An In-Depth Look at Event Sourcing with CQRS

Sebastian von Conrad
@vonconrad
Agenda

‣ Event Sourcing

‣ Event Sourcing with CQRS
  › With a lil’ dose of DDD

‣ Upside

‣ Downside
  › It’s not all sunshine and rainbows
Event Sourcing at Envato
Event Sourcing at Envato

- First generation: 1 system, >3 years in production.
Event Sourcing at Envato

- First generation: 1 system, >3 years in production.
  - Extracted framework.
Event Sourcing at Envato

‣ First generation: 1 system, >3 years in production.
  ▪ Extracted framework.

‣ Second generation: Half-a-dozen systems, in production from ~1 year to now.
Event Sourcing at Envato

- First generation: 1 system, >3 years in production.
  - Extracted framework.

- Second generation: Half-a-dozen systems, in production from ~1 year to now.
  - Used framework from first generation.
Event Sourcing at Envato

• First generation: 1 system, >3 years in production.
  • Extracted framework.

• Second generation: Half-a-dozen systems, in production from ~1 year to now.
  • Used framework from first generation.

• Third generation: Being developed now.
Event Sourcing at Envato

‣ First generation: 1 system, >3 years in production.
  ‣ Extracted framework.

‣ Second generation: Half-a-dozen systems, in production from ~1 year to now.
  ‣ Used framework from first generation.

‣ Third generation: Being developed now.
  ‣ Expanding paradigm to new frameworks and languages, including FP.
Event Sourcing
What is Event Sourcing?
Many systems log “events.”
Event Logging ≠ Event Sourcing
Most systems are built around a database that stores current state.

An “event log” would sit to the side.
Event Sourcing forces current state to be *derived* from all past history.
With Event Sourcing, all current state is disposable.
Current state is a cache.

A cache can be deleted and regenerated.
Greg Young
Event Sourcing isn’t new.
At least 500 years old.

Allegedly tracked back to Ancient Mesopotamia.
Most mature industries are event sourced.
Finance, accounting, insurance, medical, legal, etc.
Bank account.
Do you think your bank account balance is simply a column in a current state database?
What if you disagree with your bank?
The balance is an equation.
balance = sum(transaction.amount)
You can manually calculate this.
Event Sourcing works the same way.
All current state is transient.
And are the result of equations.
Your source of truth are events, not state.
…so what are events?
Overloaded term.
In this context, events describe things that have happened that are of interest to you.
Events are business facts.
Events are expressed in past tense.
Bank domain events

- New Account Application Received
- Customer Identity Verified
- Account Opened
- Funds Deposited
- Funds Withdrawn
- Funds Transferred
- Account Statement Issued
E-commerce Domain events

- Item Added To Shopping Cart
- Item Removed From Shopping Cart
- Order Placed
- Payment Received
- Goods Dispatched
Calendar domain events

- Appointment Scheduled
- Appointment Rescheduled
- Invitation Extended
- Invitation Sent
- Invitation Accepted
- Invitation Declined
- Appointment Canceled
Events are self-described.
Events **completely describe** the thing that has happened.
But they are not denormalised.
Keep your events focused on only the data for which they are the authoritative source.
“Order Placed” event

- A reference to the customer who placed the order
- A timestamp for the order placement from the customer's perspective
- A fully specified shipping address for the order
- A fully specified billing address for the order
- A list of the items ordered, each of which include:
  - A reference to the type of item
  - The quantity ordered
  - The price to be charged
- Details of any taxes charged
- Details of discounts, coupons and other adjustments that were applied
- The total amount charged for the order
- A reference to the payment that covers the order
With Event Sourcing, events are the centre of your domain.

Modelling them well is crucial.
One technique: Event Storming.
(Go watch Paul Rayner’s talk.)
Okay, so, let’s put this into practice.
Shopping Cart modelled as “current state”

Could be a document, a table in a relational database, a document in a document database, or something else.
We’ve been taught that the shape of this data is the most important thing.
But it can also be expressed in events.
Shopping Cart as Events

- Cart Emptied
- Item Added To Cart
- Item Added To Cart
- Shipping Address Provided
Shopping Cart “current state”
What we end up storing, however, are the events.
Event Sourcing just a different way of storing information.
Though it has fundamental consequences.
With Event Sourcing, data is **append-only**.
With Event Sourcing, data is immutable.
With Event Sourcing, data can be **cached forever**.
So how do we deal with mistakes?
Well, how does an accountant deal with mistakes?
We call these correction events.
Shopping Cart “correction” event

Cart emptied → Item added to cart → Item added to cart → Item removed from cart → Shipping address provided
Are these the same thing?
The current state is the same!
Let me issue you a challenge.
In your system:

Two use cases that result in the same current state.
If you can, you’re losing data.
Follow-up question:

How do you determine which data you’re happy to lose?

