastStore.from(xa)

  for
    a <- at.tx.use(_.save(Author(aid, AuthorName("gvolpe"), None, Set(fid)))).attempt
    _ <- at.tx.use(_.save(Author(aid, AuthorName("gvolpe"), None, Set())))
    b <- fc.tx.use(_.save(aid2, Forecast(fid, symbol, tag, desc, ForecastScore(1)))).attempt
    _ <- fc.tx.use(_.save(aid, Forecast(fid, symbol, tag, desc, ForecastScore(1))))
    c <- at.fetch(aid)
    d <- at.tx.use(_.save(Author(aid2, AuthorName("gvolpe"), None, Set()))).attempt
  yield NonEmptyList
    .of(
      expect.same(a, Left(ForecastNotFound)),
      expect.same(b, Left(AuthorNotFound)),
      expect.same(c.map(_.id), Some(aid)),
```

```
      expect.same(d, Left(DuplicateAuthorError))
    )
    .reduce
}
```

Notice that running this test also runs a Flyway migration, so it validates not only our SQL queries and statements but also our SQL schemas. Furthermore, the codebase contains more tests for the other methods that don't appear on this example.

7.5 Summary

We have learned about `snapshots` and `forecasts`, which might not be considered part of the core services, but these play their role in the system.

With the former, we learned how to correctly leverage the distributed lock technique to synchronize multiple consumers attached to an exclusive subscription, effectively replacing the `Failover` subscription feature with extra guarantees.

With the latter, we have seen a stateless approach to command processing that fits the `Failover` subscription, as well as stores that ultimately interact with a SQL database.

Furthermore, we learned to apply the transactional outbox and listen-to-yourself patterns, and how these help guarantee correctness and strong consistency.

We also learned about Scalacheck generators, used both for testing and random data generation in the `feed` service.

At last, we concluded by talking about the role of integration tests in our system.

8 Trading system (observability)

Observability is an integral part of any distributed system.

This chapter will teach us to run the entire system locally, manage the remote continuous integration build, and get ready to deploy to a production environment.

Before we do so, let's unravel the lengthy details of the last remaining service in our stack: the *tracing* service.

For those who wish to dive deeper into this extensive topic, I recommend checking out Observability Engineering[1] by Charity Majors, Liz Fong-Jones, and George Miranda.

[1] https://www.oreilly.com/library/view/observability-engineering/9781492076438/

8.1 Tracing

Among the Scala libraries compliant with Open Tracing[2], both Natchez[3] and Trace4cats[4] integrate perfectly with the Typelevel ecosystem, with a wide offering of out-of-the-box integrations such as Http4s, Honeycomb, Jaeger, etc.

We will be using Natchez in the following examples, but the concepts translate to any other tracing library.

The usual tracing approach involves threading spans (aka the context) throughout the application we wish to instrument. On the other hand, distributed tracing requires a so-called *kernel* to be able to continue the previous tracing span.

However, I find this approach quite invasive. We need to thread context throughout the entire application–generally done via MTL or directly via `Kleisli`—which may be hard to justify only for the tracing feature.

In Cats Effect 3, we can get a `Trace[IO]` instance given an initial `Span` (implemented via `IOLocal`). Still, besides only working for a concrete `IO`, it shares the same limitations of Cats MTL's Local[5] and Kleisli#local[6]. E.g.

```
def traceMe[F[_]: Monad: Trace]: F[String] =
  Trace[F].span("test-1") {
    Trace[F].put("tracing-msg" → "Hello!").as("Hello world!")
  }

def doTrace(ep: EntryPoint[IO]): IO[Unit] =
  ep.root("root").use { sp ⇒
    for
      given Trace[IO] <- Trace.ioTrace(sp)
      msg             <- traceMe[IO]
      _               <- IO.println(msg)
    yield ()
  }
```

Otherwise, we are left to choose between the existing `Trace` instances for datatypes capable of carrying context, such as `Kleisli` and `StateT`.

[2]https://opentracing.io/
[3]https://github.com/tpolecat/natchez
[4]https://github.com/trace4cats
[5]https://typelevel.org/cats-mtl/mtl-classes/local.html
[6]https://typelevel.org/cats/datatypes/kleisli.html

8.1.0.1 Open Telemetry

Notice that Open Tracing is slowly being replaced by Open Telemetry[7], which is the result of the merge with Open Census[8], a set of libraries for various languages focused on collecting metrics and distributed traces.

There is an ongoing effort to support it in the Typelevel ecosystem: otel4s[9].

That said, current Open Tracing and Open Census applications should continue to work with Open Telemetry, remaining relevant.

[7] https://opentelemetry.io/
[8] https://opencensus.io/
[9] https://github.com/typelevel/otel4s/

8.1.1 Distributed

In distributed systems, we usually need to continue a trace after consuming a message or receiving an HTTP request (with a kernel in the HTTP headers).

In such cases, the `Trace[IO]` instance runs short in tagless applications, as it can only be constructed at the top level, limited to the initial span.

The only way to resume a trace from a kernel is by having access to a shared `EntryPoint`, as shown in the following code snippet.

```
def continueTrace[F[_]: Console: MonadCancelThrow](
    ep: EntryPoint[F],
    kernel: Kernel
): F[Unit] =
  ep.continue("test-1", kernel).use { sp1 ⇒
    sp1.span("test-2").use { sp2 ⇒
      sp2.put("tracing-msg" → "Continuation") *>
        Console[F].println("Done with tracing")
    }
  }
```

The `EntryPoint` is only needed where a trace is expected to start from root or resume a previous one, so this should not be that much of a problem.

8.1.1.1 HTTP traces

The natchez-http4s[10] library allows us to trace HTTP requests down to the bottom layers by reading the HTTP headers and continuing a previous trace whenever possible. Otherwise, it starts a new root span.

Here we have a simple HTTP routes with a `Trace` constraint.

```
final class RoutesOne[F[_]: Monad: Trace] extends Http4sDsl[F]:

  val routes: HttpRoutes[F] = HttpRoutes.of {
    case GET → Root / "v1" / "health" ⇒
      Trace[F].span("http-health") {
        Trace[F].put("foo" → "bar") *> Ok()
      }
```

As we use `natchez-http4s`, we do not need to share the `EntryPoint`. Instead, we can lift the routes via the given syntax by instantiating it at the top level.

[10] https://github.com/tpolecat/natchez-http4s

```
import natchez.http4s.syntax.entrypoint.*

def run: IO[Unit] =
  Honeycomb.entryPoint[IO]("app")(...).flatMap { ep ⇒
    val routes = ep.liftT(RoutesOne[IO].routes)
    val server = Ember.routes[IO](port"9000", routes)
    server.useForever
  }
```

This is very convenient—and the most straightforward example you will find everywhere. However, what if our HTTP routes class needs access to other programs? E.g.

```
final class RoutesTwo[F[_]: GenUUID: Monad: Trace](
    users: UsersDB[F]
) extends Http4sDsl[F]:

  val routes: HttpRoutes[F] = HttpRoutes.of {
    case GET → Root / "v1" / "users" / UUIDVar(id) ⇒
      Trace[F].span("http") {
        Trace[F].put("get-user" → id.toString) *>
          users.get(id).flatMap {
            case Some(u) ⇒
              Trace[F].put("ok" → u.name) *> Ok(u.name)
            case None ⇒
              Trace[F].put("not-found" → id.toString) *> NotFound()
          }
      }

    case POST → Root / "v1" / "users" / name ⇒ ???
```

Where `UsersDB` defines the following two methods.

```
trait UsersDB[F[_]]:
  def get(id: UUID): F[Option[User]]
  def save(user: User): F[Either[DuplicateUser, Unit]]
```

We need an instance of `UsersDB` to construct our HTTP routes, so we need to decide whether we also want to trace the bottom layers or not.

With a single dependency, it is usually enough to trace the HTTP routes, for which we can use the following constructor.

```
object UsersDB:
  def noTrace[F[_]: MonadThrow: Ref.Make]: F[UsersDB[F]] = ???
```

Thus, lifting our HTTP routes is still pretty much the same.

```
def run: IO[Unit] =
  Honeycomb.entryPoint[IO]("app")(...).flatMap { ep ⇒
    UsersDB.noTrace[IO].flatMap { db ⇒
      val routes = ep.liftT(RoutesTwo(db).routes)
      val server = Ember.routes[IO](port"9000", routes)
      server.useForever
    }
  }
```

What if we wish to trace the `UsersDB` interpreter too? We have two options. The first one implies using a single effect type constructor.

```
object UsersDB:
  def make[F[_]: MonadThrow: Ref.Make: Trace]: F[UsersDB[F]] = ???
```

This means our `F` needs to be capable of carrying every HTTP request context. So we can no longer use `IO`.

```
type Eff = [A] ⟹» Kleisli[IO, Span[IO], A]

def run: IO[Unit] =
  Honeycomb.entryPoint[IO]("app")(...).flatMap { ep ⇒
    ep.root("demo-root").use { root ⇒
      UsersDB.make[Eff].flatMap { db ⇒
        val routes = ep.liftT(RoutesTwo(db).routes)
        val server = Ember.routes[IO](port"9000", routes)
        Kleisli.liftF(server.useForever)
      }.run(root)
    }
  }
```

If we created the `UsersDB` instance in `IO` via `Trace.ioTrace`, it would be stuck on a single span unrelated to the context of the current HTTP request.

Therefore, we use `Kleisli` both for our HTTP routes' class and its dependency. The disadvantage of this approach is that we still need to provide an initial span to get an `IO` back we can run—effectively eliminating the required context—which will never be reached (i.e. that last `run(root)`).

A cleaner alternative would be to separate the constructing effect `F` from the running effect `G`, which are inherently different.

