Event-Driven Architecture

hgraca Architecture, Development, Series, The Software Architecture Chronicles, Uncategorized October 5, 2017August 30, 2018 12 Minutes

This post is part of The Software Architecture Chronicles (https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/), a series of posts about Software Architecture (https://herbertograca.com/category/development/series/software-architecture/). In them, I write about what I've learned on Software Architecture, how I think of it, and how I use that knowledge. The contents of this post might make more sense if you read the previous posts in this series.

Using events to design applications is a practice that seems to be around since the late 1980s. We can use events anywhere in the frontend or backend. When a button is pressed, when some data changes or some backend action is performed.

But what is it exactly? When should we use it and how? What are the downsides?

What/When/Why

Just like classes, components should have low coupling between them but be high cohesive within them. When components need to collaborate, let's say a component "A" needs to trigger some logic in component "B", the natural way to do it is simply to have component A call a method in an object of component B. However, if A knows about the existence of B, they are coupled, A depends on B, making the system more difficult to change and maintain. Events can be used to **prevent coupling**.

Furthermore, as a side effect of using events and having decoupled components, if we have a team working only on component B, it can change how component B reacts to logic in component A without even talking to the team responsible for component A. Components can evolve independently: **our application becomes more organic**.

Even within the same component, sometimes we have code that needs to be executed as a result of an action, but it doesn't need to be executed immediately, in the same request/response. The most flagrant example is sending out an email. In this case, we can immediately return a response to the user and send out the email at a later time, in an **async** fashion and avoid having the user waiting for an email to be sent.

Nevertheless, there are dangers to it. If we use it indiscriminately, we run the risk of ending up with logic flows that are conceptually highly cohesive but wired up together by events which are a decoupling mechanism. In other words, code that should be together will be separated and will be difficult to track its flow (kind of like the *goto* statement), to understand it and reason about it: It will be spaghetti code!

To prevent turning our codebase into a big pile of spaghetti code, we should keep the usage of events limited to clearly identified situations. In my experience, there are three cases in which to use events:

1. To decouple components
2. To perform async tasks
3. To keep track of state changes (audit log)

1. To decouple components

When component A performs the logic that needs to trigger the component B logic, instead of calling it directly, we can trigger an event sending into an event dispatcher. Component B will be listening to that specific event in the dispatcher and will act whenever the event occurs.

This means that both A and B will be depending on the dispatcher and the event, but they will have no knowledge of each other: they will be decoupled.

Ideally, both the dispatcher and the event should live in neither component:

- The dispatcher should be a library completely independent of our application and therefore installed in a generic location using a dependency management system. In PHP world, this is something installed in the *vendor* folder, using Composer (https://getcomposer.org/).
- The event, nevertheless, is part of our application but should live outside both components as to keep them unaware of each other. The event is shared between the components and it is part of the core of the application. Events are part of what DDD calls the **Shared Kernel**. This way, both components will depend on the Shared Kernel but will remain unaware of each other.
  Nevertheless, in a Monolithic application, for convenience, it is acceptable to place it in the component that triggers the event.

> *Shared Kernel*
>
> *[…] Designate with an explicit boundary some subset of the domain model that the teams agree to share. Keep this kernel small. […] This explicitly shared stuff has special status, and shouldn't be changed without consultation with the other team.*
>
> *Eric Evans 2014, Domain-Driven Design Reference (https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-Summaries/dp/1457501198)*

2. To perform async tasks

Sometimes we have a piece of logic that we want to have executed, but it might take quite some time executing and we don't want to keep the user waiting for it to finish. In such a case, it is desirable to have it run as an async job and immediately return a message to the user informing him that his request will be executed later, asynchronously.

For example, placing an order in a webshop could be done in sync, but sending an email notifying the user could be done async.

In such cases, what we can do is trigger an event that will be queued, and will sit in the queue until a worker can pick it up and execute it whenever the system has resources for it.

In these cases, it doesn't really matter if the associated logic is in the same bounded context or not, either way, the logic is decoupled.

3. To keep track of state changes (audit log)

In the traditional way of storing data, we have entities holding some data. When the data in those entities change, we simply update a DB table row to reflect the new values.

The problem here is that we do not store exactly what changed and when.

We can store events containing the changes, in an *audit log* kind of construction.