(Can you teach me?)
I have only one rule: *never lose data.*

Because I don’t know how to value it.
I can’t predict its future value.
This is the main benefit of Event Sourcing.
CEO: “Let’s sell people some things they were interested in but didn’t end up buying.”
Adding to current state model
Shopping Cart “correction” event

Cart emptied → Item added to cart → Item added to cart → Item removed from cart → Shipping address provided
We can write a **projection**.

(More on this later.)
Starts from event 0. May take a while. That’s fine, it’s async.
We’re not losing data.

And we can come up with new and interesting ways to interpret it.
We also get a time machine.
Want to know what that report looked like at 2016-05-03 21:16:09 UTC?

No problem.
An event sourced system is deterministic.
Easy to go back to previous state, e.g. when a bug was occurring.
Sure, sure, that’s all well and good but *it can’t possibly scale!*
What if I have a lot of events?
Event Sourcing with CQRS
Command Query Responsibility Segregation
Effectively, separating *reads* from *writes*. 
It’s possible to use Event Sourcing without CQRS.
It’s possible to use CQRS without Event Sourcing.
They just go *really* well together.
Four major building blocks to the architecture

‣ Event Store
‣ Reading with Queries
‣ Writing with Commands
‣ Reactors
Event Store.
Events are kept in a database called the “Event Store.”
The Event Store is usually considered a sub-system in its own right.
There are **commercial** and **open-source** options available.
But it’s easy enough to roll your own.

(We have.)
The simplest event store has a single event “stream.”
The simplest Event Store has three features:

- Put event.
- Get all events, in order, from an offset.
- Get all events, in order, scoped by <something>.
These are the features we’ll make use of.
Reading with Queries.
Clients need to read (and display) information.
Clients are *usually* interested in *current state*. 
The Event Store is not suitable for presenting current state.
So we need something that Clients can **query** for current state.
We call this a **Query Handler**.
It’s a little detour.
The Query Handler “tails” the event stream from the Event Store.
...and builds up a projection.
When a Client wants to know something, the query handler use the projection.
And returns the data to the Client.
There are many (N) projections, each optimised for a particular “query.”
The Query Handler will pick a projection for a query from a client.
Each of which has their own “tailing” process called a projector that read from the stream.
This is **not** using the Observer pattern.
We use the following Event Store feature(s):

- Put event.
- Get all events, in order, from an offset.
- Get all events, in order, scoped by <something>.
The projectors keep a “pointer” to where in the event stream they’re at.
The projections are **disposable** and can be rebuilt by going back to 0.
The projectors and projections are completely decoupled from each other.
This allows different data storage options to solve different problems.
Relational DB  Cache
Document DB  Graph DB
Search Engine  Flat files
Data Warehouse  …
We can optimise for how we want to *read* the data.
“Current” current state read models tend to be normalised, which is optimised for writing data.
Projections can be (and often are) denormalised.
Projections are cheap.
New interpretation?
New projection!
When caught up, we can **swap** to reading from the new projection.
And then we can decommission the old projection.
No more database migrations on production tables.
Summary: Reading with Queries
Summary: Reading with Queries

› Clients never query the events directly.
Summary: Reading with Queries

- Clients never query the events directly.

- Clients query denormalised *projections* that are optimised for querying.
Summary: Reading with Queries

‣ Clients **never query the events** directly.

‣ Clients query denormalised **projections** that are optimised for querying.

‣ Projections are built with **projectors** that process the event stream.
Summary: Reading with Queries

‣ Clients *never query the events* directly.

‣ Clients query denormalised *projections* that are optimised for querying.

‣ Projections are built with *projectors* that process the event stream.

‣ Projectors are *decoupled* from each other and don’t share any state.
Summary: Reading with Queries

- Clients never query the events directly.

- Clients query denormalised projections that are optimised for querying.

- Projections are built with projectors that process the event stream.

- Projectors are decoupled from each other and don’t share any state.