```
object UsersDB:
  def alt[
    F[_]: MonadThrow: Ref.Make,
    G[_]: MonadThrow: Trace
  ]: F[UsersDB[G]] = ???
```

This allows us to avoid that last `run(root)`, keeping the two effects separately.

```
def run: IO[Unit] =
  Honeycomb.entryPoint[IO]("app")(...).flatMap { ep ⇒
    ep.root("demo-root").use { root ⇒
      UsersDB.alt[IO, Eff].flatMap { db ⇒
        val routes = ep.liftT(RoutesTwo(db).routes)
        val server = Ember.routes[IO](port"9000", routes)
        server.useForever
      }
    }
  }
```

We are now back to `HttpRoutes[IO]` instead of `HttpRoutes[Eff]`. However, the clear disadvantage of this approach is that the separation of effect types quickly spreads throughout the entire system—for as many layers as we have.

An advantage is that it allows us to combine both programs in `IO` and `Eff`, given we provide a way to go from `IO ~> Eff`—aka a *natural transformation*.

```
object UsersDB:
  def alt[
    F[_]: MonadThrow: Ref.Make,
    G[_]: MonadThrow: Trace
  ](using NT[F, G]): F[UsersDB[G]] = ???
```

In our examples, we use a custom `NT` typeclass to do so.

```
trait NT[F[_], G[_]]:
  def fk: F ~> G

object NT:
  def apply[F[_], G[_]](using nt: NT[F, G]): NT[F, G] = nt

  given NT[IO, Kleisli[IO, Span[IO], *]] = new:
    val fk = Kleisli.liftK

  object syntax:
    extension [F[_], G[_], A](using nt: NT[F, G])(fa: F[A])
      def liftK: G[A] = nt.fk(fa)
```

In addition to declaring a default `IO ~> Eff` instance, we define a `liftK` extension method to enhance the user experience. So using the `noTrace[F]` constructor as the base implementation, we can add tracing instrumentation on top of it.

```
object UsersDB:
  def alt[
    F[_]: MonadThrow: Ref.Make,
    G[_]: MonadThrow: Trace
  ](using NT[F, G]): F[UsersDB[G]] =
    noTrace[F].map { db ⇒
      new:
        def get(id: UUID): G[Option[User]] =
          Trace[G].span("users-db") {
            Trace[G].put("fetch" → id.toString) *>
              db.get(id).liftK
          }

        def save(user: User): G[Either[DuplicateUser, Unit]] =
          Trace[G].span("users-db") {
            db.save(user).liftK.flatTap {
              case Left(e) ⇒
                Trace[G].put("duplicate-error" → user.name)
              case Right(_) ⇒
                Trace[G].put("new-user" → user.name)
            }
          }
    }
```

Both `db.get` and `db.save` are in `F`, so we use `liftK` to transform them to `G`, which is the effect type used by the rest of the operations.

The downside is that it gets complex very quickly once we add more layers.

Say we have the following dependencies instead—displayed by fig. 8.1.1—where blue and orange boxes represent business logic and concrete interpreters, respectively.

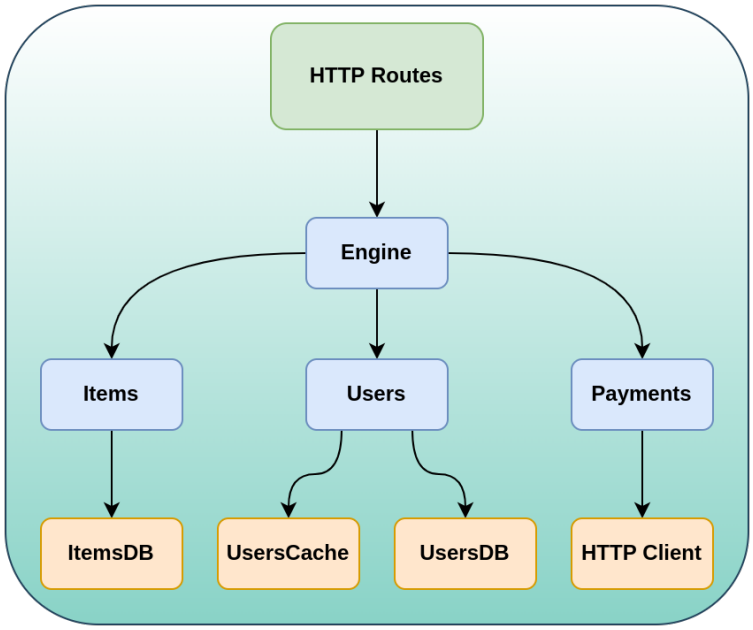


Figure 8.1.1: HTTP Routes dependencies

If we only wish to instrument the payments HTTP client, then also `Engine` needs to be instantiated in `Eff`. Fortunately, the other dependencies can still be in `IO`, though they need to be lifted into `Eff` within the engine implementation.

```
object Engine:
  def make[F[_]: Monad, G[_]: Monad: Trace](
      items: Items[F],
      users: Users[F],
      payments: Payments[G]
  )(using NT[F, G]): Engine[F] = ???
```

Suppose tomorrow we want to add tracing to the `ItemsDB` component. In that case, we would need to require `Items[G]` instead, and so on. Depending on the size of your dependency tree, this could be a bit painful to manage.

Unfortunately, it does not get any easier than this. On the bright side, you get distributed tracing for your critical components.

For completeness, here is the `noTrace` implementation.

```
def noTrace[F[_]: MonadThrow: Ref.Make]: F[UsersDB[F]] =
  (
    Ref.of[F, Map[UUID, User]](Map.empty),
    Ref.of[F, Map[String, UUID]](Map.empty)
  ).tupled.map { (users, idx) ⇒
```

```
    new:
      def get(id: UUID): F[Option[User]] =
        users.get.map(_.get(id))

      def save(user: User): F[Either[DuplicateUser, Unit]] =
        idx.get
          .map(_.get(user.name))
          .flatMap {
            case Some(_) ⇒
              DuplicateUser.raiseError
            case None ⇒
              users.update(_.updated(user.id, user)) *>
                idx.update(_.updated(user.name, user.id))
          }
          .attemptNarrow
  }
```

The `attemptNarrow` method has some remarkable properties. We can convert any `F[A]` into an `F[Either[E, A]]`, where we decide what `E` is, as long as it is a subtype of `Throwable`—i.e. what error types we wish to expose in the method's type signature.

Type inference will always yield `F[Either[Throwable, Unit]]` in such cases. Thus, it is important for us to be explicit about the expected error types.

This error modeling and handling technique are detailed in this blog post[11] I wrote in early 2022. It further describes how it is possible to achieve the same without using `Either`, all by leveraging union types.

Initially, the `forecasts` service was also modeled with explicit error types. However, the need for extended transactional support didn't make this possible, as errors must be propagated for a transaction to fail, and `attempt`/`attemptNarrow` do the opposite.

Under the `demo` module of the trading application, you will find a mini-tracing application that showcases the various designs we have discussed thus far.

[11] https://gvolpe.com/blog/error-handling-scala3/

8.1.1.2 Pulsar traces

We learned about the most common of traces: HTTP. Now is the time to talk about the tracing of messages that flow in an event-driven architecture.

When producing and consuming messages, we have two options:

1. add the kernel to our messages.
2. use metadata to send the kernel (recommended).

The first option is quite invasive, requiring us to add an extra field to all our data. Thus, making the second approach much more appealing, as the kernel can be classified as metadata.

Apache Pulsar supports sending and receiving metadata with every message; it is called *properties* in Pulsar terminology, modeled as a `Map[String, String]`.

We added the following overloaded method to our `Producer` to make this possible.

```
trait Producer[F[_], A]:
  def send(a: A, properties: Map[String, String]): F[Unit]
```

The tracing demo application also showcases this approach, as shown in the following code snippet.

```
def one[F[_]: GenUUID: Monad: Trace](
    producer: Producer[F, User],
    users: UsersDB[F],
    ack: MsgId ⇒ F[Unit]
): Msg[String] ⇒ F[Unit] =
  case Msg(msgId, _, name) ⇒
    Trace[F].span("name-consumer") {
      Trace[F].put("new-username" → name) *>
        GenUUID[F].make[UUID].flatMap { id ⇒
          users.save(User(id, name)).flatMap {
            case Left(DuplicateUser) ⇒
              Trace[F].put("duplicate" → name)
            case Right(_) ⇒
              Trace[F].put("ok" → name) *>
                Trace[F].kernel.flatMap { kernel ⇒
                  producer.send(User(id, name), kernel.toHeaders)
                }
          } *> ack(msgId)
        }
    }
```

The relevant part is the acquisition of the kernel, followed by producing the message together with the metadata.

```
Trace[F].kernel.flatMap { kernel ⇒
  producer.send(User(id, name), kernel.toHeaders)
}
```

To support metadata on the consuming side, we also enhanced the `Consumer.Msg` datatype.

```
type MsgId      = String
type Properties = Map[String, String]

final case class Msg[A](id: MsgId, props: Properties, payload: A)
```

The consuming side is showcased in the tracing demo application as well.