More about this further ahead, in the explanation about *Event Sourcing*.

Listeners Vs Subscribers

A common debate to have when implementing an event driven architecture is whether to use event listeners or event subscribers, so let's clarify my take on that:

1. **Event Listeners** react to only one event and can have multiple methods reacting to it. So we should name the listener following the event name, for example if we have a "*UserRegisteredEvent*" we will have a "*UserRegisteredEventListener*" this will make it easy to know, without even looking inside the file, to what event that listener is listening to. The methods (reactions) to the event should reflect what the method actually does, for example "*notifyNewUserAboutHisAccount()*" and "*notifyAdminThatNewUserHasRegistered()*". This should be the usual approach for most cases, as it keeps the listener small and focused on the single responsibility of reacting to a specific event. Furthermore, if we have a componentized architecture, each component (if needed) would have its own listener to an event that could be triggered from multiple locations.
2. **Event Subscribers** react to multiple events and have multiple methods reacting to them. The naming of the subscriber is more difficult as it can not be ad-hoc, nevertheless the subscriber should still comply with the Single Responsibility Principle, so the name of the subscriber needs to reflect its single intent. Using event subscribers should be a less common approach, especially in components, as it can easily break the single responsibility principle. An example of a good use case for an event subscriber is to manage transactions, more concretely we could have an event subscriber named "*RequestTransactionSubscriber*" reacting to events like "*RequestReceivedEvent*", "*ResponseSentEvent*" and "*KernelExceptionEvent*", and bind to them the start, commit and rollback of transactions, respectively, each in their own method like "*startTransaction()*", "*finishTransaction()*" and "*rollbackTransaction()*". This would be a subscriber reacting to multiple events but still focusing on the single responsibility of managing the request transaction.

Patterns

Martin Fowler identifies three different types of event patterns:

- Event Notification
- Event-Carried State Transfer
- Event-Sourcing

All of these patterns share the same key concepts:

1. Events communicate that something has happened (they occur <u>after</u> something);
2. Events are broadcasted to any code that is listening (<u>several</u> code units can react to an event).

Event Notification

Let's suppose we have an application core with clearly defined components. Ideally, those components are completely decoupled from each other but, some of their functionality requires some **logic in other components to be <u>executed</u>**.

This is the most typical case, which was described earlier: When component A performs the logic that needs to trigger the component B logic, instead of calling it directly, it triggers an event sending it to an event dispatcher. Component B will be listening to that specific event in the dispatcher and will act whenever the event occurs.

It is important to note that, a characteristic of this pattern is that **the event carries minimal data**. It carries only enough data for the listener to know what happened and carry out their code, usually just entity ID(s) and maybe the date and time that the event was created.

Advantages

- Greater resilience, if the events are queued the origin component can perform its logic even if the secondary logic can not be performed at that moment because of a bug (since they are queued, they can be executed later, when the bug is fixed);
- Reduced latency, if the event is queued the user does not need to wait for that logic to be executed;
- Teams can evolve the components independently, making their work easier, faster, less prone to problems, and more organic;

Disadvantages

- If used without criteria, it has the potential to transform the codebase in a pile of spaghetti code.

Event-Carried State Transfer

Let's consider again the previous example of an application core with clearly defined components. This time, for some of their functionality, they **need <u>data</u> from other components**. The most natural way of getting that data is to ask the other components for it, but that means the querying component will know about the queried component: the components will be coupled to each other!

Another way of sharing this data is by using events that are triggered when the component that owns the data, changes it. **The event will carry the whole new version of the data** with them. The Components interested in that data will be listening to those events and will react to them by storing a local copy of that data. This way, when they need that external data, they will have it locally, they will not need to query the other component for it.

Advantages

- Greater resilience, since the querying components can function if the queried component becomes unavailable (either because there is a bug or the remote server is unreachable);
- Reduced latency, as there's no remote call (when the queried component is remote) required to access the data;
- We don't have to worry about load on the queried component to satisfy queries from all the querying components (especially if it's a remote component);

Disadvantages

- There will be several copies of the same data, although they will be read-only copies and data storage is not a problem nowadays;
- Higher complexity of the querying component, as it will need the logic to maintain a local copy of the external data although this is pretty standard logic.