- Projections are cheap and easy to build and rebuild.
Writing with Commands.
In addition to reading, Clients also want to write.
We established that Clients should not query the Event Store directly.
Nor should Clients write directly to the Event Store.
The Event Store is not responsible for validation, and so giving Clients direct access is dangerous.
It would be like opening your SQL database to the world.
Instead, Clients express their intentions in the form of **Commands**.
Commands represent intent.
Just like we had something responsible for handling queries, we need something responsible for handling commands.
It’s called a Command Handler.
Commands express **intent**.
Event and command naming

Events (past tense)

- Account Opened
- Funds Deposited
- Funds Withdrawn
- Funds Transferred
- Account Statement Issued
Event and command naming

Events (past tense)

- Account Opened
- Funds Deposited
- Funds Withdrawn
- Funds Transferred
- Account Statement Issued

Commands (imperative)

- Open Account
- Deposit Funds
- Withdraw Funds
- Transfer Funds
- Issue Account Statement
Commands represent intent and can be rejected.

Events are facts and cannot.
A Command that is accepted results in an event.
So how does the Command Handler validate Commands?
This is where we start borrowing from DDD.
DDD doesn’t come into play on the Query side, but it does on the Command side.
Events happen to “something.”
We call that “something” an **Aggregate**.

This is a DDD term.
The Aggregates are the **nouns** in our system.
Bank domain events and aggregates

Events

- New Account Application Received
- Customer Identity Verified
- Account Opened
- Funds Deposited
- Funds Withdrawn
- Funds Transferred
- Account Statement Issued
Bank domain events and aggregates

**Events**

- New Account Application Received
- Customer Identity Verified
- Account Opened
- Funds Deposited
- Funds Withdrawn
- Funds Transferred
- Account Statement Issued

**Aggregates**

- Account Application
- Customer
- Account
- Transfer
- Account Statement
E-commerce domain aggregates

- Customer
- Shopping Cart
- Order
- Payment
- Product (or Item)
- Shipment
Calendar domain events

› Calendar

› Appointment

› Invitation
Aggregates have **identity** that we need to track over time.

Often, they have lifecycles.
Aggregates can refer to each other.

E.g. Order => Customer.
When a Command is received, it is executed against an Aggregate.
The Aggregate determines whether to accept or reject the Command.
To do that, the Aggregate needs to know its current state.
events = EventStore.load(aggregate_id)
events = EventStore.load(aggregate_id)

aggregate = Aggregate.new
events = EventStore.load(aggregate_id)
aggregate = Aggregate.new
aggregate.replay_events(events)
events = EventStore.load(aggregate_id)
aggregate = Aggregate.new
aggregate.replay_events(events)
aggregate.attempt_command(payload)
events = EventStore.load(aggregate_id)
aggregate = Aggregate.new
aggregate.replay_events(events)
aggregate.attempt_command(payload)
# => true
events = EventStore.load(aggregate_id)
aggregate = Aggregate.new
aggregate.replay_events(events)
aggregate.attempt_command(payload)
# => false
Let’s say we want to withdraw some funds.
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)

events = EventStore.load(command.account_id)
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)

events = EventStore.load(command.account_id)
# [ FundsDeposited, FundsDeposited, FundsWithdrawn ]
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)

events = EventStore.load(command.account_id)
# [ FundsDeposited, FundsDeposited, FundsWithdrawn ]

account = Account.new
account.replay_events(events)
class Account

end
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end
end

  apply FundsDeposited do |event|
    self.balance += event.amount
  end
end
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end

  apply FundsDeposited do |event|
    self.balance += event.amount
  end
end
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end

  apply FundsDeposited do |event|
    self.balance += event.amount
  end

  apply FundsWithdrawn do |event|
    self.balance -= event.amount
  end
end
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)

events = EventStore.load(command.account_id)
# [ FundsDeposited, FundsDeposited, FundsWithdrawn ]

account = Account.new
account.replay_events(events)
command = WithdrawFundsCommand.new(
    account_id: "6504630d-6ceb-4650-b516-5e720d3db941",
    amount: 500)

events = EventStore.load(command.account_id)
# [ FundsDeposited, FundsDeposited, FundsWithdrawn ]

account = Account.new
account.replay_events(events)

account.withdraw(command.amount)
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end

  apply FundsDeposited ...
  apply FundsWithdrawn ...