```
val users: Consumer[IO, User] ⇒ Stream[IO, Unit] = c ⇒
  c.receiveM.evalMap { case Msg(id, props, user) ⇒
    val k = Kernel(props)
    ep.continue("ok", k).orElse(ep.continue("duplicate", k)).use { sp ⇒
      sp.span("user-consumer").use { sp1 ⇒
        sp1.put("user" → user.name) *>
          IO.println(s"$user with kernel: $props \n") *> c.ack(id)
      }
    }
  }
```

Once we resume a trace, we could invoke other functions with a `Trace` constraint. However, the only way to construct a proper instance is via `Trace.ioTrace(span)`, or if we use `Kleisli` or MTL from there, making it somewhat inconvenient in a tagless final application.

Still, there is no escape from this approach if we wish to trace internal components in a single application, such as database calls, external HTTP calls, and so on.

So the question we need to ask ourselves is: how much do we want to trace? Once we thread the context everywhere, we might as well trace every component, but what if we could get away by tracing fewer things?

8.1.2 Centralized

The shortcomings of the `Trace` approach made me think about an *alternative-but-rather-unorthodox* method that would allow us to have the cake and eat it too.

Since every service communicates via Pulsar messages, a centralized tracing service could hook up directly on every topic we wish to inspect.

Of course, this only extends to tracing incoming and outgoing messages, but this might be a good fit for any message-driven architecture.

We will learn it is not all good news, though. As with every design decision, there are always trade-offs. In this case, increased complexity is one, but we will get to the fine details soon enough.

The following sections will explore the final `tracing` service that demonstrates this technique. It all starts with tracer interfaces for both trading and forecasting.

8.1.2.1 Forecasting tracer

The forecasting tracer is the simplest of the two.

```
trait ForecastingTracer[F[_]]:
  def trace(
    cmd: ForecastCommand,
    evt: Either[AuthorEvent, ForecastEvent]
  ): F[Unit]
```

Its primary constructor takes an `EntryPoint[F]` as an argument, so we can always start from a root span.

```
object ForecastingTracer:
  def make[F[_]: MonadCancelThrow](
      ep: EntryPoint[F]
  ): ForecastingTracer[F] = new:
    def trace(
        cmd: ForecastCommand,
        evt: Either[AuthorEvent, ForecastEvent]
    ): F[Unit] = ???
```

Upon receiving a `ForecastCommand` and either an `AuthorEvent` or a `ForecastEvent`, we create the trace. In between, we can add as much contextual information as we wish via the `put` method.

```
ep.root("forecast-root").use { root ⇒
  root.span(s"forecast-command-${cmd.cid.show}").use { sp1 ⇒
    val cid        = evt.fold(_.cid, _.cid)
    val createdAt  = evt.fold(_.createdAt, _.createdAt)
    val durationMs = createdAt.value.toEpochMilli - cmd.createdAt.value.toEpochMilli
    val evtPayload = evt.fold(_.asJson, _.asJson)

    sp1.put(
      "correlation_id" → cmd.cid.show,
      "created_at"     → cmd.createdAt.show,
      "payload"        → cmd.asJson.noSpaces
    ) *> sp1.span(s"forecast-event-${cid.show}").use { sp2 ⇒
      sp2.put(
        "correlation_id" → cid.show,
        "created_at"     → createdAt.show,
        "duration_tx_ms" → durationMs.show,
        "payload"        → evtPayload.noSpaces
      )
    }
  }
}
```

It is worth mentioning that we would be losing the automatic calculation of every span duration by using Natchez, as the internal implementation sets the `duration_ms` property to the moment the trace is created instead of setting it to the `createdAt` value of our event.

Adapting Natchez to this model would require a bit of work, but it is doable. Otherwise, we can ignore the `duration_ms` and simply base our queries on a field we have control over: `duration_tx_ms`.

This is the simplest way of doing tracing, though omitting the complexity that lives on the state machines. Let's look into that next.

8.1.2.2 Forecasting FSM

Here we have the state and input types of the forecasting state machine.

```
type ForecastState = (List[AuthorEvent], List[ForecastEvent], List[ForecastCommand])
type ForecastIn    = AuthorEvent | ForecastEvent | ForecastCommand
```

The FSM receives either a command or one of the two event types, and it then associates them via a `CorrelationId`. Once a matching pair is found, it invokes the `trace` method.

The state of our FSM is a tuple of three different lists for the commands and events that have not yet been associated.

In this case, we know it is guaranteed to receive either event for every command, so we keep it simple. Still, we need to consider that an event might arrive before a command, even if the latter is always produced first.

This is the exact approach used by Kafka Streams, which buffers incoming data until it can join correlated events by some ID, pushing the aggregated data down to the consumers.

We could also leverage Pulsar Functions, but we must add the necessary infrastructure to easily monitor and maintain such functions. Still, this could represent an additional workload on the team. For this reason, monitoring and maintaining yet another Scala service using the ordinary JVM tools can be more appealing.

Below we have a partial FSM implementation (only showing highlighted parts).

```
def forecastFsm[F[_]: Applicative: Logger](
    tracer: ForecastingTracer[F]
): FSM[F, ForecastState, ForecastIn, Unit] =
  FSM {
    case ((atEvents, fcEvents, fcCommands), cmd: ForecastCommand) ⇒
      (atEvents.find(_.cid == cmd.cid), fcEvents.find(_.cid == cmd.cid)) match
        ???

    case ((atEvents, fcEvents, fcCommands), evt: ForecastEvent) ⇒
      fcCommands.find(_.cid == evt.cid) match
        ???

    case ((atEvents, fcEvents, fcCommands), evt: AuthorEvent) ⇒
      fcCommands.find(_.cid == evt.cid) match
        ???
  }
```

Whenever either event is received, we check for existing commands in the internal state and try to match them via `CorrelationId`. If successful, we trace it; otherwise, we add the event to the internal state.

Next, when we receive a command, we also need to check whether either event has been received before, as they come from different Pulsar topics and can be unordered when merged altogether.

Furthermore, there could be an edge case where we lose a message and never associate a command to an event. In such cases, we could add an expiration mechanism and either discard the unassociated command or trace it with a dummy event.

This gets complex very quickly, though, so we omit it here, but we will see a similar idea in action with the trading tracer in the following section.

8.1.2.3 Trading tracer

This is modeled entirely differently, and we will soon learn why.

```
trait TradingTracer[F[_]]:
  def command(cmd: TradeCommand): F[CmdKernel]
  def event(kernel: CmdKernel, evt: TradeEvent): F[EvtKernel]
  def alert(kernel: EvtKernel, alt: Alert): F[Unit]
```

Both the `command` and `event` methods return a newtype over `Kernel` so that the trace can be continued someplace else. Conversely, the `alert` method returns `F[Unit]`, denoting the end of the chain of traces.

Given a `CmdKernel` produced by `command`, the `event` method can continue the trace by adding the given `TradeEvent`, and so on.

Here is a glimpse of the immediate implementation.

```
object TradingTracer:
  def make[F[_]: MonadCancelThrow](
      ep: EntryPoint[F]
  ): TradingTracer[F] = new:
    def command(cmd: TradeCommand): F[CmdKernel] =
      ep.root("trading-root").use { root ⇒ ??? }

    def event(k: CmdKernel, evt: TradeEvent): F[EvtKernel] =
      ep.continue(s"trading-command-${evt.cid.show}", k.value).use { sp ⇒
        ???
      }

    def alert(k: EvtKernel, alt: Alert): F[Unit] =
      ep.continue(s"trading-event-${alt.cid.show}", k.value).use { sp ⇒
        ???
      }
```

Many alerts can be produced from a single event. Figure 8.1.2 should give you a good idea of what to expect as a final result.

There could also be zero alerts emitted per event, so how do we know whether an event should be kept in our state machine waiting for an alert that will never arrive?

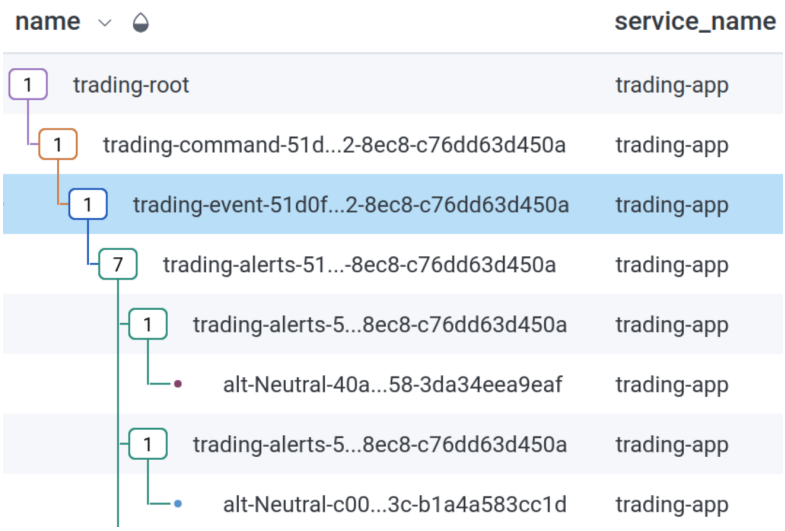


Figure 8.1.2: trading tracer

8.1.2.4 FSM

As hinted previously, we can solve it by introducing temporal windows with an expiration mechanism, but this is where complexity increases dramatically.

The following types are involved in the state machine responsible for trading traces.