Maybe this pattern is not necessary if both components execute in the same process, which makes for fast communication between components, but even then it might be interesting to use it for decoupling and maintainability sake or as a preparation for decoupling those components into different microservices, sometime in the future. It all depends on what are our current needs, future needs and how far we want/need to go with decoupling.

Event Sourcing

Let's assume an Entity in its initial state. Being an Entity, it has its own identity, it's a specific thing in the real world, which the application is modelling. Along its lifetime, the Entity data changes and, traditionally, the current state of the entity is simply stored as a row in a DB.

Transaction log

This is fine for most cases, but what happens if we need to know how the Entity reached that state (ie. we want to know the credits and debits of our bank account)? It's not possible because we only store the current state!

Using event sourcing, instead of storing the Entity state, we focus on storing the Entity state **changes** and **computing the Entity state** from those changes. Each state change is an event, stored in an event stream (ie. a table in an RDBMS). When we need the current state of an Entity, we calculate it from all of its events in the event stream.

> *The event store becomes the principal source of truth, and the system state is purely derived from it. For programmers, the best example of this is a version-control system. The log of all the commits is the event store and the working copy of the source tree is the system state.*
>
> *Greg Young 2010, CQRS Documents (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf)*

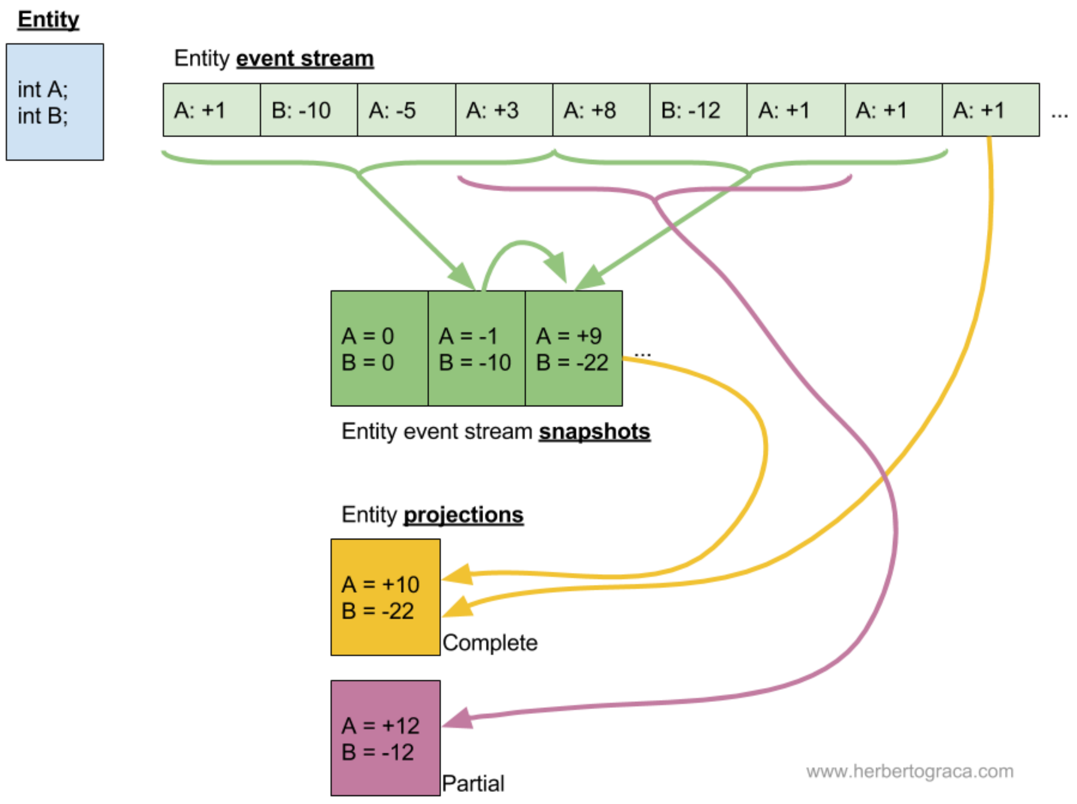
Deletions

If we have a state change (event) that is a mistake, we can not simply delete that event because that would change the state change history, and it would go against the whole idea of doing event sourcing. Instead, we create an event, in the event stream, that reverses the event that we would like to delete. This process is called a Reversal Transaction, and not only brings the Entity back to the desired state but also *leaves a trail that shows that the object had been in that state at a given point in time*.

