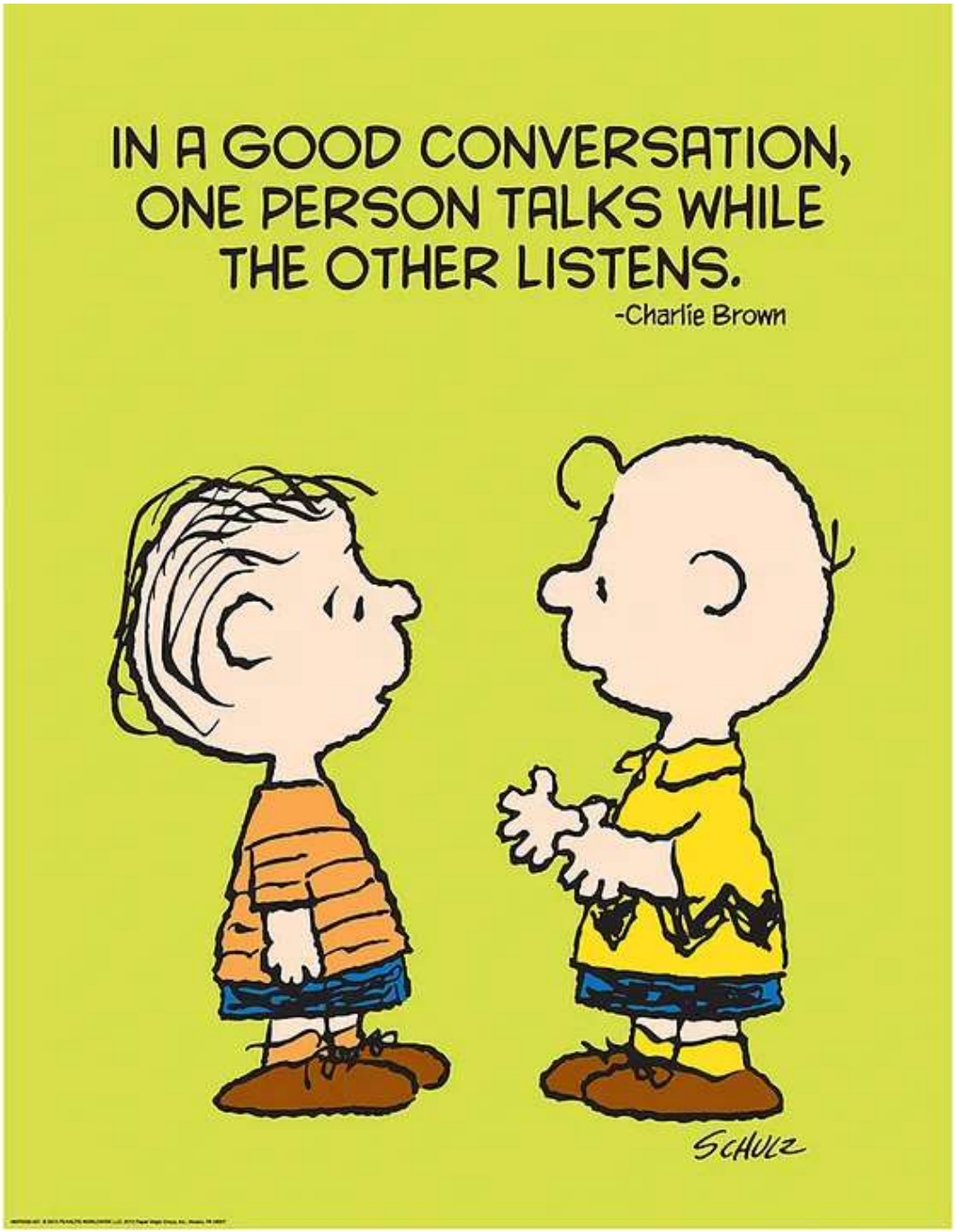
Event Driven Systems

Omar Elgabry · Follow

Published in OmarElgabry's Blog · 14 min read · Mar 31, 2019

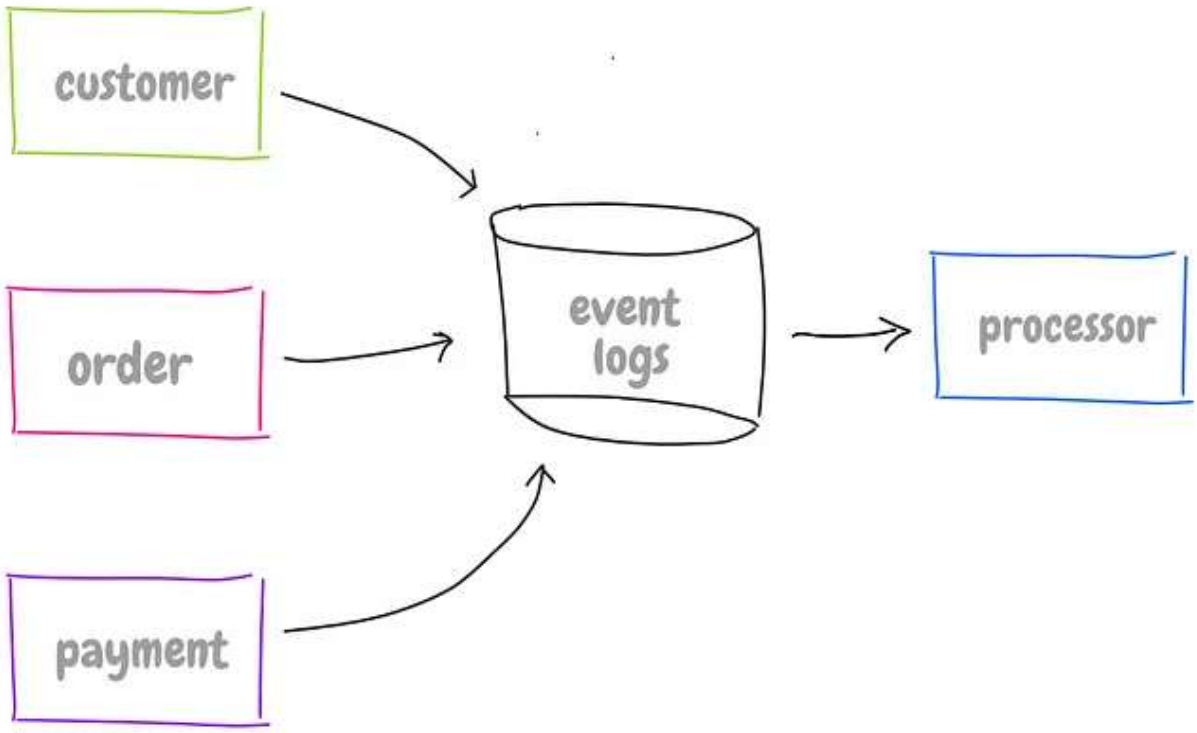


Source

An event is something happened that our application needs to react to. Changing the customer address, making a purchase, or calculating the customer bill, are all events. These events might come from the external world or triggered internally such as having a scheduled job that is being executed every some time.

And the essence here is to capture these events and then process them to cause changes to the application in addition to storing them as an audit log.

The workflow



The figure above shows three components: **Customer**, **Order** and **Payment** services. Each of these services receives requests and convert the inputs into a stream of events which are then stored in a persistent log. Each event is encapsulated into an event model, an event object.

The **event processor** is what reads the events from the log and processes them causing our application to do whatever it's supposed to do.

> *The event log can be split into muliple logs and use a separate processor for them.*

While most of the interactions are asynchronous, that is not always the case. A request sent to the event processor directly demanding the customer address change and waiting for a response can be a synchronous task.

Event Data

Events flow in the application carrying some data along. Since each event represents something that has happened in the past then it makes sense to be immutable. That is extremely important, particularly when using the event logs to keep tracking of changes to our application.

However, we may also attach some mutable data that has no side effects such as the **result of the processing**.

For instance, immutable data would include the items ordered by a customer and how much did they cost at that time, while mutable data can include the confirmation statement generated as a result.

It is also worth recording the **time** the event was captured by our application and the time the event was processed. Both might, and will, be different.

Lastly, storing the **previous state** of the application in the event object might be handy when reversing that event — *We'll get into reversing events later.*

Benefits?

We've got a log of events provides a full record of what happened and can also be used for debugging.