  def withdraw(amount)
    return false if amount > self.balance
    create_event(FundsWithdrawn, amount: amount)
    return true
  end
end
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end

  def withdraw(amount)
    return false if amount > self.balance
  end
end
class Account
  def replay_events(events)
    events.each do |event|
      apply_event(event.type, event.body)
    end
  end

  apply FundsDeposited ...
  apply FundsWithdrawn ...

  def withdraw(amount)
    return false if amount > self.balance

    EventStore.store(FundsWithdrawn, amount: amount)
    return true
  end
end
We use the following Event Store feature(s):

- Put event.
- Get all events, in order, from an offset.
- Get all events, in order, scoped by <something>. 
We use the following Event Store feature(s):

- Put event.
- Get all events, in order, from an offset.
- Get all events, in order, scoped by `aggregate_id`. 
Isn’t this slow?
It *can* be.
Keep the number of events per aggregate low.
An Aggregate that accumulates an unbounded number of events over time is a smell.
Wherever possible, have fixed life Aggregates.
We would not put OrderPlaced events under the Customer Aggregate.

Instead we have an Order Aggregate.
Domain modelling is hard.
Event sourcing doesn’t change that.
If you screw this up and find yourself with too many events per aggregate, there is another tool you can use.
Snapshots.
It’s effectively the **Memento** pattern from the Gang of Four book.

Design Patterns: Elements of Reusable Object-Oriented Software

*Gamma, E; Helm, R; Johnson, R; Vlissides, J.*
Snapshots
Snapshots
Snapshots
Snapshots are a purely technical optimisation and play no role in the conceptual model of event sourcing.
Summary: Writing with Commands
Summary: Writing with Commands

- Clients never write the events directly.
Summary: Writing with Commands

‣ Clients **never write the events** directly.

‣ Clients express an intent to do something via **commands**.
Summary: Writing with Commands

- Clients **never write the events** directly.

- Clients express an intent to do something via **commands**.

- Commands are validated by **Aggregates**, which is a concept borrowed from DDD.
Summary: Writing with Commands

‣ Clients **never write the events** directly.

‣ Clients express an intent to do something via **commands**.

‣ Commands are validated by **Aggregates**, which is a concept borrowed from DDD.

  ‣ Aggregates fetch events from the Event Store, and replay them to reconstitute their current state.
Summary: Writing with Commands

- Clients never write the events directly.
- Clients express an intent to do something via commands.
- Commands are validated by Aggregates, which is a concept borrowed from DDD.
  - Aggregates fetch events from the Event Store, and replay them to reconstitute their current state.
  - If the Aggregate accepts the Command, it results in an event.
Summary: Writing with Commands

‣ Clients **never write the events** directly.

‣ Clients express an intent to do something via **commands**.

‣ Commands are validated by **Aggregates**, which is a concept borrowed from DDD.
  
  ‣ Aggregates fetch events from the Event Store, and replay them to reconstitute their current state.

  ‣ If the Aggregate accepts the Command, it results in an event.

‣ If replaying gets slow, performance can be improved by **snapshots**.
Where does this leave us?
The Circular Architecture
Clients express intent through Commands.
Commands that are accepted result in **Events** persisted to the Event Store.
Projectors tail the Event Store and builds up projections.
Clients retrieve data from the projections via Queries.
Reads (queries) and writes (commands) are **decoupled**.
Reads and writes can scale independently.
Great if you’re **read-heavy**.

(We are.)
An In-Depth Look at Event Sourcing with CQRS

© 2017 Sebastian von Conrad
Writes are usually fast, and always consistent.
Reads are fast, and eventually consistent.

(We’ll talk more about this later.)
Reactors.
Reactors are the last major building block.
If Commands are for Client writes…
…and Queries are for Client reads…
…then Reactors are for all of the business logic in between.
Extending the Circular Architecture.
Reactors **process events**, just like projectors.
Reactors also keep a “pointer” of where they are.
(In fact, you can argue that projectors are a type of reactor.)
Instead of building projections, they react to events.
Either through triggering external behaviour...
…or by emitting new events back to the Event Store.
...or both!
Example: Raising events at a different level of abstraction.
A Reactor reads the event stream and “subscribes” to Order Placed events.
When it detects the $N$th order from a customer, it triggers its business rules.
It knows that this Customer should be promoted to gold, and emits an event.
This Reactor is **stateful**.