> *There are also architectural benefits to not deleting data. The storage system becomes an additive only architecture, it is well known that append-only architectures distribute more easily than updating architectures because there are far fewer locks to deal with.*
>
> *Greg Young 2010, CQRS Documents (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf)*

Snapshots

However, when we have many events in an event stream, computing an Entity state will be costly, it will not be performant. To solve this, every X amount of events we will create a snapshot of the Entity state at that point in time. This way, when we need the entity state, we only need to calculate it up to the last snapshot. Hell, we can even keep a permanently updated snapshot of the Entity, that way we have the best of both worlds.



Projections

In event sourcing, we also have the concept of a **projection**, which is the computation of the events in an event stream, from and to specific moments. This means that a snapshot, or the current state of an entity, fits the definition of a projection. But the most valuable idea in the concept of projections, is that we can analyse the "behaviour" of Entities during specific periods of time, which allows us to make educated guesses about the future (ie. if in the past 5 years an Entity had increased activity during August, it's likely that next August the same will happen), and this can be an extremely valuable capability for the business.

Pros and cons

Event sourcing can be very useful for both the business and the development process:

- we query these events, useful both for business and for development to understand the users and system behaviour (debugging);
- we can also use the event log to reconstruct past states, again useful both for business and for development;
- automatically adjust the state to cope with retroactive changes, great for business;
- explore alternative histories by injecting hypothetical events when replaying, awesome for business.

But not everything is good news, be aware of hidden problems:

- ### External updates

When our events trigger updates in external systems, we do not want to retrigger those events when we are replaying the events in order to create a projection. At this point, we can simply disable the external updates when we are in "replay mode", maybe encapsulating that logic in a gateway.
Another solution, depending on the actual problem, might be to buffer the updates to external systems, performing them after a certain amount of time, when it is safe to assume that the events will not be replayed.

- ### External Queries

  When our events use a query to an external system, ie. getting stock bonds ratings, what happens when we are replaying the events in order to create a projection? We might want to get the same ratings that were used when the events were run for the first time, maybe years ago. So either the remote application can give us those values or we need to store them in our system so we can simulate the remote query, again, by encapsulating that logic in the gateway.

- ### Code Changes

  Martin Fowler identifies 3 types of code changes: *new features*, *bug fixes*, and *temporal logic*. The real problem comes when replaying events which should be played with different business logic rules at different moments in time, ie. last year tax calculations are different than this year. As usual, conditional logic can be used but it will become messy, so the advice it to use a strategy pattern instead.

So, I advise caution, and I follow these rules whenever possible:

- Keep events dumb, knowing only about the state change and not how it was decided. That way we can safely replay any event and expect the result to be the same even if the business rules have changed in the meantime (although we will need to keep the legacy business rules so we can apply them when replaying past events);
- Interactions with external systems should not depend on these events, this way we can safely replay events without the danger of retriggering external logic and we don't need to ensure the reply from the external system is the same as when the event was played originally.

And, of course, like any other pattern, we don't need to use it everywhere, we should use it where it makes sense, where it brings us an advantage and solves more problems than it creates.

Conclusion

It's, again, mostly about encapsulation, low coupling and high cohesion.

Events can leverage tremendous benefits to the maintainability, performance and growth of the codebase, but, with event sourcing, also to the reliability and information that the system data can provide.

Nevertheless, it's a path with its own dangers because both the conceptual and technical complexity increase and a misuse in either of them can have disastrous outcomes.

Sources

2005 • Martin Fowler • Event Sourcing (https://martinfowler.com/eaaDev/EventSourcing.html)

2006 • Martin Fowler • Focusing on Events (https://martinfowler.com/eaaDev/EventNarrative.html)

2010 • Greg Young • CQRS Documents (https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf)

2014 • Greg Young • CQRS and Event Sourcing – Code on the Beach 2014 (https://www.youtube.com/watch?v=JHGkaShoyNs)

2014 • Eric Evans • Domain-Driven Design Reference (https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-Summaries/dp/1457501198)

2017 • Martin Fowler • What do you mean by "Event-Driven"? (https://martinfowler.com/articles/201701-event-driven.html)

2017 • Martin Fowler • The Many Meanings of Event-Driven Architecture (https://www.youtube.com/watch?v=STKCRSUsyP0)

Translations

- Vietnamese (https://edwardthienhoang.wordpress.com/2018/08/20/event-driven-architecture/), by Edward Hien Hoang
  Tagged:
  events,
  software architecture,
  software developement

Published by hgraca

View all posts by hgraca

13 thoughts on "Event-Driven Architecture"

dominikjeske says:
August 29, 2019 at 15:41
Hello Herbert,

I'm reading "The Software Architecture Chronicles" – nice work with suming all this things. In context of event architecture there is no mention about Actor Model. Do you plan to write something about this?