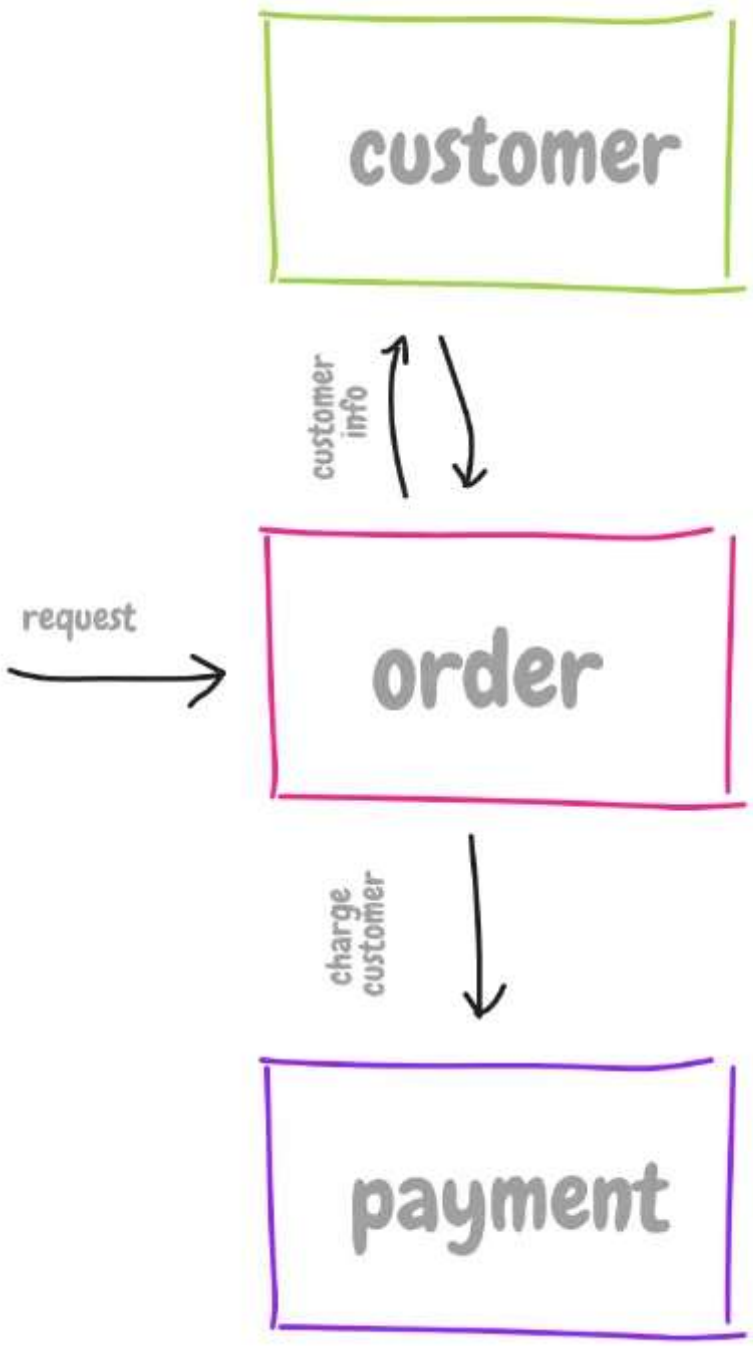
We can easily know what happened in the past at any given point of time.

Now, because these components, these services work together, that brings us to a question: How can they interact by sending and listening to events?.

Collaboration

Most of the interactions are being driven by requests. One component needs information from another so it *queries* it asking for that information. Or, perhaps, a component might issue a *command* to another demanding a change or an action to be taken.

In the event-driven world, that works differently. Instead of components talking to each other directly, they emit events that something has happened. Other components can then listen to these events if needed and react accordingly.



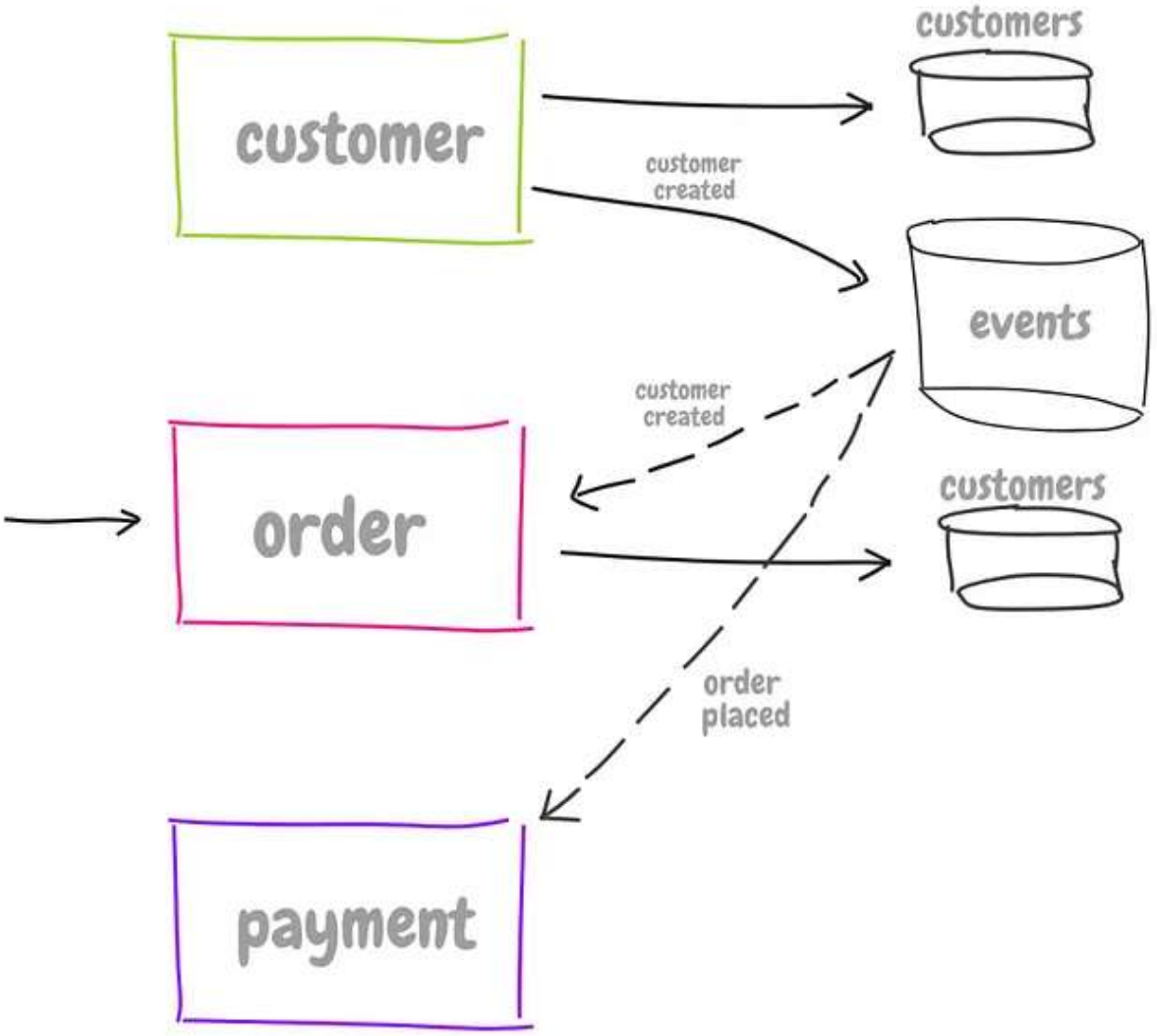
Going back to our example.