Many (but not all) Reactors are.
It holds *reference data*, which we also sometimes call *internal projections*. 
The Reactor watches the stream of events, and **reacts when its conditions are met.**
Example: **Undertake async side effects** that logically follow from an event.
Another Reactor subscribes to the Gold Class Promotion event.
It interacts with an email server to send a welcome email to the customer.
It then emits an event indicating the action it’s just taken.
This Reactor **executes a side effect** from something that has happened, and then emits an event to that effect.
Given that it’s the only responsibility of the Reactor, is it still a side effect?
What matters is that we get an **audit trail** of all the actions the system has undertaken.
We use the following Event Store feature(s):

- Put event.
- Get all events, in order, from an offset.
- Get all events, in order, scoped by aggregate_id.
Reactors are **decoupled** from each other.

(As are projectors.)
Reactors run as single-threaded processes.
Like projectors, Reactors are asynchronous and subject to Eventual Consistency.
Reactors typically **do not share code.**
Reactors generally do not share any state.
Reactors do not share their “internal projections.”

Nor do they access any of the “external projections” that Clients use.
If a Reactor needs state, it should be responsible for managing that state.
Sometimes there are exceptions.
Reactors sometimes access a canonical view.

Beware temporal coupling.
Reactors should be built so they can be deleted or lost with no loss of system integrity.
(Except delays in completing operations.)
Reactors behave a lot like Microservices.
Reactors → idempotence.

(They are not fully idempotent.)
For the most part, Reactors will not re-trigger external behaviour when rebuilt against event history.
Which means that we can reset them back to 0 as well.
Reactors should do **one thing well.**
Reactors encourage a business-oriented decomposition of the system, rather than a technical one.
Sagas are built up by reactors.
Sometimes also referred to as “Process Managers.”
Reactors are really freaking cool.
Reactors are where you find most of the complexity, whether it’s necessary complexity or accidental complexity.
Summary: Reactors
Summary: Reactors

- **Reactors** process the event stream, like projectors.
Summary: Reactors

- **Reactors** process the event stream, like projectors.

- Reactors **reacts to events** in several ways:
Summary: Reactors

- Reactors process the event stream, like projectors.

- Reactors react to events in several ways:
  - Reactors can raise new events at a different level of abstraction.
Summary: Reactors

- Reactors process the event stream, like projectors.

- Reactors react to events in several ways:
  - Reactors can raise new events at a different level of abstraction.
  - Reactors can trigger side effects that logically follow from an event.
Summary: Reactors

- **Reactors** process the event stream, like projectors.

- Reactors **reacts to events** in several ways:
  - Reactors can raise **new events at a different level of abstraction**.
  - Reactors can **trigger side effects** that logically follow from an event.

- Reactors leave an **audit trail** in the form of events.
Summary: Reactors

- **Reactors** process the event stream, like projectors.

- Reactors **reacts to events** in several ways:
  - Reactors can raise **new events at a different level of abstraction**.
  - Reactors can **trigger side effects** that logically follow from an event.
  - Reactors leave an **audit trail** in the form of events.
  - Reactors are **decoupled** from each other and don’t share any state.
Summary: Reactors

‣ Reactors process the event stream, like projectors.

‣ Reactors **reacts to events** in several ways:
  ‣ Reactors can raise **new events at a different level of abstraction**.
  ‣ Reactors can **trigger side effects** that logically follow from an event.

‣ Reactors leave an **audit trail** in the form of events.

‣ Reactors are **decoupled** from each other and don’t share any state.

‣ Reactors create the foundation for implementing **sagas**.
Summary: Reactors

- **Reactors** process the event stream, like projectors.
- Reactors **reacts to events** in several ways:
  - Reactors can raise **new events at a different level of abstraction**.
  - Reactors can **trigger side effects** that logically follow from an event.
- Reactors leave an **audit trail** in the form of events.
- Reactors are **decoupled** from each other and don’t share any state.
- Reactors create the foundation for implementing **sagas**.
- Reactors are often a **source of** necessary or accidental **complexity**.
Okay, deep breath. Time check.

Nearly there.
Recap
Event Sourcing with CQRS
Event Sourcing with CQRS

- Event Sourcing makes you store business facts as the source of truth.
Event Sourcing with CQRS

‣ **Event Sourcing** makes you store *business facts* as the source of truth.