↪ Reply
Pingback: Event-Driven Architecture – Java Việt Nam
Pingback: 事件驱动架构设计 | PHP Zendo
Pingback: November-2017.php – Jesper Jarlskov's blog
Pingback: October-2017.php – Jesper Jarlskov's blog
Pingback: Les liens de la semaine – Édition #247 | French Coding
Hernan Bazzino says:
October 23, 2017 at 18:50
Isn't EDA too much for the example cases you posted?
I mean, for getting better de coupling you can use DI, for async tak just an async process, and for the audit log also an async process, if anything needed at all. I understand and support its use as a communication pattern bewteen different apps, like in a big CRM, ERP where each of its app(or modules?) need to communicate actions to the rest, or in a SOA environment, but implementing a good EDA gets quite heavy on the coding, debugging, testing scenario, and even more if you are dealing with sync events, or when you need to keep track of its states/ results, so it seems too much for these examples, that can clearly be solved using much easier patterns…

↪ Reply
hgraca says:
October 24, 2017 at 00:35
Well, DI gives us class level decoupling. However it does not give us bounded context decoupling. A bounded context can be for example, as you say, several apps in a ERP, CRM, or SOA ecosystem. However it can also refer to different components, or modules if you will, in a monolith. I like to feel that components are decoupled between them, I feel it simplifies the code by reducing the amount of direct dependencies and makes the application more organic.

Suppose we have a service that updates an entity and triggers side effects in ten other services in different components… Without events we need to inject all ten services in the initial service . With events we only inject the event dispatcher.

The examples given are purposely simple because I want to focus on the pattern concept, and avoid getting the reader distracted with the complexity of the problem I am using as an example. My experience as a teacher tells me this strategy works better when teaching ppl that have no knowledge of the concept being taught. And once we understand the concept we can adapt it to other contexts and needs.

↪ Reply
Hernan Bazzino says:
October 24, 2017 at 09:02
With DI I meant Dependency Inversion, not Injection, sorry for not being clear enough. With DI you would only need to reference a coupling service, which will take care of the rest of the 10 other services. When you have that amount of interaction between services and or modules/ apps, you usually need to share a lot of data and state between them, and having an event driven approach for all that interaction gets messy quite fast, as you end up having lots of different events publishers and subscribers, and different messages and

timings to which you inevitably will need to synchronize.
Thanks for the explanation, I see where you're going. Its a nicely exaplained post, I just felt the examples weren't probably the best case for event handling.

↪ Reply

hgraca says:
October 24, 2017 at 18:11
Glad u understand why i structured the post that way.

However, the strategy you mention still doesn't decouple the components, we would just be moving the coupling to another intermediate class. The calling code would still know about the intermediate class, who would know about the other 10 services.

However, if those services are injected into the intermediate class through some config, that would indeed decouple the intermediate class from the other 10 services, but that is half way to an event dispatcher which is a much cleaner solution.

I agree that events have the potential to result in spaghetti code, as i state in my post, but if we use them properly they can simplify our code a lot.

Anyway, thanks for your feedback, i do appreciate it! 🙂

↪ Reply

breckenedge says:
October 11, 2017 at 17:42
May also want to research Fowler's opinion on the CQRS pattern:
https://martinfowler.com/bliki/CQRS.html

> Despite these benefits, you should be very cautious about using CQRS.

Also in event-driven systems, knowing if the event stream is strictly ordered is very important when using Event Sourcing. I've seen some implementations where deletes and updates can arrive before inserts.

↪ Reply

hgraca says:
October 11, 2017 at 18:16
Yeah, I did, my next post is about CQRS. Its gonna be published tomorrow 🙂

↪ Reply

Pingback: The Software Architecture Chronicles | @herbertograca