1. A request is sent to the order service to place an order.
2. The order will then *query* the customer service asking about customer information such as credit card, address, etc.

3. It then sends a *command* to the payment service to charge the customer.

There are two different kinds of interactions occurred in this example. The *query* that's being sent from order to customer service and the *command* from order to payment service.

In this pattern, services are coupled as they need to know to talk to each other. If one service is down, that might stop the whole application.



On the other hand, when working with events, we reverse the relationship; Instead of having order service talking to payment service, payment service listens to events emitted by the order service.

Similarly, order service listens to events emitted by customer service when customer data has been changed, and keep a note of it.

These events are saved in the event log.

This loose coupling allows the sender to broadcast the event without having to worry about who is interested and who should act. Any new component can hook up and listen to events, and so the existing component doesn't need to know about that new component, nor it is location.

Commands vs Events

Commands are imperative, in the present, can be rejected, don't get recorded, triggered by external request (user).

Events are declarative, happened in the past, immutable (can't be changed), and can be understood by all stakeholders (descriptive text).

If component A does care about having component B to do something, then this is a command. If, however, component A emits a change happened and *doesn't care* about who's interested in that change, that's an event.

Queries vs Events

Queries are when a component explicitly asks another for a piece of data.

In events, one component doesn't ask the other, instead, every component broadcasts events when something changes. And then, components that are

interested in that data keep a note of what it needs and ignore the rest.

Therefore, the order service doesn't need to query the customer service because it kept a note of the last event and thus knows the customer data. Hence, we reduced the calls to customer service.

Consequently, data might be duplicated across different services. It is no longer the case where all customer information resides in one place, it is also placed in other components.

It is up to every component to structure the data, and what data to store and what to ignore.

If a centralized source of data is necessary, we'll need to (1) emit events to update the data. When done, (2) broadcast the updates to components to act on.

Others might just store the log of events attached to the domain object itself, and there is nothing wrong about that. For example, the order object will have a property called "history", an array of all the changes.

```
customer = {
  history: [
     { orderId: 31, event: PLACE_ORDER, date: '12/05/2018' }
  ]
}
```

As a result, every component can work on its own even when other components are down, while when issuing a query or command, we can't do that without a connection to the recipient component.

If customer service went down, order service doesn't care because it has its own copy — *An event queue will deliver the messages when customer service is up again.*

However, we increased availability but consistency is a problem. Order service might not have the latest information in customer service (outdated data) when customer service is down. And so it may take actions based on out of date data. But, again, with requests, it would just not work — *which might be preferable in some scenarios.*

The other downside is, even though we have got decoupling but it gets hard to explain what exactly happens when an event is emitted. Who is listening? What happens next?. We have to look closely to logs and it gets harder when the application contains many services. With requests, it's easy to spot and see that a call from one component goes to which other components.

> *One might use a visualization tool that uses the configurations and build the event chain to get a better idea of what's happening.*

Event collaboration opens the gate to Event sourcing.

Event sourcing

Event sourcing is not only about having a sequence of events that one can query and look at but it also allows us to …

- Know what happened at a specific point of time (by **querying**).
- Find out the current state of the application, and also **how we got there**.
- **Replay** the history of all (or some of) events.
- **Revert** back to an event X by reversing all subsequent events.
- **Ignore** an incorrect event X by reversing it and all subsequent events and then replaying all the subsequent events without X. Or, by throwing away the application state and replaying all events in sequence without that X.
- Blow the application state, replay the events, and **rebuild** the application again.
- We can **predict** the future based on the list of events that resulted in the current state.

A common example of an application that uses event sourcing is a version control system.

Snapshots

One obvious problem is when the event log grows and becomes slow to query. A common solution is to take snapshots by storing the current application state, say, every night.

And so, we can cache the last snapshot in memory and use it throughout the day to build the current application state. On failure, we can quickly rebuild the application from the event logs in the persistent database.