‣ Event Sourcing makes the system *deterministic.*
Event Sourcing with CQRS

‣ **Event Sourcing** makes you store **business facts** as the source of truth.

‣ Event Sourcing makes the system **deterministic**.

‣ **CQRS** and the **Circular Architecture** works well with Event Sourcing.
Event Sourcing with CQRS

- **Event Sourcing** makes you store **business facts** as the source of truth.
- Event Sourcing makes the system **deterministic**.
- **CQRS** and the **Circular Architecture** works well with Event Sourcing.
  - Clients express intent via **commands**, which if accepted become **events**.
Event Sourcing with CQRS

- **Event Sourcing** makes you store **business facts** as the source of truth.

- Event Sourcing makes the system **deterministic**.

- **CQRS** and the **Circular Architecture** works well with Event Sourcing.

  - Clients express intent via **commands**, which if accepted become **events**.

  - Asynchronous projectors process the event stream and build up denormalised **projections**.
Event Sourcing with CQRS

‣ **Event Sourcing** makes you store *business facts* as the source of truth.

‣ Event Sourcing makes the system *deterministic*.

‣ **CQRS** and the **Circular Architecture** works well with Event Sourcing.
  
  ⁃ Clients express intent via *commands*, which if accepted become *events*.

  ⁃ Asynchronous projectors process the event stream and build up denormalised *projections*.

  ⁃ Clients *query* the projections when they want know something.
Event Sourcing with CQRS

‣ **Event Sourcing** makes you store **business facts** as the source of truth.

‣ Event Sourcing makes the system **deterministic**.

‣ **CQRS** and the **Circular Architecture** works well with Event Sourcing.

  ‣ Clients express intent via **commands**, which if accepted become **events**.

  ‣ Asynchronous projectors process the event stream and build up denormalised **projections**.

  ‣ Clients **query** the projections when they want know something.

  ‣ Asynchronous **reactors** process the event stream and reacts to events according to business logic, outputting more events.
Upside
Upside

‣ Treats the core of your business with respect.

‣ Compatibility with reality.

‣ Rapid iteration and reinterpretation.

‣ Encourages Good Practices™.

‣ Supports (some) Quality Attributes.

‣ Avoids common pain points.

‣ Lets you use the best tool for the job.

‣ You get a free time machine.
Treats the **core of your business** with respect.
You can place no more respect to your data than placing it at the core of your system and then never change it.
We’re not losing data we can’t value.
We can’t predict the future.
We don’t know how to value the data we throw away.
Events represent **fundamental business happenings** that always have been (and always will remain) of **fundamental interest** to the business.
Compatibility with reality.
Event Sourcing borrows a lot from the real world.
Just like in the real world, **time** is a first class citizen.
In the real world, the past is done and if you don’t like it, all you can do is compensate for it.
The real world is eventually consistent because there is no central control.
The real world is made up of independent actors that do the best with the information they have.
The real world doesn't have distributed transactions across autonomous agents to keep them in sync.
(There is no evidence that this is a problem, and event sourced systems don’t assume there is one either.)
In the real world, core concepts change very very very slowly.
In the real world, we frantically innovate at the *edges of systems.*
Event Sourcing, surrounded by lots of disposable and replaceable interpretations of what’s happening, is just like that.
We seek to emulate the real world.
Rapid iteration and reinterpretation.
Just because some things change quickly, doesn’t mean everything does.
The **key concepts** in business models stay relatively **stable**.
If designed well*, the rate of change for events is low.

* By designers with understanding of the business domain.
Meanwhile, we can separate recording of what happened from interpreting what it means.
Interpretations and **experiments** are easy to get rid of when no longer relevant.
Even experiments that create events can be discarded (we can ignore events).
Encourages **Good Practices™.**
Separation of concerns through isolated, asynchronous components.
If a Reactor or Projector breaks down, it only affects itself. When it comes back, it picks up where it left off.
New features tend to involve extending code, rather than changing code.
Encourages code that is expressed in business and user terms.
Removes abstractions.
Forms ubiquitous language.
Business owners *love* this.
We get unidirectional coupling.
Provides logical seams for Microservices. Slice by capability.
Supports (some) **Quality Attributes.**
Eventual consistency is inherently more **scalable** and **available** than approaches which mandate strong consistency.
Reads **scale** horizontally and can achieve **high availability**.
Writes can be very fast while remain single-threaded (with all of those benefits).
For the *integrity*-minded, consider WORM media for your Event Store.
Append-only data capturing facts provides **auditability**.
Avoids common pain points.
Database migrations.
ORMs.
Tightly coupled, lazy-loaded object graphs that are hard to understand and reason about.
Lets you use the **best tool** for the job.
Different building blocks can be written in **different languages**.
We can use the **right data storage** for individual projections.
Provides seams for **team organisation**.