```
type MatchingVals = (Timestamp, Option[CmdKernel], Option[EvtKernel])
type MatchingIds  = Map[CorrelationId, MatchingVals]
type Tick         = Unit
type TradeState   = (List[TradeEvent], List[Alert], MatchingIds)
type TradeIn      = TradeCommand | TradeEvent | Alert | Tick
```

Let's start off by analyzing the `TradeState` type alias. We accumulate events and alerts, as well as a complex type `MatchingIds`. We do not need to collect commands because they represent the beginning of a trace.

`MatchingIds` is a key-value store where the key is a `CorrelationId` to associate the different message types. As values, we have a `Timestamp` used for expirations and two optional kernels to resume traces.

We have commands, events, alerts, and a tick as inputs to the state machine. Before we dive into the tick's details, let's look at a few internal methods.

```
def expireMatchingIds[F[_]: Monad: Time](
    ids: MatchingIds
): F[MatchingIds] = ???
```

This one inspects all the existing `Timestamp`s within the `Map`, and it removes all of those considered expired (the default expiration is one minute).

```
def updateMatchingIds[F[_]: Monad: Time](
    ids: MatchingIds,
    cid: CorrelationId,
    kernel: Either[CmdKernel, EvtKernel]
): F[MatchingIds] = ???
```

This second one updates an existing `MatchingIds` with a potential new `CorrelationId` and kernel. You can refer to the source code for the complete picture.

Here's the constructor of the FSM.

```
val MatchingIdsExpiration = 1.minute

def tradingFsm[F[_]: Logger: Monad: Time](
    tracer: TradingTracer[F]
): FSM[F, TradeState, TradeIn, Unit] = FSM { ??? }
```

Next, let's examine how an incoming `TradeCommand` is processed.

```
case ((events, alerts, ids), cmd: TradeCommand) ⇒
  for
    k <- tracer.command(cmd)
    i <- updateMatchingIds(ids, cmd.cid, Left(k))
  yield (events, alerts, i) → ()
```

We invoke the `tracer.command` method and get a kernel back that is later used to update the existing `MatchingIds`. Next comes the processing of `TradeEvent`.

```
case (st @ (events, alerts, ids), evt: TradeEvent) ⇒
  ids.get(evt.cid).flatMap((_, k, _) ⇒ k) match
    case Some(cmdKernel) ⇒
      for
        k <- tracer.event(cmdKernel, evt)
        i <- updateMatchingIds(ids, evt.cid, Right(k))
      yield (events, alerts, i) → ()
    case None ⇒
      expireMatchingIds[F](ids).map(i ⇒ (events :+ evt, alerts, i) → ())
```

We first check if there is a matching `CorrelationId`. If so, we get the command kernel, trace the event, and update the state with the newly acquired event kernel. Otherwise, we trigger the expiration mechanism and add the event to the internal state.

Alerts are processed similarly, except there is no need to update any kernels.

```
case (st @ (events, alerts, ids), alt: Alert) ⇒
  ids.get(alt.cid).flatMap((_, _, k) ⇒ k) match
    case Some(evtKernel) ⇒
      tracer.alert(evtKernel, alt).as(st → ())
    case None ⇒
      expireMatchingIds[F](ids).map(i ⇒ (events, alerts :+ alt, i) → ())
```

Next comes the `Tick`, which is arguably the most intriguing one.

```
case (st @ (events, alerts, ids), tick: Tick) ⇒
  val fsm = tradingFsm(tracer)

  val processEvents: F[TradeState] =
    events.foldLeft(st.pure[F]) { (getSt, evt) ⇒
      getSt.flatMap(fsm.runS(_, evt))
    }

  def processAlerts(st1: TradeState): F[TradeState] =
    alerts.foldLeft(st1.pure[F]) { (getSt, alt) ⇒
      getSt.flatMap(fsm.runS(_, alt))
    }

  (processEvents >>= processAlerts).tupleRight(())
```

We start by creating a new FSM within the FSM, which may sound crazy until we remind ourselves that a state machine is nothing more than a function invoked repeatedly.

This is necessary so that previously accumulated events and alerts that arrived in a different order can still be traced.

Ticks are emitted every two seconds, as we do with snapshots (see Snapshots FSM).

8.1.2.5 FSM Dependent Types

In Chapter 4, we have learned about match types and dependent types (see Dependent Types). We will now see how this feature is leveraged in the tracing state machines.

We have previously defined our FSM types as follows.

```
def forecastFsm[F[_]: Applicative: Logger](
    tracer: ForecastingTracer[F]
): FSM[F, ForecastState, ForecastIn, Unit] = FSM { ??? }

def tradingFsm[F[_]: Logger: Monad: Time](
    tracer: TradingTracer[F]
): FSM[F, TradeState, TradeIn, Unit] = FSM { ??? }
```

In reality, however, we have the following isomorphic definitions.

```
def forecastFsm[F[_]: Applicative: Logger]: SM[F, ForecastIn] =
  tracer ⇒ FSM { ??? }

def tradingFsm[F[_]: Logger: Monad: Time]: SM[F, TradeIn] =
  tracer ⇒ FSM { ??? }
```

Since the tracer and state types depend on the input type, we can now enforce this relationship at the type level.

```
type St[In] = In match
  case ForecastIn ⇒ ForecastState
  case TradeIn    ⇒ TradeState

type Tracer[F[_], In] = In match
  case ForecastIn ⇒ ForecastingTracer[F]
  case TradeIn    ⇒ TradingTracer[F]

type SM[F[_], In] = Tracer[F, In] ⇒ FSM[F, St[In], In, Unit]
```

The tracer and state types are uniquely determined by the input type at compile time, making our code much safer, thanks to match types.

8.1.2.6 Main

As usual, we have a subscription type for the consumers.

```
val sub =
  Subscription.Builder
    .withName("tracing")
    .withType(Subscription.Type.Exclusive)
    .build
```

Followed by a long sequence of resources to consume messages from all the topics we wish to trace (omitting some details so the code fits on the page).

```
def resources =
  for
    config <- Resource.eval(Config.load[IO])
    pulsar <- Pulsar.make[IO](config.pulsar.url)
    ep     <- Honeycomb.makeEntryPoint(config.honeycombApiKey)
    alerts           <- Consumer.pulsar[IO, Alert](???).map(_.receive)
    tradingEvents    <- Consumer.pulsar[IO, TradeEvent](???).map(_.receive)
    tradingCommands  <- Consumer.pulsar[IO, TradeCommand](???).map(_.receive)
```

```
    authorEvents     <- Consumer.pulsar[IO, AuthorEvent](???).map(_.receive)
    forecastEvents   <- Consumer.pulsar[IO, ForecastEvent](???).map(_.receive)
    forecastCommands <- Consumer.pulsar[IO, ForecastCommand](???).map(_.receive)
    fcFsm  = forecastFsm[IO].apply(ForecastingTracer.make[IO](ep))
    tdFsm  = tradingFsm[IO].apply(TradingTracer.make[IO](ep))
    server = Ember.default[IO](config.httpPort)
  yield (server, alerts, tradingEvents, tradingCommands, ..., fcFsm, tdFsm)
```

Next comes the `run` method, where all inputs are merged. The call to `Stream.resource(...)` is also skipped for brevity.

```
val ticks: Stream[IO, TradeIn] =
  Stream.fixedDelay[IO](2.seconds)

val trading =
  tradingCommands
    .merge[IO, TradeIn](tradingEvents.merge(alerts))
    .merge(ticks)
    .evalMapAccumulate(TradeState.empty)(tdFsm.run)

val forecasting =
  authorEvents
    .merge[IO, ForecastIn](forecastEvents.merge(forecastCommands))
    .evalMapAccumulate(ForecastState.empty)(fcFsm.run)