Alternatively, we can also have two systems running at the same time. If one failed, the second takes over.

External Systems

Things get complicated when working with external systems. The payment service perhaps relies on external systems to query the exchange rate and charge the customer's credit card.

And since these external systems don't know the difference between a real request and a replay, a common solution is to introduce a *gateway*. It acts as a middle layer between our application and the external systems.

During the rebuild and replays, the gateway must not forward the requests to the external systems (disabled mode). It should differentiate between an actual request and a replay. And It should do so without having the application to worry about it.

Queries that are likely to result in different values depending on the time like exchange rate can be stored in the gateway. This means that every or some of the responses from external systems need to be stored in the gateway, and returned upon replay.

Code / Schema changes

Code changes can be lumped under either new features, bug fixes, change logic.

— New features

They usually add new capabilities to the system but don't invalidate old events. These will get executed as new events come in. To apply these new features to old events, we can just reprocess all old events.

— Bug fixes

They are just a matter of fixing the bug and reprocessing all buggy events. However, if an external system is used, that might be tricky to solve.

For example, If the payment service relies on external service to charge the customer's credit card. Now, say they charged the customer more than normal fees due to a bug in the code. A solution is by emitting a new event to refund the extra withdrawn money.

Alternatively, we can reverse the buggy event, and insert the correct one. So, we would revert all events until the buggy event (branch point). Then, we reverse it, mark it as deleted (but don't actually delete it) and insert the corrected one.

Some of these fixes have to be done manually.

— Code Logic

Changing the logic or business rules require to process old events with old rules and new event with the new rules. One way is to have a conditional statement — *If before a certain date do A else do B.* That solution can get very messy. And so, one can have rules encapsulated inside an object and mapped to dates.

Say, when placing an order, we charge the customer for an extra amount of 1.5% taxes. Starting from February 2018, we charge customers 1.8%. So, here are the steps:

```
// Event
// Can be implemented by various different events.
class Event {
  private Domain domain;

  trigger() {
     // get the rule assoc. with that event and date,
     // the rule object will then process the event
     this.getDomain().getRule(this, this.date).process(this);
  }
}
```

```
// Order
class Order implements Domain {
  // RulesByEvent: {Event => {Date => Rule}}
  private Map rulesByEvent = new HashMap();
  public void addRule(Event e, Rule rule, Date date) {
      if (rulesByEvent.get(e) == null)
          rulesByEvent.put(e, new HashMap());
      rulesByEvent(e).put(date, rule);
  }

  public Rule getRule(Event e, Date date {
     return rulesByEvent.get(e).get(date);
  }

  public placeOrder(Customer customer, double amount){
     // ... place order using the correct arguments
  }
}

// Rules
class TaxRule {
  private double percentage = 1.5;  // default
  TaxRule (double percentage) {
     this.percentage = percentage;
  }

  // This method calls the Order object passing the correct amount
  public void proces(Event e) {
    double amount = e.amount + (e.amount * this.percentage);
    e.getDomain().placeOrder(e.customer, amount);
  }
}

// Test
Order order = new Order();
order
// from february 2018, apply the new tax rule
.addRule(PlaceOrderEvent, new TaxRule(1.7), new Date(2018, 2, 1));

(new PlaceOrderEvent(...)).trigger();  // will place the order
```

1. A new order has been placed by a customer, and so an event was triggered.
2. The event object calls the `Rule` based on the event itself and the date. The rule here represents the different tax percentages.

3. The `Rule` object calculates the amount with the tax percentage and calls the order service passing the correct arguments. This calculation is again done with respect to the event date.

> *The Rule objects are called strategy objects. The idea comes from being able to add or update rules without changing the Order class.*

What happens when **data schema** changes?. It is either not to mutate the schema fields (name, age, gender, etc), and extend it by adding new optional fields. This way old events won't break the code.

Another solution is to migrate the old events to new ones. For sure, we could use If-else conditional statements, but again, that can get really messy.

Identification, as part of the schema, can also cause troubles on replaying events. If the events are associated with customer Ids, every time we replay, make sure we use the same id for the same customer. That's because if we replayed the events in a different order, or ignored some, maybe for testing, customers might have a different id, and hence events won't be applied on the respective customers.

A common solution is to use *stickyIds* (immutable). It means every time we create a customer, we assign an id, and that Id will be associated with the customer's email (unique). So, upon replay, every created customer will get the same id given the email.

Concurrency

This problem is rather a common problem and happens everywhere, but worth mentioning here as well.

Two requests may arrive, one to deposit money, and another to withdraw. Having two treads querying the same data at the same time may result in The lost update problem.

Good consistency can be achieved by:

1. Pessimistic locking: prevents any further commands until the current one gets saved. But, it is slow, and we need to handle timeouts, etc.
2. Sync queue that processes one command at a time.
3. Optimistic locking by checking the latest version of the data being updated just before saving it. In other words, reading the record again and check if any changes were made to it since the last time you read it.

Benefits & Drawbacks?