E.g. one team works on the write-side, another on read-side.
You can use Event Sourcing for parts of systems, as well.
You get a free time machine.
Downside
Downside

- Eventual consistency.
- Handing over the crown jewels.
- Opacity.
- Mistakes are forever.
- It’s not mainstream.
- The J curve of productivity.
- Complexity.
- Cultures of control.
Eventual consistency.
Commands are synchronous.
Clients issue a command and receives an **ACK/NACK**.
Queries are synchronous.
Clients make a query and receive data.
Projectors are asynchronous.
It can take *some time* for changes to show up in projections.
Reactors are asynchronous too.
And because they are autonomous, changes can show up at different times.
When not all parts of a system has the same idea of current state, this is called eventual consistency.
The **real world** is eventually consistent.
Independent actors can act without consulting each other.
Example: your car getting stolen
Example: your car getting stolen

‣ Your car may get stolen right now while you’re listening to me.
Example: your car getting stolen

- Your car may get stolen right now while you’re listening to me.

- Your “copy” of the world is not in lockstep with the real world.
Example: your car getting stolen

‣ Your car may get stolen right now while you’re listening to me.

‣ Your “copy” of the world is not in lockstep with the real world.
  ‣ In your "copy" of the world, your car will be exactly where you left it.
Example: your car getting stolen

‣ Your car may get stolen right now while you’re listening to me.

‣ Your “copy” of the world is not in lockstep with the real world.
  ‣ In your "copy" of the world, your car will be exactly where you left it.

‣ Hours from now, you find out that it’s gone.
Example: your car getting stolen

‣ Your car may get stolen right now while you’re listening to me.

‣ Your “copy” of the world is not in lockstep with the real world.
  • In your "copy" of the world, your car will be exactly where you left it.

‣ Hours from now, you find out that it’s gone.
  • That’s when your “copy” of the world is made consistent.
Example: your car getting stolen

- Your car may get stolen right now while you’re listening to me.

- Your “copy” of the world is not in lockstep with the real world.
  - In your "copy" of the world, your car will be exactly where you left it.

- Hours from now, you find out that it’s gone.
  - That’s when your “copy” of the world is made consistent.

- This is eventual consistency.
Example: your bank loan
Example: your bank loan

- Your loan application may be approved by the loan department right now.
Example: your bank loan

› Your loan application may be approved by the loan department right now.

› This evening, you may discuss with your partner about whether or when the application will get approved.
Example: your bank loan

- Your loan application may be approved by the loan department right now.

- This evening, you may discuss with your partner about whether or when the application will get approved.
  - Not knowing it already has been.
Example: your bank loan

‣ Your loan application may be approved by the loan department right now.

‣ This evening, you may discuss with your partner about whether or when the application will get approved.

  ‣ Not knowing it already has been.

‣ You get a letter in the post days later, informing you of the approval.
Example: your bank loan

- Your loan application may be approved by the loan department right now.

- This evening, you may discuss with your partner about whether or when the application will get approved.
  
  - Not knowing it already has been.

- You get a letter in the post days later, informing you of the approval.
  
  - That’s when your “copy” of the world is made consistent.
Example: your bank loan

- Your loan application may be approved by the loan department right now.

- This evening, you may discuss with your partner about whether or when the application will get approved.
  - Not knowing it already has been.

- You get a letter in the post days later, informing you of the approval.
  - That’s when your “copy” of the world is made consistent.

- This is eventual consistency.
If you look for eventual consistency, you will find it everywhere.

(The light from the Sun takes 8 mins to get to Earth.)
Modelling systems based on *lessons from the real world* is very helpful when dealing with eventual consistency.
But we’re better than this, right?
Software can push the natural boundaries of the real world, can it not?
Well, actually…
Your system is **already** eventually consistent.
As soon as you let go of a database connection, you don’t know the data you drew from it is still correct.
While returning data to the client, the data may have been changed.
You can narrow the time window, but never eliminate it.
If we can’t eliminate eventual consistency, we have to manage it.
Risk is a function of time.
Is 1 nanosecond okay?