Stream(
  Stream.eval(server.useForever),
  trading,
  forecasting
).parJoin(3)
```

Here we see how commands, events, alerts and ticks are merged and then run through the corresponding state machine. The forecasting process is similar, except there is no need for ticks.

Notice we omitted a lot of scalability areas for this service. For simplicity, we went straight with a single instance using exclusive subscriptions for all the topics, but this could quickly fail under heavy load.

Application architectures such as the one used by the snapshots service (see Snapshots scalability) could also apply to this service in case you would like a challenging task.

However, instead of persisting state and reading it back on startup (which would highly increase complexity), we could switch to manual acknowledgement mode—i.e. `receiveM` instead of `receive`— and only send the ack once the chain of traces has ended.

Beware that the acknowledgement timeout[12] needs to be tweaked accordingly to be greater than our expiration time.

This is trivial when we know when it ends. In our case, however, we would need to rely on the expiration mechanism to send the remaining acknowledgements.

Furthermore, with this strategy, we could leverage the `Failover` subscription to adhere to the single-writer principle in a stateless way.

8.1.2.7 Summary of tracing

We learned that the centralized tracing approach has pros and cons.

On the one hand, it is fabulous that all the other services are unaware of tracing code, which translates to easy maintenance. On the other hand, the tracing service pays the price for the extra complexity that entails keeping track of messages that belong to the same transaction, which could be tricky or even not possible at times.

To summarize, I think it is a formidable technique when a single message is always emitted per message, as is the case with forecasting.

Whenever we expect zero or many messages being produced for every other message—as is the case with trading—we might not always be able to get away with it.

Regardless of your final choice, you are now provided with the foundations to make an informed decision next time you are asked to add distributed tracing to your system.

For those interested in learning more about streaming systems and the temporal windowing function used in this centralized tracing service, the Streaming Systems[13] book by Tyler Akidau, Slava Chernyak, and Reuven Lax is a good read, mainly based on Apache Flink[14] and Apache Beam[15].

Furthermore, similar techniques are also used in modern technologies such as Azure Stream Analytics[16] and Kafka Streams.

[12] https://pulsar.apache.org/docs/concepts-messaging/#acknowledgement-timeout
[13] https://www.oreilly.com/library/view/streaming-systems/9781491983867/
[14] https://flink.apache.org/
[15] https://beam.apache.org/
[16] https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-window-functions

8.2 Build & run

The trading repository[17] contains most of the information you need to run the entire system. Regardless, we are now going to review the structure of the project and the running instructions, as showcased in fig. 8.2.1.

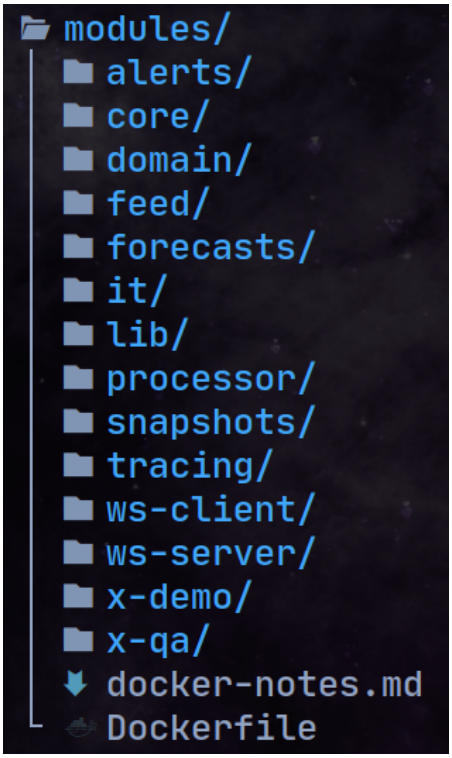


Figure 8.2.1: modules

These are all the modules in our `sbt` project, from which only `alerts`, `forecasts`, `processor`, `snapshots`, `tracing`, and `ws-server` are deployable as back-end applications.

This means all these services can be packaged as binaries, Docker images, and other types of runnable applications. It also means we can deploy them to a Kubernetes cluster.

We will shortly learn about the latter, but it is worth mentioning we can run every application directly via `sbt project:run`. In fairness, this is the recommended way of running both the `feed` application and all the examples under the `x-demo` module.

The `it` module corresponds to the integration tests we learned about in Chapter 7 (see Integration tests), whereas the `x-qa` module corresponds to the smoke tests we will learn about in a few sections (see Smoke tests).

[17] https://github.com/gvolpe/trading

8.2.1 Docker compose

At the project's top-level directory, you will find a `docker-compose.yml` file that declares all the deployable services and runtime dependencies.

Before bringing the containers up, we need to ensure all the services are published as Docker images.

```
$ docker build -t jdk17-curl modules/
$ sbt docker:publishLocal
```

All the JVM services depend on a custom `jdk17-curl` image that adds `curl` support to a base `openjdk:17-slim-buster` image, so that we can rely on it for health checks.

Software requirements

Docker[a] and docker-compose[b] need to be installed.

[a] https://www.docker.com/get-started
[b] https://docs.docker.com/compose/

Running the following command now should bring up all the containers.

```
$ docker-compose up
Creating network "trading_app" with the default driver
Creating trading_pulsar_1 ... done
Creating trading_redis_1  ... done
Creating trading_ws-server_1      ... done
Creating trading_pulsar-manager_1 ... done
Creating trading_alerts_1         ... done
Creating trading_processor_1      ... done
Creating trading_snapshots_1      ... done
Creating trading_forecasts_1      ... done
Creating trading_tracing_1        ... done
Creating trading_prometheus_1     ... done
Creating trading_grafana_1        ... done
```

Bear in mind this requires a certain amount of memory, so you may need to tweak the usage of resources when working on a low-resource machine.

This is the recommended way of running the system locally. However, we can do better in a production environment, as we will learn soon.

8.2.2 Continuous integration

In the trading repository, we use Github Actions as the default CI build mechanism. The integration tests only need Redis, so we only run this service.

```
$ docker-compose up -d redis
Creating network "trading_app" with the default driver
Creating trading_redis_1  ... done
```

We can consistently use `docker-compose` both locally and in our CI build.

The CI and development environment are guaranteed to be the same, as declared in the `flake.nix` file. Nix[18] helps us achieve reproducibility across different machines.

Though, this is completely optional for every user. The `flake.nix` file is there if you want to take advantage of it. Otherwise, you are expected to have `sbt`, `jdk`, and all the necessary software installed and set up for development on your machine.

Furthermore, the `jdk17-curl` Docker image used by all the services can also be built via Nix if you want to take advantage of the repository's flake.

```
$ nix build
$ docker load -i result
```

We do not have Continuous Delivery (CD) in the reference project, which would require a production environment. However, we have a set of workflows in place that will facilitate a CD system in the future.

Once the `Scala` build passes in the `main` branch, a `Registry` workflow responsible for building and publishing Docker images of our applications to the Github registry will be triggered.

```
name: Registry

on:
  workflow_run:
    workflows: ["Scala"]
    branches: [main]
    types:
      - completed
```

Only images for the `processor`, `alerts`, and `ws` services are published for now, which are necessary to run the smoke tests (see next section).

Once the `Registry` build succeeds, a `Smokey` workflow will be triggered.

[18] https://nixos.org/

```yaml
name: Smokey

on:
  workflow_run:
    workflows: ["Registry"]
    branches: [main]
    types:
      - completed
```

This workflow will pull the application's Docker images from the registry and proceed with the execution of the smoke tests we will learn about shortly.

A continuous delivery workflow (e.g. promoting a service to UAT or STAGING environments) could be triggered once the smoke tests succeed.

CI/CD systems are ubiquitous nowadays, and Github Actions is a solid choice.

8.2.3 Smoke tests

Smoke tests[19] give us certain guarantees about the overall functionality of our system.

Our implementation is a mix of smoke and functional testing, serving the purposes of verifying that the applications successfully connect to Pulsar and Redis (smoke tests), and that certain expectations about the communication between clients and services are fulfilled (functional tests).

In a nutshell, this is what `smokey` (our smoke testing suite) does:

1. Run essential applications (`processor`, `alerts` and `ws`) via `docker-compose`.
2. Two clients connect via WS and subscribe to a subset of symbols.
3. Generate and publish a fixed set of `TradeCommand`s.
4. Verify each WS client only gets alerts corresponding to its subscriptions.

Here's a sneak peek of the main test.

[19] https://www.guru99.com/smoke-testing.html

```
test("Trading smoke test") { case (pulsar, ws) ⇒
  val cli1 = clientSimulator(ws, symbols1, expected1.size)
  val cli2 = clientSimulator(ws, symbols2, expected2.size)
  (commandsProducer(pulsar) &> (cli1, cli2).parTupled).map {
    case (((_: WsOut.Attached) :: xs), ((_: WsOut.Attached) :: ys)) ⇒
      expect.same(xs, expected1) && expect.same(ys, expected2)
    case xs ⇒
      failure(s"Expected ${expected1.size + 1} messages")
  }
}
```

The `commandsProducer` method is the one publishing `TradeCommand`s after an initial delay, whereas the `clientSimulator` method handles the Web Socket client connections and keeps track of the messages received.

In the end, we run a few assertions expecting that both clients got the initial `Attached` messages followed by a subset of trading alerts for their subscriptions.

The full implementation can be found in the trading repository under the `x-qa` module.

8.3 Monitoring

So far, we have talked a lot about tracing, but observability is much more than that. In this section, we will look into the monitoring of our services and infrastructure.

Reports of CPU, memory, and thread usage, among other properties, can be critical to understanding the health of any distributed system.

Since we run on the JVM, specific metrics that help us understand its performance will also be crucial whenever production issues arise.

Moreover, from these metrics we can automate alerts for any team on duty.

8.3.1 Prometheus

Prometheus[20] is a standard open-source monitoring solution. We use this in our system—declared in our `docker-compose` dependencies—to collect metrics from all the JVM services.

To do so, every service has an HTTP server from which Prometheus can scrape such metrics. The pertinent code is powered by `http4s-prometheus-metrics`, which exports sane default metrics.

```
private def metrics[F[_]: Async]: Resource[F, HttpRoutes[F] ⇒ HttpRoutes[F]] =
  for
    prt <- PrometheusExportService.build[F]
    ops <- Prometheus.metricsOps[F](prt.collectorRegistry)
  yield rts ⇒ Metrics[F](ops)(prt.routes <+> rts)
```

This adds a `GET /metrics` endpoint to every service, which exposes metrics in a raw format that can be read and parsed for further processing.

[20] https://prometheus.io/

8.3.2 Grafana

To get the metrics reported by Prometheus rendered as pretty dashboards, we can leverage Grafana[21], which is another standard in the industry.

Figure 8.3.1: JVM metrics

Under the `monitoring` directory in the trading repository, you will find default Prometheus and Grafana configurations for all our services, as well as configuration for Pulsar Manager, which is slowly becoming the default UI for Pulsar.

Dashboards can be customized according to our needs. You may also find further inspiration from the wide selection of community-contributed dashboards[22].

These configurations are read by default when using `docker-compose`, so you can see it all in action when running the system locally.

[21] https://grafana.com/
[22] https://grafana.com/grafana/dashboards

8.4 Deployment

Production deployment should ideally be an automated task handled by our CD (Continuous Delivery) system. Arguably, Kubernetes (k8s)[23] has become the de-facto clustering tool in our industry, allowing for easy deployments and containers orchestration.

We will briefly touch on this area to analyze what we should expect when deploying every service in our system. This means talking about downtime, uptime, and resilience.

8.4.1 K8s cluster

We do not have a production environment for the reference trading application, but we can replicate a k8s cluster locally using minikube[24], which works on all major operating systems.

To demonstrate some of the features k8s brings to the table, we will only work with a subset of the services to minimize the required amount of resources.

Nix users can leverage the declarative development shell to access the software to run the following examples. In any other case, make sure you have the required programs available in scope.

Under the `ops` directory, we have both `apps` and `infra` directories, in which you will find k8s deployment files.

In essence, managing the local cluster boils down to these two commands.