- An **audit log** that we can use to dig to events log and see what happened.
- Get a glimpse of **historical states** and know what happened in past.
- Not only does the event log make it easy to **debug**, but it also allows us to create a separate **test environment** and replay the events into the test with the ability to stop, rewind, and replay just like using a debugger.
- **Branching** becomes fairly simple by running and replaying selective events in a separate environment. Usually, when comparing two states to fix problems.
- Running an entire application **in memory** is only possible if you can guarantee that the system can be rebuilt on failure. We can keep the application state data in memory, and on failure, we can rebuild the application from the event logs in the persistent database.

An example of this would be a cluster of servers with in-memory databases, kept up to date with each other through a stream of events. If updates are needed, they can be routed to a single master server which applies the updates to the persistent database and then broadcasts the resulting events.

- To increase **availability**, we can also have two systems running at the same time and kept in sync, if one failed, the second will take over.

Though, …

- When relying on an external system, it makes it hard to rebuild the application state.
- Changing application code, schema, identifiers, fixing bugs, can get hard to maintain, especially when using external systems.
- Inconsistency as a result of concurrency issues, Or, one service has outdated data for a while as it has not captured the new changes yet.
- The idea of event messaging is widely used but event sourcing is an odd pattern.

You don't always need event sourcing, you could also do this with more regular logging tool, and that way not have to deal with these pitfalls.

Parallel Systems

We've talked about being able to create a separate test environment and branches in the benefits above.

This gives us the ability to view the system using different paths. Maybe, we want to replay all events but remove, add, or modify some. Or, maybe we

want to fix the system state but don't want to mess with the currently running one, so we fork, fix, and merge back.

It is the same idea as with branching in Git.

So, how can we implement it?.

We can simply take a copy of the existing system, and have another separate independent parallel system running at the same time. The benefit is the change in one system doesn't affect the other.

The problem is, if we want to replace the existing system, we still need to integrate the forked one to the existing running one.

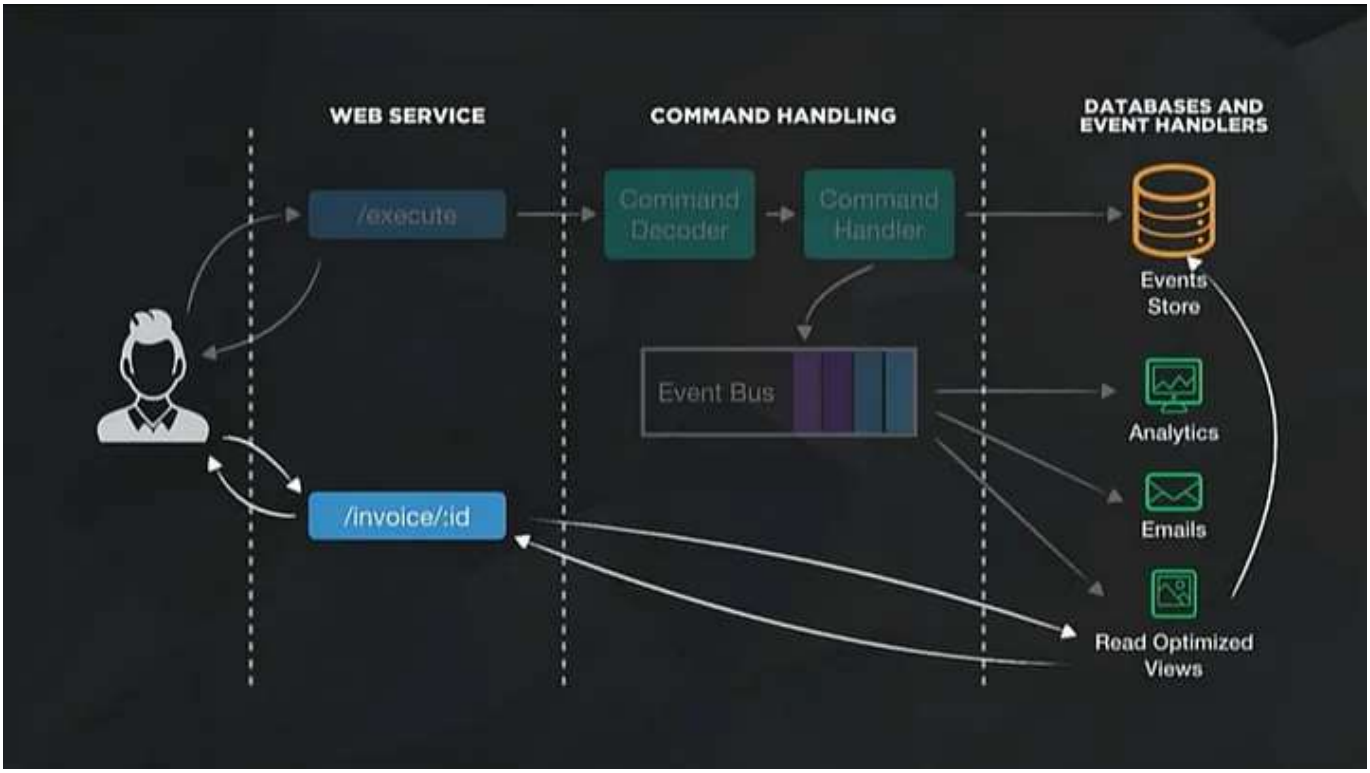
The other way is to have one running system but can point to two databases. In addition, it should be easy to switch between the production and testing databases. The drawback is since both share the same code base, changing it will reflect on both views: The current running system and the forked one.