What about 1 month?
Your threshold is contextual.
Showing order status to a user? Threshold: seconds.
Sales report?
Threshold: hours.
It’s a business risk, not a technical one.
Ask: what’s the risk of making decisions with out-of-date information?
Unlike other architectures, however, this one *emphasises* eventual consistency and makes it a *first class concern*. 
Call it a “feature.”
This presents challenges, especially for developers used to not dealing with it.

(Even though it’s always been there.)
There are ways to manage eventual consistency.
Managing eventual consistency
Managing eventual consistency

› Educate your users.
Managing eventual consistency

› Educate your users.

› Lie to your users.
Managing eventual consistency

‣ Educate your users.

‣ Lie to your users.

‣ Read your own writes.
Managing eventual consistency

- Educate your users.
- Lie to your users.
  - Read your own writes.
- Build new user interfaces.
Managing eventual consistency

› Educate your users.

› Lie to your users.
  
  › Read your own writes.

› Build new user interfaces.

› Slow down the user.
Managing eventual consistency

‣ Educate your users.

‣ Lie to your users.
  ‣ Read your own writes.

‣ Build new user interfaces.

‣ Slow down the user.

‣ Speed up the backend.
Though it is manageable, it can be an absolute blocker.
Handing over the **crown jewels.**
There is nothing more important than the data we store.
Quote from FP workshop: “Apps age like fish, data age like wine.”
Data persists even as technologies are replaced.
In any organisation, fundamentally changing the way data is stored is a risk.
The default answer to whether you should fundamentally change how to store data is “no.”
Opacity.
Traditional relational models means any developer can **poke around** in the data to answer questions.
Not just developers! Other folks with basic SQL experience as well.
An Event Store renders all of those tools and ubiquitous access *irrelevant*. This is a big deal.
Your data is more **opaque** in an Event Store.
You have to **create projections** to make sense of the data.
This can be burdensome, particularly for ad-hoc queries.
You have to create **compensating events** rather than quick “fixups.”

Both an advantage and a disadvantage.
Mistakes are forever.
Natural evolution of event schemas do happen.
You can’t change events after-the-fact.
Upcasting and compensating events become the norm.
Confidentiality can be issue.
There are strategies but they can be painful.

Some are not for the faint of heart.
It’s not mainstream.
There are not enough resources.
Books, courses, presentations, etc.

But also libraries and frameworks.
There may never be.
It can be difficult to hire.
In my experience, many teams with little prior knowledge is gonna struggle hard.
You almost have to have someone who’s done it before.
The J curve of productivity.
It will get worse before it gets better.
It takes time before the benefits kick in.
Particularly if the choice was made *for* the team, not *by* the team.
Don’t impose Event Sourcing on a team that:

- Lacks buy-in to try it.
- Lacks stakeholder support.
- Lacks intestinal fortitude.
Complexity.
First, let’s not conflate inherent necessary domain complexity with accidental complexity.
We can’t avoid complexity, but we can choose how to implement it.
Monoliths are:

a simple arrangement of complex things.
Event Sourcing with CQRS is:

a complex arrangement of simple things.
The same goes for **Microservices**.
Don’t listen to people who tell you Microservices make things simpler without acknowledging it also makes other things more complex.
Event Sourcing with CQRS has lots of moving parts.
Event Sourcing with CQRS usually means difficulty tracing across async components.
Beware devolving into a complex arrangement of complex things.
It’s easy to get here with the cognitive load introduced by a fundamental paradigm shift.
Most developers will struggle.
This is the **number one** issue I’ve encountered.

(YMMV.)
Cultures of control.
Not appropriate to allow systems you have less control of to consume events.
Events contain all the data, and you might not want (or be allowed) to share that data.
Event Sourcing with CQRS favours environments where autonomy is a cultural norm, rather than control.
You *can* do it in tighter control environments:

- Keep Event Store behind high walls.
- Build more standard APIs.
- Only contain data you want clients to have.
This is a pattern
It’s not appropriate for every use case.
It’s not a silver bullet.
It has lots of awesome benefits, but also significant disadvantages.
If you can overcome the challenges, it can be incredibly powerful.
“Event Sourcing is a multiplier.

Done well, it gives you magical powers in terms of flexibility and scalability.

Done poorly, it becomes worse than the crappiest balls of mud.”
Thank you!