```
$ minikube start
$ minikube stop
```

Once started, we need to publish our Docker images into the minikube environment before deploying anything, which is done as instructed below.

```
$ eval $(minikube docker-env)
$ docker build -t jdk17-curl modules/
$ sbt docker:publishLocal
```

Next, we can bring up the infra services and wait until they are running.

```
$ kubectl apply -f ops/infra
deployment.apps/pulsar created
deployment.apps/redis created
service/redis configured
```

[23] https://kubernetes.io/
[24] https://minikube.sigs.k8s.io

Then we can bring up all the application services after setting the `HONEYCOMB_API_KEY` value in the `ops/apps/tracing-deployment.yaml` file.

```
$ kubectl apply -f ops/apps
deployment.apps/alerts created
service/alerts configured
networkpolicy.networking.k8s.io/app configured
...
```

There are better ways of handling secrets in k8s, but to serve our purpose, we can either change it manually or use a command like `envsubst` (only available to Linux users).

```
envsubst -i ops/apps/tracing-deployment.yaml -o ops/apps/tracing-deployment.yaml
```

All these instructions are also outlined in the `ops/deployment.md` file.

Tips

You can get a Honeycomb[a] API Key by registering for free.

[a] https://www.honeycomb.io/

8.4.2 Pods management

Once services are up and running in our local cluster, we can move on to the exciting bits.

```
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
alerts-78846666d-sl2ld        1/1     Running   0          4m34s
forecasts-6dd77757bb-zczsb    1/1     Running   0          4m34s
forecasts-6dd77757bb-8wsbm    1/1     Running   0          4m34s
processor-6f89879764-8z6d9    1/1     Running   0          4m34s
pulsar-75f65c75fc-t7znc       1/1     Running   0          5m26s
redis-cd5c5d4d7-nt4dc         1/1     Running   0          5m26s
snapshots-85856767b4-pmj95    1/1     Running   0          4m34s
ws-server-668c9dccb8-244dg    1/1     Running   0          4m34s
```

8.4.2.1 Scaling

For instance, we can scale out any service with a single command.

```
$ kubectl scale --replicas=2 deployment/alerts
```

After a few seconds, we can effectively see two pods of the same service running in harmony.

```
$ kubectl get pods | rg alerts
alerts-78846666d-4qcg9       1/1     Running   0          32s
alerts-78846666d-sl2ld       1/1     Running   0          7m44s
```

This is possible with the `alerts` service because its consumer uses a combination of key-shared and unique exclusive subscriptions, so multiple consumers can run concurrently.

> **Notes**
>
> The `rg` command comes from ripgrep[a], replacing the good old `grep`.
>
> [a] https://github.com/BurntSushi/ripgrep

What if we try to scale out the `tracing` service, which instead uses an exclusive subscription type with a global name?

```
$ kubectl scale --replicas=2 deployment/tracing
$ kubectl get pods | rg tracing
tracing-5886fbd789-st66p     0/1     CrashLoopBackOff   2 (13s ago)   42s
tracing-5886fbd789-td6hz     1/1     Running            0             78s
```

Yikes! We cannot scale out any service using such a subscription type without giving it a unique name, but this is precisely what we should expect. Being an exclusive consumer means there can only be one at a single point in time.

If we inspect the logs of the crashed pod, we will find something along these lines.

```
2022-03-15T10:36:08 [io-1] INFO t.lib.Logger - Initializing tracing service
PulsarClientException: {"errorMsg":"Exclusive consumer is already connected"}
```

This is how far our design decisions can reach, making it extremely important to understand Pulsar subscriptions. We can now roll back and leave a single instance running by down-scaling the service.

```
$ kubectl scale --replicas=1 deployment/tracing
```

8.4.2.2 Fail-over subscriptions

Arguably, the most interesting case is when using the fail-over subscription type, which is the case of the `forecasts` service. By default, we fire up two replicas, but only one will get to consume messages at first.

We can see this in action with a little bit of work. First off, let's verify we have two pods running, as per the specification.

```
$ kubectl get pods | rg forecasts
forecasts-6dd77757bb-zczsb   1/1     Running   0          7m13s
forecasts-6dd77757bb-8wsbm   1/1     Running   0          7m13s
```

Next, we need to run the `feed` service to generate random forecasting data while substituting the default Pulsar URI.

```
PULSAR_URI=pulsar://192.168.49.2:32356 sbt feed/run
```

This should be the address of the Pulsar instance running on the minikube cluster. By default, it is not exposed, but we can change that with the following command.

```
$ kubectl expose deployment pulsar --type=NodePort --port=6650 --name=pulsar-svc
```

Next, to find out the URL of the Pulsar service, we ask minikube.

```
$ minikube service --url pulsar-svc
http://192.168.49.2:32356
```

Now we are ready to witness the fail-over subscription type in action.

Before we trigger the `feed` service, we need to ensure we are tailing the logs of both `forecasts` pods via `kubectl logs -f`.

Once it all starts, we will only see messages flowing in one pod. Then immediately, let's kill the working pod from a different terminal with the following command.

```
$ kubectl delete pod forecasts-6dd77757bb-zczsb
pod "forecasts-6dd77757bb-zczsb" deleted
```

We should see all the messages being redirected to the fail-over pod, namely `forecasts-6dd77757bb-8wsbm`. A little bit tricky to test but a great way to verify things work as expected.

The same kind of test can be performed on the `snapshots` service, where multiple instances use an `Exclusive` subscription synchronized via a distributed lock, making it somewhat similar to the `Failover` subscription.

Furthermore, we can test directly using `docker-compose` as well. For example, we can add a `snapshots2` service using a different HTTP port and start up both instances.

8.4.2.3 Roll-out restarts

Everything we learned about subscription types and scaling also applies to roll-out restarts (or *rolling releases*). We use this command to deploy new changes into our cluster. E.g.

```
$ kubectl rollout restart deployment/alerts
```

Or we can directly roll out all the applications that have changed.

```
$ kubectl apply -f ops/apps
```

When deploying breaking changes, we need to ensure JSON schemas are backward-compatible (see Schema evolution) to avoid any system disruption.

This command is smart enough to handle all types of Pulsar subscriptions. It always tries to bring the new container up before putting the old one down, which has zero downtime.

If this is not possible—as when using exclusive subscriptions with a global name—the old container is brought down first, and then the new container starts up, which can cause a few seconds of downtime, depending on the case.

8.5 Summary

Observability is vital in distributed systems. In this final chapter, we learned about the different techniques we can leverage for distributed tracing.

Furthermore, we learned how to run the system locally using `sbt` directly or via `docker-compose`. With the latter, we get to see monitoring in action via Prometheus and Grafana.

Last but not least, we learned the basics of k8s deployments, such as scaling and roll-out releases, and how all of these interact with the different Pulsar subscription types.

Kubernetes is ubiquitous nowadays; thus, upping your game and learning about it will undoubtedly put you ahead when looking for a job.

Now let me congratulate you for making it until the very end of this book! I hope you enjoyed reading it as much as I did writing it.

Thank you all for your support, and feel free to engage with me and other fellow readers in the discussion forum[25].

You can also submit private feedback directly to **feda@gvolpe.com**.

[25] https://github.com/gvolpe/trading/discussions

9 Bonus: Web App

As mentioned in the preface, we have two web applications written in Elm and Scala.js, respectively. In this bonus section, we will only briefly highlight the latter.

The Scala web app is built using the excellent Tyrian[1] framework, which follows the Elm architecture[2], making it easy to follow the Elm code as well.

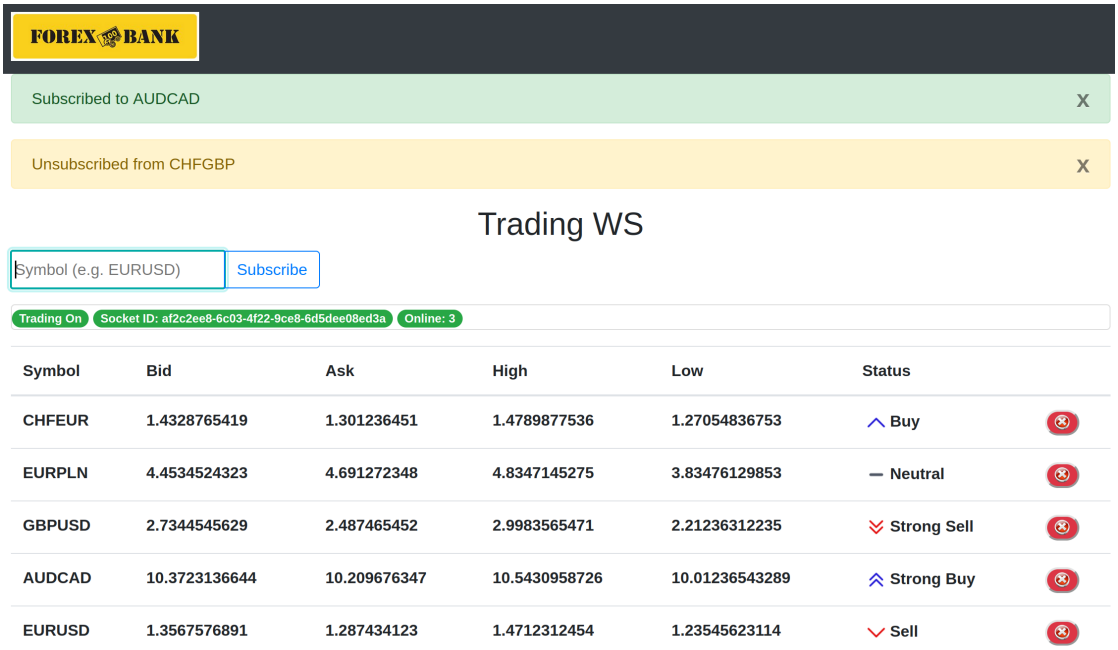


Figure 9.0.1: web app

Elm is a very user-friendly language. However, it doesn't support typeclasses[3], which can lead to repetition. With Scala and Tyrian, we get the best of both worlds!

[1] https://tyrian.indigoengine.io/
[2] https://guide.elm-lang.org/architecture/
[3] https://faq.elm-community.org/#does-elm-have-ad-hoc-polymorphism-or-typeclasses

9.1 Entry point

Every Tyrian app is modeled by extending `TyrianApp`, which takes two type arguments: the message and model types.