CQRS

Event sourcing goes hand in hand with the Command Query Responsibility Segregation (CQRS) pattern.

It is about segregation the operations WRITE from READ. And so if the read ratio is much more than write, so splitting them to scale each independently. Moreover, each has a different schema, backed by different databases, services, etc.

How event sourcing fits into CQRS world?



Source

The WRITE component handles events, and stores these events which are used to create different views: different ways of representing data (reports, analysis, etc). These views can be queried later through READ component.

> *Read is immutable, denormalized, and based on how often the user query the data. Write is normalized, validated (before save), good for updates, and not optimized for querying.*

An obvious pitfall is these different views might be inconsistent for a while until they are all updated with the latest changes. This is called eventual consistency; data becomes consistent will become consistent at a point in the future. We've talked about that before.

One of the solutions is to compare the last version of the view and the event log, if there's a gab, update the view, and return the result.

References:

- Event Sourcing
- Event Sourcing in Node.js with Microservices
- YOW! Nights March 2016 Martin Fowler — Event Sourcing #YOWNights
- The Many Meanings of Event-Driven Architecture • Martin Fowler

And finally, … it doesn't go without saying,

Thank you for reading!

Feel free to reach out on LinkedIn or Medium.

Thoughts | Event Driven Systems | Software Architecture | Software Engineering | Software Development

Written by Omar Elgabry

Follow

8.5K Followers · Editor for OmarElgabry's Blog

Software Engineer. Going to the moon. When I die, turn my blog into a story.
@https://www.linkedin.com/in/omarelgabry

More from Omar Elgabry and OmarElgabry's Blog

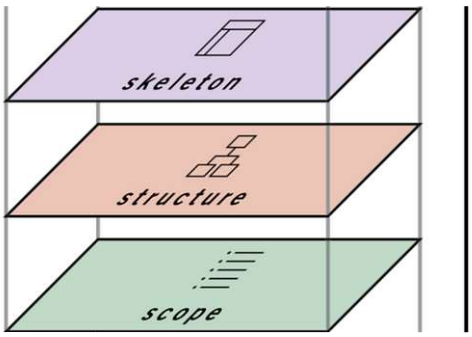
Omar Elgabry in Towards Data Science

The Ultimate Guide to Data Cleaning

When the data is spewing garbage

15 min read · Feb 28, 2019

3.3K 10



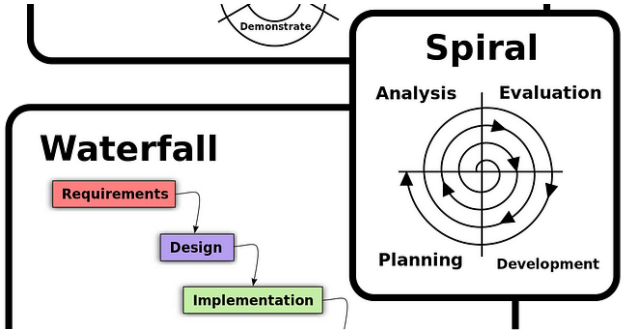
Omar Elgabry in OmarElgabry's Blog

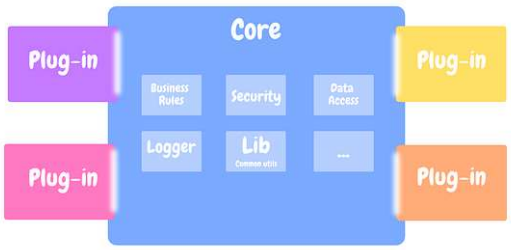
UX — A quick glance at The 5 Elements of User Experience (Par...

The Five Elements of UX came from “The Elements of User Experience” book written b...

5 min read · Sep 15, 2016

3.3K 19





Omar Elgabry in OmarElgabry's Blog

Software Engineering — Software Process and Software Process...

Understand the software process and software process models.

13 min read · Mar 17, 2017

4.1K 16

Omar Elgabry in OmarElgabry's Blog

Plug-in Architecture

and the story of the data pipeline framework

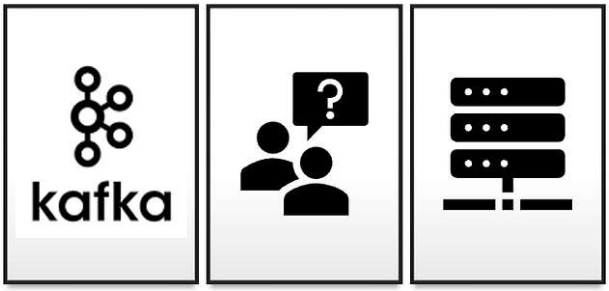
9 min read · May 1, 2019

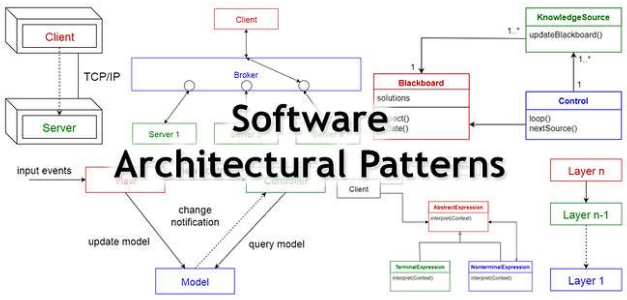
1.1K 3

See all from Omar Elgabry

See all from OmarElgabry's Blog

Recommended from Medium





Amit Sharma in Engineered @ Publicis Sapient

Event-Driven Architecture at Scale Using Kafka—Part2

How to decide on the right Kafka cluster option?

4 min read · Jul 3

1

Vijini Mallawaarachchi in Towards Data Science

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major softwar...

· 5 min read · Sep 4, 2017

39K 127

Lists

General Coding Knowledge

20 stories · 677 saves

Stories to Help You Grow as a Software Developer

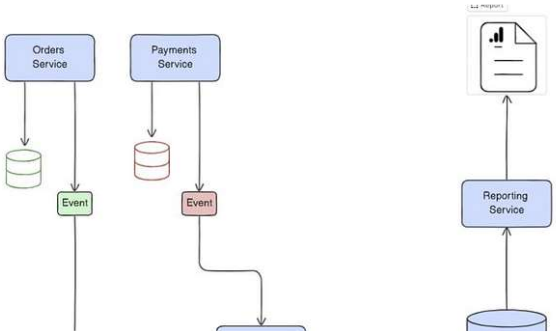
19 stories · 630 saves

Leadership

39 stories · 178 saves

Coding & Development

11 stories · 319 saves

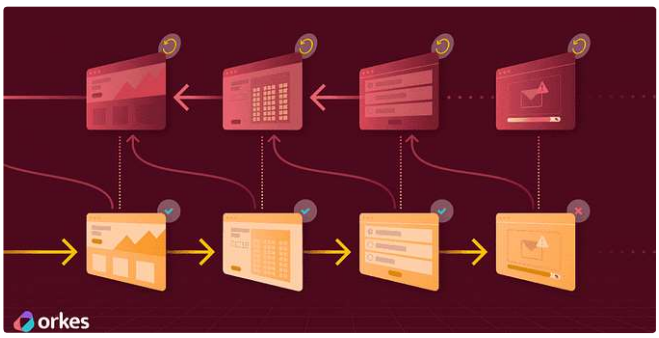


Ian Kiprono in Stackademic

Handling Reporting on Microservices Architecture

Ask around and most developers will tell you how frightening Microservices are. It should...

2 min read · Oct 26



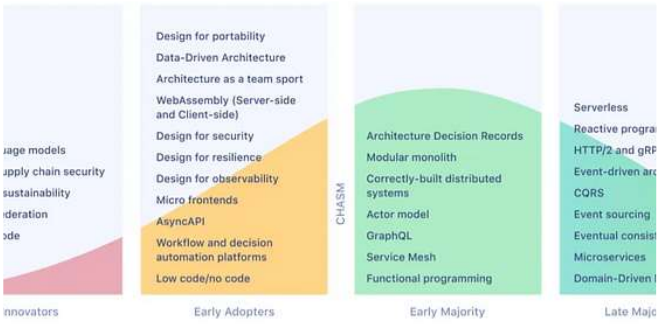
Riza Farheen in Orkes Tech Blog

Saga Pattern in Distributed Systems

Distributed systems allow for creating architecture that is easy to maintain and...

8 min read · Jul 24

52 16



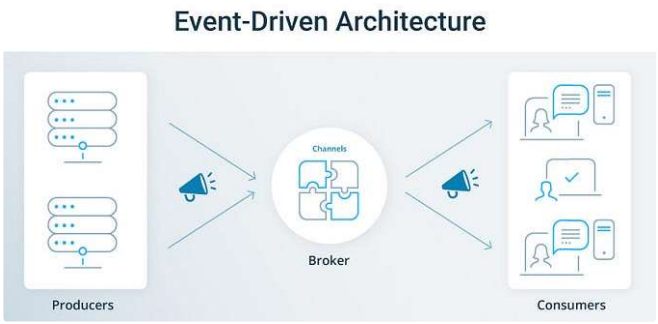
Daniel Foo

Software Architecture and Design Trend 2023

2023 is almost coming to the end. It's always a good idea to reflect back on what has been...

7 min read · Oct 1

1.2K 7



Seetharamugn

The Complete Guide to Event-Driven Architecture

What is Event-Driven Architecture?

8 min read · Aug 29

30

See more recommendations