```
@JSExportTopLevel("TyrianApp")
object Main extends TyrianApp[Msg, Model]:

  def init(flags: Map[String, String]): (Model, Cmd[IO, Msg]) =
    Model.init → Cmd.None

  def update(model: Model): Msg ⇒ (Model, Cmd[IO, Msg]) =
    runUpdates(model)

  def view(model: Model): Html[Msg] =
    render(model)

  def subscriptions(model: Model): Sub[IO, Msg] =
    model.socket.subscribe(Subs.ws)
```

We can think of the Elm architecture (and Tyrian's) as another finite state machine.

```
case class FSM[F[_], S, I, O](run: (S, I) ⇒ F[(S, O)])

type ElmArch = FSM[Cmd, Model, Msg, Unit]
```

We start with an initial state `Model.init`, and input messages are fed from `view` and `subscriptions`. The `update` method can be seen as the `run` method of the FSM.

The `view` method produces input messages as an outcome of user interaction, and it renders the final HTML view based on the current state (the model).

The `subscriptions` method produces input messages by interacting with external actors. In this case, these are Web Socket clients.

This gives us a good idea of Tyrian's architecture. However, there is more to it in practice, as Tyrian models the execution of `Cmd`s and `Sub`s via `cats.effect.IO`.

9.2 Datatypes

The `Msg` datatype represents the inputs and is defined as follows.

```
enum Msg:
  case CloseAlerts
  case SymbolChanged(input: InputText)
  case Subscribe
  case Unsubscribe(symbol: Symbol)
  case Recv(in: WsOut)
  case ConnStatus(msg: WsMsg)
  case FocusError(id: ElemId)
  case NoOp
```

The `Model` datatype represents the state of the application.

```
final case class Model(
    symbol: Symbol,
    input: InputText,
    socket: TradingSocket,
    onlineUsers: Int,
    alerts: Map[Symbol, Alert],
    tradingStatus: TradingStatus,
    sub: Option[Symbol],
    unsub: Option[Symbol],
    error: Option[String]
)
```

We start the application with an initial state: `Model.init`.

```
object Model:
  def init = Model(
    symbol = mempty,
    input = mempty,
    socket = TradingSocket.init,
    onlineUsers = mempty,
    alerts = Map.empty,
    tradingStatus = TradingStatus.On,
    sub = None,
    unsub = None,
    error = None
  )
```

The `mempty` method is an extension that works for any `Monoid`.

```
def mempty[A: Monoid]: A = Monoid[A].empty
```

We also have a `WsMsg` datatype for all the Web Socket messages, which can be converted back to `Msg` by wrapping it in `ConnStatus`, as the `asMsg` method shows.

```
enum WsMsg:
  case Error(msg: String)
  case Connecting
  case Connected(ws: WebSocket[IO])
  case Heartbeat
  case Disconnected(code: Int, reason: String)

  def asMsg: Msg = Msg.ConnStatus(this)
```

Furthermore, we have a `TradingSocket` class managing all Web Socket messages.

```
final case class TradingSocket(
    wsUrl: WsUrl,
    id: Option[SocketId],
    ws: Option[WebSocket[IO]],
    error: Option[String]
):
  val update: WsMsg ⇒ (TradingSocket, Cmd[IO, Msg]) = ???

  def publish[A: Encoder](a: A): Cmd[IO, Msg] =
    ws.fold(Cmd.None)(_.publish(a.asJson.noSpaces))

  def subscribe(f: WSEvent ⇒ Msg): Sub[IO, Msg] =
    ws.fold(Sub.emit[IO, Msg](Msg.NoOp))(_.subscribe(f))

object TradingSocket:
  def init: TradingSocket =
    TradingSocket(WsUrl("ws://localhost:9000/v1/ws"), None, None, None)
```

We will see the implementation details of the `update` function shortly (see Updates).

Finally, there are a few newtypes present in the model.

```
type InputText = InputText.Type
object InputText extends Newtype[String]:
  given Monoid[InputText] = derive

type WsUrl = WsUrl.Type
object WsUrl extends Newtype[String]
```

```
type ElemId = ElemId.Type
object ElemId extends Newtype[String]
```

The `InputText` newtype derives a `Monoid` instance used in `Model.init`.

We also have two implicit conversions[4] for our newtypes, so they can be accepted wherever the underlying `String` type is expected.

```
given Conversion[InputText, String] = _.value
given Conversion[WsUrl, String]     = _.value
```

Such implicit conversions ought to be used with caution. However, I believe it is justified in cases where we prefer to use newtypes' values as string literals. Haskell does something similar with the IsString[5] typeclass.

Let's now move over to the next section and learn how the UI is built.

[4] https://docs.scala-lang.org/scala3/book/ca-implicit-conversions.html
[5] https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-String.html

9.3 View

The view is based on Bootstrap[6], constructing the necessary HTML using Tyrian's convenient DSL. In this section, we will only highlight the essential parts.

When the application starts, we decide whether we render a "Connect" button or the current connection details based on our model.

```
def renderConnectionDetails: (Option[SocketId], Int) ⇒ Html[Msg] =
  case (Some(sid), online) ⇒
    span(
      span(
        id := "socket-id",
        `class` := "badge badge-pill badge-success"
      )(text(s"Socket ID: ${sid.show}")),
      span(" "),
      span(
        id := "online-users",
        `class` := "badge badge-pill badge-success"
      )(text(s"Online: ${online.show}"))
    )

  case (None, users) ⇒
    span(
      span(
        id := "socket-id",
        `class` := "badge badge-pill badge-secondary"
      )(text("<Disconnected>")),
      span(" "),
      button(
        `class` := "badge badge-pill badge-primary",
        onClick(WsMsg.Connecting.asMsg)
      )(text("Connect"))
    )
```

When clicking the "Connect" button, a `WsMsg.Connecting` message is produced.

[6] https://getbootstrap.com/

All the pieces are then put together via the `render` method.

```
def render(model: Model): Html[Msg] =
  div(`class` := "container")(
    h2(attribute("align", "center"))(text("Trading WS")),
    div(`class` := "input-group mb-3")(
      input(
        `type` := "text",
        id     := "symbol-input",
        autoFocus,
        placeholder := "Symbol (e.g. EURUSD)",
        onInput(s ⇒ Msg.SymbolChanged(InputText(s))),
        value := model.input.value
      ),
      div(`class` := "input-group-append")(
        button(
          `class` := "btn btn-outline-primary btn-rounded",
          onClick(Msg.Subscribe)
        )(
          text("Subscribe")
        )
      )
    )
  )
```

Overall, messages are produced when interacting with the elements on the page. In this last code snippet, we can see `SymbolChanged` and `Subscribe` in action.

9.4 Subscriptions

The only subscriptions we have in our application are related to the Web Socket client.

Let's recap on the definition of the `TradingSocket#subscribe` method.

```
def subscribe(f: WSEvent ⇒ Msg): Sub[IO, Msg] =
  ws.fold(Sub.emit[IO, Msg](Msg.NoOp))(_.subscribe(f))
```

If there is no connection (`ws` is empty), we emit a `NoOp` message. Otherwise, we call `subscribe` on the underlying `WebSocket` interface.

From the entry point, we call it with `Subs.ws`, defined as follows.

```
object Subs:
  def ws: WSEvent ⇒ Msg =
    case WSEvent.Receive(str) ⇒
      jsonDecode[WsOut](str) match
        case Right(in) ⇒ Msg.Recv(in)
        case Left(err) ⇒ WsMsg.Error(s"$err").asMsg
    case WSEvent.Error(err) ⇒
      WsMsg.Error(err).asMsg
    case WSEvent.Open ⇒
      Msg.NoOp
    case WSEvent.Close(code, reason) ⇒
      WsMsg.Disconnected(code, reason).asMsg
    case WSEvent.Heartbeat ⇒
      WsMsg.Heartbeat.asMsg
```

We pattern-match on the different events and produce a message accordingly. In the following section, we will learn how all these messages are processed.

9.5 Updates

The `runUpdates` method takes a model and an input message, resulting in another model while potentially running some effects in `Cmd`.

We start with the easiest one: `Msg.NoOp`, which results in no changes or effects.

```
def runUpdates(model: Model): Msg ⇒ (Model, Cmd[IO, Msg]) =
  case Msg.NoOp ⇒
    model → Cmd.None
```

Next, we have the Web Socket messages received in the `ConnStatus` message.

```
case Msg.ConnStatus(wsMsg) ⇒
  val (ws, cmd) = model.socket.update(wsMsg)
  model.copy(socket = ws, error = ws.error) → cmd
```

These are run via the `socket.update` function, defined as follows.

```
val update: WsMsg ⇒ (TradingSocket, Cmd[IO, Msg]) =
  case WsMsg.Connecting ⇒
    this → WebSocket.connect[IO, Msg](wsUrl, KeepAliveSettings.default) {
      case WebSocketConnect.Socket(s) ⇒ WsMsg.Connected(s).asMsg
      case WebSocketConnect.Error(e)  ⇒ WsMsg.Error(e).asMsg
    }

  case WsMsg.Connected(cws) ⇒
    this.copy(ws = cws.some, error = None) → refocusInput

  case WsMsg.Disconnected(code, reason) ⇒
    val err = s"WS socket closed. Code: $code, reason: $reason"
    this.copy(id = None, ws = None, error = err.some) → Cmd.None

  case WsMsg.Error(cause) ⇒
    this.copy(error = s"Connection error: $cause".some) → Cmd.None

  case WsMsg.Heartbeat ⇒
    this → publish[WsIn](WsIn.Heartbeat)
```

It takes a `WsMsg` as input and returns an updated `TradingSocket` and a command. The `runUpdates` method then updates its model based on this result.

When we receive the `Connecting` message (triggered after the user clicks on Connect), we initiate the Web Socket connection with the configured URL. If it all goes well, we return a `Connected` message, otherwise an `Error` message.

Once we receive the `Connected` message, we update the trading socket state with the current WS client. Conversely, if we get an error, we set the error message in our internal state, which will ultimately be displayed as a JavaScript alert.

On the other hand, the `Disconnected` message removes the current socket ID, and the `Heartbeat` message helps keep the connection alive.

Next, we have the `SymbolChanged` message—triggered when users type in a symbol.

```
case Msg.SymbolChanged(in) if in.length == 6 ⇒
  model.copy(symbol = Symbol(in), input = in) → Cmd.None

case Msg.SymbolChanged(in) ⇒
  model.copy(input = in) → Cmd.None
```

We add some basic validation by checking the length of the input text before creating a valid symbol. Next, we have the subscription messages.

On `Subscribe`, we verify both the current symbol and the existence of a socket ID, which indicates an active Web Socket connection.

If it all checks, we emit a new `Subscribe` message with the current symbol and run two different effects via `Cmd.Batch`: publishing a WS message and refocusing the input.

```
case Msg.Subscribe ⇒
  (model.socket.id, model.symbol) match
    case (_, Symbol.XEMPTY) ⇒
      model.copy(error = "Invalid symbol".some) → Cmd.None
    case (Some(_), sl) ⇒
      val nm = model.copy(sub = sl.some, symbol = mempty, input = mempty)
      nm → Cmd.Batch(model.socket.publish(WsIn.Subscribe(sl)), refocusInput)
    case (None, _) ⇒
      disconnected(model)
```

The `disconnected` method (implementation omitted for brevity) displays an error message as a JavaScript alert.

On `Unsubscribe`, we only check for an existing socket ID and proceed similarly.

```
case Msg.Unsubscribe(sl) ⇒
  model.socket.id.fold(disconnected(model)) { _ ⇒
    val nm = model.copy(unsub = sl.some, alerts = model.alerts - sl)
    nm → Cmd.Batch(
      model.socket.publish(WsIn.Unsubscribe(sl)), refocusInput
    )
  }
```

Ultimately, we have the incoming Web Socket messages: `Attached`, `OnlineUsers`, and `Notification`. These are called `WsOut` because we are reusing the domain model from the back-end application.

```
case Msg.Recv(WsOut.Attached(sid)) ⇒
  _SocketId.replace(sid.some)(model) → Cmd.None

case Msg.Recv(WsOut.OnlineUsers(online)) ⇒
  model.copy(onlineUsers = online) → Cmd.None

case Msg.Recv(WsOut.Notification(t: Alert.TradeAlert)) ⇒
  model.copy(alerts = model.alerts.updated(t.symbol, t)) → Cmd.None

case Msg.Recv(WsOut.Notification(t: Alert.TradeUpdate)) ⇒
  model.copy(tradingStatus = t.status) → Cmd.None
```

The processing of the `Attached` message is the only one using a lens, helping with the nested `model.socket.id` data structure.

```
val _SocketId: Lens[Model, Option[SocketId]] =
  Focus[Model](_.socket).andThen(Focus[TradingSocket](_.id))
```

All these messages result in the update of some properties of the model.

9.6 Build & Run

The Scala.js client lives on the same trading repository as the back-end services under the modules/ws-client[7] directory.

To get started, we need to compile the Scala app to JavaScript and copy the file to the root directory of the `ws-client` module (this last task is only required for `nix run`).

```
$ sbt 'webapp/fullLinkJS;webapp/copyJsFileTask'
```

You can then run it via Nix as follows (it requires flakes[8]).

```
$ nix run .#tyrian-webapp
Using cache dir: ~/trading/nix-parcel-cache
Server running at http://localhost:1234
```

> **Notes**
>
> The `nix run` command will create a directory for the Parcel cache on the current directory, which needs write permissions.

We use `fullLinkJS` to create a fully optimized JS file. However, we can use `fastLinkJS` for faster iterations, but the `copyJsFileTask` does not work with it.

For such cases, it may be more convenient to use `yarn` directly.

```
$ nix develop .#tyrian
$ cd modules/ws-client
$ yarn install
$ yarn build
$ yarn start
yarn run v1.22.17
parcel index.html --no-cache --dist-dir dist --log-level info
Server running at http://localhost:1234
```

However, this is not fully reproducible and can't be guaranteed to work in the future.

Users who prefer not to use Nix must install yarn[9] and parcel[10], and use the former as shown above. If it all goes well, a server should be running at localhost:1234[11].

[7] https://github.com/gvolpe/trading/tree/main/modules/ws-client
[8] https://nixos.wiki/wiki/Flakes
[9] https://yarnpkg.com/
[10] https://parceljs.org/
[11] http://localhost:1234

9.7 Summary

Although a simple enough web application, it showcases the power of functional programming and finite state machines on the client side.

The clear advantage of using Scala.js is that we get to share the domain and common libraries with the back-end applications, leading to a unified model and less duplication.

Ultimately, this wouldn't have been possible without the ongoing effort of compiler engineers to get the Scala language to the next level by compiling to JavaScript and native binaries via Scala.js and Scala Native[12], respectively. So huge props to everyone involved **#ScalaThankYou**

To the readers that made it until this very last bonus chapter, I hope you enjoyed it and give Tyrian a try. Once again, thank you all for your support.

Private feedback can be submitted directly to **feda@gvolpe.com**. Public feedback and open discussions with fellow readers can be sent to the public forum of discussions[13].

[12] https://www.scala-native.org/en/latest/
[13] https://github.com/gvolpe/trading/discussions

FUNCTIONAL EVENT-DRIVEN ARCHITECTURE

POWERED BY SCALA 3

EXPLORE THE EVENT-DRIVEN ARCHITECTURE (EDA) IN A PURELY FUNCTIONAL WAY, MAINLY POWERED BY FS2 STREAMS IN SCALA 3.

THROUGHOUT THE CHAPTERS, WE WILL DEVELOP A DISTRIBUTED SYSTEM THAT MEETS THE REQUIREMENTS OF A MODERN SOFTWARE ARCHITECTURE CAPABLE OF PROCESSING BILLIONS OF EVENTS PER DAY AT SCALE USING APACHE PULSAR.

IT ALSO INCLUDES A WEB SOCKETS SERVICE POWERED BY HTTP4S, AND TWO WEB APP LICATIONS WRITTEN IN ELM AND SCALA.JS, RESPECTIVELY.

ALTHOUGH THE APPLICATION PICKS A PARTICULAR DESIGN AND IMPLEMENTATION, THE CONCEPTS SHOULD EASILY TRANSLATE TO OTHER DESIGNS IN THE SAME SPACE THAT CAN BE BUILT ON TOP OF APACHE KAFKA, RABBIT MQ, OR OTHER MESSAGE BROKERS.

GABRIEL VOLPE IS A SOFTWARE ENGINEER, SPECIALIZED IN FUNCTIONAL PROGRAMMING, FROM BUENOS AIRES, ARGENTINA. HE IS THE AUTHOR OF PRACTICAL FP IN SCALA, AND NOWADAYS KEEPS HIMSELF BUSY WRITING HASKELL & SCALA, WHILE OBSESSING OVER REPRODUCIBLE BUILDS VIA NIX.